MINISTERUL EDUCAȚIEI, CULTURII ȘI CERCETĂRII AL REPUBLICII MOLDOVA

IP CENTRUL DE EXCELENȚĂ ÎN INFORMATICĂ ȘI TEHNOLOGII INFORMAȚIONALE

CATEDRA INFORMATICA II

ANDRIAN DASCAL

PROGRAMAREA CALCULATORULUI

METODE ȘI TEHNICI DE PROGRAMARE ÎN C++

Îndrumar pentru activitățile practice,
destinat cadrelor didactice și elevilor din IP CEITI

CHIȘINĂU, 2020

Aprobat: *Consiliul metodic-ştiinţific, IP CEITI, proces verbal Nr. 8, din data 30.03.21, _____*

_____

Elaborat conform Curriculumului modular „Programarea calculatorului”, ediţia 2020. Discutat şi examinat la şedinţa catedrei „Informatica II” din 12.11.2020, Proces Verbal Nr. 2, şef catedră, Luminița Gribineț _____

Autor: *Andrian Dascal, magistru în Informatică şi Tehnologii Informaţionale, grad didactic II, IP CEITI.*

Recenzie: *Ioan Jeleascov, prof. la discipline de informatică, IP CEITI.*

Redactare ştiinţifică: *Ioan Jeleascov, „architect advisor” în cadrul unităţii "StoneHard".*

Redactare lingvistică: *Doina Lupu, prof. de limba şi literatura română, gr. didactic I, IP CEITI.*

Dragi prieteni,

"Cu cât înveți mai multe, cu atât se leagă mai multe lucruri, iar aceasta exercită asupra voastră o puternică fascinație intelectuală."
Graham Priest

Cunoaștem că C++ este un limbaj de programare cu scop general, creat de Bjarne Stroustrup ca o extensie a limbajului de programare C sau „C cu clase". Limbajul s-a extins semnificativ de-a lungul timpului, iar C++ modern are funcții orientate spre obiecte, generice și funcționale, pe lângă facilități pentru manipularea memoriei la nivel scăzut. Acesta este aproape întotdeauna implementat ca limbaj compilat, iar mulți furnizori furnizează compilatoare C++, inclusiv Free Software Foundation, LLVM, Microsoft, Intel, Oracle și IBM, deci este disponibil pe mai multe platforme.

Din ciclul gimnazial cunoaștem că informatica este un domeniu al științei care studiază metodele de păstrare, transmitere și prelucrare a informației cu ajutorul calculatoarelor. le- Am examinat noțiunile de bază ale informaticii – date, informație, calculator, executant, algoritm – și ne-am format deprinderile practice de lucru la calculator. Am elaborat mai multe secvențe de cod pentru comanda executanților și ne-am convins că funcționarea calculatoarelor moderne este guvernată de algoritmi.

Ghidul de față are drept scop însușirea de către elevi a cunoștințelor necesare pentru dezvoltarea gândirii algoritmice și formarea culturii informaționale. Realizarea acestui obiectiv presupune extinderea capacităților fiecărei persoane de a elabora algoritmi pentru rezolvarea problemelor pe care le întâmpină în viața cotidiană. Este cunoscut faptul că algoritmii descriu foarte exact ordinea și componența operațiilor necesare pentru prelucrarea informației.

Cu ajutorul acestui ghid veți studia elaborarea de subprograme în limbajul C++ prin metodele de elaborare a produselor program. De asemenea, veți implementa în practică cele mai răspândite tehnici de programare: recursivitatea, trierea, reluarea (backtracking), metoda desparte și stăpânește (divide et impera), tehnici de sortare, tehnica Greedy și programarea dinamică. În acest ghid sunt expuse metode de estimare a necesarului de memorie și a timpului cerut de algoritmi, recomandări ce vizează utilizarea recursiei și iterativății.

O atenție deosebită se acordă metodelor de implementare a tehnicilor de programare, interdependenței dintre performanțele calculatorului și complexitatea problemelor ce pot fi rezolvate cu ajutorul mijloacelor respective. Implementarea tehnicilor de programare este ilustrată cu ajutorul mai multor probleme frecvent întâlnite în viața cotidiană și studiate în cadrul disciplinelor școlare. Totodată, în respectivul ghid au fost incluse și probleme de o reală importanță practică, rezolvarea cărora este posibilă doar cu aplicarea calculatorului.

În ansamblu, materialul inclus în acest ghid de implementare practică a metodelor și tehnicilor de programare va contribui la dezvoltarea următoarelor competențe: analiza structurală a problemei; divizarea problemelor complexe în probleme mai simple și reducerea lor la cele deja rezolvate; estimarea complexității algoritmilor destinați soluționării problemelor propuse; utilizarea metodelor formale pentru elaborarea algoritmilor și scrierea programelor respective.

Evident, aceste competențe sunt strict necesare nu numai viitorilor informaticieni, dar și fiecărui om cult care, la sigur, va trăi și va lucra într-un mediu bazat pe cele mai moderne tehnologii informaționale. Recomand acest ghid tuturor cadrelor didactice debutante, precum și tuturor tinerilor amatori de programare.

Autorul

CUPRINSUL

INTRODUCERE

1. Istoria algoritmilor și a algoritmiei

Cuvântul algoritm provine de la numele matematicianului musulman persan din secolul al IX-lea Abu Abdullah Muhammad ibn Musa Al-Khwarizmi. Cuvântul algoritm s-a referit inițial doar la regulile de executare a aritmeticii folosind cifre hinduse-arabe, dar a evoluat prin traducerea latină europeană a numelui lui Al-Khwarizmi în algoritm până în secolul XVIII. Utilizarea cuvântului a evoluat pentru a include toate procedurile definite pentru rezolvarea problemelor sau îndeplinirea sarcinilor.

Opera vechilor geometri greci, matematicianul persan Al-Khwarizmi - adesea considerat drept „părintele algebrei", matematicienii chinezi și vest-europeni culminați în noțiunea Leibniz de „calculus ratiocinator", o algebră a logicii. Euclid a creat un algoritm căruia i s-a dat numele. Algoritmul lui Arhimede oferă o aproximare a numărului Pi. Eratosthenes a definit un algoritm pentru preluarea numerelor prime. Averroès (1126-1198) a folosit metode algoritmice pentru calcule. Adelard de Bath (a 12-a) introduce termenul algorismus, de la Al-Khwarizmi ***[1]****.*

2. Importanța algoritmilor în programarea computerizată

Fie că avem de gând să construim o casă mare, dar în același timp, nu suntem sigur dacă resursele pe care le avem sunt suficiente. Ce vom face? Vom defini un plan de lucru care ne va asigura că vom cheltui puținele resurse disponibile la dispoziția noastră pentru a termina clădirea. Fie alt caz asemănător, intenționăm să parcurgem câțiva kilometri distanță, dar este foarte puțin timp disponibil. Este destul de evident că vom obține cea mai scurtă sau cea mai rapidă rută care ne va duce la destinație.

Când vine vorba de programarea computerului, algoritmii funcționează într-o manieră similară. În limbajul profanului, un algoritm poate fi definit ca o procedură pas cu pas pentru îndeplinirea unei sarcini. În lumea programării, un algoritm este o procedură de calcul bine structurată, care ia unele valori ca intrare (input) și unele valori ca ieșire (output). Algoritmii ne oferă cea mai ideală opțiune de a îndeplini o sarcină. Iată unele argumente ce exprimă o importanță a algoritmilor în programarea computerului: Pentru a îmbunătăți eficiența unui program de calculator; Utilizarea corectă a resurselor calculatorului; Scrierea corectă a unor algoritmi eficienți în timp, etc.

Pentru a vă oferi o imagine mai bună, iată cei mai frecvenți algoritmi: algoritmi de căutare, algoritmi de sortare, algoritmi de compresie, arbori și algoritmi pe bază de grafică, algoritmul de potrivire a modelului și mulți alți algoritmi pe care-i utilizăm la rezolvarea problemelor din activitatea cotidiană. Diferiți algoritmi joacă roluri diferite în programare. Trebuie doar să vă definiți problema, apoi selectați algoritmul potrivit pentru a fi utilizat ***[2]****.*

3. Noțiunea de metodă și tehnică în hermeneutica proprie

Tehnica reprezintă o strategie sau tactică elaborată/inventată de om, în timp ce o metodă este abordarea sau calea. Pentru a înțelege mai bine afirmația de mai sus, să ne imaginăm interpretarea: educația este o metodă de dezvoltare a indivizilor în mod intelectual, în timp ce abordarea specifică a cadrelor didactice din procesul educațional este o tehnică pe care au dobândit-o sau a dezvoltat-o.

O abordare este un set de presupuneri corelative care se referă la natura predării și învățării algoritmilor. Întâi descriem natura materiei care urmează să fie predată. O metodă este un plan general pentru prezentarea ordonată a materialelor lingvistice, nici o parte din care se contrazice și toate se bazează pe abordarea selectată. O abordare este axiomatică, o metodă este procedurală. În cadrul unei abordări, pot exista multe metode.

O tehnică este de implementare, cea care are loc de fapt într-o clasă. Pentru a atinge un obiectiv imediat, este un truc, un strateg sau o confidență deosebită. Tehnicile trebuie să fie în concordanță cu o metodă, prin urmare, în armonie cu o abordare. Cu alte cuvinte, abordarea este nivelul la care sunt specificate ipotezele, presupunerile și convingerile despre învățarea algoritmilor; metoda este nivelul la care teoria este pusă în practică și la care se propun alegeri cu privire la abilitățile particulare care trebuie predate, conținutul care trebuie predat și ordinea în care va fi prezentat conținutul; tehnica este nivelul la care sunt descrise procedurile de clasă.

4. *Metode didactice pentru o învățare activă*

Activizarea metodei de predare-învăţare presupune folosirea unor metode, tehnici şi procedee care să-l implice pe elev în procesul de învăţare, urmărindu-se dezvoltarea gândirii, stimularea creativităţii, dezvoltarea interesului pentru învăţare,în sensul formarii lui ca participant activ la procesul de educare. Astfel elevul este ajutat să înţeleagă lumea în care trăieşte şi să aplice în diferite situaţii de viaţă ceea ce a învăţat.

Opțiunea pentru o metodă sau alta este în strinsă relație și cu personalitatea profesorului și gradul de pregatire, predispoziție(readiness) și stilurile de învățare ale grupului cu care se lucrează. Din această perspectivă, metodele pentru o învățare activă se pot clasifica în:

- *Metode care favorizează înţelegerea conceptelor şi ideilor, valorifică experienţa proprie a elevilor, dezvoltă competenţe de comunicare şi relaţionare, de deliberare pe plan mental şi vizează formarea unei atitudini active: **discuția**, **dezbaterea**, **jocul de rol**, etc.*
- *Metode care stimulează gândirea şi creativitatea, îi determină pe elevi să caute şi să dezvolte soluţii pentru diferite probleme, să facă reflecţii critice şi judecăţi de valoare, să compare şi să analizeze situaţii date: **studiul de caz**, **rezolvarea de probleme**, **jocul didactic**, **exercițiul**, etc.*
- *Metode prin care elevii sunt învăţaţi să lucreze productiv cu alţii şi să-şi dezvolte abilităţi de colaborare şi ajutor reciproc: **mozaicul**, **cafeneaua**, **proiectul în grupuri mici**, etc.*

*Metodele constituie elementul esențial al strategiei didactice, ele reprezentând latura executorie, de punere în acțiune a întregului ansamblu ce caracterizează un curriculum dat. În acest context, metoda poate fi considerată ca instrumentul de realizare cât mai deplină a obiectivelor prestabilite ale activității instructive. De aici și o mare grijă pentru adoptarea unor metode variate, eficiente și adecvate nu numai specificului disciplinelor, profilului instituției, ci scopului general al învățământului și cerințelor de educație ale societății moderne **[3]**.*

5. *Procesul educațional centrat pe elev*

Rolul profesorului în învăţarea centrată pe elev se schimbă radical faţă de abordarea tradiţională. În perspectiva tradiţională, profesorul transmite cunoştinţe elevilor pasivi, accentul fiind pus pe predarea frontală şi ajungerea la răspunsul corect. În contradicţie cu abordarea tradiţională, în învăţarea centrată pe grupe de elevi, profesorul are rol de supraveghetor, sub directa lui îndrumare elevul asumându-şi responsabilitatea propriei învăţări, dezvoltându-şi în timpul procesului instructiv-educativ competenţe de educaţie permanentă, metacognitive şi autoevaluative.

În această abordare modernă, profesorul are mai multe roluri, el fiind pentru elevii săi şi mentor, sfătuitor, supraveghetor, dar şi model, colaborator, îndrumându-şi elevii în vederea atingerii rezultatului scontat, dorit de toţi cei implicaţi în acest proces educativ. Acum accentul se pune pe sprijinirea învăţării, folosind metode şi mijloace de predare – învăţare plăcute, acceptate de către elevi, metode şi mijloace pe care inerent şi ei învaţă să le folosească, folosirea acestora ducând la rezultatul dorit.

Dacă în predarea – învăţarea tradiţională profesorul foloseşte prelegerea ca metodă principală, sigur, împletindu-se şi cu alte metode tradiţionale, şi organizarea clasei de elevi frontală, elevii primind din discursul didactic toate cunoştinţele fără a fi implicaţi în vreun fel în aflarea lor, în învăţarea centrată pe elev prelegerea este înlocuită de învăţarea activă, nu total, iar forma de organizare a clasei de elevi se schimbă, alternând forma individuală cu cea pe grupe, forma frontală fiind forma de organizare a clasei cel mai puţin folosită.

Metodele învăţării active se îmbină cu prelegerea pe tot parcursul procesului instructiv – educativ, pretându-se cerinţelor fiecărui elev în parte. În învăţarea centrată pe elev elevii învaţă fiecare în ritmul său. Din această cauză noi, profesorii avem un rol foarte important în cest proces. Învăţarea centrată pe elev are drept scop atingerea şi asimilarea de către elevi a competenţelor planificate de noi în documentele didactice anuale şi semestriale, dar nu oricum, ci în ritmul lor propriu, dar într-o perioadă de timp rezonabilă.

*Munca profesorului nu mai este aceea de a preda, de a da elevilor cunoştinţele printr-un discurs didactic, ci este de a îndruma elevii cum să înveţe, alegând metodele şi mijloacele didactice care sunt cele mai bune pentru progresul şcolar al elevilor scontat **[4]**.*

ANALIZA ALGORITMILOR

1.1 Noțiuni generale despre teoria complexității.

Teoria complexității este o ramură a informaticii care se ocupă cu studierea complexității algoritmilor. Complexitatea reprezintă puterea de calcul necesară implementării unui algoritm. Ea are două componente principale, și anume complexitatea în timp și cea în spațiu. Complexitatea în spațiu se referă la volumul de memorie necesar calculelor, iar cea în timp se referă la timpul necesar efectuării calculelor, ambele fiind exprimate ca funcții de n, unde n este mărimea datelor de intrare.

În general, complexitatea este exprimată folosind notația big O (O), notație ce reține doar termenul care crește cel mai repede odată cu creșterea lui n, deoarece acest termen are impactul cel mai mare asupra timpului de execuție (sau al spațiului ocupat) al implementărilor algoritmului, ceilalți termeni devenind neglijabili pentru valori mari ale lui n. De exemplu, dacă un algoritm se execută în $2^n + n^5 + 8$ *unități de timp, atunci complexitatea lui în timp este* $O(2^n)$. *Ordinul de mărime* $O(g)$ *se definește în felul următor:*

$O(g) = \{ f : N \to R_+ \mid (\exists c \in R_+^*)(\exists n_0 \in N)(\forall n \geq n_0)\ (f(n) \leq cg(n)) \}$, *unde g este o funcție definită pe N cu valori în R. Se notează* $f(n) = O(g(n))$, *prin care se înțelege că funcția f aparține mulțimii* $O(g(n))$. *Se spune că f este de ordinul O al lui g, aceasta însemnând că f nu crește mai repede, din punct de vedere asimptotic, decât g, eventual multiplicată printr-o constantă* ***[5]***.

Ce trebuie să analizăm?

- *Corectitudine*
  - *Pentru date de intrare valide, algoritmul trebuie să producă rezultate cu proprietățile dorite sau stabilite.*
- *Complexitate*
  - *Un algoritm performant consumă cât mai puține resurse, resursele cele mai importante: timpul și spațiul de memorie. Cantitatea de resurse folosite depinde de dimensiunea datelor de intrare (un algoritm care dă cu aspiratorul în tot apartamentul cu n camere va produce un rezultat în mai mult timp pentru apartamentul reginei cu n=40, decât pentru garsoniera bunicii cu n=3).*

Cum măsurăm?

- *Dacă algoritmul tău rulează mai repede ca algoritmul colegului, ai vrea să conchizi că al tău este mai eficient, asta nu e neapărat adevărat, dacă lupta se desfășoară de fapt între procesorul tău de 3GHz și al lui de 500MHz. De aceea, măsurării experimentale îi preferăm o estimare matematică a numărului de pași pe care îi face algoritmul de la intrare până la ieșire.*

Cazuri esențiale pentru complexitatea temporală:

- *cazul cel mai favorabil (numărul minim de pași pe care îl execută algoritmul, pe o anumită intrare, până ce rezultatul este produs);*
- *cazul cel mai nefavorabil (numărul maxim de pași pe care îl execută algoritmul, pe o anumită intrare, până ce rezultatul este produs);*
- *cazul mediu (numarul mediu de pași pe care îl execută algoritmul, pe o anumită intrare, până ce rezultatul este produs).*

Simplificări

- *Dacă algoritmul tău începe prin a-ți ura o zi bună, această operație nu influențează decisiv eficiența sa. De aceea în analiză poți să ții cont numai de operațiile critice din cadrul algoritmului (acele operații care prin natura lor consumă foarte multe resurse, sau pur și simplu se repetă de foarte multe ori încât ajung să afecteze performanța.*

De exemplu, când sortezi un vector, compararea a 2 elemente se repetă de foarte multe ori şi este o operaţie de care trebuie să ţii cont în analiză).

- *Complexitatea temporală pe cazul mediu este mult mai greu de calculat, de aceea se calculează în general pe cazul cel mai defavorabil (în practică, adesea cazul cel mai defavorabil se apropie de cazul mediu), chiar şi cu toate aceste simplificări, un calcul exact al numărului de paşi este dificil de realizat şi mai degraba inutil.*
- *Este suficientă o estimare care încadreaza algoritmul într-o anumită clasă de complexitate (un set de probleme înrudite din punct de vedere al complexităţii; împărţirea în clase de complexitate are în spate o matematica bine pusă la punct, bazată pe notaţiile asimptotice de complexitate).*

Spaţiului de memorie utilizat de programul care implementează algoritmul într-un limbaj de programare este format dintr-o parte constantă, independentă de datele de intrare, în care se află memorat codul executabil, variabile şi structuri de date de dimensiune constantă alocate static şi o parte variabilă ca lungime care depinde de volumul de date de prelucrat, spaţiul necesar pentru structurile de date alocate dinamic, stive pentru apelul subprogramelor şi a căror lungime depinde în mod cert de algoritmul de rezolvare ***[6]****.*

Un exemplu care scoate în evidenţă diferenţa de eficienţă legată de consumul de spaţiu de memorie, îl constituie doi algoritmi, care calculează suma primelor n numere naturale. Primul algoritm constă în a construi o funcţie care să calculeze succesiv sumele:
0, 0 +1, 0 + 1 + 2, 0 + 1 + 2 + 3, 0 + 1 + 2 +3 + 4, . . . , ,1 + 2 + 3 +... + n.

Notă:

- ✓ *Atenţie! Următoarea problemă include o componentă a bibliotecii* ***graphics.h****.*
- ✓ *Pentru instalarea modulului graphic veţi găsi paşii recomandaţi la pagina 54. Dacă nu doriţi să instalaţi la moment modulul grafic în mediul dvs. de programare, va trebui să excludeţi următoarele linii:* `#include <graphics.h>` *şi* `system("color F0");`
- ✓ *Aceste note informative sunt valabile pentru fiecare secvenţă de cod din acest ghid.*

Problemă:
Se citeşte de la tastatură un număr natural n. Să se determine şi să se afişeze suma primelor n numere naturale nenule.

Metoda I	***Metoda II***
`#include <iostream>` `#include <graphics.h>` `using namespace std;` `int main(){` `  system("color F0");` `  int n, s=0, i;` `  cout<<"Introduceti n, n= \t";` `  cin>>n;` `  for (i=1; i<=n; i++){` `  s=s+i;` `  cout<<"Suma primelor "<<i;` `  cout<<" elemente este: \t"<<s;` `  cout<<endl;` `  }` `  return 0;` `}`	`#include <iostream>` `#include <graphics.h>` `using namespace std;` `int main(){` `  system("color F0");` `  int n, s;` `  cout<<"Introduceti n, n= \t";` `  cin>>n;` `  s=(n*(n+1))/2;` `  cout<<"Suma primelor "<<n;` `  cout<<" elemente este: \t"<<s;` `  cout<<endl;` `  return 0;` `}`
Rezultate: `Introduceti n, n=  5` `Suma primelor 1 elemente este:  1` `Suma primelor 2 elemente este:  3` `Suma primelor 3 elemente este:  6` `Suma primelor 4 elemente este:  10` `Suma primelor 5 elemente este:  15`	***Rezultate:*** `Introduceti n, n=  5` `Suma primelor 5 elemente este:  15`

Pentru a analiza teoretic algoritmul din punct de vedere a duratei de execuţie a programului care-l implementează, vom presupune că o operaţie elementară se execută într-o unitate de timp.

Dacă timpul de execuţie a două operaţii elementare diferă, acesta este bine stabilit (şi constant) pentru fiecare operaţie elementară şi este independent de datele cu care operează. Nu vom restrânge generalitatea dacă presupunem că timpul de execuţie este acelaşi pentru toate operaţiile elementare. Observăm că timpul de execuţie al programului este direct proporţional cu numărul de operaţii simple efectuate de algoritm, număr care oferă un criteriu de comparaţie, între algoritmi şi programele care-i implementează, fără a mai fi necesară implementarea efectivă şi execuţia.

Teoretic, aprecierea timpului de execuţie se poate face pentru orice volum de date de intrare, lucru care practic nu este realizabil, dacă am încerca să măsurăm efectiv timpul pe perioada execuţiei programelor, iar aceasta s-ar face pentru seturi particulare de date de intrare. Analiza complexităţii algoritmului ca timp de execuţie presupune determinarea numărului de operaţii elementare efectuate de algoritm, nu şi a timpului total de execuţie a acestora, ţinând cont doar de ordinul de mărime a numărului de operaţii elementare [7].

Evaluarea ordinului unui algoritm echivalează cu determinarea unei margini superioare a timpului de execuţie a algoritmului. Prin urmare:

- *un algoritm cu t(f(n)) O(1) necesită un timp de execuţie constant;*
- *un algoritm cu t(f(n)) O(log n) se numeşte logaritmic;*
- *un algoritm cu t(f(n)) O(n) se numeşte liniar;*
- *un algoritm cu t(f(n)) $O(n^2)$ se numeşte pătratic;*
- *un algoritm cu t(f(n)) $O(n^3)$, se numeşte cubic;*
- *un algoritm cu t(f(n)) $O(n^k)$ se numeşte polinomial;*
- *un algoritm cu t(f(n)) $O(2^n)$ se numeşte exponenţial.*

Observaţii:

- *Această notaţie, numită asimptotică, determină o clasificare a algoritmilor impusă de valoarea ordinului de complexitate: O(1); O(log n); O(n); O(n log n); $O(n^2)$; $O(n^k)$; $O(2^n)$; unde $k > 2$. Putem astfel clasifica algoritmii din punctul de vedere al performanţei.*

- *Dacă avem t(f((n)) $O(2^n)$ şi un calculator care face 1 bilion (10^9) de operaţii pe secundă, atunci:*
  - *pentru n = 40, îi sunt necesare aproximativ 18 minute;*
  - *pentru n = 50, îi sunt necesare aproximativ 13 zile;*
  - *pentru n = 60, îi sunt necesari aproape peste 310 ani;*
  - *pentru n = 100, îi sunt necesari aproape $4*10^{13}$ ani.*

Notă:

- ✓ *Chiar dacă timpul de execuţie al unui algoritm este direct proporţional cu numărul de operaţii elementare, totuşi, acest număr poate varia considerabil în funcţie de caracteristicile relevante ale datelor de intrare (cum ar fi ordinul de mărime al setului de date, cunoscut şi sub denumirea de volumul datelor de intrare).*
- ✓ *Respectiva complexitate este analizată prin prisma maşinilor de calcul de tip Timeshering, care posedă doar pseudo-paralelism. Cu alte cuvinte, aceste maşini de calcul doar comutează procesele de la un registru la altul. Deci, efectuează comutarea între procese, iar frecvenţa procesorului indică numărul de comutări posibile într-o secundă. De exemplu, un procesor cu frecvenţa 2GHz are maxim 2 miliarde de procese fără activarea regimului Turbo.*

1.2 Notații asimptotice. Estimarea necesarului de memorie.

Ce înseamnă și de ce depinde complexitatea ?

- *un algoritm performant consumă cât mai puține resurse;*
- *resursele cele mai importante: timpul și spațiul (memoria), vom vorbi despre complexitate temporală și spațială;*
- *cantitatea de resurse folosită depinde de:*
  - *datele de intrare (ex: în general consumi mai puțin să sortezi un vector gata sortat decât unul "bine amestecat")*
  - *dimensiunea datelor de intrare (ex: consumi mai mult sortând un vector de 1000 de elemente față de unul de 3 elemente), complexitatea va fi în funcție de dimensiunea datelor de intrare (notată tipic T(n)).*
- *măsurarea experimentală a timpului consumat de algoritm introduce dependența de hardware/software (contează mașina pe care rulezi, limbajul în care ai programat – lucruri care nu țin de calitatea intrinsecă a algoritmului), vom prefera o estimare matematică a numărului de pași parcurși de algoritm;*
- *numărul exact de pași este dificil de calculat, se impun o serie de simplificări;*
- *în calculul de complexitate ținem cont numai de operațiile critice din algoritm, acele operații care prin natura lor sunt foarte consumatoare, sau prin faptul că sunt efectuate de un număr semnificativ de ori (ex: într-un algoritm de sortare o operație critică este comparația de elemente, întrucât se produce de foarte multe ori);*
- *în continuare este dificil de calculat un număr de pași, ceea ce va interesa este:*
  - *De ce ordin este funcția de complexitate? (În ce clasă de complexitate?);*
  - *Cum crește funcția de complexitate?*
  - *Cât de repede? Crește liniar? Crește logaritmic? Crește exponential? etc;*
  - *Astfel, deducem cum se va comporta algoritmul pe dimensiuni mari ale datelor de intrare, acolo unde diferențele se simt cel mai acut;*
- *această idee (spectaculoasă) stă la baza analizei asimptotice de complexitate, bazată pe notațiile asimptotice de complexitate.*

Notă:

- ✓ *În "viața reală" problemele au anumite particularități care ne pot conduce să alegem/ modificăm un algoritm mai puțin performant din punct de vedere matematic, însă mai performant pentru necesitățile problemei în cauză (ex: dacă un algoritm începe să fie mai rapid de la n=1000000 încolo, iar problema noastră reală nu ajunge niciodată la un n atât de mare, alegem un algoritm care (doar) în teorie e mai puțin performant);*
- ✓ *Orice analiză teoretică trebuie dublată de un bun simț practic.*

Tipuri de analiza de complexitate

- *cazul cel mai defavorabil (Ce complexitate rezultă pentru cel mai neprietenos input? Analiza cea mai frecventă, e ușor de realizat și oferă utilizatorului o garanție: algoritmul nu se va purta niciodata mai rău decât atât.)*
- *cazul mediu (La ce complexitate ne putem aștepta în cazul inputurilor aleatoare? O analiză utilă, însă dificil de realizat, necesită cunoaștere a unei distribuții statistice a posibilelor inputuri, pentru a pondera pe fiecare cu probabilitatea sa de apariție și a calcula apoi o astfel de medie ponderată a complexităților.)*
- *cazul cel mai favorabil (Ce complexitate rezultă pentru cel mai prietenos input? Analiza cea mai puțin frecventă, utilă cel mult pentru probleme care tind să aibă inputuri favorabile; în plus, este ușor de trișat, prin plasarea unui test pentru un input anume la începutul algoritmului, caz în care se dă direct rezultatul, cu minim de efort) [8].*

Funcţiile care calculează complexitatea unui anumit algoritm sunt funcţii asimptotic crescătoare de tip $N \to R_+$ *(f(n) calculează numărul de paşi efectuaţi de algoritm pentru intrarea de dimensiune n).*

o, ω

1. $o(g(n)) = \{ f : N \to R_+ \mid \forall c \in R_+, c > 0, \exists n(c) \in N, unde\, 0 \le f(n) < cg(n), pentru\ \forall n > n(c) \}$
2. $\omega(g(n)) = \{ f : N \to R_+ \mid \forall c \in R_+, c > 0, \exists n(c) \in N, unde\, 0 \le cg(n) < f(n), pentru\ \forall n > n(c) \}$

Semnificaţia lui o este că g(n) creşte strict mai repede decât f(n).
În altă exprimare, $f(n) \in o(g(n)) \Leftrightarrow f(n) \in O(g(n)) \wedge not(f(n) \in \Theta(g(n)))$.
Similar, $f(n) \in \omega(g(n)) \Leftrightarrow f(n) \in \Omega(g(n)) \wedge not(f(n) \in \Theta(g(n)))$.

O, Ω, Θ

1. $O(g(n)) = \{ f : N \to R_+ \mid \exists c \in R_+, c > 0, n \in N, unde\, 0 \le f(n) \le cg(n), pentru\ \forall n \ge n_0 \}$
2. $\Omega(g(n)) = \{ f : N \to R_+ \mid \exists c \in R_+, c > 0, n \in N, unde\, 0 \le cg(n) \le f(n), pentru\ \forall n \ge n_0 \}$
3. $\Theta(g(n)) = \{ f : N \to R_+ \mid \exists c_1, c_2 \in R_+, c_1 > 0, c_2 > 0, n \in N, unde\, 0 \le c_1 g(n) \le f(n) \le c_2 g(n) \}$ $pentru\ \forall n \ge n_0$.
4. *f(n) ∈ O(g(n)) înseamnă că pentru valori mari ale dimensiunii intrării, cg(n) este o limită superioară pentru f(n); algoritmul se va purta mereu mai bine decât această limită.*
5. *f(n) ∈ Ω(g(n)) înseamnă că pentru valori mari ale dimensiunii intrării, cg(n) este o limită inferioară pentru f(n); algoritmul se va purta mereu mai prost decât această limită.*
6. *f(n) ∈ Θ(g(n)) înseamnă că pentru valori mari ale dimensiunii intrării,* $c_1 g(n)$ *este o limită inferioară pentru f(n), iar* $c_2 g(n)$ *o limită superioară.*

Proprietăţi esenţiale ale notaţiilor de complexitate

1. *Tranzitivitatea*	$f(n) \in O(h(n)) \wedge h(n) \in O(g(n)) \Rightarrow f(n) \in O(g(n))$; $f(n) \in \Omega(h(n)) \wedge h(n) \in \Omega(g(n)) \Rightarrow f(n) \in \Omega(g(n))$; $f(n) \in \Theta(h(n)) \wedge h(n) \in \Theta(g(n)) \Rightarrow f(n) \in \Theta(g(n))$.
2. *Reflexivitatea*	$f(n) \in O(f(n))$; $f(n) \in \Omega(f(n))$; $f(n) \in \Theta(f(n))$.
3. *Simetria*	$f(n) \in \Theta(g(n)) \Leftrightarrow g(n) \in \Theta(f(n))$.
4. *Antisimetria*	$f(n) \in O(g(n)) \Leftrightarrow g(n) \in \Omega(f(n))$.
5. *Altele*	$f(n) \in \Theta(g(n)) \Leftrightarrow f(n) \in O(g(n)) \wedge f(n) \in \Omega(g(n))$.

Observaţii:

- *Determinarea necesarului de memorie şi a timpului de execuţie prezintă un interes deosebit la etapa de elaborare a algoritmilor şi programelor respective.*
- *Evident, anume pe parcursul acestei etape pot fi eliminaţi din start acei algoritmi, care necesită memorii prea mari sau un timp de execuţie inacceptabil.*
- *Menţionăm că apariţia unor calculatoare cu memorii din ce în ce mai performante fac ca atenţia informaticienilor să fie îndreptată în special asupra necesarului de timp sau, cu alte cuvinte, asupra complexităţii temporale a algoritmilor* ***[9]***.

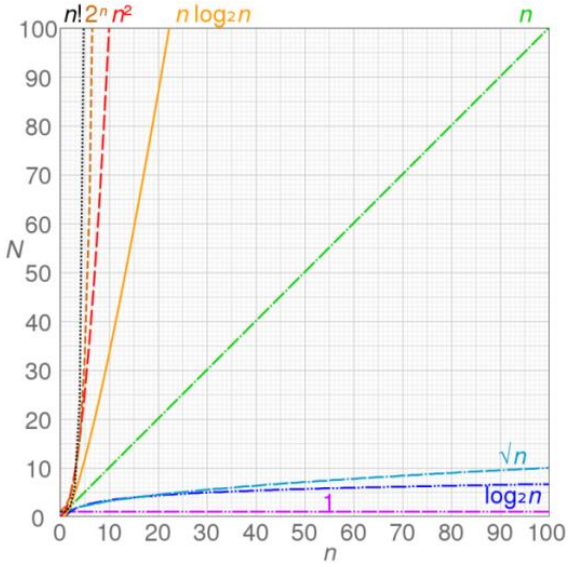
Analiza eficienței algoritmilor înseamnă: estimarea volumului de resurse de calcul necesare execuției algoritmilor.

Analiza eficienței este utilă pentru a compara algoritmii între ei și pentru a obține informații privind resursele de calcul necesare pentru execuția algoritmilor.

Vom folosi următoarele notații:

- *n – un număr natural ce caracterizează mărimea datelor de intrare ale unui algoritm.*
- *V(n) – volumul de memorie internă necesară pentru păstrarea datelor cu care operează algoritmul.*
- *T(n) – timpul de execuție al algoritmului.*

Grafic cu funcțiile utilizate frecvent în analiza algoritmilor, care trasează numărul de operațiuni N în funcție de dimensiunea intrării n pentru fiecare funcție.



Exemplu:
Fie că pentru rezolvarea unei probleme există doi algoritmi pe care-i notăm cu A1 și A2.

Algoritmul	***Necesarul de memorie***	***Timpul de execuție***
A1	$V_{A_1}(n) = 100 \cdot n^2 + 4$	$T_{A_1}(n) = n^3 + 10^{-3}$
A2	$V_{A_2}(n) = 100 \cdot n + 12$	$T_{A_2}(n) = 2^n \cdot 10^{-6}$

Resurse de calcul:

- *Spațiu de memorie = spațiul necesar stocării datelor prelucrate de către algoritm;*
- *Timp de execuție = timp necesar execuției prelucrărilor din cadrul algoritmului.*

Există două tipuri de eficiență:

- *Eficiența în raport cu spațiul de memorie = se referă la spațiul de memorie necesar algoritmului;*
- *Eficiența în raport cu timpul de execuție = se referă la timpul necesar execuției prelucrărilor din algoritm.*

Ambele tipuri de analiză a eficienței algoritmilor se bazează pe următoarea ipoteză:

- ***Volumul resurselor de calcul necesar ce depinde de volumul datelor de intrare = dimensiunea problemei.***

Dimensiunea problemei (DP) *= volumul de memorie necesar pentru a stoca toate datele de intrare ale problemei.*

Dimensiunea problemei este exprimată în una dintre următoarele variante:

- *numărul de componente (valori reale, valori întregi, caractere, etc.) ale datelor de intrare;*
- *numărul de biți necesari stocării datelor de intrare.*

Exemple:

1. *Determinarea minimului unui tablou x[1..n].* ***Răspuns:*** *DP= n.*
2. *Calculul sumei a două matrici cu m linii și n coloane.* ***Răspuns:*** *DP= (m,n) sau m*n.*
3. *Verificarea primalității unui număr n.* ***Răspuns:*** *DP= n. sau DP= $\log_2 n$.*

Exemplu de analiză de complexității:

- *Fie avem la dispoziție metoda de sortare InsertionSort (metoda respectivă o vom studia în compartimentul metodelor de sortare)*
- *Algoritmul:*
  - *Elementul curent se inserează în bucata de vector aflată în stânga lui, care este mereu sortată;*
  - *Se pornește cu al 2-lea (următorul) element.*

- *Subprogramul în limbajul C++:*

```
void insertionSort(int vect[], int n){
    int i, element, j;
    for (j = 2; j < n; j++){
        element = vect[j];
        i = j - 1;
        while (i >= 0 && vect[i] > key){
            vect[i + 1] = vect[i];
            i = i - 1;
        }
        vect[i + 1] = element;
    }
}
```

- *Cazul cel mai favorabil:*
  - *vectorul deja sortat, nu se intră în while;*
  - *$T(n) = 5n-4$, deci obținem $T(n) \in \Theta(n)$.*
- *Cazul cel mai defavorabil:*
  - *vectorul sortat invers, fiecare while merge până la i=0;*
  - *$T(n) \in \Theta(n^2)$.*

Observații:

- ❖ *Ca regulă generală când stabilim clasa de complexitate, în funcția de complexitate:*
  - *ignorăm termenii cu creștere mai înceată;*
  - *ignorăm constanta din fața termenului dominant.*

Exerciții propuse:

1) *Să se redefinească notațiile asimptotice de complexitate folosind limite de funcții.*
2) *Demonstrați proprietățile notațiilor de complexitate.*
3) *Verificați daca notația $\Theta(n2) = \Theta(n2 + n)$ este o reflexivă.*
4) *Determinați valoare de adevăr a următoarelor afirmații:*
   - *$f(n) + g(n) \in O(\max(f(n), g(n)))$;*
   - *$2^{2n} \in \Theta(2n)$;*
   - *$f(n) \in O(f^2(n))$;*
   - *$f(n) + O(f(n)) \in \Theta(f(n))$* ***[10]***.

1.3 Măsurarea timpului de execuție

De multe ori, pentru rezolvarea unei probleme, trebuie ales un algoritm dintre mai mulți posibili, două criterii principale de alegere fiind contradictorii:

- *algoritmul să fie simplu de înțeles, de codificat și de depănat;*
- *algoritmul să folosească eficient resursele calculatorului, să aibă un timp de execuție redus.*

Dacă programul care se scrie trebuie rulat de un număr mic de ori, prima cerință este mai importantă. În această situație, timpul de punere la punct a programului e mai important decât timpul lui de rulare. Deci trebuie aleasă varianta cea mai simplă a programului.

Dacă programul urmează a fi rulat de un număr mare de ori, având și un număr mare de date de prelucrat, trebuie ales algoritmul care duce la o execuție mai rapidă. Chiar în această situație ar trebui implementat mai înainte algoritmul mai simplu și calculată reducerea de timp de execuție pe care ar aduce-o implementarea algoritmului complex.

Timpul de rulare al unui program depinde de următorii factori:

- *datele de intrare;*
- *calitatea codului generat de compilator;*
- *natura și viteza de execuție a instrucțiunilor programului;*
- *complexitatea algoritmului care stă la baza programului.*

Deci timpul de rulare este o funcție de intrare a sa, de cele mai multe ori, nu depinde de valorile de intrare, ci de numărul de date. Se notează cu T(n) timpul de rulare al unui program, ale cărui date de intrare au dimensiunea n. De exemplu T(n) poate fi cn^2, unde c este o constantă. Dacă timpul de rulare depinde de valorile datelor de intrare, în calculul lui T(n) se va lua în considerare situația cea mai defavorabilă, cea care duce la timpul cel mai mare.

Cum timpul de rulare depinde nu numai de intrare (n) și de algoritm, ci și de performanțele calculatorului pe care se execută programul, T(n) se apreciază ca fiind proporțional cu o anumită funcție de n și nu se exprimă în unități reale de timp, deci T(n)=cf(n).

Timpul T(n) de rulare al unui program, poate fi apreciat printr-o funcție O(f(n)), dacă există constantele pozitive c și n0, astfel încât $T(n) \le cf(n)$, oricare ar fi $n \ge n_0$. Funcția O(f(n)) reprezintă aproximarea limitei superioare a lui T(n) și se spune că T(n) este O(f(n)), iar f(n) se numește limita superioară a ratei de creștere a lui T(n) [11].

Ex.1. *$T(n)=3n^3+2n^2$ este $O(n^3)$ pentru ca $3n^3+2n^2 \le 5n^3$, pentru orice $n \ge 0$. Pentru specificarea limitei inferioare a ratei de creștere a lui T(n), se folosește funcția O(g(n)) și se spune T(n) este O(g(n)), însemnând că există o constantă pozitivă c, astfel încât $T(n) \ge cg(n)$, pentru o infinitate de valori ale lui n.*

Ex.2. *$T(n)=n^3+2n^2$ este $O(n^3)$ pentru ca pentru c=1, $T(n)=n^3+2n^2 \ge n^3$, pentru o infinitate de valori ale lui n, cele $n \ge 0$. În comparația între timpii de rulare ai diferitelor programe (sau a diferitelor variante ale aceluiași program), constantele de proporționalitate nu pot fi întotdeauna neglijate. Spre exemplu, s-ar putea spune că un program având $O(n^2)$ este mai rapid decât unul cu $O(n^3)$, dar în cazul când constantele ar fi 100, respectiv 5, al doilea program este mai rapid pentru n<20.*

Înainte de prezentarea câtorva reguli pentru determinarea timpului de execuție al unui program, se dau cele referitoare la suma și produsul funcției O:

1. *Dacă T1(n) și T2(n) sunt timpii de execuție a două secvențe de program P1 și P2, T1(n) fiind O(f(n)), iar T2(n) fiind O(g(n)), atunci timpul de execuție T1(n)+T2(n), al secvenței P1 urmată de P2, va fi O(max(f(n),g(n))).*
2. *Dacă T1(n) este O(f(n)) și T2(n) este O(g(n)),atunci T1(n)*T2(n) este O(f(n)*g(n)).*

Există următoarele cazuri când rata de creștere a timpului de execuție, nu e cel mai bun criteriu de apreciere a performanțelor unui algoritm:

- *dacă un program se rulează de puține ori, se alege algoritmul cel mai ușor de implementat;*
- *dacă întreținerea trebuie făcută de o altă persoană decât cea care l-a scris, un algoritm simplu, chiar mai puțin eficient, e de preferat unuia performant, dar foarte complex și greu de înțeles;*
- *există algoritmi foarte eficienți, dar care necesită un spațiu de memorie foarte mare, astfel încât folosirea memoriei externe, le diminuiază foarte mult performanțele.*

Câteva reguli generale pentru evaluarea timpului de execuție, funcția de mărimea n a datelor de intrare, sunt:

- *timpul de execuție a unei instrucțiuni de asignare, citire sau scriere este O(1);*
- *timpul de rulare a unei secvențe de instrucțiuni e determinat de regula de însumare, fiind proporțional cu cel mai lung timp din cei ai instrucțiunilor secvenței;*
- *timpul de execuție a unei instrucțiuni if-else este suma dintre timpul de evaluare a condiției (O(1)) și cel mai mare dintre timpii de execuție ai instrucțiunilor pentru condiția adevarată sau falsă;*
- *timpul de execuție a unei instrucțiuni de ciclare este suma, pentru toate iterațiile, dintre timpul de execuție a corpului instrucțiunii și cel de evaluare a condiției de terminare (O(1)); pentru*
- *evaluarea timpului de execuție a unui subprogram recursiv, se asociază fiecărui subprogram recursiv un timp necunoscut T(n), unde n măsoară argumentele subprogramului recursiv, se poate obține o relație recurentă pentru T(n), adică o ecuație pentru T(n), în termeni T(k), pentru diferite valori ale lui k;*
- *timpul de execuție poate fi analizat chiar pentru programele scrise în pseudocod, pentru secvențele care cuprind operații asupra unor TDA (tipuri de date abstracte), se pot alege câteva implementări și astfel se poate face comparație între performanțele implementărilor, în contextul aplicației respective* ***[12]****.*

Ex.3. *Având trei secvențe de program cu timpii $O(n^3)$, $O(n^2)$ și O(n*log n), conform regulii anterior prezentate, timpul total de execuție a celor trei secvențe va fi: O(max(max(n^3, n^2), n*log n))= O(max(n^3,n*log n))=O(n^3).*

Ex.4. *Modalitatea de evaluare a timpului de execuție al unui subprogram recursiv, este ilustrată prin cea a funcției de calcul al factorialului:*

```
int factorial(int n){
    if(n <= 1)                              //(1)
        return 1;                           //(2)
    else return (n * factorial(n - 1));     //(3)
}
```

Dimensiunea intrării este n, valoarea numărului al cărui factorial se calculează se notează cu T(n), timpul de rulare al funcției factorial(n). Timpul de rulare pentru liniile (1) și (2) este O(1), iar pentru linia (3) este O(1)+T(n-1), deci cu constantele c și d neprecizate:

- *T(n)=c+T(n-1), pentru n>1 sau T(n)=d , pentru n<=1.*
- *Pentru n>2, expandând pe T(n-1), se obține T(n)=2c+T(n-2), pentru n>2.*
- *Pentru n>3, expandând pe T(n-2), se obține T(n)=3n+T(n-3), pentru n>3.*
- *Deci, în general T(n)=ic+T(n-i), pentru n>i.*
- *În final, când i=n-1, se obține T(n)=(n-1)c+T(1)=(n-1)c+d. Deci T(n) este O(n).*

Câteva recomandări:

- *Pe parcursul lucrărilor de laborator, se vor urmări performanțele TDA-urilor și ale implementărilor lor în aplicații diverse, atât printr-o evaluare a lor în termenii funcției O, cât și prin măsurarea efectivă a timpilor de execuție.*

1.4 Estimarea timpului cerut de program

1. Testarea duratei de execuție a unui program

Se citește de la tastatură un număr natural n. Să se determine și să se afișeze suma primelor n numere naturale nenule împreună cu numărul de microsecunde necesare execuției programului.

Implementare C++

```
#include <iostream>
#include <graphics.h>
#include <sys/time.h>
using namespace std;
long long start;
inline long long getTime(){
    struct timeval tv;
    gettimeofday(&tv, NULL);
    return tv.tv_sec * 1000000LL + tv.tv_usec;
}
int main() {
    system("color F0");long long i, suma;
    long long durata;
    start = getTime(); suma = 0;
    for(i = 0; i<1000000000; i++ ){
        suma = suma + i;
    }
    cout<<"\nSuma numerelor pana la 1000000000 este: \t"<<suma;
    durata = getTime() - start;
    cout<<"\nTimpul de executie in microsecunde: \t"<<durata;
    return 0;
}
```

Rezultate:

```
Suma numerelor pana la 1000000000 este:    499999999500000000
Timpul de executie in microsecunde:     2665343
```

2. Execuția unui program până la expirarea timpului alocat

Acest lucru ne folosește în unele situații precum:

- *Atunci când programul nostru calculează un rezultat care poate fi îmbunătățit în timp (gen mutarea în timpul unui joc).*
- *Atunci când la olimpiadă scriem un program ce folosește metoda backtracking pentru a găsi soluția, care este posibil să nu se încadreze în timpul alocat, caz în care ne oprim după timpul alocat și afișăm soluția găsită până atunci în speranța că este cea optimă.*
- *Ideea de bază este să împărțim programul în segmente de calcul ce se execută iterativ, într-o buclă while. În acea buclă while vom testa timpul, ieșind din buclă, dacă timpul rămas este mai mic de o milisecundă.*

Iată un exemplu ce calculează suma numerelor de la 1 la un miliard, având o secundă la dispoziție. El afișează numărul de iterații și suma calculată până la momentul respectiv.

Implementarea 1

```
#include <iostream>
#include <graphics.h>
#include <sys/time.h>
#define MAXTIME 1000000
using namespace std;
long long start;
inline long long getTime(){
    struct timeval tv;
```

```
    gettimeofday(&tv, NULL);
    return tv.tv_sec * 1000000LL + tv.tv_usec;
}
int main() {
    system("color F0"); long long i, suma, durata;
    start = getTime();
    suma = i = 0;
    while(i<1000000000 && (getTime()-start)<(MAXTIME - 1000)){
            suma = suma + i; i++;
    }
    durata = getTime() - start;
    cout<<"\nAm efectuat "<<i<<" iteratii.";
    cout<<"\nSuma pana la iteratia "<<i<<" este "<<suma;
    cout<<"\nTimpul de executie in microsecunde: \t"<<durata;
    cout<<endl; return 0;
}
```

Rezultate:

```
Am efectuat 29621762 iteratii.
Suma pana la iteratia 29621762 este 438724377181441
Timpul de executie in microsecunde:     999532
```

Deoarece testarea timpului ocupă și ea timp este bine să nu o facem foarte des. Dacă segmentul de calcul este extrem de scurt, vom petrece timp foarte mult calculând timpul. În caz contrar, riscăm să depășim timpul în unul dintre segmente.

Cum putem modifica exemplul anterior pentru a lungi segmentul de calcul? Vom calcula împărțirile în grupe de câte 100. Deoarece împărțirile sunt totuși foarte rapide, un astfel de segment de calcul va fi în continuare foarte rapid. Iată noua variantă:

Implementarea 2

```
#include <iostream>
#include <sys/time.h>
#include <graphics.h>
#define MAXTIME 1000000
#define BATCH 100
using namespace std;
long long start;
inline long long getTime(){
    struct timeval tv;
    gettimeofday(&tv, NULL);
    return tv.tv_sec * 1000000LL + tv.tv_usec;
}
int main() {
    system("color F0");long long i,j, suma, durata;
    start = getTime();
    suma = i = 0;
    while(i<1000000000 && (getTime()-start)<(MAXTIME - 1000)){
        for ( j = i; j < i + BATCH; j++ )
        suma = suma + j; i += BATCH;
    }
    durata = getTime() - start;
    cout<<"\nAm efectuat "<<i<<" iteratii.";
    cout<<"\nSuma pana la iteratia "<<i<<" este "<<suma;
    cout<<"\nTimpul de executie in microsecunde: \t"<<durata;
    cout<<endl; return 0;
}
```

Rezultate:

```
Am efectuat 343971000 iteratii.
Suma pana la iteratia 343971000 este 59158024248514500
Timpul de executie in microsecunde:     1004520
```

Vom calcula împărțirile în grupe de câte 1.000. Iată noua variantă:

Implementarea 3

```
#include <iostream>
#include <graphics.h>
#include <sys/time.h>
#define MAXTIME 1000000
#define BATCH 1000
using namespace std;
long long start;
inline long long getTime(){
    struct timeval tv;
    gettimeofday(&tv, NULL);
    return tv.tv_sec * 1000000LL + tv.tv_usec;
}
int main() {
    system("color F0");long long i,j, suma, durata;
    start = getTime();
    suma = i = 0;
    while(i<1000000000 && (getTime()-start)<(MAXTIME - 1000)){
        for ( j = i; j < i + BATCH; j++ )
        suma = suma + j; i += BATCH;
    }
    durata = getTime() - start; cout<<"\nAm efectuat "<<i<<" iteratii.";
    cout<<"\nSuma pana la iteratia "<<i<<" este "<<suma;
    cout<<"\nTimpul de executie in microsecunde: \t"<<durata;
    cout<<endl; return 0;
}
```

Rezultate:

```
Am efectuat 347683000 iteratii.
Suma pana la iteratia 347683000 este 60441734070658500
Timpul de executie in microsecunde:     1004922
```

Vom calcula împărțirile în grupe de câte 10.000. Iată noua variantă:

Implementarea 4

```
#include <iostream>
#include <graphics.h>
#include <sys/time.h>
#define MAXTIME 1000000
#define BATCH 10000
using namespace std;
long long start;
inline long long getTime(){
    struct timeval tv; gettimeofday(&tv, NULL);
    return tv.tv_sec * 1000000LL + tv.tv_usec;
}
int main() {
    system("color F0");long long i,j, suma, durata;
    start = getTime(); suma = i = 0;
    while(i<1000000000 && (getTime()-start)<(MAXTIME - 1000)){
        for ( j = i; j < i + BATCH; j++ )
        suma = suma + j; i += BATCH;
    }
    durata = getTime() - start; cout<<"\nAm efectuat "<<i<<" iteratii.";
    cout<<"\nSuma pana la iteratia "<<i<<" este "<<suma;
    cout<<"\nTimpul de executie in microsecunde: \t"<<durata; cout<<endl; return 0;
}
```

Rezultate:

```
Am efectuat 354890000 iteratii.
Suma pana la iteratia 354890000 este 62973455872555000
Timpul de executie in microsecunde:     1004497
```

Analizați următorul tabel pentru a observa creșterea productivității algoritmului:

Modificarea	Timpul	Iterații	Suma
DATE INIȚIALE	0.999532 sec.	29621762	438724377181441
BATCH 100	1.004520 sec.	343971000	59158024248514500
BATCH 1000	1.004922 sec.	347683000	60441734070658500
BATCH 10000	1.004497 sec.	354890000	62973455872555000

Concluzie:

- *Viteza de calcul a crescut de la 29.621.762 operații / sec. până la 354.890.000 operații / sec. Pentru cele trei optimizări se observă că timpul de execuție este 1 sec.*

Important:

- *Programele precedente au fost testate în baza procesorului intel core i7, generația 7, seria 7500U, cu două nuclee, frecvența procesorului variază de la 2.70 GHz la 3.50 GHz în regim Turbo.*
- *Rezultatele pe care le veți obține dacă veți executa codul programului pe alt calculator, acesta va depinde de puterea de calcul a calculatorului dvs.*

Analiza în cazul cel mai favorabil:

- *furnizează o margine inferioară pentru timpul de execuție;*
- *permite identificarea algoritmilor ineficienți (dacă un algoritm are un cost ridicat chiar și în cel mai favorabil caz, atunci el nu reprezintă o soluție acceptabilă).*

Analiza în cazul cel mai defavorabil:

- *furnizează cel mai mare timp de execuție în raport cu toate datele de intrare de dimensiune n (reprezintă o margine superioară a timpului de execuție);*
- *marginea superioară a timpului de execuție este mai importantă decât marginea inferioară.*

Analiza în cazul mediu:

- *aceasta analiză se bazează pe cunoașterea distribuției de probabilitate a datelor de intrare;*
- *aceasta înseamnă cunoașterea (estimarea) probabilității de apariție a fiecăreia dintre instanțele posibile ale datelor de intrare (cât de frecvent apare fiecare dintre posibilele valori ale datelor de intrare);*
- *timpul mediu de execuție este valoarea medie (în sens statistic) a timpilor de execuție corespunzători diferitelor instanțe ale datelor de intrare.*

1.5 Complexitatea temporală a algoritmilor

În informatică complexitatea temporală a algoritmilor se caracterizează prin timpul de execuţie T(n) sau numărul de operaţii elementare Q(n). Întrucât calculatoarele moderne au o viteză de calcul foarte mare (mai mult de 10^8 instrucţiuni pe secundă), problema timpului de execuţie (de calcul) se pune numai pentru valori mari ale lui n.

În consecinţă, în formulele ce exprimă numărul de operaţii elementare Q(n) prezintă interes numai termenul dominant, adică acel care tinde repede la infinit. Importanţa termenului dominant faţă de ceilalţi termeni este pusă în evidenţă în tabelul 1.

Tabelul 1. *Valorile termenilor dominanţi*

n	$\log_2 n$	n^2	n^3	n^4	2^n
2	1	4	8	16	4
4	2	16	64	256	16
8	3	64	512	4096	256
16	4	256	4096	65536	65536
32	5	1024	32768	1048576	4294967296

De exemplu, numărul de operaţii elementare ale subprogramului Sortare se exprimă prin formula $Q(n)=16n^2-13n+2$. Termenul dominant din această formulă este $16n^2$. Evident, pentru valorile mari ale lui n numărul de operaţii elementare $Q(n)\approx 16n^2$, iar timpul de calcul $T(n)\approx 16n^2\ \Delta$.

În funcţie de complexitatea temporală, algoritmii se clasifică în:

- *algoritmi polinomiali;*
  - *Un algoritm se numeşte polinomial dacă termenul dominant are forma Cn^k, adică $Q(n)=Cn^k$; $T(n)=Cn^k\ \Delta$, unde n este caracteristica datelor de intrare, C – o constantă pozitivă, iar k – un număr natural.*
  - *Complexitatea temporală a algoritmilor polinomiali este redată prin notaţia $O(n^k)$, care se citeşte „algoritm cu timpul de calcul de ordinul n^k" sau, mai scurt, „algoritm de ordinul n^k".*
  - *Evident, există algoritmi polinomiali de ordinul n, n^2, n^3 ş.a.m.d. De exemplu, algoritmul de sortare a elementelor unui vector prin metoda bulelor este un algoritm polinomial de ordinul n^2.*
- *algoritmi exponenţiali;*
  - *Un algoritm se numeşte exponenţial dacă termenul dominant are forma Ck^n, adică $Q(n)= Ck^n$; $T(n)= Ck^n\ \Delta$, unde $k>1$.*
  - *Complexitatea temporală a algoritmilor exponenţiali este redată prin notaţia $O(k^n)$. Menţionăm că tot exponenţiali se consideră şi algoritmii de complexitatea $n^{\log n}$, cu toate că această funcţie nu este exponenţială în sensul stric matematic al acestui termen.*
- *algoritmi nedeterminişti polinomiali.*

În practică, elaborarea programelor pe calculator presupune parcurgerea următoarelor etape:

- *formularea exactă a problemei;*
- *determinarea complexităţii temporale a problemei propuse - problemă uşor rezolvabilă sau dificilă;*
- *elaborarea algoritmului respectiv şi implementarea lui pe un sistem de calcul.*

Evident, în cazul unor probleme uşor rezolvabile, programatorul va depune toate eforturile pentru a inventa algoritmi polinomiali, adică algoritmi de ordinul n^k, astfel încât parametrul k să ia valori cât mai mici. În cazul problemelor dificile se va da prioritate algoritmilor care minimizează timpul de calcul cel puţin pentru datele de intrare frecvent utilizate în aplicaţiile practice ***[13]****.*

- ***Timp constant***
  - *Se spune despre un algoritm că este în timp constant (sau timp O(1)) dacă valoarea lui T(n) este delimitată de o valoare care nu depinde de mărimea datelor de intrare. De exemplu, accesarea unui singur element într-un tablou are nevoie de timp constant întrucât trebuie să fie efectuată o singură operațiune pentru a-l localiza. La fel și găsirea valorii minime într-un tablou sortat în ordine crescătoare – este primul element. Cu toate acestea, găsirea valorii minime într-un tablou neordonat nu este constantă de timp, întrucât este nevoie de o parcurgere a fiecărui element din tablou, în scopul de a determina valoarea minimă. Prin urmare, aceasta este o operațiune în timp liniar, care durează O(n). Dacă numărul de elemente este cunoscut în avans și nu se schimbă, cu toate acestea, un astfel de algoritm poate fi considerat a rula în timp constant.*
  - *În ciuda denumirii de „timp constant", timpul de rulare nu trebuie să fie independent de dimensiunea problemei, ci o limită superioară a timpului de rulare, trebuie să fie delimitată independent de dimensiunea problemei. De exemplu, sarcina de a „schimba valorile a și b dacă este necesar, astfel încât a≤b" este considerat a rula în timp constant, chiar dacă durata efectivă poate depinde dacă este sau nu este deja adevărat că a ≤ b. Cu toate acestea, există unele constante t astfel încât timpul necesar este întotdeauna cel mult t.*
- ***Timp logaritmic***
  - *Algoritmii care au durata de execuție în timp logaritmic se întâlnesc de obicei în operații pe arbori binari sau atunci când se utilizează căutarea binară. Un algoritm O(log n) este considerat extrem de eficient, deoarece numărul de operațiuni pe instanță necesar pentru a finaliza rularea scade la fiecare instanță.*
  - *Un exemplu foarte simplu de acest tip este un algoritm care taie un șir în jumătate, apoi taie jumătatea din dreapta în jumătate, și așa mai departe. El va dura O(log n) (n fiind lungimea șirului), așadar șirul se taie în jumătate înainte de fiecare tipărire. Aceasta înseamnă că, pentru a crește numărul de tipăriri cu 1, trebuie dublată lungimea șirului.*
- ***Timp polilogaritmic***
  - *Un algoritm este considerat a fi în timp polilogaritmic dacă $T(n) = O((\log n)^k)$, pentru un k constant. De exemplu, ordonarea lanțului de matrice poate fi rezolvată în timp polilogaritmic pe o mașină cu acces aleator paralel.*
- ***Timp subliniar***
  - *Se spune că un algoritm rulează în timp subliniar dacă T(n) = o(n). În particular, aici sunt incluși algoritmii cu complexitățile definite mai sus, precum și alții cum ar fi căutarea lui Grover de complexitate $O(n^{1/2})$.*
  - *Algoritmii tipici care sunt exacți și, totuși, rulează în timp subliniar folosesc prelucrări paralele (așa cum face algoritmul NC1 de calcul al determinantului matricei), prelucrări neclasice (de exemplu, căutarea lui Grover), sau li se garantează unele presupuneri despre structura datelor de intrare (cum ar fi căutarea binară și mulți algoritmi de mentenanță a arborilor, care rulează în timp logaritmic). Limbajele formale însă, așa cum este mulțimea tuturor șirurilor care au un bit de 1 pe poziția indicată de primii log(n) biți ai șirului, ar putea depinde de toți biții intrării, dar tot ar putea fi calculați în timp subliniar.*
  - *Termenul specific, algoritm în timp subliniar este de regulă rezervat algoritmilor diferiți de cei de mai sus prin aceea că sunt rulați peste modele seriale clasice de mașină și nu au dreptul la prezumții despre datele de intrare. Ei pot însă să fie randomizați și, chiar trebuie, pentru orice task (sarcină) netrivial (ă). Cum un astfel de algoritm trebuie să dea un răspuns fără a-și citi datele de intrare în întregime, detaliile lui depind masiv de accesul permis asupra intrării. De regulă, pentru o intrare reprezentată de un șir binar b1,...,bk se presupune că algoritmul poate cere și primi într-un timp O(1) valoarea lui bi oricare ar fi i.*

Observații:

- ❖ *Din comparația vitezei de creștere a funcțiilor exponențială și polinomială rezultă că algoritmii exponențiali devin inutilizabili chiar pentru valori nu prea mari ale lui n.*
- ❖ *În funcție de complexitatea temporală a algoritmilor se consideră că o problemă este ușor rezolvabilă dacă pentru soluționarea ei există un algoritm polinomial. O problemă pentru care nu există un algoritm polinomial se numește dificilă* ***[14]****.*

1.6 Clasa NP. Algoritmi nedeterminişti.

Algoritmii cu care suntem obişnuiţi să lucrăm zi de zi sunt determinişti. Asta înseamnă că la un moment dat evoluţia algoritmului este unic determinată, şi că instrucţiunea care urmează să se execute este unic precizată în fiecare moment. Acest tip de algoritmi este surprinzător de bogat în consecinţe cu valoare teoretică. Aceşti algoritmi nu sunt direct aplicabili, însă studiul lor dă naştere unor concepte foarte importante.

Surprinzătoare este şi definiţia corectitudinii unui astfel de algoritm. Un algoritm nedeterminist este corect dacă există o posibilitate de executare a sa care găseşte răspunsul corect. Pe măsură ce un algoritm nedeterminist se execută, la anumiţi paşi se confruntă cu alegeri nedeterministe. Ei bine, dacă la fiecare pas există o alegere, care făcută să ducă la găsirea soluţiei, atunci algoritmul este numit corect.

Pe scurt algoritmul se comportă aşa: dacă la nord nu e perete mergi încolo, sau, poate, dacă la sud e liber, mergi încolo, sau la est, sau la vest. În care dintre direcţii, nu se precizează (este ne-determinat). Este clar că dacă există o ieşire la care se poate ajunge, există şi o suită de aplicări ale acestor reguli care duce la ieşire.

Utilitatea practică a unui astfel de algoritm nu este imediat aparentă: în definitiv pare să nu spună nimic util: soluţia este fie spre sud, fie spre nord, fie spre este, fie spre vest. Ei şi? Este clar că aceşti algoritmi nu sunt direct implementabili pe un calculator real.

În realitate, existenţa unui astfel de algoritm deja înseamnă destul de mult. Înseamnă, în primul rând, că problema se poate rezolva algoritmic; vă reamintesc că există probleme care nu se pot rezolva deloc. În al doilea rând, se poate arăta că fiecare algoritm nedeterminist se poate transforma într-unul determinist într-un mod automat.

Deci, de îndată ce ştim să rezolvăm o problemă într-un mod nedeterminist, putem să o rezolvăm şi determinist! Transformarea este relativ simplă: încercăm să mergem pe toate drumurile posibile în paralel, pe fiecare câte un pas. (O astfel de tehnică aplicată în cazul labirintului se transformă în ceea ce se cheamă "flood fill": evoluez radial de la poziţia de plecare în toate direcţiile).

Fiind dată o anumită problemă, se pune întrebarea: există un algoritm de rezolvare a ei polinomial?

La această întrebare se poate răspunde în felul următor: există probleme pentru care nu se cunosc algoritmi de rezolvare în timp polinomial. Matematicienii nu au avut curajul să facă o afirmaţie de genul "nu există algoritmi polinomiali de rezolvare a acestor probleme", pentru că o astfel de afirmaţie presupune o demonstraţie, care nu s-a făcut. Faptul că nu s-a găsit un algoritm polinomial nu înseamnă că el nu există.

Algoritmii cu care suntem obişnuiţi să lucrăm zi de zi sunt determinişti. Asta înseamnă că la un moment dat evoluţia algoritmului este unic determinată, şi că instrucţiunea care urmează să se execute este unic precizată în fiecare moment. Acest tip de algoritmi este surprinzător de bogat în consecinţe cu valoare teoretică. Aceşti algoritmi nu sunt direct aplicabili, însă studiul lor dă naştere unor concepte foarte importante.

*Surprinzătoare este şi definiţia corectitudinii unui astfel de algoritm. Un algoritm nedeterminist este corect dacă există o posibilitate de executare a sa care găseşte răspunsul corect. Pe măsură ce un algoritm nedeterminist se execută, la anumiţi paşi se confruntă cu alegeri nedeterministe. Ei bine, dacă la fiecare pas există o alegere, care făcută să ducă la găsirea soluţiei, atunci algoritmul este numit corect **[15]**.*

Deci, de îndată ce ştim să rezolvăm o problemă într-un mod nedeterminist, putem să o rezolvăm şi determinist! Transformarea este relativ simplă: încercăm să mergem pe toate drumurile posibile în paralel, pe fiecare cîte un pas.

Clasa tuturor problemelor care se pot rezolva cu algoritmi nedeterminişti într-un timp polinomial se notează cu NP (Nedeterminist Polinomial). Este clar că orice problemă care se află în P se află şi în NP, pentru că algoritmii determinişti sunt doar un caz extrem al celor determinişti: în fiecare moment au o singură alegere posibilă.

Din păcate, transformarea într-un algoritm determinist se face pierzând din eficienţă. În general un algoritm care operează în timp nedeterminist polinomial (NP) poate fi transformat cu uşurinţă într-un algoritm care merge în timp exponenţial (EXP). Avem deci o incluziune de mulţimi între problemele de decizie: $P \subseteq NP \subseteq EXP$.

Partea cea mai interesantă este următoarea: ştim cu certitudine că $P \neq EXP$. Însă nu avem nici o idee despre relaţia de egalitate între NP şi P sau între NP şi EXP. Nu există nici o demonstraţie care să infirme că problemele din NP au algoritmi eficienţi, determinist polinomiali! Problema P=NP este cea mai importantă problemă din teoria calculatoarelor, pentru că de soluţionarea ei se leagă o mulţime de consecinţe importante.

Problema aceasta este extrem de importantă pentru întreaga matematică, pentru că însăşi demonstrarea teoremelor este un proces care încearcă să verifice algoritmic o formulă logică. Teoremele la care există demonstraţii « scurte » pot fi asimilate cu problemele din mulţimea NP. Dacă orice problemă din NP este şi în P, atunci putem automatiza o mare parte din demonstrarea de teoreme în mod eficient!

Problema P=NP este foarte importantă pentru criptografie: decriptarea este o problemă din NP (cel care ştie cheia, ştie un algoritm determinist polinomial de decriptare, dar cel care nu o ştie are în faţă o problemă pe care nedeterminist o poate rezolva în timp polinomial).

Dacă s-ar demonstra că P=NP acest lucru ar avea consecinţe extrem de importante, iar CIA (Central Intelligence Agency) și KGB (Russian: Комите́т Госуда́рственной Безопа́сности (КГБ), tr. Komitet Gosudarstvennoy Bezopasnosti) ar fi într-o situaţie destul de proastă, pentru că toate schemele lor de criptare ar putea fi sparte în timp polinomial (asta nu înseamnă neapărat foarte repede, dar oricum, mult mai repede decât în timp exponenţial)!

În 1971 Stephen Arthur Cook (n. 14 decembrie 1939, Buffalo, New York, SUA) a demonstrat că există o problemă specială în NP, pe care a numit-o NP- completă: Problema satisfacerii (problema problemelor).

Să considerăm o funcţie booleană cu n variabilele $(x_1,x_2,...,x_n)$, dată în forma canonică conjunctivă. Se cere un sistem de valori $x_1,x_2,...,x_n$, astfel încît, pentru acest sistem, funcţia să ia valoarea True. Pentru aceasta, Cook a primit în 1982 Premiul Turing.

Oricât ne străduim să procedăm altfel, trebuie să încercăm toate sistemele de valori pentru a vedea ce valoare ia funcţia, dar avem 2^n astfel de sisteme.

În concluzie, pentru această problemă vom avea timpul de calcul $O(2^n)$. Mai mult, se demonstrează că alte probleme se reduc la aceasta (există algoritm în timp polinomial care transformă o altă problemă în problema satisfacerii) ***[16]****.*

Dacă am fi capabili să rezolvăm această problemă în timp polinomial, am rezolva în timp polinomial o întreagă clasă de probleme care se reduc la aceasta. Următoarele tipuri de probleme se reduc la problema satisfacerii:

- *Problema colorării hărţilor;*
- *Problema comis-voiajorului;*
- *Problema celor n dame;*
- *Problema rucsacului.*

O mulţime de probleme cerute de practică se reduc la problema satisfacerii (de aici şi numele de problema problemelor). Cele prezentate mai sus au o semnificaţie enormă! Înseamnă că nu se poate (pe moment cel puţin) rezolva orice cu ajutorul calculatorului electronic. Mai mult, perfecţionările tehnologice aduse acestor maşini, oricât de spectaculoase vor fi, nu vor rezolva această problemă.

Ce se poate face, în absenţa unei rezolvări în timp polinomial pentru această categorie de probleme? În multe cazuri, se renunţă la optimalitatea unei soluţii (obţin o soluţie bună, dar nu cea mai bună) cu condiţia că acestea să fie obţinute în timp polinomial. Astfel de metode se numesc euristice. Nu este deloc uşor să se imagineze astfel de metode. Mai ales, este foarte dificil de arătat cât de departe de soluţia optimă este soluţia găsită.

Au apărut metode noi de cercetare, pentru obţinerea unor soluţii cât mai bune, chiar dacă nu optime: greedy euristic, algoritmi genetici, algoritmi neuronali, algoritmi de tip călire etc. Astfel de probleme sunt de cea mai mare actualitate în informatica teoretică pe plan mondial.

1.7 Algoritmi aleatori: Markov, Las Vegas şi Monte Carlo

1. *Noțiuni introductive*

Algoritmii aleatori se împart în principal în 2 clase:

- *Algoritmi care rezolvă probleme de optim: soluția calculată de algoritm este garantat corectă, dar este aproximativă (nu este optimală). În acest caz, soluția suboptimală este considerată acceptabilă având o marjă de aproximare controlată probabilistic – algoritmi de aproximare, algoritmi genetici și algoritmi aleatori de tip Las Vegas;*
- *Algoritmi care rezolvă o problemă ce acceptă o singură soluție: se renunță la exactitatea rezolvării preferându-se o soluție rapidă care se apropie cu o probabilitatea suficient de mare de soluția exactă – corectitudinea nu este garantată – algoritmii aleatori de tip Monte Carlo și stocastici (Markov).*

Printre implicațiile practice ale algoritmilor aleatori se numără:

- *optimizarea diverșilor algoritmi, în general în vederea asigurării dispersiei corespunzătoare a valorilor;*
- *diverse inițializări (ex. Algoritmi Genetici pentru indivizi) sau selecții de date după o distribuție prestabilită (în general Gaussiană), dar și reducerea complexității unor probleme specifice.*

Primele aspecte care trebuie clarificate sunt caracteristicile algoritmilor aleatori:

- *necesitatea micșorării timpului de rezolvare a problemei prin relaxarea restricțiile impuse soluțiilor;*
- *este suficientă o singură soluție care se apropie cu o probabilitate măsurabilă de soluția exactă, în final se poate obține o soluție suboptimală cu o marjă de eroare garantată prin calcul probabilistic.*

În informatica teoretică, un algoritm Markov este un sistem de rescriere a șirurilor care folosește reguli similare gramaticale pentru a opera pe șiruri de simboluri. Algoritmii Markov s-au dovedit a fi Turing-complet, ceea ce înseamnă că sunt adecvați ca un model general de calcul și pot reprezenta orice expresie matematică din notarea sa simplă. Algoritmii Markov poartă numele matematicianului sovietic Andrey Markov, Jr.

2. *Algoritmi Las Vegas*

Evoluţia unui algoritm care foloseşte numere aleatoare nu mai depinde numai de datele de intrare, ci şi de numerele aleatoare pe care le generează. Dacă are ” noroc” algoritmul poate termina repede şi bine; dacă dă prost cu zarul, ar putea eventual chiar trage o concluzie greşită. În cazul quick-sort, răspunsul este întotdeauna corect, dar câte o dată vine mai greu. Aceasta este diferenţa dintre algoritmii Monte Carlo, mereu corecţi, şi cei Las Vegas, care pot uneori, rar, greşi. Vom defini acum algoritmii probabilişti pentru probleme de decizie.

Majoritatea problemelor pot fi exprimate în forma unor probleme de decizie, deci simplificarea nu este prea drastică. Există două clase de algoritmi probabilişti, dar ne vom concentra atenţia numai asupra uneia dintre ele. Vom defini totodată clasa problemelor care pot fi rezolvate probabilist în timp polinomial, numită RP (Random Polinomial). Dacă indicăm de la început care sunt numerele aleatoare care vor fi generate, evoluţia algoritmului este perfect precizată.

***Definiţie:** O problemă de decizie este în RP dacă există un algoritm aleator A care rulează într-un timp polinomial în lungimea instanţei (n^c pentru un c oarecare), şi care are următoarele proprietăţi:*

- *Dacă răspunsul la o instanţă I este „Da”, atunci cu o probabilitate mai mare de 1/2 algoritmul va răspunde „Da”.*
- *Dacă răspunsul la o instanţă I este ”Nu”, atunci cu probabilitate 1 algoritmul va răspunde ”Nu”.*
- *De cine este dată „probabilitatea” de mai sus? De numărul de şiruri aleatoare.*

Când rulăm un algoritm aleator pentru o instanţă I, avem la dispoziţie 2^{nc} şiruri aleatoare de biţi; pentru unele dintre ele algoritmul răspunde „Da”, pentru celelalte răspunde ”Nu”. Ceea ce facem este să numărăm pentru câte şiruri algoritmul ar răspunde „Da” şi să facem raportul cu 2^{nc}. Aceasta este probabilitatea ca algoritmul să răspundă „Da”. Opusul ei este probabilitatea să răspundă ”Nu”.

O tehnică foarte simplă de amplificare poate creşte nedefinit această probabilitate: dacă executăm algoritmul de 2 ori pe aceleaşi date de intrare, probabilitatea de a greşi pentru răspunsuri ”Nu” rămîne 0. Pe de altă parte, ajunge ca algoritmul să răspundă măcar o dată „Da” pentru a şti că răspunsul este „Da” cu siguranţă! Din cauza asta, dacă probabilitatea de eroare este p pentru algoritm, executând de k ori probabilitatea coboară la p^k (nu uitaţi că p este subunitar, ba chiar sub 1/2). Această metodă se numeşte “boost”' în engleză, şi face dintr-un algoritm probabilist slab ca discriminare un instrument extrem de puternic ***[17]****.*

Caracteristicile algoritmilor de tip Las Vegas

- *Determină soluția corectă a problemei, însă timpul de rezolvare nu poate fi determinat cu exactitate;*
- *Creșterea timpului de rezolvare implică creșterea probabilității de terminare a algoritmului;*
- *După un timp infinit se ajunge la soluția optimă și algoritmul se termină sigur;*
- *Probabilitatea de găsire a soluției crește extrem de repede încât să se determine soluția corectă într-un timp suficient de scurt.*

3. Monte Carlo

O tehnică foarte spectaculoasă pentru rezolvarea problemelor este cea a folosirii numerelor aleatoare. Practic algoritmii aleatori sunt identici cu cei obişnuiţi, dar folosesc în plus o nouă instrucţiune, care s-ar putea chema “dă cu banul”. Această instrucţiune generează un bit arbitrar ca valoare. În mod paradoxal, incertitudinea ne poate oferi mai multă putere ...

La ce se foloseşte aleatorismul? Să ne amintim că în general complexitatea unei probleme este definită luând în considerare cele mai defavorabile date iniţiale. De exemplu, pentru problema comis voiajorului, faptul că această problemă este NP- completă nu înseamnă că nu putem rezolva nici o instanţă (un set de date iniţiale) a ei, ci că există instanţe pentru care algoritmii cunoscuţi nu au prea multe şanse să se termine în curând.

Acest lucru este adevărat şi pentru alte clase de algoritmi; de pildă algoritmul quick-sort are pentru majoritatea vectorilor de intrare o comportare O(n log n). Dacă însă datele de intrare sunt prost distribuite, atunci quick-sort poate face n^2 comparaţii. Pentru n=100 asta înseamnă de 10 ori mai mult! Numărul de instanţe pentru care quick- sort este slab, este mult mai mic decât numărul de instanţe pentru care merge bine. Cum procedăm atunci când într-un anumit context lui quick-sort i se dau numai date rele? O soluţie paradoxală constă în a amesteca aleator vectorul înainte de a-l sorta.

Complexitatea medie (average case) a lui quick-sort este O(n log n). Complexitatea în cazul cel mai rău (worst case) este $O(n^2)$. Dacă datele vin distribuite cu probabilitate mare în zona “rea”, atunci amestecându-le putem transforma instanţe care pică în zona „worst-case” în instanţe de tip “average-case”. Fireşte, asta nu înseamnă că nu putem avea ghinion, şi că amestecarea să producă tot o instanţă “rea”, dar probabilitatea ca acest lucru să se întâmple este foarte mică, pentru că quick-sort are puţine instanţe rele. Acesta este un caz de folosire a aleatorului pentru a îmbunătăţi performanţa medie a unui algoritm. Câteodată câştigul este şi mai mare, pentru că putem rezolva probleme NP- complete foarte rapid folosind aleatorismul ***[18]****.*

Caracteristicile algoritmilor de tip Monte Carlo

- *Determină o soluție a problemei care e garantat corectă doar după un timp infinit de rezolvare – soluție aproximativă;*
- *Presupun un număr finit de iterații după care răspunsul nu este garantat corect;*
- *Creșterea timpului de rezolvare implică creșterea probabilității ca soluția găsită să fie corectă;*
- *Soluția găsită într-un timp acceptabil este aproape sigur corectă (există o probabilitate mică ca soluţia să nu fie corectă).*

METODA RECURSIVĂ

2.1 Noțiuni generale despre recursivitate

În matematică și informatică, recursivitatea sau recursia este un mod de a defini unele funcții. Funcția este recursivă, dacă definiția ei folosește o referire la ea însăși, creând la prima vedere un cerc vicios, care însă este numai aparent, nu și real. Nu toate funcțiile matematice pot fi definite recursiv, cu alte cuvinte există și funcții nerecursive.

În matematică și informatică recursivitatea funcționează prin definirea unuia sau a mai multor cazuri de bază, foarte simple, apoi prin definirea unor reguli prin care cazurile mai complexe se reduc la cazuri mai simple.

Exemple:

1. *Definirea formală a numerelor naturale din cadrul teoriei mulțimilor:*
   - *baza recursiei este faptul că 1 este număr natural;*
   - *în plus, orice număr natural are un succesor, care este de asemenea un număr natural.*
2. *Definirea conceptului de strămoș al unei persoane:*
   - *Un părinte este strămoșul copilului ("baza");*
   - *Părinții unui strămoș sunt și ei strămoși ("pasul de recursie").*
3. *Definirea recursivă este următoarea: 'Un buchet de flori este fie (1) o floare, fie (2) o floare adăugată buchetului'.*
   - *Afirmatia (1) servește ca și condiție inițială, indicând maniera de amorțare a definiției;*
   - *Afirmatia (2) precizează definirea recursivă propriu-zisă.*
   - *Varianta iterativă a aceleiași definiții este: 'Un buchet de flori consta fie dintr-o floare, fie din două, fie din 3 flori, fie etc". După cum se observă, definiția recursivă este simplă și elegantă, dar oarecum indirectă, în schimb definiția iterativă este directă.*

Notă:

- ✓ *O definiție recursivă se referă la un obiect care se definește ca parte a propriei sale definiri. Desigur o definiție de genul 'o floare este o floare' care poate reprezenta în poezie un univers întreg, în știință, în general și în matematică, în special nu furnizează prea multe informații despre floare.*
- ✓ *O caracteristică foarte importantă a recursivității este aceea de a preciza o definiție într-un sens evolutiv, care evită circularitatea.*
- ✓ *Despre un obiect se spune ca este recursiv, dacă el constă sau este definit prin el însuși. Prin definiție orice obiect recursiv implică recursivitatea ca și proprietate intrinsecă a obiectului în cauză.*
- ✓ *Recursivitatea este utilizată cu multă eficiență în matematică, spre exemplu: în definirea numerelor naturale, a structurilor de tip arbore sau a anumitor funcții.*

Definiție 1:

Recursivitate este proprietatea funcțiilor de a se autoapela:

- *din afara subprogramului se face un prim apel al acestuia;*
- *programul se auto-apelează de un anumit număr de ori;*
- *la fiecare nouă auto-apelare a algoritmului, se execută din nou secvența de instrucțiuni ce reprezintă corpul său, cu alte date => înlănțuire.*

Definiție 2:

Dacă apelul subprogramului apare chiar în corpul său, recursivitatea se numește directă, altfel se numește indirectă.

Observații:

- ❖ *În corpul algoritmului trebuie să existe cel puțin o testare a unei condiții de oprire, la îndeplinirea căreia se întrerupe lanțul de auto-apeluri.*

- *Majoritatea algoritmilor repetitivi se pot implementa atât în varianta nerecursivă (iterativă), cât și în varianta recursivă.*
- *Varianta recursivă este recomandată, în special, pentru probleme definite prin relații de recurență.*
- *Algoritmii recursivi sunt, în general, greu de urmărit (depinde), necesită timp de execuție mai lung și spațiu de memorie mai mare.*

ÎNTREBĂRI CU RĂSPUNSURI

1. Care este condiția de bază a recursiei?

În programul recursiv, soluția cazului de bază este oferită și soluția problemei mai mari este exprimată în termeni de probleme mai mici.

```
int fact(int n){
    if (n < = 1) // cazul de bază
        return 1;
    else
        return n*fact(n-1);
}
```

În exemplul de mai sus, se definește cazul de bază pentru n <= 1 și se poate rezolva o valoare mai mare a numărului prin conversia la unul mai mic până când se ajunge la cazul de bază.

2. Cum se rezolvă o anumită problemă folosind recursivitate?

Ideea este de a reprezenta o problemă în termeni de una sau mai multe probleme mai mici și de a adăuga una sau mai multe condiții de bază care opresc recursiunea.

De exemplu, calculăm factorial n dacă cunoaștem factorial al lui (n-1). Cazul de bază pentru factorial ar fi n = 0. Revenim 1 când n = 0.

3. De ce apare o eroare de supraîncărcare în stivă?

Dacă nu este abordat cazul sau nu este definit, atunci poate apărea problema supraîncărcării stivei. Să luăm un exemplu pentru a înțelege acest lucru.

```
int fact(int n){
    // condiție de bază incorectă,poate provoca supraîncărcarea stivei
    if (n == 100)
        return 1;
    else
        return n*fact(n-1);
}
```

*Dacă fact (10), se va numi fact (9), fact (8), fact (7) și așa mai departe, dar numărul nu va ajunge niciodată la 100. Deci, cazul de bază nu este realizat. Dacă memoria este epuizată de aceste funcții pe stivă, aceasta va cauza o eroare de supraîncărcare a stivei **[19]**.*

2.2 Analiza complexității algoritmului metodei recursive

1. Cum numărăm pașii pe care îi execută un algoritm?

Măsurăm complexitatea ca pe o funcție de dimensiunea datelor de intrare: T:N->R_+. Valoarea T(n) va reprezenta estimarea numărului de pași pe care îi face algoritmul atunci când dimensiunea datelor de intrare este n.

Numărarea este în general simplă pe algoritmi nerecursivi, ceea ce trebuie să facem în general este să identificăm operațiile critice și să vedem de câte ori se execută acestea. Când algoritmul este nerecursiv, pur și simplu îl luăm de la cap la coadă și numărăm operațiile critice, înmulțim când ele se află într-unul sau mai multe cicluri.

Ceva mai dificil este cu algoritmii recursivi, nu este mereu clar sau ușor de determinat câte apeluri recursive să execute.

2. Cum determinăm complexitatea pe un algoritm recursiv?

Exemplul 1:

Pentru un număr întreg x>0 să se calculeze x^n, unde $n \geq 0$. Vom considera funcția $putere(x,n): N \times N \to N$. *Evident* $putere(x,0)=1$. *Atunci expresia recursivă este aceasta:*

$$putere(x,n)=\begin{cases} 1, dacă\ n=0; \\ x \cdot putere(x,n-1),\ dacă\ n \geq 1. \end{cases}$$

Corectitudinea definiției rezultă din $x^0=1$ *(pasul de verificare) și* $x^{n+1}=x \cdot x^n$ *(pasul de inducție). Afirmăm că funcția (1) are complexitatea O (n). Notăm cu T (n) timpul de execuţie a funcţiei putere.*

- *T (n) = T (n − 1) + O (1), unde O (1) este timpul necesar pentru a returna valoarea x .*
- *T (n) = T (n − 1) + O (1) = T (n − 2) + 2 ∗ O (1) +.... +T (n) = T (0) + n ∗ O (1).*

Rezultă complexitatea O (n).

Exemplul 2:

Exprimând altfel funcţia putere, vom obţine un algoritm cu o complexitate mai bună. Observăm că x^4 se poate calcula mai ușor decât am calculat x^2, deoarece $x^4=(x^2)^2$. La fel $x^6=(x^3)^2$. Vom nota n împărțit la 2 cu n/2. Considerăm că $x^{n/2} = putere(x, n/2)$ și notăm cu pow(x,2) ridicarea la puterea a doua. Atunci:

$$pow(putere(x,n/2),2)=\begin{cases} x^{n-1}, dacă\ n\ impar; \\ x^n,\ dacă\ n\ par. \end{cases} \Rightarrow x^n=\begin{cases} x \cdot pow(putere(x,n/2),2), dacă\ n\ impar; \\ pow(putere(x,n/2),2),\ dacă\ n\ par. \end{cases}$$

Algoritmul complet este:

$$putere(x,n)=\begin{cases} 1, dacă\ n=0; \\ x \times pow(putere(x,n/2),2), dacă\ n\ impar; \\ pow(putere(x,n/2),2),\ dacă\ n\ par. \end{cases}$$

Notăm cu T (n) timpul de execuţie pentru această formulă.

Fie $n=2^m$, *atunci*: $T(n)=\begin{cases} 1, dacă\ n=0; \\ T(\frac{n}{2}),\ dacă\ n \geq 1. \end{cases}$

Aplicarea formulei recursive:

$$T(n)=T(2^m)=T(2^{m-1})+1=T(2^{m-2})+2=\ldots=T(2^0)+m=T(1)+m=1+m=\log_2 n+1.$$

Rezultă complexitatea O ($\log_2 n$ *) [20].*

2.3 Forme specifice ale metodei recursive

1. *Recursivitatea directă*

Definiție 1:
O funcție F în corpul căreia există cel puțin un auto-apel (F apelează pe F), se numește funcție direct recursivă.

În limbajul C++ funcțiile se pot apela pe ele însele, adică sunt direct recursive. Pentru o funcționare corectă (din punct de vedere logic), apelul recursiv trebuie să fie condiționat de o decizie, care la un moment dat în cursul execuției să împiedice continuarea apelurilor recursive și să permită astfel revenirea din șirul de apeluri.

Lipsa acestei condiții sau programarea ei greșită va conduce la executarea unui șir de apeluri a cărui terminare nu mai este controlată prin program și care, la epuizarea resurselor sistemului va provoca o eroare de execuție: depășirea stivei de date.

2. *Recursivitatea indirectă*

Definiție 2:
Două funcţii F şi G se numesc funcţii indirect recursive, dacă se apelează reciproc, F apelează G şi G apelează F.

Pentru a putea fi executată, orice funcţie trebuie să fie scrisă înaintea modului apelant. Dacă dorim să scriem funcţia după modulul apelant, atunci trebuie să îi dăm prototipul înaintea modulului apelant. Presupunem că avem două funcţii A şi B care se apelează reciproc, scrise în această ordine. Funcţia A apelează funcţia B, dar modulul apelat B se află după cel apelant A. În consecinţă, pentru ca acest apel să poată fi executat, trebuie să dăm prototipul funcţiei B. Prototipul unei funcţii este o reproducere a antetului său, cu două deosebiri:

- *între paranteze, în lista parametrilor formali nu se mai scriu şi identificatorii parametrilor, ci doar se enumeră tipurile acestora, separate prin "virgulă" ;*
- *prototipul se încheie cu caracterul "punct şi virgulă"* ***[21]****.*

Exemplu:
Să se calculeze valoarea funcţiilor matematice f(x) şi g(x), pentru o valoare a argumentului x introdusă de la tastatură.

$$f(x)=\begin{cases}3+g(x), \text{ pentru } x<3\\ x^2, \text{ pentru } x>3\end{cases} \qquad g(x)=\begin{cases}5-f(x+1), \text{ pentru } x<0\\ x+f(x), \text{ pentru } x>0\end{cases}$$

Implementarea în limbajul C++ a recursiei indirecte:

```
#include <iostream>
using namespace std;
int x; int G (int x);
int F (int x){
    if (x>3) return x*x ;
    else return 3 + G (x) ;
}
int G (int x){
    if  (x<0) return 5 - F (x+1);
    else return x + F (x) ;
}
int main (){
    cout << " x= " ; cin >> x ;
    cout << " f ( " << x << " )= " << F (x) << " \n" ;
    cout << " g ( " << x << " )= " << G (x);
}
```

2.4 Implementarea metodelor recursive și iterative

Problema 1: Factorialul unui număr – 1, 2, 6, 24, 120, ...

Să se determine factorialul unui număr. Să se alcătuiască varianta recursivă şi iterativă în care va afișa rezultatul pentru factorial.

Exemplu:
Fie n=5.

$1! = 1$
$2! = 1 \cdot 2 = 2$
$3! = 1 \cdot 2 \cdot 3 = 6$
$4! = 1 \cdot 2 \cdot 3 \cdot 4 = 24$
$5! = 1 \cdot 2 \cdot 3 \cdot 4 \cdot 5 = 120$

$1! = 1$
$2! = 1! \cdot 2 = 1 \cdot 2 = 2$
$3! = 2! \cdot 3 = 2 \cdot 3 = 6$
$4! = 3! \cdot 4 = 6 \cdot 4 = 24$
$5! = 4! \cdot 5 = 24 \cdot 5 = 120$

Formula recursivă:

$$Factorial(n) = \begin{cases} 1, \text{ pentru } n = \overline{0,1}; \\ Factorial(n-1) \cdot n, \text{ pentru } n > 1. \end{cases}$$

Varianta recursivă	Varianta iterativă
`#include <iostream>` `#include <graphics.h>` `using namespace std;` `long int factorial (int n) {` `  if (n==1) return 1;` `  else return n*factorial(n-1);` `}` `int main(){` `  system("color F0");` `  int n;` `  cout<<"VARIANTA RECURSIVA\n\n";` `  cout<<"Introduceti datele de intare, n= "; cin>>n;` `  if(!n) cout<<"0!=1\n";` `  else cout<<n<<"!= "<<factorial(n)<<"\n";` `  return 0;` `}`	`#include <iostream>` `#include <graphics.h>` `using namespace std;` `long int factorial (int n) {` `  long int f=1;` `  for (int i=1; i<=n;i++)` `  f=f*i;` `  return f;` `}` `int main(){` `  system("color F0");` `  int n;` `  cout<<"VARIANTA ITERATIVA\n\n";` `  cout<<"Introduceti datele de intare, n= "; cin>>n;` `  if(!n) cout<<"0!=1\n";` `  else cout<<n<<"!= "<<factorial(n)<<"\n";` `  return 0;` `}`
Rezultatele execuției: `VARIANTA RECURSIVA` `Introduceti datele de intare, n= 5` `5!= 120`	***Rezultatele execuției:*** `VARIANTA ITERATIVA` `Introduceti datele de intare, n= 5` `5!= 120`

Problema 2: Factorialul modificat al unui număr – 1, 4, 24, 192, 1920, ...

Să se determine factorialul unui număr. Să se alcătuiască varianta recursivă şi iterativă în care va afişa rezultatul pentru factorialul modificat.

Exemplu:
Fie n=5.

$$1!=1$$
$$2!=1\cdot(2\cdot 2)=4$$
$$3!=1\cdot 2\cdot 2\cdot(2\cdot 3)=24$$
$$4!=1\cdot 2\cdot 2\cdot 2\cdot 3\cdot(2\cdot 4)=192$$
$$5!=1\cdot 2\cdot 2\cdot 2\cdot 3\cdot 2\cdot 4\cdot(2\cdot 5)=1920$$

$$1!=1$$
$$2!=1!\cdot(2\cdot 2)=1\cdot 4=4$$
$$3!=2!\cdot(2\cdot 3)=4\cdot 6=24$$
$$4!=3!\cdot(2\cdot 4)=24\cdot 8=192$$
$$5!=4!\cdot(2\cdot 5)=192\cdot 10=1920$$

Formula recursivă:

$$Factorial(n)=\begin{cases} 1, \text{ pentru } n=\overline{0,1}; \\ Factorial(n-1)\cdot 2\cdot n, \text{ pentru } n>1. \end{cases}$$

Varianta recursivă	Varianta iterativă
`#include <iostream>` `#include <graphics.h>` `using namespace std;` `long int factorial (int n) {` `  if (n==1) return 1;` `  else return 2*n*factorial(n-1);` `}` `int main(){` `  system("color F0");` `  int n;` `  cout<<"VARIANTA RECURSIVA\n\n";` `  cout<<"Introduceti datele de intare, n= "; cin>>n;` `  if(!n) cout<<"0!=1\n";` `  else cout<<n<<"!= "<<factorial(n)<<"\n";` `  return 0;` `}`	`#include <iostream>` `#include <graphics.h>` `using namespace std;` `int factorial (int n) {` `  int f=1;` `  for (int i=1; i<=n;i++)` `  f*=2*i; f/=2;` `  return f;` `}` `int main(){` `  system("color F0");` `  int n;` `  cout<<"VARIANTA ITERATIVA\n\n";` `  cout<<"Introduceti datele de intare, n= "; cin>>n;` `  if(!n) cout<<"0!=1\n";` `  else cout<<n<<"!= "<<factorial(n)<<"\n";` `  return 0;` `}`
Rezultatele execuției: `VARIANTA RECURSIVA` `Introduceti datele de intare, n= 5` `5!= 1920`	***Rezultatele execuției:*** `VARIANTA ITERATIVA` `Introduceti datele de intare, n= 5` `5!= 1920`

Problema 3: Șirul lui Fibonacci – 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, ...

Să se determine primele n elemente ale şirului lui Fibonacci. Să se alcătuiască varianta recursivă şi iterativă în care va afişa rezultatul pentru determinarea primelor n elemente.

Exemplu:
Fie n=7.

