

**Mariana Miloșescu**

# **Informatică intensiv**

**C++**

**Filierea teoretică,  
profilul real,  
specializarea  
*matematică - informatică,*  
*intensiv informatică***

**Manual pentru clasa a XI-a**



**EDITURA DIDACTICĂ ȘI PEDAGOGICĂ S.A.**

## 1. Tehnici de programare

### 1.1. Analiza algoritmilor

Prin **analiza unui algoritm** se identifică resursele necesare pentru executarea algoritmului: **timpul de execuție și memoria**.

Analiza algoritmilor este necesară atunci când există mai mulți algoritmi pentru rezolvarea aceleiași probleme și trebuie ales algoritmul cel mai eficient.

**Eficiența unui algoritm este evaluată prin timpul necesar pentru executarea algoritmului.**

Pentru a compara – din punct de vedere al eficienței – doi algoritmi care rezolvă aceeași problemă, se folosește aceeași **dimensiune a datelor de intrare** – **n** (același număr de valori pentru datele de intrare).

**Timpul de execuție** al algoritmului se exprimă prin numărul de **operații de bază** executate în funcție de **dimensiunea datelor de intrare**: **T(n)**.

Pentru a compara doi algoritmi din punct de vedere al timpului de execuție, trebuie să se stabilească unitatea de măsură care se va folosi, adică **operația de bază** executată în cadrul algoritmilor, după care, se numără de câte ori se execută operația de bază în cazul fiecărui algoritm.

**Operația de bază** este o operație elementară – sau o succesiune de operații elementare, a căror execuție nu depinde de valorile datelor de intrare.

Există algoritmi la care **timpul de execuție** depinde de **distribuția datelor de intrare**. Să considerăm doi algoritmi de sortare a unui vector cu **n** elemente – algoritmul de sortare prin metoda selecției directe și algoritmul de sortare prin metoda bulelor – și ca operație de bază **comparația**. Dacă, în cazul primului algoritm, timpul de execuție nu depinde de distribuția datelor de intrare (modul în care sunt aranjate elementele vectorului înainte de sortarea lui), el fiind  $T(n) = \frac{n \times (n - 1)}{2}$ , în cazul celui de al doilea algoritm timpul de execuție depinde de distribuția datelor de intrare (numărul de execuții ale structurii repetitive **while** depinde de modul în care sunt aranjate elementele vectorului înainte de sortare). În cazul în care numărul de execuții ale operațiilor elementare depinde de distribuția datelor de intrare, pentru analiza algoritmului se folosesc:

- **timpul maxim de execuție** – timpul de execuție pentru cazul cel mai nefavorabil de distribuție a datelor de intrare; în cazul sortării prin metoda bulelor, cazul cel mai nefavorabil este atunci când elementele vectorului sunt aranjate în ordine inversă decât aceea cerută de criteriul de sortare;
- **timpul mediu de execuție** – media timpilor de execuție pentru fiecare caz de distribuție a datelor de intrare.

Deoarece, în analiza eficienței unui algoritm, se urmărește comportamentul lui pentru o dimensiune mare a datelor de intrare, pentru a compara doi algoritmi din punct de vedere al eficienței, este suficient să se ia în considerare numai factorul care determină **timpul de execuție** – și care este denumit **ordinul de complexitate**.

**Respect pentru bântuirea grădinițelor**

**Ordinul de complexitate** al unui algoritm îl reprezintă timpul de execuție – estimat prin ordinul de mărime al numărului de execuții ale operației de bază:  $O(f(n))$ , unde  $f(n)$  reprezintă termenul determinant al timpului de execuție  $T(n)$ .

De exemplu, dacă – pentru algoritmul de sortare, prin metoda selecției directe – timpul de execuție este  $T(n) = \frac{n \times (n - 1)}{2} = \frac{n^2}{2} - \frac{n}{2}$ , ordinul de complexitate al algoritmului este  $O(n^2)$ , deoarece în calcularea lui se ia în considerare numai factorul determinant din timpul de execuție.

În funcție de **ordinul de complexitate**, există următoarele **tipuri de algoritmi**:

Ordin de complexitate	Tipul algoritmului
$O(n)$	<b>Algoritm liniar.</b>
$O(n^m)$	<b>Algoritm polinomial.</b> Dacă $m=2$ , algoritmul este <b>pătratic</b> , iar dacă $m=3$ , algoritmul este <b>cubic</b> .
$O(k^n)$	<b>Algoritm exponentiaj.</b> De exemplu: $2^n$ , $3^n$ etc. Algoritmul de tip $O(n!)$ este tot de tip exponentiaj, deoarece: $1 \times 2 \times 3 \times 4 \times \dots \times n > 2 \times 2 \times 2 \times \dots \times 2 = 2^{n-1}$ .
$O(\log n)$	<b>Algoritm logaritmic.</b>
$O(n \log n)$	<b>Algoritm liniar logaritmic.</b>

De exemplu, algoritmul de sortare prin metoda selecției directe este un algoritm **pătratic**. Ordinul de complexitate este determinat de structurile repetitive care se execută cu mulțimea de valori pentru datele de intrare. În cazul structurilor repetitive imbricate, ordinul de complexitate este dat de produsul dintre numărul de repetiții ale fiecărei structuri repetitive.

