Facultatea: **Automatică şi Calculatoare**, Iaşi
Specializarea: **Calculatoare, Tehnologia informaţiei**

# PROIECTAREA ALGORITMILOR

## Lucrarea nr. 1
## Recapitularea unor noţiuni C/C++

**Breviar teoretic:**

***Structura unui program simplu in C/C++:***

```
//includere de fisiere header
#include <stdio.h>

//functia main = punctul de „intrare” in program
int main (void) {
      //declarare de variabila
      int x;

      //apel de functie
      printf (”Hello world!\n”);

      //atribuire valoare variabila
      x = 10;

      printf (”x = %d\n”, x);
      //finalizarea unui program
      return 0;
}
```

***Variabile:***

- mecanism ce simplifică stocarea informaţiilor în cadrul unui program;
- **variabilă** este un nume simbolic asociat cu o anumită **locaţie de memorie**;
- **numele unei variabile** *începe* fie cu un caracter ***alfabetic*** fie cu caracterul _ (***underscore***) şi *nu trebuie să conţină spaţii albe sau alte caracter în afara celor alfa-numerice*;
- **variabilă** poate fi accesată (citită) şi modificată (scrisă);
- variabilă se *caracterizează* prin: **tip**, **nume** şi **valoare**;
- **variabile locale** – variabile definite în interiorul unei funcţii;
- **variabile globale** – variabile definite în exteriorul oricărei funcţii dintr-un fişier sursă;
- variabilă este ***vizibilă*** din momentul în care a fost definită până la sfârşitul unei funcţii (dacă este variabilă locală) sau până la sfârşitul fişierului sursă (dacă este variabilă globală).

***Tipurile de date fundamentale:***

**INT**:

- tip de dată utilizat pentru stocarea **numerelor întregi**, **pozitive** şi **negative**, **reprezentate în cod complementar faţă de 2**;
- dimensiune: **4 octeţi**.

**CHAR**:

- tip de dată utilizat pentru stocarea **caracterelor**:
- dimensiune: **1 octet**.

**FLOAT**:

- tip de dată utilizat pentru stocarea **numerelor reale**, **pozitive** şi **negative**, **reprezentate în format cu virgulă mobilă**, **în simplă precizie**;
- dimensiune: **4 octeţi**.

**DOUBLE**:

- tip de dată utilizat pentru stocarea **numerelor reale**, **pozitive** şi **negative**, **reprezentate în format cu virgulă mobilă**, **în dublă precizie**;
- dimensiune: **8 octeţi**.

***Tipuri de date definite utilizator – STRUCTURI:***

*Declarare:*

```
struct nume_structura {
      //lista membri structura
      tip_data1 membru1, membru2;
      tip_data2 membru3;
      …
};
```

*Declarea unei variabile de tip structură:*

```
struct nume_structura nume_variabila;
```

***Funcţii:***

- **funcţie** reprezintă o secvenţă de cod destinată să implementeze o anumită *sarcină*;
- **funcţie** se declară prin ***antet***:

```
tip_retur nume_funcţie (listă de parametri formali);
```

- **funcţie** se ***defineşte*** prin implementarea corpului funcţiei (marcat prin *{}*).

***Pointeri:***

- un **pointer** reprezintă în C/C++ un tip de variabilă ce *indică* o adresă;
- declarare unui pointer:

```
T * nume_pointer; // T reprezintă un tip de dată primar
                  // sau definit de utilizator
```

- ***iniţializarea*** pointerilor:

cu ***valoare 0***: `T* p = 0`; sau `T* p = NULL;`

cu ***adresa unei variabile*** declarate anterior: `T* p = &(var);`
cu ***valoarea*** unui alt pointer declarat şi iniţializat anterior: `T* p = ptr;`
prin ***alocare dinamică de memorie***:

```
T* p = (T*) malloc (no_elems * sizeof(T));
```

s***au***

```
T* p = (T*) calloc (no_elems, sizeof (T));
```

**Sarcini de lucru:**

1. Scrieți un program C/C++ care să permită inserarea unei valori pe o poziţie specificată, într-o *listă liniară dublu înlănţuită*.

2. Se dă o matrice pătratică a[n][n] cu elemente formate din caractere din alfabetul latin. Să se rotească elementele matricei în sens trigonometric, folosind spaţiu suplimentar de memorie constant.

**Barem de notare:**

1. P1: 4p
2. P2 5p
3. Baza: 1p

# Proiectarea algoritmilor

## Complexitatea algoritmilor

## Lucrare de laborator nr. 2

# Cuprins

# Evaluarea algorimilor

- Evaluarea algorimilor din punctul de vedere al performanțelor obținute de aceștia în rezolvarea problemelor este o etapă esențială în procesul de decizie a utilizării acestora în aplicații.
- La evaluarea (estimarea) algoritmilor se pune în evidență consumul celor două resurse fundamentale: timpul de execuție și spațiul de memorare a datelor.
- În funcție de prioritățile alese, se aleg limite pentru resursele timp și spațiu.
- Algoritmul este considerat eligibil dacă consumul celor două resurse se încadrează în limitele stabilite.

# Timp și spatiu

- Fie $P$ o problemă și $A$ un algoritm pentru $P$.
- Fie $c_0 \vdash_A c_1 \cdots \vdash_A c_n$ un calcul finit al algoritmului $A$.
- Notăm cu $t_A(c_i)$ timpul necesar obținerii configurației $c_i$ din $c_{i-1}$, $1 \leq i \leq n$, și cu $s_A(c_i)$ spațiul de memorie ocupat în configurația $c_i$, $0 \leq i \leq n$.
- Fie $A$ un algoritm pentru problema $P$, $p \in P$ o instanță a problemei $P$ și $c_0 \vdash c_1 \vdash \cdots \vdash c_n$ calculul lui $A$ corespunzător instanței $p$.
  - *Timpul* necesar algoritmului $A$ pentru rezolvarea instanței $p$ este:

$$T_A(p) = \sum_{i=1}^{n} t_A(c_i)$$

  - *Spațiul* (de memorie) necesar algoritmului $A$ pentru rezolvarea instanței $p$ este:

$$S_A(p) = \max_{0 \leq i \leq n} s_A(c_i)$$

# Mărimea unei instanțe

- Asociem unei instanțe $p \in P$ o mărime $g(p)$, care este un număr natural, pe care o numim *mărimea* instanței $p$.
  - De exemplu, $g(p)$ poate fi suma lungimilor reprezentărilor corespunzând datelor din instanța $p$.
- Dacă reprezentările datelor din $p$ au aceeași lungime, atunci se poate considera $g(p)$ egală cu numărul datelor.
  - Dacă $p$ constă dintr-un tablou atunci se poate lua $g(p)$ ca fiind numărul de elemente ale tabloului.
  - Dacă $p$ constă dintr-un polinom se poate considera $g(p)$ ca fiind gradul polinomului (= numărul coeficienților minus 1).
  - Dacă $p$ este un graf se poate lua $g(p)$ ca fiind numărul de vârfuri, numărul de muchii sau numărul de vârfuri + numărul de muchii etc.

# Cazul favorabil și nefavorabil

- Fie $A$ un algoritm pentru problema $P$.
  - Spunem că $A$ rezolvă $P$ în *timpul* $T_A^{fav}(n)$ dacă:
$$T_A^{fav}(n) = \inf\{ T_A(p) \mid p \in P, g(p) = n\}$$
  - Spunem că $A$ rezolvă $P$ în *timpul* $T_A(n)$ dacă:
$$T_A(n) = \sup\{ T_A(p) \mid p \in P, g(p) = n\}$$
  - Spunem că $A$ rezolvă $P$ în spațiul $S_A^{fav}(n)$ dacă:
$$S_A^{fav}(n) = \inf\{ S_A(p) \mid p \in P, g(p) = n\}$$
  - Spunem că $A$ rezolvă $P$ în spațiul $S_A(n)$ dacă:
$$S_A(n) = \sup\{ S_A(p) \mid p \in P, g(p) = n\}$$
- Funcția $T_A^{fav}$ ($S_A^{fav}$) se numește *timpul de execuție al algoritmului (spațiul utilizat de algoritmul) A pentru cazul cel mai favorabil*
- Funcția $T_A$ ($S_A$) se numește *timpul de execuție al algoritmului (spațiu utilizat de algoritmul) A pentru cazul cel mai nefavorabil*.

# Exemplu de analiză - Problema căutării unui element într-o secvență de numere întregi

*Intrare:* $n, (a_0, \ldots, a_{n-1}), z$ numere întregi.

*Ieșire:* $poz = \begin{cases} \min\{i \mid a_i = z\} & \text{dacă } \{i \mid a_i = z\} \neq \emptyset, \\ -1 & \text{altfel.} \end{cases}$

- Presupunem că secvența $(a_0, \ldots, a_{n-1})$ este memorată în tabloul $(\texttt{a[i]} \mid 0 \leq i \leq n-1)$.

```
/* algoritmul A1 */
i ← 0
while (a[i] ≠ z) and (i < n-1) do
    i ← i+1
if (a[i] = z)
    then poz ← i
    else poz ← -1
```

# Problema căutării unui element într-o secvență de numere întregi - continuare

- Considerăm ca dimensiune a problemei numărul $n$ al elementelor din secvența în care se caută.
- Deoarece suntem în cazul când toate datele sunt memorate pe câte un cuvânt de memorie, vom presupune că toate operațiile necesită o unitate de timp.
- Cazul cel mai favorabil este obținut când $a_0 = z$ și se efectuează trei comparații și două atribuiri. Rezultă $T_{A_1}^{fav}(n) = 3+2 = 5$.
- Cazul cel mai nefavorabil se obține când $z \notin \{a_0, \ldots, a_{n-1}\}$ sau $z = a[n-1]$, în acest caz fiind executate $2n+1$ comparații și $1+(n-1)+1 = n+1$ atribuiri. Rezultă $T_{A_1}(n) = 3n+2$.
- Pentru simplitatea prezentării, nu au mai fost luate în considerare operațiile `and` și operațiile de adunare și scădere.
- Spațiul utilizat de algoritm, pentru ambele cazuri, este $n+7$ (tabloul `a`, constantele `0`, `1` și `-1`, variabilele `i`, `poz`, `n` și `z`).

```
/* algoritmul A1 */
i ← 0
while (a[i] ≠ z) and (i < n-1) do
   i ← i+1
if (a[i] = z)
   then poz ← i
   else poz ← -1
```

## Cazul mediu

- Timpii de execuție pentru cazul cel mai favorabil nu oferă informații relevante despre eficiența algoritmului.
- Mult mai semnificative sunt informațiile oferite de timpii de execuție în cazul cel mai nefavorabil: în toate celelalte cazuri algoritmul va avea performanțe mai bune sau cel puțin la fel de bune.
- Pentru evaluarea timpului de execuție nu este necesar întotdeauna să numărăm toate operațiile.
- În exemplul anterior, observăm că operațiile de atribuire (fără cea inițială) sunt precedate de comparații.
- Putem număra numai comparațiile, pentru că numărul acestora determină numărul atribuirilor.
- Putem să mergem chiar mai departe și să numărăm numai comparațiile între $z$ și componentele tabloului.
- Uneori, numărul instanțelor $p$ cu $g(p) = n$ pentru care $T_A(p) = T_A(n)$ sau $T_A(p)$ are o valoare foarte apropiată de $T_A(n)$ este foarte mic.
- Pentru aceste cazuri, este preferabil să calculăm comportarea în medie a algoritmului.

# Cazul mediu - continuare

- Pentru a putea calcula comportarea în medie este necesar să privim mărimea $T_A(p)$ ca fiind o variabilă aleatoare (o experiență = execuția algoritmului pentru o instanță $p$, valoarea experienței = durata execuției algoritmului pentru instanța $p$) și să precizăm legea de repartiție a acestei variabile aleatoare.
- *Comportarea în medie* se calculează ca fiind media acestei variabile aleatoare (considerăm numai cazul timpului de execuție):

$$T_A^{med}(n) = M(\{T_A(p) \mid p \in P \wedge g(p) = n\})$$

- Dacă mulțimea valorilor variabilei aleatoare $T_A(p) = \{x_1, \dots\}$ este finită sau numărabilă ($T_A(p) = \{x_1, \dots, x_i \dots\}$ și probabilitatea ca $T_A(p) = x_i$ este $p_i$, atunci media variabilei aleatoare $T_A$ (timpul mediu de execuție) este:

$$T_A^{med}(n) = \sum_i x_i \cdot p_i$$

# Exemplu de calcul al timpului pentru cazul mediu

- Considerăm problema căutării unui element într-o secvență de numere întregi, definită anterior.
- Mulțimea valorilor variabilei aleatoare $T_{A_1}(p)$ este $\{3i+2 \mid 1 \leq i \leq n\}$.
- Legea de repartiție:
  - Facem următoarele presupuneri: probabilitatea ca $z \in \{a_0, \ldots, a_{n-1}\}$ este $q$ și probabilitatea ca $z$ să apară prima dată pe poziția $i-1$ este $\frac{q}{n}$ (indicii $i$ candidează cu aceeași probabilitate pentru prima apariție a lui $z$). Rezultă că probabilitatea ca $z \notin \{a_0, \ldots, a_{n-1}\}$ este $1-q$.
  - Probabilitatea ca $T_{A_1}(p) = 3i+2$ ($poz = i-1$) este $\frac{q}{n}$, pentru $1 \leq i < n$, iar probabilitatea ca $T_{A_1}(p) = 3n+2$ este $p_n = \frac{q}{n} + (1-q)$ (probabilitatea ca $poz = n-1$ sau ca $z \notin \{a_0, \ldots, a_{n-1}\}$).

# Exemplu de calcul al timpului pentru cazul mediu - continuare

- Timpul mediu de execuție este:

$$\begin{aligned} T_{A_1}^{med}(n) &= \sum_{i=1}^{n} p_i x_i = \sum_{i=1}^{n-1} \frac{q}{n} \cdot (3i+2) + (\frac{q}{n} + (1-q)) \cdot (3n+2) \\ &= \frac{3q}{n} \cdot \sum_{i=1}^{n} i + \frac{q}{n} \sum_{i=1}^{n} 2 + (1-q) \cdot (3n+2) \\ &= \frac{3q}{n} \cdot \frac{n(n+1)}{2} + 2q + (1-q) \cdot (3n+2) \\ &= \frac{3q \cdot (n+1)}{2} + 2q + (1-q) \cdot (3n+2) \\ &= 3n - \frac{3nq}{2} + \frac{3q}{2} + 2 \end{aligned}$$

- Pentru $q = 1$ ($z$ apare totdeauna în secvență) avem $T_{A_1}^{med}(n) = \frac{3n}{2} + \frac{7}{2}$ și pentru $q = \frac{1}{2}$ avem $T_{A_1}^{med}(n) = \frac{9n}{4} + \frac{11}{4}$.

# Calcul asimptotic

- În practică, atât $T_A(n)$, cât și $T_A^{med}(n)$ sunt dificil de evaluat. Din acest motiv se caută, de multe ori, margini superioare și inferioare pentru aceste mărimi.
- Următoarele clase de funcții sunt utilizate cu succes în stabilirea acestor margini:

$$O(f(n)) = \{g(n) \mid (\exists c > 0, n_0 \geq 0)(\forall n \geq n_0)|g(n)| \leq c \cdot |f(n)|\}$$
$$\Omega(f(n)) = \{g(n) \mid (\exists c > 0, n_0 \geq 0)(\forall n \geq n_0)|g(n)| \geq c \cdot |f(n)|\}$$
$$\Theta(f(n)) = \{g(n) \mid (\exists c_1, c_2 > 0, n_0 \geq 0)(\forall n \geq n_0)c_1 \cdot |f(n)| \leq |g(n)| \leq c_2 \cdot |f(n)|\}$$

- Cu notațiile $O$, $\Omega$ și $\Theta$ se pot forma expresii și ecuații. Considerăm numai cazul $O$, celelalte tratându-se similar.
- Expresiile construite cu $O$ pot fi de forma:

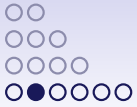
$$O(f_1(n)) \text{ op } O(f_2(n))$$

unde „op" poate fi $+, -, *$ etc. și notează mulțimile:
$\{g(n) \mid (\exists g_1(n), g_2(n), c > 0, n_0 \geq 0)$

$$[(\forall n)g(n) = g_1(n) \text{ op } g_2(n)] \wedge [(\forall n \geq n_0)g_1(n) \leq cf_1(n) \wedge g_2(n) \leq cf_2(n)]\}$$

  - De exemplu:

$$O(n) + O(n^2) = \{g(n) = g_1(n) + g_2(n) \mid (\forall n \geq n_0)g_1(n) \leq cn \wedge g_2(n) \leq cn^2\}$$



# Calcul asimptotic - continuare

- Utilizând regulile de asociere și prioritate, se obțin expresii de orice lungime:

$$O(f_1(n)) \text{ op}_1 \ O(f_2(n)) \text{ op}_2 \ \cdots$$

- Orice funcție $f(n)$ poate fi gândită ca o notație pentru mulțimea cu un singur element $f(n)$ și deci putem alcătui expresii de forma:

$$f_1(n) + O(f_2(n))$$

ca desemnând mulțimea:

$$\{f_1(n) + g(n) \mid g(n) \in O(f_2(n))\} =$$

$$\{f_1(n) + g(n) \mid (\exists c > 0, n_0 > 1)(\forall n \geq n_0) g(n) \leq c \cdot f_2(n)\}$$

- Peste expresii considerăm formule de forma:

$$expr1 = expr2$$

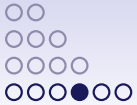
cu semnificația că mulțimea desemnată de $expr1$ este inclusă în mulțimea desemnată de $expr2$.

# Calcul asimptotic - Exemplu de formule

$$n \log n + O(n^2) = O(n^2)$$

Justificare:
$(\exists c_1 > 0, n_1 > 1)(\forall n \geq n_1) n \log n \leq c_1 n^2$, $g_1(n) \in O(n^2)$ implică
$(\exists c_2 > 0, n_2 > 1)(\forall n \geq n_2) g_1(n) \leq c_2 n^2$ c si de aici
$(\forall n \geq n_0) g(n) = n \log n + g_1(n) \leq n \log n + c_2 n^2 \leq (c_1 + c_2) n^2$, unde
$n_0 = \max\{n_1, n_2\}$.