$PASUL\ 1 \to 1$ $\quad 1 \to 1 = 1$
$PASUL\ 2 \to 1$ $\quad 2 \to 1 = 1$
$PASUL\ 3 \to 2$ $\quad 3 \to 1+1 = 2$
$PASUL\ 4 \to 3$ $\quad 4 \to 2+1 = 3$
$PASUL\ 5 \to 5$ $\quad 5 \to 3+2 = 5$
$PASUL\ 6 \to 8$ $\quad 6 \to 5+3 = 8$
$PASUL\ 7 \to 13$ $\quad 7 \to 8+5 = 13$

Formula recursivă:

$$Fibonacci(n) = \begin{cases} 1, \text{ pentru } n = \overline{1,2}; \\ Fibonacci(n-1) + Fibonacci(n-2), \text{ pentru } n > 2. \end{cases}$$

Varianta recursivă	Varianta iterativă
`#include <iostream>` `#include <graphics.h>` `using namespace std;` `int Fib(int k){` `  if (k<2) return 1;` `  else return (Fib(k-1)+Fib(k-2));` `}` `int main(){` `  system("color F0");` `  int n,k;` `  cout<<"VARIANTA RECURSIVA\n\n";` `  cout<<"Introduceti datele de intrare, n= ";` `  cin>>n;` `  cout<<"Primele  "<<n<<" elemente ale sirului: \n";` `  for (k=0;k<n;k++)` `  cout<<Fib(k)<<" ";` `}`	`#include <iostream>` `#include <graphics.h>` `using namespace std;` `int main(){` `  system("color F0");` `  int Fib[25],n,k;` `  cout<<"VARIANTA ITERATIVA\n\n";` `  cout<<"Introduceti datele de intrare, n= ";` `  cin>>n;` `  Fib[0]=1; Fib[1]=1;` `  for (k=2;k<n;k++)` `  Fib[k]=Fib[k-1]+Fib[k-2];` `  cout<<"Primele  "<<n<<" elemente ale sirului: \n";` `  for(k=0; k<n; k++)` `  cout<<Fib[k]<<" ";` `  return 0;` `}`
Rezultatele execuției: `VARIANTA RECURSIVA` `Introduceti datele de intrare, n= 5` `Primele  5 elemente ale sirului:` `1 1 2 3 5`	***Rezultatele execuției:*** `VARIANTA ITERATIVA` `Introduceti datele de intrare, n= 5` `Primele  5 elemente ale sirului:` `1 1 2 3 5`

Problema 4: Șirul lui Fibonacci modificat – 1, 1, 1, 2, 3, 7, 23, 164, 3779, 619779, ...

Să se determine primele n elemente ale şirului lui Fibonacci. Să se alcătuiască varianta recursivă şi iterativă în care va afişa rezultatul pentru determinarea primelor n elemente.

Exemplu:
Fie n=7.

$PASUL\ 1 \to 1$ $\quad 1 \to 1 = 1$
$PASUL\ 2 \to 1$ $\quad 2 \to 1 = 1$
$PASUL\ 3 \to 1$ $\quad 3 \to 1 = 1$
$PASUL\ 4 \to 2$ $\quad 4 \to 1 \cdot 1 + 1 = 1 + 1 = 2$
$PASUL\ 5 \to 3$ $\quad 5 \to 2 \cdot 1 + 1 = 2 + 1 = 3$
$PASUL\ 6 \to 7$ $\quad 6 \to 3 \cdot 2 + 1 = 6 + 1 = 7$
$PASUL\ 7 \to 23$ $\quad 7 \to 7 \cdot 3 + 2 = 21 + 2 = 23$

Formula recursivă:

$$Fibonacci(n) = \begin{cases} 1, \text{ pentru } n = \overline{1,3}; \\ Fibonacci(n-1) \cdot Fibonacci(n-2) + Fibonacci(n-3), \text{ pentru } n > 3. \end{cases}$$

Varianta recursivă	Varianta iterativă
`#include <iostream>` `#include <graphics.h>` `using namespace std;` `int Fib(int k){` `  if (k<3) return 1;` `  else return (Fib(k-1)*Fib(k-2)+Fib(k-3));` `}` `int main(){` `  system("color F0");` `  int n,k;` `  cout<<"VARIANTA RECURSIVA\n\n";` `  cout<<"Introduceti datele de intrare, n= ";` `  cin>>n;` `  cout<<"Primele  "<<n<<" elemente ale sirului: \n";` `  for (k=0;k<n;k++)` `  cout<<Fib(k)<<" ";` `  return 0;` `}`	`#include <iostream>` `#include <graphics.h>` `using namespace std;` `int main(){` `  system("color F0");` `  int Fib[25],n,k;` `  cout<<"VARIANTA ITERATIVA\n\n";` `  cout<<"Introduceti datele de intrare, n= ";` `  cin>>n;` `  Fib[0]=1; Fib[1]=1; Fib[2]=1;` `  for (k=3;k<n;k++)` `  Fib[k]=Fib[k-1]*Fib[k-2]+Fib[k-3];` `  cout<<"Primele  "<<n<<" elemente ale sirului: \n";` `  for(k=0; k<n; k++)` `  cout<<Fib[k]<<" ";` `  return 0;` `}`
Rezultatele execuției: `VARIANTA RECURSIVA` `Introduceti datele de intrare, n= 10` `Primele  10 elemente ale sirului:` `1 1 1 2 3 7 23 164 3779 619779`	***Rezultatele execuției:*** `VARIANTA ITERATIVA` `Introduceti datele de intrare, n= 10` `Primele  10 elemente ale sirului:` `1 1 1 2 3 7 23 164 3779 619779`

Problema 5: Algoritmul lui Euclid. CMMDC(X,Y)

Să se determine cel mai mare divizor comun a două numere naturale. Să se alcătuiască varianta recursivă şi iterativă în care va afişa CMMDC (X, Y).

Exemplu:

Fie X=4 și Y=12

$$PASUL\,1 \quad 4=12 \rightarrow \quad NU$$
$$PASUL\,2 \quad 4<12 \rightarrow \quad CMMDC(4,12-4)=CMMDC(4,8)$$
$$PASUL\,3 \quad 4=8 \quad \rightarrow \quad NU$$
$$PASUL\,4 \quad 4<8 \quad \rightarrow \quad CMMDC(4,8-4)=CMMDC(4,4)$$
$$PASUL\,5 \quad 4=4 \quad \rightarrow \quad DA$$
$$CMMDC(4,12)=4$$

Formula recursivă:

$$CMMDC(x,y)=\begin{cases} x, \ pentru\ x=y; \\ CMMDC(x-y,y), \ pentru\ x>y; \\ CMMDC(x,y-x), \ pentru\ x<y. \end{cases}$$

Varianta recursivă	Varianta iterativă
`#include <iostream>` `#include <graphics.h>` `using namespace std;` `int cmmdc(int a,int b){` `  if (a==b) return a;` `  if (a>b) return cmmdc(a-b,b);` `  else  return cmmdc(a,b-a);` `}` `int main(){` `  system("color F0");` `  int n,m;` `  cout<<"VARIANTA RECURSIVA\n\n";` `  cout<<"Introduceti valoarea 1, n= ";` `  cin>>n;` `  cout<<"Introduceti valoarea 2, m= ";` `  cin>>m;` `  cout<<"CMMDC("<<n<<","<<m<<")=";` `  cout<<cmmdc(n,m);` `}`	`#include <iostream>` `#include <graphics.h>` `using namespace std;` `int main(){` `  system("color F0");` `  int n,m,cmmdc;` `  cout<<"VARIANTA ITERATIVA\n\n";` `  cout<<"Introduceti valoarea 1, n= ";` `  cin>>n;` `  cout<<"Introduceti valoarea 2, m= ";` `  cin>>m;` `  int x=n,y=m;` `  while(n!=m){` `  if (n>m) n=n-m;` `  else m=m-n;` `  }` `  cout<<"CMMDC("<<x<<","<<y<<")=";` `  cout<<n;` `}`
Rezultatele execuției: `VARIANTA RECURSIVA` `Introduceti valoarea 1, n= 4` `Introduceti valoarea 2, m= 12` `CMMDC(4,12)=4`	***Rezultatele execuției:*** `VARIANTA ITERATIVA` `Introduceti valoarea 1, n= 4` `Introduceti valoarea 2, m= 12` `CMMDC(4,12)=4`

Problema 6: Algoritmul lui Euclid. CMMDC(X,Y,Z)

Să se determine cel mai mare divizor comun a trei numere naturale. Să se alcătuiască varianta recursivă şi iterativă în care va afişa CMMDC (X, Y, Z).

Exemplu:

Fie X=4, Y=12 și Z=8

$$PASUL\,1 \quad 4=12=8 \quad \rightarrow \quad NU$$
$$PASUL\,2 \quad 4<12 \quad \rightarrow \quad CMMDC(4,12-4)=CMMDC(4,8)$$
$$PASUL\,3 \quad 4=8 \quad \rightarrow \quad NU$$
$$PASUL\,4 \quad 4<8 \quad \rightarrow \quad CMMDC(4,8-4)=CMMDC(4,4)$$
$$PASUL\,5 \quad 4=4 \quad \rightarrow \quad DA$$
$$PASUL\,6 \quad 4=8 \quad \rightarrow \quad NU$$
$$PASUL\,7 \quad 4<8 \quad \rightarrow \quad CMMDC(4,8-4)=CMMDC(4,8)$$
$$PASUL\,8 \quad 4=4 \quad \rightarrow \quad DA$$

CMMDC (4, 12, 8)= CMMDC (CMMDC (4, 12), 8)= CMMDC (4, 8)= 4

Formula recursivă:

$$CMMDC(x,y,z)=\begin{cases} x, \ pentru\ x=y=z; \\ CMMDC(CMMDC(x,y),z), \ pentru\ x,y,z\in\mathbb{R}. \end{cases}$$

Varianta recursivă

```
#include <iostream>
#include <graphics.h>
using namespace std;
int cmmdc(int a,int b){
    if (a==b) return a;
    else if (a>b)
          return(cmmdc(a-b,b));
          else return(cmmdc(a,b-a));
}
int main(){
    system("color F0");
    int n,m,p;
    cout<<"VARIANTA RECURSIVA\n\n";
    cout<<"Introduceti trei numere:
n, m, p\n ";
    cin>>n>>m>>p;

cout<<"CMMDC("<<n<<","<<m<<","<<p<<"
)= ";
    cout<<cmmdc(cmmdc(n,m),p);
    return 0;
}
```

Rezultatele execuției:

```
VARIANTA RECURSIVA

Introduceti trei numere: n, m, p
 4 12 8
CMMDC(4,12,8)= 4
```

Varianta iterativă

```
#include <iostream>
#include <graphics.h>
using namespace std;
int main(){
    system("color F0");
    int m,n,p,cmmdc;
    cout<<"VARIANTA ITERATIVA\n\n";
    cout<<"Introdu trei numere: n,
m, p\n ";
    cin>>n>>m>>p;
    int x=n,y=m,z=p;
    while (n!=m){
            if (n>m) n=n-m ;
            else m=m-n ;
    }
    while (n!=p){
          if (n>p) n=n-p;
            else p=p-n ;
    }

cout<<"CMMDC("<<x<<","<<y<<","<<z<<"
)= "<<n;
     return 0;
}
```

Rezultatele execuției:

```
VARIANTA ITERATIVA

Introduceti trei numere: n, m, p
 4 12 8
CMMDC(4,12,8)= 4
```

Problema 7: Suma numerelor pare pozitive

Să se elaboreze un program care calculează pe paşi suma numerelor pare pozitive până la n. Să se alcătuiască varianta recursivă şi iterativă în care va afişa suma respectivă.

Exemplu:
Fie n=6

$$\begin{aligned}
&PASUL\,1 \quad 1\ este\ impar \quad \rightarrow \quad S=0\\
&PASUL\,2 \quad 2\ este\ par \quad \rightarrow \quad S=0+2=2\\
&PASUL\,3 \quad 3\ este\ impar \quad \rightarrow \quad S=2\\
&PASUL\,4 \quad 4\ este\ par \quad \rightarrow \quad S=2+4=6\\
&PASUL\,5 \quad 5\ este\ impar \quad \rightarrow \quad S=6\\
&PASUL\,6 \quad 6\ este\ par \quad \rightarrow \quad S=6+6=12
\end{aligned}$$

Formula recursivă:

$$SUMA(n)=\begin{cases} 0, \ pentru\ n=0; \\ SUMA(n-1)+2\cdot n, \ pentru\ n>0. \end{cases}$$

Varianta recursivă

```
#include <iostream>
#include <graphics.h>
using namespace std;
long int suma(int n) {
  if(n==0) return 0;
  else return suma(n-1)+(2*n);
}
int main() {
    system("color F0");
    int n, i;
    cout<<"VARIANTA RECURSIVA\n\n";
    cout<<"Introduceti n= ";cin>>n;
    for(i=1; i<=n;i++)
        cout<<"\t"<<i<<" \tn=
"<<(2*i)<<"\t S("<<i<<")=
"<<suma(i)<<endl;
}
```

Rezultatele execuției:

```
VARIANTA RECURSIVA

Introduceti n= 3
        1       n= 2     S(1)= 2
        2       n= 4     S(2)= 6
        3       n= 6     S(3)= 12
```

Varianta iterativă

```
#include <iostream>
#include <graphics.h>
using namespace std;
long int suma( int n){
    long int i, s=0;
    for(i=0; i<=n;i++)
        if(i==0) s=0;
            else s+=2*i;
        return s;
}
int main(){
    system("color F0");
    int n, i;
    cout<<"VARIANTA ITERATIVA\n\n";
    cout<<"Introduceti n= ";cin>>n;
    for(i=1; i<=n;i++)
        cout<<"\t"<<i<<" \tn=
"<<(2*i)<<"\t S("<<i<<")=
"<<suma(i)<<endl;
}
```

Rezultatele execuției:

```
VARIANTA ITERATIVA

Introduceti n= 3
        1       n= 2     S(1)= 2
        2       n= 4     S(2)= 6
        3       n= 6     S(3)= 12
```

Problema 8: Produsul numerelor impare pozitive

Să se elaboreze un program care calculează pe paşi produsul numerelor impare pozitive până la n. Să se alcătuiască varianta recursivă şi iterativă în care va afişa produsul respectiv.

Exemplu:
Fie n=6

$$PASUL\,1 \quad 1 \quad este\ impar \quad \rightarrow \quad P = 1 \cdot 1 = 1$$
$$PASUL\,2 \quad 2 \quad este\ par \quad \rightarrow \quad P = 1$$
$$PASUL\,3 \quad 3 \quad este\ impar \quad \rightarrow \quad P = 1 \cdot 3 = 3$$
$$PASUL\,4 \quad 4 \quad este\ par \quad \rightarrow \quad P = 3$$
$$PASUL\,5 \quad 5 \quad este\ impar \quad \rightarrow \quad P = 3 \cdot 5 = 15$$
$$PASUL\,6 \quad 6 \quad este\ par \quad \rightarrow \quad P = 15$$

Formula recursivă:

$$PRODUS(n) = \begin{cases} 1, \ pentru\ n = 1; \\ PRODUS(n-1) \cdot (2 \cdot n - 1), \ pentru\ n > 1. \end{cases}$$

Varianta recursivă

```
#include <iostream>
#include <graphics.h>
using namespace std;
long int produs(int n) {
  if(n==1) return 1;
  else return produs(n-1)*(2*n-1);
}
int main() {
    system("color F0");
    int n, i;
    cout<<"VARIANTA RECURSIVA\n\n";
    cout<<"Introdu n= ";cin>>n;
    for(i=1; i<=n;i++)
        cout<<"\t"<<i<<" \tn=
"<<(2*i-1)<<"\t P("<<i<<")=
"<<produs(i)<<endl;
}
```

Rezultatele execuției:

```
VARIANTA RECURSIVA

Introduceti n= 3
        1       n= 1     P(1)= 1
        2       n= 3     P(2)= 3
        3       n= 5     P(3)= 15
```

Varianta iterativă

```
#include <iostream>
#include <graphics.h>
using namespace std;
long int produs( int n){
    long int i, p=1;
    for(i=0; i<=n;i++)
        if(i==0) p=1;
            else p*=2*i-1;
        return p;
}
int main(){
    system("color F0");
    int n, i;
    cout<<"VARIANTA ITERATIVA\n\n";
    cout<<"Introduceti n= ";cin>>n;
    for(i=1; i<=n;i++)
        cout<<"\t"<<i<<" \tn=
"<<(2*i-1)<<"\t S("<<i<<")=
"<<produs(i)<<endl;
}
```

Rezultatele execuției:

```
VARIANTA ITERATIVA

Introduceti n= 3
        1       n= 1     P(1)= 1
        2       n= 3     P(2)= 3
        3       n= 5     P(3)= 15
```

Problema 9: Suma şirului de numere 1,5,9,13,17,...

Să se elaboreze un program care calculează suma primilor n termini ai şirului de numere. Să se alcătuiască varianta recursivă şi iterativă în care va afişa suma respectivă.

Exemplu:
Fie n=5

$$\begin{aligned} &PASUL\ 1 \quad E=1 \quad \rightarrow \quad S=0+1=1 \\ &PASUL\ 2 \quad E=5 \quad \rightarrow \quad S=1+5=6 \\ &PASUL\ 3 \quad E=9 \quad \rightarrow \quad S=6+9=15 \\ &PASUL\ 4 \quad E=13 \quad \rightarrow \quad S=15+13=28 \\ &PASUL\ 5 \quad E=17 \quad \rightarrow \quad S=28+17=45 \end{aligned}$$

Formula recursivă:

$$SUMA(n)=\begin{cases} 0, \ pentru\ n=0; \\ SUMA(n-1)+4\cdot n-3, \ pentru\ n>0. \end{cases}$$

Varianta recursivă	Varianta iterativă
`#include <iostream>` `#include <graphics.h>` `using namespace std;` `int suma(int k){` `  if (k==1) return 1;` `  else return suma(k-1)+(4*k-3);` `}` `int main(){` `  system("color F0");` `  int n;` `  cout<<"VARIANTA RECURSIVA\n\n";` `  cout<<"Introdu n= ";cin>>n;` `  cout<<"Suma sirului este: "<<suma(n);` `}`	`#include <iostream>` `#include <graphics.h>` `using namespace std;` `int main(){` `  system("color F0");` `  int n,suma,k,t;` `  cout<<"VARIANTA ITERATIVA\n\n";` `  cout<<"Introduceti n= ";cin>>n;` `  suma=0; k=1; t=4*n-3;` `  while(k<=t){` `  suma+=k; k+=4;` `  }` `  cout<<"Suma sirului este: "<<suma;` `}`
Rezultatele execuției: `VARIANTA RECURSIVA` `Introduceti n= 5` `Suma sirului este: 45`	***Rezultatele execuției:*** `VARIANTA ITERATIVA` `Introduceti n= 5` `Suma sirului este: 45`

Problema 10: Produsul şirului de numere 1,6,11,16,21,...

Să se elaboreze un program care calculează produsul primilor n termini ai şirului de numere. Să se alcătuiască varianta recursivă şi iterativă în care va afişa produsul respectiv.

Exemplu:
Fie n=5

$$PASUL\ 1 \quad E=1 \quad \rightarrow \quad P=1\cdot 1=1$$
$$PASUL\ 2 \quad E=6 \quad \rightarrow \quad P=1\cdot 6=6$$
$$PASUL\ 3 \quad E=11 \quad \rightarrow \quad P=6\cdot 11=66$$
$$PASUL\ 4 \quad E=16 \quad \rightarrow \quad P=66\cdot 16=1056$$
$$PASUL\ 5 \quad E=21 \quad \rightarrow \quad P=1056\cdot 21=22176$$

Formula recursivă:

$$PRODUS(n)=\begin{cases} 1, \ pentru\ n=0; \\ PRODUS(n-1)\cdot(5\cdot n-4), \ pentru\ n>0. \end{cases}$$

Varianta recursivă	Varianta iterativă
`#include <iostream>` `#include <graphics.h>` `using namespace std;` `int produs(int k){` `  if (k==0) return 1;` `  else return produs(k-1)*(5*k-4);` `}` `int main(){` `  system("color F0");` `  int n;` `  cout<<"VARIANTA RECURSIVA\n\n";` `  cout<<"Introduceti n= ";cin>>n;` `  cout<<"Produsul sirului este: "<<produs(n);` `}`	`#include <iostream>` `#include <graphics.h>` `using namespace std;` `int main(){` `  system("color F0");` `  int n,produs=1,k,t;` `  cout<<"VARIANTA ITERATIVA\n\n";` `  cout<<"Introduceti n= ";cin>>n;` `  k=1; t=5*n-4;` `  while(k<=t){` `  produs*=k; k+=5;` `  }` `  cout<<"Produsul sirului este: "<<produs;` `}`
Rezultatele execuției: `VARIANTA RECURSIVA` `Introduceti n= 3` `Produsul sirului este: 66`	***Rezultatele execuției:*** `VARIANTA ITERATIVA` `Introduceti n= 3` `Produsul sirului este: 66`

SARCINI PENTRU EXERSARE

1. *Să se elaboreze un program care afișează al n-lea termen al șirului de numere. Să se alcătuiască varianta recursivă şi iterativă care va afișa termenul.*

Varianta 1	Varianta 2
a) 1, 2, 3, 5, 8, 13, 21, 34, 55, ...	a) 1, 4, 8, 15, 26, 44, 73, 120, 196, ...
b) 1, 3, 4, 7, 11, 18, 29, 47, 76, ...	b) 1, 5, 6, 11, 17, 28, 45, 73, 118, ...
c) 1, 3, 6,11, 19, 32, 53, 87, 142, ...	c) 1, 6, 7, 13, 20, 33, 53, 86, 139, ...
d) 1, 4, 5, 9, 14, 23, 37, 60, 97, ...	d) 1, 7, 8, 15, 23, 38, 61, 99, 160, ...

2. *Să se elaboreze un program care afișează suma primilor n termeni al șirului de numere. Să se alcătuiască varianta recursivă şi iterativă care va afișa suma.*

Varianta 1	Varianta 2
a) 1, 3, 6,11, 19, 32, 53, 87, 142, ...	a) 1, 4, 8, 15, 26, 44, 73, 120, 196, ...
b) 1, 4, 5, 9, 14, 23, 37, 60, 97, ...	b) 1, 5, 6, 11, 17, 28, 45, 73, 118, ...
c) 1, 6, 7, 13, 20, 33, 53, 86, 139, ...	c) 1, 2, 3, 5, 8, 13, 21, 34, 55, ...
d) 1, 7, 8, 15, 23, 38, 61, 99, 160, ...	d) 1, 3, 4, 7, 11, 18, 29, 47, 76, ...

3. *Să se elaboreze un program care afișează produsul primilor n termeni al șirului de numere. Să se alcătuiască varianta recursivă şi iterativă care va afișa produsul.*

Varianta 1	Varianta 2
a) 1, 3, 6,11, 19, 32, 53, 87, 142, ...	a) 1, 2, 3, 5, 8, 13, 21, 34, 55, ...
b) 1, 7, 8, 15, 23, 38, 61, 99, 160, ...	b) 1, 3, 4, 7, 11, 18, 29, 47, 76, ...
c) 1, 5, 6, 11, 17, 28, 45, 73, 118, ...	c) 1, 4, 5, 9, 14, 23, 37, 60, 97, ...
d) 1, 4, 8, 15, 26, 44, 73, 120, 196, ...	d) 1, 6, 7, 13, 20, 33, 53, 86, 139, ...

4. *Să se elaboreze un program care stabilește dacă două șiruri de caractere sunt anagrame. Să se alcătuiască varianta recursivă şi iterativă care va afișa rezultatele.*

5. *Să se alcătuiască varianta recursivă şi iterativă pentru următoarele exemple:*
   - *A. afișează cifra minimă a unui număr de maxim 9 cifre;*
   - *B. afișează cifra minimă pară a unui număr de maxim 9 cifre;*
   - *C. afișează cifra minimă impară a unui număr de maxim 9 cifre;*
   - *D. afișează cifra maximă a unui număr de maxim 9 cifre;*
   - *E. afișează cifra maximă pară a unui număr de maxim 9 cifre;*
   - *F. afișează cifra maximă impară a unui număr de maxim 9 cifre;*
   - *G. afișează pozițiile cifrelor pare a unui număr de maxim 9 cifre;*
   - *H. afișează pozițiile cifrelor impare a unui număr de maxim 9 cifre;*
   - *I. afișează suma cifrelor unui număr de maxim 9 cifre;*
   - *J. afișează suma cifrelor pare unui număr de maxim 9 cifre;*
   - *K. afișează suma cifrelor impare unui număr de maxim 9 cifre;*
   - *L. afișează produsul cifrelor unui număr de maxim 9 cifre.*
   - *M. afișează produsul cifrelor pare unui număr de maxim 9 cifre;*
   - *N. afișează produsul cifrelor impare unui număr de maxim 9 cifre;*
   - *O. afișează inversarea cifrelor unui număr, exemplu: 123456789 → 987654321;*
   - *P. afișează toate numerele palindrom ce conțin exact 9 cifre;*
   - *Q. afișează toate numerele palindrom pare ce conțin exact 9 cifre;*
   - *R. afișează toate numerele palindrom impare ce conțin exact 9 cifre;*
   - *S. afișează toate numerele prime ce conțin exact 9 cifre;*
   - *T. afișează toate numerele pare divizibile cu 3 ce conțin exact 9 cifre;*
   - *U. afișează toate numerele impare divizibile cu 5 ce conțin exact 9 cifre;*

6. *Să se alcătuiască varianta recursivă şi iterativă pentru următoarele exemple:*
   A. *suma şi produsul primelor n pătrate perfecte;*
   B. *suma şi produsul primelor n pătrate perfecte pare;*
   C. *suma şi produsul primelor n pătrate perfecte impare;*
   D. *suma şi produsul primelor n cuburi;*
   E. *suma şi produsul primelor n cuburi pare;*
   F. *suma şi produsul primelor n cuburi impare;*
   G. *suma şi produsul primelor 2^n elemente;*
   H. *suma şi produsul primelor 2^n elemente pare;*
   I. *suma şi produsul primelor 2^n elemente impare;*
   J. *suma şi produsul primelor n elemente din şirul lui Fibonacci;*
   K. *suma şi produsul primelor n elemente pare din şirul lui Fibonacci;*
   L. *suma şi produsul primelor n elemente impare din şirul lui Fibonacci;*

7. *Să se alcătuiască varianta recursivă şi iterativă pentru următoarele exemple:*
   A. *de câte ori apare cifra x într-un vector a cu n elemente întregi;*
   B. *de câte ori apare cifra pară x într-un vector a cu n elemente întregi;*
   C. *de câte ori apare cifra impară x într-un vector a cu n elemente întregi;*
   D. *de câte ori apare vocala x într-un vector a cu n elemente de tip caracter;*
   E. *de câte ori apare consoana x într-un vector a cu n elemente de tip caracter;*
   F. *de câte ori apare diftongul "oa" într-un vector a cu n elemente de tip caracter;*
   G. *de câte ori apare triftongul "ioa" într-un vector a cu n elemente de tip caracter;*
   H. *de câte ori apare heatul "ue" într-un vector a cu n elemente de tip caracter.*

8. *Să se alcătuiască varianta recursivă şi iterativă pentru următoarele exemple:*
   A. *afişează suma elementelor unui tablou unidimensional cu elemente reale;*
   B. *afişează produsul elementelor unui tablou unidimensional cu elemente reale;*
   C. *afişează elementul minim par al unui tablou unidimensional;*
   D. *afişează elementul maxim impar al unui tablou unidimensional;*
   E. *afişează poziţia elementului maxim al unui tablou unidimensional;*
   F. *afişează poziţia elementului minim al unui tablou unidimensional.*

9. *Să se alcătuiască varianta recursivă şi iterativă pentru următoarele exemple:*
   A. *afişează suma elementelor pare pozitive ale unui tablou bidimensional;*
   B. *afişează produsul elementelor impare nepozitive ale unui tablou bidimensional;*
   C. *afişează elementul minim impar al unui tablou bidimensional;*
   D. *afişează elementul maxim par al unui tablou bidimensional;*
   E. *afişează poziţia elementului maxim al unui tablou bidimensional;*
   F. *afişează poziţia elementului minim al unui tablou bidimensional.*

10. *Să se elaboreze un program care afişează rezultatul conversiei unui număr zecimal într-o altă bază de numeraţie. Să se alcătuiască varianta recursivă şi iterativă care va afişa rezultatele.*
    A. *Din baza 10 în baza b=2;*
    B. *Din baza 10 în baza b=3;*
    C. *Din baza 10 în baza b=4;*
    D. *Din baza 10 în baza b=5;*
    E. *Din baza 10 în baza b=6;*
    F. *Din baza 10 în baza b=7;*
    G. *Din baza 10 în baza b=8;*
    H. *Din baza 10 în baza b=9;*
    I. *Din baza 10 în baza b=11;*
    J. *Din baza 10 în baza b=12;*
    K. *Din baza 10 în baza b=13;*
    L. *Din baza 10 în baza b=14;*
    M. *Din baza 10 în baza b=15;*
    N. *Din baza 10 în baza b=16;*

2.5 Implementarea metodelor recursive directe și indirecte

Problema 1: Funcția McCarthy *(numită după John McCarthy)*

Să se alcătuiască un program pentru implementarea funcției McCarthy.

Formula recursivă	Exemplu
$MC(x) = \begin{cases} x-10, \text{pentru } x>100; \\ MC(MC(x+11)), \text{pentru } x\leq 100. \end{cases}$	McCarthy(0)=91; McCarthy(100)=91; McCarthy(1)=91; McCarthy(101)=91; McCarthy(2)=91; McCarthy(102)=92; McCarthy(3)=91; McCarthy(103)=93; McCarthy(4)=91; McCarthy(104)=94;

Implementare C++

```
#include <iostream>
#include <graphics.h>
#include <math.h>
using namespace std;
int MC(int m){
    if (m>100){
       return m-10;
    }
    return MC(MC(m+11));
}
int main(){
    system("color F0");
    int s;
    cout<<" FUNCTIA McCarthy\n";
    cout<<" Introduceti numarul de pasi, n= ";
    cin>>s;
    for (int m = 0; m < s; ++m){
        cout<<"\tMC("<<m<<")= "<<MC(m)<<"\n";
    }
}
```

Rezultatele execuției:

```
FUNCTIA McCarthy
Introduceti numarul de pasi, n= 6
        MC(0)= 91
        MC(1)= 91
        MC(2)= 91
        MC(3)= 91
        MC(4)= 91
        MC(5)= 91
```

Notă:

- ✓ *John McCarthy (4 septembrie 1927 - 24 octombrie 2011) a fost un informatician și om de știință cognitiv american. McCarthy a fost unul dintre fondatorii disciplinei inteligenței artificiale. El a inventat termenul „inteligență artificială” (AI), a dezvoltat familia limbajului de programare Lisp, a influențat semnificativ proiectarea limbajului de programare ALGOL, a popularizat timesharing-ul, a inventat colectarea gunoiului și a fost foarte influent în dezvoltarea timpurie a inteligenței artificiale* ***[22]***.

Problema 2: Funcția Manna-Pnueli *(numită după Zohar Manna și Amir Pnueli)*

Să se alcătuiască un program pentru implementarea funcției Manna-Pnueli.

Formula recursivă	Exemplu
$MP(x) = \begin{cases} x-1, \text{pentru } x \geq 12; \\ MP(MP(x+2)), \text{pentru } x < 12. \end{cases}$	Manna_Pnueli(0)=11; Manna_Pnueli(10)=11; Manna_Pnueli(1)=11; Manna_Pnueli(11)=11; Manna_Pnueli(2)=11; Manna_Pnueli(12)=11; Manna_Pnueli(3)=11; Manna_Pnueli(13)=12; Manna_Pnueli(4)=11; Manna_Pnueli(14)=13;

Implementare C++

```
#include <iostream>
#include <graphics.h>
using namespace std;
int MP(int m){
    if (m>=12){
       return m-1;
    }
       else return MP(MP(m+2));
}
int main(){
    system("color F0");
    int s;
    cout<<" FUNCTIA Manna_Pnueli \n";
    cout<<" Introduceti numarul de pasi, n= ";
    cin>>s;
    for (int m = 0; m < s; ++m){
        cout<<"\tMP("<<m<<")= "<<MP(m)<<"\n";
    }
}
```

Rezultatele execuției:

```
FUNCTIA Manna_Pnueli
Introduceti numarul de pasi, n= 6
       MP(0)= 11
       MP(1)= 11
       MP(2)= 11
       MP(3)= 11
       MP(4)= 11
       MP(5)= 11
```

Notă:

- ✓ *Zohar Manna (1939 - 30 august 2018) a fost un informatician israeliano-american care a fost profesor de informatică la Universitatea Stanford. El a supravegheat 30 de doctoranzi, printre care, de exemplu, Nachum Dershowitz, Thomas Henzinger și Pierre Wolper.*
- ✓ *Amir Pnueli (în ebraică אמיר פנואלי; n. 22 aprilie 1941, Nahalal, Palestina sub mandat britanic – d. 2 noiembrie 2009, New York, SUA) a fost un informatician israelian, laureat al Premiului Turing în 1996, care, conform comisiei de acordare, i-a fost conferit pentru lucrări de referință ce au introdus logica temporală în informatică și pentru remarcabile contribuții în domeniul verificării programelor și sistemelor* ***[22]****.*

Problema 3: Funcția Takeuchi *(numită după Ikuo Takeuchi)*

Să se alcătuiască un program pentru implementarea funcției Takeuchi.

Formula recursivă	Exemplu
$TAK(x, y, z) = \begin{cases} TAK(TAK(x-1,y,z),TAK(y-1,z,x),TAK(z-1,x,y)), \text{pentru } y<x; \\ z, \text{altfel}. \end{cases}$	TAKEUCHI(0,0,0)=0; TAKEUCHI(0,0,1)=1; TAKEUCHI(0,1,0)=0; TAKEUCHI(0,1,1)=1; TAKEUCHI(1,0,0)=0; TAKEUCHI(1,0,1)=0; TAKEUCHI(1,1,0)=0; TAKEUCHI(1,1,1)=0;

Implementare C++

```
#include <iostream>
#include <graphics.h>
using namespace std;
int TAK(int m, int n, int p){
    if (n < m){
       return TAK(TAK(m-1,n,p),TAK(n-1,p,m),TAK(p-1,m,n));
    }
    return p;
}
int main(){
    system("color F0");
    cout<<" FUNCTIA TAKEUCHI\n";
    /// cazul TAKEUCHI(2,2,2), unde 2 nu este inclus
    for (int m = 0; m < 2; ++m){
              for (int n = 0; n < 2; ++n){
            for (int p = 0; p < 2; ++p){
                 cout<<"\tTAKEUCHI(";
                 cout<<m<<","<<n<<","<<p<<")= "<<TAK(m,n,p)<<"\n";
            }
        }
    }
}
```

Rezultatele execuției:

```
FUNCTIA TAKEUCHI
        TAKEUCHI(0,0,0)= 0
        TAKEUCHI(0,0,1)= 1
        TAKEUCHI(0,1,0)= 0
        TAKEUCHI(0,1,1)= 1
        TAKEUCHI(1,0,0)= 0
        TAKEUCHI(1,0,1)= 0
        TAKEUCHI(1,1,0)= 0
        TAKEUCHI(1,1,1)= 1
```

Notă:

- ✓ *Ikuo Takeuchi (竹内郁雄) (1946) este un informatician japonez, profesor de informatică și inginer la Universitatea Waseda, Tokyo* ***[22]***.

Problema 4: Funcția Ackermann *(numită după Wilhelm Ackermann)*

Să se alcătuiască un program pentru implementarea funcției Ackermann.

Formula recursivă	Exemplu
$Ack(m,n) = \begin{cases} n+1, \text{pentru } m=0; \\ Ack(m-1,1), \text{pentru } m>0, n=0; \\ Ack(m-1,Ack(m,n-1)), \text{pentru } m>0, n>0. \end{cases}$	Ackermann(0,0)=1 Ackermann(0,1)=2 Ackermann(1,0)=2 Ackermann(1,1)=3 Ackermann(1,2)=4 Ackermann(2,1)=5 Ackermann(2,2)=7

Implementare C++

```
#include <iostream>
#include <graphics.h>
using namespace std;
int ack(int m, int n){
    if (m == 0){
        return n + 1;
    }
    if (n == 0){
        return ack(m-1,1);
    }
    return ack(m - 1, ack(m,n - 1));
}
int main(){
    system("color F0");
    cout<<" FUNCTIA ACKERMANN\n";
    /// cazul ACKERMANN(3,3), unde 3 nu este inclus
    for (int m = 0; m < 3; ++m){
        for (int n = 0; n < 3; ++n){
            cout<<"\t Ackermann ("<<m<<","<<n<<")= "<<ack(m,n)<<"\n";
        }
    }
}
```

Rezultatele execuției:

```
FUNCTIA ACKERMANN
        Ackermann (0,0)= 1
        Ackermann (0,1)= 2
        Ackermann (0,2)= 3
        Ackermann (1,0)= 2
        Ackermann (1,1)= 3
        Ackermann (1,2)= 4
        Ackermann (2,0)= 3
        Ackermann (2,1)= 5
        Ackermann (2,2)= 7
```

Notă:

- ✓ *Wilhelm Friedrich Ackermann (n. 29 martie 1896 - d. 24 decembrie 1962) a fost un matematician german, cunoscut pentru ceea ce ulterior a fost numită ca funcție Ackermann și care își demonstrează utilitatea în teoria calculului. Ackermann a absolvit Universitatea Georg-August din Göttingen obținând doctoratul în 1925 și l-a avut ca profesor pe David Hilbert* ***[22]***.

Problema 5: Funcția Hermite *(numită după Charles Hermite)*

Să se alcătuiască un program pentru implementarea funcției Hermite.

Formula recursivă	Exemplu
$Hr(m,n) = \begin{cases} 1, \text{pentru } m=0; \\ 2 \cdot n, \text{pentru } m=1; \\ 2 \cdot n \cdot Hr(m-1,n) - 2 \cdot (m-1) \cdot Hr(m-2,n), \text{pentru } n \geq 2. \end{cases}$	Hermite(0,0)=1 Hermite(0,1)=1 Hermite(1,0)=0 Hermite(1,1)=2 Hermite(1,2)=4 Hermite(2,1)=2 Hermite(2,2)=14

Implementare C++

```
#include <iostream>
#include <graphics.h>
using namespace std;
int hermite(int m, int n){
    if (m == 0){
        return 1;
    }
    if (m == 1){
        return 2*n;
    }
    return 2*n*hermite(m - 1,n)-2*(m-1)*hermite(m - 2,n);
}
int main(){
    system("color F0");
    cout<<" FUNCTIA HERMITE\n";
    /// cazul HERMITE(3,3), unde 3 nu este inclus
    for (int m = 0; m < 3; ++m){
        for (int n = 0; n < 3; ++n){
            cout<<"\tHERMITE("<<m<<","<<n<<")= "<<hermite(m,n)<<"\n";
        }
    }
}
```

Rezultatele execuției:

```
 FUNCTIA HERMITE
        HERMITE(0,0)= 1
        HERMITE(0,1)= 1
        HERMITE(0,2)= 1
        HERMITE(1,0)= 0
        HERMITE(1,1)= 2
        HERMITE(1,2)= 4
        HERMITE(2,0)= -2
        HERMITE(2,1)= 2
        HERMITE(2,2)= 14
```

Notă:

- ✓ *Charles Hermite (24.12.1822 – 14.01.1901) a fost un matematician francez care a efectuat cercetări privind teoria numerelor, formele cuadratice, teoria invarianților, polinoamele ortogonale, funcțiile eliptice și algebre. Polinoamele Hermite, Forma naturală Hermite, Operatorul hermitic, și curbele spline Hermite au fost denumite în cinstea sa. Henri Poincaré i-a fost student. A fost primul care a demonstrat că* ***e****, baza logaritmului natural, este număr transcendent* ***[22]****.*

Problema 6: Funcția Cebâșev *(numită după Pafnuty Lvovich Chebyshev)*

Să se alcătuiască un program pentru implementarea funcției Cebâșev.

Formula recursivă	Exemplu
$Ceb(m,n) = \begin{cases} 1, \text{pentru } m=0; \\ 2\times n, \text{pentru } m=1; \\ 2n\cdot Ceb(m-1,n)-Ceb(m-2,n), \text{pentru } n\geq 2. \end{cases}$	Chebyshev(0,0) = 1 Chebyshev(0,1) = 1 Chebyshev(1,0) = 0 Chebyshev(1,1) = 2 Chebyshev(1,2) = 4 Chebyshev(2,1) = 3 Chebyshev(2,2) = 15

Implementare C++

```
#include <iostream>
#include <graphics.h>
using namespace std;
int chebyshev(int m, int n){
    if (m == 0){
        return 1;
    }
    if (m == 1){
        return 2*n;
    }
    return 2*n*chebyshev(m - 1,n)-chebyshev(m - 2,n);
}
int main(){
    system("color F0");
    cout<<" FUNCTIA CHEBYSHEV\n";
    /// cazul CHEBYSHEV (3,3), unde 3 nu este inclus
    for (int m = 0; m < 3; ++m){
        for (int n = 0; n < 3; ++n){
            cout<<"\tCHEBYSHEV("<<m<<","<<n<<")= "<<chebyshev(m,n)<<"\n";
        }
    }
}
```

Rezultatele execuției:

```
 FUNCTIA CHEBYSHEV
        CHEBYSHEV(0,0)= 1
        CHEBYSHEV(0,1)= 1
        CHEBYSHEV(0,2)= 1
        CHEBYSHEV(1,0)= 0
        CHEBYSHEV(1,1)= 2
        CHEBYSHEV(1,2)= 4
        CHEBYSHEV(2,0)= -1
        CHEBYSHEV(2,1)= 3
        CHEBYSHEV(2,2)= 15
```

Notă:

- ✓ *Pafnuti Lvovici Cebîșev (în rusă Пафнутий Львович Чебышёв) (n. 16.05.1821, d. 26.11.1894) a fost un matematician rus, cu contribuții în domeniul probabilităților, statisticii și teoriei numerelor. Cel mai strălucit reprezentant al școlii matematice din Petersburg, este considerat, după Nicolai Ivanovici Lobacevski, ca fiind cel mai mare matematician rus* ***[22]***.

Problema 7: Calcularea valorilor a două funcţii F(x) şi G(x).

Să se alcătuiască un program utilizând recursia indirectă pentru a calcula valorile funcţiilor.

Funcțiile F(x) și G(x)	Exemplu
$F(x) = \begin{cases} x+2, pentru\ x \le 1; \\ G(x-1), pentru\ x > 1. \end{cases}$ $G(x) = \begin{cases} -x, pentru\ x < 0; \\ F(x)+1, pentru\ x \ge 0. \end{cases}$	F(0)=2 G(0)=3 F(1)=3 G(1)=4 F(2)=4 G(2)=5 F(3)=5 G(3)=6 F(4)=6 G(4)=7 F(5)=7 G(5)=8 F(6)=8 G(6)=9 F(7)=9 G(7)=10

Implementare C++

```
#include <iostream>
#include <graphics.h>
using namespace std;
int G(int);
int F(int x){
    if (x>1) return G(x-1);
    else return x+2;
}
int G(int x){
    if(x>=0) return F(x)+1;
    else return -x;
}
int main (){
    system("color F0");
    int n;
    cout<<"Introduceti n= ";cin>>n;
    for (int m = 0; m <= n; ++m){
    cout<<"\tF("<<m<<")= "<<F(m)<<"\t\tG("<<m<<")= "<<G(m);
    cout<<endl;
    }
    return 0;
}
```

Rezultatele execuției:

```
Introduceti n= 7
        F(0)= 2         G(0)= 3
        F(1)= 3         G(1)= 4
        F(2)= 4         G(2)= 5
        F(3)= 5         G(3)= 6
        F(4)= 6         G(4)= 7
        F(5)= 7         G(5)= 8
        F(6)= 8         G(6)= 9
        F(7)= 9         G(7)= 10
```

Problema 8: Calcularea valorilor a trei funcţii F(x), G(x) și H(x).

Să se alcătuiască un program utilizând recursia indirectă pentru a calcula valorile funcţiilor.

Funcțiile F(x), G(x) și H(x)	Exemplu
$F(x)=\begin{cases} x^2, \text{pentru } x\le 3; \\ G(x-2), \text{pentru } 3<x<6; \\ H(x-1), \text{pentru } x\ge 6. \end{cases}$ $G(x)=\begin{cases} (x+1)^2, \text{pentru } x\le 0; \\ 0, \text{pentru } 0<x<5; \\ F(x)+5, \text{pentru } x\ge 5. \end{cases}$ $H(x)=\begin{cases} x^2, \text{pentru } x\le 0; \\ G(x+1), \text{pentru } 0<x<5; \\ F(x+1), \text{pentru } x\ge 5. \end{cases}$	F(1)=1 G(1)=6 H(1)=4 F(2)=2 G(2)=9 H(2)=9 F(3)=9 G(3)=14 H(3)=9 F(4)=9 G(4)=14 H(4)=14 F(5)=14 G(5)=0 H(4)=0 F(6)=14 G(5)=0 H(4)=0 F(7)=0 G(5)=0 H(4)=0

Implementare C++

```
#include <iostream>
#include <graphics.h>
#include <math.h>
using namespace std;
int G(int x);
int H(int x);
int F(int x){
    if(x<=3) return pow(x,2);
    else if(x>=6) return H(x-1);
    else return G(x-2);
}
int G(int x){
    if(x<=0) return pow(x+1,2);
    else if(x>=5) return F(x)+5;
    return 0;
}
int H(int x){
    if(x<=0) return pow(x,2);
    else if(x>=5) return F(x+1);
    return G(x+1);
}
int main(){
    system("color F0");
    int m=1, p=2, i=1;
    cout<<"F(x)"<<"\t\tG(x)"<<"\t\tH(x)"<<endl;
    for(int x=m; x<=p;x++){
       cout<<"F("<<x<<") = "<<F(x)<<";";
       cout<<"\tG("<<x<<") = "<<G(x)<<";";
       cout<<"\tH("<<x<<") = "<<H(x)<<";"<<endl;
       i++;
    }
}
```

Rezultatele execuției:

```
F(x)            G(x)            H(x)
F(1) = 1;       G(1) = 0;       H(1) = 0;
F(2) = 4;       G(2) = 0;       H(2) = 0;
```

Problema 9: Compoziția a două funcţii F(x) şi G(x).

Să se alcătuiască un program utilizând recursia indirectă pentru a găsi compozițiile:
a) F(G(F(x))); *b) G(F(G(x)));*

Funcțiile F(x) și G(x)	Exemplu
$F(x)=\begin{cases} 2x^2+1, pentru\ x<5; \\ x^4+2, pentru\ 5\le x<8; \\ 3, pentru\ x\ge 8. \end{cases}$ $G(x)=\begin{cases} 5x^2-3x+2, pentru\ x\le 1; \\ x^2-x+5, pentru\ x>1. \end{cases}$	$F(G(F(X)))$ / $G(F(G(X)))$ $X=-2$: 3 / 11 $X=-1$: 3 / 11 $X=0$: 33 / 77 $X=1$: 3 / 1061 $X=2$: 3 / 5772011

Implementare C++

```
#include <iostream>
#include <graphics.h>
#include <math.h>
using namespace std;
int f(int x){
    if (x<5){ return 2*pow(x,2)+1; }
    else if (x>=8){ return 3; }
    else { return pow(x,4)+2; }
}
int g(int x){
    if (x<=1){ return 5*pow(x,2)-3*x+2; }
    else { return pow(x,2)-x+5; }
}
int main() {
    system("color F0");
    int a, b;
    cout<<"Rezolvare prin metoda recursiva."<<endl;
    cout<<"Pentu x, introduceti intervalul [a,b]: ";
    cin>>a>>b; cout<<endl;
    cout<< "\t\tf(g(f(x)))"<<"\tg(f(g(x)))";
    cout<<endl;
    for(int x=a; x<=b;x++){
        cout<< "x= "<< x<<"\t\t"<< f(g(f(x)))<<"\t\t"<< g(f(g(x)));
        cout<<endl;
    }
}
```

Rezultatele execuției:

```
Rezolvare prin metoda recursiva.
Pentu x, introduceti intervalul [a,b]:  0 3

                f(g(f(x)))      g(f(g(x)))
x= 0            33              77
x= 1            3               1061
x= 2            3               5772011
x= 3            3               11
```

Problema 10: Compoziția a trei funcţii F(x), G(x) și H(x).

Să se alcătuiască un program utilizând recursia indirectă pentru a găsi compozițiile:

a) F(G(H(G(F(x))))); *b) H(G(F(F(x))));*

Funcțiile F(x), G(x) și H(x)	Exemplu
$F(x)=\begin{cases} 2+1, \text{pentru } x<5; \\ 3, \text{pentru } x\geq 8; \\ 2x^4+2, \text{pentru } 5\leq x<8. \end{cases}$ $G(x)=\begin{cases} 5x^2-3x+2, \text{pentru } x\leq 1; \\ x^2-x+5, \text{pentru } x>1. \end{cases}$ $H(x)=\begin{cases} 2x^2+1, \text{pentru } x\leq 1; \\ x^4+2, \text{pentru } x>1. \end{cases}$	$F(G(H(X)))$ $H(G(F(X)))$ $X=-2$ 3 3515304 $X=-1$ 3 14642 $X=0$ 33 258 $X=1$ 3 14642 $X=2$ 3 3515304

Implementare C++

```
#include <iostream>
#include <graphics.h>
#include <math.h>
using namespace std;
int f(int x){
    if (x<5){ return 2*pow(x,2)+1; }
    else if (x>=8){ return 3; }
    else { return pow(x,4)+2; }
}
int g(int x){
    if (x<=1){ return 5*pow(x,2)-3*x+2; }
    else { return pow(x,2)-x+5; }
}
int h(int x){
    if (x<=1){ return 2*pow(x,2)+1; }
    else { return pow(x,4)+2; }
}
int main() {
    system("color F0");
    int a, b;
    cout<<"Rezolvare prin metoda recursiva."<<endl;
    cout<<"Pentu x, introduceti intervalul [a,b]: "; cin>>a>>b; cout<<endl;
    cout<< "\t\tf(g(h(x)))"<<"\th(g(f(x)))";
    cout<<endl;
    for(int x=a; x<=b;x++){
        cout<< "x= "<< x<<"\t\t"<<f(g(h(x)))<<"\t\t"<<h(g(f(x)))<<endl;
    }
}
```

Rezultatele execuției:

```
Rezolvare prin metoda recursiva.
Pentu x, introduceti intervalul [a,b]: -1 1

                f(g(h(x)))      h(g(f(x)))
x= -1           3               14642
x= 0            33              258
x= 1            3               14642
```

SARCINI PENTRU EXERSARE

1. *Pentru litera a introduceți numărul de ordine corespunzător listei din registru. Să se alcătuiască un program pentru implementarea următoarelor funcții, este necesar de afișat minimum primii 10 de pași :*

Varianta 1	Varianta 2
A. $Z(x)=\begin{cases} ax-1, pentru\ x>10; \\ Z(Z(ax+1)), pentru\ x\le 10. \end{cases}$	**A.** $Z(x)=\begin{cases} a^2x-1, pentru\ x>100; \\ Z(Z((a+1)^3x+1)), pentru\ x\le 100. \end{cases}$
B. $Z(x,y)=\begin{cases} Z(Z(ax-1,y),Z(x,ay-1)), pentru\ y<x; \\ 1, altfel. \end{cases}$	**B.** $Z(x,y)=\begin{cases} Z(Z(x-1,(a+2)y),Z(x,(a+1)y)), pentru\ y<x; \\ 1, altfel. \end{cases}$

2. *Pentru litera a introduceți numărul de ordine corespunzător listei din registru. Să se alcătuiască un program utilizând recursia indirectă pentru a calcula valorile funcțiilor.*

Varianta 1	Varianta 2
C. $Z(x)=\begin{cases} ax, pentru\ x>20; \\ Z(W(ax+1)), pentru\ x\le 10. \end{cases}$ $W(x)=\begin{cases} ax, pentru\ x>30; \\ W(Z(ax+2)), pentru\ x\le 20. \end{cases}$	**C.** $Z(x)=\begin{cases} a^2x-1, pentru\ x>100; \\ Z(W(a^3x+1)), pentru\ x\le 100. \end{cases}$ $W(x)=\begin{cases} a^2x-2, pentru\ x>100; \\ W(Z(a^3x+2)), pentru\ x\le 100. \end{cases}$
D. $Z(x,y)=\begin{cases} Z(Z(ax+1,y),W(x,ay-1)), pentru\ y<x; \\ 1, altfel. \end{cases}$ $W(x,y)=\begin{cases} W(Z(ax+2,y),Z(x,ay-2)), pentru\ y<x; \\ 2, altfel. \end{cases}$	**D.** $Z(x,y)=\begin{cases} Z(W(x-1,(a+2)y),Z(x,(a+1)y)), pentru\ y<x; \\ 1, altfel. \end{cases}$ $W(x,y)=\begin{cases} W(Z(x-1,(a+2)y),W(x,(a+1)y)), pentru\ y<x; \\ 2, altfel. \end{cases}$

3. *Pentru litera a introduceți numărul de ordine corespunzător listei din registru. Să se alcătuiască un program utilizând recursia indirectă pentru a găsi compozițiile:*
   - *F(G(F(x)));*
   - *F(F(G(x)));*
   - *F(G(G(x)));*
   - *G(F(G(x)));*
   - *G(G(F(x)));*
   - *G(F(F(x)));*

Varianta 1	Varianta 2
E. $Z(x)=\begin{cases} a^2x-1, pentru\ x>100; \\ Z(Z((a+1)^3x+1)), pentru\ x\le 100. \end{cases}$	**E.** $Z(x)=\begin{cases} ax-1, pentru\ x>10; \\ Z(Z(ax+1)), pentru\ x\le 10. \end{cases}$
F. $Z(x,y)=\begin{cases} Z(Z(x-1,(a+2)y),Z(x,(a+1)y)), pentru\ y<x; \\ 1, altfel. \end{cases}$	**F.** $Z(x,y)=\begin{cases} Z(Z(ax-1,y),Z(x,ay-1)), pentru\ y<x; \\ 1, altfel. \end{cases}$

2.6 Implementarea metodelor recursive la construcția fractalilor

1. Instalarea modulului grafic în mediul de programare CodeBlocks

Biblioteca grafică WinBGIm se poate descărca de aici:

- *https://www.edusoft.ro/introp/wp-content/uploads/2017/11/winbgim.zip*

După ce descărcați și extrageți fișierele din această arhivă, veți face următoarele lucruri:

- *Extrageți fișierele **graphics.h**, **winmgim.h** si **libbgi.a** din arhiva **winbgim.zip**.*
- *Copiați fișierele **graphics.h** și **winbgim.h** în folderul **include** din compilatorul* MinGW, *care se găsește, cel mai probabil, aici*: ***c:\Program Files\CodeBlocks\MinGW\include***
- *Copiați* fișierul ***libbgi.a*** în folderul ***lib*** *al directorului compilatorului* MinGW, *care se găsește, cel mai probabil, aici*: ***c:\Program Files\CodeBlocks\MinGW\lib***
- *În Code::Blocks accesați **Settings** >> **Compiler…** >> **linker settings***
- *Faceți click pe butonul **Add** în partea **Link libraries**, apoi răsfoiți și selectați fișierul **libbgi.a***
- *În partea dreaptă (adică la **Other linker options**) scrieți următoarele comenzi: **-lbgi -lgdi32 -lcomdlg32 -luuid -loleaut32 -lole32***
- *Faceți click pe **OK**.*

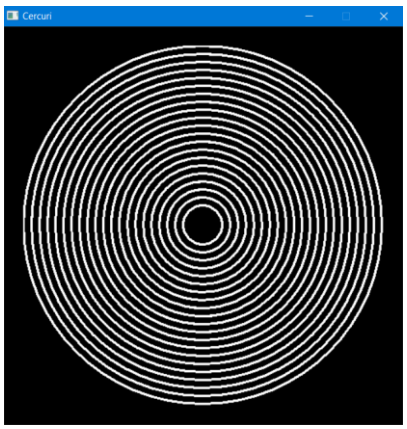
Descrierea funcțiilor grafice se găsește aici:

- *http://www.cs.colorado.edu/~main/cs1300/doc/bgi/bgi.htm*

Vezi funcţiile grafice în C/C++ la:

- *http://www.programmingsimplified.com/c/graphics.h*

2. Programe pentru testarea regimului grafic în mediul CodeBlocks

Implementare C++	Rezultatul obținut
`#include <graphics.h>` `#include <winbgim.h>` `int main(){` `  initwindow(500, 500, "Desen");` `  setcolor(WHITE);` `  setlinestyle(0,0,3);` `  circle (250,250,150);` `  circle (200,200, 10);` `  circle (290,200, 10);` `  circle (250,320, 30);` `  while (!kbhit( )){` `  delay(100);` `  }` `}`	
`#include <graphics.h>` `#include <winbgim.h>` `int main(){` `  int i,x,y,radius;` `  initwindow(500, 500, "Cercuri");` `  setcolor(WHITE);` `  setlinestyle(0,0,3);` `  x=getmaxx()/2; y=getmaxy()/2;` `  for(radius=25;radius<=225;radius+=10){` `  circle(x,y,radius);` `  }` `  while (!kbhit( )){` `  delay(100);` `  }` `}`	

Problema 1: Geometrie fractală. Fulgul lui Koch pentru triunghi echilateral

1. *Curba Koch:*

Legea de transformare - impune divizarea în trei părţi egale a dreptei, înlăturarea părţii centrale şi înlocuirea ei cu un triunghi echilateral fără bază. La fiecare iteraţie, fiecare dreaptă se consideră independentă şi i se aplică legea de transformare. După un număr foarte mare de paşi se obţine o curbă continuă, nederivabilă, cu dimensiunea fractală D=ln4/ln3=1,26.

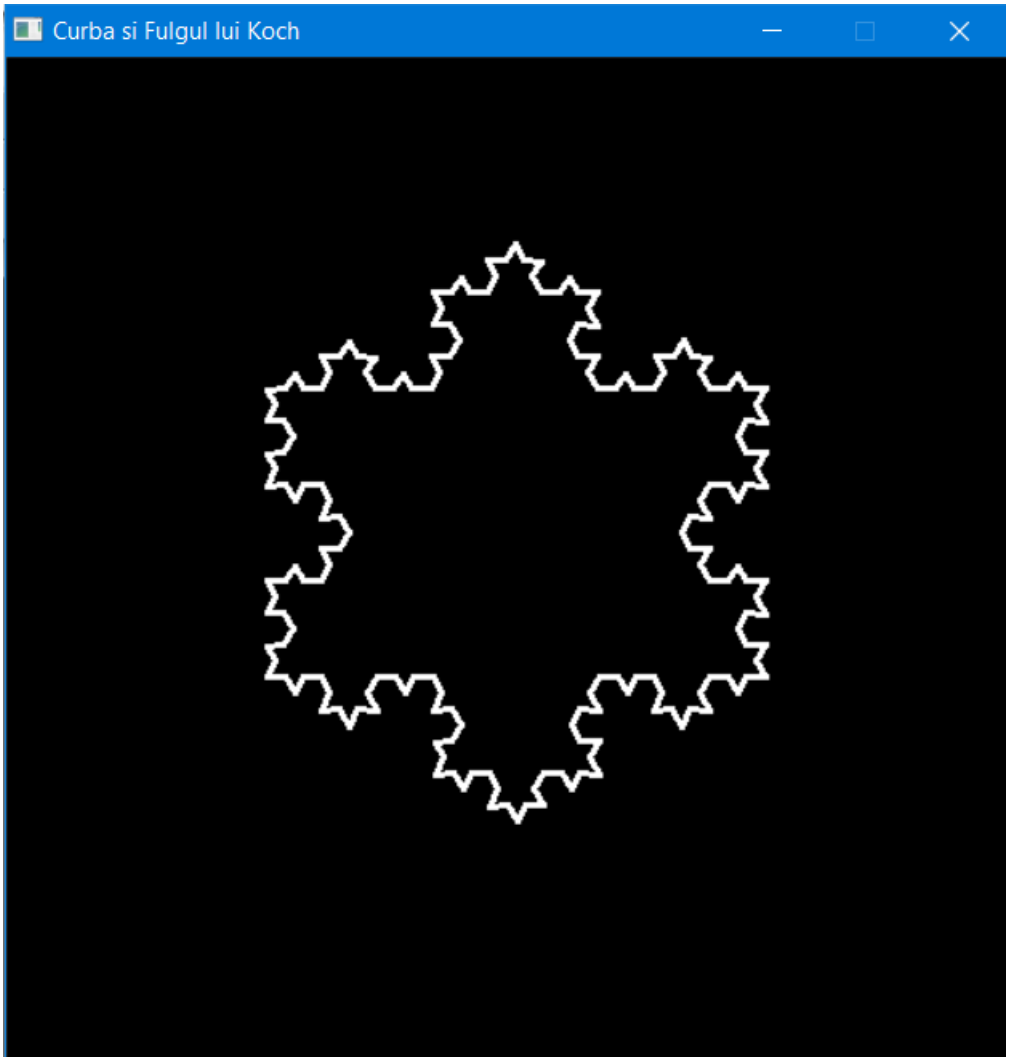
2. *Fulgul Koch:*

Legea de transformare – se împarte fiecare latură a triunghiului în trei părţi egale, în locul treimii din mijloc se reprezintă câte un triunghi echilateral, fără bază, obţinându-se o nouă figură cu 6 colţuri (steaua lui David) şi iteraţia poate continua ***[23]***.

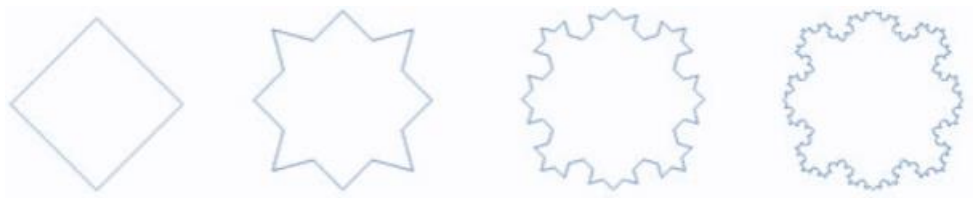
Implementare C++

```
#include <graphics.h>
#include <iostream>
#include <math.h>
using namespace std;
int L;
void rotplan(int xc,int yc, int x1, int y1,int &x,int &y,float unghi){
    x = ceil(xc+(x1-xc)*cos(unghi)-(y1-yc)*sin(unghi));
    y = ceil(yc+(x1-xc)*sin(unghi)+(y1-yc)*cos(unghi));
}
void desenez(int  x1,int  y1,  int  x2,int y2,int x3,int y3){
    moveto(x1,y1);
    lineto(div((2*x1+x2),3).quot,div((2*y1+y2),3).quot);
    lineto(x3,y3);
    lineto(div((x1+2*x2),3).quot,div((y1+2*y2),3).quot);
    lineto(x2,y2);
}
void generator(int x1,int y1,int x2,int y2,int n,int step) {
    int x,y;

rotplan(div((2*x1+x2),3).quot,div((2*y1+y2),3).quot,div((x1+2*x2),3).quot,div((y1+2
*y2),3).quot,x,y,M_PI/3);
    if (n<step){
        generator(x1,y1,div((2*x1+x2),3).quot,div((2*y1+y2),3).quot,n+1,step);
        generator(div((2*x1+x2),3).quot,div((2*y1+y2),3).quot,x,y,n+1,step);
        generator(x,y,div((x1+2*x2),3).quot,div((y1+2*y2),3).quot,n+1,step);
        generator(div((x1+2*x2),3).quot,div((y1+2*y2),3).quot,x2,y2,n+1,step);
    }
    else desenez(x1,y1,x2,y2,x,y);
}
int main(){
    int i,step=3,z=190;
    initwindow(500,500,"Curba si Fulgul lui Koch");
    setcolor(WHITE);
    for(i=1;i<=step;i++){
          setlinestyle(SOLID_LINE, 0, 3);
          L= getmaxx()/2;
          generator(130,getmaxy()-z,130+L,getmaxy()-z,1,step);
          generator(130+L,getmaxy()-z,130+div(L,2).quot, getmaxy()-z-
ceil(L*(sqrt(3)/2)),1,step);
          generator(130+div(L,2).quot, getmaxy()-z-ceil(L*(sqrt(3)/2)),130,
getmaxy()-z,1,step);
    }
    getch();
    closegraph();
}
```

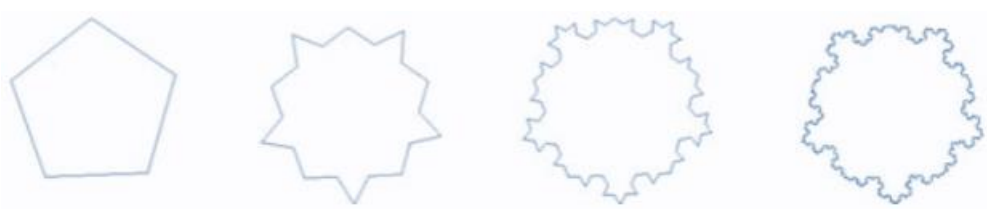
PENTRU N=3



- Generarea fractalului „Fulg de zăpadă” – iniţiator pătrat:



- Generarea fractalului „Fulg de zăpadă” – iniţiator pentagon regulat:

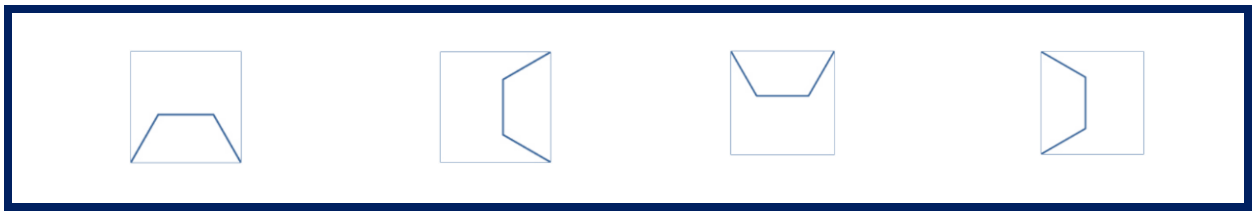


Problema 2: Geometrie fractală. Curba Sierpinski pentru pătrat

Sierpinski a găsit o curbă care poate să treacă prin fiecare punct al spațiului. Numim curba Sierpinski de ordinul k curba realizată după următoarea regulă:

1. *Trece prin fiecare nod al grilei 2^k*2^k;*
2. *Trece prin noduri vecine ale curbei* ***[23]***.

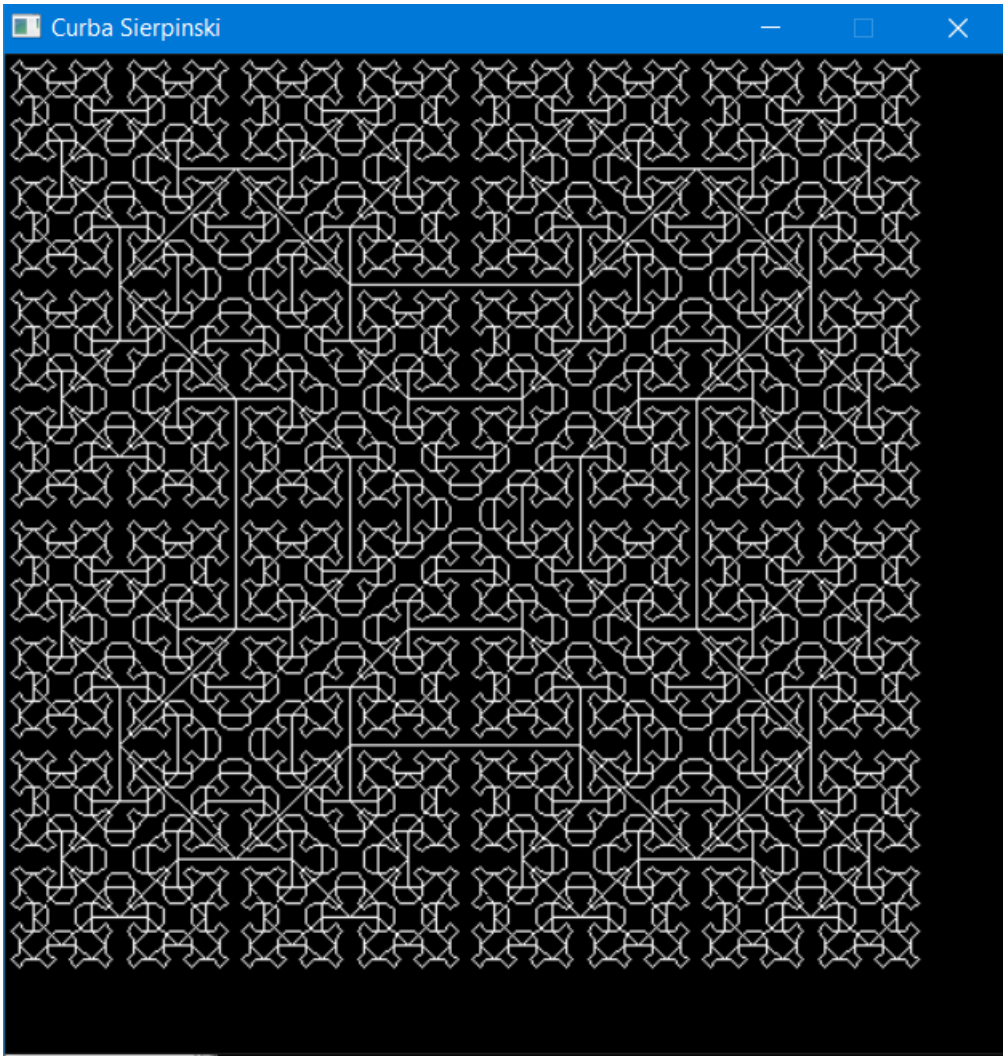
Curba Sierpinski are un element care se aplică pentru fiecare latură a pătratului. Pentru a construi curba Hilbert vom avea cele 4 combinații:



Implementare C++

```
#include <graphics.h>
#define n 5
#define Cadru 2048
#define SizeWin 460
int i,h,x,y; double k;
void A(int i); void B(int i); void C(int i); void D(int i);
void A(int i){
    if(i>0) {
        A(i-1);x=x+h;y=y-h;lineto((double)x*k,(double)y*k);
        B(i-1);x=x+h;x=x+h;lineto((double)x*k,(double)y*k);
        D(i-1);x=x+h;y=y+h;lineto((double)x*k,(double)y*k); A(i-1);    }
}
void B(int i){
    if(i>0) {
        B(i-1);x=x-h;y=y-h;lineto((double)x*k,(double)y*k);
        C(i-1);y=y-h;y=y-h;lineto((double)x*k,(double)y*k);
        A(i-1);x=x+h;y=y-h;lineto((double)x*k,(double)y*k); B(i-1);    }
}
void C(int i){
    if(i>0) {
        C(i-1);x=x-h;y=y+h;lineto((double)x*k,(double)y*k);
        D(i-1);x=x-h;x=x-h;lineto((double)x*k,(double)y*k);
        B(i-1);x=x-h;y=y-h;lineto((double)x*k,(double)y*k); C(i-1);    }
}
void D(int i){
    if(i>0) {
        D(i-1);x=x+h;y=y+h;lineto((double)x*k,(double)y*k);
        A(i-1);y=y+h;y=y+h;lineto((double)x*k,(double)y*k);
        C(i-1);x=x-h;y=y+h;lineto((double)x*k,(double)y*k); D(i-1);    }
}
int main(){
        initwindow(500,500,"Curba Sierpinski");
        i=0; h=div(Cadru, 4).quot;
        x=2*h; y=3*h; k=(double)SizeWin/Cadru;
        do{
            i=i+1; x=x-h; h=div(h, 2).quot; y=y+h;
            setcolor(15); setlinestyle(SOLID_LINE, 1, 1);
            moveto((double)x*k,(double)y*k);
            A(i);x=x+h;y=y-h;lineto((double)x*k,(double)y*k);
            B(i);x=x-h;y=y-h;lineto((double)x*k,(double)y*k);
            C(i);x=x-h;y=y+h;lineto((double)x*k,(double)y*k);
            D(i);x=x+h;y=y+h;lineto((double)x*k,(double)y*k);
            getch();
            ///cleardevice(); - curateste ecranul
        }
        while (!(i==n));
        closegraph();
}
```

PENTRU N=5, aici avem imprimate toate cele 3 etape împreună, funcția cleardevice nu se aplică pentru acest caz.

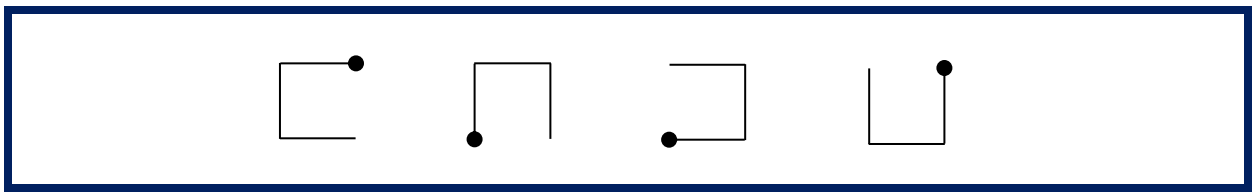


Problema 3: Geometrie fractală. Curba Hilbert.

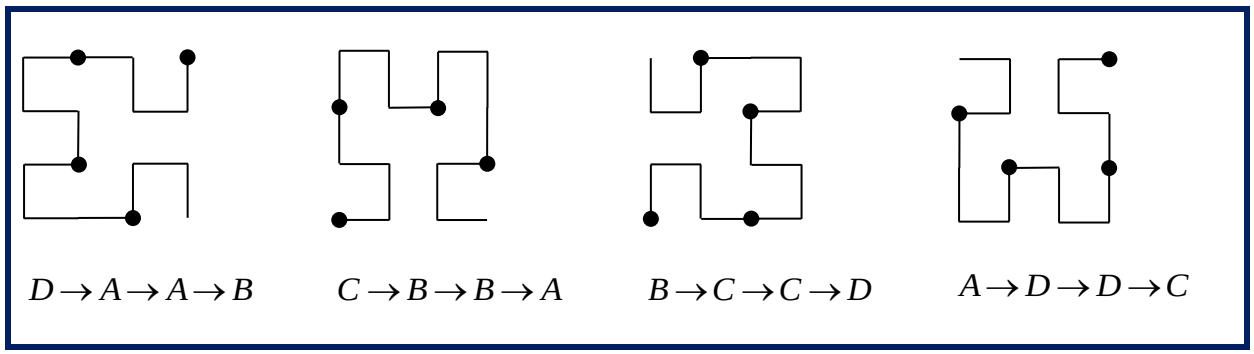
Hilbert a găsit o curbă care poate să treacă prin fiecare punct al spațiului. Numim curba Hilbert de ordinul k curba realizată după următoarea regulă:

1. *Trece prin fiecare nod al grilei 2^k*2^k;*
2. *Trece prin noduri vecine ale curbei* ***[23]***.

Curba Hilbert are următoarele elemente:



Pentru a construi curba Hilbert în baza elementelor A, B,C și D vom avea cele 4 combinații:

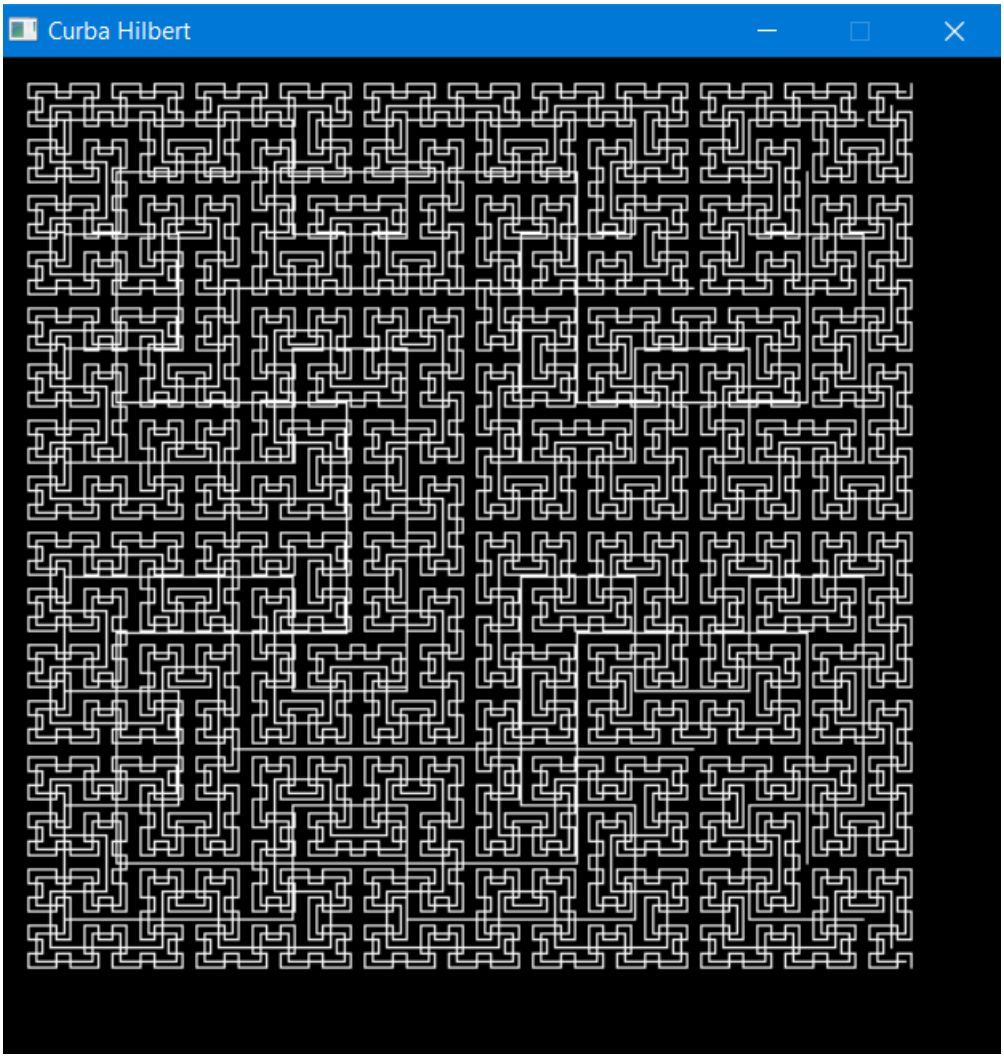


Implementare C++

```
#include <graphics.h>
#define n 3
#define h0 460
int i,h,x,y,x0,y0;
void A(int i); void B(int i); void C(int i); void D(int i);
void A(int i){
  if(i>0) {
    D(i-1); x=x-h; lineto(x,y); A(i-1); y=y-h; lineto(x,y);
    A(i-1); x=x+h; lineto(x,y); B(i-1); }
}
void B(int i){
  if(i>0) {
    C(i-1); y=y+h; lineto(x,y); B(i-1); x=x+h; lineto(x,y);
    B(i-1); y=y-h; lineto(x,y); A(i-1); }
}
void C(int i){
  if(i>0) {
    B(i-1); x=x+h; lineto(x,y); C(i-1); y=y+h; lineto(x,y);
    C(i-1); x=x-h; lineto(x,y); D(i-1); }
}
void D(int i){
  if(i>0) {
    A(i-1); y=y-h; lineto(x,y); D(i-1); x=x-h; lineto(x,y);
    D(i-1); y=y+h; lineto(x,y); C(i-1); }
}
int main(){
    initwindow(500,500,"Curba Hilbert");
    setcolor(15);
    i=0; h=h0;x0=div(h, 2).quot; y0=x0; setlinestyle(SOLID_LINE, 1, 1);
    do{
       i++; h=div(h, 2).quot; x0+=div(h, 2).quot;  y0+=div(h, 2).quot;
```

```
        x=x0; y=y0; moveto(x,y); A(i); getch();
        ///cleardevice(); - curateste ecranul
      }
      while(!(i==n));
      getch(); closegraph();
}
```

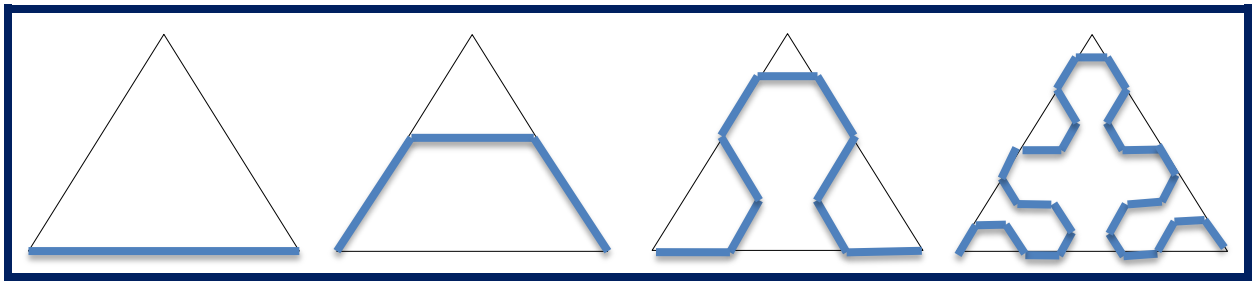
PENTRU N=6, aici avem imprimate toate cele 5 etape împreună, funcția cleardevice nu se aplică pentru acest caz.



SARCINI PENTRU EXERSARE

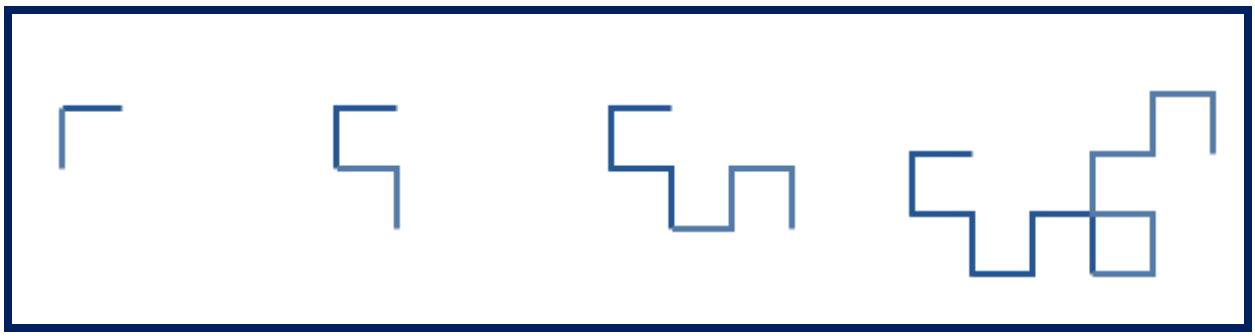
Să se scrie programe pentru fiecare curbă, afișați primele 4 iterații.

A) *Curba Sierpinski pentru triunghi, vezi mai jos primele 4 etape:*



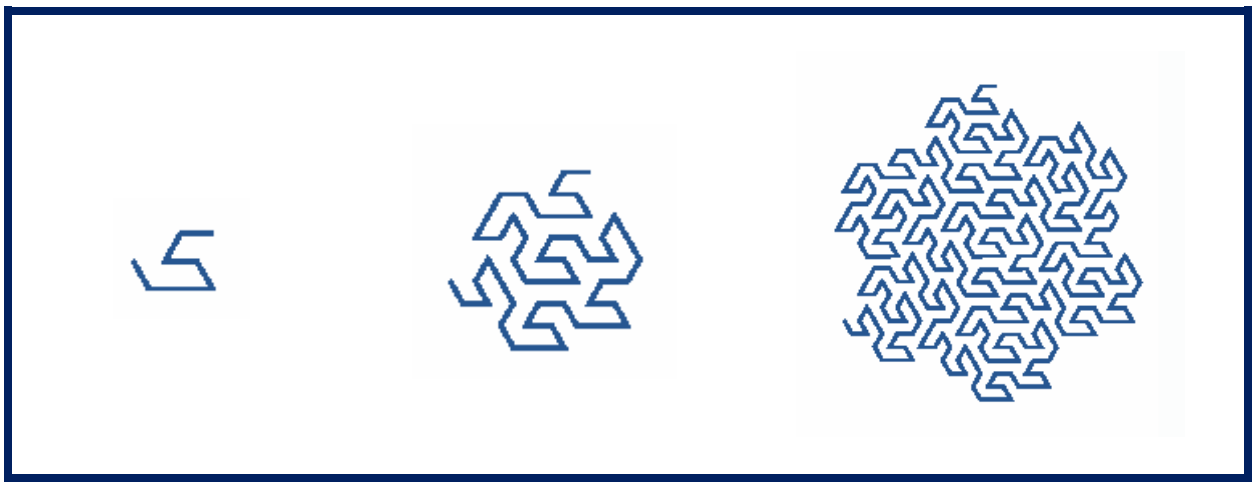
Informații cu privire la curba Sierpinski pentru triunghi găsiți aici:
https://en.wikipedia.org/wiki/Sierpiński_curve

B) *Curba Dragon, vezi mai jos primele 4 etape:*



Informații cu privire la curba Dragon găsiți aici:
https://en.wikipedia.org/wiki/Dragon_curve

C) *Curba Peano-Gosper, vezi mai jos primele 3 etape:*



Informații cu privire la curba Peano-Gosper găsiți aici:
https://en.wikipedia.org/wiki/Gosper_curve

METODA TRIERII

3.1 Noțiuni generale despre triere

Pe parcursul dezvoltării informaticii s-a stabilit că multe probleme de o reală importanţă practică pot fi rezolvate cu ajutorul unor metode standart, denumite tehnici de programare: recursia, trierea, metoda reluării, metodele euristice, etc.

Definiție:
Se numeşte metoda trierii o metodă ce indentifică toate soluţiile unei probleme în dependenţă de mulţimea soluţiilor posibile.

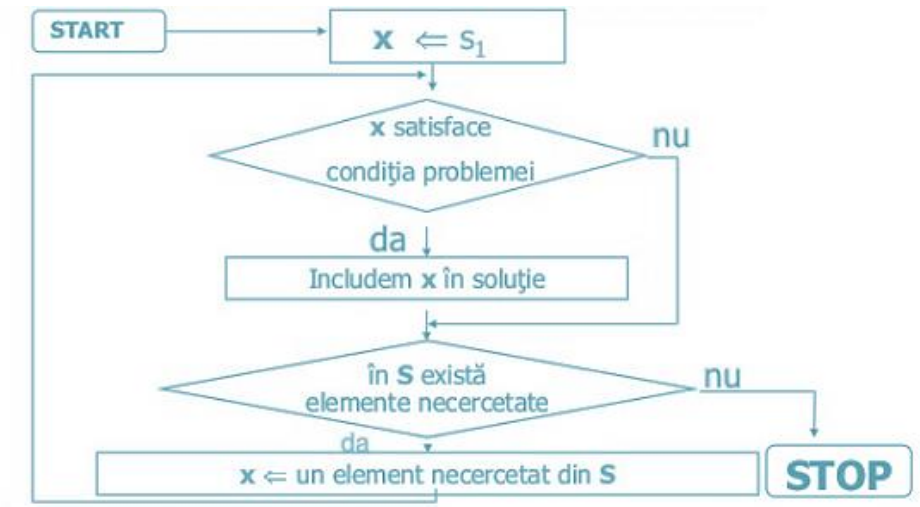
Prezentare generală
Fie P o problemă, soluţia căreia se află printre elementele mulţimii S cu un număr finit de elemente, $S = \{s_1, s_2, s_3, ..., s_n\}$. *Soluţia se determină prin analiza fiecărui element* s_i *din mulţimea S.*

Schema generală a unui algoritm bazat pe metoda trierii poate fi redată cu ajutorul unui ciclu:

```
for (i = 1; i ≤ n; i++){
        if (SolutiePosibila(s_i )) PrelucrareaSolutiei(s_i );
}
```

- ***SolutiePosibila*** *este o funcţie booleană care returnează valoarea true dacă elementul* s_i *satisface condiţiile problemei şi false în caz contrar.*
- ***PrelucrareaSolutiei*** *este un subprogram care efectuează prelucrarea elementului selectat. De obicei, în acest subprogram soluţia* s_i *este afişată la ecran.*

Schema de aplicare a metodei



Problemă prototip
Se consideră numerele naturale din mulţimea {1, 2, 3, ..., n}. Să se determine toate elementele acestei mulţimi, pentru care suma cifrelor este egală cu un număr dat m (pentru câte numere K din această mulţime).

În particular, pentru n=100 si m=2, în mulţimea {0, 1, 2, ..., 100} există 3 numere care satisfac condiţiile problemei: 2, 11 şi 20. Prin urmare, K=3.

Schema de rezolvare

Pentru i de la 1 până la n:

- *Se calculează suma cifrelor numărului i;*
- *Dacă suma cifrelor este egală cu m includem i în soluţie.*

Evident, mulţimea soluţiilor posibile S = {0, 1, 2, ..., n}. Înalgoritmul ce urmează suma cifrelor oricărui număr natural i, i∈ S, se calculează cu ajutorul funcţiei ***SumaCifrelor****. Separarea cifrelor zecimale din scrierea numărului natural "i" se efectuează de la dreapta la stânga prin împărţirea numărului "i" şi a câturilor respective la baza 10.*

Particularităţi de implementare

- *Generarea şi cercetarea consecutivă a elementelor mulţimii S.*
- *Utilizarea subprogramelor pentru fiecare din subproblemele:*
  - *Verificarea apartenenţei elementului cercetat* s_i *la soluţie;*
  - *Plasarea elementului curent în soluţie;*
  - *Generarea următorului element al mulţimii (dacă e necesar).*

Exemplu:

Să se scrie un program care determină toate secvenţele binare de lungime n, fiecare din ele conţinând nu mai puţin de k cifre de 1.

- *Intrare: numere naturale n, 1<n<20, şi k,unde k<n, se citesc de la tastatură.*
- *Ieşire: fiecare linie a fişierului text OUT.TXT va conţine câte o secvenţă binară distinctă, ce corespunde condiţiilor din enunţul problemei.*

Analiza problemei

Numărul secvenţelor binare de lungime n este 2^n*, finit, prin urmare, pentru problema dată poate fi aplicată metoda trierii.*

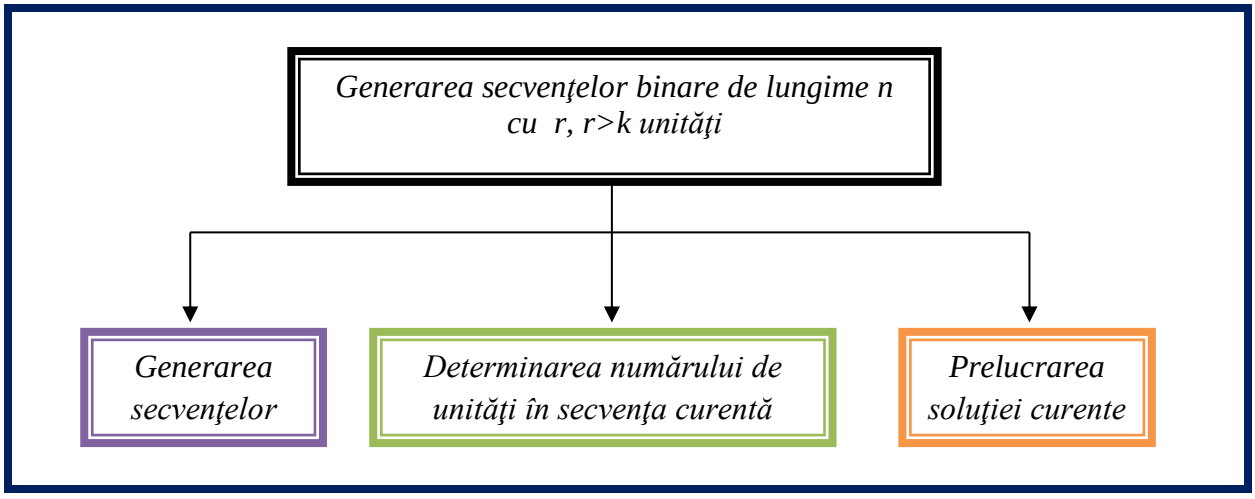
Modelul matematic

Elementele mulţimii S pot fi interpretate ca numere {0, 1,2, ..., 2^n-1}, reprezentate pe n poziţii binare. Pentru generarea consecutivă a secvenţelor binare se va utiliza formula:

$$s_0 = 0;\ \ s_1 = s_0 + 1 = 1;\ \ \ s_2 = s_1 + 1 = 2;\ \ s_3 = s_2 + 1 = 3, \ldots$$

$$s_i = s_{i-1} + 1, \ \ unde \ \ i = 1, 2, \ldots, 2^n - 1.$$

Separarea subproblemelor



Algoritmul de rezolvare

Iniţializăm variabilele n şi k, fişierul de ieşire, tabloul B.

- *Pasul 1. Cercetarea secvenţei curente. Se calculează numărul de unităţi (r) în secvenţa curentă B*
- *Pasul 2. Prelucrarea soluţiei. Dacă r ≥ k, secvenţa curentă B este înscrisă în fişierul de ieşire.*
- *Pasul 3. Verificarea prezenţei secvenţelor necercetate. Dacă r=n se închide fişierul de ieşire, apoi STOP.*
- *Pasul 4. Generarea secvenţei următoare. Dacă B[n]=0, atunci B[n] ⇐ 1*
  - *în caz contrar: i ⇐ n*
    - ✓ *atât timp cât B[i] = 1, repetăm B[i] ⇐ 0; i ⇐ i–1;*
    - ✓ *pentru indicele curent i B[i] ⇐ 1*
  - *Revenim la Pasul 1* ***[21]****.*

Aplicații ale metodei trierii

Metoda trierii poate fi folosită pentru rezolvarea următoarelor probleme din viață:

- *aflarea numărului minim de monede care pot fi date drept plată sau rest;*
- *medicii deseori se confruntă cu necesitatea aplicării metodei trierii cazurilor, când numărul răniților sau bolnavilor este foarte mare, medicul fiind suprasolicitat, în cazul unui război, sau când își periclitează propria viață în cazul unei epidemii periculoase;*
- *aflarea ariei maxime a unui lot de teren, având la dispoziție o anumită lungime de sârmă ghimpată, spre exemplu (ca perimetru dat);*
- *afișarea coordonatelor a două puncte date ce au distanță minimă sau maximă, ceea ce va fi foarte folositor dacă plănuim o călătorie;*
- *calcularea șanselor de a lua premiul mare la loterie etc.*

Diferența succintă dintre metoda trierii și tehnica greedy

Metoda trierii	Tehnica greedy
Se aplică numai în scopuri didactice sau pentru elaborarea unor programe simple.	*Se aplică mai des, cu condiția că din enunțul problemei poate fi redusă regula de selecție direct a elementelor necesare.*
Algoritmii sunt exponențiali, ceea ce înseamnă că necesită foarte mult timp și bineînțeles, resurse.	*Algoritmii sunt polinomiali - cea mai folosită clasă de complexitate, ceea ce înseamnă că necesită mai puțin timp și, bineînțeles, resurse.*

Observație:

- ❖ *Tehnica greedy va fi studiată într-un capitol aparte. De aceea în acest capitol nu se va face o abordare detaliată a tehnicii greedy.*

Notă:

- ✓ *Avantajul principal al algoritmilor bazaţi pe metoda trierii constă în faptul că programele respective sunt relativ simple, iar depanarea lor nu necesită teste sofisticate.*
- ✓ *Metoda trierii realizează operaţiile legate de prelucrarea datelor unor mulţimi:*
  - *reuniunea;*
  - *intersecţia;*
  - *diferenţa;*
  - *generarea tuturor submulţimilor;*
  - *generarea elementelor unui produs cartezian;*
  - *generarea permutărilor, aranjamentelor sau combinărilor de obiecte etc.*

3.2 Analiza complexității algoritmilor metodei trierii

Notă:

- ✓ *Complexitatea temporală a acestor algorimi este determinată de numărul de elemente k din mulţimea soluţiilor posibile S. În majoritatea problemelor de o reală importanţă practică metoda trierii conduce la algoritmiii exponenţiali.*
- ✓ *Întrucît algoritmii exponenţiali sunt inacceptabili în cazul datelor de intrare foarte mari, metoda trierii este aplicată numai în scopuri didactice sau pentru elaborarea unor programe al căror timp de execuţie nu este critic.*

Exemplul 1:

Se dă o matrice de dimensiuni NxM *de numere naturale. Se cere determinarea unei submatrici cu număr maxim de celule în care diferenţa între valoarea maximă şi valoarea minimă este mai mică decât o valoare dată* K. *Restricţii:* $2 \le N, M \le 150, 1 \le K \le 1\,000$.

