PROIECTAREA ȘI IMPLEMENTAREA ALGORITMILOR

Conf. univ. dr. COSTEL BĂLCĂU

2020

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: teorie, algoritmi -cu implementare- și probleme)

Bibliografie

[1] A.V. Aho, J.E. Hopcroft, J.D. Ullman, *Data Structures and Algorithms*, Addison-Wesley, Massachusetts, 2009.

[2] Gh. Barbu, V. Păun, *Programarea în limbajul C/C++*, Editura Matrix Rom, București, 2011.

[3] Gh. Barbu, V. Păun, *Calculatoare personale și programare în C/C++*, Editura Didactică și Pedagogică, București, 2005.

[4] Gh. Barbu, I. Văduva, M. 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*, Ed. Polirom, Iași, 2005.

[8] E. Ciurea, L. Ciupală, *Algoritmi. Introducere în algoritmica fluxurilor în rețele*, Editura Matrix Rom, București, 2006.

[9] T.H. Cormen, *Algorithms Unlocked*, MIT Press, Cambridge, 2013.

[10] T.H. Cormen, C.E. Leiserson, R.L. Rivest, C. Stein, *Introduction to Algorithms*, MIT Press, Cambridge, 2009.

[11] C. Croitoru, *Tehnici de bază în optimizarea combinatorie*, Editura Universității "Al. I. Cuza", Iași, 1992.

[12] N. Dale, C. Weems, *Programming and problem solving with JAVA*, Jones & Bartlett Publishers, Sudbury, 2008.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

[27] D. Lucanu, M. Craus, *Proiectarea algoritmilor*, Ed. Polirom, Iași, 2008.

[28] D.A. Popescu, *Bazele programării - Java după C++*, `ebooks.infobits.ro`, 2019.

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

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

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

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

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

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

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

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

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

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

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

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

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

[42] ***, *Handbook of discrete and combinatorial mathematics*, edited by K.H. Rosen, J.G. Michaels, J.L. Gross, J.W. Grossman and D.R. Shier, CRC Press, Boca Raton, 2000.

[43] ***, *Revista MATINF*, editată de Departamentul de Matematică-Informatică, Universitatea din Pitești, Editura Universității din Pitești, 2018-2020, `http://matinf.upit.ro`.

Tema 1

Elemente de complexitatea algoritmilor

1.1 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 $\lim_{n\to\infty} \frac{f(n)}{g(n)} = L$. Atunci:*

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

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

Următorul rezultat este o consecință a propoziției anterioare.

Propoziția 1.1.2. *Fie $f : \mathbb{N} \to \mathbb{R}$ 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.

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 sî 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
$R \leftarrow G$; // $R =$ greutatea disponibilă pentru rucsac
$C \leftarrow 0$;
$i \leftarrow 1$;
while $R > 0$ **do** // rucsacul nu este plin
  if $g[i] \leq R$ **then**
    // obiectul curent încape în întregime, deci
    // se adaugă în rucsac
    $x[i] \leftarrow 1$;
    $C \leftarrow C + c[i]$;
    $R \leftarrow R - g[i]$; // actualizăm greutatea disponibilă
    $i \leftarrow i + 1$; // trecem la obiectul următor
  else
    // obiectul curent nu încape în întregime, deci
    // se adaugă exact acea parte din el care
    // umple rucsacul și încărcarea se încheie
    $x[i] \leftarrow \frac{R}{g[i]}$;
    $C \leftarrow C + x[i]c[i]$;
    $R \leftarrow 0$;
    for $j = \overline{i+1, n}$ **do** $x[j] \leftarrow 0$;
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 mazimă $G = 40$, 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)**:** // $a = (a_1, \ldots, a_n)$, $b = (b_1, \ldots, b_n)$
    // $c = (c_1, \ldots, c_n)$, $m =$ numărul de spectacole selectate
SORTARE(a, b, n); // se sortează spectacolele crescător
    // după timpul lor de începere a_i
$m \leftarrow 0$; // inițializări
for $i = \overline{1, n}$ **do** $c[i] \leftarrow 0$;
$t \leftarrow b[n] + 1$; // $t =$ timpul de începere al ultimului
    // spectacol selectat
for $i = \overline{n, 1, -1}$ **do** // parcurgem spectacolele în ordinea
    // descrescătoare a timpilor de începere
  if $b[i] < t$ **then**
    $c[i] \leftarrow 1$; // selectăm intervalul (spectacolul) curent
    $m \leftarrow m + 1$;
    $t \leftarrow a[i]$; // actualizăm t
AFISARE(m, c, n); // se afișează numărul și
    // submulțimea intervalelor (spectacolelor) selectate

Funcția de afișare este aceeași ca în Algoritmul 2.4.1.