- De remarcat nesimetria ecuațiilor: părțile stânga și cea din dreapta joacă roluri distincte.
- Ca un caz particular, notația $g(n) = O(f(n))$ semnifică, de fapt, $g(n) \in O(f(n))$.



# Calculul timpului asimptotic de execuție pentru cazul cel mai nefavorabil

- Un algoritm poate avea o descriere complexă și deci evaluarea sa poate pune unele probleme.
- Deoarece orice algoritm este descris de un program, în continuare considerăm $A$ o secvență de program.
- Regulile prin care se calculează timpul de execuție sunt date în funcție de structura lui $A$:
  - $A$ este o instrucțiune de atribuire. Timpul de execuție a lui $A$ este egal cu timpul evaluării expresiei din partea dreaptă.
  - $A$ este forma $A_1$ $A_2$ Timpul de execuție al lui $A$ este egal cu suma timpilor de execuție ai algoritmilor $A_1$ și $A_2$.
  - $A$ este de forma `if` $e$ `then` $A_1$ `else` $A_2$. Timpul de execuție al lui $A$ este egal cu maximul dintre timpii de execuție ai algoritmilor $A_1$ și $A_2$ la care se adună timpul necesar evaluării expresiei $e$.
  - $A$ este de forma `while` $e$ `do` $A_1$. Se determină cazul în care se execută numărul maxim de iterații ale buclei `while` și se face suma timpilor calculați pentru fiecare iterație. Dacă nu este posibilă determinarea timpilor pentru fiecare iterație, atunci timpul de execuție al lui $A$ este egal cu produsul dintre timpul maxim de execuție al algoritmului $A_1$ și numărul maxim de execuții ale buclei $A_1$.

# Timp polinomial

## Theorem (1)

*Dacă $g$ este o funcție polinomială de grad $k$, atunci $g = O(n^k)$.*

## Demonstrație.

Presupunem $g(n) = a_k \cdot n^k + a_{k-1} \cdot n^{k-1} + \cdots + a_1 \cdot n + a_0$.
Efectuând majorări în membrul drept, obținem:
$g(n) \leq |a_k| \cdot n^k + |a_{k-1}| \cdot n^{k-1} + \cdots + |a_1| \cdot n + |a_0| < n^k \cdot \underbrace{(|a_k| + |a_{k-1}| + |a_0|)}_{c} < n^k \cdot c$

pentru $\forall n > 1 \Rightarrow g(n) < c \cdot n^k$, cu $n_0 = 1$.
Deci $g = O(n^k)$. □

## Clasificarea algoritmilor

- Următoarele incluziuni sunt valabile în cazul notației $O$:

$$O(1) \subset O(\log n) \subset O(\log^k n) \subset O(n) \subset O(n^2) \subset \cdots \subset O(n^{k+1}) \subset O(2^n)$$

- Pentru clasificarea algoritmilor cea mai utilizată totație este $O$.
- Cele mai cunoscute clase sunt:

$\{A \mid T_A(n) = O(1)\}$ = clasa algoritmilor constanți;
$\{A \mid T_A(n) = O(\log n)\}$ = clasa algoritmilor logaritmici;
$\{A \mid T_A(n) = O(log^k n)\}$ = clasa algoritmilor polilogaritmici;
$\{A \mid T_A(n) = O(n)\}$ = clasa algoritmilor liniari;
$\{A \mid T_A(n) = O(n^2)\}$ = clasa algoritmilor pătratici;
$\{A \mid T_A(n) = O(n^k)\}$ = clasa algoritmilor polinomiali;
$\{A \mid T_A(n) = O(2^n)\}$ = clasa algoritmilor exponențiali.

- Cu notațiile de mai sus, doi algoritmi, care rezolvă aceeași problemă, pot fi comparați numai dacă au timpii de execuție în clase de funcții (corespunzătoare notațiilor $O, \Omega$ și $\Theta$) diferite. De exemplu, un algoritm $A$ cu $T_A(n) = O(n)$ este mai eficient decât un algoritm $A'$ cu $T_{A'}(n) = O(n^2)$.
- Dacă cei doi algoritmi au timpii de execuție în aceeași clasă, atunci compararea lor devine mai dificilă pentru că trebuie determinate și constantele cu care se înmulțesc reprezentanții clasei.

# Sarcini de lucru și barem de notare

**Sarcini de lucru:**

Pentru crearea unei liste circulare simplu înlănțuite studentul Ionescu reține pointerul către ultimul element al listei, inserând elementul nou introdus după ultimul, actualizând apoi pointerul către ultimul element. Student Popescu reține pointerul către primul element, pentru inserția unui element în listă parcurgând lista până la găsirea ultimului, realizând inserția după acesta.

a) Pornind de la lista vidă, inserând 10000 de elemente și considerând ca pas elementar comparația a 2 pointeri, câte operații efectuează în plus studentul Popescu față de studentul Ionescu?

b) Ajutați-l pe studentul Popescu scriind pentru poblema dată o funcție C/C++ de complexitate $O(1)$, care primește ca parametru pointerul către primul element al listei circulare simplu înlănțuite și valoarea de inserat la sfârștul listei circulare!

**Observații:**

- Se vor număra operațiile elementare și se va măsura timpul experimentului (durata de execuție a programului fără a considera operațiile de citire a datelor de intrare).
- În cazul în care optați pentru limbajul C++, puteți găsi informații despre afișarea valorilor reale cu o anumită precizie urmărind link-ul următor: http://www.cplusplus.com/reference/iostream/ios_base/precision/

**Barem de notare:**

1. Problema a): 4p
2. Problema b): 5p
3. Baza: 1p

# Determinarea timpului de rulare

Pentru determinarea duratei necesare efectuării experimentului se poate adapta codul de mai jos.

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

int main( void )
{
      clock_t start, finish;
      long loop;
      double result,r2, elapsed_time;

      printf( "Inmultirea a doua numere reale de un miliard de ori...\n" );
      result = 3.1415926535897;

      start = clock(); //inceputul experimentului
      for( loop = 0; loop < 1000000000; loop++ ) {
            r2=result*16.0;
      }
      finish = clock(); //sfarsitul operatiilor cronometrate

      elapsed_time = (double)(finish - start) / CLOCKS_PER_SEC;
      printf("\nProgramul necesita %6.2f secunde.\n",elapsed_time );

      return 0;
}
```

# Proiectarea algoritmilor

## Căutarea în secvențe sortate și în arbori binari de căutare

## Lucrare de laborator nr. 3

# Cuprins

# Căutarea binară

**Problema căutării binare**

*Intrare:* $n, (a_0, \ldots, a_{n-1}), z$ numere întregi;
secvența $(a_0, \ldots, a_{n-1})$ este sortată crescător,
adică $a_i \leq a_{i+1}, i \in \{0, \ldots, n-2\}$.

*Ieșire:* $$poz = \begin{cases} k \in \{i \mid a_i = z\} & \text{dacă } \{i \mid a_i = z\} \neq \emptyset, \\ -1 & \text{altfel.} \end{cases}$$

- Esența căutării binare constă în compararea elementului căutat cu elementul din mijlocul zonei de căutare și în cazul în care elementul căutat nu este egal cu acesta, se restrânge căutarea la subzona din stânga sau din dreapta, în funcție de rezultatul comparării.
- Dacă elementul căutat este mai mic decât cel din mijlocul zonei de căutare, se alege subzona din stânga, altfel subzona din dreapta. Inițial, zona de căutare este tabloul a.
- Convenim să notăm cu $\texttt{i}_{stg}$ indicele elementului din stânga zonei de căutare în tablou, $\texttt{i}_{dr}$ indicele elementului din dreapta zonei de căutare în tablou.

# Căutarea binară - pseudocod

```
function cautareBinara(a,n,z)
    i_stg ← 0
    i_dr ← n-1
    while (i_stg ≤ i_dr) do
        i_med ← ⌊i_stg+i_dr⌋/2
        if (a[i_med]=z)
            then return i_med
            else if (a[i_med]>z)
                    then i_dr ← i_med-1 /* se cauta in stanga */
                    else i_stg ← i_med+1 /* se cauta in dreapta */
    return -1
end
```

# Analiza algoritmului de căutare binară

- Dimensiunea problemei căutării binare este dată de dimensiunea $n$ a secvenței în care se face căutarea. Și de această dată presupunem că toate operațiile necesită o unitate de timp.
- Calculul timpului de execuție al algoritmului constă în determinarea numărului de execuții ale blocului de instrucțiuni asociat cu instrucțiunea `while`. Se observă că, după fiecare iterație a buclei `while`, dimensiunea zonei de căutare se înjumătățește.
- Cazul cel mai favorabil este obținut când $a\lfloor\frac{n-1}{2}\rfloor = z$ și se efectuează două comparații și trei atribuiri. Rezultă $T_{A_4}^{fav}(n) = 2+3 = 5$.
- Cazul cel mai nefavorabil este în situația în care vectorul $a$ nu conține valoarea căutată. Pentru simplitate, se consideră $n = 2^k$, unde $k$ este numărul de înjumătățiri. Rezultă $k = \log_2 n$ și printr-o majorare, $T_{A_4}(n) \leq c\log_2 n + 1$, unde $c$ este o constantă, $c \geq 1$.
- Spațiul necesar execuției algoritmului $A_4$ este $n+7$ (tabloul `a`, constantele `0` și `-1`, variabilele $\texttt{i}_{stg}$, $\texttt{i}_{dr}$, $\texttt{i}_{med}$, `n` și `z`).

# Căutare în arbori binari de cătare oarecare

- Un *arbore binar de căutare* este un arbore binar cu proprietățile:
  - informațiile din noduri sunt elemente dintr-o mulțime total ordonată;
  - pentru fiecare nod $v$, valorile memorate în subarborele stâng sunt mai mici decât valoarea memorată în $v$, iar valorile memorate în subarborele drept sunt mai mari decât valoarea memorată în $v$.

```
function cautArboreBinar(t,a)
    p ← t
    while ((p ≠ NULL) and (a ≠ p->elt)) do
        if (a < p->elt)
            then p ← p->stg
            else p ← p->drp
    return p
end
```

- Funcția poz ia valoarea *NULL*, dacă $a \notin S$ și adresa nodului care conține pe $a$ în caz contrar.
- Operațiile de inserare și de ștergere trebuie să păstreze invariantă următoarea proprietate:
  - valorile din lista inordine a nodurilor arborelui trebuie să fie în ordine crescătoare.

# Sarcini de lucru și barem de notare

**Sarcini de lucru:**

1. Scrieți o funcție C/C++ care implementează algoritmul de căutare binară.
2. Scrieți o funcție C/C++ care implementează algoritmul de căutare într-un un arbore binar de cătare.
3. Contorizați numărul de comparații. Comentați rezultatele obținute.

**Barem de notare:**

1. Funcția C/C++ care implementează algoritmul de căutare binară: 2p
2. Funcția C/C++ care implementează algoritmul de căutare într-un arbore binar de cătare: 3p
3. Contorizarea numărului de comparații: 4p
4. Baza: 1p

# Bibliografie

 Lucanu, D. și Craus, M., *Proiectarea algoritmilor*, Editura Polirom, 2008.

# Proiectarea algoritmilor

## Căutarea în B-arbori

## Lucrare de laborator nr. 4

# Cuprins

# B-arborii și indexarea multistratificată

- B-arborii sunt frecvent utilizați la indexarea unei colecții de date.
- Un index ordonat este un fișier secvențial.
- Pe măsură ce dimensiunea indexului crește, cresc și dificultățile de administrare.
- Soluția este construirea unui index cu mai multe niveluri, iar instrumentele sunt B-arborii.
- Algoritmii de căutare într-un index nestratificat nu pot depăși performanța $O(\log_2 n)$ intrări/ieșiri.
- Indexarea multistratificată are ca rezultat algoritmi de căutare de ordinul $O(\log_d n)$ intrări/ieșiri, unde $d$ este dimensiunea elementului indexului.

# Index multistratificat

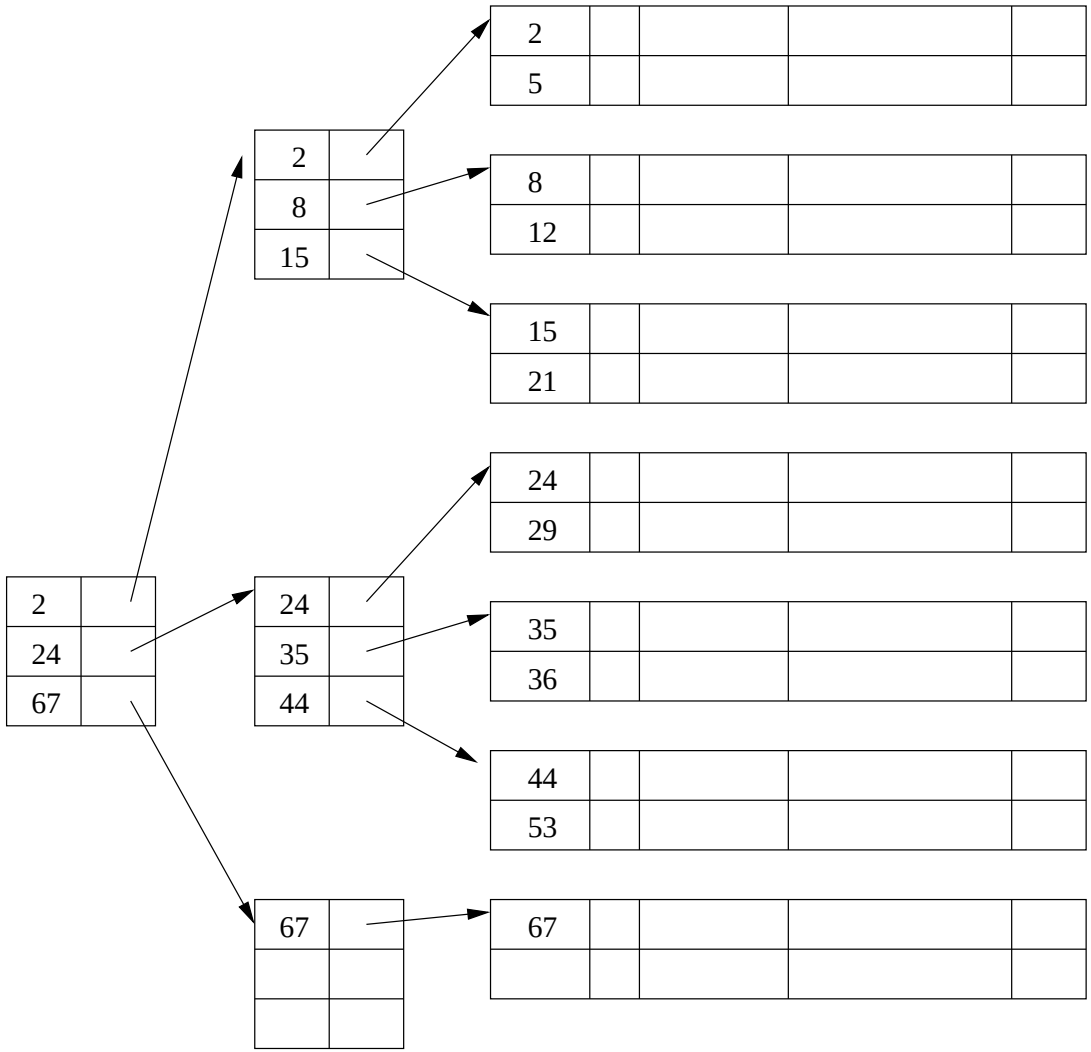



Figura 1 : Index multistratificat organizat pe 3 niveluri

# B-arbori - definiție

- Un B-arbore este un arbore cu rădăcină, în care:
  1. fiecare nod intern are un număr variabil de chei și fii;
  2. cheile dintr-un nod intern sunt memorate în ordine crescătoare;
  3. fiecare cheie dintr-un nod intern are asociat un fiu care este rădăcina unui subarbore ce conține toate nodurile cu chei mai mici sau egale cu cheia respectivă dar mai mari decât cheia precedentă;
  4. fiecare nod intern are un fiu extrem-dreapta, care este rădăcina unui subarbore ce conține toate nodurile cu chei mai mari decât oricare cheie din nod;
  5. fiecare nod intern are cel puțin un număr de $f-1$ chei ($f$ fii), $f=$ factorul de minimizare;
  6. doar rădăcina poate avea mai puțin de $f-1$ chei ($f$ fii);
  7. fiecare nod intern are cel mult $2f-1$ chei ($2f$ fii);
  8. lungimea oricărui drum de la rădăcină la o frunză este aceeași.
- Dacă fiecare nod necesită accesarea discului, atunci B-arborii vor necesita un număr minim de astfel de accesări.
- Factorul de minimizare va fi ales astfel încât dimensiunea unui nod să corespundă unui multiplu de blocuri ale dispozitivului de memorare. Această alegere optimizează accesarea discului.
- Înălțimea $h$ unui B-arbore cu $n$ noduri și $f>1$ satisface relația $h \leq \log_f \frac{n+1}{2}$.

# Implementare B-arbori

- Un nod $v$ al unui B-arbore poate fi implementat cu o structură statică formată din câmpurile `tipNod, nrChei, cheie[nrChei], data[nrChei]` și `fiu[nrChei+1]`.
  - Câmpul `tipNod` conține o valoare $tipNod \in \{frunza, interior\}$.
  - Câmpul `nrChei` conține numărul $t$ al cheilor din nodul $v$.
  - Tabloul `cheie[nrChei]` conține valorile cheilor memorate în nod$(k_0, k_1, \ldots, k_{t-1})$.
  - Tabloul `data[nrChei]` conține pointerii la structurile care conțin datele asociate nodului $v$ $((q_0, q_1, \ldots, q_{t-1})$.
  - Tabloul `fiu[nrChei+1]` conține pointerii la structurile care implementează fiii nodului $v$ $((p_0, p_1, \ldots, p_t)$.

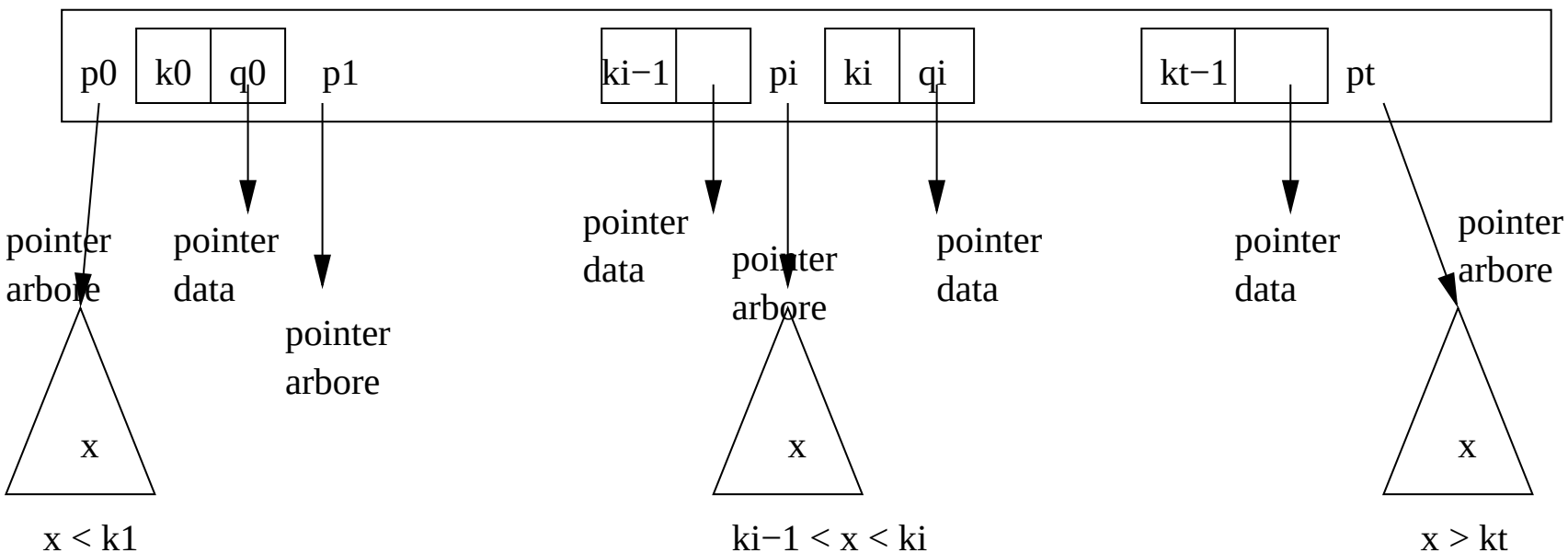



Figura 2 : Structura unui nod al unui B-arbore

# Creare B-arbori

- Operația `creeazaBarbore` creează un B-arbore vid cu rădăcina $t$ fără fii.
- Timpul de execuție este O(1).