Rezolvare:

Problema constă în determinarea submatricei cu număr maxim de celule în care diferenţa între valoarea maximă şi valoarea minimă este mai mică decât K. Vom folosi ideea de la problema cu subsecvenţa de sumă maximă pe matrice, adică vom fixa două linii i_1 *şi* i_2. *Acum pentru fiecare coloană j păstrăm în m[j] elementul minim şi în M[j] elementul maxim a[i][j] cu* $i_1 \le i \le i_2$. *Astfel, la fiecare pas trebuie acum să rezolvăm problema determinării unei subsecvenţe de lungime maximă pentru care diferenţa între elementul maxim şi minim este mai mică decât K.*

Folosind un max-heap şi un min-heap, putem parcurge elementele şirurilor M şi m în ordine, inserând la fiecare pas elemente în heap şi determinând cea mai lungă secvenţă ce se termină în i cu proprietatea dată. Vom ţine un al 2-lea indice j, iniţial 0, care atâta timp cât diferenţa între elementul maxim din max-heap cu elementul minim din min-heap este mai mare sau egală cu K, vom scoate din heapuri elementele M[j], respectiv m[j] şi îl vom incrementa pe j.

Complexitate: $O(N^3 \log N)$. *Dacă în loc de heapuri folosim deque-uri (este un acronim neregulat al cozii cu capăt dublu), atunci acest algoritm va avea complexitatea* $O(N^3)$.

Exemplul 2:

Se dă un şir de N *numere întregi şi un număr natural* K. *O secvenţă este un subşir de numere care apar pe poziţii consecutive în şirul iniţial. Se cere să se găsească secvenţa de sumă maximă de lungime cel puţin* K. *Restricţii:* $1 \le K \le N \le 50\,000$.

Rezolvare:

Putem folosi rezolvarea liniară bazată pe programare dinamică. Vom folosi şirul sumelor parţiale sum[], iar pentru a determina subsecvenţa de sumă maximă ce se termină în i *şi are lungimea cel puţin K trebuie să găsim sum[j] minim astfel ca* $j < i - K$.

Parcurgem lista, la pasul i *determinăm best[i] = sum[i] - min(sum[j]),* $j < i - K$. *Comparăm minimul curent cu sum[i - K] şi trecem la pasul următor.*

Complexitate: O(N) ***[20]***.

Observații:

- ❖ *Pentru a aplica metoda trierii, în alte țări se folosește cel mai des tipul de algoritm Greedy, care are rolul de a construi soluția optimă pas cu pas, la fiecare pas fiind selectat în soluție elementul care pare optim la momentul respectiv.*
- ❖ *De obicei, algoritmii bazați pe metoda Greedy sunt algoritmi polinomiali.*

3.3 Implementarea metodei trierii la rezolvarea problemelor elementare

Problema 1: Suma elementelor din mulțime egală cu k

Condiția problemei	*Exemplu*
Se consideră numerele naturale din mulțimea {1, 2, 3, ..., n}. Să se determine toate elementele acestei mulțimi, pentru care suma cifrelor este egală cu un număr dat k.	*Pentru n = 100 și k=10* *Programul va afișa:* `19 28 37 46 55 64 73 82 91` *Explicație:* $S=\begin{cases}19\rightarrow 1+9=10\\28\rightarrow 2+8=10\\37\rightarrow 3+7=10\end{cases}$

Implementare C++

```
#include <iostream>
using namespace std;
int n,m,i,k;
int SumaCifrelor(int num){
    int s=0;
    do{
       s=s+num%10;
       num=num/10;
    }
    while (num!=0);
    return s;
}
bool SolutiePosibila(int num){
    if (SumaCifrelor(num)==m)
        return true;
        else return false;
}
void PrelucrareaSolutiei(int num, int &k){
    cout<<num<<" "; k++;
}
int main(){
    cout<< "N= "; cin >> n;
    cout<< "M= "; cin >> m;
    cout<< "\nElemente cu proprietatea: \"";
    cout<< "Suma cifrelor numarului este "<<m<<"!\"\n\t";
    for (i=1; i<=n; ++i)
    if (SolutiePosibila(i))
        PrelucrareaSolutiei(i,k);
    if (k==0) cout <<"Nu exista!";
    return 0;
}
```

Rezultatele execuției:

```
N= 1000
M= 26

Elemente cu proprietatea: "Suma cifrelor numarului este 26!"
        899 989 998
```

Problema 2: Produsul elementelor din mulțime egală cu k

Condiția problemei	*Exemplu*
Se consideră numerele naturale din mulțimea {1, 2, 3, ..., n}. Să se determine toate elementele acestei mulțimi, pentru care produsul cifrelor este egală cu un număr dat k.	*Pentru n = 2000 și k=50* *Programul va afișa:* `255 525 552 1255 1525 1552` *Explicație:* $S=\begin{cases}255\rightarrow 2\cdot 5\cdot 5=50\\525\rightarrow 5\cdot 2\cdot 5=50\\552\rightarrow 5\cdot 5\cdot 2=50\end{cases}$

Implementare C++

```
#include <iostream>
using namespace std;
int n,m,i,k;
int ProdusulCifrelor(int num){
    int p=1;
    while (num!=0){
    p=p*(num%10);
    num=num/10;
    }
    return p;
}
bool SolutiePosibila(int num){
    if (ProdusulCifrelor(num)==m) return true;
        else return false;
}

void PrelucrareaSolutiei(int num, int &k){
    cout<<num<<" ";
    k++;
}
int main(){
    cout<< "N= "; cin >> n;
    cout<< "M= "; cin >> m;
    cout<< "\nElemente cu proprietatea:\n";
    cout<< "Produsul cifrelor numarului este "<<m<<"!\"\n\t";
    for (i=1; i<=n; ++i)
    if (SolutiePosibila(i))
        PrelucrareaSolutiei(i,k);
    if (k==0) cout <<"Nu exista!";
    return 0;
}
```

Rezultatele execuției:

```
N= 1000
M= 50

Elemente cu proprietatea:"Produsul cifrelor numarului este 50!"
        255 525 552
```

Notă:

- ✓ *Pentru problemele ce urmează, adaptați codul programului pentru elaborarea subprogramelor de bază pentru vizualizarea implementării metodei trierii în limbajul C++.*

Problema 3: Intersecția a două mulțimi A și B

Condiția problemei	*Exemplu*
Să se determine mulțimea care reprezintă intersecția a două mulțimi de numere întregi (reale). Elementele mulțimilor A și B vor fi introduse de la tastatură.	*Pentru A={1, 2, 3, 4, 5, 6, 7, 8} și B={3, 4, 5, 6, 7, 8, 9, 10}* *Programul va afișa:* 3 4 5 6 7 8 *Explicație:* *Verificăm dacă fiecare element din mulțimea A este și în mulțimea B, dacă careva element se conține în ambele mulțimi, atunci acest element îl vom transcrie în mulțimea soluției, adică C. Bine ar fi dacă soluția ar fi reprezentată de elemente aranjate ascendent sau descendent.* $A \cap B = \{3,4,5,6,7,8\}$

Implementare C++

```
#include <iostream>
using namespace std;
// Intersectia a doua multimi
int main(){
    int a[100],b[100],c[100],i,j,k=1,n,m;
    cout<<"\nNumarul de elemente al multimii A este: ";
    cin>>n;
    for (i=1;i<=n;i++) cin>>a[i];
    cout<<"\nNumarul de elemente al multimii B este: ";
    cin>>m;
    for (j=1;j<=m;j++) cin>>b[j];
    for (i=1;i<=n;i++)
        for (j=1;j<=m;j++){
            if (a[i]==b[j]){
                c[k]=a[i]; k++;
            }
        }
    cout<<"\nIntersectia multimilor: \n{ ";
    for (i=1;i<=n;i++)
        cout<<a[i]<<" "; cout<<"} si { ";
    for (j=1;j<=m;j++)
        cout<<b[j]<<" "; cout<<"} este: { ";
    for (i=1; i<k; i++)
        cout<<c[i]<<" "; cout<<" }";
    return 0;
}
```

Rezultatele execuției:

```
Numarul de elemente al multimii A este: 8
1 2 3 4 5 6 7 8

Numarul de elemente al multimii B este: 8
3 4 5 6 7 8 9 10

Intersectia multimilor:
{ 1 2 3 4 5 6 7 8 } si { 3 4 5 6 7 8 9 10 } este: { 3 4 5 6 7 8  }
```

Problema 4: Reuniunea a două mulțimi A și B

Condiția problemei	*Exemplu*
Să se determine mulțimea care reprezintă reuniunea a două mulțimi de numere întregi (reale). Elementele mulțimilor A și B vor fi introduse de la tastatură.	*Pentru A={1, 2, 3, 4, 5, 6, 7, 8} și B={3, 4, 5, 6, 7, 8, 9, 10}* *Programul va afișa:* `1 2 3 4 5 6 7 8 9 10` *Explicație:* *Pentru a obține reuniunea mulțimilor, vom uni elementele ambelor mulțimi și vom exclude toate elementele care se repetă. Bine ar fi dacă soluția ar fi reprezentată de elemente aranjate ascendent sau descendent.* $A \cup B = \{1,2,3,4,5,6,7,8,9,10\}$

Implementare C++

```
#include<iostream>
using namespace std;
int main(){
    int a[20],b[20],c[50],i,j,m,n,k=0,gasit;
    cout<<"\nNumarul de elemente al multimii A este: ";
    cin>>n;
    for(i=0;i<=n-1;i++) cin>>a[i];
    cout<<"\nNumarul de elemente al multimii B este: ";
    cin>>m;
    for(j=0;j<=m-1;j++) cin>>b[j];
       for(i=0;i<=n-1;i++){
              gasit=0;
         for(j=0;j<=m-1 && !gasit;j++)
         if (a[i]==b[j]) gasit=1;
         if (!gasit) c[k++]=a[i];
       }
    cout<<"\nReuniunea multimilor: \n{ ";
    for (i=0;i<=n-1;i++)
        cout<<a[i]<<" "; cout<<"} si { ";
    for (j=0;j<=m-1;j++)
        cout<<b[j]<<" "; cout<<"} este:\n\n{ ";
    for(i=0; i<=m-1; i++)
        cout<<b[i]<<" ";
    for(i=0; i<k; i++)
        cout<<c[i]<<" "; cout<<"}";
 }
```

Rezultatele execuției:

```
Numarul de elemente al multimii A este: 8
1 2 3 4 5 6 7 8

Numarul de elemente al multimii B este: 8
3 4 5 6 7 8 9 10

Reuniunea multimilor:
{ 1 2 3 4 5 6 7 8 } si { 3 4 5 6 7 8 9 10 } este:

{ 3 4 5 6 7 8 9 10 1 2 }
```

Problema 5: Diferența a două mulțimi A și B

Condiția problemei	*Exemplu*
Să se determine mulțimea care reprezintă diferența a două mulțimi de numere întregi (reale). Elementele mulțimilor A și B vor fi introduse de la tastatură.	*Pentru A={1, 2, 3, 4, 5, 6, 7, 8} și B={3, 4, 5, 6, 7, 8, 9, 10}* *Programul va afișa:* 1 2 *Explicație:* *Pentru a obține diferența mulțimilor, A-B, vom verifica elementele ce sunt în mulțimea A și nu sunt în mulțimea B.* *Pentru a obține diferența mulțimilor, A-B, vom verifica elementele ce sunt în mulțimea A și nu sunt în mulțimea B.* *Bine ar fi dacă soluția ar fi reprezentată de elemente aranjate ascendent sau descendent.* $A \setminus B = \{1,2\}$ & $B \setminus A = \{9,10\}$

Implementare C++

```
#include<iostream>
using namespace std;
int main(){
    int a[20],b[20],c[50],i,j,m,n,k=0,gasit;
    cout<<"\nNumarul de elemente al multimii A este: ";
    cin>>n;
    for(i=0;i<=n-1;i++) cin>>a[i];
    cout<<"\nNumarul de elemente al multimii B este: ";
    cin>>m;
    for(j=0;j<=m-1;j++) cin>>b[j];

       for(i=0;i<=n-1;i++){
             gasit=0;
         for(j=0;j<=m-1 && !gasit;j++)
         if (a[i]==b[j]) gasit=1;
         if (!gasit) c[k++]=a[i];
       }
    cout<<"\nDiferenta multimilor, A-B: \n{ ";
    for (i=0;i<=n-1;i++) cout<<a[i]<<" ";
        cout<<"} si { ";
    for (j=0;j<=m-1;j++) cout<<b[j]<<" ";
        cout<<"} este: { ";
       for(i=0; i<k; i++)
         cout<<c[i]<<" "; cout<<"}";
}
```

Rezultatele execuției:

```
Numarul de elemente al multimii A este: 8
1 2 3 4 5 6 7 8

Numarul de elemente al multimii B este: 8
3 4 5 6 7 8 9 10

Diferenta multimilor, A-B:
{ 1 2 3 4 5 6 7 8 } si { 3 4 5 6 7 8 9 10 } este: { 1 2 }
```

Problema 6: Generarea tuturor submulțimilor

Condiția problemei	*Exemplu*
Să se genereze toate submulțimile mulțimii A de numere întregi. Numărul de elementele al mulțimii A va fi introdus de la tastatură.	*Analizați cazurile pentru n=2 și n=3:*

Pentru n=2	*Pentru n=3*
1) 2 2) 1 3) 1 2	1) 3 2) 2 3) 2 3 4) 1 5) 1 3 6) 1 2 7) 1 2 3

Implementare C++

```
#include<iostream>
using namespace std;
int a[10], n, i, S;
int main(){
    cout<<"\nNumarul de elemente al multimii A este: ";
    cin>>n;
    for(i=0; i<n;i++)
    a[i]=0;
    do{
        a[n-1]++;
        for(i=n-1; i>=1; i--)
             if(a[i]>1){
               a[i]-=2; a[i-1]+=1;
             }
             S=0;
             for(i=0; i<n; i++)
             S+=a[i];
             for(i=0; i<n; i++)
             if(a[i]) cout<<i+1<<" ";
             cout<<endl;
    }
    while(S<n);
    cout<<"Multime vida!";
}
```

Rezultatele execuției:

```
Numarul de elemente al multimii A este: 4
4
3
3 4
2
2 4
2 3
2 3 4
1
1 4
1 3
1 3 4
1 2
1 2 4
1 2 3
1 2 3 4
Multime vida!
```

Problema 7: Produsul cartezian a două mulțimi A și B

Condiția problemei	*Exemplu*
Să se genereze produsul cartezian a două mulțimi A și B de numere întregi. Elementele mulțimilor A și B vor fi introduse de la tastatură.	*Pentru A={1, 2, 3, 4, 5} și B={6, 7, 8, 9, 10}* *Programul va afișa:* $\{1,6\}, \{1,7\}, \{1,8\}, \{1,9\}, \{1,10\},$ $\{2,6\}, \{2,7\}, \{2,8\}, \{2,9\}, \{2,10\},$ $\{3,6\}, \{3,7\}, \{3,8\}, \{3,9\}, \{3,10\},$ $\{4,6\}, \{4,7\}, \{4,8\}, \{4,9\}, \{4,10\},$ $\{5,6\}, \{5,7\}, \{5,8\}, \{5,9\}, \{5,10\}$ *Explicație:* *Pentru a obține produsul cartezian a două mulțimi A și B, vom cupla (împerechea) fiecare element din mulțimea A cu fiecare element din mulțimea B*

Implementare C++

```
#include<iostream>
using namespace std;
int main(){
    int i, m, n, j;
    char a[20], b[20];
    cout<<"\nNumarul de elemente al multimii A este: ";
    cin>>n;
    for(i=1; i<=n; i++) cin>>a[i];
        cout<<"\nNumarul de elemente al multimii B este: ";
    cin>>m;
    for(j=1; j<=m; j++) cin>>b[j];
        cout<<"\nProdusul cartezian, AxB: \n";
    for(i=1; i<=n; i++)
    for(j=1; j<=m; j++)
        cout<<" {"<<a[i]<<" "<<b[j]<<"}";
}
```

Rezultatele execuției:

```
Numarul de elemente al multimii A este: 8
1 2 3 4 5 6 7 8

Numarul de elemente al multimii B este: 8
9 10 11 12 13 14 15 16

Produsul cartezian, AxB:

 {1 9} {1 1} {1 0} {1 1} {1 1} {1 1} {1 2} {1 1} {2 9} {2 1} {2 0} {2 1}
 {2 1} {2 1} {2 2} {2 1} {3 9} {3 1} {3 0} {3 1} {3 1} {3 1} {3 2} {3 1}
 {4 9} {4 1} {4 0} {4 1} {4 1} {4 1} {4 2} {4 1} {5 9} {5 1} {5 0} {5 1}
 {5 1} {5 1} {5 2} {5 1} {6 9} {6 1} {6 0} {6 1} {6 1} {6 1} {6 2} {6 1}
 {7 9} {7 1} {7 0} {7 1} {7 1} {7 1} {7 2} {7 1} {8 9} {8 1} {8 0} {8 1}
 {8 1} {8 1} {8 2} {8 1}
```

Problema 8: Suma numerelor prime

Condiția problemei	*Exemplu*
Se citește de la tastatură un număr natural par. Să se decidă dacă acesta poate fi scris ca și sumă de două numere prime și să se afișeze toate soluțiile găsite (se va considera că și 1 este număr prim). (Conjectura lui Goldbach: "Orice număr par mai mare decat 2 este sumă a două numere prime.").	*Pentru X=100* *Programul va afișa:* $100 = 3 + 97$ $100 = 11 + 89$ $100 = 17 + 83$ $100 = 29 + 71$ $100 = 41 + 59$ $100 = 47 + 53$ *Explicație:* *Numerele prime până la 100 sunt: 2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53, 59, 61, 67, 71, 73, 79, 83, 89, 97. Acestea sunt numerele prime pana la 100. Singurul număr par din sirul de mai sus este 2, resul numerelor prime fiind impare.*

Implementare C++

```
#include <iostream>
using namespace std;
bool NumarPrim(int n){
    bool prim = true;
    int i = 2;
    while(i <= n/2){
        if(n % i == 0) prim = false;
        i++;
    }
    return prim;
}
int main(){
    int n,p,q=0;
    cout<<"Introduceti numarul: ";
    cin>>n;
    for (p =1; p<= n/2; p=p + 2){
        if (NumarPrim(p)){
            q = n - p;
            if (NumarPrim(q))
                cout<<n<<" = "<<p<<" + "<<q<<endl;
        }
    }
    return 0;
}
```

Rezultatele execuției:

```
Introduceti numarul: 90
        90 = 1 + 89
        90 = 7 + 83
        90 = 11 + 79
        90 = 17 + 73
        90 = 19 + 71
        90 = 23 + 67
        90 = 29 + 61
        90 = 31 + 59
        90 = 37 + 53
        90 = 43 + 47
```

Problema 9: Factori primi

Condiția problemei	*Exemplu*
Să se descompună un număr natural n ca produs de factori primi.	*Pentru N=360* *Programul va afișa:* 2 3 3 2 5 1 *Explicație:* *Considerăm toate numerele naturale începand cu 2 .Pentru fiecare număr verificăm dacă este divizor al lui n.* *Dacă da, calculăm multiplicitatea acestui divisor în n împărțind succesiv pe n la divizor și calculăm numărul de împărțiri efectuate atât timp cât n mai are divizori. Așadar descompunerea în factori primi a numărului 360 va fi: $2^3 * 3^2 * 5^1$*

Pentru n=30	*Pentru n=420*	*Pentru n=3850*
$\left.\begin{matrix}2^1\\3^1\\5^1\end{matrix}\right\}\Rightarrow 2^1\cdot 3^1\cdot 5^1$	$\left.\begin{matrix}2^2\\3^1\\5^1\\7^1\end{matrix}\right\}\Rightarrow 2^2\cdot 3^1\cdot 5^1\cdot 7^1$	$\left.\begin{matrix}2^1\\5^2\\7^1\\11^1\end{matrix}\right\}\Rightarrow 2^1\cdot 5^2\cdot 7^1\cdot 11^1$

Implementare C++

```
#include <iostream>
using namespace std;
int main(){
    int n, m, p;
    cout<<"Introduceti numarul natural: "; cin>>n;
    m = n;
    cout<<"Factorii primi din descomunerea lui "<<n<<" sunt:\n";
    for (int d=2; d<=n/2; d++){
        if (m%d==0){
            p = 0;
            while (m%d==0){
                p++; m/=d;
            }
            cout<<"\t"<<d<<"^"<<p<<endl;
        }
        if (m == 1) break;
    }
    return 0;
}
```

Rezultatele execuției:

```
Introduceti numarul natural: 360
Factorii primi din descomunerea lui 360 sunt:
        2^3
        3^2
        5^1
```

Problema 10: Număr sumă de pătrate perfecte

Condiția problemei	*Exemplu*
Se citeşte de la tastatură un număr natural. Să se decidă dacă acesta poate fi scris ca şi sumă de două pătrate şi să se afişeze toate soluţiile găsite.	*Analizați cazurile pentru n=1000 şi n=5000:*

Pentru n=1000	*Pentru n=5000*
$\left.\begin{matrix}10^2+30^2\\18^2+26^2\end{matrix}\right\}\Rightarrow 1000$	$\left.\begin{matrix}10^2+70^2\\34^2+62^2\\50^2+50^2\end{matrix}\right\}\Rightarrow 5000$

Implementare C++

```
#include <iostream>
#include <math.h>
using namespace std;
bool verif_nr(float nr){
    bool patrat=false;
    if ((sqrt(nr)*sqrt(nr)) == nr)
    patrat=true;
    return patrat;
}
int main(){
    float nr, nr1, nr2=0;
    cout<<"Introduceti numarul natural: ";
    cin>>nr;
    cout<<"Suma de patrate perfecte: \n";
    for(nr1=2; nr1<=nr/2; nr1++){
        if(verif_nr(nr1)){
            nr2=nr-nr1;
            if(verif_nr(nr2))
            cout<<"\t"<<nr<<"="<<sqrt(nr1)<<"^2"<<" + "<<sqrt(nr2)<<"^2"<<endl;
        }
    }
    return 0;
}
```

Rezultatele execuției:

```
Introduceti numarul natural: 98653210
Suma de patrate perfecte:
        9.86532e+007=802^2 + 9900^2
        9.86532e+007=2260.86^2 + 9671.7^2
        9.86532e+007=2403^2 + 9637.36^2
        9.86532e+007=4047.23^2 + 9070.45^2
```

SARCINI PENTRU EXERSARE

SETUL DE PROBLEME NR. 1

1. *Se consideră mulţimea P={ P1, P2, ..., Pn} formată din n puncte (2 ≤ n ≤ 30) pe un plan Euclidian. Fiecare punct Pj este definit prin coordonatele sale Xj, Yj. Elaboraţi un program care afişează la ecran coordonatele punctelor Pa, Pb distanţa dintre care este maximă.*

2. *Se consideră mulţimea P={ P1, P2, ..., Pn} formată din n puncte (2 ≤ n ≤ 30) pe un plan Euclidian. Fiecare punct Pj este definit prin coordonatele sale Xj, Yj. Elaboraţi un program care afişează la ecran coordonatele punctelor Pa, Pb distanţa dintre care este minimă.*

3. *Având la dispoziție n săculeti cu monede S_1, S_2, ... S_n, fiecare săculet S_i conținând N_i monede de valoare V_i să se afișeze toate modalitățile de a plăti o sumă dată, S folosind numai monezi din săculețe.*

4. *Se citesc două numere naturale n și s (n<=10, s<=20). Afișați în ordine crescătoare toate numerele cu n cifre care au suma cifrelor egală cu s și în care oricare două cifre alăturate au paritate diferită.*
Exemplu: n=4, s=8 => 1016, 1034, 1052, 1070, 1214, 1232, 7010

5. *Se citește un cuvânt s (cu cel mult 10 caractere litere mici distincte). Să se genereze și să se afișeze toate anagramele cuvântului s în care consoanele sunt puncte fixe.*
Exemplu: s=alinus => alinus, alunis, ilanus, ilunas, ulanis, ulinas.

6. *Suma numerelor nenegative a,b,c,d,e,f, g este 1. Fie S cea mai mică dintre sumele:*
a) a+b+c, b) b+c+d, c) c+d+e, d) d+e+f, e) e+f+g.
Să se determine cea mai mare valoare posibilă a lui s.

7. *Suma numerelor nenegative a,b,c,d,e,f, g este 1. Fie S cea mai mare dintre sumele:*
a) a+b+c, b) b+c+d, c) c+d+e, d) d+e+f, e) e+f+g.
Să se determine cea mai mică valoare posibilă a lui s.

8. *Două orașe A, B sunt situate respectiv la 10 km și 15 km de un râu rectiliniu, iar proiecția lungimii AB pe direcția râului este de 20 km. Cele două orașe trebuie alimentate cu apă de la o uzină amplasată pe marginea râului. Se cere pozitia uzinei pentru care lungimea conductelor ce o leagă de cele două orașe să fie minimă* ***[24]****.* ***(Soluție:*** $x = 8\,Km$ ***)***

9. *De descompus numărul 180 în sumă de 3 termeni, în așa fel ca raportul a 2 termeni să fie 1:2, iar produsul lor să fie maxim.* ***(Soluție:*** $x = 40 \;\&\; y = 60 \;\&\; z = 80$ ***)***

10. *De găsit aşa un număr strict pozitiv, care să fie mai mare decât pătratul său cu o valoare maxim posibilă.* ***(Soluție:*** $x_{max} = \frac{1}{2} \;\&\; d_{max}(x_{max}) = \frac{1}{4}$ ***)***

SETUL DE PROBLEME NR. 2

1) *Se citesc două numere naturale n și k. Generați și afișați toate numerele naturale formate din n cifre care conțin exact k cifre de 1.*

2) *Se citește un număr natural n și apoi n culori distincte date ca șiruri de caractere. Afișați toate steagurile care se pot forma cu câte 3 culori diferite.*

3) *Presupunem că puterea P(t) emisă la descărcarea unui dispozitiv electronic la fiecare moment t>0 este* $P(t)=t^3 \cdot e^{-0.2 \cdot t}$ *(puterea fiind măsurată în watz, iar timpul în secunde). Să se afle:*
- *La ce moment puterea va fi maximă;*
- *Între ce limite (margini) variază puterea P(t) în intervalul de timp, t aparține [10, 20].*

4) *Să se determine punctul de pe graficul funcției* $f(x)=2 \cdot \sqrt{x}$ *aflat la distanţa minimă de punctul A(2,0).*

5) *Într-o sferă de raza R este înscris un con, cu aria suprafeţei laterale maximă. Aflaţi aria suprafeţei laterale a acestui con.*

6) *Într-o sferă este înscris un cilindru cu aria suprafeţei laterale maximă. De aflat raportul dintre raza sferei şi raza bazei acestui cilindru.*

7) *Într-o sferă este înscris un cilindru cu volumul maxim. De cîte ori volumul sferei e mai mare decât volumul acestui cilindru.*

8) *Într-o sferă de raza R este înscris un cilindru cu aria suprafeţei laterale maximă. De aflat volumul acestui cilindru.*

9) *Într-un con este înscris un cilindru cu aria suprafeţei laterale maximă. De aflat raportul dintre lungimea înălţimii conului la lungimea înălţimii acestui cilindru.*

10) *Într-un con este înscris un cilindru cu volumul maxim. De aflat raportul dintre lungimea razei bazei conului la lungimea razei bazei cilindrului.*

11) *Prin punctul N(2; 4) este dusă o dreaptă. Punctele de intersecţie a dreptei cu axele sistemului de coordonate (x>0 şi y>0) împreună cu originea sistemului de coordonate formează un triunghi dreptunghic. Care este lungimea celei mai mari catete ale acestui triunghi, pentru ca aria triunghiului să fie minimă?*

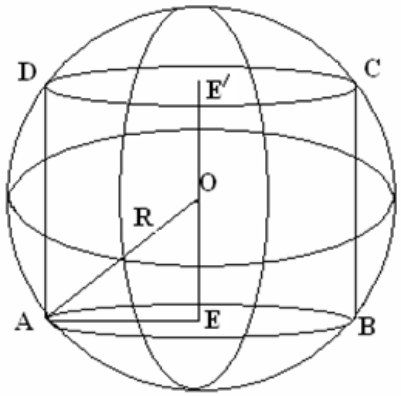
12) *Volumul unei prisme triunghiulare regulate este egală cu V. Să se afle lungimea laturii bazei prismei, astfel încât aria totală a prismei să fie minimă.*

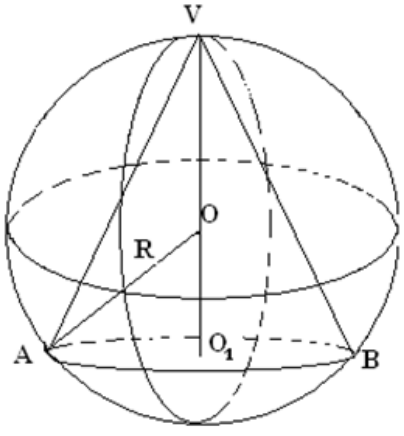
13) *Aflaţi laturile unui dreptunghi, perimetrul căruia este de 72 cm, iar aria dreptunghiului este maximă.*

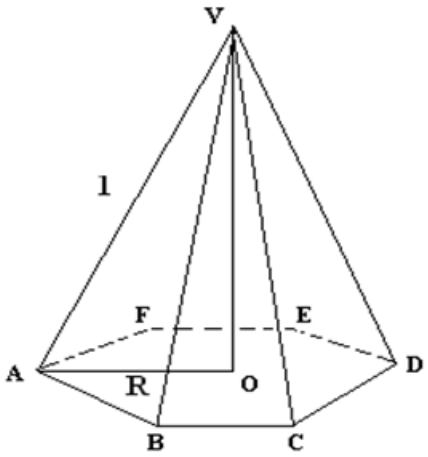
14) *O cutie are forma unui cilindru a cărui volum este de 1 dm^3 .Care este lungimea razei bazei pentru ca aria suprafeţei totale să fie minimă.*

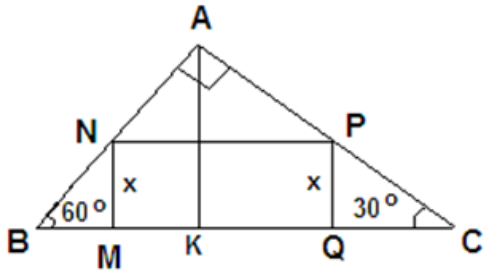
15) *Dintr-o bară metalică de formă cilindrică se obţine prin strunjire o bară paralelipipedică. Să se determine dimensiunile dreptunghiului de secţiune astfel încât pierderea de material să fie minimă* ***[25]***.

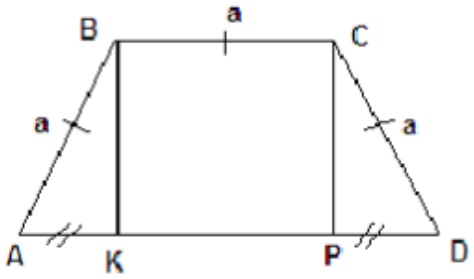
SETUL DE PROBLEME NR. 3

Condiția problemei 1	***Desenul***
Se dă o sferă de rază R, în care este înscris un cilindru cu aria suprafeţei laterale maximă. De aflat înălţimea cilindrului. ***Soluție:*** $h = R\sqrt{2}$ & $S_{Lat.} = 2\pi R^2$	

Condiția problemei 2	***Desenul***
Într-o sferă e înscris un con cu volumul maxim. De aflat raportul dintre raza sferei şi înălţimea conului ***[26]***. ***Soluție:*** $\frac{R}{H} = \frac{3}{4}$	

Condiția problemei 3	*Desenul*
Într-o piramidă hexagonală regulată lungimea muchiei laterale este de 1 cm. Care trebuie să fie lungimea laturii bazei pentru ca volumul piramidei să fie maxim? ***Soluție:*** $x = R = \sqrt{\frac{2}{3}}\,cm \quad \& \quad V_{\max} = \frac{1}{3}\,cm^3$	

Condiția problemei 4	*Desenul*
Într-un triunghi dreptunghic cu lungimea ipotenuzei de 24 cm şi măsura unui unghi ascuțit de 60^o este înscris un dreptunghi, baza căruia se află pe ipotenuză. Care sunt lungimile laturilor dreptunghiului, pentru ca aria dreptunghiului să fie maximă. ***Soluție:*** $a = 3\sqrt{3}\,cm \quad \& \quad b = 12\,cm \quad \& \quad S_{\max} = 36\sqrt{3}\,cm^2$	

Condiția problemei 5	*Desenul*
Într(un trapez isoscel, laturile laterale sunt congruente cu baza mică a trapezului, lungimea căruia este a m (metri) Care este lungimea bazei mari a trapezului, pentru ca aria trapezului să fie maximă? ***Soluție:*** $a = 3\sqrt{3}\,cm \quad \& \quad b = 12\,cm \quad \& \quad S_{\max} = 36\sqrt{3}\,cm^2$	

Remarcă:

- *Pentru problemele 1-5 din setul 3 stabiliți care va fi rezultatul dacă cerința ar fi de a obține valoare minimă în loc de maximă.*

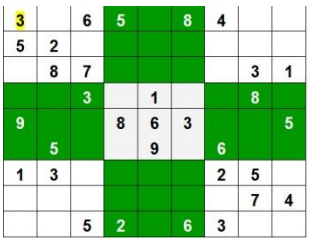
METODA BACKTRACKING

4.1 Noțiuni generale despre backtracking

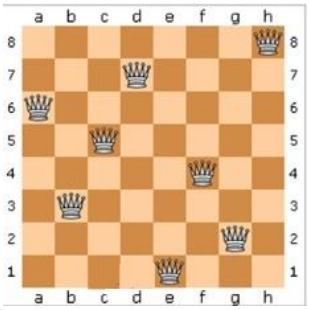
Definiție:
Backtracking este un algoritm general pentru a găsi toate (sau unele) soluții la unele probleme de calcul, în special probleme de satisfacție de constrângere, care crește gradual candidații la soluții și abandonează un candidat („backtracks") imediat ce stabilește că acest candidat nu poate fi completat cu o soluție valabilă.

Notă:

- ✓ *Metoda backtracking mai poate fi întâlnită în unele resurse ca metoda reluării.*
- ✓ *Există trei tipuri de probleme în baza metodei backtracking:*
  1. *Problema deciziei - în această situație, căutăm o soluție fezabilă.*
  2. *Problema de optimizare - în această situație, căutăm cea mai bună soluție.*
  3. *Problema de enumerare - în această situație, găsim toate soluțiile fezabile.*
- *Termenul "backtrack" a fost creat de matematicianul american D. H. Lehmer în anii '50. Limbajul pionier de prelucrare a șirurilor SNOBOL (1962) ar fi fost primul care a furnizat o instalație de backtracking generală.*



De exemplu, luăm în considerare problema de rezolvare a Sudoku, încercăm să umplem cifre una câte una. Ori de câte ori descoperim că cifra curentă nu poate duce la o soluție, o eliminăm (backtrack) și încercăm următoarea cifră. Aceasta este mai bună decât abordarea naivă (generează toate combinațiile posibile de cifre și apoi încearcă fiecare combinație una câte una), deoarece scade un set de permutări ori de câte ori se întoarce.



Exemplul clasic de utilizare a backtracking este puzzle-ul cu opt regine, care solicită toate aranjamentele a opt regine pe o tablă de șah standard, astfel încât nici o regină să nu atace pe niciuna. În abordarea comună de backtracking, candidații parțiali sunt aranjamente de k regine în primele k rânduri ale consiliului, toate în rânduri și coloane diferite. Orice soluție parțială care conține două regine care se atacă reciproc poate fi abandonată. Backtracking-ul poate fi aplicat numai pentru problemele care admit conceptul de „soluție parțială de candidat" și un test relativ rapid dacă poate fi completat cu o soluție validă. Este inutil, de exemplu, pentru localizarea unei valori date într-un tabel neordonat.

Cu toate acestea, atunci când este aplicabil, backtracking-ul este adesea mult mai rapid, decât enumerarea forței brute a tuturor candidaților, deoarece poate elimina mulți candidați cu un singur test.

Backtracking este un instrument important pentru soluționarea problemelor de satisfacție de constrângere, cum ar fi cuvintele încrucișate, aritmetica verbală, Sudoku și multe alte puzzle-uri. Este adesea cea mai convenabilă (dacă nu cea mai eficientă) tehnică pentru analizare, pentru problema rucsacului și alte probleme de optimizare combinatorie. De asemenea, este baza așa-numitelor limbaje de programare logică, cum ar fi Icon, Planificator și Prolog.

Backtracking-ul depinde de „procedurile de cutie neagră" date de utilizator, care definesc problema de rezolvat, natura candidaților parțiali și modul în care acestea sunt extinse în candidați compleți. Prin urmare, este un metaheuristic și nu un algoritm specific - deși, spre deosebire de multe alte meta-euristice, se garantează că se găsesc toate soluțiile la o problemă finită într-un timp limitat.

Descrierea metodei

Algoritmul backtracking enumeră un set de candidați parțiali care, în principiu, ar putea fi completate în diverse moduri pentru a da toate soluțiile posibile la problema dată. Finalizarea se face treptat, printr-o succesiune de pași de extensie a candidatului.

Conceptual, candidații parțiali sunt reprezentați ca noduri ale unei structuri de arbore, arborele potențial de căutare. Fiecare candidat parțial este părintele candidaților care diferă de acesta printr-o singură etapă de extindere; frunzele copacului sunt candidații parțiali care nu mai pot fi prelungiți.

Algoritmul backtracking parcurge acest arbore de căutare recursiv, de la rădăcină în jos, în profunzime ordine. La fiecare nod c, algoritmul verifică dacă c poate fi completat cu o soluție validă. Dacă nu se poate, întregul sub-arbore înrădăcinat la c este omis (tăiat). În caz contrar, algoritmul (1) verifică dacă c în sine este o soluție valabilă și, dacă este, îl raportează utilizatorului; și (2) enumeră recursiv toate sub-arborii din c. Cele două teste și copiii fiecărui nod sunt definiți prin proceduri date de utilizator.

Prin urmare, arborele de căutare efectiv traversat de algoritm este doar o parte a arborelui potențial. Costul total al algoritmului este numărul de noduri ale arborelui, efectiv, de mai multe ori costul obținerii și procesării fiecărui nod. Acest fapt trebuie luat în considerare atunci când alegeți arborele de căutare potențial și implementați testul de tăiere.

Observații:

- *Pornind de la strategiile clasice de parcurgere a spațiului de stări, algoritmii de tip backtracking, practic, enumeră un set de candidați parțiali, care după completarea definitivă, pot deveni soluții potențiale ale problemei inițiale.*
- *Exact ca strategiile de parcurgere în lățime/adâncime și backtracking-ul are la bază expandarea unui nod curent, iar determinarea soluției se face într-o manieră incrementală.*
- *Prin natura sa, backtracking-ul este recursiv, iar în arborele expandat top-down se aplică operații de tipul pruning (tăiere) dacă soluția parțială nu este validă.*

Cum se poate determina dacă o problemă poate fi rezolvată folosind Backtracking?

În general, fiecare problemă de satisfacție, care are constrângeri clare și bine definite asupra oricărei soluții obiective, care construiește treptat un candidat la soluție și abandonează un candidat („backtracks") imediat ce determină că candidatul nu poate fi completat în mod valabil, soluția, poate fi rezolvată prin Backtracking.

Cu toate acestea, majoritatea problemelor care sunt discutate pot fi rezolvate, folosind alți algoritmi cunoscuți, precum programare dinamică sau tehnica Greedy, în complexitatea timpului logaritmic, liniar, liniar-logaritmic, în ordinea mărimii de intrare.

Prin urmare, depășiți algoritmul de backtracking în toate privințele (deoarece algoritmii de backtracking sunt în general exponențiali atât în timp, cât și în spațiu). Cu toate acestea, mai rămân câteva probleme, care au doar algoritmi de backtracking care să le rezolve până acum ***[19]****.*

Exemple:

1. *Luați în considerare o situație în care aveți trei cutii în fața dvs. și doar una dintre ele are o monedă de aur în ea, dar nu știți care dintre ele. Deci, pentru a obține moneda, va trebui să deschideți toate cutiile una câte una. Mai întâi veți verifica prima cutie, dacă nu conține moneda, va trebui să o închideți și să verificați a doua cutie și așa mai departe până găsiți moneda. Aceasta este ceea ce numim backtracking, care se aplică pentru a rezolva toate sub-problemele una câte una pentru a ajunge la cea mai bună soluție posibilă.Considerați exemplul de mai jos pentru a înțelege mai formal abordarea Backtracking,*
2. *Dat fiind o instanță a unor probleme de calcul P și date D corespunzătoare fiecărei instanțe. Toate constrângerile care trebuie satisfăcute pentru a rezolva problema sunt reprezentate de C. Un algoritm de backtracking va funcționa astfel:*

Algoritmul începe să creeze o soluție, începând cu un set de soluții vide S, adică S = {}.

1. *Adăugați la S prima mișcare care rămâne în continuare (Toate mutările posibile sunt adăugate la S una câte una). Aceasta creează acum un nou sub-arbore în arborele de căutare al algoritmului.*
2. *Verificați dacă S + s îndeplinește fiecare dintre constrângerile din C.*
   - *Dacă da, atunci sub-arborele este „eligibil" pentru a adăuga mai mulți „copii".*
   - *În rest, întregul sub-arbore nu este inutil, deci recurgeți la pasul 1 folosind argumentul S.*
3. *În cazul „eligibilității" sub-arborelui nou format, recurgeți la pasul 1, folosind argumentul S + s.*

4. *Dacă se verifică S + s, se întoarce că este o soluție pentru toate datele D. Ieșirea și încheierea programului. Dacă nu, atunci întoarceți că nici o soluție nu este posibilă cu curentul s și, prin urmare, aruncați-l.*

Pseudocod

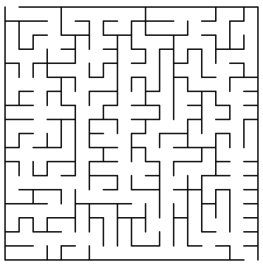
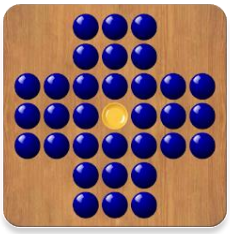
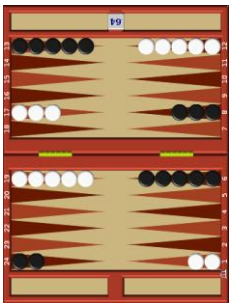
Pentru a aplica backtracking la o anumită clasă de probleme, trebuie să furnizați datele P pentru instanța particulară a problemei care urmează să fie rezolvată și șase parametri procedurali, root, resping, acceptați, întâi, următorul și ieșire. Aceste proceduri ar trebui să ia datele de instanță P ca parametru și ar trebui să facă următoarele:

- *root (P): returnează candidatul parțial la rădăcina arborelui de căutare.*
- *respinge (P,c): returnează adevărat numai dacă candidatul parțial c nu merită completat.*
- *acceptă (P, c): returnează adevărat dacă c este o soluție a lui P și fals în caz contrar.*
- *prima (P, c): generați prima extensie a candidatului c.*
- *următorul (P,s): generați următoarea extensie alternativă a unui candidat, după extensia s.*
- *ieșire (P, c): folosiți soluția c din P, după cum este cazul aplicației.*

Tipuri de probleme

Exemple în care backtracking poate fi utilizat pentru a rezolva puzzle-uri sau probleme includ:

- *Puzzle cum ar fi: problema celor opt regine, cuvinte încrucișate, aritmetică verbală, Sudoku și Peg Solitaire.*
- *Probleme de optimizare combinatorie, cum ar fi: analiza și problema rucsacului.*
- *Limbaje de programare logică precum: Icon, Planificator și Prolog, care utilizează backtracking-ul intern pentru a genera răspunsuri.*

Labirint	***Peg Solitaire***	***Șah***	***Backgammon***
			

4.2 Analiza complexității algoritmului metodei backtracking

Complexitatea temporală *este de* $O(B^d)$, *iar cea spațială* $O(d)$, *unde B este factor de ramificare (numărul mediu de stări posibil ulterioare în care nodul curent poate fi expandat) și d este adâncimea soluției.*

Cum determinăm complexitatea pe un algoritm backtracking recursiv?

Exemplul 1:
Se dă un număr N. Să se genereze toate permutările mulțimii formate din toate numerele de la 1 la N.

Notă:
- ✓ *Pentru n=3 obținem rezultatele din imaginea alăturată.*
- ✓ *Numărul total de permutații posibile sunt în total 6.*

Backtracking (algoritmul în cazul general)
Soluția va avea următoarele complexități:

A. *Complexitate temporală:* $T(n)=O(n*n!)=O(n!)$
- *Explicație: Complexitatea generării permutarilor, O(n!)O(n!), se înmulțește cu complexitatea copierii vectorilor soluție și domeniu, precum și a ștergerii elementelor din domeniu, O(n).*

B. *Complexitate spațială:* $S(n)=O(n^2)$
- *Explicație: Fiecare nivel de recursivitate are propria lui copie a soluției și a domeniului. Sunt n nivele de recursivitate, deci complexitatea spațială este* $O(n*n)= O(n^2)$.

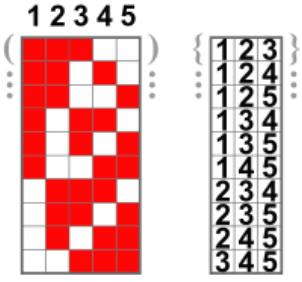
Backtracking (tăierea ramurilor nefolositoare)
Soluția va avea următoarele complexități:

A. *Complexitate temporală :* $T(n)=O(n*n!)=O(n!)$
- *Explicație: Complexitatea generării permutărilor, O(n!), se înmulțește cu complexitatea iterării prin domeniu, O(n).*

B. *Complexitatea spațială:* $S(n)=O(n)$
- *Explicație: Toate nivelele de recursivitate folosesc aceeași soluție și același domeniu.*

Observații:
- ❖ *Această soluție este optimă și are complexitatea temporală* $T(n)=O(n!)$. *Nu putem să obținem o soluție mai bună, întrucât trebuie să generăm n! permutări.*
- ❖ *De asemenea, este optimă și din punct de vedere spațial, întrucât trebuie să avem* $S(n)=O(n)$, *din cauza stocării permutării generate.*

Exemplul 2:
Se dau numerele N și K. Să se genereze toate combinările mulțimii formate din toate numerele de la 1 la N, luate câte K.



Notă:
- ✓ *Pentru n=5 și k=3 obținem rezultatele din imaginea alăturată.*
- ✓ *Numărul total de combinații posibile sunt în total 10.*

Backtracking (taierea ramurilor nefolositoare)
Soluția va avea următoarele complexități:

A. *Complexitate temporală:* $T(n)=O(Combinari(n,k))$

B. *Complexitate spațială:* $S(n)=O(n+k)=O(n)$
- *Explicație: Deoarece* $k<=n$, *deci* $O(n+k)=O(n)$ ***[20]***.

4.3 Forme specifice ale metodei backtracking

Ca orice algoritm în care sunt prezente instrucțiuni repetitive, algoritmul backtracking poate fi reprezentat într-o maniera recursivă sau iterativă (nerecursivă). Vom opta pentru varinata iterativă.

Generarea permutărilor. *Se citeşte un număr natural n. Să se genereze toate permutările mulţimii {1, 2, 3, ...,n}.Generarea permutărilor se va face ţinând cont că orice permutare va fi alcătuită din elemente distincte ale mulţimii A. Din acest motiv, la generarea unei permutări, vom urmări ca numerele să fie distincte. Metoda Backtracking se poate implementa în două moduri relativ asemănătoare.*

Notă:

- ✓ *În cazul variantei recursive rutina BackRec1 se definește ca o funcție recursivă cu parametrul k (nivelul curent din stivă). Urcarea în stivă se face prin autoapelul funcției BackRec1 cu o nouă valoare pentru k.*
- ✓ *Generarea potențialilor candidați la soluție se face printr-o instrucțiune for care parcurge valorile posibile de la limita inferioară la limita superioară.*

Varianta recursivă

```
#include <iostream>
using namespace std;
int x1[100], n1, nrsol1=0;
void Afisare(){
    int i;
    for(i=1;i<=n1;i++)
        cout<<x1[i]<<" ";
        cout<<endl;
        nrsol1++;
}
int Valid(int k){
    int i;
    for(i=1;i<=k-1;i++)
        if (x1[k]==x1[i]) return 0;
        return 1;
}
void BackRec1(int k){
    int i;
    for(i=1;i<=n1;i++){
        x1[k]=i;
        if (Valid(k))
        if (k==n1) Afisare();
            else BackRec1(k+1);
     }
}
int main(){
    cout<<"Introduceti valoarea lui n:
";
    cin>>n1;
    cout<<"Permutarile primelor "<<n1<<"
numere naturale (n<100)"<<endl;
    BackRec1(1);
    cout<<"Numar solutii: ";
    cout<<nrsol1<<endl;
    return 0;
}
```

Varianta iterativă

```
#include<iostream>
using namespace std;
typedef int stiva[100];
int n,k,ev,as,nr=0;
stiva st;
void init(){
    st[k]=0;
}
int succesor(){
    if (st[k]<n){
       st[k]=st[k]+1; return 1;
    }
    else return 0;
}
int valid(){
    for(int i=1;i<k;i++)
        if(st[k]==st[i])
            return 0;
    return 1;
 }
int solutie(){
     return k==n;
}
void tipar(){
     for(int i=1;i<=n;i++)
        cout<<st[i]<<" "; cout<<endl;
        nr++;
}
void backtrack(){
    k=1;
    init();
    while(k>0){
        as=1; ev=0;
        while(as && !ev){
            as=succesor();
            if(as) ev=valid();
        }
        if(as)
            if(solutie()) tipar();
                else {
                    k++; init();
                }
        else k--;
    }
}
int main(){
    cout<<" Introduceti valoarea lui n:
```

	"; cin>>n; cout<<"Permutarile primelor "<<n<<" numere naturale (n<100)"<<endl; backtrack(); cout <<"Numar solutii: "<<nr<<endl; }
Introduceti valoarea lui n: 3 Permutarile primelor 3 numere naturale (n<100) 1 2 3 1 3 2 2 1 3 2 3 1 3 1 2 3 2 1 Numar solutii: 6	Introduceti valoarea lui n: 3 Permutarile primelor 3 numere naturale (n<100) 1 2 3 1 3 2 2 1 3 2 3 1 3 1 2 3 2 1 Numar solutii: 6

Observații:

- *Metoda Backtracking are ca rezultat obținerea tuturor soluțiilor problemei. În cazul în care se cere o singură soluție se poate forța oprirea, atunci când a fost găsită.*
- *Problemele rezolvate prin această metodă necesită un timp îndelungat. Din acest motiv, este bine să utilizăm metoda numai atunci când nu avem la dispoziție un alt algoritm mai eficient. Cu toate că există probleme pentru care nu se pot elabora alți algoritmi mai eficienți, metoda backtracking trebuie aplicată numai în ultimă instanță* ***[21]****.*

Probleme de generare. Oportunitatea utilizării metodei backtracking.

Problemele care se rezolvă prin metoda backtracking pot fi împărțite în mai multe grupuri de probleme cu rezolvări asemănătoare, în funcție de modificările pe care le vom face în algoritm.

Principalele grupuri de probleme sunt:

- *probleme în care vectorul soluție are lungime fixă și fiecare element apare o singură dată în soluție;*
- *probleme în care vectorul soluție are lungime variabilă și fiecare element poate să apară de mai multe ori în soluție;*
- *probleme în plan, atunci când spațiul în care ne deplasăm este un tablou bidimensional.*

Vom prezenta în cele ce urmează câteva probleme care fac parte din primul grup. Cele mai cunoscute sunt:

- *generarea permutărilor unei mulțimi;*
- *generarea aranjamentelor unei mulțimi;*
- *generarea submulțimilor unei mulțimi;*
- *generarea submulțimilor cu m elemente ale unei mulțimi (combinări);*
- *generarea produsului cartezian a n mulțimi;*
- *generarea tuturor secvențelor de n (par) paranteze care se închid corect;*
- *colorarea țărilor de pe o hartă astfel încât oricare două țări vecine să aibă culori diferite, etc.*

Notă:

- ✓ *Toate problemele din acest grup au particularitatea că soluția se obține atunci când vectorul soluție ajunge să conțină un anumit număr de elemente.*

4.4 Implementarea metodei backtracking la rezolvarea problemelor elementare

Problema 1: Aranjamente

Condiția problemei	*Exemplu*
Formula generală: $A_n^k = n(n-1)\ldots(k-n+1) = \frac{n!}{(n-k)!}$ ***Formula generală recursivă:*** $A_n^k = n(n-1)\cdot A_n^{k-1},\ pentru\ \forall k \in [1,n].$	*Pentru n = 3 și m=2 programul va afișa:* *1 2* *1 3* *2 1* *2 3* *3 1* *3 2*

Implementare C++

```
#include<iostream>
using namespace std;
int st[20],n,k;
void init(){
       int i;
       cout<<"Introduceti numerele naturale n si k: \n"; cin>>n>>k;
       for(i=1;i<=n;i++)
              st[i]=0;
}
void tipar(int p){
       int j;
       for(j=1;j<=p;j++)
         cout<<"\t"<<st[j]<<" "; cout<<endl;
}
int valid(int p){
       int i,ok;
       ok=1;
       for(i=1;i<p;i++)
         if(st[p]==st[i])ok=0;
       return ok;
}
int solutie(int p){
       return (p==k);
}
void bkt(int p){
       int val;
       for (val=1;val<=n;val++){
              st[p]=val;
              if (valid(p))
              if(solutie(p))tipar(p);
              else bkt(p+1);
       }
}
int main(){
       init();
       cout<<"Aranjamente din "<<n<<" elemente luate cate "<<k<<endl;
       bkt(1);
}
```

Rezultatele execuției:

```
Introduceti numerele naturale n si k:
2 2
Aranjamente din 2 elemente luate cate 2
        1       2
        2       1
```

Problema 2: Combinări

Condiția problemei	*Exemplu*
Formula generală: $C_n^k = \frac{A_n^k}{P_k} = \frac{n!}{k!\cdot(n-k)!}$, *unde* $p \in [0,n]$. ***Formula generală recursivă:*** $C_n^k = C_{n-1}^k + C_{n-1}^{k-1}$, *pentru* $\forall k \in [1,n]$.	*Pentru n = 3 și m=2 programul va afișa:* *1 2* *1 3* *2 3*

Implementare C++

```
#include<iostream>
using namespace std;
int st[20],n,k;
void init(){
     cout<<"Introduceti numerele naturale n si k: \n"; cin>>n>>k;
     for(int i=1;i<=n;i++) st[i]=0;
}
void tipar(int p){
    for(int j=1;j<=p;j++)
        cout<<"\t"<<st[j]<<" "; cout<<endl;
}
int valid(int p){
    for(int i=1;i<=p-1;i++)
        if(st[i]>=st[p]) return 0;
    return 1;
}
int solutie(int p){
       return (p==k);
}
void bkt(int p){
    for(int val=1;val<=n;val++){
        st[p]=val;
        if(valid(p)==1)
            if (solutie(p)) tipar(p);
            else bkt(p+1);
    }
}
int main(){
    init();
    cout<<"Combinari din "<<n<<" elemente luate cate "<<k<<endl;
    bkt(1);
}
```

Rezultatele execuției:

```
Introduceti numerele naturale n si k:
6 4
Combinari din 6 elemente luate cate 4
        1       2       3       4
        1       2       3       5
        1       2       3       6
        1       2       4       5
        1       2       4       6
        1       2       5       6
        1       3       4       5
        1       3       4       6
        1       3       5       6
        1       4       5       6
        2       3       4       5
        2       3       4       6
        2       3       5       6
        2       4       5       6
        3       4       5       6
```

Problema 3: Partițiile unui număr natural

Condiția problemei	*Exemplu*
Fie n>0, natural. Să se scrie un program care să afișeze toate partițiile unui număr natural n. Numim partiție a unui număr natural nenul n o mulțime de numere naturale nenule {p1, p2, ..., pk} care îndeplinesc condiția p1+p2+ ...+pk = n.	*Pentru n = 4 programul va afișa:* *4 = 1+1+1+1* *4 = 1+1+2* *4 = 1+3* *4 = 2+2* *4 = 4*

Implementare C++

```
#include<iostream>
using namespace std;
int n, ns,sol[20];
void afis(int l){
    ns++;
    cout<<"Solutia "<< ns<<" : \t";
    for(int i=1;i<=l;i++) cout<<sol[i]<<" ";
    cout<<endl;
}
void back(int i, int sp){
    int j;
    if (sp==n) afis(i-1);
    else
        for(j=1;j<=n-sp;j++)
        if (j>=sol[i-1]){
            sol[i]=j;
            back(i+1, sp+j);
             }
}
int main(){
  cout<<"Introdu un numar natural: ";cin>>n;
  ns=0;
  back(1,0);
  cout<<endl;
  cout<<"TOTAL "<<ns<<" SOLUTII";
}
```

Rezultatele execuției:

```
Introduceti un numar natural: 6
Solutia 1 :     1 1 1 1 1 1
Solutia 2 :     1 1 1 1 2
Solutia 3 :     1 1 1 3
Solutia 4 :     1 1 2 2
Solutia 5 :     1 1 4
Solutia 6 :     1 2 3
Solutia 7 :     1 5
Solutia 8 :     2 2 2
Solutia 9 :     2 4
Solutia 10 :    3 3
Solutia 11 :    6

TOTAL 11 SOLUTII
```

Problema 4: Combinații a k paranteze rotunde care se închid corect

Condiția problemei	*Exemplu*
Se citește de la tastatură un număr natural k par, k<50. Să se genereze și să se afișeze toate combinațiile de k paranteze rotunde care se închid corect.	*Pentru k = 6 programul va afișa:* *1 : ((( )))* *2 : (( )( ))* *3 : (( ))( )* *4 : ( )(( ))* *5 : ( )( )( )*

Implementare C++

```
#include<iostream>
using namespace std;
int x[10],n,ns;
void scriesol(){
    ns++;
    cout<<"Solutia "<< ns<<" : \t";
    for(int j=1;j<=n;j++)
    if(x[j]==1) cout<<")";
    else cout<<"(";
    cout<<endl;
}
int cond(int k){
    int i,pi=0,pd=0;
    for(i=1;i<=k;i++)
    if(x[i]==0) pd++;
    else pi++;
    return (pd<=n/2 && pi<=pd);
}
void back(int k){
    int i;
    for(i=0;i<=1;i++){
        x[k]=i;
        if (cond(k))
           if (k==n) scriesol();
           else back(k+1);
   }
}
int main(){
    cout<<"Introduceti un numar natural: ";cin>>n;
    back(1);
    cout<<endl;
    cout<<"TOTAL "<<ns<<" SOLUTII";
}
```

Rezultatele execuției:

```
Introduceti un numar natural: 8
Solutia 1 :     (((())))
Solutia 2 :     (((()())))
Solutia 3 :     (((())()))
Solutia 4 :     (((()))())
Solutia 5 :     ((()(())))
Solutia 6 :     ((()()()))
Solutia 7 :     ((()())())
Solutia 8 :     ((())(()))
Solutia 9 :     ((())()())
Solutia 10 :    ()((()))
Solutia 11 :    ()(()())
Solutia 12 :    ()(())()
Solutia 13 :    ()()(())
Solutia 14 :    ()()()()

TOTAL 14 SOLUTII
```

Problema 5: Descompunerea lui k – natural ca sumă de numere naturale consecutive

Condiția problemei	*Exemplu*
Se citește un număr natural k. Să se afișeze toate modalitățile de a-l descompune ca sumă de numere naturale consecutive. Dacă acest lucru nu este posibil, se va afișa mesajul "Imposibil".	*Pentru k=15 programul va afișa:* *1+2+3+4+5* *4+5+6* *7+8* *Pentru k=8 programul va afișa:* *"Imposibil".*

Implementare C++

```
#include<iostream>
using namespace std;
int n, ns,sol[20];
void afis(int l){
    int i;
    ns++;
    for(i=1;i<=l;i++)
    cout<<sol[i]<<" ";
    cout<<endl;
}
void back(int i, int sp){
    int j;
    if (sp==n && i>2) afis(i-1);
    else
        for(j=sol[i-1]+1;j<=n-sp;j++)
        if (j==sol[i-1]+1 || i==1){
            sol[i]=j;
            back(i+1, sp+j);
             }
}
int main(){
    cout<<"Introduceti un numar natural: ";
    cin>>n;
    ns=0;
    back(1,0);
    if (ns==0) cout<<"Imposibil"; cout<<endl;
    cout<<"TOTAL "<<ns<<" SOLUTII";
}
```

Rezultatele execuției:

```
Introduceti un numar natural: 333
10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27
33 34 35 36 37 38 39 40 41
53 54 55 56 57 58
110 111 112
166 167

TOTAL 5 SOLUTII
```

Problema 6: Rațe și găini

Condiția problemei	*Exemplu*
În curtea lui moș Zorel se află n găini și m rate. Să se aranjeze în toate modurile cele n găini și m rațe astfel încat nici o găină să nu fie așezată între două rațe.	*Pentru n=2 și m=2 programul va afișa:* *G G R R* *G R R G* *R G G R* *R R G G*

Implementare C++

```
#include<iostream>
using namespace std;
int x[100],pus[100],n,m,nr=0;
void aranjare(){
    for(int i=1;i<=n+m;i++)
        if(x[i]==1) cout<<"G ";
        else cout<<"R ";
    cout<<endl; nr++;
}
int conditie(int k){
    int c=0,p=0,i;
    for(i=1;i<=k;i++)
    if(x[i]==0) c++;
    else p++;
    if(p>n || c>m) return 0;
    if(k>=3) if(x[k-2]==0 && x[k-1]==1 && x[k]==0) return 0;
    return 1;
}
void btk(int k){
    for(int i=1;i>=0;i--){
        x[k]=i;
        if(conditie(k))
        if(k==n+m) aranjare();
        else btk(k+1);
   }
}
int main(){
    cout<<"Introduceti doua numere naturale: ";cin>>n>>m;cout<<endl;
    cout<<"Pentru "<<n<<" gaini si "<<m<<" rate avem aranjarile:\n";
    btk(1);
    cout<<endl; cout<<"TOTAL "<<nr<<" SOLUTII";
    return 0;
}
```

Rezultatele execuției:

```
Introduceti doua numere naturale: 3 4
Pentru 3 gaini si 4 rate avem aranjarile:
G G G R R R R
G G R R R R G
G R G G R R R
G R R G G R R
G R R R G G R
G R R R R G G
R G G G R R R
R G G R R R G
R R G G G R R
R R G G R R G
R R R G G G R
R R R G G R G
R R R R G G G

TOTAL 13 SOLUTII
```

Problema 7: Anagramele unui cuvânt

Condiția problemei	*Exemplu*
Se citește un cuvânt format doar din litere mici distincte. Să se genereze anagramele acestui cuvânt.	*Pentru s=”abac” programul va afișa:* *abac abca aabc aacb acba acab baac baca baac baca bcaa bcaa aabc aacb abac abca acab acba caba caab cbaa cbaa caab caba* *Avem în total 24 de soluții*

Implementare C++

```
#include<iostream>
#include<string.h>
using namespace std;
int x[100],pus[100],n,nr=0;
char s[100];
void afiseaza(){
    for(int i=1;i<=n;i++)
    cout<<s[x[i]-1];
    cout<<endl;
    nr++;
}
void btk(int k){
    for(int i=1;i<=n;i++)
    if(!pus[i]){
        x[k]=i; pus[i]=1;
        if(k==n) afiseaza();
        else btk(k+1);
        pus[i]=0;
    }
}
int main(){
    cout<<"Introdu un cuvant: ";cin>>s;
    n=strlen(s);
    cout<<endl;
    cout<<"Anagramele cuvantului "<<s<<" sunt: \n";
    btk(1);
    cout<<endl;
    cout<<"TOTAL "<<nr<<" SOLUTII";
}
```

Rezultatele execuției:

```
Introduceti un cuvant: abc

Anagramele cuvantului abc sunt:
abc
acb
bac
bca
cab
cba

TOTAL 6 SOLUTII
```

Problema 8: Construcția segmentelor

Condiția problemei	*Exemplu*
Se dau în plan n puncte albe și n puncte negre prin coordonatele lor întregi. Fiecare punct alb trebuie unit cu un singur punct negru pentru a forma n segmente care să nu se intersecteze, fiecare segment având o extremitate un punct alb și cealaltă - un punct negru.	*Pentru datele de intrare:* *2* *1 2* *1 3* *2 1* *2 3* *Se obțin segmentele:* *(1,2)-(2,1)* *(1,3)-(2,3)*

Implementare C++

```
#include <iostream>
using namespace std;
struct punct{
    int x,y;
};
int x[100], p[100], n,s; punct a[100], b[100];
void citire(punct a[],punct b[],int &n){
    cout<<"Introduceti un numar natural: ";cin>>n;
    for(int i=1;i<=n;i++) cin>>a[i].x>>a[i].y;
    for(int i=1;i<=n;i++) cin>>b[i].x>>b[i].y;
}
void afis(int n){
    for(int i=1;i<=n;i++)    cout<<"("<<a[i].x<<","<<a[i].y<<")-
("<<b[x[i]].x<<","<<b[x[i]].y<<")"<<endl;
    cout<<endl;
}
//pozitia lui c fata de dreapta a-b
int dreapta(punct a, punct b, punct c){
    return (c.x-a.x)*(b.y-a.y)-(c.y-a.y)*(b.x-a.x);
}
int corect(int k){
    for(int i=1;i<k;i++)
    if(dreapta(a[i],b[x[i]],a[k])*dreapta(a[i],b[x[i]],b[x[k]])<0 &&
dreapta(a[k],b[x[k]],a[i])*dreapta(a[k],b[x[k]],b[x[i]])<0) return 0;
    return 1;
}
void back(int x[],int k){
    int i;
    for(i=1;i<=n;i++)
    if(!p[i]){
        x[k]=i; p[i]=1;
        if(corect(k))
            if(k==n) afis(k);
            else back(x,k+1);  p[i]=0;
    }
}
int main(){
    citire(a,b,n);
    cout<<"Se obtin segmentele:"<<endl;
    back(x,1); return 0;
}
```

Rezultatele execuției:

```
Introduceti un numar natural: 2
1 2
1 3
2 3
2 1
Se obtin segmentele:
(1,2)-(2,1)
(1,3)-(2,3)
```

Problema 9: Drapele din trei culori diferite

Condiția problemei	*Exemplu*
Se citește un număr natural n și apoi n culori distincte date ca șiruri de caractere. Afișați toate drapelele care se pot forma cu câte 3 culori diferite.	*Pentru datele de intrare:* *3* *GREEN* *BLACK* *BROWN* *Se obțin drapelele:* *GREEN BLACK BROWN* *GREEN BROWN BLACK* *BLACK GREEN BROWN* *BLACK BROWN GREEN* *BROWN GREEN BLACK* *BROWN BLACK GREEN*

Implementare C++

```
#include<iostream>
using namespace std;
int x[100],pus[100],n,nr=0;
char s[30][30];
void afisare(){
    for(int i=1;i<=3;i++) cout<<"\t"<<s[x[i]];
    cout<<endl;
    nr++;
}
void btk(int k){
    for(int i=1;i<=n;i++)
    if(!pus[i]){
        x[k]=i; pus[i]=1;
        if(k==3) afisare();
        else btk(k+1);
        pus[i]=0;
    }
}
int main(){
    cout<<"Introduceti un numar natural: ";cin>>n;
    cout<<"Introduceti cele "<<n<<" culori:\n";
    for(int i=1;i<=n;i++) cin>>s[i];
    cout<<"Drapelele formate din "<<n<<" culori aranjate cate 3 sunt: \n";
    btk(1);
    cout<<endl;
    cout<<"TOTAL "<<nr<<" SOLUTII";
}
```

Rezultatele execuției:

```
Introduceti un numar natural: 3
Introduceti cele 3 culori:
GREEN
BLACK
BROWN
Drapelele formate din 3 culori aranjate cate 3 sunt:
        GREEN   BLACK   BROWN
        GREEN   BROWN   BLACK
        BLACK   GREEN   BROWN
        BLACK   BROWN   GREEN
        BROWN   GREEN   BLACK
        BROWN   BLACK   GREEN

TOTAL 6 SOLUTII
```

Problema 10: Pătrat magic de numere

Condiția problemei	*Exemplu*
*Un pătrat cu latura n este numit magic dacă este format din numerele de la 1 la n*n și suma pe fiecare linie, pe fiecare coloană, precum și pe cele 2 diagonale este constantă.*	*Pentru datele de intrare:* *3* *Se obține următorul pătrat magic :* *2 7 6* *9 5 1* *4 3 8*

Implementare C++

```
#include<iostream>
using namespace std;
int n,p[100],a[10][10],s,gata;
void afis(){
    for(int i=1;i<=n;i++){
        for(int j=1;j<=n;j++) cout<<"\t"<<a[i][j]; cout<<endl;
    }
    cout<<endl;gata=1;
}
int sumap(int i,int j){
    int s=0; for(int k=1;k<=j;k++) s=s+a[i][k]; return s;
}
int sumal(int i){
    int s=0; for(int j=1;j<=n;j++) s=s+a[i][j]; return s;
}
int sumac(int j){
    int s=0; for(int i=1;i<=n;i++) s=s+a[i][j]; return s;
}
int sumad2(){
    int s=0; for(int i=1;i<=n;i++) s=s+a[i][n+1-i]; return s;
}
int sumad1(){
    int s=0; for(int i=1;i<=n;i++) s=s+a[i][i]; return s;
}
int bun(int i, int j){
    if(sumap(i,j)>s) return 0; if(j==n) if(sumal(i)!=s) return 0;
    if(i==n) if(sumac(j)!=s) return 0;
    if(i==n && j==1) if(sumad2()!=s) return 0;
    if(i==n && j==n) if(sumad1()!=s) return 0; return 1;
}
void back(int i, int j){
    if(!gata)
    for(int v=1;v<=n*n;v++)
    if(!p[v]){
        a[i][j]=v; p[v]=1;
        if(bun(i,j))
            if(i==n&&j==n) afis();
            else if(j<n) back(i,j+1); else back(i+1,1); p[v]=0;
    }
}
int main(){
    cout<<"Introduceti un numar natural: ";cin>>n;
    s=n*(n*n+1)/2; gata=0;
    cout<<"Patratul magic de dimensiunea "<<n<<" este: \n";
    back(1,1); return 0;
}
```

Rezultatele execuției:

```
Introduceti un numar natural: 4
Patratul magic de dimensiunea 4 este:
        1       2       15      16
        12      14      3       5
        13      7       10      4
        8       11      6       9
```

SARCINI PENTRU EXERSARE

SETUL DE PROBLEME NR. 1

1. *Să se afişeze toate soluţiile ecuaţiei în mulţimea numerelor naturale: 3x+y+4xz=100.*
2. *Folosind metoda backtracking să se genereze în ordine lexicografică cuvintele de câte patru litere din mulțimea A={a,b,c,d,e}, cuvinte care nu conțin două vocale alăturate. Primele 8 cuvinte generate sunt, în ordinea: abab, abac, abad, abba, abbb, abbc, abbd, abbe.*
3. *Să se determine toate cuvintele ce conţin numai literele a,b,c de lungime 10 care conţin exact 3 simboluri 'a'; 4 simboluri 'b' şi 3 simboluri 'c'.*
4. *Să se determine toate numerele în baza 8 de lungime 10 care conţin cel mult 5 cifre de 7 şi exact 3 cifre de 0.*
5. *Să se afişeze toate posibilităţile de colorare a unui graf neorientat cu n vârfuri, folosind m culori astfel încât două noduri vecine să aibă culori diferite.*
6. *N copii se așează în șir indian. Se cunosc numele celor n copii. Să se găsească toate posibilitățile de aranjare în șir.*
7. *N copii se așează în cerc. Se cunosc numele celor n copii. Să se găsească toate posibilitățile de rearanjare în cerc.*
8. *Se cer toate soluțiile de așezare în linie a m câini și n pisici astfel încât să nu existe o pisică între doi câini.*
9. *Se citește un număr. Să se genereze toate numerele având aceleași cifre ca el. Care este cel mai mare?*
10. *Să se genereze toate:*
    - *A. triunghiurile de perimetru n, afișați lungimile celor 3 laturi;*
    - *B. pătratele de perimetru n, afișați lungimile celor 4 laturi;*
    - *C. pentagoanele de perimetru n, afișați lungimile celor 5 laturi.*
11. *Să se genereze toate numerele de lungime n formate:*
    - *A. doar cu cifre pare;*
    - *B. doar cu cifre impare;*
    - *C. doar cu cifre prime.*
12. *Scrieți un program care să afișeze toate numerele de n (n<=10) cifre, formate numai din cifre distincte și care sunt divizibile cu 5.*
13. *Fiind dată o mulțime de n cuburi, fiecare cub fiind caracterizat de lungimea laturii și culoarea sa. Să se scrie un program care să genereze toate turnurile care se pot forma cu p cuburi astfel încât două cuburi vecine să nu aibă aceeași culoare, iar deasupra unui cub să nu se poată aseza un cub cu latura mai mare.*
14. *O persoană a uitat numărul de telefon al unui prieten. Știe doar că numărul are 6 cifre, începe cu 4 și conține 3 zerouri dintre care două sunt alăturate. Afișați toate variantele pe care trebuie să le încerce pentru a vorbi cu prietenul său.*
15. *O persoană a uitat codul PIN de la telefon. Știe doar că codul avea 4 sau 6 cifre, începe cu 9 și conține 2 zerouri care nu sunt alăturate. Afișați toate variantele pe care trebuie să le încerce pentru a debloca telefonul său* ***[27]***.

SETUL DE PROBLEME NR. 2

Problema 1	***Date de intrare***	***Date de ieșire***
Se dă un număr natural s cu cel mult 9 cifre. Afişaţi în ordine lexicografică, toate modalităţile de a-l scrie pe s ca produs de divizori proprii distincți ai lui s.	`30`	`2 3 5` `2 15` `3 10` `5 6`

Problema 2	***Date de intrare***	***Date de ieșire***
Se citește un cuvânt s (cu cel mult 10 caractere litere mici distincte) și un număr natural n (n<=10). Să se genereze și să se afișeze toate cuvintele care se pot obține din s eliminând exact n litere.	`cosmin` `4`	`co cs cm ci cn` `os om oi on` `...` `in`

Problema 3	***Date de intrare***	***Date de ieșire***
Se citește un număr natural n, unde n<10 și apoi o mulțime A cu n elemente naturale ordonate crescător. Afișați în ordine lexicografică toate permutările mulțimii A în care elementele pare sunt puncte fixe.	`5` `1 4 6 7 9`	`Se vor genera permutările în care 4 și 6 nu își modifică poziția, veți obține 6 soluți.`

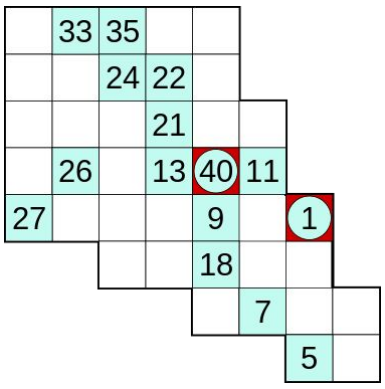
Problema 4	***Date de intrare***	***Date de ieșire***
Se citesc n numere naturale. Determinați o aranjare a acestor numere sub forma unui cerc, astfel încât suma produselor de câte două numere alăturate să fie maximă.	`6` `1 8 3 2 5 4`	`1 2 4 8 5 3`

Problema 5	***Date de intrare***	***Date de ieșire***
Fie n puncte în plan prin coordonatele lor. Găsiți cel mai mare pătrat care se poate forma cu vârfurile în 4 dintre punctele citite.Afișați coordonatele vârfurilor pătratului.	`9` `1 3` `0 0` `1 1` `1 0` `0 1` `3 5` `5 1` `5 6` `0 6`	`0 1` `5 1` `5 6` `0 6`

Problema 6	***Date de intrare***	***Date de ieșire***
Fie n puncte în plan prin coordonatele lor. Găsiți cel mai mic pătrat care se poate forma cu vârfurile în 4 dintre punctele citite.Afișați coordonatele vârfurilor pătratului.	`9` `1 3` `0 0` `1 1` `1 0` `0 1` `3 5` `5 1` `5 6` `0 6`	`0 0` `0 1` `1 1` `1 0`

Problema 7	***Date de intrare***	***Date de ieșire***
Se dau următoarele 6 culori: alb, galben, roșu, verde, orange și negru. Costruiți toate drapelele formate din 3 culori care îndeplinesc următoarele condiții: • *orice drapel trebuie să conțină culoarea verde sau culoarea galben la mijloc;* • *culorile din fiecare drapel trebuie să fie distincte.*	`6` `a` `g` `r` `v` `o` `n`	`a g r` `a g v` `a g o` `a g n` `a v g` `a v r` `a v o` `a v n` `...` `n v o`

*Problema 8**



Hidato (ebraică: חידאתו, provenind din cuvântul ebraic Hida = Enigmă), este un joc de puzzle logic inventat de Dr. Gyora M. Benedek, matematician israelian. Scopul Hidato este să umple grila cu numere consecutive care se conectează orizontal, vertical sau în diagonală. Numele Hidato este o marcă înregistrată a Doo-Bee Toys and Games LTD, o companie co-fondată de Benebek însuși. Unii editori folosesc diferite denumiri pentru acest puzzle, cum ar fi Numărul șarpe, Snakepit (ambele jucând pe similitudinea jocului în concept cu șarpele jocului video), Jadium sau Numbrix.

Vă propun ca să încercați să rezolvați această problemă, deși este foarte asemănătoare cu Sudoku, Hidato are ceva specific. Pentru mai multe detalii despre rezolvarea acestor tipuri de puzzle puteți cerceta unele resurse gratuite disponibile în internet.

4.5 Implementarea metodei backtracking la rezolvarea problemelor complexe

Problema 1: Jocul "FAZAN"

Condiția problemei	*Exemplu*
Se citesc două numerele naturale n și m (n<=15, m<=n), apoi n cuvinte distincte cu cel mult 10 litere fiecare. Să se afișeze toate secvențele a câte m cuvinte dintre cele citite care să respecte condițiile jocului "Fazan".	*Pentru n = 8 și m=3 și cele 8 cuvinte:paul, alina, asfalt, nas, ultim, imagine, nasture, real, programul va afișa:* `paul ultim imagine` `alina nas asfalt` `alina nasture real` `nasture real alina` `real alina nas` `real alina nasture`

Implementare C++

```
#include<iostream>
#include<string.h>
using namespace std;
char C[16][11]; int X[16],P[16],n,m;
void afisare(){
    for(int i=1;i<=n;i++)
        cout<<"\t"<<C[X[i]]; cout<<endl;
}
int fazan(char x[], char y[]){
    if(x[strlen(x)-2]==y[0] && x[strlen(x)-1]==y[1]) return 1;
    return 0;
}
void back(int k){
    for(int i=1;i<=n;i++)
        if(!P[i]){
            X[k]=i; P[i]=1;
            if(k==1 || fazan(C[X[k-1]],C[X[k]]))
                if(k==m) afisare();
                else back(k+1);
                P[i]=0;
        }
}
int main(){
    cout<<"Introduceti doua numere naturale: ";
    cin>>n>>m;
    for(int i=1;i<=n;i++) cin>>C[i];
    cout<<"Modalitatile de aranjare sunt: "<<endl;
    back(1); return 0;
}
```

Rezultatele execuției:

```
Introduceti doua numere naturale: 8 3
paul alina asfalt nas ultim imagine nasture real
Modalitatile de aranjare sunt:
        paul    ultim   imagine
        alina   nas     asfalt
        alina   nasture real
        nasture real    alina
        real    alina   nas
        real    alina   nasture
```

Problema 2: Aranjare soldați

Condiția problemei	*Exemplu*
*Se dă un număr natural n (n<=5) și un număr 2*n numere naturale cu cel mult 3 cifre, fiecare reprezentând înălțimile în cm a 2*n soldați. Să se aranjeze soldații pe două rânduri a câte n soldați fiecare astfel încât fiecare soldat în afară de primul de pe rând să aibă în stâga un soldat mai înalt decât el și fiecare soldat de pe rândul 2 să aibă în față un soldat mai înalt decât el.*	*Pentru n = 2 și înălțimile a 4 soldați: 153 155 157 159, programul va afișa:* **159 157** **155 153** *...modalitatea 1* **159 155** **157 153** *...modalitatea 2*

Implementare C++

```
#include <iostream>
using namespace std;
int X[15],P[15],A[15],n;
void afisare(){
    for(int i=1;i<=n/2;i++) //randul 1
        cout<<"\t"<<A[X[i]]; cout<<endl;
    for(int i=n/2+1;i<=n;i++) //randul 2
        cout<<"\t"<<A[X[i]]; cout<<endl<<endl;
}
int verif(int k){
    if(k!=1 && k!=n/2+1)
        if(A[X[k]]>=A[X[k-1]]) return 0;//stanga
    if(k>n/2)
        if(A[X[k]]>=A[X[k-n/2]]) return 0;//fata
    return 1;
}
void back(int k){
    for(int i=1;i<=n;i++)
        if(P[i]==0){
            X[k]=i; P[i]=1;
            if(verif(k))
                if(k==n) afisare();
                else back(k+1); P[i]=0;
        }
}
int main(){
    cout<<"Introdu un numar natural n:"; cin>>n; n=n*2;
    for(int i=1;i<=n;i++) cin>>A[i];
    for(int i=1;i<n;i++)
    for(int j=i+1;j<=n;j++)
        if(A[i]<A[j]){
            int aux=A[i]; A[i]=A[j]; A[j]=aux;
        }
    cout<<"Modalitatile de aranjare in doua randuri cate 3 soldati sunt: "<<endl;
    back(1); return 0;
}
```

Rezultatele execuției:

```
Introdu un numar natural n:2
123 125 127 129
Modalitatile de aranjare in doua randuri cate 3 soldati sunt:
        129     127
        125     123
        129     125
        127     123
```

Problema 3: Suma de bani

Condiția problemei	*Exemplu*
Se citește un număr natural n și apoi n numere naturale ordonate strict crescător, reprezentând valorile a n bancnote. Se citește apoi o sumă de bani s și se cere să se plătească în toate modurile posibile suma s cu bancnote de valorile precizate. Se presupune că avem la dispoziție oricate bancnote de fiecare valoare.	*Pentru n = 2 și s=20 și cele 2 tipuri de bancnote: 1 și 5, programul va afișa:* 4*5 5*1 3*5 10*1 2*5 15*1 1*5 20*1

Implementare C++

```
#include <iostream>
using namespace std;
int x[100], a[100], n,s;
void citire(){
    cout<<"Introduceti un numar natural n= "; cin>>n;
    cout<<" Introduceti valorile celor "<<n<<" bancnote: ";
    for(int i=1;i<=n;i++) cin>>a[i];
    cout<<" Introduceti un numar natural s= "; cin>>s;
}
void afis(){
    for(int i=1;i<=n;i++)
        if(x[i]!=0) cout<<"\t"<<x[i]<<"*"<<a[i]; cout<<endl;
}
void back(int k,int sp){
    for(int i=0;i<=(s-sp)/a[k];i++){
        x[k]=i; sp=sp+x[k]*a[k];
        if(sp<=s && k<=n)
        if(k==n && sp==s) afis();
        else if(k<n) back(k+1,sp);
        sp=sp-x[k]*a[k];
   }
}
int main(){
    citire();
    cout<<"Metodele de plata a celor "<<s<<" bancnote sunt: \n";
    back(1,0); return 0;
}
```

Rezultatele execuției:

```
Introduceti un numar natural n= 2
Introduceti valorile celor 2 bancnote: 1 5
Introduceti un numar natural s= 50

Metodele de plata a celor 50 bancnote sunt:
        10*5
        5*1     9*5
        10*1    8*5
        15*1    7*5
        20*1    6*5
        25*1    5*5
        30*1    4*5
        35*1    3*5
        40*1    2*5
        45*1    1*5
        50*1
```

Problema 4: Submulțime YZ

Condiția problemei	*Exemplu*
Se citesc numerele naturale x, y, z (x<=20, y<=x, z<=1000) și apoi n numere naturale distincte cu cel mult 5 cifre fiecare, reprezentând elementele unei mulțimi A. Numim submulțime "yz" o submulțime cu y elemente a mulțimii A care să aibă cmmdc al elementor cel puțin egal cu z. Să se afișeze toate submulțimile " yz " ale mulțimii A.	*Pentru x=7 și y=3 și z=5 și mulțimea de elemente: 3 6 9 15 20 24 30, programul va afișa:* `6 24 30` `15 20 30` *Ambele submulțimi au câte 3 elemente și cmmdc al elementelor cel puțin egal cu 5 (prima are 6, iar a doua 5).*

Implementare C++

```
#include <iostream>
using namespace std;
int X[21],A[21],n,k,p,gasit;
int cmmdc(int a, int b){
    if(b==0) return a;
    else return cmmdc(b,a%b);
}
void afisare(){
    for(int i=1;i<=k;i++)
    cout<<"\t"<<A[X[i]];
    cout<<endl;
    gasit=1;
}
void back(int pas, int c){
    for(int i=X[pas-1]+1;i<=n;i++){
        X[pas]=i;
        int cc=c;//copie cmmdc
        c=cmmdc(c,A[X[pas]]);
        if(pas==k){
            if(c>=p) afisare();
        }
        else back(pas+1,c);
        c=cc;//revin la cmmdc anterior
    }
}
int main(){
    cout<<" Introduceti valorile numerelor naturale n, k, p: \t";
    cin>>n>>k>>p;
    cout<<" Introduceti cele "<<n<<" elemente ale multimii: \t\t";
    for(int i=1;i<=n;i++)
    cin>>A[i];
    cout<<"Submultimile sunt: \n";
    back(1,0);
    if(!gasit) cout<<"Nu are solutie!"; return 0;
}
```

Rezultatele execuției:

```
Introduceti valorile numerelor naturale n, k, p:    7 3 5
Introduceti cele 7 elemente ale multimii:           3 6 9 15 20 24 30
Submultimile sunt:
        6       24      30
        15      20      30
```

Problema 5: Generarea funcțiilor injective

Condiția problemei	*Exemplu*
Să se genereze toate funcțiile injective $F:\{1,2,3,...,n\}\to\{1,2,3,...,m\}$	*Pentru n=3 și m=3, programul va afișa:* *f(1)=1 f(2)=2 f(3)=3* *f(1)=1 f(2)=3 f(3)=2* *f(1)=2 f(2)=1 f(3)=3* *f(1)=2 f(2)=3 f(3)=1* *f(1)=3 f(2)=1 f(3)=2* *f(1)=3 f(2)=2 f(3)=1* *Am obținut 6 soluții*

Implementare C++

```
#include <iostream>
using namespace std;
int x[20],n,m;
void scriere(){
    for (int i=1; i<=n; i++){
        cout <<"\tf("<<i<<")="<<x[i];cout <<"\t";
    } cout <<"\n";
}
int valid(int k){
    for (int i=1;i<k;i++)
    if (x[k]==x[i]) return 0;
    return 1;
}
void back(int k){
    for (int i=1;i<=m;i++){
        x[k]=i;
        if (valid(k))
        if (k==n) scriere();
        else back(k+1);
    }
}
int main(){
    cout<<" Introduceti valorile numerelor naturale n, m: \t"; cin>>n>>m;
    cout<<"Generarea functiilor injective: \n";
    if (m<n) cout <<"Nu exista functii injective.";
    else back(1);
    return 0;
}
```

Rezultatele execuției:

```
Introduceti valorile numerelor naturale n, m:       3 3
Generarea functiilor injective:
        f(1)=1          f(2)=2          f(3)=3
        f(1)=1          f(2)=3          f(3)=2
        f(1)=2          f(2)=1          f(3)=3
        f(1)=2          f(2)=3          f(3)=1
        f(1)=3          f(2)=1          f(3)=2
        f(1)=3          f(2)=2          f(3)=1
```

Problema 6: Generarea funcțiilor surjective

Condiția problemei	*Exemplu*
Să se genereze toate funcțiile surjective $F:\{1,2,3,...,n\}\to\{1,2,3,...,m\}$	*Pentru n=3 și m=2, programul va afișa:* *f(1)=1 f(2)=1 f(3)=2* *f(1)=1 f(2)=2 f(3)=1* *f(1)=1 f(2)=2 f(3)=2* *f(1)=2 f(2)=1 f(3)=1* *f(1)=2 f(2)=1 f(3)=2* *f(1)=2 f(2)=2 f(3)=1* *Am obținut 6 soluții*

Implementare C++

```
#include <iostream>
using namespace std;
int x[20],m,n;
void afisare(){
    for(int i=1;i<=n;i++){
        cout <<"\tf("<<(i)<<")="<<x[i];
        cout <<"\t";
    } cout <<"\n";
}
int valid(int k){
    int sw=1,i,ap[20]={0};
    for(i=1;i<=k;i++) ap[x[i]]=1;
    for(i=1;i<=m;i++)
    if (ap[i]==0) sw=0;
    return sw;
}
void back(int k){
    for (int i=1;i<=m;i++){
        x[k]=i;
        if(k==n){
            if(valid(k)) afisare();
        }
        else back(k+1);
    }
}
int main(){
    cout<<" Introduceti valorile numerelor naturale n, m: \t";
    cin>>n>>m;
    cout<<"Generarea functiilor surjective: \n";
    if (m>n) cout <<"Nu exista functii surjective.";
    else back(1);
    return 0;
}
```

Rezultatele execuției:

```
Introduceti valorile numerelor naturale n, m:       3 2
Generarea functiilor surjective:
        f(1)=1          f(2)=1          f(3)=2
        f(1)=1          f(2)=2          f(3)=1
        f(1)=1          f(2)=2          f(3)=2
        f(1)=2          f(2)=1          f(3)=1
        f(1)=2          f(2)=1          f(3)=2
        f(1)=2          f(2)=2          f(3)=1
```

Problema 7: Echipe din k elevi

Condiția problemei	*Exemplu*
Într-o clasă de k elevi sunt f fete (f<=k). Afişaţi toate echipele care se pot constitui din m elevi, dintre care p fete, unde avem relația: $p \le m \le k$, $p \le f$. Numerele k, f, p şi m se citesc de la tastatură. Soluţiile posibile se afişează în felul următor: mai întâi fetele (f1, f3, etc.) şi apoi băieţii. *Indicaţii: numerotăm fetele de la 1 la f şi băieţii de la f+1 la k şi generăm toate echipele de p elevi, punând condiţia suplimentară să fie m fete în echipă.*	*Pentru k=4, f=3, m=3 și p=2, programul va afișa:* `f1 f2 b1` `f1 f3 b1` `f2 f3 b1` *Am obținut 3 soluții*

Implementare C++

```
#include <iostream>
using namespace std;
int n,m,p,f,st[20],nr_f;
void afiseaza(){
    for (int j=1;j<=m;j++)
    if (st[j]<=f) cout<<"f"<<st[j]<<" ";
    for (int j=1;j<=m;j++)
    if (st[j]>f) cout<<"b"<<st[j]-f<<" ";
    cout<<"\n";
}
void back(int k){
    for(int i=st[k-1]+1;i<=n;i++){
        st[k]=i;
        if(i<=f) nr_f++;
        if (k==m){
            if (nr_f==p) afiseaza();
        }
        else back(k+1);
        if(i<=f) nr_f--;
    }
}
int main(){
    cout<<"Numarul total de elevi=";cin>>n;
    cout<<"Numarul total de fete=";cin>>f;
    cout<<"Numarul de elevi din echipe=";cin>>m;
    cout<<"Numarul de fete din echipe=";cin>>p;
    back(1);
    return 0;
}
```

Rezultatele execuției:

```
Numarul total de elevi=5
Numarul total de fete=2
Numarul de elevi din echipe=4
Numarul de fete din echipe=2
f1 f2 b1 b2
f1 f2 b1 b3
f1 f2 b2 b3
```

Problema 8: Șiruri strict crescătoare de divizori ai lui k

Condiția problemei	*Exemplu*
Să se genereze toate şirurile strict crescătoare de lungime m formate din divizori ai unui număr dat k (0<k, m<1000). În cazul în care nu există soluţie, se va scrie pe ecran mesajul "Fără soluţie". Indicaţii: se formează şirul divizorilor lui n şi apoi se foloseşte algoritmul de generare al combinărilor.	*Pentru k=10 și m=3, programul va afișa:* 1 2 5 1 2 10 1 5 10 2 5 10 *Am obținut 4 soluții*

Implementare C++

```
#include <iostream>
using namespace std;
int n,m,nr,st[20],diviz[20];
void formare(){
    nr=0;
    for (int i=1;i<=n/2;i++)
    if(!(n%i)) diviz[++nr]=i;
    diviz[++nr]=n;
}
void scriere(){
    for (int j=1;j<=m;j++)
    cout<<diviz[st[j]]<<" ";
    cout<<"\n";
}
void back(int k){
    for(int i=st[k-1]+1;i<=nr;i++){
        st[k]=i;
        if (k==m) scriere();
        else back(k+1);
    }
}
int main(){
    cout<<" Introduceti valorile numerelor naturale n, m: \t";
    cin>>n>>m;
    formare();
    cout<<"Generarea sirurilor strict crescatoare de divizori: \n";
    if(m>nr) cout<<"Fara solutie!";
    else back(1);
    return 0;
}
```

Rezultatele execuției:

```
Introduceti valorile numerelor naturale n, m:       50 5
Generarea sirurilor strict crescatoare de divizori:
        1       2       5       10      25
        1       2       5       10      50
        1       2       5       25      50
        1       2       10      25      50
        1       5       10      25      50
        2       5       10      25      50
```

Problema 9: Promovarea unui examen

Condiția problemei	*Exemplu*
Să se elaboreze toate modalitățile de a promova un examen care conține N itemi, știind că la fiecare item (i) se poate obține un punctaj între 1 și (P[i]), iar pentru a fi promovat, un candidat trebuie să obțină măcar M puncte.	*Pentru N=3 și punctajele maxime pentru fiecare din cei 3 itemi: 10, 20, 30, unde punctajul minim de trecere: M=20, programul va afișa toate cele 5115 soluții.*

Implementare C++

```
#include <iostream>
using namespace std;
int N, M,V[10], P[10],nr=0;
void Citeste (){
    cout << "Numar de itemi: \t\t"; cin >> N;
    cout << "Punctaj maxim pentru "<<N<<" itemi:  \t";
    for (int i=1; i<=N; i++){
        cin >> P[i];
    } cout << "Punctaj minim necesar: \t\t" ; cin >> M;
}
int Punctaj (int k){
    int S=0;
      for (int i=1; i<=k; i++) S+=V[i]; return S;
}
int Solutie (int k){
      return (k==N) && (Punctaj(k)>=M);
}
void Tipar (int k){
    for (int j=1; j<=k; j++){
        cout<<"Itemul "<<j << "  "<<V[j]<<" p"<<"\t";nr++;
    }cout<<endl;
}
void back (){
    for (int k=1; k<=N; k++)
    V[k]=0;
    int k=1;
      while (k>0){
             while (V[k]<P[k]){
                    V[k] ++;
            if (Solutie(k)) Tipar (k);
            else k++;
             } V[k--]=0;
      }
}
int main (int){
      Citeste(); back ();
     cout<<"TOTAL "<<nr/N<<" SOLUTII";
}
```

Rezultatele execuției:

```
Numar de itemi:                 3
Punctaj maxim pentru 3 itemi:   10 20 30
Punctaj minim necesar:          20
Itemul 1  1 p   Itemul 2  1 p   Itemul 3  18 p

Itemul 1  10 p  Itemul 2  20 p  Itemul 3  30 p
TOTAL 5115 SOLUTIII
```

Notă:

- ✓ *Am afișat doar prima și a 5115-a soluție. Nu am afișat toate soluțiile, deoarece sunt multe.*

Problema 10: Matrice pătratică binară și simetrică

Condiția problemei	*Exemplu*
Să se genereze toate matricile pătratice de dimensiune N, formate doar din elemente 0 şi 1, simetrice faţă de diagonala principală şi cu diagonala principală 0.	*Pentru N=2, programul va afișa:* *0 0* *0 0* _____*soluția 1* *0 1* *1 0* _____*soluția 2*

Implementare C++

```
#include <iostream>
using namespace std;
int N, V[100], A[10][10],nr=0;
int Solutie (int k){
       return (k==(N*(N-1)/2));
}
void Afisare (){
       cout << "-------------------------"<<endl;
       for (int i=1; i<=N; i++){
              for (int j=1; j<=N; j++) cout << A[i][j]<< " ";nr++;
              cout << endl;
       }
}
void Tipar (){
       for (int i=1, p=1; i<= N-1; i++)
              for (int j=i+1; j<=N; j++){
                     A[i][j] = V[p]; A[j][i] = V[p]; p++;
              }
       Afisare();
}
void Back (){
    int k=1;
       for (int k=1; k<=N*(N-1)/2; k++)
    V[k]=-1;
       while (k>0){
              while (V[k]<1){
                     V[k] ++;
             if (Solutie(k)) Tipar ();
             else k++;
              } V[k--]=-1;
       }
}
int main (){
       cout << "Introduceti dimensiunea matricii patratice: \t"; cin >> N;
       Back ();
       cout<<"\nTOTAL "<<nr/N<<" SOLUTII";
}
```

Rezultatele execuției:

```
Introduceti dimensiunea matricii patratice:         2
-------------------------
0 0
0 0
-------------------------
0 1
1 0

TOTAL 2 SOLUTII
```

SARCINI PENTRU EXERSARE

Problema 1	***Date de intrare***	***Date de ieşire***
Magazinul ZorelShop vinde k specii de peşti despre care se cunosc m perechi de peşti care nu pot fi puşi în acelaşi acvariu, deoarece se atacă. Borel are un acvariu şi vrea să îşi cumpere un număr maxim de specii de peşti de la magazinul ZorelShop. Ajutaţi-l pe Borel să aleagă speciile de peşti astfel încât să poată avea un număr maxim de specii în acvariul său.	`6` `5` `1 2` `1 3` `1 4` `3 5` `3 6`	`2 4 5 6`

Problema 2	***Date de intrare***	***Date de ieşire***
Se consideră k piese de domino citite ca perechi de numere naturale, fiecare pe câte un rând de intrare. Se citeşte apoi un număr natural g. Să se afişeze cel mai lung lanţ domino care se poate forma cu piesele date, fără a roti piesele. (Un lanţ domino se alcătuieşte din piese domino astfel încât o piesă este urmată de alta a cărei prima jumătate coincide cu jumatatea a doua a piesei curente [28].)	`6` `1 2` `1 3` `3 4` `2 3` `3 5` `4 5`	`1 2` `2 3` `3 4` `4 5`

Problema 3	***Date de intrare***	***Date de ieşire***
Un student are de dat k examene numerotate de la 1 la k într-o sesiune formata din t zile (t este cel puţin de 2 ori mai mare decât k). Afişaţi toate modurile în care îşi poate programa studentul examenele astfel încât să nu dea 2 examene în zile consecutive şi să dea examenele în ordine de la 1 la k.	`3` `6`	`0 1 0 2 0 3` `1 0 0 2 0 3` `1 0 2 0 0 3` `1 0 2 0 3 0` `0 = zi liberă`

Problema 4	***Date de intrare***	***Date de ieşire***
Din fişierul cub.in se citesc de pe prima linie două numere naturale n şi m, apoi de pe următoarele n linii se citesc n perechi (l şi c, unde l este un număr natural ce determină lungimea laturii, iar c este sir de caractere de lungime maxim 20 ce determină culoarea pentru n cuburi). Să se construiască toate turnurile formate din cel puţin m cuburi care se pot forma din cuburile citite din fişier, ştiind că un cub se poate pune peste un altul doar dacă are latura strict mai mică şi culoarea diferită de a celui peste care vrem să-l punem. Să se afişeze turnurile obţinute şi turnul format din cele mai multe cuburi. Un turn se afişează începând cu cel mai de sus cub.	`3 2` `3 verde` `4 rosu` `1 rosu`	`Turnul 1:` `1 rosu` `3 verde` `Turnul 2:` `3 verde` `4 rosu` `Turnul maxim:` `1 rosu` `3 verde` `4 rosu`

Problema 5	***Date de intrare***	***Date de ieşire***
Se dă un cuvânt de lungime N, format din caractere distincte. Să se afişeze toate cuvintele de lungime N care se pot forma cu caracterele cuvântului dat şi care îndeplinesc condiţia că nu există două consoane sau două vocale alăturate.	`house`	`ohuse ohesu osuhe` `osehu uhose uheso` `usohe useho ehosu` `ehuso esohu esuho`

Problema 6	***Date de intrare***	***Date de ieşire***
Fiind dată o tablă de şah de dimensiunea nxn şi un cal în colţul stânga sus al acesteia, se cere să se afişeze toate posibilităţile de mutare a acestei piese de şah astfel încât să treacă o singură dată prin fiecare pătrat al tablei. O soluţie va fi afişată ca o matrice nxn în care sunt numerotate săriturile calului [29].	`5`	`1  14 9  20 23` `10 19 22 15 8` `5  2  13 24 21` `18 11 4  7  16` `3  6  17 12 25`

Problema 7	**Date de intrare**	**Date de ieşire**
Fiind dată o tablă de şah de dimensiunea nxn şi un cal în colţul stânga sus al acesteia (x0,y0). Se cere cel mai scurt traseu pe care trebuie să-l parcurgă calul pentru a ajunge în poziţia (x1,y1) fără a trece de două ori prin aceeaşi poziţie şi ştiind că există anumite căsuţe inundate, unde calul nu poate sări (notate cu 1).	5 1 1 5 5 0 0 0 0 0 0 0 1 0 0 0 0 0 1 0 0 0 0 0 0 0 1 1 0 0	1 0 0 0 0 0 0 0 3 0 0 2 0 0 0 0 0 4 0 0 0 0 0 0 5

Problema 8	**Date de intrare**	**Date de ieşire**
Fiind date n discuri etichetate de la 1 la n de raze r[i] şi grosime g[i], r<=i<=n, să se afişeze toate turnurile de k discuri care se pot forma astfel încât discurile din turn să aibă razele în ordine descrescătoare, iar grosimea discurilor alăturate să fie diferite ***[30]****.*	5 3 1 2 3 4 5 1 2 1 3 4	3 2 1 4 2 1 4 3 2 5 2 1 5 3 2 5 4 1 5 4 2 5 4 3

Problema 9	**Date de intrare**	**Date de ieşire**
Se citeşte un număr natural n. Să se genereze toate numerele naturale a căror reprezentare binară au acelaşi număr de cifre 0 (semnificative) şi respectiv 1 ca în reprezentarea binară a numărului n.	53	39 43 45 46 51 53 54 57 58 60

Problema 10	**Date de intrare**	**Date de ieşire**
Se citeşte un număr natural n. Să se genereze toate numerele naturale de 2 cifre care se împart fără rest la n şi au suma cifrelor egală cu un număr mai mic ca n .	5	10 20 30 40

Problema 11	**Date de intrare**	**Date de ieşire**
Să se genereze toate numerele naturale de 3 cifre care au suma cifrelor egală cu n.	3	102 111 120 210

Problema 12	**Date de intrare**	**Date de ieşire**
Se citeşte un număr natural n. Să se genereze toate numerele naturale de 3 cifre care se împart fără rest la n şi au produsul cifrelor un număr egală cu 2n .	2	114 122 141 212 221 411

Problema 13	**Date de intrare**	**Date de ieşire**
Se citeşte un număr natural n. Să se genereze toate numerele naturale de 2 cifre care se împart fără rest la n şi au suma cifrelor egală cu n+3 .	7	28 91

Problema 14	**Date de intrare**	**Date de ieşire**
Într-un magazin se găsesc spre vânzare n (n<=20) produse, numerotate de la 1 la n. Un cumpărător, care dispune de o sumă de bani S, doreşte să cheltuiască toţi banii pe care îi are în acest magazin, dar fără a cumpăra două produse de acelaşi fel. Cunoscând preţul fiecărui produs, scrieţi un program care să afişeze toate posibilităţile acestui cumpărător de a-şi cheltui banii. Se citesc de la tastatură valorile S, n şi apoi cele n numere naturale reprezentând preţurile celor n produse. În fişierul de ieşire MAGAZIN.OUT se vor scrie toate soluţiile, câte o soluţie pe fiecare linie. O soluţie este constituită din numerele de ordine ale produselor cumpărate, separate prin câte un spaţiu. Dacă nu există soluţii, fişierul de ieşire va fi gol .	5 20 50 70 30 60 100	3 4

4.6 Implementarea metodei backtracking la rezolvarea problemelor avansate

Problema 1: Tabla de șah. Aranjarea celor n dame.