Structura repetitivă	Numărul de execuții ale corpului structurii	Tipul algoritmului
<b>for</b> ( $i=1; i <= n; i=i+k$ ) {....}	$f(n)=n/k \rightarrow O(n)=n$	Liniar
<b>for</b> ( $i=1; i <= n; i=i*k$ ) {....}	$f(n)=\log_k n \rightarrow O(n)=\log n$	Logaritmic
<b>for</b> ( $i=n; i >= n; i=i/k$ ) {....}	$f(n)=\log_k n \rightarrow O(n)=\log n$	Logaritmic
<b>for</b> ( $i=n; i <= n; i=i+p$ ) {....} <b>for</b> ( $j=n; j <= n; j=j+q$ ) {....}	$f(n)=(n/p)*(n/q) = n^2/(p*q) \rightarrow O(n)=n^2$	Polinomial pătratic
<b>for</b> ( $i=n; i <= n; i=i++$ ) {....} <b>for</b> ( $j=i; j <= n; j=j++$ ) {....}	$f(n)=1+2+3+\dots+n=(n*(n+1))/2 \rightarrow O(n)=n^2$	Polinomial pătratic

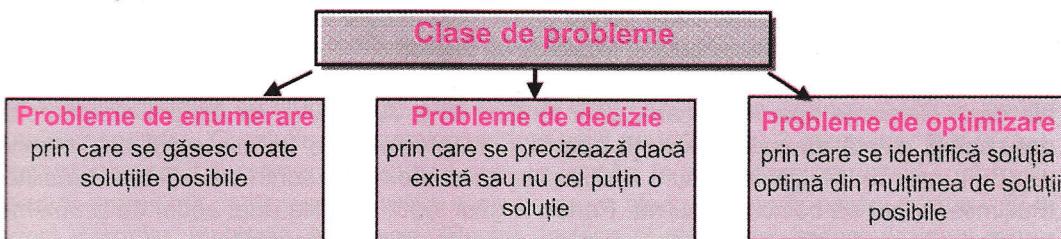
**Temă**



- Determinați complexitatea următorilor algoritmi și precizați tipul algoritmului. Pentru fiecare algoritm se va considera dimensiunea datelor de intrare –  $n$ .
- determinarea valorii minime dintr-un sir de numere;
  - inserarea unui element într-un vector, după un element cu valoare precizată;
  - ștergerea dintr-un vector a unui element cu valoare precizată,
  - stabilirea dacă un sir de numere conține numai numere distincte;
  - sortarea unui vector folosind metoda bulelor;
  - căutarea unui element cu valoare precizată, într-un vector nesortat;
  - căutarea unui element cu valoare precizată, într-un vector sortat;
  - determinarea tuturor permutărilor unei multimi de numere.

## 1.2. Metode de construire a algoritmilor

În funcție de procesul de calcul necesar pentru rezolvarea unei probleme, există următoarele clase de probleme:



Generarea tuturor permutărilor unei mulțimi de numere este o problemă de enumerare, căutarea unei valori precizate într-un sir de numere este o problemă de decizie, iar găsirea modalității de plată a unei sume s cu un număr minim de bancnote de valori date este o problemă de optimizare.

Pentru rezolvarea aceleiași probleme se pot folosi mai multe metode de construire a algoritmilor. Ați învățat deja că – pentru rezolvarea aceleiași probleme – puteți folosi un:

- **algoritm iterativ**;
- **algoritm recursiv**.

Soluțiile recursive sunt mult mai clare, mai scurte și mai ușor de urmărit. Alegerea algoritmului recursiv în locul celui iterativ este mai avantajoasă în cazul în care soluțiile problemei sunt definite recursiv sau dacă cerințele problemei sunt formulate recursiv.

Timpul de execuție a unui algoritm recursiv este dat de o formulă recursivă. De exemplu, pentru algoritmul de calculare a sumei primelor  $n$  numere naturale, funcția pentru timpul de execuție este prezentată alăturat, unde

$$T(n) = \begin{cases} \Theta(1) & \text{pentru } n=0 \\ \Theta(1)+T(n-1) & \text{pentru } n \neq 0 \end{cases}$$

$\Theta(1)$  reprezintă timpul de execuție a unei operații elementare de atribuire a unei valori sumei. Rezultă că  $T(n)=(n+1)\times\Theta(1)$ , iar ordinul de complexitate a algoritmului este  $O(n)$  la fel ca și cel al algoritmului iterativ. În cazul implementării recursive, fiecare apel al unui subprogram recurrent înseamnă încă o zonă de memorie rezervată pentru execuția subprogramului (variabilele locale și instrucțiunile). Din această cauză, în alegerea între un algoritm iterativ și un algoritm recursiv trebuie ținut cont nu numai de ordinul de complexitate, dar și de faptul că, pentru o **adâncime mare a recursivității**, algoritmii recursivi nu mai sunt eficienți, deoarece timpul de execuție crește, din cauza timpilor necesari pentru mecanismul de apel și pentru administrarea stivei de sistem.

