Proiectarea Algoritmilor

Mitică Craus

Universitatea Tehnică "Gheorghe Asachi" din Iași

Cuprins

Ce se doreste?

Obiectivele cursului

- Formarea unui stil științific de analiză și modelare a unei probleme;
- Construirea de algoritmi corecți și eficienți;
- Adaptarea algoritmilor generici la aplicații.

Rezultatele învățării

- Cunoașterea metodelor de proiectare a algoritmilor: modelul matematic, implementarea, analiza;
- Cunoașterea problemelor generice care pot fi rezolvate printr-o paradigmă de proiectare specificată;
- Abilitatea de a aplica algoritmul adecvat unei probleme;
- Capacitatea de a proiecta, implementa și testa algoritmi;
- Abilități de programare.

Sistemul de evaluare

Evaluarea continuă:

- Activitatea la laborator (M):
  - Ponderea în nota finală: 30%
  - Criterii de evaluare: rezolvarea temelor propuse și calitatea soluțiilor.
- Teme de casă (CC)
  - Ponderea în nota finală: 20%
  - Evaluarea pe parcursul semestrului a cunoștințelor practice acumulate la orele de aplicații.

Evaluarea finală: examen

- Ponderea în nota finală: 50%
- Proba 1:
  - categoria de sarcini: test de cunoștințe cu întrebări deschise;
  - condițiile de lucru: fără mijloace de informare accesibile studentului în timpul probei;
  - ponderea în nota examenului: 50%
- Proba 2:
  - categoria de sarcini: rezolvare de probleme;
  - condițiile de lucru: fără mijloace de informare accesibile studentului în timpul probei;
  - ponderea în nota examenului: 50%

Lecții

- Limbaj algoritmic, algoritmi și probleme, evaluarea algoritmilor.
- Metode de proiectare a algoritmilor orientate pe problemă:
  - Problema căutării
  - Problema sortării
- Metode generale de proiectare a algoritmilor:
  - Divide-et-impera (divide and conquer)
  - Greedy
  - Programare dinamică
  - Backtracking
  - Branch and bound
- Algoritmica grafurilor

Lucrări de laborator

- Recapitularea unor noțiuni C/C++ și structuri de date.
- Complexitatea algoritmilor: analiza unor algortmi fundamentali.
- Căutare secvențială și binară, căutarea pe arbori binari de căutare: programe C/C++; experimente; evaluare.
- Cautarea pe arbori B: programe C/C++ de inserare și căutare a unei chei; experimente; evaluare.,
- Algoritmi de sortare: programe C/C++ pentru algoritmii Buble_Sort, Insertion_Sort și Naiv_Sort; experimente; evaluare.
- Algoritmi de sortare: programme C/C++ pentru algoritmul Radix_Sort și algoritmul de sortare topologică; experimente; evaluare.
- Metoda divide-et-impera: implementarea în C/C++ algoritmului Quick_Sort; experimente; evaluare..
- Metoda divide-et-impera: implementarea în C/C++ a algoritmului de inmulțire a două numere întregi mari și a algoritmului lui Strassen.
- Metoda greedy: implementarea în C/C++ a algoritmului de interclasare optimală; aplicații.
- Metoda greedy: implementarea în C/C++ a algoritmului comprimare de date prin metoda arborilor Huffman; aplicații.
- Programare dinamică: implementarea în C/C++ a algoritmului Floyd-Warshall.
- Programare dinamică: implementarea în C/C++ a algoritmului de rezolvare a problemei rucsacului, varianta discretă.
- Backtracking: implementarea în C/C++ a algoritmului pentru rezolvarea problemei submulțimilor de sumă dată.

Profesori

Curs

- Mitică Craus

Laborator

- Mitică Craus
- Cristian Nicolae Buțincu
- Adrian Alexandrescu

Bibliografie selectivă

- D. Lucanu, M. Craus, (2008), Proiectarea algoritmilor, Editura Polirom
- T. Cormen, C. Leiserson, R. Rivest, (2000), Introducere în algoritmi, Computer Libris Agora, Cluj
- C.Croitou, (1992), Tehnici de bază în optimizarea combinatorie, Editura Universității "Al. I. Cuza", Iași

Proiectarea algoritmilor

Algorimi, probleme, performanțe

Mitică Craus

Univeristatea Tehnică "Gheorghe Asachi" din Iași

Cuprins

Algoritmi

- Un *algoritm* este o secvență finită de operații (pași) care, atunci când este executată, produce o soluție corectă pentru o problemă precizată.
- Tipul operațiilor și ordinea lor în secvență respectă o logică specifică.
- Algoritmii pot fi descriși în orice limbaj, pornind de la limbajul natural până la limbajul nativ al unui calculator specific.
- Un limbaj al cărui scop unic este cel de a descrie algoritmi se numește *limbaj algoritmic*.
- Limbajele de programare sunt exemple de limbaje algoritmice.

Tipuri de date

- Un *tip de date* constă dintr-o mulțime de entități (componente) de tip dată (informație reprezentabilă în memoria unui calculator), numită și *domeniul* tipului, și o mulțime de operații peste aceste entități.
- Convenim să grupăm tipurile de date în trei categorii:
  - *tipuri de date elementare*, în care entitățile sunt indivizibile;
  - *tipuri de date structurate de nivel jos*, în care entitățile sunt structuri relativ simple obținute prin asamblarea de date elementare sau date structurate, iar operațiile sunt definite la nivel de componentă;
  - *tipuri de date structurate de nivel înalt*, în care componentele sunt structuri mai complexe, iar operațiile sunt implementate de algoritmi proiectați de către utilizatori.
- Primele două categorii sunt dependente de limbaj.
- Tipurile de nivel înalt pot fi descrise într-o manieră independentă de limbaj.
- Un tip de date descris într-o manieră independentă de reprezentarea valorilor și implementarea operațiilor se numește *tip de date abstract*.

Programe

- Pașii unui algoritm și ordinea logică a acestora sunt descrise cu ajutorul *instrucțiunilor*.
- O secvență de instrucțiuni care acționează asupra unor structuri de date precizate se numește *program*.
- Memoria este reprezentată ca o secvență de celule (locații), fiecare celulă având asociată o *adresă* și putând memora (stoca) o dată de un anumit tip (figura 1).

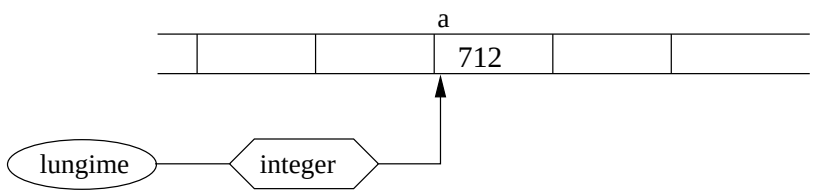


Figura 1 : Memoria

Variabile

- Accesul la memorie este realizat cu ajutorul variabilelor.
- O *variabilă* este caracterizată de:
  - un *nume* cu ajutorul căruia variabila este referită,
  - o *adresă* care desemnează o locație de memorie și
  - un *tip de date* care descrie natura datelor memorate în locația de memorie asociată variabilei.
- Dacă în plus adăugăm și data memorată la un moment dat în locație, atunci obținem o *instanță a variabilei*.
- O variabilă este reprezentată grafic ca în figura 2.a. Atunci când tipul se subînțelege din context, vom utiliza reprezentarea scurtă sugerată în 2.b.
- Convenim să utilizăm fontul `type writer` pentru notarea variabilelor și fontul *mathnormal* pentru notarea valorilor memorate de variabile.