```
creeazaBarbore(t)
    new t
    t->tipNod ← frunza
    t->nrChei ← 0
    scrieMemmorieExterna(t)
end
```

# Spargere nod B-arbore - descriere

- Dacă un nod $v$ este încărcat la maxim ($2f-1$ chei), pentru a insera o cheie nouă este necesară spargerea acestuia.
- Prin spargere, cheia mediană a nodului $v$ este mutată în părintele $u$ al acestuia ($v$ este al $i$-lea fiu).
- Este creat un nou nod $w$ și toate cheile din $v$ situate la dreapta cheii mediane sunt mutate în $w$.
- Cheile din $v$ situate la stânga cheii mediane rămân în $v$.
- Nodul nou $w$ devine fiu imediat la dreapta cheii mediane care a fost mutată în părintele $u$, iar $v$ devine fiu imediat la stânga cheii mediane care a fost mutată în părintele $u$.
- Spargerea transformă nodul cu $2f-1$ chei în două noduri cu $f-1$ chei (o cheie este mutată în părinte).
- Timpul de execuție este $O(t)$ unde $t =$ constant.

# Spargere nod B-arbore - pseudocod

```
procedure spargeNod(u, i, v)
    new nod w
    w->tipNod ← v->tipNod
    w->nrChei ← f-1
    for j ← 0 to f-2 do w->cheie[j] ← v->cheie[j+f]
    if (v->tipNod = interior)
        then for j ← 0 to f-1 do w->fiu[j] ← v->fiu[j+f]
    v->nrChei ← f-1
    for j ← u->nrChei downto i+1 do u->fiu[j+1] ← u->fiu[j]
    u->fiu[i+1] ← w
    for j ← u->nrChei-1 downto i do u->cheie[j+1] ← u->cheie[j]
    u->cheie[i] ← v->cheie[f-1]
    u->nrChei ← u->nrChei + 1
    scrieMemmorieExterna(u)
    scrieMemmorieExterna(v)
    scrieMemmorieExterna(w)
end
```

# Inserare în nod incomplet - pseudocod

```
procedure insereazainNodIncomplet(v, k)
    i ← v->nrChei-1
    if (v->tipNod = frunza)
        then while (i >= 0 and k < v->cheie[i]) do
                v->cheie[i+1] ← v->cheie[i]
                i ← i-1
             v->cheie[i+1] ← k
             v-nrChei ← v->nrChei + 1
             scrieMemmorieExterna(v)
        else while (i >= 0 and k < v->cheie[i]) do i ← i-1
             i ← i+1
             citesteMemmorieExterna(v->fiu[i])
             if (v->fiu[i]->nrChei = 2f-1)
                then spargeNod(v, i, v->fiu[i])
                     if (k > v->cheie[i]) then i ← i+1
             insereazainNodIncomplet(v->fiu[i], k)
end
```

# Inserare în B-arbore - descriere

- Pentru a efectua o inserție într-un B-arbore trebuie întâi găsit nodul în care urmează a se face inserția.
- Pentru a găsi nodul în care urmează a se face inserția, se aplică un algoritm similar cu `cautareBarbore`.
- Apoi cheia urmează a fi inserată.
  1. Dacă nodul determinat anterior conține mai puțin de $2f-1$ chei, se face inserarea în nodul respectiv.
  2. Dacă acest nod conține $2f-1$ chei, urmează spargerea acestuia.
  3. Procesul de spargere poate continua până în rădăcină.
- Pentru a evita două citiri de pe disc ale aceluiași nod, algoritmul sparge fiecare nod plin ($2f-1$ chei) întâlnit la parcurgea *top-down* în procesul de căutare a nodului în care urmează a se face inserarea.
- Timpul de spargere a unui nod este $O(f)$. Rezultă pentru inserție complexitatea timp $O(f \log n)$.

# Inserare în B-arbore - pseudocod

```
procedure insereazaBarbore(t, k)
    v ← t
    if (v->nrChei = 2f-1)
        then new nod u
             t ← u
             u->tipNod ← interior
             u->nrChei ← 0
             u->fiu[0]= v
             spargeNod(u, 0, v)
             insereazainNodIncomplet(u, k)
        else insereazainNodIncomplet(v, k)
end
```

# Căutare în B-arbori

- Căutarea într-un B-arbore este asemănătoare cu căutarea într-un arbore binar.
- Deoarece timpul de căutare depinde de adâncimea arborelui, `căutareBarbore` are timpul de execuție $O(\log_f n)$.

```
cautareBarbore(v, k)
    i ← 0
    while (i < v->nrChei and k > v->cheie[i]) do i ← i+1
    if (i < v->nrChei and k = v->cheie[i]) then return (v, i)
    if (v->tipNod = frunza)
        then return NULL
        else citesteMemmorieExterna(v->fiu[i])
             return cautareBarbore(v->fiu[i], k)
end
```

# Link-uri utile

http://www.bluerwhite.org/btree/
http://cis.stvincent.edu/html/tutorials/swd/btree/btree.html

# Sarcini de lucru și barem de notare

**Sarcini de lucru:** Implementați următoarele operații pentru structuri de tip B-Arbore:

1. Scrieți o funcție C/C++ care implementează operația de inserare a unui element într-un B-Arbore.
2. Scrieți o funcție C/C++ care implementează operația de parcurgere a unui B-Arbore.
3. Scrieți o funcție C/C++ care implementează operația de căutare a unui element într-un B-Arbore.

**Barem de notare:**

1. Funcția C/C++ care implementează operația de inserare a unui element într-un B-Arbore: 5p
2. Funcția C/C++ care implementează operația de parcurgere a unui B-Arbore: 2p
3. Funcția C/C++ care implementează de căutare a unui element într-un B-Arbore: 2p
4. Baza: 1p

# Bibliografie

 Lucanu, D. și Craus, M., *Proiectarea algoritmilor*, Editura Polirom, 2008.

# Proiectarea algoritmilor

**Sortare - bubleSort, naivSort, InsertionSort**

**Lucrare de laborator nr. 5**

# Cuprins

# Sortarea prin interschimbarea elementelor vecine (*bubble-sort*)

- Notăm cu $SORT(\mathtt{a})$ predicatul care ia valoarea *true* dacă și numai dacă tabloul a este sortat.
- Metoda *bubble-sort* se bazează pe următoarea definiție a predicatului $SORT(\mathtt{a})$:

$$SORT(\mathtt{a}) \iff (\forall i)(0 \leq i < n-1) \Rightarrow a[i] \leq a[i+1]$$

- O pereche $(i,j)$, cu $i<j$, formează o *inversiune (inversare)*, dacă $a[i] > a[j]$.
- Pe baza definiției de mai sus vom spune că tabloul a este sortat dacă și numai dacă nu există nici o inversiune $(i, i+1)$.
- Metoda *bubble-sort* propune parcurgerea iterativă a tabloului a și, la fiecare parcurgere, ori de câte ori se întâlnește o inversiune $(i, i+1)$ se procedează la interschimbarea $\mathtt{a[i]} \leftrightarrow \mathtt{a[i+1]}$.

# Sortarea prin interschimbarea elementelor vecine - continuare

- La prima parcurgere, elementul cel mai mare din secvență formează inversiuni cu toate elementele aflate după el și, în urma interschimbărilor realizate, acesta va fi deplasat pe ultimul loc care este și locul său final.
- În iterația următoare, se va întâmpla la fel cu cel de-al doilea element cel mai mare.
- În general, dacă subsecvența $\mathtt{a}[\mathtt{r}+1..\mathtt{n}-1]$ nu are nici o inversiune la iterația curentă, atunci ea nu va avea inversiuni la nici una din iterațiile următoare.
- Aceasta permite ca la iterația următoare să fie verificată numai subsecvența $\mathtt{a}[0..\mathtt{r}]$.
- Terminarea algoritmului este dată de faptul că la fiecare iterație numărul de interschimbari este micșorat cu cel puțin 1.

# Algoritmul *bubbleSort* - pseudocod

```
procedure bubbleSort(a, n)
    ultim ← n-1
    while (ultim > 0) do
        n1  ← ultim - 1
        ultim  ← 0
        for i ← 0 to n1 do
            if (a[i] > a[i+1])
                then interschimba(a[i], a[i+1])
                     ultim  ← i
end
```

# Evaluarea algoritmului

- Cazul cel mai favorabil este întâlnit atunci când secvența de intrare este deja sortată, caz în care algoritmul `bubbleSort` execută $O(n)$ operații.
- Cazul cel mai nefavorabil este obținut când secvența de intrare este ordonată descrescător și, în această situație, procedura execută $O(n^2)$ operații.

# Sortare prin inserție directă

- Algoritmul sortării prin inserție directă consideră că în pasul $k$, elementele `a[0..k-1]` sunt sortate crescător, iar elementul `a[k]` va fi inserat, astfel încât, după această inserare, primele elemente `a[0..k]` să fie sortate crescător.
- Inserarea elementului `a[k]` în secvența `a[0..k-1]` presupune:
  1. memorarea elementului într-o variabilă temporară;
  2. deplasarea tuturor elementelor din vectorul `a[0..k-1]` care sunt mai mari decât `a[k]`, cu o poziție la dreapta (aceasta presupune o parcurgere de la dreapta la stânga);
  3. plasarea lui `a[k]` în locul ultimului element deplasat.

# Algoritmul de sortare prin inserție directă - pseudocod

```
procedure insertionSort(a, n)
    for k ← 1 to n-1 do
        i ← k-1
        temp ← a[k]
        while ((i ≥ 0) and (temp < a[i])) do
            a[i+1] ← a[i]
            i ← i-1
        if (i ≠ k-1) then a[i+1] ← temp
end
```

# Evaluarea algoritmului

- Căutarea poziției $i$ în subsecvența `a[0..k-1]` necesită $O(k-1)$ timp.
- Timpul total în cazul cel mai nefavorabil este $O(1+\cdots+n-1) = O(n^2)$.
- Pentru cazul cel mai favorabil, când valoarea tabloului la intrare este deja în ordine crescătoare, timpul de execuție este $O(n)$.

# Sortare prin selecţie naivă

- Este o metodă mai puţin eficientă, dar foarte simplă în prezentare.
- Se bazează pe următoarea caracterizare a predicatului $SORT(\mathrm{a})$:

$$SORT(\mathrm{a}) \iff (\forall i)(0 \leq i < n) \Rightarrow a[i] = \max\{a[0], \ldots, a[i]\}$$

- Ordinea în care sunt aşezate elementele pe poziţiile lor finale este $n-1, n-2, \ldots, 0$.
- O formulare echivalentă este:

$$SORT(\mathrm{a}) \iff (\forall i)(0 \leq i < n) : a[i] = \min\{a[i], \ldots, a[n]\},$$

caz în care ordinea de aşezare este $0, 1, \ldots, n-1$.

# Algoritmul de sortare prin selecţie naivă - pseudocod

