PROIECTAREA ALGORITMILOR

Conf. univ. dr. COSTEL BĂLCĂU

2025

Tematica

Evaluare

- Activitate laborator: 30% (Programe și probleme din Temele de laborator)
- Teme de casă: 20% (Programe și probleme suplimentare)
- Examen final: 50% (Probă scrisă: 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. Bolosteanu, *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*, Editura 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] P. Deitel, H. Deitel, *Java SE 8 for programmers*, Deitel & Associates, Boston, 2014.

[14] D. Du, X. Hu, *Steiner Tree Problems in Computer Communication Networks*, World Scientific Publishing Co. Pte. Ltd., Hackensack, 2008.

[15] S. Even, *Graph Algorithms*, Cambridge University Press, Cambridge, 2012.

[16] F. Gebali, *Algorithms and parallel computing*, John Wiley & Sons, New Jersey, 2011.

[17] H. Georgescu, *Tehnici de programare*, Editura Universității din București, București, 2005.

[18] C.A. Giumale, *Introducere în analiza algoritmilor. Teorie și aplicații*, Editura Polirom, Iași, 2004.

[19] M. Goodrich, R. Tamassia, *Algorithm Design. Foundations, Analysis and Internet Examples*, Wiley, New Delhi, 2011.

[20] R. Harper, *Practical foundations for programming languages*, Cambridge University Press, Cambridge, 2013.

[21] F.V. Jensen, T.D. Nielsen, *Bayesian Networks and Decision Graphs*, Springer, New York, 2007.

[22] D. Jungnickel, *Graphs, Networks and Algorithms*, Springer, Heidelberg, 2013.

[23] D.E. Knuth, *The Art Of Computer Programming. Vol. 4A: Combinatorial Algorithms*, Addison-Wesley, Massachusetts, 2011.

[24] B. Korte, J. Vygen, *Combinatorial Optimization. Theory and Algorithms*, Springer, Heidelberg, 2012.

[25] R. Lafore, *Data Structures and Algorithms in Java*, Sams Publishing, Indianapolis, 2002.

[26] A. Levitin, *Introduction to The Design and Analysis of Algorithms*, Pearson, Boston, 2012.

[27] L. Livovschi, H. Georgescu, *Sinteza și analiza algoritmilor*, Editura Științifică și Enciclopedică, București, 1986.

[28] D. Logofătu, *Algoritmi fundamentali în C++: Aplicații*, Editura Polirom, Iași, 2007.

[29] D. Logofătu, *Algoritmi fundamentali în Java: Aplicații*, Editura Polirom, Iași, 2007.

[30] D. Lucanu, M. Craus, *Proiectarea algoritmilor*, Editura Polirom, Iași, 2008.

[31] S. Miller, *Mathematics of Optimization: How to do Things Faster*, AMS, Providence, 2017.

[32] D.A. Popescu, *Bazele programării - JAVA după C++*, ebooks.infobits.ro, 2019.

[33] D.R. Popescu, *Combinatorică și teoria grafurilor*, Societatea de Științe Matematice din România, București, 2005.

[34] N. Popescu, *Data structures and algorithms using Java*, Editura Politehnica Press, București, 2008.

[35] C.P. Popovici, H. Georgescu, L. State, *Bazele informaticii. Vol. I, II*, Editura Universității din București, București, 1990, 1991.

[36] V. Preda, C. Bălcău, *Entropy optimization with applications*, Editura Academiei Române, București, 2010.

[37] O.A. Schipor, S.G. Pentiuc, F. Gîză-Belciug, *Limbajul C - Tehnici de programare eficientă*, Editura Matrix Rom, București, 2014.

[38] R. Sedgewick, P. Flajolet, *An Introduction to the Analysis of Algorithms*, Addison-Wesley, New Jersey, 2013.

[39] R. Sedgewick, K. Wayne, *Algorithms*, Addison-Wesley, Massachusetts, 2011.

[40] R. Stephens, *Essential Algorithms: A Practical Approach to Computer Algorithms*, Wiley, Indianopolis, 2013.

[41] Ș. Tănasă, C. Olaru, Ș. Andrei, *Java de la 0 la expert*, Editura Polirom, Iași, 2007.

[42] T. Toadere, *Grafe. Teorie, algoritmi și aplicații*, Editura Albastră, Cluj-Napoca, 2002.

[43] I. Tomescu, *Combinatorică și teoria grafurilor*, Tipografia Universității din București, București, 1978.

[44] I. Tomescu, *Probleme de combinatorică și teoria grafurilor*, Editura Didactică și Pedagogică, București, 1981.

[45] I. Tomescu, *Data structures*, Editura Universității din București, București, 2004.

[46] M.A. Weiss, *Data Structures and Algorithm Analysis in Java*, Addison-Wesley, New Jersey, 2012.

[47] ***, *Handbook of combinatorics*, edited by R.L. Graham, M. Grőtschel and L. Lovász, Elsevier, Amsterdam, 1995.

[48] ***, *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.

[49] ***, *Revista MATINF. Publicație bianuală de matematică și informatică pentru elevi și profesori*, editată de Departamentul Matematică-Informatică, Universitatea din Pitești, Editura Universității din Pitești.

Tema 1

Elemente de complexitatea algoritmilor

1.1 Notații 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 algoritmii 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* $f : \mathbb{N} \setminus A \to \mathbb{R}$ *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

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

Exemplul 1.1.1. Funcția $f : D \to \mathbb{R},\ f(n) = \dfrac{(3n^4 + n + 3)\sqrt{n-5}}{(5n+1)(n-8)}$, unde $D = \{n \in \mathbb{N} \,|\, n \geq 5,\ n \neq 8\}$, 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 $g : \mathbb{N} \setminus \{1, 6\} \to \mathbb{R},\ g(n) = \dfrac{\ln(n^5+1) - n}{(n-1)(n-6)}$, nu este asimptotic pozitivă, deoarece $(n-1)(n-6) > 0,\ \forall\, n \geq 7$, dar $\lim\limits_{n\to\infty}[\ln(n^5 + 1) - n] = -\infty$, 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* $f : \mathbb{N} \to \mathbb{R}$*, de grad* p*,*

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

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* $\exists \lim\limits_{n\to\infty} \dfrac{f(n)}{g(n)} = 1$.

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* $\lim\limits_{n\to\infty} \dfrac{f(n)}{g(n)} = L$. *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* $f : \mathbb{N} \to \mathbb{R}$ *o funcție polinomială de grad p,*

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

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. Notația $\mathcal{O}$ se utilizează pentru a exprima complexitatea unui algoritm corespunzătoare *timpului de execuție în cazul cel mai defavorabil*, fiind astfel cea mai adecvată analizei algoritmilor. Notația Ω este corespunzătoare *timpului de execuție în cazul cel mai favorabil*, caz practic irelevant, fiind astfel mai puțin utilizată. Notațiile $\sim$ și Θ 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 Θ), 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 Θ). 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 $T_{\mathcal{A}}(n) = \mathcal{O}\left(T_{\mathcal{A}'}(n)\right)$.*

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 $A = (a_1, a_2, \ldots, a_n)$, $n \geq 1$. Se cere să se determine maximul și minimul dintre elementele $a_1, a_2, \ldots, a_n$, 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 $a_1, a_2, ..., a_{k-1}, a_k$ 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

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

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. □

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, $T_{\text{MAX-MIN}}(n) = \mathcal{O}\left(T_{\mathcal{A}}(n)\right)$, deci am obținut următorul rezultat:

Propoziția 1.2.1. *Algoritmul MAX-MIN este asimptotic-optim (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, ..., și calculează maximul dintre maximele acestor perechi și minimul dintre minimele perechilor.

MAX-MIN-PER(A, n, M, m)**:**
$M \leftarrow a_n$; $m \leftarrow a_n$;
for $i = \overline{1, \lfloor n/2 \rfloor}$ **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 $3 \cdot \left\lfloor \frac{n}{2} \right\rfloor$ comparații de chei, indiferent de ordinea dintre elementele vectorului A. □

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

1.3 Determinarea valorii unui polinom dat într-un punct dat

Problema determinării valorii unui polinom dat într-un punct dat este următoarea:

Se consideră un polinom

$$f = a_n \cdot X^n + a_{n-1} \cdot X^{n-1} + \cdots + a_1 \cdot X + a_0,\ n \in \mathbb{N},\ a_0, a_1, \ldots, a_n \in \mathbb{R},\ a_n \neq 0$$

și un punct (valoare) $x \in \mathbb{R}$.

Se cere să se determine valoarea polinomului f în x, adică valoarea

$$f(x) = a_n \cdot x^n + a_{n-1} \cdot x^{n-1} + \cdots + a_1 \cdot x + a_0 \in \mathbb{R}.$$

Exemplul 1.3.1. De exemplu, pentru polinomul

$$f = 2X^4 - 4X^3 + 5X^2 + 7X - 8$$

și $x = 3$ avem

$$\begin{aligned} f(3) &= 2 \cdot 3^4 - 4 \cdot 3^3 + 5 \cdot 3^2 + 7 \cdot 3 - 8 \\ &= 2 \cdot 81 - 4 \cdot 27 + 5 \cdot 9 + 7 \cdot 3 - 8 \\ &= 162 - 108 + 45 + 21 - 8 \\ &= 112. \end{aligned}$$

Metoda 1) Calculăm valoarea $v = f(x)$ ca *sumă a monoamelor* componente a_0, $a_1 \cdot x$, $a_2 \cdot x^2$, $\ldots$, $a_{n-1} \cdot x^{n-1}$, $a_n \cdot x^n$ (ca în exemplul de mai sus).

Descrierea în pseudocod a algoritmului are următoarea formă.

Algoritmul 1.3.1.

VALPOL1(A, n, x, v)**:**
$v \leftarrow A[0]$;
for $i = \overline{1, n}$ **do**
  $p \leftarrow A[i]$; // $p = a_i \cdot x^i$
  for $j = \overline{1, i}$ **do**
    $p \leftarrow p \cdot x$;
  $v \leftarrow v + p$;
returnează v;

Analiza complexității algoritmului:

Vom număra numai operațiile de adunare și de înmulțire (celelalte operații care se efectuează, atribuiri și comparații, au același ordin de creștere cu cele pe care le analizăm).

Pentru fiecare $i = \overline{1, n}$ algoritmul efectuează i înmulțiri (de forma $p \cdot x$) și o adunare (de forma $v + p$), deci în total algoritmul efectuează:

- $1 + 2 + \cdots + n = \dfrac{n(n+1)}{2}$ înmulțiri;
- n adunări.

Astfel am justificat următorul rezultat.

Propoziția 1.3.1. *Algoritmul 1.3.1 are complexitatea* $\Theta(n^2)$.

Metoda 2) Calculăm valoarea $f(x)$ utilizând *Schema lui Horner*:

	X^n	X^{n-1}	X^{n-2}	$\dots$	X	X^0
	a_n	a_{n-1}	a_{n-2}	$\dots$	a_1	a_0
x	$\underbrace{a_n}_{v_n}$	$\underbrace{x \cdot v_n + a_{n-1}}_{v_{n-1}}$	$\underbrace{x \cdot v_{n-1} + a_{n-2}}_{v_{n-2}}$	$\dots$	$\underbrace{x \cdot v_2 + a_1}_{v_1}$	$\underbrace{x \cdot v_1 + a_0}_{v_0}$

Valoarea cerută, $f(x)$, este calculată în ultima celulă a tabelului, adică $f(x) = v_0$.

Exemplul 1.3.2. De exemplu, pentru polinomul

$$f = 2X^4 - 4X^3 + 5X^2 + 7X - 8$$

și punctul $x = 3$ din exemplul anterior, Schema lui Horner este

	X^4	X^3	X^2	X	X^0
	2	-4	5	7	-8
$x = 3$	2	$3 \cdot 2 - 4$ $= 2$	$3 \cdot 2 + 5$ $= 11$	$3 \cdot 11 + 7$ $= 40$	$3 \cdot 40 - 8$ $= 112$

Deci $f(3) = 112$.

Pentru implementarea algoritmului putem utiliza o singură variabilă v pentru calculul valorilor $v_n, v_{n-1}, \dots, v_1, v_0$, deci la final avem $v = f(x)$.

Descrierea în pseudocod a algoritmului are următoarea formă.

Algoritmul 1.3.2.

VALPOL2(A, n, x, v) **:**
$v \leftarrow A[n]$;
for $i = \overline{n-1,\ 0,\ -1}$ **do**
  $v \leftarrow x \cdot v + A[i]$;
returnează v;

Analiza complexității algoritmului:

Vom număra, din nou, numai operațiile de adunare și de înmulțire (celelalte operații care se efectuează, atribuiri și comparații, au același ordin de creștere cu cele pe care le analizăm).

Pentru fiecare $i \in \{n-1, n-2, \dots, 1, 0\}$ algoritmul efectuează o înmulțire (de forma $x \cdot v$) și o adunare (de forma $x \cdot v + A[i]$), deci în total algoritmul efectuează:

- n înmulțiri;

- n adunări.

Astfel am justificat următorul rezultat.

Propoziția 1.3.2. *Algoritmul 1.3.2 are complexitatea* $\Theta(n)$*, deci este un algoritm liniar.*

Observația 1.3.1. Conform Propozițiilor 1.3.1 și 1.3.2 rezultă că Metoda 2 este mai eficientă decât Metoda 1.

1.4 Problema votului majoritar

Problema votului majoritar este următoarea:

Se consideră n voturi $v_1, v_2, \ldots, v_n$. Un candidat este majoritar dacă a obținut mai mult de $\frac{n}{2}$ voturi.

Se cere să se determine dacă există un candidat majoritar și, în caz afirmativ, să se determine acest candidat.

Exemplul 1.4.1. De exemplu, pentru voturile

$$2, 3, 3, 3, 2, 1, 4, 3, 1, 1, 5, 3, 3, 4, 2, 2, 5, 2, 2, 3, 1$$

avem un total de $n = 21$ de voturi repartizate conform tabelului următor.

Candidat	1	2	3	4	5
Nr. voturi	4	6	7	2	2

Niciun candidat nu are mai mult de $\frac{n}{2} = 10{,}5$ voturi, deci nu există candidat majoritar.

Exemplul 1.4.2. Pentru voturile

$$2, 3, 3, 3, 2, 1, 4, 3, 1, 1, 3, 5, 3, 3, 3, 2, 3, 5, 3, 2, 3, 1, 3$$

avem un total de $n = 23$ de voturi repartizate conform tabelului următor.

Candidat	1	2	3	4	5
Nr. voturi	4	4	12	1	2

Candidatul cu cele mai multe voturi este candidatul 3, care are 12 voturi, deci are mai mult de $\frac{n}{2} = 11{,}5$ voturi, deci este majoritar.

Metoda 1) Determinăm candidatul cu cele mai multe voturi și verificăm dacă este sau nu majoritar (ca în exemplele de mai sus).

Descrierea în pseudocod a algoritmului are următoarea formă.

Algoritmul 1.4.1.

VOTMAJ1$(V, n, cand)$**:** // $V = (v_1, v_2, \ldots, v_n)$
// $cand =$ candidatul cu cele mai multe voturi
$max \leftarrow 0$; // $max =$ numărul său de voturi
for $i = \overline{1, n}$ **do**
  if $V[i] \neq 0$ **then** // candidatul $V[i]$ nu a fost analizat
    $nr \leftarrow 1$; // numărul său de voturi
    for $j = \overline{i+1, n}$ **do**
      if $V[j] = V[i]$ **then**
        $nr \leftarrow nr + 1$;
        $V[j] \leftarrow 0$; // candidatul $V[j]$ a fost analizat
    if $nr > max$ **then** // actualizăm max și $cand$
      $max \leftarrow nr$;
      $cand \leftarrow V[i]$;
      // verificăm dacă votul pt. $cand$ este majoritar
      if $max > \dfrac{n}{2}$ **then**
        returnează $cand$; // $cand$ este majoritar
returnează 0; // nu există candidat majoritar

Observația 1.4.1. Deoarece un candidat care nu are niciun vot printre primele $\left\lfloor \dfrac{n+1}{2} \right\rfloor$ voturi nu poate fi majoritar, rezultă că în algoritmul anterior putem înlocui instrucțiunea **for** $i = \overline{1, n}$ **do** cu

$$\textbf{for } i = \overline{1, \left\lfloor \frac{n+1}{2} \right\rfloor} \textbf{ do}$$

Analiza complexității algoritmului

Cazul cel mai defavorabil este cel în care voturile sunt distincte două câte două. Vom număra numai comparațiile de forma $V[j] = V[i]$ (celelalte operații care se efectuează au cel mult același ordin de creștere cu cele pe care le analizăm).

În cazul cel mai defavorabil algoritmul efectuează câte o astfel de comparație pentru fiecare pereche

$$(i, j) \text{ cu } i \in \{1, 2, \ldots, n-1\} \text{ și } j \in \{i+1, i+2, \ldots, n\},$$

deci un total de

$$(n-1)+(n-2)+\ldots+2+1=\frac{n(n-1)}{2}$$

astfel de comparații.

Cu modificarea din observația anterioară, numărul acestor comparații se reduce la

$$(n-1)+(n-2)+\ldots+\left(n-\left\lfloor\frac{n+1}{2}\right\rfloor\right)=\frac{1}{2}\cdot\left(2n-1-\left\lfloor\frac{n+1}{2}\right\rfloor\right)\cdot\left\lfloor\frac{n+1}{2}\right\rfloor.$$

Astfel am justificat următorul rezultat.

Propoziția 1.4.1. *Algoritmul 1.4.1 are complexitatea* $\mathcal{O}(n^2)$.

Metoda 2) Sortăm voturile în ordine crescătoare. Apoi determinăm candidatul cu cele mai multe voturi, corespunzător celei mai lungi subsecvențe de voturi egale, și verificăm dacă este sau nu majoritar.

Descrierea în pseudocod a algoritmului are următoarea formă.

Algoritmul 1.4.2.

VOTMAJ2$(V, n, cand)$ **:**
SORTARE(V, n); // se sortează voturile crescător,
// adică $v_1 \leq v_2 \leq \ldots \leq v_n$
$i \leftarrow 1$;
$max \leftarrow 0$;
repeat
  // determinăm subsecvența voturilor egale cu $V[i]$
  $j \leftarrow i+1$;
  while $j \leq n$ *and* $V[j]=V[i]$ **do**
    $j \leftarrow j+1$;
  if $j-i > max$ **then** // actualizăm max și $cand$
    $max \leftarrow j-i$;
    $cand \leftarrow V[i]$;
    // verificăm dacă votul pt. $cand$ este majoritar
    if $max > \frac{n}{2}$ **then**
      returnează $cand$; // $cand$ este majoritar
  $i \leftarrow j$; // trecem la următorul candidat
while $i \leq \left\lfloor\frac{n+1}{2}\right\rfloor$;
returnează 0; // nu există candidat majoritar

Analiza complexității algoritmului

Așa cum se va demonstra în Capitolul 5, orice algoritm eficient de sortare în ordine crescătoare a celor n voturi are complexitatea $\Theta(n \log_2 n)$.

Pentru restul algoritmului vom număra, din nou, numai comparațiile de forma $V[j] = V[i]$ (celelalte operații care se efectuează au cel mult același ordin de creștere cu cele pe care le analizăm). În cazul cel mai defavorabil algoritmul efectuează $n - 1$ astfel de comparații (câte una pentru fiecare $j \in \{2, 3, \ldots, n\}$), deci restul algoritmului are complexitatea $\mathcal{O}(n)$.

Astfel complexitatea întregului algoritm este de ordinul $\Theta(n \log_2 n)$.

Am justificat următorul rezultat.

Propoziția 1.4.2. *Algoritmul 1.4.2 are complexitatea* $\Theta(n \log_2 n)$.

Exemplul 1.4.3. Pentru voturile din Exemplul 1.4.1, voturile sortate crescător sunt:

$$1, 1, 1, 1, 2, 2, 2, 2, 2, 2, 2, 3, 3, 3, 3, 3, 3, 3, 4, 4, 5, 5.$$

Subsecvența maximă de voturi egale este cea a candidatului 3, de lungime $7 \leq \frac{n}{2} = 10{,}5$, deci nu există candidat majoritar.

Exemplul 1.4.4. Pentru voturile din Exemplul 1.4.2, voturile sortate crescător sunt:

$$1, 1, 1, 1, 2, 2, 2, 2, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 4, 5, 5.$$

Subsecvența maximă de voturi egale este cea a candidatului 3, de lungime $12 > \frac{n}{2} = 11{,}5$, deci acest candidat este majoritar.

Metoda 3) Parcurgem voturile împerechind voturi diferite (adică voturi pentru candidați diferiți). La final, dacă nu rămân voturi neîmperecheate atunci nu există candidat majoritar, iar dacă rămân voturi neîmperecheate atunci ele vor fi ale unui aceluiași candidat și verificăm dacă acesta este sau nu majoritar.

Descrierea în pseudocod a algoritmului are următoarea formă.