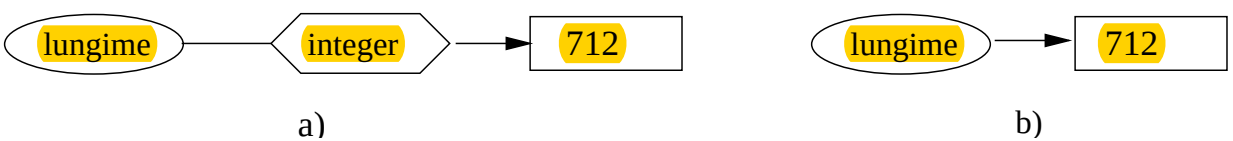


Figura 2 : Variabilă

Instrucțiunea atribuire

- *Sintaxa*:
  - ⟨variabilă⟩ ← ⟨expresie⟩
    unde ⟨variabilă⟩ este numele unei variabile, iar ⟨expresie⟩ este o expresie corect formată, de același tip cu ⟨variabilă⟩.
- *Semantica*:
  1. Se evaluează ⟨expresie⟩ și rezultatul obținut se memorează în locația de memorie desemnată de ⟨variabilă⟩.
  2. Valorile tuturor celorlalte variabile rămân neschimbate.
- Atribuirea este singura instrucțiune cu ajutorul căreia se poate modifică memoria.
- O reprezentare intuitivă a efectului instrucțiunii de atribuire este dată în figura 3.

a) Inainte de atribuire

b) Dupa atribuirea "a <- a*b"

Figura 3 : Atribuirea

Instrucțiunea `if`

- *Sintaxa*:
  - `if` ⟨expresie⟩
    `then` ⟨secvență-instrucțiuni$_1$⟩
    `else` ⟨secvență-instrucțiuni$_2$⟩
    unde ⟨expresie⟩ este o expresie care prin evaluare dă rezultat boolean, iar ⟨secvență-instrucțiuni$_i$⟩, $i = 1, 2$, sunt secvențe de instrucțiuni scrise una sub alta și aliniate corespunzător.
  - Partea `else` este facultativă.
  - Dacă partea `else` lipsește și ⟨secvență-instrucțiuni$_1$⟩ este formată dintr-o singură instrucțiune, atunci instrucțiunea `if` poate fi scrisă și pe un singur rând.
- *Semantica*:
  1. Se evaluează ⟨expresie⟩.
  2. Dacă rezultatul evaluării este `true`, atunci se execută ⟨secvență-instrucțiuni$_1$⟩, după care execuția instrucțiunii `if` se termină;
  3. dacă rezultatul evaluării este `false`, atunci se execută ⟨secvență-instrucțiuni$_2$⟩ după care execuția instrucțiunii `if` se termină.

Instrucțiunea `while`

- *Sintaxa*:
  - `while` ⟨expresie⟩ `do`
    ⟨secvență-instrucțiuni⟩
    unde ⟨expresie⟩ este o expresie care prin evaluare dă rezultat boolean, iar ⟨secvență-instrucțiuni⟩ este o secvență de instrucțiuni scrise una sub alta și aliniate corespunzător.
- *Semantica*:
  1. Se evaluează ⟨expresie⟩.
  2. Dacă rezultatul evaluării este `true`, atunci se execută ⟨secvență-instrucțiuni⟩, după care se reia procesul începând cu pasul 1.
  3. Dacă rezultatul evaluării este `false`, atunci execuția instrucțiunii `while` se termină.

Instrucțiunea `for`

- *Sintaxa*:
  - `for` ⟨variabilă⟩ ← ⟨expresie$_1$⟩ `to` ⟨expresie$_2$⟩ `do`
    ⟨secvență-instrucțiuni⟩

    sau

    `for` ⟨variabilă⟩ ← ⟨expresie$_1$⟩ `downto` ⟨expresie$_2$⟩ `do`
    ⟨secvență-instrucțiuni⟩

    unde ⟨variabilă⟩ este o variabilă de tip întreg, ⟨expresie$_i$⟩, $i = 1, 2$, sunt expresii care prin evaluare dau valori întregi, ⟨secvență-instrucțiuni⟩ este o secvență de instrucțiuni scrise una sub alta și aliniate corespunzător.
- *Semantica*:

Instrucțiunea

```
for i ← e1 to e2 do
    S
```

simulează execuția următorului program:

```
i ← e1
temp ← e2
while (i ≤ temp) do
    S
    i ← succ(i)
    /* succ(i) întoarce
   succesorul numărului întreg i */
```

Instrucțiunea

```
for i ← e1 downto e2 do
    S
```

simulează execuția următorului program:

```
i ← e1
temp ← e2
while (i ≥ temp) do
    S
    i ← pred(i)
    /* pred(i) întoarce
   predecesorul numărului întreg i */
```

Instrucțiunea `for each`

- *Sintaxa*:
  - `for each` ⟨variabilă⟩ $\in$ ⟨mulțime⟩ `do`
    ⟨secvență-instrucțiuni⟩

    unde ⟨variabilă⟩ este o variabilă de tip $\mathscr{T}$, ⟨mulțime⟩ este o mulțime finită de valori din $\mathscr{T}$, ⟨secvență-instrucțiuni⟩ este o secvență de instrucțiuni scrise una sub alta și aliniate corespunzător.
- *Semantica*:
  - Instrucțiunea

    `for each x` $\in \mathscr{T}$ `do`
    S

    simulează execuția următorului program:

    `while (`$\mathscr{T} \neq \emptyset$`) do`
    `x` $\leftarrow$ *un element din* $\mathscr{T}$
    S
    $\mathscr{T} \leftarrow \mathscr{T} \setminus \{$`x`$\}$

Instrucțiunea `repeat`

- *Sintaxa*:
  - `repeat`
    ⟨secvență-instrucțiuni⟩
    `until` ⟨expresie⟩
    unde ⟨expresie⟩ este o expresie care prin evaluare dă rezultat boolean, iar ⟨secvență-instrucțiuni⟩ este o secvență de instrucțiuni scrise una sub alta și aliniate corespunzător.
- *Semantica*:
  - Instrucțiunea

    `repeat`
    S
    `until` *e*

    simulează execuția următorului program:

    S
    `while (not` *e*`) do`
    S

Excepții

- *Sintaxa*:
  - `throw` ⟨mesaj⟩
    unde ⟨mesaj⟩ este un șir de caractere (un text).
- *Semantica*:
  - Execuția programului se oprește și este afișat textul ⟨mesaj⟩. Cel mai adesea, `throw` este utilizată împreună cu `if`:

    `if` ⟨expresie⟩ `then throw` ⟨mesaj⟩
  - Obținerea rezultatului `true` în urma evaluării expresiei are ca semnificație apariția unei excepții, caz în care execuția programului se oprește.
  - Un exemplu de excepție este cel când procedura `new` nu poate aloca memorie pentru variabilele dinamice:

```
new(p)
if (p = NULL) then throw 'memorie insuficienta'
```

Subprograme

- Limbajul nostru algoritmic este unul modular, unde un modul este identificat de un subprogram.
- Există două tipuri de subprograme: proceduri și funcții.

Proceduri

- Interfața dintre o procedură și modulul care o apelează este realizată numai prin parametri și variabilele globale.
- Apelul unei proceduri apare ca o instrucțiune separată.
- Forma generală a unei proceduri este:

  `procedure` ⟨nume⟩(⟨lista-parametri⟩)
  ⟨secvență-instrucțiuni⟩
  `end`
- Lista parametrilor este opțională. Considerăm ca exemplu o procedură care interschimbă valorile a două variabile:

```
procedure swap(x, y)
    aux ← x
    x ← y
    y ← aux
end
```

- Permutarea circulară a valorilor a trei variabile `a, b, c` se face apelând de două ori procedura `swap`:

```
swap(a, b)
swap(b, c)
```

Funcții.

- În plus față de proceduri, o funcție întoarce o valoare asociată cu numele funcției.
- Apelul unei funcții poate participa la formarea de expresii.
- Forma generală a unei funcții este:

  `function` ⟨nume⟩(⟨lista-parametri⟩)
  ⟨secvență-instrucțiuni⟩
  `return` ⟨expresie⟩
  `end`
- Lista parametrilor este opțională.
- Valoarea întoarsă de funcție este cea obținută prin evaluarea expresiei.
- O instrucțiune `return` poate apărea în mai multe locuri în definiția unei funcții și execuția ei implică terminarea evaluării funcției.
- Considerăm ca exemplu o funcție care calculează maximul dintre valorile a trei variabile:

```
function max3(x, y, z)
    temp ← x
    if (y > temp) then temp ← y
    if (z > temp) then temp ← z
    return temp
end
```

- Are sens să scriem `2*max3(a, b, c)` sau `max3(a, b, c) < 5`.

Comentarii

- Comentariile sunt notate în mod similar cu limbajul C, utilizând combinațiile de caractere tt /* și */.
- Comentariile au rolul de a introduce explicații suplimentare privind descrierea algoritmului:

```
procedure absDec(x)
   if (x > 0) /* testeaza daca x este pozitiv */
      then x ← x-1 /* decrementeaza x */
      else x ← x+1 /* incrementeaza x */
end
```

Execuția programelor

- Intuitiv, o execuție a unui program constă în succesiunea de pași elementari determinați de execuțiile instrucțiunilor ce compun programul.
- Convenim să utilizăm noțiunea de *calcul* pentru execuția unui program.
- Exemplu:

```
x ← 0
i ← 1
while (i < 10) do
    x ← x*10+i
    i ← i+2
```

- Calculul descris de acest program ar putea fi reprezentat de tabelul din dreapta.