Condiția problemei	*Exemplu*
Fie o tablă de şah de dimensiune NxN, unde N>3. Să se aranjeze pe ea N regine fără ca ele să se atace. Reamintim că o regină atacă linia, coloana şi cele 2 diagonale pe care se află [31].	*Pentru N=4, un exemplu ar fi:* 0 **D** 0 0 0 0 0 **D** **D** 0 0 0 0 0 **D** 0 *În total am avea 2 soluții*

Implementare C++

```
#include <iostream>
#include <stdlib.h>
using namespace std;
int n,v[100],sol,i,j;
void afisare(){
    sol++;
    cout<<"\n Solutia: "<<sol<<"\n";
    for (i=1;i<=n;i++){
        for (j=1;j<=n;j++)
        if (v[i]==j) cout<<"D ";
        else cout<<"0 "; cout<<'\n';
    }
}
int valid(int k){
    for (i=1;i<=k-1;i++)
    if ((v[i]==v[k])||(abs(v[k]-v[i])==(k-i))) return 0; return 1;
}
int solutie(int k){
    if (k==n) return 1; return 0;
}
void BK(int k){
    for (int i=1;i<=n;i++){
        v[k]=i;
        if (valid(k)==1){
            if (solutie(k)==1) afisare();
            else BK(k+1);
        }
    }
}
int main(){
    cout << "Introduceti dimensiunea tablei de sah: \t"; cin >> n;
    BK(1);
}
```

Rezultatele execuției:

```
Introduceti dimensiunea tablei de sah:      4

 Solutia: 1
0 D 0 0
0 0 0 D
D 0 0 0
0 0 D 0

 Solutia: 2
0 0 D 0
D 0 0 0
0 0 0 D
0 D 0 0
```

Problema 2: Tabla de șah. Aranjarea celor n ture.

Condiția problemei	*Exemplu*
Fie o tablă de șah de dimensiune NxN, unde N>3. Să se aranjeze pe ea N ture fără ca ele să se atace. Reamintim că o tură atacă linia și coloana pe care se află ***[31]****.*	*Pentru N=4, un exemplu ar fi:* **T** 0 0 0 0 **T** 0 0 0 0 **T** 0 0 0 0 **T** *În total am avea 24 soluții*

Implementare C++

```
#include <iostream>
#include <stdlib.h>
using namespace std;
int n,v[100],sol,i,j;
void afisare(){
    sol++;
    cout<<"\n Solutia: "<<sol<<"\n";
    for (i=1;i<=n;i++){
        for (j=1;j<=n;j++)
        if (v[i]==j) cout<<"T ";
        else cout<<"0 "; cout<<'\n';
    }
}
int valid(int k){
    for (i=1;i<=k-1;i++)
    if (v[i]==v[k]) return 0; return 1;
}
int solutie(int k){
    if (k==n) return 1; return 0;
}
void BK(int k){
    for (int i=1;i<=n;i++){
        v[k]=i;
        if (valid(k)==1){
            if (solutie(k)==1) afisare();
            else BK(k+1);
        }
    }
}
int main(){
    cout << " Introduceti dimensiunea tablei de sah: \t"; cin >> n;
    BK(1);
}
```

Rezultatele execuției:

```
Introduceti dimensiunea tablei de sah:      4

 Solutia: 1
T 0 0 0
0 T 0 0
0 0 T 0
0 0 0 T

 Solutia: 24
0 0 0 T
0 0 T 0
0 T 0 0
T 0 0 0
```

Notă:

- ✓ *Am afișat doar prima și a 24-a soluție. Nu am afișat toate soluțiile, deoarece sunt multe.*

Problema 3: Tabla de șah. Aranjarea celor n nebuni (ofițeri).

Condiția problemei	*Exemplu*
Fie o tablă de şah de dimensiune NxN, unde N>3. Să se aranjeze pe ea N nebuni fără ca ei să se atace. Reamintim că un nebun atacă cele 2 diagonale pe care se află ***[31]****.*	*Pentru N=4, un exemplu ar fi:* **N** 0 0 0 **N** 0 0 0 **N** 0 0 0 **N** 0 0 0 *În total am avea 24 soluții*

Implementare C++

```
#include <iostream>
#include <stdlib.h>
using namespace std;
int n,v[100],sol,i,j;
void afisare(){
    sol++;
    cout<<"\n Solutia: "<<sol<<"\n";
    for (i=1;i<=n;i++){
        for (j=1;j<=n;j++)
        if (v[i]==j) cout<<"N ";
        else cout<<"0 "; cout<<'\n';
    }
}
int valid(int k){
    for (i=1;i<=k-1;i++)
    if (abs(v[k]-v[i])==(k-i)) return 0; return 1;
}
int solutie(int k){
    if (k==n) return 1; return 0;
}
void BK(int k){
    for (int i=1;i<=n;i++){
        v[k]=i;
        if (valid(k)==1){
            if (solutie(k)==1) afisare();
            else BK(k+1);
        }
    }
}
int main(){
    cout << "Introduceti dimensiunea tablei de sah: \t"; cin >> n; BK(1);
}
```

Rezultatele execuției:

```
Introduceti dimensiunea tablei de sah:      4

 Solutia: 1
N 0 0 0
N 0 0 0
N 0 0 0
N 0 0 0

 Solutia: 24
0 0 0 N
0 0 0 N
0 0 0 N
0 0 0 N
```

Notă:

- ✓ *Am afișat doar prima și a 24-a soluție. Nu am afișat toate soluțiile, deoarece sunt multe.*

Problema 4: Tabla de șah. Traseu pion.

Condiția problemei	*Exemplu*
Pe o tablă de șah NxN sunt plasate figuri marcate prin valoarea 1, iar prin valoarea 0 sunt marcate pozițiile libere. Într-o pozitie j0 de pe prima linie se află un pion. Determinați toate traseele pe care poate ajunge pionul până pe ultima linie. Traseele vor fi afișate în matrice, dar și ca șir de poziții.	*Pentru N=3 și matricea de dimensiunea 3:* 1 0 0 1 0 0 0 0 0 *Poziția pionului pe prima linie este 3.* *Vom avea o soluție unică:* 1 0 0 1 0 1 0 0 2 *Șirul de poziții este: (1,3),(2,3) și (3,3)*

Implementare C++

```
#include<iostream>
using namespace std;
int a[20][20],n,jo,b[20][3],i,j;
void afis(){
    for(int i=1;i<=n;i++){
        for(int j=1;j<=n;j++)
             cout<<a[i][j]<<" "; cout<<endl;
    }
    for(int i=1;i<=n;i++)
    cout<<b[i][1]<<","<<b[i][2]<<" "; cout<<endl<<endl;
 }
int inside(int i,int j){
    return (i<=n && j>=1 && j<=n);
}
void back(int i, int j, int pas){
    b[pas][1]=i; b[pas][2]=j;
    if (i==n)  afis();
    else{
        if(a[i+1][j]==0) { a[i+1][j]=pas; back(i+1,j,pas+1); a[i+1][j]=0; }
        if(a[i+1][j-1]==1) { a[i+1][j-1]=pas; back(i+1,j-1,pas+1); a[i+1][j-1]=1; }
        if(a[i+1][j+1]==1) { a[i+1][j+1]=pas; back(i+1,j+1,pas+1); a[i+1][j+1]=1; }
    }
    a[i][j]++;
}
int main(){
    cout << "Introduceti dimensiunea tablei de sah: \t"; cin >> n;
    cout << "Introduceti matricea patratica de dimensiunea "<<n<<":\n";
    for(i=1;i<=n;i++)
    for(j=1;j<=n;j++) cin>>a[i][j];
    cout << "Introdu pozitia pionului pe prima linie: \t"; cin>>jo;
    back(1,jo,1); return 0;
}
```

Rezultatele execuției:

```
Introduceti dimensiunea tablei de sah:      3
Introduceti matricea patratica de dimensiunea 3:
1 0 0
0 0 1
0 1 0
Introdu pozitia pionului pe prima linie:        2
1 0 0
0 0 1
0 1 2
1,2 2,3 3,3

1 0 0
0 0 1
0 2 0
1,2 2,3 3,2
```

Problema 5: Tabla de șah. Pionul bate maxim piese.

Condiția problemei	*Exemplu*
Pe o tabla de sah nXn sunt plasate marcate prin valoarea -1, iar prin valoarea 0 sunt marcate pozițiile libere. Într-o poziție jo de pe prima linie se află un pion. Determinați toate traseele pe care poate ajunge pionul până pe ultima linie. Determinați traseul pe care pionul ia număr maxim de piese. Traseele vor fi afisate ca matrice de pași.	*Pentru N=3 și matricea de dimensiunea 3:* 0 -1 0 0 -1 0 0 0 -1 *Poziția pionului pe prima linie este 3.* *Pionul va bate maxim 2 piese:* 0 # 1 0 2 0 0 0 3

Implementare C++

```
#include<iostream>
using namespace std;
int a[20][20],n,jo,i,j,maxx,amax[20][20];
void afis(int a[20][20], int n){
    for(int i=1;i<=n;i++){
       for(int j=1;j<=n;j++)
             if(a[i][j]==-1) cout<<"# ";
             else cout<<a[i][j]<<" "; cout<<endl;
    } cout<<endl;
 }
void back(int i, int j, int pas, int np){
    int inou,jnou;
    if (i==n){
        afis(a,n);
        if(np>maxx){
            maxx=np;
            for(int l=1;l<=n;l++) for(int c=1;c<=n;c++) amax[l][c]=a[l][c];
        }
    }
    else{
        for(int k=-1;k<=1;k++){
            inou=i+1; jnou=j+k;
            if(a[inou][jnou]==0 && k==0){
                a[inou][jnou]=pas+1; back(inou,jnou,pas+1,np); a[inou][jnou]=0;
            }
            else if(a[inou][jnou]==-1 && k!=0){
                a[inou][jnou]=pas+1; back(inou,jnou,pas+1,np+1); a[inou][jnou]=-1;
            }
        }
    } a[i][j]++;
}
int main(){
    cout << "Introduceti dimensiunea tablei de sah: \t"; cin >> n;
    for(i=1;i<=n;i++) for(j=1;j<=n;j++) cin>>a[i][j];
    cout << "Introduceti pozitia pionului din randul 1: \t"; cin>>jo;
    a[1][jo]=1; back(1,jo,1,0);
    cout<<"Bate maxim "<<maxx<<" piese si solutia este: \n";
    afis(amax,n);
}
```

Rezultatele execuției:

```
Introduceti dimensiunea tablei de sah:      3
0 -1 0
0 -1 0
0 0 -1
Introduceti pozitia pionului din randul 1:  3
Bate maxim 2 piese si solutia este:
0 # 1
0 2 0
0 0 3
```

Problema 6: Tabla de șah. Săritura calului.

Condiția problemei	*Exemplu*
Fiind dată o tablă de șah de dimensiunea NxN și un cal în colțul stânga sus al acesteia, se cere să se afișeze toate posibilitățile de mutare a acestei piese de șah astfel încât să treacă o singură dată prin fiecare pătrat al tablei. O soluție va fi afișată ca o matrice NxN în care sunt numerotate săriturile calului.	*Pentru N=5.Vom scrie una din soluții:* 1 14 9 20 23 10 19 22 15 8 5 2 13 24 21 18 11 4 7 16 3 6 17 12 25 *Numărul total de soluții este: 304*

Implementare C++

```
#include<iostream>
using namespace std;
const int dx[8]={-1,1,2,2,1,-1,-2,-2};
const int dy[8]={-2,-2,-1,1,2,2,1,-1};
int a[100][100],n,nr=0;
void afis(){
    int i,j;
    for(i=1;i<=n;i++){
        for(j=1;j<=n;j++)
        cout<<a[i][j]<<"\t";
        cout<<endl;
    }
    cout<<endl;nr++;
}
int inside(int i,int j){
    return (i>=1 && i<=n && j>=1 && j<=n);
}
void back(int i, int j, int pas){
    int k,inou,jnou;
    a[i][j]=pas;
    if (pas==n*n)  afis();
    else for(k=0;k<8;k++){
            inou=i+dx[k]; jnou=j+dy[k];
            if (inside(inou,jnou) && a[inou][jnou]==0)
               back(inou,jnou,pas+1);
        }
        a[i][j]=0;
}
int main(){
    cout << "Introduceti dimensiunea tablei de sah: \t";
    cin >> n;
    back(1,1,1);
    cout<<"NUMARUL TOTAL DE SOLUTII: "<<nr;
}
```

Rezultatele execuției:

```
Introduceti dimensiunea tablei de sah:      5
1       14      9       20      23
10      19      22      15      8
5       2       13      24      21
18      11      4       7       16
3       6       17      12      25

NUMARUL TOTAL DE SOLUTII: 304
```

Problema 7: Roboțelul TOBBY

Condiția problemei	*Exemplu*
Se citește o matrice NxM cu elemente numere naturale și o poziție i0, j0 din matrice în care se află Tobby. El este un roboțel care se poate deplasa paralel cu liniile și coloanele matricii pe poziții alăturate. Afișați toate drumurile pe care poate parcurge robotul matricea, trecând prin fiecare poziție de atâtea ori cât este valoarea poziției respective. Drumurile vor fi afișate ca perechi de coordonate.	*Pentru N=4, M= și matricea:* 0 0 0 0 0 1 1 0 0 1 2 1 0 0 0 1 *Vom avea următoarea parcurgere a lui Tobby:* 2,2 3,2 3,3 2,3 3,3 3,4 4,4 2,2 2,3 3,3 3,2 3,3 3,4 4,4

Implementare C++

```
#include<iostream>
using namespace std;
const int dx[4]={-1,1,0,0};
const int dy[4]={0,0,-1,1};
int a[20][20],p=0,n,m,io,jo,b[400][3],i,j;
void afis(int k){
    for(int i=1;i<=k;i++)
    cout<<b[i][1]<<","<<b[i][2]<<" "; cout<<endl;
}
int inside(int i,int j){
    return (i>=1 && i<=n && j>=1 && j<=m);
}
void back(int i, int j, int pas){
    int k,inou,jnou;
    a[i][j]--; b[pas][1]=i; b[pas][2]=j;
    if (pas==p)  afis(p);
    else for(k=0;k<4;k++){
            inou=i+dx[k]; jnou=j+dy[k];
            if (inside(inou,jnou) && a[inou][jnou]>0){
                back(inou,jnou,pas+1);
            }
    } a[i][j]++;
}
int main(){
    cout<<"Introduceti dimensiunea matricei: \t"; cin >> n>> m;;
    cout<<"Introduceti matricea: \n";
    for(i=1;i<=n;i++)
    for(j=1;j<=m;j++){
        cin>>a[i][j]; p=p+a[i][j];
    }
    cout<<"Introduceti pozitia lui Tobby: \t"; cin>>io>>jo;
    cout<<"Afisarea drumului parcurs de Tobby: \n"; back(io,jo,1); return 0;
}
```

Rezultatele execuției:

```
Introduceti dimensiunea matricei:   4 4
Introduceti matricea:
0 0 0 0
0 1 1 0
0 1 2 1
0 0 0 1
Introduceti pozitia lui Tobby:      2 2
Afisarea drumului parcurs de Tobby:
2,2 3,2 3,3 2,3 3,3 3,4 4,4
2,2 2,3 3,3 3,2 3,3 3,4 4,4
```

Problema 8: Cel mai lung șir din vocale

Condiția problemei	*Exemplu*
Se citește o matrice NxM care conține litere mici. Găsiți cel mai lung șir format numai din vocale care se poate construi cu literele din matrice prin deplasare paralelă cu liniile și coloanele matricii, fără a trece de mai multe ori prin aceeași literă.	*Pentru N=5, M=5 un exemplu ar fi:* `q w e r t` `y u i o p` `a s d f g` `h j k l z` `x c v b n` *Cel mai lung sir de vocale:* `eiu`

Implementare C++

```
#include<iostream>
#include<cstring>
using namespace std;
const int dx[4]={-1,1,0,0}, dy[4]={0,0,-1,1};
char a[20][20],s[50],smax[50],v[]="aeiou";
int n,m,b[20][20],i,j,maxx;
int inside(int i,int j){
    return i>=1 && i<=n && j>=1 && j<=m;
}
void back(int i, int j, int pas){
    int k,inou,jnou; b[i][j]=1;
    if (pas>maxx) {
        maxx=pas; s[pas]=0;
        strcpy(smax,s);
    }
    for(k=0;k<4;k++){
        inou=i+dx[k]; jnou=j+dy[k];
        if (inside(inou,jnou)&&strchr(v,a[inou][jnou])&&b[inou][jnou]==0){
                s[pas]=a[inou][jnou];
                back(inou,jnou,pas+1);
        }
       } b[i][j]=0;
}
int main(){
    cout << "Introduceti dimensiunea matricei: \t";
    cin>>n>>m;
    cout << "Introduceti elementele matricei: \n";
    for(i=1;i<=n;i++)
    for(j=1;j<=m;j++)
    cin>>a[i][j];
    for(i=1;i<=n;i++)
    for(j=1;j<=m;j++)
    if(strchr(v,a[i][j])){
        s[0]=a[i][j]; back(i,j,1);
    }
    cout << "Cel mai lung sir de vocale: \t";
    cout<<smax; return 0;
}
```

Rezultatele execuției:

```
Introduceti dimensiunea matricei:   3 3
Introduceti elementele matricei:
a e t
i o s
d u b
Cel mai lung sir de vocale:     eaiou
```

Problema 9: Cățelul Rorel

Condiția problemei	*Exemplu*
Harta unui oraș este codificată ca o matrice NxM. în care 0 reprezintă poziție accesibilă (strada) și -1 reprezintă poziție inaccesibilă (clărire, zid). În poziția i, j se află Jorel, iar în poziția x,y se află stăpânul său. Rorel se poate deplasa pe poziții cu valoarea 0, alăturate pe linii și coloane cu poziția curentă, fără să treacă de două ori prin aceeași poziție. Determinați și afișați cel mai scurt traseu pe care poate ajunge Rorel până la stăpân. Soluția se va afișa prin pași marcați în matrice și ca șir de coordonate prin care trece Rorel [32].	*Pentru N=3, M=4 și matricea:* 0 -1 0 0 0 0 0 0 0 -1 -1 -1 *Poziția lui Rorel:* 3 1 *Poziția stăpânului:* 1 4 *Drumul parcurs în matrice:* 0 -1 5 6 2 3 4 0 1 -1 -1 -1 *Vom avea următoarea parcurgere a lui Jorel:* 3,1 2,1 2,2 2,3 1,3 1,4

Implementare C++

```
#include<iostream>
using namespace std;
const int di[4]={-1,0,1,0}, dj[4]={0,1,0,-1};
int a[10][10], b[10][10],sol[100][3],x[100][3],n,m,pmin=100;
int ir,jr,ij,jj;
void citire(){
    cout<<"Introdu dimensiunea matricei: \t"; cin >> n>> m;;
    cout<<"Introdu matricea: \n";
    for(int i=1;i<=n;i++)
    for(int j=1;j<=m;j++) cin>>a[i][j];
    cout<<"Introdu pozitia lui Rorel: \t"; cin>>ir>>jr;
    cout<<"Introdu pozitia stapanului: \t"; cin>>ij>>jj;
}
int inside(int i, int j){
      return i>=1 && i<=n && j>=1 && j<=m;
}
void alege(int pas){
      if(pas<pmin){
            pmin=pas;
            for(int i=1;i<=n;i++)
            for(int j=1;j<=m;j++) b[i][j]=a[i][j];
            for(int i=1;i<=pas;i++){
                sol[i][1]=x[i][1];  sol[i][2]=x[i][2];
            }
      }
}
void afis() {
      for (int i=1;i<=n;i++){
          for(int j=1;j<=m;j++) cout<<b[i][j]<<" "; cout<<endl;
      }
      for(int i=1;i<=pmin;i++) cout<<sol[i][1]<<","<<sol[i][2]<<"  ";
}
void back(int i,int j, int pas) {
      int inou, jnou,k;
      if(i==ij && j==jj) alege(pas-1);
      else for(k=0;k<=3;k++){
                  inou=i+di[k]; jnou=j+dj[k];
                  if(inside(inou,jnou) && a[inou][jnou]==0){
                        a[inou][jnou]=pas; x[pas][1]=inou; x[pas][2]=jnou;
                        back(inou,jnou,pas+1); a[inou][jnou]=0;
                  }
      }
}
int main(){
      citire();
       a[ir][jr]=1; x[1][1]=ir; x[1][2]=jr; back(ir,jr,2);
      cout<<"Afisarea drumului parcurs de Rorel: \n"; afis();
     return 0;
}
```

Problema 10: Soldatul Gerald

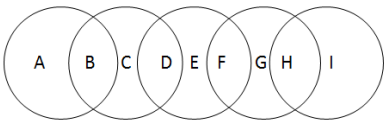
Condiția problemei	*Exemplu*
Gerald trebuie să străbată un labirint cu capcane reprezentat de o matrice NxN. Pentru fiecare celulă a labirintului, se cunoaşte timpul în minute după care celula respectivă devine capcană. După ce o celulă devine capcană, Gerald moare dacă intră în acea celulă. Gerald porneşte din colțul stâng-sus al labirintului și trebuie să ajungă în colțul dreapt-jos. El are nevoie de un minut ca să treacă dintr-o celulă într-una vecină și se poate deplasa în sus, în jos, spre stânga sau spre dreapta. Să se afişeze timpul minim în care poate Gerald să străbată labirintul, numărul de drumuri de timp minim, precum și toate drumurile minime pe care le poate urma Gerald prin labirint de la intrare la ieşire, astfel încât Gerald să nu moară. Drumurile vor fi afişate ca matrici în care sunt indicați pașii lui Gerald [32].	*Pentru N=3, M=3 și matricea:* 1 2 3 4 5 6 7 8 9 *Vom determina timpul minim:* 5 sec *Vom determina nr. de drumuri:* 6 *Vom afișa o modalitate de parcurgere a labirintului de către soldatul Gerald:* 1 2 3 0 0 4 0 0 5 *În total vom avea 6 soluții*

Implementare C++

```
#include <iostream>
using namespace std;
const int di[]={-1,0,1,0}, dj[]={0,1,0,-1};
int a[100][100], x[100][100], m,n,pmin=10000,nr=0,z=0;
void citire(){
    cout<<"Introdu dimensiunea matricei: \t"; cin >> n>> m;
    cout<<"Introdu elementele matricii: \n";
    for(int i=1;i<=m;i++) for(int j=1;j<=n;j++) cin>>a[i][j];
}
void afis(int x[100][100]){
    cout<<"\nMatricea solutie: \n";
    for(int i=1;i<=m;i++){
        for(int j=1;j<=n;j++)
        cout<<x[i][j]<<" "; cout<<endl;
    } z++;cout<<"Parcurgerea "<<z<<endl;
}
void alege(int pas){
    if(pas<pmin){ pmin=pas; nr=1;}
    else if(pas==pmin) nr++;
}
void back(int i, int j, int pas){
    x[i][j]=pas;
    if(i==m && j==n) alege(pas);
    else for(int d=0;d<4;d++){
        int inou=i+di[d]; int jnou=j+dj[d];
        if(pas<a[inou][jnou] && x[inou][jnou]==0 && pas<pmin)
        back(inou,jnou,pas+1);
    } x[i][j]=0;
}
void backmin(int i, int j, int pas){
    x[i][j]=pas;
    if(i==m && j==n && pas==pmin) afis(x);
    else for(int d=0;d<4;d++){
        int inou=i+di[d]; int jnou=j+dj[d];
        if(pas<a[inou][jnou] && x[inou][jnou]==0 && pas<pmin)
        backmin(inou,jnou,pas+1);
    } x[i][j]=0;
}
int main() {
    citire(); back(1,1,1); cout<<"Tmin si Ndrumuri: \t"<<pmin<<" "<<nr<<endl;
    backmin(1,1,1);return 0;
}
```

SARCINI PENTRU EXERSARE

1. *Fie o tablă dreptunghiulară, împărţită în n*m căsuţe identice. Iniţial, într-una din căsuţe se află un pion care se poate deplasa pe orizontală sau verticală fără a putea ieşi din cadrul tablei. Dându-se poziţia iniţială şi cea finală a pionului, precum şi de câte ori poate să treacă acesta prin fiecare poziţie, se cere să se reconstituie traseul parcurs de pion.*
2. *Un soldat având la dispoziţie un detector de mine trebuie să parcurgă un teren minat având forma unei table dreptunghiulare de dimensiune m*n. În fiecare din pătratele tablei poate fi amplasată o mină. Soldatul porneşte dintr-un colţ al terenului şi trebuie să ajungă în colţul opus. Să se determine toate modalităţile în care soldatul poate parcurge terenul minat.*
3. *Se dă un lac îngheţat sub forma unui tablou m*n. O broscuţă se găseşte în poziţia (1,1), iar în poziţia (m,n) se găseşte o insectă. În gheaţă sunt găuri ale căror coordonate se cunosc. Găsiţi drumul cel mai scurt până la insectă şi înapoi ştiind că: deoarece broscuţa visa să fie cal, ea va sări ca un cal de şah. În punctele în care păşeşte broscuţa gheaţa cedează. Broscuţa nu poate păşi decât pe gheaţă.*
4. *Pe un teren de dimensiune dreptunghiulară cu denivelări se află un sportiv care doreşte să se antreneze pentru un concurs de alpinism. Cunoscând altitudinea fiecărei porţiuni din teren şi poziţia iniţială a alpinistului, să se determine traseele de lungime minimă pe care trebuie să le parcurgă sportivul pentru a ajunge într-o poziţie de altitudine maximă. Se ştie că sportivul nu doreşte să coboare deloc, dar poate merge pe loc drept. De asemenea el se poate deplasa ortogonal sau diagonal cu un singur pas.*
5. *Trebuie trecute cu barca peste o apa Z zebre şi T tigrii. Un transport conţine 2 animale. Tigrii nu pot ataca zebrele dacă sunt în minoritate sau egali cu ele. Afişati toate posibilităţile de transport astfel încât animalele să nu se atace. Animalele trecute nu se pot întoarce.*
6. *Trebuie trecute cu barca peste o apă M misionari şi M canimali. Un transport conţine 2 persoane. Cu o barcă ce poate transporta la un moment dat unul sau doi oameni. Se cere să se găsească toate modalităţile de a transporta pe toţi pe celălalt mal, fără a permite la vreun transport ca numărul de canibali dintr-o locaţie să depăşească numărul de misionari.*
7. *Se dă un teren sub forma de matrice cu M linii şi N coloane. Fiecare element al matricei reprezintă un subteren cu o anumită altitudine dată de valoarea reţinută de element (număr natural). Într-un astfel de subteren, de coordonate (lin,col) se găseşte o bilă. Ştiind că bila se poate deplasa în orice porţiune de teren aflată la nord, est, sud sau vest, de altitudine strict inferioară porţiunii pe care se găseşte bila. Se cere să se găsească toate posibilităţile ca bila să părăsească terenul **[29]**.*
8. *Scrieţi un program care afişează în fişierul cercuri.out toate modalităţitile de înlocuire a literelor din imaginea alăturată cu cifre de la 1 la 9 astfel încât suma cifrelor din fiecare cerc să fie aceeaşi.*



9. *Se citeşte un număr natural n. Generaţi şi afişaţi toate combinaţiile de câte n cifre binare care nu au două cifre de 1 alăturate.*
10. *Se dă o listă de n cuvinte. Să se formeze cel mai lung şir în care fiecare cuvânt începe cu litera cu care se termină predecesorul.*
11. *Să se scrie în fişierul soluţii.in toate numerele de n, (0<n<10) cifre, care adunate fiecare cu numărul obţinut prin inversarea ordinii cifrelor sale, dau un număr pătrat perfect.*
12. *Să se genereze toate permutările de dimensiune n cu valori din 1..n cu proprietatea că oricare ar fi 2<=i<=n, există un 1<=j<=i , astfel încât |v(i)-v(j)|=1.*
13. *Se dau N puncte albe şi N puncte negre în plan ,de coordonate întregi .Fiecare punct alb se uneşte cu câte un punct negru, astfel încât din fiecare punct, fie el alb sau negru, pleacă exact un segment. Să se determine o astfel de configuraţie de segmente încât oricare două segmente să nu se intersecteze. Se citesc 2N perechi de coordonate corespunzând punctelor **[31]**.*
14. *Se citeşte un număr natural n. Să se genereze toate numerele care au în reprezentarea lor binară atâtea cifre de 1 şi de 0 câte are şi reprezentare numărului n în baza 2.*
15. *Se citeşte un număr natural n. Să se genereze toate numerele care au în reprezentarea lor binară atâtea cifre de 1 şi cu n cifre de 0 în reprezentare numărului n în baza 2.*

METODA DIVIDE ET IMPERA

5.1 Noțiuni generale despre divide et impera

Divide et impera este o clasă de algoritmi care funcționează pe baza tacticii divide et impera. Divide et impera se bazează pe principiul descompunerii problemei în două sau mai multe subprobleme (mai ușoare), care se rezolvă, iar soluția pentru problema inițială se obține combinând soluțiile subproblemelor.

De multe ori, subproblemele sunt de același tip și pentru fiecare din ele se poate aplica aceeași tactică a descompunerii în (alte) subprobleme, până când (în urma descompunerilor repetate) se ajunge la probleme care admit rezolvare imediată.

Nu toate problemele pot fi rezolvate prin utilizarea acestei tehnici. Se poate afirma că numărul celor rezolvabile prin "divide et impera" este relativ mic, tocmai datorită cerinței ca problema să admită o descompunere repetată.

Divide et impera admite o implementare recursivă, deorece subproblemele sunt similare problemei inițiale, dar de dimensiuni mai mici. Principiul general prin care se elaborează algoritmi recursivi este: "ce se întâmplă la un nivel, se întâmplă la orice nivel" (având grijă să asigurăm condițiile de terminare) ***[21]****.*

Principiul fundamental al recursivității este autoapelarea unui subprogram când acesta este activ; ceea ce se întâmplă la un nivel, se întâmplă la orice nivel, având grija să asigurăm condiția de terminare ale apelurilor repetate. Asemănător se întamplă și în cazul metodei divide et impera la un anumit nivel sunt două posibilități:

- *s-a ajuns la o (sub)problemă simplă ce admite o rezolvare imediată, caz în care se rezolvă (sub) problema și se revine din apel (la subproblema anterioară, de dimensiuni mai mari);*
- *s-a ajuns la o (sub) problemă care nu admite o rezolvare imediată, caz în care o descompunem în două sau mai multe subprobleme și pentru fiecare din ele se continuă apelurile recursive (ale funcției).*

Așadar, un algoritm prin divide et impera se elaborează astfel: la un anumit nivel avem două posibilități:

- *s-a ajuns la o problemă care admite o rezolvare imediată (condiția de terminare), caz în care se rezolvă și se revine din apel;*
- *nu s-a ajuns în situația de la punctul precedent, caz în care problema curentă este descompusă în (două sau mai multe) subprobleme, pentru fiecare din ele urmează un apel recursiv al funcției, după care combinarea rezultatelor are loc fie pentru fiecare subproblemă, fie la final, înaintea revenirii din apel.*

Metoda divide et impera se poate aplica în rezolvarea unei probleme care îndeplinesc următoarele condiții :

- *se poate descompune în (două sau mai multe) suprobleme ;*
- *aceste suprobleme sunt independente una față de alta (o subproblemă nu se rezolvă pe baza alteia și nu folosește rezultate celeilalte);*
- *aceste subprobleme sunt similare cu problema inițială;*
- *la randul lor subproblemele se pot descompune (dacă este necesar) în alte subprobleme mai simple;*
- *aceste subprobleme simple se pot soluționa imediat prin algoritmul simplificat.*

Deoarece puține probleme îndeplinesc condițiile de mai sus, aplicarea metodei este destul de rară. După cum sugerează și numele 'desparte și stăpânește ', etapele rezolvării unei probleme (numită problema inițială) în divide et impera sunt :

- *descompunerea problemei inițiale în subprobleme independente, smilare problemei de bază, de dimensiuni mai mici ;*

- *descompunerea treptată a subproblemelor în alte subprobleme din ce în ce mai simple, până când se pot rezolva imediat prin algoritmul simplificat ;*
- *rezolvarea subproblemelor simple ;*
- *combinarea soluțiilor găsite pentru construirea soluțiilor subproblemelor de dimensiuni din ce în ce mai mari ;*
- *combinarea ultimelor soluții determină obținerea soluției problemei inițiale .*

În etapa finală a metodei divide et impera se produce combinarea subproblemelor (rezolvate deja) prin secvențele de revenire din apelurile recursive ***[33]****.*

Etapele metodei divide et impera (prezentate anterior) se pot reprezenta prin următorul subprogram general recursiv exprimat în limbaj natural:

```
Subprogram DIVIMP (PROB);
Dacă PROBLEMA PROB este simplă
Atunci se rezolvă și se obține soluția SOL
Altfel pentru i=1,k execută DIVIMP(PROB) și se obține SOL1;
Se combină soluțiile SOL 1,…, SOL K și se obține SOL;
Sfârșit_subprogram;
```

Deci, subprogramul DIVIMP se apelează pentru problema inițială PROB; aceasta admite descompunerea în K subprobleme simple; pentru acestea se reapelează recursiv subprogramul; în final se combină soluțiile acestor K subprobleme.

De obicei, problema inițială se descompune în două subprobleme mai simple; în acest caz etapele generale ale metodei divide et impera se pot reprezenta concret, în limbaj pseudocod, printr-un subprogram recursiv astfel:

```
Subprogram  DIVIMP(li,ls,sol);
Dacă ((ls-li)<=eps)
    Atunci REZOLVA (li,ls,sol);
Altfel
    DIVIDE (li,m,ls);
    DIVIMP(li,msol1);
    DIVIMP(m,ls,sol2);
    COMBINA(sol1,sol2,sol);
Sfârșit_subprogram;
```

Subprogramul DIVIMP se apelează pentru problema inițială care are dimensiunea între limita inferioară (li) și limita superioară (ls); dacă (sub)problema este simplă (ls-li<=eps), atunci subprogramul REZOLVA îi află soluția imediat și se produce întoarcerea din apelul recursiv; dacă (sub)problema este (încă) complexă, atunci subprogramul DIVIDE o împarte în două subprobleme, alegând poziția m între limitele li și ls; pentru fiecare din cele două subprobleme se reapelează recursiv subprogramul DIVIMP; în final, la întoarcerile din apeluri se produce combinarea celor două soluții sol1 și sol2 prin apelul subprogramului COMBINA ***[34]****.*

5.2 Analiza complexității algoritmilor metodei divide et impera

Un algoritm D&I împarte problema în mai multe subprobleme similare cu problema inițială şi de dimensiuni mai mici, rezolva sub-problemele recursiv şi apoi combina soluțiile obținute pentru a obține soluția problemei inițiale. Sunt trei pași pentru aplicarea algoritmului D&I:

- *Divide: împarte problema în una sau mai multe probleme similare de dimensiuni mai mici.*
- *Impera (stăpânește sau guvernează): rezolva subprobleme recursiv; dacă dimensiunea (sub) problemelor este mică se rezolva iterativ.*
- *Combină: combină soluțiile sub-problemelor pentru a obține soluția problemei inițiale.*

Complexitatea algoritmilor D&I se calculează după formula: T(n)=D(n)+S(n)+C(n), unde D(n), S(n) şi C(n) reprezintă complexitățile celor 3 pași descriși mai sus: divide, stăpânește respectiv combină. Această paradigmă conduce în g eneral la un timp de c alcul polinomial.

Vom presupune că dimensiunea n_i a subproblemei P_i satisface $n_i \le n/b$, unde $b>1$, în acest fel pasul de divizare reduce o subproblemă la altele de dimensiuni mai mici, ceea ce sigură proprietatea de terminare a subprogramului recursiv. Divizarea problemei în subprobleme şi asamblarea soluțiilor necesită timpul $O(n^k)$. Complexitatea timp T(n) a algoritmului Divide et impera este dată de următoare relaţie de recurenţă:

$$T(n) = \begin{cases} (1), & \text{dacă } n \le n_0 \\ a \cdot T(\frac{n}{b}) + O(n^k), & \text{dacă } n > n_0 \end{cases}$$

Teoremă:

Dacă $n > n_0$ atunci:

$$T(n) = \begin{cases} O(n^{\log_b a}), \text{ dacă } a > b^k, \\ O(n^k \cdot \log_b n), \text{ dacă } a = b^k, \\ O(n^k), \text{ dacă } a < b^k. \end{cases}$$

Algoritmii divide et impera au o bună comportare în timp, dacă dimensiunile subproblemelor în care se împarte subproblema sunt aproximativ egale. Dacă lipsesc fazele de combinare a soluțiilor, viteza acestor algoritmi este şi mai mare (ex. c ăutarea binară). În majoritatea cazurilor, descompunerile înjumătăţescdimensiunea problemei, un exemplu tipic reprezentându-l sortarea prin interclasare.

Exemplul 1

Se dă un vector sortat crescător (v[1], v[2], …, v[n]) ce conține valori reale distincte și o valoare x. Să se găsească la ce poziție apare x în vectorul dat.

Rezolvare:

BinarySearch (Cautare Binara), se rezolva cu un algoritm D&I:

- *Divide: împărțim vectorul în doi sub-vectori de dimensiune n/2.*
- *Stăpânește: aplicăm algoritmul de căutare binară pe sub-vectorul care conține valoarea căutată.*
- *Combină: soluția sub-problemei devine soluția problemei inițiale, motiv pentru care nu mai este nevoie de etapa de combinare.*

Complexitate:

- *complexitate temporală : T=O(log(n)), se deduce din recurenta T(n)=T(n/2)+O(1).*
- *complexitate spațiala : S=O(1), nu avem structuri de date complexe auxiliare, atragem atenția ca acest algoritm se poate implementa și recursiv, caz în care complexitatea spațiala devine O(log(n)), întrucât salvăm pe stivă O(log(n)) parametri (întregi, referințe) **[35]**.*

Exemplul 2

Se consideră 3 tije S (sursa), D (destinație), aux (auxiliar) şi n discuri de dimensiuni distincte (1,2,...,n - ordinea crescătoare a dimensiunilor) situate inițial toate pe tija S în ordinea 1,2,...,n (de la vârf către bază). Singura operație care se poate efectua este de a selecta un disc ce se află în vârful unei tije şi plasarea lui în vârful altei tije astfel încât să fie așezat deasupra unui disc de dimensiune mai mare decât a sa. Să se găsească un algoritm prin care se mută toate discurile de pe tija S pe tija D (problema turnurilor din Hanoi).

Rezolvare:

Pentru rezolvarea problemei folosim următoarea strategie:

- *mutăm primele $n-1$ discuri de pe tija S pe tija aux folosindu-ne de tija D.*
- *mutăm discul n pe tija D.*
- *mutăm apoi cele $n-1$ discuri de pe tija aux pe tija D, folosindu-ne de tija S.*

Ideea din spate este că avem mereu o singură sursă și o singură destinație să atingem un scop. Întotdeauna a 3-a tijă va fi considerată auxiliară și poate fi folosită pentru a atinge scopul propus.

Complexitate:

- *complexitate temporală : $T(n)=O(2n)$, se deduce din recurenta $T(n)=2*T(n-1)+O(1)$.*
- *complexitate spațială : $S(n)=O(n)$, la un moment dat, nivelul maxim de recursivitate este n.*

Exemplul 3

Fie că avem la dispoziție o tablă pătratică de dimensiuni 2^n*2^n. Dorim să scriem pe pătrățelele tablei numere naturale cuprinse între 1 și 2^n*2^n, conform unei parcurgeri mai deosebite pe care o numim Zoro. O parcurgere Zoro vizitează recursiv cele patru cadrane ale tablei în ordinea: stanga-sus, dreapta-sus, stânga-jos, dreapta-jos. La un moment dat dorim să aflăm ce număr de ordine trebuie să scriem conform parcurgerii Zoro pe anumite pătrățele date prin coordonatele lor (x,y). Vom începe umplerea tablei întotdeauna din colțul din stânga-sus.

Rezolvare:

Analizând modul în care se completează matricea din enunțul nostru, observăm că la fiecare etapă împărțim matricea în 4 submatrici (4 subprobleme). De asemenea, șirul de numere pe care dorim să-l punem în matrice se împarte în 4 secvențe, fiecare corespunzând unei submatrici. Mai observăm astfel că problema suportă descompunerea în subprobleme disjuncte și cu structură similară, ceea ce ne face să ne gândim la o soluție cu Divide et Impera.

Complexitate:

- *complexitate temporală : $T=O(n)$, adică $\log_4(2^n)=1/2*\log_2(2^n)=1/2*n$.*
- *complexitate spațială : $S=O(n)$, stocăm parametrii pentru recursivitate; soluția se poate implementa și iterativ, caz în care $S=O(1)$; deoarece dimensinile spațiului de căutare sunt 2^n*2^n, n este foarte mic ($n<=15$), de aceea o soluție iterativă nu aduce niciun câștig efectiv în această situație* ***[36]****.*

5.3 Forme specifice de căutare ale metodei divide et impera

1. *Căutare binară*

Algoritmul de căutare binară este un algoritm de căutare folosit pentru a găsi un element într-o listă ordonată (tablou unidimensional/vector). Algoritmul funcționează pe baza tehnicii divide et impera. Valoarea căutată este comparată cu cea a elementului din mijlocul listei. Dacă e egală cu cea a acelui element, algoritmul se termină. Dacă e mai mare decât acea valoare, algoritmul se reia, de la mijlocul listei până la sfârșit, iar dacă e mai mică, algoritmul se reia pentru elementele de la începutul listei până la mijloc. Întrucât la fiecare pas cardinalul mulțimii de elemente în care se efectuează căutarea se înjumătățește, algoritmul are complexitate logaritmică.

Se consideră un tablou unidimensional v de n elemente deja sortat și trei variabile: i=început, s=sfârșit și m=mijloc. Metoda verifică de mai multe ori dacă mijlocul vectorului/tabloului unidimensional este egal cu elementul căutat:

- *în cazul în care este egală, variabila m reprezintă poziția elementului în vector;*
- *dacă nu se îndeplinește condiția de egalitate se trece la verificarea poziției elementului căutat în vector astfel: dacă elementul căutat este mai mic decât elementul din mijlocul vectorului, variabila "s" ia valoarea lui "m", iar dacă nu variabila i ia valoarea lui m.*

Totul se repetă atât timp cât i este mai mic decât s.

Se aplică pentru șiruri ordonate, principiul algoritmului, constând în înjumătățirea repetată a intervalului în care se caută elementul specificat. La fiecare pas se compară elementul căutat cu elementul din mijlocul intervalului de căutare, dacă avem egalitatea algoritmul se termină. Dacă nu avem egalitate căutăm doar în jumătatea din stânga sau doar în jumătatea din dreapta în funcție de sensul inegalității și de ordinea crescătoare sau descrescătoare a elementelor șirului. Procesul se repetă cu jumătatea de interval aleasă. Algoritmul se opreşte fie când am găsit elementul, fie când am terminat de inspectat șirul și nu am găsit nimic.

Practic, ignorăm jumătate din elemente imediat după o singură comparație:

- *Comparăm x cu elementul din mijloc.*
- *Dacă x se potrivește cu elementul din mijloc, returnăm indicele din mijloc.*
- *Altfel Dacă x este mai mare decât elementul mijlociu, atunci x poate sta doar în jumătatea submulțimii din dreapta după elementul mijlociu. Așa că reapărăm pentru jumătatea dreaptă.*
- *În rest (x este mai mic) se repetă pentru jumătatea stângă.*

2. *Căutare binară uniformă*

Căutare binară uniformă este o optimizare a algoritmului de căutare binară atunci când multe căutări sunt făcute pe același tablou sau pe multe tablouri de aceeași dimensiune. În căutarea binară normală, efectuăm operații aritmetice pentru a găsi punctele intermediare. Aici precomputăm punctele intermediare și le completăm în tabelul de căutare. Aspectul de tablă funcționează, în general, mai repede decât aritmetica făcută (adăugarea și deplasarea) pentru a găsi punctul intermediar.

Algoritmul este foarte similar cu algoritmul de căutare binară. Singura diferență este că un tabel de căutare este creat pentru un tablou și tabelul de căutare este utilizat pentru a modifica indexul indicelui din tablou care face căutarea mai rapidă. În loc să mențină legăturile inferioare și superioare, algoritmul menține un index și indexul este modificat, folosind tabelul de căutare.

Algoritmul utilizează o tabelă de căutare - notată delta – pentru a actualiza indexul poziției în care se caută în vectorul dat. În funcție de n (numărul de elemente din vectorul dat), delta trebuie să ia valori astfel încât să înjumătățească la fiecare pas pe n.

Algoritmul înlocuieşte utilizarea limitei inferioare, a limitei superioare şi a mijlocului cu doi indecşi, unul pentru poziția curentă și unul pentru rata lui de modificare: δ. După fiecare comparație de inegalitate se setează $i = i \pm \delta$ și $\delta = \delta/2$ **[37]** .

3. Căutare ternară

Căutarea ternară este un algoritm divide et impera care poate fi utilizat pentru a găsi un element într-un tablou. Este similar cu acea căutarea binară în care împărțim tabloul în două. În acest algoritm, împărțim tabloul dat în trei și determinăm care are cheia (elementul căutat).

Putem împărți tabloul în trei părți luând mid1 și mid2 care pot fi calculate așa cum se arată mai jos. Inițial, l și r vor fi egale cu 0, respectiv n-1, unde n este lungimea tabloului:

- *mid1 = l + (r-l)/3;*
- *mid2 = r – (r-l)/3.*

Este important ca tabloul unidimensional să fie sortat pentru a efectua căutărea ternară în el.

Pași pentru efectuarea căutării Ternare:

- *În primul rând, comparăm cheia cu elementul de la mid1. Dacă este găsit egal, ne întoarcem la mid1.*
- *Dacă nu, atunci comparăm cheia cu elementul de la mid2. Dacă este găsit egal, ne întoarcem la mid2.*
- *Dacă nu, atunci verificăm dacă cheia este mai mică decât elementul de la mid1. Dacă da, atunci reveniți la prima parte.*
- *Dacă nu, atunci verificăm dacă cheia este mai mare decât elementul la mid2. Dacă da, atunci reveniți la a treia parte.*
- *Dacă nu, atunci vom recidiva la a doua parte (din mijloc) **[38]**.*

4. Căutare Fibonacci

Căutarea Fibonacci este o tehnică bazată pe comparație care folosește numere Fibonacci pentru a căuta un element într-un tablou sortat. Asemănări cu căutarea binară:funcționează pentru tabelele sortate; utilizează un algoritm divide et impera și are complexitatea temporală T(n)=log n.

Pași pentru efectuarea căutării Fibonacci:

- *Fie elementul căutat să fie x.*
- *Ideea este de a găsi mai întâi cel mai mic număr Fibonacci care este mai mare sau egal cu lungimea tabloului dat.*
- *Fie numărul Fibonacci găsit este fib (numărul lui Fibonacci găsit). Folosim al (m-2)-lea număr Fibonacci drept index (dacă este o poziție validă).*
- *Fie al (m-2-lea) număr Fibonacci este i, comparăm arr [i] cu x, dacă x este același, revenim i. În rest, dacă x este mai mare, vom recidiva pentru subarray după i, altfel repetăm pentru subarray înainte de i **[39]**.*

5. Căutare prin interpolare

Căutarea binară se referă întotdeauna la elementul de mijloc pentru a verifica. Pe de altă parte, căutarea interpolării poate merge în diferite locații în funcție de valoarea cheii căutate. De exemplu, dacă valoarea cheii este mai aproape de ultimul element, căutarea interpolării este probabil să înceapă căutarea către partea finală.

Pentru a găsi poziția de căutat, utilizează următoarea formulă:

*pos = lo + [(x-array [lo]) * (hi-lo) / (array [hi] -array [Lo])], unde array [] este vectorul unde trebuie căutat elementul, x este elementul de căutat, lo este indicele de pornire în array [], este indicele de încheiere în array []. Ideea formulei este de a returna o valoare mai mare a pos când elementul de căutat este mai aproape de array [hi]. Și valoare mai mică atunci când este mai aproape de array [lo].*

Căutarea prin interpolare se bazează pe strategia adoptată de o persoană când caută un cuvânt într-un dicționar. Astfel, dacă cuvântul căutat începe cu litera C, deschidem dicționarul undeva mai la început, iar când cuvântul începe cu litera V, deschidem dicționarul mai pe la sfârșit. Dacă e este valoarea căutată în vectorul a[ic..sf], atunci partiționăm spațiul de căutare pe poziția m=ic+(e-a[ic])(sf-ic)/(a[sf]-a[ic]). Această partiționare permite o estimare mai bună în cazul în care elementele lui a sunt numere distribuite uniform **[37]** .*

5.4 Implementarea metodei divide et impera la rezolvarea problemelor elementare

Problema 1: Radical de ordinul n din x

Condiția problemei	*Exemplu*
Se citesc două numere, n și x, n natural și x real pozitiv. Fără a folosi funcția pow, extrageți cu 4 zecimale exacte radicalul de ordinul n din x, folosind metoda divide et impera.	*Pentru N=2 și X=5.* *Soluție:* N=2 X=1 obținem 1 N=2 X=2 obținem 1.41421 N=2 X=3 obținem 1.73205 N=2 X=4 obținem 2 N=2 X=5 obținem 2.23607

Implementare C++

```
#include<iostream>
#include<cmath>
using namespace std;
float f(float y, int n, float x){
    float p=1;
    for(int i=1;i<=n;i++)
    p=p*y;
    return p-x;
}
float DEI(float s, float d, int n, float x){
   if(d-s<=0.000001) return s;
   else{
        float m=(s+d)/2;
        if(f(m,n,x)==0) return m;
        else
            if(f(m,n,x)<0) return DEI(m,d,n,x);
            else return DEI(s,m,n,x);
   }
}
int main(){
    int n; float x;
    cout<<"Introduceti ordinul radicalului: \t\t"; cin>>n;
    cout<<"Introduceti numarul de sub radical: \t"; cin>>x;
    cout<<endl;
    for (int i=2;i<=x;i++){
        cout<<"Radical de ordinul "<<n<<" din "<<i<<" este: \t";
        cout<<DEI(0,sqrt(i),n,i)<<"\n";
    }
    return 0;
}
```

Rezultatele execuției:

```
Introduceti ordinul radicalului:                2
Introduceti numarul de sub radical:     10

Radical de ordinul 2 din 2 este:        1.41421
Radical de ordinul 2 din 3 este:        1.73205
Radical de ordinul 2 din 4 este:        2
Radical de ordinul 2 din 5 este:        2.23607
Radical de ordinul 2 din 6 este:        2.44949
Radical de ordinul 2 din 7 este:        2.64575
Radical de ordinul 2 din 8 este:        2.82843
Radical de ordinul 2 din 9 este:        3
Radical de ordinul 2 din 10 este:       3.16228
```

Problema 2: Ridicarea la putere

Condiția problemei	*Exemplu*
Se citesc două numere întregi n și x. Scrieți un program care va calcula pe x^n , folosind metoda divide et impera. Este posibil să presupunem că x și n sunt mici și că surplusul nu se întâmplă.	*Pentru N=5 și X=10.* *Soluție:* 1^{10} = 1; 2^{10} = 1024; 3^{10} = 59049; 4^{10} = 1048576; 5^{10} = 9765625;

Implementare C++

```
#include<iostream>
#include<cmath>
using namespace std;
int DEI(int x, unsigned int y){
    if (y == 0) return 1;
    else
        if (y%2 == 0)
            return DEI(x, y/2)*DEI(x, y/2);
    else
        return x*DEI(x, y/2)*DEI(x, y/2);
}
int main(){
    int n, x;
    cout<<"Introduceti un numar ca baza: \t\t";
    cin>>x;
    cout<<"Introduceti un numar ca putere: \t";
    cin>>n;
    cout<<endl;
    for (int i=1;i<=x;i++){
        cout<<"Solutie: "<<i<<"^"<<n<<" = ";
        cout<<DEI(i,n)<<"\n";
    }
    return 0;
}
```

Rezultatele execuției:

```
Introduceti un numar ca baza:           20
Introduceti un numar ca putere:         2

Solutie: 1^2 = 1
Solutie: 2^2 = 4
Solutie: 3^2 = 9
Solutie: 4^2 = 16
Solutie: 5^2 = 25
Solutie: 6^2 = 36
Solutie: 7^2 = 49
Solutie: 8^2 = 64
Solutie: 9^2 = 81
Solutie: 10^2 = 100
Solutie: 11^2 = 121
Solutie: 12^2 = 144
Solutie: 13^2 = 169
Solutie: 14^2 = 196
Solutie: 15^2 = 225
Solutie: 16^2 = 256
Solutie: 17^2 = 289
Solutie: 18^2 = 324
Solutie: 19^2 = 361
Solutie: 20^2 = 400
```

Problema 3: Polinom

Condiția problemei	*Exemplu*
Să se rezolve ecuația $x^3+x-1=0$ pe intervalul [0,1], folosind metoda divide et impera.	*Soluție:* Găsim $\frac{d}{dx}(x^3+x-1)=3x^2+1;$ Fie $x_0=1,$ atunci calculăm x_{n+1} până când $\Delta x_{n+1}<0.000001.$ Calculele se vor face în felul următor: PASUL 1: $f(x_0)=1^3+1-1=1;\quad f'(x_0)=3\cdot 1^2+1=4;$ $x_1=1-\frac{1}{4}=0.75;\quad \Delta x_1=\lvert 0.75-1\rvert=0.25;$ PASUL 2: $f(x_1)=0.75^3+0.75-1=0.171875;$ $f'(x_1)=30.75^2+1=2.6875;$ $x_2=0.75-\frac{0.171875}{2.6875}=0.68604\ldots;$ $\Delta x_2=\lvert 0.68604\ldots-0.75\rvert=0.06395\ldots;$ Celelalte se calculează asemănător celor de mai sus.Rezultatele finale ale calculelor manuale sunt: 1 x1=0.75 Δx1=0.25 2 x2=0.68604... Δx2=0.06395... 3 x3=0.68233... Δx3=0.00370... 4 x4=0.68232... Δx4=0.00001... 5 x5=0.68232... Δx5=1.18493...E-10

Implementare C++

```
#include<iostream>
#include<cmath>
using namespace std;
float f(float x){
    return pow(x,3)+x-1;
}
float DEI(float s, float d){
    if(d-s<=0.0000001) return s;
    else {
        float m=(s+d)/2;
        if(f(m)==0) return m;
        else if(f(m)<0) return DEI(m,d);
                else return DEI(s,m);
    }
}
int main(){
    cout<<"Solutia este: \t"<<DEI(0,1);
    return 0;
}
```

Rezultatele execuției:

```
Solutia este:   0.682328
```

Problema 4: Sumă și produs din vector

Condiția problemei	*Exemplu*
Să se determine suma și produsul elementelor unui vector, flosind algoritmul divide et impera. Se vor utiliza subprograme aparte.	*Pentru N=8 și elementele tabloului unidimensional: 2 3 4 1 5 6 8 9.* *Soluție:* `Stânga Dreapta` `1 2 3 4 1 5 6 8 9` `2 2 3 4 1 5 6 8 9` `3 2+3 4+1 5+6 8+9` `4 5 5 11 17` `5 5+5 11+17` `6 10 28` `7 10+28` `8 38` `Soluția în 8pași: suma=38` `În așa mod obținem și produsul=51840`

Implementare C++

```
#include<iostream>
#include <stdlib.h>
using namespace std;
int v[20],n;
int DEI1(int li,int ls){
    int mij, d1 ,d2;
    if(li!=ls){
        mij=(li+ls)/2; d1= DEI1(li,mij); d2= DEI1(mij+1,ls);
        return d1+d2;
    }   else return v[li];
}
int DEI2(int li,int ls){
    int mij, d1 ,d2;
    if(li!=ls){
        mij=(li+ls)/2; d1= DEI2(li,mij); d2= DEI2(mij+1,ls);
        return d1*d2;
    }   else return v[li];
}
int main(){
    cout<<"Introdu dimensiunea tabloului unidimensional: \t";
    cin>>n;
    for(int i=1;i<=n;i++){
        cout<<"v["<<i<<"]= ";cin>>v[i];
    }
    system("cls");
    cout<<"Elementele tabloului unidimensional: \n\t";
    for(int i=1;i<=n;i++){
        cout<<v[i]<<" ";
    }   cout<<endl;
    cout<<"\nSuma celor "<<n<<" elemente:\t\t"<< DEI1(1,n);
    cout<<"\nProdusul celor "<<n<<" elemente:\t"<< DEI2(1,n);
}
```

Rezultatele execuției:

```
Elementele tabloului unidimensional:
        2 3 4 1 5 6 8 9

Suma celor 8 elemente:          38
Produsul celor 8 elemente:      51840
```

Problema 5: Min și Max din vector

Condiția problemei	*Exemplu*
Se citește un vector cu n elemente numere naturale. Să se determine elementul minim și maxim din vector, folosind divide et impera.	*Pentru N=8 și elementele tabloului unidimensional: 2 3 4 1 5 6 8 9.* *Soluție:* 1 (2 3 4 1) (5 6 8 9) 2 (2 3) (4 1) (5 6) (8 9) 3 2 1 5 8 4 (2 1) (5 8) 5 1 5 6 (1 5) 7 1 Soluția în 7 pași: MINIM(2,3,4,1,5,6,8,9)=1 În așa mod obținem și MAXIM=9

Implementare C++

```
#include<iostream>
#include <stdlib.h>
using namespace std;
int v[20],n;
int DEI1(int v[100],int s , int d){
    if ( s == d ) return v[s];
    else{
        int m = (s+d)/2; int m1 = DEI1(v,s,m); int m2 = DEI1(v,m+1,d);
        if ( m1 < m2 )  return m1;
        else return m2;
    }
}
int DEI2(int v[100],int s , int d){
    if ( s == d ) return v[s];
    else{
        int m = (s+d)/2; int m1 = DEI2(v,s,m); int m2 = DEI2(v,m+1,d);
        if ( m1 > m2 )  return m1;
        else return m2;
    }
}
int main(){
    cout<<"Introdu dimensiunea tabloului unidimensional: \t";cin>>n;
    for(int i=1;i<=n;i++){
        cout<<"v["<<i<<"]= ";cin>>v[i];
    }
    system("cls");
    cout<<"Elementele tabloului unidimensional: \n\t";
    for(int i=1;i<=n;i++){
        cout<<v[i]<<" ";
    }   cout<<endl;
    cout<<"\nElementul minim:\t"<< DEI1(v,1,n);
    cout<<"\nElementul maxim:\t"<< DEI2(v,1,n);
}
```

Rezultatele execuției:

```
Elementele tabloului unidimensional:
        10 20 30 5 8

Elementul minim:        5
Elementul maxim:        30
```

Problema 6: CMMDC din vector

Condiția problemei	*Exemplu*
Se citește un vector cu n elemente numere naturale. Să se calculeze CMMDC al elementelor vectorului, folosind divide et impera. Se vor utiliza subprograme aparte.	*Pentru N=8 și elementele tabloului unidimensional: 2 3 4 1 5 6 8 9.* *Soluție:* 1 (2 3 4 1) (5 6 8 9) 2 (2 3) (4 1) (5 6) (8 9) 3 1 1 1 1 4 (1 1) (1 1) 5 1 1 6 (1 1) 7 1 Soluția în 7 pași: CMMDC(2,3,4,1,5,6,8,9)=1

Implementare C++

```
#include<iostream>
#include <stdlib.h>
using namespace std;
int v[20],n;
int DEI(int v[20], int s, int d){
    if(s==d) return v[s];
    else{
            int x,y;
            x=DEI(v,s,(s+d)/2);
            y=DEI(v,(s+d)/2+1,d);
            while(x!=y)
                if(x>y) x=x-y;
                else y=y-x;
            return x;
    }
}
int main(){
    cout<<"Introdu dimensiunea tabloului unidimensional: \t";
    cin>>n;
    for(int i=1;i<=n;i++){
        cout<<"v["<<i<<"]= ";cin>>v[i];
    }
    system("cls");
    cout<<"Elementele tabloului unidimensional: \n\t";
    for(int i=1;i<=n;i++){
        cout<<v[i]<<" ";
    }
    cout<<"\nCMMDC ( ";
    for(int i=1;i<=n;i++){
        cout<<v[i]<<" ";
    }
    cout<<" ) = "<<DEI(v,1,n);
}
```

Rezultatele execuției:

```
Elementele tabloului unidimensional:
        6 12 18 24 30
CMMDC ( 6 12 18 24 30  ) = 6
```

Problema 7: Valoarea unei expresii

Condiția problemei	*Exemplu*
Să se claculeze valoarea următoarei expresii: E=n(n+1), folosind divide et impera. Se vor utiliza subprograme aparte.*	*Pentru n=1 vom avea:* • *1*(1+1)=1*2=2* *Pentru n=2 vom avea:* • *1*(1+1) 2*(2+1)* • *1*2 2*3* • *2 + 6 = 8* *Pentru n>3 vom avea același procedeu de descompunere în conformitate cu principiul metodei divide et impera.*

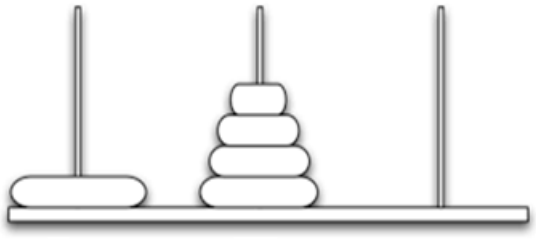
Implementare C++

```
#include <iostream>
using namespace std;
int n;
void DEI(int p,int u,int &z){
    int m,x1,x2;
    if(p==u) z=p*(p+1);
    else {
        m=(p+u)/2; //divizeaza
        DEI(p,m,x1);
        DEI(m+1,u,x2);
        z=x1+x2; //combina
    }
}
int main(){
    int z=0;
    cout<<"Introduceti valoarea lui n, n= \t";cin>>n;
    DEI(1,n,z);
    cout<<"\nValoarea sumei conform formulei este: \n";
    for(int i=1;i<=n;i++){
        DEI(1,i,z);
        cout<<"\tPentru n= "<<i<<" suma este: "<<z<<endl;
    }
}
```

Rezultatele execuției:

```
Introduceti valoarea lui n, n=      10

Valoarea sumei conform formulei este:
        Pentru n= 1 suma este: 2
        Pentru n= 2 suma este: 8
        Pentru n= 3 suma este: 20
        Pentru n= 4 suma este: 40
        Pentru n= 5 suma este: 70
        Pentru n= 6 suma este: 112
        Pentru n= 7 suma este: 168
        Pentru n= 8 suma este: 240
        Pentru n= 9 suma este: 330
        Pentru n= 10 suma este: 440
        Pentru n= 11 suma este: 572
        Pentru n= 12 suma este: 728
        Pentru n= 13 suma este: 910
        Pentru n= 14 suma este: 1120
        Pentru n= 15 suma este: 1360
```

Problema 8: Turnurile din Hanoi.

Condiția problemei	*Exemplu*
Sunt date 3 tije x, y, z. Pe tija x sunt plasate n discuri aşezate în ordinea descrescătoare a diametrelor. Se cere de trecut toate discurile de pe tija x pe tija y, folosind tija intermediară z, astfel încât să se respecte următoarea condiţie: discul mai mic totdeauna se află deasupra celui mai mare la orice transfer.	

Implementare C++

```
#include <iostream>
#include <math.h>
using namespace std;
int numarare;
void Hanoi(int n,char din_tija,char in_tija,char aux){
    if (n == 1){
        cout << "Muta discul 1 din tija " <<din_tija<<" in tija ";
        cout<<in_tija<<endl;
        return;
    }
    Hanoi(n - 1,din_tija,aux,in_tija);
    cout << "Muta discul " <<n<< " din tija " <<din_tija<<" in tija ";
    cout<<in_tija<<endl;
    Hanoi(n - 1,aux,in_tija,din_tija);
}
int main(){
    int n = 3; // Numarul de discuri
    cout << "Turnurile din Hanoi cu 3 tije si "<<n<<" discuri.\n"<<endl;
    Hanoi(n,'A','C','B'); // A, B si C sunt numele tijelor
    numarare=pow(2,n)-1;
    cout << "\nNumarul total de miscari este: "<<numarare<<endl;
    return 0;
}
```

Rezultatele execuției:

```
Turnurile din Hanoi cu 3 tije si 4 discuri.

Muta discul 1 din tija A in tija B
Muta discul 2 din tija A in tija C
Muta discul 1 din tija B in tija C
Muta discul 3 din tija A in tija B
Muta discul 1 din tija C in tija A
Muta discul 2 din tija C in tija B
Muta discul 1 din tija A in tija B
Muta discul 4 din tija A in tija C
Muta discul 1 din tija B in tija C
Muta discul 2 din tija B in tija A
Muta discul 1 din tija C in tija A
Muta discul 3 din tija B in tija C
Muta discul 1 din tija A in tija B
Muta discul 2 din tija A in tija C
Muta discul 1 din tija B in tija C

Numarul total de miscari este: 15
```

Problema 9: Contorizarea elementelor prime

Condiția problemei	*Exemplu*
Se citește un vector cu n elemente numere naturale. Să se numere câte elemente prime sunt în acest vector, folosind divide et impera. Se vor utiliza subprograme aparte.	*Pentru N=8 şi elementele tabloului unidimensional: 2 3 4 1 5 6 8 9.* *Soluție:* 1 (2 3 4 1) (5 6 8 9) 2 (2 3) (4 1) (5 6) (8 9) 3 (2 3) 1 5 Soluția în 3 pași: Numere Prime(2,3,1,5)=4

Implementare C++

```
#include <iostream>
using namespace std;
int v[100],n,p;
int prim(int x){
    for(int d=2;d<=x/2;d++)
    if(x%d==0) return 0;
    return 1;
}
void afisare_prim(){
    for(int i=1; i<=n; i++){
        int x=0;
        for(int d=2;d<=v[i]/2;d++)
        if(v[i]%d==0) x++;
        if(x==0) cout<<v[i]<<" ";
    }
}
int DEI(int p,int u){
    int m,x,y;
    if(p==u)
        if(prim(v[p]) and (v[p]!=1 || v[p]==1)) return 1;
        else return 0;
    else{
        m=(p+u)/2;
        return DEI(p,m) + DEI(m+1,u);
    }
}
int main(){
    cout<<"Introdu dimensiunea tabloului: \t";cin>>n;
    cout<<"Introdu elementele tabloului unidimensional: \n\t";
    for(int i=1;i<=n;i++)
    cin>>v[i];
    cout<<"\nNumarul de elemente prime din tablou este: \t";
    cout<<DEI(1,n);
    cout<<"\nElementele prime din tablou sunt: \t\t";
    afisare_prim();
}
```

Rezultatele execuției:

```
Introdu dimensiunea tabloului:  8
Introdu elementele tabloului unidimensional:
        2 3 4 1 5 6 8 9

Numarul de elemente prime din tablou este:      4
Elementele prime din tablou sunt:               2 3 1 5
```

Problema 10: Fibonacci

Condiția problemei	*Exemplu*
În termeni matematici, secvența F(n) a numerelor Fibonacci este definită prin relația de recurență: *F(n) = F(n-1) + F(n-2), cu valorile predefinite: F(0) = 0 și F(1) = 1.*	*Italianul Leonardo din Pisa (cunoscut în mathematică drept Fibonacci) a descoperit un șir de numere destul de interesant: 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, 987, 1597, și așa mai departe.* *Primele două elemente ale șirului sunt 0 și 1, iar al treilea element este obținut,adunând primele doua: 0 + 1 = 1. Al patrulea se obține adunând al treilea număr cu cel de-al doilea (2 + 1 = 3). Astfel, fiecare număr Fibonacci este suma celor două numere Fibonacci anterioare.*

Implementare C++

```
#include <iostream>
using namespace std;
void inmultire(int F[2][2], int M[2][2]);
void putere(int F[2][2], int n);
int fib(int n){
    int F[2][2] = {{1,1},{1,0}};
    if (n == 0) return 0;
    putere(F, n-1);
    return F[0][0];
}
void inmultire(int F[2][2], int M[2][2]){
    int x =  F[0][0]*M[0][0] + F[0][1]*M[1][0];
    int y =  F[0][0]*M[0][1] + F[0][1]*M[1][1];
    int z =  F[1][0]*M[0][0] + F[1][1]*M[1][0];
    int w =  F[1][0]*M[0][1] + F[1][1]*M[1][1];
    F[0][0] = x; F[0][1] = y; F[1][0] = z; F[1][1] = w;
}
void putere(int F[2][2], int n){
    if( n == 0 || n == 1) return;
    int M[2][2] = {{1,1},{1,0}};
    putere(F, n/2);
    inmultire(F, F);
    if (n%2 != 0) inmultire(F, M);
}
int main(){
    int n = 9;
    for(int i=0; i<=n;i++){
        cout<<"\nElementul "<<i<<" cu valoarea: \t"<<fib(i);
    }
    return 0;
}
```

Rezultatele execuției:

```
Elementul 0 cu valoarea:        0
Elementul 1 cu valoarea:        1
Elementul 2 cu valoarea:        1
Elementul 3 cu valoarea:        2
Elementul 4 cu valoarea:        3
Elementul 5 cu valoarea:        5
Elementul 6 cu valoarea:        8
Elementul 7 cu valoarea:        13
Elementul 8 cu valoarea:        21
Elementul 9 cu valoarea:        34
```

SARCINI PENTRU EXERSARE

Probleme aritmetice

1. *Să se calculeze sumele:*
   a. $1*2+2*3+3*4+ \ldots +n*(n+1);$
   b. $1*2+1*2*3+ \ldots +1*2*3*...*n;$
   c. $1^1+2^2+3^3+ \ldots +(n-1)^{n-1}+n^n;$
   d. $1+3+5+7+9+ \ldots + (2*n-1);$
   e. $3+7+11+15+ \ldots + (4*n-1);$
   f. $1/2+1/4+1/6+ \ldots + 1/(2*n);$
2. *Să se calculeze factorialul unui număr introdus de la tastatură.*
3. *Să se calculeze suma primilor n factorial, unde n este un număr introdus de la tastatură.*
4. *Să se determine valoarea unui polinom într-un punct.*
5. *Să se determine cel puțin o soluție a unei funcții într-un interval.*

Probleme cu tablouri

6. *Să se verifice dacă vectorul conține sau nu elemente pare cu exact k divizori primi.*
7. *Să verifice dacă vectorul conține sau nu elemente impare cu exact d divizori primi.*
8. *Să se contorizeze elementele pare și cele impare dintr-un vector.*
9. *Să se contorizeze elementele din vector mai mici decât o valoare dată.*
10. *Să se contorizeze elementele din seria lui Fibonacci ce se află într-un vector.*
11. *Să se găsească într-un vector cu n elemente un element cu proprietatea că valoarea lui este egală cu poziția pe care apare, (a[k]=k).*
12. *Să se găsească într-un vector cu n elemente un element cu proprietatea că valoarea lui este egală cu poziția pe care apare cel mai mic element din vector.*
13. *Să se găsească într-un vector cu n elemente un element cu proprietatea că valoarea lui este egală cu poziția pe care apare cel mai mare element din vector.*
14. *Se dă un şir cu n elemente, numere naturale. Să se verifice dacă toate elementele şirului au număr par de cifre. Programul citeşte de la tastatură numărul n, iar apoi cele n elemente ale şirului, separate prin spaţii. Programul afişează pe ecran mesajul DA, dacă toate elementele şirului au număr par de cifre, respectiv NU, în caz contrar.*
15. *Se dă un şir x format din n numere naturale nenule. Pentru fiecare element x_i din şir să se verifice dacă există un număr k astfel încât elementul x_i să fie egal cu suma primelor k elemente din şir. Fişierul de intrare date.in conține pe prima linie numărul n, iar pe a doua linie n numere naturale separate prin spații. Fișierul de ieșire date.out va conține pe linia i valoarea k dacă elementul x_i este egal cu suma primelor k elemente din şir, sau 0 în caz contrar, pentru fiecare i de la 1 la n.*

Probleme cu șiruri de caractere

16. *Să se afișeze în ordine inversă literele unui cuvânt citit de la tastatură.*
17. *Să se afişeze dacă un cuvânt este palindrom.*
18. *Să se afişeze cuvintele care conțin minim 3 vocale.*
19. *Să se afişeze cuvintele care conțin exact v vocale și c consoane.*
20. *Să se afişeze împărțirea pe silabe a cuvintelor care conțin exact 4 vocale.*
21. *Să se afișeze cuvintele care conțin exact k silabe dintr-un text.*
22. *Să se contorizeze numărul de vocale și numărul de consoane dintr-un text.*
23. *Să se contorizeze numărul de apariții a cel puțin două vocale alăturate dintr-un text, indiferent dacă există spațiu între ele.*
24. *Să se contorizeze numărul de apariții a cel puțin două consoane alăturate dintr-un text, indiferent dacă există spațiu între ele.*
25. *Să se contorizeze numărul de apariții pentru diftong, triftong și heat dintr-un text.*

5.5 Implementarea metodei divide et impera la rezolvarea problemelor complexe

Problema 1: Căutare binară

Exemplu
Fie vectorul: 16 27 43 45 49 60 68 81 92 96. Fie că dorim să căutăm în vector numărul 43. ■ *Avem Left = 0, Right = 9, Mid = 4. Dacă înlocuim V[Mid] = 49. Deoarece 49 > 43, Right devine 3.* ■ *Avem Left = 0, Right = 3, Mid = 1. Dacă înlocuim V[Mid] = 27. Deoarece 27 < 43, Left devine 2.* ■ *Avem Left = 2, Right = 3, Mid = 2. Dacă înlocuim V[Mid] = 43. Am obținutd numărul căutat.*

Varianta clasică

```
#include <iostream>
using namespace std;
const int N=10;
int
V[N]={16,27,43,45,49,60,68,81,92,96};
int CautareBinara(int x){
    int Sol=-1,Left=0,Right=N;
    while(Left <= Right){
        int Mid = (Left+Right) / 2;
        if(V[Mid] == x){
            Sol = Mid;
            break;
        }
        if(V[Mid] > x)
            Right = Mid - 1;
        if(V[Mid] < x)
            Left = Mid + 1;
    }
    return Sol;
}
int main(){
    int k=96;
    cout<<"Elementele vectorului
sunt:\n";
    for(int i=0;i<N;i++){
        cout<<V[i]<<" ";
    }
    cout<<"\n\nNumarul cautat \t\t"<<k;
    cout<<"\nSe afla in pozitia \t";
    cout << CautareBinara(k);
    cout<<"\n";
    return 0;
}
```

Rezultatele execuției:

```
Elementele vectorului sunt:
16 27 43 45 49 60 68 81 92 96

Numarul cautat          96
Se afla in pozitia      9
```

Varianta recursivă

```
#include <iostream>
using namespace std;
const int N=10;
int
V[N]={16,27,43,45,49,60,68,81,92,96};
int CBinara(int Left,int Right,int x){
    if(Left > Right) return -1;
    else{
        int Mid =(Left+Right)/2;
        if(x == V[Mid])
        return Mid;
        if(x < V[Mid])
        return CBinara(Left,Mid-1,x);
        else
        return CBinara(Mid+1,Right,x);
    }
}
int main(){
    int k=96;
    cout<<"Elementele vectorului
sunt:\n";
    for(int i=0;i<N;i++){
        cout<<V[i]<<" ";
    }
    cout<<"\n\nNumarul cautat \t\t"<<k;
    cout<<"\nSe afla in pozitia \t";
    cout <<CBinara(0,N-1,k);
    cout<<"\n";
    return 0;
}
```

Rezultatele execuției:

```
Elementele vectorului sunt:
16 27 43 45 49 60 68 81 92 96

Numarul cautat          96
Se afla in pozitia      9
```

Problema 2: Căutare binară uniformă

Condiția problemei	*Exemplu*
Fiind dat un tablou unidimensional de numere întregi și o valoare țintă ce trebuie căutată, utilizați algoritmul de căutare binară uniformă, dacă există o țintă în tablou, imprimați poziția acestuia, evident se va utiliza algoritmul divide et impera.	*Fie vectorul: 2 3 4 10 40 .* *Elementul țintă: 10.* *Soluție:* *Elementul 10 este găsit pe poziția 3.*

Implementare C++

```
#include <iostream>
using namespace std;
const int dimensiune = 1000;
int tabel[dimensiune];
void delta(int n){
    int putere = 1, numar = 0; /// variabilele putere si numar
    do {
        putere <<= 1; /// inmultim cu 2
        /// initializam tabelul de cautare
        tabel[numar] = (n + (putere >> 1)) / putere;
    } while (tabel[numar++] != 0);
}
int cautare_uniforma(int a[], int v){
    int index = tabel[0] - 1; /// punctul din mijlocul al tabloului
    int numar = 0;
    while (1) {
        /// daca valoarea este gasita
        if (v == a[index]){ return index; }
        else if (tabel[numar] == 0){ return -1; }
        else{
        /// daca valoarea este mai mica decat valoarea medie
            if (v < a[index]){ index -= tabel[++numar]; }
        /// daca valoarea este mai mare decat valoarea medie
            else{ index += tabel[++numar]; }
        }
    }
}
int main(){
    int a[] = { 1, 2, 3, 4, 5}; int x,i,n = sizeof(a) / sizeof(int);
    delta(n); /// cream tabelul de cautare
    cout<<"Elementele sirului: \t\t";
    for(i = 0; i < n; ++i){ cout<<a[i]<<"\t"; }
    cout<<"\nPozitiile elementele sirului: \t";
    for(i = 0; i < n; ++i){cout<<i<<"\t"; } cout<<endl;
    /*cout<<"\nPozitia fiecarui element al sirului: \n";
    for (i=1; i <= n; i++){
        cout<<"\tElementul "<<i<<" se afla pe pozitia "<<cautare_uniforma(a,i);
        cout<<endl;
   }*/
   cout<<"\n\nIntrodu elementul care trebuie gasit: \t\t"; cin>>x;
   cout<<"Elementul "<<x<<"  se afla pe pozitia "<<cautare_uniforma(a,x)<<endl;
   return 0;
}
```

Rezultatele execuției:

```
Elementele sirului:             1       2       3       4       5
Pozitiile elementele sirului:   0       1       2       3       4

Introdu elementul care trebuie gasit:           4
Elementul 4  se afla pe pozitia 3
```

Problema 3: Căutare ternară

Condiția problemei	*Exemplu*
Fiind dat un tablou unidimensional de numere întregi și o valoare țintă ce trebuie căutată, utilizați algoritmul de căutare ternară, dacă există o țintă în tablou, imprimați poziția acestuia, evident, se va utiliza algoritmul divide et impera.	*Fie vectorul: 2 3 4 10 40 .* *Elementul țintă: 10.* *Soluție:* *Elementul 10 este găsit pe poziția 3.*

Implementare C++

```
#include <iostream>
using namespace std;
int CautareTernara(int l, int r, int key, int a[]){
    if (r >= l){
        int mid1 = l + (r - l) / 3, mid2 = r - (r - l) / 3;
        if (a[mid1] == key){
            return mid1;
        }
        if (a[mid2] == key){
            return mid2;
        }
        if (key < a[mid1]){
            return CautareTernara(l, mid1 - 1, key, a);
        }
        else if (key > a[mid2]){
            return CautareTernara(mid2 + 1, r, key, a);
        }
        else {
            return CautareTernara(mid1 + 1, mid2 - 1, key, a);
        }
    }
    return -1;
}
int main(){
    int l, r, a[] = { 1, 3, 5, 6, 7 };
    int lungime = sizeof(a) / sizeof(a[0]);
    l = 0; r = 6;
    cout<<"\nVectorul introdus este:\t";
    for(int i=0;i<lungime;i++){
        cout<<a[i]<<" ";
    }
    cout<<"\nPozitiile fiecarui element din vector:\n";
    for(int i=0;i<=lungime;i++){
        cout << "\tPozitia numarului " << i<< " este: ";
        cout<< CautareTernara(l, r, i, a) << endl;
    }
}
```

Rezultatele execuției:

```
Vectorul introdus este: 1 3 5 6 7
Pozitiile fiecarui element din vector:
        Pozitia numarului 0 este: -1
        Pozitia numarului 1 este: 0
        Pozitia numarului 2 este: -1
        Pozitia numarului 3 este: 1
        Pozitia numarului 4 este: -1
        Pozitia numarului 5 este: 2
```

Problema 4: Căutare Fibonacci

Condiția problemei	*Exemplu*
Fiind dat un tablou unidimensional de numere întregi și o valoare țintă ce trebuie căutată, utilizați algoritmul de căutare Fibonacci, dacă există o țintă în tablou, imprimați poziția acestuia, evident se va utiliza algoritmul divide et impera.	*Fie vectorul: 2 3 4 10 40 .* *Elementul țintă: 10.* *Soluție:* *Elementul 10 este găsit pe poziția 3.*

Implementare C++

```
#include <iostream>
using namespace std;
int min(int x, int y) {
    return (x<=y)? x : y;
}
int Fibonacci(int a[], int x, int n){
    int F1 = 1, F2 = 0, F = F2 + F1;
    while (F < n){ F2 = F1; F1 = F;  F = F2 + F1; }
    int offset = -1;
    while (F > 1) {
        int i = min(offset + F2, n-1);
        if (a[i] < x) { F  = F1; F1 = F2; F2 = F - F1; offset = i; }
        else if (a[i] > x) { F  = F2; F1 = F1 - F2; F2 = F - F1; }
        else return i;
    }
    if(F1 && a[offset+1]==x)return offset+1;
    return -1;
}
int main(){
    int a[] = { 1, 2, 3, 4, 5 }, n = sizeof(a)/sizeof(a[0]);
    cout<<"\nVectorul introdus:\t";
    for(int i=0;i<n;i++){ cout<<a[i]<<" "; }
    cout<<"\nPozitiile fiecarui element din vector:\n";
    for(int i=0;i<=n;i++){
        cout << "\tPozitia numarului " << i<< " este: ";
    cout<< Fibonacci(a, i, n) << endl;
    }
}
```

Rezultatele execuției:

```
Vectorul introdus este: 1 2 3 4 5
Pozitiile fiecarui element din vector:
        Pozitia numarului 0 este: -1
        Pozitia numarului 1 este: 0
        Pozitia numarului 2 este: 1
        Pozitia numarului 3 este: 2
        Pozitia numarului 4 este: 3
        Pozitia numarului 5 este: 4
```

Notă:

- ✓ **Diferențe față de căutarea binară:**
  - *Căutarea Fibonacci împarte tabloul dat în părți inegale.*
  - *Căutarea binară folosește operatorul de divizare pentru a împărți intervalul. Căutarea Fibonacci nu utilizează /, ci folosește + și -. Operatorul de divizie poate fi costisitor pe unele procesoare.*
  - *Căutarea Fibonacci examinează elemente relativ mai apropiate în etapele următoare. Deci, atunci când matricea de intrare este mare, care nu se pot încadra în memoria cache CPU sau chiar în memoria RAM, Fibonacci Search poate fi util.*

Problema 5: Căutare prin interpolare

Exemplu
Fie vectorul: 16 27 43 45 49 60 68 81 92 96. Fie că dorim să căutăm în vector numărul 43. ■ *m=1+((10-1)*(43-16))/(96-16)=1+(9*27)/80=1+3.03=4.03* *deoarece 43 < v[4], adică 43<45* ■ *m=1+((9-1)*(43-16))/(92-16)=1+(8*27)/76=1+2.84=3.84* *deoarece 43 = v[3], adică 43=43. Răspuns poziția 3* *Fie vectorul: 16 27 43 45 49 60 68 81 92 96. Fie că dorim să căutăm în vector numărul 60.* ■ *m=1+((10-1)*(60-16))/(96-16)=1+(9*44)/80=1+4.95=5.95* *deoarece 60 > v[5], adică 60>49* ■ *m=2+((10-1)*(60-27))/(96-27)=2+(9*33)/69=2+4.30=6.30* *deoarece 60 = v[6], adică 60=60. Răspuns poziția 6*

Varianta clasică

```
#include <iostream>
using namespace std;
int interpolare(int V[],int n,int x){
    int lo = 0, hi = (n - 1);
    while (lo<=hi && x>=V[lo] &&
x<=V[hi]){
        if (lo == hi){
            if (V[lo] == x) return lo;
            return -1;
        }
        int k=(hi-lo)/(V[hi]-V[lo]);
        int pos=lo+(k*(x-V[lo]));
        if (V[pos] == x)
            return pos;
        if (V[pos] < x)
            lo = pos + 1;
        else
            hi = pos - 1;
    }
    return -1;
}
int main(){
    int
V[]={16,27,43,45,49,60,68,81,92,96};
    int n = sizeof(V)/sizeof(V[0]);
    int x = 49;
    cout<<"\nElementele sirului: \n";
    for(int i = 0; i < n; ++i){
        cout<<V[i]<<" ";
    }
    cout<<"\nElementele indicilor: \n";
    for(int i = 0; i < n; ++i){
        cout<<i<<"  ";
    }
    int index = interpolare(V, n, x);
    if (index != -1)
        cout<<"\nElementul "<<x<<" este
gasit pe pozitia: \n"<<index<<endl;
    else
        cout<<"\nElementul "<<x<<" nu
este gasit! "; }
```

Rezultatele execuției:

```
Elementele sirului:
16 27 43 45 49 60 68 81 92 96
Elementele indicilor:
0  1  2  3  4  5  6  7  8  9
Elementul 49 este gasit pe pozitia:4
```

Varianta recursivă

```
#include <iostream>
#include <vector>
using namespace std;
int CautInterp(int V[10], int e, int ic,
int sf){
      if (ic <= sf){
            int m;
        m = ic+(e-V[ic])*(sf-ic)/(V[sf]-
V[ic]);
            if (e == V[m])
                  return m + 1;
            else
            if (e<V[m])
           return CautInterp(V, e, ic,
m - 1);
            else
                  return CautInterp(V,
e, m + 1, sf);
      }
      else return 0;
}
int main(){
      int
V[]={16,27,43,45,49,60,68,81,92,96};
    int n = sizeof(V)/sizeof(V[0]);
    int x = 43,p;
    cout<<"\nElementele sirului: \n";
    for(int i = 0; i <n; i++){
        cout<<V[i]<<" ";
    }
    cout<<"\nElementele indicilor: \n";
    for(int i = 1; i <= n; i++){
        cout<<i<<"  ";
    }
      cout<<endl;
      if (p = CautInterp(V, x, 0, n -
1))
            cout<<"Elementul "<<x<<"
este gasit pe pozitia: "<<p;
      else cout<<"Elementul "<<x<<" nu
este gasit!"; }
```

Rezultatele execuției:

```
Elementele sirului:
16 27 43 45 49 60 68 81 92 96
Elementele indicilor:
1  2  3  4  5  6  7  8  9  10
Elementul 49 este gasit pe pozitia:5
```

Problema 6: Căutare număr lipsă

Condiția problemei	*Exemplu*
Fie că avem un șir de numere întregi n-1 și aceste numere întregi sunt în intervalul de la 1 la n. Nu există duplicate în listă. Unul dintre numerele întregi lipsește din listă. Scrieți un cod eficient pentru a găsi numărul întreg care lipsește, evident se va utiliza algoritmul divide et impera.	*Fie vectorul: 1 2 3 4 5 6 8 9 10.* *Soluție:* *Elementele vectorului sunt cuprinse în intervalul [1,10].* *Ca rezultat vom obține elementul lipsă =7*

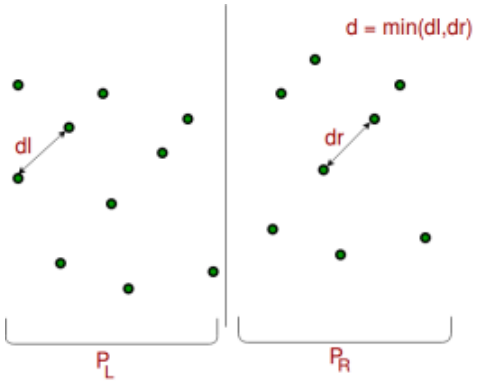
Implementare C++

```
#include <iostream>
using namespace std;
int DEI(int arr[], int lungime){
    int a = 0, b = lungime - 1;
    int mid;
    while ((b - a) > 1){
        mid = (a + b) / 2;
        if ((arr[a] - a) != (arr[mid] - mid))
            b = mid;
        else if ((arr[b] - b) != (arr[mid] - mid))
            a = mid;
    }
    return (arr[mid] + 1);
}
int main(){
    ///Vectorul 1
    int arr1[] = { 1, 2, 3, 4, 5, 6, 8, 9, 10 };
    int lungime1 = sizeof(arr1) / sizeof(arr1[0]);
    cout<<"\nVectorul 1:\t";
    for(int i=0;i<lungime1;i++){
        cout<<arr1[i]<<" ";
    }
    cout << "\nNumarul lipsa: \t" << DEI(arr1, lungime1);
    ///Vectorul 2
    int arr2[] = { 1, 2, 4, 5, 6, 7, 8, 9, 10 };
    int lungime2 = sizeof(arr2) / sizeof(arr2[0]);
    cout<<"\n\nVectorul 2:\t";
    for(int i=0;i<lungime2;i++){
        cout<<arr2[i]<<" ";
    }
    cout << "\nNumarul lipsa: \t" << DEI(arr2, lungime2);
}
```

Rezultatele execuției:

```
Vectorul 1:     1 2 3 4 5 6 8 9 10
Numarul lipsa:  7

Vectorul 2:     1 2 4 5 6 7 8 9 10
Numarul lipsa:  3
```

Problema 7: Cea mai apropiată pereche de puncte

Condiția problemei	*Exemplu*
Fie că avem o mulțime de n puncte în plan, iar problema este să aflăm cea mai apropiată pereche de puncte. Această problemă apare într-o serie de aplicații. De exemplu, în controlul traficului aerian, dacă dorim să monitorizăm avioanele care se apropie prea mult, deoarece acest lucru poate indica o posibilă coliziune. La rezolvarea problemei se va lucra cu clasa Point (Punct).	

