

# PROIECTAREA ȘI IMPLEMENTAREA ALGORITMILOR

Conf. univ. dr. COSTEL BĂLCĂU

2020

# Tematica

<b>1</b>	<b>Elemente de complexitatea algoritmilor</b>	<b>7</b>
1.1	Notății asimptotice. Ordine de complexitate . . . . .	7
1.2	Determinarea maximului și minimului dintr-un vector . . . . .	10
<b>2</b>	<b>Metoda Greedy</b>	<b>13</b>
2.1	Descrierea metodei. Algoritmi generali . . . . .	13
2.2	Aplicații ale Inegalității rearanjamentelor . . . . .	16
2.2.1	Inegalitatea rearanjamentelor . . . . .	16
2.2.2	Produs scalar maxim/minim . . . . .	18
2.2.3	Memorarea optimă a textelor pe benzi . . . . .	22
2.3	Problema rucsacului, varianta continuă . . . . .	24
2.4	Problema planificării spectacolelor . . . . .	31
<b>3</b>	<b>Metoda Backtracking</b>	<b>38</b>
3.1	Descrierea metodei. Algoritmi generali . . . . .	38
3.2	Colorarea grafurilor . . . . .	46
3.3	Problema celor $n$ dame pe tabla de șah . . . . .	49
3.4	Problema nebunilor pe tabla de șah . . . . .	53
3.5	Generarea obiectelor combinatoriale . . . . .	62
3.5.1	Preliminarii . . . . .	62
3.5.2	Produs cartezian . . . . .	63
3.5.3	Submulțimi . . . . .	64
3.5.4	Aranjamente cu repetiție . . . . .	66
3.5.5	Aranjamente . . . . .	67
3.5.6	Permutări . . . . .	70
3.5.7	Combinări . . . . .	73
3.5.8	Combinări cu repetiție . . . . .	75
3.5.9	Permutări cu repetiție . . . . .	77
3.5.10	Compuneri ale unui număr natural . . . . .	82
3.5.11	Partiții ale unui număr natural . . . . .	88

<b>4</b>	<b>Metoda Divide et Impera</b>	<b>93</b>
4.1	Descrierea metodei. Algoritmi generali . . . . .	93
4.2	Problema turnurilor din Hanoi . . . . .	96
4.3	O problemă de acoperire . . . . .	99
4.4	Căutarea binară . . . . .	102
4.5	Algoritmi de sortare . . . . .	104
4.5.1	Problema sortării . . . . .	104
4.5.2	Interclasarea a doi vectori . . . . .	104
4.5.3	Sortarea prin interclasare (mergesort) . . . . .	108

# Evaluare

- Activitate laborator: 30% (Programe și probleme din Temele de laborator)
- Teme de casă: 20% (Programe și probleme suplimentare)
- Examen final: 50% (Probă scrisă: teorie, algoritmi -cu implementare- și probleme)

# Bibliografie

- [1] A.V. Aho, J.E. Hopcroft, J.D. Ullman, *Data Structures and Algorithms*, Addison-Wesley, Massachusetts, 2009.
- [2] Gh. Barbu, V. Păun, *Programarea în limbajul C/C++*, Editura Matrix Rom, București, 2011.
- [3] Gh. Barbu, V. Păun, *Calculatoare personale și programare în C/C++*, Editura Didactică și Pedagogică, București, 2005.
- [4] Gh. Barbu, I. Văduva, M. Boloșteanu, *Bazele informaticii*, Editura Tehnică, București, 1997.
- [5] C. Bălcău, *Combinatorică și teoria grafurilor*, Editura Universității din Pitești, Pitești, 2007.
- [6] O. Bâscă, L. Livovschi, *Algoritmi euristici*, Editura Universității din București, București, 2003.
- [7] E. Cerchez, M. Șerban, *Programarea în limbajul C/C++ pentru liceu. Vol. 2: Metode și tehnici de programare*, Ed. Polirom, Iași, 2005.
- [8] E. Ciurea, L. Ciupală, *Algoritmi. Introducere în algoritmica fluxurilor în rețele*, Editura Matrix Rom, București, 2006.
- [9] T.H. Cormen, *Algorithms Unlocked*, MIT Press, Cambridge, 2013.
- [10] T.H. Cormen, C.E. Leiserson, R.L. Rivest, C. Stein, *Introduction to Algorithms*, MIT Press, Cambridge, 2009.
- [11] C. Croitoru, *Tehnici de bază în optimizarea combinatorie*, Editura Universității "Al. I. Cuza", Iași, 1992.
- [12] N. Dale, C. Weems, *Programming and problem solving with JAVA*, Jones & Bartlett Publishers, Sudbury, 2008.
- [13] D. Du, X. Hu, *Steiner Tree Problems in Computer Communication Networks*, World Scientific Publishing Co. Pte. Ltd., Hackensack, 2008.
- [14] S. Even, *Graph Algorithms*, Cambridge University Press, Cambridge, 2012.

- [15] H. Georgescu, *Tehnici de programare*, Editura Universității din București, București, 2005.
- [16] C.A. Giumale, *Introducere în analiza algoritmilor. Teorie și aplicații*, Ed. Polirom, Iași, 2004.
- [17] M. Goodrich, R. Tamassia, *Algorithm Design. Foundations, Analysis and Internet Examples*, Wiley, New Delhi, 2011.
- [18] F.V. Jensen, T.D. Nielsen, *Bayesian Networks and Decision Graphs*, Springer, New York, 2007.
- [19] D. Jungnickel, *Graphs, Networks and Algorithms*, Springer, Heidelberg, 2013.
- [20] D.E. Knuth, *The Art Of Computer Programming. Vol. 4A: Combinatorial Algorithms*, Addison-Wesley, Massachusetts, 2011.
- [21] B. Korte, J. Vygen, *Combinatorial Optimization. Theory and Algorithms*, Springer, Heidelberg, 2012.
- [22] R. Lafore, *Data Structures and Algorithms in Java*, Sams Publishing, Indianapolis, 2002.
- [23] A. Levitin, *Introduction to The Design and Analysis of Algorithms*, Pearson, Boston, 2012.
- [24] L. Livovschi, H. Georgescu, *Sinteza și analiza algoritmilor*, Editura Științifică și Enciclopedică, București, 1986.
- [25] D. Logofătu, *Algoritmi fundamentali în C++: Aplicații*, Ed. Polirom, Iași, 2007.
- [26] D. Logofătu, *Algoritmi fundamentali în Java: Aplicații*, Ed. Polirom, Iași, 2007.
- [27] D. Lucanu, M. Craus, *Proiectarea algoritmilor*, Ed. Polirom, Iași, 2008.
- [28] D.A. Popescu, *Bazele programării - Java după C++*, [ebooks.infobits.ro](http://ebooks.infobits.ro), 2019.
- [29] D.R. Popescu, *Combinatorică și teoria grafurilor*, Societatea de Științe Matematice din România, București, 2005.
- [30] N. Popescu, *Data structures and algorithms using Java*, Editura Politehnica Press, București, 2008.
- [31] V. Preda, C. Bălcău, *Entropy optimization with applications*, Editura Academiei Române, București, 2010.
- [32] R. Sedgewick, P. Flajolet, *An Introduction to the Analysis of Algorithms*, Addison-Wesley, New Jersey, 2013.
- [33] R. Sedgewick, K. Wayne, *Algorithms*, Addison-Wesley, Massachusetts, 2011.
- [34] R. Stephens, *Essential Algorithms: A Practical Approach to Computer Algorithms*, Wiley, Indianapolis, 2013.

- [35] Ș. Tănasă, C. Olaru, Ș. Andrei, *Java de la 0 la expert*, Ed. Polirom, Iași, 2007.
- [36] T. Toadere, *Grafe. Teorie, algoritmi și aplicații*, Editura Albastră, Cluj-Napoca, 2002.
- [37] I. Tomescu, *Combinatorică și teoria grafurilor*, Tipografia Universității din București, București, 1978.
- [38] I. Tomescu, *Probleme de combinatorică și teoria grafurilor*, Editura Didactică și Pedagogică, București, 1981.
- [39] I. Tomescu, *Data structures*, Editura Universității din București, București, 2004.
- [40] M.A. Weiss, *Data Structures and Algorithm Analysis in Java*, Addison-Wesley, New Jersey, 2012.
- [41] \*\*\*, *Handbook of combinatorics*, edited by R.L. Graham, M. Grótschel and L. Lovász, Elsevier, Amsterdam, 1995.
- [42] \*\*\*, *Handbook of discrete and combinatorial mathematics*, edited by K.H. Rosen, J.G. Michaels, J.L. Gross, J.W. Grossman and D.R. Shier, CRC Press, Boca Raton, 2000.
- [43] \*\*\*, *Revista MATINF*, editată de Departamentul de Matematică-Informatică, Universitatea din Pitești, Editura Universității din Pitești, 2018-2020, <http://matinf.upit.ro>.

# Tema 1

## Elemente de complexitatea algoritmilor

### 1.1 Notății asimptotice. Ordine de complexitate

Timpul de execuție al unui algoritm depinde, în general, de setul datelor de intrare, iar pentru fiecare astfel de set el este bine determinat de numărul de operații executate și de tipul acestora. Astfel timpul de execuție al unui algoritm poate fi interpretat și analizat drept o funcție pozitivă ce are ca argument dimensiunea datelor de intrare.

**Definiția 1.1.1.** Pentru orice algoritm  $\mathcal{A}$ , notăm cu  $T_{\mathcal{A}}(n)$  **timpul de execuție** pentru algoritmul  $\mathcal{A}$  corespunzător unui set de date de intrare având dimensiunea totală  $n$ .

*Observația 1.1.1.* Pentru simplificarea calculelor, de cele mai multe ori sunt analizate numai anumite operații, semnificative pentru algoritmi respectivi, evaluarea timpului de execuție rezumându-se astfel la numărarea sau estimarea acestor operații.

**Definiția 1.1.2.** Un algoritm  $\mathcal{A}$  este considerat **optim** dacă (se demonstrează că) nu există un algoritm având un timp de execuție mai bun pentru rezolvarea problemei date, adică pentru orice algoritm  $\mathcal{A}'$  care rezolvă problema dată avem  $T_{\mathcal{A}}(n) \leq T_{\mathcal{A}'}(n)$ , pentru orice  $n$ .

Obținerea de algoritmi pur optimi - în sensul definiției anterioare - este posibilă în puține situații, iar demonstrarea optimalității acestora este de obicei dificilă. Mult mai des se întâlnesc algoritmi cu o comportare apropiată de cea optimă, pentru valori suficient de mari ale dimensiunii setului datelor



de intrare. Prezentăm în continuare câteva noțiuni prin care se cuantifică această apropiere.

**Definiția 1.1.3.** O funcție **asimptotic pozitivă** (prescurtat **a.p.**) este o funcție  $f : \mathbb{N} \setminus A \rightarrow \mathbb{R}$  a.î.

- $A \subset \mathbb{N}$  este o mulțime finită;
- $\exists n_0 \in \mathbb{N} \setminus A$  astfel încât  $f(n) > 0, \forall n \geq n_0$ .

*Observația 1.1.2.* De cele mai multe ori, mulțimea  $A$  este de forma

$$A = \underbrace{\{0, 1, 2, \dots, k\}}_{\text{primele numere naturale}}, \text{ unde } k \in \mathbb{N}.$$

*Exemplul 1.1.1.* Funcția  $f : D \rightarrow \mathbb{R}, f(n) = \frac{(3n^4 + n + 3)\sqrt{n-5}}{(5n+1)(n-8)}$ , unde  $D = \{n \in \mathbb{N} \mid n \geq 5, n \neq 8\}$ , este asimptotic pozitivă, deoarece  $D = \mathbb{N} \setminus A$  cu  $A = \{0, 1, 2, 3, 4, 8\}$  (mulțime finită) și  $f(n) > 0, \forall n \geq 9$ .

*Exemplul 1.1.2.* Funcția  $g : \mathbb{N} \setminus \{1, 6\} \rightarrow \mathbb{R}, g(n) = \frac{\ln(n^5 + 1) - n}{(n-1)(n-6)}$ , nu este asimptotic pozitivă, deoarece  $(n-1)(n-6) > 0, \forall n \geq 7$ , dar  $\lim_{n \rightarrow \infty} [\ln(n^5 + 1) - n] = -\infty$ , deci  $\exists n_0 \in \mathbb{N}, n_0 \geq 7$  a.î.  $g(n) < 0, \forall n \geq n_0$ .

Următorul rezultat este o consecință a definiției anterioare.

**Lema 1.1.1.** O funcție polinomială  $f : \mathbb{N} \rightarrow \mathbb{R}$ , de grad  $p$ ,

$$f(n) = a_p \cdot n^p + a_{p-1} \cdot n^{p-1} + \dots + a_1 \cdot n + a_0, \quad a_0, a_1, \dots, a_p \in \mathbb{R}, \quad a_p \neq 0,$$

este asimptotic pozitivă dacă și numai dacă  $a_p > 0$ .

**Definiția 1.1.4.** Fie  $f$  și  $g$  două funcții asimptotic pozitive.

a)  $f$  și  $g$  se numesc **asimptotic echivalente** și notăm  $f(n) \sim g(n)$  dacă  $\exists \lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = 1$ .

b) Spunem că  $f$  este **asimptotic mărginită superior** de  $g$ , iar  $g$  este **asimptotic mărginită inferior** de  $f$  și notăm  $f(n) = \mathcal{O}(g(n))$  și  $g(n) = \Omega(f(n))$  dacă  $\exists c > 0, \exists n_0 \in \mathbb{N}$  astfel încât  $f(n) \leq c \cdot g(n), \forall n \geq n_0$ .

c) Spunem că  $f$  și  $g$  **au același ordin de creștere** și notăm  $f(n) = \Theta(g(n))$  dacă  $f(n) = \mathcal{O}(g(n))$  și  $f(n) = \Omega(g(n))$ .

Următorul rezultat este o consecință a definiției anterioare.

**Propoziția 1.1.1.** Fie  $f$  și  $g$  două funcții asimptotic pozitive. Presupunem că există  $\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = L$ . Atunci:

- a)  $f(n) = \mathcal{O}(g(n))$  dacă și numai dacă  $L \in [0, +\infty)$ ;
- b)  $f(n) = \Omega(g(n))$  dacă și numai dacă  $L \in (0, +\infty]$ ;
- c)  $f(n) = \Theta(g(n))$  dacă și numai dacă  $L \in (0, +\infty)$ .

**Corolarul 1.1.1.** Fie  $f$  și  $g$  două funcții asimptotic pozitive. Dacă  $f(n) \sim g(n)$ , atunci  $f(n) = \Theta(g(n))$ .

Următorul rezultat este o consecință a propoziției anterioare.

**Propoziția 1.1.2.** Fie  $f : \mathbb{N} \rightarrow \mathbb{R}$  o funcție polinomială de grad  $p$ ,

$$f(n) = a_p \cdot n^p + a_{p-1} \cdot n^{p-1} + \cdots + a_1 \cdot n + a_0, \quad a_0, a_1, \dots, a_p \in \mathbb{R},$$

astfel încât  $a_p > 0$ .

Atunci

- a)  $f(n) = \mathcal{O}(n^k)$ ,  $\forall k \geq p$ ;
- b)  $f(n) = \Omega(n^k)$ ,  $\forall k \leq p$ ;
- c)  $f(n) = \Theta(n^p)$ ;
- d)  $f(n) \sim a_p \cdot n^p$ .

**Definiția 1.1.5.** Fie  $\mathcal{A}$  un algoritm și  $f$  o funcție asimptotic pozitivă. Spunem că algoritmul  $\mathcal{A}$  are **ordinul de complexitate (complexitatea)**  $\mathcal{O}(f(n))$ ,  $\Omega(f(n))$ , respectiv  $\Theta(f(n))$  dacă  $T_{\mathcal{A}}(n) = \mathcal{O}(f(n))$ ,  $T_{\mathcal{A}}(n) = \Omega(f(n))$ , respectiv  $T_{\mathcal{A}}(n) = \Theta(f(n))$ .

**Definiția 1.1.6.** Fie  $\mathcal{A}$  un algoritm,  $n$  dimensiunea datelor de intrare și  $T(n)$  timpul de execuție estimat pentru algoritmul  $\mathcal{A}$ .

- 1) Spunem că algoritmul  $\mathcal{A}$  are **complexitate (comportare) polinomială** (sau că este **polinomial** sau că **aparține clasei  $P$** ) dacă  $\exists p > 0$  astfel încât  $T(n) = \mathcal{O}(n^p)$ .
- 2) În particular, dacă  $T(n) = \mathcal{O}(n)$  atunci spunem că algoritmul  $\mathcal{A}$  are **complexitate (comportare) liniară** (sau că este **liniar**).

*Observația 1.1.3.* Algoritmii polinomiali sunt, în general, acceptabili în practică. Algoritmii care necesită un timp de calcul exponențial sunt utilizați numai în cazuri excepționale și doar pentru date de intrare de dimensiuni relativ mici.

*Observația 1.1.4.* Notăția  $\mathcal{O}$  se utilizează pentru a exprima complexitatea unui algoritm corespunzătoare  *timpului de execuție în cazul cel mai defavorabil*, fiind astfel cea mai adecvată analizei algoritmilor. Notăția  $\Omega$  este corespunzătoare  *timpului de execuție în cazul cel mai favorabil*, caz practic irelevant, fiind astfel mai puțin utilizată. Notățiile  $\sim$  și  $\Theta$  se utilizează atunci când se constată că timpii de execuție corespunzători cazurilor cel mai defavorabil și cel mai favorabil fie sunt chiar asimptotic echivalenți (notația  $\sim$ , deci și notația  $\Theta$ ), cazul cel mai simplu fiind acela al algoritmilor a căror executare depinde doar de dimensiunea setului de date de intrare, nu și de valorile acestor date, fie au măcar același ordin de creștere (notația  $\Theta$ ). Tot aceste notații se utilizează și atunci când se poate determina  *timpul mediu de execuție*  al algoritmului, calculat ca medie aritmetică ponderată a timpilor de execuție pentru toate seturile de date de intrare posibile, ponderile fiind frecvențele de apariție ale acestor seturi.

**Definiția 1.1.7.** *Un algoritm  $\mathcal{A}$  este considerat **asimptotic-optim** dacă (se demonstrează că) nu există un algoritm având un ordin de complexitate mai bun pentru rezolvarea problemei date, adică pentru orice algoritm  $\mathcal{A}'$  care rezolvă problema dată avem  $T_{\mathcal{A}}(n) = \mathcal{O}(T_{\mathcal{A}'}(n))$ .*

*Observația 1.1.5.* Evident, orice algoritm optim este asimptotic-optim. Reciproca acestei afirmații nu este adevărată, în continuare fiind prezentat un exemplu în acest sens.

## 1.2 Determinarea maximului și minimului dintr-un vector

**Problema determinării maximului și minimului dintr-un vector** este următoarea: Se consideră un vector  $A = (a_1, a_2, \dots, a_n)$ ,  $n \geq 1$ . Se cere să se determine maximul și minimul dintre elementele  $a_1, a_2, \dots, a_n$ , adică perechea  $(M, m)$ , unde  $M = \max\{a_i \mid 1 \leq i \leq n\}$ ,  $m = \min\{a_i \mid 1 \leq i \leq n\}$ .

Un algoritm uzual de rezolvare este următorul.

**MAX-MIN**( $A, n, M, m$ ) :

$M \leftarrow a_1; m \leftarrow a_1;$

**for**  $i = \overline{2, n}$  **do**

**if**  $a_i > M$  **then**

$M \leftarrow a_i;$

**else**

**if**  $a_i < m$  **then**

$m \leftarrow a_i;$

Pentru evaluarea complexității algoritmilor care rezolvă problema dată, vom analiza numai comparațiile în care intervin elemente ale vectorului sau valorile  $M$  și  $m$ , numite *comparații de chei*. Evident, algoritmul MAX-MIN efectuează cel puțin  $n - 1$  și cel mult  $2n - 2$  astfel de comparații, iar celelalte operații nu depășesc ordinul de creștere al acestora, deci are complexitatea  $\Theta(n)$ .

Vom utiliza următorul rezultat.

**Lema 1.2.1.** *Pentru determinarea maximului dintre  $n$  numere,  $n \in \mathbb{N}^*$ , sunt necesare  $n - 1$  comparații.*

*Demonstrație.* Notăm proprietatea din enunț cu  $P(n)$  și îi demonstrăm valabilitatea prin inducție după  $n$ .

$P(1)$ : Pentru  $n = 1$  sunt necesare  $0 = 1 - 1$  comparații, deci  $P(1)$  este adevărată.

$P(k - 1) \Rightarrow P(k)$ : Fie  $k \in \mathbb{N}$ ,  $k \geq 2$ . Presupunem afirmația adevărată pentru orice  $k - 1$  numere și o demonstrăm pentru  $k$  numere.

Fie  $a_1, a_2, \dots, a_{k-1}, a_k$  aceste numere.

Fie  $a_i$  și  $a_j$ ,  $i \neq j$ , numerele care sunt supuse primei comparații.

Presupunem că  $a_i \geq a_j$  (raționamentul este similar în cazul când  $a_j \geq a_i$ ).

Atunci

$$\max\{a_1, a_2, \dots, a_{k-1}, a_k\} = \max\{a_1, a_2, \dots, a_{j-1}, a_{j+1}, \dots, a_{k-1}, a_k\},$$

adică avem de determinat în continuare maximul dintre  $k - 1$  numere (celelalte  $k - 2$  numere și  $a_i$ ). Pentru aceasta, conform ipotezei de inducție, sunt necesare încă  $k - 2$  comparații. Deci obținem un total de  $1 + (k - 2) = k - 1$  comparații.

Demonstrația prin inducție este astfel încheiată.  $\square$

Conform lemei anterioare, orice algoritm  $\mathcal{A}$  care calculează maximul dintre  $n$  elemente, bazat pe comparații de chei, necesită cel puțin  $n - 1$  astfel de comparații, deci are complexitatea  $\Omega(n)$ . Astfel  $T_{\mathcal{A}}(n) = \Omega(T_{\text{MAX-MIN}}(n))$ , sau, echivalent,  $T_{\text{MAX-MIN}}(n) = \mathcal{O}(T_{\mathcal{A}}(n))$ , deci am obținut următorul rezultat:

**Propoziția 1.2.1.** *Algoritmul MAX-MIN este asimptotic-optimal (în clasa algoritmilor bazați pe comparații de chei).*

Pe de altă parte, avem:

**Propoziția 1.2.2.** *Din punct de vedere al timpului de execuție în cazul cel mai defavorabil, algoritmul MAX-MIN nu este optim.*

*Demonstrație.* Există algoritmi (bazați pe comparații de chei) care în cazul cel mai defavorabil efectuează mai puțin de  $2n - 2$  comparații de chei, cât efectuează algoritmul MAX-MIN. Un astfel de exemplu este următorul algoritm, ce compară  $a_1$  cu  $a_2$ ,  $a_3$  cu  $a_4$ ,  $\dots$ , și calculează maximumul dintre maximele acestor perechi și minimumul dintre minimele perechilor.

**MAX-MIN-PER**( $A, n, M, m$ ) :

$M \leftarrow a_n; m \leftarrow a_n;$

**for**  $i = 1, \lfloor n/2 \rfloor$  **do**

**if**  $a_{2i-1} > a_{2i}$  **then**

**if**  $a_{2i-1} > M$  **then**

$M \leftarrow a_{2i-1};$

**if**  $a_{2i} < m$  **then**

$m \leftarrow a_{2i};$

**else**

**if**  $a_{2i} > M$  **then**

$M \leftarrow a_{2i};$

**if**  $a_{2i-1} < m$  **then**

$m \leftarrow a_{2i-1};$

( $\lfloor x \rfloor$  reprezintă partea întreagă a numărului real  $x$ ). Evident, algoritmul MAX-MIN-PER efectuează exact  $3 \cdot \lfloor \frac{n}{2} \rfloor$  comparații de chei, indiferent de ordinea dintre elementele vectorului  $A$ .  $\square$

*Observația 1.2.1.* Algoritmul MAX-MIN-PER are tot complexitatea  $\Theta(n)$ , deoarece efectuează  $3 \cdot \lfloor \frac{n}{2} \rfloor$  comparații de chei, iar celelalte operații nu depășesc ordinul de creștere al acestora.

# Tema 2

## Metoda Greedy

### 2.1 Descrierea metodei. Algoritmi generali

Metoda **Greedy** (a **optimului local**) presupune elaborarea unor strategii de rezolvare a *problemelor de optim*, în care se urmărește *maximizarea* sau *minimizarea* unei *funcții obiectiv*.

Se aplică problemelor în care se dă o mulțime finită  $A = \{a_1, a_2, \dots, a_n\}$  (**mulțimea de candidați**), conținând  $n$  date de intrare, pentru care se cere să se determine o submulțime  $B \subseteq A$  care să îndeplinească anumite condiții pentru a fi acceptată. Această submulțime se numește **soluție posibilă** (**soluție admisibilă**, pe scurt **soluție**).

Deoarece, în general, există mai multe *soluții posibile*, trebuie avut în vedere și un *criteriu de selecție*, conform căruia, dintre acestea, să fie aleasă una singură ca rezultat final, numită **soluție optimă**.

**Soluțiile posibile** au următoarele proprietăți:

- mulțimea vidă  $\emptyset$  este întotdeauna soluție posibilă;
- dacă  $B$  este soluție posibilă și  $C \subseteq B$ , atunci și  $C$  este soluție posibilă.

În continuare sunt prezentate două scheme de lucru, care urmează aceeași idee, diferențiindu-se doar prin ordinea de efectuare a unor operații:

*Algoritmul 2.1.1 (Metoda Greedy, varianta I).*

- Se pleacă de la soluția vidă ( $\emptyset$ );
- Se alege, pe rând, într-un anumit fel, un element din  $A$  neales la pași precedenți.

- Dacă includerea elementului ales în soluția parțială construită anterior conduce la o soluție posibilă, atunci construim noua soluție prin adăugarea elementului ales.

**GREEDY1**( $A, n, B$ ):  
 $B \leftarrow \emptyset$ ;  
**for**  $i = \overline{1, n}$  **do**  
     $x \leftarrow \mathbf{ALEGE}(A, i, n)$ ;  
    **if** **SOLUTIE\_POSIBILA**( $B, x$ ) **then**  
         $B \leftarrow B \cup \{x\}$ ;

*Observația 2.1.1.*

- Funcția **ALEGE**( $A, i, n$ ) returnează un element  $x = a_j \in \{a_i, \dots, a_n\}$  și efectuează interschimbarea  $a_i \leftrightarrow a_j$ ;
- Funcția **SOLUTIE\_POSIBILA**( $B, x$ ) verifică dacă  $B \cup \{x\}$  este soluție posibilă a problemei.
- Funcția **ALEGE** este cea mai dificil de realizat, deoarece trebuie să implementeze criteriul conform căruia alegerea la fiecare pas a câte unui candidat să conducă în final la obținerea soluției optime.

*Algoritmul 2.1.2 (Metoda Greedy, varianta a II-a).*

Metoda e asemănătoare primeia, cu excepția faptului că se stabilește de la început ordinea în care trebuie analizate elementele din  $A$ .

**GREEDY2**( $A, n, B$ ):  
**PRELUCREAZA**( $A, n$ );  
 $B \leftarrow \emptyset$ ;  
**for**  $i = \overline{1, n}$  **do**  
    **if** **SOLUTIE\_POSIBILA**( $B, a_i$ ) **then**  
         $B \leftarrow B \cup \{a_i\}$ ;

*Observația 2.1.2.* Prin apelul procedurii **PRELUCREAZA**( $A, n$ ) se efectuează o permutare a elementelor mulțimii  $A$ , stabilind ordinea de analiză a acestora. Aceasta este procedura cea mai dificil de realizat.

*Observația 2.1.3.*

- Metoda Greedy nu caută să determine toate soluțiile posibile și apoi să aleagă pe cea optimă conform criteriului de optimizare dat (ceea ce ar necesita în general un timp de calcul și spațiu de memorie mari), ci constă în a alege pe rând câte un element, urmând să-l ”înghită” eventual în soluția optimă. De aici vine și numele metodei (*Greedy = lacom*).
- Astfel, dacă trebuie determinat maximul unei funcții de cost depinzând de  $a_1, \dots, a_n$ , ideea generală a metodei este de a alege la fiecare pas acel element care face să crească cât mai mult valoarea acestei funcții. Din acest motiv metoda se mai numește și **a optimului local**.
- *Optimul global* se obține prin alegeri succesive, la fiecare pas, ale *optimului local*, ceea ce permite rezolvarea problemelor fără revenire la deciziile anterioare (așa cum se întâmplă la *metoda backtracking*).
- În general metoda Greedy oferă o soluție posibilă și nu întotdeauna soluția optimă. De aceea, dacă problema cere soluția optimă, algoritmul trebuie să fie însoțit și de justificarea faptului că soluția generată este optimă. Pentru aceasta, este frecvent întâlnit următorul procedeu:
  - se demonstrează prin *inducție matematică* faptul că pentru orice pas  $i \in \{0, 1, \dots, n\}$ , dacă  $B_i$  este soluția posibilă construită la pasul  $i$ , atunci există o soluție optimă  $B^*$  astfel încât  $B_i \subseteq B^*$ ;
  - se arată că pentru soluția finală,  $B_n$ , incluziunea  $B_n \subseteq B^*$  devine egalitate,  $B_n = B^*$ , deci  $B_n$  este soluție optimă.

*Exemplul 2.1.1.* Se dă o mulțime  $A = \{a_1, a_2, \dots, a_n\}$  cu  $a_i \in \mathbb{R}$ ,  $i = \overline{1, n}$ . Se cere să se determine o submulțime  $B \subseteq A$ , astfel încât  $\sum_{b \in B} b$  să fie maximă.

*Rezolvare.* Dacă  $B \subseteq A$  și  $b_0 \in B$ , cu  $b_0 \leq 0$ , atunci

$$\sum_{b \in B} b \leq \sum_{b \in B \setminus \{b_0\}} b.$$

Rezultă că putem înțelege prin **soluție posibilă** o submulțime  $B$  a lui  $A$  cu toate elementele strict pozitive.

Vom aplica metoda Greedy, în **varianta I**, în care

- funcția **ALEGE** furnizează  $x = a_i$ ;
- funcția **SOLUTIE\_POSIBILA** returnează 1 (adevărat) dacă  $x > 0$  și 0 (fals) în caz contrar.

□



**ALEGE** ( $A, i, n$ ):

$x \leftarrow a_i$ ;

**returnează**  $x$ ;

**SOLUTIE\_POSIBILA** ( $B, x$ ):

**if**  $x > 0$  **then**

**returnează** 1;

// adevărat

**else**

**returnează** 0;

// fals

## 2.2 Aplicații ale Inegalității rearanjamentelor

### 2.2.1 Inegalitatea rearanjamentelor

**Teorema 2.2.1 (Inegalitatea rearanjamentelor).** *Fie șirurile crescătoare de numere reale*

$$a_1 \leq a_2 \leq \dots \leq a_n \text{ și } b_1 \leq b_2 \leq \dots \leq b_n, \quad n \in \mathbb{N}^*.$$

*Atunci, pentru orice permutare  $p \in S_n$  ( $S_n =$  grupul permutărilor de ordin  $n$ ), avem*

$$\sum_{i=1}^n a_i \cdot b_{n+1-i} \leq \sum_{i=1}^n a_i \cdot b_{p(i)} \leq \sum_{i=1}^n a_i \cdot b_i. \quad (2.2.1)$$

*Demonstrație.* Pentru orice permutare  $p \in S_n$ , notăm

$$s(p) = \sum_{i=1}^n a_i \cdot b_{p(i)}.$$

Fie

$$M = \max_{p \in S_n} s(p). \quad (2.2.2)$$

Demonstrăm că pentru orice  $k \in \{0, 1, \dots, n\}$  există  $p \in S_n$  astfel încât

$$s(p) = M \text{ și } p(i) = i \quad \forall 1 \leq i \leq k, \quad (2.2.3)$$

prin inducție după  $k$ .

Pentru  $k = 0$  afirmația este evidentă, luând orice permutare  $p \in S_n$  astfel încât  $s(p) = M$ .

Presupunem (2.2.3) adevărată pentru  $k - 1$ , adică există  $p \in S_n$  astfel încât

$$s(p) = M \text{ și } p(i) = i \quad \forall 1 \leq i \leq k - 1$$

și o demonstrăm pentru  $k$  ( $k \in \{1, 2, \dots, n\}$ ).

Avem două cazuri.

Cazul 1)  $p(k) = k$ . Atunci  $p(i) = i \forall 1 \leq i \leq k$ .