Algoritmul 1.4.3.

```
VOTMAJ3(V, n, cand) :
nv ← 0;    // nv = numărul de voturi ce rămân neîmperecheate
for i = 1, n do                                  // analizăm votul V[i]
    if nv = 0 then
        // toate voturile anterioare au fost împerecheate
        nv ← 1;                  // nu putem împerechea votul curent
        cand ← V[i];                  // cand = candidatul ce rămâne
                                  // în urma împerecherii voturilor
    else
                                  // există voturi neîmperecheate
        if V[i] = cand then
            nv ← nv + 1;          // nu putem împerechea votul V[i]
        else
            nv ← nv − 1;                    // împerechem votul V[i]
if nv = 0 then
                              // nu au rămas voturi neîmperecheate
    returnează 0;                  // nu există candidat majoritar
else
                               // au rămas voturi neîmperecheate,
                // verificăm dacă votul pt. cand este majoritar
    nr ← 0;
    for i = 1, n do
        if V[i] = cand then
            nr ← nr + 1;
            if nr > n/2 then
                returnează cand;              // cand este majoritar
    returnează 0;                  // nu există candidat majoritar
```

Analiza complexității algoritmului

- Prima instrucțiune este o atribuire.
- Urmează ciclul **for** ce implică $n+1$ atribuiri și tot $n+1$ comparații, iar în cadrul corpului său se efectuează, pentru fiecare $i \in \{1, 2, \ldots, n\}$, fie câte o comparație și două atribuiri, fie câte două comparații și o atribuire; deci numărul total de atribuiri și numărul total de comparații sunt cuprinse în intervalul $[2n+1, 3n+1]$.
- Urmează comparația **if** $nv = 0$.

- Ramura **then** a acesteia nu efectuează instrucțiuni de atribuire sau de comparație.
- Ramura **else** efectuează doar o instrucțiune de atribuire, apoi ciclul **for** ce implică cel mult $n+1$ atribuiri și tot cel mult $n+1$ comparații, iar în cadrul corpului său se efectuează cel mult n atribuiri și cel mult $2n$ comparații.
- Indiferent de ramură, se apelează și o instrucțiune **return**.

Adunând toate aceste operații obținem că algoritmul efectuează:

- un număr de atribuiri cuprins în intervalul $[2n+2, 5n+4]$;
- un număr de comparații cuprins în intervalul $[2n+2, 6n+3]$;
- o instrucțiune **return**.

Astfel am justificat următorul rezultat.

Propoziția 1.4.3. *Algoritmul 1.4.3 are complexitatea* $\Theta(n)$*, deci este un algoritm liniar.*

Exemplul 1.4.5. Pentru voturile din Exemplul 1.4.1, aplicarea algoritmului anterior este evidențiată în tabelul următor.

i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
v_i	-	2	3	3	3	2	1	4	3	1	1	5	3	3	4	2
nv	0	1	0	1	2	1	0	1	0	1	2	1	0	1	0	1
$cand$	-	2	2	3	3	3	3	4	4	1	1	1	1	3	3	2

i	16	17	18	19	20	21
v_i	2	5	2	2	3	1
nv	2	1	2	3	2	1
$cand$	2	2	2	2	2	2

Avem $nv = 1$ și $cand = 2$, deci au rămas voturi neîmperecheate, ale candidatului 2. Acest candidat are 6 voturi, deci nu este majoritar. Astfel nu există candidat majoritar.

Exemplul 1.4.6. Pentru voturile din Exemplul 1.4.2, aplicarea algoritmului anterior este evidențiată în tabelul următor.

i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
v_i	-	2	3	3	3	2	1	4	3	1	1	3	5	3	3	3
nv	0	1	0	1	2	1	0	1	0	1	2	1	0	1	2	3
$cand$	-	2	2	3	3	3	3	4	4	1	1	1	1	3	3	3

i	16	17	18	19	20	21	22	23
v_i	2	3	5	3	2	3	1	3
nv	2	3	2	3	2	3	2	3
$cand$	3	3	3	3	3	3	3	3

Avem $nv = 3$ și $cand = 3$, deci au rămas voturi neîmperecheate, ale candidatului 3. Acest candidat are 12 voturi, deci este majoritar.

Observația 1.4.2. Conform Propozițiilor 1.4.1, 1.4.2 și 1.4.3 rezultă că Metoda 2 este mai eficientă decât Metoda 1, iar Metoda 3 este mai eficientă decât Metodele 1 și 2.

Observația 1.4.3. Evident, orice algoritm care rezolvă problema dată necesită parcurgerea (citirea și analiza) celor n voturi, deci conform Propoziției 1.4.3 rezultă că Algoritmul 1.4.3 este asimptotic-optim.

1.5 Determinarea ultimei cifre nenule a factorialului unui număr natural

În această secțiune ne propunem să determinăm ultima cifră nenulă a lui $n!$, unde n este un număr natural arbitrar. Avem nevoie de câteva noțiuni și rezultate pregătitoare.

Definiția 1.5.1. *a) Pentru orice număr natural m (scris în baza 10), notăm cu $u(m)$* ***ultima cifră*** *a numărului m.*

b) Pentru orice număr natural nenul m, notăm cu $r(m)$ ***ultima cifră nenulă*** *a numărului m.*

Observația 1.5.1. Evident, $u(m) = m \text{ MOD } 10$, pentru orice $m \in \mathbb{N}$. Pe de altă parte, pentru orice $m \in \mathbb{N}^*$ avem $r(m) = \frac{m \text{ MOD } 10^k}{10^{k-1}}$, unde k este cel mai mic număr natural nenul cu proprietatea că $m \text{ MOD } 10^k \neq 0$.

Cifra $r(m)$ poate fi calculată prin împărțiri succesive la 10, cât timp restul este egal cu zero, algoritm care are complexitatea $\mathcal{O}(\ln m)$. Într-adevăr, numărul de impărțiri succesive la 10 este egal cu 1 plus numărul de zerouri în care se termină m, deci în cazul cel mai defavorabil, și anume când $m = \overline{c_1 00 \ldots 0}$, se efectuează $1 + \lfloor \lg m \rfloor$ împărțiri.

Un algoritm echivalent pentru calculul lui $r(m)$ se bazează pe următoarea proprietate.

Propoziția 1.5.1. *Pentru orice număr natural nenul m există un unic triplet (a, b, c) cu $a, b \in \mathbb{N}$ și $c \in \{1, 3, 7, 9\}$ astfel încât*

$$m = 2^a \cdot 5^b \cdot (\mathcal{M}10 + c) \tag{1.5.1}$$

(notația $\mathcal{M}10$ reprezintă un multiplu al lui 10).

Demonstrație. Fie

$$a = \max\{k \in \mathbb{N} \mid 2^k \text{ divide } m\} \text{ și } b = \max\{k \in \mathbb{N} \mid 5^k \text{ divide } m\}.$$

Atunci $m = 2^a \cdot 5^b \cdot p$, unde p este un număr natural care nu se divide nici cu 2, nici cu 5. Notând $c = u(p)$, rezultă că $m = 2^a \cdot 5^b \cdot (\mathcal{M}10 + c)$ și $c \in \{1, 3, 7, 9\}$.

Demonstrăm acum unicitatea scrierii (1.5.1). Într-adevăr, fie

$$m = 2^a \cdot 5^b \cdot (\mathcal{M}10 + c) = 2^{a'} \cdot 5^{b'} \cdot (\mathcal{M}10 + c'), \text{ cu } a, b, a', b' \in \mathbb{N} \text{ și } c, c' \in \{1, 3, 7, 9\}.$$

Cum $5^b \cdot (\mathcal{M}10 + c)$ nu se divide cu 2, rezultă că $2^{a'}$ divide 2^a, deci $a' \leq a$. Analog se obține că 2^a divide $2^{a'}$, deci $a \leq a'$ și astfel $a = a'$. Analog se arată că $b = b'$. Rezultă că $\mathcal{M}10 + c = \mathcal{M}10 + c'$, deci și $c = c'$. □

Definiția 1.5.2. *Pentru orice număr natural nenul m, notăm cu $a(m)$, $b(m)$ și $c(m)$ numerele a, b, respectiv c din scrierea (1.5.1).*

Propoziția 1.5.2. *Pentru orice număr natural nenul m avem*

$$r(m) = \begin{cases} c(m), & \text{dacă } a(m) = b(m), \\ 5, & \text{dacă } a(m) < b(m), \\ u\left(2^{a(m)-b(m)} \cdot c(m)\right), & \text{dacă } a(m) > b(m). \end{cases}$$

Demonstrație. Conform definiției anterioare, $m = 2^{a(m)} \cdot 5^{b(m)} \cdot (\mathcal{M}10 + c(m))$, cu $a(m), b(m) \in \mathbb{N}$ și $c(m) \in \{1, 3, 7, 9\}$.

Cazul 1. Dacă $a(m) = b(m)$, atunci $m = 10^{a(m)} \cdot (\mathcal{M}10 + c(m))$, deci m se termină în $a(m)$ zerouri și ultima cifră nenulă a sa este $c(m)$.

Cazul 2. Dacă $a(m) < b(m)$, atunci $m = 10^{a(m)} \cdot 5^{b(m)-a(m)} \cdot (\mathcal{M}10 + c(m))$, deci m se termină în $a(m)$ zerouri și ultima sa cifră nenulă este $u\left(5^{b(m)-a(m)} \cdot c(m)\right) = 5$.

Cazul 3. Dacă $a(m) > b(m)$, atunci $m = 10^{b(m)} \cdot 2^{a(m)-b(m)} \cdot (\mathcal{M}10 + c(m))$, deci m se termină în $b(m)$ zerouri și ultima sa cifră nenulă este $u\left(2^{a(m)-b(m)} \cdot c(m)\right)$. □

Observația 1.5.2. Pentru orice $k \in \mathbb{N}^*$ avem $u\left(2^k\right) = u\left(6 \cdot 2^{k \bmod 4}\right)$, deci formula lui $r(m)$ dată de Propoziția 1.5.2 poate fi rescrisă sub forma:

$$r(m) = \begin{cases} c(m), & \text{dacă } a(m) = b(m), \\ 5, & \text{dacă } a(m) < b(m), \\ u\left(6 \cdot 2^{(a(m)-b(m)) \bmod 4} \cdot c(m)\right), & \text{dacă } a(m) > b(m). \end{cases}$$

Exemplul 1.5.1. $15300 = 2^2 \cdot 5^2 \cdot 153$, deci $a(15300) = b(15300) = 2$, $c(15300) = 3$ și $r(15300) = 3$.

$23500 = 2^2 \cdot 5^3 \cdot 47$, deci $a(23500) = 2$, $b(23500) = 3$, $c(23500) = 7$ și $r(23500) = 5$.

$42240 = 2^8 \cdot 5 \cdot 33$, deci $a(42240) = 8$, $b(42240) = 1$, $c(42240) = 3$ și $r(42240) = u\left(6 \cdot 2^{(8-1) \bmod 4} \cdot 3\right) = u(6 \cdot 2^3 \cdot 3) = 4$.

Obținem astfel următorul algoritm pentru determinarea lui $r(m)$.

UCN(m)**:**
$p \leftarrow m$; $a \leftarrow 0$; $b \leftarrow 0$;
while p *MOD* $2 = 0$ **do**
  $a \leftarrow a + 1$; $p \leftarrow p/2$;
while p *MOD* $5 = 0$ **do**
  $b \leftarrow b + 1$; $p \leftarrow p/5$;
$c \leftarrow p$ MOD 10;
if $a = b$ **then**
  return c;
else
  if $a < b$ **then**
    return 5;
  else
    for $i = \overline{1, (a-b) \text{ MOD } 4}$ **do**
      $c \leftarrow 2 \cdot c$;
    return $(6 \cdot c)$ MOD 10;

Propoziția 1.5.3. *Algoritmul UCN are complexitatea* $\Theta(a(m) + b(m) + 1)$.

Demonstrație. Conform Definiției 1.5.2 avem $m = 2^{a(m)} \cdot 5^{b(m)} \cdot (\mathcal{M}10 + c(m))$, cu $a(m), b(m) \in \mathbb{N}$ și $c(m) \in \{1, 3, 7, 9\}$. Corpul primului ciclu **while** din algoritmul UCN are două instrucțiuni și se execută de $a(m)$ ori, corpul celui de-al doilea ciclu **while** are tot două instrucțiuni și se execută de $b(m)$ ori, iar corpul ciclului **for** are o instrucțiune și se execută de cel mult 3 ori. Algoritmul mai conține 4 atribuiri și una sau două comparații, deci are complexitatea $\Theta(a(m) + b(m) + 1)$. □

Corolarul 1.5.1. *Algoritmul UCN are complexitatea* $\mathcal{O}(\ln m)$.

Demonstrație. Avem $2^{a(m)} \leq m$ și $5^{b(m)} \leq m$, deci $a(m) \leq \log_2 m$ și $b(m) \leq \log_5 m$. Obținem că $a(m) + b(m) + 1 \leq \left(\frac{1}{\ln 2} + \frac{1}{\ln 5}\right) \ln m + 1$, deci conform propoziției anterioare rezultă că algoritmul UCN are complexitatea $\mathcal{O}(\ln m)$. □

Observația 1.5.3. Mai mult, în cazul când m este o putere a lui 2, adică $m = 2^{a(m)}$, avem $b(m) = 0$ și astfel $a(m) + b(m) + 1 = \log_2 m + 1 = \frac{\ln m}{\ln 2} + 1$. Conform rezultatelor anterioare rezultă că algoritmul UCN are timpul de execuție în acest caz particular chiar de ordinul $\Theta(\ln m)$, care este mai precis decât $\mathcal{O}(\ln m)$. Evident, același ordin se obține și în cazul când m este o putere a lui 5 sau a lui 10, și nu numai.

Obținem astfel o primă metodă pentru calculul ultimei cifre nenule a lui $n!$, adică $r(n!)$.

Metoda 1) Calculăm $m = n!$ și determinăm $r(n!) = r(m)$, ca mai sus. Descrierea în pseudocod a algoritmului are următoarea formă.

Algoritmul 1.5.1.

UCNF1(n)**:**
$m \leftarrow 1$;
for $i = \overline{2, n}$ **do**
  $m \leftarrow m \cdot i$;
return **UCN**(m) ;

Analiza complexității algoritmului

Propoziția 1.5.4. *Dacă* n *este un număr natural nenul, atunci*

$$a(n!) = \left\lfloor \frac{n}{2} \right\rfloor + \left\lfloor \frac{n}{2^2} \right\rfloor + \left\lfloor \frac{n}{2^3} \right\rfloor + \ldots + \left\lfloor \frac{n}{2^k} \right\rfloor, \text{ unde } k = \lfloor \log_2 n \rfloor$$

și

$$b(n!) = \left\lfloor \frac{n}{5} \right\rfloor + \left\lfloor \frac{n}{5^2} \right\rfloor + \left\lfloor \frac{n}{5^3} \right\rfloor + \ldots + \left\lfloor \frac{n}{5^{k'}} \right\rfloor, \text{ unde } k' = \lfloor \log_5 n \rfloor .$$

Demonstrație. $a(n!)$ reprezintă puterea lui 2 din descompunerea în factori primi a produsului $1 \cdot 2 \cdot \ldots \cdot n$. Printre numerele $1, 2, \ldots, n$ avem exact $\left\lfloor \frac{n}{2} \right\rfloor$ numere divizibile cu 2, dintre care exact $\left\lfloor \frac{n}{2^2} \right\rfloor$ sunt divizibile și cu 2^2, dintre care exact $\left\lfloor \frac{n}{2^3} \right\rfloor$ sunt divizibile și cu 2^3, ș.a.m.d., deci

$$a(n!) = \left\lfloor \frac{n}{2} \right\rfloor + \left\lfloor \frac{n}{2^2} \right\rfloor + \left\lfloor \frac{n}{2^3} \right\rfloor + \ldots .$$

Cum $2^{k+1} > n$, deci printre numerele $1, 2, \ldots, n$ nu există numere divizibile cu 2^{k+1}, rezultă că suma anterioară se poate încheia cu termenul $\left\lfloor \frac{n}{2^k} \right\rfloor$. Analog se demonstrează și formula lui $b(n!)$ din enunț. □

Corolarul 1.5.2. *Avem* $a(n!) = \Theta(n)$ *și* $b(n!) = \Theta(n)$.

Demonstrație. Fie $n \in \mathbb{N}^*$ și $k = \lfloor \log_2 n \rfloor$. Conform propoziției anterioare,

$$a(n!) = \left\lfloor \frac{n}{2} \right\rfloor + \left\lfloor \frac{n}{2^2} \right\rfloor + \left\lfloor \frac{n}{2^3} \right\rfloor + \ldots + \left\lfloor \frac{n}{2^k} \right\rfloor.$$

Avem $k \leq \log_2 n < k+1$, deci $2^k \leq n < 2^{k+1}$. Deducem că $2^{k-1} \leq \left\lfloor \frac{n}{2} \right\rfloor < 2^k$, $2^{k-2} \leq \left\lfloor \frac{n}{2^2} \right\rfloor < 2^{k-1}$,..., $2^0 \leq \left\lfloor \frac{n}{2^k} \right\rfloor < 2^1$, deci prin adunare obținem că

$$2^k - 1 \leq a(n!) < 2(2^k - 1).$$

Cum $\log_2 n - 1 < k \leq \log_2 n$, rezultă că

$$\frac{n-2}{2} < a(n!) < 2n - 2,$$

deci $a(n!) = \Theta(n)$.

Analog se demonstrează că

$$\frac{n-5}{20} < b(n!) < \frac{5n-5}{4},$$

deci și $b(n!) = \Theta(n)$. □

Corolarul 1.5.3. *Avem* $a(0!) = b(0!) = a(1!) = b(1!) = 0$ *și* $a(n!) > b(n!)$ *pentru orice* $n \geq 2$.

Demonstrație. Evident, $a(1) = b(1) = 0$. Pentru $n \geq 2$, cu notațiile din Propoziția 1.5.4, avem $k \geq k'$, $\left\lfloor \frac{n}{2} \right\rfloor > \left\lfloor \frac{n}{5} \right\rfloor$, $\left\lfloor \frac{n}{2^2} \right\rfloor \geq \left\lfloor \frac{n}{5^2} \right\rfloor$, ..., $\left\lfloor \frac{n}{2^{k'}} \right\rfloor \geq \left\lfloor \frac{n}{5^{k'}} \right\rfloor$, de unde prin adunare obținem că $a(n!) > b(n!)$. □

Corolarul 1.5.4. *Pentru orice* $n \in \mathbb{N}$*, numărul de zerouri în care se termină numărul* $n!$ *este egal cu* $b(n!)$.

Demonstrație. Conform corolarului anterior, $a(n!) \geq b(n!)$, deci conform (1.5.1) avem $n! = 10^{b(n!)} \cdot 2^{a(n!)-b(n!)} \cdot (\mathcal{M}10 + c(n!))$, de unde rezultă afirmația din enunț. □

Conform Propoziției 1.5.3 și Corolarului 1.5.2 obținem următorul rezultat.

Propoziția 1.5.5. *Algoritmul UCNF1 are complexitatea* $\Theta(n)$.

Observația 1.5.4. Algoritmul UCNF1, bazat pe calculul lui $n!$, este un algoritm liniar, dar este practic inoperabil pentru valori mari ale lui n din cauza mărimii numărului $n!$.

Metoda 2) Vom determina $r(n!)$ tot pe baza Propoziției 1.5.2, dar vom calcula numerele $a(n!)$, $b(n!)$ și $c(n!)$ evitând calculul efectiv al lui $n!$. Ne vom baza pe următorul rezultat.

Propoziția 1.5.6. *Pentru orice numere naturale nenule* $m_1, m_2, \ldots, m_n$ *avem*

$$\begin{aligned} a(m_1 \cdot m_2 \cdot \ldots \cdot m_n) &= a(m_1) + a(m_2) + \ldots + a(m_n), \\ b(m_1 \cdot m_2 \cdot \ldots \cdot m_n) &= b(m_1) + b(m_2) + \ldots + b(m_n), \\ c(m_1 \cdot m_2 \cdot \ldots \cdot m_n) &= u\left(c(m_1) \cdot c(m_2) \cdot \ldots \cdot c(m_n)\right). \end{aligned}$$

Demonstrație. Avem $m_1 \cdot m_2 \cdot \ldots \cdot m_n = \prod_{i=1}^{n} \left(2^{a(m_i)} \cdot 5^{b(m_i)} \cdot (\mathcal{M}10 + c(m_i))\right) = 2^{\sum_{i=1}^{n} a(m_i)} \cdot 5^{\sum_{i=1}^{n} b(m_i)} \cdot \left(\mathcal{M}10 + u\left(\prod_{i=1}^{n} c(m_i)\right)\right)$, de unde rezultă egalitățile din enunț. □

Următorul rezultat este o consecință imediată a Propozițiilor 1.5.6 și 1.5.2 și a Observației 1.5.2.