Implementare C++

```
#include <bits/stdc++.h>
using namespace std;
class Point {
    public: int x, y;
};
int compareX(const void* a, const void* b){
    Point *p1=(Point *)a, *p2=(Point *)b; return (p1->x-p2->x); }
int compareY(const void* a, const void* b){
    Point *p1=(Point *)a, *p2=(Point *)b; return (p1->y-p2->y); }
float dist(Point p1, Point p2){
    return sqrt((p1.x-p2.x)*(p1.x-p2.x)+(p1.y-p2.y)*(p1.y-p2.y)); }
float bruteForce(Point P[], int n){
    float minim = FLT_MAX;
    for (int i = 0; i < n; ++i) for (int j = i+1; j < n; ++j)
    if (dist(P[i],P[j])<minim) minim=dist(P[i],P[j]); return minim; }
float minim(float x, float y){
    return (x < y)? x : y; }
float stripClosest(Point strip[], int size, float d){
    float minim = d; qsort(strip, size, sizeof(Point), compareY);
    for (int i = 0; i < size; ++i)
    for (int j=i+1; j<size && (strip[j].y-strip[i].y)<minim; ++j)
    if (dist(strip[i],strip[j])<minim)
        minim=dist(strip[i],strip[j]); return minim; }
float closestUtil(Point P[], int n){
    if (n <= 3) return bruteForce(P, n);
    int mid = n/2; Point midPoint = P[mid];
    float dl=closestUtil(P,mid); float dr = closestUtil(P+mid, n-mid);
    float d = min(dl, dr);  Point strip[n]; int j = 0;
    for (int i = 0; i < n; i++)
        if (abs(P[i].x - midPoint.x) < d) strip[j] = P[i], j++;
    return min(d, stripClosest(strip, j, d) ); }
float closest(Point P[], int n){
    qsort(P, n, sizeof(Point), compareX); return closestUtil(P, n); }
int main(){
    Point P[] = {{2, 3}, {12, 30}, {40, 50}, {5, 1}, {12, 10}, {3, 4}};
    int n=sizeof(P)/sizeof(P[0]);
    cout<<"Distanta minima este: "<<closest(P,n);
}
```

Rezultatele execuției:

```
Distanta minima este: 1.41421
```

Problema 8: Algoritmul Karatsuba pentru multiplicare rapidă

Condiția problemei	*Exemplu*
Fie că avem două șiruri binare care reprezintă valoarea a două numere întregi. Cerința este de a găsi produsul celor două șiruri. De exemplu, dacă primul șir de biți este „1100” și al doilea șir de biți este „1010”, ieșirea ar trebui să fie 120. Pentru simplitate, lungimea a două șiruri să fie aceeași și să fie n.	*Fie avem numerele m=1010001 și n=1010010. Trebuie să inmulțim aceste numere.* *Soluție:* 1010001 =81 1010010 =82 1010010 =82 + 1010001 =81 1010001 =81 11010111010 =1722

Implementare C++

```
#include<iostream>
using namespace std;
int Lung_egale(string &str1, string &str2){
    int len1=str1.size(), len2=str2.size();
    if (len1<len2){
        for(int i=0;i<len2-len1;i++) str1='0'+str1;
        return len2;
    }
    else if (len1>len2){
        for (int i=0;i<len1-len2;i++) str2='0'+str2;
    }   return len1;
}
string suma( string primul, string al_doilea){
    string result;
    int length = Lung_egale(primul, al_doilea),transporta = 0;
    for (int i = length-1 ; i >= 0 ; i--){
        int primulBit=primul.at(i) - '0';
        int al_doileaBit = al_doilea.at(i) - '0';
        int sum=(primulBit ^ al_doileaBit ^ transporta)+'0';
        result=(char)sum + result;
        transporta=(primulBit&al_doileaBit) | (al_doileaBit&transporta) |
(primulBit&transporta);
    }
    if (transporta)  result = '1' + result;
    return result;
}
int multiplyiSingleBit(string a, string b){
    return (a[0] - '0')*(b[0] - '0');
}
long int multiply(string X, string Y){
    int n = Lung_egale(X, Y); int fh = n/2, sh = (n-fh);
    if (n == 0) return 0;
    if (n == 1) return multiplyiSingleBit(X, Y);
    string Xl = X.substr(0, fh); string Xr = X.substr(fh, sh);
    string Yl = Y.substr(0, fh); string Yr = Y.substr(fh, sh);
    long int P1 = multiply(Xl, Yl), P2 = multiply(Xr, Yr);
    long int P3 = multiply(suma(Xl, Xr), suma(Yl, Yr));
    return P1*(1<<(2*sh)) + (P3 - P1 - P2)*(1<<sh) + P2;
}
int main(){
    string m="1000", n="1000";
    cout<<"\nProdusul dintre numerele binare: \t"<<m<<" si "<<n;
    cout<<"\nRezultatul este numarul zecimal: \t"<<multiply(m, n);
}
```

Rezultatele execuției:

```
Produsul dintre numerele binare:        1000 si 1000
Rezultatul este numarul zecimal:        64
```

Problema 9: Valoarea XXX…. (N ori) % M

Condiția problemei	*Exemplu*
Fie că avem trei numere întregi X, N și M. Sarcina este de a găsi XXXX ... (N ori) % M unde X poate fi orice cifră din intervalul [1, 9].	*Fie avem numerele X=7, N=7 și M=50. Deci trebuie să găsim soluția pentru: 7777777 % 50* *Soluție:* `7777777 % 50 = 27`

Implementare C++

```
#include <iostream>
#include <sstream>
using namespace std;
int Putere(int x, unsigned int y, int p){
    int rez = 1;
    x = x % p;
    while (y > 0) {
        if (y & 1)
        rez=(rez*x)%p; y=y>>1; x=(x*x)%p;
    }   return rez;
}
int DEI(int X, int N, int M){
    if (N < 6) {
        string temp(N, (char)(48 + X));
        stringstream degree(temp);
        int z=0; degree>>z; int rez = z % M;
        return rez;
    }
    if (N % 2 == 0) {
        int half=DEI(X,N/2,M)%M; int rez=(half*Putere(10,N/2,M)+half)%M;
        return rez;
    }
    else {
        int half = DEI(X,N/2,M)%M;
        int rez=(half*Putere(10,N/2+1,M)+half*10+X)%M;
        return rez;
    }
}
int exp(int b,int p){
    int result = 1;
    for(int i = 1; i <= p; i++){
        result = result * b;
    }   return result;
}
long long numar(int X,int N){
    int i,s=X;
    for(i=1;i<N;i++){
            s+=X*exp(10,i);
    }   return s;
}
int main(){
    /// Conditia de baza: N < 10
    int X = 7, N = 5, M = 50;
    cout<<"Numarul format din "<<N<<" cifre de "<<X<<" este: "<<numar(X,N);
    cout<<"\n\nNumarul "<<numar(X,N)<<" % "<<M<<" este: "<<DEI(X, N, M)<<endl;
}
```

Rezultatele execuției:

```
Numarul format din 5 cifre de 7 este: 77777

Numarul 77777 % 50 este: 27
```

Problema 10: Căutare element într-o matrice sortată sub formă de spirală

Condiția problemei	*Exemplu*
Fie că avem o matrice NxN de elemente întregi sortată și un număr întreg X. Sarcina este de a găsi poziția acestui număr întreg dat în matrice dacă există, altfel tipăriți -1. Rețineți că toate elementele matricei sunt distincte aranjate sub formă de spirală.	*Fie avem numerele N=3 și X=7. Matricea 3x3 are elementele sortate sub formă circulară:* 1 2 3 8 9 4 7 6 5 *Soluție:* Numărul X=7 va avea poziția (2,0)în matrice

Implementare C++

```
#include <iostream>
#define n 3
using namespace std;
int cauta(int arr[][n], int x){
    if (arr[0][0]>x) return -1;
    int l=0, r=(n+1)/2-1;
    if (n%2==1 && arr[r][r]<x) return -1;
    if (n%2==0 && arr[r+1][r]<x) return -1;
    while (l<r) {
        int mid=(l+r)/2;
        if (arr[mid][mid] <= x)
        if (mid==(n+1)/2-1 || arr[mid+1][mid+1]>x) return mid;
        else l = mid + 1;
        else r = mid - 1;
    }   return r;
}
/// cazul de cautare cand exista sortare crescatoare
int cautareRow(int arr[][n],int row,int l,int r,int x){
    while (l<=r){
        int mid=(l+r)/2;
        if (arr[row][mid] == x) return mid;
        if (arr[row][mid] < x) l = mid + 1;
        else r = mid - 1;
    }   return -1;
}
int cautareColumn(int arr[][n],int col,int t,int b,int x){
    while (t<=b){
        int mid=(t+b)/2;
        if (arr[mid][col]==x) return mid;
        if (arr[mid][col]<x) t=mid+1;
        else b=mid-1;
    }   return -1;
}
/// cazul de cautare cand exista sortare descrescatoare
int cautareRow2(int arr[][n],int row,int l,int r,int x){
    while (l<=r){
        int mid=(l+r)/2;
        if (arr[row][mid]==x) return mid;
        if (arr[row][mid]<x) r=mid-1;
        else l = mid + 1;
    }   return -1;
}
int cautareColumn2(int arr[][n],int col,int t,int b,int x){
    while (t<=b){
        int mid=(t+b)/2;
        if (arr[mid][col]==x) return mid;
        if (arr[mid][col]<x) b=mid-1;
        else t=mid+1;
    }   return -1;
}
void spiralBinary(int arr[][n],int x){
    int f1=cauta(arr,x); int r,c;
    if (f1==-1){
        cout << "-1"; return;
```

```
    }
    if (n%2==1 && f1==(n+1)/2-1) {
        cout<<f1<<" "<<f1<<endl; return;
    }
    if (x<arr[f1][n-f1-1]){
        c=cautareRow(arr,f1,f1,n-f1-2,x); r=f1;
    }
    else if (x<arr[n-f1-1][n-f1-1]){
        c=n-f1-1;
        r=cautareColumn(arr,n-f1-1,f1,n-f1-2,x);
    }
    else if (x<arr[n-f1-1][f1]){
        c = cautareRow2(arr,n-f1-1,f1+1,n-f1-1,x);
        r = n-f1-1;
    }
    else {
        r = cautareColumn2(arr,f1,f1+1,n-f1-1,x);
        c = f1;
    }
    if (c==-1 || r==-1) cout << "-1";
    else cout << r << " " << c;
    return;
}
int main() {
    int arr[][n] = { {1,2,3},
                     {8,9,4},
                     {7,6,5}};
    int k=(sizeof(arr)/sizeof(arr[0][0]))/n;
    int i,j, cautat;
    cout<<"Introduceti numarul ce trebuie de cautat in matrice: \t"; cin>>cautat;
    cout<<"Matricea sortata sub forma de spirala este: \n\t";
    for(i=0;i<k;i++){
        for(j=0;j<k;j++)
            cout<<arr[i][j]<<"\t"; cout<<endl<<"\t";
    }
    cout<<"\nPozitiile alementelor in matrice incep cu a[0][0].";
    cout<<"\nPozitia numarului "<<cautat<<" este: ";
    spiralBinary(arr, cautat);
    return 0;
}
```

Rezultatele execuției:

```
Introduceti numarul ce trebuie de cautat in matrice:        7
Matricea sortata sub forma de spirala este:
        1       2       3
        8       9       4
        7       6       5

Pozitiile alementelor in matrice incep cu a[0][0].
Pozitia numarului 7 este: 2 0
```

SARCINI PENTRU EXERSARE

1. *Se dau n-1 numere distincte de la 1 la n, să se scrie un algoritm cât mai eficient care să determine numărul lipsă.*
2. *Se dă un șir de n+1 numere de la 1 la n în care unul se repetă, iar restul sunt distincte. Să se propună un algoritm cât mai eficient care să determine numărul ce se repetă.*
3. *Se dă o listă înlănțuită prin primul ei element. Se cere un algoritm cât mai eficient care să determine dacă lista are sau nu ciclu.*
4. *Un șir de lungime n conține numere întregi din mulțimea {1, 2, ..., n-1}. Folosind Principiul lui Dirichlet deducem că cel puțin un element se repetă. Să se scrie un algoritm liniar care afișează o valoare ce se repetă, folosind memorie suplimentară constantă și nemodificând la nici un pas vre-un element din șir* ***[21]****.*
5. *Se dă un șir de N întregi ce conține numere între 1 și N. Să se determine cât mai eficient dacă acest număr este o permutare, fără a distruge șirul.*
6. *Se dau n numere de la 1 la n. Unul din ele apare de două ori în șir, iar restul sunt distincte. Evident că un număr nu va apărea niciodată. Să se scrie un algoritm cât mai eficient care să determine numărul lipsă și numărul ce apare de două ori.*
7. *Într-o structură de date avem n - 1 numere întregi (pentru simplitate n = 2*b - 1, cu numere distincte de la 0 la n). Asupra elementelor din structura de date putem face următoarea operație getBit(i, j), care returnează al j-lea bit din reprezentarea binară a nuărului a[i]. Astfel, dacă a[4] = 11, atunci getBit(4, 3) returneaza 0, pentru că 11 se scrie în baza 2 ca 1011. Să se scrie o soluție eficientă care găsește numărul lipsă.*
8. *Se dau n numere întregi, astfel încât fiecare număr apare de un număr par de ori, în afară de unul singur, care apare de un număr impar de ori. Se cere determinarea acelui număr. De exemplu: în șirul 1 2 2 3 1 2 2 2 3 3 3, numai elementul 2 apare de un număr impar de ori.*
9. *Se dau n șiruri formate din cifre de la 1 la 8 (inclusiv), de lungime maximă, 500. Toate șirurile sunt repetate de k ori sau de multiplu de k ori, cu excepția unui singur șir, care nu este repetat de k ori, sau de multiplu de k ori. Să se afișeze șirul care nu se repetă. Restricții: k < 2005, 1<= n <= 32000* ***[26]****.*
10. *Se dă un șir de n numere naturale, în care orice element apare de exact k ori, mai puțin n%k dintre acestea care, de asemenea, sunt egale. Să se determine valoarea elementelor care apar de exact n%k ori. Se garantează că n % k >0.*
11. *Fie avem o matrice sortată de dimensiunea NxN ce conține elemente întregi și un număr întreg X. Sarcina este de a găsi poziția acestui număr întreg din matrice dacă există, altfel tipăriți -1. Rețineți că toate elementele matricei sunt distincte aranjate sub formă de spirală:*

Cazul A	Cazul B	Cazul C	Cazul D
N=3 1 2 3 8 9 4 7 6 5	N=3 3 2 1 4 9 8 5 6 7	N=3 1 8 7 2 9 6 3 4 5	N=3 7 8 1 6 9 2 5 4 3

12. *Fie avem o matrice sortată NxN, de elemente întregi și un număr întreg X. Sarcina este de a găsi poziția acestui număr întreg din matrice dacă există, altfel tipăriți -1. Rețineți că toate elementele matricei sunt distincte aranjate sub formă de zigzag (șerpuită):*

Cazul A	Cazul B	Cazul C	Cazul D
N=3 1 2 3 6 5 4 7 8 9	N=3 3 2 1 4 5 6 9 8 7	N=3 1 6 7 2 5 8 3 4 9	N=3 7 6 1 8 5 2 9 4 3

5.6 Implementarea metodei divide et impera la rezolvarea problemelor complexe

Problema 1: Problema plierii

Condiția problemei

Se consideră un vector de lungime n. Definim plierea vectorului prin suprapunerea unei jumătăți, numită donatoare, peste cealaltă jumătate, numită receptoare, cu precizarea, dacă numărul de elemente este impar, elementul din mijloc este eliminat. Prin pliere, elementele subvectorului obținut vor avea numerotarea jumătății receptoare. Plierea se poate aplica în mod repetat, până când se ajunge 1a un subvector format dintr-un singur element, numit element final. Scrieți un program care să afișeze toate elementele finale posibile și să se figureze (afișeze) pe ecran pentru fiecare element final o succesiune de plieri.

Indicații de rezolvare

Pentru a determina toate elementele finale și succesiunile de plieri corespunzătoare unui vector cu numerotarea elementelor p... q, vom utiliza funcția Pliaza (p, q). Dacă p=q, atunci vectorul este format dintr-un singur element, deci element final. Dacă p<q, calculăm numărul de elemente din vector (q-p+l). Dacă numărul de elemente din vector este impar, elementul din mijloc ((p+q) /2) este eliminat, plierea la stânga se face de la poziția (p+q) /2-1, iar plierea la dreapta de la poziția (p +q) /2+1. Dacă numărul de elemente din vector este par, plierea la stanga se poate face de la poziția (p+q) /2, iar plierea la dreapta de la poziția (p+q) /2+1. Vom codifica plierea la stânga reținând în șirul mutărilor simbolul "S", urmat de poziția Ls, de 1a care se face plierea spre stânga, iar o pliere la dreapta prin simbolul "D", urmat de poziția Ld de 1a care se face plierea spre dreapta. Pentru a determina toate elementele finale din lintervalul p...q apelăm recursiv funcția Pliaza() pentru prima jumătate a intervalului, precedând toate succesiunile de plieri cu o pliere spre stânga, apoi apelăm funcția Pliaza () pentru cea de-a doua jumătate a intervalului, precedând toate succesiunile de plieri corespunzătoare cu o pliere 1a dreapta.

Implementare C++

```
#include <iostream>
#include <string.h>
#include <stdlib.h>
#define NMax 50
using namespace std;
int n,efinal[NMax];
char m[NMax][50];
void Pliaza(int p, int q){
    int Ls, Ld, i;
    char ss[50], sd[50], aux[50];
    if (p==q) efinal[p]=1;
    else{
        if ((q-p+1)%2) Ls=(p+q)/2-1;
        else Ls=(p+q)/2;
        Ld=(p+q)/2+1;
        Pliaza(p,Ls);
        itoa(Ls, ss, 10);
        itoa(Ld, sd, 10);
        for (i=p; i<=Ls; i++){
            strcpy(aux,"S");
            strcat(aux,ss);
            strcat(aux," ");
            strcat(aux,m[i]);
            strcpy(m[i],aux);
        }
        Pliaza (Ld, q);
        for (i=Ld; i<=q; i++){
            strcpy(aux,"D");
```

```
            strcat(aux,sd);
            strcat(aux ," ");
            strcat (aux,m[i]);
            strcpy(m[i],aux);
        }
    }
}
int main(){
    cout<<"Introduceti numarul de gauri, n= "; cin>>n;
    Pliaza(1,n);
    cout<<"Elementele finale sunt: "<<endl;
    for(int i=1;i<=n;i++)
    if(efinal[1]) cout<<"\t"<<i<<": \t"<<m[i]<<endl;
    return 0;
}
```

Rezultatele execuției:

```
Introduceti numarul de gauri, n= 35
Elementele finale sunt:
        1:      S17 S8 S4 S2 S1
        2:      S17 S8 S4 S2 D2
        3:      S17 S8 S4 D3 S3
        4:      S17 S8 S4 D3 D4
        5:      S17 S8 D5 S6 S5
        6:      S17 S8 D5 S6 D6
        7:      S17 S8 D5 D7 S7
        8:      S17 S8 D5 D7 D8
        9:      S17
        10:     S17 D10 S13 S11 S10
        11:     S17 D10 S13 S11 D11
        12:     S17 D10 S13 D12 S12
        13:     S17 D10 S13 D12 D13
        14:     S17 D10 D14 S15 S14
        15:     S17 D10 D14 S15 D15
        16:     S17 D10 D14 D16 S16
        17:     S17 D10 D14 D16 D17
        18:
        19:     D19 S26 S22 S20 S19
        20:     D19 S26 S22 S20 D20
        21:     D19 S26 S22 D21 S21
        22:     D19 S26 S22 D21 D22
        23:     D19 S26 D23 S24 S23
        24:     D19 S26 D23 S24 D24
        25:     D19 S26 D23 D25 S25
        26:     D19 S26 D23 D25 D26
        27:     D19
        28:     D19 D28 S31 S29 S28
        29:     D19 D28 S31 S29 D29
        30:     D19 D28 S31 D30 S30
        31:     D19 D28 S31 D30 D31
        32:     D19 D28 D32 S33 S32
        33:     D19 D28 D32 S33 D33
        34:     D19 D28 D32 D34 S34
        35:     D19 D28 D32 D34 D35
```

Problema 2: Problema foaia de tablă

Condiția problemei

O placă de tablă dreptunghiulară, de dimensiuni LxH , are n găuri, în puncte de coordonate întregi. Să se decupeze din placă o bucată de arie maximă (sau minimă), dreptunghiulară și fără găuri, știind că sunt permise doar tăieturi pe direcții paralele cu laturile plăcii.

Indicații de rezolvare

Pentru a identifica o zonă dreptunghiulară având laturile paralele cu laturile plăcii este suficient să reținem coordonatele colțului din stânga-jos (xs,ys) și cele două dimensiuni: L și H. Funcția Taie (xs,ys,1,h) determină zonele dreptunghiulare fără găuri ce se pot obține prin tăieturi duse prin găurile plăcii cu colțul stânga-jos (xs,ys) și dimensiunile l, h pe direcții paralele cu laturile plăcii, retinând în variabilele globale: Amax, xmax, ymax, LMax, HMax, aria maxima, respectiv coordonatele unei plăci dreptunghiulare, fără găuri, de arie maximă. Pentru aceasta, funcția caută o gaură din interiorul plăcii de coordonate (xs, ys, l, h). Dacă o astfel de gaură nu există, placa dreptunghiulară de arie maximă fără găuri este chiar (x s , ys, 1, h) și o comparăm cu maximul global. Dacă a fost găsită o gaură de coordonate (x_i, y_i) în interiorul plăcii (xa , ys, 1, h), reducem problema la patru subprobleme: determinarea unei zone dreptunghiulare de arie maximă, fără găuri, ce se poate obține din cele două plăci obținute printr-o tăietură orizontală prin gaura (x_i, y_i), respectiv cele două placi obținute printr-o tăietura verticală prin gaura (x_i, y_i).

Implementare C++

```
#include <iostream>
#define NMaxGauri 20
using namespace std;
int L,H,n,Amax,lmax,hmax,xmax,ymax;
int ox[NMaxGauri],oy[NMaxGauri];
void citire(){
    cout<<"Introduceti numarul de gauri: \t";cin>>n;
    cout<<"\nIntroduceti coordonatele celor "<<n<<" gauri: \n";
    for (int i=0; i<n; i++){
        cout<<"x["<<i<<"]= "; cin>>ox[i];
        cout<<"y["<<i<<"]= "; cin>>oy[i];
    }
    cout<<"Introduceti lungimea: \t";cin>>L;
    cout<<"Introduceti inaltimea: \t";cin>>H;
}
int interior(int i, int xs, int ys, int l, int h){
    if ((ox[i]>xs) && (ox[i]<xs+l) && (oy[i]>ys) && (oy[i]<ys+h))
        return 1;
    return 0;
}
void taiere(int xs, int ys, int l, int h){
    int i=0;
    while ((i<n) && !interior(i,xs,ys,l,h)) i++;
    if (i==n){
        if (l*h>Amax)
            Amax=l*h, lmax=l, hmax=h, xmax=xs, ymax=ys;
    }
    else{

        /// taieturi pe orizontala
        taiere(xs,ys,l,oy[i]-ys);
        taiere(xs,oy[i],l,ys+h-oy[i]);

        /// taieturi pe verticala
        taiere(xs, ys, ox[i]-xs, h);
        taiere(ox[i], ys, xs + l - ox[i], h);
    }
}
int main(){
    citire();
    taiere(0,0,L,H);
```

```
    cout<<endl;
    cout<<"Placa de arie maxima are coltul stanga-jos (";
    cout<<xmax<<", "<<ymax<<") \n";
    cout<<"Lungimea = "<<lmax<<" si inaltimea = "<<hmax<<endl;
    cout<<"Aria dreptunghiului este: \t"<<lmax*hmax<<endl;
    return 0;
}
```

Rezultatele execuției:

```
Introduceti numarul de gauri:   10

Introduceti coordonatele celor 10 gauri:
x[0]= 2
y[0]= 3
x[1]= 4
y[1]= 6
x[2]= 5
y[2]= 4
x[3]= 7
y[3]= 3
x[4]= 8
y[4]= 6
x[5]= 3
y[5]= 4
x[6]= 5
y[6]= 7
x[7]= 6
y[7]= 5
x[8]= 5
y[8]= 6
x[9]= 6
y[9]= 4
Introduceti lungimea:   10
Introduceti inaltimea:  7

Placa de arie maxima are coltul stanga-jos (0, 0)
Lungimea = 10 si inaltimea = 3
Aria dreptunghiului este:       30
```

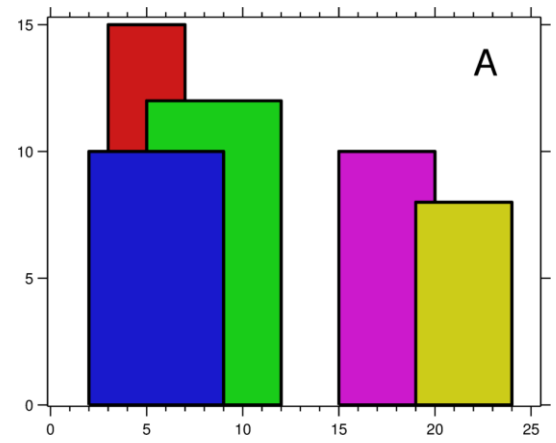
Problema 3: Problema orizont

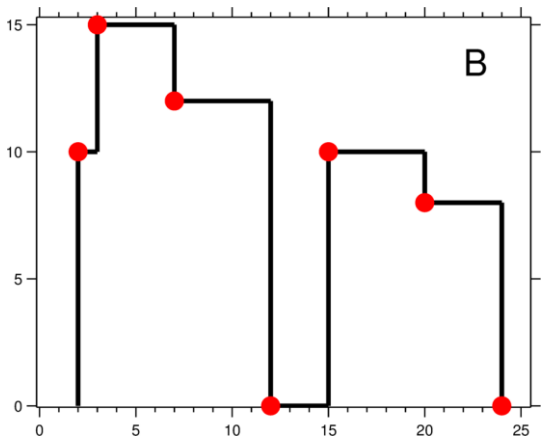
Condiția problemei

Fie avem n clădiri dreptunghiulare într-un oraș bidimensional, calculează orizontul acestor clădiri, eliminând liniile ascunse. Sarcina principală este să vizualizați clădirile dintr-o parte și să eliminați toate secțiunile care nu sunt vizibile. Toate clădirile au partea de jos comună și fiecare clădire este reprezentată de triplete (stânga, h, dreapta), unde: „Stânga": este coordonata din partea stângă, „Dreapta": este coordonata din partea dreaptă și „H": este înălțimea clădirii. Un orizont este o colecție de benzi dreptunghiulare. O bandă dreptunghiulară este reprezentată ca o pereche (stânga, h) unde stânga este coordonata părții stângi a benzii și h este înălțimea benzii.

Indicații de rezolvare

Orizontul unui oraș este conturul exterior al siluetei formate de toate clădirile din oraș atunci când sunt privite de la distanță. Acum, să presupunem că vi se oferă locațiile și înălțimea tuturor clădirilor, așa cum se arată pe o fotografie de peisaj urban (Figura A), scrieți un program pentru a scoate orizontul format de aceste clădiri în mod colectiv (figura B).





Implementare C++

```
#include<iostream>
using namespace std;
struct cladire {
    int stanga, ht, dreapta;
};
class banda {
    int stanga, inaltime;
public:
    banda(int l=0, int h=0) {
        stanga = l; inaltime = h;
    }
    friend class Orizont;
};
class Orizont {
    banda *arr;
    int capacitate, n;
public:
    ~Orizont() {  delete[] arr;  }
    int count()  { return n;   }
    Orizont* Imbina(Orizont *other);
    Orizont(int cap){
        capacitate = cap; arr = new banda[cap]; n = 0;
    }
    void adauga(banda *st){
        if (n>0 && arr[n-1].inaltime == st->inaltime) return;
        if (n>0 && arr[n-1].stanga == st->stanga) {
            arr[n-1].inaltime = max(arr[n-1].inaltime, st->inaltime);
```

```
            return;
        }
        arr[n] = *st; n++;
    }
    void afisare() {
        for (int i=0; i<n; i++) {
            cout<<"\t(" <<arr[i].stanga<<","<<arr[i].inaltime<<")\n";
        }
    }
};
Orizont *Gaseste(cladire arr[], int l, int h){
    if (l == h){
        Orizont *res = new Orizont(2);
        res->adauga(new banda(arr[l].stanga, arr[l].ht));
        res->adauga(new banda(arr[l].dreapta, 0));
        return res;
    }
    int mid = (l + h)/2;
    Orizont *sl = Gaseste(arr, l, mid); Orizont *sr = Gaseste(arr, mid+1, h);
    Orizont *res = sl->Imbina(sr); delete sl; delete sr; return res;
}
Orizont *Orizont::Imbina(Orizont *other){
    Orizont *res = new Orizont(this->n + other->n);
    int h1 = 0, h2 = 0, i = 0, j = 0;
    while (i < this->n && j < other->n){
        if (this->arr[i].stanga < other->arr[j].stanga){
            int x1 = this->arr[i].stanga; h1 = this->arr[i].inaltime;
            int maxh = max(h1, h2); res->adauga(new banda(x1, maxh)); i++;
        }
        else {
            int x2 = other->arr[j].stanga; h2 = other->arr[j].inaltime;
            int maxh = max(h1, h2); res->adauga(new banda(x2, maxh));
            j++;
        }
    }
    while (i < this->n){
        res->adauga(&arr[i]); i++;
    }
    while (j < other->n){
        res->adauga(&other->arr[j]); j++;
    }
    return res;
}
int main(){
    cladire arr[] = {{1,11,5}, {2,6,7}, {3,13,9}, {12,7,16},
                     {14,3,25}, {19,18,22}, {23,13,29}, {24,4,28}};
    int n = sizeof(arr)/sizeof(arr[0]);
    Orizont *ptr = Gaseste(arr, 0, n-1);
    cout <<"Orizontul pentru datele cladirilor introduse este: \n";
    ptr->afisare();
    return 0;
}
```

Rezultatele execuției:

```
Orizontul pentru datele cladirilor introduse este:
        (1,11)
        (3,13)
        (9,0)
        (12,7)
        (16,3)
        (19,18)
        (22,3)
        (23,13)
        (29,0)
```

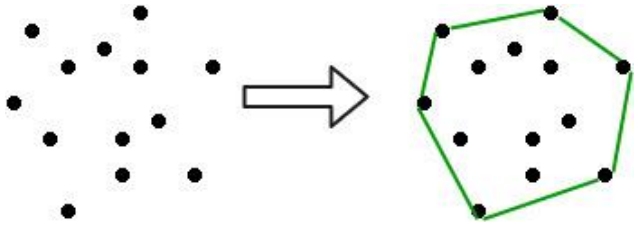
Problema 4: Înfășurătoarea convexă / Coca convexă

Condiția problemei

O înfășurătoare convexă este cel mai mic poligon convex care conține toate punctele date. Fie că avem o mulțime de puncte, specificate de coordonatele lor x și y. Să se construiască înfășurătoarea a convexă a acestei mulțimi de puncte.

Indicații de rezolvare

1. *Găsiți punctul cu coordonată x minimă: min_x și în mod similar punctul cu coordonată x maximă: max_x.*
2. *Alcătuiț o linie care unește aceste două puncte: L. Această linie va împărți întreaga mulțime în două părți. Luați ambele părți una câte una și continuați mai departe.*
3. *Pentru o parte, găsiți punctul P cu distanța maximă de la linia L. P formează un triunghi cu punctele min_x, max_x. Este clar că punctele care se află în interiorul acestui triunghi nu pot fi niciodată partea unei înfășurători convexe.*
4. *Etapa de mai sus împarte problema în două sub-probleme (rezolvate recursiv). Acum linia care unește punctele P și min_x și linia care unește punctele P și max_x sunt linii noi, iar punctele care stau în afara triunghiului sunt mulțimea de puncte. Repetați punctul nr. 3 până nu a mai rămas niciun punct cu linia (distanța). Adăugați punctele finale ale acestui punct la înfășurătoarea convexă.*



Implementare C++

```
#include<bits/stdc++.h>
using namespace std;
#define iPereche pair<int, int>
set<iPereche> infasuratoare;
int Gasit(iPereche p1, iPereche p2, iPereche p){
    int val=(p.second-p1.second)*(p2.first-p1.first)-(p2.second-
p1.second)*(p.first-p1.first);
    if (val > 0) return 1;
    if (val < 0) return -1;
    return 0;
}
int Distanta(iPereche p1, iPereche p2, iPereche p){
    return abs((p.second-p1.second)*(p2.first-p1.first)-(p2.second-
p1.second)*(p.first-p1.first));
}
void infasuratoare_rapida(iPereche a[], int n, iPereche p1, iPereche p2, int side){
    int ind = -1, max_dist = 0;
    for (int i=0; i<n; i++){
        int temp = Distanta(p1, p2, a[i]);
        if (Gasit(p1, p2, a[i]) == side && temp > max_dist){
            ind = i; max_dist = temp;
        }
    }
    if (ind == -1){
        infasuratoare.insert(p1); infasuratoare.insert(p2);
        return;
    }
    infasuratoare_rapida(a, n, a[ind], p1, -Gasit(a[ind], p1, p2));
    infasuratoare_rapida(a, n, a[ind], p2, -Gasit(a[ind], p2, p1));
}
```

```
void afisare(iPereche a[], int n){
    if (n < 3){
        cout << "Nu este posibil de construit infasuratoarea! \n";
        return;
    }
    int min_x = 0, max_x = 0;
    for (int i=1; i<n; i++){
        if (a[i].first < a[min_x].first) min_x = i;
        if (a[i].first > a[max_x].first) max_x = i;
    }
    infasuratoare_rapida(a, n, a[min_x], a[max_x], 1);
    infasuratoare_rapida(a, n, a[min_x], a[max_x], -1);
    cout << "Infasuratoare convexa posibila este formata din punctele: \n";
    while (!infasuratoare.empty()){
        cout << "\t(" <<( *infasuratoare.begin()).first << ", ";
        cout<< (*infasuratoare.begin()).second << ")\n";
        infasuratoare.erase(infasuratoare.begin());
    }
}
int main(){
    iPereche a[] = {{2,3}, {3,1}, {3,5}, {4,6}, {5,4}, {1,2}, {7,3}, {8,6}, {9,1},
                    {5,3}, {4,1}, {8,5}, {4,10}, {5,9}, {1,3}, {7,10}, {8,2} };
    int n = sizeof(a)/sizeof(a[0]);
    afisare(a, n);
    return 0;
}
```

Rezultatele execuției:

```
Infasuratoare convexa posibila este formata din punctele:
        (1, 2)
        (1, 3)
        (3, 1)
        (4, 10)
        (7, 10)
        (8, 6)
        (9, 1)
```

Problema 5: Suma și produsul a două polinoame

Condiția problemei

*Fie avem două polinoame A(x) și B(x). Să se efectueze suma și produsul dintre aceste polinoame: S(x)= A(x) + B(x) și C(x)= A(x) * B(x).*

Exemplu:

Pentru polinoamele: $A(x)=6x^3+7x^2-10x+9$ *și* $B(x)=-2x^3+4x-5$.

Suma polinoamelor: $S(x)= 4x^3+7x^2-6x+4$

Produsul polinoamelor: $C(x)=-12x^6-14x^5+44x^4-20x^3-75x^2+86x-45$.

Indicații de rezolvare

Vom obține vectorii coeficienți în felul următor: A[]={9, -10, 7, 6} și B[]={-5, 4, 0, -2}

- *Adăugarea este mai simplă decât înmulțirea polinoamelor. Inițializăm rezultatul ca unul dintre cele două polinoame, apoi traversăm celălalt polinom și adăugăm toți termenii la rezultat.*
- *soluție simplă este să luați în considerare unul câte unul fiecare termen al primului polinom și să-l multiplicați cu fiecare termen de la al doilea polinom. Urmează algoritmul acestei metode:*
  1) *Creați un vector Produs [] cu dimensiunea m + n-1.*
  2) *Inițializați toate intrările din Produs [] cu valoarea 0.*
  3) *Parcurgeți vectorul A [] și faceți următoarele pentru fiecare element A [i].*
     Parcurgeți vectorul B [] și faceți următoarele pentru fiecare element B [i].
  4) *Produs [i + j] = Produs [i + j] + A [i] * B [j].*
  5) *Returnați Produs [].*

Implementare C++

```
#include <iostream>
using namespace std;
int *inmultire(int A[], int B[], int m, int n){
    int *prod = new int[m+n-1];
    for (int i = 0; i<m+n-1; i++)
    prod[i] = 0;
    for (int i=0; i<m; i++){
        for (int j=0; j<n; j++)
        prod[i+j] += A[i]*B[j];
   }
   return prod;
}
int *suma(int A[], int B[], int m, int n) {
    int size = max(m, n);
    int *sum = new int[size];
    for (int i = 0; i<m; i++)
        sum[i] = A[i];
    for (int i=0; i<n; i++)
        sum[i] += B[i];
    return sum;
}
void afisare(int poly[], int n){
    for (int i=0; i<n; i++){
       cout <<"\t"<<poly[i];
    }
}
int main(){
    /// Vectorul A reprezinta polinomul 9 - 10x + 7x^2 + 6x^3
    int A[] = {9, -10, 7, 6};
    int m = sizeof(A)/sizeof(A[0]);

    /// Vectorul B reprezinta polinomul -5 + 4x - 2x^3
    int B[] = {-5, 4, 0, -2};
    int n = sizeof(B)/sizeof(B[0]);

    /// Afisarea exponentilor pentru polinomul A: 9 - 10x + 7x^2 + 6x^3
    cout << "\nPolinomul A:\n";
    for(int j=0; j<m; j++){
```

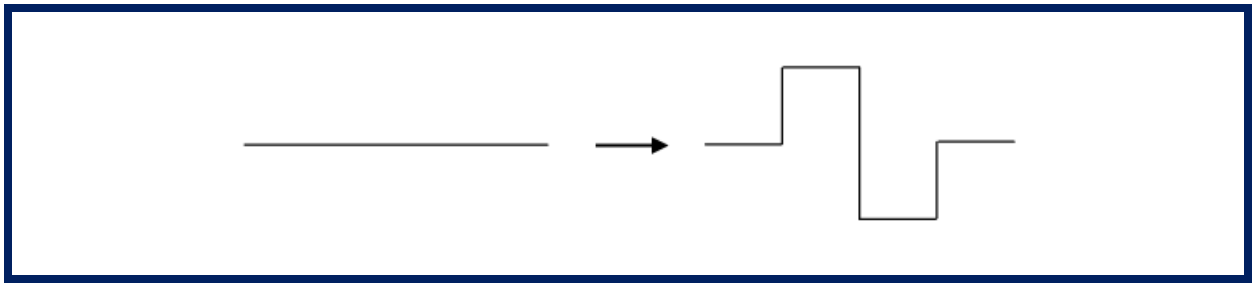
```
        cout<<"\t"<<j;
    }cout<<endl;
    afisare(A, m);

    /// Afisarea exponentilor pentru polinomul B: -5 + 4x - 2x^3
    cout << "\n\nPolinomul B:\n";
    for(int j=0; j<n; j++){
        cout<<"\t"<<j;
    }cout<<endl;
    afisare(B, n);

    /// Afisarea exponentilor pentru suma polinoamelor A+B
    cout << "\n\nSuma polinoamelor, S = A + B:\n";
    int *S = suma(A, B, m, n);
    for(int j=0; j<max(n,m); j++){
        cout<<"\t"<<j;
    }cout<<endl;
    afisare(S, max(n,m));

    /// Afisarea exponentilor pentru produsul polinoamelor A*B
    cout << "\n\nProdusul polinoamelor, C = A * B:\n";
    int *C = inmultire(A, B, m, n);
    for(int j=0; j<n+m-1; j++){
        cout<<"\t"<<j;
    }cout<<endl;
    afisare(C, m+n-1);
    cout << "\n\n";
    return 0;
}
```

Rezultatele execuției:

```
Polinomul A:
        0       1       2       3
        9       -10     7       6

Polinomul B:
        0       1       2       3
        -5      4       0       -2

Suma polinoamelor, S = A + B:
        0       1       2       3
        4       -6      7       4

Produsul polinoamelor, C = A * B:
        0       1       2       3       4       5       6
        -45     86      -75     -20     44      -14     -12
```

Problema 6: Geometrie fractală. Fulgul lui Koch pentru pătrat

Se consideră un pătrat. Fiecare latură a sa se transformă după cum se vede în figura de mai jos:



*Fiecare segment al liniei frânte astfel formate se transformă din nou după aceeaşi regulă. Se cere să se vizualizeze curba după **M** transformări (valoare citită de la tastatură).*

Indicaţii de rezolvare

*Transformarea şi desenarea unui segment sunt realizate de subprogramul **desen**. Aceasta are ca parametri de intrare coordonatele punctului care determină segmentul, numărul de transformări efectuate (**N**) şi numărul de transformări cerut (**M**).*
Subprogramul conţine următorul algoritm:

- *dacă nu a fost efectuat numărul de transformări necesar, se calculează coordonatele punctelor care determină linia frântă obţinută pornind de la segment şi pentru fiecare segment din această linie se reapelează subprogramul desen;*
- *contrar, se desenează linia frântă obţinută.*

Implementare C++

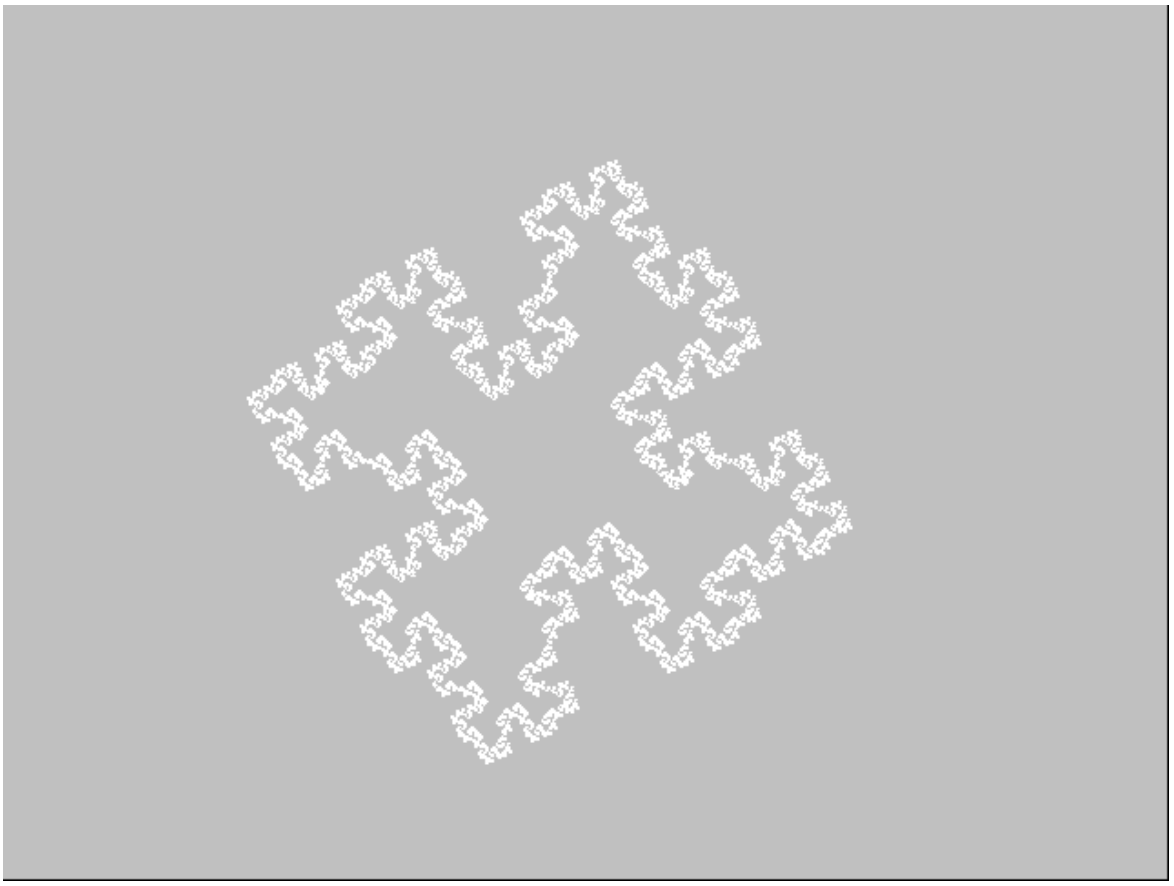
```
#include <iostream>
#include "graphics.h"
#include <math.h>
#include <stdio.h>
using namespace std;
int gdriver,gmode,ls,lc=15,lu,L;
void init() {
    gdriver = VGA; gmode = VGAHI;
    initgraph(&gdriver,&gmode," ");
    if (graphresult()) {
        cout<<"Tentativa nereusita.";
        cout<<"Apasa o tasta pentru a inchide...";
        getch();
        exit(1);
    }
}
void rotplan(int xc,int yc,int x1,int y1,int &x,int &y, float unghi) {
    x = ceil(xc+(x1-xc)*cos(unghi)-(y1-yc)*sin(unghi));
    y = ceil(yc+(x1-xc)*sin(unghi)+(y1-yc)*cos(unghi));
}
void desen(int x1,int y1,int x2,int y2,int n,int ls){
    int x3,x4,x5,x6,x7,x8,xc,y3,y4,y5,y6,y7,y8,yc;
    if (n<=ls){
        x3=div(3*x1+x2,4).quot; y3=div(3*y1+y2, 4).quot;
        rotplan(x3,y3,x1,y1,x4,y4,-M_PI/2);
        xc=div(x1+x2,2).quot; yc=div(y1+y2,2).quot;
        rotplan(xc,yc,x3,y3,x5,y5,-M_PI/2);
        rotplan(xc,yc,x3,y3,x6,y6,M_PI/2);
        x8=div(x1+3*x2, 4).quot; y8=div(y1+3*y2,4).quot;
        rotplan(x8,y8,xc,yc,x7,y7,M_PI/2);
        desen(x1,y1,x3,y3,n+1,ls); desen(x3,y3,x4,y4,n+1,ls);
        desen(x4,y4,x5,y5,n+1,ls); desen(x5,y5,xc,yc,n+1,ls);
        desen(xc,yc,x6,y6,n+1,ls); desen(x6,y6,x7,y7,n+1,ls);
        desen(x7,y7,x8,y8,n+1,ls); desen(x8,y8,x2,y2,n+1,ls);
        if (n == ls){
            moveto(x1,y1); lineto(x3,y3); lineto(x4,y4); lineto(x5,y5);
```

```
            lineto(x6,y6); lineto(x7,y7); lineto(x8,y8); lineto(x2,y2);
        }
    }
}
int main(){
    cout<<"Vom utiliza un fundal de background de culoare LIGHTGRAY (GRI
DESCHIS).\n";
    cout<<"Vom utiliza un contur de desen de culoare WHITE (ALB).\n";
    cout<<"\nIntroduceti numarul de iteratii:\t "; cin>>ls;
    init();
    setbkcolor(7);
    for(int i=0;i<=ls;i++){
        setcolor(lc);
        desen(200,150,400,150,1,i); desen(400,150,400,350,1,i);
        desen(400,350,200,350,1,i); desen(200,350,200,150,1,i);
        getchar(); delay(10);
        cleardevice();
    }
    closegraph();
}
```

Rezultatele execuției:

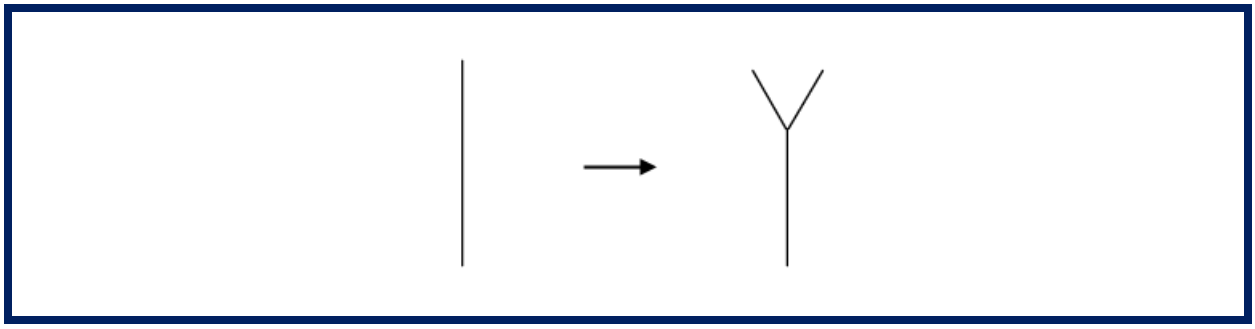
```
Vom utiliza un fundal de background de culoare LIGHTGRAY (GRI DESCHIS).
Vom utiliza un contur de desen de culoare WHITE (ALB).

Introduceti numarul de iteratii:         4
```

Modulul grafic:



Problema 7: Geometrie fractală. Arbore simetric

Se dă un segment AB. Cu ajutorul lui se construieşte un arbore, aşa cum se vede în figura de mai jos:



Lungimea fiecărei ramuri este o treime din lungimea iniţială a segmentului. Fiecare latură se transformă în mod asemănător. Se cere să se vizualizeze figura astfel rezultată, după ***M*** *transformări.*

Indicaţii de rezolvare

Pentru obţinerea ramurilor se procedează astfel:

- *se consideră punctul situat pe dreapta determinată de segment şi pentru care avem:*

$$k=\frac{CA}{CB}=3;\quad x_c=\frac{x_1-3\cdot x_2}{1-3}=\frac{3\cdot x_2-x_1}{2},\quad y_p=\frac{y_1-3\cdot y_2}{1-3}=\frac{3\cdot y_2-y_1}{2}.$$

- *se roteşte acest punct în jurul punctului $B(x_2,y_2)$ cu un unghi alfa = Pi / 4;*
- *se roteşte punctul în jurul lui B cu unghiul beta = - Pi / 4*

În urma acestor rotaţii se obţin coordonatele punctelor care, împreună cu punctul B, determină segmentele ce costituie ramurile arborelui. Subprogramul ***desenez*** *are ca parametri de intrare coordonatele unui segment, numărul de transformări efectuate (**n**) şi numărul de transformări care trebuie efectuate (**m**). În cazul în care nu s-au efectuat toate transformările, se trasează segmentul (cu o culoare oarecare), se calculează coordonatele punctelor care determină ramurile şi, pentru fiecare segment, se reapelează subprogramul.*

Implementare C++

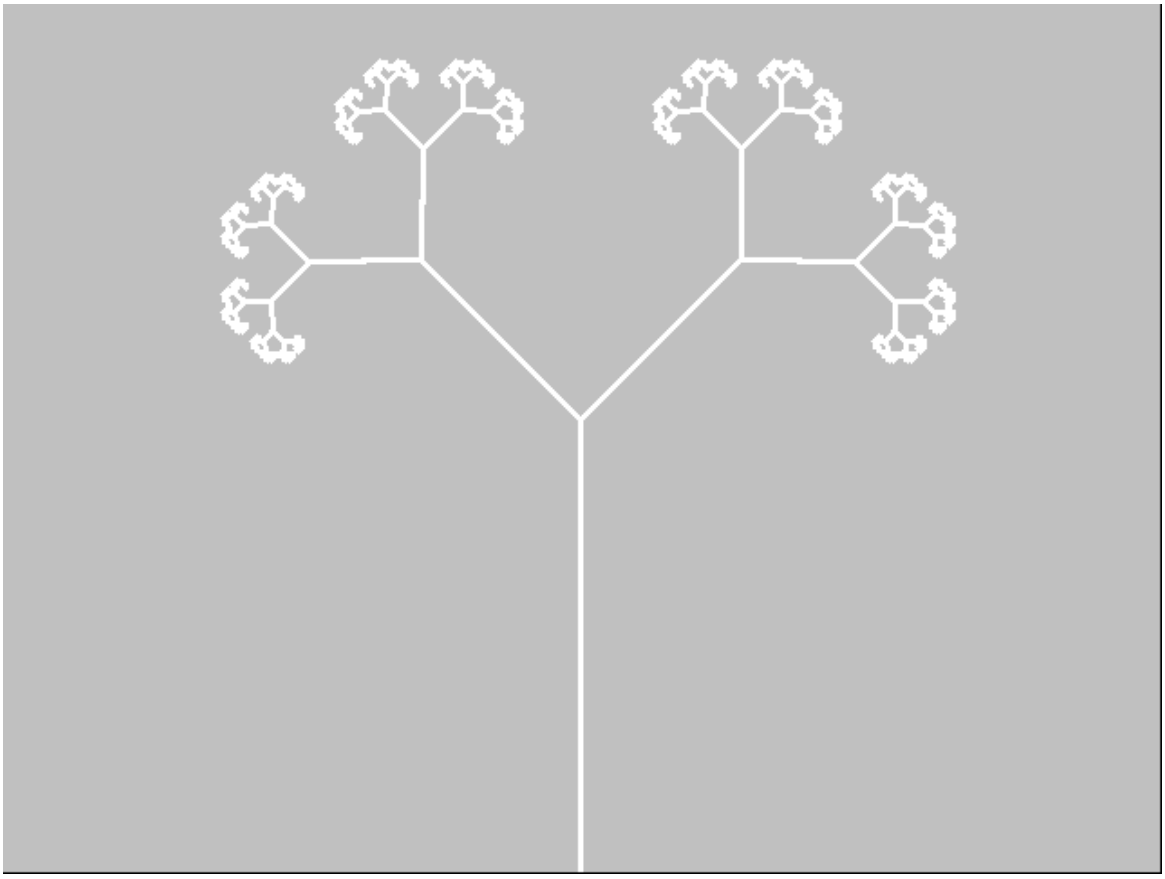
```
#include <iostream>
#include "graphics.h"
#include <math.h>
#include <stdio.h>
using namespace std;
int gdriver,gmode,ls,lc,lu,L;
void init() {
    gdriver = VGA; gmode = VGAHI;
    initgraph(&gdriver,&gmode," ");
    if (graphresult()) {
        cout<<"Tentativa nereusita!";
        cout<<"Apasati o tasta pentru a inchide!";
        getch();
        exit(1);
    }
}
void rotplan(int xc,int yc,int x1,int y1,int &x,int &y, float unghi) {
    x = ceil(xc+(x1-xc)*cos(unghi)-(y1-yc)*sin(unghi));
    y = ceil(yc+(x1-xc)*sin(unghi)+(y1-yc)*cos(unghi));
}
void desenez(int x1,int y1,int x2,int y2,int n,int ls){
    int x,y;
    //setcolor(rand()%15 + 1);
    setcolor(15);
    setlinestyle(0,0,3);
    if (n<=ls){
        moveto(x1,y1); lineto(x2,y2);
        rotplan(x2,y2,div(3*x2-x1,2).quot,div(3*y2-y1,2).quot,x,y,M_PI/4);
        desenez(x2,y2,x,y,n+1,ls);
```

```
        rotplan(x2,y2,div(3*x2-x1,2).quot,div(3*y2-y1,2).quot,x,y,-M_PI/4);
        desenez(x2,y2,x,y,n+1,ls);
    }
}
int main(){
    cout<<"Vom utiliza un fundal de background de culoare LIGHTGRAY (GRI
DESCHIS).\n";
    cout<<"Vom utiliza un contur de desen de culoare WHITE (ALB).\n";
    cout<<"\nIntroduceti numarul de iteratii:\t "; cin>>ls;
    init();
    setbkcolor(7);
    for(int i=0;i<=ls;i++){
        desenez(div(getmaxx(),2).quot,getmaxy(),div(getmaxx(),2).quot,getmaxy()-
250,1,i);
        getchar(); cleardevice();
    }
    closegraph();
}
```

Rezultatele execuției:

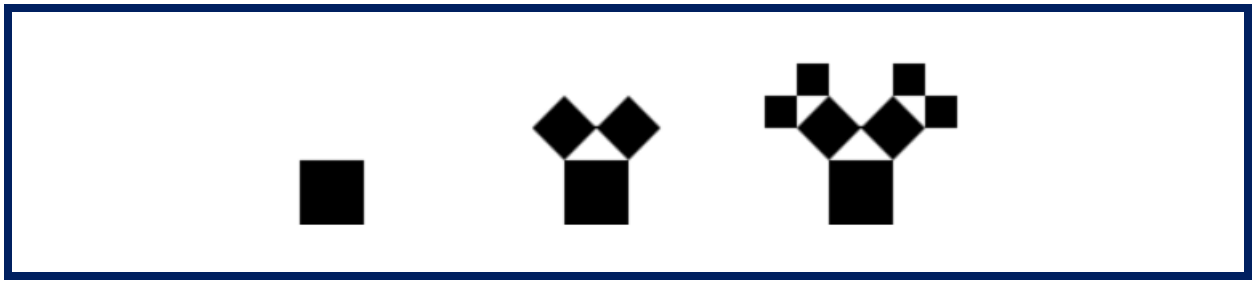
```
Vom utiliza un fundal de background de culoare LIGHTGRAY (GRI DESCHIS).
Vom utiliza un contur de desen de culoare WHITE (ALB).

Introduceti numarul de iteratii:         10
```

Modulul grafic:



Problema 8: Geometrie fractală. Arborele simetric Pitagorean

Se dă un patrat. Cu ajutorul lui se construieşte un arbore, aşa cum se vede în figura de mai jos:



Indicații de rezolvare

Construcția arborelui lui Pitagora începe cu un pătrat. Construiți un triunghi isoscel drept a cărui ipotenuză este marginea superioară a pătratului. Construiți pătrate de-a lungul fiecăreia din celelalte două laturi ale acestui triunghi izoscel. Repetați această construcție recursiv pe fiecare din cele două pătrate noi. Limita acestei construcții se numește Arborele Pitagoreului (sau Arborele lui Pitagora).
Când arborele pitagorean este desenat cu triunghiuri isoscele ($\alpha = 45°$) şi un pătrat unitar ca mulțime inițială, arborele se va potrivi exact în interiorul unui dreptunghi cu lățimea 6 și înălțimea 4. Pătratele nu se vor suprapune pentru primele patru iterații, dar după aceea, pătratele vor începe să se suprapună și să înceapă să crească înapoi, precum și să se extindă spre exterior pentru a crea efectul „frunze" în jurul perimetrului arborelui. Arborele va rămâne întotdeauna în interiorul acestui dreptunghi.

Implementare C++

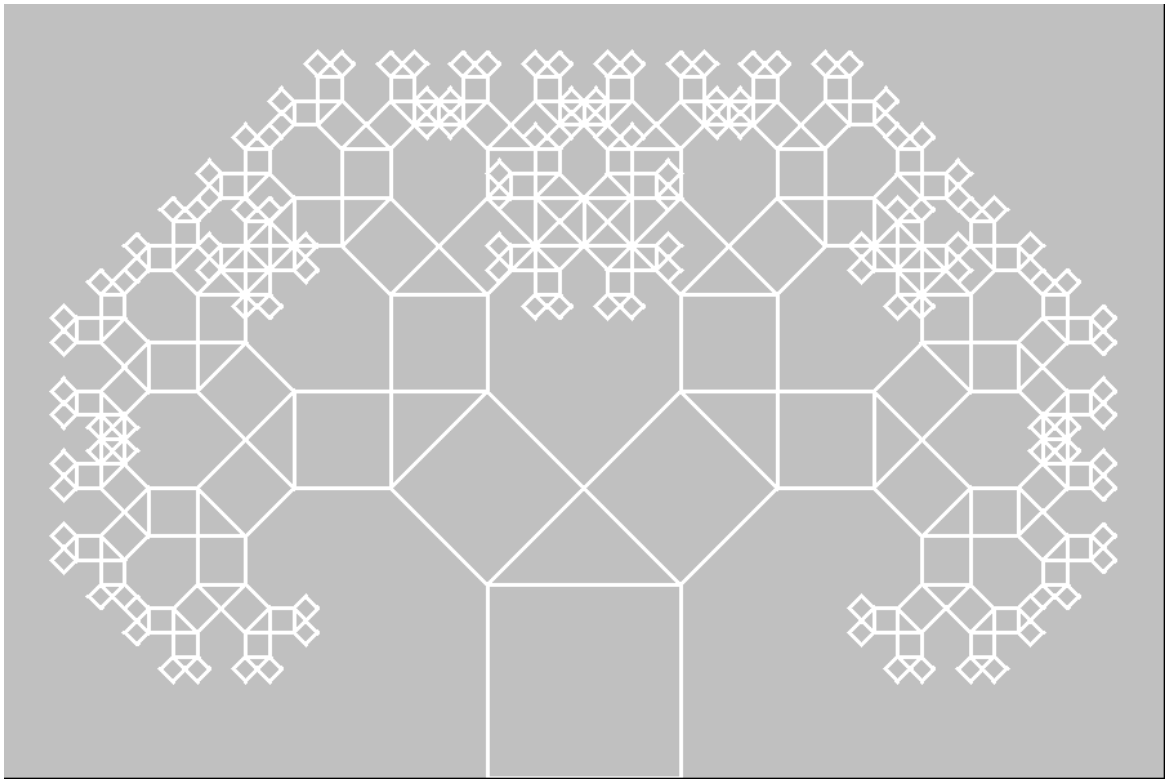
```
#include<iostream>
#include<graphics.h>
#include <stdio.h>
using namespace std;
typedef struct{
       double x,y;
}Punct;
void Pitagora(Punct a,Punct b,int i_ori){
       Punct c,d,e;
       c.x = b.x - (a.y -  b.y);
       c.y = b.y - (b.x - a.x);
       d.x = a.x - (a.y -  b.y);
       d.y = a.y - (b.x - a.x);
       e.x = d.x + ( b.x - a.x - (a.y - b.y) ) / 2;
       e.y = d.y - ( b.x - a.x + a.y - b.y ) / 2;
       //setcolor(rand()%15 + 1);
       setcolor(15);
    setlinestyle(0,0,3);
    if(i_ori>0){
              line(a.x,a.y,b.x,b.y); line(c.x,c.y,b.x,b.y);
              line(c.x,c.y,d.x,d.y); line(a.x,a.y,d.x,d.y);
              Pitagora(d,e,i_ori-1); Pitagora(e,c,i_ori-1);
       }
}
int main(){
       Punct a,b;
       double Latura=150;
       int iter,lc;
       time_t t;
       cout<<"Vom utiliza un fundal de background de culoare LIGHTGRAY (GRI
DESCHIS).\n";
     cout<<"Vom utiliza un contur de desen de culoare WHITE (ALB).\n";
     cout<<"\nIntroduceti numarul de iteratii:\t "; cin>>iter;
       initwindow(6*Latura,4*Latura,"Arborele lui pitagora");
       setbkcolor(7);
     for(int j=0;j<=iter;j++){
           a.x = 6*Latura/2 - Latura/2; a.y = 4*Latura;
           b.x = 6*Latura/2 + Latura/2; b.y = 4*Latura;
```

```
            Pitagora(a,b,j);
            getchar();
            cleardevice();
        }
        closegraph();
        return 0;
}
```

Rezultatele execuției:

```
Vom utiliza un fundal de background de culoare LIGHTGRAY (GRI DESCHIS).
Vom utiliza un contur de desen de culoare WHITE (ALB).

Introduceti numarul de iteratii:         8
```

Modulul grafic:



SARCINI PENTRU EXERSARE

Problema 1

Într-un fișier text sunt scrise următoarele informații:

- *pe primul rând numărul N de elevi din clasă,*
- *pe următoarele N rânduri, pentru fiecare elev, mediile semestriale la disciplina informatică.*

În cadrul unui rând, mediile sunt separate prin spațiu. Fiecare elev se identifică prin numărul de ordine la citirea din fișier. Să se scrie un program care să realizeze următoarele cerințe:

a) *Se citesc datele din fișier și se calculează media anuală a fiecărui elev;*
b) *Pentru identificatorul unui elev, citit de la tastatură, se afișează mediile semestriale și media anuală;*
c) *Se rearanjează elevii în ordinea crescătoare a mediilor și se afișează mediile fiecărui elev;*
d) *Se scriu informațiile obținute în urma prelucrărilor într-un alt fișier.*

Problema 2

Fie N un număr natural nenul. Fie A un vector cu N poziții numerotate de la 1 la N și elemente numere naturale diferite, de la 1 la N, într-o ordine oarecare. Pentru i și j numere naturale între 1 și N, numim pocnet (N, A, i, j) operația care inversează ordinea elementelor din A situate pe pozițiile de la i la j.

a) *Să se scrie într-un limbaj de programare un subprogram care implementează operația pocnet (N, A, i, j).*
b) *Să se scrie un program care sortează crescător vectorul A, folosind pentru schimbarea ordinii elementelor în A doar operația pocnet (N, A, 1, k), cu k de la 2 la N.*
c) *Considerăm că n este o putere a lui 2 (n = 2^m, cu m număr natural nenul) și vectorul A are proprietatea că pentru orice i de la 1 la m și orice j de la 1 la 2^{m-i} , există k de la 1 la 2^{m-i}, astfel încât pe pozițiile din A de la 2^i * (j-1)+1 la 2^i * j se află numerele naturale de la 2^i*(k-1)+1 la 2^i*k, într-o ordine oarecare. Să se scrie un program care sortează crescător vectorul A, folosind pentru schimbarea ordinii elementelor în A doar operația pocnet(N, A, 2^i* (j-1)+1, 2^i* j), cu i de la 1 la m și j de la 1 la 2^{m-i}, printr-un algoritm mai eficient decât cel implementat la punctul precedent, care se bazează pe proprietatea vectorului A* ***[24]***.

Problema 3

Ghiciţ numărul. Realizaţ un joc astfel încât calculatorul să poată ghici un număr cuprins între 1 şi 1000 , evident din cât mai puţine întrebări, la care dumneavoastră să răspundeţi prin "da" sau "nu". Întrebările pot fi doar de forma: Numărul este egal cu X? Numărul este mai mare decât X? Numărul este mai mic decât X?

Problema 4

Un arbore cartezian al unui vector este un arbore binar definit recursiv astfel:

- *rădăcina arborelui este elementul cel mai mic din vector;*
- *subarborele stâng este arborele cartezian al subvectorului stâng (faţă de poziţia elementului din rădăcină);*
- *subarborele drept este arborele cartezian al subvectorului drept.*

Se dă un vector de dimensiune n. Să se afişeze arborele său cartezian.

Problema 5

Demonstrați că pentru orice intregi m și n sunt adevărate următoarele proprietăți:

1. *dacă m și n sunt pare, atunci cmmdc(m, n) = 2cmmdc(m/2, n/2);*
2. *dacă m este impar și n este par, atunci cmmdc(m, n) = cmmdc(m, n/2);*
3. *dacă m și n sunt impare, atunci cmmdc(m, n) = cmmdc((m-n)/2, n).*

Pe majoritatea calculatoarelor, operațiile de scădere, testare a parității unui întreg și împărțirea la doi sunt mai rapide decât calcularea restului împărțirii întregi. Elaborați un algoritm divide et impera pentru a calcula cel mai mare divizor comun a doi întregi, evitând calcularea restului împărțirii întregi. Folosiți proprietățile de mai sus ***[25]***.

TEHNICI DE SORTARE

6.1 Noțiuni generale despre sortare

În informatică, „a sorta" înseamnă a rearanja elementele dintr-o structura de date în ordine crescătoare (sau descrescătoare). Un vector (tablou unidimensional) este numit sortat atunci când elementele lui sunt într-o anumită ordine (crescătoare sau descrescătoare). Există mai multe tipuri de sortări, în funcție de timpul de răspuns. Printre cele mai cunoscute se numără: BubbleSort, SelectionSort, InsertionSort, etc.

Sortarea este o operație fundamentală în informatică, deoarece:

- *Putem sorta datele pentru prezentarea către utilizator (de exemplu în agenda telefonică, sau atunci când navigăm printre foldere, sau selectăm o anumită melodie din telefon);*
- *Sortarea este o piatră fundamentală în algoritmii mai mari, pentru a simplifica anumite cerințe (precum găsirea unei chei unice, sau detectarea elementelor duplicate).*

Putem sorta: numere, texte, cuvinte, orice, care are un criteriu de sortare. Dar pentru usurință vom folosii numere în toate exemplele noastre. Aveți grijă să distingeți poziția elementului (i) de elementul efectiv (V[i])

Evaluarea algoritmilor de sortare

- *Cât de mult spațiu necesita algoritmul, în afară de spațiul efectiv pe care îl ocupă vectorul? În cel mai bun caz acel spațiu este O(1), dar există și algoritmi care au nevoie de spațiu extra pentru a sorta mai efficient. Acest procedeu se numeste „trading space for time" – ne vom mai întâlni cu el.*
- *Cât de repede se execută sortarea? Ar trebui să numarăm de câte ori se compară două numere și să luăm în considerare atât cazul mediu (average case), cât și cazul cel mai nefavorabil (worst case).*
- *Ce se întâmplă dacă există elemente care se repetă?*

Din punct de vedere al eficienței, avem:

- *algoritmi neeficienți, de complexitate $O(n^2)$:*
  - *metoda bulelor*
  - *sortarea prin selecție*
  - *sortarea prin inserție*
  - *etc.*
- *algoritmi eficienți, de complexitate $O(n \cdot \log n)$:*
  - *QuickSort*
  - *MergeSort*
  - *HeapSort*
  - *etc.*

Notă:

- ✓ *Pentru structuri de date particulare există și algoritmi de complexitate O(n).*
- ✓ *De asemenea, există algoritmi exponențiali, de complexitate O(n!), fără utilitate practică.*

Exemplul 1

Considerăm mulțimea de valori întregi: (5,8,3,1,6). În acest caz cheia de sortare coincide cu valoarea elementului. Prin sortare crescătoare se obține mulțimea (1,3,5,6,8), iar prin sortare descrescătoare se obține (8,6,5,3,1).

Exemplul 2

Considerăm un tabel constant din nume ale studenților și note: {(Popescu, 9), (Ionescu, 10), (Voinescu, 8), (Adam, 9)}. În acest caz cheia de sortare poate fi numele sau nota. Prin ordonare crescătoare după nume se obține {(Adam, 9), (Ionescu, 10), (Popescu, 9), (Voinescu, 8) }, iar prin ordonare descrescătoare după notă se obține {(Ionescu, 10), (Popescu, 9), (Adam, 9), (Voinescu, 8) }.

În funcție de spațiul de manevră necesar pentru efectuarea sortării există:

- *Sortare ce folosește o zonă de manevră de dimensiunea mulțimii de date. Dacă mulțimea inițială de date este reprezentat de tabloul x[1..n], cel sortat se va obține într-un alt tablou y[1..n].*
- *Sortare în aceeași zonă de memorie. Elementele tabloului x[1..n] își schimbă pozițiile astfel încât după încheierea procesului să fie ordonate. Este posibil ca și în acest caz să se folosească o zonă de memorie, însă aceasta este de regulă de dimensiunea unui element și nu de dimensiunea întregului tablou* ***[40]****.*

Metodele de sortare pot fi caracterizate prin:

- ***Stabilitate.*** *O metodă de sortare este considerată stabilă dacă ordinea relativă a elementelor ce au aceeași valoare a cheii nu se modifică în procesul de sortare. De exemplu, dacă asupra tabelului cu note din Exemplul 2 se aplică o metodă stabilă de ordonare descrescătoare după notă se obține {(Ionescu, 10), (Popescu, 9), (Adam, 9), (Voinescu, 8)} pe când dacă se aplică una care nu este stabilă se va putea obține {(Ionescu, 10), (Popescu, 9), (Adam, 9), (Voinescu, 8)}.*
- ***Naturalețe.*** *O metodă de sortare este considerată naturală dacă numărul de operații scade odată cu distanța dintre tabloul inițial și cel sortat. O măsură a acestei distanțe poate fi numărul de inversiuni ale permutării corespunzătoare tabloului inițial.*
- ***Eficiență.*** *O metodă este considerată eficientă dacă nu necesită un volum mare de resurse. Din punctul de vedere al spațiului de memorie o metodă de sortare pe loc este mai eficientă decât una bazată pe o zonă de manevră de dimensiunea tabloului. Din punct de vedere al timpului de execuție este important să fie efectuate cât mai puține operații. În general, în analiză se iau în considerare doar operațiile efectuate asupra elementelor tabloului (comparații și mutări). O metodă este considerată optimală dacă ordinul său de complexitate este cel mai mic din clasa de metode din care face parte.*
- ***Simplitate.*** *O metodă este considerată simplă dacă este intuitivă și ușor de înțeles.*

Observație:

- *Primele două proprietăți sunt specifice algoritmilor de sortare, pe când ultimele sunt cu caracter general.*
- *În continuare vom considera câteva metode elementare de sortare caracterizate prin faptul că sunt simple, nu sunt cele mai eficiente metode, dar reprezintă punct de pornire pentru metode avansate. Pentru fiecare dintre aceste metode se va prezenta: principiul și analiza complexității.*

Menţionăm următoarele operaţii pe biţi ce se pot folosi în C++ :

1. *Operaţii logice*
   - *& şi pe biţi(bitwise AND)*
   - *| sau pe biţi(bitwise OR)*
   - *^ sau exclusiv pe biţi(bitwise XOR)*
   - *~ complement pe biţi(bitwise NOT)*
2. *Deplasări*
   - *» la dreapta(right shift)*
   - *« la stânga(left shift)*

Descriem numai operațiile pe care le vom folosi în cadrul exemplului de mai jos: » și &.

- *Operaţia n » k are ca rezultat valoarea obţinută prin mutarea la dreapta a tuturor biţilor lui n(pe primii k biţi se obţine 0, iar ultimii k biţi din n sunt ignoraţi).*
- *Operația n & k are ca rezultat valoarea obținută prin păstrarea biților nenuli din n pentru pozițiile pe care și k are biți nenuli(0 în rest)* ***[41]****.*

Notă:

- ✓ *Dacă n este număr natural şi k = 2p, atunci:*
  - *n » p == n / k*
  - *n & (k - 1) == n % k*
- ✓ *Apar diferenţe în cazul numerelor negative.*

6.2 Tehnici de sortare directe

Sortarea cu bule *(BubbleSort)*

Metoda constă în parcurgerea șirului ori de câte ori este nevoie până când șirul devine sortat sau cât timp șirul nu este sortat. Înainte de parcurgerea șirului se presupune de fiecare dată ca șirul ar fi sortat utilizând o variabilă logică. La fiecare iterație șirul se va parcurge până la ultima componentă care nu este sortată. Parcurgerea șirului constă în compararea de fiecare dată a două componente de pe poziții consecutive, de forma a[i] și a[i+1]. Dacă a[i]>a[i+1] cele două componente se vor interschimba. Dacă la o parcurgere a șirului există cel puțin o interschimbare înseamnă că șirul încă nu este sigur sortat și se reia parcurgerea șirului. Dacă la o parcurgere a șirului nu există nici o interschimbare, atunci cu siguranță șirul este sortat. Metoda are rolul de a transporta, la fiecare parcurgere a șirului, valoarea maximă dintre componentele nesortate și de a o plasa pe poziția finală în șirul sortat a o plasa pe poziția finală în șirul sortat.

Timp mediu	Timp la limită	Memorie	Stabilitate
$O(N^2)$	$O(N^2)$	O(1)	DA

Implementare:

```
void bubbleSort(int a[],int n){
    int i,schimbat,aux;
    do {
        schimbat = 0;
        for(i = 0; i < n-1; i++) {
            if (a[i] < a[i+1]) {
                aux = a[i]; a[i] = a[i+1]; a[i+1] = aux; schimbat = 1;
            }
        }
    } while(schimbat);
}
```

Sortarea prin selecție *(SelectionSort)*

Metoda constă în parcurgerea șirului de la prima componentă până la componenta n-1 inclusiv. La pasul i componentele a[1],a[2],…,a[i-1] sunt deja sortate. La pasul i se determină valoarea minimă, respectiv, poziția elementului minim dintre componentele a[i],a[i+1],…,a[n]. Fie p poziția elementului minim . Dacă p este diferit de i atunci componentele a[i] și a[p] se vor interschimba.

Timp mediu	Timp la limită	Memorie	Stabilitate
$O(N^2)$	$O(N^2)$	O(1)	DA

Implementare:

```
void selectionSort(int a[],int n){
      int i,j,aux,minim,minPoz;
      for(i = 0; i < n - 1;i++) {
            minPoz = i; minim = a[i];
            for(j = i + 1;j < n;j++){
           if(minim > a[j]){
                        minPoz = j; minim = a[j];
                  }
            }
            aux = a[i]; a[i] = a[minPoz]; a[minPoz] = aux;
      }
}
```

Sortarea prin inserție *(InsertionSort)*

Această metodă constă în parcurgerea șirului, începând cu a doua componentă până la final. La pasul i elementele a[1],a[2],..,a[i-1] sunt deja sortate și trebuie să plasăm elementul a[i] printre elementele deja sortate. Pentru a plasa elementul a[i] vom parcurge următorii pași :

- *Se reține a[i] într-o variabilă temporară t;*
- *Indicele j parcurge în ordine inversă elementele deja sortate;*
- *Cât timp a[j]>t și j>0, elementul a[j] se deplasează la dreapta cu o unitate, iar j descrește cu o unitate;*
- *În final t se va poziționa pe următoarea componentă pentru care nu a mai fost adevărată condiția: a[j+1]=t .*

Timp mediu	Timp la limită	Memorie	Stabilitate
$O(N^2)$	$O(N^2)$	O(1)	DA

Implementare:

```
void insertionSort(int a[], int n){
    int i, j, aux;
    for (i = 1; i < n; i++){
        j = i;
        while (j > 0 && a[j - 1] > a[j]){
            aux = a[j]; a[j] = a[j - 1]; a[--j] = aux;
        }
    }
}
```

Exemplu practic pas cu pas

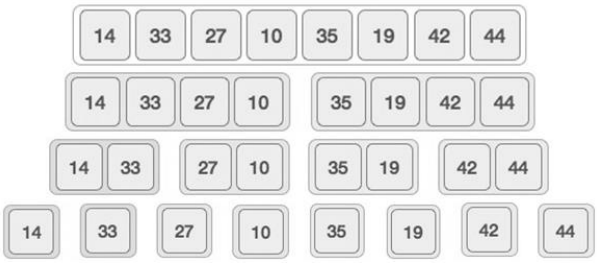
Se dă un vector ce conține valori întregi distincte, V={5, 7, 9, 4, 2}. Să se sorteze elementele vectorului prin cele trei metode menționate mai sus. Să se precizeze fiecare pas pentru a sorta elementele vectorului în ordine crescătoare ***[42]*** .

Bubble Sort	Selection Sort	Insertion Sort
Schimbare 1 5 7 9 4 2 Schimbare 2 5 7 4 9 2 Schimbare 3 5 7 4 2 9 Schimbare 4 5 4 7 2 9 Schimbare 5 5 4 2 7 9 Schimbare 6 4 5 2 7 9 Schimbare 7 4 2 5 7 9 Schimbare 8 2 4 5 7 9	Schimbare 1 5 7 9 4 2 Schimbare 2 5 7 2 4 9 Schimbare 3 5 4 2 7 9 Schimbare 4 2 4 5 7 9	Schimbare 1 5 7 9 4 2 Schimbare 2 5 7 4 9 2 Schimbare 3 5 4 7 9 2 Schimbare 4 4 5 7 9 2 Schimbare 5 4 5 7 2 9 Schimbare 6 4 5 2 7 9 Schimbare 7 4 2 5 7 9 Schimbare 8 2 4 5 7 9
Notă: • *Numărul total de pași rezlizați este: 32*	***Notă:*** • *Numărul total de pași rezlizați este: 41*	***Notă:*** • *Numărul total de pași rezlizați este: 18*

6.3 Tehnici de sortare indirecte

Sortarea prin interclasare *(MergeSort)*

Timp mediu	Timp la limită	Memorie	Stabilitate
O(N*logN)	O(N*logN)	O(N)	DA

Descriere	Exemplu
În cazul sortării prin interclasare, vectorii care se interclasează sunt două secvenţe ordonate din acelaşi vector. Sortarea prin interclasare utilizează metoda ***Divide et Impera****:* ❖ *se împarte vectorul în secvenţe din ce în ce mai mici, astfel încât fiecare secvenţă să fie ordonată la un moment dat şi interclasată cu o altă secvenţă din vector corespunzătoare.* ❖ *practic, interclasarea va începe când se ajunge la o secvenţă formată din două elemente. Aceasta, odată ordonată, se va interclasa cu o alta corespunzătoare(cu 2 elemente). Cele două secvenţe vor alcătui un subşir ordonat din vector mai mare(cu 4 elemente) care, la rândul lui, se va interclasa cu un subşir corespunzător(cu 4 elemente) ş.a.m.d.*	***Etapa 1***  ***Etapa 2*** 

Sortarea prin asamblare *(HeapSort)*

Timp mediu	Timp la limită	Memorie	Stabilitate
O(N*logN)	O(N*logN)	O(1)	NU

Există mai mutle tipuri de heap, dar ne vom referi numai la binary heap pentru implementarea algoritmului Heap sort. Un heap binar este un arbore binar cu următoarele proprietăţi:

- *este „complet"(toate nivelele sunt pline, cu posibila excepţie a ultimului nivel), adică de înălţime minimă*
- *există aceeaşi relaţie de ordine între orice nod şi părintele acestuia(excepţie - nodul rădăcină).*

Dacă nodurile conţin numere întregi după care stabilim relaţia de ordine, heap-ul poate fi de două feluri:

- *max-heap(rădăcina are cel mai mare număr, de la orice copil la părinte avem relaţia mai mic sau egal)*
- *min-heap(rădăcina are cel mai mic număr, de la orice copil la părinte avem relaţia mai mare sau egal)*

Descriere :

- *Metoda de sortare prin selecţie directă se bazează pe selecţia repetată a ultimei chei dintre n elemente, apoi dintre n-1 elemente rămase, etc.*
- *Pentru a găsi cea mai mică cheie dintre n elemente sunt necesare n-1 comparaţii, apoi găsirea următoarei dintre n-1 elemente are nevoie de n-2 comparaţii, etc.* ⇒ ***n(n-1)/2*** *comparaţii.*
- *Această sortare se poate îmbunătăţi prin reţinerea, de la fiecare scanare, de mai multă informaţie decât identificarea unui singur element, cel mai mic.*
- *De exemplu, cu n/2 comparaţii se poate determina cheia mai mică pentru fiecare pereche de elemente dintre cele n elemente, apoi cu alte n/4 comparaţii se poate determina cheia cea mai mică pentru fiecare pereche ale cheilor determinate anterior, şi aşa mai departe. Astfel, cu* ***n-1*** *comparaţii se poate construi arborele de selecţie.*

Descriem următorii paşi pentru o variantă de implementare a algoritmului Heap sort(ordonare crescătoare):

- *presupunem că vectorul formează un arbore binar, fiecare poziţie din vector reprezentând un nod, cu **rădăcina** pe poziţia 0(**zero**) şi cu fiecare **nod k** având copiii **2k+1** şi **2k+2**(dacă nu există poziţia din vector cu indicele respectiv, atunci nu există nod copil ⇒ NULL)*
- *formăm un **max-heap** cu aceeaşi reprezentare(pe vector, fără a construi altă structură pentru noduri)*
- *extragem maximul din rădăcina heap-ului(poziţia 0 din vector) şi facem o intersschimbare între poziţia maximului şi ultima poziţie din vector. Acum maximul se află pe poziţia dorită şi putem să îl excludem din heap.*
- *repetăm paşii(refacem forma de heap, extragem noul maxim, reducem cu 1 numărul de elemente nesortate), cât timp mai sunt elemente în heap.*

Definim următoarele două funcţii pentru a prezenta mai uşor algoritmul Heap sort:

- *funcţia „**cerne**“(asigură „cernerea“/„scurgerea“/„căderea“ nodului k până poziţia necesară pentru heap) - dacă nodul k nu are valoarea mai mare decât a copiilor lui(nu se păstrează relaţia de ordine a heap-ului), atunci nodul k va fi „cernut“ până va fi respectată relaţia.*
- *funcţia „**makeHeap**“(formează max heap-ul) - funcţia cerne toate nodurile pentru a obţine un heap **[43]**.*

Sortarea prin inserţie cu pas variabil *(ShellSort)*

Timp mediu	Timp la limită	Memorie	Stabilitate
O(N*log^{2}N)	O(N*log^{2}N)	O(1)	NU

Descriere	Exemplu
Algoritmul shell sort este o generalizare a algoritmului insertion sort. ❖ *La algoritmul insertion sort, pentru a insera un nou element în lista de elemente deja sortate, se deplasează fiecare element cu **câte o poziţie** spre dreapta atât timp cât avem elemente mai mari decât el. Practic, fiecare element înaintează spre poziţia sa finală cu câte o poziţie.* ❖ *Algoritmul shell sort lucrează similar, doar că deplasează elementele spre poziţia finală **cu mai mult de o poziţie**. Se lucrează în iteraţii.* ❖ *În prima iteraţie se aplică un insertion sort cu salt s1 mai mare decât 1. Asta înseamnă că fiecare element din şirul iniţial este deplasat spre stânga cu câte s1 poziţii atât timp cât întâlneste elemente mai mari decât el.* ❖ *Se repetă asemenea iteraţii cu salturi din ce în ce mai mici s2, s3, s4, etc. Ultima iteraţie se face cu saltul 1. Această ultimă iteraţie este, practic, un insetion sort clasic.* ❖ *Principiul este că, după fiecare iteraţie, şirul devine din ce în ce "mai sortat". Iar, cum algoritmul insertion sort funcţionează cu atât mai repede cu cât sirul este mai sortat, per ansamblu, vom obţine o îmbunătăţire de viteză.*	

Sortarea rapidă *(QuickSort)*

Timp mediu	Timp la limită	Memorie	Stabilitate
O(N*logN)	$O(N^2)$	O(logN)	NU

Descriere :

- ❖ *Quick Sort este unul dintre cei mai rapizi şi mai utilizaţi algoritmi de sortare până în acest moment,bazându-se pe tehnica „Divide et impera“.*
- ❖ *Deşi cazul cel mai nefavorabil este $O(N^2)$, în practică, QuickSort oferă rezultate mai bune decât restul algoritmilor de sortare din clasa „O(N log N)“.*

Algoritmul se bazează pe următorii paşi:

- *alegerea unui element pe post de pivot;*
- *parcurgerea vectorului din două părţi(de la stânga la pivot, de la dreapta la pivot, ambele în acelaşi timp);*
- *interschimbarea elementelor care se află pe „partea greşită“ a pivotului(mutăm la dreapta pivotului elementele mai mari, la stânga pivotului elementel mai mici);*
- *divizarea algoritmului: după ce mutăm elementele pe „partea corectă“ a pivotului, avem 2 subşiruri de sortat, iar pivotul se află pe poziţia bună.*

Urmează algoritmul QuickSort pentru Pivot

Pasul 1 - Alegeți cea mai mare valoare a indexului are pivot
Pasul 2 - Ia două variabile pentru a indica stânga și dreapta din listă, cu excepția pivotului
Pasul 3 - punct stânga către indicele scăzut
Pasul 4 - punct corect către înălțime
Pasul 5 - în timp ce valoarea din stânga este mai mică decât pivotul se deplasează spre dreapta
Pasul 6 - în timp ce valoarea din dreapta este mai mare decât deplasarea pivotului la stânga
Pasul 7 - dacă atât pasul 5 cât și pasul 6 nu se potrivesc cu schimbarea la stânga și la dreapta
Pasul 8 - dacă stânga ≥ dreapta, punctul în care s-au întâlnit este pivotul nou

Urmează algoritmul QuickSort

Pasul 1 - Faceți pivotul cu cea mai bună valoare a indexului
Pasul 2 - partiți tabloul folosind valoarea pivot
Pasul 3 - quicksort partiție stângă recursiv
Pasul 4 - quicksort partiție dreapta recursiv

Notă:

- ✓ *Nu există restricţii pentru alegerea pivotului.*
- ✓ *Algoritmul prezentat alege mereu elementul din mijloc.*
- ✓ *Quicksort este un algoritm de sortare eficient în loc, care, de obicei, se execută de aproximativ două până la trei ori mai rapid decât MergeSort și HeapSort atunci când este implementat bine.*
- ✓ *Quicksort este un fel de comparație, ceea ce înseamnă că poate sorta elemente de orice tip pentru care este definită o relație mai mică decât aceea. În implementări eficiente, de obicei, nu este stabil.*
- ✓ *Quicksort are în medie O (N*logN) comparații pentru a sorta n elemente. În cel mai rău caz, face comparații O (n), deși acest comportament este foarte rar* ***[44]*** .

Analiza comparativă dintre *MergeSort* și *QuickSort*

MergeSort este un algoritm extern care se bazează pe strategia Divide et Impera. În acest algoritm:

- *Elementele sunt împărțite în două submulțimi (n / 2) din nou și din nou, până când rămâne doar un element.*
- *MergeSort folosește stocarea suplimentară pentru sortarea tabloului auxiliar.*
- *MergeSort folosește trei tablouri în care două sunt utilizate pentru stocarea fiecărei jumătăți, iar cel de-al treilea extern este folosit pentru a stoca lista finală sortată prin comasarea altor două și fiecare tablou este apoi sortat recursiv.*
- *În cele din urmă, toate sub-tablourile sunt îmbinate pentru a face „n" - dimensiunea elementului tabloului.*

QuickSort este un algoritm intern care se bazează pe strategia Divide et Impera. În acest algoritm:

- *Gama de elemente este împărțită în părți în mod repetat, până când nu este posibil să o împărțim în continuare.*
- *Este, de asemenea, cunoscut sub numele de "sortare de schimb de partiții".*
- *Folosește un element cheie (pivot) pentru împărțirea elementelor.*
- *O partiție stângă conține toate acele elemente care sunt mai mici decât pivotul și o partiție din dreapta conține toate acele elemente care sunt mai mari decât elementul cheie* ***[41]***.

CRITERIUL	QUICK SORT	MERGE SORT
Partiția elementelor din tablou	*Împărțirea unui vector de elemente este în orice raport, nu neapărat împărțit în jumătate.*	*Împărțirea unui vector de elemente este în orice raport, nu neapărat împărțit în jumătate.*
Complexitatea cea mai rea	$O(N^2)$	*O(N*logN)*
Funcționează bine	*Funcționează bine pe un vector de dimensiune mai mică*	*Funcționează bine pe un vector de orice dimensiune*
Viteza de execuție	*Funcționează mai repede decât alți algoritmi de sortare pentru vectori de dimensiuni mai mici precum sortare SelectionSort etc.*	*Are o viteză constantă pentru vectori de orice dimensiune*
Cerință suplimentară de spațiu de stocare	*Mai puțin (în loc)*	*Mai mult (nu în loc)*
Eficiență	*Ineficient pentru vectori de dimensiuni mai mari*	*Mai eficient*
Metoda de sortare	*Intern*	*Extern*
Stabilitate	*Instabil*	*Stabil*
Preferat	*Pentru vectori*	*Pentru Liste de legături*
Localitatea de referință	*bun*	*slab*

6.4 Implementarea tehnicilor de sortare directe

Problema 1: Sortarea cu bule (BubbleSort)

Exemplu
Fie avem vectorul: arr [] = { 5 1 4 2 8 }. Trebuie de sortat elementele ascendant. *Soluție:* • (**5 1** 4 2 8)->(**1 5** 4 2 8), (1 **5 4** 2 8)->(1 **4 5** 2 8), (1 4 **5 2** 8)->(1 4 **2 5** 8), (1 4 2 **5 8**)->(1 4 2 **5 8**). • (**1 4** 2 5 8)->(**1 4** 2 5 8), (1 **4 2** 5 8)->(1 **2 4** 5 8), (1 2 **4 5** 8)->(1 2 **4 5** 8), (1 2 4 **5 8**)->(1 2 4 **5 8**). • (**1 2** 4 5 8)->(**1 2** 4 5 8), (1 **2 4** 5 8)->(1 **2 4** 5 8), (1 2 **4 5** 8)->(1 2 **4 5** 8), (1 2 4 **5 8**)->(1 2 4 **5 8**).