```
procedure naivSort(a, n)
for i ← n-1 downto 1 do
    locmax ← 0
    maxtemp ← a[0]
    for j ← 1 to i do
        if (a[j] > maxtemp)
           then locmax ← j
                maxtemp ← a[j]
    a[locmax] ← a[i]
    a[i] ← maxtemp
end
```

# Evaluarea algoritmului

- Timpul de execuţie este $O(n^2)$ pentru toate cazurile, adică algoritmul `NaivSort` are timpul de execuţie $\Theta(n^2)$.

# Sarcini de lucru și barem de notare

**Sarcini de lucru:**

1. Scrieți o funcție C/C++ care implementează algoritmul bubbleSort.
2. Scrieți o funcție C/C++ care implementează algoritmul insertionSort.
3. Scrieți o funcție C/C++ care implementează algoritmul naivSort.
4. Măsurați timpii de execuție pentru $n$ numere, unde $10.000 \leq n \leq 10.000.000$. Comparați rezultatele.

**Barem de notare:**

1. Funcția bubbleSort: 2p
2. Funcția insertionSort: 2p
3. Funcția naivSort: 2p
4. Măsurarea și compararea timpilor de execuție: 3p
5. Baza: 1p

# Bibliografie

 Lucanu, D. şi Craus, M., *Proiectarea algoritmilor*, Editura Polirom, 2008.

# Proiectarea algoritmilor

## Sortarea prin metoda distribuirii - Algoritmul radixSort

## Lucrare de laborator nr. 6

# Cuprins

# Sortare prin distribuire

- Algoritmii de sortare prin distribuire presupun cunoașterea de informații privind distribuția acestor elemente.
- Aceste informații sunt utilizate pentru a distribui elementele secvenței de sortat în „pachete" care vor fi sortate în același mod sau prin altă metodă, după care pachetele se combină pentru a obține lista finală sortată.

# Sortarea cuvintelor

- Presupunem că avem $n$ fișe, iar fiecare fișă conține un nume ce identifică în mod unic fișa (cheia).
- Se pune problema sortării manuale a fișelor. Pe baza experienței câștigate se procedează astfel:
  - Se împart fișele în pachete, fiecare pachet conținând fișele ale căror cheie începe cu aceeași literă.
  - Apoi se sortează fiecare pachet în aceeași manieră după a doua literă, apoi etc.
  - După sortarea tuturor pachetelor, acestea se concatenează rezultând o listă liniară sortată.
- Vom încerca să formalizăm metoda de mai sus într-un algoritm de sortare a șirurilor de caractere (cuvinte).
- Presupunem că elementele secvenței de sortat sunt șiruri de lungime fixată $m$ definite peste un alfabet cu $k$ litere.
- Echivalent, se poate presupune că elementele de sortat sunt numere reprezentate în baza $k$. Din acest motiv, sortarea cuvintelor este denumită în engleză *radix-sort* (cuvântul *radix* traducându-se prin *bază*).

# Sortarea cuvintelor - continuare

- Dacă urmăm ideea din exemplul cu fișele, atunci algoritmul ar putea fi descris recursiv astfel:
  1. Se împart cele $n$ cuvinte în $k$ pachete, cuvintele din același pachet având aceeași literă pe poziția $i$ (numărând de la stânga la dreapta).
  2. Apoi, fiecare pachet este sortat în aceeași manieră după literele de pe pozițiile $i+1, \ldots, m-1$.
  3. Se concatenează cele $k$ pachete în ordinea dată de literele de pe poziția $i$. Lista obținută este sortată după subcuvintele formate din literele de pe pozițiile $i, i+1, \ldots, m-1$.
- Inițial se consideră $i = 0$. Apare următoarea problemă:
  - Un grup de $k$ pachete nu va putea fi combinat într-o listă sortată decât dacă cele $k$ pachete au fost sortate complet pentru subcuvintele corespunzătoare.
  - Este deci necesară ținerea unei evidențe a pachetelor, fapt care conduce la utilizarea de memorie suplimentară și creșterea gradului de complexitate a metodei.

# Sortarea cuvintelor - continuare

- O simplificare majoră apare dacă împărțirea cuvintelor în pachete se face parcurgând literele acestora de la dreapta la stânga.
- Procedând așa, observăm următorul fapt surprinzător:
  - după ce cuvintele au fost distribuite în $k$ pachete după litera de pe poziția $i$, cele $k$ pachete pot fi combinate înainte de a le distribui după litera de pe poziția $i-1$.
- Exemplu: Presupunem că alfabetul este $\{0 < 1 < 2\}$ și $m = 3$. Cele trei faze care cuprind distribuirea elementelor listei în pachete și apoi concatenarea acestora într-o singură listă sunt sugerate grafic în figura 1.

# Sortarea cuvintelor - continuare

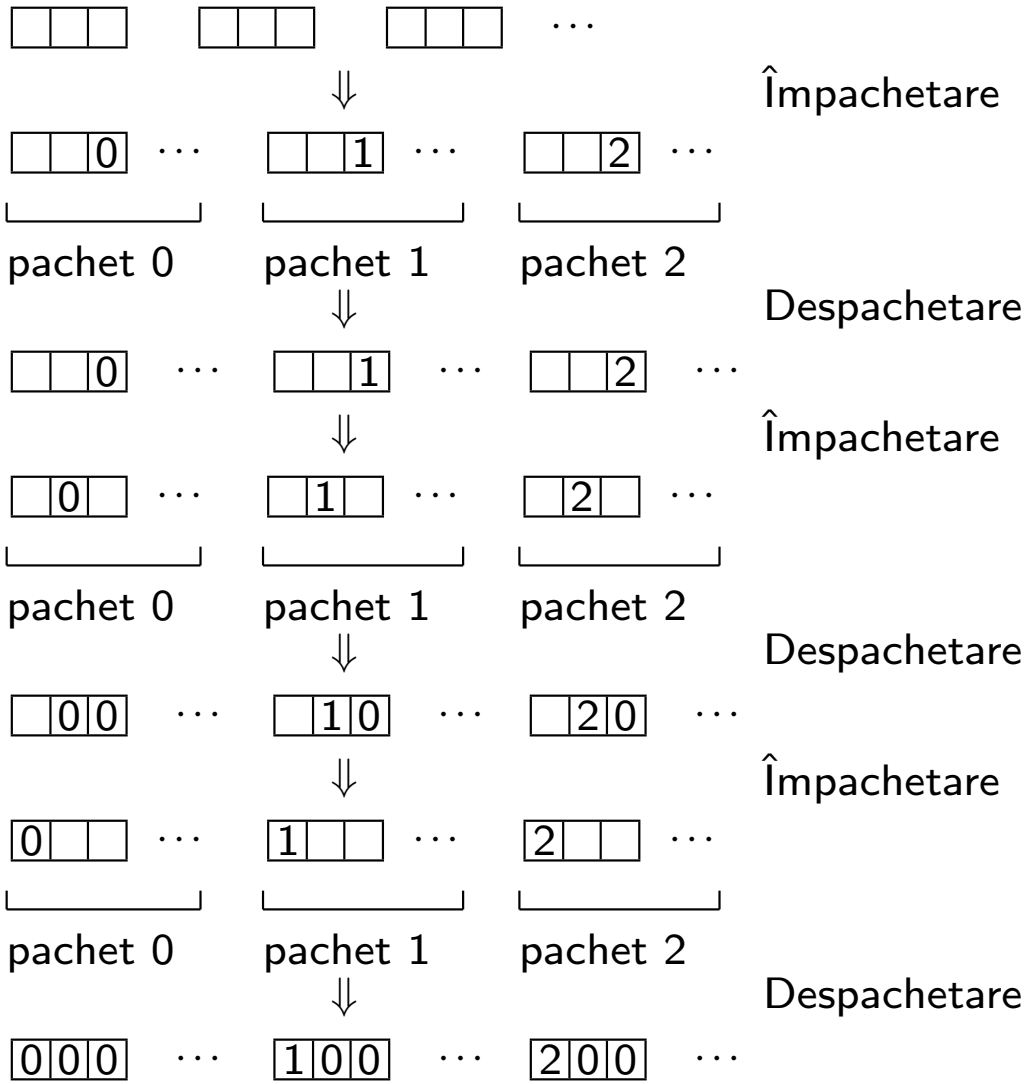



Figura 1 : Sortare prin distribuire

# Algoritmul radixSort - descriere

- Pentru gestionarea pachetelor vom utiliza un tablou de structuri de pointeri numit `pachet`, cu semnificația următoare:
  - *pachet*[*i*] este structura de pointeri *pachet*[*i*].*prim* și *pachet*[*i*].*ultim*,
  - *pachet*[*i*].*prim* face referire la primul element din lista ce reprezintă pachetul *i* și
  - *pachet*[*i*].*ultim* face referire la ultimul element din lista corespunzătoare pachetului *i*.
- Etapa de distribuire este realizată în modul următor:
  1. Inițial, se consideră listele *pachet*[*i*] vide.
  2. Apoi se parcurge secvențial lista supusă distribuirii și fiecare element al acesteia este distribuit în pachetul corespunzător.
- Etapa de combinare a pachetelor constă în concatenarea celor $k$ liste *pachet*[*i*], $i = 0, \ldots, k-1$.

# Algoritmul radixSort - pseudocod

```
procedure radixSort(L,m)
for i ← m-1 downto 0 do
    for j ← 0 to k-1 do
        pachet[j] ← listaVida()
    while (not esteVida(L)) do /* împachetare */
       w ← citeste(L, 0)
       elimina(L, 0)
       insereaza(pachet[w[i]], w)
    for j ← 0 to k-1 do /* despachetare */
        concateneaza(L, pachet[j])
end
```

# Evaluarea algoritmului

- Distribuirea în pachete presupune parcurgerea completă a listei de intrare, iar procesarea fiecărui element al listei necesită $O(1)$ operații.
  - Faza de distribuire se face în timpul $O(n)$, unde $n$ este numărul de elemente din listă.
- Combinarea pachetelor presupune o parcurgere a tabloului `pachet`, iar adăugarea unui pachet se face cu $O(1)$ operații, cu ajutorul tabloului `ultim`.
  - Faza de combinare a pachetelor necesită $O(k)$ timp.
- Algoritmul `radixSort` are un timp de execuție de $O(m \cdot n)$.

# Sarcini de lucru și barem de notare

**Sarcini de lucru:**

1. Scrieți o funcție C/C++ care implementează algoritmul radixSort. Se presupune că secvența de sortat este formată din $n$ numere întregi, cu cifre din baza 10: $s = (a_0, a_1, \ldots, a_{n-1})$. Fiecare număr $a_i$, $i = 0, 1, \ldots, n-1$, din secvența de sortat are maxim $k$ cifre ($a_i = c_1 c_2 \ldots c_k$).
2. Măsurați timpul de execuție pentru $n$ numere, unde $10.000 \leq n \leq 10.000.000$.

**Barem de notare:**

1. Funcția radixSort: 7p
2. Măsurarea timpului de execuție: 2p
3. Baza: 1p

# Bibliografie

 Lucanu, D. și Craus, M., *Proiectarea algoritmilor*, Editura Polirom, 2008.

# Proiectarea algoritmilor

**Paradigma** *Divide_et_Impera*

**Lucrare de laborator nr. 7**

# Cuprins

# Sortare rapidă (Quick Sort) - descriere

- Ca și în cazul algoritmului *Merge Sort*, vom presupune că trebuie sortată o secvență memorată într-un tablou a[p..q].
- Divizarea problemei constă în alegerea unei valori $x$ din $a[p..q]$ și determinarea prin interschimbări a unui indice $k$ cu proprietățile:
  - $p \leq k \leq q$ și $a[k] = x$;
  - $\forall i : p \leq i \leq k \Rightarrow a[i] \leq a[k]$;
  - $\forall j : k < j \leq q \Rightarrow a[k] \leq a[j]$;
- Elementul $x$ este numit *pivot*. În general, se alege pivotul $x = a[p]$, dar nu este obligatoriu.
- Partiționarea tabloului se face prin interschimbări care mențin invariante proprietăți asemănătoare cu cele de mai sus.
- Se consideră două variabile index: `i` cu care se parcurge tabloul de la stânga la dreapta și `j` cu care se parcurge tabloul de la dreapta la stânga. Inițial se ia $i = p+1$ și $j = q$.
- Proprietățile menținute invariante în timpul procesului de partiționare sunt:

$$\forall i' : p \leq i' < i \Rightarrow a[i'] \leq x \tag{1}$$

și

$$\forall j' : j < j' \leq q \Rightarrow a[j'] \geq x \tag{2}$$

# Sortare rapidă (Quick Sort) - descriere (continuare)

- Presupunem că la momentul curent sunt comparate elementele a[i] și a[j] cu $i < j$.
- Distingem următoarele cazuri:
  1. $a[i] \leq x$. Transformarea $\texttt{i} \leftarrow \texttt{i} + 1$ păstrează proprietatea (1).
  2. $a[j] \geq x$. Transformarea $\texttt{j} \leftarrow \texttt{j} - 1$ păstrează proprietatea (2).
  3. $a[i] > x > a[j]$. Dacă se realizează interschimbarea $\texttt{a[i]} \leftrightarrow \texttt{a[j]}$ și se face $\texttt{i} \leftarrow \texttt{i} + 1$ și $\texttt{j} \leftarrow \texttt{j} - 1$, atunci sunt păstrate ambele predicate (1) și (2).
- Operațiile de mai sus sunt repetate până când $i$ devine mai mare decât $j$.
- Considerând $k = i - 1$ și interschimbând a[p] cu a[k] obținem partiționarea dorită a tabloului.
- După sortarea recursivă a subtablourilor $\texttt{a[p..k} - 1]$ și $\texttt{a[k} + 1\texttt{..q]}$ se observă că tabloul este sortat deja. Astfel partea de asamblare a soluțiilor este vidă.

# Quick Sort - pseudocod

```
procedure quickSort1(a, p, q)
   if (p < q)
      then / * determină prin interschimbări indicele k pentru care:
                p ≤ k ≤ q
                (∀)i : p ≤ i ≤ k ⇒ a[i] ≤ a[k]
                (∀)j : k < j ≤ q ⇒ a[k] ≥ a[j]  */
           partitioneaza(a, p, q, k)
           quickSort(a, p, k-1)
           quickSort(a, k+1, q)
end
```

# Quick Sort - pseudocod

```
procedure partitioneaza1(a, p, q, k)
   x ← a[p]
   i ← p + 1
   j ← q
   while (i ≤ j) do
      if (a[i] ≤ x then i ← i + 1
      if (a[j] ≥ x then j ← j - 1
      if (i < j)
         then if ((a[i] > x) and (x > a[j]))
                 then interschimba(a[i],a[j])
                      i ← i + 1
                      j ← j - 1
   k ← i-1
   a[p] ← a[k]
   a[k] ← x
end
```

# Quick Sort - pseudocod (continuare)

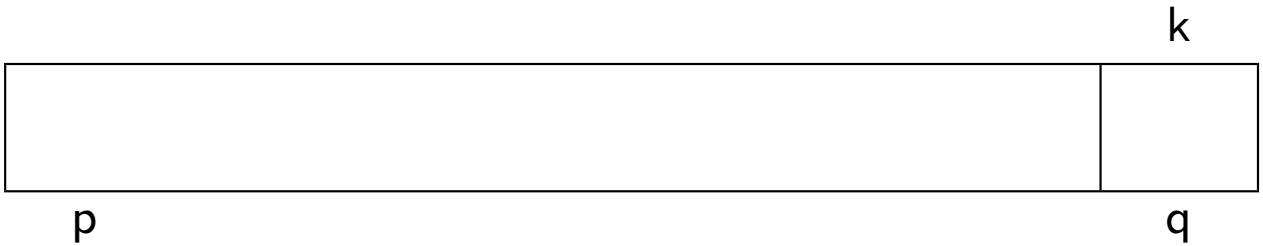
```
procedure partitioneaza2(a, p, q, k)
   x ← a[p]
   i ← p
   j ← q
   while (i < j) do
      while (a[i] ≤ x and i ≤ q) do i ← i + 1
      while (a[j] > x and j ≥ p) do j ← j - 1
      if (i < j)
         then interschimba(a[i],a[j])
   k ← j
   a[p] ← a[k]
   a[k] ← x
end
```

# Complexitatea timp a algoritmului Quick Sort

- Cazul cel mai nefavorabil se obține atunci când la fiecare partiționare se obține una din subprobleme cu dimensiunea 1.
- Deoarece operația de partiționare necesită $O(q-p)$ comparații, rezultă că pentru acest caz numărul de comparații este $O(n^2)$.
- Acest rezultat este oarecum surprinzător, având în vedere că numele algoritmului este „sortare rapidă”.
- Numele algoritmului se justifică prin faptul că într-o distribuție normală, cazurile pentru care quickSort execută $n^2$ comparații sunt rare, fapt care conduce la o complexitate medie foarte bună a algoritmului.
- Complexitatea medie a algoritmului QuickSort este $O(n \log_2 n)$.

# Consumul de memorie

- La execuția algoritmilor recursivi, importantă este și spațiul de memorie ocupat de stivă.
- Considerăm spațiul de memorie ocupat de stivă în cazul cel mai nefavorabil, $k = q$:




- În acest caz, spațiul de memorie ocupat de stivă este $M(n) = c + M(n-1)$, ce implică $M(n) = O(n)$.
- În general, pivotul împarte secvența de sortat în două subsecvențe.
- Dacă subsecvența mică este rezolvată recursiv, iar subsecvența mare este rezolvată iterativ, atunci consumul de memorie se reduce.

# Quick Sort - pseudocod îmbunătățit

```
procedure quickSort2(a, p, q)
   while (p < q) do
      partitioneaza(a, p, q, k)
      if (k-p > q-k)
         then quickSort(a, k+1, q)
              q ← k-1
         else quickSort(a, p, k-1)
              p ← k+1
end
```

# Consumul de memorie în cazul algoritmului îmbunătățit

- Spațiul de memorie ocupat de stivă pentru algoritmul îmbunătățit satisface relația $M(n) \leq c + M(n/2)$, de unde rezultă $M(n) = O(\log n)$.
- Dacă tabloul a conține multe elemente egale, atunci algoritmul `partitioneaza` realizează multe interschimbări inutile (de elemente egale).
  *Exercițiu*: Să se modifice algoritmul astfel încât noua versiune să elimine acest inconvenient.

# Sarcini de lucru și barem de notare

Sarcini de lucru:

1. Scrieți o funcție C/C++ care implementează o algoritmul quickSort1;
2. Scrieți o funcție C/C++ care implementează o algoritmul quickSort2;
3. Comparați funcțiile quickSort1, quickSort1 și qSort (din biblioteca STL). Pentru aceasta, măsurați timpii de execuție pentru $n$ chei de sortare ($10.000 \leq n \leq 10.000.000$).

Barem de notare:

1. Funcția quickSort1: 4p
2. Funcția quickSort2: 3p
3. Compararea funcțiilor quickSort1, quickSort1 și qSort: 2p
4. Baza: 1p

# Bibliografie

 Lucanu, D. și Craus, M., *Proiectarea algoritmilor*, Editura Polirom, 2008.

# Proiectarea algoritmilor

**Paradigma** *Divide_et_Impera*

**Lucrare de laborator nr. 8**

# Cuprins

# Înmulțirea matricelor pătratice - algoritm clasic

- Considerăm algoritmul clasic de înmulțire a matricelor pătratice:

```
procedure inmultireMatrice(A, B, C, n )
   for i ← 0 to n-1
       for j ← 0 to n-1
           C[i,j] ← 0
           for k ← 0 to n-1
               C[i,j] ← C[i,j] + A[i,k] * B[k,j]
end
```

- Mult timp s-a crezut că bariera de $O(n^3)$ nu poate fi depășită.
- *Strassen* a demonstrat însă că se poate obține un algoritm mai bun.
- Metoda lui *Strassen* folosește un algoritm de înmulțire bazat pe descompunerea matricelor în sferturi.

# Înmulțirea matricelor pătratice - algoritm *Divide_et_Impera* clasic

$$\begin{bmatrix} A_{1,1} & A_{1,2} \\ A_{2,1} & A_{2,2} \end{bmatrix} \begin{bmatrix} B_{1,1} & B_{1,2} \\ B_{2,1} & B_{2,2} \end{bmatrix} = \begin{bmatrix} C_{1,1} & C_{1,2} \\ C_{2,1} & C_{2,2} \end{bmatrix}$$

$$\begin{aligned} C_{1,1} &= A_{1,1}B_{1,1} + A_{1,2}B_{2,1} \\ C_{1,2} &= A_{1,1}B_{1,2} + A_{1,2}B_{2,2} \\ C_{2,1} &= A_{2,1}B_{1,1} + A_{2,2}B_{2,1} \\ C_{2,2} &= A_{2,1}B_{1,2} + A_{2,2}B_{2,2} \end{aligned}$$

$$\mathbf{AB} = \begin{bmatrix} 3 & 4 & 1 & 6 \\ 1 & 2 & 5 & 7 \\ 5 & 1 & 2 & 9 \\ 4 & 3 & 5 & 6 \end{bmatrix} \begin{bmatrix} 5 & 6 & 9 & 3 \\ 4 & 5 & 3 & 1 \\ 1 & 1 & 8 & 4 \\ 3 & 1 & 4 & 1 \end{bmatrix}$$

$$A_{1,1} = \begin{bmatrix} 3 & 4 \\ 1 & 2 \end{bmatrix} \quad A_{1,2} = \begin{bmatrix} 1 & 6 \\ 5 & 7 \end{bmatrix} \quad B_{1,1} = \begin{bmatrix} 5 & 6 \\ 4 & 5 \end{bmatrix} \quad B_{1,2} = \begin{bmatrix} 9 & 3 \\ 3 & 1 \end{bmatrix}$$

$$A_{2,1} = \begin{bmatrix} 5 & 1 \\ 4 & 3 \end{bmatrix} \quad A_{2,2} = \begin{bmatrix} 2 & 9 \\ 5 & 6 \end{bmatrix} \quad B_{2,1} = \begin{bmatrix} 1 & 1 \\ 3 & 1 \end{bmatrix} \quad B_{2,2} = \begin{bmatrix} 8 & 4 \\ 4 & 1 \end{bmatrix}$$

Figura 1 : Exemplu de înmulțire a matricelor prin descompunerea în sferturi

- $T(n) = 8T(n/2) + O(n^2)$
- Din teorema complexității *Divide_et_Impera* rezultă $T(n) = O(n^3)$ !! Nici un progres !

# Înmulțirea matricelor pătratice - Metoda lui Strassen

- Strassen a utilizat o strategie similară inmulțirii numerelor întregi și a arătat că pot fi utilizate doar 7 înmulțiri în loc de 8.
- Cele 7 înmulțiri sunt următoarele:
$$M_1 = (A_{1,2} - A_{2,2})(B_{2,1} + B_{2,2})$$
$$M_2 = (A_{1,1} + A_{2,2})(B_{1,1} + B_{2,2})$$
$$M_3 = (A_{1,1} - A_{2,1})(B_{1,1} + B_{1,2})$$
$$M_4 = (A_{1,1} + A_{1,2})B_{2,2}$$
$$M_5 = A_{1,1}(B_{1,2} - B_{2,2})$$
$$M_6 = A_{2,2}(B_{2,1} - B_{1,1})$$
$$M_7 = (A_{2,1} + A_{2,2})B_{1,1}$$
- Apoi:
$$C_{1,1} = M_1 + M_2 - M_4 + M_6$$
$$C_{1,2} = M_4 + M_5$$
$$C_{1,3} = M_6 + M_7$$
$$C_{1,4} = M_2 - M_3 + M_5 - M_7$$
- $T(n) = 7T(n/2) + O(n^2)$. Rezultă $T(n) = O(n^{\log 27}) = O(n^{2.81})$ !!!

# Sarcini de lucru și barem de notare

Sarcini de lucru:

1. Scrieți un program C/C++ care implementează metoda *Divide_et_Impera* clasică pentru înmulțirea a două matrice pătratice.
2. Scrieți un program C/C++ care implementează metoda lui Strassen pentru înmulțirea a două matrice pătratice.

Barem de notare:

1. Scrierea pseudocodului algoritmului *Divide_et_Impera* clasic: 3p
2. Implementarea algoritmului *Divide_et_Impera* clasic: 3p
3. Aplicarea metodei lui Strassen: 3p
4. Baza: 1p

# Bibliografie


M. A. Weiss, *Data Structures and Algorithm Analysis in C*, The Benjamin/Cummings Publishing Company, Inc., 1992.

# Proiectarea algoritmilor

**Paradigma *Divide_et_Impera***

**Lucrare de laborator nr. 9**

# Cuprins

# Înmulțirea numerelor întregi - algoritm *Divide_et_Impera* clasic

- Să presupunem că dorim să înmulțim două numere întregi $x$ și $y$, formate din $n$ cifre, in baza $b$.
- Putem presupune ca $x$ si $y$ sunt pozitive. Algoritmul clasic necesită $O(n^2)$ operații.
- De exemplu, dacă baza $b = 10$ și $x = 61,438,521$ și $y = 94,736,407$, atunci $xy = 5,820,464,730,934,047$.
- Să spargem acum $x$ și $y$ în două jumătăți. Rezultă $x_s = 6,143$, $x_d = 8,521$, $y_s = 9,473$, și $y_d = 6,407$.
- Vom avea $x = x_s 10^4 + x_d$ și $y = y_s 10^4 + y_d$. Urmează că $xy = x_s y_s 10^8 + (x_s y_d + x_d y_s)10^4 + x_d y_d$.
- Astfel, pentru înmulțirea numerelor întregi $x$ și $y$, sunt necesare 4 înmulțiri de numere formate din $n/2$ cifre: $x_s y_s$, $x_s y_d$, $x_d y_s$ și $x_d y_d$.
- Inmulțirea cu $10^8$ și $10^4$ înseamnă adăugarea de zerouri, ceea ce implică $O(n)$ operații suplimentare.

# Înmulțirea numerelor întregi - algoritm *Divide_et_Impera* îmbunătățit

- Dacă înmulțim recursiv obținem recurența $T(n) = 4T(n/2) + O(n)$
- Din teorema complexității *Divide_et_Impera* rezultă $T(n) = O(n^2)$.
- Pentru a obține un algoritm subpătratic, trebuie să reducem numărul apelurilor recursive.
- Observația cheie este $x_s y_d + x_d y_s = (x_s - x_d)(y_d - y_s) + x_s y_s + x_d y_d$
- În locul a două înmulțiri pentru a obține coeficientul lui $10^4$, putem face o înmulțire și apoi să folosim rezultatul a două înmultiri deja efectuate.
- Astfel, pentru înmulțirea numerelor întregi $x$ și $y$, sunt necesare 3 înmulțiri de numere formate din $n/2$ cifre: $x_s y_s$, $x_d y_d$ și $(x_s - x_d)(y_d - y_s)$ .
- În acest fel, numărul apelurilor recursive este redus la 3.
- Rezultă $T(n) = 3T(n/2) + O(n)$, adică $T(n) = O(n^{\log_2 3}) = O(n^{1.59})$.

# Sarcini de lucru și barem de notare

Sarcini de lucru:

1. Scrieți un program C/C++ care implementează o metodă *Divide_et_Impera* pentru înmulțirea a două numere întregi formate din 128 cifre binare.

Barem de notare:

1. Aplicarea corectă a metodei *Divide_et_Impera*: 6p
2. Funcția de înmulțire conține 3 apeluri recursive: 3p
3. Baza: 1p

# Bibliografie


M. A. Weiss, *Data Structures and Algorithm Analysis in C*, The Benjamin/Cummings Publishing Company, Inc., 1992.

# Proiectarea algoritmilor

**Paradigma *Greedy***

**Lucrare de laborator nr. 10**

# Cuprins

# Arbori binari ponderați pe frontieră - descriere

- Considerăm arbori binari cu proprietatea că orice vârf are 0 sau 2 succesori și vârfurile de pe frontieră au ca informații (etichete, ponderi) numere, notate cu $info(v)$.
- Convenim să numim acești arbori ca fiind *ponderați pe frontieră*.
- Pentru un vârf $v$ din arborele $t$ notăm cu $d_v$ lungimea drumului de la rădăcina lui $t$ la vârful $v$.
- *Lungimea externă ponderată* a arborelui $t$ este:

$$\mathrm{LEP}(t) = \sum_{v \text{ pe frontiera lui } t} d_v \cdot info(v)$$

- Modificăm acești arbori etichetând vârfurile interne cu numere ce reprezintă suma etichetelor din cele două vârfuri fii.
- Pentru orice vârf intern $v$ avem $info(v) = info(v_1) + info(v_2)$, unde $v_1, v_2$ sunt fiii lui $v$ (Figura 1).

# Arbori binari ponderați pe frontieră - exemplu

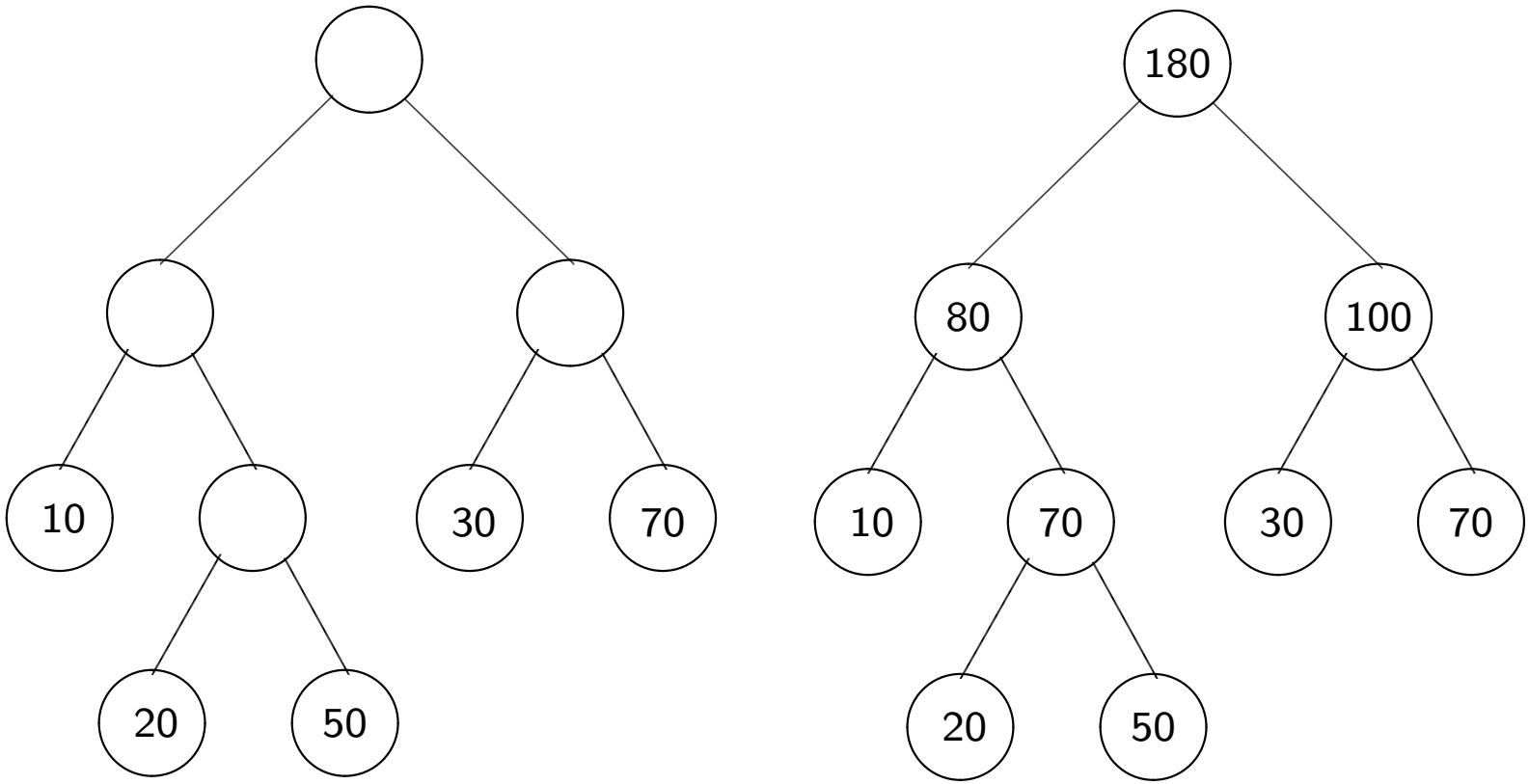



Figura 1: Arbore ponderat pe frontieră, înainte și după modificare

# Lungimea externă ponderată

## Lemma (1)

*Fie $t$ un arbore binar ponderat pe frontieră.*
*Atunci*

$$\mathrm{LEP}(t) = \sum_{v \text{ intern în } t} info(v)$$

## Lemma (2)

*Fie $t$ un arbore din $\mathscr{T}(x)$ cu LEP minimă și $v_1, v_2$ două vârfuri pe frontiera lui $t$. Dacă $info(v_1) < info(v_2)$ atunci $d_{v_1} \geq d_{v_2}$.*

## Lemma (3)

*Presupunem $x_0 \leq x_1 \leq \cdots \leq x_{n-1}$. Există un arbore în $\mathscr{T}(x)$ cu LEP minimă și în care vârfurile etichetate cu $x_0$ și $x_1$ (vârfurile sunt situate pe frontieră) sunt frați.*

# Algoritm pentru construirea unui arbore cu lungimea externă ponderată minimă - descriere

- Ideea algoritmului rezultă direct din Lema 3.
- Presupunem $x_0 \leq x_1 \leq \cdots \leq x_{n-1}$.
- Știm că există un arbore optim $t$ în care $x_0$ și $x_1$ sunt memorate în vârfuri frate. Tatăl celor două vârfuri va memora $x_0 + x_1$.
- Prin ștergerea celor două vârfuri ce memorează $x_0$ și $x_1$ se obține un arbore $t'$.
- Fie $t1'$ un arbore optim pentru secvența $y = (x_0 + x_1, x_2, \ldots, x_{n-1})$ și $t1$ arborele obținut din $t1'$ prin „agățarea" a două vârfuri cu informațiile $x_0$ și $x_1$ de vârful ce memorează $x_0 + x_1$.
- Avem $\mathrm{LEP}(t1') \leq \mathrm{LEP}(t')$ ce implică

$$\mathrm{LEP}(t1) = \mathrm{LEP}(t1') + x_0 + x_1 \leq \mathrm{LEP}(t') + x_0 + x_1 = \mathrm{LEP}(t)$$

.

- Cum $t$ este optim, rezultă $\mathrm{LEP}(t1) = \mathrm{LEP}(t)$ și de aici $t'$ este optim pentru secvența $y$.

# Algoritm pentru construirea unui arbore cu lungimea externă ponderată minimă - pseudocod

- Considerăm în loc de secvențe de numere secvențe de arbori.
- *Notații:* $t(x_i)$ desemnează arborele format dintr-un singur vârf etichetat cu $x_i$ iar $rad(t)$ rădăcina arborelui $t$.
- *Premise*: Inițial se consideră $n$ arbori cu un singur vârf, care memorează numerele $x_i, i = 0, \ldots, n-1$.