Pasul	Instrucțiunea	i	x
0	x ← 0	–	–
1	i ← 1	–	0
2	1<10	1	0
3	x ← x*10+i	1	0
4	i ← i+2	1	1
5	3<10	3	1
6	x ← x*10+i	3	1
7	i ← i+2	3	13
8	5<10	5	13
9	x ← x*10+i	5	13
10	i ← i+2	5	135
11	7<10	7	135
12	x ← x*10+i	7	135
13	i ← i+2	7	1357
14	9<10	9	1357
15	x ← x*10+i	9	1357
16	i ← i+2	9	13579
17	11<10	11	13579
18		11	13579

Configurații

- O *configurație c* include instrucțiunea curentă (starea programului) și starea memoriei (valorile curente ale variabilelor din program).
- Relația $c_{i-1} \vdash c_i$ are următoarea semnificație:
  - Prin execuția instrucțiunii din c_{i-1}, se transformă c_{i-1} în c_i.
- În exemplul anterior, o configurație este reprezentată de o linie în tabel.
  - Calculul este notat formal printr-o secvență $c_0 \vdash c_1 \vdash \cdots \vdash c_{18}$.
  - c_0 se numește *configurație inițială*, iar c_{18} *configurație finală*.
- Pot exista și calcule infinite. Exemplu:
  - Instrucțiunea

    ```
    while (true) do
       i ← i+1
    ```

    generează un calcul infinit.

Algoritmi și probleme

- Noțiunile de algoritmi, programe și probleme sunt diferite și trebuie înțeles bine care sunt relațiile dintre ele.
- Un algoritm exprimă o modalitate de a obține soluția unei probleme specifice.
- Descrierea algoritmului într-un limbaj algoritmic se face prin intermediul unui program.
- Toate noțiunile definite pentru program sunt extinse în mod natural la algoritm. Astfel, putem vorbi despre execuția (calculul) unui algoritm, configurație inițială, etc.

Noțiunea de problemă

- O problemă are două componente: domeniul, care descrie elementele ce intervin în problemă și relațiile dintre aceste elemente și o cerință de determinare a unor elemente ce au o anumită proprietate sau o întrebare despre proprietățile anumitor elemente.
- În funcție de scopul urmărit, există mai multe moduri de a formaliza o problemă.
- Vom utiliza numai două dintre ele.
- *Intrare/Ieșire.*
  - Putem formaliza problema rezolvată de algoritm ca o pereche (*Intrare*, *Ieșire*).
  - Componenta *Intrare* descrie datele de intrare, iar componenta *Ieșire* descrie datele de ieșire.
  - Un exemplu simplu de problemă reprezentată astfel este următorul:

  Intrare: Un număr întreg pozitiv x.
  Ieșire: Cel mai mare număr prim mai mic decât sau egal cu x.
- *Problemă de decizie.*
  - Este un caz particular de problemă când ieșirea este de forma $'DA'$ sau $'NU'$.
  - O astfel de problemă este reprezentată ca o pereche (*Instanță*, *Întrebare*) unde componenta *Instanță* descrie datele de intrare, iar componenta *Întrebare* se referă, în general, la existența unui obiect sau a unei proprietăți.
  - Un exemplu tipic îl reprezintă următoarea problemă:

  Instanță: Un număr întreg pozitiv x.
  Întrebare: Este x număr prim?

Problemele de decizie

- Problemele de decizie sunt preferate atât în teoria complexității, cât și în teoria calculabilității datorită reprezentării foarte simple a ieșirilor.
- Orice problemă admite o reprezentare sub formă de problemă de decizie, indiferent de reprezentarea sa inițială.
- De obicei, pentru reprezentarea problemelor de decizie se consideră o mulțime $\mathscr{U}$, iar o instanță este de forma $\mathscr{R} \subseteq \mathscr{U}, x \in \mathscr{U}$ și întrebarea de forma $x \in \mathscr{R}$?.
- Problema din exemplul anterior poate fi reprezentată astfel:

  Instanță: $\mathscr{P}$ = mulțimea numerelor prime ($\mathscr{P} \subset \mathbb{Z}$), $x \in \mathbb{Z}^+$.

  Întrebare: $x \in \mathscr{P}$?

Problemă rezolvată de un algoritm

- Un algoritm A *rezolvă o problemă* P *în sensul corectitudinii totale* dacă pentru orice instanță p, calculul unic determinat de configurația inițială ce include p este finit și configurația finală include ieșirea $P(p)$
- Un algoritm A *rezolvă o problemă* P *în sensul corectitudinii parțiale* dacă pentru orice instanță p pentru care calculul unic determinat de configurația inițială ce include p este finit, configurația finală include ieșirea $P(p)$.
- Ori de câte ori spunem că un algoritm A rezolvă o problemă P vom înțelege de fapt că A rezolvă o problemă P în sensul corectitudinii totale.
- O problemă P este *rezolvabilă* dacă există un algoritm care să o rezolve în sensul corectitudinii totale. Dacă P este o problemă de decizie, atunci spunem că P este *decidabilă*.
- O problemă de decizie P este *semidecidabilă* sau *parțial decidabilă* dacă există un algoritm A care rezolvă P în sensul corectitudinii parțiale astfel încât calculul lui A este finit pentru orice instanță p pentru care răspunsul la întrebare este $'DA'$.
- O problemă P este *nerezolvabilă* dacă nu este rezolvabilă, adică nu există un algoritm care să o rezolve în sensul corectitudinii totale. Dacă P este o problemă de decizie, atunci spunem că P este *nedecidabilă*.

Probleme nedecidabile

- Pentru a arăta că o problemă este decidabilă este suficient să găsim un algoritm care să o rezolve. Mai complicat este cazul problemelor nedecidabile.
- În legătură cu acestea din urmă se pun, în mod firesc, următoarele întrebări:
  - Există probleme nedecidabile?
  - Cum putem demonstra că o problemă nu este decidabilă?
- Răspunsul la prima întrebare este afirmativ. Un exemplu de problemă nedicidabilă este *problema opririi*.
- Problema opririi:
  - Notăm cu $\mathscr{U}$ mulțimea perechilor de forma $\langle A, \overline{x} \rangle$ unde A este un algoritm și $\overline{x}$ este o intrare pentru A, iar $\mathscr{R}$ este submulțimea formată din acele perechi $\langle A, \overline{x} \rangle$ pentru care calculul lui A pentru intrarea $\overline{x}$ este finit.
  - Dacă notăm prin $A(\overline{x})$ (a se citi $A(\overline{x}) = \mathit{true}$) faptul că $\langle A, \overline{x} \rangle \in \mathscr{R}$, atunci problema opririi poate fi scrisă astfel:

Instanță: Un algoritm A, $\overline{x} \in \mathbb{Z}^*$.

Întrebare: $A(\overline{x})$?

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*.

Problema căutării unui element într-o secvență de numere întregi

- **Problema** P_1
  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 $(\mathtt{a[i]} \mid 0 \leq i \leq n-1)$.
- Algoritmul A_1 descris de următorul program rezolvă P_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 P_1 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
```

Problema determinării elementului de valoare maximă dintr-o secvență de numere întregi

- **Problema** P_2
  Intrare: $n, (a_0, \ldots, a_{n-1})$ numere întregi.
  Ieșire: $max = \max\{a_i \mid 0 \leq i \leq n-1\}$.
- Presupunem că secvența $(a_0, \ldots, a_{n-1})$ este memorată în tabloul $(\texttt{a[i]} \mid 0 \leq i \leq n-1)$.
- Algoritmul A_2 descris de următorul program rezolvă P_2:

```
/* algoritmul A2 */
max ← a[0]
for i ← 1 to n-1 do
   if (a[i]>max)
      then max ← a[i]
```

Problema determinării elementului de valoare maximă dintr-o secvență de numere întregi - continuare

- Dimensiunea problemei P_2 este n, numărul elementelor din secvența în care se caută maximul. Vom presupune și în acest caz că toate operațiile necesită o unitate de timp.
- Cazul cel mai favorabil este obținut când a_0 este elementul de valoare maximă. Se efectuează $n+n-1=2n-1$ comparații și $1+n$ atribuiri. Rezultă $T_{A_1}^{fav}(n)=2n-1+1+n=3n$.
- Cazul cel mai nefavorabil se obține in situația în care tabloul este ordonat crescător (pentru că de fiecare dată instrucțiunea `if` se execută pe ramura `then`, adică se face și comparație și atribuire). În acest caz numărul comparațiilor este $2n-1$ și cel al atribuirilor este $2n$. Rezulă $T_{A_2}(n)=2n-1+2n=4n-1$.
- Dimensiunea memoriei utilizate de algoritm este $n+5$ (tabloul `a`, constanta `1`, variabilele `i`, `n` și `z`), în ambele cazuri.