Cazul 2)  $p(k) \neq k$ . Cum  $p$  este o permutare și  $p(i) = i \forall 1 \leq i \leq k-1$ , rezultă că  $p(k) > k$  și există  $j > k$  a.î.  $p(j) = k$ . Definim permutarea  $p' \in S_n$  prin

$$\begin{cases} p'(k) &= p(j), \\ p'(j) &= p(k), \\ p'(i) &= p(i), \forall i \in \{1, \dots, n\} \setminus \{k, j\}. \end{cases}$$

Avem

$$\begin{aligned} s(p') - s(p) &= \sum_{i=1}^n a_i \cdot b_{p'(i)} - \sum_{i=1}^n a_i \cdot b_{p(i)} \\ &= a_k \cdot b_{p'(k)} + a_j \cdot b_{p'(j)} - a_k \cdot b_{p(k)} - a_j \cdot b_{p(j)} \\ &= a_k \cdot b_{p(j)} + a_j \cdot b_{p(k)} - a_k \cdot b_{p(k)} - a_j \cdot b_{p(j)} \\ &= (a_j - a_k)(b_{p(k)} - b_{p(j)}) \\ &= \underbrace{(a_j - a_k)}_{\geq 0} \underbrace{(b_{p(k)} - b_k)}_{\geq 0} \geq 0 \end{aligned}$$

(deoarece  $j > k$ ,  $p(k) > k$ , iar șirurile  $(a_i)_{i=\overline{1,n}}$  și  $(b_i)_{i=\overline{1,n}}$  sunt crescătoare). Deci  $s(p') \geq s(p) = M$ .

Cum, conform (2.2.2), avem  $s(p') \leq M$ , rezultă că

$$s(p') = s(p) = M$$

(în plus,  $a_j = a_k$  sau  $b_{p(k)} = b_k$ ).

Evident,  $p'(i) = i \forall 1 \leq i \leq k$ , deci relația (2.2.3) este adevărată pentru  $k$ , ceea ce încheie demonstrația prin inducție a acestei relații.

Luând  $k = n$  în această relație rezultă că

$$s(e) = M, \tag{2.2.4}$$

unde  $e \in S_n$  este permutarea identică, definită prin  $e(i) = i \forall i \in \{1, \dots, n\}$ .

Fie  $p \in S_n$  o permutare arbitrară. Din (2.2.2) și (2.2.4) rezultă că  $s(p) \leq s(e)$ , adică

$$\sum_{i=1}^n a_i \cdot b_{p(i)} \leq \sum_{i=1}^n a_i \cdot b_i.$$

Aplicând această inegalitate pentru șirurile crescătoare

$$a_1 \leq a_2 \leq \dots \leq a_n \text{ și } -b_n \leq -b_{n-1} \leq \dots \leq -b_1$$

rezultă că

$$\sum_{i=1}^n a_i \cdot (-b_{p(i)}) \leq \sum_{i=1}^n a_i \cdot (-b_{n+1-i}),$$

adică

$$\sum_{i=1}^n a_i \cdot b_{n+1-i} \leq \sum_{i=1}^n a_i \cdot b_{p(i)}.$$

□

## 2.2.2 Produs scalar maxim/minim

Se consideră şirurile de numere reale

$$a_1, a_2, \dots, a_n \text{ şi } b_1, b_2, \dots, b_n, \quad n \in \mathbb{N}^*.$$

Se cere să se determine două permutări  $a_{q(1)}, a_{q(2)}, \dots, a_{q(n)}$  şi  $b_{p(1)}, b_{p(2)}, \dots, b_{p(n)}$  ale celor două şiruri,  $q, p \in S_n$  ( $S_n =$  grupul permutărilor de ordin  $n$ ), astfel

încât suma  $\sum_{i=1}^n a_{q(i)} \cdot b_{p(i)}$  să fie

- a) maximă;
- b) minimă.

*Observația 2.2.1.* Suma  $\sum_{i=1}^n a_{q(i)} \cdot b_{p(i)}$  reprezintă **produsul scalar** al vectorilor  $(a_{q(i)})_{i=\overline{1,n}}$  şi  $(b_{p(i)})_{i=\overline{1,n}}$ . Astfel problema anterioară cere determinarea unor permutări ale elementelor vectorilor  $(a_1, a_2, \dots, a_n)$  şi  $(b_1, b_2, \dots, b_n)$  astfel încât după permutare produsul lor scalar să fie maxim, respectiv minim.

### Rezolvarea problemei de maxim

*Algoritmul 2.2.1.* Conform Teoremei 2.2.1 deducem următoarea *strategie Greedy* în varianta I pentru rezolvarea problemei:

- Pentru obținerea celor  $n$  termeni ai sumei maxime, la fiecare pas  $i = \overline{1, n}$  luăm produsul dintre:
  - cel mai mic dintre termenii şirului  $(a_1, a_2, \dots, a_n)$  neales la pașii anteriori;
  - cel mai mic dintre termenii şirului  $(b_1, b_2, \dots, b_n)$  neales la pașii anteriori.

Descrierea în pseudocod a algoritmului are următoarea formă.

```

MAXIM1 ( $a, b, n, s$ ) :           //  $a = (a_1, \dots, a_n)$ ,  $b = (b_1, \dots, b_n)$ 
 $s \leftarrow 0$ ;                      //  $s = \text{suma maximă}$ 
for  $i = \overline{1, n}$  do                // pasul  $i$ 
     $k \leftarrow i$ ;                // calculăm termenul minim  $a_k$  din  $(a_i, \dots, a_n)$ 
     $m \leftarrow a[i]$ ;
    for  $j = \overline{i+1, n}$  do
        if  $a[j] < m$  then
             $k \leftarrow j$ ;
             $m \leftarrow a[j]$ ;
     $a[i] \leftrightarrow a[k]$ ;                // interschimbăm termenii  $a_i$  și  $a_k$ 
     $k \leftarrow i$ ;                // calculăm termenul minim  $b_k$  din  $(b_i, \dots, b_n)$ 
     $m \leftarrow b[i]$ ;
    for  $j = \overline{i+1, n}$  do
        if  $b[j] < m$  then
             $k \leftarrow j$ ;
             $m \leftarrow b[j]$ ;
     $b[i] \leftrightarrow b[k]$ ;                // interschimbăm termenii  $b_i$  și  $b_k$ 
     $s \leftarrow s + a[i] \cdot b[i]$ ;    // adunăm produsul termenilor minimi
                                        // la suma  $s$ 
AFISARE( $s, a, b, n$ );                // se afișează suma maximă și
                                        // permutările obținute

```

Funcția de afișare este

```

AFISARE ( $s, a, b, n$ ) :
    afișează  $s$ ;
    for  $i = \overline{1, n}$  do
        afișează  $a[i]$ ;
    for  $i = \overline{1, n}$  do
        afișează  $b[i]$ ;

```

*Observația 2.2.2.* Algoritmul necesită câte două comparații și câte cel mult patru atribuiri pentru fiecare pereche de indici  $(i, j)$  cu  $i \in \{1, 2, \dots, n\}$  și  $j \in \{i+1, i+2, \dots, n\}$ . Numărul acestor perechi este  $(n-1) + (n-2) + \dots + 2 + 1 = \frac{n(n-1)}{2}$ , deci algoritmul are complexitatea  $\Theta(n^2)$ .

*Algoritmul 2.2.2.* Conform Teoremei 2.2.1 obținem și următoarea *strategie Greedy în varianta a II-a* pentru rezolvarea problemei:

- ordonăm crescător elementele primului șir:  $a_1 \leq a_2 \leq \dots \leq a_n$ ;

- ordonăm crescător elementele celui de-al doilea șir:  $b_1 \leq b_2 \leq \dots \leq b_n$ ;
- suma maximă este  $s = \sum_{i=1}^n a_i \cdot b_i$ .

Descrierea în pseudocod a algoritmului are următoarea formă.

```

MAXIM2 ( $a, b, n, s$ ) :           //  $a = (a_1, \dots, a_n)$ ,  $b = (b_1, \dots, b_n)$ 
SORTARE ( $a, n$ );                // se sortează crescător vectorul  $a$ 
SORTARE ( $b, n$ );                // se sortează crescător vectorul  $b$ 
 $s \leftarrow 0$ ;                    //  $s$  = suma maximă
for  $i = \overline{1, n}$  do             // calculăm suma maximă  $s$ 
     $s \leftarrow s + a[i] \cdot b[i]$ ;
AFISARE ( $s, a, b, n$ );           // se afișează suma maximă și
                                   // permutările obținute

```

unde funcția de afișare este aceeași ca în Algoritmul 2.2.1.

*Observația 2.2.3.* Algoritmul anterior are complexitatea  $\Theta(n \log_2 n)$ , deoarece necesită sortarea celor doi vectori de dimensiune  $n$ . Rezultă că Algoritmul 2.2.2 este mai eficient decât Algoritmul 2.2.1.

## Rezolvarea problemei de minim

*Algoritmul 2.2.3.* Conform Teoremei 2.2.1 deducem următoarea *strategie Greedy* în varianta I pentru rezolvarea problemei de minim:

- Pentru obținerea celor  $n$  termeni ai sumei minime, la fiecare pas  $i = \overline{1, n}$  luăm produsul dintre:
  - cel mai mic dintre termenii șirului  $(a_1, a_2, \dots, a_n)$  neales la pașii anteriori;
  - cel mai mare dintre termenii șirului  $(b_1, b_2, \dots, b_n)$  neales la pașii anteriori.

Descrierea în pseudocod a algoritmului are următoarea formă.

```

MINIM1 ( $a, b, n, s$ ):           //  $a = (a_1, \dots, a_n)$ ,  $b = (b_1, \dots, b_n)$ 
 $s \leftarrow 0$ ;                      //  $s =$  suma minimă
for  $i = \overline{1, n}$  do                // pasul  $i$ 
     $k \leftarrow i$ ;                // calculăm termenul minim  $a_k$  din  $(a_i, \dots, a_n)$ 
     $m \leftarrow a[i]$ ;
    for  $j = \overline{i+1, n}$  do
        if  $a[j] < m$  then
             $k \leftarrow j$ ;
             $m \leftarrow a[j]$ ;
     $a[i] \leftrightarrow a[k]$ ;                // interschimbăm termenii  $a_i$  și  $a_k$ 
     $k \leftarrow i$ ;                // calculăm termenul maxim  $b_k$  din  $(b_i, \dots, b_n)$ 
     $m \leftarrow b[i]$ ;
    for  $j = \overline{i+1, n}$  do
        if  $b[j] > m$  then
             $k \leftarrow j$ ;
             $m \leftarrow b[j]$ ;
     $b[i] \leftrightarrow b[k]$ ;                // interschimbăm termenii  $b_i$  și  $b_k$ 
     $s \leftarrow s + a[i] \cdot b[i]$ ; // adunăm produsul termenilor calculați
                                     // la suma  $s$ 
AFISARE( $s, a, b, n$ );           // se afișează suma minimă și
                                     // permutările obținute

```

Funcția de afișare este aceeași ca în Algoritmul 2.2.1.

*Algoritmul 2.2.4.* Conform Teoremei 2.2.1 obținem și următoarea *strategie Greedy în varianta a II-a* pentru rezolvarea problemei de minim:

- ordonăm crescător elementele primului șir:  $a_1 \leq a_2 \leq \dots \leq a_n$ ;
- ordonăm descrescător elementele celui de-al doilea șir:  $b_1 \geq b_2 \geq \dots \geq b_n$ ;
- suma minimă este  $s = \sum_{i=1}^n a_i \cdot b_i$ .

Descrierea în pseudocod a algoritmului are următoarea formă.

```

MINIM2 ( $a, b, n, s$ ):           //  $a = (a_1, \dots, a_n)$ ,  $b = (b_1, \dots, b_n)$ 
SORTARE1 ( $a, n$ );               // se sortează crescător vectorul  $a$ 
SORTARE2 ( $b, n$ );               // se sortează descrescător vectorul  $b$ 
 $s \leftarrow 0$ ;                  //  $s$  = suma minimă
for  $i = \overline{1, n}$  do           // calculăm suma minimă  $s$ 
     $s \leftarrow s + a[i] \cdot b[i]$ ;
AFISARE ( $s, a, b, n$ );          // se afișează suma minimă și
                                   // permutările obținute

```

unde funcția de afișare este aceeași ca în Algoritmul 2.2.1.

*Observația 2.2.4.* Analog problemei de maxim, Algoritmul 2.2.3 are complexitatea  $\Theta(n^2)$ , iar Algoritmul 2.2.4 are complexitatea  $\Theta(n \log_2 n)$ , fiind astfel mai eficient decât Algoritmul 2.2.3.

### 2.2.3 Memorarea optimă a textelor pe benzi

Se dă o bandă magnetică suficient de lungă pentru a memora  $n$  texte (sau fișiere)

$$T_1, T_2, \dots, T_n$$

de lungimi date (de exemplu, în octeți)

$$L_1, L_2, \dots, \text{ respectiv } L_n.$$

La citirea unui text de pe bandă, trebuie citite și textele aflate înaintea lui.

Presupunând că frecvența de citire a celor  $n$  texte este aceeași, se cere să se determine o ordine de poziționare (memorare) optimă a acestora pe bandă, adică o poziționare astfel încât timpul mediu de citire să fie minim.

#### Modelarea problemei

- Evident, orice poziționare a celor  $n$  texte pe bandă este o permutare a vectorului  $(T_1, T_2, \dots, T_n)$ , adică are forma

$$(T_{p(1)}, T_{p(2)}, \dots, T_{p(n)}),$$

unde  $p \in S_n$  este o permutare de ordin  $n$  (pentru orice  $i \in \{1, \dots, n\}$ , pe poziția  $i$  pe bandă se memorează textul  $T_{p(i)}$ ).

- Evident, timpul de citire doar a unui text  $T_k$  este direct proporțional cu lungimea lui, deci putem considera că acest timp este egal cu lungimea  $L_k$  a textului.

- Pentru orice  $k \in \{1, \dots, n\}$ , citirea textului  $T_{p(k)}$  necesită timpul

$$t_k = \sum_{i=1}^k L_{p(i)},$$

deoarece la timpul de citire efectivă a textului  $T_{p(k)}$  trebuie adăugați și timpii de citire a textelor precedente  $T_{p(1)}, T_{p(2)}, \dots, T_{p(k-1)}$ .

- Frecvența de citire a celor  $n$  texte fiind aceeași, rezultă că *timpul mediu de citire* pentru o poziționare  $p \in S_n$  este

$$\begin{aligned} t(p) &= \frac{1}{n} \sum_{k=1}^n t_k \\ &= \frac{1}{n} \sum_{k=1}^n \sum_{i=1}^k L_{p(i)} \\ &= \frac{1}{n} \sum_{i=1}^n \sum_{k=i}^n L_{p(i)} \\ &= \frac{1}{n} \sum_{i=1}^n \left( L_{p(i)} \sum_{k=i}^n 1 \right) \\ &= \frac{1}{n} \sum_{i=1}^n (n - i + 1) \cdot L_{p(i)}. \end{aligned}$$

**Propoziția 2.2.1.** Dacă  $L_1 \leq L_2 \leq \dots \leq L_n$ , atunci

$$\min_{p \in S_n} t(p) = t(e),$$

adică poziționarea corespunzătoare permutării identice este optimă.

*Demonstrație.* Aplicând Teorema 2.2.1 pentru șirurile crescătoare

$$-\frac{n}{n} < -\frac{n-1}{n} < \dots < -\frac{1}{n} \text{ și } L_1 \leq L_2 \leq \dots \leq L_n,$$

rezultă că pentru orice permutare  $p \in S_n$  avem

$$\sum_{i=1}^n \left( -\frac{n-i+1}{n} \right) \cdot L_{p(i)} \leq \sum_{i=1}^n \left( -\frac{n-i+1}{n} \right) \cdot L_i,$$

deci

$$\frac{1}{n} \sum_{i=1}^n (n-i+1) \cdot L_{p(i)} \geq \frac{1}{n} \sum_{i=1}^n (n-i+1) \cdot L_i,$$

adică  $t(p) \geq t(e)$ . Rezultă că  $\min_{p \in S_n} t(p) = t(e)$ . □



*Observația 2.2.5.* Mai mult, deoarece șirul

$$-\frac{n}{n} < -\frac{n-1}{n} < \dots < -\frac{1}{n}$$

este strict crescător, conform demonstrației Teoremei 2.2.1 rezultă că orice poziționare optimă presupune memorarea textelor pe bandă în ordinea crescătoare a lungimilor lor.

*Observația 2.2.6.* Conform propoziției anterioare deducem următoarea *strategie Greedy* pentru rezolvarea problemei:

*Varianta I:* La fiecare pas  $i = \overline{1, n}$  se poziționează pe bandă pe poziția curentă,  $i$ , textul de lungime minimă dintre cele nepoziționate la pașii anteriori;

*Varianta II:* Se sortează textele în ordinea crescătoare a lungimilor lor și se poziționează pe bandă în această ordine.

Analog problemelor de la secțiunea anterioară, Varianta I are complexitatea  $\Theta(n^2)$ , iar Varianta II are complexitatea  $\Theta(n \log_2 n)$  (fiind astfel mai eficientă decât Varianta I).

## 2.3 Problema rucsacului, varianta continuă

**Problema rucsacului (Knapsack)** este următoarea:

Se consideră un rucsac în care se poate încărcă greutatea maximă  $G$ , unde  $G > 0$ , și  $n$  obiecte  $O_1, \dots, O_n$ ,  $n \in \mathbb{N}^*$ . Pentru fiecare obiect  $O_i$ ,  $i \in \{1, \dots, n\}$ , se cunoaște greutatea sa,  $g_i$ , unde  $g_i > 0$ , și câștigul obținut la transportul său în întregime,  $c_i$ , unde  $c_i > 0$ .

Se cere să se determine o modalitate de încărcare a rucsacului cu obiecte astfel încât câștigul total al obiectelor încărcate să fie maxim.

În **varianta continuă (fracționară)** a problemei, pentru fiecare obiect  $O_i$  poate fi încărcată orice parte (fracțiune)  $x_i \in [0, 1]$  din el, câștigul obținut fiind proporțional cu partea încărcată, adică este egal cu  $x_i c_i$ .

### Modelarea problemei

- O *soluție (soluție posibilă)* a problemei este orice vector  $x = (x_1, \dots, x_n)$  astfel încât

$$\begin{cases} x_i \in [0, 1], \forall i \in \{1, \dots, n\}, \\ \sum_{i=1}^n x_i g_i \leq G, \end{cases}$$

ultima inegalitate exprimând faptul că greutatea totală încărcată în rucsac nu trebuie să depășească greutatea maximă.

- Câștigul (*total*) corespunzător soluției  $x = (x_1, \dots, x_n)$  este

$$f(x) = \sum_{i=1}^n x_i c_i.$$

- O soluție optimă a problemei este orice soluție  $x^* = (x_1^*, \dots, x_n^*)$  astfel încât

$$f(x^*) = \max\{f(x) \mid x = \text{soluție a problemei}\}.$$

- Dacă suma greutatea tuturor obiectelor este mai mică sau egală cu greutatea maximă a rucsacului, adică  $\sum_{i=1}^n g_i \leq G$ , atunci problema este trivială, soluția  $x^* = (1, 1, \dots, 1)$ , corespunzătoare încărcării integrale a tuturor celor  $n$  obiecte în rucsac, fiind evident singura soluție optimă. Astfel în continuare putem presupune că

$$\sum_{i=1}^n g_i > G. \quad (2.3.1)$$

- Pentru orice soluție optimă  $x^* = (x_1^*, \dots, x_n^*)$  avem

$$\sum_{i=1}^n x_i^* g_i = G \quad (2.3.2)$$

(adică rucsacul trebuie încărcat complet). Demonstrăm această afirmație prin reducere la absurd. Într-adevăr, dacă  $\sum_{i=1}^n x_i^* g_i < G$ , cum  $\sum_{i=1}^n g_i > G$  rezultă că există un indice  $k \in \{1, \dots, n\}$  a.î.  $x_k^* < 1$ . Considerând vectorul  $x' = (x'_1, \dots, x'_n)$  definit prin

$$x'_i = \begin{cases} x_i^*, & \text{dacă } i \neq k, \\ x_i^* + u, & \text{dacă } i = k, \end{cases}$$

unde

$$u = \min \left\{ 1 - x_k^*, \frac{1}{g_k} \left( G - \sum_{i=1}^n x_i^* g_i \right) \right\},$$

avem  $u > 0$ ,  $x'_k \leq 1$ ,  $\sum_{i=1}^n x'_i g_i = \sum_{i=1}^n x_i^* g_i + u g_k \leq \sum_{i=1}^n x_i^* g_i + G - \sum_{i=1}^n x_i^* g_i = G$  și  $f(x') = \sum_{i=1}^n x'_i c_i = \sum_{i=1}^n x_i^* c_i + u c_k = f(x^*) + u c_k$ , deci  $x'$  este o soluție a problemei și  $f(x') > f(x^*)$ , ceea ce contrazice optimalitatea soluției  $x^*$ .

Prezentăm în continuare un *algorithm Greedy* pentru rezolvarea problemei.

*Algoritmul 2.3.1.* Vom utiliza următoarea *strategie Greedy*:

- Ordonăm obiectele descrescător după câștigul lor unitar:

$$\frac{c_1}{g_1} \geq \frac{c_2}{g_2} \geq \dots \geq \frac{c_n}{g_n}. \quad (2.3.3)$$

- Încărcăm obiectele în rucsac, în această ordine, cât timp nu se depășește greutatea maximă  $G$ . Încărcarea obiectelor se face în întregime, cât timp este posibil; în acest fel doar ultimul obiect adăugat poate fi încărcat parțial.

Descrierea în pseudocod a algoritmului are următoarea formă.

```

RUCSAC ( $G, n, g, c, x, C$ ):      //  $g = (g_1, \dots, g_n)$ ,  $c = (c_1, \dots, c_n)$ 
                                   //  $C =$  câștigul total
SORTARE( $g, c, n$ );           // se sortează obiectele descrescător
                                   // după câștigul lor unitar
 $R \leftarrow G$ ;                //  $R =$  greutatea disponibilă pentru rucsac
 $C \leftarrow 0$ ;
 $i \leftarrow 1$ ;
while  $R > 0$  do                // rucsacul nu este plin
    if  $g[i] \leq R$  then
        // obiectul curent încape în întregime, deci
        // se adaugă în rucsac
         $x[i] \leftarrow 1$ ;
         $C \leftarrow C + c[i]$ ;
         $R \leftarrow R - g[i]$ ;      // actualizăm greutatea disponibilă
         $i \leftarrow i + 1$ ;        // trecem la obiectul următor
    else
        // obiectul curent nu încape în întregime, deci
        // se adaugă exact acea parte din el care
        // umple rucsacul și încărcarea se încheie
         $x[i] \leftarrow \frac{R}{g[i]}$ ;
         $C \leftarrow C + x[i]c[i]$ ;
         $R \leftarrow 0$ ;
        for  $j = i + 1, n$  do  $x[j] \leftarrow 0$ ;
AFISARE( $C, x, n$ );           // se afișează câștigul total maxim  $C$ 
                                   // și soluția optimă  $x = (x_1, \dots, x_n)$ 

```

**Teorema 2.3.1 (de corectitudine a Algoritmului 2.3.1).** *În contextul Algoritmului 2.3.1, vectorul  $x = (x_1, \dots, x_n)$  calculat de algoritm este o soluție optimă a problemei rucsacului.*

*Demonstrație.* Evident, vectorul  $x = (x_1, \dots, x_n)$  calculat de algoritm verifică relațiile

$$\begin{cases} x_i \in [0, 1], \forall i \in \{1, \dots, n\}, \\ \sum_{i=1}^n x_i g_i = G, \end{cases}$$

deci este o soluție a problemei. Rămâne să demonstrăm optimalitatea acestei soluții.

Demonstrăm prin inducție după  $k \in \{0, 1, \dots, n\}$  că există o soluție optimă  $x^* = (x_1^*, \dots, x_n^*)$  a problemei pentru care

$$x_i = x_i^*, \forall i \text{ a.î. } 1 \leq i \leq k. \quad (2.3.4)$$

Pentru  $k = 0$  afirmația este evidentă, luând  $x^*$  orice soluție optimă a problemei.

Presupunem (2.3.4) adevărată pentru  $k - 1$ , adică există o soluție optimă  $x^* = (x_1^*, \dots, x_n^*)$  a problemei pentru care

$$x_i = x_i^*, \forall i \text{ a.î. } 1 \leq i \leq k - 1$$

și o demonstrăm pentru  $k$  ( $k \in \{1, 2, \dots, n\}$ ).

Cum

$$\sum_{i=1}^{k-1} x_i g_i + x_k^* g_k = \sum_{i=1}^{k-1} x_i^* g_i + x_k^* g_k \leq \sum_{i=1}^n x_i^* g_i = G,$$

din descrierea algoritmului (alegerea maximală a lui  $x_k$ ) rezultă că

$$x_k \geq x_k^*.$$

Avem două cazuri.

Cazul 1)  $x_k = x_k^*$ . Atunci  $x_i = x_i^*, \forall i \in \{1, \dots, k\}$ , deci (2.3.4) este adevărată pentru  $k$ .

Cazul 2)  $x_k > x_k^*$ . În acest caz avem  $k < n$ , deoarece dacă, prin reducere la absurd, am avea  $k = n$ , atunci ar rezulta că

$$f(x) = \sum_{i=1}^n x_i c_i = \sum_{i=1}^{n-1} x_i^* c_i + x_n c_n > \sum_{i=1}^{n-1} x_i^* c_i + x_n^* c_n = f(x^*),$$

ceea ce contrazice optimalitatea soluției  $x^*$ .

Definim vectorul  $x^{**} = (x_1^{**}, \dots, x_n^{**})$  prin

$$x_i^{**} = \begin{cases} x_i, & \text{dacă } 1 \leq i \leq k, \\ \alpha^* x_i^*, & \text{dacă } k+1 \leq i \leq n, \end{cases} \quad (2.3.5)$$

unde  $\alpha^* \in [0, 1)$  este o soluție a ecuației

$$h(\alpha) = 0, \text{ unde } h(\alpha) = \sum_{i=1}^k x_i g_i + \alpha \sum_{i=k+1}^n x_i^* g_i - G. \quad (2.3.6)$$

O astfel de soluție există, deoarece

$$\begin{aligned} h(0) &= \sum_{i=1}^k x_i g_i - G \leq \sum_{i=1}^n x_i g_i - G = G - G = 0, \\ h(1) &= \sum_{i=1}^k x_i g_i + \sum_{i=k+1}^n x_i^* g_i - G = \sum_{i=1}^{k-1} x_i^* g_i + x_k g_k + \sum_{i=k+1}^n x_i^* g_i - G \\ &= \sum_{i=1}^n x_i^* g_i - x_k^* g_k + x_k g_k - G = G + g_k(x_k - x_k^*) - G \\ &= g_k(x_k - x_k^*) > 0, \end{aligned}$$

iar  $h$  este o funcție continuă pe intervalul  $[0, 1]$ .

Conform (2.3.5) și (2.3.6) rezultă că  $x_i^{**} \in [0, 1]$ ,  $\forall i \in \{1, \dots, n\}$  și

$$\sum_{i=1}^n x_i^{**} g_i = \sum_{i=1}^k x_i g_i + \alpha^* \sum_{i=k+1}^n x_i^* g_i = h(\alpha^*) + G = 0 + G = G,$$

deci vectorul  $x^{**} = (x_1^{**}, \dots, x_n^{**})$  este o soluție a problemei.

Avem

$$\begin{aligned} f(x^{**}) - f(x^*) &= \sum_{i=1}^{k-1} x_i^* c_i + x_k c_k + \alpha^* \sum_{i=k+1}^n x_i^* c_i - \sum_{i=1}^n x_i^* c_i \\ &= x_k c_k + \alpha^* \sum_{i=k+1}^n x_i^* c_i - \sum_{i=k}^n x_i^* c_i \\ &= x_k c_k + \alpha^* \sum_{i=k+1}^n x_i^* c_i - x_k^* c_k - \sum_{i=k+1}^n x_i^* c_i \\ &= c_k(x_k - x_k^*) - (1 - \alpha^*) \sum_{i=k+1}^n x_i^* c_i \\ &= \frac{c_k}{g_k} \cdot (x_k g_k - x_k^* g_k) - (1 - \alpha^*) \sum_{i=k+1}^n \frac{c_i}{g_i} \cdot x_i^* g_i. \end{aligned} \quad (2.3.7)$$

Conform (2.3.3) rezultă că

$$\frac{c_i}{g_i} \leq \frac{c_k}{g_k}, \forall i \in \{k+1, \dots, n\}. \quad (2.3.8)$$

Din (2.3.7) și (2.3.8) obținem că

$$\begin{aligned} f(x^{**}) - f(x^*) &\geq \frac{c_k}{g_k} \cdot \left[ (x_k g_k - x_k^* g_k) - (1 - \alpha^*) \sum_{i=k+1}^n x_i^* g_i \right] \\ &= \frac{c_k}{g_k} \cdot \left( x_k g_k - x_k^* g_k - \sum_{i=k+1}^n x_i^* g_i + \alpha^* \sum_{i=k+1}^n x_i^* g_i \right) \\ &= \frac{c_k}{g_k} \cdot \left[ x_k g_k - \left( \sum_{i=1}^n x_i^* g_i - \sum_{i=1}^{k-1} x_i^* g_i \right) + \alpha^* \sum_{i=k+1}^n x_i^* g_i \right] \\ &= \frac{c_k}{g_k} \cdot \left( x_k g_k - G + \sum_{i=1}^{k-1} x_i g_i + \alpha^* \sum_{i=k+1}^n x_i^* g_i \right) \\ &= \frac{c_k}{g_k} \cdot \left( \sum_{i=1}^k x_i g_i + \alpha^* \sum_{i=k+1}^n x_i^* g_i - G \right), \end{aligned}$$

și conform (2.3.6) rezultă că

$$f(x^{**}) - f(x^*) \geq \frac{c_k}{g_k} \cdot h(\alpha^*) = 0,$$

deci

$$f(x^{**}) \geq f(x^*).$$

Cum  $x^*$  este soluție optimă, rezultă că și  $x^{**}$  este soluție optimă (și, în plus,  $f(x^{**}) = f(x^*)$ ). Conform (2.3.5) avem

$$x_i = x_i^{**}, \forall i \in \{1, \dots, k\},$$

deci relația (2.3.4) este adevărată pentru  $k$ , ceea ce încheie demonstrația prin inducție a acestei relații.

Luând  $k = n$  în această relație rezultă că există o soluție optimă  $x^* = (x_1^*, \dots, x_n^*)$  pentru care

$$x_i = x_i^*, \forall i \in \{1, \dots, n\},$$

deci  $x = x^*$  și astfel  $x$  este o soluție optimă a problemei. □

*Exemplul 2.3.1.* Considerăm un rucsac în care se poate încărca o greutate maximă  $G = 40$ , din  $n = 10$  obiecte ce au greutatea și câștigurile date în următorul tabel:

Obiect	$O_1$	$O_2$	$O_3$	$O_4$	$O_5$	$O_6$	$O_7$	$O_8$	$O_9$	$O_{10}$
Greutate $g_i$	10	7	10	5	6	10	8	15	3	12
Câștig $c_i$	27	9	40	20	11	20	50	22	4	33

Ordinea descrescătoare a obiectelor după câștigul unitar  $c_i/g_i$  este evidențiată în următorul tabel:

Obiect	$O_7$	$O_3$	$O_4$	$O_{10}$	$O_1$	$O_6$	$O_5$	$O_8$	$O_9$	$O_2$
Câștig $c_i$	50	40	20	33	27	20	11	22	4	9
Greutate $g_i$	8	10	5	12	10	10	6	15	3	7

Aplicarea strategiei Greedy (algoritmul de mai sus) conduce la soluția optimă

$$x = (1, 1, 1, 1, 5/10, 0, 0, 0, 0, 0),$$

adică umplem rucsacul încărcând, în ordine, obiectele:

- $O_7$ , după care greutatea disponibilă devine  $R = 40 - 8 = 32$ ,
- $O_3$ , după care  $R = 32 - 10 = 22$ ,
- $O_4$ , după care  $R = 22 - 5 = 17$ ,
- $O_{10}$  după care  $R = 17 - 12 = 5$ ,
- $5/10$  din  $O_1$ , după care  $R = 5 - 5 = 0$ .