Varianta clasică

```
#include <iostream>
using namespace std;
int i,j;
void swap(int *xp, int *yp){
    int temp = *xp;
    *xp = *yp;
    *yp = temp;
}
void bubbleSort1(int arr[], int n){
    for (i = 0; i < n-1; i++)
    for (j = 0; j < n-i-1; j++)
        if (arr[j] > arr[j+1])
            swap(&arr[j], &arr[j+1]);
}
void bubbleSort2(int arr[], int n){
    for (i = 0; i < n-1; i++)
    for (j = 0; j < n-i-1; j++)
        if (arr[j] < arr[j+1])
            swap(&arr[j], &arr[j+1]);
}
void afisare(int arr[], int size){
    for (i = 0; i < size; i++)
        cout<<arr[i]<<" "; cout<<endl;}
int main(){
    int arr[] = {5, 7, 9, 4, 2};
    int n = sizeof(arr)/sizeof(arr[0]);
    cout<<"Vectorul introdus: \n\t";
    afisare(arr, n);
    bubbleSort1(arr, n);
    cout<<"Vectorul sortat crescator:
\n\t";
    afisare(arr, n);
    bubbleSort2(arr, n);
    cout<<"Vectorul sortat descrescator:
\n\t";
    afisare(arr, n); return 0;
}
```

Rezultatele execuției:

```
Vectorul introdus:
        5 7 9 4 2
Vectorul sortat crescator:
        2 4 5 7 9
Vectorul sortat descrescator:
        9 7 5 4 2
```

Varianta recursivă

```
#include <iostream>
using namespace std;
void bubbleSort1(int arr[], int n){
    if (n == 1) return;
    for (int i=0; i<n-1; i++)
        if (arr[i] > arr[i+1])
            swap(arr[i], arr[i+1]);
    bubbleSort1(arr, n-1);
}
void bubbleSort2(int arr[], int n){
    if (n == 1) return;
    for (int i=0; i<n-1; i++)
        if (arr[i] < arr[i+1])
            swap(arr[i], arr[i+1]);
    bubbleSort2(arr, n-1);
}
void afisare(int arr[], int n){
    for (int i=0; i < n; i++)
        cout<<arr[i]<<" "; cout<<endl;}
int main(){
    int arr[] = {5, 7, 9, 4, 2};
    int n = sizeof(arr)/sizeof(arr[0]);
    cout<<"Vectorul introdus: \n\t";
    afisare(arr, n);
    bubbleSort1(arr, n);
    cout<<"Vectorul sortat crescator:
\n\t";
    afisare(arr, n);
    bubbleSort2(arr, n);
    cout<<"Vectorul sortat descrescator:
\n\t";
    afisare(arr, n); return 0;
}
```

Rezultatele execuției:

```
Vectorul introdus:
        5 7 9 4 2
Vectorul sortat crescator:
        2 4 5 7 9
Vectorul sortat descrescator:
        9 7 5 4 2
```

Problema 2: Sortarea prin selecție (SelectionSort)

Exemplu
Fie avem vectorul: arr [] = { 64 25 12 22 11 }. Trebuie de sortat elementele ascendant. *Soluție:* • *Găsim elementul minim în arr [0 ... 4] și-l așezăm la început: 11 25 12 22 64.* • *Găsim elementul minim în arr [1 ... 4] și-l așezăm la început: 11 12 25 22 64.* • *Găsim elementul minim în arr [2 ... 4] și-l așezăm la început: 11 12 22 25 64.* • *Găsim elementul minim în arr [3 ... 4] și-l așezăm la început: 11 12 22 25 64*

Varianta clasică

```
#include <iostream>
using namespace std;
void swap(int *xp, int *yp){
    int temp = *xp;
    *xp = *yp;
    *yp = temp;
}
void selectionSort1(int arr[], int n){
    int i, j, min_idx;
    for (i = 0; i < n-1; i++){
        min_idx = i;
        for (j = i+1; j < n; j++)
        if (arr[j] < arr[min_idx])
            min_idx = j;
        swap(&arr[min_idx], &arr[i]);
    }
}
void selectionSort2(int arr[], int n){
    int i, j, min_idx;
    for (i = 0; i < n-1; i++){
        min_idx = i;
        for (j = i+1; j < n; j++)
        if (arr[j] > arr[min_idx])
            min_idx = j;
        swap(&arr[min_idx], &arr[i]);
    }
}
void afisare(int arr[], int size){
    int i;
    for (i=0; i < size; i++)
        cout << arr[i] << " ";
    cout << endl;
}
int main(){
    int arr[] = {5, 7, 9, 4, 2};
    int n = sizeof(arr)/sizeof(arr[0]);
    selectionSort1(arr, n);
    cout<<"Vectorul sortat crescator:
\n\t";
    afisare(arr, n);
    selectionSort2(arr, n);
    cout<<"Vectorul sortat descrescator:
\n\t";
    afisare(arr, n);
}
```

Rezultatele execuției:

```
Vectorul sortat crescator:
        2 4 5 7 9
Vectorul sortat descrescator:
        9 7 5 4 2
```

Varianta recursivă

```
#include <iostream>
using namespace std;
int i;
int minIndex1(int a[], int i, int j){
    if (i == j) return i;
    int k = minIndex1(a, i + 1, j);
    return (a[i] < a[k])? i : k;
}
int minIndex2(int a[], int i, int j){
    if (i == j) return i;
    int k = minIndex2(a, i + 1, j);
    return (a[i] > a[k])? i : k;
}
void SelectionSort1(int a[], int n, int
index=0){
    if (index == n) return;
    int k = minIndex1(a, index, n-1);
    if (k != index)
       swap(a[k], a[index]);
    SelectionSort1(a, n, index + 1);
}
void SelectionSort2(int a[], int n, int
index=0){
    if (index == n) return;
    int k = minIndex2(a, index, n-1);
    if (k != index)
       swap(a[k], a[index]);
    SelectionSort2(a, n, index + 1);
}
void afisare(int a[], int size){
    for (i = 0; i < size; i++)
        cout << a[i] << " ";
        cout << endl;
}
int main(){
    int a[] = {5, 7, 9, 4, 2};
    int n = sizeof(a)/sizeof(a[0]);
    SelectionSort1(a, n);
    cout<<"Vectorul sortat crescator:
\n\t";
    afisare(a, n);
    SelectionSort2(a, n);
    cout<<"Vectorul sortat descrescator:
\n\t";
    afisare(a, n);
}
```

Rezultatele execuției:

```
Vectorul sortat crescator:
        2 4 5 7 9
Vectorul sortat descrescator:
        9 7 5 4 2
```

Problema 3: Sortarea prin inserție (InsertionSort)

Exemplu

Fie avem vectorul: a [] = { 12 11 13 5 6 }. Trebuie de sortat elementele ascendant.
Soluție:

Începem parcurgerea de la al doilea indice (1) până la ultimul (4):

- i = 1. Deoarece 11 este mai mic decât 12, deplasăm 12 și introducem 11 înainte de 12, obținem: 11, 12, 13, 5, 6
- i = 2. Numărul 13 va rămâne la poziția sa, deoarece toate elementele din A [0..i-1] sunt mai mici decât 13, obținem: 11, 12, 13, 5, 6
- i = 3. Numărul 5 se va muta la început și toate celelalte elemente de la 11 la 13 vor muta o poziție înaintea poziției lor actuale, obținem: 5, 11, 12, 13, 6
- i = 4. Numărul 6 se va muta pe poziția după 5, iar elementele de la 11 la 13 vor muta o poziție înaintea poziției lor actuale, obținem: 5, 6, 11, 12, 13.

Varianta clasică

```
#include <iostream>
using namespace std;
int i;
void insertionSort1(int a[], int n){
    int i, key, j;
    for (i = 1; i < n; i++){
        key = a[i]; j = i - 1;
        while (j >= 0 && a[j] > key) {
            a[j + 1] = a[j];
            j = j - 1;
        } a[j + 1] = key;
     }
}
void insertionSort2(int a[], int n){
    int i, key, j;
    for (i = 1; i < n; i++){
        key = a[i]; j = i - 1;
        while (j >= 0 && a[j] < key) {
            a[j + 1] = a[j];
            j = j - 1;
        }  a[j + 1] = key;
    }
}
void afisare(int a[], int n){
    for (i = 0; i < n; i++)
        cout<<a[i]<<" "; cout<<endl;
}
int main(){
    int a[] = {5, 7, 9, 4, 2};
    int n = sizeof(a)/sizeof(a[0]);
    insertionSort1(a, n);
    cout<<"Vectorul sortat crescator:
\n\t";
    afisare(a, n);
    insertionSort2(a, n);
    cout<<"Vectorul sortat descrescator:
\n\t";
    afisare(a, n); return 0;
}
```

Rezultatele execuției:

```
Vectorul sortat crescator:
        2 4 5 7 9
Vectorul sortat descrescator:
        9 7 5 4 2
```

Varianta recursivă

```
#include <iostream>
using namespace std;
void insertionSort1(int a[], int n){
    if (n <= 1) return;
    insertionSort1( a, n-1);
    int last = a[n-1];
    int j = n-2;
    while (j >= 0 && a[j] > last){
        a[j+1] = a[j];
        j--;
    } a[j+1] = last;
}
void insertionSort2(int a[], int n){
    if (n <= 1) return;
    insertionSort2(a, n-1);
    int last = a[n-1];
    int j = n-2;
    while (j >= 0 && a[j] < last){
        a[j+1] = a[j];
        j--;
    } a[j+1] = last;
}
void afisare(int a[], int n){
    for (int i=0; i < n; i++)
        cout<<a[i]<<" "; cout<<endl;
}
int main() {
    int a[] = {5, 7, 9, 4, 2};
    int n = sizeof(a)/sizeof(a[0]);
    insertionSort1(a, n);
    cout<<"Vectorul sortat crescator:
\n\t";
    afisare(a, n);
    insertionSort2(a, n);
    cout<<"Vectorul sortat descrescator:
\n\t";
    afisare(a, n); return 0;
}
```

Rezultatele execuției:

```
Vectorul sortat crescator:
        2 4 5 7 9
Vectorul sortat descrescator:
        9 7 5 4 2
```

6.5 Implementarea tehnicilor de sortare indirecte

Problema 1: Sortarea prin interclasare (MergeSort)

Exemplu	
Fie avem vectorul: a [] = { 43 16 60 27 45 }. *Trebuie de sortat elementele ascendent.* *Soluție:* *Pasul 1: 16 43* *Pasul 2: 16 43 60* *Pasul 3: 27 45* *Pasul 4: 16 27 43 45 60*	*Fie avem vectorul: a [] = { 43 16 60 27 45 }.* *Trebuie de sortat elementele descendent.* *Soluție:* *Pasul 1: 43 16* *Pasul 2: 60 43 16* *Pasul 3: 45 27* *Pasul 4: 60 45 43 27 16*

Implementare C++

```
#include <iostream>
#include <stdlib.h>
using namespace std;
/// sortare ascendenta
void combina1(int a[], int l, int m, int r) {
    int i, j, k, n1 = m - l + 1, n2 =  r - m;
    int L[n1], R[n2];  /* cream tablouri temporare */
    /* Copiam datele in tablourile temporare L[] si R[] */
    for (i = 0; i < n1; i++) L[i] = a[l + i];
    for (j = 0; j < n2; j++) R[j] = a[m + 1+ j];
    /* Unirea celor doua tablouri temporare*/
    i = 0; j = 0; k = l;
    while (i < n1 && j < n2) {
        if (L[i] <= R[j]) { a[k] = L[i]; i++; }
        else { a[k] = R[j]; j++; }
        k++;
    }
    /* Copiam elementele ramase din L [], daca exista */
    while (i < n1) { a[k] = L[i]; i++; k++; }
    /* Copiam elementele ramase din R [], daca exista */
    while (j < n2) { a[k] = R[j]; j++; k++; }
}
void mergeSort1(int a[], int l, int r) {
    if (l < r){
        int m=l+(r-l)/2; mergeSort1(a,l,m); mergeSort1(a,m+1,r); combina1(a,l,m,r);
    }
}
/// sortare descendenta
void combina2(int a[], int l, int m, int r) {
    int i, j, k, n1 = m - l + 1, n2 =  r - m;
    int L[n1], R[n2];  /* cream tablouri temporare */
    /* Copiam datele in tablourile temporare L[] si R[] */
    for (i = 0; i < n1; i++) L[i] = a[l + i];
    for (j = 0; j < n2; j++) R[j] = a[m + 1+ j];
    /* Unirea celor doua tablouri temporare*/
    i = 0; j = 0; k = l;
    while (i < n1 && j < n2) {
        if (L[i] >= R[j]) { a[k] = L[i]; i++; }
        else { a[k] = R[j]; j++; }
        k++;
    }
    /* Copiam elementele ramase din L [], daca exista */
    while (i < n1) { a[k] = L[i]; i++; k++; }
    /* Copiam elementele ramase din R [], daca exista */
    while (j < n2) { a[k] = R[j]; j++; k++; }
}
void mergeSort2(int a[], int l, int r) {
    if (l < r){
        int m=l+(r-l)/2; mergeSort2(a,l,m); mergeSort2(a,m+1,r); combina2(a,l,m,r);
    }
}
```

```
void afisare(int A[], int size){
    int i;
    for (i=0; i < size; i++)
        cout<<A[i]<<"\t"; cout<<endl;
}
int main() {
    int i,n, a[20];
    cout<<"Introdu dimensiunea vectorului: "; cin>>n;
    cout<<"Introdu elementele vectorului: \n";
    for (i=0; i<n; i++){
        cin>>a[i];
    }
    system("cls");
    cout<<"Vectorul introdus este: \n\t"; afisare(a, n);
    mergeSort1(a, 0, n-1);
    cout<<"Vectorul sortat crescator este: \n\t"; afisare(a, n);
    mergeSort2(a, 0, n-1);
    cout<<"Vectorul sortat descrescator este: \n\t"; afisare(a, n);
    return 0;
}
```

Rezultatele execuției:

```
Vectorul introdus este:
        43      16      60      27      45
Vectorul sortat crescator este:
        16      27      43      45      60
Vectorul sortat descrescator este:
        60      45      43      27      16
```

Problema 2: Sortarea prin asamblare (HeapSort)

Exemplu	
Fie avem vectorul: a [] = { 30 13 64 27 47 }. *Trebuie de sortat elementele ascendent.* *Soluție:* *Pasul 1: 47 27 30 13 64* *Pasul 2: 30 27 13 47 64* *Pasul 3: 27 13 30 47 64* *Pasul 4: 13 27 30 47 64* *Pasul 5: 13 27 30 47 64*	*Fie avem vectorul: a [] = { 30 13 64 27 47 }.* *Trebuie de sortat elementele descendent.* *Soluție:* *Pasul 1: 64 13 30 27 47* *Pasul 2: 64 47 13 27 30* *Pasul 3: 64 47 30 13 27* *Pasul 4: 64 47 30 27 13* *Pasul 5: 64 47 30 27 13*

Implementare C++

```
#include <iostream>
#include <stdlib.h>
using namespace std;
/*Pentru a heapifica un subarbore in radacinata cu nodul i care este un index in
vectorul a[], n este marimea heapului. */
/// sortare ascendenta
void heapify1(int a[], int n, int i) {
    int z = i, l = 2*i + 1, r = 2*i + 2;
    // Daca copilul stang este mai mare decat radacina
    if (l < n && a[l] > a[z]) z = l;
    // Daca copilul drept este mai mare decat cea mai mare de pana acum
    if (r < n && a[r] > a[z]) z = r;
    // Daca cea mai mare de pana acum nu este radacina
    if (z != i) {
        swap(a[i], a[z]);
        heapify1(a, n, z);
    }
}
void heapSort1(int a[], int n){
    // Construim heapul (rearanjam vectorul)
    for (int i = n / 2 - 1; i >= 0; i--)
        heapify1(a, n, i);
    for (int i=n-1; i>=0; i--){
        swap(a[0], a[i]); // Mutam radacina curenta la sfarsit
        heapify1(a, i, 0); // Apelam max heapify pe heapul redus
    }
}
/// sortare descendenta
void heapify2(int a[], int n, int i) {
    int z = i, l = 2*i + 1, r = 2*i + 2;
    // Daca copilul stang este mai mic decat radacina
    if (l < n && a[l] < a[z]) z = l;
    // Daca copilul drept este mai mic decat cea mai mica de pana acum
    if (r < n && a[r] < a[z]) z = r;
    // Daca cea mai mica de pana acum nu este radacina
    if (z != i) {
        swap(a[i], a[z]);
        heapify2(a, n, z);
    }
}
void heapSort2(int a[], int n){
    // Construim heapul (rearanjam vectorul)
    for (int i = n / 2 - 1; i >= 0; i--)
        heapify2(a, n, i);
    for (int i=n-1; i>=0; i--){
        swap(a[0], a[i]); // Mutam radacina curenta la sfarsit
        heapify2(a, i, 0); // Apelam max heapify pe heapul redus
    }
}
void afisare(int a[], int n){
    for (int i=0; i<n; ++i)
        cout << a[i] << "\t";
        cout << "\n";
}
```

```
int main(){
    int i,n, a[20];
    cout<<"Introdu dimensiunea vectorului: "; cin>>n;
    cout<<"Introdu elementele vectorului: \n";
    for (i=0; i<n; i++){
        cin>>a[i];
    }
    system("cls");
    cout<<"Vectorul introdus este: \n\t"; afisare(a, n);
    heapSort1(a, n);
    cout<<"Vectorul sortat crescator este: \n\t"; afisare(a, n);
    heapSort2(a, n);
    cout<<"Vectorul sortat descrescator este: \n\t"; afisare(a, n);
    return 0;
}
```

Rezultatele execuției:

```
Vectorul introdus este:
        30      13      64      27      47
Vectorul sortat crescator este:
        13      27      30      47      64
Vectorul sortat descrescator este:
        64      47      30      27      13
```

Problema 3: Sortarea prin inserție cu pas variabil (ShellSort)

Exemplu

Fie avem vectorul: a [] = { 23 12 52 27 40 }.
Trebuie de sortat elementele ascendent.
Soluție:

```
Pasul 1:    40 27 23 12 52
Pasul 2:    27 12 23 40 52
Pasul 3:    23 12 27 40 52
Pasul 4:    12 23 27 40 52
Pasul 5:    12 23 27 40 52
```

Fie avem vectorul: a [] = { 23 12 52 27 40 }.
Trebuie de sortat elementele descendent.
Soluție:

```
Pasul 1:    52 12 23 27 40
Pasul 2:    52 40 23 12 27
Pasul 3:    52 40 27 12 23
Pasul 4:    52 40 27 23 12
Pasul 5:    52 40 27 23 12
```

Implementare C++

```
#include  <iostream>
using namespace std;
/// sortare ascendenta
int shellSort1(int a[], int n){
    /// Incepem cu un decalaj mare, apoi reducem decalajul
    for (int decalaj = n/2; decalaj > 0; decalaj /= 2){
        for (int i = decalaj; i < n; i += 1) {
            int j, temp = a[i];
            for (j = i; j >= decalaj && a[j - decalaj] > temp; j -= decalaj)
                a[j] = a[j - decalaj];
            ///  Punem temp (originalul lui a[i]) in locatia corecta
            a[j] = temp;
        }
    }
}
/// sortare descendenta
int shellSort2(int a[], int n){
    /// Incepem cu un decalaj mare, apoi reducem decalajul
    for (int decalaj = n/2; decalaj > 0; decalaj /= 2){
        for (int i = decalaj; i < n; i += 1) {
            int j, temp = a[i];
            for (j = i; j >= decalaj && a[j - decalaj] < temp; j -= decalaj)
                a[j] = a[j - decalaj];
            ///  Punem temp (originalul lui a[i]) in locatia corecta
            a[j] = temp;
        }
    }
}
void afisare(int a[], int n){
    for (int i=0; i<n; i++)
        cout << a[i] <<"\t"; cout << "\n";
}
int main(){
    int i, a[] = {23, 12, 52, 27, 40}, n = sizeof(a)/sizeof(a[0]);
    cout<<"Vectorul introdus este: \n\t"; afisare(a, n);
    shellSort1(a, n);
    cout<<"Vectorul sortat crescator este: \n\t"; afisare(a, n);
    shellSort2(a, n);
    cout<<"Vectorul sortat descrescator este: \n\t"; afisare(a, n);
    return 0;
}
```

Rezultatele execuției:

```
Vectorul introdus este:
        23      12      52      27      40
Vectorul sortat crescator este:
        12      23      27      40      52
Vectorul sortat descrescator este:
        52      40      27      23      12
```

Problema 4: Sortarea rapidă (QuickSort)

Exemplu	
Fie avem vectorul: a [] = { 15 53 38 84 40 }. *Trebuie de sortat elementele ascendent.* *Soluție:* *Pasul 1: 15 38 40 84 53* *Pasul 2: 15 38 40 84 53* *Pasul 3: 15 38 40 53 84*	*Fie avem vectorul: a [] = { 15 53 38 84 40 }.* *Trebuie de sortat elementele descendent.* *Soluție:* *Pasul 1: 53 84 40 38 15* *Pasul 2: 53 84 40 38 15* *Pasul 3: 84 53 40 38 15*

Implementare C++

```
#include <iostream>
#include <stdlib.h>
using namespace std;
// O functie de utilitate pentru a schimba doua elemente
void swap(int* a, int* b){
    int t = *a; *a = *b; *b = t;
}
/* Aceasta functie ia ultimul element ca pivot, plaseaza elementul pivot la pozitia
sa corecta intr-un tablou sortat si plaseazs toate elementele mai mici decat
pivotul la stanga pivotului si toate elementele mai mari la dreapta pivotului. */
/// sortare ascendenta
int partitionare1 (int arr[], int l, int h){
    int pivot = arr[h]; // Pivotul
    int i = (l - 1); // Indicele elementului cel mai mic.
    for (int j = l; j <= h - 1; j++){
        // Daca elementul curent este mai mic decat pivotul.
        if (arr[j] < pivot) {
            i++; // Incrementarea indicelui elementului cel mai mic.
            swap(&arr[i], &arr[j]);
        }
    }
    swap(&arr[i + 1], &arr[h]); return (i + 1);
}
void quickSort1(int arr[], int x, int y){
    if (x < y)    {
        /* pi este indexul de partitionare, arr [p] este acum la locul potrivit */
        int pi = partitionare1(arr, x, y);
        // Sorteaza separat elementele inainte de partitionare si dupa partitionare
        quickSort1(arr, x, pi - 1); quickSort1(arr, pi + 1, y);
    }
}
/// sortare descendenta
int partitionare2 (int arr[], int l, int h){
    int pivot = arr[h]; // Pivotul
    int i = (l - 1); // Indicele elementului cel mai mare.
    for (int j = l; j <= h - 1; j++){
        // Daca elementul curent este mai mare decat pivotul.
        if (arr[j] > pivot) {
            i++; // Incrementarea indicelui elementului cel mai mare.
            swap(&arr[i], &arr[j]);
        }
    }
    swap(&arr[i + 1], &arr[h]); return (i + 1);
}
void quickSort2(int arr[], int x, int y){
    if (x < y)    {
        /* pi este indexul de partitionare, arr [p] este acum la locul potrivit */
        int pi = partitionare2(arr, x, y);
        // Sorteaza separat elementele inainte de partitionare si dupa partitionare
        quickSort2(arr, x, pi - 1); quickSort2(arr, pi + 1, y);
    }
}
void afiseaza(int arr[], int m){
    int i;
    for (i = 0; i < m; i++)
        cout << arr[i] << "\t"; cout << endl;
```

```
}
int main(){
    int i,n, a[20];
    cout<<"Introdu dimensiunea vectorului: "; cin>>n;
    cout<<"Introdu elementele vectorului: \n";
    for (i=0; i<n; i++){
        cin>>a[i];
    }
    system("cls");
    cout<<"Vectorul introdus este: \n\t"; afiseaza(a, n);
    quickSort1(a, 0, n - 1);
    cout<<"Vectorul sortat crescator este: \n\t"; afiseaza(a, n);
    quickSort2(a, 0, n - 1);
    cout<<"Vectorul sortat descrescator este: \n\t"; afiseaza(a, n);
    return 0;
}
```

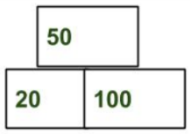
Rezultatele execuției:

```
Vectorul introdus este:
        15      53      38      84      40
Vectorul sortat crescator este:
        15      38      40      53      84
Vectorul sortat descrescator este:
        84      53      40      38      15
```

SARCINI PENTRU EXERSARE

1. *Creaţi un alt vector pentru salturi, de exemplu: {1e5,1e4,1e3,1e2,1e1,1}, unde 1ek = 10^k. Încercaţi să folosiţi acest vector pentru sortare şiobservaţi diferenţa de performanţă. Sortaţi vectorul cu cei trei algoritmi (metodele directe de sortare) şi comparaţi rezultatele.*
2. *Creaţi un alt vector pentru salturi, de exemplu: {1e5,1e4,1e3,1e2,1e1,1}, unde 1ek = 10^k. Încercaţi să folosiţi acest vector pentru sortare şiobservaţi diferenţa de performanţă. Sortaţi vectorul cu cei patru algoritmi (metodele indirecte de sortare) şi comparaţi rezultatele.*
3. *Alegeţi un algoritm A(dintre Bubble, Insertion şiSelection) şiun algoritm B(dintre Merge şi Quick). Introduceţi niştevariabile globale cu care să contorizaţi numărul de comparaţii pentru algoritmii A şi B. Comparaţirezultatele pentru un vector de întregi de lungime n = 20.*
4. *Introduceţi o variabilă globală cu care să contorizaţi numărul de apelări ale funcţiei „cerne“. Afişaţinumărul de apelări necesare pentru construirea heap-ului(makeHeap) şinumărul de apelări necesare pentru tot algoritmul(heapSort). Ce observaţi referitor la complexitatea funcţiilor?*
5. *Implementaţi un algoritm(dintre Bubble, Insertion şiSelection) pentru sortarea unui vector cu n cuvinte de maxim 4 litere fiecare.*
6. *Implementaţi un algoritm(dintre Merge şiQuick) pentru sortarea unui vector de structuri, unde fiecare structură reprezintă un moment de timp(int ora,min,sec).*
7. *Se dă un vector de n întregi, iar toate valorile din vector sunt între 0 şi1000. Sortaţi vectorul în timp O(n).*
8. *Fiind date două tablouri A1 [] şi A2 [], sortaţi A1 astfel încât ordinea relativă dintre elemente să fie aceeaşi cu cele din A2. Pentru elementele care nu sunt prezente în A2, adăugaţi-le în cele din urmă în ordine ordonată (ascendent sau descendent).*
9. *Fiind dat un tablou de dimensiunea k şi un număr x,sortaţi acest tablou, apoi găsiţi o pereche în tablou a cărei sumă este cea mai apropiată de x.*
10. *Fiind dat un tablou de numere întregi nesortate, sortaţi tabloul într-un nou tablou în formă de undă. Un tablou 'a [0..n-1]' este sortat în formă de undă dacă: a [0]> = a[1] <= a [2]> = a [3] <= a [4]> = ...*
11. *Fie că aveţi un tablou de dimensiunea k, ce conţine infirmaţii despre data de naştere a celor k elevi. Cum veţi proceda pentru a sorta datele respective în forma ascendentă / descendentă **[20]**.*
12. *Fie că aveţi un tablou de dimensiunea k şi conţine exact k elemente distincte, găsiţi numărul minim de schimbări (mutări) necesare pentru a sorta tabloul.*
13. *Fiind date două tablouri care au aceleaşi valori, dar în ordine diferită, trebuie să facem ca al doilea tablou să fie la fel ca primul tablou, folosind un număr minim de schimbări (mutări).*
14. *Fie că aveţi o matrice pătrată de ordinul N * N având elemente distincte. Sarcina este de a sorta matricea dată, astfel încât rândurile, coloanele şi ambele diagonale (diagonală şi anti-diagonală) să fie în ordine crescătoare / descrescătoare.*
15. *Fiind date n obiecte, fiecare obiect are lăţime wi. Trebuie să le aranjăm într-un mod piramidal, astfel încât:*
    - *Lăţimea totală a i-lea este mai mică decât (i + 1)-lea.*
    - *Numărul total de obiecte în a i-lea este mai mic decât (i + 1)-lea.*

    *Găsi înălţimea maximă care poate fi obţinută din obiectele date **[24]**.*



6.6 Elaborarea unei biblioteci pentru tehnicile de sortare

1. *Elaborarea unei biblioteci cu numele vsortdirect.h, care va implementa toate cele trei tehnici de sortare directe studiate pentru sortarea elementelor unui vector în ordine ascendentă și descendentă.*

Biblioteca cu numele vsortdirect.h
Implementare C++

```
#ifndef VSORTDIRECT_H_INCLUDED
#define VSORTDIRECT_H_INCLUDED
using namespace std;
int i,j;
void afisare(int arr[], int size){
    for (i = 0; i < size; i++)
    cout<<arr[i]<<" "; cout<<endl;
}
void swap(int *xp, int *yp){
    int temp = *xp; *xp = *yp; *yp = temp;
}
/// BubbleSort crescator
void bubbleSort1(int arr[], int n){
    for (i = 0; i < n-1; i++)
    for (j = 0; j < n-i-1; j++)
        if (arr[j] > arr[j+1]) swap(&arr[j], &arr[j+1]);
}
/// BubbleSort descrescator
void bubbleSort2(int arr[], int n){
    for (i = 0; i < n-1; i++)
    for (j = 0; j < n-i-1; j++)
        if (arr[j] < arr[j+1]) swap(&arr[j], &arr[j+1]);
}
/// SelectionSort crescator
void selectionSort1(int arr[], int n){
    int i, j, min_idx;
    for (i = 0; i < n-1; i++){
        min_idx = i;
        for (j = i+1; j < n; j++)
        if (arr[j] < arr[min_idx]) min_idx = j; swap(&arr[min_idx], &arr[i]);
    }
}
/// SelectionSort descrescator
void selectionSort2(int arr[], int n){
    int i, j, min_idx;
    for (i = 0; i < n-1; i++){
        min_idx = i;
        for (j = i+1; j < n; j++)
        if (arr[j] > arr[min_idx]) min_idx = j; swap(&arr[min_idx], &arr[i]);
    }
}
/// InsertionSort crescator
void insertionSort1(int arr[], int n){
    int i, key, j;
    for (i = 1; i < n; i++){
        key = arr[i]; j = i - 1;
        while (j >= 0 && arr[j] > key) {
            arr[j + 1] = arr[j]; j = j - 1;
        }
        arr[j + 1] = key;
     }
}
/// InsertionSort descrescator
void insertionSort2(int arr[], int n){
    int i, key, j;
    for (i = 1; i < n; i++){
        key = arr[i]; j = i - 1;
        while (j >= 0 && arr[j] < key) {
            arr[j + 1] = arr[j]; j = j - 1;
        }
```

```
        arr[j + 1] = key;
    }
}
#endif // VSORTDIRECT_H_INCLUDED
```

Apelarea bibliotecii cu numele vsortdirect.h
Implementare C++

```
#include <iostream>
#include "vsortdirect.h"
using namespace std;
int main(){
    int a[] = {5,7,9,4,2,6,1,3,0,8};
    int n = sizeof(a)/sizeof(a[0]);
    cout<<"Vectorul introdus: \n\t"; afisare(a, n);
    ///*****************************************************************
    cout<<"\nTehnica BubbleSort: \n";
    cout<<"\t\t crescator: \t"; bubbleSort1(a, n); afisare(a, n);
    cout<<"\t\t descrescator: \t"; bubbleSort2(a, n); afisare(a, n);
    ///*****************************************************************
    cout<<"\nTehnica SelectionSort: \n";
    cout<<"\t\t crescator: \t"; selectionSort1(a, n); afisare(a, n);
    cout<<"\t\t descrescator: \t"; selectionSort2(a, n); afisare(a, n);
    ///*****************************************************************
    cout<<"\nTehnica InsertionSort: \n";
    cout<<"\t\t crescator: \t"; insertionSort1(a, n); afisare(a, n);
    cout<<"\t\t descrescator: \t"; insertionSort2(a, n); afisare(a, n);
    return 0;
}
```

Rezultatele execuției:

```
Vectorul introdus:
        5 7 9 4 2 6 1 3 0 8

Tehnica BubbleSort:
                crescator:      0 1 2 3 4 5 6 7 8 9
                descrescator:   9 8 7 6 5 4 3 2 1 0

Tehnica SelectionSort:
                crescator:      0 1 2 3 4 5 6 7 8 9
                descrescator:   9 8 7 6 5 4 3 2 1 0

Tehnica InsertionSort:
                crescator:      0 1 2 3 4 5 6 7 8 9
                descrescator:   9 8 7 6 5 4 3 2 1 0
```

2. *Elaborarea unei biblioteci cu numele vsortindirect.h, care va implementa toate cele patru tehnici de sortare indirecte studiate pentru sortarea elementelor unui vector în ordine ascendentă și descendentă.*

Biblioteca cu numele vsortindirect.h
Implementare C++

```
#ifndef VSORTINDIRECT_H_INCLUDED
#define VSORTINDIRECT_H_INCLUDED
using namespace std;
int i,j;
void afisare(int arr[], int size){
    for (i = 0; i < size; i++)
    cout<<arr[i]<<" "; cout<<endl;
}
void swap(int* a, int* b){
    int t = *a; *a = *b; *b = t;
}
/// MergeSort crescator ************************************************************
void combina1(int a[], int l, int m, int r) {
    int i, j, k, n1 = m - l + 1, n2 =  r - m;
    int L[n1], R[n2];  /* cream tablouri temporare */
    /* Copiam datele in tablourile temporare L[] si R[] */
    for (i = 0; i < n1; i++) L[i] = a[l + i];
    for (j = 0; j < n2; j++) R[j] = a[m + 1+ j];
    /* Unirea celor doua tablouri temporare*/
    i = 0; j = 0; k = l;
    while (i < n1 && j < n2) {
        if (L[i] <= R[j]) { a[k] = L[i]; i++; }
        else { a[k] = R[j]; j++; }
        k++;
    }
    /* Copiam elementele ramase din L [], daca exista */
    while (i < n1) { a[k] = L[i]; i++; k++; }
    /* Copiam elementele ramase din R [], daca exista */
    while (j < n2) { a[k] = R[j]; j++; k++; }
}
void MergeSort1(int a[], int l, int r) {
    if (l < r){
        int m=l+(r-l)/2; MergeSort1(a,l,m); MergeSort1(a,m+1,r); combina1(a,l,m,r);
    }
}
/// MergeSort descrescator *********************************************************
void combina2(int a[], int l, int m, int r) {
    int i, j, k, n1 = m - l + 1, n2 =  r - m;
    int L[n1], R[n2];  /* cream tablouri temporare */
    /* Copiam datele in tablourile temporare L[] si R[] */
    for (i = 0; i < n1; i++) L[i] = a[l + i];
    for (j = 0; j < n2; j++) R[j] = a[m + 1+ j];
    /* Unirea celor doua tablouri temporare*/
    i = 0; j = 0; k = l;
    while (i < n1 && j < n2) {
        if (L[i] >= R[j]) { a[k] = L[i]; i++; }
        else { a[k] = R[j]; j++; }
        k++;
    }
    /* Copiam elementele ramase din L [], daca exista */
    while (i < n1) { a[k] = L[i]; i++; k++; }
    /* Copiam elementele ramase din R [], daca exista */
    while (j < n2) { a[k] = R[j]; j++; k++; }
}
void MergeSort2(int a[], int l, int r) {
    if (l < r){
        int m=l+(r-l)/2; MergeSort2(a,l,m); MergeSort2(a,m+1,r); combina2(a,l,m,r);
    }
}
/// HeapSort crescator *************************************************************
void heapify1(int a[], int n, int i) {
    int z = i, l = 2*i + 1, r = 2*i + 2;
    // Daca copilul stang este mai mare decat radacina
    if (l < n && a[l] > a[z]) z = l;
```

```
    // Daca copilul drept este mai mare decat cea mai mare de pana acum
    if (r < n && a[r] > a[z]) z = r;
    // Daca cea mai mare de pana acum nu este radacina
    if (z != i) {
        swap(a[i], a[z]); heapify1(a, n, z);
    }
}
void HeapSort1(int a[], int n){
    // Construim heapul (rearanjam vectorul)
    for (int i = n / 2 - 1; i >= 0; i--)
        heapify1(a, n, i);
    for (int i=n-1; i>=0; i--){
        swap(a[0], a[i]); // Mutam radacina curenta la sfarsit
        heapify1(a, i, 0); // Apelam max heapify pe heapul redus
    }
}
/// HeapSort descrescator ***********************************************************
void heapify2(int a[], int n, int i) {
    int z = i, l = 2*i + 1, r = 2*i + 2;
    // Daca copilul stang este mai mic decat radacina
    if (l < n && a[l] < a[z]) z = l;
    // Daca copilul drept este mai mic decat cea mai mica de pana acum
    if (r < n && a[r] < a[z]) z = r;
    // Daca cea mai mica de pana acum nu este radacina
    if (z != i) {
        swap(a[i], a[z]); heapify2(a, n, z);
    }
}
void HeapSort2(int a[], int n){
    // Construim heapul (rearanjam vectorul)
    for (int i = n / 2 - 1; i >= 0; i--)
        heapify2(a, n, i);
    for (int i=n-1; i>=0; i--){
        swap(a[0], a[i]); // Mutam radacina curenta la sfarsit
        heapify2(a, i, 0); // Apelam max heapify pe heapul redus
    }
}
/// ShellSort crescator *************************************************************
int ShellSort1(int a[], int n){
    /// Incepem cu un decalaj mare, apoi reducem decalajul
    for (int decalaj = n/2; decalaj > 0; decalaj /= 2){
        for (int i = decalaj; i < n; i += 1) {
            int j, temp = a[i];
            for (j = i; j >= decalaj && a[j - decalaj] > temp; j -= decalaj)
                a[j] = a[j - decalaj];
            ///  Punem temp (originalul lui a[i]) in locatia corecta
            a[j] = temp;
        }
    }
}
/// ShellSort descrescator **********************************************************
int ShellSort2(int a[], int n){
    /// Incepem cu un decalaj mare, apoi reducem decalajul
    for (int decalaj = n/2; decalaj > 0; decalaj /= 2){
        for (int i = decalaj; i < n; i += 1) {
            int j, temp = a[i];
            for (j = i; j >= decalaj && a[j - decalaj] < temp; j -= decalaj)
                a[j] = a[j - decalaj];
            ///  Punem temp (originalul lui a[i]) in locatia corecta
            a[j] = temp;
        }
    }
}
/// QuickSort crescator *************************************************************
int partitionare1 (int arr[], int l, int h){
    int pivot = arr[h]; // Pivotul
    int i = (l - 1); // Indicele elementului cel mai mic.
    for (int j = l; j <= h - 1; j++){
        // Daca elementul curent este mai mic decat pivotul.
        if (arr[j] < pivot) {
            i++; // Incrementarea indicelui elementului cel mai mic.
            swap(&arr[i], &arr[j]);
```

```
        }
    }
    swap(&arr[i + 1], &arr[h]); return (i + 1);
}
void QuickSort1(int arr[], int x, int y){
    if (x < y)    {
        /* pi este indexul de partitionare, arr [p] este acum la locul potrivit */
        int pi = partitionare1(arr, x, y);
        // Sorteaza separat elementele inainte de partitionare si dupa partitionare
        QuickSort1(arr, x, pi - 1); QuickSort1(arr, pi + 1, y);
    }
}
/// QuickSort descrescator ************************************************************
int partitionare2 (int arr[], int l, int h){
    int pivot = arr[h]; // Pivotul
    int i = (l - 1); // Indicele elementului cel mai mare.
    for (int j = l; j <= h - 1; j++){
        // Daca elementul curent este mai mare decat pivotul.
        if (arr[j] > pivot) {
            i++; // Incrementarea indicelui elementului cel mai mare.
            swap(&arr[i], &arr[j]);
        }
    }
    swap(&arr[i + 1], &arr[h]); return (i + 1);
}
void QuickSort2(int arr[], int x, int y){
    if (x < y)    {
        /* pi este indexul de partitionare, arr [p] este acum la locul potrivit */
        int pi = partitionare2(arr, x, y);
        // Sorteaza separat elementele inainte de partitionare si dupa partitionare
        QuickSort2(arr, x, pi - 1); QuickSort2(arr, pi + 1, y);
    }
}
#endif // VSORTINDIRECT_H_INCLUDED
```

Apelarea bibliotecii cu numele vsortindirect.h Implementare C++

```
#include <iostream>
#include "vsortindirect.h"
using namespace std;
int main(){
    int a[] = {42,26,61,19,90,0,83,35,57,74}, n = sizeof(a)/sizeof(a[0]);
    cout<<"Vectorul introdus: \n\t"; afisare(a, n);
    ///*********************************************************************
    cout<<"\nTehnica MergeSort: \n";
    cout<<"\t\t crescator: \t";
    MergeSort1(a,0,n-1); afisare(a, n);
    cout<<"\t\t descrescator: \t";
    MergeSort2(a,0,n-1); afisare(a, n);
    ///*********************************************************************
    cout<<"\nTehnica HeapSort: \n";
    cout<<"\t\t crescator: \t";
    HeapSort1(a, n); afisare(a, n);
    cout<<"\t\t descrescator: \t";
    HeapSort2(a,n); afisare(a, n);
    ///*********************************************************************
    cout<<"\nTehnica ShellSort: \n";
    cout<<"\t\t crescator: \t";
    ShellSort1(a, n); afisare(a, n);
    cout<<"\t\t descrescator: \t";
    ShellSort2(a,n); afisare(a, n);
    ///*********************************************************************
    cout<<"\nTehnica QuickSort: \n";
    cout<<"\t\t crescator: \t";
    QuickSort1(a,0,n-1); afisare(a, n);
    cout<<"\t\t descrescator: \t";
    QuickSort2(a,0,n-1); afisare(a, n);
    return 0;
}
```

Rezultatele execuției:

```
Vectorul introdus:
        42 26 61 19 90 0 83 35 57 74

Tehnica MergeSort:
                crescator:     0 19 26 35 42 57 61 74 83 90
                descrescator:  90 83 74 61 57 42 35 26 19 0

Tehnica HeapSort:
                crescator:     0 19 26 35 42 57 61 74 83 90
                descrescator:  90 83 74 61 57 42 35 26 19 0

Tehnica ShellSort:
                crescator:     0 19 26 35 42 57 61 74 83 90
                descrescator:  90 83 74 61 57 42 35 26 19 0

Tehnica QuickSort:
                crescator:     0 19 26 35 42 57 61 74 83 90
                descrescator:  90 83 74 61 57 42 35 26 19 0
```

TEHNICA GREEDY

7.1 Noțiuni generale despre greedy

7.1.1 Prezentare generală

Algoritmii greedy formează o paradigmă algoritmică care urmează euristica rezolvării de probleme care face la nivel local alegerea optimă pentru fiecare etapă în speranța de a găsi un optim global. În multe probleme, o strategie greedy produce, în general, o soluție optimă, dar cu toate acestea o euristică greedy poate produce la nivel local soluții optime care aproximează o soluție optimă globală într-un timp rezonabil.

Tehnica Greedy se aplică problemelor de optimizare, ea constă în faptul că se construiește soluția optimă pas cu pas. La fiecare pas, fiind selectat în soluție, elementul care pare „cel mai bun/ cel mai optim" la momentul respectiv, în speranța că această alegere locală va conduce la optimul global. De exemplu, o strategie greedy pentru problema comis-voiajorului (care este de mare complexitate computațională) are următoarea euristică: „La fiecare etapă, se vizitează un oraș nevizitat aflat cel mai apropiat de actualul oraș". Această euristică nu găsește neapărat o soluție mai bună, dar se termină într-un număr rezonabil de pași; găsirea unei soluții optime necesită, de obicei, un număr nejustificat de pași. Tehnica Greedy se aplică problemelor pentru care se dă o mulţime A cu n elemente şi pentru care trebuie determinată o submulţime a sa, S cu m elemente, care îndeplinesc anumite condiţii, numite și condiții de optim. Algoritmii greedy produc soluții bune la unele probleme de optimizare matematică, rezolvând probleme combinatorice, cu proprietăți de matroide ***[45]****.*

Notă:

- ✓ *Un matroid este o structură care abstractizează și generalizează noțiunea de independență liniară în spațiile vectoriale.*
- ✓ *Teoria matroizilor se preia foarte mult din terminologia algebrei liniare și a teoriei grafice, în mare parte pentru că este abstracția diferitelor noțiuni de importanță centrală în aceste domenii. Matroizii au găsit aplicații în geometrie, topologie, optimizare combinatorie, teoria rețelei și teoria codificării.*

Algoritmii Greedy apar și în rutarea rețelelor. Folosind rutarea greedy, un mesaj este transmis nodului vecin care este „cel mai apropiat" de destinație. Noțiunea de locație a unui nod (și, prin urmare, de „apropiere") poate fi determinată de localizarea fizică, ca în rutarea geografică utilizată de către rețelele ad-hoc. Locația poate fi și o construcție complet artificială, ca în rutarea într-o lume mică și în tabelele de dispersie distribuite. Algoritmii Greedy sunt foarte eficienți, dar nu conduc în mod necesar la o soluție optimă. Şi nici nu este posibilă formularea unui criteriu general conform căruia să putem stabili exact dacă metoda Greedy rezolvă sau nu o anumită problemă de optimizare. Din acest motiv, orice algoritm Greedy trebuie însotit de o demonstrație a corectitudinii sale . Demonstrația faptului că o anumită problemă are proprietatea alegerii Greedy se face, de obicei, prin inducție matematică.

Algoritm Greedy:

se dă o mulţime A;

- *se cere o submulţime S din multimea A care sa:*
- *să îndeplinească anumite condiţii interne (să fie acceptabilă);*
- *să fie optimală (să realizeze un maxim sau un minim).*

Principiul metodei Greedy:

- *se iniţializează mulţimea soluţiilor S cu mulţimea vidă, S=Ø;*
- *la fiecare pas se alege un anumit element x∈A (cel mai promiţător element la momentul*
- *respectiv) care poate conduce la o soluţie optimă;*

se verifică dacă elementul ales poate fi adăugat la mulţimea soluţiilor:

- *dacă da, atunci va fi adăugat şi mulţimea soluţiilor devine S=S∪{x} - un element introdus în mulţimea S nu va mai putea fi eliminat;*
- *altfel, el nu se mai testează ulterior, procedeul continuă, până când au fost determinate toate elementele din mulţimea soluţiilor.*

În general, algoritmii greedy au cinci componente:

1. *O mulțime de candidați, din care se creează o soluție;*
2. *Funcția de selecție, care alege cel mai bun candidat pentru a fi adăugat la soluție;*
3. *O funcție de fezabilitate, care este folosită pentru a determina dacă un candidat poate fi utilizat pentru a contribui la o soluție;*
4. *O funcție obiectiv, care atribuie o valoare unei soluții sau unei soluții parțiale;*
5. *O funcție de soluție, care va indica atunci când s-a descoperit o soluție complete.*

Cele mai multe probleme la care funcționează vor avea două proprietăți:

- ***Substructură optimă***
  „O problemă prezintă o substructură optimă dacă o soluție optimă a problemei conține soluții optime pentru subprobleme”.
- ***Proprietatea alegerii greedy***
  Putem face orice alegere ce pare mai bună pe moment și apoi se pot rezolva subproblemele care apar mai târziu. Alegerea făcută de către un algoritm greedy poate depinde de alegerile făcute până atunci, dar nu de viitoarele alegeri sau de toate soluțiile subproblemelor. El face iterativ o alegere greedy după alta, reducând fiecare problemă dată într-una mai mică.

Există câteva variante de algoritm greedy:

- *Algoritmi greedy puri;*
- *Algoritmi greedy ortogonali;*
- *Algoritmi greedy relaxați.*

Important:

- ✓ *Un algoritm greedy nu își reconsideră alegerile. Aceasta este principala diferență față de programarea dinamică, care este exhaustivă găsește garantat soluția.*
- ✓ *După fiecare etapă, programarea dinamică ia decizii pe baza tuturor deciziilor luate în etapa anterioară, și poate reconsidera calea găsită în etapa algoritmică anterioară.*

Observație:

- ❖ *Algoritmii greedy pot fi caracterizați ca având „viziune scurtă”, și „nerecuperabili”. Aceștia sunt ideali doar pentru probleme care au „substructură optima”. În ciuda acestui fapt, pentru multe probleme simple (de exemplu, problema restului de bani), cei mai potriviți algoritmi sunt algoritmii greedy.*
- ❖ *Algoritmii greedy pot fi folosiți ca algoritmi de selecție pentru a prioritiza opțiuni într-o căutare, sau într-un algioritm branch-and-bound.*

Algoritmii greedy dau greș în găsirea soluției optime globale mai ales pentru că nu operează exhaustiv pe toate datele. Ei își pot lua angajamente pentru anumite alegeri prea devreme, ceea ce îi împiedică să găsească cele mai bune soluții globale mai târziu.

De exemplu, toți algoritmii greedy de colorare pentru problema colorării grafurilor și pentru toate celelalte probleme NP-complete nu găsesc în mod consistent soluții optime. Cu toate acestea, ele sunt utile, deoarece acestea fac rapid alegeri și de multe ori dau aproximări bune ale soluției optime.

Dacă se poate demonstra că un algoritm greedy dă randament global optim pentru o anumită clasă de probleme, de obicei, acesta devine metoda aleasă, pentru că este mai rapid decât alte metode de optimizare ca programarea dinamică. Exemple de astfel de algoritmi greedy sunt algoritmul lui Kruskal și algoritmul lui Prim pentru găsirea arborilor minimi de acoperire, precum algoritmul pentru găsirea arborilor Huffman optimi. Teoria matroidelor și teoria mai generală a greedoidelor oferă clase întregi de astfel de algoritmi ***[1]****.*

7.1.2 Minimizarea timpului mediu de așteptare

Să presupunem că un frizer are mai mulţi clienţi pentru diverse servicii (tuns simplu, tuns cu șampon, permanent, vopsit). Serviciile nu vor dura la fel, dar stilistul știe cât va lua fiecare. Un scop al problemei, este ca frizerul să programeze clienţii astfel ca să minimizeze timpul de așteptare în salon (atât cât stau cât și când sunt serviți). Timpul total petrecut în salon se numește timp în sistem. Problema minimizării timpului de așteptare are multe aplicații: putem permite accesul la disc în funcţie de utilizator. Astfel, utilizatorii vor aștepta cât mai puţin pentru a citi un fișier ***[25]****.*

O altă problemă apare când un client va avea nevoie de același timp pentru a completa o sarcină, dar are un anumit deadline ceea ce înseamnă uneori că trebuie să înceapă mai devreme sau să fie servit la timp. Scopul este evident, de a minimiza timpul pentru a maximiza profitul. Vom discuta și acest aspect.

Să presupunem că există trei sarcini care trebuie îndeplinite și timpii pentru a le îndeplini sunt: $T_1 = 5$, $T_2 = 10$ *și* $T_3 = 4$. *Unităţile de timp nu sunt relevante. Dacă ordonăm cronologic cele trei sarcini obţinem următoarea secvenţă de acţiuni:*

Sarcina	Timp în sistem	Total
1	*5 (timp de servire)*	**5**
2	*5 (timp așteptare după 1)* *10 (timp de servire)*	**15**
3	*5 (timp așteptare după 1)* *10 (timp de așteptare după 2)* *4 (timp de servire)*	**19**
	Total:	**39**

Aceeași metodă de calcul va duce la următoarele posibilităţi de a servi cele trei cereri:

Modul de servire	Timp în sistem	Total
{1, 2, 3}	*5+(5+10)+(5+10+4)*	**39**
{1, 3, 2}	*5+(5+4)+(5+4+10)*	**33**
{2, 1, 3}	*10+(10+5)+(10+5+4)*	**44**
{2, 3, 1}	*10+(10+4)+(10+4+5)*	**43**
{3, 1, 2}	*4+(4+5)+(4+5+10)*	**32**
{3, 2, 1}	*4+(4+10)+(4+10+5)*	**37**

Observație:

- *Se observă că cel mai bun mod de a servi cererile este [3, 1, 2] cu un timp total de 32.*
- *Există clar un algoritm care poate verifica toate permutările posibile. Ordinul de timp al acestui algoritm este însă factorial. Această soluţie, numită și brute force poate fi înlocuită cu un algoritm de tip greedy mult mai simplist și mai eficient. Se observă că soluţia cea mai bună conţine timpii sortaţi crescător.*

Vom descrie în continuare elaborarea unui subprogram de determinare a timpului optim:

1. *sortăm sarcinile după timp în ordine ascendentă*
2. *while (instanța nu este rezolvată) do*
3. *programează următoarea sarcină*
4. *if (nu mai există sarcini) instanța este rezolvată*

*Evident că algoritmul poate fi adaptat după sensul problemei, în cazul de faţă putem înlocui pasul 3 cu adăugarea timpului sarcinii următoare la timpul total. Complexitatea acestui algoritm este: O (n*log n).*

7.1.3 Programarea deservirii folosind termeni limită

În această problemă de programare, fiecare sarcină va avea nevoie de o unitate de timp pentru a termina și are un termen limită și un profit ***[46]****.*

Adică, dacă operaţia începe mai devreme de termenul limită, se obţine un profit. Scopul este de a programa operaţia ca profitul să fie maxim. Nu toate sarcinile trebuie să fie executate. Iată un exemplu pentru a ilustra mai bine problema:

Sarcina	Timp în sistem	Profit
1	**2**	**30**
2	**1**	**35**
3	**2**	**25**
4	**1**	**40**

Atunci când spunem că o sarcină 1 are un termen limită de 2, înseamnă că sarcina 1 poate începe la timpul 1 sau timpul 2. Nu există timp 0. Deoarece sarcina 2 are timpul limită 1, această sarcină poate porni doar la timpul 1. Iată posibilele profituri:

Programare	Profit	Total
{1, 3}	**30+25**	**55**
{2, 1}	**35+30**	**65**
{2, 3}	**35+25**	**60**
{3, 1}	**25+30**	**55**
{4, 1}	**40+30**	**70**
{4, 3}	**40+25**	**65**

Notă:

- ✓ *Alte programări imposibile (de exemplu [1,2]), nu au mai fost listate. Pe de altă parte, programări ca [1,3] sunt posibile, deoarece sarcina 1 a pornit înaintea termenului limită, iar sarcina 3, a pornit exact la termenul limită.*
- ✓ *Se poate vedea că programarea [1,4] este optimă și are cel mai mare profit.*
- ✓ *Se poate modifica algoritmul de mai sus pentru a obține un algoritm mai bun decât cel de ordin factorial pentru a rezolva această problemă.*

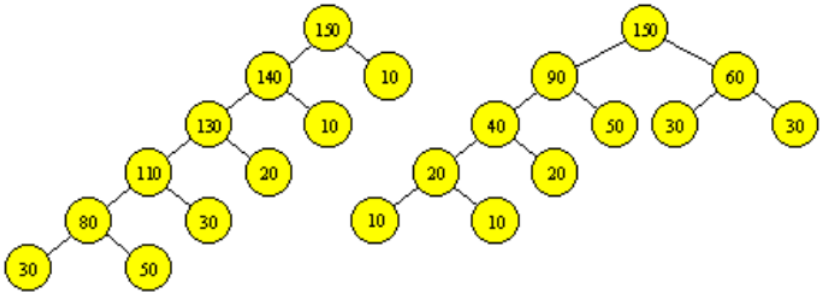
7.1.4 Interclasarea optimă a șirurilor ordonate

Să presupunem că avem două șiruri S_1, S_2 ordonate crescător și că dorim să obţinem prin interclasarea lor, șirul ordonat crescător ce conţine elementele din ambele șiruri. Dacă interclasarea are loc prin deplasarea elementelor din cele două șiruri, atunci numărul deplasărilor este: $S_1 + S_2$.

Dacă generalizăm, considerăm n șiruri S_1, S_2 , ... , S_n unde fiecare șir S_i, $1 \leq i \leq n$ fiind format din q_i elemente ordonate crescător (vom denumi q_i lungimea lui S_i). Ne propunem să obţinem șirul S, ordonat crescător, unde S conţine toate elementele din cele n șiruri. Vom realiza aceasta prin interclasări succesive de câte două șiruri. Problema constă în determinarea ordinii optime în care trebuie efectuate aceste interclasări, astfel ca numărul total de deplasări să fie minim. Exemplul de mai jos ne arată că problema nu este una banală ***[47]****.*

Fie șirurile S_1, S_2 , S_3 de lungimi $q_1 = 30$, $q_2 = 20$, $q_3 = 10$. Dacă interclasăm pe S_1 cu S_2, iar rezultatul îl interclasăm cu S_3, numărul total de deplasări este $(30 + 20) + (50 + 10) = 110$. Dacă interclasăm pe S_1 cu S_2, iar rezultatul îl interclasăm cu S_3, numărul total de deplasări este $(10 + 20)+ (30 + 30) = 90$. Dacă interclasăm pe S_1 cu S_3, iar rezultatul în interclasăm cu S_2, atunci numărul total de deplasări va fi $(10 + 30) + (40 + 20) = 100$. După cum se poate vedea numărul minim de deplasări este 90.

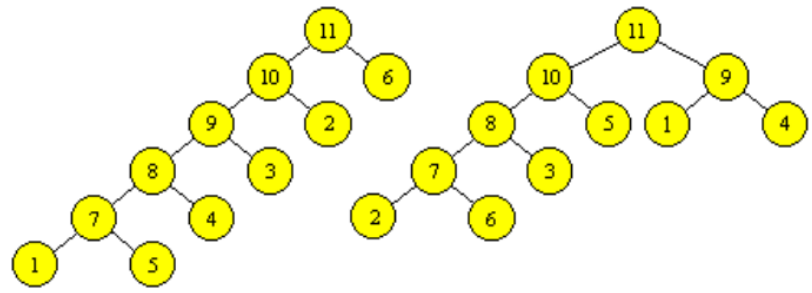
Atașăm fiecărei strategii de interclasare un arbore binar în care valoarea fiecărui vârf este dată de lungimea șirului pe care îl reprezintă. De exemplu, dacă șirurile S_1, S_2 , ... , S_6 au lungimile $q_1 = 30$, $q_2 = 10$, $q_3 = 20$, $q_4 = 30$, $q_5 = 50$, $q_6 = 10$, două dintre strategiile de interclasare sunt reprezentate prin arborii din figura de mai jos:



Cazul A Cazul B

Figura 1. Reprezentarea strategiilor de interclasare

Observăm că fiecare arbore are 6 vârfuri terminale, corespunzând celor 6 şiruri iniţiale şi 5 vârfuri neterminale, adică cele 5 interclasări care definesc strategia. Numerotăm vârfurile astfel: vârful terminal i, $1 \le i \le 6$ va corespunde şirului S_i, iar vârfurile neterminale se numerotează de la 7 la 11 în ordinea obţinerii lor, ca în Figura 2.



Cazul A Cazul B

Figura 2. Numerotarea vârfurilor arborilor din Figura 1.

Strategia greedy apare în Figura 1(Cazul B), şi constă în a interclasa mereu cele mai scurte şiruri disponibile la momentul curent. Interclasând şirurile S_1, S_2, ... , S_n de lungimile q_1, q_2, ... , q_n, obţinem pentru fiecare strategie câte un arbore binar cu n vârfuri terminale, numerotate de la 1 la n, şi n-1 vârfuri neterminale, numerotate de la n+1 la 2n-1. Definim pentru un arbore oarecare A de acest tip, lungimea externă ponderată ca fiind:

$L(A) = \sum_{i=1}^{n} a_i q_i$ *, unde a_i este adâncimea vârfului i.*

Observaţie:

- *Numărul total de deplasări de elemente pentru strategia corespunzătoare lui A este L(A).*
- *Soluţia optimă este arborele pentru care lungimea ponderată este minimă.*

Proprietate 1:
Prin metoda greedy se obţine întotdeauna interclasarea optimă a n şiruri ordonate, deci strategia cu arborele de lungime externă ponderată minimă.

La scrierea algoritmului care generează arborele strategiei greedy de interclasare vom folosi un min-heap. Fiecare element al min-heap-ului este o pereche (q, i) unde i este numărul unui vârf din arborele strategiei de interclasare, iar q este lungimea şirului pe care îl reprezintă. Proprietatea de min-heap se referă la valoarea lui q.

Algoritmul interopt va construi arborele strategiei greedy. Un varf i al arborelui va fi memorat în trei locații diferite conținând:

1. *LU[i] = lungimea șirului reprezentat de vârf;*
2. *ST[i], DR[i] = numărul fiului stâng, respectiv numărul fiului drept.*

Vom descrie în continuare elaborarea unui subprogram pentru algoritmul interopt:

1. *interopt*($Q[1 .. n]$) {construiește arborele strategiei greedy de interclasare a șirurilor de lungimi $Q[i] = q_i$, $1 \le i \le n$}
2. $H \leftarrow$ min-heap vid
3. for $i \leftarrow 1$ to n do
4. $(Q[i], i) => H$ {insereaza in min-heap}
5. $LU[i] \leftarrow Q[i]$; $ST[i] \leftarrow 0$; $DR[i] \leftarrow 0$
6. for $i \leftarrow n+1$ to $2n-1$ do
7. $(s, j) <= H$ {extrage radacina lui H}
8. $(r, k) <= H$ {extrage radacina lui H}
9. $ST[i] \leftarrow j$; $DR[i] \leftarrow k$; $LU[i] \leftarrow s+r$; $(LU[i], i) => H$ {inserează în min-heap}

Observație:

- *În cazul cel mai nefavorabil, operațiile de inserare în min-heap și de extragere din min-heap necesită un timp în ordinul lui log n. Restul operațiilor necesită un timp constant.*
- *Timpul total pentru interopt este O(n log n).*

7.1.5 Implementarea arborilor de interclasare

Transpunerea procedurii interopt într-un limbaj de programare prezinta o singură dificultate generată de utilizarea unui min-heap de perechi vârf-lungime. În limbajul C++, implementarea arborilor de interclasare este aproape o operație de rutină, deoarece clasa parametrică heap permite manipularea unor heap-uri cu elemente de orice tip în care este definit operatorul de comparare ">".

Altfel nu avem decât să construim o clasă formată din perechi vârf-lungime (pondere) și să o completăm cu operatorul ">" corespunzător. Vom numi aceasta clasa vp, adică vârf-pondere. Elaborarea bibliotecii în limbajul C++ va avea următoarea secvență de cod:

```
#ifndef __VP_H
#define __VP_H
#include <iostream.h>
class vp {
public:
  vp( int vf = 0, float pd = 0 ) { v = vf; p = pd; }
  operator int  ( ) const { return v; }
  operator float( ) const { return p; }
  int v; float p;
};
inline operator > ( const vp& a, const vp& b) {
  return a.p <  b.p;
}
inline istream& operator >>( istream& is, vp& element ) {
  is >> element.v >> element.p; element.v--;
  return is;
}
inline ostream& operator <<( ostream& os, vp& element ) {
  os << "{ " << (element.v+1) << "; " << element.p << " }";
  return os;
}
#endif
```

Scopul clasei vp (definită în fișierul vp.h) nu este de a introduce un nou tip de date, ci mai curând de a facilita manipularea structurii vârf-pondere, structură utilă la reprezentarea grafurilor. Din acest motiv, nu exista nici un fel de incapsulare, toți membrii fiind publici ***[46]****.*

Notă:

- ✓ *Incapsularea este proprietatea claselor de obiecte de a grupa sub aceeași structură datele și metodele aplicabile asupra datelor.*
- ✓ *În sensul programării orientata pe obiecte, incapsularea definește modalitatea în care diverse obiecte și restul programului se pot referi la date specifice obiectelor. Dupa cum s-a precizat clasele din C++ permit separarea datelor în date private și date publice. Programele utilizatorilor pot accesa și utiliza datele unui obiect, declarate private, numai prin utilizarea unor metode definite public. Separarea datelor în secțiuni de date publice și private determină protejarea datelor față de utilizarea lor eronată în programe.*

Ne mai rămâne să precizăm structura arborelui de interclasare. Cel mai simplu este să preluăm structura folosită în subprogramul interopt din secțiunea 7.1.4, unde arborele este format din trei tablouri paralele, care conțin lungimea șirului reprezentat de vârful respectiv și indicii celor doi fii. Pentru o scriere mai compacta, vom folosi o structură parțial diferită: un tablou de elemente de tip nod, fiecare nod conținând trei câmpuri corespunzătoare informațiilor de mai sus. Clasa nod este similară clasei vp, atât ca structură, cât și prin motivația introducerii acesteia.

Elaborarea clasei nod în limbajul C++ va avea următoarea secvență de cod:

```
class nod {
public:
  int lu; // lungimea
  int st; // fiul stang
  int dr; // fiul drept
};

inline ostream& operator <<( ostream& os, nod& nd ) {
  os << " <" << nd.st << "< "
     << nd.lu
     << " >" << nd.dr << "> ";
  return os;
}
```

În limbajul C++, funcția de construire a arborelui strategiei greedy se obține direct, prin transcrierea subprogramului interopt.

```
tablou<nod> interopt( const tablou<int>& Q ) {
   int n = Q.size( );
   tablou<nod> A( 2 * n - 1 ); // arborele de interclasare
   heap  <vp>  H( 2 * n - 1 );
   for ( int i = 0; i < n; i++ ) {
      H.insert( vp(i, Q[i]) );
      A[i].lu = Q[i]; A[i].st = A[i].dr = -1;
   }
   for ( i = n; i < 2 * n - 1; i++ ) {
      vp s; H.delete_max( s ); vp r; H.delete_max( r );
      A[i].st = s; A[i].dr = r; A[i].lu = (float)s + (float)r;
      H.insert( vp(i, A[i].lu) );
   }
   return A;
}
```

Observație:

- ❖ *Constructorul vp(int, float) este invocat explicit în funcția de inserare în heap-ul H. Efectul acestei invocări constă în crearea unui obiect temporar de tip vp, obiect distrus după inserare. O notație foarte simplă ascunde o anumită ineficiență, datorită creării și distrugerii obiectului temporar.*

- *Operatorul de conversie la int este invocat implicit în expresiile A[i].st=s și A[i].dr=r, iar în expresia A[i].lu=(float)s+(float)r, operatorul de conversie la float trebuie să fie specificat explicit.*
- *Semantica limbajului C++ este foarte clară relativ la conversii: cele utilizator au prioritate față de cele standard, iar ambiguitatea în selectarea conversiilor posibile este semnalată ca eroare. Dacă în primele două atribuiri conversia lui s și r la int este singura posibilitate, scrierea celei de-a treia sub forma A[i].lu=s+r este ambiguă, expresia s+r poate fi evaluată atât ca int, cât și ca float.*

În final, nu ne mai rămâne decât să testăm funcția interopt(). Vom folosi un tablou l cu lungimi de șiruri, lungimi extrase din stream-ul standard de intrare.

```
main( ) {
  tablou<int> l;
  cout << "Siruri: "; cin >> l;
  cout << "Arborele de interclasare: ";
  cout << interopt( l ) << '\n';
  return 1;
}
```

Strategia de interclasare optimă pentru cele șase lungimi folosite ca exemplu în secțiunea 7.1.4:	*Rezultatul obținut ca urmare a interclasării optime pentru exemplul din secțiunea 7.1.4:*
6 30 10 20 30 50 10	Arborele de interclasare: 11 <-1<30>-1> <-1<10>-1> <-1<20>-1> <-1<30>-1> <-1<50>-1> <-1<10>-1> <1<20>5> <2<40>6> <3<60>0> <7<90>4> <8<150>9>

Observație:

- *Valoarea fiecărui nod este precedată de indicele fiului stâng și urmată de cel al fiului drept, indicele -1 reprezentând legătura inexistentă.*
- *Formatele de citire și scriere ale tablourilor sunt cele stabilite pentru clasa parametrică tablou.*

7.1.6 Coduri Huffman.

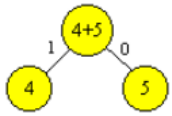
O altă aplicație a strategiei greedy și a arborilor binari cu lungime externă ponderată minimă este obținerea unei codificări cât mai compacte a unui text.

Un principiu general de codificare a unui șir de caractere este următorul: Se măsoară frecvența de apariție a diferitelor caractere dintr-un eșantion de text și se atribuie cele mai scurte coduri celor mai frecvente caractere, și cele mai lungi coduri - celor mai puțin frecvente caractere.

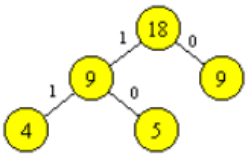
Acest principiu stă la baza codului Morse. Pentru situația în care codificarea este binară, există o metodă elegantă pentru a obține codul respectiv. Aceasta metoda, a fost descoperită de Huffman (1952), care folosește o strategie greedy și se numește codificarea Huffman. O vom descrie pe baza unui exemplu.

Fie un text compus din următoarele litere (în paranteze figurează frecvențele lor de apariție):
S (10), I (29), P (4), O (9), T (5).

Conform metodei greedy, vom construi un arbore binar fuzionând cele două litere cu frecvențele cele mai mici. Valoarea fiecărui vârf este dată de frecvența pe care o reprezintă:



Etichetăm muchia stângă cu 1 și muchia dreaptă cu 0. Rearanjăm tabelul de frecvențe: S(10), I(29), O(9), {P,T} (4+5=9). Mulțimea {P, T} semnifică evenimentul reuniune a celor două evenimente independente corespunzătoare apariției literelor P și T. Continuăm procesul, obținând arborele:



În final, ajungem la arborele din Figura 3, în care fiecare vârf terminal corespunde unei litere din text.

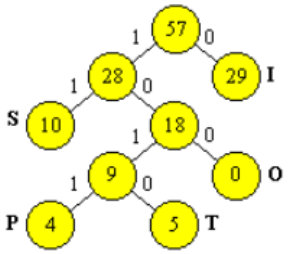


Figura 3. Arborele de codificare Huffman

Pentru a obține codificarea binară a literei P, trebuie să scriem secvența de 0-uri și 1-uri în ordinea apariției lor pe drumul de la rădăcină către vâful corespunzător lui P: 1011. Procedăm similar și pentru restul literelor: S(11), I(0), P(1011), O(100), T(1010).

Pentru un text format din n litere care apar cu frecventele $f_1, f_2, ..., f_n$, un arbore de codificare este un arbore binar cu vârfurile terminale având valorile $f_1, f_2, ..., f_n$, prin care se obține o codificare binară a textului.

Un arbore de codificare nu trebuie în mod necesar să fie construit după metoda greedy a lui Huffman, alegerea vârfurilor care sunt fuzionate la fiecare pas se pot obține după diverse criterii ***[47]****.*

Notă:

- ✓ *Lungimea externă ponderată a unui arbore de codificare este :* $\sum_{i=1}^{n} a_i f_i$, *unde* a_i *este adâncimea vârfului terminal corespunzător literei i.*

Observație:

- ❖ *Lungimea externă ponderată este egală cu numărul total de caractere din codificarea textului considerat.*
- ❖ *Codificarea cea mai compactă a unui text corespunde, deci. arborelui de codificare de lungime externă ponderată minimă.*

- *Se poate demonstra că arborele de codificare Huffman minimizează lungimea externă ponderată pentru toți arborii de codificare cu vârfurile terminale având valorile $f_1, f_2, ..., f_n$.*
- *Prin strategia greedy se obține întotdeauna codificarea binară cea mai compactă a unui text.*
- *Arborii de codificare pe care i-am considerat în acestă secțiune corespund unei codificări de tip special: codificarea unei litere nu este prefixul codificării nici unei alte litere. O astfel de codificare este de tip prefix. Codul Morse nu face parte din aceasta categorie.*
- *Codificarea cea mai compactă a unui șir de caractere poate fi întotdeauna obținută printr-un cod de tip prefix.*

Exemplu:

Coduri de prefix, înseamnă că aceste coduri (secvențe de biți) sunt atribuite în așa fel încât codul atribuit unui caracter nu este prefixul codului atribuit niciunui alt caracter. Așa se face că această codificare Huffman se asigură că nu există ambiguitate atunci când se decodează fluxul generat.

Să înțelegem codurile prefixului cu un exemplu contrar. Fie patru caractere a, b, c și d, iar codurile lor de lungime variabilă corespunzătoare să fie 00, 01, 0 și 1. Această codificare duce la ambiguitate, deoarece codul atribuit lui c este prefixul codurilor atribuite lui a și b. Dacă fluxul de biți comprimat este 0001, ieșirea decomprimată poate fi „cccd" sau „ccb" sau „acd" sau „ab".

Există, în principal, două părți majore în Huffman Coding:

1. *Construiți un arbore Huffman din caractere de intrare.*
2. *Traversați arborele Huffman și atribuiți coduri caracterelor.*

Pași pentru construirea arborelui Huffman

Datele de intrare reprezintă o serie de caractere unice, împreună cu frecvența lor de apariții, iar ieșirea este Huffman Tree.

1. *Creați un nod de frunze pentru fiecare personaj unic și construiți un heap min de toate nodurile de frunze (Min Heap este utilizat ca coadă prioritară. Valoarea câmpului de frecvență este utilizată pentru a compara două noduri în min heap. Inițial, caracterul cel mai puțin frecvent este la rădăcină).*
2. *Extrageți două noduri cu frecvența minimă din mormanul minim.*
3. *Creați un nou nod intern cu o frecvență egală cu suma celor două noduri. Faceți primul nod extras ca copilul său stâng și celălalt nod extras ca copilul său drept. Adăugați acest nod la mormanul minim.*
4. *Repetați pașii nr. 2 și nr. 3 până când grămada nu conține un singur nod. Nodul rămas este nodul rădăcină și arborele este complet* ***[48]****.*

Date de intrare		Date de ieșire	
a	5	f	0
b	9	c	100
c	12	d	101
d	13	a	1100
e	16	b	1101
f	45	e	111

Complexitatea în timp: O (nlogn) unde n este numărul de caractere unice. Dacă există n noduri, extractMin () se numește de 2 (n - 1). extractMin () durează O (logn) timp în care apelează minHeapify (). Deci, complexitatea generală este O (nlogn).*

Observație:

- *Dacă tabloul de intrare este sortat, există un algoritm liniar de timp.*
- *Implementarea codului în limbajul C++ pentru algoritmul lui Huffman va fi reprezentat în compartimentul: Implementarea tehnicii greedy la rezolvarea problemelor complexe.*

7.1.7 Arbori parţiali de cost minim

Fie G = <V, M> un graf neorientat conex, unde V este mulțimea vârfurilor și M este mulțimea muchiilor. Fiecare muchie are un cost nenegativ (sau o lungime nenegativă). Problema este să găsim o submulțime A ⊆ M, astfel încât toate vârfurile din V să rămână conectate atunci când sunt folosite doar muchii din A, iar suma lungimilor muchiilor din A să fie cât mai mică. Căutăm deci o submulțime A de cost total minim. Această problemă se mai numește și problema conectării orașelor cu cost minim, având numeroase aplicații.

Graful parțial <V, A> este un arbore și este numit arborele parțial de cost minim al grafului G (minimal spanning tree). Un graf poate avea mai mulți arbori parțiali de cost minim și acest lucru se poate verifica pe un exemplu.

Vom prezenta doi algoritmi greedy care determină arborele parțial de cost minim al unui graf. În terminologia metodei greedy, vom spune că o mulțime de muchii este o soluție, dacă constituie un arbore parțial al grafului G și este fezabilă, dacă nu conține cicluri. O mulțime fezabilă de muchii este promițătoare, dacă poate fi completată pentru a forma soluția optimă. O muchie atinge o mulțime dată de vârfuri, dacă exact un capăt al muchiei este în mulțime. Următoarea proprietate va fi folosită pentru a demonstra corectitudinea celor doi algoritmi.

Proprietate 2:
Fie G = <V, M> un graf neorientat conex în care fiecare muchie are un cost nenegativ. Fie W ⊂ V o submulțime strictă a vârfurilor lui G și fie A ⊆ M o mulțime promițătoare de muchii, astfel încât nici o muchie din A nu atinge W. Fie m muchia de cost minim care atinge W. Atunci, A ∪ {m} este promițătoare ***[49]****.*

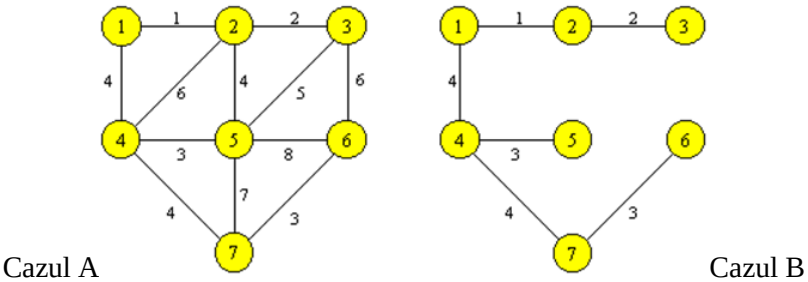
Notă:

- ✓ *Mulțimea inițială a candidaților este M. Cei doi algoritmi greedy aleg muchiile una câte una într-o anumită ordine, această ordine fiind specifică fiecarui algoritm.*

Prezentare generală a algoritmului lui Kruskal

Arborele parțial de cost minim poate fi construit muchie cu muchie, după următoarea metoda a lui Kruskal (1956): se alege întâi muchia de cost minim, iar apoi se adaugă repetat muchia de cost minim nealeasă anterior și care nu formează cu precedentele un ciclu. Alegem astfel V–1 muchii. Este ușor de dedus că obținem în final un arbore. Este acesta chiar arborele parțial de cost minim căutat?

Înainte de a răspunde la întrebare, fie avem graful din Figura 4 (Cazul A):



Cazul A Cazul B

Figura 4. Un graf și arborele său parțial de cost minim

Ordonăm crescător (în funcție de cost) muchiile grafului:
{1, 2}, {2, 3}, {4, 5}, {6, 7}, {1, 4}, {2, 5}, {4, 7}, {3, 5}, {2, 4}, {3, 6}, {5, 7}, {5, 6}, apoi aplicăm algoritmul.

Structura componentelor conexe este ilustrata, pentru fiecare pas, în tabelul de mai jos:

Pasul	Muchia considerată	Componentele conexe ale subgrafului <V, A>
inițializare	-	{1}, {2}, {3}, {4}, {5}, {6}, {7}
1	{1, 2}	{1, 2}, {3}, {4}, {5}, {6}, {7}
2	{2, 3}	{1, 2, 3}, {4}, {5}, {6}, {7}
3	{4, 5}	{1, 2, 3}, {4, 5}, {6}, {7}
4	{6, 7}	{1, 2, 3}, {4, 5}, {6, 7}
5	{1, 4}	{1, 2, 3, 4, 5}, {6, 7}
6	{2, 5}	respinsă (formează ciclu)
7	{4, 7}	{1, 2, 3, 4, 5, 6, 7}

Mulțimea A este inițial vidă și se completează pe parcurs cu muchii acceptate (care nu formează un ciclu cu muchiile deja existente în A). În final, mulțimea A va conține muchiile {1, 2}, {2, 3}, {4, 5}, {6, 7}, {1, 4}, {4, 7}. La fiecare pas, graful parțial <V, A> formează o pădure de componente conexe, obținută din pădurea precedentă unind două componente. Fiecare componentă conexă este, la rândul ei un arbore parțial de cost minim pentru vârfurile pe care le conectează. Inițial, fiecare vârf formează o componentă conexă. La sfârșit, vom avea o singură componentă conexă, care este arborele parțial de cost minim căutat (Figura 4 (Cazul B)).

Ceea ce am observat în acest caz particular este valabil și pentru cazul general, din Proprietatea 2 rezultă următoarea proprietate:

Proprietate 3:
În algoritmul lui Kruskal, la fiecare pas, graful parțial <V, A> formează o pădure de componente conexe, în care fiecare componentă conexă este la rândul ei un arbore parțial de cost minim pentru vârfurile pe care le conectează ***[50]****. În final, se obține arborele parțial de cost minim al grafului G.*

Prezentare generală a algoritmului lui Prim

Cel de-al doilea algoritm greedy pentru determinarea arborelui parțial de cost minim al unui graf se datorează lui Prim (1957). În acest algoritm, la fiecare pas, mulțimea A de muchii alese împreună cu mulțimea U a vârfurilor pe care le conectează formează un arbore parțial de cost minim pentru subgraful <U, A> al lui G.

Inițial, mulțimea U a vârfurilor acestui arbore conține un singur vârf oarecare din V, care va fi rădăcină, iar mulțimea A a muchiilor este vidă. La fiecare pas, se alege o muchie de cost minim, care se adaugă la arborele precedent, oferind naștere unui nou arbore parțial de cost minim (deci, exact una dintre extremitățile acestei muchii este un vârf în arborele precedent). Arborele parțial de cost minim crește "natural", cu câte o ramură, până când va atinge toate vârfurile din V, adică până când U = V. Funcționarea algoritmului, pentru exemplul din Figura 4 (Cazul A), este ilustrată în tabelul de mai jos:

Pasul	Muchia considerată	Componentele conexe ale subgrafului <V, A>
inițializare	—	{1}
1	{2, 1}	{1, 2}
2	{3, 2}	{1, 2, 3}
3	{4, 1}	{1, 2, 3, 4}
4	{5, 4}	{1, 2, 3, 4, 5}
5	{7, 4}	{1, 2, 3, 4, 5, 6}
6	{6, 7}	{1, 2, 3, 4, 5, 6, 7}

La sfârșit, A va conține aceleași muchii ca și în cazul algoritmului lui Kruskal. Faptul că algoritmul funcționează întotdeauna corect este exprimat de următoarea proprietate, pe care o puteți demonstra folosind Proprietatea 2.

Proprietate 4:
În algoritmul lui Prim, la fiecare pas, <U, A> formează un arbore parțial de cost minim pentru subgraful <U, A> al lui G. În final, se obține arborele parțial de cost minim al grafului G.

Algoritmul lui Kruskal	Algoritmul lui Prim
Pasul 1: ○ *creează o pădure F (o mulțime de arbori), unde fiecare vârf din graf este un arbore separat;* *Pasul 2:* ○ *creează o mulțime S care conține toate muchiile din graf;* *Pasul 3:* ○ *atât timp cât S este nevidă:* • *elimină o muchie de cost minim din S* • *dacă acea muchie conectează doi arbori distincți, atunci adaugă muchia în pădure, combinând cei doi arbori într-unul singur* • *altfel, ignoră muchia* *Pasul 4:* ○ *La sfârșitul algoritmului, pădurea are doar o componentă care reprezintă un arbore parțial de cost minim al grafului.*	*Pasul 1:* ○ *se alege un vârf de start (x);* *Pasul 2:* ○ *se marchează vârful într-un vector viz, în care vedem nodurile prin care am trecut (viz[x]=1);* *Pasul 3:* ○ *se inițializează vectorul s astfel : s[x]=0, s[i]=x (pentru i!=x); → la final, vectorul s va reprezenta și vectorul de tați;* *Pasul 4:* ○ *se alege muchia de cost minim cu un capăt selectat;* *Pasul 5:* ○ *se marchează capătul neselectat;* *Pasul 6:* ○ *se actualizează vectorii viz și s;* *Pasul 7:* ○ *se reia pasul 2 de n-2 ori (n fiind numărul de noduri din graf).*

Notă:

- ✓ *Timpul pentru algoritmul lui Kruskal este în O(m log n), unde m = M.*
- ✓ *Pentru un graf dens (adică, cu foarte multe muchii), se deduce că m se apropie de n(n–1)/2. În acest caz, algoritmul Kruskal necesită un timp în $O(n^2 \log n)$ și algoritmul Prim este probabil mai bun.*
- ✓ *Pentru un graf rar (adică, cu un număr foarte mic de muchii), m se apropie de n și algoritmul Kruskal necesită un timp în O(n log n), fiind probabil mai eficient decât algoritmul Prim.*

Analiza comparativă a algoritmilor Kruskal și Prim

1. *Se poate observa că în cazul algoritmului lui Prim există două bucle repetitive imbricate, fiecare executând n-1 iterații. Execuția acestor instrucțiuni, în cazul în care avem n noduri va produce un ordin de timp: $O(n)= 2(n-1)(n-1)= O(n^2)$.*
2. *Algoritmului lui Kruskal este de ordin $O(n^2 \log n)$.*
3. *Dacă m este numărul de muchii dintr-un graf conex, atunci următoarea relație este valabilă:*
$$n-1 \le m \le \frac{n(n-1)}{2}.$$
4. *Pentru un graf a cărui număr de muchii se apropie de n-1, algoritmul lui Kruskal este în O(n log n), ceea ce înseamnă că este mai rapid. Totuși dacă numărul de muchii tinde către limita superioară, atunci algoritmul lui Prim este mai bun* ***[51]****.*

7.1.8 Cele mai scurte drumuri care pleacă din același punct

Fie G = <V, M> un graf orientat, unde V este mulțimea vârfurilor și M este mulțimea muchiilor. Fiecare muchie are o lungime nenegativă. Unul din vârfuri este desemnat ca vârf sursă. Problema este să determinăm lungimea celui mai scurt drum de la sursă către fiecare vârf din graf.

Vom folosi un algoritm greedy, propus de Dijkstra (1959). Algoritmul lui Dijkstra este un algoritm foarte popular in Teoria Grafurilor. Algoritmul lui Dijkstra funcționează atât pe grafuri conexe, cât și pe grafuri neconexe ***[52]****.*

Notăm cu C mulțimea vârfurilor disponibile (candidații) și cu S mulțimea vârfurilor deja selectate. În fiecare moment, S conține acele vârfuri a căror distanță minimă de la sursă este deja cunoscută, în timp ce mulțimea C conține toate celelalte vârfuri. La început, S conține doar vârful sursă, iar în final S conține toate vârfurile grafului. La fiecare pas, adăugăm în S acel vârf din C a cărui distanță de la sursă este cea mai mică.

Spunem ca un drum de la sursa către un alt vârf este special, dacă toate vârfurile intermediare de-a lungul drumului aparțin lui S. Algoritmul lui Dijkstra lucrează în felul următor: La fiecare pas al algoritmului, un tablou D conține lungimea celui mai scurt drum special către fiecare vârf al grafului. După ce adăugăm un nou vârf v la S, cel mai scurt drum special către v va fi, de asemenea, cel mai scurt dintre toate drumurile către v. Când algoritmul se termină, toate vârfurile din graf sunt în S, deci toate drumurile de la sursă către celelalte vârfuri sunt speciale și valorile din D reprezintă soluția problemei.

Presupunem, pentru simplificare, că vârfurile sunt numerotate, V = {1, 2, ..., n}, vârful 1 fiind sursa, iar matricea L oferă lungimea fiecărei muchii, cu L[i, j] = +∞, dacă muchia (i, j) nu există. Soluția se va construi în tabloul D[2 .. n]. Fie avem graful din Figura 5:

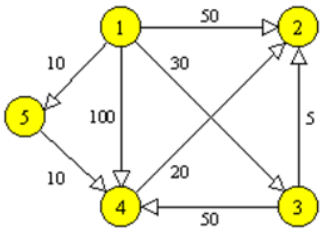


Figura 5. Un graf orientat

Functionarea algoritmului, pentru exemplul din Figura 5, este ilustrată în tabelul de mai jos:

Pasul	v	C	D
inițializare	—	{2, 3, 4, 5}	[50, 30, 100, 10]
1	5	{2, 3, 4}	[50, 30, 20, 10]
2	4	{2, 3}	[40, 30, 20, 10]
3	3	{2}	[35, 30, 20, 10]

Proprietate 5:
În algoritmul lui Dijkstra:

- *dacă un vârf i este în S, atunci D[i] oferă lungimea celui mai scurt drum de la sursă către i;*
- *dacă un vârf i nu este în S, atunci D[i] oferă lungimea celui mai scurt drum special de la sursă către i.*

Algoritmul lui Dijktra

1. Se creează o listă cu distanțe, o listă cu nodul anterior, o listă cu nodurile vizitate și un nod curent.
2. Toate valorile din lista cu distanțe sunt inițializate cu o valoare infinită, cu excepția nodului de start, care este setat cu 0.
3. Toate valorile din lista cu nodurile vizitate sunt setate cu fals.
4. Toate valorile din lista cu nodurile anterioare sunt inițializate cu -1.
5. Nodul de start este setat ca nodul curent.
6. Se marchează ca vizitat nodul curent.
7. Se actualizează distanțele, pe baza nodurilor care pot fi vizitate imediat din nodul curent.
8. Se actualizează nodul curent la nodul nevizitat care poate fi vizitat prin calea cea mai scurtă de la nodul de start.
9. Se repetă (de la punctul 6) până când toate nodurile sunt vizitate.

Observație:

- *Lungimea celui mai scurt drum special către fiecare vârf al grafului (adică D), nu se schimbă dacă mai efectuăm o iterație pentru a-l scoate și pe {2} din mulțimea vârfurilor disponibile (adică C). De aceea, bucla greedy se repetă de doar n-2 ori.*

- *La terminarea algoritmului, toate vârfurile grafului, cu excepția unuia, sunt în S. Din proprietatea precedentă, rezultă că algoritmul lui Dijkstra funcționează corect.*

Optimizarea algoritmului lui Dijktra

Să încercăm să îmbunătăţim acest algoritm. Vom reprezenta graful nu sub forma unei matrice de adiacenţă L, ci sub forma a n liste de adiacenţă, continuând pentru fiecare vârf, lungimea muchiilor care pleacă din el. Bucla for devine mai rapidă, deoarece putem considera doar nodurile adiacente lui v. Trebuie totuși să scădem și ordinul de timp pentru alegerea lui v din bucla for. Vom ţine vârfurile v $\in C$ într-un min-heap în care fiecare element este de forma (v, D[v]), proprietatea de min- heap referindu-se la valoarea lui D[v]. Acesta este Dijkstra modificat ***[53]****.*

Vom analiza în cele ce urmează ordinul de timp al acestui algoritm. Iniţializarea min-heap-ului necesită un timp O(n). Instrucţiunea C ← C\{v} devine acum extragerea rădăcinii min-heap-ului și necesită un timp în O(log n) pentru că presupune și refacerea min-heap-ului. Pentru cele n-2 extrageri avem nevoie de un timp O(n log n).

Pentru a testa dacă D[w] > D[v] + L[v, w], bucla for constă în inspectarea fiecărui vârf w adiacent lui v, ceea ce înseamnă un maxim de m operații. Dacă testul este adevărat, va trebui să modificăm, D[w] și să operăm un percolate cu w în min-heap, ceea ce presupune din nou un timp în O(log n). Timpul total este deci O(m log n).

În concluzie, algoritmul lui Dijkstra modificat necesită un timp în O(max (n, m) · log n). Dacă graful este conex, atunci m ≥ n și timpul este în O(m log n). Pentru un graf rar este preferabil să folosim algoritmul Dijkstra-modificat, iar pentru un graf dens algoritmul Dijkstra este mai eficient. Este ușor de observat că, într-un graf G neorientat conex, muchiile celor mai scurte drumuri de la un vârf i la celelalte vârfuri formează un arbore parțial al celor mai scurte drumuri pentru G. Desigur, acest arbore depinde de alegerea rădăcinii i și el diferă, în general, de arborele parțial de cost minim al lui G. Problema găsirii celor mai scurte drumuri care pleacă din același punct se poate pune și în cazul unui graf neorientat.

7.1.9 Euristica greedy

Pentru anumite probleme, se poate accepta utilizarea unor algoritmi despre care nu se știe dacă furnizează soluția optimă, dar care furnizează rezultate "acceptabile", sunt mai ușor de implementat și mai eficienți decât algoritmii care dau soluția optimă. Un astfel de algoritm se numește euristic.

Una din ideile frecvent utilizate în elaborarea algoritmilor euristici constă în descompunerea procesului de căutare a soluției optime în mai multe subprocese succesive, fiecare din aceste subprocese constând dintr-o optimizare. O astfel de strategie nu poate conduce întotdeauna la o soluție optimă, deoarece alegerea unei soluții optime la o anumită etapă poate împiedica atingerea în final a unei soluții optime a întregii probleme; cu alte cuvinte, optimizarea locală nu implică, în general, optimizarea globală. Regăsim principiul care stă la baza metodei greedy. Un algoritm greedy, despre care nu se poate demonstra că furnizează soluția optimă, este un algoritm euristic ***[50]****.*

Problema comis-voiajorului

Se cunosc distanțele dintre mai multe orașe. Un comis-voiajor pleacă dintr-un oraș și dorește să se întoarcă în același oraș, după ce a vizitat fiecare din celelalte orașe exact o dată. Problema este de a minimiza lungimea drumului parcurs. Pentru această problemă, toți algoritmii care găsesc soluția optimă sunt exponențiali.

Problema poate fi reprezentată printr-un graf neorientat, în care oricare două vârfuri diferite ale grafului sunt unite între ele printr-o muchie, de lungime pozitivă. Căutăm un ciclu de lungime minimă, care să se închidă în vârful inițial și care să treacă prin toate vârfurile grafului ***[40]****.*

Conform strategiei greedy, vom construi ciclul pas cu pas, adăugând la fiecare iterație cea mai scurtă muchie disponibilă cu următoarele proprietăți:

1. *nu formează un ciclu cu muchiile deja selectate (exceptând pentru ultima muchie aleasă, care completeaza ciclul);*
2. *nu există încă două muchii deja selectate, astfel încât cele trei muchii să fie incidente în același vârf.*

De exemplu, pentru șase orașe a căror matrice a distanțelor este dată în tabelul de mai jos (matricea distantelor pentru problema comis-voiajorului), muchiile se aleg în ordinea: {1, 2}, {3, 5}, {4, 5}, {2, 3}, {4, 6}, {1, 6}, se obține ciclul (1, 2, 3, 5, 4, 6, 1) de lungime 58.

	1	2	3	4	5	6
1		3	10	11	7	25
2			6	12	8	26
3				9	4	20
4					5	15
5						18

Notă:

- ✓ *Algoritmul greedy nu a găsit ciclul optim, deoarece ciclul (1, 2, 3, 6, 4, 5, 1) are lungimea 56.*

Colorarea unui graf

Fie G = <V, M> un graf neorientat, ale cărui vârfuri trebuie colorate, astfel încât oricare două vârfuri adiacente să fie colorate diferit. Problema este de a obține o colorare cu un număr minim de culori. Folosim următorul algoritm greedy: alegem o culoare și un vârf arbitrar de pornire, apoi considerăm vârfurile rămase, încercând să le colorăm, fără a schimba culoarea. Când niciun vârf nu mai poate fi colorat, schimbăm culoarea și vârful de start, repetând procedeul.

Fie avem graful din Figura 6:

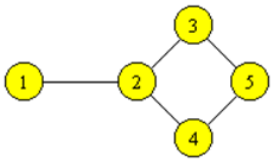


Figura 6. Un graf care va fi colorat

Pornim cu vârful 1 și îl colorăm în roșu, mai putem colora tot în roșu vârfurile 3 și 4. Apoi, schimbăm culoarea și pornim cu vârful 2, colorându-l în albastru. Mai putem colora cu albastru și vârful 5. Deci, ne-au fost suficiente două culori. Dacă colorăm vârfurile în ordinea 1, 5, 2, 3, 4, atunci se obține o colorare cu trei culori ***[51]****.*

Prin metoda greedy, nu obținem decât o soluție euristică, care nu este în mod necesar soluția optimă a problemei. De ce suntem atunci interesați într-o astfel de rezolvare? Toți algoritmii cunoscuți, care rezolvă optim această problemă, sunt exponențiali, deci, practic, nu pot fi folosiți pentru cazuri mari. Algoritmul greedy euristic propus furnizează doar o soluție "acceptabilă", dar este simplu și eficient.

Un caz particular al problemei colorării unui graf corespunde celebrei probleme a colorării hărților: O hartă oarecare trebuie colorată cu un număr minim de culori, astfel încât două țări cu frontiera comună să fie colorate diferit. Dacă fiecărui vârf îi corespunde o țară, iar două vârfuri adiacente reprezintă țări cu frontiera comună, atunci hărții îi corespunde un graf planar, adică un graf care poate fi desenat în plan fără ca două muchii să se intersecteze. Celebritatea problemei constă în faptul că în toate exemplele întâlnite, colorarea s-a putut face cu cel mult 4 culori. Aceasta în timp ce, teoretic, se putea demonstra că pentru o hartă oarecare este nevoie de cel mult 5 culori.

Problema colorării unui graf poate fi interpretată și în contextul planificării unor activități. De exemplu, să presupunem că dorim să executăm simultan o mulțime de activități, în cadrul unor săli de clasa. În acest caz, vârfurile grafului reprezintă activități, iar muchiile unesc activitățile incompatibile. Numărul minim de culori necesare pentru a colora graful corespunde numărului minim de săli necesare.