```
/* algoritmul A2 */
max ← a[0]
for i ← 1 to n-1 do
   if (a[i] > max)
      then max ← a[i]
```

Sortarea prin inserare

- **Problema** P_3
  Intrare: $n, (a_0, \ldots, a_{n-1})$ numere întregi.
  Ieșire: $(a_{i_0}, \ldots, a_{i_{n-1}})$ unde $(i_0, \ldots, i_{n-1})$ este o permutare a șirului $(0, \ldots, n-1)$ și $a_{i_j} \leq a_{i_{j+1}}, \forall j \in \{0, \ldots, n-2\}$.
- Presupunem din nou că secvența $(a_0, \ldots, a_{n-1})$ este memorată în tabloul $(\texttt{a[i]} \mid 0 \leq i \leq n-1)$.
- Algoritmul sortării prin inserare 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 A_3 care rezolvă P_3 este descris de următorul program:

```
/* algoritmul A3 */
for k ← 1 to n-1 do
    temp ← a[k]
    i ← k-1
    while (i ≥ 0 and a[i] > temp) do
        a[i+1] ← a[i]
        i ← i-1
    a[i+1] ← temp
```

Sortarea prin inserare - continuare

- Dimensiunea problemei P_3 este dată de numărul n al elementelor din secvența ce urmează a fi sortată. Presupunem și aici că toate operațiile necesită o unitate de timp.
- Cazul cel mai favorabil este obținut când elementele secvenței sunt sortate crescător. În această situație se efectuează $n+2(n-1)$ comparații și $n+3(n-1)$ atribuiri. Rezultă $T^{fav}_{A_1}(n) = 3n-2+4n-3 = 6n-5$.
- Cazul cel mai nefavorabil este dat de situația în care deplasarea (la dreapta cu o poziție în vederea inserării) se face de la începutul tabloului, adică șirul este ordonat descrescător. Estimarea timpului de execuție pentru cazul cel mai nefavorabil este următoarea: numărul de comparații este $n+2(2+3+\cdots+n) = n+n(n+1)-2 = n^2+2n-2$, iar cel al atribuirilor este $n+3(n-1)+2(1+2+\cdots+n-1) = 4n-3+(n-1)n = n^2+3n-3$. Adunând, avem $T_{A_3}(n) = (n^2+2n-2)+(n^2+3n-3) = 2n^2+5n-5$.
- Spațiul este $n+5$ (tabloul a, constantele 0 și 1, variabilele i, temp, n și z).

```
/* algoritmul A3 */
for k ← 1 to n-1 do
    temp ← a[k]
    i  ← k-1
    while (i ≥ 0 and a[i] > temp) do
        a[i+1] ← a[i]
        i  ← i-1
    a[i+1]  ← temp
```

Căutarea binară

- Reconsiderăm problema căutării unui element într-un tablou, dar cu presupunerea suplimentară că tabloul este ordonat.

  Problema P_4

  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 $\mathtt{i}_{stg}$ indicele elementului din stânga zonei de căutare în tablou, $\mathtt{i}_{dr}$ indicele elementului din dreapta zonei de căutare în tablou.

Căutarea binară - continuare

Algoritmul căutării binare (A_4)este descris de subprogramul următor:

```
/* algoritmul A4 */
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
```

Căutarea binară - continuare

- Dimensiunea problemei P_4 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 $\mathtt{i}_{stg}$, $\mathtt{i}_{dr}$, $\mathtt{i}_{med}$, `n` și `z`).

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 exemplele anterioare, 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 P_1 definită anterior.
- Mulțimea valorilor variabilei aleatoare $T_A(p)$ 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 $T_A(p) = 3i+2$ ($poz = i-1$) este $\frac{q}{n}$, pentru $1 \leq i < n$, iar probabilitatea ca $T_A(p) = 3n+2$ 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:

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

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

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, Ω și Θ 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

Teorema (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, Ω și Θ) 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.

Încadrarea algoritmilor în clasele lor

- Putem acum încadra algoritmii prezentați în exemplele anterioare în clasele lor.
- Pentru algoritmul ce rezolva problema P_1 (căutarea unui element într-o secvență de numere), $T_{A_1}(n) = 3n+2$. Deci, timpul de execuție al acestui algoritm pentru cazul cel mai nefavorabil este în clasa $O(n)$.
- Pentru problema P_2 (căutarea într-un șir a elementului de valoare maximă), $T_{A_2}(n) = 1+2(n-1)$. Timpul de execuție a algoritmului ce rezolvă această problemă pentru cazul cel mai nefavorabil este în clasa $O(n)$.
- În cazul problemei P_3 (sortarea prin inserare), $T_{A_3}(n) = 3(n-1)+3n(n-1)/2$. Rezultă că timpul de execuție al algoritmului respectiv este în clasa $O(n^2)$.
- Pentru problema P_4 (căutarea binară) nu se poate aplica teorema 1. Deoarece $T_{A_4}(n) \leq \log_2 n+1$, timpul de execuție al acestui algoritm este în clasa $O(\log_2 n)$. Baza logaritmului se poate ignora deoarece: $\log_a x = \log_a b \log_b x$ și $\log_a b$ este o constantă, deci rămâne $O(\log n)$, adică o funcție logaritmică.

Proiectarea algoritmilor

Problema căutării

Mitică Craus

Univeristatea Tehnică "Gheorghe Asachi" din Iași

Cuprins


Introducere

- Alături de sortare, căutarea în diferite structuri de date constituie una dintre operațiile cele mai des utilizate în activitatea de programare.
- Problema căutării poate fi formulată în diverse moduri:
  - Dat un tablou $(\texttt{s[i]} \mid 0 \leq i < n)$ și un element a, să se decidă dacă există $i \in \{0, \ldots, n-1\}$ cu proprietatea $s[i] = a$.
  - Dată o structură înlănțuită (liniară, arbore, etc.) și un element a, să se decidă dacă există un nod în structură a cărui informație este egală cu a.
  - Dat un fișier și un element a, să se decidă dacă există o componentă a fișierului care este egală cu a etc.
- În plus, fiecare dintre aceste structuri poate avea sau nu anumite proprietăți:.
  - Informațiile din componente sunt distincte două câte două sau nu.
  - Componentele sunt ordonate în conformitate cu o relație de ordine peste mulțimea informațiilor sau nu.
  - Căutarea se poate face pentru toată informația memorată într-o componentă a structurii sau numai pentru o parte a sa numită *cheie*.
    - Cheile pot fi unice (ele identifică în mod unic componentele) sau multiple (o cheie poate identifica mai multe componente).
  - Între oricare două căutări, structura de date nu suferă modificări (aspectul static) sau poate face obiectul operațiilor de inserare/ștergere (aspectul dinamic).

Formularea abstractă a problemei căutării

Instanță: o mulțime univers $\mathscr{U}$, o submulțime $S \subseteq \mathscr{U}$ și un element a din $\mathscr{U}$;

Întrebare: $x \in S$?

- Aspectul *static* este dat de cazul când, între oricare două căutări, mulțimea S nu suferă nici o modificare.
- Aspectul *dinamic* este obținut atunci când între două căutări mulțimea S poate face obiectul următoarelor operații:
  - Inserare.

  Intrare: S, x;
  Ieșire: $S \cup \{x\}$.
  - Ștergere.

  Intrare: S, x;
  Ieșire: $S \setminus \{x\}$.
- Evident, realizarea eficientă a căutării și, în cazul aspectului dinamic, a operațiilor de inserare și de ștergere, depinde de structura de date aleasă pentru reprezentarea mulțimii S.

Căutare în liste liniare

- Mulțimea S este reprezentată printr-o listă liniară. Dacă mulțimea $\mathscr{U}$ este total ordonată, atunci S poate fi reprezentată de o listă liniară ordonată.
- Timpul de execuție pentru cazul cel mai nefavorabil este dependent de implementarea listei. Tabelul de mai jos include un sumar al valorilor acestor timpi de execuție.
- Facem observația că valorile pentru operațiile de inserare și ștergere nu presupun și componenta de căutare.
- În mod obișnuit, un element x este adăugat la S numai dacă el nu apare în S; analog, un element x este șters din S numai dacă el apare în S.
  - Ambele operații ar trebui precedate de căutare.
  - La valorile timpilor de execuție pentru inserare și ștergere ar trebui adaugătă și valoarea corespunzătoare pentru căutare.
- De exemplu, timpul de execuție în cazul cel mai nefavorabil pentru inserare în cazul în care S este reprezentată prin tablouri neordonate devine $O(n)$,iar pentru cazul tablourilor ordonate rămâne aceeași, $O(n)$.

Tip de date	**Implementare**	**Căutare**	**Inserare**	**Ștergere**
Listă liniară	Tablouri	$O(n)$	$O(1)$	$O(n)$
	Liste înlănțuite	$O(n)$	$O(1)$	$O(1)$
Listă liniară ordonată	Tablouri	$O(\log_2 n)$	$O(n)$	$O(n)$
	Liste înlănțuite	$O(n)$	$O(1)$	$O(1)$

Timpului mediu de execuție

- Pentru calculul timpului mediu de execuție vom presupune că $a \in S$ cu probabilitatea q și că a poate apărea în S la adresa adr cu aceeași probabilitate $\frac{q}{n}$.
- Timpul mediu de execuție al căutărilor cu succes (a este găsit în S) este:

$$T^{med,s}(n) = \frac{3q(1+2+\cdots+n)}{n} + 2q = \frac{3q(n+1)}{2} + 2q,$$

iar în cazul general avem:

$$T^{med}(n) = 3n - \frac{3nq}{2} + \frac{3q}{2} + 2.$$

- **Cazul în care se ia în considerare frecvența căutărilor.**
  - Presupunem că x_i este căutat cu frecvența f_i.
  - Se poate demonstra că se obține o comportare în medie bună, atunci când $f_1 \geq \cdots \geq f_n$.
  - Dacă aceste frecvențe nu se cunosc aprioric, se pot utiliza *tablourile cu autoorganizare*.
  - Într-un tablou cu autoorganizare, ori de câte ori se caută un $a = s[i]$, acesta este deplasat la începutul tabloului în modul următor: elementele de pe pozițiile $1, \ldots, i-1$ sunt deplasate la dreapta cu o poziție după care se pune a pe prima poziție.
  - Dacă în loc de tablouri se utilizează liste înlănțuite, atunci deplasările la dreapta nu mai sunt necesare.
  - Se poate arăta că pentru tablourile cu autoorganizare timpul mediu de execuție este:

$$T^{med,s}(n) \approx \frac{2n}{\log_2 n}$$

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.
- Căutarea. Operația de căutare într-un asemenea arbore este descrisă de următoarea procedură:

```
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.

Căutare în arbori de căutare echilibrați

- Considerăm arbori binari de căutare. Este posibil ca în urma operațiilor de inserare și de ștergere structura arborelui binar de căutare să se modifice foarte mult și operația de căutare să nu mai poată fi executată în timpul $O(\log_2 n)$.
- Suntem interesați să găsim algoritmi pentru inserție și ștergere care să mențină o structură "echilibrată" a arborilor de căutare (nu neaparat binari) astfel încât operația de căutare să se execute în timpul $O(\log n)$ totdeauna.
- Reamintim că înălțimea unui arbore t, pe care o notăm cu $h(t)$, este lungimea drumului maxim de la rădăcină la un vârf de pe frontieră.
- Fie $\mathscr{C}$ o mulțime de arbori. $\mathscr{C}$ se numește *clasă de arbori echilibrați (balansați)* dacă pentru orice $t \in \mathscr{C}$, înălțimea lui t este mai mică decât sau egală cu $c \cdot \log n$, unde c este o constantă ce poate depinde de clasa $\mathscr{C}$ (dar nu depinde de t) și n este numărul de vârfuri din t.
- O clasă $\mathscr{C}$ de arbori echilibrați se numește $O(\log n)$-*stabilă* dacă există algoritmi pentru operațiile de căutare, inserare și de ștergere care necesită timpul $O(\log n)$ și arborii rezultați în urma execuției acestor operații fac parte din clasa $\mathscr{C}$.

Căutare în arbori AVL

- Arborii AVL sunt arbori binari de căutare echilibrați.
- **Definiție**: Un arbore *AVL* este un arbore binar de căutare în care, pentru orice vârf, diferența dintre înălțimea subarborelui din stânga vârfului și cea a subarborelui din dreapta este -1, 0 sau 1. Numim această diferență *factor de echilibrare*
- Pentru orice arbore AVL t, cu n noduri interne, are loc $h(t) = \Theta(\log_2 n)$.
- Clasa arborilor AVL este $O(\log n)$-stabilă.

.

Căutare în arbori bicolori

- Arborii bicolori sunt arbori binari de căutare echilibrați. Echilibrarea este menținută cu ajutorul unei colorări adecvate ale nodurilor.
- Sunt suficiente numai două culori pentru a putea defini o clasă $O(\log n)$-stabilă.
- Deoarece culorile au fost inițial roșu și negru, arborii sunt cunoscuți sub numele de *red-black-trees*.
- **Definiție**: Un *arbore bicolor* este un arbore binar de căutare care satisface următoarele proprietăți:
  1. Nodurile sunt colorate. Un nod are sau culoarea roșie sau culoarea neagră.
  2. Nodurile pendante (acestea sunt considerate ca făcând parte din structură) sunt colorate cu negru.
  3. Dacă un nod este roșu, atunci fiii săi sunt colorați cu negru.
  4. Pentru orice nod, drumurile de la acel nod la nodurile de pe frontieră au același număr de noduri colorate cu negru.

Arbori bicolori - exemplu

- Un exemplu de arbore bicolor este reprezentat în figura 1.
- Nodurile reprezentate cu cercuri albe reprezintă nodurile colorate cu roșu.
- Dacă v este un nod într-un arbore bicolor t, atunci notăm cu $hn(v)$ numărul de noduri negre aflate pe un drum de la v la un nod de pe frontieră, excluzând v.
- Definiția lui hn nu depinde de alegerea drumului datorită condiției 4 din definiție.

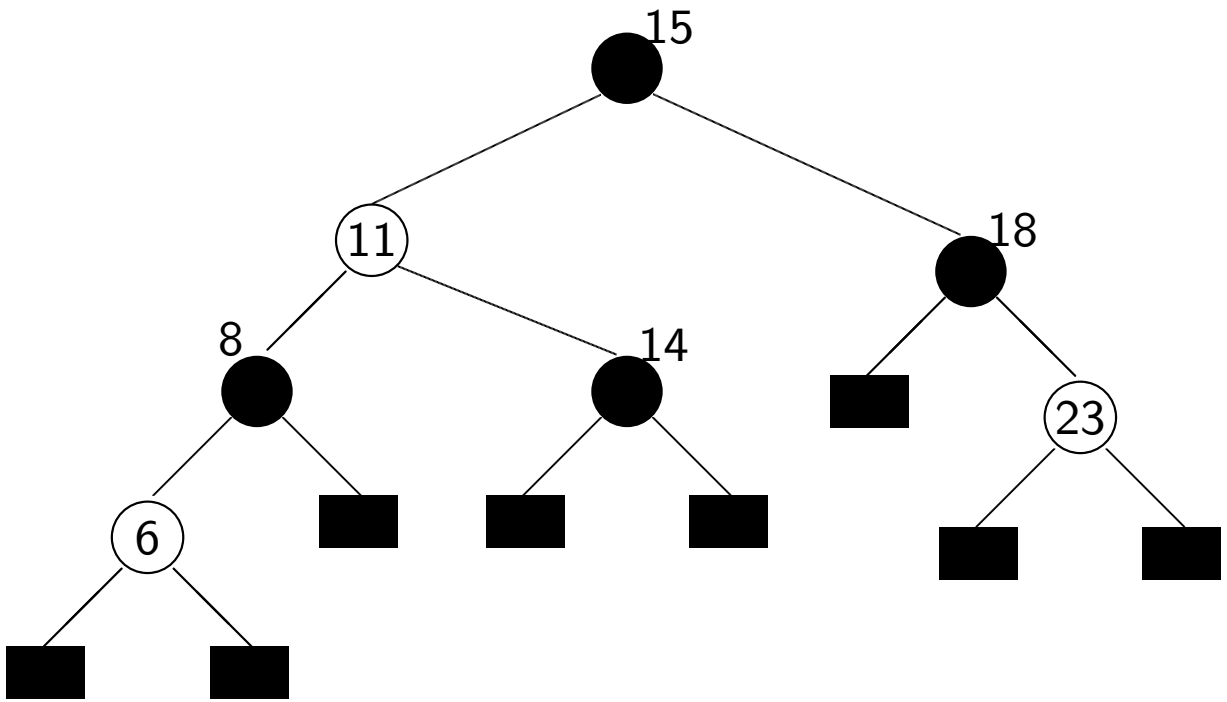


Figura 1 : Arbore bicolor

Clasa arborilor bicolori este $O(\log n)$-stabilă

Lema

Fie t un arbore bicolor. Orice subarbore al lui t are cel puțin $2^{hn(v)}-1$ noduri interne, unde v este rădăcina subarborelui.