Câștigul (total) obținut este

$$f(x) = 50 + 40 + 20 + 33 + \frac{5}{10} \cdot 27 = 156,5.$$

*Observația 2.3.1.* Algoritmul 2.3.1 are complexitatea  $\mathcal{O}(n \log_2 n)$ , deoarece este necesară sortarea obiectelor descrescător după câștigul unitar iar blocul ”**while**” se execută de cel mult  $n$  ori (câte o dată pentru fiecare obiect) și necesită de fiecare dată o comparație și 3 operații aritmetice.

*Observația 2.3.2.* În **varianta discretă a problemei rucsacului**, fiecare obiect  $O_i$  poate fi încărcat doar în întregime. În această variantă, **soluția produsă de strategia Greedy (de mai sus) nu este neapărat optimă!**

De exemplu, pentru datele din exemplul anterior, aplicarea strategiei Greedy conduce la soluția

$$x = (1, 1, 1, 1, 0, 0, 0, 0, 1, 0),$$

adică încărcăm în rucsac, în ordine, obiectele:

- $O_7$ , după care greutatea disponibilă devine  $R = 40 - 8 = 32$ ,
- $O_3$ , după care  $R = 32 - 10 = 22$ ,
- $O_4$ , după care  $R = 22 - 5 = 17$ ,
- $O_{10}$  după care  $R = 17 - 12 = 5$ ,
- $O_9$ , după care  $R = 5 - 3 = 2$  și nu mai există niciun obiect care să mai încapă în rucsac, deci încărcarea se încheie.

Câștigul (total) obținut este

$$f(x) = 50 + 40 + 20 + 33 + 4 = 147.$$

Soluția obținută nu este optimă, o soluție mai bună fiind

$$x' = (1, 1, 1, 0, 1, 0, 1, 0, 0, 0),$$

corespunzătoare încărcării obiectelor  $O_7$ ,  $O_3$ ,  $O_4$ ,  $O_1$  și  $O_5$ , având greutatea totală  $8 + 10 + 5 + 10 + 6 = 39$  (deci rucsacul nu este plin) și câștigul (total)

$$f(x') = 50 + 40 + 20 + 27 + 11 = 148.$$

## 2.4 Problema planificării spectacolelor

**Problema planificării spectacolelor** este următoarea:

Se consideră  $n$  spectacole  $S_1, \dots, S_n$ ,  $n \in \mathbb{N}^*$ . Pentru fiecare spectacol  $S_i$ ,  $i \in \{1, \dots, n\}$ , se cunoaște intervalul orar  $I_i = [a_i, b_i]$  de desfășurare, unde  $a_i < b_i$ .

O persoană dorește să vizioneze cât mai multe dintre aceste  $n$  spectacole. Fiecare spectacol trebuie vizionat integral, nu pot fi vizionate simultan mai multe spectacole, iar timpii necesari deplasării de la un spectacol la altul sunt nesemnificativi (egali cu zero).

Se cere să se selecteze un număr cât mai mare de spectacole ce pot fi vizionate de o singură persoană, cu respectarea cerințelor de mai sus.

### Modelarea problemei

- O *soluție* (*soluție posibilă*) a problemei este orice submulțime  $P \subseteq \{I_1, \dots, I_n\}$  astfel încât

$$I_i \cap I_j = \emptyset, \forall I_i, I_j \in P, i \neq j$$

(adică orice submulțime de intervale disjuncte două câte două).



- O *soluție optimă* a problemei este orice soluție  $P^* \subseteq \{I_1, \dots, I_n\}$  astfel încât

$$\text{card}(P^*) = \max\{\text{card}(P) \mid P = \text{soluție a problemei}\}.$$

Prezentăm în continuare doi *algoritmi Greedy* pentru rezolvarea problemei.

*Algoritmul 2.4.1.* Vom utiliza următoarea *strategie Greedy*:

- Ordonăm spectacolele crescător după timpul lor de încheiere:

$$b_1 \leq b_2 \leq \dots \leq b_n. \quad (2.4.1)$$

- Parcurgem spectacolele, în această ordine, și:
  - selectăm primul spectacol;
  - de fiecare dată, spectacolul curent,  $S_i$ , se selectează doar dacă nu se suprapune cu niciunul dintre spectacolele selectate anterior, adică dacă timpul său de începere este mai mare decât timpul de încheiere al ultimului spectacol  $S_j$  selectat:

$$a_i > b_j.$$

Pentru memorarea soluției utilizăm un vector caracteristic  $c = (c_1, \dots, c_n)$ , cu semnificația

$$c_i = \begin{cases} 1, & \text{dacă intervalul } I_i \text{ a fost selectat,} \\ 0, & \text{în caz contrar.} \end{cases}$$

Descrierea în pseudocod a algoritmului are următoarea formă.

```

SPECTACOLE1 ( $a, b, n, c, m$ ): //  $a = (a_1, \dots, a_n)$ ,  $b = (b_1, \dots, b_n)$ 
    //  $c = (c_1, \dots, c_n)$ ,  $m = \text{numărul de spectacole selectate}$ 
SORTARE( $a, b, n$ ); // se sortează spectacolele crescător
    // după timpul lor de încheiere  $b_i$ 
 $m \leftarrow 0$ ; // inițializări
for  $i = \overline{1, n}$  do  $c[i] \leftarrow 0$ ;
 $t \leftarrow a[1] - 1$ ; //  $t = \text{timpul de încheiere al ultimului}$ 
    // spectacol selectat
for  $i = \overline{1, n}$  do // parcurgem spectacolele
    if  $a[i] > t$  then
         $c[i] \leftarrow 1$ ; // selectăm intervalul (spectacolul) curent
         $m \leftarrow m + 1$ ;
         $t \leftarrow b[i]$ ; // actualizăm  $t$ 
AFISARE( $m, c, n$ ); // se afișează numărul și
    // submulțimea intervalelor (spectacolelor) selectate
  
```

Funcția de afișare este

**AFISARE** ( $m, c, n$ ) :

**afișează**  $m$ ;

**for**  $i = \overline{1, n}$  **do**

**if**  $c[i] = 1$  **then** **afișează**  $[a[i], b[i]]$ ;

**Teorema 2.4.1** (de corectitudine a Algoritmului 2.4.1). *În contextul Algoritmului 2.4.1, submulțimea intervalelor (spectacolelor) selectate de algoritm este o soluție optimă a problemei planificării spectacolelor.*

*Demonstrație.* Fie  $P = \{I'_1, \dots, I'_m\}$  submulțimea de intervale calculată de algoritm, unde

$$I'_1 = [a'_1, b'_1], I'_2 = [a'_2, b'_2], \dots, I'_m = [a'_m, b'_m]$$

sunt intervalele selectate, în această ordine, de algoritm.

Evident,  $m \geq 1$  (după sortare, primul interval este întotdeauna selectat).

Din descrierea algoritmului (alegerea intervalului curent  $I'_i$ ) rezultă că

$$a'_i > b'_{i-1}, \forall i \in \{2, \dots, m\},$$

deci

$$a'_1 < b'_1 < a'_2 < b'_2 < \dots < a'_m < b'_m.$$

Rezultă că

$$I'_i \cap I'_j = \emptyset, \forall i, j \in \{1, \dots, m\}, i \neq j,$$

deci submulțimea  $P = \{I'_1, \dots, I'_m\}$  a intervalelor selectate de algoritm este o soluție a problemei. Rămâne să demonstrăm optimalitatea acestei soluții.

Demonstrăm prin inducție după  $k \in \{0, 1, \dots, m\}$  că există o soluție optimă  $P^* = \{I_1^*, \dots, I_p^*\}$  a problemei,  $p \in \mathbb{N}^*$ , cu

$$I_1^* = [a_1^*, b_1^*], I_2^* = [a_2^*, b_2^*], \dots, I_p^* = [a_p^*, b_p^*], \quad b_1^* < b_2^* < \dots < b_p^*, \quad (2.4.2)$$

pentru care

$$I'_i = I_i^*, \forall i \text{ a.î. } 1 \leq i \leq k. \quad (2.4.3)$$

Pentru  $k = 0$  afirmația este evidentă, luând  $P^*$  orice soluție optimă a problemei (și sortând intervalele componente  $I_i^*$  crescător după extremitățile  $b_i^*$ ).

Presupunem (2.4.3) adevărată pentru  $k - 1$ , adică există o soluție optimă  $P^* = \{I_1^*, \dots, I_p^*\}$  a problemei, ce verifică (2.4.2), pentru care

$$I'_i = I_i^*, \forall i \text{ a.î. } 1 \leq i \leq k - 1. \quad (2.4.4)$$

și o demonstrăm pentru  $k$  ( $k \in \{1, 2, \dots, m\}$ ).

Din optimalitatea soluției  $P^* = \{I_1^*, \dots, I_p^*\}$  rezultă că  $p \geq m$ , deci

$$p \geq m \geq k.$$

Avem două cazuri.

Cazul 1)  $I'_k = I_k^*$ . Atunci  $I'_i = I_i^*$ ,  $\forall i \in \{1, \dots, k\}$ , deci (2.4.3) este adevărată pentru  $k$ .

Cazul 2)  $I'_k \neq I_k^*$ , adică  $[a'_k, b'_k] \neq [a_k^*, b_k^*]$ . În acest caz, pentru  $k \geq 2$  avem  $a_k^* > b_{k-1}^*$  (deoarece  $I_k^* \cap I_{k-1}^* = \emptyset$  și  $b_k^* > b_{k-1}^*$ ) și  $b'_{k-1} = b_{k-1}^*$  (deoarece  $I'_{k-1} = I_{k-1}^*$ ), deci

$$a_k^* > b'_{k-1}.$$

Atunci, din descrierea algoritmului, deoarece  $I'_k = [a'_k, b'_k]$  este primul interval selectat după intervalul  $I'_{k-1} = [a'_{k-1}, b'_{k-1}]$ , rezultă că

$$b'_k \leq b_k^*. \quad (2.4.5)$$

Din descrierea algoritmului, această inegalitate este valabilă și pentru  $k = 1$ , deoarece  $I'_1 = [a'_1, b'_1]$  este primul interval selectat.

Definim submulțimea de intervale  $P^{**} = \{I_1^{**}, \dots, I_p^{**}\}$  prin

$$I_i^{**} = [a_i^{**}, b_i^{**}] = \begin{cases} I_i^*, & \text{dacă } i \neq k, \\ I'_i, & \text{dacă } i = k. \end{cases} \quad (2.4.6)$$

Deoarece  $P^* = \{I_1^*, \dots, I_p^*\}$  este soluție a problemei și verifică (2.4.2), rezultă că

$$\begin{aligned} a_1^* < b_1^* < a_2^* < b_2^* < \dots < a_{k-1}^* < b_{k-1}^* < a_k^* < b_k^* < \\ < a_{k+1}^* < b_{k+1}^* < \dots < a_p^* < b_p^*. \end{aligned} \quad (2.4.7)$$

Pentru  $k \geq 2$  avem  $a'_k > b'_{k-1}$  (din descrierea algoritmului) și  $b'_{k-1} = b_{k-1}^*$  (deoarece  $I'_{k-1} = I_{k-1}^*$ ), deci

$$a'_k > b_{k-1}^*. \quad (2.4.8)$$

Din (2.4.7), (2.4.8) și (2.4.5) rezultă că

$$\begin{aligned} a_1^* < b_1^* < a_2^* < b_2^* < \dots < a_{k-1}^* < b_{k-1}^* < a'_k < b'_k < \\ < a_{k+1}^* < b_{k+1}^* < \dots < a_p^* < b_p^* \end{aligned}$$

(inegalitate valabilă și pentru  $k = 1$ ), deci submulțimea  $P^{**} = \{I_1^{**}, \dots, I_p^{**}\}$ , definită de (2.4.6), este o soluție a problemei și

$$b_1^{**} < b_2^{**} < \dots < b_p^{**}.$$

Cum

$$\text{card}(P^{**}) = p = \text{card}(P^*)$$

și  $P^*$  este soluție optimă, rezultă că și  $P^{**}$  este soluție optimă.

Conform (2.4.6) și (2.4.4) avem

$$I'_i = I_i^{**}, \forall i \in \{1, \dots, k\},$$

deci relația (2.4.3) este adevărată pentru  $k$ , ceea ce încheie demonstrația prin inducție a acestei relații.

Luând  $k = m$  în această relație rezultă că există o soluție optimă  $P^* = \{I_1^*, \dots, I_p^*\}$  pentru care

$$I'_i = I_i^*, \forall i \in \{1, \dots, m\}.$$

Demonstrăm că  $p = m$  prin reducere la absurd. Într-adevăr, dacă  $p > m$  atunci ar exista intervalul  $I_{m+1}^* = [a_{m+1}^*, b_{m+1}^*]$  astfel încât

$$a_{m+1}^* > b_m^* = b'_m$$

ceea ce ar contrazice faptul că algoritmul se încheie cu selectarea intervalului  $I'_m = [a'_m, b'_m]$ .

Astfel  $p = m$ , deci

$$P = \{I'_1, \dots, I'_m\} = \{I_1^*, \dots, I_p^*\} = P^*$$

și astfel submulțimea  $P$  este o soluție optimă a problemei.

□

*Exemplul 2.4.1.* Considerăm  $n = 14$  spectacole ce au timpii de începere și de încheiere dați în următorul tabel (în ordinea crescătoare a timpilor de începere  $a_i$ ):

Spectacol	$S_1$	$S_2$	$S_3$	$S_4$	$S_5$	$S_6$	$S_7$
Timp de începere $a_i$	8:00	8:10	8:15	8:50	9:10	9:20	9:20
Timp de încheiere $b_i$	9:10	9:00	9:00	10:20	10:40	10:30	11:00

Spectacol	$S_8$	$S_9$	$S_{10}$	$S_{11}$	$S_{12}$	$S_{13}$	$S_{14}$
Timp de începere $a_i$	10:45	11:00	12:00	12:10	12:30	13:00	13:40
Timp de încheiere $b_i$	12:00	12:30	13:30	14:00	13:50	14:30	15:00

Ordonarea spectacolele crescător după timpul lor de încheiere  $b_i$  este evidențiată în următorul tabel:

Spectacol	$S_2$	$S_3$	$S_1$	$S_4$	$S_6$	$S_5$	$S_7$
Timp de începere $a_i$	8:10	8:15	8:00	8:50	9:20	9:10	9:20
Timp de încheiere $b_i$	9:00	9:00	9:10	10:20	10:30	10:40	11:00

Spectacol	$S_8$	$S_9$	$S_{10}$	$S_{12}$	$S_{11}$	$S_{13}$	$S_{14}$
Timp de începere $a_i$	10:45	11:00	12:00	12:30	12:10	13:00	13:40
Timp de încheiere $b_i$	12:00	12:30	13:30	13:50	14:00	14:30	15:00

Aplicarea strategiei Greedy din algoritmul de mai sus conduce la soluția optimă dată de selectarea (vizionarea), în ordine, a spectacolelor:

- $S_2$  (primul, în ordinea impusă),
- $S_6$  (primul situat după  $S_2$  și care are timpul de începere mai mare decât timpul de încheiere al lui  $S_2$ ),
- $S_8$  (primul situat după  $S_6$  și care are timpul de începere mai mare decât timpul de încheiere al lui  $S_6$ ),
- $S_{12}$  (primul situat după  $S_8$  și care are timpul de începere mai mare decât timpul de încheiere al lui  $S_8$ ), după care nu mai urmează niciun spectacol care să înceapă după încheierea lui  $S_{12}$ , deci selectarea se termină.

Numărul maxim de spectacole ce pot fi vizionate este deci egal cu 4.

*Observația 2.4.1.* Algoritmul 2.4.1 are complexitatea  $\mathcal{O}(n \log_2 n)$ , deoarece este necesară sortarea spectacolelor crescător după timpul lor de încheiere, iar blocul ”**for**” prin care se parcurg spectacolele se execută de  $n$  ori (câte o dată pentru fiecare spectacol) și necesită de fiecare dată o comparație, cel mult o adunare și cel mult 3 operații de atribuire.

*Algoritmul 2.4.2.* O altă rezolvare a problemei spectacolelor se obține prin utilizarea următoarei *strategie Greedy*, similară cu cea de mai sus.

- Ordonăm spectacolele descrescător după timpul lor de începere:

$$a_1 \geq a_2 \geq \dots \geq a_n.$$

- Parcurgem spectacolele, în această ordine, și:
  - selectăm primul spectacol;
  - de fiecare dată, spectacolul curent,  $S_i$ , se selectează doar dacă nu se suprapune cu niciunul dintre spectacolele selectate anterior, adică dacă timpul său de încheiere este mai mic decât timpul de începere al ultimului spectacol  $S_j$  selectat:

$$b_i < a_j.$$

Pentru memorarea soluției se utilizează din nou un vector caracteristic  $c = (c_1, \dots, c_n)$ , cu aceeași semnificație ca în algoritmul de mai sus.

Descrierea în pseudocod a noului algoritm are următoarea formă.

```

SPECTACOLE2 ( $a, b, n, c, m$ ): //  $a = (a_1, \dots, a_n)$ ,  $b = (b_1, \dots, b_n)$ 
    //  $c = (c_1, \dots, c_n)$ ,  $m =$  numărul de spectacole selectate
SORTARE( $a, b, n$ ); // se sortează spectacolele crescător
    // după timpul lor de începere  $a_i$ 
     $m \leftarrow 0$ ; // inițializări
    for  $i = \overline{1, n}$  do  $c[i] \leftarrow 0$ ;
     $t \leftarrow b[n] + 1$ ; //  $t =$  timpul de începere al ultimului
    // spectacol selectat
    for  $i = \overline{n, 1, -1}$  do // parcurgem spectacolele în ordinea
    // descrescătoare a timpilor de începere
    |   if  $b[i] < t$  then
    |   |    $c[i] \leftarrow 1$ ; // selectăm intervalul (spectacolul) curent
    |   |    $m \leftarrow m + 1$ ;
    |   |    $t \leftarrow a[i]$ ; // actualizăm  $t$ 
    AFISARE( $m, c, n$ ); // se afișează numărul și
    // submulțimea intervalelor (spectacolelor) selectate

```

Funcția de afișare este aceeași ca în Algoritmul 2.4.1.

*Observația 2.4.2.* Demonstrația corectitudinii și evaluarea complexității Algoritmului 2.4.2 sunt analoage cu cele ale Algoritmul 2.4.1.

*Exemplul 2.4.2.* Pentru spectacolele din Exemplul 2.4.1, ordonate crescător după timpii lor de începere  $a_i$  în primul tabel, aplicarea strategiei Greedy din Algoritmul 2.4.2 conduce la soluția optimă dată de următoarea selectare (vizionare în ordine inversă) a spectacolelor:

- $S_{14}$  (ultimul, în ordinea descrescătoare a timpilor de începere),
- $S_{10}$  (ultimul situat înainte de  $S_{14}$  și care are timpul de încheiere mai mic decât timpul de începere al lui  $S_{14}$ ),
- $S_7$  (ultimul situat înainte de  $S_{10}$  și care are timpul de încheiere mai mic decât timpul de începere al lui  $S_{10}$ ),
- $S_3$  (ultimul situat înainte de  $S_7$  și care are timpul de încheiere mai mic decât timpul de începere al lui  $S_7$ ), înainte de care nu mai avem niciun spectacol care să se încheie înainte de începerea lui  $S_3$ , deci selectarea se termină.

Numărul maxim de spectacole ce pot fi vizionate este egal, din nou, cu 4.

## Tema 3

# Metoda Backtracking

### 3.1 Descrierea metodei. Algoritmi generali

**Metoda Backtracking (metoda căutării cu revenire)** se aplică problemelor a căror soluție se poate reprezenta sub forma unui vector

$$x = (x_1, x_2, \dots, x_n) \in S_1 \times S_2 \times \dots \times S_n,$$

unde:

- $S_1, S_2, \dots, S_n$  sunt mulțimi finite și nevide, elementele lor aflându-se într-o **relație de ordine** bine stabilită;
- între componentele  $x_1, x_2, \dots, x_n$  ale vectorului  $x$  sunt precizate anumite relații, numite **condiții interne**.

*Observația 3.1.1.* Pentru unele probleme, numărul de componente  $n$  al soluțiilor nu este de la început cunoscut, el urmând a fi determinat pe parcurs.

De asemenea, pentru unele probleme, două soluții pot avea numere diferite de componente.

**Definiția 3.1.1.** Mulțimea finită  $S = S_1 \times S_2 \times \dots \times S_n$  se numește **spațiul soluțiilor posibile**.

Un vector  $x = (x_1, \dots, x_n) \in S$  se numește **soluție posibilă**.

Soluțiile posibile care satisfac condițiile interne se numesc **soluții rezultat**.

*Observația 3.1.2.* Metoda Backtracking își propune să determine:

- fie o soluție rezultat,
- fie toate soluțiile rezultat.

Obținerea tuturor soluțiilor rezultat poate constitui o etapă intermediară în rezolvarea unei alte probleme, urmând ca, în continuare, dintre acestea să fie alese soluțiile care optimizează (minimizează sau maximizează) o anumită funcție obiectiv dată.

*Observația 3.1.3.* O variantă de determinare a soluțiilor rezultat ar putea fi următoarea metodă, numită **metoda forței brute**:

- se generează succesiv toate *soluțiile posibile*, adică toate elementele produsului cartezian  $S_1 \times S_2 \times \cdots \times S_n$ ;
- pentru fiecare soluție posibilă se verifică dacă sunt satisfăcute *condițiile interne*;
- se rețin cele care satisfac aceste condiții.

Această variantă are dezavantajul că timpul de execuție este foarte mare.

De exemplu, dacă  $\text{card}(S_i) = 2$ ,  $i = \overline{1, n}$ , atunci spațiul soluțiilor posibile ar avea  $2^n$  elemente, iar complexitatea ar fi de ordinul  $\Omega(2^n)$  (nesatisfăcătoare!!).

Metoda Backtracking urmărește să evite generarea tuturor soluțiilor posibile, ceea ce duce la scurtarea timpului de execuție.

**Definiția 3.1.2.** Fie  $k \in \{1, \dots, n\}$ . Un set de relații definite pentru componenta  $x_k \in S_k$  prin intermediul unor eventuale componente ale vectorului  $(x_1, \dots, x_{k-1}) \in S_1 \times S_2 \times \cdots \times S_{k-1}$  reprezintă **condiții de continuare** pentru  $x_k$  împreună cu  $(x_1, \dots, x_{k-1})$  dacă:

1. aceste relații sunt necesare pentru existența unei soluții rezultat de forma

$$(x_1, \dots, x_{k-1}, x_k, x_{k+1}, \dots, x_n),$$

adică neîndeplinirea acestor condiții implică faptul că oricum am alege  $x_{k+1} \in S_{k+1}, \dots, x_n \in S_n$  vectorul  $x = (x_1, \dots, x_n)$  nu poate fi o soluție rezultat (nu verifică condițiile interne);

2. pentru  $k = n$  condițiile de continuare coincid cu condițiile interne.

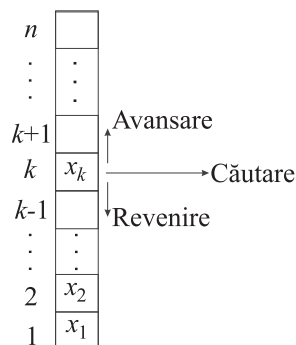
Orice vector  $(x_1, \dots, x_k) \in S_1 \times S_2 \times \cdots \times S_k$ , cu  $1 \leq k \leq n$ , care satisface condițiile de continuare, se numește **soluție parțială (soluție validă)**.

De asemenea, vectorul vid (fără elemente) este considerat ca fiind soluție parțială.



## Mecanismul metodei Backtracking

Prin metoda Backtracking soluția se construiește pas cu pas (componentă cu componentă) pe **principiul stivei**.



La fiecare nivel  $k$ , se caută un element din **mulțimea de nivel**  $S_k$ ,  
care să fie atribuit componentei  $x_k$   
și care împreună cu  $(x_1, \dots, x_{k-1})$  să verifice condițiile de continuare

Mai precis, avem următorii pași:

1. Se începe cu soluția parțială dată de vectorul  $x$  vid.
2. Se ia primul element din mulțimea  $S_1$  și se atribuie lui  $x_1$ .
3. Presupunând generate elementele  $(x_1, x_2, \dots, x_{k-1})$ ,  $x_i \in S_i$ ,  $i = \overline{1, k-1}$ , se **avansează** la nivelul  $k$  și se caută primul element disponibil din  $S_k$  a cărui valoare să fie atribuită lui  $x_k$ .

Avem următoarele cazuri:

**3.1.** A fost găsit în  $S_k$  un element disponibil  $v$ . Atunci:

- îl atribuim lui  $x_k$ ;
- se verifică dacă acesta împreună cu elementele deja generate  $x_1, \dots, x_{k-1}$  îndeplinește condițiile de continuare.

Avem următoarele subcazuri:

**3.1.1.**  $x_k$  verifică condițiile de continuare. Spunem că  $v$  este o **valoare validă** pentru componenta  $x_k$ .

Atunci:

- se extinde soluția parțială la  $(x_1, \dots, x_{k-1}, x_k)$ ;
- se verifică dacă  $k = n$ :

**3.1.1.1.** Dacă **da**, atunci s-a obținut o soluție rezultat.

Acum, fie ne oprim, fie continuăm cu căutarea altei soluții rezultat, reluând algoritmul (cu pasul **3.**), considerând generate  $(x_1, \dots, x_{k-1})$  și căutând în continuare pentru  $x_k$  un element netestat din  $S_k$ .

**3.1.1.2.** Dacă **nu** ( $k < n$ ), se reia algoritmul cu pasul **3.**, căutând extinderea soluției parțiale  $(x_1, \dots, x_{k-1}, x_k)$  cu testarea primului element din  $S_{k+1}$  (**se avansează**).

**3.1.2.**  $x_k$  nu verifică condițiile de continuare. Atunci, oricum am alege următoarele componente pentru  $x$  (adică pe  $x_{k+1}, \dots, x_n$ ), nu vom obține o soluție rezultat. Prin urmare, se va relua algoritmul (cu pasul **3.**), având generate  $x_1, \dots, x_{k-1}$  și căutând extinderea soluției cu următorul element netestat din  $S_k$  (continuă **căutarea**).

**3.2.** Nu a fost găsit în  $S_k$  un element disponibil (netestat). Atunci se consideră generate  $x_1, \dots, x_{k-2}$  (se **revine**) și se reia **căutarea** cu următorul element din  $S_{k-1}$  netestat, pentru extinderea soluției parțiale  $(x_1, \dots, x_{k-2})$  (pas **3.**).

*Observația 3.1.4.* Algoritmul se termină atunci când nu mai există nici un element din  $S_1$  netestat.

*Observația 3.1.5.* După **avansare**, **căutarea** se va face începând cu testarea primului element din mulțimea de nivel  $S_k$  corespunzătoare.

*Observația 3.1.6.* După **revenire**, **căutarea** se va face începând cu următorul element netestat (la etapele anterioare) din mulțimea de nivel  $S_k$  corespunzătoare.

*Observația 3.1.7.* Condițiile de continuare sunt necesare pentru existența unei soluții rezultat, iar ideal este ca ele să fie și suficiente, ceea ce, de obicei, este imposibil!

Alegerea acestor condiții este esențială în utilizarea metodei Backtracking, deoarece cu cât condițiile de continuare sunt mai restrictive, cu atât se limitează mai mult numărul de căutări (în consecință și numărul de avansări și reveniri), deci metoda este mai eficientă.

De obicei condițiile de continuare sunt restricțiile condițiilor interne la primele  $k$  componente.

*Observația 3.1.8.* În general, problemele rezolvate cu Backtracking necesită un timp foarte mare de execuție. De aceea, metoda se utilizează atunci când nu avem la dispoziție un algoritm mai eficient. Problemele tipice care se pot rezolva utilizând această metodă sunt cele *nedeterminist-polinomial complete* (*NP-complete*).

În continuare considerăm **cazul particular** în care mulțimile  $S_k$ ,  $k = \overline{1, n}$ , conțin termeni succesivi ai unor progresii aritmetice. Pentru fiecare  $k$ ,  $k = \overline{1, n}$ , mulțimea  $S_k$  va fi dată de:

- $a_k$  – primul termen al progresiei;
- $r_k > 0$  – rația progresiei;
- $b_k$  – ultimul termen al progresiei,

adică

$$S_k = \{a_k, a_k + r_k, a_k + 2r_k, \dots, b_k\}, \forall k = \overline{1, n}.$$

Pentru acest caz particular, vom prezenta în continuare șase variante pentru schema Backtracking: și anume două variante iterative și patru variante recursive.

*Algoritmul 3.1.1 (Schema Backtracking iterativă, varianta 1).*

**BACKTRACKING1 :**

```

 $k \leftarrow 1;$ 
 $x[1] \leftarrow a[1] - r[1];$  // pregătim introducerea valorii inițiale  $a_1$ 
                                     // pentru  $x_1$ 

while  $k > 0$  do
    if  $x[k] < b[k]$  then
        // mai există valori netestate pentru  $x_k$ 
         $x[k] \leftarrow x[k] + r[k];$  // Căutare
        if  $[(x[1], \dots, x[k]) \text{ verifică condițiile de continuare}]$  then
            if  $k = n$  then
                //  $(x_1, \dots, x_n)$  este soluție rezultat
                PRELUCREAZĂ $(x[1], \dots, x[n]);$ 
                (*) // STOP, dacă se dorește
                // o singură soluție rezultat,
                // sau  $k \leftarrow k - 1$  (revenire forțată) dacă
                // nu mai există valori valide pentru  $x_n$ 
            else
                 $k \leftarrow k + 1;$  // Avansare
                 $x[k] \leftarrow a[k] - r[k];$  // pregătim introducerea
                                     // valorii inițiale  $a_k$  pentru  $x_k$ 
        else
            // nu mai există valori netestate pentru  $x_k$ 
             $k \leftarrow k - 1;$  // Revenire

```

*Observația 3.1.9.* Dacă după determinarea și prelucrarea soluției rezultat nu mai există valori valide pentru componenta  $x[n]$ , atunci în loc de (\*) se poate forța revenirea ( $k \leftarrow k - 1$ ).

Următoarea schemă este echivalentă cu cea de mai sus.

*Algoritmul 3.1.2 (Schema Backtracking iterativă, varianta 2).*

**BACKTRACKING2:**

```

 $k \leftarrow 1;$ 
 $x[1] \leftarrow a[1] - r[1];$  // pregătim introducerea valorii inițiale  $a_1$ 
                                     // pentru  $x_1$ 

while  $k > 0$  do
    if  $k \leq n$  then
        if  $x[k] < b[k]$  then
            // mai există valori netestate pentru  $x_k$ 
             $x[k] \leftarrow x[k] + r[k];$  // Căutare
            if  $[(x[1], \dots, x[k]) \text{ verifică condițiile de continuare}]$  then
                 $k \leftarrow k + 1;$  // Avansare
                 $x[k] \leftarrow a[k] - r[k];$  // pregătim introducerea
                                     // valorii inițiale  $a_k$  pentru  $x_k$ 
            else
                 $k \leftarrow k - 1;$  // Revenire
        else
            //  $(x_1, \dots, x_n)$  este soluție rezultat
            PRELUCREAZĂ $(x[1], \dots, x[n]);$ 
            (*) // STOP, dacă se dorește
                                     // o singură soluție rezultat
             $k \leftarrow k - 1;$  // Revenire, dacă se doresc
                                     // toate soluțiile rezultat

```

*Observația 3.1.10.* Schemele Backtracking de mai sus generează toate soluțiile problemei date. Dacă se dorește obținerea unei singure soluții, atunci în loc de (\*) se poate insera o comandă de oprire (**STOP**).

Algoritmul 3.1.3 (Schema Backtracking recursivă, varianta 1).

```

BACKR1( $k$ ):                                     // se generează  $(x[k], \dots, x[n])$ 
 $x[k] \leftarrow a[k] - r[k];$     // pregătim introducerea valorii inițiale
                                     //  $a[k]$  pentru  $x[k]$ 

while  $x[k] < b[k]$  do
     $x[k] \leftarrow x[k] + r[k];$                                 // Căutare
    if  $[(x[1], \dots, x[k]) \text{ verifică condițiile de continuare}]$  then
        if  $k = n$  then
            //  $(x[1], \dots, x[n])$  este soluție rezultat
            PRELUCREAZĂ $(x[1], \dots, x[n]);$ 
        else
            BACKR1( $k + 1$ );                                // Avansare, adică
            // se generează, RECURSIV,  $(x[k + 1], \dots, x[n])$ .
            // La încheierea apelului BACKR1( $k + 1$ ) se produce
            // revenirea la funcția BACKR1( $k$ )

```

Apelare: **BACKR1**(1).

Următoarea schemă este echivalentă cu cea de mai sus.