```
procedure lep(x, n)
   1:  B ← {t(x_0), ..., t(x_{n-1})}
   2:  while (#B > 1) do
   3:      alege t1, t2 din B cu info(rad(t1)),info(rad(t2)) minime
   4:      construieşte arborele t în care subarborii rădăcinii
   5:        sunt t1, t2 şi info(rad(t))=info(rad(t1))+info(rad(t2))
   6:      B ← (B \ {t1,t2}) ∪ {t}
end
```

# Implementarea algoritmului pentru construirea unui arbore cu lungimea externă ponderată minimă

a) Dacă mulțimea $B$ este implementată printr-o listă liniară, atunci în cazul cel mai nefavorabil operația 3 este are timpul de execuție $O(n)$, iar operația 6 are timpul de execuție $O(1)$.

b) Dacă mulțimea $B$ este implementată printr-o listă liniară ordonată, atunci în cazul cel mai nefavorabil operația 3 are timpul de execuție $O(1)$, iar operația 6 are timpul de execuție $O(n)$.

c) Dacă mulțimea $B$ este implementată printr-un *heap*, atunci în cazul cel mai nefavorabil operația 3 are timpul de execuție $O(\log n)$, iar operația 6 are timpul de execuție $O(\log n)$.

*Concluzie*: *heapul* este alegerea cea mai bună pentru implementarea mulțimii $B$.

# Interclasarea optimală a unei mulțimi de secvențe sortate
## Descrierea problemei

- Se consideră $m$ secvențe sortate $a_0, \ldots, a_{m-1}$ care conțin $n_0, \ldots, n_{m-1}$, respectiv, elemente dintr-o mulțime total ordonată.
- Interclasarea celor $m$ secvențe constă în execuția repetată a următorului proces:
  - Se extrag din mulțime două secvențe și se pune în locul lor secvența obținută prin interclasarea acestora.
- Procesul se continuă până când se obține o singură secvențe sortată cu cele $n_0 + \cdots + n_{m-1}$ elemente.
- Problema constă în determinarea unei alegeri pentru care numărul total de transferuri de elemente să fie minim.
- Un exemplu este este dat de *sortarea externă*
  - Presupunem că avem de sortat un volum mare de date ce nu poate fi încărcat în memoria internă.
  - Se partiționează colecția de date în în mai multe secvențe ce pot fi ordonate cu unul dintre algoritmii de sortare internă.
  - Secvențele sortate sunt memorate în fișiere pe suport extern.
  - Sortarea întregii colecții se face prin interclasarea fișierelor ce memorează secvențele sortate.

# Interclasarea unei mulțimi de secvențe sortate - exemplu

- Considerăm problema interclasării a două secvențe sortate:

  Fie date două secvențe sortate $x = (x_0, \ldots, x_{p-1})$ și $y = (y_0, \ldots, y_{q-1})$ ce conțin elemente dintr-o mulțime total ordonată. Să se construiască o secvență sortată $z = (z_0, \ldots, z_{p+q-1})$ care să conțină cele $p+q$ elemente ce apar în $x$ și $y$.

- Utilizăm notația $z = \text{merge}(x, y)$ pentru a nota faptul că $z$ este rezultatul interclasării secvențelor $x$ și $y$.
- Numărul de comparații executate de algoritmul $\text{merge}(x, y)$ este cel mult $p+q-1$, iar numărul de elemente transferate este $p+q$.
- Revenim la problema interclasării a $m$ secvențe.
  - Considerăm un exemplu: Fie $m = 5, n_0 = 20, n_1 = 60, n_2 = 70, n_3 = 40, n_4 = 30$.
  - Un mod de alegere a secvențelor pentru interclasare este următorul:

    $b_0 = \text{merge}(a_0, a_1)$
    $b_1 = \text{merge}(b_0, a_2)$
    $b_2 = \text{merge}(a_3, a_4)$
    $b = \text{merge}(b_1, b_2)$

    (vezi Figura 7.a)
  - Numărul de transferuri al acestei soluții este
    $(20+60)+(80+70)+(40+30)+(150+70) = 80+150+70+220 = 520$.
  - Există alegeri mai bune?
  - Răspunsul este afirmativ!!

# Interclasarea optimală a unei mulțimi de secvențe sortate - algoritm

- Unei alegeri i se poate atașa un arbore binar în modul următor:
  - informațiile din vârfuri sunt lungimi de secvențe;
  - vârfurile de pe frontieră corespund secvențelor inițiale $a_0, \ldots, a_{m-1}$;
  - vârfurile interne corespund secvențelor intermediare.
- Se observă ușor că aceștia sunt arbori ponderați pe frontieră și numărul de transferuri de elemente corespunzător unei alegeri este egală cu $\mathrm{LEP}$ a arborelui asociat.
- Așadar, alegerea optimă corespunde arborelui cu $\mathrm{LEP}$ minimă.
- Pentru exemplul anterior ($m = 5, n_0 = 20, n_1 = 60, n_2 = 70, n_3 = 40, n_4 = 30.$), soluția optimă dată de algoritmul greedy este:

  $b_0 = merge(a_0, a_4)$
  $b_1 = merge(a_3, b_0)$
  $b_2 = merge(a_1, a_2)$
  $b = merge(b_1, b_2)$

  (vezi Figura 7.b)
- Numărul de comparații este $50 + 90 + 130 + 220 = 490$.

# Arborii asociați celor doua moduri de alegere a secvențelor pentru interclasare

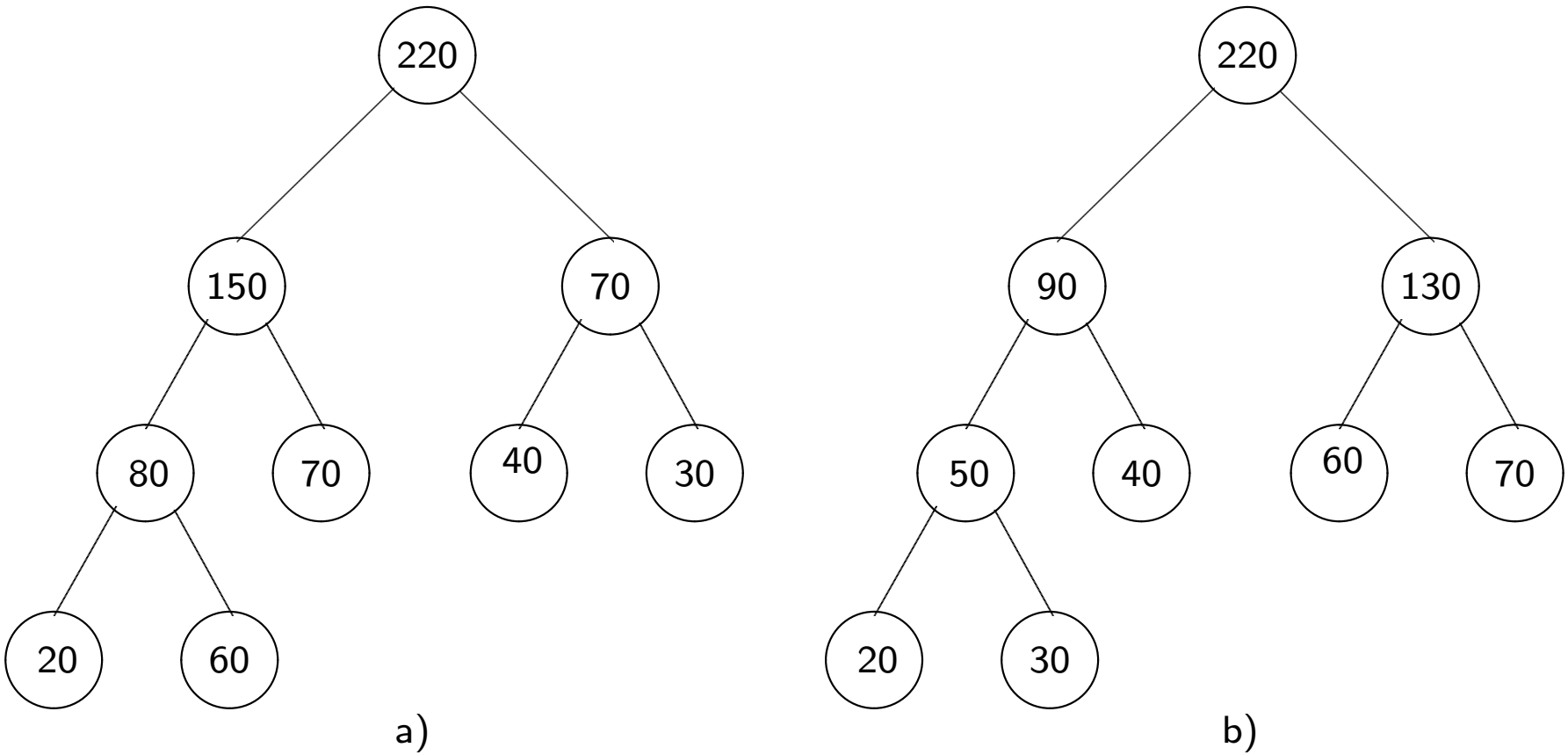



Figura 2: Arborii asociați celor doua soluții de intreclasare

# Algoritm de construcție a arborelui de interclasare optimală a unei mulțimi de secvențe sortate - descriere

- Presupunem că intrarea este memorată într-un tablou `T` de structuri cu două câmpuri:
  - `T[i].secv` conține adresa secvenței sortate $a_i$;
  - `T[i].n` conține numărul de elemente din secvența $a_i$.
- Algoritmul care urmează utilizează reprezentarea arborilor prin tablouri de structuri.
- Notăm cu `H` tabloul ce reprezintă arborele de interclasare.
- Tabloul `H` conține structuri formate din trei câmpuri:
  - `H[i].elt`;
  - `H[i].fst` = indicele fiului de pe partea stângă;
  - `H[i].fdr` = indicele fiului de pe partea dreaptă.
- Semnificația câmpului `H[i].elt` este următoarea:
  - dacă `i` este nod intern, atunci `H[i].elt` reprezintă informația calculată din nod;
  - dacă `i` este pe frontieră (corespunde unui mesaj), atunci `H[i].elt` este adresa din `T` a secvenței corespunzătoare.
- Notăm cu `val(i)` funcția care intoarce informația din nodul `i`, calculată ca mai sus.
- Tabloul `H`, care în final va memora arborele de interclasare optimală, va memora pe parcursul construcției acestuia colecțiile intermediare de arbori.

# Algoritm de construcție a arborelui de interclasare optimală a unei mulțimi de secvențe sortate - descriere (continuare)

- În timpul execuției algoritmului de construcție a arborelui, `H` este compus din trei părți (Figura 3):

  Partea I: un min-*heap* care va conține rădăcinile arborilor din colecție;
  Partea a II-a: conține nodurile care nu sunt rădăcini;
  Partea a III-a: zonă vidă în care se poate extinde partea din mijloc.

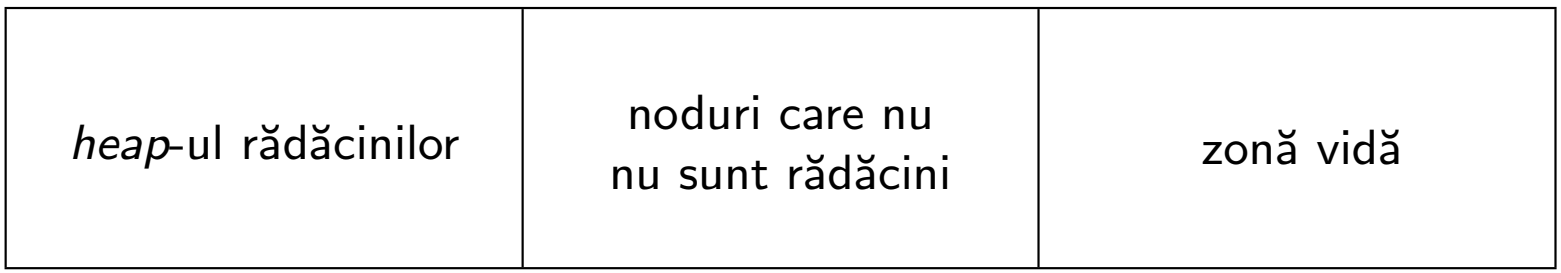



Figura 3: Organizarea tabloului `H`

# Algoritm de construcție a arborelui de interclasare optimală a unei mulțimi de secvențe sortate (continuare)

Un pas al algoritmului de construcție ce realizează selecția *greedy* presupune parcurgerea următoarelor etape:

1. Mutarea rădăcinii cu informația cea mai mică pe prima poziție liberă din zona a treia, să zicem $k$. Aceasta este realizată de următoarele operații:
   a) copierea rădăcinii de pe prima poziție din heap pe poziția $k$:
   `H[k]` $\leftarrow$ `H[1]`
   `k` $\leftarrow$ `k + 1`
   b) mutarea ultimului element din heap pe prima poziție:
   `H[1]` $\leftarrow$ `H[m]`
   `m` $\leftarrow$ `m - 1`
   c) refacerea min-*heapului*.
2. Copierea rădăcinii cu informația cea mai mică pe prima poziție liberă din zona a treia, fără a o elimina din min-*heap*:
   `H[k]` $\leftarrow$ `H[1]`
   `k` $\leftarrow$ `k + 1`
3. Construirea noii rădăcini și memorarea acesteia pe prima poziție în min-*heap* (în locul celei copiate anterior).
4. Refacerea min-*heapului*. .

- Algoritmul rezultat are timpul de execuție $O(n \log n)$.

# Sarcini de lucru și barem de notare

Sarcini de lucru:

1. Scrieți o funcție C/C++ care implementează un algoritm de construcție a arborelui de interclasare optimală a unei mulțimi de secvențe sortate.
2. Date fiind $n$ secvențe sortate, scriețiun program care să determine o alegere optimă în cazul interclasării a $n$ secvențe sortate.

Barem de notare:

1. Implementarea algoritmului de construcție a arborelui de interclasare optimală: 6p
2. Determinarea unei alegeri optime în cazul interclasării a $n$ secvențe sortate: 3p
3. Baza: 1p

Temă suplimentară:

1. Consderăm un graf $G = (V, E)$. Scrieți un program C/C++ pentru determinarea drumurilor minime între un vârf sursă și celelalte vărfuri.

# Bibliografie


Lucanu, D. și Craus, M., *Proiectarea algoritmilor*, Editura Polirom, 2008.


Moret, B.M.E.și Shapiro, H.D. , *Algorithms from P to NP: Design and Efficiency*, The Benjamin/Cummings Publishing Company, Inc., 1991.

# Proiectarea algoritmilor

## Paradigma programării dinamice

## Lucrare de laborator nr. 11

# Cuprins

# Problema drumurilor minime între oricare două vârfuri

- Se consideră un digraf ponderat $D = (\langle V, A \rangle, \ell)$.
- Problema constă în a determina, pentru oricare două vârfuri $i, j$, un drum de lungime minimă de la vârful $i$ la vârful $j$ (dacă există).
- Metoda utilizată este programarea dinamică.

# Problema drumurilor minime - modelul matematic

- Extindem funcția $\ell$ la $\ell : V \times V \to \mathscr{R}$, prin asignarea $\ell_{ij} = \infty$ pentru acele perechi de vârfuri distincte cu $\langle i,j \rangle \notin E$ și $\ell_{ii} = 0$ pentru orice $i = 0, \ldots, n-1$.
- Definim starea problemei ca fiind subproblema corespunzătoare determinării drumurilor de lungime minimă cu vârfuri intermediare din mulțimea $X \subseteq V$, $\mathrm{DM2VD}(X)$ (Drum Minim între oricare două Vârfuri ale unui Digraf).
- Evident, $\mathrm{DM2VD}(V)$ este chiar problema inițială.
- Notăm cu $\ell_{ij}^X$ lungimea drumului minim de la $i$ la $j$ construit cu vârfuri intermediare din $X$. Dacă $X = \emptyset$, atunci $\ell_{ij}^{\emptyset} = \ell_{ij}$.
- Considerăm decizia optimă care transformă starea $\mathrm{DM2VD}(X \cup \{k\})$ în $\mathrm{DM2VD}(X)$.
- Presupunem că $(G, \ell)$ *este un digraf ponderat fără circuite negative*.
- Fie $\rho$ un drum optim de la $i$ la $j$ ce conține vârfuri intermediare din mulțimea $X \cup \{k\}$.
- Avem $\mathrm{lung}(\rho) = \ell_{ij}^{X \cup \{k\}}$, unde $\mathrm{lung}(\rho)$ este lungimea drumului $\rho$.
- Dacă vârful $k$ nu aparține lui $\rho$, atunci politica obținerii lui $\rho$ corespunde stării $\mathrm{DM2VD}(X)$ și, aplicând principiul de optim, obținem:

$$\ell_{ij}^X = \mathrm{lung}(\rho) = \ell_{ij}^{X \cup \{k\}}$$

# Problema drumurilor minime - modelul matematic (continuare)

- În cazul în care $k$ aparține drumului $\rho$, notăm cu $\rho_1$ subdrumul lui $\rho$ de la $i$ la $k$ și cu $\rho_2$ subdrumul de la $k$ la $j$.
- Aceste două subdrumuri au vârfuri intermediare numai din $X$.
- Conform principiului de optim, politica optimă corespunzătoare stării $\mathrm{DM2VD}(X)$ este subpolitică a politicii optime corespunzătoare stării $\mathrm{DM2VD}(X \cup \{k\})$.
- Rezultă că $\rho_1$ și $\rho_2$ sunt optime în $\mathrm{DM2VD}(X)$.
- De aici rezultă:

$$\ell_{ij}^{X\cup\{k\}} = \mathrm{lung}(\rho) = \mathrm{lung}(\rho_1) + \mathrm{lung}(\rho_2) = \ell_{ik}^{X} + \ell_{kj}^{X}$$

- Acum, ecuația funcțională analitică pentru valorile optime $\ell_{ij}^{X}$ are următoarea formă:

$$\ell_{ij}^{X\cup\{k\}} = \min\{\ell_{ij}^{X}, \ell_{ik}^{X} + \ell_{kj}^{X}\}$$

# Problema drumurilor minime - modelul matematic (continuare)

- **Corolar:** Dacă $\langle D, \ell \rangle$ nu are circuite de lungime negativă, atunci au loc următoarele relații:

$$\ell_{kk}^{X \cup \{k\}} = 0$$
$$\ell_{ik}^{X \cup \{k\}} = \ell_{ik}^{X}$$
$$\ell_{kj}^{X \cup \{k\}} = \ell_{kj}^{X}$$

pentru orice $i, j, k \in V$.

- Calculul valorilor optime rezultă din rezolvarea subproblemelor

$$\mathrm{DM2VD}(\emptyset), \mathrm{DM2VD}(\{0\}), \mathrm{DM2VD}(\{0,1\}), \ldots, \mathrm{DM2VD}(\{0,1,\ldots,n-1\}) =$$
$$\mathrm{DM2VD}(V)$$

- Convenim să notăm $\ell_{ij}^{k}$ în loc de $\ell_{ij}^{\{0,\ldots,k\}}$.
- Pe baza corolarului rezultă că valorile optime pot fi memorate într-un același tablou.
- Maniera de determinare a acestora este asemănătoare cu cea utilizată la determinarea matricei drumurilor de către algoritmul Floyd–Warshall.

# Problema drumurilor minime - modelul matematic (continuare)

- Pe baza ecuațiilor anterioare, proprietatea de substructură optimă se caracterizează prin proprietatea următoare:
  - Un drum optim de la $i$ la $j$ include drumurile optime de la $i$ la $k$ și de la $k$ la $j$, pentru orice vârf intermediar $k$ al său.
- Astfel, drumurile minime din $\mathrm{DM2VD}(X \cup \{k\})$ pot fi determinate utilizând drumurile minime din $\mathrm{DM2VD}(X)$.

# Problema drumurilor minime - modelul matematic (continuare)

- În continuare considerăvem numai cazurile $X = \{0, 1, \ldots, k-1\}$ și $X \cup \{k\} = \{0, 1, \ldots, k-1, k\}$
- Determinarea drumurilor optime poate fi efectuată cu ajutorul unor matrice $P^k = (P^k_{ij})$, care au semnificația următoare: $P^k_{ij}$ este penultimul vârf din drumul optim de la $i$ la $j$.
- Inițial, avem $P^{init}_{ij} = i$, dacă $\langle i, j \rangle \in E$ și $P^{init}_{ij} = -1$, în celelalte cazuri.
- Decizia $k$ determină matricele $\ell^k = (\ell^k_{ij})$ și $P^k = (P^k_{ij})$.
  - Dacă $\ell^{k-1}_{ik} + \ell^{k-1}_{kj} < \ell^{k-1}_{ij}$, atunci drumul optim de la $i$ la $j$ este format din concatenarea drumului optim de la $i$ la $k$ cu drumul optim de la $k$ la $j$ și penultimul vârf din drumul de la $i$ la $j$ coincide cu penultimul vârf din drumul de la $k$ la $j$: $P^k_{ij} = P^{k-1}_{kj}$.
  - În caz contrar, avem $P^k_{ij} = P^{k-1}_{ij}$.
- Cu ajutorul matricei $P^{n-1}_{ij}$ pot fi determinate drumurile optime: ultimul vârf pe drumul de la $i$ la $j$ este $j_t = j$, penultimul vârf este $j_{t-1} = P^{n-1}_{ij_t}$, antipenultimul este $j_{t-2} = P^{n-1}_{ij_{t-1}}$ ș.a.m.d.
- În acest mod, toate drumurile pot fi memorate utilizând numai $O(n^2)$ spațiu.

# Algoritmul Floyd-Warshall - pseudocod

```
procedure Floyd-Warshall(G, ℓ, P)
    for i ← 0 to n-1 do
        for j ← 0 to n-1 do
```

$$\ell_{ij}^{init} = \begin{cases} 0 & ,i = j \\ \ell_{ij} & ,\langle i,j\rangle \in A \\ \infty & ,\text{altfel} \end{cases}$$

$$P_{ij}^{init} = \begin{cases} i & ,i \neq j, \langle i,j\rangle \in A \\ -1 & ,\text{altfel} \end{cases}$$

```
    for i ← 0 to n-1 do
        for j ← 0 to n-1 do
```

$$\ell_{ij}^{0} = \min\{\ell_{ij}^{init}, \ell_{i0}^{init} + \ell_{0j}^{init}\}$$

$$P_{ij}^{0} = \begin{cases} P_{ij}^{init} & ,\ell_{ij}^{0} = \ell_{ij}^{init} \\ P_{0j}^{init} & ,\ell_{ij}^{0} = \ell_{i0}^{init} + \ell_{0j}^{init} \end{cases}$$