Corolarul 1.5.5. *Fie* $m_1, m_2, \ldots, m_n$ *numere naturale nenule.*

a) Dacă $a(m_1) + a(m_2) + \ldots + a(m_n) = b(m_1) + b(m_2) + \ldots + b(m_n)$, *atunci*

$$r(m_1 \cdot m_2 \cdot \ldots \cdot m_n) = u\left(c(m_1) \cdot c(m_2) \cdot \ldots \cdot c(m_n)\right).$$

b) Dacă $a(m_1) + a(m_2) + \ldots + a(m_n) < b(m_1) + b(m_2) + \ldots + b(m_n)$, *atunci*

$$r(m_1 \cdot m_2 \cdot \ldots \cdot m_n) = 5.$$

c) Dacă $a(m_1) + a(m_2) + \ldots + a(m_n) > b(m_1) + b(m_2) + \ldots + b(m_n)$, *atunci*

$$r(m_1 \cdot m_2 \cdot \ldots \cdot m_n) = u\Big(6 \cdot 2^{(a(m_1)+a(m_2)+\ldots+a(m_n)-b(m_1)-b(m_2)-\ldots-b(m_n)) \ MOD \ 4} \cdot c(m_1) \cdot c(m_2) \cdot \ldots \cdot c(m_n)\Big).$$

Exemplul 1.5.2. $320 = 2^6 \cdot 5$ și $150 = 2 \cdot 5^2 \cdot 3$, deci $a(320) = 6$, $b(320) = 1$, $c(320) = 1$, $a(150) = 1$, $b(150) = 2$, $c(150) = 3$, $a(320) + a(150) = 7$, $b(320) + b(150) = 3$ și $r(320 \cdot 150) = u\left(6 \cdot 2^{(7-3) \text{ MOD } 4} \cdot 1 \cdot 3\right) = u(6 \cdot 2^0 \cdot 3) = 8$.

Obținem următorul algoritm pentru calculul lui $r(n!) = r(1 \cdot 2 \cdot \ldots \cdot n)$.

Algoritmul 1.5.2.

UCNF2(n)**:**
$a \leftarrow 0$; $b \leftarrow 0$; $c \leftarrow 1$;
for $k = \overline{1,\, n}$ **do**
  $p \leftarrow k$;
  while p *MOD* $2 = 0$ **do**
    $a \leftarrow a + 1$; $p \leftarrow p/2$;
  while p *MOD* $5 = 0$ **do**
    $b \leftarrow b + 1$; $p \leftarrow p/5$;
  $c \leftarrow (c \cdot p)$ MOD 10;
if $a = b$ **then**
  return c;
else
  if $a < b$ **then**
    return 5;
  else
    for $i = \overline{1,\, (a-b)\ MOD\ 4}$ **do**
      $c \leftarrow 2 \cdot c$;
    return $(6 \cdot c)$ MOD 10;

Analiza complexității algoritmului

Propoziția 1.5.7. *Algoritmul UCNF2 are complexitatea* $\Theta(n)$.

Demonstrație. Conform Propoziției 1.5.6 și Corolarului 1.5.2, complexitatea algoritmului UCNF1 este $\Theta\left(\sum_{k=1}^{n}(1 + a(k) + b(k))\right) = \Theta\left(n + a(n!) + b(n!)\right) = \Theta(n)$. □

Exemplul 1.5.3. Aplicând Metoda 2, avem $r(11!) = r(1 \cdot 2 \cdot 3 \cdot \ldots \cdot 11) = r(1 \cdot 2 \cdot 3 \cdot 2^2 \cdot 5 \cdot 2 \cdot 3 \cdot 7 \cdot 2^3 \cdot 9 \cdot 2 \cdot 5 \cdot 11) = r\left(2^8 \cdot 5^2 \cdot 3 \cdot 3 \cdot 7 \cdot 9 \cdot 1\right) = u\left(6 \cdot 2^{(8-2)\ \mathrm{MOD}\ 4} \cdot 3 \cdot 3 \cdot 7 \cdot 9\right) = 8$.

Observația 1.5.5. Pentru valori mari ale lui n, acest procedeu de calcul este practic imposibil de aplicat fără utilizarea unui program pentru calculator.

Metoda 3) Prezentăm în continuare un algoritm subliniar pentru determinarea lui $r(n!)$. Avem nevoie de alte câteva rezultate.

Observația 1.5.6. Dacă m este un număr natural nedivizibil cu 10, atunci $u(m) \neq 0$, deci $r(m) = u(m)$.

Observația 1.5.7. Evident, $u(m_1 \cdot m_2 \cdot \ldots \cdot m_n) = u\left(u(m_1) \cdot u(m_2) \cdot \ldots \cdot u(m_n)\right)$, pentru orice $m_1, m_2, \ldots, m_n \in \mathbb{N}$ $(n \in \mathbb{N},\ n \geq 2)$.

Pe de altă parte, egalitatea

$$r(m_1 \cdot m_2 \cdot \ldots \cdot m_n) = u\left(r(m_1) \cdot r(m_2) \cdot \ldots \cdot r(m_n)\right) \tag{1.5.2}$$

este adevărată doar dacă $u\left(r(m_1)\cdot r(m_2)\cdot\ldots\cdot r(m_n)\right)\neq 0$. Mai mult, egalitatea $r(m_1\cdot m_2\cdot\ldots\cdot m_n)=r\left(r(m_1)\cdot r(m_2)\cdot\ldots\cdot r(m_n)\right)$ este adevărată dacă $u\left(r(m_1)\cdot r(m_2)\cdot\ldots\cdot r(m_n)\right)\neq 0$ și nu este neapărat adevărată în caz contrar.

De exemplu, $r(320\cdot 170)=4=u\left(r(320)\cdot r(170)\right)=r\left(r(320)\cdot r(170)\right)$, dar $r(320\cdot 150)=8$ iar $u\left(r(320)\cdot r(150)\right)=0$ și $r\left(r(320)\cdot r(150)\right)=1$.

Observația 1.5.8. Fie $m_1, m_2, \ldots, m_n$ numere naturale nenule astfel încât $r(m_i)\neq 5$ pentru orice $i=\overline{1,n}$. Atunci $u\left(r(m_1)\cdot r(m_2)\cdot\ldots\cdot r(m_n)\right)\neq 0$ și conform observației anterioare are loc egalitatea (1.5.2).

Lema 1.5.1. *Fie n un număr natural, $n\geq 2$. Atunci $r(n!)\in\{2,4,6,8\}$.*

Demonstrație. Conform Corolarului 1.5.3, $a(n!)>b(n!)$, deci conform (1.5.1) avem

$$n!=10^{b(n!)}\cdot 2^{a(n!)-b(n!)}\cdot\left(\mathcal{M}10+c(n!)\right),$$

de unde rezultă că

$$r(n!)=r\left(2^{a(n!)-b(n!)}\cdot\left(\mathcal{M}10+c(n!)\right)\right).$$

Dar

$$r\left(2^{a(n!)-b(n!)}\right)=u\left(2^{a(n!)-b(n!)}\right)\in\{2,4,6,8\},$$
$$r\left(\mathcal{M}10+c(n!)\right)=u\left(\mathcal{M}10+c(n!)\right)=c(n!)\in\{1,3,7,9\},$$

deci, folosind și Observația 1.5.8, $r(n!)=u\left(r\left(2^{a(n!)-b(n!)}\right)\cdot c(n!)\right)\in\{2,4,6,8\}$. □

Lema 1.5.2. *Fie m și k două numere naturale astfel încât m este divizibil cu 2^{k+1}. Atunci*

$$u\left(m:2^k\right)=u\left(m:2^{k\ MOD\ 4}\right)=u\left(m\cdot 2^{4-(k\ MOD\ 4)}\right).$$

Demonstrație. Din ipoteză rezultă că $m=2^{k+p}\cdot y$, cu $p\in\mathbb{N}^*$ și y impar. Avem

$$\begin{aligned}u\left(m:2^k\right)&=u\left(2^p\cdot y\right)=u\left(u(2^p)\cdot u(y)\right),\\ u\left(m:2^{k\ \mathrm{MOD}\ 4}\right)&=u\left(2^p\cdot 2^{k-(k\ \mathrm{MOD}\ 4)}\cdot y\right)=u\left(u(2^p)\cdot u(2^{k-(k\ \mathrm{MOD}\ 4)})\cdot u(y)\right)\\ &=u\left(u(2^p)\cdot 6\cdot u(y)\right)=u\left(u(2^p\cdot 6)\cdot u(y)\right)=u\left(u(2^p)\cdot u(y)\right)\end{aligned}$$

și

$$\begin{aligned}u\left(m\cdot 2^{4-(k\ \mathrm{MOD}\ 4)}\right)&=u\left(m:2^{k\ \mathrm{MOD}\ 4}\cdot 2^4\right)=u\left(u\left(m:2^{k\ \mathrm{MOD}\ 4}\right)\cdot u(2^4)\right)\\ &=u\left(u\left(u(2^p)\cdot u(y)\right)\cdot 6\right)=u\left(u(2^p)\cdot 6\cdot u(y)\right)\\ &=u\left(u(2^p\cdot 6)\cdot u(y)\right)=u\left(u(2^p)\cdot u(y)\right),\end{aligned}$$

deci au loc egalitățile din enunț. □

Propoziția 1.5.8. *Pentru orice număr natural n, $n \geq 2$, avem*

$$r(n!) = u\left(r\left(\left\lfloor\frac{n}{5}\right\rfloor!\right) \cdot u\left(\prod_{\substack{1\leq i\leq n\\ i\neq \mathcal{M}5}} i\right) \cdot u\left(2^{4-(\lfloor\frac{n}{5}\rfloor \ MOD\ 4)}\right)\right).$$

Demonstrație. Avem

$$\begin{aligned} n! &= \prod_{\substack{1\leq i\leq n\\ i=\mathcal{M}5}} i \cdot \prod_{\substack{1\leq i\leq n\\ i\neq \mathcal{M}5}} i = \prod_{k=1}^{[n/5]}(5k) \cdot \prod_{\substack{1\leq i\leq n\\ i\neq \mathcal{M}5}} i = 5^{\lfloor\frac{n}{5}\rfloor} \cdot \left\lfloor\frac{n}{5}\right\rfloor! \cdot \prod_{\substack{1\leq i\leq n\\ i\neq \mathcal{M}5}} i \\ &= 10^{\lfloor\frac{n}{5}\rfloor} \cdot \left\lfloor\frac{n}{5}\right\rfloor! \cdot \left(\prod_{\substack{1\leq i\leq n\\ i\neq \mathcal{M}5}} i : 2^{\lfloor\frac{n}{5}\rfloor}\right), \end{aligned}$$

deci utilizând Lema 1.5.1 și Observațiile 1.5.8 și 1.5.6 rezultă că

$$r(n!) = r\left(\left\lfloor\frac{n}{5}\right\rfloor! \cdot \left(\prod_{\substack{1\leq i\leq n\\ i\neq \mathcal{M}5}} i : 2^{\lfloor\frac{n}{5}\rfloor}\right)\right) = u\left(r\left(\left\lfloor\frac{n}{5}\right\rfloor!\right) \cdot u\left(\prod_{\substack{1\leq i\leq n\\ i\neq \mathcal{M}5}} i : 2^{\lfloor\frac{n}{5}\rfloor}\right)\right).$$

Cum $\prod_{\substack{1\leq i\leq n\\ i\neq \mathcal{M}5}} i$ se divide cu $2^{\lfloor\frac{n}{2}\rfloor}$ și $\lfloor\frac{n}{2}\rfloor > \lfloor\frac{n}{5}\rfloor$, aplicând Lema 1.5.2 obținem relația din enunț. □

Definiția 1.5.3. *Pentru orice număr natural n, $n \geq 2$, notăm*

$$s(n) = u\left(\prod_{\substack{1\leq i\leq n\\ i\neq \mathcal{M}5}} i\right).$$

Prin convenție, definim și $s(0) = s(1) = 6$.

Lema 1.5.3. *Pentru orice număr natural n, $n \geq 2$, avem $s(n) = s(u(n))$.*

Demonstrație. Avem

$$\begin{aligned} s(n+10) &= u(\left(u\left(\prod_{\substack{1\leq i\leq n\\ i\neq \mathcal{M}5}} i\right) \cdot u\left(\prod_{\substack{n+1\leq i\leq n+10\\ i\neq \mathcal{M}5}} i\right)\right) \\ &= u\left(s(n) \cdot u(1\cdot 2\cdot 3\cdot 4\cdot 6\cdot 7\cdot 8\cdot 9)\right) = u\left(s(n)\cdot 6\right) = s(n), \end{aligned}$$

deoarece $s(n)$ este evident o cifră pară și nedivizibilă cu 5. Cum $s(10) = s(0) = 6$ și $s(11) = s(1) = 6$, obținem că șirul $(s(n))_{n\geq 0}$ este periodic de perioadă 10 și astfel rezultă egalitatea din enunț. □

Observația 1.5.9. Conform Definiției 1.5.3 avem

$$s(0) = s(1) = s(3) = s(9) = 6,\ s(2) = 2,$$
$$s(4) = s(5) = s(6) = s(8) = 4,\ s(7) = 8.$$

Definiția 1.5.4. *Notăm* $t(k) = u\left(2^{4-k}\right),\ k = \overline{0,3}$.

Observația 1.5.10. Conform definiției anterioare avem

$$t(0) = 6,\ t(1) = 8,\ t(2) = 4,\ t(3) = 2.$$

Teorema 1.5.1 (**relația de recurență a numerelor** $r(n!)$)**.** *Avem*

$$r(n!) = \begin{cases} 1, & \text{dacă } n \in \{0,1\}, \\ u\left(s(u(n)) \cdot t\left(\left\lfloor \frac{n}{5} \right\rfloor \ MOD\ 4\right) \cdot r\left(\left\lfloor \frac{n}{5} \right\rfloor !\right)\right), & \text{dacă } n \geq 2. \end{cases}$$

Demonstrație. Evident, $r(0!) = r(1!) = r(1) = 1$. Pentru $n \geq 2$ egalitatea din enunț este o consecință imediată a Propoziției 1.5.8, Definițiilor 1.5.3 și 1.5.4 și Lemei 1.5.3. □

Exemplul 1.5.4. Pentru a calcula $r(1918!)$ procedăm astfel:

$$\left\lfloor \frac{1918}{5} \right\rfloor = 383, \quad 383 \text{ MOD } 4 = 3,$$
$$\left\lfloor \frac{383}{5} \right\rfloor = 76, \quad 76 \text{ MOD } 4 = 0,$$
$$\left\lfloor \frac{76}{5} \right\rfloor = 15, \quad 15 \text{ MOD } 4 = 3,$$
$$\left\lfloor \frac{15}{5} \right\rfloor = 3, \quad 3 \text{ MOD } 4 = 3,$$
$$\left\lfloor \frac{3}{5} \right\rfloor = 0, \quad 0 \text{ MOD } 4 = 0.$$

Astfel avem

$$\begin{aligned} r(1918!) &= u\left(s(u(1918)) \cdot t\left(\left\lfloor \frac{1918}{5} \right\rfloor \text{ MOD } 4\right) \cdot r\left(\left\lfloor \frac{1918}{5} \right\rfloor !\right)\right) \\ &= u\left(s(8) \cdot t(3) \cdot r(383!)\right) = u\left(4 \cdot 2 \cdot r(383!)\right), \\ r(383!) &= u\left(s(3) \cdot t(0) \cdot r(76!)\right) = u\left(6 \cdot 6 \cdot r(76!)\right), \\ r(76!) &= u\left(s(6) \cdot t(3) \cdot r(15!)\right) = u\left(4 \cdot 2 \cdot r(15!)\right), \\ r(15!) &= u\left(s(5) \cdot t(3) \cdot r(3!)\right) = u\left(4 \cdot 2 \cdot r(3!)\right), \\ r(3!) &= u\left(s(3) \cdot t(0) \cdot r(0!)\right) = u\left(6 \cdot 6 \cdot 1\right) = 6, \end{aligned}$$

deci revenind la relațiile anterioare obținem că

$$\begin{aligned} r(15!) &= u\,(4 \cdot 2 \cdot 6) = 8, \\ r(76!) &= u\,(4 \cdot 2 \cdot 8) = 4, \\ r(383!) &= u\,(6 \cdot 6 \cdot 4) = 4, \\ r(1918!) &= u\,(4 \cdot 2 \cdot 4) = 2. \end{aligned}$$

Conform rezultatelor de mai sus obținem următorul algoritm pentru determinarea lui $r(n!)$.

Algoritmul 1.5.3.

$s_0 \leftarrow 6$; $s_1 \leftarrow 6$; $s_2 \leftarrow 2$; $s_3 \leftarrow 6$; $s_4 \leftarrow 4$;
$s_5 \leftarrow 4$; $s_6 \leftarrow 4$; $s_7 \leftarrow 8$; $s_8 \leftarrow 4$; $s_9 \leftarrow 6$;
$t_0 \leftarrow 6$; $t_1 \leftarrow 8$; $t_2 \leftarrow 4$; $t_3 \leftarrow 2$;
UCNF3(n)**:** `// calculează` $r(n!)$`, recursiv`
if $n \leq 1$ **then**
  return 1;
else
  $p \leftarrow [n/5]$; $i \leftarrow n$ MOD 10; $j \leftarrow p$ MOD 4;
  return $(s_i \cdot t_j \cdot$ **UCNF3**$(p))$ MOD 10;

Analiza complexității algoritmului

Propoziția 1.5.9. *Algoritmul 1.5.3 are complexitatea* $\Theta(\ln n)$.

Demonstrație. Fie $T(n)$ numărul de instrucțiuni executate de algoritm. Evident, avem $T(0) = T(1) = 1$ și

$$T(n) = T\left(\left\lfloor \frac{n}{5} \right\rfloor\right) + 5 \text{ pentru } n \geq 2.$$

Fie $n \geq 2$ și $k = \lfloor \log_5 n \rfloor$, deci $k \leq \log_5 n < k+1$ și astfel $5^k \leq n < 5^{k+1}$.

Cazul 1. Dacă $5^k < n < 5^{k+1}$, atunci $1 < \frac{n}{5^k} < 5$, $\frac{n}{5^{k+1}} < 1$, $\left\lfloor \frac{n}{5^{k+1}} \right\rfloor = 0$, deci adunând egalitățile

$$\begin{aligned} T(n) &= T\left(\left\lfloor \frac{n}{5} \right\rfloor\right) + 5, \\ T\left(\left\lfloor \frac{n}{5} \right\rfloor\right) &= T\left(\left\lfloor \frac{n}{5^2} \right\rfloor\right) + 5, \\ &\cdots \\ T\left(\left\lfloor \frac{n}{5^k} \right\rfloor\right) &= T\left(\left\lfloor \frac{n}{5^{k+1}} \right\rfloor\right) + 5 \end{aligned}$$

obținem $T(n) = T(0) + 5(k+1)$, adică $T(n) = 5\lfloor \log_5 n \rfloor + 6$.

Cazul 2. Dacă $n = 5^k$, atunci $\frac{n}{5^{k+1}} = 1$ și analog cazului 1 obținem că $T(n) = T(1) + 5k = 5\lfloor \log_5 n \rfloor + 1$. □

Observația 1.5.11. Conform Propozițiilor 1.5.5, 1.5.7 și 1.5.9 rezultă că Metoda 3 este mai eficientă decât Metodele 1 și 2.

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 $A = \{a_1, a_2, \ldots, a_n\}$ (**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 $\varnothing$ 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 ($\varnothing$);
- Se alege, pe rând, într-un anumit fel, un element din A neales la pașii 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)**:**
$B \leftarrow \varnothing$;
for $i = \overline{1, n}$ **do**
  $x \leftarrow$**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 $x = a_j \in \{a_i, \ldots, a_n\}$ 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);
$B \leftarrow \varnothing$;
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 $a_1, \ldots, a_n$, 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 $i \in \{0, 1, \ldots, n\}$, 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 $A = \{a_1, a_2, \ldots, a_n\}$ 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*

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

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

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

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

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

Fie

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

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

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

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

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

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

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

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

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 $e(i) = i \; \forall i \in \{1, \ldots, n\}$.

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

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

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 șirurile de numere reale

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

Se cere să se determine două permutări $a_{q(1)}, a_{q(2)}, \ldots, a_{q(n)}$ și $b_{p(1)}, b_{p(2)}, \ldots, b_{p(n)}$ ale celor două șiruri, $q, p \in S_n (S_n =$ 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 $(a_{q(i)})_{i=\overline{1,n}}$ și $(b_{p(i)})_{i=\overline{1,n}}$. Astfel problema anterioară cere determinarea unor permutări ale elementelor vectorilor $(a_1, a_2, \ldots, a_n)$ și $(b_1, b_2, \ldots, b_n)$ 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 șirului $(a_1, a_2, \ldots, a_n)$ neales la pașii anteriori;
  - cel mai mic dintre termenii șirului $(b_1, b_2, \ldots, b_n)$ neales la pașii anteriori.