Algoritmul 3.1.4 (Schema Backtracking recursivă, varianta 2).

```

BACKR2( $k$ ):
if  $k \leq n$  then
     $x[k] \leftarrow a[k] - r[k];$ 
    while  $x[k] < b[k]$  do
         $x[k] \leftarrow x[k] + r[k];$ 
        if  $[(x[1], \dots, x[k]) \text{ verifică condițiile de continuare}]$  then
            BACKR2( $k + 1$ );
    else
        PRELUCREAZĂ $(x[1], \dots, x[n]);$ 

```

Apelare: **BACKR2**(1).

Algoritmul 3.1.5 (Schema Backtracking recursivă, varianta 3).

```

BACKR3( $k$ ):
  for  $v = \overline{a[k], b[k], r[k]}$  do    // Căutarea lui  $x[k]$  în mulțimea  $S_k$ 
     $x[k] \leftarrow v$ ;
    if  $[(x[1], \dots, x[k]) \text{ verifică condițiile de continuare}]$  then
      if  $k = n$  then
        //  $(x[1], \dots, x[n])$  este soluție rezultat
        PRELUCREAZĂ $(x[1], \dots, x[n])$ ;
      else
        BACKR3( $k + 1$ );           // Avansare, adică
        // se generează, RECURSIV,  $(x[k + 1], \dots, x[n])$ .
        // La încheierea apelului BACKR3( $k + 1$ ) se produce
        // revenirea la funcția BACKR3( $k$ )

```

Apelare: **BACKR3**(1).

Următoarea schemă este echivalentă cu cea de mai sus.

Algoritmul 3.1.6 (Schema Backtracking recursivă, varianta 4).

```

BACKR4( $k$ ):
  if  $k \leq n$  then
    for  $v = \overline{a[k], b[k], r[k]}$  do
       $x[k] \leftarrow v$ ;
      if  $[(x[1], \dots, x[k]) \text{ verifică condițiile de continuare}]$  then
        BACKR4( $k + 1$ );
    else
      PRELUCREAZĂ $(x[1], \dots, x[n])$ ;

```

Apelare: **BACKR4**(1).

*Observația 3.1.11.* Schemele Backtracking din Algoritmii 3.1.1, 3.1.2, 3.1.3 și 3.1.4 pot fi ușor adaptate și pentru situații în care elementele fiecărei mulțimi  $S_k$  nu sunt în progresie aritmetică.

*Observația 3.1.12.* În oricare din cele șase scheme de mai sus, verificarea (validarea) condițiilor de continuare

$$[(x[1], x[2], \dots, x[k]) \text{ verifică condițiile de continuare}]$$

se face adesea prin intermediul unei funcții **VALID**( $x, k$ ).

*Observația 3.1.13.* Dacă, în oricare din cele șase scheme de mai sus, se renunță la testul

$$[(x[1], x[2], \dots, x[k]) \text{ verifică condițiile de continuare}],$$

considerându-se astfel că orice soluție posibilă verifică condițiile interne, atunci se vor obține ca soluții rezultat ale problemei toate elementele produsului cartezian  $S_1 \times S_2 \times \cdots \times S_n$ .

## 3.2 Colorarea grafurilor

**Problema colorării grafurilor** este următoarea:

Se consideră un graf neorientat fără bucle  $G = (V, E)$ , cu mulțimea nodurilor

$$V = \{1, 2, \dots, n\},$$

și un număr de  $m$  culori, numerotate cu  $1, 2, \dots, m$ .

Se cere să se determine toate modalitățile de colorare ale nodurilor grafului, utilizând cele  $m$  culori, astfel încât oricare două noduri adiacente să fie colorate cu culori diferite.

### Modelarea problemei

Orice soluție a problemei se poate scrie sub forma

$$x = (x_1, x_2, \dots, x_n),$$

unde  $x_i$  este culoarea atașată nodului  $i$ ,  $x_i \in \{1, 2, \dots, m\}$ .

Avem

$$x = (x_1, x_2, \dots, x_n) \in S_1 \times S_2 \times \cdots \times S_n,$$

unde

$$S_1 = S_2 = \cdots = S_n = \{1, 2, \dots, m\}.$$

Așadar mulțimile  $S_k$  conțin termenii succesivi ai unei progresii aritmetice, în care

$$\begin{cases} a_k = 1, \\ r_k = 1, \\ b_k = m, \end{cases} \quad \forall k = \overline{1, n}.$$

### Condițiile interne:

Orice două noduri adiacente sunt colorate diferit, adică

$$x_i \neq x_j, \quad \forall [i, j] \in E.$$

**Condițiile de continuare:**

Dacă  $x_1, \dots, x_{k-1}$  sunt deja alese, atunci  $x_k$  *verifică condițiile de continuare* (*este valid*) dacă

$$x_i \neq x_k, \forall i \in \{1, \dots, k-1\} \text{ cu } [i, k] \in E.$$

Altfel spus,  $x_k$  *nu verifică condițiile de continuare* (*nu este valid*), dacă există  $i \in \{1, 2, \dots, k-1\}$  astfel încât  $[i, k] \in E$  și  $x_i = x_k$ .

*Observația 3.2.1.* Pentru reprezentarea grafului utilizăm matricea de adiacență

$$A = (a_{ij})_{i,j=\overline{1,n}}.$$

Graful fiind neorientat, matricea de adiacență este simetrică.

Obținem următorul *algorithm Backtracking*, descris în pseudocod, pentru rezolvarea problemei.

*Algoritmul 3.2.1.*

```

COLORARE( $A, n, m$ ) :
   $k \leftarrow 1$ ;
   $x[1] \leftarrow 0$ ;
  while  $k > 0$  do
    if  $x[k] < m$  then
       $x[k] \leftarrow x[k] + 1$ ;
      if VALID( $x, k$ ) then
        if  $k = n$  then
          | AFISARE( $x$ );
        else
          |  $k \leftarrow k + 1$ ;
          |  $x[k] \leftarrow 0$ ;
      else
        |  $k \leftarrow k - 1$ ;

```

Funcția de verificare (validare) a condițiilor de continuare este

```

VALID( $x, k$ ) :
  for  $i = \overline{1, k-1}$  do
    if ( $a_{ik} \geq 1$  and  $x[i] = x[k]$ ) then
      | returnează 0;
  returnează 1;

```



## Aplicație: colorarea hărților

**Problema colorării hărților** este următoarea:

Se consideră o hartă și un număr de culori. Se cere să se coloreze fiecare țară cu câte una din culori astfel încât orice două țări ce au frontieră comună să fie colorate diferit.

### Modelarea problemei

Oricărei hărți i se poate asocia un graf neorientat simplu astfel:

- fiecărei țări îi corespunde un nod;
- între două noduri există muchie dacă și numai dacă ele corespund unor țări ce au frontieră comună.

O astfel de corespondență este evidențiată în Figura 3.2.1.

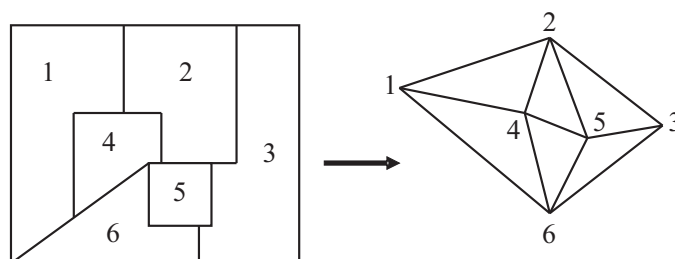


Figura 3.2.1:

Astfel problema colorării unei hărți se reduce la problema colorării grafului asociat.

**Observația 3.2.2.** Graful asociat unei hărți este **planar**, adică există o reprezentare grafică în plan a acestuia astfel încât muchiile să nu se intersecteze decât în noduri.

Avem următorul rezultat celebru.

**Teorema 3.2.1 (Teorema celor 4 culori).** Pentru colorarea unui graf planar sunt suficiente patru culori.

**Observația 3.2.3.** Celebritatea problemei colorării hărților a constatat în faptul că în toate exemplele întâlnite colorarea s-a putut face cu numai 4 culori, dar teoretic se demonstrase că sunt suficiente 5 culori.

- Problema a apărut în 1852, când un matematician amator, Francis Guthrie, a încercat să coloreze harta comitatelor britanice, folosind numai 4 culori.
- În anul 1976, Wolfgang Haken și Kenneth Appel, de la Universitatea Illinois (SUA), au demonstrat că pentru colorare sunt suficiente 4 culori.
- Demonstrația s-a efectuat cu ajutorul calculatorului electronic, analizându-se 1482 configurații în circa 1200 ore calculator.

### 3.3 Problema celor $n$ dame pe tabla de șah

**Problema celor  $n$  dame** este următoarea:

Se consideră o tablă de șah de dimensiune  $n \times n$ ,  $n \geq 2$ . Se cere să se determine toate modalitățile de aranjare a  $n$  dame astfel încât oricare două dame să nu se atace reciproc (pe linii, coloane sau diagonale).

*Observația 3.3.1.* Prin convenție, orice pătrățel al tablei este reprezentat prin coordonatele sale  $(l, c)$ , unde  $l$  reprezintă linia, iar  $c$  reprezintă coloana acestuia. Liniile tablei sunt numerotate cu  $1, 2, \dots, n$  de sus în jos, iar coloanele sunt numerotate cu  $1, 2, \dots, n$  de la stânga la dreapta.

**Propoziția 3.3.1.** *Pe o tablă de dimensiune  $n \times n$  nu putem așeza mai mult de  $n$  dame astfel încât oricare două să nu se atace reciproc.*

*Demonstrație.* Oricum așezăm mai mult de  $n$  dame, cum tabla are  $n$  linii rezultă că există cel puțin două dame așezate pe aceeași linie, și acestea se atacă reciproc.  $\square$

#### Modelarea problemei

În orice soluție rezultat, conform demonstrației propoziției anterioare obținem că pe fiecare linie a tablei se află exact o damă. Rezultă că orice soluție pentru problema celor  $n$  dame se poate reprezenta sub forma unui vector

$$x = (x_1, x_2, \dots, x_n),$$

unde  $x_i$  reprezintă coloana pe care se află dama de pe linia  $i$ , pentru  $i = \overline{1, n}$ .

De exemplu, pentru  $n = 5$  o soluție rezultat este

$$x = (1, 3, 5, 2, 4),$$

corespunzătoare aranjării damelor din Figura 3.3.1.






	1	2	3	4	5
1					
2					
3					
4					
5					

Figura 3.3.1:

Pentru cazul general, avem

$$x = (x_1, x_2, \dots, x_n) \in S_1 \times S_2 \times \dots \times S_n,$$

unde

$$S_1 = S_2 = \dots = S_n = \{1, 2, \dots, n\}.$$

Așadar mulțimile  $S_k$  conțin termenii succesivi ai unei progresii aritmetice, în care

$$\begin{cases} a_k = 1, \\ r_k = 1, \quad \forall k = \overline{1, n}. \\ b_k = n, \end{cases}$$

**Condițiile interne:**

- Damele de pe liniile  $i$  și  $j$  ( $i \neq j$ ) nu se pot afla pe aceeași coloană:

$$x_i \neq x_j, \quad \forall i \neq j, \quad i, j = \overline{1, n};$$

- Damele de pe liniile  $i$  și  $j$  ( $i \neq j$ ) nu se pot afla pe o aceeași diagonală:

$$|i - j| \neq |x_i - x_j|, \quad \forall i \neq j, \quad i, j = \overline{1, n}$$

sau, echivalent,

$$j - i \neq |x_j - x_i|, \quad \forall i, j \text{ a.î. } 1 \leq i < j \leq n.$$

**Condițiile de continuare:**

$$x_k \neq x_i \text{ si } k - i \neq |x_k - x_i|, \forall i = \overline{1, k-1}.$$

$\exists i \in \{1, 2, \dots, k-1\}$  astfel încât  $x_k = x_i$  sau  $k-i = |x_k - x_i|$ .

*Algorithmul 3.3.1.*

 $k \leftarrow 1;$ 
$$x[1] \leftarrow 0;$$
**while**  $k > 0$  **do****if**  $x[k] < n$  **then**
$$x[k] \leftarrow x[k] + 1;$$

if  $VALID(x, k)$  then

**if  $k = n$  then**
$$\mathbf{AFISARE}(x);$$
$$k \leftarrow k - 1;$$

```
-1;           // forțăăm revenirea, deoarece
// nu mai există valori valide pentru  $x[n]$ 
```

else

$$k \leftarrow k + 1;$$
$$x[k] \leftarrow 0;$$

else

$$k \leftarrow k - 1;$$

Funcția de verificare (validare) a condițiilor de continuare este

$$\text{VALID}(x, k) :$$
**for**  $i = \overline{1, k-1}$  **do**

**if**  $(x[k] = x[i] \text{ or } k - i = \lfloor x[k] - x[i] \rfloor)$  **then**

```
returnează 0;
```

```
// fals
```

```
returnează 1;
```

```
// adevărat
```

- Pentru  $n = 2$  problema nu are soluție, deoarece două dame așezate pe linii și coloane diferite se află pe aceeași diagonală, deci se atacă reciproc.

- Nici pentru  $n = 3$  problema nu are soluție, deoarece orice două dame așezate pe o aceeași culoare se atacă reciproc.
- Pentru  $n \geq 4$  avem următorul rezultat celebru.

**Teorema 3.3.1 (E. Pauls, 1874).** *Pentru orice  $n \geq 4$  problema celor  $n$  dame are cel puțin o soluție.*

*Demonstrație.* Avem următoarele cazuri.

Cazul 1)  $n = 6k$  sau  $n = 6k + 4$ . Atunci o soluție este dată de așezarea celor  $n$  dame în pătrățelele de coordonate  $(l, c)$  ce aparțin mulțimii

$$\left\{ (2i, i) \mid 1 \leq i \leq \frac{n}{2} \right\} \cup \left\{ (2i - 1, \frac{n}{2} + i) \mid 1 \leq i \leq \frac{n}{2} \right\}.$$

Cazul 2)  $n = 6k + 1$  sau  $n = 6k + 5$ . Atunci o soluție este dată de așezarea celor  $n$  dame în pătrățelele de coordonate  $(l, c)$  ce aparțin mulțimii

$$\{(n, 1)\} \cup \left\{ (2i, i + 1) \mid 1 \leq i \leq \frac{n-1}{2} \right\} \cup \left\{ (2i - 1, \frac{n+1}{2} + i) \mid 1 \leq i \leq \frac{n-1}{2} \right\}.$$

Cazul 3)  $n = 6k + 2$ . Atunci o soluție este dată de așezarea celor  $n$  dame în pătrățelele de coordonate  $(l, c)$  ce aparțin mulțimii

$$\begin{aligned} & \left\{ (4, 1), (n, \frac{n}{2} - 1), (2, \frac{n}{2}), (n - 1, \frac{n}{2} + 1), (1, \frac{n}{2} + 2), (n - 3, n) \right\} \cup \\ & \cup \left\{ (n - 2i, i + 1) \mid 1 \leq i \leq \frac{n}{2} - 3 \right\} \cup \\ & \cup \left\{ (n - 2i - 3, \frac{n}{2} + i + 2) \mid 1 \leq i \leq \frac{n}{2} - 3 \right\}. \end{aligned}$$

Cazul 4)  $n = 6k + 3$ . Atunci o soluție este dată de așezarea celor  $n$  dame în pătrățelele de coordonate  $(l, c)$  ce aparțin mulțimii

$$\begin{aligned} & \left\{ (1, n), (5, 1), (n, \frac{n-1}{2} - 1), (3, \frac{n-1}{2}), (n - 1, \frac{n-1}{2} + 1) \right\} \cup \\ & \left\{ (2, \frac{n-1}{2} + 2), (n - 3, n - 1) \right\} \cup \left\{ (n - 2i, i + 1) \mid 1 \leq i \leq \frac{n-1}{2} - 3 \right\} \cup \\ & \cup \left\{ (n - 2i - 3, \frac{n-1}{2} + i + 2) \mid 1 \leq i \leq \frac{n-1}{2} - 3 \right\}, \end{aligned}$$

adică se așează o damă în colțul din dreapta sus (pătrățelul de coordonate  $(1, n)$ ) iar celelalte  $n - 1$  dame se așează conform Cazului 3 în subtabla de dimensiune  $(n - 1) \times (n - 1)$  rămasă prin eliminarea primei linii și ultimei coloane.  $\square$

Conform Teoremei 3.3.1, Propoziției 3.3.1 și cazurilor  $n = 2$  și  $n = 3$  considerate mai sus obținem următorul rezultat.

**Corolarul 3.3.1.** *Numărul maxim de dame ce pot fi așezate pe o tablă de șah de dimensiune  $n \times n$  astfel încât oricare două dame să nu se atace reciproc este egal cu*

$$\begin{cases} n - 1, & \text{dacă } n \in \{2, 3\}, \\ n, & \text{dacă } n \geq 4. \end{cases}$$

- Problema a fost expusă inițial pentru tabla de șah obișnuită  $8 \times 8$  în anul 1848 de către șahistul german Max Bezzel.
- Prima rezolvare a problemei celor 8 dame a fost publicată de către Franz Nauck în anul 1850, care a găsit toate cele 92 de soluții ale problemei. Acesta a propus și varianta generalizată la tabla de dimensiune  $n \times n$ .
- Problema a fost imediat studiată de mulți matematicieni, printre care și celebrul matematician Carl Friedrich Gauss.
- Emil Pauls a fost primul care a demonstrat existența soluției pentru orice  $n \geq 4$ , în anul 1874, prin construcția din demonstrația teoremei de mai sus.

### 3.4 Problema nebunilor pe tabla de șah

**Problema nebunilor** este următoarea:

Se consideră o tablă de șah de dimensiune  $n \times n$ ,  $n \geq 2$ . Se cere să se determine toate modalitățile de aranjare a  $n - 1$  nebuni pe pătrățele de aceeași culoare astfel încât oricare doi nebuni să nu se atace reciproc (pe diagonale).

*Observația 3.4.1.* Utilizăm în continuare convenția din Observația 3.3.1. Tot prin convenție, pătrățelul situat în colțul din stânga sus al tablei, de coordonate  $(1, 1)$ , este colorat cu alb.

**Propoziția 3.4.1.** *Pe o tablă de dimensiune  $n \times n$  nu putem așeza mai mult de  $n - 1$  nebuni pe pătrățele de aceeași culoare astfel încât oricare doi nebuni să nu se atace reciproc.*

*Demonstrație.* Mulțimea  $\mathcal{N}$  a pătrățelelor de culoare neagră poate fi partiționată în următoarele diagonale (cu direcția *stânga-jos*  $\rightarrow$  *dreapta-sus*, ordonate crescător după distanța față de colțul din stânga-sus):

$$\mathcal{N} = D_1 \cup D_2 \cup \dots \cup D_{n-1}, \text{ unde} \\ D_k = \{(l, c) \mid l + c = 2k + 1, 1 \leq l \leq n, 1 \leq c \leq n\}, \forall k = \overline{1, n-1}. \quad (3.4.1)$$

Rezultă că oricum așezăm mai mult de  $n - 1$  nebuni pe pătrățele de culoare neagră există cel puțin doi nebuni așezați pe o aceeași diagonală  $D_k$ , și aceștia se atacă reciproc.

Pe de altă parte, mulțimea  $\mathcal{A}$  a pătrățelelor de culoare albă poate fi partitionată în următoarele diagonale (având, de asemenea, direcția *stânga-jos*  $\rightarrow$  *dreapta-sus* și ordonate crescător după distanța față de colțul din stânga-sus):

$$\mathcal{A} = \overline{D}_1 \cup \overline{D}_2 \cup \dots \cup \overline{D}_n, \text{ unde} \\ \overline{D}_k = \{(l, c) \mid l + c = 2k, 1 \leq l \leq n, 1 \leq c \leq n\}, \forall k = \overline{1, n}. \quad (3.4.2)$$

Rezultă că oricum așezăm mai mult de  $n - 1$  nebuni pe pătrățele de culoare albă, fie există cel puțin doi nebuni așezați pe o aceeași diagonală  $\overline{D}_k$  și aceștia se atacă reciproc, fie există câte cel puțin un nebun pe fiecare diagonală  $\overline{D}_k$  și atunci nebunii de pe diagonalele  $\overline{D}_1 = \{(1, 1)\}$  și  $\overline{D}_n = \{(n, n)\}$  se atacă reciproc. □

### Modelarea problemei

Cazul 1) Considerăm cazul așezării celor  $n - 1$  nebuni pe pătrățele de culoare neagră.

Pentru orice soluție rezultat, conform demonstrației propoziției anterioare obținem că pe fiecare din cele  $n - 1$  diagonale  $D_k$  definite prin relația (3.4.1) se află exact un nebun.

Fiecare diagonală  $D_k$ ,  $k = \overline{1, n-1}$ , poate fi rescrisă astfel

$$\begin{aligned} D_k &= \{(l, c) \mid l + c = 2k + 1, 1 \leq l \leq n, 1 \leq c \leq n\} \\ &= \{(l, 2k + 1 - l) \mid 1 \leq l \leq n, 1 \leq 2k + 1 - l \leq n\} \\ &= \{(l, 2k + 1 - l) \mid 1 \leq l \leq n, 2k + 1 - n \leq l \leq 2k\} \\ &= \{(l, 2k + 1 - l) \mid \max\{2k + 1 - n, 1\} \leq l \leq \min\{2k, n\}\}, \end{aligned}$$

deci

$$D_k = \begin{cases} \{(l, 2k + 1 - l) \mid 1 \leq l \leq 2k\}, & \text{dacă } 1 \leq k \leq \left\lfloor \frac{n}{2} \right\rfloor, \\ \{(l, 2k + 1 - l) \mid 2k + 1 - n \leq l \leq n\}, & \text{dacă } \left\lfloor \frac{n}{2} \right\rfloor < k \leq n - 1 \end{cases}$$

( $[t]$  reprezintă *partea întreagă* a numărului  $t \in \mathbb{R}$ ).

Pentru orice  $k \in \{1, 2, \dots, n - 1\}$ , coloana pe care se află nebunul de pe diagonala  $D_k$  este

$$c = 2k + 1 - l,$$

deci este bine determinată de linia  $l$  a acestuia.

Rezultă că orice soluție se poate reprezenta sub forma unui vector

$$x = (x_1, x_2, \dots, x_{n-1}),$$

unde  $x_k$  reprezintă linia pe care se află nebunul de pe diagonala  $D_k$ , pentru  $k = \overline{1, n-1}$ .

De exemplu, pentru  $n = 5$  o soluție rezultat este

$$x = (1, 4, 2, 5),$$

corespunzătoare aranjării nebunilor din Figura 3.4.1.





	1	2	3	4	5
1					
2					
3					
4					
5					

Figura 3.4.1:

Pentru cazul general, avem

$$x = (x_1, x_2, \dots, x_{n-1}) \in S_1 \times S_2 \times \dots \times S_{n-1},$$

unde

$$S_k = \begin{cases} \{1, 2, \dots, 2k\}, & \text{dacă } 1 \leq k \leq \left\lfloor \frac{n}{2} \right\rfloor, \\ \{2k - n + 1, 2k - n + 2, \dots, n\}, & \text{dacă } \left\lfloor \frac{n}{2} \right\rfloor + 1 \leq k \leq n - 1. \end{cases}$$

Așadar mulțimile  $S_k$  conțin termenii succesivi ai unei progresii aritmetice, în care

$$\begin{cases} a_k = 1, \\ r_k = 1, \\ b_k = 2k, \end{cases} \quad \forall k = \overline{1, \left\lfloor \frac{n}{2} \right\rfloor},$$



$$\begin{cases} a_k = 2k - n + 1, \\ r_k = 1, \\ b_k = n, \end{cases} \quad \forall k = \overline{\left\lceil \frac{n}{2} \right\rceil + 1, n - 1}.$$

**Condițiile interne:**

Pentru orice  $i \neq j$ , cei doi nebuni de pe diagonalele  $D_i$  și  $D_j$ , așezați deci pe liniile  $x_i$  respectiv  $x_j$ , și pe coloanele  $c_i = 2i + 1 - x_i$  respectiv  $c_j = 2j + 1 - x_j$ , nu se pot afla nici pe o aceeași diagonală având direcția *stânga-sus*  $\rightarrow$  *dreapta-jos*:

$$x_i - x_j \neq c_i - c_j, \quad \forall i \neq j, \quad i, j = \overline{1, n - 1},$$

sau, echivalent,

$$x_i - x_j \neq 2i + 1 - x_i - 2j - 1 + x_j, \quad \forall i \neq j, \quad i, j = \overline{1, n - 1},$$

adică

$$x_i - x_j \neq i - j, \quad \forall i \neq j, \quad i, j = \overline{1, n - 1}.$$

**Condițiile de continuare:**

Dacă avem soluția parțială  $(x_1, \dots, x_{k-1})$ , atunci  $x_k$  *verifică condițiile de continuare* (*este valid*) dacă

$$x_k - x_i \neq k - i, \quad \forall i = \overline{1, k - 1}.$$

Altfel spus, dacă avem soluția parțială  $x = (x_1, \dots, x_{k-1})$ , atunci  $x_k$  *nu verifică condițiile de continuare* (*nu este valid*) dacă

$$\exists i \in \{1, 2, \dots, k - 1\} \text{ astfel încât } x_k - x_i = k - i.$$

Obținem următorul *algoritm Backtracking*, descris în pseudocod, pentru rezolvarea problemei.

Algoritmul 3.4.1.

**NEBUNI1** ( $n$ ):

$k \leftarrow 1$ ;

$x[1] \leftarrow 0$ ;

$ult \leftarrow 2$ ; //  $ult$  este valoarea finală  $b_k$  pentru  $x_k$

**while**  $k > 0$  **do**

**if**  $x[k] < ult$  **then**

$x[k] \leftarrow x[k] + 1$ ;

**if** **VALID1**( $x, k$ ) **then**

**if**  $k = n - 1$  **then**

**AFISARE**( $x$ );

**else**

$k \leftarrow k + 1$ ;

            // pregătim introducerea valorii inițiale  $a_k$  și

            // actualizăm valoarea finală  $b_k$  pentru  $x_k$

**if**  $k \leq \frac{n}{2}$  **then**

$x[k] \leftarrow 0$ ;

$ult \leftarrow 2k$ ;

**else**

$x[k] \leftarrow 2k - n$ ;

$ult \leftarrow n$ ;

**else**

$k \leftarrow k - 1$ ;

        // actualizăm valoarea finală  $b_k$  pentru  $x_k$

**if**  $k \leq \frac{n}{2}$  **then**

$ult \leftarrow 2k$ ;

**else**

$ult \leftarrow n$ ;

Funcția de verificare (validare) a condițiilor de continuare este

**VALID1**( $x, k$ ):

**for**  $i = \overline{1, k - 1}$  **do**

**if**  $x[k] - x[i] = k - i$  **then**

**returnează** 0;

// fals

**returnează** 1;

// adevărat

Cazul 2) Considerăm acum cazul așezării celor  $n - 1$  nebuni pe pătrățele de culoare albă.

Pentru orice soluție rezultat, conform demonstrației propoziției anterioare

obținem că un prim nebun se așează fie pe pătrățelul  $(1, 1)$  ce formează diagonală  $\overline{D}_1$ , fie pe pătrățelul  $(n, n)$  ce formează diagonală  $\overline{D}_n$ , iar restul de  $n - 2$  nebuni se așează câte unul pe fiecare din diagonalele  $\overline{D}_2, \dots, \overline{D}_{n-1}$  definite prin relația (3.4.2).

Fiecare diagonală  $\overline{D}_k$ ,  $k = \overline{2, n-1}$ , poate fi rescrisă astfel

$$\begin{aligned}\overline{D}_k &= \{(l, c) \mid l + c = 2k, 1 \leq l \leq n, 1 \leq c \leq n\} \\ &= \{(l, 2k - l) \mid 1 \leq l \leq n, 1 \leq 2k - l \leq n\} \\ &= \{(l, 2k - l) \mid 1 \leq l \leq n, 2k - n \leq l \leq 2k - 1\} \\ &= \{(l, 2k - l) \mid \max\{2k - n, 1\} \leq l \leq \min\{2k - 1, n\}\},\end{aligned}$$

deci

$$\overline{D}_k = \begin{cases} \{(l, 2k - l) \mid 1 \leq l \leq 2k - 1\}, & \text{dacă } 2 \leq k \leq \left\lfloor \frac{n+1}{2} \right\rfloor, \\ \{(l, 2k - l) \mid 2k - n \leq l \leq n\}, & \text{dacă } \left\lfloor \frac{n+1}{2} \right\rfloor < k \leq n - 1. \end{cases}$$

Pentru orice  $k \in \{2, \dots, n - 1\}$ , coloana pe care se află nebunul de pe diagonală  $\overline{D}_k$  este

$$c = 2k - l,$$

deci este bine determinată de linia  $l$  a acestuia.

Rezultă că orice soluție se poate reprezenta sub forma unui vector

$$y = (y_1, y_2, \dots, y_{n-1}),$$

unde

$$y_1 = \begin{cases} 1, & \text{dacă primul nebun se află pe pătrățelul } (1, 1), \\ n, & \text{dacă primul nebun se află pe pătrățelul } (n, n), \end{cases}$$

iar pentru  $k \in \{2, \dots, n - 1\}$   $y_k$  reprezintă linia pe care se află nebunul de pe diagonală  $\overline{D}_k$ .

Avem

$$y = (y_1, y_2, \dots, y_{n-1}) \in S_1 \times S_2 \times \dots \times S_{n-1},$$

unde

$$S_k = \begin{cases} \{1, n\}, & \text{dacă } k = 1, \\ \{1, 2, \dots, 2k - 1\}, & \text{dacă } 2 \leq k \leq \left\lfloor \frac{n+1}{2} \right\rfloor, \\ \{2k - n, 2k - n + 1, \dots, n\}, & \text{dacă } \left\lfloor \frac{n+1}{2} \right\rfloor + 1 \leq k \leq n - 1. \end{cases}$$

Pentru  $k \in \{2, \dots, n-1\}$  mulțimile  $S_k$  conțin termenii succesivi ai unei progresii aritmetice, în care

$$\begin{cases} a_k = 1, \\ r_k = 1, \\ b_k = 2k - 1, \end{cases} \quad \forall k = 2, \overline{\left[ \frac{n+1}{2} \right]},$$

$$\begin{cases} a_k = 2k - n, \\ r_k = 1, \\ b_k = n, \end{cases} \quad \forall k = \overline{\left[ \frac{n+1}{2} \right] + 1, n-1}.$$

**Condițiile interne:**

Analog cazului 1) obținem:

$$y_i - y_j \neq i - j, \quad \forall i \neq j, \quad i, j = \overline{2, n-1}.$$

În plus,

$$y_1 \in \{1, n\} \text{ și } y_i \neq i, \quad \forall i = \overline{2, n-1}$$

(nebulul de pe diagonala  $\overline{D_i i = 2, n-1}$ , nu se poate afla pe diagonala dintre pătrățelele  $(1, 1)$  și  $(n, n)$ , deoarece s-ar ataca cu nebunul aflat pe unul din aceste două pătrățelele).

**Condițiile de continuare:**

Dacă avem soluția parțială  $(y_1, \dots, y_{k-1})$ , cu  $k \geq 2$ , atunci  $y_k$  verifică condițiile de continuare (este valid) dacă

$$y_k \neq k \text{ și } y_k - y_i \neq k - i, \quad \forall i = \overline{2, k-1}.$$

Altfel spus, dacă avem soluția parțială  $(y_1, \dots, y_{k-1})$ , cu  $k \geq 2$ , atunci  $y_k$  nu verifică condițiile de continuare (nu este valid) dacă

$$y_k = k \text{ sau } \exists i \in \{2, \dots, k-1\} \text{ astfel încât } y_k - y_i = k - i.$$

Obținem următorul *algoritm Backtracking*, descris în pseudocod, pentru rezolvarea problemei.