Veți învăța noi **metode de construire** a algoritmilor – care vă oferă avantajul că prezintă fiecare o **metodă generală de rezolvare** care se poate aplica unei clase de probleme:

- **metoda backtracking**;
- **metoda divide et impera**;
- **metoda greedy**;
- **metoda programării dinamice**.

Fiecare dintre aceste metode de construire a algoritmilor se poate folosi pentru anumite clase de probleme, iar în cazul în care – pentru aceeași clasă de probleme – se pot folosi mai multe metode de construire a algoritmilor, criteriul de alegere va fi **eficiența** algoritmului.

## 1.3. Metoda backtracking

### 1.3.1. Descrierea metodei backtracking

Metoda backtracking se poate folosi pentru problemele în care trebuie să se genereze toate soluțiile, o soluție a problemei putând fi dată de un **vector**:

$$\mathbf{S} = \{\mathbf{x}_1, \mathbf{x}_2, \dots, \mathbf{x}_n\}$$

ale cărui elemente aparțin, fiecare, unor multimi finite  $\mathbf{A}_i$  ( $x_i \in \mathbf{A}_i$ ), iar asupra elementelor unei soluții există anumite **restricții** specifice problemei care trebuie rezolvată, numite **condiții interne**. Multimile  $\mathbf{A}_i$  sunt multimi ale căror elemente sunt în relații bine stabilită. Multimile  $\mathbf{A}_i$  pot să coincidă sau nu. Pentru a găsi toate soluțiile unei astfel de probleme folosind o **metodă clasică de rezolvare**, se execută următorul algoritm:

**PAS1.** Se generează toate elementele produsului cartezian  $\mathbf{A}_1 \times \mathbf{A}_2 \times \mathbf{A}_3 \times \dots \times \mathbf{A}_n$ .

**PAS2.** Se verifică fiecare element al produsului cartezian, dacă îndeplinește **condițiile interne** impuse ca să fie soluție a problemei.

### Studiu de caz

**Scop:** identificarea problemelor pentru care trebuie enumerate toate soluțiile, fiecare soluție fiind formată din **n** elemente  $\mathbf{x}_i$ , care aparțin fiecare unor multimi finite  $\mathbf{A}_i$  și care trebuie să respecte anumite **condiții interne**.

**Enunțul problemei 1:** Să se genereze toate permutările multimii {1, 2, 3}.

Cerința este de a enumera toate posibilitățile de generare a **3** numere naturale din multimea {1, 2, 3}, astfel încât numerele generate să fie distincte (**condiția internă** a soluției). O soluție a acestei probleme va fi un vector cu **3** elemente:  $\mathbf{S} = \{\mathbf{x}_1, \mathbf{x}_2, \mathbf{x}_3\}$ , în care elementul  $\mathbf{x}_i$  reprezintă numărul care se va găsi, în permutare, pe poziția  $i$ , iar multimea  $\mathbf{A}_i$  reprezintă multimea numerelor din care se va alege un număr pentru poziția  $i$ . În acest exemplu, multimile  $\mathbf{A}_i$  coincid. Ele au aceleași 3 elemente, fiecare element reprezentând un număr.

$$\mathbf{A}_i = \{1, 2, 3\} = \mathbf{A}$$

Dacă s-ar rezolva clasic această problemă, ar însemna să se genereze toate elementele produsului cartezian  $\mathbf{A}_1 \times \mathbf{A}_2 \times \mathbf{A}_3 = \mathbf{A} \times \mathbf{A} \times \mathbf{A} = \mathbf{A}^3$ , adică multimea:

$$\{(1,1,1), (1,1,2), (1,1,3), (1,2,1), \dots, (3,3,2), (3,3,3)\}$$

după care se va verifica fiecare element al multimii dacă este o soluție a problemei, adică dacă cele trei numere dintr-o soluție sunt distincte. Soluțiile obținute sunt:

$$\{(1,2,3), (1,3,2), (2,1,3), (2,3,1), (3,1,2), (3,2,1)\}$$

**Enunțul problemei 2:** Să se genereze toate aranjamentele de **2** elemente ale multimii {1, 2, 3}.

Cerința este de a enumera toate posibilitățile de generare a **2** numere naturale din multimea {1, 2, 3}, astfel încât numerele generate să fie distincte (**condiția internă** a soluției). O soluție a acestei probleme va fi un vector cu **2** elemente:  $\mathbf{S} = \{\mathbf{x}_1, \mathbf{x}_2\}$ , în care elementul  $\mathbf{x}_i$  reprezintă numărul care se va găsi în aranjament pe poziția  $i$ , iar multimea  $\mathbf{A}_i$  reprezintă multimea numerelor din care se va alege un număr pentru poziția  $i$ . Își în acest exemplu, multimile  $\mathbf{A}_i$  coincid. Ele au aceleași 3 elemente, fiecare element reprezentând un număr.