Descrierea în pseudocod a algoritmului are următoarea formă.

```
MAXIM1 (a, b, n, s) :                    // a = (a_1, ..., a_n), b = (b_1, ..., b_n)
s ← 0;                                                  // s = suma maximă
for i = 1, n do                                                  // pasul i
    k ← i;                  // calculăm termenul minim a_k din (a_i, ..., a_n)
    m ← a[i];
    for j = i + 1, n do
        if a[j] < m then
            k ← j;
            m ← a[j];
    a[i] ↔ a[k];                          // interschimbăm termenii a_i și a_k
    k ← i;                   // calculăm termenul minim b_k din (b_i, ..., b_n)
    m ← b[i];
    for j = i + 1, n do
        if b[j] < m then
            k ← j;
            m ← b[j];
    b[i] ↔ b[k];                            // interschimbăm termenii b_i și b_k
    s ← s + a[i] · 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 = 1, n do
    afișează a[i];
for i = 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 $i \in \{1, 2, \ldots, n\}$ și $j \in \{i+1, i+2, \ldots, n\}$. Numărul acestor perechi este $(n-1) + (n-2) + \cdots + 2 + 1 = \dfrac{n(n-1)}{2}$, 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: $a_1 \leq a_2 \leq \ldots \leq a_n$;

- ordonăm crescător elementele celui de-al doilea șir: $b_1 \leq b_2 \leq \ldots \leq b_n$;
- 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) **:** // $a = (a_1, \ldots, a_n)$, $b = (b_1, \ldots, b_n)$
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 $(a_1, a_2, \ldots, a_n)$ neales la pașii anteriori;
  - cel mai mare dintre termenii șirului $(b_1, b_2, \ldots, b_n)$ neales la pașii anteriori.

Descrierea în pseudocod a algoritmului are următoarea formă.

```
MINIM1 (a, b, n, s) :                  // a = (a_1, ..., a_n), b = (b_1, ..., b_n)
s ← 0;                                                    // s = suma minimă
for i = 1, n do                                                  // pasul i
    k ← i;          // calculăm termenul minim a_k din (a_i, ..., a_n)
    m ← a[i];
    for j = i + 1, n do
        if a[j] < m then
            k ← j;
            m ← a[j];
    a[i] ↔ a[k];                  // interschimbăm termenii a_i și a_k
    k ← i;           // calculăm termenul maxim b_k din (b_i, ..., b_n)
    m ← b[i];
    for j = i + 1, n do
        if b[j] > m then
            k ← j;
            m ← b[j];
    b[i] ↔ b[k];                  // interschimbăm termenii b_i și b_k
    s ← s + a[i] · 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: $a_1 \leq a_2 \leq \ldots \leq a_n$;
- ordonăm descrescător elementele celui de-al doilea șir: $b_1 \geq b_2 \geq \ldots \geq b_n$;
- 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)**:** // $a = (a_1, \ldots, a_n)$, $b = (b_1, \ldots, b_n)$
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)

$$T_1, T_2, \ldots, T_n$$

de lungimi date (de exemplu, în octeți)

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

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 $(T_1, T_2, \ldots, T_n)$, adică are forma

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

  unde $p \in S_n$ este o permutare de ordin n (pentru orice $i \in \{1, \ldots, n\}$, 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 $k \in \{1, \ldots, n\}$, 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 $T_{p(1)}, T_{p(2)}, \ldots, T_{p(k-1)}$.

- 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 $L_1 \leq L_2 \leq \ldots \leq L_n$, 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

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

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 $\min\limits_{p \in S_n} t(p) = t(e)$. □

Observația 2.2.5. Mai mult, deoarece șirul

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

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 curenta, 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 încărca greutatea maximă G, unde $G > 0$, și n obiecte $O_1, \ldots, O_n$, $n \in \mathbb{N}^*$. Pentru fiecare obiect O_i, $i \in \{1, \ldots, n\}$, 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 incărcare 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 $x = (x_1, \ldots, x_n)$ astfel încât
$$\begin{cases} x_i \in [0,1], \ \forall\, i \in \{1, \ldots, n\}, \\ \sum_{i=1}^{n} x_i g_i \leq G, \end{cases}$$
ultima inegalitate exprimând faptul că greutatea totală încărcată în rucsac nu trebuie să depășească greutatea maximă.

- *Câștigul (total)* corespunzător soluției $x = (x_1, \ldots, x_n)$ este
$$f(x) = \sum_{i=1}^{n} x_i c_i.$$

- O *soluție optimă* a problemei este orice soluție $x^* = (x_1^*, \ldots, x_n^*)$ astfel încât
$$f(x^*) = \max\{f(x) \mid x = \text{soluție a problemei}\}.$$

- Dacă suma greutăților 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 $x^* = (1, 1, \ldots, 1)$, corespunzătoare încărcării integrale a tuturor celor n obiecte în rucsac, fiind evident singura soluție optimă. Astfel în continuare putem presupune că
$$\sum_{i=1}^{n} g_i > G. \tag{2.3.1}$$

- Pentru orice soluție optimă $x^* = (x_1^*, \ldots, x_n^*)$ avem
$$\sum_{i=1}^{n} x_i^* g_i = G \tag{2.3.2}$$
(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 $k \in \{1, \ldots, n\}$ a.î. $x_k^* < 1$. Considerând vectorul $x' = (x_1', \ldots, x_n')$ 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 *algoritm Greedy* pentru rezolvarea problemei.

Algoritmul 2.3.1. Vom utiliza următoarea *strategie Greedy*:

- Ordonăm obiectele descrescător după câștigul lor unitar:

$$\frac{c_1}{g_1} \geq \frac{c_2}{g_2} \geq \cdots \geq \frac{c_n}{g_n}. \tag{2.3.3}$$

- 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)**:** // $g = (g_1, \ldots, g_n)$, $c = (c_1, \ldots, c_n)$
// $C =$ câștigul total
SORTARE(g, c, n); // se sortează obiectele descrescător
// după câștigul lor unitar
$C \leftarrow 0$;
for $i = \overline{1, n}$ **do** $x[i] \leftarrow 0$;
$R \leftarrow G$; // $R =$ greutatea disponibilă pentru rucsac
$i \leftarrow 1$;
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
    $x[i] \leftarrow \frac{R}{g[i]}$;
    $C \leftarrow C + x[i]c[i]$;
    $R \leftarrow 0$;
AFISARE(C, x, n); // se afișează câștigul total maxim C
// și soluția optimă $x = (x_1, \ldots, x_n)$

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

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

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

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

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

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

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 $x^* = (x_1^*, \ldots, x_n^*)$ a problemei pentru care

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

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

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 $x_i = x_i^*, \ \forall\, i \in \{1, \ldots, k\}$, 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 $x^{**} = (x_1^{**}, \ldots, x_n^{**})$ prin

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

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

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

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 $x_i^{**} \in [0, 1]$, $\forall\, i \in \{1, \ldots, n\}$ 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 $x^{**} = (x_1^{**}, \ldots, x_n^{**})$ este o soluție a problemei.

Avem

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

Conform (2.3.3) rezultă că

$$\frac{c_i}{g_i} \leq \frac{c_k}{g_k},\ \forall\, i \in \{k+1, \ldots, n\}. \tag{2.3.8}$$

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

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

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 $x^* = (x_1^*, \ldots, x_n^*)$ pentru care

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

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 greutatea maximă $G = 40$, din $n = 10$ obiecte ce au greutățile 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 eemplu, 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 $S_1, \ldots, S_n$, $n \in \mathbb{N}^*$. Pentru fiecare spectacol S_i, $i \in \{1, \ldots, n\}$, 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 $P \subseteq \{I_1, \ldots, I_n\}$ 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 cîte două).

- O *soluție optimă* a problemei este orice soluție $P^* \subseteq \{I_1, \ldots, I_n\}$ 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:

$$b_1 \leq b_2 \leq \cdots \leq b_n. \tag{2.4.1}$$

- 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 $c = (c_1, \ldots, c_n)$, 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)**:** // $a = (a_1, \ldots, a_n)$, $b = (b_1, \ldots, b_n)$
    // $c = (c_1, \ldots, c_n)$, $m =$ 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** $c[i] \leftarrow 0$;
$t \leftarrow a[1] - 1$; // $t =$ 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 $P = \{I'_1, \ldots, I'_m\}$ submulțimea de intervale calculată de algoritm, unde

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

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

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

deci

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

Rezultă că

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

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

Demonstrăm prin inducție după $k \in \{0, 1, \ldots, m\}$ că există o soluție optimă $P^* = \{I^*_1, \ldots, I^*_p\}$ a problemei, $p \in \mathbb{N}^*$, cu

$$I^*_1 = [a^*_1, b^*_1],\ I^*_2 = [a^*_2, b^*_2], \ldots, I^*_p = [a^*_p, b^*_p], \quad b^*_1 < b^*_2 < \cdots < b^*_p, \tag{2.4.2}$$

pentru care

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

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 $P^* = \{I^*_1, \ldots, I^*_p\}$ a problemei, ce verifică (2.4.2), pentru care

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

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

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

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

Avem două cazuri.

Cazul 1) $I_k' = I_k^*$. Atunci $I_i' = I_i^*$, $\forall\, i \in \{1, \ldots, k\}$, 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

$$b_k' \leq b_k^*. \tag{2.4.5}$$

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 $P^{**} = \{I_1^{**}, \ldots, I_p^{**}\}$ prin

$$I_i^{**} = [a_i^{**}, b_i^{**}] = \begin{cases} I_i^*, \text{ dacă } i \neq k, \\ I_i', \text{ dacă } i = k. \end{cases} \tag{2.4.6}$$

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

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

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

$$a_k' > b_{k-1}^*. \tag{2.4.8}$$

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

$$\begin{aligned} a_1^* < b_1^* < a_2^* < b_2^* < \cdots < a_{k-1}^* < b_{k-1}^* < a_k' < b_k' < \\ < a_{k+1}^* < b_{k+1}^* < \cdots < a_p^* < b_p^* \end{aligned}$$

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

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

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

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

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 $P^* = \{I_1^*, \ldots, I_p^*\}$ pentru care

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

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

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

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:

$$a_1 \geq a_2 \geq \cdots \geq a_n.$$

- 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 $c = (c_1, \ldots, c_n)$, 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, ..., a_n), b = (b_1, ..., b_n)
      // c = (c_1, ..., c_n), m = numărul de spectacole selectate
SORTARE(a, b, n);      // se sortează spectacolele crescător
                            // după timpul lor de începere a_i
m ← 0;                                          // inițializări
for i = 1, n do c[i] ← 0;
t ← b[n] + 1;              // t = timpul de începere al ultimului
                                              // spectacol selectat
for i = n, 1, −1 do         // parcurgem spectacolele în ordinea
    |                  // descrescătoare a timpilor de începere
    if b[i] < t then
    |   c[i] ← 1;    // selectăm intervalul (spectacolul) curent
    |   m ← m + 1;
    |   t ← 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 crescătoare 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.

2.5 Arbori parțiali de cost minim

Definiția 2.5.1. *Un **graf ponderat** este o pereche (G, c), unde $G = (V, E)$ este un graf (V fiind mulțimea nodurilor iar E mulțimea muchiilor sau arcelor) iar $c : E \to \mathbb{R}$ este o funcție numită **pondere** (**cost**). Pentru orice $e \in E$, $c(e)$ se numește **ponderea** (**costul**) muchiei (în cazul grafurilor neorientate) sau arcului (în cazul grafurilor orientate) e.*

Definiția 2.5.2. *Fie (G, c) un graf ponderat, $G = (V, E)$.*

*a) Dacă $H = (U, F)$ este un subgraf al lui G, atunci **costul** (**ponderea**) lui H este*

$$c(H) = \sum_{e \in F} c(e)$$

(adică suma costurilor muchiilor sau arcelor sale).

b) Un arbore parțial $T^\star = (V, F)$ al lui G cu proprietatea că

$$c(T^\star) = \min\{c(T) | T = \text{ arbore parțial al lui } G\}$$

*se numește **arbore parțial de cost minim** (**APM**) al grafului ponderat (G, c).*

Observația 2.5.1. Un graf ponderat are arbori parțiali de cost minim dacă și numai dacă este conex (adică între orice două noduri distincte există cel puțin un lanț).

Un arbore este un graf conex și fără cicluri.

Problema determinării arborilor parțiali de cost minim are numeroase aplicații practice. Prezentăm în continuare doi algoritmi fundamentali pentru rezolvarea acestei probleme.

Algoritmul 2.5.1 (**Kruskal**). Fie (G, c) un graf ponderat conex cu $G = (V, E)$, $V = \{v_1, \ldots, v_n\}$. Algoritmul are $n - 1$ pași.

- La pasul i, $i = \overline{1, n-1}$, dintre muchiile neselectate la pașii anteriori se selectează o muchie $e_i \in E$ de cost minim cu proprietatea că nu formează cicluri cu muchiile $\{e_1, \ldots, e_{i-1}\}$ selectate la pașii anteriori.

Algoritmul 2.5.2 (**Prim**). Fie (G, c) un graf ponderat conex cu $G = (V, E)$, $V = \{v_1, \ldots, v_n\}$. Algoritmul are n pași.

- La pasul 0 se selectează un nod arbitrar $x_0 \in V$.

- La pasul i, $i = \overline{1, n-1}$, se selectează o muchie $e_i = [x_j, x_i] \in E$ de cost minim cu proprietatea că are ca extremități un nod $x_j \in V$ selectat la un pas anterior și celălalt nod $x_i \in V$ neselectat la pașii anteriori; se selectează și nodul x_i.

Teorema 2.5.1 (**de corectitudine a algoritmilor Kruskal și Prim**). *În contextul Algoritmilor Kruskal sau Prim, fie $F = \{e_1, \ldots, e_{n-1}\}$ mulțimea muchiilor selectate. Atunci $T = (V, F)$ este un arbore parțial de cost minim al grafului ponderat (G, c).*

Demonstrație. Fie $T_0 = (V, \emptyset)$ și $T_i = (V, \{e_1, \ldots, e_i\})$, $\forall i \in \{1, 2, \ldots, n-1\}$, unde $e_1, e_2, \ldots, e_{n-1}$ sunt muchiile selectate, în această ordine, de Algoritmul Kruskal sau de Algoritmul Prim.

Graful G fiind conex, selectarea muchiei e_i este posibilă la fiecare pas i, iar T_i este o pădure (graf fără cicluri) parțială a lui G (afirmație evidentă pentru Algoritmul Kruskal, iar pentru Algoritmul Prim este o consecință a faptului că nodurile neselectate la pasul i sunt izolate în T_i).

Se arată prin inducție după $i \in \{0, 1, \ldots, n-1\}$ că există un APM $T^\star = (V, F^\star)$ astfel încât

$$T_i \subseteq T^\star \text{ (adică } \{e_1, \ldots, e_i\} \subseteq F^\star).$$

Luând $i = n - 1$ în această relație obținem că $T \subseteq T^\star$, unde $T = T_{n-1} = (V, F)$, $F = \{e_1, \ldots, e_{n-1}\}$ (mulțimea muchiilor selectate de algoritm) iar $T^\star$ este un APM. Dar T și $T^\star$ au fiecare câte $n - 1$ muchii, deci $T = T^\star$ și astfel T este un APM al grafului dat. □

Exemplul 2.5.1. Fie graful ponderat (G, c) reprezentat în Figura 2.5.1, unde costul fiecărei muchii este scris lângă segmentul corespunzător acesteia.

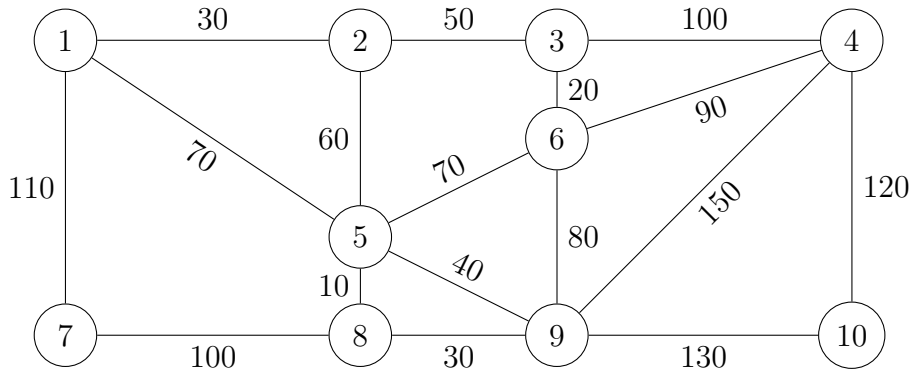

Figura 2.5.1:

Aplicarea Algoritmului Kruskal este evidențiată în următorul tabel:

Pas	Muchia selectată	Costul ei
1	[5, 8]	10
2	[3, 6]	20
3	[1, 2]	30
4	[8, 9]	30
5	[2, 3]	50
6	[2, 5]	60
7	[4, 6]	90
8	[7, 8]	100
9	[4, 10]	120

Arborele parțial de cost minim obținut este reprezentat în Figura 2.5.2. Costul acestui APM este de 510.

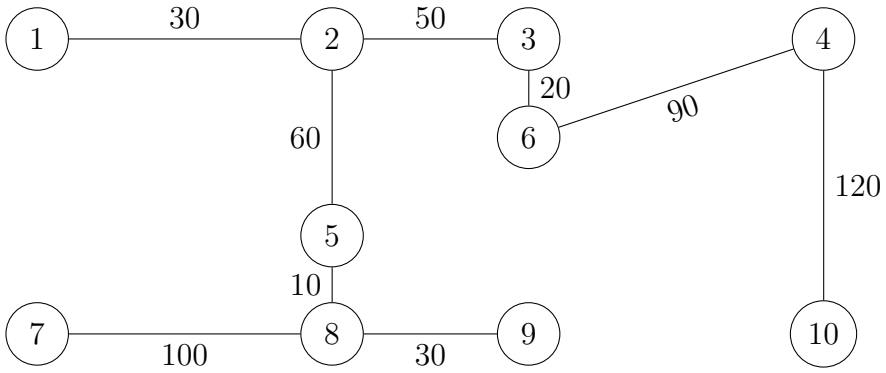

Figura 2.5.2:

Aplicarea Algoritmului Prim pentru același graf este evidențiată în următorul tabel:

Pas	Muchia selectată	Costul ei	Nodul selectat
0	-	-	1
1	$[1,2]$	30	2
2	$[2,3]$	50	3
3	$[3,6]$	20	6
4	$[2,5]$	60	5
5	$[5,8]$	10	8
6	$[8,9]$	30	9
7	$[6,4]$	90	4
8	$[8,7]$	100	7
9	$[4,10]$	120	10

Arborele parțial de cost minim obținut este deci același cu cel obținut prin aplicarea Algoritmului Kruskal.

Observația 2.5.2. Algoritmii Kruskal și Prim sunt specifici **metodei de programare Greedy**. Algoritmul Kruskal selectează muchii, în ordinea crescătoare a costurilor, subgrafurile induse pe parcurs de acestea nefiind neapărat conexe. Algoritmul Prim selectează muchii și noduri, nu neapărat în ordinea crescătoare a costurilor muchiilor, iar subgrafurile induse pe parcurs de muchiile selectate sunt conexe.

În implementări optime, se poate arăta că Algoritmul Kruskal are complexitatea $\mathcal{O}(m \ln n)$ (fiind necesară sortarea muchiilor după cost), iar Algoritmul Prim are complexitatea $\mathcal{O}(n^2)$ în cazul memorării grafului prin matricea de adiacență (o astfel de implementare va fi prezentată în continuare), unde n și m reprezintă numerele de noduri, respectiv de muchii ale grafului dat. Graful fiind conex, $m \geq n-1$.

Pentru grafuri simple, $m \leq \frac{n(n-1)}{2}$. Folosind și inegalitatea $\ln n \leq n-1$, obținem că Algoritmul Kruskal este mai eficient pentru grafuri "sărace" în muchii, iar Algoritmul Prim este mai eficient pentru grafuri "bogate" în muchii.

Observația 2.5.3. Pentru implementarea Algoritmului Kruskal, memorăm graful ponderat conex (G,c), unde $G=(V,E)$, $V=\{1,\ldots,n\}$, $E=\{e_1,\ldots,e_m\}$, într-o matrice cu 3 linii și m coloane $P=(p_{ik})_{\substack{i=\overline{1,3}\\ k=\overline{1,m}}}$ având semnificația:

$$\text{dacă } e_k=[x_k,y_k]\in E, \text{ atunci } p_{1k}=x_k,\ p_{2k}=y_k \text{ și } p_{3k}=c(e_k).$$

Utilizăm un vector S cu semnificația

$$S[k]=\begin{cases} 1, \text{ dacă } e_k \text{ a fost selectată},\\ 0, \text{ în caz contrar}, \end{cases}$$

$\forall\, k \in \{1, \ldots, m\}$ și un vector CC cu semnificația

> $CC[i]$ = numărul componentei conexe în care se află nodul i în graful indus de muchiile selectate, $\forall\, i \in \{1, \ldots, n\}$.

Astfel o muchie $[x, y]$ nu formează cicluri cu muchiile selectate dacă și numai dacă

$$CC[x] \neq CC[y].$$

Descrierea în pseudocod a algoritmului are următoarea formă.