Algoritmul 3.4.2.

```

NEBUNI2( $n$ ):                                     // presupunem că  $n \geq 3$ 
 $k \leftarrow 2$ ;
 $y[2] \leftarrow 0$ ;
 $ult \leftarrow 3$ ;                                //  $ult$  este valoarea finală  $b_k$  pentru  $y_k$ 
while  $k > 1$  do
    if  $y[k] < ult$  then
         $y[k] \leftarrow y[k] + 1$ ;
        if VALID2( $y, k$ ) then
            if  $k = n - 1$  then
                 $y[1] \leftarrow 1$ ;                // primul nebun pe pătrățelul (1,1)
                AFISARE( $y$ );
                 $y[1] \leftarrow n$ ;                // primul nebun pe pătrățelul (n,n)
                AFISARE( $y$ );
            else
                 $k \leftarrow k + 1$ ;
                // pregătim introducerea valorii inițiale  $a_k$  și
                // actualizăm valoarea finală  $b_k$  pentru  $y_k$ 
                if  $k \leq \frac{n+1}{2}$  then
                     $y[k] \leftarrow 0$ ;
                     $ult \leftarrow 2k - 1$ ;
                else
                     $y[k] \leftarrow 2k - n - 1$ ;
                     $ult \leftarrow n$ ;
        else
             $k \leftarrow k - 1$ ;
            // actualizăm valoarea finală  $b_k$  pentru  $y_k$ 
            if  $k \leq \frac{n+1}{2}$  then
                 $ult \leftarrow 2k - 1$ ;
            else
                 $ult \leftarrow n$ ;

```

Funcția de verificare (validare) a condițiilor de continuare este

```

VALID2( $y, k$ ) :
  if  $y[k] = k$  then
    returnează 0; // fals
  for  $i = \overline{2, k-1}$  do
    if  $y[k] - y[i] = k - i$  then
      returnează 0; // fals
  returnează 1; // adevărat

```

### Considerații

**Teorema 3.4.1.** *Pentru orice  $n \geq 2$  problema nebunilor are cel puțin o soluție.*

*Demonstrație.* Avem din nou următoarele două cazuri.

Cazul 1) Cei  $n - 1$  nebuni trebuie așezați pe pătrățele de culoare neagră. Atunci o soluție este dată de așezarea celor  $n - 1$  nebuni în pătrățelele de coordonate  $(l, c)$  ce aparțin mulțimii

$$\left\{ (1, 2i) \mid 1 \leq i \leq \frac{n-1}{2} \right\} \cup \left\{ (n, 2i) \mid 1 \leq i \leq \frac{n-1}{2} \right\}, \text{ dacă } n \text{ este impar,}$$

respectiv mulțimii

$$\left\{ (1, 2i) \mid 1 \leq i \leq \frac{n}{2} \right\} \cup \left\{ (n, 2i-1) \mid 2 \leq i \leq \frac{n}{2} \right\}, \text{ dacă } n \text{ este par.}$$

Cazul 2) Cei  $n - 1$  nebuni trebuie așezați pe pătrățele de culoare albă. Atunci o soluție este dată de așezarea celor  $n - 1$  nebuni în pătrățelele de coordonate  $(l, c)$  ce aparțin mulțimii

$$\left\{ (1, 2i-1) \mid 1 \leq i \leq \frac{n+1}{2} \right\} \cup \left\{ (n, 2i-1) \mid 2 \leq i \leq \frac{n-1}{2} \right\}, \text{ dacă } n \text{ este}$$

impar, respectiv mulțimii

$$\left\{ (1, 2i-1) \mid 1 \leq i \leq \frac{n}{2} \right\} \cup \left\{ (n, 2i-2) \mid 2 \leq i \leq \frac{n}{2} \right\}, \text{ dacă } n \text{ este par.}$$

□

Conform Teoremei 3.4.1 și Propoziției 3.4.1 obținem următorul rezultat.

**Corolarul 3.4.1.** *Numărul maxim de nebuni ce pot fi așezați pe pătrățele de aceeași culoare ale unei table de șah de dimensiune  $n \times n$ ,  $n \geq 2$ , astfel încât oricare doi nebuni să nu se atace reciproc este egal cu  $n - 1$ .*

Evident, orice doi nebuni așezați pe pătrățele de culori diferite nu se atacă reciproc. Astfel, conform corolarului anterior obținem următorul rezultat.

**Corolarul 3.4.2.** *Numărul maxim de nebuni ce pot fi așezați pe o tablă de șah de dimensiune  $n \times n$ ,  $n \geq 2$ , astfel încât oricare doi nebuni să nu se atace reciproc este egal cu  $2n - 2$ .*

*Observația 3.4.2.* Pentru generarea tuturor modalităților de aranjare a  $2n - 2$  nebuni pe o tablă de șah de dimensiune  $n \times n$  astfel încât oricare doi nebuni să nu se atace reciproc putem proceda astfel:

- se generează toate modalitățile de așezare a  $n - 1$  nebuni pe pătrățelele de culoare albă astfel încât oricare doi nebuni să nu se atace reciproc;
- pentru fiecare astfel de așezare, se generează toate modalitățile de așezare a restului de  $n - 1$  nebuni pe pătrățelele de culoare neagră astfel încât oricare doi din aceștia să nu se atace reciproc.

Pentru aceasta putem utiliza funcția **NEBUNI2** ( $n$ ) din Algoritmul 3.4.2 în care înlocuim funcția **AFISARE**( $y$ ) cu funcția **NEBUNI1** ( $n$ ) din Algoritmul 3.4.1.

## 3.5 Generarea obiectelor combinatoriale

### 3.5.1 Preliminarii

În continuare vom prezenta formule de numărare și algoritmi de generare (enumerare) pentru cele mai cunoscute familii de obiecte combinatoriale: produs cartezian, submulțimi, aranjamente (fără repetiție sau cu repetiție), combinări (fără repetiție sau cu repetiție), permutări (fără repetiție sau cu repetiție).

Reamintim pentru început câteva noțiuni uzuale.

**Definiția 3.5.1.** *Fie  $A$  un alfabet (adică o mulțime finită) și  $n \in \mathbb{N}^*$ . O secvență de forma*

$$a = a_1 a_2 \dots a_n, \text{ cu } a_1, a_2, \dots, a_n \in A,$$

*se numește **cuvânt de lungime  $n$**  peste alfabetul  $A$ . Lungimea cuvântului  $a$  se notează cu  $|a|$ .*

*Observația 3.5.1.* Evident, cuvântul  $a_1 a_2 \dots a_n$  poate fi identificat cu vectorul  $(a_1, a_2, \dots, a_n)$ .

**Definiția 3.5.2.** Fie  $x \in \mathbb{R}$  și  $n \in \mathbb{N}$ . Notăm

$$[x]_n = \begin{cases} 1, & \text{dacă } n = 0, \\ \underbrace{x(x-1) \dots (x-n+1)}_{n \text{ factori}}, & \text{dacă } n \geq 1, \end{cases}$$

$$[x]^n = \begin{cases} 1, & \text{dacă } n = 0, \\ \underbrace{x(x+1) \dots (x+n-1)}_{n \text{ factori}}, & \text{dacă } n \geq 1. \end{cases}$$

$[x]_n$  se numește **polinomul factorial descrescător** de gradul  $n$ , iar  $[x]^n$  se numește **polinomul factorial crescător** de gradul  $n$ .

### 3.5.2 Produs cartezian

**Definiția 3.5.3. Produsul cartezian** al mulțimilor  $A_1, A_2, \dots, A_n$  ( $n \in \mathbb{N}^*$ ) este mulțimea

$$A_1 \times A_2 \times \dots \times A_n = \{(x_1, x_2, \dots, x_n) | x_1 \in A_1, x_2 \in A_2, \dots, x_n \in A_n\}.$$

*Exemplul 3.5.1.*  $\{a, b\} \times \{+, -\} \times \{c\} = \{(a, +, c), (a, -, c), (b, +, c), (b, -, c)\}$ .

**Propoziția 3.5.1 (de numărare a produsului cartezian).** Fie  $n \in \mathbb{N}^*$  și  $A_1, A_2, \dots, A_n$  mulțimi finite. Atunci  $\text{card}(A_1 \times A_2 \times \dots \times A_n) = \text{card}(A_1) \cdot \text{card}(A_2) \cdot \dots \cdot \text{card}(A_n)$ .

*Demonstrație.* Se utilizează metoda inducției matematice după  $n$ . □

**Algoritmul 3.5.1 (de generare a produsului cartezian prin metoda Backtracking).** Fie mulțimile standard

$$A_1 = \{1, 2, \dots, m_1\}, A_2 = \{1, 2, \dots, m_2\}, \dots, A_n = \{1, 2, \dots, m_n\}.$$

Vom utiliza metoda Backtracking. Avem

$$x = (x_1, x_2, \dots, x_n) \in S_1 \times S_2 \times \dots \times S_n,$$

unde

$$S_1 = A_1, S_2 = A_2, \dots, S_n = A_n.$$

Așadar, mulțimile  $S_k$ , conțin termeni succesivi ai unei progresii aritmetice, în care

$$\begin{cases} a_k = 1, \\ r_k = 1, \\ b_k = m_k, \end{cases} \quad \forall k = \overline{1, n}.$$



Conform Observației 3.1.13, eliminăm verificarea condițiilor de continuare.

**PRODUS\_CARTEZIAN**( $n, m$ ) :

```

 $k \leftarrow 1$ ;
 $x[1] \leftarrow 0$ ;
while  $k > 0$  do
    if  $x[k] < m[k]$  then
         $x[k] \leftarrow x[k] + 1$ ;
        if  $k = n$  then
            | AFISARE( $x, n$ );
        else
            |  $k \leftarrow k + 1$ ;
            |  $x[k] \leftarrow 0$ ;
    else
        |  $k \leftarrow k - 1$ ;

```

Funcția de afișare este

**AFISARE**( $x, n$ ) :

```

for  $i = \overline{1, n}$  do
    | afișează  $x[i]$ ;

```

*Observația 3.5.2.* Pentru generarea produsului cartezian  $A_1 \times A_2 \times \dots \times A_n$  al unor mulțimi finite arbitrare

$$\begin{aligned}
 A_1 &= \{a_{11}, a_{12}, \dots, a_{1m_1}\}, \\
 A_2 &= \{a_{21}, a_{22}, \dots, a_{2m_2}\}, \\
 &\dots, \\
 A_n &= \{a_{n1}, a_{n2}, \dots, a_{nm_n}\}
 \end{aligned}$$

se poate folosi algoritmul anterior, bazat pe generarea indicilor, înlocuind afișarea indicilor  $x_i$  cu afișarea elementelor corespunzătoare  $a_{ix_i}$  din mulțimile  $A_i$ , adică înlocuind funcția **AFISARE**( $x, n$ ) cu funcția

**AFISARE**( $x, a, n$ ) :

```

for  $i = \overline{1, n}$  do
    | afișează  $a[i, x[i]]$ ;

```

### 3.5.3 Submulțimi

**Propoziția 3.5.2 (de numărare a submulțimilor).** *Fie  $A$  o mulțime finită și  $\mathcal{P}(A)$  mulțimea tuturor submulțimilor (părților) lui  $A$ . Atunci*

$$\text{card}(\mathcal{P}(A)) = 2^{\text{card}(A)}.$$

*Demonstrație.* Fie  $A = \{a_1, a_2, \dots, a_n\}$ ,  $n \in \mathbb{N}^*$ . Notăm

$$\{1, 2\}^n = \underbrace{\{1, 2\} \times \dots \times \{1, 2\}}_{\text{de } n \text{ ori}}.$$

Definim funcțiile  $\alpha : \mathcal{P}(A) \rightarrow \{1, 2\}^n$  și  $\beta : \{1, 2\}^n \rightarrow \mathcal{P}(A)$  prin:

- $\forall B \in \mathcal{P}(A)$ ,  $\alpha(B) = (x_1, x_2, \dots, x_n)$ , unde  $x_i = \begin{cases} 1, & \text{dacă } a_i \in B, \\ 2, & \text{dacă } a_i \notin B; \end{cases}$
- $\forall (x_1, x_2, \dots, x_n) \in \{1, 2\}^n$ ,  $\beta(x_1, x_2, \dots, x_n) = \{a_i | x_i = 1, i \in \{1, \dots, n\}\}$ .

Funcțiile  $\alpha$  și  $\beta$  sunt inverse una celeilalte, deci sunt bijective și

$$\text{card}(\mathcal{P}(A)) = \text{card}(\{1, 2\}^n) = 2^n.$$

□

*Exemplul 3.5.2.* Pentru  $A = \{a, b, c\}$ , corespondența *submulțimi*  $\leftrightarrow$  *produs cartezian* din demonstrația anterioară este redată în următorul tabel:

$(x_1, x_2, x_3) \in \{1, 2\}^3$	$B \in \mathcal{P}(A)$
(1,1,1)	$\{a, b, c\}$
(1,1,2)	$\{a, b\}$
(1,2,1)	$\{a, c\}$
(1,2,2)	$\{a\}$
(2,1,1)	$\{b, c\}$
(2,1,2)	$\{b\}$
(2,2,1)	$\{c\}$
(2,2,2)	$\emptyset$

Deci  $A$  are  $2^3 = 8$  submulțimi.

Fie mulțimea  $A = \{a_1, a_2, \dots, a_n\}$ . Conform demonstrației anterioare, putem genera produsul cartezian  $\{1, 2\}^n$  cu Algoritmul 3.5.1 înlocuind afișarea elementelor  $(x_1, \dots, x_n)$  cu afișarea submulțimilor corespunzătoare

$$B = \{a_i | x_i = 1, i \in \{1, \dots, n\}\}.$$

Obținem următorul algoritm descris în pseudocod.

Algoritmul 3.5.2 (de generare a submulțimilor, folosind metoda Backtracking).

```

SUBMULȚIMI( $a, n$ ) :
 $k \leftarrow 1$ ;
 $x[1] \leftarrow 0$ ;
while  $k > 0$  do
    if  $x[k] < 2$  then
         $x[k] \leftarrow x[k] + 1$ ;
        if  $k = n$  then
            AFISARE( $x, a, n$ );
        else
             $k \leftarrow k + 1$ ;
             $x[k] \leftarrow 0$ ;
    else
         $k \leftarrow k - 1$ ;

```

Funcția de afișare este

```

AFISARE( $x, a, n$ ) :
for  $i = \overline{1, n}$  do
    if  $x[i] = 1$  then afișează  $a[i]$ ;

```

### 3.5.4 Aranjamente cu repetiție

**Propoziția 3.5.3** (de numărare a aranjamentelor cu repetiție). *Fie  $m, n \in \mathbb{N}$ . Atunci numărul de cuvinte de lungime  $n$  peste un alfabet cu  $m$  litere este egal cu  $m^n$ .*

*Demonstrație.* Fie  $B = \{1, 2, \dots, m\}$  și  $\mathcal{C} = \{(x_1, x_2, \dots, x_n) | x_i \in B \forall i\}$ . Cum  $\mathcal{C} = B^n$ , conform Propoziției 3.5.1 avem  $\text{card}(\mathcal{C}) = m^n$ .  $\square$

**Definiția 3.5.4.** *Cuvintele numărate în propoziția anterioară se numesc **aranjamente cu repetiție** de  $m$  luate câte  $n$ . Prin abuz de limbaj, și numărul lor, adică  $m^n$ , se numește tot **aranjamente cu repetiție de  $m$  luate câte  $n$** .*

*Exemplul 3.5.3.* Pentru  $m = 2$  și  $n = 3$ , aranjamente cu repetiție sunt, în ordine lexicografică:

$(1, 1, 1), (1, 1, 2), (1, 2, 1), (1, 2, 2), (2, 1, 1), (2, 1, 2), (2, 2, 1), (2, 2, 2)$ .

Deci avem  $2^3 = 8$  aranjamente cu repetiție.

**Algoritmul 3.5.3 (de generare a aranjamentelor cu repetiție).** Pentru mulțimea standard  $B = \{1, 2, \dots, m\}$ , conform demonstrației anterioare putem genera produsul cartezian  $B^n$ , cu Algoritmul 3.5.1 luând  $m_1 = m_2 = \dots = m_n = m$ .

**Observația 3.5.3.** Pentru o mulțime  $A = \{a_1, a_2, \dots, a_n\}$  oarecare putem genera indicii  $(x_1, \dots, x_n)$  cu algoritmul anterior și afișa elementele corespunzătoare acestor indici  $(a_{x_1}, \dots, a_{x_n})$ .

### 3.5.5 Aranjamente

**Propoziția 3.5.4 (de numărare a aranjamentelor).** Fie  $m, n \in \mathbb{N}$ . Atunci numărul de cuvinte de lungime  $n$  cu litere distincte peste un alfabet cu  $m$  litere este egal cu  $[m]_n$ .

*Demonstrație.* Fie  $A = \{1, 2, \dots, m\}$  și

$$\mathcal{C}_1 = \{(x_1, x_2, \dots, x_n) | x_i \in A \forall i, x_i \neq x_j \forall i \neq j\}.$$

Avem

$$\begin{aligned} \mathcal{C}_1 = \{ & (x_1, x_2, \dots, x_n) | x_1 \in \{1, \dots, m\}, \\ & x_2 \in \{1, \dots, m\} \setminus \{x_1\}, \\ & x_3 \in \{1, \dots, m\} \setminus \{x_1, x_2\}, \\ & \dots, \\ & x_n \in \{1, \dots, m\} \setminus \{x_1, \dots, x_{n-1}\} \} \end{aligned}$$

deci  $\text{card}(\mathcal{C}_1) = m(m-1)(m-2)\dots(m-n+1) = [m]_n$ .  $\square$

**Definiția 3.5.5.** Cuvintele numărate în propoziția anterioară se numesc **aranjamente (fără repetiție)** de  $m$  luate câte  $n$ . De asemenea și numărul lor, adică  $[m]_n$ , se numește tot **aranjamente (fără repetiție) de  $m$  luate câte  $n$**  și se mai notează cu  $A_m^n$ .

**Observația 3.5.4.** Pentru  $n > m$  avem

$$[m]_n = m \dots (m-m) \dots (m-n+1) = 0.$$

**Exemplul 3.5.4.** Pentru  $m = 4$  și  $n = 2$ , aranjamentele sunt, în ordine lexicografică:

$$(1, 2), (1, 3), (1, 4), (2, 1), (2, 3), (2, 4), (3, 1), (3, 2), (3, 4), (4, 1), (4, 2), (4, 3).$$

Deci avem  $[4]_2 = 4 \cdot 3 = 12$  aranjamente.

**Algoritmul 3.5.4 (de generare a aranjamentelor prin metoda Backtracking, varianta 1).** Fie mulțimea standard  $A = \{1, 2, \dots, m\}$  și  $n \leq m$ .

Vom utiliza metoda Backtracking. Avem

$$x = (x_1, x_2, \dots, x_n) \in S_1 \times S_2 \times \dots \times S_n,$$

unde

$$S_1 = S_2 = \dots = S_n = A.$$

Așadar, mulțimile  $S_k$ , conțin termeni succesivi ai unei progresii aritmetice, în care

$$\begin{cases} a_k = 1, \\ r_k = 1, \\ b_k = m, \end{cases} \quad \forall k = \overline{1, n}.$$

**Condițiile interne:**

$$x_i \neq x_j, \quad \forall i \neq j, \quad i, j = \overline{1, n}.$$

**Condițiile de continuare:**

Dacă avem soluția parțială  $(x_1, \dots, x_{k-1})$ , atunci  $x_k$  verifică condițiile de continuare dacă

$$x_k \neq x_i, \quad \forall i = \overline{1, k-1}.$$

Altfel spus, dacă avem soluția parțială  $x = (x_1, \dots, x_{k-1})$ , atunci  $x_k$  nu verifică condițiile de continuare dacă

$$\exists i \in \{1, 2, \dots, k-1\} \text{ astfel încât } x_k = x_i.$$

**ARANJAMENTE1( $m, n$ ):**

$k \leftarrow 1;$

$x[1] \leftarrow 0;$

**while**  $k > 0$  **do**

**if**  $x[k] < m$  **then**

$x[k] \leftarrow x[k] + 1;$

**if**  $VALID(x, k)$  **then**

**if**  $k = n$  **then**

**AFISARE**( $x, n$ );

**else**

$k \leftarrow k + 1;$

$x[k] \leftarrow 0;$

**else**

$k \leftarrow k - 1;$

```

VALID( $x, k$ ):
for  $i = 1, k - 1$  do
    if  $x[k] = x[i]$  then
        returnează 0;                                // fals
returnează 1;                                       // adevărat

```

Funcția de afișare este, din nou:

```

AFISARE( $x, n$ ):
for  $i = \overline{1, n}$  do
    afișează  $x[i]$ ;

```

*Algoritmul 3.5.5 (de generare a aranjamentelor prin metoda Backtracking, varianta 2).* Putem îmbunătăți algoritmul anterior prin utilizarea unui vector  $(y_1, y_2, \dots, y_m)$  cu semnificația: dacă avem soluția parțială  $(x_1, \dots, x_{k-1})$ , atunci

$$y_i = \begin{cases} 1, & \text{dacă } i \in \{x_1, \dots, x_{k-1}\}, \\ 0, & \text{dacă } i \notin \{x_1, \dots, x_{k-1}\}, \end{cases} \quad \forall i = \overline{1, m}.$$

Astfel  $x_k$  verifică condițiile de continuare dacă și numai dacă

$$y_{x_k} = 0.$$

Obținem următorul algoritm:

```

ARANJAMENTE2( $m, n$ ):
  for  $i = \overline{1, m}$  do  $y[i] \leftarrow 0$ ;      // inițial, soluția parțială este
                                          // vectorul vid

   $k \leftarrow 1$ ;
   $x[1] \leftarrow 0$ ;
  while  $k > 0$  do
    if  $x[k] < m$  then
       $x[k] \leftarrow x[k] + 1$ ;
      if  $y[x[k]] = 0$  then                //  $x[k]$  verifică condițiile
                                          // de continuare
        if  $k = n$  then
          | AFISARE( $x, n$ );
        else
          |  $y[x[k]] \leftarrow 1$ ;        // înainte de avansare, se adaugă
                                          //  $x[k]$  la soluția parțială
          |  $k \leftarrow k + 1$ ;
          |  $x[k] \leftarrow 0$ ;
      else
        |  $k \leftarrow k - 1$ ;
        |  $y[x[k]] \leftarrow 0$ ;          // după revenire, se elimină
                                          //  $x[k]$  din soluția parțială

```

Funcția de afișare este aceeași ca în algoritmul anterior.

*Observația 3.5.5.* Pentru generarea aranjamentelor de  $m$  luate câte  $n$  ale unor mulțimi finite arbitrare

$$A = \{a_1, a_2, \dots, a_m\}$$

se pot folosi algoritmi anteriori, bazați pe generarea indicilor, înlocuind afișarea indicilor  $x_i$  cu afișarea elementelor corespunzătoare  $a_{x_i}$  din mulțimile  $A_i$ , adică înlocuind funcția **AFISARE**( $x, n$ ) cu funcția

```

AFISARE( $x, a, n$ ):
  for  $i = \overline{1, n}$  do
    | afișează  $a[x[i]]$ ;

```

### 3.5.6 Permutări

**Propoziția 3.5.5 (de numărare a permutărilor).** Fie  $n \in \mathbb{N}$ . Atunci numărul de cuvinte ce conțin exact o dată fiecare literă a unui alfabet cu  $n$

litere este egal cu  $n!$ , unde

$$n! = 1 \cdot 2 \cdot 3 \cdot \dots \cdot n \text{ (} n \text{ factorial)}, 0! = 1.$$

*Demonstrație.* Luăm  $m = n$  în Propoziția 3.5.4 și folosim egalitatea  $[n]_n = n(n-1) \dots 1 = n!$ .  $\square$

**Definiția 3.5.6.** Cuvintele numărate în propoziția anterioară se numesc **permutări (fără repetiție)** de ordinul  $n$ . De asemenea și numărul lor, adică  $n!$ , se numește tot **permutări (fără repetiție) de  $n$** .

*Exemplul 3.5.5.* Pentru  $n = 3$ , permutările mulțimii standard  $\{1, 2, 3\}$  sunt, în ordine lexicografică:

$$(1, 2, 3), (1, 3, 2), (2, 1, 3), (2, 3, 1), (3, 1, 2), (3, 2, 1).$$

Deci avem  $3! = 1 \cdot 2 \cdot 3 = 6$  permutări.

**Algoritmul 3.5.6 (de generare a permutărilor prin metoda Backtracking, varianta 1).** Fie mulțimea standard  $A = \{1, 2, \dots, n\}$ . Luând  $m = n$  în Algoritmul 3.5.4 de generare a aranjamentelor și forțând revenirea după determinarea fiecărei permutări (deoarece după determinarea unei permutări  $(x_1, x_2, \dots, x_n)$  nu mai există alte valori valide pentru componenta  $x_n$ ) obținem următorul algoritm.

**PERMUTĂRI1( $n$ ):**

$k \leftarrow 1;$

$x[1] \leftarrow 0;$

**while**  $k > 0$  **do**

**if**  $x[k] < n$  **then**

$x[k] \leftarrow x[k] + 1;$

**if** **VALID**( $x, k$ ) **then**

**if**  $k = n$  **then**

**AFISARE**( $x, n$ );

$k \leftarrow k - 1;$

                // forțăm revenirea, deoarece  
                // nu mai există valori valide pentru  $x[n]$

**else**

$k \leftarrow k + 1;$

$x[k] \leftarrow 0;$

**else**

$k \leftarrow k - 1;$



```

VALID( $x, k$ ):
  for  $i = 1, k - 1$  do
    if  $x[k] = x[i]$  then
      returnează 0;                                // fals
  returnează 1;                                    // adevărat

```

Funcția de afișare este, din nou:

```

AFISARE( $x, n$ ):
  for  $i = \overline{1, n}$  do
    afișează  $x[i]$ ;

```

**Algoritmul 3.5.7 (de generare a permutărilor prin metoda Backtracking, varianta 2).** Luând  $m = n$  în Algoritmul 3.5.5 de generare a aranjamentelor și forțând din nou revenirea după determinarea fiecărei permutări obținem următorul algoritm.

```

PERMUTĂRI2( $n$ ):
  for  $i = \overline{1, n}$  do  $y[i] \leftarrow 0$ ;
   $k \leftarrow 1$ ;
   $x[1] \leftarrow 0$ ;
  while  $k > 0$  do
    if  $x[k] < n$  then
       $x[k] \leftarrow x[k] + 1$ ;
      if  $y[x[k]] = 0$  then
        if  $k = n$  then
          AFISARE( $x, n$ );
           $k \leftarrow k - 1$ ;
           $y[x[k]] \leftarrow 0$ ;
        else
           $y[x[k]] \leftarrow 1$ ;
           $k \leftarrow k + 1$ ;
           $x[k] \leftarrow 0$ ;
      else
         $k \leftarrow k - 1$ ;
         $y[x[k]] \leftarrow 0$ ;

```

Funcția de afișare este aceeași ca în algoritmul anterior.

**Observația 3.5.6.** Pentru generarea permutărilor unei mulțimi arbitrare  $A = \{a_1, a_2, \dots, a_n\}$  înlocuim afișarea indicilor  $(x_1, \dots, x_n)$  cu afișarea elementelor corespunzătoare  $(a_{x_1}, \dots, a_{x_n})$ .

### 3.5.7 Combinări

**Propoziția 3.5.6 (de numărare a combinațiilor).** Fie  $m, n \in \mathbb{N}$ . Atunci numărul de cuvinte strict crescătoare de lungime  $n$  peste un alfabet (ordonat) cu  $m$  litere

$$= \text{numărul de submulțimi cu } n \text{ elemente ale unei mulțimi cu } m \text{ elemente} \\ = \frac{[m]_n}{n!} \stackrel{\text{def}}{=} \binom{m}{n}.$$

*Demonstrație.* Fie  $A = \{1, 2, \dots, m\}$ . Notăm mulțimile din enunț astfel:

$$\mathcal{C}_3 = \{(c_1, c_2, \dots, c_n) \mid c_i \in A \forall i, c_i < c_{i+1} \forall i\}, \mathcal{S}_3 = \{S \mid S \subseteq A, \text{card}(S) = n\}.$$

Între aceste mulțimi definim funcțiile  $\mu : \mathcal{C}_3 \rightarrow \mathcal{S}_3$ ,  $\nu : \mathcal{S}_3 \rightarrow \mathcal{C}_3$  prin:

- $\forall (c_1, c_2, \dots, c_n) \in \mathcal{C}_3$ ,  $\mu(c_1, c_2, \dots, c_n) = \{c_1, c_2, \dots, c_n\}$ ;
- $\forall \{c_1, c_2, \dots, c_n\} \in \mathcal{S}_3$  cu  $c_1 < c_2 < \dots < c_n$ ,  $\nu(\{c_1, c_2, \dots, c_n\}) = (c_1, c_2, \dots, c_n)$ .

Aceste funcții sunt inverse una celeilalte, deci sunt bijective și astfel

$$\text{card}(\mathcal{C}_3) = \text{card}(\mathcal{S}_3).$$

Permutând fiecare combinație  $(c_1, c_2, \dots, c_n) \in \mathcal{C}_3$  în toate cele  $n!$  moduri posibile, obținem fără repetare toate aranjamentele din mulțimea  $\mathcal{C}_1 = \{(a_1, a_2, \dots, a_n) \mid a_i \in A \forall i, a_i \neq a_j \forall i \neq j\}$ .

Deci  $\text{card}(\mathcal{C}_3) \cdot n! = \text{card}(\mathcal{C}_1)$ . Conform Propoziției 3.5.4,  $\text{card}(\mathcal{C}_1) = [m]_n$ , deci  $\text{card}(\mathcal{C}_3) = \frac{[m]_n}{n!}$ . □

**Definiția 3.5.7.** Oricare obiecte din cele 2 tipuri numărate în propoziția anterioară se numesc **combinări (fără repetiție)** de  $m$  luate câte  $n$ . De asemenea și numărul lor, adică  $\binom{m}{n}$ , se numește tot **combinări (fără repetiție) de  $m$  luate câte  $n$**  și se mai notează cu  $C_m^n$ .

*Observația 3.5.7.* Pentru  $n > m$  avem  $\binom{m}{n} = 0$ , deoarece  $[m]_n = 0$ .

Pentru  $n \leq m$ , deoarece  $[m]_n = m(m-1)\dots(m-n+1) = \frac{m!}{(m-n)!}$  avem

$$\binom{m}{n} = \frac{m!}{n!(m-n)!}.$$

*Exemplul 3.5.6.* Pentru  $m = 5$  și  $n = 3$ , corespondențele din demonstrația anterioară sunt redată în următorul tabel:

$(c_1, c_2, c_3) \in \mathcal{C}_3$	$S \in \mathcal{S}_3$
(1,2,3)	{1, 2, 3}
(1,2,4)	{1, 2, 4}
(1,2,5)	{1, 2, 5}
(1,3,4)	{1, 3, 4}
(1,3,5)	{1, 3, 5}
(1,4,5)	{1, 4, 5}
(2,3,4)	{2, 3, 4}
(2,3,5)	{2, 3, 5}
(2,4,5)	{2, 4, 5}
(3,4,5)	{3, 4, 5}

Deci avem  $\binom{5}{3} = \frac{[5]_3}{3!} = \frac{5 \cdot 4 \cdot 3}{1 \cdot 2 \cdot 3} = 10$  combinații.

**Algoritmul 3.5.8 (de generare a combinațiilor prin metoda Backtracking).** Fie mulțimea standard  $A = \{1, 2, \dots, m\}$  și  $n \leq m$ .

Vom utiliza metoda Backtracking. Fie soluția (combinația)

$$x = (x_1, x_2, \dots, x_n).$$

**Condițiile interne:**

$$1 \leq x_1 < x_2 < \dots < x_{n-1} < x_n \leq m,$$

deci

$$x_n \leq m, \quad x_{n-1} \leq m-1, \quad x_{n-2} \leq m-2, \quad \dots, \quad x_2 \leq m-n+2, \quad x_1 \leq m-n+1.$$

Astfel

$$\begin{aligned} x_1 &\in S_1 = \{1, 2, \dots, m-n+1\}, \\ x_2 &\in S_2 = \{x_1+1, x_1+2, \dots, m-n+2\}, \\ &\dots \\ x_k &\in S_k = \{x_{k-1}+1, x_{k-1}+2, \dots, m-n+k\}, \\ &\dots \\ x_n &\in S_n = \{x_{n-1}+1, x_{n-1}+2, \dots, m\}. \end{aligned}$$

Așadar, mulțimile  $S_k$ , conțin termeni succesivi ai unei progresii aritmetice, în care

$$\begin{cases} a_k = x_{k-1} + 1, \\ r_k = 1, \\ b_k = m - n + k, \end{cases} \quad \forall k = \overline{1, n},$$

considerând  $x_0 = 0$ .

Prin alegerea mulțimilor  $S_1, S_2, \dots, S_n$  am eliminat verificarea condițiilor de continuare.