$$\mathbf{A}_i = \{1, 2, 3\} = \mathbf{A}$$

Dacă s-ar rezolva clasic această problemă, ar însemna să se genereze toate elementele produsului cartezian  $\mathbf{A}_1 \times \mathbf{A}_2 = \mathbf{A} \times \mathbf{A} = \mathbf{A}^2$ , adică multimea:

$$\{(1,1), (1,2), (1,3), (2,1), \dots, (3,2), (3,3)\}$$

după care se va verifica fiecare element al mulțimii, dacă este o soluție a problemei, adică dacă cele două numere dintr-o soluție sunt distincte. Soluțiile obținute sunt:

$$\{(1,2), (1,3), (2,1), (2,3), (3,1), (3,2)\}$$

**Enunțul problemei 3:** Să se genereze toate combinările de 2 elemente ale mulțimii {1, 2, 3}.

Cerința este de a enumera toate posibilitățile de generare a 2 numere naturale din mulțimea {1,2,3}, astfel încât numerele generate să fie distincte (**condiția internă** a soluției), iar soluțiile obținute să fie distincte. Două soluții sunt considerate distincte dacă nu conțin aceleași numere. O soluție a acestei probleme va fi un vector cu 2 elemente:  $S = \{x_1, x_2\}$ , în care elementul  $x_i$  reprezintă numărul care se va găsi în combinare pe poziția i, iar mulțimea  $A_i$  reprezintă mulțimea numerelor din care se va alege un număr pentru poziția i. Își în acest exemplu, mulțimile  $A_i$  coincid. Ele au aceleași 3 elemente, fiecare element reprezentând un număr.

$$A_i = \{1, 2, 3\} = A$$

Dacă s-ar rezolva clasic această problemă, ar însemna să se genereze toate elementele produsului cartezian  $A_1 \times A_2 = A \times A = A^2$ , adică mulțimea:

$$\{(1,1), (1, 2), (1,3), (2,1), \dots, (3,2), (3,3)\}$$

după care se va verifica fiecare element al mulțimii dacă este o soluție a problemei, adică dacă cele două numere dintr-o soluție sunt distincte și dacă soluția obținută este distinctă de soluțiile obținute anterior. Soluțiile obținute sunt:  $\{(1,2), (1,3), (2,3)\}$

**Enunțul problemei 4:** Să se genereze toate permutările mulțimii {1,2,3,4} care îndeplinesc condiția că 1 nu este vecin cu 3, și 2 nu este vecin cu 4.

Cerința este de a enumera toate posibilitățile de generare a 4 numere naturale din mulțimea {1, 2, 3, 4}, astfel încât numerele generate să fie distincte, iar 1 să nu se învecineze cu 3, și 2 să nu se învecineze cu 4 (**condiția internă** a soluției). O soluție a acestei probleme va fi un vector cu 4 elemente:  $S = \{x_1, x_2, x_3, x_4\}$ , în care elementul  $x_i$  reprezintă numărul care se va găsi în permuteare pe poziția i, iar mulțimea  $A_i$  reprezintă mulțimea numerelor din care se va alege un număr pentru poziția i. În acest exemplu, mulțimile  $A_i$  coincid. Ele au aceleași 4 elemente, fiecare element reprezentând un număr.  $A_i = \{1, 2, 3, 4\} = A$

Dacă s-ar rezolva clasic această problemă, ar însemna să se genereze toate elementele produsului cartezian  $A_1 \times A_2 \times A_3 \times A_4 = A \times A \times A \times A = A^4$ , adică mulțimea:

$$\{(1,1,1,1), (1,1,1,2), (1,1,1,3), (1,1,1,4), \dots, (4,4,4,3), (4,4,4,4)\}$$

după care se va verifica fiecare element al mulțimii dacă este o soluție a problemei, adică dacă cele patru numere dintr-o soluție sunt distincte și dacă 1 nu se învecinează cu 3, iar 2 cu 4. Soluțiile obținute sunt:

$$\{(1,2,3,4), (1,4,3,2), (2,1,4,3), (2,3,4,1), (3,2,1,4), (3,4,1,2), (4,1,2,3), (4,3,2,1)\}$$