KRUSKAL:
SORTARE(P); // se sortează coloanele matricei P
// crescător după costurile muchiilor
for $i = \overline{1, m}$ **do** $S[i] \leftarrow 0$;
for $i = \overline{1, n}$ **do** $CC[i] \leftarrow i$;
$cost \leftarrow 0$; // costul APM
$poz \leftarrow 0$; // căutarea următoarei muchii e_k ce va fi
// selectată începe de pe poziția $poz + 1$
for $l = \overline{1, n-1}$ **do** // pasul l
  $k \leftarrow poz$;
  repeat
    $k \leftarrow k + 1$; $x \leftarrow p_{1k}$; $y \leftarrow p_{2k}$; $c \leftarrow p_{3k}$;
  while $(CC[x] = CC[y])$;
  $S[k] \leftarrow 1$; // selectăm $e_k = [x, y]$
  $cost \leftarrow cost + c$; $poz \leftarrow k$;
  $aux \leftarrow CC[y]$; // actualizăm vectorul CC prin unificarea
  // componentelor conexe ale lui x și y
  for $i = \overline{1, n}$ **do**
    if $(CC[i] = aux)$ **then** $CC[i] \leftarrow CC[x]$;

Observația 2.5.4. Pentru implementarea Algoritmului Prim, memorăm graful ponderat conex și simplu (G, c), unde $G = (V, E)$, $V = \{1, \ldots, n\}$, $E = \{e_1, \ldots, e_m\}$, cu ajutorul unei matrice $C = (c_{ij})_{i,j=\overline{1,n}}$ a **costurilor** (**directe**) având semnificația

$$c_{ij} = \begin{cases} c([i, j]), \text{ dacă } [i, j] \in E, \\ 0, \text{ dacă } i = j, \\ \infty, \text{ în rest,} \end{cases} \tag{2.5.1}$$

$\forall\, i, j \in \{1, \ldots, n\}$. Pentru grafuri neorientate, matricea C este simetrică. În cazul grafurilor nesimple putem lua

$$c_{ij} = \min\{c(e) | e = [i, j] \in E\}.$$

Utilizăm un vector S cu semnificația

$$S[i] = \begin{cases} 1, \text{ dacă nodul } i \text{ a fost selectat,} \\ 0, \text{ în caz contrar} \end{cases}$$

și doi vectori t și $TATA$ având semnificația

$t[i] =$ costul minim al unei muchii $[i, j]$ de la nodul i la un nod selectat j,
$TATA[i] =$ nodul j ce atinge minimul în $t[i]$, $\forall\, i \in \{1, \ldots, n\}$.

Descrierea în pseudocod a algoritmului are următoarea formă.

PRIM :
$S[1] \leftarrow 1$; // selectăm nodul 1
$cost \leftarrow 0$; // costul APM
for $i = \overline{2, n}$ **do** // inițializări
  $S[i] \leftarrow 0$; $t[i] \leftarrow c_{i1}$; $TATA[i] \leftarrow 1$;
for $l = \overline{1, n-1}$ **do** // căutăm nodul y și muchia $[x, y]$
  // ce vor fi selectate la pasul l
  $min \leftarrow \infty$;
  for $i = \overline{2, n}$ **do**
    if $(S[i] = 0)$ *and* $(t[i] < min)$ **then**
      $min \leftarrow t[i]$; $y \leftarrow i$;
  $S[y] \leftarrow 1$; // selectăm nodul y
  $x \leftarrow TATA[y]$; // și muchia $[x, y]$
  $cost \leftarrow cost + c_{xy}$;
  for $i = \overline{2, n}$ **do** // actualizăm vectorii t și $TATA$
    if $(S[i] = 0)$ *and* $(t[i] > c_{iy})$ **then**
      $t[i] \leftarrow c_{iy}$; $TATA[i] \leftarrow y$;

2.6 Distanțe și drumuri minime. Algoritmul lui Dijkstra

Problema determinării distanțelor și drumurilor minime între nodurile unui graf ponderat apare în numeroase aplicații practice. În continuare vom prezenta un algoritm clasic pentru rezolvarea acestei probleme.

Definiția 2.6.1. *Fie* (G, c) *un graf ponderat, unde* $G = (V, E)$, $V = \{v_1, \ldots, v_n\}$ *iar* $c : E \to \mathbb{R}_+$.

a) Dacă $\mu = (x_0, e_1, x_1, \ldots, x_{k-1}, e_k, x_k)$ *este un drum al grafului* G*, unde* $x_0, x_1, \ldots, x_k \in V$*,* $e_1, \ldots, e_k \in E$*,* $k \in \mathbb{N}$*, atunci* ***costul (ponderea)*** *lui* μ *este*

$$c(\mu) = \begin{cases} 0, & \text{dacă } k = 0, \\ \sum_{i=1}^{k} c(e_i), & \text{dacă } k \geq 1 \end{cases}$$

(adică suma costurilor arcelor sau muchiilor sale).

b) Fie $x, y \in V$*. Un drum* $\mu^\star = (x, \ldots, y)$ *în graful* G *cu proprietatea că*

$$c(\mu^\star) = \min\{c(\mu) | \mu \text{ este drum de la } x \text{ la } y \text{ în } G\}$$

se numește ***drum minim (drum de cost minim, drum de pondere minimă)*** *de la* x *la* y *în graful ponderat* (G, c)*. Costul* $c(\mu^\star)$ *al acestui drum minim se numește* ***distanța minimă*** *de la* x *la* y *în graful* (G, c)*.*

Observația 2.6.1. Eliminând eventualele circuite $C_1, \ldots, C_p$ dintr-un drum μ de la nodul x la nodul y obținem un drum elementar (adică un drum având nodurile distincte două câte două, cu excepția extremităților, care pot fi egale) μ' de la x la y cu proprietatea că $c(\mu') = c(\mu) - \sum_{k=1}^{p} c(C_k) \leq c(\mu)$.

Dacă drumul μ este minim atunci și drumul elementar μ' este minim și $c(e) = 0$ pentru orice muchie sau arc e al circuitelor $C_1, \ldots, C_p$. Deci existența unui drum minim de la x la y implică existența unui drum minim elementar de la x la y. Astfel distanța minimă de la x la y poate fi considerată ca fiind costul minim al unui drum elementar de la x la y. Mai mult, dacă funcția cost c este strict pozitivă, atunci orice drum minim este elementar.

Observația 2.6.2. Mulțimea drumurilor elementare fiind evident finită, avem echivalența: există drumuri minime de la x la y dacă și numai dacă există drumuri de la x la y.

Observația 2.6.3. Distanța minimă de la un nod x la el însuși este egală cu zero, drumul minim elementar de la x la x fiind drumul de lungime zero, $\mu = (x)$.

Observația 2.6.4. Dacă $\mu = (x_0, x_1, \ldots, x_k)$ este un drum minim de la x_0 la x_k, atunci orice subdrum $\mu' = (x_i, x_{i+1}, \ldots, x_j)$ $(0 \leq i \leq j \leq k)$ al său este un drum minim de la x_i la x_j (**principiul optimalității al lui Bellman**). Afirmația poate fi justificată ușor prin reducere la absurd.

Exemplul 2.6.1. Fie graful ponderat (G, c) reprezentat în Figura 2.6.1.

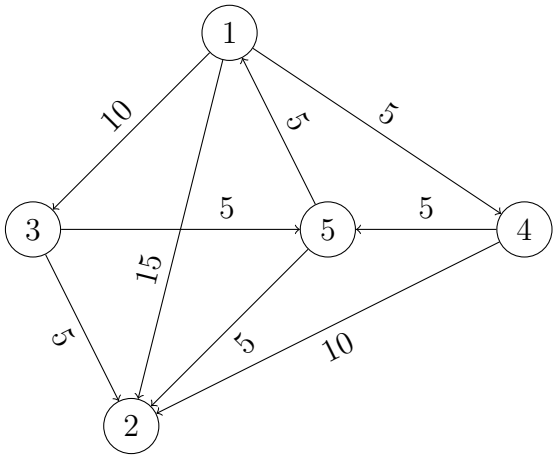

Figura 2.6.1:

Drumurile elementare de la nodul 1 la nodul 5 sunt $\mu_1 = (1, 3, 5)$, având costul $c(\mu_1) = 10 + 5 = 15$, $\mu_2 = (1, 4, 5)$, având costul $c(\mu_2) = 5 + 5 = 10$, deci μ_2 este un drum minim de la 1 la 5. Astfel distanța minimă de la nodul 1 la nodul 5 este $c(\mu_2) = 10$.

Definiția 2.6.2. *Fie (G, c) un graf ponderat, unde $G = (V, E)$, $V = \{v_1, \ldots, v_n\}$, $c : E \to \mathbb{R}_+$.*

a) ***Matricea distanțelor (costurilor) directe*** *asociată grafului (G, c) este matricea $C = (c_{ij})_{i,j=\overline{1,n}}$ definită prin*

$$c_{ij} = \begin{cases} 0, \text{ dacă } i = j, \\ \min\{c(e) | e = (v_i, v_j) \in E\}, \text{ dacă } i \neq j \text{ și } \exists\, (v_i, v_j) \in E, \\ \infty, \text{ dacă } i \neq j \text{ și } \nexists\, (v_i, v_j) \in E \end{cases}$$

(pentru grafuri neorientate (v_i, v_j) desemnând de fapt muchia $[v_i, v_j]$).

b) ***Matricea distanțelor (costurilor) minime*** *asociată grafului (G, c) este matricea $C^\star = (c^\star_{ij})_{i,j=\overline{1,n}}$ definită prin*

$$c^\star_{ij} = \begin{cases} c(\mu^\star), \ \mu^\star = \text{ drum minim de la } v_i \text{ la } v_j, \\ \qquad \text{dacă } \exists\, \mu = (v_i, \ldots, v_j) \text{ drum în } G, \\ \infty, \text{ în caz contrar.} \end{cases}$$

Observația 2.6.5. Evident, pentru orice graf neorientat atât matricea distanțelor directe cât și matricea distanțelor minime sunt matrice simetrice.

Observația 2.6.6. Conform Observației 2.6.1, putem să înlocuim termenul de "drum" cu cel de "drum elementar" în definiția anterioară.

Conform Observației 2.6.2, punctul b) din definiția anterioară este o extindere a definiției distanței minime de la punctul b) al Definiției 2.6.1.

Conform Observației 2.6.3, $c^{\star}_{ii} = 0 \ \forall \, i \in \{1, \ldots, n\}$.

Exemplul 2.6.2. Matricele distanțelor directe, respectiv minime asociate grafului din Exemplul 2.6.1 sunt

$$C = \begin{pmatrix} 0 & 15 & 10 & 5 & \infty \\ \infty & 0 & \infty & \infty & \infty \\ \infty & 5 & 0 & \infty & 5 \\ \infty & 10 & \infty & 0 & 5 \\ 5 & 5 & \infty & \infty & 0 \end{pmatrix}, \ C^{\star} = \begin{pmatrix} 0 & 15 & 10 & 5 & 10 \\ \infty & 0 & \infty & \infty & \infty \\ 10 & 5 & 0 & 15 & 5 \\ 10 & 10 & 20 & 0 & 5 \\ 5 & 5 & 15 & 10 & 0 \end{pmatrix}.$$

Vom expune în continuare un algoritm pentru determinarea distanțelor minime și a drumurilor minime de la un nod fixat, numit și nod sursă, la toate nodurile grafului ponderat dat.

Algoritmul 2.6.1 (**Dijkstra**). Fie (G, c) un graf ponderat, $G = (V, E)$, $V = \{v_1, v_2, \ldots, v_n\}$, $c : E \to \mathbb{R}_+$. Fie $C = (c_{ij})_{i,j=\overline{1,n}}$ matricea distanțelor directe asociată grafului (G, c) și fie $v_s \in V$ un nod arbitrar fixat, numit **nod sursă**. Distanțele minime de la nodul v_s la nodurile grafului sunt calculate și memorate într-un vector $t = (t_1, \ldots, t_n)$ astfel:

- La pasul 1 se selectează nodul sursă v_s și se ia $t_s = 0$;
- La pasul k, $2 \leq k \leq n$, se cunosc nodurile $v_{i_1}, \ldots, v_{i_{k-1}}$ selectate la pașii anteriori și distanțele corespondente $t_{i_1}, \ldots, t_{i_{k-1}}$.

  a) Dacă nu mai există nici-o muchie sau arc de la un nod selectat $v_j \in \{v_{i_1}, \ldots, v_{i_{k-1}}\}$ la un nod neselectat $v_i \in V \setminus \{v_{i_1}, \ldots, v_{i_{k-1}}\}$, atunci se ia $t_i = \infty$ pentru orice nod neselectat $v_i \in V \setminus \{v_{i_1}, \ldots, v_{i_{k-1}}\}$ și algoritmul se încheie.

  b) În caz contrar se selectează un nod $v_{i_k} \in V \setminus \{v_{i_1}, \ldots, v_{i_{k-1}}\}$ cu proprietatea că există un nod selectat $v_{j_k} \in \{v_{i_1}, \ldots, v_{i_{k-1}}\}$ astfel încât

  $$t_{j_k} + c_{j_k i_k} = \min\{t_j + c_{ji} | v_j \in \{v_{i_1}, \ldots, v_{i_{k-1}}\}, v_i \in V \setminus \{v_{i_1}, \ldots, v_{i_{k-1}}\}\}. \tag{2.6.1}$$

  Se ia

  $$t_{i_k} = t_{j_k} + c_{j_k i_k} \tag{2.6.2}$$

  și se trece la pasul $k + 1$.

Observația 2.6.7. Evident, algoritmul execută cel mult n pași.

Teorema 2.6.1 (**de corectitudine a Algoritmului Dijkstra**). *În contextul Algoritmului Dijkstra, avem*

$$t_i = c^{\star}_{si},\ \forall\, i \in \{1, \ldots, n\}$$

(adică distanța t_i calculată de algoritm este chiar distanța minimă de la v_s la v_i).

Demonstrație. Se arată prin inducție după k că nodul v_{i_k} selectat la pasul k și distanța corespondentă t_{i_k} calculată la acel pas verifică egalitatea din enunț, adică

$$t_{i_k} = c^{\star}_{si_k},$$

și, în plus, $t_{i_k} < \infty$. Evident, pentru orice nod v_i rămas neselectat în urma executării ultimului pas al algoritmului avem $t_i = \infty = c^{\star}_{si}$.

Într-adevăr, dacă ar exista un drum elementar minim

$$\mu = (v_s = y_0, y_1, \ldots, y_p = v_i),$$

luând $l \in \{0, 1, \ldots, p-1\}$ indicele maxim pentru care y_l este selectat (există, deoarece $y_0 = v_s$ este selectat) atunci y_{l+1} ar fi neselectat deși există o muchie sau un arc de la y_l la y_{l+1}, contradicție cu descrierea modului de încheiere a algoritmului. □

Exemplul 2.6.3. Pentru graful ponderat din Exemplul 2.6.1, luând ca nod sursă nodul 1, aplicarea Algoritmului Dijkstra este evidențiată în următorul tabel:

Pas	Nodul selectat	Distanța minimă
1	1	0
2	4	5
3	3	10
4	5	10
5	2	15

De exemplu, la pasul 3 avem deja selectate nodurile $i_1 = 1$ și $i_2 = 4$, cu distanțele minime $t_1 = c^{\star}_{11} = 0$ și $t_4 = c^{\star}_{14} = 5$. Se selectează nodul $i_3 = 3$, cu distanța minimă $t_3 = 10$, deoarece

$$\begin{aligned}&\min\{t_1 + c_{12}, t_1 + c_{13}, t_1 + c_{15}, t_4 + c_{42}, t_4 + c_{43}, t_4 + c_{45}\}\\ &\quad = \min\{0 + 15, 0 + 10, 0 + \infty, 5 + 10, 5 + \infty, 5 + 5\} = 10 = t_1 + c_{13}.\end{aligned}$$

Observația 2.6.8. Algoritmul Dijkstra este specific **metodei de programare Greedy**, el selectând nodurile în ordinea crescătoare a distanței față de nodul sursă.

Observația 2.6.9. Pentru implementarea Algoritmului Dijkstra, considerăm că $V = \{1, \ldots, n\}$ și că nodul sursă este $s \in V$. Utilizăm un vector S având semnificația

$$S[i] = \begin{cases} 1, \text{ dacă nodul } i \text{ a fost selectat,} \\ 0, \text{ în caz contrar,} \end{cases} \quad \forall\, i \in \{1, \ldots, n\}$$

și un vector t având semnificația

$t[i]$ = distanța minimă de la nodul sursă s la nodul i, $\forall\, i \in \{1, \ldots, n\}$, calculat conform (2.6.1) și (2.6.2).

Pentru determinarea drumurilor minime de la nodul s la nodurile grafului vom utiliza și un vector $TATA$ având semnificația

$TATA[i]$ = nodul j ce este predecesorul direct al nodului i pe drumul minim de la s la i, $\forall\, i \in \{1, \ldots, n\}$.

Astfel în vectorul $TATA$ se memorează un arbore compus din drumuri minime de la nodul sursă la nodurile grafului, numit **arborele drumurilor minime**.

Dacă $i = i_k$ este nodul selectat la pasul k, atunci $j = j_k$ se determină conform egalității (2.6.1).

Descrierea în pseudocod a algoritmului are forma următoare.

```
DIJKSTRA(s) :
for i = 1,n do                                          // iniţializări
    S[i] ← 0; t[i] ← ∞; TATA[i] ← ∞;
t[s] ← 0; TATA[s] ← 0;                                  // s este nodul sursă
repeat
        // selectăm următorul nod x, în ordinea crescătoare
                        // a distanţelor minime de la s la x
    min ← ∞;
    for i = 1,n do
        if (S[i] = 0) and (t[i] < min) then
            min ← t[i];
            x ← i;
    if (min < ∞) then                                   // există x, îl selectăm
        S[x] ← 1;
        for i = 1,n do                  // actualizăm vectorii t şi TATA
            if (S[i] = 0) and (c_xi < ∞) then
                if (t[i] > t[x] + c_xi) then
                    t[i] ← t[x] + c_xi;
                    TATA[i] ← x;
while (min < ∞);
```

Exemplul 2.6.4. Pentru graful ponderat din Exemplul 2.5.1, luând ca nod sursă nodul $s = 1$, aplicarea Algoritmului Dijkstra este evidenţiată în următorul tabel:

Pas	Nodul selectat x	$TATA[x]$	Distanţa minimă $t[x]$
1	1	0	0
2	2	1	30
3	5	1	70
4	3	2	80
5	8	5	80
6	6	3	100
7	7	1	110
8	9	5	110
9	4	3	180
10	10	9	240

Arborele drumurilor minime, memorat în vectorul $TATA$, este reprezentat în Figura 2.6.2.

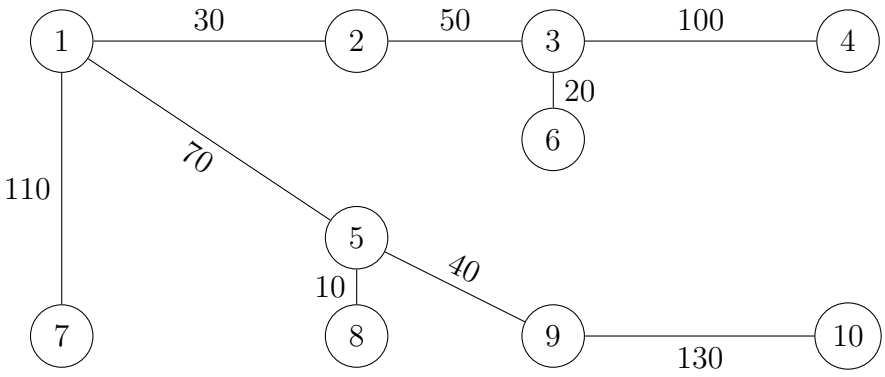

Figura 2.6.2:

Deci drumurile minime determinate de algoritm sunt:

- de la 1 la 1: $[1]$;
- de la 1 la 2: $[1, 2]$;
- de la 1 la 3: $[1, 2, 3]$;
- de la 1 la 4: $[1, 2, 3, 4]$;
- de la 1 la 5: $[1, 5]$;
- de la 1 la 6: $[1, 2, 3, 6]$;
- de la 1 la 7: $[1, 7]$;
- de la 1 la 8: $[1, 5, 8]$;
- de la 1 la 9: $[1, 5, 9]$;
- de la 1 la 10: $[1, 5, 9, 10]$.

Observația 2.6.10. Implementarea anterioară necesită $\mathcal{O}(n^2)$ operații (deoarece blocul "**repeat**" se execută de cel mult n ori și necesită de fiecare dată cel mult n comparații pentru determinarea nodului selectat x și cel mult n comparații și n adunări pentru actualizarea vectorilor t și $TATA$). Aceasta este de fapt și complexitatea Algoritmului Dijkstra (în implementarea optimă) în cazul memorării grafului prin matricea distanțelor directe.

2.7 Fluxuri în rețele

Definiția 2.7.1. *O **rețea** (**rețea de transport**) are forma $R = (G, s, t, c)$, unde:*

- *$G = (V, E)$ este un graf orientat simplu (adică fără bucle sau arce egale).*

  *V se numește și **mulțimea nodurilor** rețelei, iar E se numește și **mulțimea arcelor** rețelei;*

- *$s, t \in V$ sunt două noduri a.î. $s \neq t$.*

  *Nodul s se numește **nodul sursă** (**intrarea**) al rețelei, iar nodul t se numește **nodul destinație** (**ieșirea**) al rețelei;*

- *$c : E \to \mathbb{R}_+$ este o funcție numită **funcție capacitate**.*

  *Pentru orice $e \in E$, $c(e)$ se numește **capacitatea arcului** e.*

Pentru simplificarea notațiilor, extindem funcția capacitate

$$c : V \times V \to \mathbb{R}_+,\ c(i,j) = \begin{cases} c(i,j), & \text{dacă } (i,j) \in E, \\ 0, & \text{dacă } (i,j) \notin E. \end{cases} \tag{2.7.1}$$

Exemplul 2.7.1. În Figura 2.7.1 este reprezentată o rețea R, capacitățile fiind menționate pe arce. Considerăm că această rețea are intrarea $s = 1$ și ieșirea $t = 6$.