```
    for k ← 1 to n-1 do
        for i ← 0 to n-1 do
            for j ← 0 to n-1 do
```

$$\ell_{ij}^{k} = \min\{\ell_{ij}^{k-1}, \ell_{ik}^{k-1} + \ell_{kj}^{k-1}\}$$

$$P_{ij}^{k} = \begin{cases} P_{ij}^{k-1} & ,\ell_{ij}^{k} = \ell_{ij}^{k-1} \\ P_{kj}^{k-1} & ,\ell_{ij}^{k} = \ell_{ik}^{k-1} + \ell_{kj}^{k-1} \end{cases}$$

```
end
```

# Algoritmul Floyd-Warshall - implementare (descriere)

- Presupunem că digraful $G = (V, A)$ este reprezentat prin matricea de ponderilor (lungimilor) arcelor, pe care convenim să o notăm aici cu `G.L` (este ușor de văzut că matricea ponderilor include și reprezentarea lui $A$).
- Datorită corolarului (1), matricele $\ell^k$ și $\ell^{k-1}$ pot fi memorate de același tablou bidimensional `G.L`.
- Simbolul $\infty$ este reprezentat de o constantă `plusInf` cu valoare foarte mare.
- Dacă digraful are circuite negative, atunci acest lucru poate fi depistat:
  - Dacă la un moment dat se obține $G.L[i, i] < 0$, pentru un $i$ oarecare, atunci există un circuit de lungime negativă care trece prin $i$.
- Funcția `Floyd-Warshall` întoarce valoarea *true* dacă digraful ponderat reprezentat de matricea `G.L` nu are circuite negative:
  - `G.L` conține la ieșire ponderile (lungimile) drumurilor minime între oricare două vârfuri;
  - `G.P` conține la ieșire reprezentarea drumurilor minime.

# Algoritmul Floyd-Warshall - implementare (pseudocod)

```
procedure Floyd-Warshall(G, P)
   for i ← 0 to n-1 do
       for j ← 0 to n-1 do
           if ((i ≠ j) and (L[i,j] ≠ plusInf))
              then P[i,j] ← i
              else P[i,j] ← -1
   for k ← 0 to n-1 do
       for i ← 0 to n-1 do
           for j ← 1 to n do
               if ((L[i,k] = PlusInf) or (L[k,j] = PlusInf))
                  then temp ← plusInf
                  else temp ← L[i,k]+L[k,j]
               if (temp < L[i,j])
                  then L[i,j] ← temp
                       P[i,j] ← P[k,j]
               if ((i = j) and (L[i,j] < 0))
                  then throw '(di)graful are circuite negative'
end
```

# Algoritmul Floyd-Warshall - evaluare

Se verifică ușor că execuția algoritmului `Floyd-Warshall` necesită $O(n^3)$ timp și utilizează $O(n^2)$ spațiu.

# Sarcini de lucru și barem de notare

**Sarcini de lucru**:

1. Scrieți o funcție C/C++ care implementează un algoritmul Floyd-Warshall.
2. Dat fiind un graf $G = (V, E)$, scrieți un program care să afișeze drumurile minime între oricare două vârfuri

**Barem de notare**:

1. Implementarea algoritmului Floyd-Warshall: 7p
2. Afișarea drumurilor minime între oricare două vârfuri: 2p
3. Baza: 1p

# Bibliografie

 Lucanu, D. și Craus, M., *Proiectarea algoritmilor*, Editura Polirom, 2008.

# Proiectarea algoritmilor

## Paradigma programării dinamice

## Lucrare de laborator nr. 12

# Cuprins

# Problema rucsacului, varianta discretă - descrierea problemei

- Se consideră un rucsac de capacitate $M \in \mathbb{Z}_+$ și $n$ obiecte $1, \ldots, n$ de dimensiuni (greutăți) $w_1, \ldots, w_n \in \mathbb{Z}_+$.
- Un obiect $i$ este introdus în totalitate în rucsac, $x_i = 1$, sau nu este introdus deloc, $x_i = 0$, astfel că o umplere a rucsacului constă dintr-o secvență $x_1, \ldots, x_n$ cu $x_i \in \{0,1\}$ și $\sum_{i=1}^{n} x_i \cdot w_i \leq M$.
- Introducerea obiectului $i$ în rucsac aduce profitul $p_i \in \mathbb{Z}$, iar profitul total este $\sum_{i=1}^{n} x_i p_i$.
- *Problema constă în a determina o alegere* $(x_1, \ldots, x_n)$ *care să aducă un profit maxim.*
- Singura deosebire față de varianta continuă studiată la metoda greedy constă în condiția $x_i \in \{0,1\}$, în loc de $x_i \in [0,1]$.

# Problema rucsacului, varianta discretă - model matematic

Problema inițială (starea $\mathrm{RUCSAC}(n, M)$):

- Funcția obiectiv:

$$\max \sum_{i=1}^{n} x_i \cdot p_i$$

- Restricții:

$$\sum_{i=1}^{n} x_i \cdot w_i \leq M$$
$$x_i \in \{0,1\}, i = 1, \ldots, n$$
$$w_i \in \mathbb{Z}_+, p_i \in \mathbb{Z}, i = 1, \ldots, n$$
$$M \in \mathbb{Z}_+$$

# Problema rucsacului, varianta discretă - model matematic (continuare)

Generalizarea problemei inițiale (starea $\mathrm{RUCSAC}(j, X)$):

- Funcția obiectiv:

$$\max \sum_{i=1}^{j} x_i \cdot p_i$$

- Restricții:

$$\sum_{i=1}^{j} x_i \cdot w_i \leq X$$
$$x_i \in \{0,1\}, i = 1, \ldots, j$$
$$w_i \in \mathbb{Z}_+, p_i \in \mathbb{Z}, i = 1, \ldots, j$$
$$X \in \mathbb{Z}_+$$

# Problema rucsacului, varianta discretă - model matematic (continuare)

- Notăm cu $f_j(X)$ valoarea optimă pentru instanța $\text{RUCSAC}(j, X)$.
- Dacă $j = 0$ și $X \geq 0$, atunci $f_j(X) = 0$.
- Presupunem $j > 0$. Notăm cu $(x_1, \ldots, x_j)$ alegerea care dă valoarea optimă $f_j(X)$.
  - Dacă $x_j = 0$ (obiectul $j$ nu este pus în rucsac), atunci, conform principiului de optim, $f_j(X)$ este valoarea optimă pentru starea $\text{RUCSAC}(j-1, X)$ și de aici $f_j(X) = f_{j-1}(X)$.
  - Dacă $x_j = 1$ (obiectul $j$ este pus în rucsac), atunci, din nou conform principiului de optim, $f_j(X)$ este valoarea optimă pentru starea $\text{RUCSAC}(j-1, X - w_j)$ plus $p_j$ și, de aici, $f_j(X) = f_{j-1}(X - w_j) + p_j$.
- Combinând relațiile de mai sus obținem:

$$f_j(X) = \begin{cases} -\infty, & \text{dacă } X < 0 \\ 0, & \text{dacă } j = 0 \text{ şi } X \geq 0 \\ \max\{f_{j-1}(X), f_{j-1}(X - w_j) + p_j\}, & \text{dacă } j > 0 \text{ şi } X \geq 0 \end{cases} \quad (1)$$

- Am considerat $f_j(X) = -\infty$, dacă $X < 0$.

# Problema rucsacului, varianta discretă - model matematic (continuare)

- Din relația (1) rezultă că proprietatea de substructură optimă se caracterizează astfel:
  - Soluția optimă $(x_1, \ldots, x_j)$ a problemei $\text{RUCSAC}(j, X)$ include soluția optimă $(x_1, \ldots, x_{j-1})$ a subproblemei $\text{RUCSAC}(j-1, X - x_j w_j)$.
- Soluția optimă pentru $\text{RUCSAC}(j, X)$ se poate obține utilizând soluțiile optime pentru subproblemele $\text{RUCSAC}(i, Y)$ cu $1 \leq i < j, 0 \leq Y \leq X$.
- Relația (1) implică o recursie în cascadă și deci numărul de subprobleme de rezolvat este $O(2^n)$, fapt pentru care calculul și memorarea eficientă a valorilor optime pentru subprobleme devine un task foarte important.

# Problema rucsacului, varianta discretă - exemplu

- Fie $M = 10, n = 3$ și greutățile și profiturile date de următorul tabel:

| $i$ | 1 | 2 | 3 |
|---|---|---|---|
| $w_i$ | 3 | 5 | 6 |
| $p_i$ | 10 | 30 | 20 |

- Valorile optime pentru subprobleme sunt calculate cu ajutorul relației (1)≡(2)

$$f_j(X) = \begin{cases} -\infty, & \text{dacă } X < 0 \\ 0, & \text{dacă } j = 0 \text{ şi } X \geq 0 \\ \max\{f_{j-1}(X), f_{j-1}(X - w_j) + p_j\}, & \text{dacă } j > 0 \text{ şi } X \geq 0 \end{cases} \quad (2)$$

- Valorile optime pot fi memorate într-un tablou bidimensional astfel:

| $X$ | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| $f_0$ | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| $f_1$ | 0 | 0 | 0 | 10 | 10 | 10 | 10 | 10 | 10 | 10 | 10 |
| $f_2$ | 0 | 0 | 0 | 10 | 10 | 30 | 30 | 30 | 40 | 40 | 40 |
| $f_3$ | 0 | 0 | 0 | 10 | 10 | 30 | 30 | 30 | 40 | 40 | 40 |

- Tabloul de mai sus este calculat linie cu linie.
  - Pentru a calcula valorile de pe o linie sunt consultate numai valorile de pe linia precedentă.
  - Exemplu: $f_2(8) = \max\{f_1(8), f_1(8-5)+30\} = \max\{10, 40\} = 40$.

# Problema rucsacului, varianta discretă - exemplu

- Tabloul valorilor optime are dimensiunea $n \cdot M$ (au fost ignorate prima linie și prima coloană).
- Dacă $M = O(2^n)$ rezultă că atât complexitatea spațiu, cât și cea timp sunt exponențiale.
- Privind tabloul de mai sus observăm că există multe valori care se repetă.
- *Cum putem memora mai compact tabloul valorilor optime*?
- *Soluție*: Construim graficele funcțiilor $f_0, f_1, f_2 \cdots$

# Problema rucsacului, varianta discretă - exemplu

$$f_0(X) = \begin{cases} -\infty & ,X < 0 \\ 0 & ,X \geq 0 \end{cases}$$

$$g_0(X) = f_0(X - w_1) + p_1 = \begin{cases} -\infty & ,X < 3 \\ 10 & ,3 \leq X \end{cases}$$

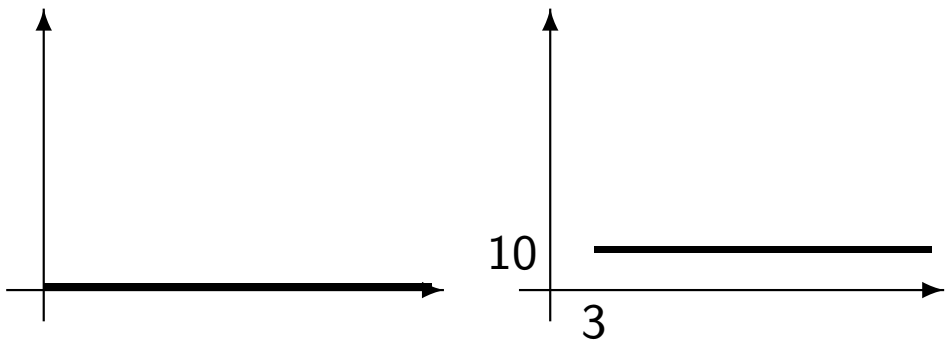



Figura 1: Funcțiile $f_0$ și $g_0$

# Problema rucsacului, varianta discretă - exemplu

$$f_1(X) = \max\{f_0(X), g_0(X)\} = \begin{cases} -\infty & , X < 0 \\ 0 & , 0 \leq X < 3 \\ 10 & , 3 \leq X \end{cases}$$

$$g_1(X) = f_1(X - w_2) + p_2 = \begin{cases} -\infty & , X < 5 \\ 30 & , 5 \leq X < 8 \\ 40 & , 8 \leq X \end{cases}$$

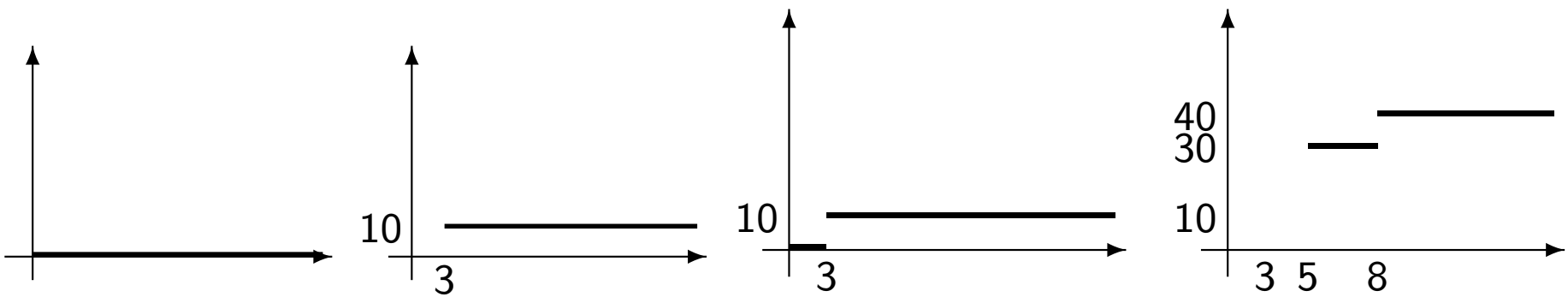



Figura 2: Funcțiile $f_0$ și $g_0$; Funcțiile $f_1$ și $g_1$

# Problema rucsacului, varianta discretă - exemplu

$$f_2(X) = \max\{f_1(X), g_1(X)\} = \begin{cases} -\infty & , X < 0 \\ 0 & , 0 \leq X < 3 \\ 10 & , 3 \leq X < 5 \\ 30 & , 5 \leq X < 8 \\ 40 & , 8 \leq X \end{cases}$$

$$g_2(X) = f_2(X - w_3) + p_3 = \begin{cases} -\infty & , X < 6 \\ 20 & , 6 \leq X < 9 \\ 30 & , 9 \leq X < 11 \\ 50 & , 11 \leq X < 14 \\ 60 & , 14 \leq X \end{cases}$$

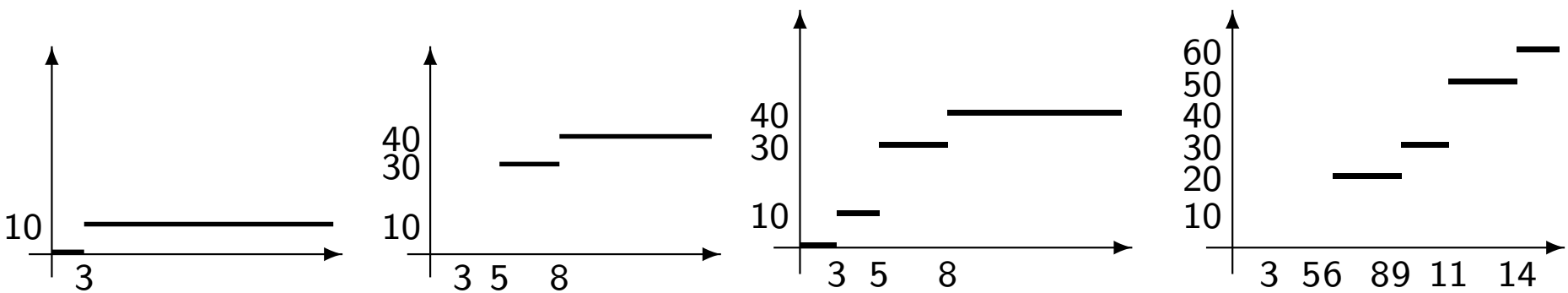



Figura 3: Funcțiile $f_1$ și $g_1$; Funcțiile $f_2$ și $g_2$

# Problema rucsacului, varianta discretă - exemplu

$$f_3(X) = \max\{f_2(X), g_2(X)\} = \begin{cases} -\infty & , X < 0 \\ 0 & , 0 \leq X < 3 \\ 10 & , 3 \leq X < 5 \\ 30 & , 5 \leq X < 8 \\ 40 & , 8 < X \leq 11 \\ 50 & , 11 \leq X < 14 \\ 60 & , 14 \leq X \end{cases}$$

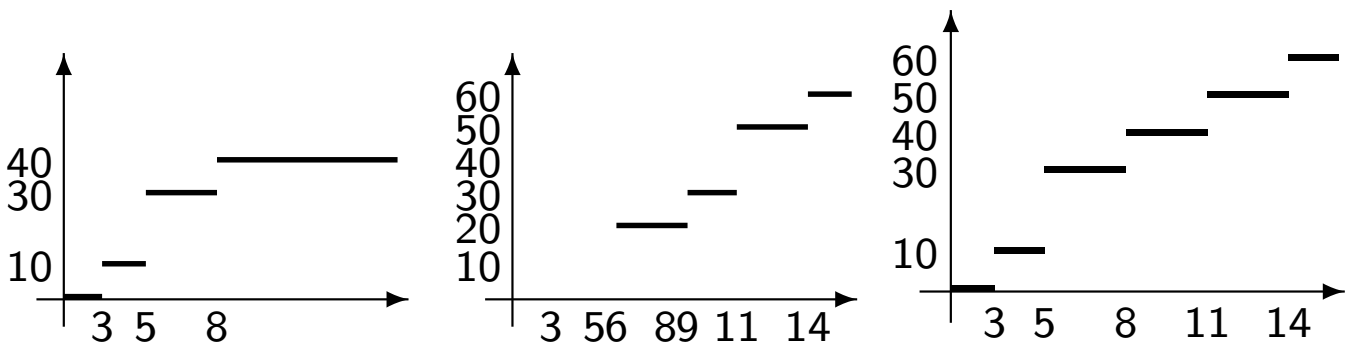



Figura 4: Funcțiile $f_2$ și $g_2$; Funcția $f_3$

# Problema rucsacului, varianta discretă - exemplu (comentarii)

- Se remarcă faptul că funcțiile $f_i$ și $g_i$ sunt funcții în scară. Graficele acestor funcții pot fi reprezentate prin mulțimi finite din puncte din plan.
  - De exemplu, graficul funcției $f_2$ este reprezentat prin mulțimea $\{(0,0),(3,10),(5,30),(8,40)\}$.
- O mulțime care reprezintă o funcție în scară conține acele puncte în care funcția face salturi.
- Graficul funcției $g_i$ se obține din graficul funcției $f_i$ printr-o translație.
- Graficul funcției $f_{i+1}$ se obține prin interclasarea graficelor funcțiilor $f_i$ și $g_i$.

# Problema rucsacului, varianta discretă - algoritm (descriere)

- În general, fiecare $f_i$ este complet specificat de o mulțime $S_i = \{(X_j, Y_j) \mid j = 0, \ldots, r\}$, unde $Y_j = f_i(X_j)$.
  - Presupunem $X_1 < \cdots < X_r$.
- Analog, funcțiile $g_i$ sunt reprezentate prin mulțimile $T_i = \{(X + w_{i+1}, Y + p_{i+1}) \mid (X, Y) \in S_i\}$.
  - Notăm $T_i = \tau(S_i)$ și $S_{i+1} = \mu(S_i, T_i)$.
- Mulțimea $S_{i+1}$ se obține din $S_i$ și $T_i$ prin interclasare.
  - Operația de interclasare se realizează într-un mod asemănător cu cel de la interclasarea a două linii ale orizontului.
- Se consideră o variabilă $L$ care ia valoarea 1 dacă graficul lui $f_{i+1}$ coincide cu cel al lui $f_i$ și cu 2 dacă el coincide cu cel al lui $g_i$.
  - Deoarece $(0,0)$ aparține graficului rezultat, considerăm $L = 1, j = 1$ și $k = 1$.

# Problema rucsacului, varianta discretă - algoritm de interclasare grafice (descriere)

Presupunând că la un pas al interclasării se compară $(X_j, Y_j) \in S_i$ cu $(X_k, Y_k) \in T_i$, atunci:

- dacă $L = 1$:
  - dacă $X_j < X_k$, atunci se adaugă $(X_j, Y_j)$ în $S_{i+1}$ și se incrementează $j$;
  - dacă $X_j = X_k$:
    - dacă $Y_j \geq Y_k$, atunci se adaugă $(X_j, Y_j)$ în $S_{i+1}$ și se incrementează $j$ și $k$;
    - dacă $Y_j < Y_k$, atunci se adaugă $(X_k, Y_k)$ în $S_{i+1}$, $L = 2$ și se incrementează $j$ și $k$;
  - dacă $X_j > X_k$ sau $j > |S_i|$:
    - dacă $Y_{j-1} \geq Y_k$, atunci se incrementează $k$;
    - dacă $Y_{j-1} < Y_k$, atunci $L = 2$;
- dacă $L = 2$:
  - dacă $X_k < X_j$, atunci se adaugă $(X_k, Y_k)$ în $S_{i+1}$ și se incrementează $k$;
  - dacă $X_k = X_j$:
    - dacă $Y_k \geq Y_j$, atunci se adaugă $(X_k, Y_k)$ în $S_{i+1}$ și se incrementează $j$ și $k$;
    - dacă $Y_k < Y_j$, atunci se adaugă $(X_j, Y_j)$ în $S_{i+1}$, $L = 1$ și se incrementează $j$ și $k$;
  - dacă $X_k > X_j$ sau $k > |T_i|$:
    - dacă $Y_{k-1} \geq Y_j$, atunci se incrementează $j$;
    - dacă $Y_{k-1} < Y_j$, atunci $L = 1$;

Dacă se termină mulțimea $S_i$, atunci se adauga la $S_{i+1}$ restul din $T_i$.
Dacă se termină mulțimea $T_i$, atunci se adauga la $S_{i+1}$ restul din $S_i$.
Notăm cu `interclGrafice(S_i,T_i)` funcția care determină $S_{i+1}$ conform algoritmului de mai sus.

# Problema rucsacului, varianta discretă - algoritm de extragere a solutiei (exemplu)

- $S_3 = \{(0,0),(3,10),(5,30),(8,40),(11,50),(14,60)\}$.
- $S_2 = \{(0,0),(3,10),(5,30),(8,40)\}$.
- $S_1 = \{(0,0),(3,10)\}$.
- $S_0 = \{(0,0)\}$.
- Se caută în $S_n = S_3$ perechea $(X_j, Y_j)$ cu cel mai mare $X_j$ pentru care $X_j \leq M$. Obținem $(X_j, Y_j) = (8,40)$. Deoarece $(8,40) \in S_3$ și $(8,40) \in S_2$ rezultă $f_{optim}(M) = f_{optim}(8) = f_3(8) = f_2(8)$ și deci $x_3 = 0$. Perechea $(X_j, Y_j)$ rămâne neschimbată.
- Pentru că $(X_j, Y_j) = (8,40)$ este în $S_2$ și nu este în $S_1$, rezultă că $f_{optim}(8) = f_1(8 - w_2) + p_2$ și deci $x_2 = 1$. În continuare se ia $(X_j, Y_j) = (X_j - w_2, Y_j - p_2) = (8-5, 40-30) = (3,10)$.
- Pentru că $(X_j, Y_j) = (3,10)$ este în $S_1$ și nu este în $S_0$, rezultă că $f_{optim}(3) = f_1(3 - w_1) + p_1$ și deci $x_1 = 1$.

# Problema rucsacului, varianta discretă - algoritm de extragere a solutiei (descriere)

- Inițial se determină perechea $(X_j, Y_j) \in S_n$ cu cel mai mare $X_j$ pentru care $X_j \leq M$. Valoarea $Y_j$ constituie încărcarea optimă a rucsacului, *i.e.*, valoarea funcției obiectiv din problema inițială.
- Pentru $i = n-1, \ldots, 0$:
  - dacă $(X_j, Y_j)$ este în $S_i$, atunci $f_{i+1}(X_j) = f_i(X_j) = Y_j$ și se consideră $x_{i+1} = 0$ (obiectul $i+1$ nu este ales);
  - dacă $(X_j, Y_j)$ nu este în $S_i$, atunci $f_{i+1}(X_j) = f_i(X_j - w_{i+1}) + p_{i+1} = Y_j$ și se consideră $x_{i+1} = 1$ (obiectul $i+1$ este ales), $X_j = X_j - w_{i+1}$ și $Y_j = Y_j - p_{i+1}$.

# Problema rucsacului, varianta discretă - algoritm (pseudocod)