7.2 Analiza algoritmului pentru tehnica greedy

1. *Problema spectacolelor*

Enunț

- *Se dau mai multe spectacole, prin timpii de start și timpii de final.*

Cerință

- *Se cere o planificare, astfel încât o persoană să poată vedea cât mai multe spectacole.*

Soluție

- *Rezolvarea constă în sortarea spectacolelor crescător după timpii de final, apoi la fiecare pas se alege primul spectacol care are timpul de start mai mare decât ultimul timp de final. Timpul inițial de final este inițializat la −inf (spectacolul care se termină cel mai devreme va fi mereu selectat, având timp de start mai mare decât timpul inițial) .*

Complexitate temporală	Complexitate spațială
$T(n)=O(n*log(n))$ *Explicație:* • *sortarea are $O(n*log(n))$;* • *facem încă o parcurgere în $O(n)$.*	*Depinde de algoritmul de sortare folosit.*

2. *Problema florarului*

Enunț

- *Se dă un grup de k oameni care vor să cumpere împreună n flori. Fiecare floare are un preț de bază, însă prețul cu care este cumpărată variază în funcție de numărul de flori cumpărate anterior de persoana respectivă. De exemplu dacă Jorel a cumpărat 3 flori (diferite) și vrea să cumpere o floare cu prețul 2, el va plăti $(3+1)*2=8$. Practic, el va plăti un preț proporțional cu numărul florii cumpărate până atunci tot de el.*

Cerință

- *Pentru un număr k de oameni și n flori, se cere să se deterimne care este costul minim cu care grupul poate să achiziționeze toate cele n flori o singură dată.*

Soluție

- *Se observă că prețul efectiv de cumpărare va fi mai mare cu cât cumpărăm acea floare mai târziu. Dacă considerăm cazul în care avem o singură persoană în grup, observăm că are sens să cumpărăm obiectele în ordine descrescătoare (deoarece vrem să minimizăm costul fiecărui tip de flori și acesta crește cu cât cumpărăm floarea mai târziu) . De aici, gândindu-ne la versiunea cu k persoane, observăm că ar fi mai ieftin dacă am repartiza următoarea cea mai scumpă floare la alt individ. Deci, împărțim florile sortate descrescător după preț în grupuri de câte k, fiecare individ luând o floare din acest grup și ne asigurăm că prețul va crește doar în funcție de numărul de grupuri anterioare* ***[25]****.*

Complexitate temporală	Complexitate spațială
$T(n)=O(n*log(n))$ *Explicație:* • *sortarea are $O(n*log(n))$;* • *facem încă o parcurgere în $O(n)$.*	*Depinde de algoritmul de sortare folosit. Fără partea de sortare, spațiul este constant (nu se ia în considerare vectorul de elemente).*

Observație:

- *Un tip de floare se cumpără o singură dată.*
- *O persoană poate cumpăra mai multe tipuri de flori.*
- *În final, în grup va exista un singur exemplar din fiecare tip de floare.*

7.3 Implementarea tehnicii greedy la rezolvarea problemelor elementare

Problema 1: Suma maximă

Condiția problemei	*Exemplu*
Se consideră o mulțime de n numere reale. Se cere o submulțime a sa cu un număr maxim de elemente, astfel încât suma elementelor sale să fie maximă.	*Pentru n = 10 și elementele mulțimii:* `2 4 6 8 -1 -3 -5 -7 -9 0` *Programul va afișa:* `2 4 6 8` – *elementele mulțimii* `20` – *suma maximă* *Explicație:* `Suma = 2 + 4 + 6 + 8` `Suma = 20`

Implementare C++

```
#include<iostream>
using namespace std;
float A[100],B[100];
int n,m,i,suma=0;
void Greedy(){
    for(i=1;i<=n;i++)
    if(A[i]>0){
        m++;
        B[m]=A[i];
        suma+=B[m];
    }
}
int main(){
    cout<<"Introduceti dimensiunea multimii, n: \t";
    cin>>n;
    cout<<"Introduceti elementele multimii: \n";
    for(i=1;i<=n;i++){
        cout<<"A["<<i<<"]= ";
        cin>>A[i];
    }
    Greedy();
    cout<<"\nElementele multimii cautate sunt: \t";
    for(i=1;i<=m;i++)
    cout<<B[i]<<" ";
    cout<<"\nSuma maxima a elementelor este: \t";
    cout<<suma<<endl;
    return 0;
}
```

Rezultatele execuției:

```
Introduceti dimensiunea multimii, n:        5
Introduceti elementele multimii:
A[1]= -1
A[2]= 2
A[3]= -3
A[4]= 4
A[5]= -5

Elementele multimii cautate sunt:       2 4
Suma maxima a elementelor este:         6
```

Problema 2: Expresie maximă

Condiția problemei	*Exemplu*
Se dau o mulțime A cu m numere întregi nenule și o mulțime B cu n>=m numere întregi nenule. Se cere să se selecteze un șir cu m elemente din B,x1,x2,...,xm, astfel încât expresia următoare să fie maximă: $E=a_1*x_1+a_2*x_2+...+a_n*x_n$, unde, a_1, a_2,..., a_n sunt elemente ale mulțimii A într-o anumită ordine pe care trebuie să o determinați.	*Pentru n = 5 și elementele mulțimii:* `2 4 6 8 0` *Pentru n = 5 și elementele mulțimii:* `1 3 5 7 9` *Programul va afișa:* `140` – *valoarea maximă a expresiei* *Explicație:* `E = 8*9 + 6*7 + 4*5 + 2*3` `E = 72 + 42 + 20 + 6` `E = 140`

Implementare C++

```
#include<iostream>
using namespace std;
int A[30],B[30],m,n,i,E;
void Sort(int k,int X[20]){
    int inversari,man;
    do{
        inversari=0;
        for(i=1;i<=k-1;i++)
        if(X[i]>X[i+1]){
            man=X[i];
            X[i]=X[i+1]; X[i+1]=man;
            inversari=1;
        }
    }
    while(inversari);
}
int main(){
    cout<<"Introduceti dimensiunea multimii A, m: \t";
    cin>>m;
    cout<<"Introduceti elementele multimii A: \t\t";
    for(i=1;i<=m;i++)
    cin>>A[i];
    cout<<"Introduceti dimensiunea multimii B, n: \t";
    cin>>n;
    cout<<"Introduceti elementele multimii B: \t\t";
    for(i=1;i<=n;i++)
    cin>>B[i];
    Sort(m,A);
    Sort(n,B);
    for(i=1;i<=m;i++){
        E+=A[i]*B[n-m+i];
        cout<<"\tPasul: "<<i<<" valoarea: "<<E<<endl;
    }
    cout<<"Valoarea maxima a expresiei va fi: \t"<<E;
}
```

Rezultatele execuției:

```
Introduceti dimensiunea multimii A, m:      5
Introduceti elementele multimii A:          1 2 3 4 5
Introduceti dimensiunea multimii B, n:      5
Introduceti elementele multimii B:          6 7 8 9 0
        Pasul: 1 valoarea: 0
        Pasul: 2 valoarea: 12
        Pasul: 3 valoarea: 33
        Pasul: 4 valoarea: 65
        Pasul: 5 valoarea: 110
Valoarea maxima a expresiei va fi:      110
```

Problema 3: Fracție egipteană

Condiția problemei	*Exemplu*
Fiecare fracție pozitivă poate fi reprezentată ca sumă de fracții unitare unice. O fracție se numește fracție unitară dacă numărătorul este 1 și numitorul este un număr întreg pozitiv, de exemplu 1/3 este o fracție unitară. O astfel de reprezentare se numește fracție egipteană, așa cum a fost folosită de egiptenii antici. Pentru un număr dat al formularului „nr / dr" unde dr> nr, mai întâi găsiți cea mai mare fracțiune unitară posibilă, apoi recurgem pentru partea rămasă.	*Pentru nr = 6 și dr=14 programul va afișa:* `1/2 + 1/6` *Explicație:* `1. 14/6 = 2.33, deci obținem 1/3` `2. 6/14-1/3 = 4/42` `3. 42/4 = 10.5, deci obținem 1/11` `4. 4/42-1/11 = 1/231` `Soluția:` `6/14 = 1/3 + 1/11 + 1/231`

Implementare C++

```
#include <iostream>
using namespace std;
void afisare(int nr, int dr){
    if (dr == 0 || nr == 0) return;
    if (dr%nr == 0){
        cout << "1/" << dr/nr; return;
    }
    if (nr%dr == 0){
        cout << nr/dr; return;
    }
    if (nr > dr){
        cout << nr/dr << " + ";
        afisare(nr%dr, dr); return;
    }
    int n = dr/nr + 1;
    cout << "1/" << n << " + ";
    afisare(nr*n-dr, dr*n);
 }
int main(){
    int nr, dr;
    cout<<"Introduceti numaratorul fractiei: \t";
    cin>>nr;
    cout<<"Introduceti numitorul fractiei: \t";
    cin>>dr;
    cout << "Reprezentarea fractiei egiptene "<<nr<<"/"<<dr<<" este: \t";
    afisare(nr, dr);
    cout<<endl;
    return 0;
}
```

Rezultatele execuției:

```
Introduceti numaratorul fractiei:   2
Introduceti numitorul fractiei:     3
Reprezentarea fractiei egiptene 2/3 este:       1/2 + 1/6
```

Problema 4: Locuri de muncă

Condiția problemei	*Exemplu*
Având în vedere o serie de locuri de muncă în care fiecare loc de muncă are un termen limită și profit asociat dacă lucrarea este terminată înainte de termen. De asemenea, se consideră că fiecare loc de muncă are o singură unitate de timp, deci termenul minim posibil pentru orice job este 1 (o oră). Cum să maximizezi profitul total, dacă poate fi programat un singur job la un moment dat.	*Pentru 4 locuri de muncă cu următoarele termene și profituri:* a 4 20 b 1 10 c 1 40 d 1 30 *Programul va afișa:* c 1 40 a 4 20

Implementare C++

```
#include<iostream>
#include<algorithm>
using namespace std;
struct Job{
   char id;
   int dead;
   int profit;
};
bool compar(Job a, Job b){
     return (a.profit > b.profit);
}
void afisare(Job arr[], int n){
    sort(arr, arr+n, compar);
    int result[n];
    bool slot[n];
    for (int i=0; i<n; i++)
    slot[i] = false;
    for (int i=0; i<n; i++){
       for (int j=min(n, arr[i].dead)-1; j>=0; j--){
          if (slot[j]==false){
             result[j] = i; slot[j] = true; break;
          }
       }
    }
    for (int i=0; i<n; i++)
    if (slot[i]){
        cout<<arr[result[i]].id<<" "<<arr[result[i]].dead<<"
"<<arr[result[i]].profit<<endl;
        cout<<"\t";
    }
}
int main(){
    Job arr[] = { {'a', 2, 100}, {'b', 1, 19}, {'c', 2, 27},
                  {'d', 1, 25}, {'e', 3, 15}, {'f', 2, 30} };
    int n = sizeof(arr)/sizeof(arr[0]);
    cout << "Urmeaza succesiunea maxima a locurilor de munca: \n"; cout<<"\t";
    afisare(arr, n); return 0;
}
```

Rezultatele execuției:

```
Urmeaza succesiunea maxima a locurilor de munca:
        f 2 30
        a 2 100
        e 3 15
```

Problema 5: Conducte de apă

Condiția problemei	*Exemplu*
Fiecare casă din colonie are cel puțin o conductă care intră în ea și cel puțin o țeavă care iese din ea. Rezervoarele și robinetele trebuie să fie instalate într-o manieră, astfel încât fiecare casă cu o conductă de ieșire, dar nicio conductă de intrare nu primește un rezervor instalat pe acoperișul său și fiecare casă cu o conductă de intrare și nicio țeavă de ieșire nu primește un robinet. Fiind date două numere întregi n și p care indică numărul de case și numărul de conducte. Conexiunile conductei dintre case conțin trei valori de intrare: a_i, b_i, și d_i , care indică conducta cu diametrul d_i de la casa a_i la casa b_i. Aflați soluția eficientă pentru rețea. Ieșirea va conține numărul de perechi de rezervoare și robinete t instalate în prima linie, iar următoarele linii t vor conține trei întregi: numărul casei rezervorului, numărul casei de robinet și diametrul minim al conductei dintre ele.	*Pentru n=4 și p=2, apoi celelalte date:* 1 2 60 3 4 50 *Programul va afișa:* 2 1 2 60 3 4 50 *Explicație:* Componentele conectate sunt: 1-> 2 și 3-> 4. Prin urmare, răspunsul nostru este 2 urmat de 1 2 60 și 3 4 50

Implementare C++

```
#include <iostream>
#include <vector>
#include <string.h>
using namespace std;
int n, p, ans;
int rd[1100], wt[1100], cd[1100];
vector<int> a; vector<int> b; vector<int> c;
int dfs(int w){
    if (cd[w] == 0) return w;
    if (wt[w] < ans) ans = wt[w];
    return dfs(cd[w]);
}
void afiseaza(int arr[][3]){
    int i = 0;
    while (i < p) {
        int q=arr[i][0], h=arr[i][1], t=arr[i][2];
        cd[q] = h; wt[q] = t; rd[h] = q; i++;
    }
    a.clear(); b.clear(); c.clear();
    for (int j = 1; j <= n; ++j)
    if (rd[j] == 0 && cd[j]) {
        ans = 1000000000; int w = dfs(j);
        a.push_back(j); b.push_back(w); c.push_back(ans);
    }
    cout <<"Numarul total de solutii: \t"<< a.size() << endl;
    cout <<"Afisarea solutiilor: \n";
    for (int j = 0; j < a.size(); ++j)
        cout << a[j] << " " << b[j]<< " " << c[j] << endl;
}
int main(){
    n = 9, p = 6;
    memset(rd, 0, sizeof(rd)); memset(cd, 0, sizeof(cd));
    memset(wt, 0, sizeof(wt));
    int arr[][3] = { { 7, 4, 98 }, { 5, 9, 72 }, { 4, 6, 10 },
                     { 2, 8, 22 }, {9, 7, 17 }, { 3, 1, 66 } };
    afiseaza(arr); return 0;
}
```

Rezultatele execuției:

```
Numarul total de solutii:       3
Afisarea solutiilor:
2 8 22
3 1 66
5 6 10
```

Problema 6: Polițiști și hoți

Condiția problemei	*Exemplu*
Fie dat un tablou unidimensional de dimensiunea n, care are următoarele specificații: • *Fiecare element din tablă conține fie un polițist, fie un hoț.* • *Fiecare polițist poate prinde un singur hoț.* • *Un polițist nu poate prinde un hoț care se află la mai mult de unități k distanță de polițist.* *Trebuie să găsim numărul maxim de hoți care pot fi prinși.*	*Pentru n=5, k=1 și vectorul cu elementele:* P H H P H *Programul va afișa:* 2 *Explicație:* Aici pot fi prinși maxim 2 hoți, mai întâi polițistul prinde primul hoț și cel de-al doilea polițist poate prinde fie al doilea, fie al treilea hoț.

Implementare C++

```
#include <iostream>
#include <vector>
#include <cmath>
using namespace std;
int afisare(char arr[],int n,int k){
    int res = 0;
    vector<int> thi;
    vector<int> pol;
    for (int i = 0; i < n; i++){
        if (arr[i] == 'P') pol.push_back(i);
        else if (arr[i] == 'H') thi.push_back(i);
    }
    int l = 0, r = 0;
    while (l < thi.size() && r < pol.size()){
        if (abs(thi[l] - pol[r]) <= k){
            res++; l++; r++;
        }
        else if (thi[l] < pol[r]) l++;
        else r++;
    }
    return res;
}
int main(){
    int k1,k2,k3,n1,n2,n3;
    char arr1[] = { 'P', 'H', 'P', 'H', 'P', 'H', 'P', 'H', 'P', 'H'};
    k1 = 2; n1 = sizeof(arr1) / sizeof(arr1[0]);
    cout<<"Numarul maxim de hoti prinsi pentru cazul 1: \t"<< afisare(arr1, n1, k1)
<< endl;
    char arr2[] = { 'P', 'H', 'H', 'P', 'H', 'H', 'P', 'H', 'H', 'P'};
    k2 = 3; n2 = sizeof(arr2) / sizeof(arr2[0]);
    cout<<"Numarul maxim de hoti prinsi pentru cazul 2: \t"<< afisare(arr2, n2, k2)
<< endl;
    char arr3[] = { 'P', 'H', 'H', 'H', 'P', 'H', 'H', 'H', 'P', 'H'};
    k3 = 4; n3 = sizeof(arr2) / sizeof(arr3[0]);
    cout<<"Numarul maxim de hoti prinsi pentru cazul 3: \t"<< afisare(arr3, n3, k3)
<< endl;
    return 0;
}
```

Rezultatele execuției:

```
Numarul maxim de hoti prinsi pentru cazul 1:    5
Numarul maxim de hoti prinsi pentru cazul 2:    4
Numarul maxim de hoti prinsi pentru cazul 3:    3
```

Problema 7: Rafturi

Condiția problemei	*Exemplu*
Având în vedere lungimea peretelui p și rafturile de două lungimi m și n, găsiți numărul fiecărui tip de raft care trebuie utilizat și spațiul gol rămas în soluția optimă, astfel încât spațiul gol să fie minim. Cea mai mare dintre cele două rafturi este mai ieftină, de aceea este preferată. Cu toate acestea, costul este secundar și prima prioritate este minimizarea spațiului gol pe perete.	*Pentru p=24, m=3 și n=5* *Programul va afișa:* 3 3 0 *Explicație:* Folosim trei unități ale ambelor rafturi,unde 0 este spațiul rămas. 3 * 3 + 3 * 5 = 24 Deci spațiu gol = 24 - 24 = 0 O altă soluție ar fi: 8 0 0, dar din moment ce raftul mai mare de lungime 5 este mai ieftin, primul va fi răspunsul.

Implementare C++

```
#include <iostream>
using namespace std;
void afisare(int perete, int m, int n){
    int num_m = 0, num_n = 0, min_empty = perete;
    int p = 0, q = 0, rem;
    while (perete >= n) {
        p = perete / m;
        rem = perete % m;
        if (rem <= min_empty){
            num_m = p; num_n = q; min_empty = rem;
        }
        q += 1; perete = perete - n;
    }
    cout<<num_m<<" "<<num_n<<" "<<min_empty<<endl;
}
int main(){
    int perete = 30, m = 3, n = 5;
    cout<<"Afisare rezultat pentru cazul 1: \n"; cout<<"\t";
    afisare(perete, m, n);
    perete = 30, m = 4, n = 6;
    cout<<"Afisare rezultat pentru cazul 2: \n"; cout<<"\t";
    afisare(perete, m, n);
    perete = 30, m = 5, n = 7;
    cout<<"Afisare rezultat pentru cazul 3: \n"; cout<<"\t";
    afisare(perete, m, n);
    perete = 30, m = 6, n = 8;
    cout<<"Afisare rezultat pentru cazul 4: \n"; cout<<"\t";
    afisare(perete, m, n);
    perete = 30, m = 7, n = 9;
    cout<<"Afisare rezultat pentru cazul 5: \n"; cout<<"\t";
    afisare(perete, m, n);
    return 0;
}
```

Rezultatele execuției:

```
Afisare rezultat pentru cazul 1:
        5 3 0
Afisare rezultat pentru cazul 2:
        3 3 0
Afisare rezultat pentru cazul 3:
        6 0 0
Afisare rezultat pentru cazul 4:
        5 0 0
Afisare rezultat pentru cazul 5:
        3 1 0
```

Problema 8: Produs minim

Condiția problemei	*Exemplu*
Având în vedere un tablou a, trebuie să găsim un produs minim posibil cu submulțimea de elemente prezente în tablou. Produsul minim poate fi, de asemenea, un singur element.	*Pentru vectorul -1, -1, -2, 4, 3* *Programul va afișa:* `-24` *Explicație:* `Produsul minim va fi:` `(-2 * -1 * -1 * 4 * 3) = -24`

Implementare C++

```
#include <iostream>
using namespace std;
int minProductSubset(int a[], int n){
    if (n == 1) return a[0];
    int max_neg = INT_MIN, min_pos = INT_MAX;
    int count_neg = 0, count_zero = 0, prod = 1;
    for (int i = 0; i < n; i++){
        if (a[i] == 0){
            count_zero++; continue;
        }
        if (a[i] < 0){
            count_neg++;
            max_neg = max(max_neg, a[i]);
        }
        if (a[i] > 0) min_pos = min(min_pos, a[i]);
        prod = prod * a[i];
    }
    if (count_zero == n || (count_neg == 0 && count_zero > 0))
        return 0;
    if (count_neg == 0) return min_pos;
    if (!(count_neg & 1) && count_neg != 0) {
        prod = prod / max_neg;
    }
    return prod;
}
int main() {
    int a1[] = { -5, -4, -3, -2, -1 };
    int n1 = sizeof(a1) / sizeof(a1[0]);
    cout <<"\nPentru cazul 1, produsul minim este: \t";
    cout << minProductSubset(a1, n1);
    int a2[] = { -1, 0, 1, 2, 3 };
    int n2 = sizeof(a2) / sizeof(a2[0]);
    cout <<"\nPentru cazul 2, produsul minim este: \t";
    cout << minProductSubset(a2, n2);
    int a3[] = { -2, -1, 0, 1, 2 };
    int n3 = sizeof(a3) / sizeof(a3[0]);
    cout <<"\nPentru cazul 3, produsul minim este: \t";
    cout << minProductSubset(a3, n3);
    return 0;
}
```

Rezultatele execuției:

```
Pentru cazul 1, produsul minim este:
    -120
Pentru cazul 2, produsul minim este:
    -6
Pentru cazul 3, produsul minim este:
    -4
```

Problema 9: Suma minimă a produsului dintre două tablouri unidimensionale

Condiția problemei	*Exemplu*
Găsiți suma minimă a produsului dintre două tablouri unidimensionale de aceeași dimensiune, având în vedere că k modificări sunt permise pe primul tablou. În fiecare modificare, un element din primul tablou poate fi mărit sau micșorat cu 2.	*Pentru N=3, k=5, unde vectorul A: 1 2 -3 și vectorul B: -2 3 -5* *Programul va afișa:* `-31` *Explicație:* `Deci, am modificat elemtul a[2]= -3 mărindu-l cu 10 (pentru că 5 modificări sunt permise, 5*2=10).` `Suma finală va fi:` `(1 * -2)+(2 * 3)+(7 * -5) = -31`

Implementare C++

```
#include <iostream>
#include <cmath>
using namespace std;
int minim(int a[], int b[], int n, int k){
    int diff = 0, res = 0, temp;
    for (int i = 0; i < n; i++){
        int pro = a[i] * b[i];
        res = res + pro;
        if (pro < 0 && b[i] < 0) temp = (a[i] + 2 * k) * b[i];
        else if (pro < 0 && a[i] < 0) temp = (a[i] - 2 * k) * b[i];
        else if (pro > 0 && a[i] < 0) temp = (a[i] + 2 * k) * b[i];
        else if (pro > 0 && a[i] > 0) temp = (a[i] - 2 * k) * b[i];
        int d = abs(pro - temp);
        if (d > diff) diff = d;
    }
    return (res - diff);
}
int main(){
    cout <<"Cazul 1: \n";
    int n1 = 3, k1 = 5, a1[] = {1,2,-3}, b1[] = {-2,3,-5};
    cout <<"\tSuma minima conform conditiei: \t"<< minim(a1, b1, n1, k1)<< endl;
    cout <<"Cazul 2: \n";
    int n2 = 5, k2 = 3, a2[] = {2,3,4,5,4}, b2[] = {3,4,2,3,2};
    cout <<"\tSuma minima conform conditiei: \t"<< minim(a2, b2, n2, k2)<< endl;
    cout <<"Cazul 3: \n";
    int n3 = 3, k3 = 7, a3[] = {1,2,-3}, b3[] = {-2,3,-5};
    cout <<"\tSuma minima conform conditiei: \t"<< minim(a3, b3, n3, k3)<< endl;
    cout <<"Cazul 4: \n";
    int n4 = 5, k4 = 5, a4[] = {2,3,4,5,4}, b4[] = {3,4,2,3,2};
    cout <<"\tSuma minima conform conditiei: \t"<< minim(a4, b4, n4, k4)<< endl;
    return 0;
}
```

Rezultatele execuției:

```
Cazul 1:
        Suma minima conform conditiei:  -31
Cazul 2:
        Suma minima conform conditiei:  25
Cazul 3:
        Suma minima conform conditiei:  -51
Cazul 4:
        Suma minima conform conditiei:  9
```

Problema 10: Cabinet stomatologic

Condiția problemei	*Exemplu*
La un cabinet stomatologic se prezintă simultan n pacienţi. Să se determine ordinea în care medicul stomatolog va trata pacienţii, astfel încât să se minimizeze timpul mediu de aşteptare dacă se cunosc duratele tratamentelor celor n pacienţi.	*Pentru n=3 și timpul pentru fiecare pacient: t1=60, t2=10 și t3=30.* *Programul va afișa:* `Timp de asteptare optimizat: 16.6667` *Explicaţie:* *Pentru ordinea 1,2,3 tm=130/3, în timp ce pentru ordinea 2,3,1 tm=50/3 (timpul minim).*

Implementare C++

```
#include<iostream>
using namespace std;
int main(){
     int n,a[100],i,j,b[100],t=0;
     cout<<"Introduceti numarul de paciennti: \t"; cin>>n;
     for(i=1;i<=n;i++){
          cout<<i<<"\t";cin>>a[i];
     }
     for(i=1;i<=n;i++)
     b[i]=a[i];
     for(i=1;i<n;i++)
     for(j=i+1;j<=n;j++)
     if(b[i]>b[j]) swap(b[i],b[j]);
     cout<<"\nOrdinea optima:"<<endl;
     for(i=1;i<=n;i++)
     for(j=1;j<=n;j++)
     if(b[i]==a[j]){
          cout<<j<<"\t"<<b[i]<<endl;
     }
     for(i=1;i<=n;i++)
     t=t+b[i]*(n-i);
     cout<<"\nTimp de asteptare optimizat: "<<endl;
     cout<<(double)t/n;
}
```

Rezultatele execuției:

```
Introduceti numarul de paciennti:       3
1       60
2       30
3       10

Ordinea optima:
3       10
2       30
1       60

Timp de asteptare optimizat:
16.6667
```

SARCINI PENTRU EXERSARE

Problema 1 (Problema bancnotelor)	***Date de intrare***	***Date de ieşire***
Scrieţi un program, care afişează modalitatea de plată, folosind un număr minim de bancnote, a unei sume întregi S de lei (S<20000). Plata se efectuează folosind bancnote de n tipuri distincte cu valorile b1=1 leu, b2,...bn, cu valoarea de lei. Din fiecare tip de bancnote avem la dispoziţie un număr nelimitat.	67 3 1 5 10	1 2 5 1 10 6

Problema 2 (Problema ZigZag)	***Date de intrare***	***Date de ieşire***
Dintr-un fişier se citesc N numere întregi (N≤1000). Afişaţi pe prima linie a fişierului de ieşire cel mai lung ZigZag care se poate construi din numerele citite. Numim ZigZag o secvenţă de numere $a_1,a_2,...,a_m$, astfel încât să obţinem relaţia: $a_1 \le a_2 \ge a_3 \le a_4 \ge a_5 \le ... \ a_{m-1} \ge a_m$.	7 7 5 0 1 4 9 3	0 9 3 7 1 5 4

Problema 3 (Problema cuielor)	***Date de intrare***	***Date de ieşire***
Fie N scânduri de lemn, descrise ca nişte intervale închise cu capete reale. Găsiţi o mulţime minimă de cuie, astfel încât fiecare scândură să fie bătută de cel putin un cui. Se cere poziţia cuielor.	5 0 2 1 7 2 6 5 14 8 16	2 14

Problema 4 (Problema săritura calului)	***Date de intrare***	***Date de ieşire***
Fiind dată o tablă de şah de dimensiunea NxN şi un cal în colţul stânga sus al acesteia, se cere să se deplaseze calul pe tablă astfel încât să treacă o singură dată prin fiecare pătrat al tablei. Soluţia va fi afişată ca o matrice NxN în care sunt numerotate săriturile calului.	5	1 24 13 18 7 14 19 8 25 12 9 2 23 6 17 20 15 4 11 22 3 10 21 16 5

Problema 5 (Problema florarului)	***Date de intrare***	***Date de ieşire***
Se dă un grup de k oameni care vor să cumpere împreună n flori. Fiecare floare are un preţ de bază, însă preţul cu care este cumpărată variază în funcţie de numărul de flori cumpărate anterior de persoana respectivă. De exemplu, dacă Jorel a cumpărat 3 flori (diferite) şi vrea să cumpere o floare cu preţul 2, el va plăti (3+1)∗2=8. Practic, el va plăti un preţ proporţional cu numărul flori cumpărate până atunci tot de el. Pentru un număr k de oameni şi n flori, se cere să se deterimne care este costul minim cu care grupul poate să achiziţioneze toate cele n flori o singură dată.	3 5 1 2 3 4 5	25

Problema 6 (Subşiruri strict crescătoare)	***Date de intrare***	***Date de ieşire***
Se dă un şir de n numere naturale. Şirul poate fi partiţionat în mai multe moduri într-un număr de subşiruri strict crescătoare. De exemplu, şirul 4 6 2 5 8 1 3 7 poate fi partiţionat astfel: 4 6 8 (primul subşir), 2 5 7 (al doilea) şi 1 3 (al treilea). O altă modalitate este formând patru subşiruri: 4 5 7, 6 8, 2 3 şi 1. Să se determine numărul minim de subşiruri strict crescătoare în care se poate partiţiona şirul. Programul citeşte de la tastatură numărul n, iar apoi şirul de n numere naturale, separate prin spaţii. Programul va afişa pe ecran numărul minim de subşiruri strict crescătoare în care se poate partiţiona şirul.	8 4 6 2 5 8 1 3 7	3

7.4 Implementarea tehnicii greedy la rezolvarea problemelor complexe

Problema 1: Algoritmul lui Huffman pentru codificarea și decodificarea uui mesaj

Implementarea codului în limbajul C++ pentru algoritmul lui Huffman prezentat în compartimentul 7.1.6

Implementare C++

```
#include <iostream>
#include <stdlib.h>
#include <map>
#include <queue>
#define MAX_TREE_HT 100
using namespace std;
map<char, string> codes;
map<char, int> freq;
struct MinHeapNode{
     char data;
     int freq;
     MinHeapNode *left, *right;
     MinHeapNode(char data, int freq){
          left = right = NULL;
          this->data = data;
          this->freq = freq;
    }
};
struct compara{
     bool operator()(MinHeapNode* l, MinHeapNode* r){
     return (l->freq > r->freq);
    }
};
void afisCod(struct MinHeapNode* root, string str){
    if (!root) return;
    if (root->data != '$')
        cout << root->data << ": " << str << "\n";
    afisCod(root->left, str + "0");
    afisCod(root->right, str + "1");
}
void librarieCod(struct MinHeapNode* root, string str){
    if (root==NULL) return;
    if (root->data != '$')
        codes[root->data]=str;
    librarieCod(root->left, str + "0");
    librarieCod(root->right, str + "1");
}
priority_queue<MinHeapNode*, vector<MinHeapNode*>, compara> minHeap;
void HuffmanCod(int size){
    struct MinHeapNode *left, *right, *top;
    for (map<char, int>::iterator v=freq.begin(); v!=freq.end(); v++)
        minHeap.push(new MinHeapNode(v->first, v->second));
    while (minHeap.size() != 1){
        left = minHeap.top(); minHeap.pop();
        right = minHeap.top(); minHeap.pop();
        top = new MinHeapNode('$', left->freq + right->freq);
        top->left = left; top->right = right; minHeap.push(top);
    }
    librarieCod(minHeap.top(),"");
}
void calcFrecventa(string str, int n){
    for (int i=0; i<str.size(); i++)
        freq[str[i]]++;
}
string decodificaMesaj(struct MinHeapNode* root, string s){
    string ans = "";
    struct MinHeapNode* curr = root;
    for (int i=0;i<s.size();i++){
          if (s[i] == '0') curr = curr->left;
```

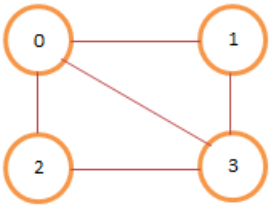
```
            else curr = curr->right;
            if (curr->left==NULL and curr->right==NULL){
                 ans += curr->data; curr = root;
            }
    }
    return ans+'\0';
}
int main(){
    string str = "ANDRIAN DASCAL";
    string codificare, decodificare;
    calcFrecventa(str, str.length());
    HuffmanCod(str.length());
    cout << "Tabelul cu datele despre fiecare caracter si frecventa:\n";
    for (auto v=codes.begin(); v!=codes.end(); v++)
        cout << "\t" << v->first << "\t" << v->second << endl;
    for (auto i: str)
        codificare+=codes[i];
    cout << "\nMesajul codificat cu ajutorul algoritmului lui Huffman:\n\t" <<
codificare << endl;
    decodificare = decodificaMesaj(minHeap.top(), codificare);
    cout << "\nMesajul decodificat cu ajutorul algoritmului lui Huffman:\n\t" <<
decodificare << endl;
    return 0;
}
```

Rezultatele execuției:

```
Tabelul cu datele despre fiecare caracter si frecventa:
                1110
        A       01
        C       1111
        D       110
        I       1010
        L       000
        N       100
        R       1011
        S       001

Mesajul codificat cu ajutorul algoritmului lui Huffman:
        01100110101110100110011101100100111110100 0

Mesajul decodificat cu ajutorul algoritmului lui Huffman:
        ANDRIAN DASCAL
```

Problema 2: Algoritmul lui Kruskal pentru determinarea costului minim

Algoritmul lui Kruskal este un algoritm în teoria grafurilor care găsește arborele parțial de cost minim pentru un graf conex ponderat.

Graful	***Soluție***
Muchiile grafului și costul pentru fiecare: 0 1 10 0 2 6 0 3 5 1 3 15 2 3 4  	*Ramurile grafului dupa sortare vor fi:* 2 3 4 0 3 5 0 2 6 0 1 10 1 3 15 *Ramurile ce alcătuiesc costul minim sunt:* 2 3 obținem costul 4 0 3 obținem costul 4+5=9 0 1 obținem costul 9+10=19 *Costul minim este: 19*

Implementare C++

```
#include    <iostream>
#include    <algorithm>
using namespace std;
const int NMax = 400005;
pair <int,int> P[NMax];
int N, M, Total, TT[NMax], k, RG[NMax];
struct Edge{
    int x,y,c;
}V[NMax];
bool Compare(Edge a, Edge b){
    return a.c < b.c;
}
void Citeste(){
    cout<<"Introduceti varfurile si muchiile: \t";
    cin >> N >> M;
    cout<<"Introduceti ramurile grafului si costurile: \n";
    for(int i = 1; i <= M; i++)
    cin >> V[i].x >> V[i].y >> V[i].c;
    sort(V+1, V+M+1, Compare);
    cout<<"Afisarea ramurile grafului dupa sortare: \n";
    for(int i = 1; i <= M; i++)
    cout<<V[i].x<<" "<<V[i].y<<" "<<V[i].c<<"\n";
}
int Find(int nod){
    while(TT[nod] != nod) nod = TT[nod];
    return nod;
}
void Unite(int x, int y){
    if(RG[x] < RG[y]) TT[x] = y;
    if(RG[y] < RG[x]) TT[y] = x;
    if(RG[x] == RG[y]){
        TT[x]=y; RG[y]++;
    }
}
void Solve(){
    for(int i=1;i<=M;i++){
        if(Find(V[i].x) != Find(V[i].y)){
            Unite(Find(V[i].x), Find(V[i].y));
            P[++k].first = V[i].x;
            P[k].second = V[i].y;
            Total+= V[i].c;
        }
    }
}
int main(){
```

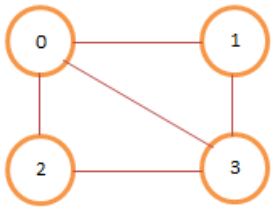
```
    Citeste();
    for(int i = 1; i <= M; i++){
        TT[i]=i; RG[i]=1;
    }
    Solve();
    cout <<"Cost minim obtinut: "<< Total << " cu "<< N-1 <<" ramuri:\n";
    for(int i = 1; i <= k; i++)
    cout << P[i].first << " " << P[i].second << "\n";
    return 0;
}
```

Rezultatele execuției:

```
Introduceti varfurile si muchiile:      4 5
Introduceti ramurile grafului si costurile:
0 1 10
0 2 6
0 3 5
1 3 15
2 3 4

Afisarea ramurile grafului dupa sortare:
2 3 4
0 3 5
0 2 6
0 1 10
1 3 15

Cost minim obtinut: 19 cu 3 ramuri:
2 3
0 3
0 1
```

Problema 3: Algoritmul lui Prim pentru determinarea costului minim

Algoritmul lui Prim este un algoritm din teoria grafurilor care găsește arborele parțial de cost minim al unui graf conex ponderat.

Graful	*Soluție*
Muchiile grafului și costul pentru fiecare: 0 1 10 0 2 6 0 3 5 1 3 15 2 3 4 	*Costul minim va fi:* 2 3 obținem costul 4 0 3 obținem costul 4+5 0 1 obținem costul 9+10 *Costul minim este: 19*

Implementare C++

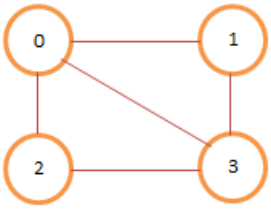
```
#include    <iostream>
#include    <algorithm>
using namespace std;
#define V 4
int Total=0;
int minKey(int key[], bool Set[]){
    int min = INT_MAX, min_index;
    for (int v = 0; v < V; v++)
    if (Set[v] == false && key[v] < min)
        min = key[v], min_index = v;
    return min_index;
}
void afisare(int parent[], int graph[V][V]){
    cout<<"Muchie\t\tCost\n";
    for (int i = 1; i < V; i++){
        cout<<parent[i]<<" "<<i<<"\t\t"<<graph[i][parent[i]]<<"\n";
        Total+=graph[i][parent[i]];
    }
}
void prim(int graph[V][V]){
    int parent[V], key[V]; bool mstSet[V];
    for (int i = 0; i < V; i++)
    key[i] = INT_MAX, mstSet[i] = false; key[0] = 0;
    parent[0] = -1;
    for (int count = 0; count < V - 1; count++){
        int u = minKey(key, mstSet);
        mstSet[u] = true;
        for (int v = 0; v < V; v++)
        if (graph[u][v] && mstSet[v] == false && graph[u][v] < key[v])
            parent[v] = u, key[v] = graph[u][v];
    }
    afisare(parent, graph);
}
int main(){
    int graph[V][V] = { { 0, 10, 6, 5 },
                        { 10, 0, 0, 15 },
                        { 6, 0, 0, 4 },
                        { 5, 15, 4, 0 } };
    prim(graph);
    cout <<"Cost minim obtinut: "<<Total; return 0;
}
```

Rezultatele execuției:

```
Muchie          Cost
0 1             10
3 2             4
0 3             5
Cost minim obtinut: 19
```

Problema 4: Algoritmul lui Dijkstra pentru determinarea drumului cel mai scurt

Algoritmul lui Dijkstra este o metodă de a stabili drumul de cost minim de la un nod de start la oricare altul dintr-un graf.

Graful	*Soluție*
Muchiile grafului și costul pentru fiecare: 0 1 10 0 2 6 0 3 5 1 3 15 2 3 4 	*Vârful sursă este 3 (al patrulea). Distanța până la celelalte vârfuri va fi:* 0 5 1 15 2 4 3 0

Implementare C++

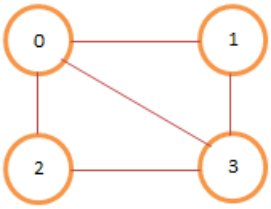
```
#include <iostream>
#include <limits.h>
#define V 4
using namespace std;
int distanta(int dist[], bool sptSet[]){
    int min = INT_MAX, min_index;
    for (int v = 0; v < V; v++)
    if (sptSet[v] == false && dist[v] <= min)
        min = dist[v], min_index = v;
    return min_index;
}
int afisare(int dist[]){
    cout<<"Varfurile \t\t Distanta de la sursa\n";
    for (int i = 0; i < V; i++){
        cout<<i<<" \t\t\t "<<dist[i]<<endl;
    }
}
void dijkstra(int graph[V][V], int src){
    int dist[V];
    bool sptSet[V];
    for (int i = 1; i < V; i++)
        dist[i] = INT_MAX, sptSet[i] = false;
        dist[src] = 0;
    for (int count = 0; count < V-1; count++) {
        int u = distanta(dist, sptSet);
        sptSet[u] = true;
        for (int v = 0; v < V; v++)
        if (!sptSet[v] && graph[u][v] && dist[u] != INT_MAX && dist[u] +
graph[u][v] < dist[v])
            dist[v] = dist[u] + graph[u][v];
    }
    afisare(dist);
}
int main(){
    int graph[V][V] = { { 0, 10, 6, 5 },
                        { 10, 0, 0, 15 },
                        { 6, 0, 0, 4 },
                        { 5, 15, 4, 0 } };
    dijkstra(graph, 3);
}
```

Rezultatele execuției:

```
Varfurile               Distanta de la sursa
0                       5
1                       15
2                       4
3                       0
```

Problema 5: Algoritmul Ford-Fulkerson pentru numărul maxim de fluxuri

Algoritmul Ford-Fulkerson este unul din algoritmii cei mai simpli care rezolvă problema "Debitului maxim". Constă în identificarea succesivă a unor drumuri de creștere până în momentul în care nu mai există niciun astfel de drum. Ca urmare, ordinul de complexitate al algoritmului Ford-Falkerson este O(F • (M + N)).

Graful	***Soluție***
Muchiile grafului și costul pentru fiecare: 0 1 10 0 2 6 0 3 5 1 3 15 2 3 4 	*Numărul total de fluxuri din graful nostru este: 19*

Implementare C++

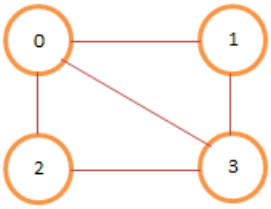
```
#include <iostream>
#include <limits.h>
#include <string.h>
#include <queue>
#define V 4
using namespace std;
bool bfs(int rGraph[V][V], int s, int t, int parent[]){
     bool visited[V]; memset(visited, 0, sizeof(visited));
     queue <int> q; q.push(s); visited[s] = true; parent[s] = -1;
     while (!q.empty()){
          int u = q.front(); q.pop();
          for (int v=0; v<V; v++){
               if (visited[v]==false && rGraph[u][v] > 0){
                    q.push(v); parent[v] = u; visited[v] = true;
               }
          }
     } return (visited[t] == true);
}
int fordFulkerson(int graph[V][V], int s, int t){
     int u, v, rGraph[V][V];
     for (u = 0; u < V; u++)
          for (v = 0; v < V; v++)
               rGraph[u][v] = graph[u][v];
     int parent[V], max_flow = 0;
     while (bfs(rGraph, s, t, parent)){
          int path_flow = INT_MAX;
          for (v=t; v!=s; v=parent[v]){
               u = parent[v]; path_flow = min(path_flow, rGraph[u][v]);
          }
          for (v=t; v != s; v=parent[v]){
               u = parent[v]; rGraph[u][v] -= path_flow; rGraph[v][u] += path_flow;
          } max_flow += path_flow;
     } return max_flow;
}
int main(){
    int graph[V][V] = { { 0, 10, 6, 5 },
                        { 10, 0, 0, 15 },
                        { 6, 0, 0, 4 },
                        { 5, 15, 4, 0 } };
     cout<<"\nNumarul maxim de fluxuri este: "<<fordFulkerson(graph, 0, V-1);
     cout <<endl; return 0;
}
```

Rezultatele execuției:

```
Numarul maxim de fluxuri este:  19
```

Problema 6: Algoritmul Edmonds – Karp pentru numărul maxim de fluxuri

*Algoritmul Edmonds-Karp, publicat de Jack Edmonds și Richard Karp reprezintă o implementare eficientă a algoritmului Ford Fulkerson. Ideea care stă la baza algoritmului este de a identifica la fiecare pas un drum de creștere care conține un număr minim de arce. După cum vom arăta în continuare, o astfel de alegere ne asigură că se vor efectua cel mult O(n*m) iterații.*

Graful	*Soluție*
Muchiile grafului și costul pentru fiecare: 0 1 10 0 2 6 0 3 5 1 3 15 2 3 4 	*Numărul total de fluxuri din graful nostru este: 19*

Implementare C++

```
#include<iostream>
#include<cstdio>
#include<queue>
#include<cstring>
#include<vector>
using namespace std;
int c[10][10];
int flowPassed[10][10];
vector<int> g[10];
int parList[10];
int currentPathC[10];
int bfs(int sNode, int eNode){
     memset(parList, -1, sizeof(parList));
     memset(currentPathC, 0, sizeof(currentPathC));
     queue<int> q; q.push(sNode);
     parList[sNode] = -1; currentPathC[sNode] = 999;
     while(!q.empty()){
          int currNode = q.front(); q.pop();
          for(int i=0; i<g[currNode].size(); i++){
               int to = g[currNode][i];
               if(parList[to] == -1){
                    if(c[currNode][to] - flowPassed[currNode][to] > 0){
                         parList[to] = currNode;
                         currentPathC[to] = min(currentPathC[currNode],
c[currNode][to] - flowPassed[currNode][to]);
                         if(to == eNode){
                              return currentPathC[eNode];
                         }
                         q.push(to);
                    }
               }
          }
   }
}
int edmondsKarp(int sNode, int eNode){
     int maxFlow = 0;
     while(true){
          int flow = bfs(sNode, eNode);
          if (flow == 0){ break; }
          maxFlow += flow;
          int currNode = eNode;
          while(currNode != sNode){
               int prevNode = parList[currNode];
               flowPassed[prevNode][currNode] += flow;
               flowPassed[currNode][prevNode] -= flow;
               currNode = prevNode;
          }
     }
     return maxFlow;
}
```

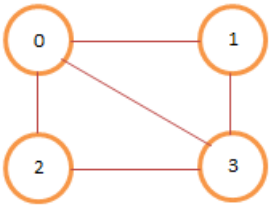
```
int main(){
     int nodCount, edCount, source, destination;
     cout<<"Introduceti numarul de varfuri si ramuri ale grafului: \n";
     cin>>nodCount>>edCount;
     cout<<"Introduceti varfurile sursa si destinatie ale grafului: \n";
     cin>>source>>destination;
     for(int ed = 0; ed < edCount; ed++){
          cout<<"Introduceti ramura "<<ed+1<<" si costul acesteia:\t";
          int from, to, cap;
          cin>>from>>to>>cap;
          c[from][to] = cap;
          g[from].push_back(to); g[to].push_back(from);
     }
     int maxFlow = edmondsKarp(source, destination);
     cout<<endl<<"Numarul maxim de fluxuri este: \t"<<maxFlow<<endl;
}
```

Rezultatele execuției:

```
Introduceti numarul de varfuri si ramuri ale grafului:
4 5
Introduceti varfurile sursa si destinatie ale grafului:
0 3
Introduceti ramura 1 si costul acesteia:        0 1 10
Introduceti ramura 2 si costul acesteia:        0 2 6
Introduceti ramura 3 si costul acesteia:        0 3 5
Introduceti ramura 4 si costul acesteia:        1 3 15
Introduceti ramura 5 si costul acesteia:        2 3 4

Numarul maxim de fluxuri este:  19
```

Problema 7: Algoritmul lui Dinic pentru numărul maxim de fluxuri

Algoritmul rulează în timp $O(v^2e)$ și este similar cu algoritmul Edmonds – Karp, care se execută în $O(ve^2)$ timpi, prin faptul că folosește cele mai scurte căi de mărire. Introducerea conceptelor grafului de nivel și a fluxului de blocare permit algoritmul lui Dinic să-și atingă performanțele.

Graful	Soluție
Muchiile grafului și costul pentru fiecare: 0 1 10 0 2 6 0 3 5 1 3 15 2 3 4 	*Numărul total de fluxuri din graful nostru este: 19*

Implementare C++

```
#include<iostream>
#include <vector>
#include <list>
using namespace std;
struct muchie{
    int v, flow, C, rev ;
};
class Graph{
    int V, *nivel ; vector< muchie > *adj;
public :
    Graph(int V){
        adj = new vector<muchie>[V];
        this->V = V; nivel = new int[V];
    }
    void adaug_muchie(int u, int v, int C){
        muchie a{v, 0, C, adj[v].size()};
        muchie b{u, 0, 0, adj[u].size()};
        adj[u].push_back(a); adj[v].push_back(b);
    }
    bool BFS(int s, int t); //Breadth First Search
    int sendFlow(int s, int flow, int t, int ptr[]);
    int DinicMaxflow(int s, int t);
};
bool Graph::BFS(int s, int t){
    for (int i = 0 ; i < V ; i++)
    nivel[i] = -1; nivel[s] = 0;
    list< int > q; q.push_back(s);
    vector<muchie>::iterator i ;
    while (!q.empty()){
        int u = q.front(); q.pop_front();
        for (i = adj[u].begin(); i != adj[u].end(); i++){
            muchie &e = *i;
            if (nivel[e.v] < 0  && e.flow < e.C){
                nivel[e.v] = nivel[u] + 1; q.push_back(e.v);
            }
        }
    }
    return nivel[t] < 0 ? false : true ;
}
int Graph::sendFlow(int u, int flow, int t, int start[]){
    if (u == t) return flow;
    for (  ; start[u] < adj[u].size(); start[u]++){
        muchie &e = adj[u][start[u]];
        if (nivel[e.v] == nivel[u]+1 && e.flow < e.C){
            int curr_flow = min(flow, e.C - e.flow);
            int temp_flow = sendFlow(e.v, curr_flow, t, start);
            if (temp_flow > 0){
                e.flow += temp_flow;
                adj[e.v][e.rev].flow -= temp_flow;
                return temp_flow;
            }
```

```
        }
    }
    return 0;
}
int Graph::DinicMaxflow(int s, int t){
    if (s == t) return -1;
    int total = 0;
    while (BFS(s, t) == true){
        int *start = new int[V+1];
        while (int flow = sendFlow(s, INT_MAX, t, start))
            total += flow;
    }
    return total;
}
int main(){
    int z=4; Graph g(z);
    g.adaug_muchie(0, 1, 10); g.adaug_muchie(0, 2, 6);
    g.adaug_muchie(0, 3, 5); g.adaug_muchie(1, 3, 15);
    g.adaug_muchie(2, 3, 4);
    cout << "\nNumarul maxim de fluxuri pentru cazul 1: \n\t";
    cout << g.DinicMaxflow(0, z-1);

    int y=5; Graph g1(y);
    g1.adaug_muchie(0, 1, 10); g1.adaug_muchie(0, 2, 6);
    g1.adaug_muchie(0, 3, 5); g1.adaug_muchie(1, 3, 15);
    g1.adaug_muchie(2, 3, 4); g1.adaug_muchie(2, 4, 7);
    cout << "\nNumarul maxim de fluxuri pentru cazul 2: \n\t";
    cout << g1.DinicMaxflow(0, y-1);

    int x=6; Graph g2(x);
    g2.adaug_muchie(0, 1, 4); g2.adaug_muchie(0, 2, 7);
    g2.adaug_muchie(0, 3, 5); g2.adaug_muchie(0, 4, 6);
    g2.adaug_muchie(0, 5, 10); g2.adaug_muchie(1, 2, 6);
    g2.adaug_muchie(1, 3, 3); g2.adaug_muchie(1, 4, 3);
    cout << "\nNumarul maxim de fluxuri pentru cazul 3: \n\t";
    cout << g2.DinicMaxflow(0, x-1);

    return 0;
}
```

Rezultatele execuției:

```
Numarul maxim de fluxuri pentru cazul 1:
        19
Numarul maxim de fluxuri pentru cazul 2:
        6
Numarul maxim de fluxuri pentru cazul 3:
        10
```

Problema 8: Comis-voiajor

Condiția problemei	*Exemplu*
Un comis-voiajor pleacă dintr-un oraș, trebuie să viziteze un număr de orase și să nu se întoarcă în orașul de unde a plecat cu efort minim. Orice oraș i este legat printr-o șosea de orice alt oraș j printr-un drum de A[i,j] kilometri. Se cere traseul pe care trebuie să-l urmeze comis-voiajorul, astfel încât să parcurgă un număr minim de kilometri.	*Pentru n=3 și nodurile: 1 5 3, unde nodul de pornire este 1.* *Programul va afișa:* `Drumul trece prin:1 2 3 1` `Cost=9` *Explicație:* `Parcurgem din 1 spre 3, d=3km` `Parcurgem din 3 spre 5, d=(3+5)km` `Parcurgem din 5 spre 1, d=(3+5+1)km.`

Implementare C++

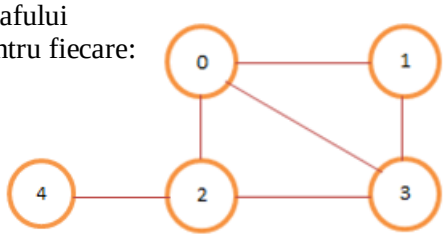
```
#include <iostream>
using namespace std;
int s[10],a[10][10],n,i,j,v,p,vs,vs1,mint,distanta;
int main(){
    cout<<"Numarul de noduri (orase) este: \t";cin>>n;
    for(i=1;i<=n;i++)
    for(j=i+1;j<=n;j++)
    cout<<"A["<<i<<"]["<<j<<"]= ",cin>>a[i][j],a[j][i]=a[i][j];
    cout<<"\nNod de pornire: \t\t";cin>>v;
    s[v]=1;
    vs1=v;
    cout<<"Drumul trece prin nodurile:\t"<<v<<" ";
    p=v;
    for(i=1;i<n;i++){
        mint=3000;
        for(j=1;j<=n;j++)
        if(a[v][j]!=0&&s[j]==0 && mint>a[v][j]){
            mint=a[v][j];
            vs=j;
        }
        distanta+=a[v][vs];
        cout<<vs<<" ";
        s[vs]=1; v=vs;
    }
    cout<<p;
    distanta+=a[vs1][v];
    cout<<endl<<"Distanta este de "<<distanta<<" km!\n";
}
```

Rezultatele execuției:

```
Numarul de noduri (orase) este:         5
A[1][2]= 2
A[1][3]= 4
A[1][4]= 6
A[1][5]= 8
A[2][3]= 10
A[2][4]= 1
A[2][5]= 3
A[3][4]= 5
A[3][5]= 7
A[4][5]= 9

Nod de pornire:                 1
Drumul trece prin nodurile:     1 2 4 3 5 1
Distanta este de 23 km!
```

Problema 9: Colorarea muchiilor unui graf

Fie avem un graf cu n vârfuri m muchii. Se cere să colorăm muchiile acestui graf, astfel încât două muchii adiacente să nu fie colorate cu aceeași culoare. Se va folosi un număr minim de culori.

Graful	*Soluție*
Muchiile grafului și costul pentru fiecare: 0 1 10 0 2 6 0 3 5 1 3 15 2 3 4 4 2 7  	*Muchiile grafului nostru vor fi colorate astfel:* 0 1 va primi culoarea 1 0 2 va primi culoarea 2 0 3 va primi culoarea 3 1 3 va primi culoarea 2 2 3 va primi culoarea 1 4 2 va primi culoarea 3

Implementare C++

```
#include <iostream>
#include <list>
#include <vector>
#include <queue>
#include <set>
#include <string.h>
using namespace std;
int n, e, i, j;
vector<vector<pair<int, int> > > g;
vector<int> color;
bool v[111001];
void col(int n) {
	queue<int> q;
	int c = 0;
	set<int> vertex_colored;
	if(v[n]) return;
	v[n] = 1;
	for(i = 0;i<g[n].size();i++){
		if(color[g[n][i].second]!=-1){
			vertex_colored.insert(color[g[n][i].second]);
		}
	}
	for(i = 0;i<g[n].size();i++){
		if(!v[g[n][i].first]){
			q.push(g[n][i].first);
		}
		if(color[g[n][i].second]==-1){
			while(vertex_colored.find(c)!=vertex_colored.end())
			c++; color[g[n][i].second] = c;
			vertex_colored.insert(c); c++;
		}
	}
	while(!q.empty()){
		int temp = q.front();
		q.pop(); col(temp);
	}
	return;
}
int main() {
	int u,w;
	set<int> empty;
	cout<<"Introduceti numarul de varfuri si ramuri ale grafului: \n";
	cin>>n>>e;
	g.resize(n); color.resize(e,-1); memset(v,0,sizeof(v));
	cout<<endl;
	for(i = 0;i<e;i++){
		cout<<"Introduceti ramura "<<i+1<<":\t";
		cin>>u>>w;
		u--; w--;
		g[u].push_back(make_pair(w,i));
```

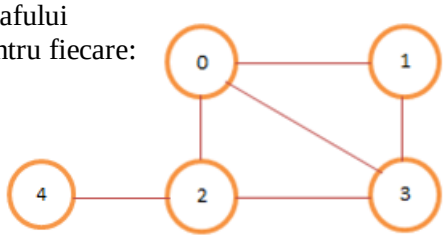
```
            g[w].push_back(make_pair(u,i));
      }
      col(0);
      for(i = 0;i<e;i++){
            cout<<"\nRamura "<<i+1<<" va fi colorata in culoarea: "<<color[i]+1;
      }
      return 0;
}
```

Rezultatele execuției:

```
Introduceti numarul de varfuri si ramuri ale grafului:
5 6

Introduceti ramura 1:   1 2
Introduceti ramura 2:   1 3
Introduceti ramura 3:   1 4
Introduceti ramura 4:   2 4
Introduceti ramura 5:   3 4
Introduceti ramura 6:   5 3

Ramura 1 va fi colorata in culoarea: 1
Ramura 2 va fi colorata in culoarea: 2
Ramura 3 va fi colorata in culoarea: 3
Ramura 4 va fi colorata in culoarea: 2
Ramura 5 va fi colorata in culoarea: 1
Ramura 6 va fi colorata in culoarea: 3
```

Problema 10: Colorarea vârfurilor unui graf

Fie avem un graf cu n vârfuri m muchii. Se cere să colorăm vârfurile acestui graf, astfel încât două vârfuri adiacente ce sunt legate între ele să nu fie colorate cu aceeași culoare. Se va folosi un număr minim de culori.

Graful	***Soluție***
Muchiile grafului și costul pentru fiecare: 0 1 10 0 2 6 0 3 5 1 3 15 2 3 4 4 2 7  	*Vârfurile grafului nostru vor fi colorate astfel:* 0 va primi culoarea 1 1 va primi culoarea 2 2 va primi culoarea 2 3 va primi culoarea 3 4 va primi culoarea 1

Implementare C++

```
#include <iostream>
#include <list>
#include <vector>
#include <queue>
#include <set>
#include <string.h>
using namespace std;
int n,e,i,j;
vector<vector<int> > graph;
vector<int> color;
bool vis[100011];
void greedyColor(){
     color[0]  = 0;
     for (i=1;i<n;i++)
          color[i] = -1;
     bool neutilizat[n];
     for (i=0;i<n;i++)
          neutilizat[i]=0;
     for (i = 1; i < n; i++){
          for (j=0;j<graph[i].size();j++)
               if (color[graph[i][j]] != -1)
                    neutilizat[color[graph[i][j]]] = true;
          int cr;
          for (cr=0;cr<n;cr++)
               if (neutilizat[cr] == false) break;
          color[i] = cr;
          for (j=0;j<graph[i].size();j++)
               if (color[graph[i][j]] != -1)
                    neutilizat[color[graph[i][j]]] = false;
     }
}
int main(){
     int x,y;
     cout<<"Introduceti numarul de varfuri si ramuri ale grafului: \n";
     cin>>n>>e;
     cout<<"\n";
     graph.resize(n); color.resize(n);
     memset(vis,0,sizeof(vis));
     for(i=0;i<e;i++) {
          cout<<"Introduceti ramura "<<i+1<<":\t";
          cin>>x>>y;
          x--; y--;
          graph[x].push_back(y); graph[y].push_back(x);
     }
     greedyColor();
     for(i=0;i<n;i++){
          cout<<"\nVarful "<<i+1<<" va fi colorata in culoarea: "<<color[i]+1;
     }
}
```

Rezultatele execuției:

```
Introduceti numarul de varfuri si ramuri ale grafului:
5 6

Introduceti ramura 1:   1 2
Introduceti ramura 2:   1 3
Introduceti ramura 3:   1 4
Introduceti ramura 4:   2 4
Introduceti ramura 5:   3 4
Introduceti ramura 6:   5 3

Varful 1 va fi colorata in culoarea: 1
Varful 2 va fi colorata in culoarea: 2
Varful 3 va fi colorata in culoarea: 2
Varful 4 va fi colorata in culoarea: 3
Varful 5 va fi colorata in culoarea: 1
```

SARCINI PENTRU EXERSARE

Problema 1 (Problema plopi)	***Date de intrare***	***Date de ieșire***
De-a lungul bulevardului principal al capitalei sunt plantați n plopi, pentru fiecare cunoscându-se înălțimea. Primarul orașului dorește ca plopii să aibă înălțimile în ordine descrescătoare. Pentru aceasta, este posibilă tăierea dintr-un plop a unei bucăți – este o tehnică ecologică, nevătămătoare, în urma căreia plopul nu are de suferit. Determinați numărul minim de plopi din care se va tăia și lungimea totală minimă a bucăților tăiate. Fișierul de intrare plopi.in *conține pe prima linie numărul de plopi n. Urmează n numere naturale nenule, separate prin spații, care pot fi dispuse pe mai multe linii, reprezentând înălțimile plopilor.* *Fișierul de ieșire plopi.out va conține pe prima linie numerele C și T, separate prin exact un spațiu, reprezentând numărul minim de plopi din care se va tăia și lungimea totală minimă a bucăților tăiate.*	`8` `5 7 3 6 4 4 2 6`	`5 11`

Problema 2 (Problema Moș Crăciun)	***Date de intrare***	***Date de ieșire***
Pentru cadourile pe care Moș Crăciun urmează să le cumpere copiilor cuminți, Consiliul Polului Nord a alocat suma de S eureni. Știind că în comerțul polar se utilizează n+1 tipuri de bancnote de valori 1, e1, e2, e3,..., en și faptul că Moșul trebuie să primească un număr minim de bancnote pentru suma aprobată, să se determine numărul de bancnote din fiecare tip utilizat în plata sumei și numărul total de bancnote care i s-au alocat. Fișierul de intrare mos.in conține pe prima linie numeral S n e. Fișierul de ieșire mos.out va conține mai multe linii: pe fiecare linie va fi scrisă valoare unei bancnote folosită în plata sumei S și numărul de bancnote folosite, separate printr-un spațiu, în ordinea descrescătoare a valorilor bancnotelor folosite. Pe ultima linie se va scrie numai numărul total de bancnote folosite.	`107 4 5`	`25 4` `5 1` `1 2` `7`

Problema 3 (Proiecte de investiții)	***Date de intrare***	***Date de ieșire***
La un birou care se ocupă cu analiza proiectelor de investiții, n investitori au depus până la termenul legal, câte un proiect.Cunoscând timpul necesar pentru analizarea fiecărui proiect, scrieți un program care determină ordinea în care vor fi analizate proiectele, astfel încât timpul mediu de așteptare pentru investitori să fie minim. Pe prima linie a fișierului proiecte.in se găsește un număr natural n, reprezentând numărul de proiecte depuse. Pe linia a doua, separate prin câte un spațiu, se găsesc n numere naturale t1, t2, ..., tn, reprezentând timpii necesari pentru analizarea fiecărui proiect. Pe prima linie a fișierului proiecte.out se vor găsi n numere naturale cuprinse între 1 și n, reprezentând ordinea în care vor fi analizate proiectele.	`5` `60 50 30 10 40`	`4 3 5 2 1`

Problema 4 (Problema depozit)	***Date de intrare***	***Date de ieșire***
Într-un depozit există un raft cu n+1 spații de depozitare, numerotate de la 1 la n+1. Primele n spații de depozitare sunt ocupate cu n pachete numerotate cu valori între 1 și n, iar spațiul de depozitare n+1 este gol. Administratorul depozitului decide mutarea pachetelor, astfel încât pentru orice i, pachetul numerotat cu i să se afle în spațiul de depozitare i. Pentru aceasta se va folosi spațiul de depozitare suplimentar, n+1, singura manevră validă fiind mutarea unui pachet dintr-un spațiu de depozitare în altul, cu condiția ca acesta să fie gol. Determinați o succesiune de manevre prin care fiecare pachet să fie în spațiul corect. Fișierul de intrare pachete.in conține pe prima linie numărul n, iar pe a doua linie n numere naturale separate prin spații. Al i-lea număr reprezintă numărul pachetului aflat în spațiul de depozitare i. Fișierul de ieșire pachete.out va conține pe prima linie numărul M, reprezentând numărul de manevre efectuate. Pe fiecare dintre următoarele M linii se descrie o manevră, prin două numere i j, cu semnificația: se ia pachetul din spațiul i și se mută în spațiul j.	`8` `4 6 2 5 8 1 3 7`	`3`

Problema 5 (Subșiruri strict crescătoare)	***Date de intrare***	***Date de ieșire***
Se dă un șir de n numere naturale. Șirul poate fi partiționat în mai multe moduri într-un număr de subșiruri strict crescătoare. De exemplu, șirul 4 6 2 5 8 1 3 7 poate fi partiționat astfel: 4 6 8 (primul subșir), 2 5 7 (al doilea) și 1 3 (al treilea). O altă modalitate este formând patru subșiruri: 4 5 7, 6 8, 2 3 și 1. Să se determine numărul minim de subșiruri strict crescătoare în care se poate partiționa șirul. Programul citește de la tastatură numărul n, iar apoi șirul de n numere naturale, separate prin spații. Programul va afișa pe ecran numărul minim de subșiruri strict crescătoare în care se poate partiționa șirul.	`8` `4 6 2 5 8 1 3 7`	`3`

Problema 6 (Problema rucsacului)	***Date de intrare***	***Date de ieșire***
O persoană are un rucsac cu care poate transporta o greutate maximă g. Persoana are la dispoziție n obiecte pentru care știe greutatea și câștigul obținut dacă transportă obiectul. Fiecare obiect poate fi transportat integral nu poate fi tăiat. Să se precizeze ce obiecte alege persoana și care este câștigul. Să nu se depășească greutatea maximă a rucsacului.	`5` `100` `10 1` `20 5` `30 10` `40 20` `50 50`	`5 4 1` `71`

Problema 7 (Problema rucsacului, cazul continuu)	***Date de intrare***	***Date de ieșire***
Condiția problemei este asemănătoare problemei precedente, doar că în acest caz obiectele pot fi tăiate. Să se precizeze ce obiecte alege persoana și în ce proporție le ia, astfel încât câștigul total să fie maxim..	`3` `100` `20 5` `40 10` `60 15`	`20 5 1` `40 10 1` `60 15 0.667` `25`

METODA PROGRAMĂRII DINAMICE

8.1 Noțiuni generale despre programare dinamică

Programarea dinamică rezolvă problemele prin descompunerea lor în subprobleme şi prin combinarea rezolvărilor acestora (termenul „programare" se referă aici la o metodă tabulară). Spre deosebire de divide et impera, care considera că subproblemele sunt independente, programarea dinamică se aplică atunci când subproblemele nu sunt independente.

Într-un astfel de caz, divide et impera ar efectua calcule redundante (surplus de informație transmis față de strictul necesar, abundență inutilă de expresii etc.) rezolvând fiecare subproblemă ca şi când nu ar mai fi întâlnit-o. Programarea dinamică, însă, salvează rezultatul fiecărei subprobleme într-o tabelă, evitând astfel rezolvarea redundantă a aceleiaşi probleme.

Programarea dinamică se aplică, în general, problemelor de optimizare, atunci când dorim să determinăm rapid soluția optimă pentru o problemă. De fapt, aplicând această tehnică determinăm una din soluţiile optime, problema poate avea mai multe soluţii optime.

Ce este programarea dinamică?

- *Este o tehnică de proiectare a algoritmilor pentru rezolvarea problemelor care pot fi descompuse în subprobleme care se suprapun – poate fi aplicată problemelor de optimizare care au proprietatea de substructură optimă.*
- *Particularitatea metodei constă în faptul că fiecare suproblemă este rezolvată o singură dată iar soluția ei este stocată (într-o structură tabelară) pentru a putea fi ulterior folosită pentru rezolvarea problemei inițiale.*

Observație:

- ❖ *Programarea dinamică a fost dezvoltată de către Richard Bellman in 1950 ca metodă generală de optimizare a proceselor de decizie. În 1957 a publicat o carte cu titlul "Dinamic programming", introducând astfel pentru prima dată conceptul de PD (programare dinamică).*
- ❖ *În programarea dinamică cuvântul programare se referă la planificare şi nu la programare în sens informatic. Cuvântul dinamic se referă la maniera în care sunt construite tabelele în care se rețin informațiile referitoare la soluțiile parțiale.*

Care este principiul optimalității?

- *Dacă* $d_1, d_2, d_3, ..., d_n$ *este un şir de decizii care duc la soluţia optimă, atunci oricare ar fi* $d_1, d_2, d_3, ..., d_i$ *o subsecvenţă de decizii, aceasta trebuie să fie optimă pentru* $\forall i \in [1, n-1]$.
- *Richard Bellman a enunţat principiul atfel: "Oricare ar fi starea iniţială şi decizia iniţială, deciziile rămase trebuie să constituie o strategie optimă privitoare la starea care rezultă din decizia anterioară".*

Notă:

- ✓ *Aplicarea acestei tehnici de programare poate fi descompusă în următoarea secvenţă de paşi:*
  1. *Descoperirea structurii şi "măsurii" pe care o are o soluţie optimă.*
  2. *Definirea recursivă a valorii care caracterizează o soluţie optimă.*
  3. *Calcularea "de jos în sus" a acestei valori.*
  4. *Construirea soluției optime pornind de la calculele efectuate anterior.*
- ✓ *Primii trei paşi reprezintă baza programării dinamice. Dacă trebuie să aflăm doar valoarea soluției optime, ultimul pas nu mai este necesar. De aceea, pentru cazul în care dorim să determinăm şi soluția optimă poate fi nevoie de construirea unor structuri de date suplimentare* ***[54]****.*

Observație:

- *Programarea dinamică este corelată cu tehnica divizării întrucât se bazează pe divizarea problemei inițiale în subprobleme. Există însă câteva diferențe semnificative între cele două abordări:*
  - *divizare: subproblemele în care se divide problema inițială sunt independente, astfel că soluția unei subprobleme nu poate fi utilizată în construirea soluției unei alte subprobleme;*
  - *programare dinamică: subproblemele sunt dependente (se suprapun), astfel că soluția unei subprobleme se utilizează în construirea soluțiilor altor subprobleme (din acest motiv este important ca soluția fiecărei subprobleme rezolvate să fie stocată pentru a putea fi reutilizată).*
- *Programarea dinamică este corelată și cu strategia căutării local optimale (greedy) întrucât ambele se aplică problemelor de optimizare care au proprietatea de substructură optimă*

Etapele principale în aplicarea programării dinamice

1. ***Analizarea structurii soluției:*** *se stabilește modul în care soluția problemei depinde de soluțiile subproblemelor. Această etapă se referă, de fapt, la verificarea proprietății de substructură optimă și la identificarea problemei generice (forma generală a problemei inițiale și a fiecărei subprobleme).*
2. ***Identificarea relației de recurență:*** *exprimă legătura între soluția problemei și soluțiile subproblemelor. De regulă, în relația de recurență intervine valoarea criteriului de optim.*
3. ***Dezvoltarea relației de recurență:*** *Relația este dezvoltată în manieră ascendentă, astfel încât să se construiască tabelul cu valorile asociate subproblemelor.*
4. ***Construirea propriu-zisă a soluției:*** *se bazează pe informațiile determinate în etapa anterioară* ***[55]***.

Dezvoltarea relațiilor de recurență

Există două abordări principale:

1. *Ascendentă (bottom up): se pornește de la cazul particular și se generează noi valori pe baza celor existente.*
2. *Descendentă (top down): valoarea de calculat se exprimă prin valori anterioare, care trebuie la rândul lor calculate. Această abordare se implementează, de regulă, recursiv (și de cele mai multe ori conduce la variante ineficiente – eficientizarea se poate realiza prin tehnica memoizării).*

Notă:

- ✓ *Memoizare este metoda prin care atunci cînd calculăm valoarea unui calcul scump, de obicei făcut într-o funcție, o păstrăm într-un tablou pentru a economisi timp în caz că acel calcul trebuie refăcut în viitor, respectiv atunci când funcția este chemată cu aceiași parametri.*
- ✓ *Un exemplu de folosire a memoizării este când scriem o funcție recursivă care să calculeze al n-ulea număr din șirul lui Fibonacci. Un mod naiv și direct de a scrie funcția este:*

```
int fib( int n ) {
    if ( n == 1 || n == 2 ) return 1;
    return (fib( n-1 ) + fib( n-2 ));
}
```

Observație:

- *Problema cu această implementare va chema de multe ori funcția fib cu aceleași valori, refăcînd frecvent aceleași calcule. De exemplu, fib(n-2) se va calcula de două ori, fib(n-3) se va calcula de 3 ori, iar fib(n-4) se va calcula de 5 ori! Faceți desenul și convingeți-vă. Numărul de apeluri crește exponențial.*

- *O soluție ar fi să implementăm funcția nerecursiv, așa cum deja știm. În unele cazuri acest lucru complică foarte mult codul. Putem însă modifica funcția fib pentru a memora valoarea o dată calculată, astfel încît la un următor apel să o returneze direct. Pentru aceasta vom folosi un vector de valori, să-i spunem val, în care vom memora valorile șirului lui Fibonacci gata calculate.*

Iată modifcarea (exemplu de funcție fibonacci implementată folosind memoizare):

```
int val[1000] = { 1, 1 };
int fib( int n ){
    if ( val[n] > 0 ) return val[n];
    return val[n] = fib( n-1 ) + fib( n-2 );
}
```

Etapele rezolvării unei probleme de programare dinamică

1. *Se identifică subproblemele problemei date;*
2. *Se alege o structură de date suplimentară capabilă să reţină soluţiile din subprobleme, eventual alte structuri în vederea reconstituirii soluţiei;*
3. *Se caracterizează substructura optimală a problemei printr-o relaţie de recurenţă;*
4. *Pentru a determina soluţia optimă, se aplică relaţia de recurenţă în mod bottom-up (se rezolvă problemele în ordine crescătoare a dimensiunii acestora)*
5. *Pe baza datelor memorate se reconstituie soluţia de sus în jos* ***[56]****.*

Abordări ale programării dinamice (verificarea principiului optimalității)

a) *Metoda înainte (în rezolvarea problemei se pleacă de la starea finală);*
b) *Metoda înapoi (în rezolvarea problemei se pleacă de la starea inițială);*
c) *Metoda mixtă (combinare înainte-înapoi).*

Analiza comparativă dintre programarea dinamică și tehnica greedy

Atât programarea dinamică, cât și tehnica greedy, pot fi folosite atunci când soluția unei probleme este privită ca rezultatul unei secvențe de decizii. Deoarece principiul optimalității poate fi exploatat de ambele metode, s-ar putea să fim tentați să elaborăm o soluție prin programare dinamică, acolo unde este suficientă o soluție greedy, sau să aplicăm în mod eronat o metodă greedy, atunci când este necesară, de fapt, aplicarea programării dinamice. Vom considera ca exemplu o problemă clasica de optimizare.

Un hoț pătrunde într-un magazin și găsește n obiecte, un obiect i având valoarea v_i și greutatea g_i. Cum să-și optimizeze hoțul profitul, dacă poate transporta cu un rucsac cel mult o greutate G?

Deosebim două cazuri:

- *în primul dintre ele, pentru orice obiect i, se poate lua orice fracțiune* $0 \le x_i \le 1$ *dine;*
- *în al doilea caz* $x_i \in \{0,1\}$*, adică orice obiect poate fi încărcat numai în întregime în rucsac.*

Observație:

- *Corespunzător acestor două cazuri, obținem problema continuă a rucsacului, respectiv, problema 0/1 a rucsacului. Evident, hoțul va selecta obiectele, astfel încât să maximizeze funcția obiectiv:* $f(x) = \sum_{i=1}^{n} v_i x_i$*, unde* $x = (x_1, x_2, \ldots, x_n)$*, verifică condiția:* $\sum_{i=1}^{n} g_i x_i \le G$.

- *Soluția problemei rucsacului poate fi privită ca rezultatul unei secvențe de decizii. De exemplu, hoțul va decide pentru început asupra valorii lui x_1, apoi asupra valorii lui x_2 etc. Printr-o secvență optimă de decizii, el va încerca să maximizeze funcția obiectiv. Se observă că este valabil principiul optimalității. Ordinea deciziilor poate fi, desigur, oricare alta.*

Partea 1

Problema continuă a rucsacului se poate rezolva prin metoda greedy, selectând la fiecare pas, pe cât posibil în întregime, obiectul pentru care v_i/g_i este maxim. Fără a restrânge generalitatea, vom presupune că $v_1 / g_1 \geq v_2 / g_2 \geq \ldots \geq v_n / g_n$.

Putem demonstra că prin acest algoritm obținem soluția optimă și că aceasta este de forma $x^* = (1, \ldots, 1, x_k^*, 0, \ldots, 0)$, *k fiind un indice unde* $1 \leq k \leq n$, *astfel încât* $0 \leq x_k \leq 1$. *Algoritmul greedy găsește secvența optimă de decizii, luând la fiecare pas câte o decizie care este optimă local. Algoritmul este corect, deoarece nicio decizie din secvență nu este eronată. Dacă nu considerăm timpul necesar sortării inițiale a obiectelor, timpul este în ordinul lui* n.

Partea 2

Să trecem la problema 0/1 a rucsacului. Se observă imediat că tehnica greedy nu conduce în general la rezultatul dorit. De exemplu, pentru $g = (1, 2, 3)$, $v = (6, 10, 12)$, $G = 5$, algoritmul greedy furnizează soluția (1, 1, 0), în timp ce soluția optimă este (0, 1, 1).

Tehnica greedy nu poate fi aplicată, deoarece este generată o decizie ($x_1 = 1$) optimă local, nu și global. Cu alte cuvinte, la primul pas, nu avem suficientă informație locală pentru a decide asupra valorii lui x_1. Strategia greedy exploatează insuficient principiul optimalității, considerând că într-o secvență optimă de decizii fiecare decizie (și nu fiecare subsecvență de decizii, cum procedează programarea dinamică) trebuie să fie optimă.

Problema se poate rezolva printr-un algoritm de programare dinamică, în această situație exploatându-se complet principiul optimalității. Spre deosebire de problema continuă, nu se cunoaște nici un algoritm polinomial pentru problema 0/1 a rucsacului.

Observație:

- *Diferența esențială dintre tehnica greedy și programarea dinamică constă în faptul că tehnica greedy generează o singură secvență de decizii, exploatând incomplet principiul optimalității.*
- *În programarea dinamică, se generează mai multe subsecvențe de decizii; ținând cont de principiul optimalității, se consideră însă doar subsecvențele optime, combinându-se acestea în soluția optimă finală.*
- *Cu toate că numărul total de secvențe de decizii este exponențial (dacă pentru fiecare din cele n decizii sunt d posibilitati, atunci sunt posibile d^n secvențe de decizii), algoritmii de programare dinamică sunt de multe ori polinomiali, această reducere a complexității datorându-se utilizării principiului optimalății.*
- *O altă caracteristică importantă a programării dinamice este că se memorează subsecvențele optime, evitându-se astfel recalcularea lor* ***[57]***.

Probleme specifice

1. *Suma maximă în triunghi.*
2. *Subşir crescător de lungime maximă.*
3. *Subşir comun de lungime maximă.*
4. *Distanţa Lowenstein.*
5. *Problema rucsacului, cazul discret.*
6. *O problemă cu sume.*
7. *Triangulaţia optimală a unui poligon convex.*
8. *Înmulţirea optimă a unui şir de matrici.*
9. *Algoritmul lui Lee. Distanţa minimă dintre două coordonate date pe o matrice.*

8.2 Analiza algoritmilor metodei de programare dinamică

Similar cu greedy, metoda de programare dinamică este folosită, în general, pentru rezolvarea problemelor de optimizare. În continuare vom folosi acronimul DP (dynamic programming). DP se poate folosi și pentru probleme în care nu căutăm un optim, cum ar fi problemele de numărare.

Exemplul 1

Fie un vector v cu n elemente întregi. O subsecvență de numere din șir este de forma: $s_i, s_{i+1},..., s_j$; *unde* $i \leq j$, *având suma asociată* $s_{ij} = s_i + s_{i+1},..., s_j$. *O subsecvență nu poate fi vidă. Să se determine subsecvența de sumă maximă.*

Rezolvare:

Tiparul acestei probleme ne sugerează că o soluție este obținută incremental, în sensul că putem privi problema astfel: găsim cea mai bună soluție folosind primele i-1 elemente din șir, apoi încercăm să o extindem folosind elementul i (adică ne extindem la dreapta cu v[i]).

Întrucât la fiecare pas trebuie să reținem cea mai bună soluție folosind un prefix din vectorul v, soluția va fi salvată într-un tablou auxiliar definit astfel: dp[i] = suma subsecvenței de sumă maximă folosind doar primele i elemente din vectorul v și care se termină pe poziția i.

Notă:

- ✓ *Pentru a menține o convenție, toate tablourile de acest tip din laborator vor fi notate cu dp (dynamic programming).*
- ✓ *Ca să rezolvăm problema dată, trebuie să rezolvăm o mulțime de subprobleme, unde dp[i] reprezintă soluția pentru problema v[1],...,v[i] și care se termină cu v[i].*
- ✓ *Soluția pentru problema inițială este maximul din vectorul* dp[i].

Găsirea recurenței

Întrucât dorim ca această problemă să fie rezolvabilă printr-un algoritm/secvență de cod, trebuie să descriem o metodă concretă prin care vom calcula dp[i].

1. *Cazul de bază*
   - *În general, în probleme putem avea mai multe cazuri de bază, care în principiu se leagă de valori extreme are dimensiunilor subproblemelor. În cazul* subsecvenței de sumă maximă (*SSM), avem un singur caz de bază, când avem un singur element în prefix: dp[1]=v[1]. Dacă avem un singur element, atunci acesta formează singura subsecvență posibilă, deci SSM=v[1].*
2. *Cazul general*
   - *Presupunem inductiv că avem rezolvate toate subproblemele mai mici. În cazul SSM, presupunem că avem calculat dp[i−1] și dorim să calculăm dp[i] (cunoaștem cea mai bună soluție folosind primele i-1 elememente și vedem dacă elementul de pe poziția i o poate îmbunătăți).*
   - *La fiecare pas avem de ales dacă v[i] extinde cea mai bună soluție care se termină pe v[i−1] sau se începe o nouă secvență cu v[i]. Decidem în funcție de dp[i−1] și v[i]:*
     - *dacă dp[i−1]>=0 (cea mai bună soluție care se termină pe i - 1 are cost nenegativ), extindem secvența care se termină cu v[i-1] folosind elementul v[i]: dp[i]=dp[i−1]+v[i], deoarece dp[i−1]+v[i]>=v[i] (încă are rost să extind);*
     - *dacă dp[i−1]<0 (cea mai bună soluție care se termină pe i - 1 are cost negativ), vom începe o nouă secvență cu v[i], adică dp[i]=v[i], deoarece v[i]>dp[i−1]+v[i], deci prin extindere nu obținem soluția maximă* ***[58]***.

Notă:

- ✓ *Întrucât această soluție presupune calculul iterativ (coloană cu coloană) a matricei dp, complexitatea este liniară. De asemenea, se mai parcurge o dată dp pentru a găsi maximul.*
- ✓ *Astfel, avem complexitate temporala : T=O(n) și complexitate spațială : S=O(n).*

Exemplul 2

Fie un vector cu n obiecte (care nu pot fi tăiate - varianta discretă a problemei). Fiecare obiect i are asociată o pereche (w_i,p_i) cu semnificația:

- *w_i = $weight_i$ = greutatea obiectului cu numărul i;*
- *p_i = $price_i$ = prețul obiectului cu numărul i, unde w_i>=0 și p_i>0.*

Un hoț are la dispoziție un rucsac de volum infinit, dar care suportă o greutate maximă (notată cu W - weight knapsack). El vrea să găsească o submulțime de obiecte pe care să le introducă în rucsac, astfel încât suma profiturilor să fie maximă. Dacă hoțul introduce în rucsac obiectul i, caracterizat de (w_i,p_i), atunci profitul adus de obiect este p_i (presupunem că îl vinde cu cât valorează obiectul). Să se determine profitul maxim pentru hoț ***[59]****.*

Rezolvare:

Întrucât la fiecare pas trebuie să reținem cea mai bună soluție folosind un prefix din vectorul de obiecte, dar pentru că trebuie să punem o restricție de greutate necesară (ocupată în rucsac), soluția va fi salvată într-un tablou auxiliar definit astfel: dp[i][cap] = profitul maxim (profit RUCSAC) obținut folosind (doar o parte) din primele i obiecte și având un rucsac de capacitate maximă cap.

Observație:

- ❖ *NU există restricție dacă în soluția menționată de dp[i][cap] este folosit OBLIGATORIU elementul i.*
- ❖ *Soluția problemei se găsește în dp[n][W] (profitul maxim folosind (doar o parte) din primele n elemente - adică toate; capacitatea maximă folosită este W - adică capacitatea maximă a rucsacului).*

Găsirea recurenței

1. *Cazul de bază*
   - *Dacă avem o submulțime vidă de obiecte selectate, dp[0][cap]=0.*
   - *Dacă nu alegem obiecte, atunci profitul este 0 indiferent de capacitate.*
2. *Cazul general*
   - *dp[i][cap]=? Pentru aceasta presupune inductiv că avem rezolvate toate subproblemele mai mici: subprobleme mai mici înseamnă să folosească mai puține obiecte sau un rucsac cu capacitatea mai mică; vedem dacă prin folosirea obiectului i, obținem cea mai bună soluție în dp[i][cap].*
   - *NU folosesc obiectul i. În acest caz, o să alegem cea mai bună soluție formată cu celelalte i−1 elemente și aceeași capacitate a rucsacului. Soluția generată de acest caz este următoarea: dp[i][cap]=dp[i−1][cap].*
   - *Folosesc obiectul i. Dacă îl folosesc, înseamnă că pentru el trebuie să am rezervată în rucsac o capacitate egala cu w_i, adică când am selectat dintre primele i−1 elemente, nu trebuia să ocup mai mult de cap− w_i din capacitatea rucsacului. Față de subproblema menționată, câștig în plus p_i (profitul pe care îl aduce acest obiect). Soluția generată de acest caz este următoarea: dp[i][cap]=dp[i−1][cap−w_i]+p_i. Reunind cele spuse mai sus, obținem:*
     - *dp[0][cap]=0, pentru cap=0:G (de la 0 până la G);*
     - *dp[i][cap]=max(dp[i−1],cap],dp[i−1][cap−w_i]+p_i), pentru i=1:n, cap=0:W;*

Notă:

- ✓ *Întrucât această soluție presupune calculul iterativ (linie cu linie) a matricei dp, complexitatea este polinomială.*
- ✓ *Astfel, avem complexitate temporală : T=O(n*W) și complexitatea spațială : S=O(n*W).*

8.3 Implementarea metodei de programare dinamică la rezolvarea problemelor elementare

Problema 1: Cel mai lung subșir crescător de valori

Condiția problemei	*Exemplu*
Se citește de la tastatură un vector v cu n elemente numere întregi. Să se afișeze lungimea celui mai lung subșir de valori alese din v, astfel încât subșirul să fie ordonat crescător și sirul respectiv.	v : 30 1 4 2 0 7 1 5 10 8 21 13 p : 1 5 4 4 5 3 4 3 2 2 1 1 d : 0 4 6 6 7 9 8 9 11 11 0 0 • *Unde p[i] este lungimea maximă a unui subșir până în punctul respectiv, care îl conține pe v[i], iar d[i] este poziția următorului element pentru reconstituirea subșirului maximal.* • *Lungimea maxima este 5. Există mai multe moduri de a scrie cel mai lung subșir maximal în acest exemplu :* 1) 1 4 7 8 21; 3) 1 4 7 8 13; 2) 1 4 7 10 21; 4) 0 1 5 10 21, etc.