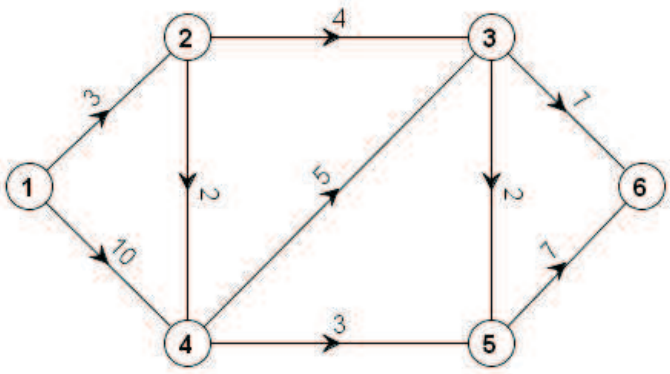

Figura 2.7.1:

Definiția 2.7.2. *Fie $R = (G, s, t, c)$ o rețea, unde $G = (V, E)$. Un* ***flux*** *în rețeaua R este o funcție $f : V \times V \to \mathbb{R}$ ce verifică următoarele două proprietăți:*

$$0 \leq f(i, j) \leq c(i, j),\ \forall i, j \in V, \tag{2.7.2}$$

$$\sum_{j \in V} f(j, i) = \sum_{j \in V} f(i, j),\ \forall i \in V \setminus \{s, t\}. \tag{2.7.3}$$

$f(i, j)$ reprezintă ***fluxul transportat pe arcul*** *(i, j).*

Relația (2.7.2) se numește ***condiția de mărginire a fluxului****, iar relația (2.7.3) se numește* ***condiția de conservare a fluxului****.*

Observația 2.7.1. Evident,

$$f(i, j) = 0\ \forall (i, j) \notin E,$$

deci, analog funcției capacitate c, și funcția flux f poate fi definită (restrânsă) doar pe mulțimea arcelor rețelei (adică $f : E \to \mathbb{R}$).

Observația 2.7.2. În particular, $f(i, j) = 0\ \forall i, j \in V$ este un flux în orice rețea, numit **fluxul nul**.

Exemplul 2.7.2. Pentru rețeaua din Figura 2.7.1, un exemplu de flux este reprezentat în Figura 2.7.2. Pe fiecare arc (i, j) sunt menționate fluxul $f(i, j)$ și capacitatea $c(i, j)$, în această ordine.

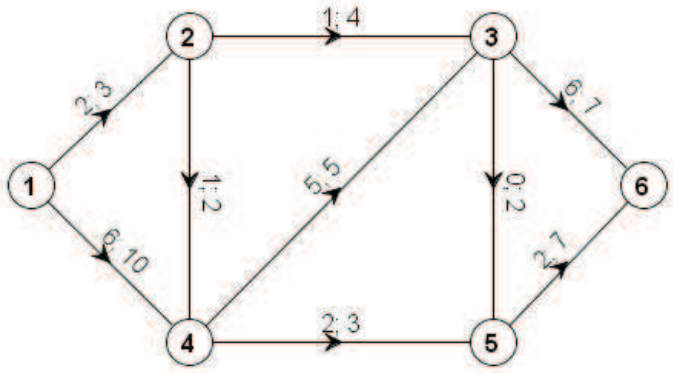

Figura 2.7.2:

Lema 2.7.1. *Fie $R = (G, s, t, c)$ o rețea, unde $G = (V, E)$. Pentru orice flux f în rețeaua R are loc egalitatea*

$$\sum_{i \in V} f(i, t) - \sum_{i \in V} f(t, i) = -\left(\sum_{i \in V} f(i, s) - \sum_{i \in V} f(s, i) \right).$$

Definiția 2.7.3. *Fie $R = (G, s, t, c)$ o rețea, unde $G = (V, E)$. Pentru orice flux f în rețeaua R, numărul*

$$v(f) = \sum_{i \in V} f(i,t) - \sum_{i \in V} f(t,i) = -\left(\sum_{i \in V} f(i,s) - \sum_{i \in V} f(s,i) \right)$$

*se numește **valoarea fluxului** f (fluxul net ce ajunge la nodul destinație, fluxul net ce iese din nodul sursă).*

Exemplul 2.7.3. Fluxul din Figura 2.7.2 are valoarea $v(f) = 2 + 6 = 8$.

Definiția 2.7.4. *Fie $R = (G, s, t, c)$ o rețea. Un flux f^* în rețeaua R cu proprietatea că*

$$v(f^*) = \max\{v(f) \mid f = \text{flux în } R\}$$

*se numește **flux de valoare maximă (flux maxim)** în rețeaua R.*

Problema determinării fluxurilor maxime într-o rețea are numeroase aplicații practice. Vom prezenta un algoritm pentru rezolvarea acestei probleme. Definim câteva noțiuni ajutătoare.

Definiția 2.7.5. *Dacă $\mu = [x_0, x_1, \ldots, x_k]$ este un lanț elementar în graful orientat $G = (V, E)$, atunci arcele sale de forma $(x_i, x_{i+1}) \in E$ $(i \in \{0, \ldots, k-1\})$ se numesc **arce directe** pentru lanțul μ, iar arcele sale de forma $(x_{i+1}, x_i) \in E$ $(i \in \{0, \ldots, k-1\})$ se numesc **arce inverse** pentru lanțul μ.*

Exemplul 2.7.4. Pentru graful din Figura 2.7.2, lanțul $[1, 2, 3, 4, 5, 6]$ are arcele directe $(1,2), (2,3), (4,5), (5,6)$ și arcul invers $(4,3)$.

Definiția 2.7.6. *Fie f un flux în rețeaua $R = (G, s, t, c)$. Un **C-lanț** în rețeaua R relativ la fluxul f este un lanț elementar μ în graful G ce verifică următoarele două proprietăți:*

$$f(i,j) < c(i,j),\ \forall (i,j) = \text{arc direct al lui } \mu,$$
$$f(i,j) > 0,\ \forall (i,j) = \text{arc invers al lui } \mu.$$

Definiția 2.7.7. *Fie f un flux în rețeaua $R = (G, s, t, c)$ și fie μ un C-lanț în rețeaua R relativ la fluxul f.*

a) Pentru orice arc (i, j) al lui μ, numărul

$$r_\mu(i,j) = \begin{cases} c(i,j) - f(i,j), & \text{dacă } (i,j) \text{ este arc direct al lui } \mu, \\ f(i,j), & \text{dacă } (i,j) \text{ este arc invers al lui } \mu \end{cases}$$

*se numește **capacitatea reziduală (reziduul)** a arcului (i, j) relativ la C-lanțul μ.*

b) Numărul

$$r(\mu) = \min\{r_\mu(i,j) \mid (i,j) = \textit{arc al lui } \mu\}$$

se numește ***capacitatea reziduală (reziduul)*** *a C-lanțului* μ.

Observația 2.7.3. Conform definițiilor anterioare, pentru orice C-lanț μ avem $r(\mu) > 0$.

Exemplul 2.7.5. Pentru rețeaua și fluxul reprezentate în Figura 2.7.2, lanțul $\mu = [1, 2, 3, 4, 5, 6]$ este un C-lanț. Reziduurile pe arcele acestui C-lanț sunt

$$r_\mu(1,2) = 3 - 2 = 1,\ r_\mu(2,3) = 4 - 1 = 3,\ r_\mu(4,3) = 5,$$

$$r_\mu(4,5) = 3 - 2 = 1,\ r_\mu(5,6) = 7 - 2 = 5,$$

deci reziduul pe acest C-lanț este $r(\mu) = 1$.

Definiția 2.7.8. *Fie* f *un flux în rețeaua* $R = (G, s, t, c)$. *Un C-lanț de la* s *la* t *în rețeaua* R *relativ la fluxul* f *se numește* ***lanț de creștere*** *(în rețeaua* R *relativ la fluxul* f*).*

Exemplul 2.7.6. Lanțul $\mu = [1, 2, 3, 4, 5, 6]$ din exemplul anterior este un lanț de creștere.

Lema 2.7.2. *Fie* $R = (G, s, t, c)$ *o rețea, unde* $G = (V, E)$. *Fie* f *un flux în rețeaua* R *și fie* μ *un lanț de creștere în rețeaua* R *relativ la fluxul* f. *Atunci funcția* $f' : V \times V \to \mathbb{R}$ *definită prin*

$$f'(i,j) = \begin{cases} f(i,j) + r(\mu), & \textit{dacă } (i,j) \textit{ este arc direct al lui } \mu, \\ f(i,j) - r(\mu), & \textit{dacă } (i,j) \textit{ este arc invers al lui } \mu, \\ f(i,j), & \textit{dacă } (i,j) \textit{ nu este arc al lui } \mu \end{cases}$$

este un flux în rețeaua R *și*

$$v(f') = v(f) + r(\mu).$$

Observația 2.7.4. În contextul lemei anterioare, conform Observației 2.7.3 avem

$$v(f') = v(f) + r(\mu) > v(f).$$

Inegalitatea justifică denumirea lui μ drept *lanț de creștere* a fluxului f, iar egalitatea justifică denumirea lui $r(\mu)$ drept *capacitatea reziduală* a lanțului μ.

Definiția 2.7.9. *Fluxul* f' *definit în lema anterioară se numește* ***fluxul obținut prin mărirea fluxului*** f ***de-a lungul lanțului de creștere*** μ *și se notează cu*

$$f' = f \oplus r(\mu).$$

Exemplul 2.7.7. Pentru rețeaua R și fluxul f reprezentate în Figura 2.7.2, lanțul $\mu = [1, 2, 3, 4, 5, 6]$ este un lanț de creștere având reziduul $r(\mu) = 1$. Fluxul $f' = f \oplus r(\mu)$ obținut prin aplicarea lemei anterioare este reprezentat în Figura 2.7.3. Valoarea acestui flux este $v(f') = v(f) + r(\mu) = 8 + 1 = 9$.

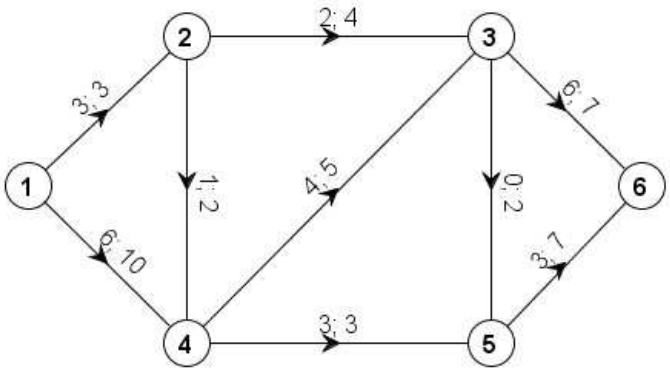

Figura 2.7.3:

Definiția 2.7.10. *Fie $R = (G, s, t, c)$ o rețea, unde $G = (V, E)$. O **secțiune** (**tăietură**) în rețeaua R este o pereche $(S, T) \in V \times V$ ce verifică următoarele proprietăți:*

$$S \cup T = V,\ S \cap T = \emptyset,$$
$$s \in S,\ t \in T.$$

Observația 2.7.5. Dacă (S, T) este o secțiune în rețeaua $R = (G, s, t, c)$, unde $G = (V, E)$, atunci $V = S \cup T$ este o partiție a mulțimii V a nodurilor rețelei (adică $V = S \cup T$, $S \neq \emptyset$, $T \neq \emptyset$, $S \cap T = \emptyset$) a.î. $s \in S$ și $t \in T$. Rezultă că numărul de secțiuni ale rețelei R este egal cu numărul de submulțimi $S \setminus \{s\} \subseteq V \setminus \{s, t\}$, deci cu

$$2^{n-2},\ \text{unde } n = \text{card}\,(V).$$

Definiția 2.7.11. *Fie $R = (G, s, t, c)$ o rețea, unde $G = (V, E)$. Pentru orice secțiune (S, T) în rețeaua R, numărul*

$$c(S, T) = \sum_{i \in S} \sum_{j \in T} c(i, j)$$

*se numește **capacitatea secțiunii** (S, T).*

Observația 2.7.6. Conform relației (2.7.1), capacitatea unei secțiuni (S, T) este suma capacităților tuturor arcelor ce au extremitatea inițială în S și extremitatea finală în T.

Exemplul 2.7.8. Pentru rețeaua reprezentată în Figura 2.7.1,

$$(S, T) = (\{1, 2, 5\}, \{3, 4, 6\})$$

este o secțiune având capacitatea

$$c(S, T) = c(1, 4) + c(2, 3) + c(2, 4) + c(5, 6) = 10 + 4 + 2 + 7 = 23.$$

Lema 2.7.3. *Fie $R = (G, s, t, c)$ o rețea. Pentru orice flux f și orice secțiune (S, T) în rețeaua R avem*

$$v(f) = \sum_{i \in S} \sum_{j \in T} (f(i, j) - f(j, i))$$

(adică valoarea fluxului este egală cu fluxul net ce traversează secțiunea).

Lema 2.7.4. *Fie $R = (G, s, t, c)$ o rețea.*

a) Pentru orice flux f și orice secțiune (S, T) în rețeaua R avem

$$v(f) \leq c(S, T).$$

b) Dacă f^* este un flux și (S^*, T^*) este o secțiune în rețeaua R astfel încât

$$v(f^*) = c(S^*, T^*),$$

atunci f^* este un flux de valoare maximă și (S^*, T^*) este o secțiune de capacitate minimă în rețeaua R.

Teorema 2.7.1 (de caracterizare a unui flux de valoare maximă). *Fie $R = (G, s, t, c)$ o rețea. Un flux f în rețeaua R este un flux de valoare maximă dacă și numai dacă nu există lanțuri de creștere în rețeaua R relativ la fluxul f.*

Demonstrație. "$\Rightarrow$" Fie f un flux de valoare maximă în R. Dacă ar exista un lanț de creștere μ în R relativ la fluxul f, atunci conform Lemei 2.7.2 ar rezulta că funcția $f' = f \oplus r(\mu)$ ar fi un flux în R și $v(f') = v(f) + r(\mu) > v(f)$ (conform Observației 2.7.4) contradicție. Deci nu există lanțuri de creștere în R relativ la fluxul f.

"$\Leftarrow$" Fie f un flux în R a.î. nu există lanțuri de creștere în R relativ la f. Fie

$$\begin{cases} S = \{i \in V \mid \text{există } C\text{-lanțuri de la } s \text{ la } i \text{ în } R \text{ relativ la } f\}, \\ T = \{i \in V \mid \text{nu există } C\text{-lanțuri de la } s \text{ la } i \text{ în } R \text{ relativ la } f\}. \end{cases} \tag{2.7.4}$$

Evident, $S \cup T = V$ și $S \cap T = \emptyset$.

De asemenea, avem $s \in S$ (deoarece $[s]$ este un C-lanț de la s la s în R relativ la f) și $t \in T$ (deoarece nu există lanțuri de creștere, adică C-lanțuri de la s la t în R relativ la f). Deci (S, T) este o secțiune în rețeaua R.

Fie $i \in S$ și $j \in T$ arbitrar fixați. Deoarece $i \in S$ rezultă că există un C-lanț $\mu = [s = x_0, x_1, \ldots, x_k = i]$ în R relativ la f. Atunci $x_0, x_1, \ldots, x_k \in S$ (sublanțurile C-lanțului μ sunt tot C-lanțuri), deci $j \notin V(\mu)$.

Dacă am avea $f(i, j) < c(i, j)$ sau $f(j, i) = 0$, atunci ar rezulta că $[s = x_0, x_1, \ldots, x_k = i, j]$ ar fi un C-lanț în R relativ la f, ceea ce ar contrazice apartenența $j \in T$. Deci

$$f(i, j) = c(i, j) \text{ și } f(j, i) = 0,$$

pentru orice $i \in S$ și $j \in T$.

Aplicând Lema 2.7.3, avem

$$v(f) = \sum_{i \in S} \sum_{j \in T} (f(i, j) - f(j, i)) = \sum_{i \in S} \sum_{j \in T} (c(i, j) - 0) = c(S, T),$$

deci conform Lemei 2.7.4 rezultă că f este un flux de valoare maximă în rețeaua R (iar (S, T) este o secțiune de capacitate minimă în rețeaua R). □

Observația 2.7.7. Dacă se cunoaște un flux f de valoare maximă într-o rețea R, atunci formulele (2.7.4) determină o secțiune (S, T) de capacitate minimă, deci orice algoritm pentru determinarea unui flux de valoare maximă bazat pe caracterizarea dată în teorema anterioară rezolvă și problema determinării unei secțiuni de capacitate minimă.

Definiția 2.7.12. *Fie f un flux în rețeaua $R = (G, s, t, c)$, unde $G = (V, E)$.*

Funcția $r : V \times V \to \mathbb{R}_+$ definită prin

$$r(i, j) = \begin{cases} c(i, j) - f(i, j), & \text{dacă } f(i, j) < c(i, j), \\ f(j, i), & \text{dacă } f(j, i) > 0 \text{ și } f(i, j) = c(i, j), \\ 0, & \text{în rest} \end{cases}$$

se numește ***capacitatea reziduală*** *(****reziduul****) a rețelei R relativ la fluxul f, iar graful orientat $G_f = (V, E_f)$ definit prin*

$$E_f = \{(i, j) \in V \times V \mid r(i, j) > 0\}$$

se numește ***graful rezidual*** *al rețelei R relativ la fluxul f.*

Observația 2.7.8. Fie f un flux în rețeaua $R = (G, s, t, c)$. Fie r și G_f capacitatea reziduală, respectiv graful rezidual ale rețelei R relativ la fluxul f. Fie $\mu = [x_0, x_1, \ldots, x_k]$ un lanț elementar în graful G. Evident, avem echivalențele:

μ este un C-lanț în R relativ la $f \Leftrightarrow r(x_i, x_{i+1}) > 0,\ \forall i \in \{0, \ldots, k-1\} \Leftrightarrow$

$\Leftrightarrow (x_0, x_1, \ldots, x_k)$ este un drum elementar în graful rezidual G_f;

μ este un lanț de creștere în R relativ la $f \Leftrightarrow$

$\Leftrightarrow (x_0, x_1, \ldots, x_k)$ este un drum elementar de la s la t în graful rezidual G_f.

Exemplul 2.7.9. Pentru rețeaua R și fluxul f' reprezentate în Figura 2.7.3, graful rezidual $G_{f'}$ este reprezentat în Figura 2.7.4, capacitățile reziduale fiind menționate pe arce.

$(1, 4, 2, 3, 6)$ este un drum elementar în graful rezidual $G_{f'}$, deci $\mu' = [1, 4, 2, 3, 6]$ este un lanț de creștere în rețeaua R relativ la fluxul f'.

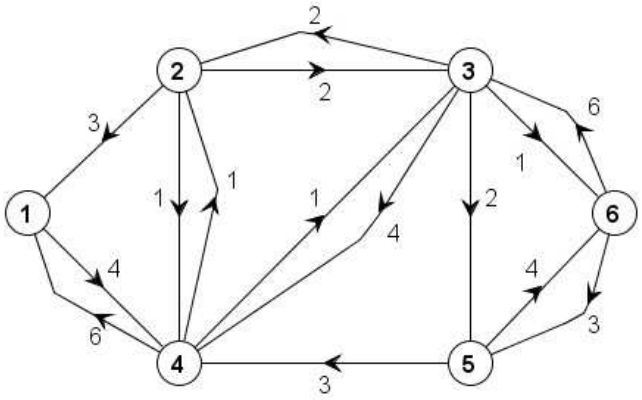

Figura 2.7.4:

Conform rezultatelor de mai sus obținem următorul algoritm pentru determinarea unui flux de valoare maximă într-o rețea.

Algoritmul 2.7.1 (**Ford-Fulkerson**). Fie $R = (G, s, t, c)$ o rețea, unde $G = (V, E)$.

Conform Teoremei 2.7.1, schița algoritmului, descrisă în pseudocod, are forma

FORD_FULKERSON :
$f \leftarrow 0$; `// sau` $f \leftarrow f_0$`, unde` f_0 `este un flux`
`// disponibil inițial`
repeat
  if ***(EXISTĂ_LANȚ_DE_CREȘTERE)*** **then**
    `// există lanțuri de creștere`
    `// relativ la fluxul curent` f
    $\mu \leftarrow$ **LANȚ_DE_CREȘTERE**; `// se determină`
    `// un astfel de lanț de creștere`
    $f \leftarrow f \oplus r(\mu)$; `// se mărește valoarea fluxului curent,`
    `// de-a lungul lanțului de creștere`
while ***(EXISTĂ_LANȚ_DE_CREȘTERE)***;
`// nu mai există lanțuri de creștere,`
`// deci fluxul curent este de valoare maximă`
AFIȘARE(f); `// se afișează fluxul de valoare maximă`

În continuare detaliem implementarea acestui algoritm. Presupunem că

$$V = \{1, 2, \ldots, n\},\ s = 1,\ t = n,$$

cu $n \geq 2$.