Demonstrație.

Procedăm prin inducție după înălțimea h a subarborelui. Dacă $h=0$, atunci v este singurul nod al subarborelui și este un nod pendant. Numărul de noduri interne este 0 și coincide cu $2^{hn(v)}-1=2^0-1$. Presupunem $h>0$. Distingem două situații.

1. v este roșu. Atunci fiii lui v sunt negri. Fie x și y fiii lui v. Avem $hn(v)=hn(x)+1=hn(y)+1$. Din ipoteza inductivă, fiecare subarbore al lui v are cel puțin $2^{hn(v)-1}-1$ noduri interne. Rezultă că t are cel puțin $2\cdot(2^{hn(v)-1}-1)+1=2^{hn(v)}-1$ noduri interne.
2. v este negru. Dacă x și y sunt ambii roșii atunci $hn(v)=hn(x)=hn(y)$. Din ipoteza inductivă, fiecare subarbore al lui v are cel puțin $2^{hn(v)}-1$ noduri interne. Rezultă că t are cel puțin $2\cdot(2^{hn(v)}-1)+1>2^{hn(v)}-1$. Celelalte cazuri se tratează asemănător.

□

Clasa arborilor bicolori este $O(\log n)$-stabilă - continuare

Teorema

Un arbore bicolor cu n noduri interne are înălțimea cel mult $2\log_2(n+1)$.

Demonstrație.

Fie t un arbore bicolor cu rădăcina v și înălțimea h. Din lema 1 rezultă că t are cel puțin $2^{hn(v)}-1$ noduri interne. Din condiția 3 din definiția arborilor bicolori, rezultă că pe orice drum de la v la un nod de pe frontieră cel puțin jumătate dintre noduri sunt negre. Avem $hn(v) \geq \frac{h}{2}$ care implică $n \geq 2^{hn(v)}-1 \geq 2^{\frac{h}{2}}-1$. De aici $h \leq 2\log_2(n+1)$. □

Teorema

Clasa arborilor bicolori este $O(\log n)$*-stabilă.*

Căutare în 2-3-arbori

- 2-3-arborii constituie o clasă de structuri de date arborescente $O(\log n)$-stabilă ce generalizează arborii binari de căutare.
- **Definiție**: Un *2-3-arbore* este un arbore de căutare care satisface următoarele proprietăți:
  1. Fiecare nod intern este de aritate 2 sau 3 (are 2 sau 3 fii). Un nod $*v$ de aritate 2 memorează o singură valoare v->*eltStg*, iar un nod de aritate 3 memorează două valori: v->*eltStg* și v->*eltMij* cu v->*eltStg* $<$ v->*eltMij*.
  2. Fiii unui nod $*v$ îi vom numi fiu stânga (notat $*(v$->*stg*$)$), mijlociu ($*(v$->*mij*$)$) și dreapta ($*(v$->*drp*$)$). Un nod de aritate 2 are fiii stânga și mijlociu.
  3. Pentru orice nod intern $*v$ de aritate 2, valorile memorate în subarborele cu rădăcina în $*(v$->*stg*$)$ sunt mai mici decât v->*eltStg* și valorile din subarborele cu rădăcina în $*(v$->*mij*$)$ sunt mai mari decât v->*eltStg*.
  4. Pentru orice nod intern $*v$ de aritate 3 are loc:
     - 4.1 valorile din subarborele cu rădăcina $*(v$->*stg*$)$ sunt mai mici decât v->*eltStg*;
     - 4.2 valorile din subarborele cu rădăcina $*(v$->*mij*$)$ sunt mai mari decât v->*eltStg* și mai mici decât v->*eltMij*;
     - 4.3 valorile din subarborele cu rădăcina $*(v$->*drp*$)$ sunt mai mari decât v->*eltMij*.
  5. Toate nodurile de pe frontieră se află pe același nivel.

2-3-arbori - exemplu

- Un exemplu de 2-3-arbore este reprezentat grafic în figura 2.
- Nodurile reprezentate grafic prin pătrate nu intră în definiția reprezentării, ele având rolul numai de a sugera mai bine structura de 2-3-arbore.
- Se poate stabili o corespondență între aceste noduri și elementele mulțimii reprezentate.
- Facem presupunerea ca în cazul nodurilor de aritate 2, câmpul *v->eltMij* să memoreze o valoare artificială *MaxVal* mai mare decât orice element din mulțimea univers.

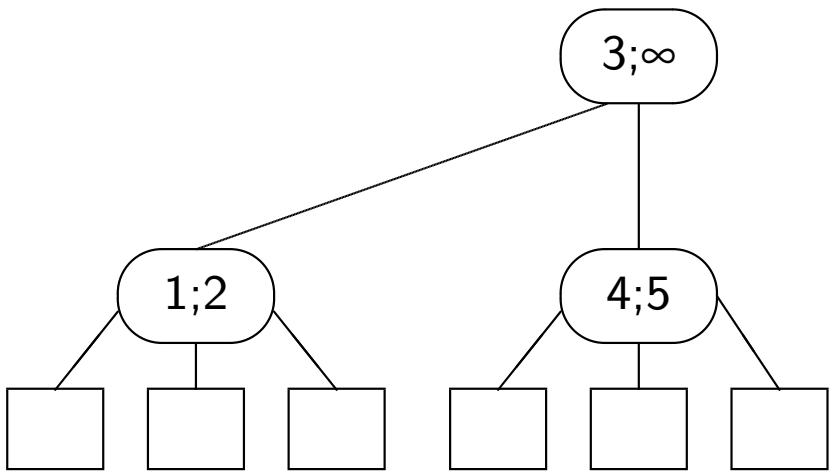


Figura 2 : Exemplu de 2-3-arbore

Operația de căutare într-un 2-3-arbore

- Operația de căutare se realizează într-o manieră asemănătoare cu cea de la arborii binari de căutare.
- Procesul de căutare pentru a în arborele t pleacă din rădăcină și în fiecare nod vizitat $*v$ se compară a cu valorile memorate în $*v$:
  - dacă $a = v\text{->}eltStg$ sau $a = v\text{->}eltMij$, atunci căutarea se termină cu succes;
  - dacă subarborele în care se caută este vid, atunci căutarea se termină cu insucces;
  - dacă $a < v\text{->}eltStg$, atunci procesul de căutare continuă în subarborele cu rădăcina în $*(v\text{->}stg)$;
  - dacă $v\text{->}eltStg < a < v\text{->}eltMij$, atunci procesul de căutare continuă în subarborele cu rădăcina în $*(v\text{->}mij)$;
  - dacă $a > v\text{->}eltMij$, atunci procesul de căutare continuă în subarborele cu rădăcina în $*(v\text{->}drp)$.

Operația de căutare într-un 2-3-arbore - pseudocod

```
function caut23arbore(t, a)
   p ← t
   while (p ≠ NULL)
      if ((p->eltStg = a) or (p->eltMij = a))
      then return p
      else if (a < p->eltStg)
           then p ← p->stg
           else if (a < p->eltMij)
                then p ← p->mij
                else p ← p->drp
   return p
end
```