**Enunțul problemei 5:** Să se aranjeze pe tabla de șah opt dame care nu se atacă între ele (problema celor 8 dame).

Cerința este de a enumera toate posibilitățile de aranjare a 8 dame pe o tablă de șah cu dimensiunea 8x8 (8 linii și 8 coloane), astfel încât toate cele 8 dame să nu se atace între ele (**condiția internă** a soluției). Deoarece nu se pot aranja două dame pe aceeași coloană (s-ar ataca între ele), înseamnă că pe fiecare coloană a tablei de șah se va pune o damă. O soluție a acestei probleme va fi un vector cu 8 elemente,  $S = \{x_1, x_2, x_3, x_4, x_5, x_6, x_7, x_8\}$ , în care elementul  $x_i$  reprezintă numărul liniei pe care se va pune dama în coloana i, iar mulțimea  $A_i$  reprezintă mulțimea liniilor pe care se poate aranja dama din coloana i. Își în acest caz mulțimile  $A_i$  coincid. Ele au aceleași opt elemente, fiecare element reprezentând un număr de linie:

$$A_i = \{1, 2, 3, 4, 5, 6, 7, 8\} = A$$

Dacă s-ar rezolva clasic această problemă, ar însemna să se genereze toate elementele produsului cartezian  $A_1 \times A_2 \times A_3 \times \dots \times A_8 = A \times A \times A \times \dots \times A = A^8$ , adică mulțimea:

$\{(1,1,1,1,1,1,1,1), (1,1,1,1,1,1,1,2), (1,1,1,1,1,1,1,3), \dots, (8,8,8,8,8,8,8,7), (8,8,8,8,8,8,8,8)\}$  după care se va verifica fiecare element al mulțimii, dacă este o soluție a problemei, adică dacă cele opt numere dintr-o soluție pot reprezenta coloanele pe care pot fi aranjate damele pe fiecare linie, astfel încât să nu se atace între ele. Soluțiile obținute sunt:

$$\{(1,5,8,6,3,7,2,4), (1,6,8,3,7,4,2,5), (1,7,4,6,8,2,5,3), \dots, (8,3,1,6,2,5,7,4), (8,4,1,3,6,2,7,5)\}$$

**Observație.** Metoda clasică de rezolvare a acestui tip de probleme necesită foarte multe operații din partea calculatorului, pentru a verifica fiecare element al produsului cartezian. Presupunând (pentru simplificare) că fiecare mulțime  $A_i$  are  $m$  elemente, atunci algoritmul de generare a elementelor produsului cartezian are complexitatea  $O(\text{card}(A_1) \times \text{card}(A_2) \times \dots \times \text{card}(A_n)) = O(m \times m \times m \times \dots \times m) = O(m^n)$ . Considerând că algoritmul prin care se verifică dacă un element al produsului cartezian este o soluție a problemei (respectă **condiția internă** a soluției) are complexitatea  $O(p)$ , atunci **complexitatea algoritmului** de rezolvare a problemei va fi  $O(p \times m^n)$ . De exemplu, în algoritmul de generare a permutărilor, complexitatea algoritmului de verificare a condiției interne este dată de complexitatea algoritmului prin care se verifică dacă numerele dintr-un sir sunt distințe. În acest algoritm, se parcurge sirul de  $m$  numere – și pentru fiecare număr din sir – se parcurge din nou sirul pentru a verifica dacă acel număr mai există în sir. Complexitatea algoritmului este dată de cele două structuri for imbricate:  $O(m^2) \rightarrow p = m^2$ .



**Metoda recomandată** pentru acest gen de probleme este metoda **backtracking** sau **metoda căutării cu revenire** – prin care se reduce volumul operațiilor de găsire a tuturor soluțiilor.

Metoda **backtracking** construiește progresiv vectorul soluției, pornind de la primul element și adăugând la vector următoarele elemente, cu revenire la elementul anterior din vector, în caz de insucces. Elementul care trebuie adăugat se caută în mulțime, printre elementele care respectă condițiile interne.

Prin metoda **backtracking** se obțin **toate soluțiile problemei**, dacă ele există. Pentru exemplificarea modului în care sunt construite soluțiile, considerăm problema generării permutărilor mulțimii  $\{1, 2, 3, \dots, n\}$  ( $A_1=A_2=\dots=A_n=A=\{1, 2, 3, \dots, n\}$ ).