Implementare C++

```
#include <iostream>
#include <fstream>
using namespace std;
ifstream fin ("intrare.txt");
int v[100], d[100], p[100], n, k;
void citire (){
     int i;
     fin>>n;
     for (i=1;i<=n;i++) fin>>v[i];
     fin.close();
}
void afisare (int k){
     while (k>0){
          cout<<v[k]<<" "; k=p[k];
     }
}
void PDinamnica (){
     int i, j, maxim, max2=n, poz;
     d[n]=1; p[n]=0;
     for(i=n-1;i>=1;i--){
          poz=0; maxim=0;
          for (j=i+1; j<=n; j++)
          if (v[i]<v[j] && d[j]>maxim){
               maxim=d[j]; poz=j;
          }
          d[i]=1+maxim; p[i]=poz;
          if (d[i]>d[max2]) max2=i;
     }
     cout<<"Lungimea maxima este: "<<d[max2]<<endl; afisare (max2);
}
int main(){
     citire(); PDinamnica(); return 0;
}
```

Rezultatele execuției:

```
Lungimea maxima este: 5
0 1 5 10 21
```

Problema 2: Combinări de n luate câte m (se vor utiliza numere mari)

Condiția problemei	*Exemplu*
Se citește de la tastatură o valoare n, apoi se va citi o valoare m, numerele m și n sunt numere mari. Să se afișeze numărul de combinări pentru valorile lui n și m, introduse de la tastatură.	*Fie n = 3 și m = 2, vom avea 3 combinații: 1 2; 1 3 și 2 3.* *Fie n = 10 și m = 5, vom avea 252 combinații.* *Fie n = 100 și m = 5, vom avea 75287520 combinații.* *Fie n = 200 și m = 10, vom avea 22451004309013280 combinații.* *Fie n = 300 și m = 15, vom avea 76878751498679488625467 20 combinații.* *Fie n = 1000 și m = 5, vom avea 8250291250200 combinații.* *Aceste calcule se pot efectua online accesând link-ul de mai jos:* *http://www.lotoxp.ro/calculatorCombinari.php*

Implementare C++

```
#include <iostream>
using namespace std;
struct num{
    unsigned char C[1200];
    int n;
}; num A[1200][1200];
void PDinamnica(num A, num B, num &S){  //S=A+B
     int i,t,c,p;
       if(A.n<B.n){
          p=B.n;//nr max de cifre
          for(i=A.n+1;i<=B.n;i++) A.C[i]=0;//completez cu 0
     }
       else {
          p=A.n;//nr max de cifre
          for(i=B.n+1;i<=A.n;i++) B.C[i]=0;//completez cu 0
     }
       t=0;
       for(i=1;i<=p;i++){
              c=A.C[i]+B.C[i]+t; //adun cifra cu cifra
              S.C[i]=c%10; t=c/10;
       }
       if(t==1){
              p++;   S.C[p]=t;
       }
       S.n=p;
}
void afisare(num A){
     for(int i=A.n;i>=1;i--)
     cout<<(int)A.C[i];
}
int main(){
     int n,k;
     cout<<"Introduceti numerele n si k: "; cin>>n>>k;
     A[0][0].n=1; A[0][0].C[1]=1;
     for(int i=1;i<=n;i++)
     for(int j=0;j<=i;j++)
     if(j==0){
          A[i][j].n=1; A[i][j].C[1]=1;
     }
     else PDinamnica(A[i-1][j-1],A[i-1][j],A[i][j]);
     cout<<"Numarul de combinari de "<<n<<" luate cate "<<k<<" este: ";
     afisare(A[n][k]); cout<<"\n"; return 0;
}
```

Rezultatele execuției:

```
Introduceti numerele n si k: 10 5
Numarul de combinari de 10 luate cate 5 este: 252
```

Problema 3: Secvența de sumă maximă

Condiția problemei	*Exemplu*
Se citește un număr natural n și apoi un vector cu n elemente întregi. Determinați secvența din vector care are suma elementelor maximă.	✓ *Fie n=9 și vectorul:* `-2 11 -3 13 -10 4 -6 25 3` *Secvența de sumă maximă este:* **`11 -3 13 -10 4 -6 25 3`** *Secvența are suma egală cu* **37** ✓ *Fie n=10 și vectorul:* `30 4 -2 7 -1 5 10 -8 21 -13` *Secvența de sumă maximă este:* **`30 4 -2 7 -1 5 10 -8 21`** *Secvența are suma egală cu* **66**

Implementare C++

```
#include <iostream>
#include <fstream>
using namespace std;
ifstream fin("intrare.txt");
ofstream fout("iesire.txt");
int afisare(int maxim){
     fout<<"\nSecventa are suma maxima egala cu: \t"<<maxim<<endl;
     cout<<"\nSecventa are suma maxima egala cu: \t"<<maxim<<endl;
}
int main(){
     int n,i,a[10000],s[10000]={0},maxim,im,jm;
     fin>>n;
     for(i=1;i<=n;i++)
          fin>>a[i];
     maxim = a[1];
     for(i=1;i<=n;i++){
          s[i]=a[i];
          if(s[i]<s[i-1]+a[i]) s[i]=s[i-1]+a[i];
          if(s[i]>maxim){
               maxim=s[i]; jm=i;
          }
     }
     im=jm;
     while(im>0 && s[im]>=0) im--;
     im++;
     afisare(maxim);
     fout<<"Secventa de suma maxima este: \t\t";
     cout<<"Secventa de suma maxima este: \t\t";
     for(i=im;i<=jm;i++){
          fout<<a[i]<<" ";
          cout<<a[i]<<" ";
     } cout<<"\n";
     return 0;
}
```

Rezultatele execuției:

Cazul 1

```
Secventa are suma maxima egala cu:      37
Secventa de suma maxima este:           11 -3 13 -10 4 -6 25 3
```

Cazul 2

```
Secventa are suma maxima egala cu:      66
Secventa de suma maxima este:           30 4 -2 7 -1 5 10 -8 21
```

Problema 4: Numărul de submulțimi cu k elemente dintr-o mulțime cu n elemente

Condiția problemei	*Exemplu*
Se citește un număr natural n și apoi un vector cu n elemente întregi. Determinați numărul de submulțimi cu k elemente din vectorul dat.	✓ *Fie n=10 și k=2, iar vectorul:* `3 1 4 2 0 7 6 5 9 8` *Numărul de submulțimi cu 2 elemente dintr-o mulțime cu 10 elemente este:* ***45*** ✓ *Fie n=10 și k=3, iar vectorul:* `3 1 4 2 0 7 6 5 9 8` *Numărul de submulțimi cu 3 elemente dintr-o mulțime cu 10 elemente este:* ***120*** ✓ *Fie n=10 și k=4, iar vectorul:* `3 1 4 2 0 7 6 5 9 8` *Numărul de submulțimi cu 4 elemente dintr-o mulțime cu 10 elemente este:* ***210*** ✓ *Fie n=10 și k=5, iar vectorul:* `3 1 4 2 0 7 6 5 9 8` *Numărul de submulțimi cu 5 elemente dintr-o mulțime cu 10 elemente este:* ***252***

Implementare C++

```
#include <iostream>
#include <fstream>
using namespace std;
ifstream fin("intrare.txt");
ofstream fout("iesire.txt");
int main(){
     unsigned long int a[500][500]={0};
     int i,j,k,n;
     /// citim din fisierul de intrare valorile lui n si k
     fin>>n>>k;
     a[0][0]=1;
     for(i=1;i<=n;i++)
     for(j=0;j<=i;j++)
     if(j==0 || j==i) a[i][j]=1;
     else a[i][j]=a[i-1][j]+a[i-1][j-1];
     /// afisarea rezultatului in fisierul de iesire
     for (i=1;i<=k;i++){
          fout<<"Numarul de submultimi cu "<<i<<" elemente dintr-o multime cu "<<n<<"
elemente este: \t";
          fout<<a[n][i]<<endl;
     }
     /// afisarea rezultatului la ecran
     for (i=1;i<=k;i++){
          cout<<"Numarul de submultimi cu "<<i<<" elemente dintr-o multime cu "<<n<<"
elemente este: \t";
          cout<<a[n][i]<<endl;
     }
     fin.close(); fout.close();
     return 0;
}
```

Rezultatele execuției:

```
Numarul de submultimi cu 1 elemente dintr-o multime cu 10 elemente este:        10
Numarul de submultimi cu 2 elemente dintr-o multime cu 10 elemente este:        45
Numarul de submultimi cu 3 elemente dintr-o multime cu 10 elemente este:        120
Numarul de submultimi cu 4 elemente dintr-o multime cu 10 elemente este:        210
Numarul de submultimi cu 5 elemente dintr-o multime cu 10 elemente este:        252
```

Problema 5: Cel mai lung subșir a două șiruri

Condiția problemei	*Exemplu*
Se citesc dintr-un fișier două șiruri x și y cu n, respectiv m elemente numere intregi. Se cere să se determine cel mai lung subșir comun al celor două șiruri și elementele comune să se afișeze în odinea inversă apariției lor în aceste șiruri.	✓ *Fie primul șir n=6 , unde x:* `30 15 12 10 60 12.` *Fie al doilea șir n=6 , unde y:* `15 70 12 71 81 10.` *Cel mai lung subșir a celo două șiruri este:* **15 12 10** ✓ *Fie primul șir n=8 , unde x:* `30 25 13 70 60 12 19 21.` *Fie al doilea șir n=6 , unde y:* `22 25 70 12 71 81 19.` *Cel mai lung subșir a celo două șiruri este:* **25 70 12 19**

Implementare C++

```
#include <iostream>
#include <fstream>
using namespace std;
ifstream fin("intrare.txt");
ofstream fout("iesire.txt");
int x[100],y[100],a[100][100],m,n,v[100],k;
void citire(){
     int i,j;
     fin>>n; for(i=1;i<=n;i++) fin>>x[i];
     fin>>m; for(i=1;i<=m;i++) fin>>y[i];
     fin.close();
}
int maxim(int x,int y){
    if(x>y)return x;
    return y;
}
void PDinamnica(){
     int i,j;
     fout<<"Lungimea maxima este: \t"; cout<<"Lungimea maxima este: \t";
     for(i=1;i<=n;i++)
     for(j=1;j<=m;j++){
          if(x[i]==y[j]) a[i][j]=a[i-1][j-1]+1;
          else a[i][j]=maxim(a[i-1][j],a[i][j-1]);
     }
     fout<<a[n][m]<<endl; cout<<a[n][m]<<endl;
}
void afisare(){
     int i,j;
     k=a[n][m];
     fout<<"Solutia este: \t\t"; cout<<"Solutia este: \t\t";
     for (i=1;i<=n && k>0;i++){
          for (j=1;j<=m && k>0;j++){
               if (x[i]==y[j]){
                    k--; fout<<x[i]<<" "; cout<<x[i]<<" ";
               }
          }
     }
}
int main(){
    citire(); PDinamnica(); afisare();
    return 0;
}
```

Rezultatele execuției:

```
Lungimea maxima este:   4
Solutia este:           25 70 12 19
```

Problema 6: Suma maximă și minimă din matrice

Condiția problemei	*Exemplu*
Se citește un număr natural n (2<=n<=20) și apoi o matrice cu n linii și n coloane având elementele numere întregi cu cel mult 4 cifre fiecare. Parcurgerea matricii se face din colțul (n,1) spre colțul (1,n) și se poate face pe direcțiile: nord, nord-est și est. *a) Afișați numărul de moduri în care se poate ajunge din colțul (n,1) în colțul (1,n).* *b) Afișați suma maximă (minimă) care se poate obține parcurgând matricea din colțul (n,1) în colțul (1,n).* *Pentru citire se va folosi un fișier de intrare, iar pentru afișare se va folosi un alt fișier.*	**Date de intrare** 4 1 2 3 -5 7 -1 3 4 2 9 -3 1 -2 8 1 9 **Date de ieșire** 63 19

Implementare C++

```
#include <iostream>
#include <fstream>
using namespace std;
ifstream fin("intrare.txt");
ofstream fout("iesire.txt");
int A[20][20],n,m,D[20][20],S1[20][20],S2[20][20];
int main(){
     fin>>n;
     for(int i=1;i<=n;i++)
     for(int j=1;j<=n;j++)
     fin>>A[i][j];
     fout<<"Numarul de moduri in care se poate ajunge din punctul ("<<n<<",1) in
(1,"<<n<<") :\n\t";
     cout<<"Numarul de moduri in care se poate ajunge din punctul ("<<n<<",1) in
(1,"<<n<<") :\n\t";
     for(int i=n;i>=1;i--)
     for(int j=1;j<=n;j++)
     if(i==n || j==1) D[i][j]=1;
     else D[i][j]=D[i][j-1]+D[i+1][j-1]+D[i+1][j];
     fout<<D[1][n]<<endl;
     cout<<D[1][n]<<endl;
     fout<<"Suma maxima care se poate obtine parcurgand matricea din punctul ("<<n<<",1)
in (1,"<<n<<") :\n\t";
     cout<<"Suma maxima care se poate obtine parcurgand matricea din punctul ("<<n<<",1)
in (1,"<<n<<") :\n\t";
     for(int i=n;i>=1;i--)
     for(int j=1;j<=n;j++)
     S1[i][j]=A[i][j]+max(max(S1[i][j-1],S1[i+1][j-1]),S1[i+1][j]);
     fout<<S1[1][n]<<endl; cout<<S1[1][n]<<endl;
     fout<<"Suma minima care se poate obtine parcurgand matricea din punctul ("<<n<<",1)
in (1,"<<n<<") :\n\t";
     cout<<"Suma minima care se poate obtine parcurgand matricea din punctul ("<<n<<",1)
in (1,"<<n<<") :\n\t";
     for(int i=n;i>=1;i--)
     for(int j=1;j<=n;j++)
     S2[i][j]=A[i][j]+min(min(S2[i][j-1],S2[i+1][j-1]),S2[i+1][j]);
     fout<<S2[1][n]<<endl; cout<<S2[1][n]<<endl;
}
```

Rezultatele execuției:

```
Numarul de moduri in care se poate ajunge din punctul (4,1) in (1,4) :
        63
Suma maxima care se poate obtine parcurgand matricea din punctul (4,1) in (1,4) :
        19
Suma minima care se poate obtine parcurgand matricea din punctul (4,1) in (1,4) :
        -4
```

Problema 7: Suma maximă într-un triunghi de numere

Condiția problemei	*Exemplu*
Se dau n(n+1)/2 numere naturale aranjate într-un triunghi format din elementele de sub și de pe diagonala unei matrici pătratice de ordin n. Se calculează sume pornind din elementul de pe prima linie prin deplasări în vecinii de sub și din dreapta. Găsiți suma maximă care se poate calcula astfel și care sunt valorile din care se obține această sumă maximă.	**Date de intrare** 6 2 3 5 6 3 4 5 6 1 4 7 8 2 1 9 2 5 3 1 7 8 **Date de ieșire** 17 2 3 6 6

Implementare C++

```
#include <iostream>
#include <fstream>
using namespace std;
ifstream fin("intrare.txt");
ofstream fout("iesire.txt");
void citire(int &n, int a[100][100]){
     int i,j;
     fin>>n;
     for(i=1;i<=n;i++)
     for(j=1;j<=i;j++)
     fin>>a[i][j];
}
void drum(int n, int a[100][100], int s[100][100], int i, int j){
     if(i>1){
          if(s[i][j]-s[i-1][j-1]==a[i][j]) drum (n,a,s,i-1,j-1);
          else drum (n,a,s,i-1,j);
          fout<<a[i][j]<<" "; cout<<a[i][j]<<" ";
     }
     else {
          fout<<a[1][1]<<" ";
          cout<<a[1][1]<<" ";
     }
}
int main(){
     int i,j,a[100][100],n,s[100][100],maxx=0,mj;
     citire(n,a);
     s[1][1]=a[1][1];
     for(i=2;i<=n;i++)
     for(j=1;j<=i;j++)
     if(j==1) s[i][j]=s[i-1][j]+a[i][j];
     else if(j==i) s[i][j]=s[i-1][j-1]+a[i][j];
          else if(s[i-1][j]<s[i-1][j-1]) s[i][j]=a[i][j]+s[i-1][j-1];
               else s[i][j]=a[i][j]+s[i-1][j];
     for(j=1;j<=n;j++) if(s[n][j]>maxx) { maxx=s[n][j]; mj=j;}
     fout<<maxx<<endl;
     cout<<"Suma maxima care se poate calcula este: \t\t"<<maxx<<endl;
     cout<<"Valorile din care s-a obtinut suma maxima sunt: \t";
     drum(n,a,s,n,mj); cout<<endl;
     fin.close(); fout.close(); return 0;
}
```

Rezultatele execuției:

```
Suma maxima care se poate calcula este:                 32
Valorile din care s-a obtinut suma maxima sunt:         2 5 4 4 9 8
```

Problema 8: Muncitorii la culesul alunelor

Condiția problemei	*Exemplu*
*O pădure este împărțită în n*m zone, în fiecare zonă crește câte un alun. Din fiecare alun cade pe jos o cantitate de alune estimată în kg. În zona stângă sus se află o echipă de muncitori care doresc să ajungă în zona dreaptă jos pentru a putea pleca acasă. Muncitorii se pot deplasa doar în două direcții: în jos sau spre dreapta. Determinați cantitatea maximă de alune pe care le poate aduna echipa de muncitori prin deplasarea din poziția inițială în cea dorită. Citirea datelor se face dintr-un fișier de intrare, care conține pe prima linie dimensiunile pădurii, adică n și m, iar apoi cantitatea de alune din fiecare dintre cele n*m zone. Afișați cantitatea maximă de alune a muncitorilor cu care pot încheia ziua de muncă. Reprezentați această cantitate în procente față de cantitatea totală de alune din pădure.*	

Date de intrare	*Date de ieșire*
7 7 0 2 1 2 1 1 0 0 1 1 0 0 0 1 1 0 1 4 0 0 0 0 1 1 0 5 0 1 0 2 1 0 0 2 3 6 0 1 0 1 5 1 0 1 1 0 0 0 1	23

Implementare C++

```
#include <iostream>
#include <fstream>
using namespace std;
ifstream fin("intrare.txt");
ofstream fout("iesire.txt");
int a[100][100],n,m,i,j,s=0;
double raport;
int main(){
       fin>>n>>m;
       for(i=1;i<=n;i++)
      for(j=1;j<=m;j++){
           fin>>a[i][j];
           s+=a[i][j]; /// cantitatea totala de alune din padure
      }
      cout<<"\nCantitatea totala de alune din padure este: \t\t"<<s<<" kg.";
      cout<<"\nCantitatea maxima de alune a muncitorilor este: \t";
      for(i=1;i<=n;i++)
      for(j=1;j<=m;j++){
           if(a[i-1][j]>a[i][j-1]) a[i][j]=a[i][j]+a[i-1][j];
           else a[i][j]=a[i][j]+a[i][j-1];
      }
      fout<<a[n][m]; cout<<a[n][m]<<" kg.";
      cout<<"\nMuncitorii au depus un efort pe parcursul zilei: \t";
      raport= double (a[n][m])/s;
      cout<<raport*100<<" %.";
      cout<<"\nMuncitorilor le-au ramas pentru ziua urmatoare: \t";
      raport= double (a[n][m])/s;
      cout<<100-(raport*100)<<" %.";
       fin.close(); fout.close();
       return 0;
}
```

Rezultatele execuției:

```
Cantitatea totala de alune din padure este:             49 kg.
Cantitatea maxima de alune a muncitorilor este:         23 kg.
Muncitorii au depus un efort pe parcursul zilei:        46.9388 %.
Muncitorilor le-au ramas pentru ziua urmatoare:         53.0612 %.
```

Problema 9: Pionul și tabla de șah.

Condiția problemei	*Exemplu*
*O tablă de șah se citește ca o matrice n*n în care pozițiile libere au valoarea 0, iar piesele sunt marcate prin valoarea 1. Pe prima linie, pe coloana js se află un pion. Să se determine drumul pe care poate ajunge pionul pe ultima linie luând un număr maxim de piese. Poziția inițială a pionului se consideră liberă. Pionul aflat în pozitia i,j se poate deplasa astfel:* *- în poziția i+1,j dacă e liberă;* *- în poziția i+1, j-1 dacă este piesă în această poziție;* *- în poziția i+1, j+1 dacă este piesă în această poziție.*	**Date de intrare** 5 3 0 0 0 0 0 0 1 0 1 0 0 1 1 1 1 0 0 0 1 1 0 1 0 1 1 **Date de ieșire** 4 1 3 2 2 3 3 4 4 5 5

Implementare C++

```
#include <iostream>
#include <fstream>
using namespace std;
ifstream fin("intrare.txt");
ofstream fout("iesire.txt");
int n,i,j, a[50][50], c[50][50],b[50][50],is,js;
void citire(){
     int i,j; fin>>n>>js; is=1;
     for(i=1;i<=n;i++)
     for(j=1;j<=n;j++) fin>>a[i][j];
}
void PDinamnica(){
     int i,j; b[is][js]=1;
     for(i=2;i<=n;i++){
          for(j=1;j<=n;j++){
               if(a[i][j]==0){
                    if(b[i-1][j]>0){ c[i][j]=c[i-1][j]; b[i][j]=1;}
               }
               else if(c[i-1][j-1]>c[i-1][j+1]){
                         if(b[i-1][j-1]>0){ c[i][j]=c[i-1][j-1]+1;b[i][j]=1;}
                         else if(b[i-1][j+1]>0){ c[i][j]=c[i-1][j+1]+1;b[i][j]=1;}
               }
               else {
                    if(b[i-1][j+1]>0){ c[i][j]=c[i-1][j+1]+1;b[i][j]=1;}
                    else if(b[i-1][j-1]>0){ c[i][j]=c[i-1][j-1]+1;b[i][j]=1;}
               }
            }
       }
}
void drum(int i, int j){
     if(i==1) fout<<i<<" "<<j<<endl;
     else{
          if(a[i][j]==0) drum(i-1,j);
          else if(c[i-1][j-1]+1==c[i][j]) drum(i-1,j-1);
               else drum(i-1,j+1);
          fout<<i<<" "<<j<<endl;
     }
}
void afis(){
     int max=0,jm;
     for(j=1;j<=n;j++)
     if(c[n][j]>max) { max=c[n][j]; jm=j; }
     fout<<max<<endl; drum(n,jm);
}
int main(){
     citire(); PDinamnica(); afis();
     fin.close(); fout.close(); return 0;
}
```

Problema 10: Tăierea unei tije

Condiția problemei	*Exemplu*
Având în vedere o tijă de lungime n și un vector de prețuri care conține prețuri la toate piesele cu dimensiuni mai mici decât n, determinați valoarea maximă și cea minimă obținută prin tăierea tijei și vânzarea pieselor. ✓ *Dacă lungimea tijei este 5 și valorile diferitelor piese sunt date, atunci valoarea maximă obținută este 13 (prin tăierea în două bucăți de lungimi 2 și 3).*	**Date de intrare** / **Date de ieșire** 5 1 2 3 4 5 2 5 8 9 7 — 13 7 6 1 2 3 4 5 6 2 5 1 8 6 4 — 15 2

Implementare C++

```
#include <iostream>
#include<limits.h>
using namespace std;
int taiere_maxima(int pret[], int n){
     if (n <= 0) return 0;
     int max_val = INT_MIN;
     for (int i = 0; i<n; i++)
     max_val = max(max_val, pret[i] + taiere_maxima(pret, n-i-1));
     return max_val;
}
int taiere_minima(int pret[], int n){
     if (n <= 0) return 0;
     int min_val = INT_MAX;
     for (int i = 0; i<n; i++)
     min_val = min(min_val, pret[i] + taiere_minima(pret, n-i-1));
     return min_val;
}
int main(){
    int a1[] = {2, 5, 8, 9, 7};
    int size = sizeof(a1)/sizeof(a1[0]);
    cout<<"Cazul 1: ";
    cout<<"\n\tValoarea maxima obtinuta este: \t"<< taiere_maxima(a1, size);
    cout<<"\n\tValoarea minima obtinuta este: \t"<< taiere_minima(a1, size);
    getchar();

    cout<<"\nCazul 2: ";
    int a2[] = {2, 5, 1, 8, 6, 4};
    size = sizeof(a2)/sizeof(a2[0]);
    cout<<"\n\tValoarea maxima obtinuta este: \t"<< taiere_maxima(a2, size);
    cout<<"\n\tValoarea minima obtinuta este: \t"<< taiere_minima(a2, size);
    getchar();
    return 0;
}
```

Rezultatele execuției:

```
Cazul 1:
        Valoarea maxima obtinuta este:  13
        Valoarea minima obtinuta este:  7

Cazul 2:
        Valoarea maxima obtinuta este:  15
        Valoarea minima obtinuta este:  2
```

SARCINI PENTRU EXERSARE

Problema 1	***Date de intrare***	***Date de ieșire***
Numerele urâte sunt numere ai căror unici factori primi sunt 2, 3 sau 5. Secvența 1, 2, 3, 4, 5, 6, 8, 9, 10, 12, 15, ... arată primele 11 numere urâte. Prin convenție, 1 este inclus. Fiind dat un număr n, sarcina este de a găsi cel de-al n-lea număr urât.	7	8
	10	12
	15	24
	150	5832

Problema 2	***Date de intrare***	***Date de ieșire***
Numerele Fibonacci sunt numerele din următoarea secvență de numere întregi: 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, etc. În termeni matematici, secvența Fn a numerelor Fibonacci este definită prin relația de recurență: Fn = Fn-1 + Fn-2, cu valori inițiale: F0 = 0 și F1 = 1. Fiind dat un număr n, sarcina este de a găsi cel de-al n-lea număr Fibonacci.	2	1
	9	34
	15	610
	30	832040
	45	1134903170

Problema 3	***Date de intrare***	***Date de ieșire***
Numerele Catalane sunt numerele din următoarea secvență de numere întregi: 1, 1, 2, 5, 14, 42, 132, 429, 1430, 4862, etc. Numerele Catalane satisfac următoarea formulă recursivă: $C_n = \frac{1}{n+1}\binom{2n}{n} = \frac{(2n)!}{(n+1)!\,n!} = \prod_{k=2}^{n}\frac{n+k}{k}$ *Fiind dat un număr n, sarcina este de a găsi cel de-al n-lea număr Catalan.*	5	42
	9	4862
	12	208012
	15	9694845
	25	43422380
	35	2527522190

Problema 4	***Date de intrare***	***Date de ieșire***
Fiind dat un număr n, n> 0, care indică numărul de cifre, sarcina este de a găsi numărul total de numere întregi pozitive cu n cifre, care nu au o natură descrescătoare. Un număr întreg care nu descrește este unul în care toate cifrele de la stânga la dreapta sunt într-o formă nedescrescătoare, exemplu: 1234, 1135, etc. Zerourile principale contează, de asemenea, în numere întregi care nu scad, cum ar fi: 0000, 0001, 0023, etc., sunt de asemenea, numere întregi cu 4 cifre care nu descresc ***[60]****.*	1	10
	2	55
	4	715
	10	92378
	12	293930
	15	1307504

Problema 5	***Date de intrare***	***Date de ieșire***
Numim un număr zecimal monoton dacă: D [i] <= D [i + 1], unde 0 <=i <=| D |. Scrieți un program care are un număr pozitiv n la intrare și returnează un număr de zecimale de lungime n care sunt strict monotone. Numărul nu poate începe cu 0.	2	36
	3	84
	4	126
	9	1

Problema 6	***Date de intrare***	***Date de ieșire***
Cea mai lungă subsecvență Zig-Zag presupune de a găsi lungimea celei mai lungi subsecvențe a secvenței date, astfel încât toate elementele acesteia să alterneze. Dacă o secvență {x1, x2, .. xn} alternează o secvență, atunci elementul ei satisface una dintre următoarele relații: $x_1 <x_2> x_3 <x_4> x_5 <... x_n$ *sau* $x_1> x_2 <x_3> x_4 <x_5> ... x_n$.	1 5 4	3
	1 4 5	2
	2 4 9 1	3
	2 4 9 1 5 7	4
	2 8 9 7 6 5	3

Problema 7	***Date de intrare***	***Date de ieșire***
Fiind date numerele n și m. Sarcina este de a găsi numărul de modalități prin care numerele mai mari sau egale cu m pot fi adăugate pentru a obține suma n.	`3 1`	`3`
	`5 1`	`7`
	`15 3`	`17`

Problema 8	***Date de intrare***	***Date de ieșire***
*Fiind dat un număr N, găsiți numărul de modalități prin care puteți desena N coarde într-un cerc cu 2 * N puncte, astfel încât să nu se intersecteze 2 acorduri. Două moduri sunt diferite dacă există o coardă care este prezentă într-un fel și nu în alta.*	`1`	`1`
	`3`	`5`
	`5`	`42`
	`7`	`429`

Problema 9	***Date de intrare***	***Date de ieșire***
Fiind dată o mulțime de m numere întregi pozitive distincte și o valoare „N”. Problema constă în numărarea numărului total de modalități prin care putem forma „N” realizând suma elementelor tabloului. Repetările și acordurile diferite sunt permise.	`7` `1 5 6`	`6`
	`14` `12 3 1 9`	`150`

Problema 10	***Date de intrare***	***Date de ieșire***
*Problema este să numărați toate căile posibile de la stânga sus la dreapta jos a unei matrice m*n cu constrângerile că de la fiecare celulă puteți muta doar spre dreapta sau în jos.*	`2 2`	`2`
	`3 4`	`10`
	`5 3`	`15`

Problema 11	***Date de intrare***	***Date de ieșire***
Având înălțimea h, numărați și afișați numărul maxim de arbori binari echilibrați posibil cu înălțimea h. Un arbore binar echilibrat este acela în care pentru fiecare nod, diferența dintre înălțimile din subarborele stâng și drept nu este mai mare decât 1.	`3`	`15`
	`4`	`315`
	`5`	`108675`
	`6`	`878720798`

Problema 12	***Date de intrare***	***Date de ieșire***
Vă aflați pe un punct (n, m) și doriți să mergeți la origine (0, 0) făcând pași fie spre stânga, fie în jos, adică din fiecare punct aveți voie să vă deplasați fie în (n-1, m), fie în (n, m-1). Găsiți numărul de căi de la un punct la origine.	`2 2`	`6`
	`2 3`	`10`
	`3 6`	`84`
	`4 5`	`126`

Problema 13	***Date de intrare***	***Date de ieșire***
Luați în considerare un joc în care un jucător poate înscrie 3 sau 5 sau 10 puncte într-o mișcare. Având un punctaj n total, găsiți un număr de moduri de a atinge scorul dat.	`15`	`3`
	`35`	`8`
	`69`	`19`

Problema 14	***Date de intrare***	***Date de ieșire***
Un copil urcă o scară cu n trepte și poate sări fie 1 pas, 2 pași sau 3 pași simultan. Aplicați metoda programării dinamice pentru a număra în câte moduri posibile copilul poate urca scările.	`3`	`4`
	`4`	`7`
	`5`	`13`

8.4 Implementarea metodei de programare dinamică la rezolvarea problemelor complexe

Problema 1: Submulțimi de sume egale

Condiția problemei	*Exemplu*
Problema partiției constă în a determina dacă o mulțime dată poate fi partiționată în două submulțimi, astfel încât suma elementelor din ambele submulțimi să fie aceeași.	✓ *Fie avem vectorul: 1, 5, 11, 5. Ca rezultat obținem partiționarea care confirmă această posibilitate, astfel avem mulțimile: {1, 5, 5} și {11}.* ✓ *Fie avem vectorul: 1, 3, 15, 5, 9. Ca rezultat nu obținem partiționarea care confirmă această posibilitate, astfel nu putem forma mulțimile.*

Implementare C++

```
#include <iostream>

using namespace std;

bool Exista (int arr[], int n, int sum){
     if (sum == 0) return true;
     if (n == 0 && sum != 0) return false;
     if (arr[n-1] > sum) return Exista (arr, n-1, sum);
     return Exista (arr, n-1, sum) || Exista (arr, n-1, sum-arr[n-1]);
}

bool Partitionare (int arr[], int n){
    // Calculam suma elementelor din tablou
    int sum = 0;
    for (int i = 0; i < n; i++)
    sum += arr[i];
    // Daca suma nu este para, nu pot exista doua submultimi cu sume egale
    if (sum%2 != 0) return false;
    return Exista(arr, n, sum/2);
}

int main(){
     cout<<"Cazul 1:\n\t";
     int a1[] = {1, 5, 11, 5};int n = sizeof(a1)/sizeof(a1[0]);
     for (int i=0;i<n;i++) cout<<a1[i]<<" ";
     if (Partitionare(a1, n) == true)
     cout << "\n\tMultimea poate fi divizata in doua submultimi de sume egale.";
     else cout << "\n\tMultimea nu poate fi divizata in doua submultimi de sume egale.";
     cout<<"\nCazul 2:\n\t";
     int a2[] = {1, 3, 15, 5, 9}, m = sizeof(a2)/sizeof(a2[0]);
     for (int i=0;i<m;i++) cout<<a2[i]<<" ";
     if (Partitionare(a2, m) == true)
     cout << "\n\tMultimea poate fi divizata in doua submultimi de sume egale.";
     else cout << "\n\tMultimea nu poate fi divizata in doua submultimi de sume egale.";
    return 0;
}
```

Rezultatele execuției:

```
Cazul 1:
        1 5 11 5
        Multimea poate fi divizata in doua submultimi de sume egale.
Cazul 2:
        1 3 15 5 9
        Multimea nu poate fi divizata in doua submultimi de sume egale.
```

Problema 2: Secvență palindrom de lungime maximă

Condiția problemei	*Exemplu*
Fiind dat un cuvânt, găsiți lungimea celei mai lungi secvențe palindromice.	✓ *Fie cuvântul: PORTOCALA, pentru acest cuvânt vom găsi segvențe (cuvinte) palindromice de maxim 3 caractere.* ✓ *Fie cuvântul: MATEMATICA, pentru acest cuvânt vom găsi segvențe (cuvinte) palindromice de maxim 5 caractere.* ✓ *Fie cuvântul: ELECTRONEGATIVITATE, pentru acest cuvânt vom găsi segvențe (cuvinte) palindromice de maxim 11 caractere.*

Implementare C++

```
#include <iostream>
#include <string.h>

using namespace std;

/// O functie pentru a obtine maximum de la doua numere intregi
int max (int x, int y) {
     return (x > y)? x : y;
}

/// Returneaza lungimea celei mai lungi secvente palindromice
int lps(char *str){
     int n = strlen(str), i, j, cl;
     /// Cream un tabel pentru a stoca rezultatele subproblemelor
     int L[n][n];
     /// Sirurile cu lungimea 1 sunt palindrom de lungime 1
     for (i = 0; i < n; i++)
     L[i][i] = 1;
     for (cl=2; cl<=n; cl++){
          for (i=0; i<n-cl+1; i++){
               j = i+cl-1;
               if (str[i] == str[j] && cl == 2) L[i][j] = 2;
               else if (str[i] == str[j]) L[i][j] = L[i+1][j-1] + 2;
               else L[i][j] = max(L[i][j-1], L[i+1][j]);
          }
     }
     return L[0][n-1];
}
int main(){
     char s1[] = "PORTOCALA", s2[] = "MATEMATICA", s3[] = "ELECTRONEGATIVITATE";
     cout<<"Cazul 1:\n\t";
     cout<<s1<<"\n\tLungimea maxima a secventei palindromice este: "<<lps(s1);
     getchar();
     cout<<"Cazul 2:\n\t";
     cout<<s2<<"\n\tLungimea maxima a secventei palindromice este: "<<lps(s2);
     getchar();
     cout<<"Cazul 3:\n\t";
     cout<<s3<<"\n\tLungimea maxima a secventei palindromice este: "<<lps(s3);
     getchar(); return 0;
}
```

Rezultatele execuției:

```
Cazul 1:
        PORTOCALA
        Lungimea maxima a secventei palindromice este: 3
Cazul 2:
        MATEMATICA
        Lungimea maxima a secventei palindromice este: 5
Cazul 3:
        ELECTRONEGATIVITATE
        Lungimea maxima a secventei palindromice este: 11
```

Problema 3: Numărul maxim de secvențe palindrom

Condiția problemei	*Exemplu*
Fiind dat un cuvânt, găsiți câte secvențe palindromice (care nu trebuie neapărat să fie distincte) pot fi formate din cuvântul dat. Rețineți că un șir gol nu este considerat palindrom.	✓ *Pentru cuvântul "ABCD", vom avea soluția egală cu 4, deoarece identificăm următoarele subsecvențe: "A", "B", "C" și "D".* ✓ *Pentru cuvântul "AAB", vom avea soluția egală cu 4, deoarece identificăm următoarele subsecvențe: "A", "A", "B" și "AA".* ✓ *Pentru cuvântul "AABC", vom avea soluția egală cu 5, deoarece identificăm următoarele subsecvențe: "A", "A", "B", "C" și "AA".*

Implementare C++

```
#include<iostream>
#include<cstring>

using namespace std;

int countPS(string str){
     int N = str.length();
     int cps[N+1][N+1];
     memset(cps, 0 ,sizeof(cps));

     // palindrom de lungimea 1
     for (int i=0; i<N; i++)
     cps[i][i] = 1;

     // verificam subsecventa de lungime L daca este palindrom
     for (int L=2; L<=N; L++){
          for (int i=0; i<N; i++){
               int k = L+i-1;
               if (str[i] == str[k]) cps[i][k] = cps[i][k-1] + cps[i+1][k] + 1;
               else cps[i][k] = cps[i][k-1] + cps[i+1][k] - cps[i+1][k-1];
          }
     }
     return cps[0][N-1];
}

int main(){
     string str = "PORTOCALA";
     cout << "Cuvantul introdus este: "<< str << endl;
     cout << "Numarul maxim de secvente palindromice este: ";
     cout<< countPS(str) << endl;
     return 0;
}
```

Rezultatele execuției:

```
Cuvantul introdus este: PORTOCALA
Numarul maxim de secvente palindromice este: 14
```

Notă:

- ✓ *Fie cuvântul: PORTOCALA, pentru acest cuvânt vom găsi 14 segvențe (cuvinte) palindromice ce pot fi formate.*
- ✓ *Fie cuvântul: MATEMATICA, pentru acest cuvânt vom găsi 38 segvențe (cuvinte) palindromice ce pot fi formate.*
- ✓ *Fie cuvântul: ELECTRONEGATIVITATE, pentru acest cuvânt vom găsi 378 segvențe (cuvinte) palindromice ce pot fi formate.*

Problema 4: Cea mai lungă creștere a succesiunii consecutive

Condiția problemei	*Exemplu*
Fie că avem n elemente, scrieți un program care tipărește cea mai lungă subsecvență în creștere a cărei diferență de element adiacent este una.	✓ *Pentru vectorul: 3 10 3 11 4 5 6 7 8 12, vom avea drept soluție următorul vector: 3 4 5 6 7 8, deoarece este cea mai lungă subsecvență în creștere al cărei element adiacent diferă cu unul.* ✓ *Pentru vectorul: 6 7 8 3 4 5 9 10, vom avea drept soluție următorul vector: 6 7 8 9 10, deoarece este cea mai lungă subsecvență în creștere al cărei element adiacent diferă cu unul.*

Implementare C++

```
#include <iostream>
#include <string.h>
#include <unordered_map>

using namespace std;

void PDinamnica(int a[], int n){
     // stocheaza indicele de elemente
     unordered_map<int, int> mp;
     // stocheaza lungimea celei mai lungi secvente care se termina cu a[i]
     int dp[n];
     memset(dp, 0, sizeof(dp));
     int maximum = INT_MIN;
     // repetam pentru toate elementele
     int index = -1;
     for (int i = 0; i < n; i++){
          // daca a[i]-1 este prezent inainte de a i-lea indice
          if (mp.find(a[i] - 1) != mp.end()){
               // ultimul index pentru a[i]-1
               int lastIndex = mp[a[i] - 1] - 1;
               // relatie
               dp[i] = 1 + dp[lastIndex];
          }
          else dp[i] = 1;
          mp[a[i]] = i + 1;
          // stocheaza cea mai lunga lungime
          if (maximum < dp[i]){
               maximum = dp[i]; index = i;
          }
     }
     for (int curr = a[index] - maximum + 1; curr <= a[index]; curr++)
     cout << curr << " ";
}

int main(){
     int a[] = { 3, 10, 3, 11, 4, 5, 6, 7, 8, 12 };
     int n = sizeof(a) / sizeof(a[0]); cout<<endl;
     cout<<"Elementele vectorului introdus sunt:\t";
     for (int i=0; i<n;i++)
          cout<<a[i]<<" "; cout<<endl;
     cout<<"Elementele vectorului solutie este:\t";
     PDinamnica(a, n); cout<<endl;
     return 0;
}
```

Rezultatele execuției:

```
Elementele vectorului introdus sunt:    3 10 3 11 4 5 6 7 8 12
Elementele vectorului solutie este:     3 4 5 6 7 8
```

Problema 5: Programarea ponderată a ofertelor de muncă

Condiția problemei	*Exemplu*
Fie că avem n locuri de muncă în care fiecare loc de muncă este reprezentat de urmărirea a trei elemente caracteristici ale acesteia: Timp de început, Timp de sfârșit și Profit. Găsiți mulțimea de profit maxim al locurilor de muncă astfel încât să nu se suprapună două locuri de muncă din mulțime.	*Fie că avem n=4 oferte de muncă cu caracteristicile de mai jos:* *Oferta 1: {8, 9, 50}; Oferta 2: {10, 12, 25}; Oferta 3: {13, 18, 100}; Oferta 4: {9, 20, 200};* *Pentru aceste oferte obținem un profit maxim de 250.* *Putem obține profitul maxim programând ofertele de muncă 1 și 4.* *Rețineți că există posibilitatea de a programa mai multe oferte de muncă: 1, 2 și 3, dar profitul cu acest program este de 50 + 25 + 100, care este mai mic de 250.*

Implementare C++

```
#include <iostream>
#include <algorithm>
using namespace std;
struct Job{
     int start, finish, profit;
};

bool verificare(Job s1, Job s2){
     return (s1.finish < s2.finish);
}

int cautareBinara(Job jobs[], int index){
     int lo = 0, hi = index - 1;
     while (lo <= hi){
          int mediu = (lo + hi) / 2;
          if (jobs[mediu].finish <= jobs[index].start){
               if (jobs[mediu + 1].finish <= jobs[index].start) lo = mediu + 1;
               else return mediu;
          }
          else hi = mediu - 1;
     }
     return -1;
}

int Profit(Job arr[], int n){
     sort(arr, arr+n, verificare);
     int *tabel = new int[n];
     tabel[0] = arr[0].profit;
     for (int i=1; i<n; i++){
          int inclProf = arr[i].profit, l = cautareBinara(arr, i);
          if (l != -1) inclProf += tabel[l];
          tabel[i] = max(inclProf, tabel[i-1]);
     }
     int result = tabel[n-1];
     delete[] tabel;
     return result;
}

int main(){
     Job a[] = {{8, 9, 50}, {10, 12, 25}, {13, 18, 100}, {9, 20, 200}};
     int n = sizeof(a)/sizeof(a[0]);
     cout << "Profitul optimal este: " << Profit(a, n);
}
```

Rezultatele execuției:

```
Profitul optimal este: 250
```

Problema 6: Numărul de căi cu exact k monede

Condiția problemei	*Exemplu*
Fie că avem o matrice în care fiecare celulă are un număr de monede. Numărați numărul de modalități de a ajunge în partea dreaptă jos din partea stângă sus cu exact k monede. Putem trece la (i + 1, j) și (i, j + 1) dintr-o celulă (i, j).	*Fie că avem k=12 și matricea:* 1 2 3 4 6 5 3 2 1 *Soluția acestei probleme este: 2* *Căile pentru a atinge numărul de monede sunt:* ▪ 1 2 6 2 1 ▪ 1 2 3 5 1

Implementare C++

```
#include <iostream>
#include <algorithm>
#include <string.h>
#include <stdlib.h>

#define R 9 //randul
#define C 5 // coloana
#define MAX_K 1000

using namespace std;

int dp[R][C][MAX_K];
int calea(int mat[][C], int m, int n, int k){
     if (m < 0 || n < 0) return 0;
     if (m==0 && n==0) return (k == mat[m][n]);
     if (dp[m][n][k] != -1) return dp[m][n][k];
     dp[m][n][k] = calea(mat,m-1,n,k-mat[m][n]) + calea(mat,m,n-1,k-mat[m][n]);
     return dp[m][n][k];
}

int PDinamnica(int mat[][C], int k){
     memset(dp, -1, sizeof dp);
     return calea(mat, R-1, C-1, k);
}

int main(){
     int k;
     int mat[R][C] = { {1, 2, 3, 4, 5},
                       {2, 3, 4, 5, 6},
                       {3, 4, 5, 6, 7},
                       {0, 1, 2, 3, 4},
                       {8, 6, 4, 2, 0},
                       {9, 7, 5, 3, 1},
                       {1, 0, 3, 2, 7}
                     };
     cout << "Introdu numarul de monede: \t"; cin>>k;
     system("CLS");
     cout << "\nNumarul de monede introduse este: \t"<<k;
     cout << "\nNumarul solutiilor pentru "<<k<<" monede: \t";
     cout << PDinamnica(mat, k);
     return 0;
}
```

Rezultatele execuției:

```
Numarul de monede introduse este:       24
Numarul solutiilor pentru 24 monede:    30
```

Problema 7: Numărul de tripletele a căror sumă este egală cu un cub perfect

Condiția problemei	*Exemplu*
Fie că avem un vector de n numere întregi, numărați toate tripletele diferite a căror sumă este egală cu cubul perfect adică, pentru orice i, j, k (i <j <k) îndeplinesc condiția ca a [i] + a [j] + a [j] = X3 unde X este orice număr întreg. 3 ≤ n ≤ 1000, 1 ≤ a [i, j, k] ≤ 5000	*Fie că avem n=5 și vectorul: 2 5 1 20 6* *Soluția acestei probleme este: 3* *Există doar 3 triplete a căror sumă totală este un cub perfect.* Indici Valori Suma 0 1 2 2 5 1 8 0 1 3 2 5 20 27 2 3 4 1 20 6 27 *Cunoaștem că 8 și 27 sunt cuburile perfecte ale lui 2 și 3.*

Implementare C++

```
#include <iostream>
#include <math.h>
using namespace std;
int dp[1000][15000];

void numarare(int arr[], int n){
     for (int i = 0; i < n; ++i){
          for (int j = 1; j <= 15000; ++j){
               if (i == 0) dp[i][j] = (j == arr[i]);
               else dp[i][j] = dp[i - 1][j] + (arr[i] == j);
          }
     }
}

int PDinamnica(int arr[], int n){
     numarare(arr, n);
     int ans = 0;  // initializarea raspunsului
     for (int i = 0; i < n - 2; ++i){
          for (int j = i + 1; j < n - 1; ++j){
               for (int k = 1; k <= 24; ++k){
                    int cube = pow(k,3); // adica k * k * k;
                    int rem = cube - (arr[i] + arr[j]);
                    if (rem > 0) ans += dp[n - 1][rem] - dp[j][rem];
               }
          }
     }
     return ans;
}

int main(){
     int a[] = { 2, 5, 1, 20, 6 };
     int n = sizeof(a) / sizeof(a[0]);
     cout<<"Elementele vectorului introdus sunt:\t";
     for (int i=0; i<n;i++)
          cout<<a[i]<<" "; cout<<endl;
     cout<<"Numarul de triplete a caror suma este egala cu un cub perfect este:\t";
     cout << PDinamnica(a, n);
     return 0;
}
```

Rezultatele execuției:

```
Elementele vectorului introdus sunt:    2 5 1 20 6
Numarul de triplete a caror suma este egala cu un cub perfect este:     3
```

Problema 8: Valoarea maximă și minimă ale unei expresii algebrice

Condiția problemei	*Exemplu*
*Fie că avem o expresie algebrică ce are următoarea formă: (x1 + x2 + x3 +.. + Xn) * (y1 + y2 +.. + Ym) și (n + m) numere întregi. Găsiți valoarea maximă și cea minimă a expresiei folosind numerele întregi. Constrângeri: n <= 50, m <= 50, -50 <= x1, x2, .. xn <= 50.*	*Fie că avem n=2, m=2 și vectorul: 1 2 3 4* *Soluția acestei probleme este: 25 21* *Expresia este (x1 + x2) * (y1 + y2) și numerele întregi date sunt 1, 2, 3 și 4. Atunci valoarea maximă este (1 + 4) * (2 + 3) = 25, iar valoarea minimă este (4 + 3) * (2 + 1) = 21.*

Implementare C++

```
#include <bits/stdc++.h>
using namespace std;
#define INF 1e9
#define MAX 50

int PDinamnica(int a[], int n, int m){
     int suma = 0;
     for (int i = 0; i < (n + m); i++){
          suma+= a[i]; a[i] += 50;
     }
     bool dp[MAX+1][MAX * MAX + 1];
     memset(dp, 0, sizeof(dp));
     dp[0][0] = 1;
     // daca dp[i][j] este adevarat, inseamna ca este posibil sa selectam numerele i din
numere(n+m) pana la j
     for (int i = 0; i < (n + m); i++){
          // k poate avea valoarea maxim n, deoarece expresia din stanga are n numere
          for (int k = min(n, i + 1); k >= 1; k--){
               for (int j = 0; j < MAX * MAX + 1; j++){
                    if (dp[k - 1][j]) dp[k][j + a[i]] = 1;
               }
          }
     }
     int maxim = -INF, minim = INF;
     for (int i = 0; i < MAX * MAX + 1; i++){
          if (dp[n][i]){
               int temp = i - 50 * n;
               maxim=max(maxim,temp*(suma-temp)); minim=min(minim,temp*(suma-temp));
          }
     }
     cout << "\nValoarea maxima a expresiei este: " << maxim << "\n";
     cout << "Valoarea minima a expresiei este: " << minim << endl;
}
int main(){
     int n = 2, m = 2, a[] = { 1, 2, 3, 4 }, dim = sizeof(a) / sizeof(a[0]);
     cout<<"Fie expresia:\t(x1+x2+x3+...+xn)*(y1+y2+y3+...+ym)\n";
     cout<<"Elementele vectorului introdus sunt:\t";
     for (int i=0; i<dim;i++)
          cout<<a[i]<<" "; cout<<endl;
     PDinamnica(a, n, m); return 0;
}
```

Rezultatele execuției:

```
Fie expresia:   (x1+x2+x3+...+xn)*(y1+y2+y3+...+ym)
Elementele vectorului introdus sunt:    1 2 3 4

Valoarea maxima a expresiei este: 25
Valoarea minima a expresiei este: 21
```

Problema 9: Submatrice de sumă minimă

Condiția problemei	*Exemplu*
*Fie că avem o matrice de dimensiunea n*m. Se cere de a afișa o submatrice care să posede suma minimă a elementelor.*	Fie avem matricea: 1 2 -1 -4 -20 -8 -3 4 2 1 3 8 10 1 3 -4 -1 1 7 -6 *Soluția acestei probleme este: -26* *Submatricea începe din (sus, stânga): (0, 0) și se închide în (jos, dreapta): (1, 4). Elementele din submatrice sunt:* 1 2 -1 -4 -20 -8 -3 4 2 1

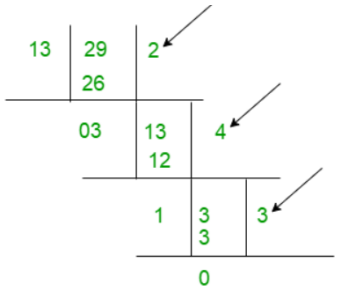
Implementare C++

```
#include <iostream>
#include <string.h>
#define R 4 // Randul
#define C 5 // Coloana
using namespace std;
int indice (int *a,int* start,int *finish,int n){
     int suma = 0, SumaMin = INT_MAX, i, inceput = 0;
     *finish = -1;
     for (i = 0; i < n; ++i){
          suma += a[i];
          if (suma > 0){suma = 0; inceput = i + 1;}
          else if (suma < SumaMin){
               SumaMin = suma; *start = inceput; *finish = i;
          }
     }
     if (*finish != -1) return SumaMin;
     SumaMin = a[0]; *start = *finish = 0;
     for (i = 1; i < n; i++){
          if (a[i] < SumaMin){
               SumaMin = a[i]; *start = *finish = i;
          }
     }
     return SumaMin;
}
void PDinamnica(int M[][C]){
     int SumaMin = INT_MAX,Stanga,Dreapta,Sus,Jos;
     int left,right,i,temp[R],sum,start,finish;
     for (left = 0; left < C; ++left){
          memset(temp, 0, sizeof(temp));
          for (right = left; right < C; ++right){
               for (i = 0; i < R; ++i)
               temp[i] += M[i][right]; sum = indice(temp, &start, &finish, R);
               if (sum < SumaMin){
                    SumaMin = sum; Stanga = left; Dreapta = right;
                    Sus = start; Jos = finish;
               }
          }
     }
     cout << "Indicii:\t\t("<<Sus<<","<<Stanga<<") si ("<<Jos<<","<<Dreapta<<")\n";
     cout << "Suma minima este: \t" << SumaMin; cout<<endl;
}
int main(){
     int M[R][C] = { {1,2,-1,-4,-20}, {-8,-3,4,2,1}, {3,8,10,1,3}, {-4,-1,1,7,-6} };
     PDinamnica(M); return 0;
}
```

Rezultatele execuției:

```
Indicii:                (0,0) si (1,4)
Suma minima este:       -26
```

Problema 10: Tăierea unei fișe într-un număr minim de pătrate

Condiția problemei	*Exemplu*
Fie avem o hârtie de dimensiunea A x B. Sarcina este de a tăia hârtia în pătrate de orice dimensiune. Găsiți numărul minim de pătrate care pot fi tăiate din hârtie.	Fie avem dimensiunile fișei de hârtie: 36*30 *Soluția acestei probleme este: 5* *Explicația este următoarea:* ▪ *3 pătrate cu dimensiunea (12*12);* ▪ *2 pătrate cu dimensiunea (18*18).* Fie avem dimensiunile fișei de hârtie: 13*29. Pentru a rezolva problema este important de a studia următoarea imagine ce indică soluția:  Așadar, numărul de pătrate ce poate fi tăiat este: 2+4+3=9.

Implementare C++

```
#include <iostream>
using namespace std;
const int MAX = 300;
int dp[MAX][MAX];
int PDinamnica(int m, int n){
     int vertical = INT_MAX, horizontal = INT_MAX;
     if (m == n) return 1;
     if (dp[m][n]) return dp[m][n];
     for (int i = 1;i<= m/2;i++){
          horizontal = min(PDinamnica(i, n)+PDinamnica(m-i,n),horizontal);
     }
     for (int j = 1;j<= n/2;j++){
          vertical = min(PDinamnica(m,j)+PDinamnica(m,n-j),vertical);
     }
     dp[m][n] = min(vertical, horizontal);
     return dp[m][n];
}
int main(){
     int m,n;
     cout<< "Introdu lungime foii de hartie: \t"; cin>>m;
     cout<< "Introdu latimea foii de hartie: \t"; cin>>n;
     cout<< "Numarul minim de patrate taiate: \t";
     cout << PDinamnica(m, n); return 0;
}
```

Rezultatele execuției:

```
Introdu lungime foii de hartie:         13
Introdu latimea foii de hartie:         29
Numarul minim de patrate taiate:        9
```

SARCINI PENTRU EXERSARE

Problema 1	***Date de intrare***	***Date de ieșire***
Există bile cu următoarele notații: „p” de tip P, bile „q” de tip Q și bile „r” de tip R. Folosind bilele dorim să creăm o linie dreaptă, astfel încât să nu fie adiacente două bile de același tip.	`1 1 0`	`2`
	`1 1 1`	`6`
	`2 2 1`	`12`

Problema 2	***Date de intrare***	***Date de ieșire***
Fie că avem un șir „str” de cifre, găsiți lungimea celei mai lungi secvențe din „str”, astfel încât lungimea secvenței este de 2k cifre și suma cifrelor k stânga este egală cu suma cifrelor k dreapta.	`123123`	`6` `123123`
	`25340851`	`8` `25340851`

Problema 3	***Date de intrare***	***Date de ieșire***
Fie că avem un tablou, trebuie să modificăm valorile acestui tablou astfel încât să se maximizeze suma diferențelor absolute între două elemente consecutive. Dacă valoarea unui element din tablou este X, atunci îl putem schimba în 1 sau în X.	`3 2 1 4`	`5` `|3-1|+|1-4|`
	`1 6 5 2`	`11` `|1-6|+|8-2|`

Problema 4	***Date de intrare***	***Date de ieșire***
Fie că avem un șir de cifre „0-9”. Sarcina este de a găsi numărul de secvențe care sunt divizibile cu 8, dar nu cu 3. Să se afișeze secvențele respective sub formă de indecși.	`168`	`2` `(1,2)` `(1,2,3)`

Problema 5	***Date de intrare***	***Date de ieșire***
Fie că avem un număr întreg n, găsiți numărul total de n cifre Stepping. Un număr Stepping se numește număr de treaptă, dacă toate cifrele adiacente au o diferență absolută de 1, astfel numărul 21 este un număr de treaptă, în timp ce 31 nu este.	`2`	`17` `10 12 21 23` `32 34 43 45` `54 56 65 67` `76 78 87 89` `98`

Problema 6	***Date de intrare***	***Date de ieșire***
Fie că avem o matrice a[][] de numere întregi, găsiți suma matricială a prefixului pentru aceasta. Fie matricea sumei prefixului este spa [] []. Valoarea spa [i] [j] conține suma tuturor valorilor care sunt deasupra acesteia sau la stânga acesteia. Formula generală este: *spa[i][j]=spa[i-1][j]+ spa[i][j-1]- spa[i-1][j-1]+a[i][j]*	`4 5` `1 1 1 1 1` `1 1 1 1 1` `1 1 1 1 1` `1 1 1 1 1`	`4 5` `1 2 3 4 5` `2 4 6 8 10` `3 6 9 12 15` `4 8 12 16 20`

Problema 7	***Date de intrare***	***Date de ieșire***
Vom numi un număr crescător, dacă fiecare cifră (cu excepția primei) este mai mare sau egală cu cifra anterioară. Deci, având în vedere numărul de cifre n, vi se cere să găsiți numărul total de numere crescătoare cu n cifre.	`1`	`10`
	`2`	`55`
	`3`	`220`
	`4`	`715`

Problema 8	***Date de intrare***	***Date de ieșire***
Fie că avem un tablou unidimensional de N elemente întregi, sarcina este de a găsi suma mediei tuturor submulțimilor din acest tablou. Fie vectorul {2, 3, 5}, suma mediei tuturor submulțimilor din acest tablou este:2+3+5+2,5+3,5+4+3,33= 23,33.	`2 3 5`	`23.33`
	`2 3 4 5`	`52.50`
	`2 5 4 7 6`	`148.80`
	`3 5 7 9 0 2`	`273.00`

Problema 9	***Date de intrare***	***Date de ieșire***
Fie că avem un tablou unidimensional, o inversare este definită ca o pereche a [i], a [j] astfel încât a [i]> a [j] și i <j. Se dau două numere N și K, trebuie să spunem câte permutări ale primului număr N au exact inversarea K. Fie N=3 și K=1, atunci petrmutările vor fi: 123, 132, 213, 231, 321 și 321, permutările cu K=1 inversări sunt:132 și 213	`3 1`	`2`
	`4 2`	`5`
	`4 3`	`6`
	`5 2`	`9`
	`5 3`	`15`

Problema 10	***Date de intrare***	***Date de ieșire***
Fie că avem un tablou unidimensional de N elemente întregi, folosind operațiile „+” și „-” între elemente, verificată dacă există o modalitate de a forma o secvență de numere care se evaluează la un număr divizibil cu M. Fie vectorul {1, 2, 3, 4, 6} și M=4, în acest caz există posibilitatea de a găsi un număr divizibil cu M, secvența este: 1-2+3+4+6=12, iar 12:4=3.	`1 2 3 4 6`	`Adevarat`
	`2 4 6 3 5`	`Adevarat`
	`1 4 6 7 9`	`Fals`
	`3 4 8`	`Fals`
	`7 4 8 5 2 6`	`Adevarat`

Problema 11	***Date de intrare***	***Date de ieșire***
Fie că avem o matrice de caractere. Găsiți lungimea celei mai lungi căi de la un anumit caracter, astfel încât toate caracterele din traseu să fie consecutive, adică fiecare caracter din traseu este alături de cele anterioare în ordine alfabetică. Este permis să se deplaseze în toate cele 8 direcții dintr-o celulă.	`3 3` `A C D` `H B E` `I G F`	`E 5` `E F G H I`
		`B 8` `B C D E F` `G H I`

Problema 12	***Date de intrare***	***Date de ieșire***
Fie că avem timpul pentru n sarcini. Găsiți timpul necesar pentru a termina sarcinile, astfel încât să fie omise unele sarcini, dar nu puteți sări peste două sarcini consecutive.	`8 5 2 7 6`	`12`
	`6 5 7 4`	`9`
	`9 6 2 4 7`	`10`

Problema 13	***Date de intrare***	***Date de ieșire***
*Un număr poate fi întotdeauna reprezentat ca o sumă a pătratelor altor numere. Rețineți că 1 este un pătrat și putem întotdeauna scrie un număr ca (1 * 1 + 1 * 1 + 1 * 1 + ...). Având un număr n, găsiți numărul minim de pătrate care însumează valoarea n.*	`6`	`3`
	`15`	`4`
	`50`	`2`
	`100`	`1`

Problema 14	***Date de intrare***	***Date de ieșire***
Fie că avem o matrice care conține numere întregi, în care fiecare celulă a matricei reprezintă înălțimea unei clădiri. Găsiți salturile minime necesare de la prima clădire (0, 0) până la ultima (n-1, m-1). Saltul de la o celulă la celula următoare este o diferență absolută între două înălțimi ale clădirii.	`4 3` `5 4 2` `9 2 1` `2 5 9` `1 3 11`	`12` `11 salturi` `5 2 5 11` `3+3+6=11`

8.5 Implementarea metodei de programare dinamică la rezolvarea problemelor avansate

Problema 1: Distanța Levenshtein între două șiruri

Condiția problemei	*Exemplu*
Distanța Levenshtein între două șiruri înseamnă numărul minim de modificări necesare pentru a transforma un șir în altul, cu operațiile de editare adică; inserarea, ștergerea sau substituirea unui singur caracter. Determinați distanța Levenshtein și afișați rezultatul.	• Fie avem S1= "casa" și S2= "masa" Soluția acestei probleme este: 1 Explicația este următoarea: pentru a obține în S1 valoarea lui S2, este necesar de a efectua doar o singură operație, de substituire a unui caracter, deoarece aceste două șiruri se diferențiază printr-un singur caracter. • Fie avem S1= "Informatica" și S2= "matematica" Soluția acestei probleme este: 5 Explicația este următoarea: pentru a obține în S1 valoarea lui S2, este necesar de a efectua 5 operații, deoarece aceste două șiruri se diferențiază prin 5 caractere.

Implementare C++

```
#include <iostream>
#include <string.h>
using namespace std;
int levenshtein(const char *s, int ls, const char *t, int lt){
     int a, b, c;
     if (!ls) return lt;
     if (!lt) return ls;
     if (s[ls - 1] == t[lt - 1])
     return levenshtein(s, ls - 1, t, lt - 1);
     a = levenshtein(s, ls - 1, t, lt - 1);
     b = levenshtein(s, ls,     t, lt - 1);
     c = levenshtein(s, ls - 1, t, lt    );
     if (a > b) a = b;
     if (a > c) a = c;
     return a + 1;
}
int main(){
     cout<<"Cazul 1\n";
     const char *s1 = "informatica";
     const char *s2 = "matematica";
     int x=strlen(s1), y=strlen(s2);
     cout<<"\tDistanta intre sirul \""<<s1<<"\" si \""<<s2<<"\" este: ";
     cout<<levenshtein(s1, x, s2, y);
     cout<<"\nCazul 2\n";
     const char *k1 = "floare";
     const char *k2 = "soare";
     int x1=strlen(k1), y1=strlen(k2);
     cout<<"\tDistanta intre sirul \""<<k1<<"\" si \""<<k2<<"\" este: ";
     cout<<levenshtein(k1, x1, k2, y1);
     return 0;
}
```

Rezultatele execuției:

```
Cazul 1
        Distanta intre sirul "informatica" si "matematica" este: 5
Cazul 2
        Distanta intre sirul "floare" si "soare" este: 2
```

Problema 2: Tastatura numerică mobilă

Condiția problemei	*Exemplu*
*Fie că avem o tastatura numerică mobilă (a unui telefon). Puteți apăsa doar butoanele care sunt în sus, la stânga, la dreapta sau în jos până la butonul curent. Nu aveți voie să apăsați butoanele din colțul de jos al rândului (adică * și #). Dat fiind un număr N, aflați numărul de numere posibile de lungime dată.*	Fie avem N= 2 Soluția acestei probleme este: 36 Explicația este următoarea: numere posibile: 00, 08, 11, 12, 14, 22, 21, 23, 25 și așa mai departe. • Dacă începem cu 0, numerele valide: 00, 08; • Dacă începem cu 1, numerele valide: 11, 12, 14; • Dacă începem cu 2, numerele valide: 22, 21, 23, 25; • Dacă începem cu 3, numerele valide: 33, 32, 36; • Dacă începem cu 4, numerele valide: 44, 41, 45, 47; Etc.

Implementare C++

```
#include <iostream>
using namespace std;
int PDinamica(char taste[][3], int n){
     if(taste == NULL || n <= 0) return 0;
     if(n == 1) return 10;
     int r[] = {0, 0, -1, 0, 1}, c[] = {0, -1, 0, 1, 0}; //rand si coloana
     int calcul[10][n+1];
     int i=0, j=0, k=0, miscare=0, ro=0, co=0, num = 0, nextNum=0, totalCalcul = 0;
     for (i=0; i<=9; i++){
          calcul[i][0] = 0; calcul[i][1] = 1;
     }
     for (k=2; k<=n; k++){
          for (i=0; i<4; i++){
               for (j=0; j<3; j++){
                    // Process for 0 to 9 digits
                    if (taste[i][j] != '*' && taste[i][j] != '#'){
                         num = taste[i][j] - '0'; calcul[num][k] = 0;
                         for (miscare=0; miscare<5; miscare++){
                              ro = i + r[miscare]; co = j + c[miscare];
                              if (ro>=0 && ro<=3 && co>=0 && co<=2 && taste[ro][co]!='*'
&& taste[ro][co]!='#'){
                                   nextNum = taste[ro][co] - '0'; calcul[num][k] +=
calcul[nextNum][k-1];
                              }
                         }
                    }
               }
          }
     }
     totalCalcul = 0;
     for (i=0; i<=9; i++)
     totalCalcul += calcul[i][n];
     return totalCalcul;
}
int main(int argc, char *argv[]){
     char taste[4][3] = {{'1','2','3'},
                         {'4','5','6'},
                         {'7','8','9'},
                         {'*','0','#'}};
     for (int i=1; i<=3; i++){
          cout<<"\nNumaram numerele de lungime "<<i<<" este: \t"<<PDinamica(taste, i);
     }
     return 0;
}
```

Rezultatele execuției:

```
Numaram numerele de lungime 1 este:     10
Numaram numerele de lungime 2 este:     36
```

Problema 3: Numărul de subsecvențe dintr-un șir divizibil cu m

Condiția problemei	*Exemplu*
Fiind dat un șir format din cifre cuprinse între 0-9. Determinați numărul de subsecvențe divizibile cu m.	Fie avem s1= “1234” și m=4 Soluția acestei probleme este: 4 Explicația este următoarea: subsecvențele 4, 12, 24 și 124 sunt divizibile cu 4.

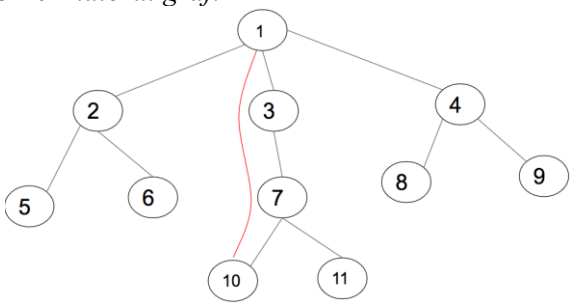
Implementare C++

```
#include <iostream>
#include <string.h>
using namespace std;

int PDonamica(string str, int n){
     int len = str.length(), dp[len][n];
     memset(dp, 0, sizeof(dp));
     dp[0][(str[0]-'0')%n]++;
     for (int i=1; i<len; i++){
          dp[i][(str[i]-'0')%n]++;
          for (int j=0; j<n; j++){
      // exclude al i-lea caracter din toate subsecventele actuale ale sirului [0...i-1]
               dp[i][j] += dp[i-1][j];
      // include al i-lea caracter in toate subsecventele actuale ale sirului [0...i-1]
               dp[i][(j*10 + (str[i]-'0'))%n] += dp[i-1][j];
          }
     }
     return dp[len-1][0];
}

int main(){
     string str1 = "1234", str2 = "123456";
     cout<<"Cazul 1";
     for (int i=1; i<=5; i++){
          cout <<"\n\tPentru sirul \""<<str1<<"\" si numarul "<<i;
          cout <<" solutia problemei este: "<< PDonamica(str1, i);
     }
     cout<<"\nCazul 2";
     for (int i=1; i<=5; i++){
          cout <<"\n\tPentru sirul \""<<str2<<"\" si numarul "<<i;
          cout <<" solutia problemei este: "<< PDonamica(str2, i);
     }
     return 0;
}
```

Rezultatele execuției:

```
Cazul 1
        Pentru sirul "1234" si numarul 1 solutia problemei este: 15
        Pentru sirul "1234" si numarul 2 solutia problemei este: 10
        Pentru sirul "1234" si numarul 3 solutia problemei este: 5
        Pentru sirul "1234" si numarul 4 solutia problemei este: 4
        Pentru sirul "1234" si numarul 5 solutia problemei este: 0
Cazul 2
        Pentru sirul "123456" si numarul 1 solutia problemei este: 63
        Pentru sirul "123456" si numarul 2 solutia problemei este: 42
        Pentru sirul "123456" si numarul 3 solutia problemei este: 23
        Pentru sirul "123456" si numarul 4 solutia problemei este: 25
        Pentru sirul "123456" si numarul 5 solutia problemei este: 16
```

Problema 4: Înălțimea maximă într-un graf în care orice nod poate fi rădăcină

Condiția problemei	*Exemplu*
Având un graf cu N noduri și N-1margini, aflați înălțimea maximă a grafului atunci când orice nod din graf este considerat rădăcina grafului.	*Fie avem următorul graf:*  *Linia roșie denotă înălțimea maximă a grafului unde nodul 1 este rădăcină. Problema respectivă are mai multe noduri ca soluții.*

Implementare C++

```
#include <bits/stdc++.h>
using namespace std;
const int MAX_NOD = 100;
int in[MAX_NOD], out[MAX_NOD];
///stocam inaltimea maxima atunci cand este parcurs prin ramuri
void dfs1(vector<int> v[], int u, int parinte){
     in[u] = 0;
     for (int copil : v[u]) {
          if (copil == parinte) continue;
          dfs1(v, copil, u); in[u] = max(in[u], 1 + in[copil]);
    }
}
///stocam inaltimea maxima atunci cand este parcurs prin parinte
void dfs2(vector<int> v[], int u, int parinte){
     int mx1 = -1, mx2 = -1;
     for (int copil : v[u]){
          if (copil == parinte) continue;
          if (in[copil] >= mx1){
               mx2 = mx1; mx1 = in[copil];
          }
          else if (in[copil] > mx2) mx2 = in[copil];
     }
     for (int copil : v[u]){
          if (copil == parinte) continue;
          int longest = mx1;
          if (mx1 == in[copil]) longest = mx2;
          out[copil] = 1 + max(out[u], 1 + longest); dfs2(v, copil, u);
     }
}
void PDinamica(vector<int> v[], int n){
     dfs1(v, 1, 0); dfs2(v, 1, 0);
     for (int i = 1; i <= n; i++)
     cout << "Inaltimea maxima unde nodul " << i << " este considerat radacina este: " <<
max(in[i], out[i]) << "\n";
}
int main(){
     int n = 11; vector<int> v[n + 1];
     // initialize the tree given in the diagram
     v[1].push_back(2), v[2].push_back(1); v[1].push_back(3), v[3].push_back(1);
     v[1].push_back(4), v[4].push_back(1); v[2].push_back(5), v[5].push_back(2);
     v[2].push_back(6), v[6].push_back(2); v[3].push_back(7), v[7].push_back(3);
     v[7].push_back(10), v[10].push_back(7); v[7].push_back(11), v[11].push_back(7);
     v[4].push_back(8), v[8].push_back(4); v[4].push_back(9), v[9].push_back(4);
     PDinamica(v, n);
     return 0;
}
```

Problema 5: Triunghiul lui Hosoya

Condiția problemei	*Exemplu*
Triunghiul Fibonnaci sau triunghiul lui Hosoya este un aranjament triunghiular de numere bazat pe numere Fibonacci. Fiecare număr este suma a două numere de mai sus, fie în diagonala stângă, fie în diagonala dreaptă. Dat fiind un număr întreg pozitiv n. Sarcina este tipărirea triunghiului lui Hosoya cu dimensiunea n.	Fie avem n=6 ***Soluția acestei probleme este***: 1 1 1 2 1 2 3 2 2 3 5 3 4 3 5 8 5 6 6 5 8

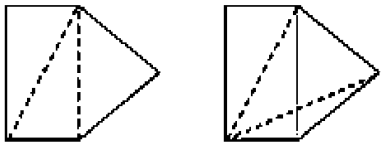
Implementare C++

```
#include <iostream>
#include <string.h>
#define N 10
using namespace std;
void PDinamica(int n){
     int dp[N][N];
     memset(dp, 0, sizeof(dp));
     // cazul de baza
     dp[0][0] = dp[1][0] = dp[1][1] = 1;
     // pentru fiecare rand
     for (int i = 2; i < n; i++){
     // pentru fiecare coloana
          for (int j = 0; j < n; j++){
          // pasii recursivi
               if (i > j) dp[i][j] = dp[i - 1][j] + dp[i - 2][j];
               else dp[i][j] = dp[i - 1][j - 1] + dp[i - 2][j - 2];
          }
     }
     // afisarea solutiei
     for (int i = 0; i < n; i++){
          for (int j = 0; j <= i; j++)
          cout <<"\t"<< dp[i][j] << " "; cout << endl;
     }
}
int main(){
    int n = 10;
    cout<<"Pentru numarul n="<<n<<", solutia problemei este: \n";
    PDinamica(n);
    return 0;
}
```

Rezultatele execuției:

```
Pentru numarul n=10, solutia problemei este:
        1
        1       1
        2       1       2
        3       2       2       3
        5       3       4       3       5
        8       5       6       6       5       8
        13      8       10      9       10      8       13
        21      13      16      15      15      16      13      21
        34      21      26      24      25      24      26      21      34
        55      34      42      39      40      40      39      42      34      55
```

Problema 6: Triangularea poligonului de cost minim

Condiția problemei	*Exemplu*
O triangulație a unui poligon convex este formată prin trasarea diagonalelor între vârfurile care nu sunt adiacente, astfel încât diagonalele să nu se intersecteze niciodată. Problema este de a găsi costul triangulării de cost minim. Costul unei triangulări este suma ponderilor triunghiurilor sale componente. Greutatea fiecărui triunghi este perimetrul său (suma lungimilor tuturor părților).	Fie avem două figuri convexe, cu coordonatele: (0,0); (0,2); (1,2); (2,1) și (1,0);  ***Soluția acestei probleme este:*** *Triangularea poligonului din stânga are un cost de 8 + 2√2 + 2√5 (aproximativ 15.30), iar pentru cel din dreapta are un cost de 4 + 2√2 + 4√5 (aproximativ 15.77).*

Implementare C++

```
#include <iostream>
#include <cmath>
#define MAX 1000000.0
using namespace std;
struct Punct{
     int x, y;
};
double min(double x, double y){
     return (x <= y)? x : y;
}
double dist(Punct p1, Punct p2){
     return sqrt((p1.x - p2.x)*(p1.x - p2.x) + (p1.y - p2.y)*(p1.y - p2.y));
}
double cost(Punct points[], int i, int j, int k){
     Punct p1 = points[i], p2 = points[j], p3 = points[k];
     return dist(p1, p2) + dist(p2, p3) + dist(p3, p1);
}
// functia pentru a gasi costul minim pentru triangularea poligonului convex.
double PDinamica(Punct points[], int n){
     if (n < 3) return 0;
     double table[n][n];
     for (int gap = 0; gap < n; gap++){
          for (int i = 0, j = gap; j < n; i++, j++){
               if (j < i+2) table[i][j] = 0.0;
               else {
                    table[i][j] = MAX;
                    for (int k = i+1; k < j; k++){
                         double val = table[i][k] + table[k][j] + cost(points,i,j,k);
                         if (table[i][j] > val) table[i][j] = val;
                    }
               }
          }
     }
     return table[0][n-1];
}
int main() {
     Punct points[] = {{0, 0}, {0, 2}, {1, 2}, {2, 1}, {1, 0}};
     int n = sizeof(points)/sizeof(points[0]);
     cout << "Triangularea poligonului de cost minim cu "<<n<<" laturi este:\t";
     cout << PDinamica(points, n); return 0;
}
```

Rezultatele execuției:

```
Triangularea poligonului de cost minim cu 5 laturi este:     15.3006
```

Problema 7: Dimensiunea celei mai lungi progresii aritmetice

Condiția problemei	*Exemplu*
*Având în vedere un set de numere, găsiți în ea lungimea progresiei aritmetice cele mai lungi (LLAP = **L**ength of the **L**ongest **A**rithmetic **P**rogression).*	Fie avem mulțimea A={1, 7, 10, 15, 27, 29} ***Soluția acestei probleme este:*** 3 *Dimensiunea celei mai lungi progresii aritmetică este3. Această mulțime este alcătuită din elementele: {1, 15, 29}.*

Implementare C++

```
#include <iostream>
using namespace std;

int DProgramming(int set[], int n){
    if (n <= 2)  return n;
    int L[n][n];
    int i0 = 2;
    for (int i = 0; i < n; i++)
        L[i][n-1] = 2;
    for (int j=n-2; j>=1; j--){
        int i = j-1, k = j+1;
        while (i >= 0 && k <= n-1){
           if (set[i] + set[k] < 2*set[j]) k++;
           else if (set[i] + set[k] > 2*set[j]){
                    L[i][j] = 2, i--;
                }
                else{
                    L[i][j] = L[j][k] + 1;
                    i0 = max(i0, L[i][j]);
                    i--; k++;
                }
        }
        while (i >= 0){
            L[i][j] = 2; i--;
        }
    }
    return i0;
}
int main(){
    int a[] = {1, 7, 10, 13, 14, 19};
    int n1 = sizeof(a)/sizeof(a[0]);
    cout<<"Elementele vectorului introdus sunt:\t";
    for (int i=0; i<n1;i++)
          cout<<a[i]<<" "; cout<<endl;
    cout<<"Dimensiunea celei mai lungi progresii aritmetice este: \t";
    cout<<DProgramming(a, n1)<<endl;
    return 0;
}
```

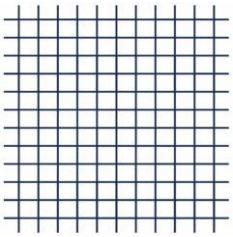
Rezultatele execuției:

```
Elementele vectorului introdus sunt:    1 7 10 13 14 19
Dimensiunea celei mai lungi progresii aritmetice este:  4
```

Notă:

- ✓ *Complexitatea temporală: O (n^2)*
- ✓ *Complexitatea spațială: O (n^2)*
- ✓ *Cum se reduce complexitatea spațială pentru soluția de mai sus?*
- ✓ *De asemenea, putem reduce complexitatea spațiului la O (n).*

Problema 8: Experiment biologic

Condiția problemei	*Exemplu*
Fie că există o insulă sub formă de matrice pătrată și un punct din interiorul matricei în care se află o persoană în picioare. Persoanei i se permite să se deplaseze cu un pas în orice direcție (dreapta, stânga, sus, jos) pe matrice. Dacă iese în afara matricei, el moare din cauza unor animale asupra cărora a fost efectuat un experiment biologic secret. Persoana care se află pe insulă nu trebuie să se afle în apropierea mediului acvatic. Calculați probabilitatea că savantul este în viață după ce merge pe pași pe insulă.	Fie avem următoarele date: M[10][10], (x_0, y_0) = (5,5) și numărul de pași este egal cu 4:  ***Soluția acestei probleme este:*** *100%* *Indiferent de direcție, savantul va fi protejat, excepție ar fi dacă savantul s-ar afla cu o celulă mai departe de centrul insulei, astfel probabilitatea ar începe să scadă.*

Implementare C++

```
#include <iostream>
#include <map>
#include <string>
using namespace std;
#define N 10
float PDinamica(int x, int y, int n, map<string, float> &dp){
	// Cazul de baza
	if (n == 0) return 1.0;
	 // Cazul special
	string s = to_string(x) + "|" + to_string(y) + "|" + to_string(n);
	if (dp.find(s) == dp.end()){
		float p = 0.0;
		// Mutam un pas in sus
		if (x > 0) p += 0.25 * PDinamica(x - 1, y, n - 1, dp);
		// Mutam un pas in jos
		if (x < N - 1) p += 0.25 * PDinamica(x + 1, y, n - 1, dp);
		// Mutam un pas la stinga
		if (y > 0) p += 0.25 * PDinamica(x, y - 1, n - 1, dp);
		// Mutam un pas la dreapta
		if (y < N - 1) p += 0.25 * PDinamica(x, y + 1, n - 1, dp);
		dp[s] = p;
	}
	return dp[s];
}
int main(){
	int n = 4;		// Numarul de pasi de realizat de catre savant
	int x = 5, y = 5; // Coordonatele de start
	// Harta pentru a stoca solutia la sub-problemele deja calculate
	map<string, float> dp;
	cout << "Coordonatele de start ale savantului: \t\t "<<"("<<x<<","<< y<<")"<<endl;
	cout << "Dimensiunea insulei savantului: \t\t "<<"("<<N<<","<< N<<")"<<endl;
	cout << "Numarul de pasi de realizat a savantului: \t "<<n<<endl;
	cout << "Probabilitatea savantului de a ramane in viata:  ";
	 cout << PDinamica(x, y, n, dp)*100<<" %"<< endl;
}
```

Rezultatele execuției:

```
Coordonatele de start ale savantului:            (5,5)
Dimensiunea insulei savantului:                  (10,10)
Numarul de pasi de realizat a savantului:        4
Probabilitatea savantului de a ramane in viata:  100 %
```

Problema 9: Ecuație liniară de k variabile

Condiția problemei	*Exemplu*
Fiind o ecuație liniară a k variabile, numărați numărul total de soluții posibile ale acesteia.	Fie avem 4 variabile cu coeficienții: (1, 3, 5, 7) și soluția 8. Deci ecuația este: a+3b+5c+7d=8. ***Soluția acestei probleme este:*** *6 soluții* `( a = 1, b = 0, c = 0, d = 1 )` `( a = 0, b = 1, c = 1, d = 0 )` `( a = 2, b = 2, c = 0, d = 0 )` `( a = 3, b = 0, c = 1, d = 0 )` `( a = 5, b = 1, c = 0, d = 0 )` `( a = 8, b = 0, c = 0, d = 0 )`

Implementare C++

```
#include <iostream>
#include <unordered_map>
using namespace std;

int count(int coeficienti[], int k, int rezultat, auto &respect){
       // Dacă rezultat devine 0, returnez 1 (solutia gasita)
       if (rezultat == 0) return 1;
       // Returnez 0 (solutia nu exista), dacă rezultat este negativ sau nu exista
       if (rezultat < 0 || k < 0) return 0;
       string key = to_string(k) + "|" + to_string(rezultat);
       if (respect.find(key) == respect.end()){
        // cazul 1
              int include = count(coeficienti, k, rezultat - coeficienti[k], respect);
        // cazul 2
        int exclude = count(coeficienti, k - 1, rezultat, respect);
        respect[key] = include + exclude;
       }
       // Returnez solutia la sub-problema actuala
       return respect[key];
}
int main(){
       // Coeficientii ecuatiei
       int coef[] = { 1, 3, 5, 7 }, rezultat=8;
       int k = sizeof(coef) / sizeof(coef[0]);
       unordered_map<string, int> respect;
       cout << "Numarul coeficintilor ecuatiei este: \t"<<k<<endl;
       cout << "Coeficintii ecuatiei sunt: \t\t";
       for (int i=0; i<k;i++)
          cout<<coef[i]<<" "; cout<<endl;
       cout << "Numarul total de solutii ale problemei: ";
       cout << count(coef, k - 1, rezultat, respect);
       return 0;
}
```

Rezultatele execuției:

```
Numarul coeficintilor ecuatiei este:    4
Coeficintii ecuatiei sunt:              1 3 5 7
Numarul total de solutii ale problemei: 6
```

Problema 10: Problema submulțimilor de sumă K

Condiția problemei	*Exemplu*
Fie că avem o mulțime S de numere întregi pozitive, determinați dacă poate fi partiționat în trei submulțimi disjuncte, care toate au aceeași sumă și acoperă o valoare K – număr întreg pozitiv.	Fie avem mulțimea S = {7, 3, 2, 1, 5, 4, 8}. Ne propunem să determinăm dacă putem diviza S în trei partiții, fiecare având suma egală cu 10. ***Soluția acestei probleme este:*** *Da, se poate de partiționat!* *S = {7, 3}* *S = {5, 4, 1}* *S = {8, 2}*

Implementare C++

```
#include <iostream>
#include <numeric>
#include <unordered_map>
using namespace std;
bool subsetSum(int S[], int n, int a, int b, int c, auto &respecta){
	// returnez adevarat daca se gaseste submultimea
	if (a == 0 && b == 0 && c == 0)return true;
	// cazul de baza
	if (n < 0) return false;
	string key = to_string(a) + "|" + to_string(b) + "|" + to_string(c) + "|" +
to_string(n);
	if (respecta.find(key) == respecta.end()){
		// Cazul 1. Elementul curent devine parte a primei submultimi
		bool A = false;
		if (a - S[n] >= 0) A = subsetSum(S, n - 1, a - S[n], b, c, respecta);
		// Cazul 2. Elementul curent devine parte a submultimii a doua
		bool B = false;
		if (!A && (b - S[n] >= 0)) B = subsetSum(S, n - 1, a, b - S[n], c,
respecta);
		// Cazul 3. Elementul curent devine parte a submultimii a treia
		bool C = false;
		if ((!A && !B) && (c - S[n] >= 0)) C = subsetSum(S, n - 1, a, b, c - S[n],
respecta);
		// returnez adevarat daca obtin solutia
		respecta[key] = A || B || C;
	}
    // returnez solutia subproblemei din harta solutiilor
	return respecta[key];
}
bool PD(int S[], int n){
	if (n < 3) return false;
	// cream o harta de stocare a solutiilor subproblemelor
	unordered_map<string, bool> respecta;
	// Obtinem suma tuturor elementelor din submultimi
	int sum = accumulate(S, S + n, 0);
	//returnez adevarat daca suma este divizibila cu 3
	//returnez adevarat dacă mulțimea S poate fi impartita in 3 submultimi cu suma
egala
	return !(sum % 3) && subsetSum(S, n - 1, sum/3, sum/3, sum/3, respecta);
}
int main(){
	// Datele de intrare ale problemei
	int S[] = { 7, 3, 2, 1, 5, 4, 8 }, n = sizeof(S) / sizeof(S[0]);
	if (PD(S, n)) cout << "Da, se poate de partitionat!";
	else cout << "Nu se poate de partitionat!";
}
```

Rezultatele execuției:

```
Da, se poate de partitionat!
```

SARCINI PENTRU EXERSARE

Problema 1	***input***	***output***
Fiind dat un șir de intrări și un model (p), implementați o potrivire regulată a expresiilor cu suport pentru '.' și ''. Unde '.' se potrivește cu orice personaj unic, iar '*' se potrivește cu zero sau mai mult din elementul precedent. Potrivirea ar trebui să acopere întregul șir de intrare (nu parțial).* ***Notă:*** • *s ar putea fi gol și conține doar litere minuscule a-z.* • *p ar putea fi gol și conține doar litere minuscule a-z și caractere ca '.' sau '*'.*	`s=”aa”` `p=”a”`	`Fals, deoarece „a” nu se potrivește cu întregul șir „aa”.`
	`s ="aa"` `p ="a*"`	`Adevarat, deoarece „*” înseamnă zero sau mai mult din elementul precedent, „a”. Prin urmare, repetând „a” o dată, devine „aa”.`

Problema 2	***input***	***output***
Fie un tablou de date pentru care elementul al i-lea este prețul unui stoc dat în ziua i. Proiectați un algoritm pentru a găsi profitul maxim. Puteți finaliza cel mult k tranzacții. ***Notă:*** • *Este posibil să nu angajați mai multe tranzacții în același timp (adică, trebuie să vindeți stocul înainte de a cumpăra din nou).*	`[2,4,1]` `k = 2`	`2, deoarece cumpărați în ziua 1 (preț = 2) și vindeți în ziua 2 (preț = 4), profitul este 4-2 = 2.`
	`[3,2,6,5 ,0,3] k = 2`	`7, deoarece cumpărați în ziua 2 (preț = 2) și vindeți în ziua 3 (preț = 6), profit = 6-2 = 4. Apoi cumpărați în ziua 5 (preț = 0) și vindeți în ziua 6 (preț = 3), profitați = 3-0 = 3.`

Problema 3	***input***	***output***
Dragonul o capturaseră pe prințesă (P) și o încarcerase în colțul din dreapta jos al unei temnițe. Temnița este formată din camere M x N, dispuse într-un tablou bidimensional. Viteazul nostru cavaler (K) a fost poziționat inițial în camera din stânga sus și trebuie să străbată drum prin temniță cea mare pentru a salva prințesa iubită. Cavalerul are un punct de sănătate inițial reprezentat de un număr întreg pozitiv. Dacă în orice moment punctul său de sănătate scade la 0 sau mai jos, el moare imediat. Unele dintre camere sunt păzite de soldații dragonului, astfel încât cavalerul își pierde sănătatea (numere întregi negative) la intrarea în aceste camere; alte camere sunt goale (0) sau conțin orbe magice care cresc sănătatea cavalerului (numere întregi pozitive). Pentru a ajunge cât mai repede la prințesă, cavalerul decide să se deplaseze doar spre dreapta sau în jos la fiecare pas.	`P=7` `-2 -3 3` `-5 -10 1` `10 30 -5`	`Printesa este salvată!` `Drumul cavalerului:` `D – dreapta;` `D – dreapta;` `J – jos;` `J – jos.`
	`P=6` `-2 -3 3` `-5 -10 1` `10 30 -5`	`Printesa nu poate fi salvată, deoarece cavalerul ajunge 0 puncte de viață la prințesă.`
	`P=5` `-2 -3 3` `-5 -10 1` `10 30 -5`	`Printesa nu poate fi salvată, deoarece cavalerul își pierde toate punctele de viață la ieșirea din camera în care se află la început, nu are nici o șansă să ajungă la prințesă.`

Problema 4	***input***	***Output***
*Mașina dvs. pornește de la poziția 0 și viteza +1 pe o linie de număr infinit. (Mașina dvs. poate merge în poziții negative.) Mașina dvs. conduce automat în conformitate cu o secvență de instrucțiuni A (accelerați) și R (invers). Când primiți o instrucțiune „A”, mașina dvs. face următoarele: poziția + = viteza, viteza * = 2. Când primiți o instrucțiune „R”, mașina dvs. face următoarele: dacă viteza dvs. este pozitivă, atunci viteza = -1 , în caz contrar, viteza = 1. (Poziția dvs. rămâne aceeași.) De exemplu, după comenzile „AAR”, mașina dvs. merge pe pozițiile 0-> 1-> 3-> 3, iar viteza dvs. va trece la 1-> 2-> 4 -> - 1. Acum pentru unele poziții țintă, spuneți lungimea celei mai scurte secvențe de instrucțiuni pentru a ajunge acolo.*	`3`	`Rezultat egal cu 2` `Secvența este ”AA”` `Deoarece poziția dvs. merge de la 0-> 1-> 3.`
	`6`	`Rezultat egal cu 5` `Secvența este ”AAARA”` `Deoarece poziția dvs. merge de la 0-> 1-> 3-> 7-> 7-> 6.`

Problema 5	***input***	***Output***
Există N grămezi de pietre dispuse la rând. Mormanul i are pietre[i] pietre. O mișcare constă în contopirea exactă a K grămezi de pietre consecutive într-o singură grămadă, iar costul acestei mișcări este egal cu numărul total de pietre din aceste grămezi de pietre K. Găsiți costul minim pentru a îmbina toate grămezile de pietre într-o singură grămadă. Dacă este imposibil, afișați valoarea -1. ***Notă:*** • *1 <= i <= 30* • *2 <= K <= 30* • *1 <= pietre [i] <= 100*	`3 2 4 1` `k=2`	`20` `Avem: {3,2} și {5,4,1},` `deci obținem {5,5}=10,` `astfel avem 20`
	`3 5 2 1` `6` `k=3`	`25` `Avem: {5,1,2} și {3,8,6}, deci {3,8,6}=17, astfel avem 25`

Problema 6	***input***	***output***
Fiind dată o pizza dreptunghiulară reprezentată ca matrice cu R x C care conține următoarele caractere: „B” (brânză) și „.” (celulă goală) și fiind dat numărul întreg k. Trebuie să tăiați pizza în k bucăți folosind k-1 tăieturi. Pentru fiecare tăietură alegeți direcția: verticală sau orizontală, apoi alegeți o poziție de tăiere la limita celulei și tăiați pizza în două bucăți. Dacă tăiați pizza vertical, dați partea stângă a pizza unei persoane. Dacă tăiați pizza orizontal, dați partea superioară din pizză unei persoane. Dă ultima bucată de pizza ultimei persoane. Întoarceți numărul de moduri de a tăia pizza astfel încât fiecare bucată să conțină cel puțin o bucată de brânză. Deoarece răspunsul poate fi un număr uriaș, afișați acest rezultat ca modul $10^9 + 7$.	`{”B..”,` `”BBB”,` `”...”}` `k=3`	`Rezultat egal cu 3`
	`{”B..”,` `”BB.”,` `”...”}` `k=3`	`Rezultat egal cu 1`
	`{”B..”,` `”B..”,` `”...”}` `k=1`	`Rezultat egal cu 1`

MATERIALE DIDACTICE DIGITALE ELABORATE

calaméo

Nr.	Denumirea capitolului	Link la prezentare digitală
1	*Analiza algoritmilor*	*https://www.calameo.com/read/005335614d76a984a89fd*
2	*Metoda recursivă*	*https://www.calameo.com/read/00533561404f22b08b90e*
3	*Metoda trierii*	*https://www.calameo.com/read/0053356148a6555e0d03a*
4	*Metoda backtracking*	*https://www.calameo.com/read/0053356144b61c809ab33*
5	*Metoda divide et impera*	*https://www.calameo.com/read/005335614a0594aa97420*
6	*Tehnici de sortare*	*https://www.calameo.com/read/0053356146429l3092d98*
7	*Tehnica greedy*	*https://www.calameo.com/read/0053356147112ba317e29*
8	*Metoda programării dinamice*	*https://www.calameo.com/read/0053356140ef5a3149129*

Testmoz

Nr.	Denumirea capitolului	Link la test grilă
1	*Metoda recursivă*	*testmoz.com/2556773*
2	*Metoda trierii*	*testmoz.com/2556793*
3	*Metoda backtracking*	*testmoz.com/2556817*
4	*Metoda divide et impera*	*testmoz.com/2556823*
5	*Tehnica greedy*	*testmoz.com/2556827*
6	*Metoda programării dinamice*	*testmoz.com/2556833*

Google Sites

Nr.	Diverse materiale suport, conspecte, fișe de lucru la unitățile de curs
1	***Programarea structurată în limbajul C++*** *(lecții de teorie, lecții de laborator, sarcini individuale)*
2	***Programarea procedurală în limbajul C++*** *(lecții de teorie, lecții de laborator, sarcini individuale)*
3	***Programarea calculatorului în limbajul C++*** *(lecții de teorie, lecții de laborator, sarcini individuale)*
4	*Utilizarea tehnicilor clasice de programare în limbajul C++* ***(curs opțional)***

MULȚUMIRE

Acum când am ajuns la etapa finală de editare a acestei lucrări pe care mi-am propus-o încă din luna septembrie 2019, aș dori să aduc mulțumiri tuturor persoanelor care au contribuit cu bunătate, imaginație, încurajare și multă dedicare la realizarea acestei lucrări.

Aș dori să menționez unele persoane care au avut o contribuție deosebită în a mă ajuta să realizez această lucrare: Dl Liubomir Chiriac (dr. hab. prof. univ., UST), Dna Irina Pasecinic (profesoară la discipline de informatică, grad didactic I, IP CEITI), Dl Ioan Jeleascov (arhitector de soft la "StoneHard" din Marea Britanie, de asemenea activează în calitate de profesor la discipline de informatică, IP CEITI) și Dna Olga Cerbu (dr. conf. univ., USM)

Stimați colegi,

E o plăcere reală pentru mine să vă transmit cele mai sincere mulțumiri pentru disponibilitatea dumneavoastră. Doresc să ştiţi cât de mult apreciez remarcabilele Dumneavoastră. sfaturi pe care le-aţi acordat. Sunt mândru să am printre colegi oameni ca Dumneavoastră şi ţin mult la calităţile deosebite cu care v-aţi afirmat în preajma mea. Vreau să vă spun cât sunt de mândru să vă am în echipa mea. Aş vrea să vă urez mult succes în activitatea profesională şi fie ca Dumneavoastră şi familia Dumneavoastră să vă bucuraţi de mulţi ani de viaţă fără umbre şi regrete. Această zi este un prilej de mândrie pentru mine, dar şi pentru voi, cei care v-ați implicat cu multă dragoste și înțelegere. Consideraţiile mele faţă de profesionalismul Domniei Voastre nu pot fi estimate. Vă doresc ca discipolii Dumneavoastră să fie demni de marii pedagogi. Să vă însoţească mereu tot ce e mai bun pe lume: sănătatea, înţelepciunea, dragostea şi visele împlinite. Fie-vă alături mereu copiii, prietenii, norocul şi amintirile frumoase.

BIBLIOGRAFIE

1. *J. Erickson, "Algorithms", 2019, ISBN: 978-1-792-64483-2*
2. *G. Barnett, L. D. Tongo, "Data Structures and Algorithms", 2008*
3. *C. Petrovici, T. Stanciu, "Metode și tehnici de învățare", 2010*
4. *Găină V. G., "Învățarea centrată pe elevi și grupe de elevi", 2015, ISSN: 2393–0810*
5. *Noțiuni din teoria complexității, https://www.cs.cmu.edu/~mihaib/articole/complex/complex-html.html*
6. *Principiul de localitate a datelor, https://ro.wikipedia.org/wiki/Principiul_de_localitate_(informatică)*
7. *Eficiența algoritmilor, https://ro.wikipedia.org/wiki/Eficiența_algoritmilor*
8. *R. Andonie, I. Gârbecea, "Algoritmi fundamentali - o perspectivă C++", 1995*
9. *Notații asimptotice, http://andrei.clubcisco.ro/cursuri/2aa/notatii%20de%20complexitate.pdf*
10. *Complexitatea în timp a algoritmilor, https://en.wikipedia.org/wiki/Time_complexity*
11. *Timpul de execuție al unui program, http://staff.cs.upt.ro/~chirila/teaching/upt/cti21-sda/lab01/sda01.html*
12. *Reguli de determinare a timpului de execuție T(n) și O(f(n)), https://dokumen.tips/documents/1complexitatea-algoritmilor.html*
13. *Analiza eficienței unui algoritm, http://www.ududec.com/wp-content/uploads/2013/09/18.-Analiza-eficientei-unui-algoritm.pdf*
14. *В.И. Поляков, В.И. Скорубский, "Основы теории алгоритмов", 2012*
15. *D. P. Bovet, P. Crescenzi, "Introduction to the theory of complexity", 2006*
16. *В.Х. ХАХАНЯН, "ЭЛЕМЕНТЫ ТЕОРИИ СЛОЖНОСТИ АЛГОРИТМОВ И ВЫЧИСЛЕНИЙ", 2010*
17. *Лас-Вегас алгоритм - Las Vegas algorithm, https://ru.qwe.wiki/wiki/Las_Vegas_algorithm*
18. *Метод Монте-Карло, http://www.machinelearning.ru/wiki/index.php?title=Метод_Монте-Карло*
19. *A. Gremalschi, "Informatica. Tehnici de programare: Manual pentru clasa a Xl-a ", 2003, ISBN 9975-67-309-0*
20. *E. Cerchez, M. Șerban, "Metode și tehnici de programare", 2005, ISBN 973-46-0092-3*
21. *A. Gremalschi, "Informatica: Manual pentru clasa a Xl-a ", 2014, ISBN 978-9975-67-877-3*
22. *Wikipedia, https://ro.wikipedia.org/wiki/*
23. *Heinz-Otto Peitgen, H. Jürgens, D. Saupe, "Chaos and Fractals", 2003, ISBN 0-387-20229-3*
24. *A. Atanasiu, "Concursuri de informatică: Probleme propuse", 1994, ISBN 973-9116-17-05*
25. *D. Hrinciuc-Logofătu, "Probleme rezolvate și algoritmi", 2001, ISBN: 973-683-690-8*
26. *M. Miloșescu, "Informatica: Manual pentru clasa a Xl-a ", 2006, ISBN 973-30-1566-0*
27. *Колдаев В.Д., Павлова Е.Ю. Сборник задач и упражнений по информатике. - М.: ИД «ФОРУМ»: ИНФРА-М, 2010*
28. *Web: https://courses.cs.washington.edu/courses/cse143/11wi/lectures/02-16/18-recursive-backtracking-2.pdf*
29. *Web: https://www.hackerearth.com/ru/practice/basic-programming/recursion/recursion-and-backtracking/practice-problems/*
30. *Web: https://medium.com/@codingfreak/backtracking-practice-problems-and-interview-questions-6a17cb6d08a7*
31. *Web: https://algorithms.tutorialhorizon.com/top-10-problems-on-backtracking/*
32. *Web: https://www.geeksforgeeks.org/backtracking-algorithms/*
33. *Web: http://www.cs.cmu.edu/afs/cs/academic/class/15210-f14/www/lectures/dandc.pdf*
34. *Web: https://people.eecs.berkeley.edu/~vazirani/algorithms/chap2.pdf*
35. *S.Dhanalakshmi, "Analysis and Performance of Divide and Conquer Methodology", International Journal of Advanced Research in Computer Engineering & Technology (IJARCET). Volume 6, Issue 8, August 2017, ISSN: 2278 – 1323*
36. *С. Дасгупта, Х. Пападимитриу, У. Вазирани, "Алгоритмы", 2014, ISBN 978-5-4439-0236-4*

37. *Web: http://www.runceanu.ro/adrian/wp-content/cursuri/S1progr2012/curs18-PC.pdf*
38. *Web: https://merascu.github.io/links/WS2018/ASD1/alg2018_seminar9.pdf*
39. *Web: http://nataliadplesca.blogspot.com*
40. *Livovschi, L.,Georgescu, H., Sinteza şi Analiza algoritmilor, Editura Ştiinţifică şi Enciclopedică, Bucureşti, 1986.*
41. *Creţu, V., Structuri de date şi algoritmi, vol.1 – Structuri de date fundamentale, Editura Orizonturi Universitare, Timişoara, 2000.*
42. *Dogaru, O., Tehnici de programare, Editura MIRTON, Timişoara, 2004.*
43. *Web: https://kupdf.net/download/cartea-de-algoritmi-_59625d35dc0d60c8632be30b_pdf#*
44. *Web: https://dokumen.tips/documents/bazele-programarii-doina-logofatu.html*
45. *Web: http://jeffe.cs.illinois.edu/teaching/algorithms/notes/E-matroids.pdf*
46. *T.H. Cormen, C.E. Leiserson, R.L. Rivest: Introduction to Algorithms, MIT Press, 1990.*
47. *T.H. Cormen, C.E. Leiserson, R.L. Rivest: Introducere in Algoritmi, Computer Libris Agora, 2000.*
48. *Dorel Lucanu, Mitica Craus. Proiectarea algoritmilor. Polirom, 2008.*
49. *Л. Н. Домнин, Элементы теории графов, 2007*
50. *Web: https://www.ntu.edu.sg/home/guohua/mas331-2006summer.pdf*
51. *Ш.Ф.Арасланов, ТЕОРИЯ ГРАФОВ - ЛЕКЦИИ И ПРАКТИЧЕСКИЕ ЗАНЯТИЯ, 2013*
52. *Д. В. Карпов, Теория графов, 2017*
53. *Keijo Ruohonen, Graph theory, 2013*
54. *Wirth, N., Algorithms and Data Structures, Prentice Hall, Inc.,Englewood, New Jersey, 1986*
55. *Dr. Kris Jamsa & Lars Klander, Totul despre C si C++ - Manualul fundamental de programare în C si C++, ed. Teora, 1999-2006.*
56. *A. Ruceanu, Proiectarea algoritmilor. http://www.runceanu.ro/adrian/wpcontent/cursuri/pa2014.php*
57. *Cristian Uscatu, Cătălina Cocianu,Cătălin Silvestru, Algoritmi în programarea calculatoarelor, 2010, ISBN 978-606-505-465-3*
58. *А.М. РОМАНОВСКАЯ, М.В. МЕНДЗИВ, ДИНАМИЧЕСКОЕ ПРОГРАММИРОВАНИЕ, 2010, ISBN 978-5-91892-030-5*
59. *Струченков В. И., Динамическое программирование в примерах и задачах, 2015, ISBN: 978-5-4475-3820-0*
60. *G. Vasilache, S. Gîncu Culegere de probleme la informatica, Chişinău, 2012. http://en.calameo.com/read/002801569a611d413be1c*