```
procedure rucsac_II(M, n, w, p, x)
    S_0 ← {(0,0)}
    T_0 ← {(w_1,p_1)}
    for i ← 1 to n
        S_i ← interclGrafice(S_{i-1},T_{i-1})
        T_i ← {(X+w_{i+1},Y+p_{i+1}) | (X,Y) ∈ S_i}
    determină (X_j,Y_j) cu X_j = max{X_i | (X_i,Y_i) ∈ S_n,X_i ≤ M}
    for i ← n-1 downto 0 do
        if (X_j,Y_j) ∈ S_i
           then x_{i+1} ← 0
           else x_{i+1} ← 1
                X_j ← X_j − w_{i+1}
                Y_j ← Y_j − p_{i+1}
end
```

# Sarcini de lucru și barem de notare

**Sarcini de lucru**:

1. Scrieți o funcție C/C++ care implementează algoritmul `rucsac`.
2. Se consideră un rucsac de capacitate $M \in \mathbb{Z}_+$ și $n$ obiecte $1, \ldots, n$ de dimensiuni (greutăți) $w_1, \ldots, w_n \in \mathbb{Z}_+$.. Scrieți un program care să afișeze soluția optimă.

**Barem de notare**:

1. Implementarea algoritmului `rucsac`: 7p
2. Afișarea soluției optime: 2p
3. Baza: 1p

# Bibliografie



Lucanu, D. și Craus, M., *Proiectarea algoritmilor*, Editura Polirom, 2008.


R.E. Bellman și S.E. Dreyfus, *Applied Dynamic Programming*, Princeton University Press, 1962.


Moret, B.M.E.și Shapiro, H.D. , *Algorithms from P to NP: Design and Efficiency*, The Benjamin/Cummings Publishing Company, Inc., 1991.

# Proiectarea algoritmilor

## Paradigma Backtracking

## Lucrare de laborator nr. 13

# Cuprins

# Problema submulțimilor de sumă dată - descriere

- Se consideră o mulțime $A$ cu $n$ elemente, fiecare element $a \in A$ având o dimensiune $s(a) \in \mathbb{Z}_+$ și un număr întreg pozitiv $M$.
- Problema constă în determinarea tuturor submulțimilor $A' \subseteq A$ cu proprietatea $\sum_{a \in A'} s(a) = M$.

# Problema submulțimilor de sumă dată - modelul matematic

- Presupunem $A = \{1, \ldots, n\}$ și $s(i) = w_i, 1 \leq i \leq n$.
- Pentru reprezentarea soluțiilor avem două posibilități.
  1. Prin vectori care să conțină elementele care compun soluția.
     - Această reprezentare are dezavantajul că trebuie utilizat un algoritm de enumerare a vectorilor de lungime variabilă.
     - De asemenea, testarea condiției $a \in A \setminus A'$? nu mai poate fi realizată în timpul $O(1)$ dacă nu se utilizează spațiu de memorie suplimentar.
  2. Prin vectori de lungime $n$, $(x_1, \ldots, x_n)$ cu $x_i \in \{0,1\}$ având semnificația: $x_i = 1$ dacă și numai dacă $w_i$ aparține soluției (vectorii caracteristici).
- *Exemplu:* Fie $n = 4$, $(w_1, w_2, w_3, w_4) = (4, 7, 11, 14)$ și $M = 25$. Soluțiile sunt
  - $(4, 7, 14)$ care mai poate fi reprezentată prin $(1, 2, 4)$ sau $(1, 1, 0, 1)$ și
  - $(11, 14)$ care mai poate fi reprezentată prin $(3, 4)$ sau $(0, 0, 1, 1)$.
- Vom opta pentru ultima variantă, deoarece vectorii au lungime fixă.

# Problema submulțimilor de sumă dată - modelul matematic (continuare)

- Remarcăm faptul că spațiul soluțiilor conține $2^n$ posibilități (elementele mulțimii $\{0,1\}^n$) și poate fi reprezentat printr-un arbore binar.
- În procesul de generare a soluțiilor potențiale, mulțimea $A$ este partiționată astfel:
  - o parte $\{1,\ldots,k\}$ care a fost luată în considerare pentru a stabili candidații la soluție și
  - a doua parte $\{k+1,\ldots,n\}$ ce urmează a fi luată în considerare.
- Cele două părți trebuie să satisfacă următoarele două inegalități:
  - suma parțială dată de prima parte (adică de candidații aleși) să nu depășească $M$:

$$\sum_{i=1}^{k} x_i \cdot w_i \leq M \tag{1}$$

  - ceea ce rămâne să fie suficient pentru a forma suma $M$:

$$\sum_{i=1}^{k} x_i \cdot w_i + \sum_{i=k+1}^{n} w_i \geq M \tag{2}$$

- Cele două inegalități pot constitui criteriul de mărginire.
- Cu acest criteriu de tăiere, arborele parțial rezultat pentru exemplul anterior este cel reprezentat în figura 1.

# Problema submulțimilor de sumă dată - modelul matematic (continuare)

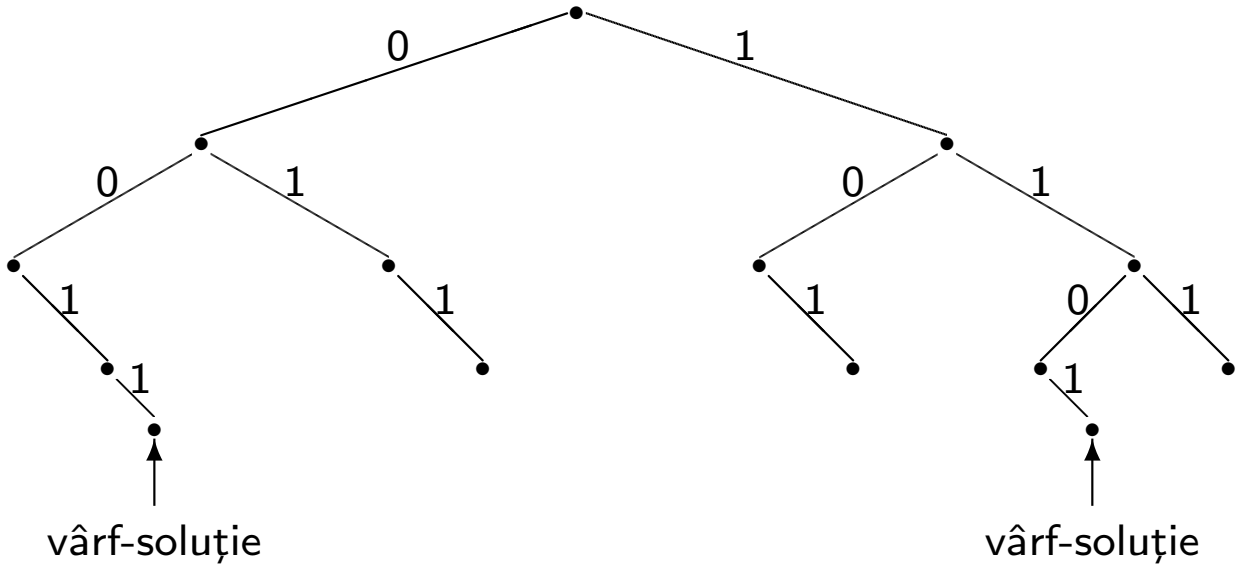



Figura 1 : Arbore parțial pentru submulțimi de sumă dată

- Criteriul de mărginire nu elimină toți subarborii care nu conțin vârfuri-soluție, dar elimină foarte mulți, restrângând astfel spațiul de căutare.
- Atingerea unui vârf pe frontieră presupune imediat determinarea unei soluții: suma $\sum_{i=k+1}^{n} w_i$ este zero (deoarece $k = n$) și dubla inegalitate dată de relațiile 1 și 2 implică $\sum_{i=1}^{n} w_i = M$.
- *Observație*: Dacă se utilizează drept criteriu de mărginire numai inegalitatea 1, atunci atingerea unui vârf pe frontieră în arborele parțial nu presupune neapărat și obținerea unei soluții. Mai trebuie verificat dacă suma submulțimii alese este exact $M$.

# Problema submulțimilor de sumă dată - model matematic îmbunătățit

- Se consideră $w_1, w_2, ..., w_n$ în ordine crescătoare (fără a restrânge generalitatea);
- Pentru cazul în care suma parțială dată de candidații aleși este strict mai mică decât $M$ ($\sum_{i=1}^{k} x_i w_i < M$), se introduce un criteriu de mărginire suplimentar:

$$\sum_{i=1}^{k} x_i w_i + w_{k+1} \leq M.$$

- Presupunem valorile $x_1, ..., x_{k-1}$ calculate.
- Notăm cu $s$ suma parțială corespunzătoare valorilor $x_1, ..., x_{k-1}$ ($s = \sum_{i=1}^{k-1} x_i w_i$) și cu $r$ suma $\sum_{i=k}^{n} w_i$.
- Presupunem $w_1 \leq M$ și $\sum_{i=1}^{n} w_i \geq M$.

# Problema submulțimilor de sumă dată - algoritm

```
procedure submultimiOpt(s,k,r)
    x_k ← 1
    if (s+w_k=M)
      then scrie(x^k) /* x^k=(x_1,x_2,...,x_k)) */
      else if (s+w_k+w_{k+1} ≤ M)
              then submultimiOpt(s+w_k,k+1,r-w_k)
           if ((s+r-w_k ≥ M) and (s+w_{k+1} ≤ M))
              then x_k ← 0
              submultimiOpt(s,k+1,r-w_k)
end
```

- *Precondiții*: $w_1 \leq M$ și $\sum_{i=1}^{n} w_i \geq M$. Astfel, înainte de apelul inițial sunt asigurate condițiile $s + w_k \leq M$ și $s + r \geq M$. Apelul inițial este `submultimiOpt`$\left(0, 1, \sum_{i=1}^{n} w_i\right)$.
- Condițiile $s + w_k \leq M$ și $s + r \geq M$ sunt asigurate și la apelul recursiv.
- Înainte de apelul recursiv `submultimiOpt(s+w_k,k+1,r-w_k)` nu este nevoie să se mai verifice dacă $\sum_{i=1}^{k} x_i w_i + \sum_{i=k+1}^{n} w_i \geq M$, deoarece $s + r > M$ și $x_k = 1$.
- Nu se verifică explicit nici $k > n$.
  - Inițial, $s = 0 < M$ și $s + r \geq M$ și $k = 1$.
  - De asemenea, în linia „`if` $(s + w_k + w_{k+1} \leq M)$”, deoarece $s + w_k < M$, rezultă $r \neq w_k$, deci $k + 1 \leq n$.

# Sarcini de lucru și barem de notare

Sarcini de lucru:

1. Scrieți o funcție C/C++ care implementează algoritmul `submultimiOpt`.
2. Se consideră o mulțimea $A = \{1, \ldots, n\}$ și un număr întreg pozitiv $M$. Fiecare element $i \in A$ are o dimensiune $w_i \in \mathbb{Z}_+$. Scrieți un program care să afișeze submulțimile $A' \subseteq A$ cu proprietatea $\sum_{i \in A'} w_i = M$.

Barem de notare:

1. Implementarea algoritmului `submultimiOpt`: 7p
2. Afișarea soluției: 2p
3. Baza: 1p

# Bibliografie

 Lucanu, D. și Craus, M., *Proiectarea algoritmilor*, Editura Polirom, 2008.

# Proiectarea algoritmilor

**Lucrare de laborator nr. 14**

**Verificarea temelor de casa**