**COMBINĂRI**( $m, n$ ) :

$k \leftarrow 1$ ;

$x[1] \leftarrow 0$ ;

**while**  $k > 0$  **do**

**if**  $x[k] < m - n + k$  **then**

$x[k] \leftarrow x[k] + 1$ ;

**if**  $k = n$  **then**

**AFISARE**( $x, n$ );

**else**

$k \leftarrow k + 1$ ;

$x[k] \leftarrow x[k - 1]$ ;

**else**

$k \leftarrow k - 1$ ;

Funcția de afișare este, din nou:

**AFISARE**( $x, n$ ) :

**for**  $i = \overline{1, n}$  **do**

**afișează**  $x[i]$ ;

*Observația 3.5.8.* Analog Observației 3.5.6, algoritmul anterior poate fi ușor adaptat pentru generarea combinărilor pentru mulțimi oarecare.

### 3.5.8 Combinări cu repetiție

**Propoziția 3.5.7** (de numărare a combinărilor cu repetiție). *Fie  $m, n \in \mathbb{N}$ . Atunci numărul de cuvinte crescătoare de lungime  $n$  peste un alfabet (ordonat) cu  $m$  litere este egal cu  $\frac{[m]^n}{n!} \stackrel{\text{def}}{=} \binom{m}{n}$ .*

*Demonstrație.* Fie  $B = \{1, 2, \dots, m\}$ ,  $A = \{1, 2, \dots, m + n - 1\}$ ,

$$\mathcal{C}_4 = \{(c_1, c_2, \dots, c_n) \mid c_i \in B \forall i, c_i \leq c_{i+1} \forall i\}$$

$$\mathcal{C}_3 = \{(d_1, d_2, \dots, d_n) \mid d_i \in A \forall i, d_i < d_{i+1} \forall i\}.$$

Definim corespondențele  $\rho : \mathcal{C}_4 \rightarrow \mathcal{C}_3$ ,  $\sigma : \mathcal{C}_3 \rightarrow \mathcal{C}_4$  astfel:

- $\rho(c_1, c_2, c_3, \dots, c_n) = (c_1, c_2 + 1, c_3 + 2, \dots, c_n + n - 1)$ ;
- $\sigma(d_1, d_2, d_3, \dots, d_n) = (d_1, d_2 - 1, d_3 - 2, \dots, d_n - n + 1)$ .

Acestea sunt funcții inverse, deci utilizând Propoziția 3.5.6 avem  
 $\text{card}(\mathcal{C}_4) = \text{card}(\mathcal{C}_3) = \binom{m+n-1}{n} = \frac{[m+n-1]_n}{n!} = \frac{[m]^n}{n!} = \left(\binom{m}{n}\right).$   $\square$

**Definiția 3.5.8.** Cuvintele numărate în propoziția anterioară se numesc **combinări cu repetiție** de  $m$  luate câte  $n$ . De asemenea și numărul lor, adică  $\left(\binom{m}{n}\right)$ , se numește tot **combinări cu repetiție de  $m$  luate câte  $n$** .

*Exemplul 3.5.7.* Pentru  $m = 3$  și  $n = 4$ , combinările cu repetiție sunt, în ordine lexicografică:

$$\begin{aligned} &(1, 1, 1, 1), (1, 1, 1, 2), (1, 1, 1, 3), (1, 1, 2, 2), (1, 1, 2, 3), \\ &(1, 1, 3, 3), (1, 2, 2, 2), (1, 2, 2, 3), (1, 2, 3, 3), (1, 3, 3, 3), \\ &(2, 2, 2, 2), (2, 2, 2, 3), (2, 2, 3, 3), (2, 3, 3, 3), (3, 3, 3, 3). \end{aligned}$$

Deci avem  $\left(\binom{3}{4}\right) = \frac{[3]^4}{4!} = \frac{3 \cdot 4 \cdot 5 \cdot 6}{1 \cdot 2 \cdot 3 \cdot 4} = 15$  combinări cu repetiție.

**Algoritmul 3.5.9 (de generare a combinărilor cu repetiție prin metoda Backtracking).** Fie mulțimea standard  $A = \{1, 2, \dots, m\}$  și  $n \geq 1$ .

Vom utiliza metoda Backtracking. Fie soluția (combinarea cu repetiție)

$$x = (x_1, x_2, \dots, x_n).$$

**Condițiile interne:**

$$1 \leq x_1 \leq x_2 \leq \dots \leq x_{n-1} \leq x_n \leq m.$$

Astfel

$$\begin{aligned} x_1 &\in S_1 = \{1, 2, \dots, m\}, \\ x_2 &\in S_2 = \{x_1, x_1 + 1, \dots, m\}, \\ &\dots \\ x_k &\in S_k = \{x_{k-1}, x_{k-1} + 1, \dots, m\}, \\ &\dots \\ x_n &\in S_n = \{x_{n-1}, x_{n-1} + 1, \dots, m\}. \end{aligned}$$

Așadar, mulțimile  $S_k$ , conțin termeni succesivi ai unei progresii aritmetice, în care

$$\begin{cases} a_k = x_{k-1}, \\ r_k = 1, \\ b_k = m, \end{cases} \quad \forall k = \overline{1, n},$$

considerând  $x_0 = 1$ .

Prin alegerea mulțimilor  $S_1, S_2, \dots, S_n$  am eliminat verificarea condițiilor de continuare.

**COMBINĂRI\_CU\_REPETIȚIE**( $m, n$ ) :

```

 $k \leftarrow 1$ ;
 $x[1] \leftarrow 0$ ;
while  $k > 0$  do
    if  $x[k] < m$  then
         $x[k] \leftarrow x[k] + 1$ ;
        if  $k = n$  then
            | AFISARE( $x, n$ );
        else
            |  $k \leftarrow k + 1$ ;
            |  $x[k] \leftarrow x[k - 1] - 1$ ;
    else
        |  $k \leftarrow k - 1$ ;

```

Funcția de afișare este, din nou:

**AFISARE**( $x, n$ ) :

```

for  $i = \overline{1, n}$  do
    | afișează  $x[i]$ ;

```

*Observația 3.5.9.* Analog Observației 3.5.6, algoritmul anterior poate fi ușor adaptat pentru generarea combinațiilor cu repetiție pentru mulțimi oarecare.

### 3.5.9 Permutări cu repetiție

**Propoziția 3.5.8** (de numărare a permutărilor cu repetiție). *Fie  $m, t_1, t_2, \dots, t_m \in \mathbb{N}$  și  $n = t_1 + t_2 + \dots + t_m$ . Atunci numărul de cuvinte de lungime  $n$  ce pot fi formate peste un alfabet cu  $m$  litere astfel încât litera numărul  $i$  să apară de exact  $t_i$  ori pentru orice  $i \in \{1, \dots, m\}$  este egal cu*

$$\frac{n!}{t_1! t_2! \dots t_m!} \stackrel{\text{def}}{=} \binom{n}{t_1, t_2, \dots, t_m}.$$

*Demonstrație.* Fie  $A = \{1, 2, \dots, m\}$  și

$$\mathcal{C}_5 = \{(x_1, x_2, \dots, x_n) | x_i \in A \forall i, \text{card}(\{i | x_i = j\}) = t_j \forall j\}.$$

Numărăm cuvintele din  $\mathcal{C}_5$  astfel:

- alegem cei  $t_1$  indici (din totalul de  $n$ ) ai literelor egale cu 1, rezultă  $\binom{n}{t_1}$  moduri posibile;

- pentru fiecare alegere de mai sus, alegem cei  $t_2$  indici (din restul de  $n - t_1$ ) ai literelor egale cu 2, rezultă  $\binom{n-t_1}{t_2}$  moduri posibile, deci obținem  $\binom{n}{t_1} \binom{n-t_1}{t_2}$  moduri posibile de alegere a indicilor literelor 1 și 2;
- ...
- pentru fiecare alegere de mai sus, alegem cei  $t_m$  indici (din restul de  $n - t_1 - \dots - t_{m-1}$ ) ai literelor egale cu  $m$ ; rezultă  $\binom{n-t_1-\dots-t_{m-1}}{t_m}$  moduri posibile, deci obținem

$$\binom{n}{t_1} \binom{n-t_1}{t_2} \dots \binom{n-t_1-\dots-t_{m-1}}{t_m}$$

moduri posibile de alegere a tuturor indicilor literelor  $1, 2, \dots, m$ .

Astfel

$$\begin{aligned} \text{card}(\mathcal{C}_5) &= \binom{n}{t_1} \binom{n-t_1}{t_2} \dots \binom{n-t_1-\dots-t_{m-1}}{t_m} \\ &= \frac{n!}{t_1!(n-t_1)!} \cdot \frac{(n-t_1)!}{t_2!(n-t_1-t_2)!} \cdot \dots \cdot \frac{(n-t_1-\dots-t_{m-1})!}{t_m!(n-t_1-\dots-t_m)!} \\ &= \frac{n!}{t_1!t_2! \dots t_m!} \end{aligned}$$

(deoarece  $(n - t_1 - \dots - t_m)! = 0! = 1$ ).

□

**Definiția 3.5.9.** Cuvintele numărate în propoziția anterioară se numesc **permutări cu repetiție (anagrame)** de  $n$  luate câte  $t_1, t_2, \dots, t_m$ . De asemenea și numărul lor, adică  $\binom{n}{t_1, t_2, \dots, t_m}$ , se numește tot **permutări cu repetiție de  $n$  luate câte  $t_1, t_2, \dots, t_m$** .

*Observația 3.5.10.* Luând  $t_1 = t_2 = \dots = t_m = 1$  obținem  $n = m$  și  $\binom{n}{1, 1, \dots, 1} = n!$ , deci permutările (fără repetiție) sunt un caz particular al permutărilor cu repetiție. Pe de altă parte, luând  $m = 2$  obținem  $\binom{n}{t_1, t_2} = \frac{n!}{t_1!t_2!}$ , deci și combinările (fără repetiție) sunt un caz particular al permutărilor cu repetiție.

*Exemplul 3.5.8.* Pentru  $m = 3$  și  $t_1 = 2, t_2 = t_3 = 1$ , deci  $n = 4$ , permutările cu repetiție sunt, în ordine lexicografică:

$$(1, 1, 2, 3), (1, 1, 3, 2), (1, 2, 1, 3), (1, 2, 3, 1), (1, 3, 1, 2), (1, 3, 2, 1), \\ (2, 1, 1, 3), (2, 1, 3, 1), (2, 3, 1, 1), (3, 1, 1, 2), (3, 1, 2, 1), (3, 2, 1, 1).$$

Deci avem  $\binom{4}{2, 1, 1} = \frac{4!}{2!1!1!} = 12$  permutări cu repetiție.

**Algoritmul 3.5.10 (de generare a permutărilor cu repetiție prin metoda Backtracking, varianta 1).** Fie mulțimea standard  $A = \{1, 2, \dots, m\}$ ,  $t_1, t_2, \dots, t_m \in \mathbb{N}$  și  $n = t_1 + t_2 + \dots + t_m$ .

Vom utiliza metoda Backtracking. Avem

$$x = (x_1, x_2, \dots, x_n) \in S_1 \times S_2 \times \dots \times S_n,$$

unde

$$S_1 = S_2 = \dots = S_n = A.$$

Așadar, mulțimile  $S_k$ , conțin termeni succesivi ai unei progresii aritmetice, în care

$$\begin{cases} a_k = 1, \\ r_k = 1, \\ b_k = m, \end{cases} \quad \forall k = \overline{1, n}.$$

**Condițiile interne:**

$$\text{card}(\{i | i = \overline{1, n}, x_i = j\}) = t_j \quad \forall j = \overline{1, m}.$$

**Condițiile de continuare:**

Dacă avem soluția parțială  $(x_1, \dots, x_{k-1})$ , atunci  $x_k$  verifică condițiile de continuare dacă

$$\text{card}(\{i | i = \overline{1, k-1}, x_i = x_k\}) < t_{x_k}.$$



**PERMUTĂRI\_CU\_REPETIȚIE1**( $n, t, m$ ):

```

 $n \leftarrow 0$ ;
for  $i = \overline{1, m}$  do  $n \leftarrow n + t[i]$ ;
 $k \leftarrow 1$ ;
 $x[1] \leftarrow 0$ ;
while  $k > 0$  do
    if  $x[k] < m$  then
         $x[k] \leftarrow x[k] + 1$ ;
         $p \leftarrow t[x[k]]$ ;
        if VALID( $p, x, k$ ) then
            if  $k = n$  then
                AFISARE( $x, n$ );
                 $k \leftarrow k - 1$ ;           // forțăm revenirea, deoarece
                // nu mai există valori valide pentru  $x[n]$ 
            else
                 $k \leftarrow k + 1$ ;
                 $x[k] \leftarrow 0$ ;
        else
             $k \leftarrow k - 1$ ;

```

**VALID**( $p, x, k$ ): //  $p = t[x[k]]$   
 $q \leftarrow 0$ ; //  $q$  reprezintă  $\text{card}(\{i | i = \overline{1, k-1}, x[i] = x[k]\})$   
**if**  $p = 0$  **then** **returnează** 0;  
**for**  $i = \overline{1, k-1}$  **do**  
**if**  $x[i] = x[k]$  **then**  
 $q \leftarrow q + 1$ ;  
**if**  $q \geq p$  **then** **returnează** 0;  
**returnează** 1;

Funcția de afișare este, din nou:

**AFISARE**( $x, n$ ):  
**for**  $i = \overline{1, n}$  **do**  
 $\lfloor$  **afișează**  $x[i]$ ;

*Algoritmul 3.5.11 (de generare a permutărilor cu repetiție prin metoda Backtracking, varianta 2).* Putem îmbunătăți algoritmul anterior prin utilizarea unui vector  $(y_1, y_2, \dots, y_m)$  cu semnificația: dacă avem soluția parțială  $(x_1, \dots, x_{k-1})$ , atunci

$$y_j = \text{card}(\{i | i = \overline{1, k-1}, x_i = j\}), \quad \forall j = \overline{1, m}.$$

Astfel  $x_k$  verifică condițiile de continuare dacă și numai dacă

$$y_{x_k} < t_{x_k}.$$

Forțând din nou revenirea după determinarea fiecărei permutări cu repetiție, obținem următorul algoritm.

```

PERMUTĂRI_CU_REPETIȚIE2( $n, t, m$ ):
 $n \leftarrow 0$ ;
for  $i = \overline{1, m}$  do  $n \leftarrow n + t[i]$ ;
for  $i = \overline{1, m}$  do  $y[i] \leftarrow 0$ ;           // inițial, soluția parțială este
                                           // vectorul vid

 $k \leftarrow 1$ ;
 $x[1] \leftarrow 0$ ;
while  $k > 0$  do
    if  $x[k] < m$  then
         $x[k] \leftarrow x[k] + 1$ ;
        if  $y[x[k]] < t[x[k]]$  then           //  $x[k]$  verifică condițiile
                                           // de continuare
            if  $k = n$  then
                AFISARE( $x, n$ );
                 $k \leftarrow k - 1$ ;
                 $y[x[k]] \leftarrow y[x[k]] - 1$ ; // după forțarea revenirii,
                                           // se elimină  $x[k]$  din soluția parțială
            else
                 $y[x[k]] \leftarrow y[x[k]] + 1$ ; // înainte de avansare, se
                                           // adaugă  $x[k]$  la soluția parțială
                 $k \leftarrow k + 1$ ;
                 $x[k] \leftarrow 0$ ;
        else
             $k \leftarrow k - 1$ ;
             $y[x[k]] \leftarrow y[x[k]] - 1$ ; // după revenire, se elimină
                                           //  $x[k]$  din soluția parțială

```

Funcția de afișare este aceeași ca în algoritmul anterior.

*Observația 3.5.11.* Analog Observației 3.5.6, algoritmul anterior poate fi ușor adaptat pentru generarea permutărilor cu repetiție pentru mulțimi arbitrare.

### 3.5.10 Compuneri ale unui număr natural

**Definiția 3.5.10.** Fie  $m, n \in \mathbb{N}$ . O **compunere** a lui  $m$  este o scriere de forma

$$m = t_1 + t_2 + \cdots + t_n,$$

unde  $t_1, t_2, \dots, t_n \in \mathbb{N}$  și contează ordinea dintre termenii  $t_1, t_2, \dots, t_n$ .

**Propoziția 3.5.9 (de numărare a compunerilor unui număr natural).**

Fie  $m, n \in \mathbb{N}$ . Atunci:

- a) numărul de compuneri ale lui  $m$  cu  $n$  termeni este egal cu  $\left(\binom{n}{m}\right)$ ;
- b) numărul de compuneri ale lui  $m$  cu  $n$  termeni nenuli este egal cu  $\binom{m-1}{n-1}$ .

*Demonstrație.* a) Fie mulțimile

$$\mathcal{N} = \{(t_1, t_2, \dots, t_n) | t_i \in \mathbb{N} \forall i, t_1 + t_2 + \cdots + t_n = m\},$$

$$\mathcal{C}_4 = \{(c_1, c_2, \dots, c_m) | c_i \in \{1, 2, \dots, n\} \forall i, c_i \leq c_{i+1} \forall i\}.$$

Definim corespondențele  $\alpha : \mathcal{N} \rightarrow \mathcal{C}_4$ ,  $\beta : \mathcal{C}_4 \rightarrow \mathcal{N}$  prin:

- $\alpha(t_1, \dots, t_n) = (c_1, \dots, c_m)$ , unde

$$\begin{cases} c_1 = \cdots = c_{t_1} = 1 \text{ (} t_1 \text{ litere)}, \\ c_{t_1+1} = \cdots = c_{t_1+t_2} = 2 \text{ (} t_2 \text{ litere)}, \\ \dots \\ c_{t_1+\cdots+t_{n-1}+1} = \cdots = c_{t_1+\cdots+t_{n-1}+t_n} = n \text{ (} t_n \text{ litere)}; \end{cases}$$

- $\beta(c_1, \dots, c_m) = (t_1, \dots, t_n)$ , unde  $t_i =$  numărul de litere  $i$  ale cuvântului  $(c_1, \dots, c_m)$ ,  $\forall i$ .

Acestea sunt funcții inverse, deci conform Propoziției 3.5.7 avem  $\text{card}(\mathcal{N}) = \text{card}(\mathcal{C}_4) = \left(\binom{n}{m}\right)$ .

- b) Fie mulțimile  $\mathcal{N}_1 = \{(t_1, t_2, \dots, t_n) | t_i \in \mathbb{N}^* \forall i, t_1 + t_2 + \cdots + t_n = m\}$ ,

$$\mathcal{N}_2 = \{(u_1, u_2, \dots, u_n) | u_i \in \mathbb{N} \forall i, u_1 + u_2 + \cdots + u_n = m - n\}.$$

Definim corespondențele  $\varphi : \mathcal{N}_1 \rightarrow \mathcal{N}_2$ ,  $\psi : \mathcal{N}_2 \rightarrow \mathcal{N}_1$  prin:

- $\forall (t_1, \dots, t_n) \in \mathcal{N}_1$ ,  $\varphi(t_1, \dots, t_n) = (t_1 - 1, \dots, t_n - 1)$ ;
- $\forall (u_1, \dots, u_n) \in \mathcal{N}_2$ ,  $\psi(u_1, \dots, u_n) = (u_1 + 1, \dots, u_n + 1)$ .

Acestea sunt funcții inverse, deci conform punctului a) avem

$$\begin{aligned} \text{card}(\mathcal{N}_1) = \text{card}(\mathcal{N}_2) &= \left( \binom{n}{m-n} \right) = \frac{[n]^{m-n}}{(m-n)!} = \frac{n(n+1) \dots (m-1)}{(m-n)!} \\ &= \frac{(m-1)!}{(n-1)!(m-n)!} = \binom{m-1}{n-1}. \end{aligned}$$

□

*Exemplul 3.5.9.* Pentru  $m = 4$  și  $n = 3$  compunerile sunt

$$\begin{aligned} 4 &= 0 + 0 + 4 = 0 + 1 + 3 = 0 + 2 + 2 = 0 + 3 + 1 = 0 + 4 + 0 \\ &= 1 + 0 + 3 = 1 + 1 + 2 = 1 + 2 + 1 = 1 + 3 + 0 = 2 + 0 + 2 \\ &= 2 + 1 + 1 = 2 + 2 + 0 = 3 + 0 + 1 = 3 + 1 + 0 = 4 + 0 + 0. \end{aligned}$$

Deci avem  $\left( \binom{3}{4} \right) = \frac{3 \cdot 4 \cdot 5 \cdot 6}{1 \cdot 2 \cdot 3 \cdot 4} = 15$  compuneri, dintre care  $\binom{4-1}{3-1} = \binom{3}{2} = 3$  compuneri cu termenii nenuli.

**Algoritmul 3.5.12 (de generare a compunerilor unui număr natural prin metoda Backtracking).** Fie  $n, m \in \mathbb{N}^*$ . Vom utiliza metoda Backtracking. Fie soluția (compunere a lui  $m$  cu  $n$  termeni)

$$x = (x_1, x_2, \dots, x_n).$$

**Condițiile interne:**

$$x_1 + x_2 + \dots + x_n = m, \quad x_1, x_2, \dots, x_n \in \mathbb{N},$$

deci

$$\begin{aligned} x_1 \leq m, \quad x_1 + x_2 \leq m, \quad x_1 + x_2 + x_3 \leq m, \quad \dots, \quad x_1 + x_2 + \dots + x_k \leq m, \quad \dots, \\ x_1 + x_2 + \dots + x_{n-1} \leq m. \end{aligned}$$

Astfel

$$\begin{aligned} x_1 &\in S_1 = \{0, 1, \dots, m\}, \\ x_2 &\in S_2 = \{0, 1, \dots, m - x_1\}, \\ x_3 &\in S_3 = \{0, 1, \dots, m - (x_1 + x_2)\}, \\ &\dots \\ x_k &\in S_k = \{0, 1, \dots, m - (x_1 + x_2 + \dots + x_{k-1})\}, \\ &\dots \\ x_{n-1} &\in S_{n-1} = \{0, 1, \dots, m - (x_1 + x_2 + \dots + x_{n-2})\}, \end{aligned}$$

iar

$$x_n = m - (x_1 + x_2 + \dots + x_{n-1}).$$

Așadar, generăm termenii  $(x_1, x_2, \dots, x_{n-1})$ , cu  $x_k \in S_k$ ,  $k = \overline{1, n-1}$ , iar ultimul termen se calculează cu formula  $x_n = m - (x_1 + x_2 + \dots + x_{n-1})$ .

Mulțimile  $S_k$ ,  $k = \overline{1, n-1}$  conțin termeni succesivi ai unei progresii aritmetice, în care

$$\begin{cases} a_k = 0, \\ r_k = 1, \\ b_k = m - (x_1 + x_2 + \dots + x_{k-1}), \end{cases} \quad \forall k = \overline{1, n-1}.$$

Prin alegerea mulțimilor  $S_1, S_2, \dots, S_{n-1}$  am eliminat verificarea condițiilor de continuare.

Vom utiliza o variabilă  $s$  cu semnificația: dacă avem soluția parțială  $(x_1, x_2, \dots, x_{k-1})$ , atunci

$$s = x_1 + x_2 + \dots + x_{k-1}.$$

Aplicând *Schema Backtracking iterativă, varianta 1* (Algoritmul 3.1.1), obținem următorul algoritm.

**COMPUNERI1**( $m, n$ ):

$k \leftarrow 1$ ;

$x[1] \leftarrow -1$ ;

$s \leftarrow 0$ ;      // inițial, soluția parțială este vectorul vid

**while**  $k > 0$  **do**

**if**  $x[k] < m - s$  **then**

$x[k] \leftarrow x[k] + 1$ ;

**if**  $k = n - 1$  **then**

$x[n] \leftarrow m - s - x[k]$ ;

**AFISARE**( $x, n$ );

**else**

$s \leftarrow s + x[k]$ ;      // înainte de avansare, se adaugă

                                    //  $x[k]$  la soluția parțială

$k \leftarrow k + 1$ ;

$x[k] \leftarrow -1$ ;

**else**

$k \leftarrow k - 1$ ;

$s \leftarrow s - x[k]$ ;      // după revenire, se elimină

                                    //  $x[k]$  din soluția parțială

Funcția de afișare este:

```
AFISARE( $x, n$ ):  
for  $i = \overline{1, n}$  do  
  afişează  $x[i]$ ;
```

*Observația 3.5.12.* Algoritmul anterior funcționează pentru  $n \geq 2$ . Pentru  $n = 1$  soluția rezultat  $(x_1, x_2, \dots, x_{n-1})$  este chiar vectorul inițial, vectorul vid, și nu poate fi obținută prin utilizarea schemei menționate, deoarece în această schemă testarea dacă soluția parțială este soluție rezultat se efectuează prin condiția **if**  $k = n - 1$ , care este falsă,  $n - 1$  fiind egal cu 0, iar  $k$  fiind egal inițial cu 1.

Acest inconvenient poate fi ușor depășit prin utilizarea algoritmului anterior doar în cazul  $n \geq 2$  și considerând separat cazul  $n = 1$ , în care se afișează direct singura soluție, corespunzătoare compunerii  $m = m$ .

Inconvenientul discutat în observația anterioară nu mai apare dacă aplicăm *Schema Backtracking iterativă, varianta 2* (Algoritmul 3.1.2). În această schemă vectorul inițial, vectorul vid, este soluție rezultat pentru  $n - 1 = 0$ , deoarece condiția **if**  $k \leq n - 1$  este falsă,  $k$  fiind egal inițial cu 1.

Astfel obținem următorul algoritm.

**COMPUNERI2**( $m, n$ ):

```

 $k \leftarrow 1$ ;
 $x[1] \leftarrow -1$ ;
 $s \leftarrow 0$ ;           // inițial, soluția parțială este vectorul vid
while  $k > 0$  do
    if  $k \leq n - 1$  then
        if  $x[k] < m - s$  then
             $x[k] \leftarrow x[k] + 1$ ;
             $s \leftarrow s + x[k]$ ;           // înainte de avansare, se adaugă
                                           //  $x[k]$  la soluția parțială
             $k \leftarrow k + 1$ ;
             $x[k] \leftarrow -1$ ;
        else
             $k \leftarrow k - 1$ ;
             $s \leftarrow s - x[k]$ ;           // după revenire, se elimină
                                           //  $x[k]$  din soluția parțială
    else
         $x[n] \leftarrow m - s$ ;
        AFISARE( $x, n$ );
         $k \leftarrow k - 1$ ;
         $s \leftarrow s - x[k]$ ;           // după revenire, se elimină
                                           //  $x[k]$  din soluția parțială

```

Funcția de afișare este aceeași ca în algoritmul anterior.

**Algoritmul 3.5.13 (de generare a compunerilor unui număr natural cu termeni nenuli prin metoda Backtracking).** Fie  $n, m \in \mathbb{N}^*$ . Vom utiliza metoda Backtracking. Fie soluția (compunere a lui  $m$  cu  $n$  termeni nenuli)

$$x = (x_1, x_2, \dots, x_n).$$

**Condițiile interne:**

$$x_1 + x_2 + \dots + x_n = m, \quad x_1, x_2, \dots, x_n \in \mathbb{N}^*,$$

deci

$$\begin{aligned}
 &x_1 + x_2 + \dots + x_{n-1} \leq m - 1, \quad x_1 + x_2 + \dots + x_{n-2} \leq m - 2, \quad \dots, \\
 &x_1 + x_2 + \dots + x_k \leq m - n + k, \quad \dots, \quad x_1 + x_2 \leq m - n + 2, \quad x_1 \leq m - n + 1.
 \end{aligned}$$

Astfel

$$\begin{aligned}
 x_1 &\in S_1 = \{1, 2, \dots, m - n + 1\}, \\
 x_2 &\in S_2 = \{1, 2, \dots, m - n + 2 - x_1\}, \\
 x_3 &\in S_3 = \{1, 2, \dots, m - n + 3 - (x_1 + x_2)\}, \\
 &\dots \\
 x_k &\in S_k = \{1, 2, \dots, m - n + k - (x_1 + x_2 + \dots + x_{k-1})\}, \\
 &\dots \\
 x_{n-1} &\in S_{n-1} = \{1, 2, \dots, m - 1 - (x_1 + x_2 + \dots + x_{n-2})\},
 \end{aligned}$$

iar

$$x_n = m - (x_1 + x_2 + \dots + x_{n-1}).$$

Așadar, din nou generăm termenii  $(x_1, x_2, \dots, x_{n-1})$ , cu  $x_k \in S_k$ ,  $k = \overline{1, n-1}$ , iar ultimul termen se calculează cu formula  $x_n = m - (x_1 + x_2 + \dots + x_{n-1})$ .

Mulțimile  $S_k$ ,  $k = \overline{1, n-1}$  conțin termeni succesivi ai unei progresii aritmetice, în care

$$\begin{cases} a_k = 1, \\ r_k = 1, \\ b_k = m - n + k - (x_1 + x_2 + \dots + x_{k-1}), \end{cases} \quad \forall k = \overline{1, n-1}.$$

Prin alegerea mulțimilor  $S_1, S_2, \dots, S_{n-1}$  am eliminat din nou verificarea condițiilor de continuare.

Vom utiliza din nou o variabilă  $s$  cu semnificația: dacă avem soluția parțială  $(x_1, x_2, \dots, x_{k-1})$ , atunci

$$s = x_1 + x_2 + \dots + x_{k-1}.$$

Pentru a nu discuta separat cazul  $n = 1$ , din nou preferăm *Schema Backtracking iterativă, varianta 2* (Algoritmul 3.1.2), și obținem următorul algo-



ritm.

```

COMPUNERI_CU_TERMENI_NENULI( $m, n$ ):
 $k \leftarrow 1$ ;
 $x[1] \leftarrow 0$ ;
 $s \leftarrow 0$ ;           // inițial, soluția parțială este vectorul vid
while  $k > 0$  do
    if  $k \leq n - 1$  then
        if  $x[k] < m - n + k - s$  then
             $x[k] \leftarrow x[k] + 1$ ;
             $s \leftarrow s + x[k]$ ;           // înainte de avansare, se adaugă
                                           //  $x[k]$  la soluția parțială
             $k \leftarrow k + 1$ ;
             $x[k] \leftarrow 0$ ;
        else
             $k \leftarrow k - 1$ ;
             $s \leftarrow s - x[k]$ ;           // după revenire, se elimină
                                           //  $x[k]$  din soluția parțială
    else
         $x[n] \leftarrow m - s$ ;
        AFISARE( $x, n$ );
         $k \leftarrow k - 1$ ;
         $s \leftarrow s - x[k]$ ;           // după revenire, se elimină
                                           //  $x[k]$  din soluția parțială

```

Funcția de afișare este aceeași ca în algoritmul anterior.

### 3.5.11 Partiții ale unui număr natural

**Definiția 3.5.11.** O *partiție (descompunere)* a numărului  $m \in \mathbb{N}^*$  este o scriere de forma

$$m = t_1 + t_2 + \cdots + t_n,$$

unde  $t_1, t_2, \dots, t_n \in \mathbb{N}^*$  ( $n \in \mathbb{N}^*$ ) și nu contează ordinea dintre termenii  $t_1, t_2, \dots, t_n$ .

*Observația 3.5.13.* Deoarece într-o partiție ca mai sus nu contează ordinea dintre termeni, putem presupune că aceștia sunt scriși în ordine crescătoare.

**Definiția 3.5.12.** Fie  $m, n \in \mathbb{N}^*$ . Notăm cu  $P(m, n)$  numărul de partiții ale lui  $m$  cu  $n$  termeni, iar cu  $P(m)$  numărul tuturor partițiilor lui  $m$ .

*Exemplul 3.5.10.* Numărul  $m = 6$  are partițiile

$$\begin{aligned} 6 &= 6 = 1 + 5 = 2 + 4 = 3 + 3 = 1 + 1 + 4 = 1 + 2 + 3 = 2 + 2 + 2 \\ &= 1 + 1 + 1 + 3 = 1 + 1 + 2 + 2 = 1 + 1 + 1 + 1 + 2 = 1 + 1 + 1 + 1 + 1 + 1, \end{aligned}$$

deci  $P(6, 1) = 1$ ,  $P(6, 2) = 3$ ,  $P(6, 3) = 3$ ,  $P(6, 4) = 2$ ,  $P(6, 5) = 1$ ,  $P(6, 6) = 1$  și  $P(6) = 11$ .

*Observația 3.5.14.* Avem  $P(m) = P(m, 1) + P(m, 2) + \dots + P(m, m)$ ,  $\forall m \in \mathbb{N}^*$ .