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

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

- S_{14} (ultimul, în ordinea 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.

Tema 3

Metoda Backtracking

3.1 Descrierea metodei. Algoritmi generali

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

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

unde:

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

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

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

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

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

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

Observația 3.1.2. Metoda Backtracking își propune să determine:

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

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

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

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

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

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

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

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

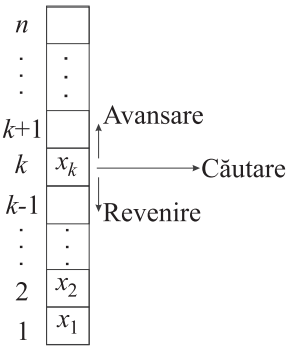
1. *aceste relații sunt necesare pentru existența unei soluții rezultat de forma*
$$(x_1, \ldots, x_{k-1}, x_k, x_{k+1}, \ldots, x_n),$$
adică neîndeplinirea acestor condiții implică faptul că oricum am alege $x_{k+1} \in S_{k+1}, \ldots, x_n \in S_n$ vectorul $x = (x_1, \ldots, x_n)$ nu poate fi o soluție rezultat (nu verifică conditiile interne);
2. *pentru $k = n$ condițiile de continuare coincid cu conditiile interne.*

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

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

Mecanismul metodei Backtracking

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



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

Mai precis, avem următorii pași:

1. Se începe cu soluția parțială dată de vectorul x vid.

2. Se ia primul element din mulțimea S_1 și se atribuie lui x_1.

3. Presupunând generate elementele $(x_1, x_2, \ldots, x_{k-1})$, $x_i \in S_i$, $i = \overline{1, k-1}$, se **avansează** la nivelul k și se caută primul element disponibil din S_k a cărui valoare să fie atribuită lui x_k.

Avem următoarele cazuri:

3.1. A fost găsit în S_k un element disponibil v. Atunci:
– îl atribuim lui x_k;
– se verifică dacă acesta împreună cu elementele deja generate $x_1, \ldots, x_{k-1}$ îndeplinește condițiile de continuare.
Avem următoarele subcazuri:

3.1.1. x_k verifică condițiile de continuare. Spunem că v este o **valoare validă** pentru componenta x_k.
Atunci:
– se extinde soluția parțială la $(x_1, \ldots, x_{k-1}, x_k)$;
– se verifică dacă $k = n$:

3.1.1.1. Dacă **da**, atunci s-a obținut o soluție rezultat. Acum, fie ne oprim, fie continuăm cu căutarea altei soluții rezultat, reluând algoritmul (cu pasul **3.**), considerând generate $(x_1, \ldots, x_{k-1})$ și căutând în continuare pentru x_k un element netestat din S_k.

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

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

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

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

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

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

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

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

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

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

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

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

adică

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

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

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

BACKTRACKING1 :

$k \leftarrow 1$;
$x[1] \leftarrow a[1] - r[1]$;// pregătim introducerea valorii inițiale a_1
// pentru x_1
while $k > 0$ **do**
  if $x[k] < b[k]$ **then**
    // mai există valori netestate pentru x_k
    $x[k] \leftarrow x[k] + r[k]$; // Căutare
    if $\big[(x[1], \ldots, x[k])$ *verifică condițiile de continuare*$\big]$ **then**
      if $k = n$ **then**
        // $(x_1, \ldots, x_n)$ este soluție rezultat
        PRELUCREAZĂ$(x[1], \ldots, x[n])$;
        (*) // STOP, dacă se dorește
        // o singură soluție rezultat,
        // sau $k \leftarrow k - 1$ (revenire forțată) dacă
        // nu mai există valori valide pentru x_n
      else
        $k \leftarrow k + 1$; // Avansare
        $x[k] \leftarrow a[k] - r[k]$; // pregătim introducerea
        // valorii inițiale a_k pentru x_k
  else
    // nu mai există valori netestate pentru x_k
    $k \leftarrow k - 1$; // Revenire

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

Următoarea schemă este echivalentă cu cea de mai sus.