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

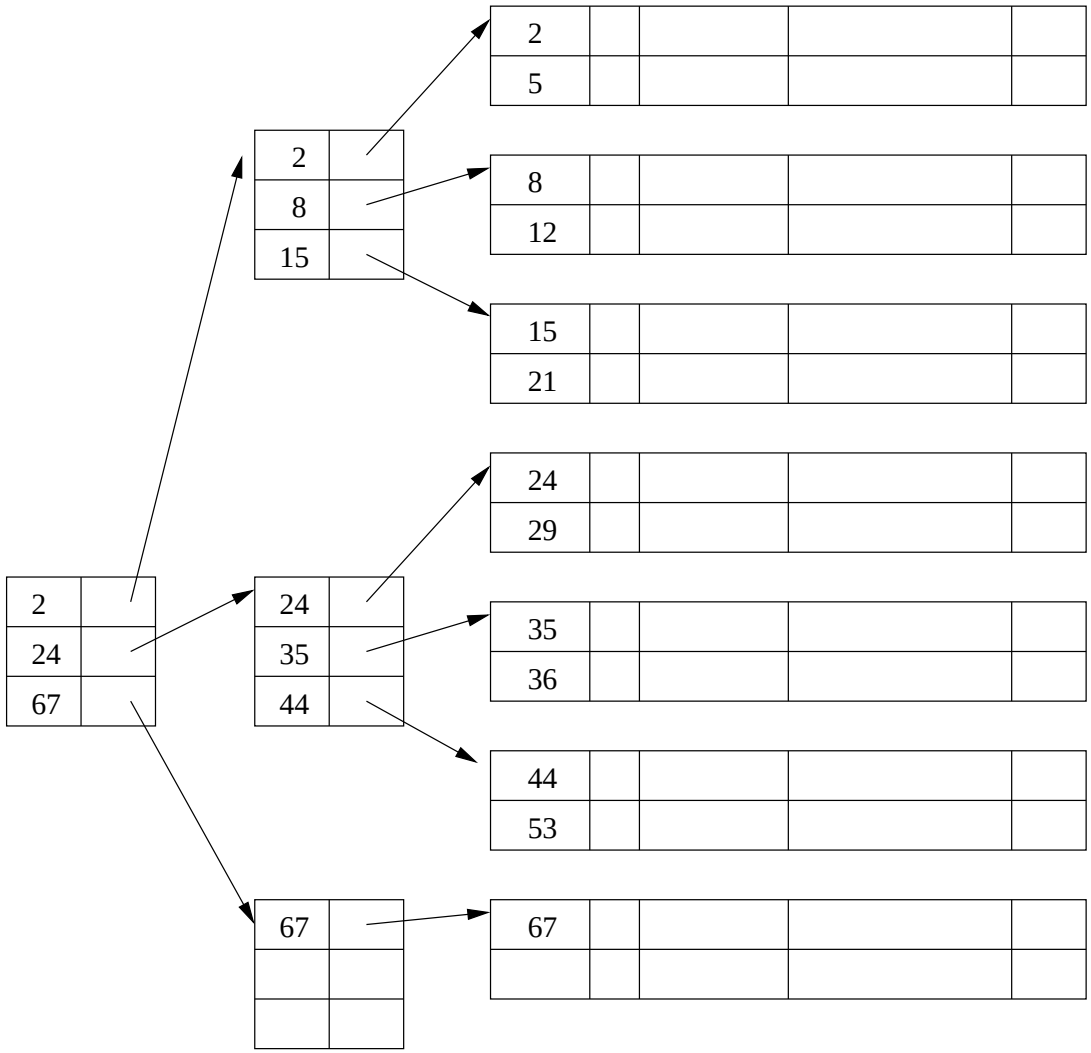


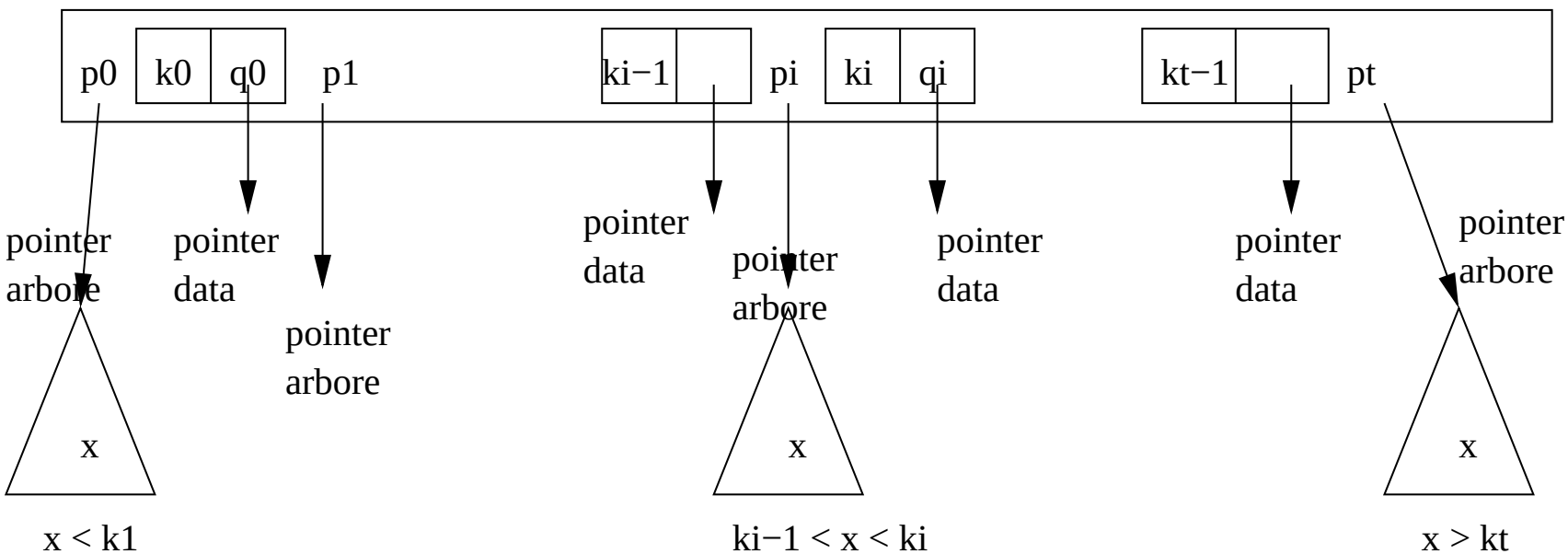
Figura 3 : Index multistratificat organizat pe 3 niveluri

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 $((p_0, p_1, \ldots, p_t)$.



nui nod al unui B-arbore

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
```

Creare B-arbore

- 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
```

Ștergere în B-arbore - descriere

1. Dacă nodul-gazdă al cheii ce urmează a fi ștearsă nu este frunză, atunci se efectuează o interschimbare între aceasta și succesorul ei în ordinea naturală (crescătoare) a cheilor. Deoarece cheia succesoare se află într-o frunză, operația se reduce la ștergerea unei chei dintr-o frunză.
2. Se șterge intrarea corespunzătoare cheii.
3. Dacă nodul curent conține cel puțin $f-1$ chei, operația de ștergere se consideră terminată.
4. Dacă nodul curent conține mai puțin decât $f-1$ chei, se consideră frații vecini.
   - 4.1 Dacă unul din frații vecin are mai mult decât $f-1$ chei, atunci se redistribuie una dintre intrările acestui frate în nodul-părinte și una din intrarile din nodul părinte se redistribuie nodului curent (deficitar).
   - 4.2 Dacă ambii frații au exact $f-1$ chei, atunci se unește nodul curent cu unul dintre frații vecini și cu o intrare din părinte.
5. Dacă nodul-părinte devine deficitar (conține mai puțin decât $f-1$ chei), acesta devine nod curent și se reia pasul 4.

Tabele de simboluri

- Dispersia este o tehnică aplicată în implementarea tabelelor de simboluri.
- O *tabelă de simboluri* este un tip de date abstract în care obiectele sunt perechi (*nume*, *atribute*) și asupra cărora se pot executa următoarele operații:
  1. căutarea unui nume în tabelă,
  2. regăsirea atributelor unui nume,
  3. modificarea atributelor unui nume,
  4. inserarea unui nou nume și a atributelor sale și
  5. ștergerea unui nume și a atributelor sale.
- Exemple tipice pentru tabelele de simboluri sunt:
  1. dicționarele de sinonime (tezaurele) (nume = cuvânt și atribute = sinonime) și
  2. tabela de simboluri a unui compilator (nume = identificator și atribute = (valoare inițială, lista liniilor în care apare etc.)).

Implementarea tabelelor de simboluri

- Întrucât operația de căutare se realizează după nume, notăm cu $\mathbb{U}$ mulțimea tuturor numelor.
- Implementarea tabelelor de simboluri prin tehnica dispersiei presupune:
  - un tablou $(\texttt{T[i]} \mid 0 \leq i \leq p-1)$, numit și *tabelă de dispersie (hash)*, pentru memorarea numelor și a referințelor la mulțimile de atribute corespunzătoare;
  - o funcție $h : \mathbb{U} \to [0, p-1]$, numită și *funcție de dispersie (hash)*, care asociază unui nume o adresă în tabelă.
- O submulțime $S \subseteq \mathbb{U}$ este reprezentată în modul următor:
  - Pentru fiecare $x \in S$ se determină $i = h(x)$ și numele x va fi memorat în componenta $T[i]$.
  - Valoarea funcției de dispersie se mai numește și *index* sau *adresă în tabelă*.
  - Dacă pentru două elemente diferite x și y avem $h(x) = h(y)$, atunci spunem că între cele două elemente există *coliziune*.
  - Pentru ca operația de dispersie să fie eficientă, trebuie rezolvate corect următoarele două probleme: alegerea funcției de dispersie și rezolvarea coliziunilor.