**Propoziția 3.5.10 (relația de recurență a numerelor  $P(m, n)$ ).** Pentru orice  $m \in \mathbb{N}^*$  și orice  $n \in \{1, \dots, m-1\}$  avem

$$P(m, n) = P(m-n, 1) + P(m-n, 2) + \dots + P(m-n, n).$$

*Demonstrație.* Pentru orice  $m \in \mathbb{N}^*$  și orice  $n \in \{1, \dots, m-1\}$  notăm

$$\mathcal{P}(m, n) = \{(t_1, t_2, \dots, t_n) | t_i \in \mathbb{N}^* \forall i, t_1 \leq t_2 \leq \dots \leq t_n, t_1 + \dots + t_n = m\}.$$

Definim corespondențele

$$\alpha : \mathcal{P}(m, n) \rightarrow \bigcup_{i=1}^n \mathcal{P}(m-n, i) \text{ și } \beta : \bigcup_{i=1}^n \mathcal{P}(m-n, i) \rightarrow \mathcal{P}(m, n)$$

prin:

- $\forall (t_1, \dots, t_n) \in \mathcal{P}(m, n)$ ,  $\alpha(t_1, \dots, t_n) = (t_j - 1, \dots, t_n - 1)$ , unde  $j = \min\{i | t_i \geq 2, i \in \{1, \dots, n\}\}$  (există  $j$ , deoarece  $n \leq m-1$ );
- $\forall i \in \{1, \dots, n\}$ ,  $\forall (t_1, \dots, t_i) \in \mathcal{P}(m-n, i)$ ,

$$\beta(t_1, \dots, t_i) = (\underbrace{1, \dots, 1}_{\text{de } n-i \text{ ori}}, t_1 + 1, \dots, t_i + 1).$$

Interpretarea acestor funcții este următoarea: aplicarea funcției  $\alpha$  unei partiții a lui  $m$  cu  $n$  termeni constă în micșorarea cu 1 a fiecărui termen și eliminarea termenilor care astfel devin egali cu zero, obținându-se o partiție a lui  $m-n$  cu cel mult  $n$  termeni. Reciproc, aplicarea funcției  $\beta$  unei partiții a lui  $m-n$  cu cel mult  $n$  termeni constă în mărirea cu 1 a fiecărui termen și adăugarea de termeni egali cu 1 pentru a obține  $n$  termeni, astfel obținându-se o partiție a lui  $m$  cu  $n$  termeni.

Funcțiile  $\alpha$  și  $\beta$  sunt bine definite și inverse una celeilalte, deci

$$\text{card}(\mathcal{P}(m, n)) = \text{card}\left(\bigcup_{i=1}^n \mathcal{P}(m-n, i)\right).$$

Cum mulțimile  $\mathcal{P}(m-n, 1), \dots, \mathcal{P}(m-n, n)$  sunt evident disjuncte două câte două, rezultă că  $P(m, n) = \sum_{i=1}^n P(m-n, i)$ .  $\square$

*Observația 3.5.15.* Relația de recurență din Propoziția 3.5.10, împreună cu condițiile inițiale evidente

$$P(m, 1) = P(m, m) = 1, P(m, n) = 0 \quad \forall n > m$$

permit calculul tuturor numerelor  $P(m, n)$ , deci și al numerelor  $P(m)$ , conform Corolarului 3.5.14. De exemplu, tabelul numerelor  $P(m, n)$  și  $P(m)$  pentru  $m \leq 7$  (și  $n \leq 7$ ) este:

$P(m, n)$	$n = 1$	$n = 2$	$n = 3$	$n = 4$	$n = 5$	$n = 6$	$n = 7$	$P(m)$
$m = 1$	1	0	0	0	0	0	0	1
$m = 2$	1	1	0	0	0	0	0	2
$m = 3$	1	1	1	0	0	0	0	3
$m = 4$	1	2	1	1	0	0	0	5
$m = 5$	1	2	2	1	1	0	0	7
$m = 6$	1	3	3	2	1	1	0	11
$m = 7$	1	3	4	3	2	1	1	15

Numerele  $P(m, n)$  nenule, aflate doar pe diagonala principală și sub această diagonală (adică  $1 \leq n \leq m$ ) formează **triunghiul numerelor**  $P(m, n)$ .

**Algoritmul 3.5.14 (de generare a partițiilor lui  $m$  cu  $n$  termeni prin metoda Backtracking).** Fie  $n, m \in \mathbb{N}^*$ . Vom utiliza metoda Backtracking. Fie soluția (partiție a lui  $m$  cu  $n$  termeni)

$$x = (x_1, x_2, \dots, x_n).$$

**Condițiile interne:**

$$x_1 + x_2 + \dots + x_n = m, \quad x_1, x_2, \dots, x_n \in \mathbb{N}^*, \quad x_1 \leq x_2 \leq \dots \leq x_n,$$

deci

$$\begin{aligned} x_1 + x_2 + \dots + x_{n-2} + 2x_{n-1} &\leq m, \quad x_1 + x_2 + \dots + x_{n-3} + 3x_{n-2} \leq m, \quad \dots, \\ x_1 + x_2 + \dots + x_{k-1} + (n-k+1)x_k &\leq m, \quad \dots, \quad x_1 + (n-1)x_2 \leq m, \quad nx_1 \leq m. \end{aligned}$$

Astfel

$$\begin{aligned}
x_1 &\in S_1 = \left\{1, 2, \dots, \left\lfloor \frac{m}{n} \right\rfloor\right\}, \\
x_2 &\in S_2 = \left\{x_1, x_1 + 1, \dots, \left\lfloor \frac{m - x_1}{n - 1} \right\rfloor\right\}, \\
x_3 &\in S_3 = \left\{x_2, x_2 + 1, \dots, \left\lfloor \frac{m - (x_1 + x_2)}{n - 2} \right\rfloor\right\}, \\
&\dots \\
x_k &\in S_k = \left\{x_{k-1}, x_{k-1} + 1, \dots, \left\lfloor \frac{m - (x_1 + x_2 + \dots + x_{k-1})}{n - k + 1} \right\rfloor\right\}, \\
&\dots \\
x_{n-1} &\in S_{n-1} = \left\{x_{n-2}, x_{n-2} + 1, \dots, \left\lfloor \frac{m - (x_1 + x_2 + \dots + x_{n-2})}{2} \right\rfloor\right\},
\end{aligned}$$

iar

$$x_n = m - (x_1 + x_2 + \dots + x_{n-1}).$$

Așadar, generăm termenii  $(x_1, x_2, \dots, x_{n-1})$ , cu  $x_k \in S_k$ ,  $k = \overline{1, n-1}$ , iar ultimul termen se calculează cu formula  $x_n = m - (x_1 + x_2 + \dots + x_{n-1})$ .

Mulțimile  $S_k$ ,  $k = \overline{1, n-1}$  conțin termeni succesivi ai unei progresii aritmetice, în care

$$\begin{cases} a_k = x_{k-1}, \\ r_k = 1, \\ b_k = \left\lfloor \frac{m - (x_1 + x_2 + \dots + x_{k-1})}{n - k + 1} \right\rfloor, \end{cases} \quad \forall k = \overline{1, n-1},$$

considerând  $x_0 = 1$ .

Prin alegerea mulțimilor  $S_1, S_2, \dots, S_{n-1}$  am eliminat verificarea condițiilor de continuare.

Vom utiliza o variabilă  $s$  cu semnificația: dacă avem soluția parțială  $(x_1, x_2, \dots, x_{k-1})$ , atunci

$$s = x_1 + x_2 + \dots + x_{k-1}.$$

Pentru a nu discuta separat cazul  $n = 1$ , din nou preferăm *Schema Backtracking iterativă, varianta 2* (Algoritmul 3.1.2), și obținem următorul algo-

ritm.

**PARTIȚII**( $m, n$ ): $k \leftarrow 1$ ; $x[1] \leftarrow 0$ ; $s \leftarrow 0$ ;      // inițial, soluția parțială este vectorul vid**while**  $k > 0$  **do**    **if**  $k \leq n - 1$  **then**        **if**  $x[k] < \left\lfloor \frac{m - s}{n - k + 1} \right\rfloor$  **then**             $x[k] \leftarrow x[k] + 1$ ;             $s \leftarrow s + x[k]$ ;      // înainte de avansare, se adaugă  
  //  $x[k]$  la soluția parțială             $k \leftarrow k + 1$ ;             $x[k] \leftarrow x[k - 1] - 1$ ;        **else**             $k \leftarrow k - 1$ ;             $s \leftarrow s - x[k]$ ;      // după revenire, se elimină  
  //  $x[k]$  din soluția parțială    **else**         $x[n] \leftarrow m - s$ ;        **AFISARE**( $x, n$ );         $k \leftarrow k - 1$ ;         $s \leftarrow s - x[k]$ ;      // după revenire, se elimină  
  //  $x[k]$  din soluția parțială

Funcția de afișare este:

**AFISARE**( $x, n$ ):**for**  $i = 1, n$  **do**    **afișează**  $x[i]$ ;

## Tema 4

# Metoda Divide et Impera

### 4.1 Descrierea metodei. Algoritmi generali

Metoda **Divide et Impera** ("împarte și stăpânește") este o metodă generală de elaborare a algoritmilor care implică trei etape:

- **Divide:** Problema dată este împărțită în două sau mai multe subprobleme de același tip, dar de dimensiuni mai mici.
- **Impera:** Stăpânește subproblemele obținute. Subproblemele se rezolvă direct, dacă dimensiunea lor permite aceasta (cazuri elementare), sau, fiind de același tip, se rezolvă în mod recursiv, prin același procedeu.
- **Combină:** Se combină soluțiile tuturor subproblemelor, pentru a obține soluția problemei inițiale.

În continuare prezentăm o schemă generală de lucru.

*Algoritmul 4.1.1 (Metoda Divide et Impera).*

Fie  $P(n)$  o problemă de dimensiune  $n \in \mathbb{N}^*$ . Presupunem că există  $n_0 \in \mathbb{N}^*$ ,  $a \in \mathbb{N}$ ,  $a \geq 1$  și  $b \in \mathbb{R}$ ,  $b > 1$  astfel încât:

- $\forall n \in \mathbb{N}^*$  cu  $n \leq n_0$ , rezolvarea problemei  $P(n)$  se poate face direct, fără descompunere.
- $\forall n \in \mathbb{N}^*$  cu  $n > n_0$ ,  $\exists m_1, \dots, m_a \in \mathbb{N}^*$  cu  $m_i \leq \frac{n}{b}$ ,  $\forall i = \overline{1, a}$ , astfel încât rezolvarea problemei  $P(n)$  se poate face rezolvând succesiv  $a$  subprobleme de același tip

$$P(m_1), \dots, P(m_a)$$

după care, combinând soluțiile celor  $a$  subprobleme, se obține soluția dorită a întregii probleme  $P(n)$ .

Algoritmul poate fi descris recursiv astfel:

```

DIVIMP( $n, S$ ):           // Rezolvarea problemei  $P(n)$ 
if  $n \leq n_0$  then
    | PRELUCREAZA( $n, S$ );    // Rezolvarea se face direct,
    |                       // rezultând soluția  $S$  a problemei  $P(n)$ 
else
    | DESCOMPUNERE( $n, a, m$ ); // Se descompune problema,
    |                       // adică se determină  $m_1, \dots, m_a$ 
    | for  $i = \overline{1, a}$  do
    | | DIVIMP( $m_i, s_i$ );    // Se obține soluția  $s_i$  pentru
    | |                       // subproblema  $P(m_i)$ 
    | COMBINA( $s, a, S$ );    // Se combină soluțiile  $s_1, \dots, s_a$ ,
    |                       // rezultând soluția  $S$  a problemei  $P(n)$ 

```

*Observația 4.1.1.* Pentru orice  $i = \overline{1, a}$  avem

$$m_i \leq \frac{n}{b} < n$$

(deoarece  $b > 1$ ), adică subproblemele  $P(m_1), \dots, P(m_a)$  au dimensiuni mai mici decât problema mare  $P(n)$ .

În continuare prezentăm o schemă particulară de lucru.

**Algoritmul 4.1.2 (Metoda Divide et Impera, variantă vectorială).**

Fie  $A = (a_1, a_2, \dots, a_n)$ ,  $n \in \mathbb{N}^*$ , un vector asupra componentelor căruia trebuie efectuată o prelucrare. Presupunem că există  $\varepsilon \in \mathbb{N}$  astfel încât:

- $\forall p, q \in \{1, 2, \dots, n\}$  cu  $q - p \leq \varepsilon$ , prelucrarea secvenței  $(a_p, \dots, a_q)$  se poate face direct, fără descompunere.
- $\forall p, q \in \{1, 2, \dots, n\}$  cu  $q - p > \varepsilon$ ,  $\exists m \in \{p, \dots, q - 1\}$  astfel încât prelucrarea secvenței

$$(a_p, \dots, a_q)$$

se poate face prelucrând succesiv subsecvențele

$$(a_p, \dots, a_m) \text{ și } (a_{m+1}, \dots, a_q),$$

după care, combinând rezultatele celor două prelucrări, se obține prelucrarea dorită a întregii secvențe  $(a_p, \dots, a_q)$ .

Algoritmul poate fi descris recursiv astfel:

```

DIVIMP( $p, q, S$ ):           // Prelucrarea secvenței ( $a_p, \dots, a_q$ )
if  $q - p \leq \varepsilon$  then
    | PRELUCREAZA( $p, q, S$ ); // Prelucrarea se face direct,
    |                       // rezultând soluția  $S$  a secvenței ( $a_p, \dots, a_q$ )
else
    | DESCOMPUNERE( $p, q, m$ ); // Se descompune problema,
    |                       // adică se determină  $m$ 
    | DIVIMP( $p, m, S_1$ );      // Se obține soluția  $S_1$  pentru
    |                       // subsecvența ( $a_p, \dots, a_m$ )
    | DIVIMP( $m + 1, q, S_2$ ); // Se obține soluția  $S_2$  pentru
    |                       // subsecvența ( $a_{m+1}, \dots, a_q$ )
    | COMBINA( $S_1, S_2, S$ ); // Se combină soluțiile  $S_1$  și  $S_2$ ,
    |                       // rezultând soluția  $S$  a secvenței ( $a_p, \dots, a_q$ )

```

Procedura trebuie apelată prin:

**DIVIMP**(1,  $n, S$ ).

În acest algoritm:

- $\varepsilon$  reprezintă lungimea maximă a unei secvențe ( $a_p, \dots, a_q$ ), notată prescurtat prin  $(p, q)$ , pentru care prelucrarea se poate face direct, fără a mai fi necesară împărțirea în subprobleme; procedura

**PRELUCREAZA**( $p, q, S$ )

realizează prelucrarea secvențelor de acest tip, furnizând rezultatul în  $S$ .

- Procedura

**COMBINA**( $S_1, S_2, S$ )

realizează combinarea rezultatelor  $S_1, S_2$  ale prelucrării celor două subsecvențe vecine  $(p, m)$  și  $(m + 1, q)$ , obținându-se rezultatul  $S$  al prelucrării secvenței  $(p, q)$ .

- Valoarea  $m$  este obținută apelând procedura

**DESCOMPUNERE**( $p, q, m$ ).



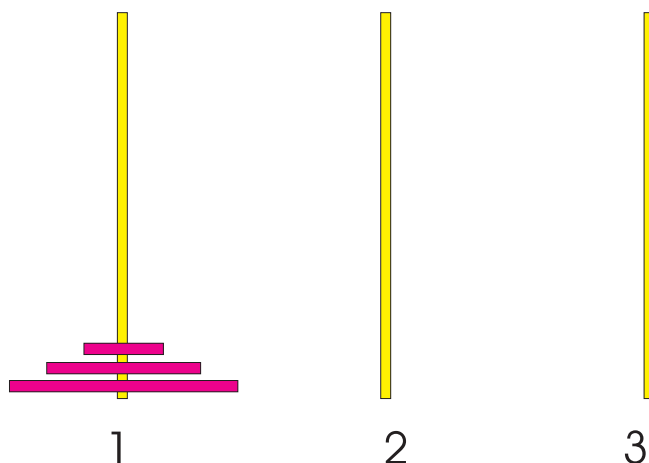
## 4.2 Problema turnurilor din Hanoi

**Problema turnurilor din Hanoi** este următoarea:

Se consideră trei tije notate cu 1, 2, 3 și  $n \geq 1$  discuri perforate de diametre diferite. Inițial, toate discurile se află pe tija 1, în ordinea crescătoare a diametrelor, considerând sensul de la vârful către bază.

Se cere mutarea discurilor pe tija 2, în aceeași ordine, utilizând tija 3 și respectând următoarele reguli:

- la fiecare pas se mută un singur disc;
- nu este permisă așezarea unui disc peste unul cu diametru mai mic.



### Modelarea problemei

Vom nota o mutare cu perechea  $(i, j)$ , ceea ce semnifică faptul că se mută un disc de pe tija  $i$  pe tija  $j$  ( $i, j \in \{1, 2, 3\}$ ).

Dacă  $n = 1$ , soluția directă este mutarea  $(1, 2)$ .

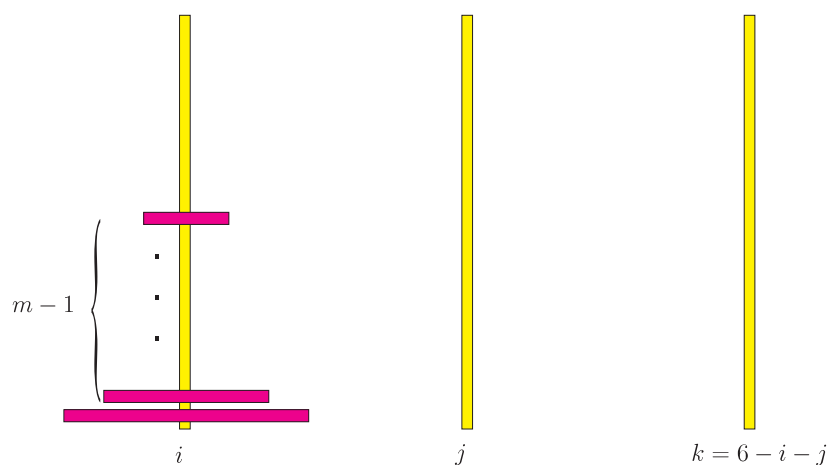
Dacă  $n = 2$ , o soluție este formată din mutările  $(1, 3)$ ,  $(1, 2)$ ,  $(3, 2)$ .

Dacă  $n > 2$ , problema se complică.

Notăm cu  $H(m, i, j)$  șirul mutărilor necesare pentru a muta primele  $m$  discuri de pe tija  $i$  pe tija  $j$ , folosind pentru manevre cealaltă tijă  $k := 6 - i - j$ . (Avem  $i + j + k = 1 + 2 + 3 = 6$ .)

Rezultă că, pentru problema noastră, avem de determinat șirul  $H(n, 1, 2)$ .  
Avem

$$H(m, i, j) = \begin{cases} (i, j), & \text{dacă } m = 1, \\ H(m-1, i, k), (i, j), H(m-1, k, j), & \text{dacă } m \geq 2. \end{cases}$$



Așadar, pentru  $m \geq 2$ , mutarea a  $m$  discuri de pe tija  $i$  pe tija  $j$  este echivalentă cu:

- mutarea a  $m - 1$  discuri de pe tija  $i$  pe tija  $k = 6 - i - j$ , utilizând pentru manevre tija  $j$ ;
- mutarea discului rămas de pe tija  $i$  pe tija  $j$ ;
- mutarea celor  $m - 1$  discuri de pe tija  $k$  pe tija  $j$ , utilizând pentru manevre tija  $i$ .

Deci rezolvarea unei probleme de dimensiune  $m \geq 2$  se reduce la rezolvarea a trei probleme, de același tip, dar de dimensiuni mai mici:

- două de dimensiune  $m - 1$ , care se vor rezolva, recursiv, în același mod;
- una de dimensiune 1, care se rezolvă direct.

Obținem următorul *algorithm Divide et Impera*, descris în pseudocod, pentru rezolvarea problemei.

*Algoritmul 4.2.1.*

```

HANOI( $m, i, j$ ) :
if  $m = 1$  then
    | afișează( $i, j$ );                                // se afișează mutarea ( $i, j$ )
else
    |  $k \leftarrow 6 - i - j$ ;
    | HANOI( $m - 1, i, k$ );
    | afișează( $i, j$ );                                // se afișează mutarea ( $i, j$ )
    | HANOI( $m - 1, k, j$ );

```

Procedura trebuie apelată prin:

$$\mathbf{HANOI}(n, 1, 2).$$

*Observația 4.2.1.* Pentru evaluarea complexității algoritmului anterior, notăm cu  $M(n)$  numărul de mutări necesare (afișate). Avem

$$\begin{aligned} M(n) &= \begin{cases} 1, & \text{dacă } n = 1, \\ M(n-1) + 1 + M(n-1), & \text{dacă } n \geq 2, \end{cases} \\ &= \begin{cases} 1, & \text{dacă } n = 1, \\ 2 \cdot M(n-1) + 1, & \text{dacă } n \geq 2. \end{cases} \end{aligned}$$

Rescriind această relație de recurență sub forma

$$M(n) + 1 = 2(M(n-1) + 1), \quad \forall n \geq 2,$$

obținem că șirul  $(M(n) + 1)_{n \geq 1}$  este o progresie geometrică având rația 2 și primul termen  $M(1) + 1 = 2$ , deci  $M(n) + 1 = 2^n$ ,  $\forall n \geq 1$  și astfel se obține că

$$M(n) = 2^n - 1, \quad \forall n \geq 1.$$

Rezultă că timpul de execuție al algoritmului este

$$T(n) = \Theta(2^n),$$

deci algoritmul este exponențial.

**Propoziția 4.2.1.** *Orice algoritm care rezolvă problema turnurilor din Hanoi necesită cel puțin  $2^n - 1$  mutări.*

*Demonstrație.* Demonstrăm afirmația din enunț prin inducție după  $n \in \mathbb{N}^*$ .

Pentru  $n = 1$ , este evident că numărul de mutări necesare este de cel puțin  $1 = 2^1 - 1$ .

Presupunem afirmația adevărată pentru  $n - 1$  și o demonstrăm pentru  $n$ ,  $n \geq 2$ . Într-adevăr, pentru orice rezolvare a problemei, sunt necesare cel puțin următoarele subetape:

- o mutare a discului  $n$  (cel mai mare) de pe tija 1 pe o altă tijă  $i$ ;
- această mutare trebuie obligatoriu precedată de mutarea primelor  $n - 1$  discuri de pe tija 1 pe tija rămasă  $6 - 1 - i$ , deci, conform ipotezei de inducție sunt necesare cel puțin  $2^{n-1} - 1$  mutări;
- o mutare a discului  $n$  (cel mai mare) de pe o tijă  $j \neq 2$  pe tija 2, mutare ce poate să coincidă cu mutarea descrisă la prima subetapă;

- această mutare trebuie obligatoriu urmată de mutarea primelor  $n - 1$  discuri de pe tija rămasă  $6 - 2 - j$  pe tija 2, deci, conform ipotezei de inducție sunt necesare încă cel puțin  $2^{n-1} - 1$  mutări.

Astfel numărul total de mutări necesare este de cel puțin

$$1 + 2(2^{n-1} - 1) = 2^n - 1.$$

□

Următorul rezultat este o consecință directă a Observației 4.2.1 și propoziției anterioare.

**Propoziția 4.2.2.** *Algoritmul 4.2.1 este optim.*

### 4.3 O problemă de acoperire

Considerăm următoarea **problemă de acoperire**:

Se consideră o suprafață pătrată cu lungimea laturii  $n = 2^k$ ,  $k \geq 1$ , divizată în pătrate unitare, ca o tablă de șah. Unul dintre aceste pătrate unitare este deja acoperit.

Se cere să se determine o acoperire a suprafeței rămase cu plăci de forma



constând din trei pătrate unitare.

#### Modelarea problemei

Memorăm suprafața pătrată într-o matrice  $S = (s_{ij})_{i,j=\overline{1,n}}$ , fiecare pătrat unitar fiind deci reprezentat de un element al acestei matrice.

Evident, o acoperire cu piese de forma dată utilizează un număr de

$$\frac{n^2 - 1}{3} = \frac{4^k - 1}{3}$$

piese.

Considerăm că piesele sunt numerotate cu  $1, 2, \dots, \frac{n^2 - 1}{3}$  și că așezarea piesei numărul  $i$  constă în completarea elementelor corespunzătoare din matricea  $S$  cu numărul  $i$  al piesei, pentru fiecare valoare  $i \in \left\{1, 2, \dots, \frac{n^2 - 1}{3}\right\}$ .

Pentru rezolvarea problemei utilizăm următoarea strategie *Divide et Impera*:

- Se împarte suprafața pătrată în patru pătrate egale, formându-se patru suprafețe pătrate cu lungimea laturii  $2^{k-1}$ .
- Una dintre cele patru suprafețe are deja un pătrat unitar acoperit.
- Acoperim cu o placă trei pătrate unitare din centrul suprafeței mari astfel încât să acoperim câte un pătrat unitar și din celelalte trei suprafețe. Un exemplu în acest sens este redat în Figura 4.3.1.

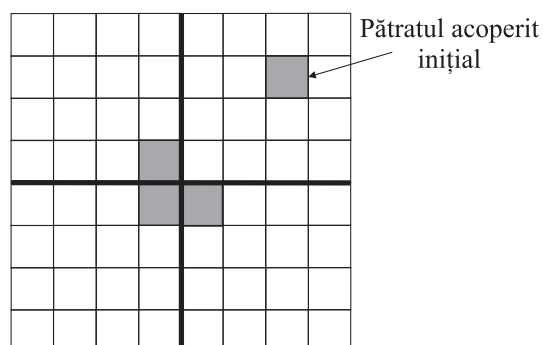


Figura 4.3.1:

- Fiecare dintre cele patru suprafețe este acum în condițiile problemei, deci pentru fiecare suprafață se repetă procedeul de descompunere descris anterior.

Obținem următorul *algorithm Divide et Impera*, descris în pseudocod, pentru rezolvarea problemei.

Algoritmul 4.3.1.

```

ACOPERIRE( $S, i, j, n, ip, jp$ ):
    //  $S$ =matricea ce va fi completată cu numerele plăcilor
    //  $(i, j)$ =colțul stânga sus
    //  $n$ =lungimea laturii
    //  $(ip, jp)$  = coordonatele pătratului unitar acoperit
    if ( $n > 1$ ) then
         $p \leftarrow p + 1$ ; // numărul plăcii
         $m \leftarrow n/2$ ; // noua dimensiune a laturii
        // Coordonatele stânga sus ale celor 4 pătrate:
         $i1 \leftarrow i$ ;  $j1 \leftarrow j$ ;
         $i2 \leftarrow i$ ;  $j2 \leftarrow j + m$ ;
         $i3 \leftarrow i + m$ ;  $j3 \leftarrow j$ ;
         $i4 \leftarrow i + m$ ;  $j4 \leftarrow j + m$ ;
        // Coordonatele pătratelor unitare ce vor fi acoperite
        // în cele 4 pătrate:
         $ip1 \leftarrow ip$ ;  $jp1 \leftarrow jp$ ;
         $ip2 \leftarrow ip$ ;  $jp2 \leftarrow jp$ ;
         $ip3 \leftarrow ip$ ;  $jp3 \leftarrow jp$ ;
         $ip4 \leftarrow ip$ ;  $jp4 \leftarrow jp$ ;
        if ( $i3 \leq ip$ ) then
             $ip1 \leftarrow i1 + m - 1$ ;  $jp1 \leftarrow j1 + m - 1$ ;
             $ip2 \leftarrow i2 + m - 1$ ;  $jp2 \leftarrow j2$ ;
             $S[ip1][jp1] \leftarrow p$ ;  $S[ip2][jp2] \leftarrow p$ ;
        else
             $ip3 \leftarrow i3$ ;  $jp3 \leftarrow j3 + m - 1$ ;
             $ip4 \leftarrow i4$ ;  $jp4 \leftarrow j4$ ;
             $S[ip3][jp3] \leftarrow p$ ;  $S[ip4][jp4] \leftarrow p$ ;
        if ( $j2 \leq jp$ ) then
             $ip1 \leftarrow i1 + m - 1$ ;  $jp1 \leftarrow j1 + m - 1$ ;
             $ip3 \leftarrow i3$ ;  $jp3 \leftarrow j3 + m - 1$ ;
             $S[ip1][jp1] \leftarrow p$ ;  $S[ip3][jp3] \leftarrow p$ ;
        else
             $ip2 \leftarrow i2 + m - 1$ ;  $jp2 \leftarrow j2$ ;
             $ip4 \leftarrow i4$ ;  $jp4 \leftarrow j4$ ;
             $S[ip2][jp2] \leftarrow p$ ;  $S[ip4][jp4] \leftarrow p$ ;
        ACOPERIRE( $S, i1, j1, m, ip1, jp1$ );
        ACOPERIRE( $S, i2, j2, m, ip2, jp2$ );
        ACOPERIRE( $S, i3, j3, m, ip3, jp3$ );
        ACOPERIRE( $S, i4, j4, m, ip4, jp4$ );

```

Apel: **ACOPERIRE**( $S, 1, 1, n, ip, jp$ ).

*Exemplul 4.3.1.* Pentru exemplul din Figura 4.3.1, aplicarea algoritmului anterior produce acoperirea dată de matricea

$$S = \begin{pmatrix} 3 & 3 & 4 & 4 & 8 & 8 & 9 & 9 \\ 3 & 2 & 2 & 4 & 8 & 7 & 0 & 9 \\ 5 & 2 & 6 & 6 & 10 & 7 & 7 & 11 \\ 5 & 5 & 6 & 1 & 10 & 10 & 11 & 11 \\ 13 & 13 & 14 & 1 & 1 & 18 & 19 & 19 \\ 13 & 12 & 14 & 14 & 18 & 18 & 17 & 19 \\ 15 & 12 & 12 & 16 & 20 & 17 & 17 & 21 \\ 15 & 15 & 16 & 16 & 20 & 20 & 21 & 21 \end{pmatrix}.$$

*Observația 4.3.1.* Algoritmul 4.3.1 are complexitatea  $\Theta(n^2)$ .

## 4.4 Căutarea binară

**Problema căutării** este următoarea:

Se consideră un vector  $A = (a_1, a_2, \dots, a_n)$ ,  $n \geq 1$ , și o valoare  $x$ . Se cere să se determine dacă  $x$  se află printre componentele vectorului  $A$ .

O căutare se poate încheia:

- fie **cu succes**, caz în care se returnează o poziție în vector pe care se află valoarea căutată;
- fie **fără succes**, caz în care se returnează  $-1$ .

*Observația 4.4.1.* În practică, problema căutării constă în regăsirea unei înregistrări într-un fișier sau bază de date după valoarea unui câmp, numit **cheie de căutare**.

*Algoritmul 4.4.1 (Căutarea binară).*

Presupunem că vectorul  $A$  are elementele în ordine crescătoare

$$a_1 \leq a_2 \leq \dots \leq a_n.$$

Vom utiliza metoda *Divide et Impera* astfel:

- se compară  $x$  cu elementul median (din "mijloc")  $a_m$ , unde

$$m = \left\lfloor \frac{1+n}{2} \right\rfloor;$$

- dacă  $a_m = x$ , atunci căutarea se termină cu succes;

- dacă  $a_m > x$ , atunci se va căuta  $x$  în subsecvența

$$(a_1, \dots, a_{m-1});$$

- dacă  $a_m < x$ , atunci se va căuta  $x$  în subsecvența

$$(a_{m+1}, \dots, a_n).$$

**Varianta recursivă** de implementare a algoritmului de căutare binară este descrisă în pseudocod astfel:

**CAUTBIN\_REC**( $A, n, p, u, x$ ):

// căutarea valorii  $x$  în subsecvența  $(a_p, \dots, a_u)$  a  
// vectorului  $(a_1, \dots, a_n)$

**if**  $p > u$  **then**

| **returnează**  $-1$ ;

**else**

|  $m \leftarrow \left\lfloor \frac{p+u}{2} \right\rfloor$ ;

| **if**  $x = A[m]$  **then**

| **returnează**  $m$ ;

| **else**

| **if**  $x < A[m]$  **then**

| **returnează** **CAUTBIN\_REC**( $A, n, p, m-1, x$ );

| **else**

| **returnează** **CAUTBIN\_REC**( $A, n, m+1, u, x$ );

Procedura trebuie apelată prin:

$$poz \leftarrow \mathbf{CAUTBIN\_REC}(A, n, 1, n, x).$$

**Varianta iterativă** de implementare a algoritmului de căutare binară



este descrisă în pseudocod astfel:

```

CAUTBIN( $A, n, x$ ):
 $p \leftarrow 1; u \leftarrow n;$     // ( $a_p, \dots, a_u$ ) reprezintă subsecvența curentă
                                // la care se restrânge căutarea

while  $p \leq u$  do
     $m \leftarrow \left\lfloor \frac{p+u}{2} \right\rfloor;$ 
    if  $x = A[m]$  then
        | returnează  $m;$ 
    else
        | if  $x < A[m]$  then
            | |  $u \leftarrow m - 1;$ 
        | else
            | |  $p \leftarrow m + 1;$ 
    returnează  $-1;$ 

```

Procedura trebuie apelată prin:

$$poz \leftarrow \mathbf{CAUTBIN}(A, n, x).$$

*Observația 4.4.2.* Algoritmul de căutare binară într-un vector sortat cu  $n$  componente are complexitatea  $\Theta(\log_2 n)$ .