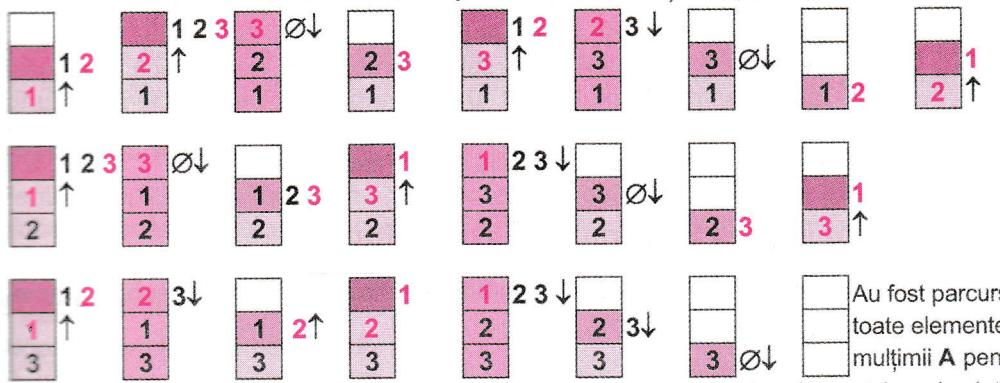
- PAS1.** Se alege primul element al soluției ca fiind primul element din mulțimea  $A$ . În exemplu,  $x_1=1$ , adică primul număr din permutare este 1.
- PAS2.** Se caută al doilea element al soluției ( $x_2$ ). Pentru a-l găsi, se parcurg pe rând elementele mulțimii  $A$  și, pentru fiecare element  $i$  al mulțimii, se verifică dacă respectă **condițiile interne**. Căutarea continuă până când se găsește primul element din mulțimea  $A$  care îndeplinește **condiția internă**, după care se oprește. În exemplu, se caută numărul de pe a doua poziție a permutării, verificându-se dacă al doilea număr din permutare este diferit de primul număr. Se parcurg primele două elemente ale mulțimii  $A$  și se găsește elementul  $x_2=2$ , după care procesul de căutare se oprește.
- PAS3.** Se caută al treilea element al soluției ( $x_3$ ). Căutarea va folosi același algoritm de la Pasul 2. În exemplu, se caută numărul din poziția a treia din permutare. Se găsește elementul  $x_3=3$ .
- PAS4.** Presupunând că s-au găsit primele  $k$  elemente ale soluției,  $x_1, x_2, x_3, \dots, x_k$ , se trece la căutarea celui de al  $k+1$ -lea element al soluției,  $x_{k+1}$ . Căutarea se va face astfel: se atribuie pe rând lui  $x_{k+1}$  elementele mulțimii  $A$ , până se găsește primul element  $i$  care îndeplinește condiția internă. În exemplu, condiția internă este ca

Respect pentru români și cărti  
numărul din poziția  $k+1$  a permutării să nu fie egal cu nici unul dintre numerele din pozițiile anterioare lui  $k+1$ . Pot să apară două situații:

- Există un element  $i$  în mulțimea  $A$ , astfel încât  $x_{k+1} = i$  să fie element al soluției problemei. În acest caz, se atribuie elementului  $x_{k+1}$  al soluției valoarea  $i$ , după care se verifică dacă s-a găsit soluția problemei. În exemplu, presupunem că pe nivelul  $k+1$  s-a găsit numărul 4. Se verifică dacă s-au generat toate cele  $n$  elemente ale mulțimii  $S$ , adică dacă s-a găsit numere pentru toate cele  $n$  poziții din permutare ( $k=n$ ). Dacă s-a găsit soluția problemei, atunci se afișează soluția; altfel, se caută următorul element al soluției, reluându-se operațiile de la Pasul 4.
- S-au parcurs toate elementele mulțimii  $A$  și nu s-a găsit nici un element  $i$  care să fie elementul  $x_{k+1}$  al soluției problemei. Înseamnă că trebuie să revenim la elementul  $k$  al soluției –  $x_k$ . Așadar, se consideră generate primele  $k-1$  elemente ale soluției:  $x_1, x_2, \dots, x_{k-1}$  și, pentru elementul  $x_k$  al soluției, se reia căutarea, cu **următorul element din mulțimea  $A$** , adică se reiau operațiile de la Pasul 4 pentru elementul  $x_k$  al soluției, însă nu cu primul element din mulțimea  $A$ , ci cu elementul din mulțimea  $A$  care se găsește imediat după cel care a fost atribuit anterior pentru elementul  $x_k$  al soluției. În exemplu, luând în considerare modul în care au fost generate primele  $k$  numere ale permutării, în poziția  $k+1$ , orice număr s-ar alege, el mai există pe una dintre cele  $k$  poziții anterioare, și se revine la elementul  $k$ , care presupunem că are valoarea 3. Se generează în această poziție următorul număr din mulțimea  $A$  (4) și se verifică dacă el nu mai există pe primele  $k-1$  poziții ale permutării, iar dacă există, se generează următorul element din mulțimea  $A$  (5) și.a.m.d.

- PAS5.** Algoritmul se încheie după ce au fost parcuse toate elementele mulțimii  $A$  pentru elementul  $x_1$  al soluției. În exemplu, algoritmul se încheie după ce s-au atribuit pe rând valorile 1, 2, ..., n, elementului de pe prima poziție a permutării.

#### Generarea tuturor permutărilor mulțimii {1, 2, 3}



Au fost parcuse toate elementele mulțimii  $A$  pentru elementul  $x_1$  al soluției.

**Observație.** În metoda backtracking, dacă s-a găsit elementul  $x_k$  al soluției, elementului  $x_{k+1}$  al soluției i se atribuie o valoare numai dacă mai există o valoare care să îndeplinească **condiția de continuare** a construirii soluției – adică dacă, prin atribuirea acelei valori, se poate ajunge la o soluție finală pentru care condițiile interne sunt îndeplinite.



