MINISTERUL EDUCAŢIEI NAŢIONALE

Mariana Miloşescu

Informatică intensiv

C++

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

Manual pentru clasa a XI-a

EDITURA DIDACTICĂ ŞI 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 $\mathbf{T(n) = \frac{n \times (n-1)}{2}}$, 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**.

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)	**Algoritm liniar**.
$O(n^m)$	**Algoritm polinomial**. Dacă m=2, algoritmul este **pătratic**, iar dacă m=3, algoritmul este **cubic**.
$O(k^n)$	**Algoritm exponențial**. De exemplu: 2^n, 3^n etc. Algoritmul de tip O(n!) este tot de tip exponențial, deoarece: $1 \times 2 \times 3 \times 4 \times ... \times n > 2 \times 2 \times 2 \times ... \times 2 = 2^{n-1}$.
O(logn)	**Algoritm logaritmic**.
O(nlogn)	**Algoritm liniar logaritmic**.

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.

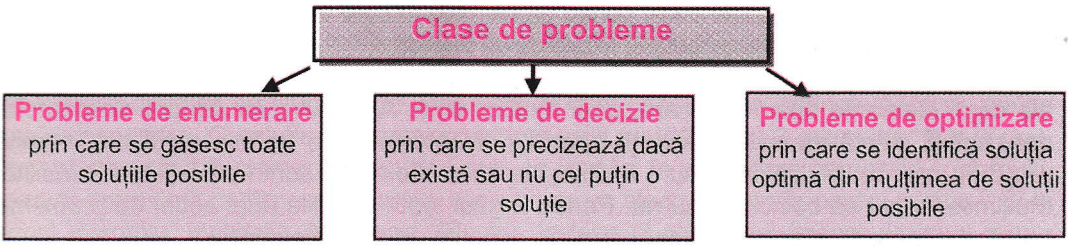
Structura repetitivă	Numărul de execuții ale corpului structurii	Tipul algoritmului
`for (i=1;i<=n;i=i+k) {....}`	$f(n)=n/k \rightarrow O(n)=n$	Liniar
`for (i=1;i<=n;i=i*k) {....}`	$f(n)=\log_k n \rightarrow O(n)=\log n$	Logaritmic
`for (i=n;i>=n;i=i/k) {....}`	$f(n)=\log_k n \rightarrow O(n)=\log n$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

Determinați complexitatea următorilor algoritmi și precizați tipul algoritmului. Pentru fiecare algoritm se va considera dimensiunea datelor de intrare – **n**.

a. determinarea valorii minime dintr-un șir de numere;
b. inserarea unui element într-un vector, după un element cu valoare precizată;
c. ștergerea dintr-un vector a unui element cu valoare precizată,
d. stabilirea dacă un șir de numere conține numai numere distincte;
e. sortarea unui vector folosind metoda bulelor;
f. căutarea unui element cu valoare precizată, într-un vector nesortat;
g. căutarea unui element cu valoare precizată, într-un vector sortat;
h. determinarea tuturor permutărilor unei mulțimi 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 şir 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 aceleiaşi probleme se pot folosi mai multe metode de construire a algoritmilor. Ați învățat deja că – pentru rezolvarea aceleiaşi probleme – puteți folosi un:

→ **algoritm iterativ**;
→ **algoritm recursiv**.

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

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

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 aceeaşi 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**:

$$S = \{x_1, x_2, \ldots, x_n\}$$

ale cărui elemente aparțin, fiecare, unor mulțimi finite A_i ($x_i \in A_i$), iar asupra elementelor unei soluții există anumite **restricții** specifice problemei care trebuie rezolvată, numite **condiții interne**. Mulțimile A_i sunt mulțimi ale căror elemente sunt în relații bine stabilite. Mulțimile 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 $A_1 \times A_2 \times A_3 \times \ldots \times A_n$.

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 x_i, care aparțin fiecare unor mulțimi finite A_i și care trebuie să respecte anumite **condiții interne**.