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

BACKTRACKING2 :

$k \leftarrow 1$;

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

while $k > 0$ **do**

  if $k \leq n$ **then**

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

      // mai există valori netestate pentru x_k

      $x[k] \leftarrow x[k] + r[k]$; // Căutare

      if $\left[(x[1], \ldots, x[k])\ \textit{verifică condițiile de continuare}\right]$ **then**

        $k \leftarrow k + 1$; // Avansare

        $x[k] \leftarrow a[k] - r[k]$; // pregătim introducerea // valorii inițiale a_k pentru x_k

    else

      $k \leftarrow k - 1$; // Revenire

  else

    // $(x_1, \ldots, x_n)$ este soluție rezultat

    PRELUCREAZĂ$(x[1], \ldots, x[n])$;

    $(*)$ // STOP, dacă se dorește // o singură soluție rezultat

    $k \leftarrow k - 1$; // Revenire, dacă se doresc // toate soluțiile rezultat

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

Algoritmul 3.1.3 (**Schema Backtracking recursivă, varianta 1**).

BACKR1(k)**:** // se generează $(x[k], \ldots, x[n])$
$x[k] \leftarrow a[k] - r[k]$; // pregătim introducerea valorii inițiale
// $a[k]$ pentru $x[k]$
while $x[k] < b[k]$ **do**
  $x[k] \leftarrow x[k] + r[k]$; // Căutare
  if $\big[(x[1], \ldots, x[k])$ *verifică condițiile de continuare*$\big]$ **then**
    if $k = n$ **then**
      // $(x[1], \ldots, x[n])$ este soluție rezultat
      PRELUCREAZĂ$(x[1], \ldots, x[n])$;
    else
      BACKR1$(k+1)$; // Avansare, adică
      // se generează, RECURSIV, $(x[k+1], \ldots, x[n])$.
      // La încheierea apelului BACKR1$(k+1)$ se produce
      // revenirea la funcţia BACKR1(k)

Apelare: **BACKR1**(1).

Următoarea schemă este echivalentă cu cea de mai sus.

Algoritmul 3.1.4 (**Schema Backtracking recursivă, varianta 2**).

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

Apelare: **BACKR2**(1).

Algoritmul 3.1.5 (**Schema Backtracking recursivă, varianta 3**).

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

Apelare: **BACKR3**(1).

Următoarea schemă este echivalentă cu cea de mai sus.

Algoritmul 3.1.6 (**Schema Backtracking recursivă, varianta 4**).

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

Apelare: **BACKR4**(1).

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

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

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

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

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

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

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

3.2 Colorarea grafurilor

Problema colorării grafurilor este următoarea:

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

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

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

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

Modelarea problemei

Orice soluție a problemei se poate scrie sub forma

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

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

Avem

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

unde

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

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

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

Condițiile interne:

Orice două noduri adiacente sunt colorate diferit, adică

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

Condițiile de continuare:

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

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

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

Observația 3.2.1. Pentru reprezentarea grafului utilizăm matricea de adiacență

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

Graful fiind neorientat, matricea de adiacență este simetrică.

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

Algoritmul 3.2.1.

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

Funcția de verificare (validare) a condițiilor de continuare este

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

Aplicație: colorarea hărților

Problema colorării hărților este următoarea:

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

Modelarea problemei

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

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

O astfel de corespondență este evidențiată în Figura 3.2.1.

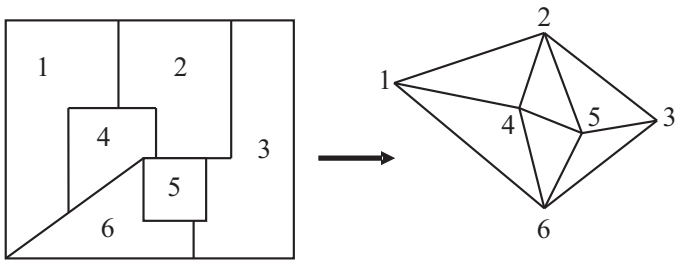


Figura 3.2.1:

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

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

Avem următorul rezultat celebru.

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

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

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