Pentru depistarea și memorarea eventualelor lanțuri de creștere relativ la fluxul curent vom utiliza doi vectori SEL și $TATA$ având semnificația

$$SEL[i] = \begin{cases} 1, & \text{dacă există un } C\text{-lanț de la nodul } s = 1 \text{ la nodul } i, \\ 0, & \text{în caz contrar,} \end{cases}$$

$$TATA[i] = \text{predecesorul direct al nodului } i \text{ pe } C\text{-lanțul de la nodul } s = 1 \text{ la nodul } i,$$

$\forall i \in \{1, \ldots, n\}$.

Algoritmul descris în pseudocod are forma (detaliată)

```
FORD_FULKERSON :
CITIRE_REȚEA;                          // se citește rețeaua dată
for i = 1,n do
    for j = 1,n do  f[i,j] ← 0;                    // fluxul inițial
vmax ← 0;                                  // valoarea fluxul maxim
repeat   // se caută lanțuri de creștere a fluxului curent,
                                          // folosind Observația 2.7.8
    CALCUL_REZIDUURI;      // se determină reziduul și
     // graful rezidual ale rețelei relativ la fluxul curent
    SEL[1] ← 1;                        // se selectează nodul sursă
    for i = 2,n do SEL[i] ← 0;
    TATA[1] ← 0;                    // pt.  memorarea C-lanțurilor
                                     // ce pornesc din nodul sursă
    DF(1);            // se parcurge graful rezidual, memorând
                    // C-lanțurile ce pornesc din nodul sursă;
                      // parcurgerea DF, poate fi înlocuită cu
                   // parcurgerea BF (Algoritmul Edmonds-Karp)
    if (SEL[n] = 1) then      // s-a selectat nodul destinație,
                     // deci există lanț de creștere a fluxului
        rmin ← ∞;                  // reziduul minim, de-a lungul
                                           // lanțului de creștere
        DET_LANT_CR;   // se determină lanțul de creștere
                                 // a fluxului și reziduul minim
        MĂRIRE_FLUX;    // se mărește fluxul curent, de-a
                                 // lungul lanțului de creștere
        vmax ← vmax + rmin;;       // se actualizează valoarea
                                                // fluxului curent
while (SEL[n] = 1);
              // nu mai există lanțuri de creștere a fluxului,
                              // deci fluxul curent este maxim
AFIȘARE_REZULTATE;
                    // se afișează fluxul maxim și valoarea sa,
                 // eventual și secțiunea de capacitate minimă,
                         // calculată conform (2.7.4) astfel:
                             // S = {i ∈ {1,...,n} | SEL[i] = 1},
                             // T = {i ∈ {1,...,n} | SEL[i] = 0}
```

Funcțiile utilizate sunt descrise în continuare.

```
CALCUL_REZIDUURI :
for i = 1,n do
    for j = 1,n do
        if (f[i,j] < c[i,j]) then
            r[i,j] ← c[i,j] − f[i,j];
        else
            if (f[j,i] > 0) then
                r[i,j] ← f[j,i];
            else
                r[i,j] ← 0;
```

```
DF(i) :                                          // recursiv
for j = 1,n do
    if (r[i,j] > 0 and SEL[j] = 0) then
        TATA[j] ← i; SEL[j] ← 1;;
        DF(j);
```

```
DET_LANT_CR :        // se determină reziduul minim de-a
    // lungul lanțului de creștere, parcurs de la n către 1
j ← n;
while (j ≠ 1) do
    i ← TATA[j];
    if (rmin > r[i,j]) then  rmin ← r[i,j];
    j ← i;
```

```
MĂRIRE_FLUX :          // se mărește fluxul de-a lungul
          // lanțului de creștere, parcurs de la n către 1
j ← n;
while (j ≠ 1) do
    i ← TATA[j];
    if (c[i,j] > f[i,j]) then
        f[i,j] ← f[i,j] + rmin;
    else
        f[j,i] ← f[j,i] − rmin;
    j ← i;
```

Teorema 2.7.2 (**de corectitudine a Algoritmului Ford-Fulkerson**). *În contextul Algoritmului 2.7.1, fie $R = (G, s, t, c)$ rețeaua dată, unde $G = (V, E)$. Presupunem că toate capacitățile arcelor rețelei sunt numere întregi, adică*

$$c(i, j) \in \mathbb{N},\ \forall (i, j) \in E.$$

Fie f_0 fluxul inițial. Presupunem că toate componentele fluxului f_0 sunt numere întregi, adică

$$f_0(i,j) \in \mathbb{N},\ \forall (i,j) \in E,$$

de exemplu f_0 = fluxul nul.

Fie μ_1, μ_2, ..., μ_k lanțurile de creștere succesive obținute, $k \geq 0$, și fie $f_1 = f_0 \oplus r_0(\mu_1)$, $f_2 = f_1 \oplus r_1(\mu_2)$, ..., $f_k = f_{k-1} \oplus r_{k-1}(\mu_k)$ fluxurile succesive obținute, unde μ_1 este lanț de creștere relativ la f_0, μ_2 este lanț de creștere relativ la f_1, ..., μ_k este lanț de creștere relativ la f_{k-1}, și nu mai există lanțuri de creștere relativ la f_k. Atunci f_k este un flux de valoare maximă în rețeaua R.

Mai mult, toate componentele fluxului f_k și valoarea acestuia sunt numere întregi.

Demonstrație. Conform Lemei 2.7.2, $f_1, f_2, \ldots, f_k$ sunt fluxuri și $v(f_1) = v(f_0) + r_0(\mu_1)$, $v(f_2) = v(f_1) + r_1(\mu_2) = v(f_0) + r_0(\mu_1) + r_1(\mu_2)$, ..., $v(f_k) = v(f_{k-1}) + r_{k-1}(\mu_k) = v(f_0) + r_0(\mu_1) + r_1(\mu_2) + \cdots + r_{k-1}(\mu_k)$.

Deoarece toate capacități $c(i,j)$, $i,j \in V$, și toate componentele $f_0(i,j)$, $i,j \in V$ ale fluxului inițial sunt numere întregi, conform Definiției 2.7.7 și Lemei 2.7.2 rezultă succesiv că

$$\begin{aligned}
(r_0)_{\mu_1}(i,j) \in \mathbb{N}^*,\ \forall (i,j) \in E(\mu_1),\ r_0(\mu_1) \in \mathbb{N}^*,\\
f_1(i,j) \in \mathbb{N},\ \forall i,j \in V,\ v(f_1) \in \mathbb{N}^*;\\
(r_1)_{\mu_2}(i,j) \in \mathbb{N}^*,\ \forall (i,j) \in E(\mu_2),\ r_1(\mu_2) \in \mathbb{N}^*,\\
f_2(i,j) \in \mathbb{N},\ \forall i,j \in V,\ v(f_2) \in \mathbb{N}^*; \ldots,\\
(r_{k-1})_{\mu_k}(i,j) \in \mathbb{N}^*,\ \forall (i,j) \in E(\mu_k),\ r_{k-1}(\mu_k) \in \mathbb{N}^*,\\
f_k(i,j) \in \mathbb{N},\ \forall i,j \in V,\ v(f_k) \in \mathbb{N}^*. \qquad (2.7.5)
\end{aligned}$$

Deci

$$v(f_k) = v(f_0) + r_0(\mu_1) + r_1(\mu_2) + \cdots + r_{k-1}(\mu_k) \geq v(f_0) + k.$$

Evident, $(\{s\}, V \setminus \{s\})$ este o secțiune în R având capacitatea

$$c(\{s\}, V \setminus \{s\}) = \sum_{j \in V \setminus \{s\}} c(s,j) \leq (\text{card}\,(V) - 1) c_{max},$$

unde

$$c_{max} = \max\{c(i,j) \mid (i,j) \in E\}$$

(capacitatea maximă a arcelor rețelei). Conform Lemei 2.7.4 avem

$$v(f_k) \leq c(\{s\}, V \setminus \{s\}),$$

și astfel obținem

$$k \leq (\text{card}\,(V) - 1)c_{max} - v(f_0),$$

deci numărul k de pași ai algoritmului (numărul de lanțuri de creștere succesive construite) este finit, adică există $k \in \mathbb{N}$ astfel încât rețeaua R nu mai conține lanțuri de creștere relativ la fluxul f_k. Conform Teoremei 2.7.1 rezultă că f_k este un flux de valoare maximă în R. Conform (2.7.5), toate componentele fluxului f_k și valoarea acestuia sunt numere întregi. □

Exemplul 2.7.10. Pentru rețeaua R din Figura 2.7.1, aplicarea Algoritmului Ford-Fulkerson este evidențiată în Figurile 2.7.5-2.7.10.

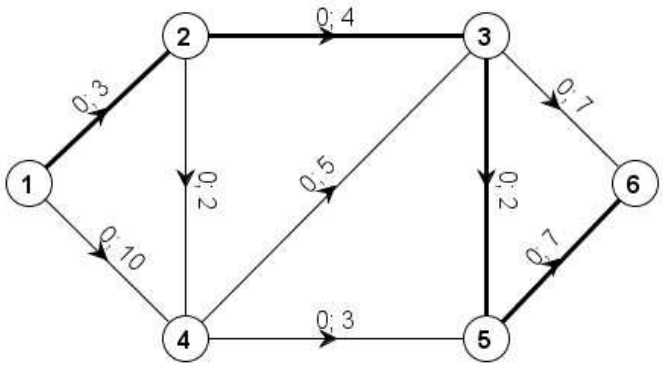

Figura 2.7.5:

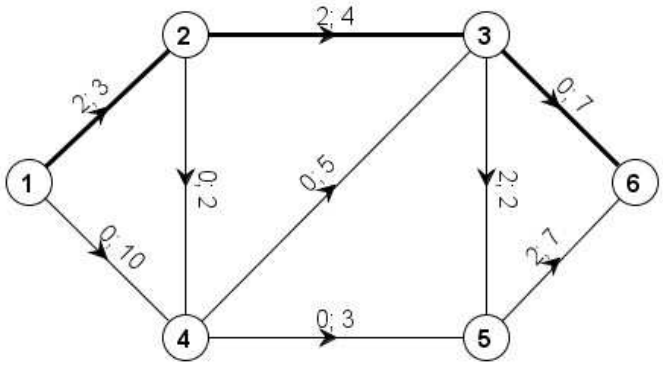

Figura 2.7.6:

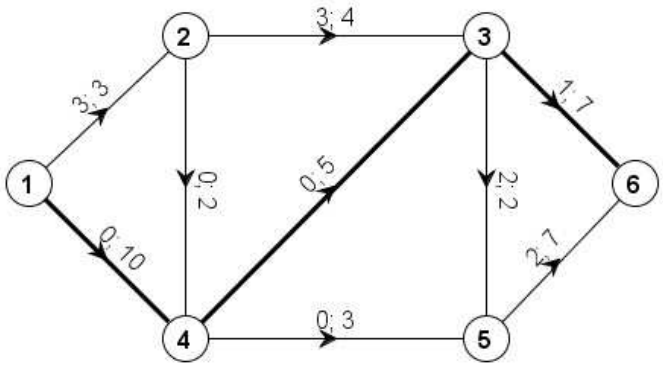

Figura 2.7.7:

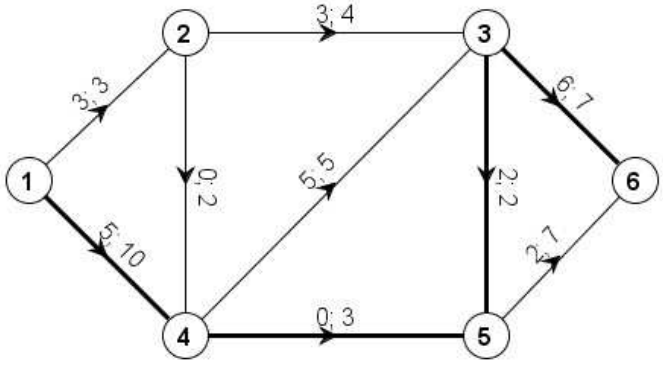

Figura 2.7.8:

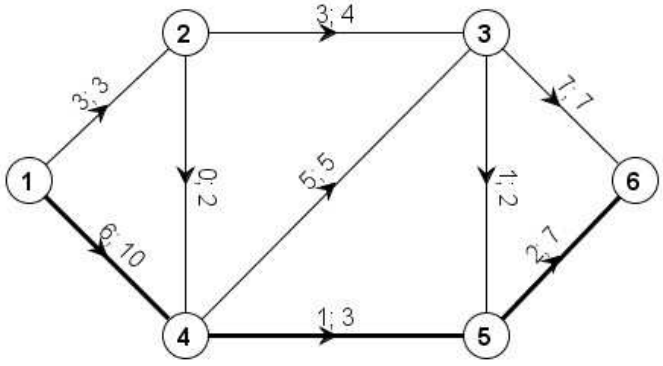

Figura 2.7.9:

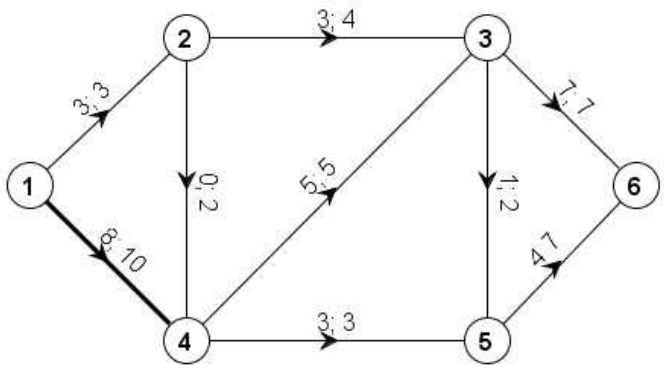

Figura 2.7.10:

Lanțurile succesive de creștere relativ la fluxul curent sunt evidențiate prin îngroșare, și anume:

$$\mu_1 = [1, 2, 3, 5, 6],$$
$$\mu_2 = [1, 2, 3, 6],$$
$$\mu_3 = [1, 4, 3, 6],$$
$$\mu_4 = [1, 4, 5, 3, 6],$$
$$\mu_5 = [1, 4, 5, 6].$$

Fluxul reprezentat în Figura 2.7.10 este de valoare maximă, deoarece nu mai există lanțuri de creștere în R relativ la acest flux. Valoarea acestui flux este 11.

Pentru acest flux maxim, avem C-lanțuri de la nodul sursă $s = 1$ doar la nodurile 1 și 4, aceste C-lanțuri sunt evidențiate prin îngroșare în Figura 2.7.10. Conform (2.7.4) rezultă că

$$(S, T) = (\{1, 4\}, \{2, 3, 5, 6\})$$

este o secțiune de capacitate minimă în R.

Observația 2.7.9. Algoritmul Ford-Fulkerson este specific **metodei de programare Greedy**. Pentru o rețea cu n noduri și m arce având toate capacitățile numere întregi și pentru un flux inițial având toate componentele numere întregi, de exemplu fluxul nul, conform demonstrației Teoremei 2.7.2 rezultă că numărul de lanțuri de creștere necesare măririi succesive a fluxului inițial, până se ajunge la un flux de valoare maximă, este de cel mult $(n-1)c_{max}$, unde c_{max} este capacitatea maximă a arcelor rețelei. Pentru fiecare flux curent, algoritmul necesită calculul reziduurilor celor m arce, parcurgerea acestora pentru depistarea unui eventual lanț de creștere, determinarea acestui lanț (de lungime cel mult m) și mărirea fluxului curent de-a

lungul lanțului de creștere, operații având complexitatea $O(m)$. Rezultă că Algoritmul Ford-Fulkerson are complexitatea $O(mnc_{max})$.

Observația 2.7.10. Pentru o rețea având toate capacitățile numere raționale, putem aplica Algoritmul Ford-Fulkerson astfel:

- se înmulțesc toate capacitățile arcelor cu numitorul lor comun M, obținându-se o rețea având toate capacitățile numere întregi;
- se determină un flux de valoare maximă în această rețea, aplicând Algoritmul Ford-Fulkerson (pentru un flux inițial având toate componentele numere întregi, de exemplu fluxul nul);
- se împart toate componentele acestui flux prin M, obținându-se un flux de valoare maximă în rețeaua inițială.

Evident, toate componentele acestui flux maxim și valoarea lui sunt numere raționale.

Observația 2.7.11. Pentru o rețea având și capacități numere iraționale, Algoritmul Ford-Fulkerson poate să fie inoperabil, fiind posibil ca numărul de pași ai algoritmului (numărul de lanțuri de creștere succesive construite) să fie infinit, deoarece valorile reziduale ale lanțurilor de creștere succesive construite pot forma o serie convergentă.

Totuși, prin impunerea unor modalități judicioase de construire a lanțurilor de creștere succesive în cadrul Algoritmului Ford-Fulkerson, se obțin algoritmi care rezolvă problema fluxului maxim și în cazul capacităților iraționale. Un astfel de algoritm, bazat pe parcurgerea în lățime (BF) a grafului rezidual, este **Algoritmul Edmonds-Karp**.

Tema 3

Recursivitate

3.1 Introducere

Recurența este o tehnică de rezolvare a problemelor prin reducere repetată la subprobleme asemănătoare și de dimensiuni mai mici, până când dimensiunea obținută permite rezolvarea directă.

Recursivitatea este modalitatea naturală de implementare algoritmică a recurențelor și constă în definirea unor funcții având ca parametru dimensiunea problemei și care se autoapelează (*apel recursiv*) fie direct, fie indirect (prin intermediul altor funcții), până când dimensiunea permite rezolvarea directă.

Condițiile a căror verificare asigură posibilitatea rezolvării directe, deci oprirea procesului recursiv de calcul, sunt numite *condiții inițiale* pentru recurența considerată.

În mod uzual, o recurență este exprimată prin relații ce exprimă soluția problemei în funcție de soluțiile subproblemelor apelate, numite *relații de recurență*.

Implementările recursive ale recurențelor au de obicei ca dezavantaj faptul că autoapelarea funcției recursive impune nu numai memorarea valorilor deja calculate, ci și apelarea și calculul aceleiași valori de mai multe ori, ceea ce poate conduce la creșterea exponențială a timpului de execuție. În astfel de situații este de preferat ca rezolvarea problemei (determinarea soluțiilor ce verifică relațiile de recurență date) să se efectueze, atunci când este posibil, prin implementări nerecursive.

De exemplu, *metoda programării dinamice* presupune rezolvarea unei probleme prin descompunerea în subprobleme de același tip și de dimensiuni mai mici, ce pot avea (sub)subprobleme comune. Rezolvarea relațiilor de recurență aferente prin implementări recursive, în care soluția problemei

mari apelează soluțiile subproblemelor de dimensiuni mai mici (*strategia top-down*), necesită de obicei un timp de calcul exponențial, deoarece rezolvarea unei aceleiași subprobleme poate fi reluată de foarte multe ori, din cauza apelării ei repetate. Mult mai eficient este ca subproblemele să fie rezolvate în ordinea crescătoare a dimensiunii lor (*strategia bottom-up*), iar, pentru a evita repetarea procesului de calcul necesar rezolvării aceleiași subprobleme, soluțiile subproblemelor să fie memorate temporar, pentru utilizarea la posibilele reapelări.

Domeniul de aplicabilitate al tehnicilor recurente (recursive) este vast: combinatorica și teoria numerelor, metode de proiectare și de analiză a algoritmilor, algebra computațională, cercetările operaționale, inteligența artificială, etc.

Exemplul 3.1.1. Considerăm următorul algoritm recursiv, descris în pseudocod.

```
f(A, n):        // calculează f(A,n), A = (a_0, a_1, ..., a_n), n ∈ ℕ
if n = 0 then                                   // condiție inițială
    return a_0 + 2;
else
    if n = 1 then                               // condiție inițială
        return a_1 − a_0 + 1;
    else
        r ← f(A, n − 2);                            // apel recursiv
        a_{n−2} ← a_{n−2} + 1; a_{n−1} ← a_{n−1} + 1;
        b ← f(A, n − 2); c ← f(A, n − 1);           // apeluri recursive
        r ← r + b + c;
        for i = 1, n − 1 do
            a_i ← a_i + a_{i+1} − a_{i−1} + 1;
        b ← f(A, n − 2); c ← f(A, n − 1);           // apeluri recursive
        r ← r + b + c;
        for i = 0, n − 1 do
            a_i ← a_i + a_n;
        b ← f(A, n − 2); c ← f(A, n − 1);           // apeluri recursive
        r ← r + b + c;
        return r;
```

Pentru analiza complexității acestui algoritm, vom evalua doar numărul de adunări și scăderi, celelalte operații nedepășind ordinul de creștere al acestora. Observăm că acest număr depinde doar dimensiunea $n+1$ a vectorului A și nu și de valorile componentelor $a_0, a_1, \ldots, a_n$.

Fie x_n numărul de adunări și scăderi efectuate de algoritm pentru un vector (arbitrar) $A = (a_0, a_1, \ldots, a_n)$.