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

Cerința este de a enumera toate posibilitățile de generare a **3** numere naturale din mulțimea {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: $S = \{x_1, x_2, x_3\}$, în care elementul x_i reprezintă numărul care se va găsi, în permutare, 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 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 $A_1 \times A_2 \times A_3 = A \times A \times A = A^3$, adică mulțimea:

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

după care se va verifica fiecare element al mulțimii 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 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). 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 aranjament pe poziția **i**, iar mulțimea A_i reprezintă mulțimea numerelor din care se va alege un număr pentru poziția **i**. Și 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:

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

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: $\mathbf{S = \{x_1, x_2\}}$, în care elementul $\mathbf{x_i}$ reprezintă numărul care se va găsi în combinare pe poziția **i**, iar mulțimea $\mathbf{A_i}$ reprezintă mulțimea numerelor din care se va alege un număr pentru poziția **i**. Și în acest exemplu, mulțimile $\mathbf{A_i}$ coincid. Ele au aceleași 3 elemente, fiecare element reprezentând un număr. $\mathbf{A_i = \{1, 2, 3\} = A}$

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

{(1,1), (1, 2), (1,3), (2,1), …, (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: $\mathbf{S = \{x_1, x_2, x_3, x_4\}}$, în care elementul $\mathbf{x_i}$ reprezintă numărul care se va găsi în permutare pe poziția **i**, iar mulțimea $\mathbf{A_i}$ reprezintă mulțimea numerelor din care se va alege un număr pentru poziția **i**. În acest exemplu, mulțimile $\mathbf{A_i}$ coincid. Ele au aceleași 4 elemente, fiecare element reprezentând un număr. $\mathbf{A_i = \{1, 2, 3, 4\} = A}$

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

{(1,1,1,1), (1,1,1,2), (1,1,1,3), (1,1,1,4), …,(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, $\mathbf{S = \{x_1, x_2, x_3, x_4, x_5, x_6, x_7, x_8\}}$, în care elementul $\mathbf{x_i}$ reprezintă numărul liniei pe care se va pune dama în coloana **i**, iar mulțimea $\mathbf{A_i}$ reprezintă mulțimea liniilor pe care se poate aranja dama din coloana **i**. Și în acest caz mulțimile $\mathbf{A_i}$ coincid. Ele au aceleași opt elemente, fiecare element reprezentând un număr de linie: $\mathbf{A_i = \{1, 2, 3, 4, 5, 6, 7, 8\} = A}$

Dacă s-ar rezolva clasic această problemă, ar însemna să se genereze toate elementele produsului cartezian $A_1 \times A_2 \times A_3 \times ... \times A_8 = A \times A \times A \times ... \times A = A^8$, 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), ..., (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), ..., (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 $O(card(A_1) \times card(A_2) \times ... \times card(A_n)) = Q(m \times m \times m \times ... \times m) = O(m^n)$. 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 șir sunt distincte. În acest algoritm, se parcurge șirul de **m** numere – și pentru fiecare număr din șir – se parcurge din nou șirul pentru a verifica dacă acel număr mai există în șir. 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, ..., n} ($A_1=A_2= ... =A_n=A=\{1, 2, 3, ..., n\}$).

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, $x_1, x_2, x_3, ..., x_k$, 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

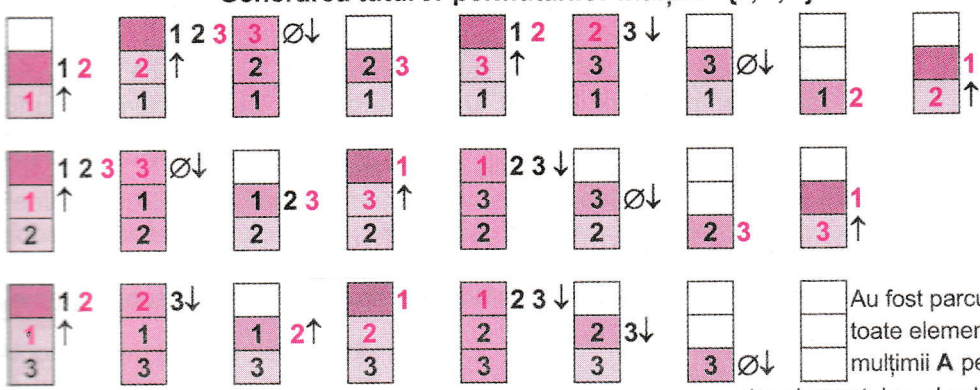
numărul din poziția **k+1** a permutării să nu fie egal cu nici unul dintre numerele din pozițiile anterioare lui **k+1**. Pot să apară două situații:

a. 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-au 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.

b. 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, ..., 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) ș.a.m.d.

PAS5. Algoritmul se încheie după ce au fost parcurse 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}



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.

Temă

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

a. tuturor aranjamentelor de 2 elemente ale mulțimii {1, 2, 3};

b. tuturor combinărilor de 2 elemente ale mulțimii {1, 2, 3};

c. tuturor permutărilor mulțimii {1, 2, 3, 4} care îndeplinesc condiția că 1 nu este vecin cu 3, și 2 nu este vecin cu 4.

Cuprins