## 4.5 Algoritmi de sortare

### 4.5.1 Problema sortării

**Problema sortării** este următoarea:

Se consideră un vector  $A = (a_1, a_2, \dots, a_n)$ ,  $n \geq 1$ . Se cere să se ordoneze crescător (sau descrescător) elementele acestui vector.

*Observația 4.5.1.* În practică, problema apare, în general, în ordonarea crescătoare (sau descrescătoare) a unei mulțimi finite de articole (înregistrări) după un câmp (sau mai multe) numit **cheie de sortare**.

### 4.5.2 Interclasarea a doi vectori

**Problema interclasării** este următoarea:

Se consideră doi vectori

$$\begin{aligned}
 A &= (a_1, a_2, \dots, a_m), \quad m \geq 1, \\
 B &= (b_1, b_2, \dots, b_n), \quad n \geq 1,
 \end{aligned}$$

ale căror componente sunt sortate (ordonate) crescător, adică

$$\begin{aligned} a_1 &\leq a_2 \leq \dots \leq a_m, \\ b_1 &\leq b_2 \leq \dots \leq b_n. \end{aligned}$$

Se cere să se construiască un vector cu  $m + n$  componente

$$C = (c_1, c_2, \dots, c_{m+n}),$$

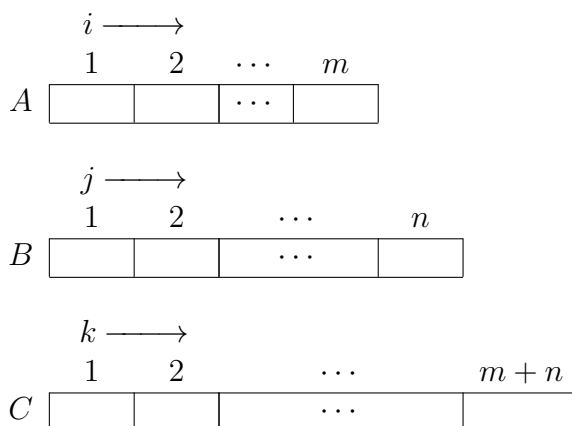
care să conțină toate componentele vectorilor  $A$  și  $B$  și care să fie, de asemenea, *sortat crescător*, adică

$$c_1 \leq c_2 \leq \dots \leq c_{m+n}.$$

**Definiția 4.5.1.** Spunem că elementele vectorului  $C$  se obțin prin **interclasarea** (intercalarea) elementelor vectorilor  $A$  și  $B$ .

*Algoritmul 4.5.1 (Interclasarea a doi vectori).*

- Pentru parcurgerea componentelor celor trei vectori  $A$ ,  $B$  și  $C$  vom utiliza trei indici  $i$ ,  $j$  și respectiv  $k$ .
- Inițial toți acești trei indici au valoarea 1.



- Se compară elementele curente din  $A$  și  $B$  (adică  $a_i$  cu  $b_j$ ), iar cel mai mic se depune în  $C$  pe poziția  $k$ .
- Se incrementează  $k$  și indicele corespunzător vectorului din care s-a făcut depunerea ( $i$  sau  $j$ ).
- Procesul continuă până când este epuizat unul dintre vectori. Elementele rămase în celălalt vector se vor adăuga, în ordinea în care se află, la sfârșitul lui  $C$ .

Descrierea în pseudocod a algoritmului are următoarea formă.

**INTERCLASARE**( $A, B, C, m, n$ ) :

$i \leftarrow 1; j \leftarrow 1; k \leftarrow 1;$

**while** ( $i \leq m$  and  $j \leq n$ ) **do** // (1)

**if**  $A[i] \leq B[j]$  **then**

$C[k] \leftarrow A[i];$

$i \leftarrow i + 1;$

**else**

$C[k] \leftarrow B[j];$

$j \leftarrow j + 1;$

$k \leftarrow k + 1;$

**while**  $i \leq m$  **do**

$C[k] \leftarrow A[i];$

$i \leftarrow i + 1;$

$k \leftarrow k + 1;$

**while**  $j \leq n$  **do**

$C[k] \leftarrow B[j];$

$j \leftarrow j + 1;$

$k \leftarrow k + 1;$

**Propoziția 4.5.1 (complexitatea algoritmului de interclasare).** Algoritmul de interclasare a doi vectori cu  $m$  și respectiv  $n$  componente are complexitatea  $\Theta(m + n)$ .

*Demonstrație.* Fie  $T(m, n)$  timpul de execuție al algoritmului.

Vom număra numai operațiile de comparație și de atribuire în care intervin elemente ale vectorilor. Celelalte operații care se efectuează au același ordin de creștere cu cele pe care le analizăm.

- Numărul de atribuirii în care intervin elemente ale vectorilor, numite și **deplasări**, notat cu  $N_d(m, n)$ , este egal cu numărul de componente ale lui  $C$ , deci

$$N_d(m, n) = m + n. \quad (4.5.1)$$

- Numărul de **comparații de chei** (comparații în care intervin elemente ale vectorilor), notat cu  $N_c(m, n)$ , este egal cu numărul de execuții ale ciclului **while** numerotat cu (1). Cu fiecare astfel de comparație are loc și o atribuire pentru o componentă a vectorului  $C$ , adică o *deplasare*. La terminarea ciclului **while** numerotat cu (1) elementele unuia dintre vectorii  $A$  sau  $B$  au fost depuse integral în  $C$ , iar în celălalt vector au

mai rămas  $r$  elemente nedepuse în  $C$ , care vor fi depuse în  $C$  după încheierea acestui ciclu. Rezultă că

$$N_c(m, n) = m + n - r.$$

Evident,

$$1 \leq r \leq \max\{m, n\},$$

deci

$$\underbrace{\min\{m, n\}}_{\substack{\text{în cazul} \\ \text{cel mai favorabil}}} \leq N_c(m, n) \leq \underbrace{m + n - 1}_{\substack{\text{în cazul} \\ \text{cel mai defavorabil}}}. \quad (4.5.2)$$

Notând cu  $c_1, c_2 > 0$  constantele care reprezintă costurile în timp ale executării unei *comparații de chei* și, respectiv, ale unei *deplasări*, obținem că

$$T(m, n) = c_1 \cdot N_c(m, n) + c_2 \cdot N_d(m, n).$$

Utilizând (4.5.1) și (4.5.2) rezultă că

$$c_1 \cdot \min\{m, n\} + c_2 \cdot (m + n) \leq T(m, n) \leq c_1 \cdot (m + n - 1) + c_2 \cdot (m + n).$$

Deducem că

$$c_2 \cdot (m + n) \leq T(m, n) \leq (c_1 + c_2) \cdot (m + n),$$

deci  $T(m, n) = \Theta(m + n)$ . □

*Observația 4.5.2.* În contextul demonstrației anterioare, pentru orice  $r \in \{1, 2, \dots, \max\{m, n\}\}$  avem

$$\begin{aligned} T(m, n) &= c_1 \cdot N_c(m, n) + c_2 \cdot (m + n) \\ &= c_1 \cdot (m + n - r) + c_2 \cdot (m + n) \\ &= (c_1 + c_2) \cdot (m + n) - c_1 \cdot r. \end{aligned}$$

Pentru a avea cazuri elementare egal posibile (egal probabile), trebuie să luăm separat în calcul cazurile în care cele  $r$  elemente nedepuse în  $C$  sunt din  $A$ , deci  $r \in \{1, 2, \dots, m\}$ , sau din  $B$ , deci  $r \in \{1, 2, \dots, n\}$ .

Astfel putem calcula efectiv timpul mediu de execuție astfel:

$$\begin{aligned}
 T_{\text{mediu}}(m, n) &= \frac{1}{m+n} \left\{ \sum_{r=1}^m ((c_1 + c_2) \cdot (m+n) - c_1 \cdot r) + \right. \\
 &\quad \left. + \sum_{r=1}^n ((c_1 + c_2) \cdot (m+n) - c_1 \cdot r) \right\} \\
 &= (c_1 + c_2) \cdot (m+n) - c_1 \cdot \frac{m(m+1) + n(n+1)}{2(m+n)} \\
 &= c_1 \cdot \left( \frac{m+n-1}{2} + \frac{m \cdot n}{m+n} \right) + c_2 \cdot (m+n).
 \end{aligned}$$

*Observația 4.5.3.* Evident, orice algoritm care rezolvă problema dată necesită cel puțin  $m+n$  instrucțiuni de atribuire pentru completarea elementelor vectorului  $C$ , deci conform propoziției anterioare rezultă că algoritmul de interclasare este asimptotic-optimal.

### 4.5.3 Sortarea prin interclasare (mergesort)

*Algoritmul 4.5.2 (Sortarea prin interclasare).* Pentru a sorta crescător vectorul  $A = (a_1, a_2, \dots, a_n)$ ,  $n \geq 1$ , se interclasează succesiv secvențe ordonate crescător ale vectorului, până se obține secvența cu toate elementele sortate ale vectorului.

Se utilizează metoda *Divide et Impera*:

- pentru a sorta un vector cu  $n$  componente îl împărțim în doi vectori care odată sortați se interclasează;
- un vector cu o singură componentă este sortat.

Descrierea în pseudocod a algoritmului are următoarea formă.

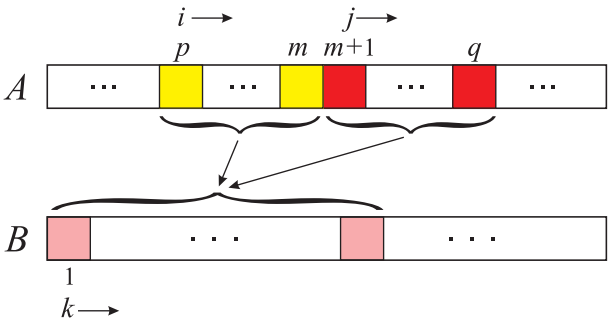
```

SORTINT( $A, p, q$ ):           // sortarea secvenței ( $a_p, \dots, a_q$ )
if  $p < q$  then
     $m \leftarrow \left\lfloor \frac{p+q}{2} \right\rfloor$ ;
    SORTINT( $A, p, m$ );
    SORTINT( $A, m+1, q$ );
    INTERCLAS( $A, p, q, m$ );

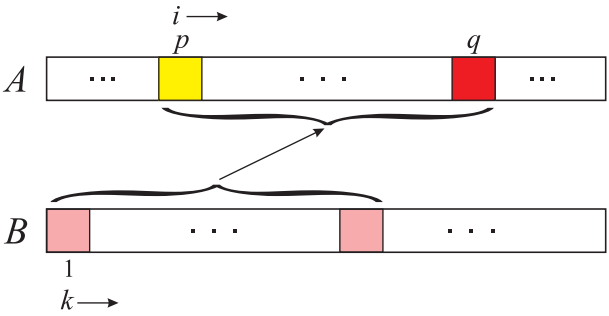
```

Apel: **SORTINT** ( $A, 1, n$ ).

Pentru interclasare vom utiliza un vector intermediar, de lucru,  $B$ . Mai întâi se interclasează cele două secvențe ( $a_p, \dots, a_m$ ) și ( $a_{m+1}, \dots, a_q$ ), rezultatul interclasării depunându-se în vectorul  $B$ :



Apoi rezultatul interclasării va fi copiat din  $B$  în  $A$ , între pozițiile  $p$  și  $q$ :



Descrierea în pseudocod a procedurii de interclasare are următoarea formă.

```

INTERCLAS( $A, p, q, m$ ) :
   $i \leftarrow p$ ;
   $j \leftarrow m + 1$ ;
   $k \leftarrow 1$ ;
  while ( $i \leq m$ ) and ( $j \leq q$ ) do
    if  $A[i] \leq A[j]$  then
       $B[k] \leftarrow A[i]$ ;
       $i \leftarrow i + 1$ ;
    else
       $B[k] \leftarrow A[j]$ ;
       $j \leftarrow j + 1$ ;
     $k \leftarrow k + 1$ ;
  while  $i \leq m$  do
     $B[k] \leftarrow A[i]$ ;
     $i \leftarrow i + 1$ ;
     $k \leftarrow k + 1$ ;
  while  $j \leq q$  do
     $B[k] \leftarrow A[j]$ ;
     $j \leftarrow j + 1$ ;
     $k \leftarrow k + 1$ ;
   $k \leftarrow 1$ ;
  for  $i = \overline{p, q}$  do
     $A[i] \leftarrow B[k]$ ;
     $k \leftarrow k + 1$ ;

```

**Teorema 4.5.1 (complexitatea algoritmului de sortare prin interclasare).** *Algoritmul de sortare prin interclasare a unui vector cu  $n$  componente are complexitatea  $\Theta(n \log_2 n)$ .*

*Demonstrație.* Fie  $T(n)$  timpul de execuție al algoritmului de sortare prin interclasare. Vom estima efectiv timpul de execuție  $T(n)$ . Ca și la analiza complexității algoritmului de interclasare a doi vectori (Propoziția 4.5.1), vom număra numai operațiile de *comparație de chei* (comparații în care intervin elemente ale vectorului) și de *deplasare* (atribuiri în care intervin elemente ale vectorului). Celelalte operații care se efectuează au același ordin de creștere cu cele pe care le analizăm.

Avem

$$T(n) = c_1 \cdot N_c(n) + c_2 \cdot N_d(n), \quad (4.5.3)$$

unde  $N_c(n)$  și  $N_d(n)$  reprezintă numărul de *comparații de chei*, respectiv numărul de *deplasări* efectuate de algoritm, iar  $c_1, c_2 > 0$  sunt constante ce

reprezintă costurile în timp ale executării unei *comparații de chei* și, respectiv, ale unei *deplasări*.

Conform descrierii recursive a algoritmului, rezultă că numerele  $N_c(n)$  și  $N_d(n)$  verifică, respectiv, relațiile de recurență

$$N_c(n) = \begin{cases} 0, & \text{dacă } n = 1, \\ N_c\left(\left\lceil \frac{n}{2} \right\rceil\right) + N_c\left(\left\lfloor \frac{n}{2} \right\rfloor\right) + \overline{N}_c\left(\left\lceil \frac{n}{2} \right\rceil, \left\lfloor \frac{n}{2} \right\rfloor\right), & \text{dacă } n > 1, \end{cases} \quad (4.5.4)$$

$$N_d(n) = \begin{cases} 0, & \text{dacă } n = 1, \\ N_d\left(\left\lceil \frac{n}{2} \right\rceil\right) + N_d\left(\left\lfloor \frac{n}{2} \right\rfloor\right) + \overline{N}_d\left(\left\lceil \frac{n}{2} \right\rceil, \left\lfloor \frac{n}{2} \right\rfloor\right), & \text{dacă } n > 1, \end{cases} \quad (4.5.5)$$

unde  $\overline{N}_c\left(\left\lceil \frac{n}{2} \right\rceil, \left\lfloor \frac{n}{2} \right\rfloor\right)$  și  $\overline{N}_d\left(\left\lceil \frac{n}{2} \right\rceil, \left\lfloor \frac{n}{2} \right\rfloor\right)$  reprezintă numărul de *comparații de chei*, respectiv numărul de *deplasări* efectuate la interclasarea celor doi vectori sortați (de dimensiuni  $\left\lceil \frac{n}{2} \right\rceil$  și  $\left\lfloor \frac{n}{2} \right\rfloor$ ).

Conform demonstrației Propoziției 4.5.1, și anume relațiilor (4.5.1) și (4.5.2), avem

$$\overline{N}_d\left(\left\lceil \frac{n}{2} \right\rceil, \left\lfloor \frac{n}{2} \right\rfloor\right) = \left\lceil \frac{n}{2} \right\rceil + \left\lfloor \frac{n}{2} \right\rfloor = n, \quad (4.5.6)$$

$$\overline{N}_c\left(\left\lceil \frac{n}{2} \right\rceil, \left\lfloor \frac{n}{2} \right\rfloor\right) \in \left\{ \left\lfloor \frac{n}{2} \right\rfloor, \left\lfloor \frac{n}{2} \right\rfloor + 1, \dots, n-1 \right\}. \quad (4.5.7)$$

Notând cu  $N_c^+(n)$  numărul de *comparații de chei* în cazul cel mai defavorabil, din (4.5.4) și (4.5.7) rezultă relația de recurență

$$N_c^+(n) = \begin{cases} 0, & \text{dacă } n = 1, \\ N_c^+\left(\left\lceil \frac{n}{2} \right\rceil\right) + N_c^+\left(\left\lfloor \frac{n}{2} \right\rfloor\right) + n - 1, & \text{dacă } n > 1. \end{cases} \quad (4.5.8)$$

Rezolvăm această relație de recurență.

Pentru  $n =$  număr par, adică  $n = 2m$ ,  $m \geq 1$ , avem

$$\begin{aligned} N_c^+(n+1) - N_c^+(n) &= N_c^+(2m+1) - N_c^+(2m) \\ &= (N_c^+(m+1) + N_c^+(m) + 2m) - (N_c^+(m) + N_c^+(m) + 2m - 1) \\ &= N_c^+(m+1) - N_c^+(m) + 1, \end{aligned}$$

iar pentru  $n =$  număr impar, adică  $n = 2m+1$ ,  $m \geq 1$ , avem

$$\begin{aligned} N_c^+(n+1) - N_c^+(n) &= N_c^+(2m+2) - N_c^+(2m+1) \\ &= (N_c^+(m+1) + N_c^+(m+1) + 2m+1) - (N_c^+(m+1) + N_c^+(m) + 2m) \\ &= N_c^+(m+1) - N_c^+(m) + 1, \end{aligned}$$



deci pentru orice  $n > 1$  avem

$$N_c^+(n+1) - N_c^+(n) = N_c^+\left(\left\lfloor \frac{n}{2} \right\rfloor + 1\right) - N_c^+\left(\left\lfloor \frac{n}{2} \right\rfloor\right) + 1.$$

Notând

$$u(n) = N_c^+(n+1) - N_c^+(n), \quad \forall n \in \mathbb{N}^*, \quad (4.5.9)$$

obținem că

$$u(n) = \begin{cases} 1, & \text{dacă } n = 1, \\ u\left(\left\lfloor \frac{n}{2} \right\rfloor\right) + 1, & \text{dacă } n > 1. \end{cases} \quad (4.5.10)$$

Demonstrăm că

$$u(n) = \lceil \log_2(n+1) \rceil, \quad \forall n \in \mathbb{N}^*, \quad (4.5.11)$$

prin inducție.

Pentru  $n = 1$  avem  $u(1) = 1 = \lceil \log_2 2 \rceil$ .

Presupunem egalitatea adevărată pentru orice  $k \in \{1, \dots, n-1\}$  și o demonstrăm pentru  $n$  ( $n \geq 2$ ). Într-adevăr, folosind relația de recurență (4.5.10) și ipoteza de inducție pentru  $\left\lfloor \frac{n}{2} \right\rfloor$  avem

$$u(n) = u\left(\left\lfloor \frac{n}{2} \right\rfloor\right) + 1 = \lceil \log_2\left(\left\lfloor \frac{n}{2} \right\rfloor + 1\right) \rceil + 1.$$

Dar, notând  $\lceil \log_2(n+1) \rceil = k$ , avem  $k \in \mathbb{N}^*$ ,  $k \geq 2$  și, succesiv:

$$\begin{aligned} k-1 &< \log_2(n+1) \leq k; \quad 2^{k-1} < n+1 \leq 2^k; \\ 2^{k-1} &\leq n < 2^k; \quad 2^{k-2} \leq \frac{n}{2} < 2^{k-1}; \\ 2^{k-2} &\leq \left\lfloor \frac{n}{2} \right\rfloor < 2^{k-1}; \quad 2^{k-2} < \left\lfloor \frac{n}{2} \right\rfloor + 1 \leq 2^{k-1}; \\ k-2 &< \log_2\left(\left\lfloor \frac{n}{2} \right\rfloor + 1\right) \leq k-1; \\ \lceil \log_2\left(\left\lfloor \frac{n}{2} \right\rfloor + 1\right) \rceil &= k-1, \end{aligned}$$

deci

$$u(n) = k-1+1 = k = \lceil \log_2(n+1) \rceil,$$

ceea ce încheie demonstrația prin inducție a relației (4.5.11).

Din (4.5.9) și (4.5.11) rezultă că

$$N_c^+(n+1) - N_c^+(n) = \lceil \log_2(n+1) \rceil, \quad \forall n \in \mathbb{N}^*.$$

Aplicând succesiv această relație avem

$$\begin{aligned}
 N_c^+(n) &= N_c^+(n-1) + \lceil \log_2 n \rceil \\
 &= N_c^+(n-2) + \lceil \log_2(n-1) \rceil + \lceil \log_2 n \rceil \\
 &\dots \\
 &= N_c^+(1) + \lceil \log_2 2 \rceil + \lceil \log_2 3 \rceil + \dots + \lceil \log_2 n \rceil.
 \end{aligned}$$

Cum  $N_c^+(1) = 0$  obținem că

$$N_c^+(n) = \lceil \log_2 1 \rceil + \lceil \log_2 2 \rceil + \dots + \lceil \log_2 n \rceil, \quad \forall n \in \mathbb{N}^*. \quad (4.5.12)$$

Fie  $n \geq 2$  și  $p = \lceil \log_2 n \rceil$ , deci  $p \in \mathbb{N}^*$  și  $p-1 < \log_2 n \leq p$ , deci  $2^{p-1} < n \leq 2^p$ .

Pentru orice  $i \in \{2, 3, \dots, n\}$  avem echivalențele

$$\lceil \log_2 i \rceil = j \Leftrightarrow j-1 < \log_2 i \leq j \Leftrightarrow 2^{j-1} < i \leq 2^j \Leftrightarrow 2^{j-1} + 1 \leq i \leq 2^j.$$

Astfel avem

$$\begin{aligned}
 N_c^+(n) &= \sum_{i=2}^n \lceil \log_2 i \rceil = \sum_{j=1}^{p-1} \sum_{i=2^{j-1}+1}^{2^j} \lceil \log_2 i \rceil + \sum_{i=2^{p-1}+1}^n \lceil \log_2 i \rceil \\
 &= \sum_{j=1}^{p-1} \sum_{i=2^{j-1}+1}^{2^j} j + \sum_{i=2^{p-1}+1}^n p = \sum_{j=1}^{p-1} j(2^j - 2^{j-1}) + p(n - 2^{p-1}) \\
 &= \sum_{j=1}^{p-1} j \cdot 2^{j-1} + p(n - 2^{p-1}) = p \cdot 2^{p-1} - 2^p + 1 + p(n - 2^{p-1}) \\
 &= np - 2^p + 1,
 \end{aligned}$$

deci

$$N_c^+(n) = \sum_{i=2}^n \lceil \log_2 i \rceil = n \lceil \log_2 n \rceil - 2^{\lceil \log_2 n \rceil} + 1, \quad \forall n \in \mathbb{N}^*, \quad (4.5.13)$$

egalitatea fiind evident valabilă și pentru  $n = 1$ . Dar

$$\log_2 n \leq \lceil \log_2 n \rceil < \log_2 n + 1,$$

deci

$$n \leq 2^{\lceil \log_2 n \rceil} < 2n.$$

Astfel rezultă că în cazul cel mai defavorabil numărul de comparații de chei verifică inegalitățile

$$n \log_2 n - 2n + 1 < N_c^+(n) < n \log_2 n + 1, \quad \forall n \in \mathbb{N}^*. \quad (4.5.14)$$

Notând cu  $N_c^-(n)$  numărul de *comparații de chei* în cazul cel mai favorabil, din (4.5.4) și (4.5.7) rezultă relația de recurență

$$N_c^-(n) = \begin{cases} 0, & \text{dacă } n = 1, \\ N_c^-\left(\left\lceil \frac{n}{2} \right\rceil\right) + N_c^-\left(\left\lfloor \frac{n}{2} \right\rfloor\right) + \left\lfloor \frac{n}{2} \right\rfloor, & \text{dacă } n > 1. \end{cases} \quad (4.5.15)$$

Demonstrăm că șirul  $(N_c^-(n))_{n \geq 1}$  este crescător, adică

$$N_c^-(n+1) \geq N_c^-(n), \quad \forall n \in \mathbb{N}^*,$$

prin inducție.

Pentru  $n = 1$  avem  $N_c^-(2) = 1 > 0 = N_c^-(1)$ .

Presupunem inegalitatea adevărată pentru orice  $k \in \{1, \dots, n-1\}$  și o demonstrăm pentru  $n$  ( $n \geq 2$ ). Într-adevăr, folosind relația de recurență (4.5.15), ipoteza de inducție pentru  $\left\lceil \frac{n}{2} \right\rceil$  când  $n$  este par, respectiv pentru  $\left\lfloor \frac{n}{2} \right\rfloor$  când  $n$  este impar, precum și inegalitatea evidentă  $\left\lfloor \frac{n+1}{2} \right\rfloor \geq \left\lfloor \frac{n}{2} \right\rfloor$ , avem

$$\begin{aligned} N_c^-(n+1) &= N_c^-\left(\left\lceil \frac{n+1}{2} \right\rceil\right) + N_c^-\left(\left\lfloor \frac{n+1}{2} \right\rfloor\right) + \left\lfloor \frac{n+1}{2} \right\rfloor \\ &\geq N_c^-\left(\left\lceil \frac{n}{2} \right\rceil\right) + N_c^-\left(\left\lfloor \frac{n}{2} \right\rfloor\right) + \left\lfloor \frac{n}{2} \right\rfloor \\ &= N_c^-(n). \end{aligned}$$

Demonstrația prin inducție este încheiată.

Fie  $n \in \mathbb{N}^*$  și

$$r = \lfloor \log_2 n \rfloor, \quad (4.5.16)$$

deci  $r \in \mathbb{N}$  și  $r \leq \log_2 n < r+1$ , deci  $2^r \leq n < 2^{r+1}$ . Deoarece șirul  $(N_c^-(n))_{n \geq 1}$  este crescător, rezultă că

$$N_c^-(n) \geq N_c^-(2^r). \quad (4.5.17)$$

Dar, conform relației de recurență (4.5.15) avem

$$N_c^-(2^r) = 2N_c^-(2^{r-1}) + 2^{r-1}, \quad \forall r \in \mathbb{N}^*.$$

Rescriind această egalitate sub forma

$$N_c^-(2^r) - r \cdot 2^{r-1} = 2(N_c^-(2^{r-1}) - (r-1)2^{r-2}), \quad \forall r \in \mathbb{N}^*,$$

Aplicând succesiv această relație avem

$$\begin{aligned} N_c^-(2^r) - r \cdot 2^{r-1} &= 2 (N_c^-(2^{r-1}) - (r-1)2^{r-2}) \\ &= 2^2 (N_c^-(2^{r-2}) - (r-2)2^{r-3}) \\ &= \dots \\ &= 2^r (N_c^-(2^{r-r}) - (r-r)2^{r-r-1}) \end{aligned}$$

Cum  $N_c^-(1) = 0$  obținem că  $N_c^-(2^r) - r \cdot 2^{r-1} = 0$ , deci

$$N_c^-(2^r) = r \cdot 2^{r-1}, \quad \forall r \in \mathbb{N}. \quad (4.5.18)$$

Din (4.5.17), (4.5.18) și (4.5.16) rezultă că în *cazul cel mai favorabil* numărul de *comparații de chei* verifică inegalitățile

$$N_c^-(n) \geq \lfloor \log_2 n \rfloor \cdot 2^{\lfloor \log_2 n \rfloor - 1} \geq \frac{1}{2} \cdot n \log_2 n, \quad \forall n \in \mathbb{N}^*. \quad (4.5.19)$$

Din (4.5.14) și (4.5.19) rezultă că numărul de *comparații de chei* verifică inegalitățile

$$\frac{1}{2} \cdot n \log_2 n \leq N_c(n) < n \log_2 n + 1, \quad \forall n \in \mathbb{N}^*. \quad (4.5.20)$$

Din relațiile (4.5.5) și (4.5.6) rezultă că numărul de *deplasări* efectuate de algoritm verifică relația de recurență

$$N_d(n) = \begin{cases} 0, & \text{dacă } n = 1, \\ N_d\left(\left\lceil \frac{n}{2} \right\rceil\right) + N_d\left(\left\lfloor \frac{n}{2} \right\rfloor\right) + n, & \text{dacă } n > 1. \end{cases}$$

Această relație se rezolvă analog relației (4.5.8), obținându-se

$$N_d(n) = n \lceil \log_2 n \rceil - 2^{\lceil \log_2 n \rceil} + n, \quad \forall n \in \mathbb{N}^*,$$

deci

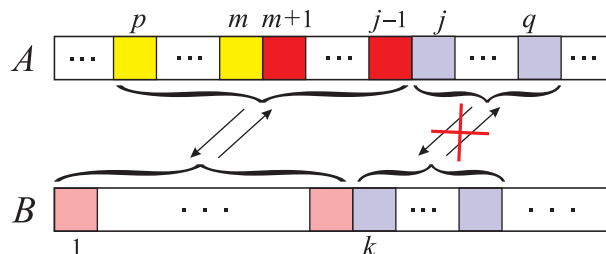
$$n \log_2 n - 2n + n < N_d(n) < n \log_2 n + n, \quad \forall n \in \mathbb{N}^*. \quad (4.5.21)$$

Din relațiile (4.5.3), (4.5.20) și (4.5.21) rezultă că  $T(n) = \Theta(n \log_2 n)$ .  $\square$

**Observația 4.5.4.** Complexitatea temporală a algoritmului de sortare prin interclasare este una dintre cele mai bune din clasa algoritmilor de sortare bazați pe comparații de chei. Un dezavantaj este faptul că se utilizează un vector suplimentar de lucru.

**Observația 4.5.5.** Se poate îmbunătăți procedura de interclasare, micșorând numărul de *deplasări* (atribuiri în care intervin elemente ale vectorului). Ținem seama de următoarele:

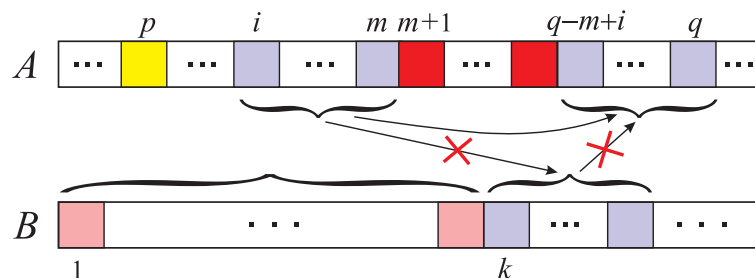
- 1) Dacă primul ciclu se termină când  $i > m$ , atunci elementele rămase în secvența  $(a_j, \dots, a_q)$  nu vor mai fi copiate în vectorul  $B$ , pentru ca apoi să fie readuse în  $A$ , deoarece ele ocupă deja pozițiile corecte.



- 2) Dacă primul ciclu se termină când  $j > q$ , atunci elementele rămase în secvența  $(a_i, \dots, a_m)$  vor fi copiate direct în vectorul  $A$ , pe pozițiile

$$q - m + i, q - m + i + 1, \dots, q,$$

fără a le mai transfera în  $B$  și apoi, de aici, în  $A$ .



Pentru a evita eventualele suprapuneri nedorite, copierea se va face în ordine inversă.

Obținem următoarea variantă îmbunătățită, descrisă în pseudocod.

**INTERCLAS2**( $A, p, q, m$ ) :

$i \leftarrow p$ ;

$j \leftarrow m + 1$ ;

$k \leftarrow 1$ ;

**while** ( $i \leq m$ ) *and* ( $j \leq q$ ) **do**

**if**  $A[i] \leq A[j]$  **then**

$B[k] \leftarrow A[i]$ ;

$i \leftarrow i + 1$ ;

**else**

$B[k] \leftarrow A[j]$ ;

$j \leftarrow j + 1$ ;

$k \leftarrow k + 1$ ;

**if**  $j \leq q$  **then**

$lim \leftarrow j - 1$ ;

**else**

$lim \leftarrow q - m + i - 1$ ;

**for**  $j = \overline{m, i, -1}$  **do**

$A[q - m + j] \leftarrow A[j]$ ;

$k \leftarrow 1$ ;

**for**  $i = \overline{p, lim}$  **do**

$A[i] \leftarrow B[k]$ ;

$k \leftarrow k + 1$ ;