Desenați diagramele pentru generarea prin metoda **backtracking** a:

- tuturor aranjamentelor de 2 elemente ale mulțimii {1, 2, 3};
- tuturor combinărilor de 2 elemente ale mulțimii {1, 2, 3};
- tuturor permutărilor mulțimii {1, 2, 3, 4} care îndeplinesc condiția că 1 nu este vecin cu 3, și 2 nu este vecin cu 4.

<b>1. Tehnici de programare .....</b>	3
<b>1.1. Analiza algoritmilor .....</b>	3
<b>1.2. Metode de construire a algoritmilor .....</b>	5
<b>1.3. Metoda backtracking .....</b>	6
1.3.1. Descrierea metodei backtracking .....	6
1.3.2. Implementarea metodei backtracking .....	10
1.3.3. Probleme rezolvabile prin metoda backtracking .....	14
1.3.3.1. Generarea permutărilor .....	15
1.3.3.2. Generarea produsului cartezian .....	17
1.3.3.3. Generarea aranjamentelor .....	20
1.3.3.4. Generarea combinărilor .....	22
1.3.3.5. Generarea tuturor partițiilor unui număr natural .....	24
1.3.3.6. Generarea tuturor partițiilor unei mulțimi .....	27
1.3.3.7. Generarea tuturor funcțiilor surjective .....	28
1.3.3.8. Problema celor n dame .....	30
1.3.3.9. Parcugerea tablei de șah cu un cal .....	31
1.3.3.10. Generarea tuturor posibilităților de ieșire din labirint .....	34
<b>1.4. Metoda „Divide et Impera“ .....</b>	40
1.4.1. Descrierea metodei „Divide et Impera“ .....	40
1.4.2. Implementarea metodei „Divide et Impera“ .....	41
1.4.3. Căutarea binară .....	48
1.4.4. Sortarea rapidă (QuickSort) .....	50
1.4.5. Sortarea prin interclasare (MergeSort) .....	53
1.4.6. Problema turnurilor din Hanoi .....	54
1.4.7. Generarea modelelor fractale .....	56
<b>1.5. Metoda greedy .....</b>	59
1.5.1. Descrierea metodei greedy .....	59
1.5.2. Implementarea metodei greedy .....	61
<b>1.6. Metoda programării dinamice .....</b>	70
1.6.1. Descrierea metodei programării dinamice .....	70
1.6.2. Implementarea metodei programării dinamice .....	73
<b>1.7. Compararea metodelor de construire a algoritmilor .....</b>	83
<b>Evaluare .....</b>	85
<b>2. Implementarea structurilor de date .....</b>	90
<b>2.1. Tipuri de date specifice pentru adresarea memoriei .....</b>	90
<b>2.2. Tipul de date pointer .....</b>	91
2.2.1. Declarația variabilei de tip pointer .....	92
2.2.2. Constante de tip adresă .....	92
2.2.3. Operatori pentru variabile de tip pointer .....	94
2.2.3.1. Operatorii specifici .....	94
2.2.3.2. Operatorul de atribuire .....	97
2.2.3.3. Operatorii aritmetici .....	100
2.2.3.4. Operatorii relaționali .....	101
<b>2.3. Tipul de date referință .....</b>	102
<b>2.4. Alocarea dinamică a memoriei .....</b>	106
<b>2.5. Clasificarea structurilor de date .....</b>	109
<b>2.6. Lista liniară .....</b>	113
2.6.1. Implementarea dinamică a listelor în limbajul C++ .....	115
2.6.2. Clasificarea listelor .....	117
2.6.3. Algoritmi pentru prelucrarea listelor simplu înlățuite .....	118
2.6.3.1. Inițializarea listei .....	118
2.6.3.2. Crearea listei .....	118