Din descrierea algoritmului avem $x_0 = 1$, $x_1 = 2$ și

$$x_n = x_{n-2} + 2 + x_{n-2} + x_{n-1} + 2 + 3(n-1) + x_{n-2} + x_{n-1} + 2 + n + x_{n-2} + x_{n-1} + 2,\ \forall n \geq 2.$$

Astfel x_n verifică relația de recurență

$$x_n = 3x_{n-1} + 4x_{n-2} + 4n + 5,\ \forall n \in \mathbb{N},\ n \geq 2,\ x_0 = 1,\ x_1 = 2. \quad (3.1.1)$$

Vom rezolva această recurență în una din secțiunile următoare și astfel vom putea evalua ordinul de complexitate al algoritmului de mai sus.

În continuare vom prezenta metode de rezolvare a unor relații de recurență clasice.

3.2 Recurența liniară de ordinul I cu termen liber constant

Această recurență are forma

$$x_n = ax_{n-1} + b,\ \forall n \in \mathbb{N}^*, \quad (3.2.1)$$

unde parametrii $a, b \in \mathbb{R}$ și primul termen $x_0 \in \mathbb{R}$ sunt numere date (cunoscute).

Exemplul 3.2.1. Considerăm relația de recurență liniară

$$x_n = 2x_{n-1} + 3,\ \forall n \in \mathbb{N}^*,\ x_0 = 1.$$

Avem

$$\begin{aligned} x_1 &= 2x_0 + 3 = 2 \cdot 1 + 3 = 5, \\ x_2 &= 2x_1 + 3 = 2 \cdot 5 + 3 = 13, \\ x_3 &= 2x_2 + 3 = 2 \cdot 13 + 3 = 29, \\ x_4 &= 2x_3 + 3 = 2 \cdot 29 + 3 = 61, \\ x_5 &= 2x_4 + 3 = 2 \cdot 61 + 3 = 125. \end{aligned}$$

Rezolvarea matematică

Avem următoarele două cazuri.

Cazul 1. Dacă $a = 1$, atunci recurența (3.2.1) devine

$$x_n = x_{n-1} + b,\ \forall n \in \mathbb{N}^*,$$

unde parametrul $b \in \mathbb{R}$ și primul termen $x_0 \in \mathbb{R}$ sunt numere date. Astfel șirul (x_n) este o *progresie aritmetică* de rație b, deci

$$x_n = x_0 + nb, \ \forall n \in \mathbb{N}$$

(relație ce poate fi demonstrată, de exemplu, prin inducție).

Cazul 2. Dacă $a \neq 1$, atunci notând

$$x_n = y_n + c, \ \forall n \in \mathbb{N}, \tag{3.2.2}$$

cu $c \in \mathbb{R}$, recurența (3.2.1) devine

$$y_n + c = a(y_{n-1} + c) + b, \ \forall n \in \mathbb{N}^*,$$

adică

$$y_n = ay_{n-1} + c(a-1) + b, \ \forall n \in \mathbb{N}^*.$$

Luând $c(a-1) + b = 0$, adică

$$c = -\frac{b}{a-1},$$

obținem

$$y_n = ay_{n-1}, \ \forall n \in \mathbb{N}^*.$$

Aceasta este o *progresie geometrică* de rație a, numită și *recurență liniară omogenă de ordinul I*. Rezultă că

$$y_n = a^n y_0, \ \forall n \in \mathbb{N}$$

(relație ce poate fi demonstrată, de exemplu, prin inducție), iar

$$y_0 = x_0 - c = x_0 + \frac{b}{a-1},$$

deci înlocuind în (3.2.2) obținem că soluția generală a recurenței (3.2.1) are forma

$$x_n = \left(x_0 + \frac{b}{a-1}\right) a^n - \frac{b}{a-1}, \ \forall n \in \mathbb{N}. \tag{3.2.3}$$

Exemplul 3.2.2. Reluăm relația de recurență liniară

$$x_n = 2x_{n-1} + 3, \ \forall n \in \mathbb{N}^*, \ x_0 = 1,$$

din Exemplul 3.2.1. Aplicând (3.2.3) obținem

$$x_n = \left(1 + \frac{3}{2-1}\right) 2^n - \frac{3}{2-1} = 4 \cdot 2^n - 3 = 2^{n+2} - 3, \ \forall n \in \mathbb{N}.$$

De exemplu, $x_5 = 2^7 - 3 = 125$.

3.3 Recurența liniară omogenă de ordinul al II-lea

Această recurență are forma

$$x_n = ax_{n-1} + bx_{n-2},\ \forall n \in \mathbb{N},\ n \geq 2, \tag{3.3.1}$$

unde parametrii $a, b \in \mathbb{R}$ și primii doi termeni $x_0, x_1 \in \mathbb{R}$ sunt numere date (cunoscute). Presupunem că

$$b \neq 0$$

(deoarece pentru $b = 0$ recurența devine de ordinul I).

Exemplul 3.3.1. Considerăm relația de recurență liniară

$$x_n = 2x_{n-1} + 15x_{n-2},\ \forall n \in \mathbb{N},\ n \geq 2,\ x_0 = 1,\ x_1 = 2.$$

Avem

$$\begin{aligned} x_2 &= 2x_1 + 15x_0 = 2 \cdot 2 + 15 \cdot 1 = 19, \\ x_3 &= 2x_2 + 15x_1 = 2 \cdot 19 + 15 \cdot 2 = 68, \\ x_4 &= 2x_3 + 15x_2 = 2 \cdot 68 + 15 \cdot 19 = 421, \\ x_5 &= 2x_4 + 15x_3 = 2 \cdot 421 + 15 \cdot 68 = 1862. \end{aligned}$$

Rezolvarea matematică

Pentru rezolvarea recurenței (3.3.1), îi asociem ecuația de gradul al doilea

$$r^2 = ar + b,\ \text{adică}\ \ r^2 - ar - b = 0, \tag{3.3.2}$$

numită *ecuația caracteristică* a acestei recurențe. Determinantul ecuației este

$$\Delta = a^2 + 4b.$$

Fie $r_1, r_2 \in \mathbb{C}$ rădăcinile acestei ecuații, adică

$$r_{1,2} = \begin{cases} \dfrac{a \pm \sqrt{\Delta}}{2}, & \text{dacă } \Delta \geq 0, \\ \dfrac{a \pm i\sqrt{-\Delta}}{2}, & \text{dacă } \Delta < 0 \end{cases}$$

(unde $i \in \mathbb{C} \setminus \mathbb{R}$, $i^2 = -1$).

Avem următoarele două cazuri.

Cazul 1. Dacă $r_1 \neq r_2$, atunci soluția generală a recurenței (3.3.1) are forma

$$x_n = c_1 r_1^n + c_2 r_2^n, \ \forall n \in \mathbb{N},$$

unde constantele $c_1, c_2 \in \mathbb{C}$ se determină în mod unic ca soluții ale sistemului de ecuații liniare

$$\begin{cases} c_1 + c_2 = x_0, \\ c_1 r_1 + c_2 r_2 = x_1 \end{cases}$$

(această formă poate fi demonstrată, de exemplu, prin inducție). Determinantul asociat sistemului este

$$\Delta_0 = \begin{vmatrix} 1 & 1 \\ r_1 & r_2 \end{vmatrix} = r_2 - r_1 \neq 0,$$

deci sistemul are o soluție unică, dată de *formulele lui Cramer*:

$$c_1 = \frac{\Delta_1}{\Delta_0} = \frac{\begin{vmatrix} x_0 & 1 \\ x_1 & r_2 \end{vmatrix}}{\Delta_0} = \frac{x_0 r_2 - x_1}{r_2 - r_1},$$

$$c_2 = \frac{\Delta_2}{\Delta_0} = \frac{\begin{vmatrix} 1 & x_0 \\ r_1 & x_1 \end{vmatrix}}{\Delta_0} = \frac{x_1 - x_0 r_1}{r_2 - r_1}.$$

Astfel soluția generală a recurenței (3.3.1) este, în acest caz,

$$x_n = \frac{(x_0 r_2 - x_1) r_1^n + (x_1 - x_0 r_1) r_2^n}{r_2 - r_1}, \ \forall n \in \mathbb{N}. \tag{3.3.3}$$

Exemplul 3.3.2. Reluăm relația de recurență liniară

$$x_n = 2x_{n-1} + 15x_{n-2}, \ \forall n \in \mathbb{N}, \ n \geq 2, \ x_0 = 1, \ x_1 = 2$$

din Exemplul 3.3.1. Ecuația caracteristică este

$$r^2 = 2r + 15, \text{ adică } r^2 - 2r - 15 = 0.$$

Avem $\Delta = 4 + 60 = 64$, $r_1 = \dfrac{2-8}{2} = -3$, $r_2 = \dfrac{2+8}{2} = 5$, deci $r_1 \neq r_2$. Aplicând (3.3.3) obținem

$$x_n = \frac{3 \cdot (-3)^n + 5 \cdot 5^n}{8} = \frac{5^{n+1} + (-1)^n \cdot 3^{n+1}}{8}, \ \forall n \in \mathbb{N}.$$

De exemplu, $x_5 = \dfrac{5^6 - 3^6}{8} = \dfrac{15625 - 729}{8} = \dfrac{14896}{8} = 1862$.

Cazul 2. Dacă $r_1 = r_2$, atunci soluția generală a recurenței (3.3.1) are forma

$$x_n = r_1^n(c_1 + c_2 n),\ \forall n \in \mathbb{N},$$

unde constantele $c_1, c_2 \in \mathbb{C}$ se determină în mod unic ca soluții ale sistemului de ecuații liniare

$$\begin{cases} c_1 = x_0, \\ r_1(c_1 + c_2) = x_1 \end{cases}$$

(această formă poate fi demonstrată, din nou, prin inducție). Evident, sistemul are o soluție unică, dată de:

$$\begin{aligned} c_1 &= x_0, \\ c_2 &= \frac{x_1 - x_0 r_1}{r_1}. \end{aligned}$$

Astfel soluția generală a recurenței (3.3.1) este, în acest caz,

$$x_n = \left[(x_1 - x_0 r_1)n + x_0 r_1\right] r_1^{n-1},\ \forall n \in \mathbb{N}. \tag{3.3.4}$$

Exemplul 3.3.3. Considerăm relația de recurență liniară

$$x_n = 4x_{n-1} - 4x_{n-2},\ \forall n \in \mathbb{N},\ n \geq 2,\ x_0 = 3,\ x_1 = 9.$$

Ecuația caracteristică este

$$r^2 = 4r - 4,\ \text{adică}\ \ r^2 - 4r + 4 = 0.$$

Avem $\Delta = 16 - 16 = 0$, $r_1 = r_2 = \dfrac{4}{2} = 2$. Aplicând (3.3.4) obținem

$$x_n = (3n + 6)2^{n-1},\ \forall n \in \mathbb{N}.$$

De exemplu, $x_5 = 21 \cdot 16 = 336$.

Observația 3.3.1. Analog se definesc și rezolvă relațiile de recurență liniară omogene de ordine mai mari sau egale cu trei.

3.4 Recurența liniară de ordinul al II-lea cu termen liber constant

Această recurență are forma

$$x_n = ax_{n-1} + bx_{n-2} + c,\ \forall n \in \mathbb{N},\ n \geq 2, \tag{3.4.1}$$

unde parametrii $a, b, c \in \mathbb{R}$ și primii doi termeni $x_0, x_1 \in \mathbb{R}$ sunt numere date (cunoscute). Presupunem că

$$b \neq 0$$

(deoarece pentru $b = 0$ recurența devine de ordinul I).

Exemplul 3.4.1. Considerăm relația de recurență liniară

$$x_n = 5x_{n-1} - 6x_{n-2} + 4,\ \forall n \in \mathbb{N},\ n \geq 2,\ x_0 = 2,\ x_1 = 3.$$

Avem

$$\begin{aligned} x_2 &= 5x_1 - 6x_0 + 4 = 5 \cdot 3 - 6 \cdot 2 + 4 = 7, \\ x_3 &= 5x_2 - 6x_1 + 4 = 5 \cdot 7 - 6 \cdot 3 + 4 = 21, \\ x_4 &= 5x_3 - 6x_2 + 4 = 5 \cdot 21 - 6 \cdot 7 + 4 = 67, \\ x_5 &= 5x_4 - 6x_3 + 4 = 5 \cdot 67 - 6 \cdot 21 + 4 = 213. \end{aligned}$$

Rezolvarea matematică

Avem următoarele două cazuri.

Cazul 1. Dacă $a + b \neq 1$, atunci notând

$$x_n = y_n + d,\ \forall n \in \mathbb{N}, \tag{3.4.2}$$

cu $d \in \mathbb{R}$, recurența (3.4.1) devine

$$y_n + d = a(y_{n-1} + d) + b(y_{n-2} + d) + c,\ \forall n \in \mathbb{N},\ n \geq 2,$$

adică

$$y_n = ay_{n-1} + by_{n-2} + d(a + b - 1) + c,\ \forall n \in \mathbb{N},\ n \geq 2.$$

Luând $d(a + b - 1) + c = 0$, adică

$$d = -\frac{c}{a + b - 1},$$

obținem

$$y_n = ay_{n-1} + by_{n-2},\ \forall n \in \mathbb{N},\ n \geq 2, \tag{3.4.3}$$

cu

$$y_0 = x_0 - d = x_0 + \frac{c}{a + b - 1},\ y_1 = x_1 - d = x_1 + \frac{c}{a + b - 1}. \tag{3.4.4}$$

Aceasta este o recurență liniară omogenă de ordinul al II-lea și poate fi rezolvată conform metodei descrise în secțiunea anterioară. Înlocuind în (3.4.2), obținem că soluția recurenței neomogene (3.4.1) în cazul $a + b \neq 1$ este

$$x_n = y_n - \frac{c}{a + b - 1},\ \forall n \in \mathbb{N}. \tag{3.4.5}$$

Exemplul 3.4.2. Reluăm relația de recurență liniară cu termen liber constant

$$x_n = 5x_{n-1} - 6x_{n-2} + 4,\ \forall n \in \mathbb{N},\ n \geq 2,\ x_0 = 2,\ x_1 = 3$$

din Exemplul 3.4.1. Avem $a + b = 5 - 6 \neq 1$. Conform relațiilor (3.4.5), (3.4.3) și (3.4.4), luând

$$y_n = x_n + \frac{c}{a+b-1} = x_n + \frac{4}{-2} = x_n - 2,\ \forall n \in \mathbb{N},$$

obținem că y_n verifică recurența liniară omogenă

$$y_n = 5y_{n-1} - 6y_{n-2},\ \forall n \in \mathbb{N},\ n \geq 2,\ y_0 = x_0 - 2 = 0,\ y_1 = x_1 - 2 = 1.$$

Ecuația caracteristică este

$$r^2 = 5r - 6,\ \text{adică}\ \ r^2 - 5r + 6 = 0.$$

Avem $\Delta = 25 - 24 = 1$, $r_1 = \dfrac{5-1}{2} = 2$, $r_2 = \dfrac{5+1}{2} = 3$, deci $r_1 \neq r_2$. Aplicând (3.3.3) obținem

$$y_n = \frac{(y_0 r_2 - y_1)r_1^n + (y_1 - y_0 r_1)r_2^n}{r_2 - r_1} = \frac{-2^n + 3^n}{1} = 3^n - 2^n,\ \forall n \in \mathbb{N}.$$

Astfel

$$x_n = y_n + 2 = 3^n - 2^n + 2,\ \forall n \in \mathbb{N}.$$

De exemplu, $x_5 = 3^5 - 2^5 + 2 = 243 - 32 + 2 = 213$.

Cazul 2. Dacă $a + b = 1$, atunci $a = 1 - b$ și recurența (3.4.1) devine

$$x_n = (1-b)x_{n-1} + bx_{n-2} + c,\ \forall n \in \mathbb{N},\ n \geq 2,$$

adică

$$x_n - x_{n-1} = -b(x_{n-1} - x_{n-2}) + c,\ \forall n \in \mathbb{N},\ n \geq 2.$$

Notând

$$y_{n-1} = x_n - x_{n-1},\ \forall n \in \mathbb{N}^*, \tag{3.4.6}$$

obținem

$$y_{n-1} = -by_{n-2} + c,\ \forall n \in \mathbb{N},\ n \geq 2,$$

adică

$$y_n = -by_{n-1} + c,\ \forall n \in \mathbb{N}^*, \tag{3.4.7}$$

cu

$$y_0 = x_1 - x_0. \tag{3.4.8}$$

Înlocuind în (3.4.6), obținem

$$\begin{aligned} x_1 - x_0 &= y_0, \\ x_2 - x_1 &= y_1, \\ x_3 - x_2 &= y_2, \\ &\dots \\ x_n - x_{n-1} &= y_{n-1}, \end{aligned}$$

deci prin adunare rezultă că

$$x_n - x_0 = y_0 + y_1 + y_2 + \ldots + y_{n-1}$$

și astfel avem

$$x_n = x_0 + \sum_{k=0}^{n-1} y_k, \ \forall n \in \mathbb{N}. \tag{3.4.9}$$

Relațiile (3.4.7) și (3.4.8) definesc o recurență liniară de ordinul I cu termen liber constant, care se rezolvă conform rezultatelor descrise în Secțiunea 3.2. Astfel avem următoarele două subcazuri.

i) Dacă $b = -1$ și deci $a = 2$, atunci relația (3.4.7) devine

$$y_n = y_{n-1} + c, \ \forall n \in \mathbb{N}^*,$$

adică șirul (y_n) este o progresie aritmetică de rație c. Deci

$$y_n = y_0 + nc = x_1 - x_0 + nc, \ \forall n \in \mathbb{N}.$$

Înlocuind în (3.4.9), obținem

$$x_n = x_0 + \sum_{k=0}^{n-1}(x_1 - x_0 + ck) = x_0 + n(x_1 - x_0) + c \cdot \frac{(n-1)n}{2},$$

deci soluția recurenței (3.4.1) în cazul $a = 2$, $b = -1$ este

$$x_n = \frac{c}{2} \cdot n^2 + \left(x_1 - x_0 - \frac{c}{2}\right) n + x_0, \ \forall n \in \mathbb{N}. \tag{3.4.10}$$

Exemplul 3.4.3. Considerăm relația de recurență liniară

$$x_n = 2x_{n-1} - x_{n-2} + 10, \ \forall n \in \mathbb{N}, \ n \geq 2, \ x_0 = 3, \ x_1 = 4.$$

Avem $a = 2$, $b = -1$, $c = 10$. Aplicând (3.4.10) obținem

$$x_n = 5n^2 - 4n + 3, \ \forall n \in \mathbb{N}.$$

De exemplu, $x_5 = 5 \cdot 25 - 4 \cdot 5 + 3 = 108$.

ii) Dacă $b \neq -1$ și $a = 1 - b$, deci $a \neq 2$, atunci conform (3.2.3) obținem că soluția recurenței (3.4.7) este

$$\begin{aligned} y_n &= \left(y_0 + \frac{c}{-b-1}\right)(-b)^n - \frac{c}{-b-1} \\ &= \left(x_1 - x_0 - \frac{c}{b+1}\right)(-b)^n + \frac{c}{b+1},\ \forall n \in \mathbb{N}. \end{aligned}$$

Înlocuind în (3.4.9), obținem

$$\begin{aligned} x_n &= x_0 + \sum_{k=0}^{n-1}\left[\left(x_1 - x_0 - \frac{c}{b+1}\right)(-b)^k + \frac{c}{b+1}\right] \\ &= x_0 + \frac{c}{b+1}\cdot n + \left(x_1 - x_0 - \frac{c}{b+1}\right)\sum_{k=0}^{n-1}(-b)^k \\ &= x_0 + \frac{c}{b+1}\cdot n + \left(x_1 - x_0 - \frac{c}{b+1}\right)\cdot\frac{(-b)^n - 1}{-b-1}, \end{aligned}$$

deci soluția recurenței (3.4.1) în cazul $a + b = 1$, $b \neq -1$ este

$$x_n = \left[\frac{c}{(b+1)^2} - \frac{x_1 - x_0}{b+1}\right][(-b)^n - 1] + \frac{c}{b+1}\cdot n + x_0,\ \forall n \in \mathbb{N}. \quad (3.4.11)$$

Exemplul 3.4.4. Considerăm relația de recurență liniară

$$x_n = 4x_{n-1} - 3x_{n-2} + 8,\ \forall n \in \mathbb{N},\ n \geq 2,\ x_0 = 1,\ x_1 = 5.$$

Avem $a = 4$, $b = -3$, $c = 8$, deci $a + b = 1$. Aplicând (3.4.11) obținem

$$x_n = \left(\frac{8}{4} - \frac{4}{-2}\right)(3^n - 1) + \frac{8}{-2}\cdot n + 1 = 4\cdot 3^n - 4n - 3,\ \forall n \in \mathbb{N}.$$

De exemplu, $x_5 = 4 \cdot 243 - 20 - 3 = 949$.

Observația 3.4.1. Analog se definesc și rezolvă relațiile de recurență liniară de ordine mai mari sau egale cu trei cu termen liber constant.

Tema 4

Metoda Backtracking

Tema 5

Metoda Divide et Impera

Tema 6

Metoda programării dinamice

Tema 7

Metoda Branch and Bound