2.6.3.3. Adăugarea primului nod la listă .....	118
2.6.3.4. Adăugarea unui nod la listă .....	119
2.6.3.5. Parcurgerea listei .....	121
2.6.3.6. Căutarea unui nod în listă .....	121
2.6.3.7. Eliminarea unui nod din listă .....	122
2.6.3.8. Eliberarea spațiului de memorie ocupat de listă .....	123
2.6.3.9. Liste ordonate .....	124
2.6.3.10. Prelucrarea listelor simplu înlățuite .....	127
2.6.4. Algoritmi pentru prelucrarea listelor circulare simplu înlățuite .....	137
2.6.4.1. Crearea listei .....	137
2.6.4.2. Parcurgerea listei .....	138
2.6.4.3. Eliminarea unui nod din listă .....	138
2.6.5. Algoritmi pentru prelucrarea listelor dublu înlățuite .....	140
2.6.5.1. Adăugarea primului nod la listă .....	140
2.6.5.2. Adăugarea unui nod la listă .....	140
2.6.5.3. Parcurgerea listei .....	141
2.6.5.4. Eliminarea unui nod din listă .....	141
2.6.6. Algoritmi pentru prelucrarea stivelor .....	145
2.6.6.1. Inițializarea stivei .....	145
2.6.6.2. Adăugarea unui nod la stivă .....	146
2.6.6.3. Extragerea unui nod din stivă .....	146
2.6.6.4. Prelucrarea stivei .....	146
2.6.7. Algoritmi pentru prelucrarea cozilor .....	148
2.6.7.1. Inițializarea cozii .....	149
2.6.7.2. Adăugarea unui nod la coadă .....	149
2.6.7.3. Extragerea unui nod din coadă .....	149
2.6.7.4. Prelucrarea cozii .....	149
2.6.8. Aplicații practice .....	151
<b>Evaluare .....</b>	<b>152</b>
<b>2.7. Grafuri .....</b>	<b>159</b>
2.7.1. Definiția matematică a grafului .....	159
2.7.2. Graful neorientat .....	160
2.7.2.1. Terminologie .....	160
2.7.2.2. Gradul unui nod al grafului neorientat .....	162
2.7.2.3. Sirul grafic .....	164
2.7.3. Graful orientat .....	165
2.7.3.1. Terminologie .....	165
2.7.3.2. Gradele unui nod al grafului orientat .....	167
2.7.4. Reprezentarea și implementarea grafului .....	169
2.7.4.1. Reprezentarea prin matricea de adiacență .....	169
2.7.4.2. Reprezentarea prin matricea de incidentă .....	175
2.7.4.3. Reprezentarea prin lista muchiilor (arcelor) .....	182
2.7.4.4. Reprezentarea prin lista de adiacență (listele vecinilor) .....	187
2.7.4.5. Aplicații practice .....	197
2.7.5. Grafuri speciale .....	201
2.7.5.1. Graful nul .....	201
2.7.5.2. Graful complet .....	201
2.7.6. Grafuri derivate dintr-un graf .....	203
2.7.6.1. Graful parțial .....	203
2.7.6.2. Subgraful .....	207
2.7.6.3. Graful complementar .....	211
2.7.6.4. Aplicații practice .....	212

2.7.7. Conexitatea grafurilor .....	213
2.7.7.1. Lanțul .....	213
2.7.7.2. Ciclul .....	218
2.7.7.3. Drumul .....	220
2.7.7.4. Circuitul .....	223
2.7.7.5. Graful conex .....	225
2.7.7.6. Graful tare conex .....	228
2.7.4.7. Aplicații practice .....	235
2.7.8. Parcurgerea grafului .....	235
2.7.8.1. Parcurgerea în lățime – Breadth First .....	236
2.7.8.2. Parcurgerea în adâncime – Depth First .....	240
2.7.8.3. Aplicații practice .....	246
2.7.9. Graful ponderat .....	246
2.7.9.1. Definiția grafului ponderat .....	246
2.7.9.2. Matricea costurilor .....	247
2.7.9.3. Algoritmi pentru determinarea costului minim (maxim) .....	249
2.7.9.4. Aplicații practice .....	255
2.7.10. Grafuri speciale .....	256
2.7.10.1. Graful bipartit .....	256
2.7.10.2. Graful hamiltonian .....	260
2.7.10.3. Graful eulerian .....	263
2.7.10.4. Graful turneu .....	268
2.7.10.5. Aplicații practice .....	271
<b>Evaluare .....</b>	<b>272</b>
<b>2.8. Arboarele</b> .....	<b>281</b>
2.8.1. Arborele liber .....	281
2.8.1.1. Definiția arborelui liber .....	281
2.8.1.2. Proprietățile arborilor liberi .....	281
2.8.2. Arborele parțial .....	283
2.8.2.1. Definiția arborelui parțial .....	283
2.8.2.2. Definiția arborelui parțial de cost minim .....	283
2.8.2.3. Algoritmi de determinare a arborelui parțial de cost minim .....	285
2.8.2.4. Aplicații practice .....	292
2.8.3. Arborele cu rădăcină .....	292
2.8.3.1. Definiția arborelui cu rădăcină .....	292
2.8.3.2. Implementarea arborelui cu rădăcină .....	297
2.8.3.3. Algoritmi pentru parcugerea unui arbore cu rădăcină .....	299
2.8.3.4. Aplicații practice .....	304
2.8.4. Arborele binar .....	305
2.8.4.1. Definiția arborelui binar .....	305
2.8.4.2. Implementarea arborelui binar .....	305
2.8.4.3. Algoritmi pentru parcugerea unui arbore binar .....	312
2.8.4.4. Aplicații practice .....	315
2.8.5. Arborele binar de căutare .....	315
2.8.5.1. Definiția arborelui binar de căutare .....	315
2.8.5.2. Algoritmi pentru prelucrarea unui arbore binar de căutare .....	315
2.8.5.3. Aplicații practice .....	322
2.8.6. Ansamblul Heap .....	323
2.8.6.1. Definiția ansamblului Heap .....	323
2.8.6.2. Algoritmi pentru prelucrarea unui ansamblu Heap .....	324
2.8.6.3. Algoritmul HeapSort .....	327
2.8.5.4. Aplicații practice .....	329
<b>Evaluare .....</b>	<b>329</b>