Tehnici elementare de alegere a funcției de dispersie

- **Trunchierea.** Se ignoră o parte din reprezentarea numelui. De exemplu, dacă numele sunt reprezentate prin secvențe de cifre și $p = 1000$, atunci $f(x)$ poate fi numărul format din ultimele trei cifre ale reprezentării: $f(62539194) = 194$.
- **Folding.** Reprezentarea numelui este partiționată în câteva părți și apoi aceste părți sunt combinate pentru a obține indexul. De exemplu, reprezentatarea 62539194 este partiționată în părțile 625, 391 și 94. Combinarea părților ar putea consta, de exemplu, în adunarea lor: $625 + 391 + 94 = 1110$. În continuare se poate aplica și trunchierea: $1110 \mapsto 110$. Deci $f(62539194) = 110$.

Tehnici elementare de alegere a funcției de dispersie - continuare

- **Aritmetică modulară.** Se convertește reprezentarea numelui într-un număr și se ia ca rezultat restul împărțirii la p. Este de preferat ca p să fie prim, pentru a avea o repartizare cât mai uniformă a elementelor în tabelă.
- *Exemplu:* Presupunem că un nume este reprezentat printr-un șir de caractere ASCII. Notăm cu `int(`c`)` funcția care întoarce codul zecimal al caracterului c.

```
function hash(x)
    h ← 0
    for i ← 0 to lung(x)-1 do
        h ← h + int(x[i])
    return h%p
end
```

- **Multiplicare.** Fie w cel mai mare număr ce poate fi memorat într-un cuvânt al calculatorului (de exemplu, $w = 2^{32}$, dacă cuvântul are 32 de biți). Funcția de dispersie prin multiplicare este dată de

$$h(x) = \left\lfloor p \cdot \left\{ x \frac{A}{w} \right\} \right\rfloor$$

unde $\{y\} = y - \lfloor y \rfloor$ și A este o constantă convenabil aleasă. De obicei, se ia A prim cu w și p.

Tehnici de bază pentru rezolvarea coliziunii

- **Dispersie cu înlănțuire**: Numele cu aceeași adresă sunt memorate într-o listă liniară simplu înlănțuită
- **Dispersie cu adresare deschisă**: Pentru un nume $x \in \mathbb{U}$ se definește o secvență $h(x, i)$, $i = 0, 1, 2, \ldots$, de poziții în tabelă. Această secvență, numită și secvență de *încercări*, va fi cercetată ori de câte ori apare o coliziune.

Dispersie cu înlănțuire - exemplu

- Notăm cu `s.nume` câmpul care memorează numele unui simbol `s` și cu `s.latr` câmpul ce memorează lista (sau adresa listei) de atribute a lui `s`.

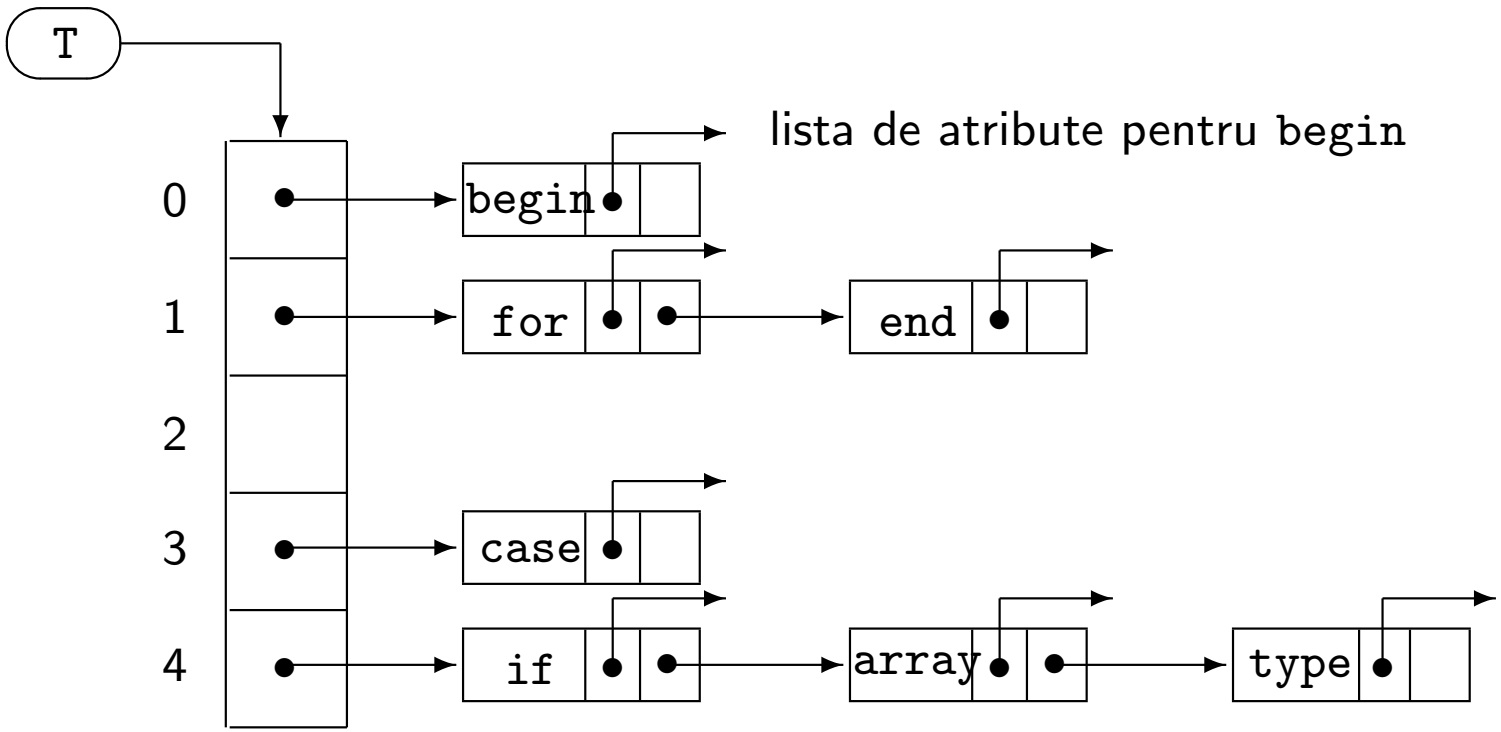


Figura 5 : Dispersie cu înlănțuire

Dispersie cu înlănțuire - căutarea unui nume

- Căutarea unui nume în tabelă se realizează în două etape:
  - mai întâi se calculează adresa în tabelă corespunzătoare numelui și apoi
  - se caută secvențial în lista simplu înlănțuită.

```
function cautTabHash(x, T)
    i ← hash(x)
    q ← T[i]
    while (q ≠ NULL) do
        if (q->nume = x)
            then return q
            else q ← q->succ
    return NULL
end
```

Dispersie cu înlănțuire - regăsirea atributelor unui element

- Regăsirea atributelor unui nume presupune mai întâi căutarea numelui in tabelă:

```
function atribute(x, T)
    q ← cautTabHash(x, T)
    if (q ≠ NULL)
        then return q->latr
        else return NULL
end
```

Dispersie cu înlănțuire - adăugarea unui element

- Operația de adăugare a unui element x în tabelă presupune calculul lui $i = h(x)$ și inserarea acestuia în lista $T[i]$.
- Pentru ca operația de adăugare să se realizeze cu cât mai puține operații, inserarea se va face la începutul listei.

```
procedure insereazaTabHash(x, xatr, T)
    i ← hash(x)
    new(q)
    q->nume ← x
    q->latr ← xatr
    q->succ ← T[i]
    T[i] ← q
end
```

Dispersie cu înlănțuire - ștergerea unui element

- Operația de ștergere a numelui x presupune căutarea pentru x în lista $T[h(x)]$ și eliminarea nodului corespunzător.
- Odată cu ștergerea numelui sunt șterse și atributele acestuia.

```
procedure stergeTabHash(x, T)
    i ← hash(x)
    q ← T[i]
    predq ← NULL
    while ((q ≠ NULL) and (q->nume ≠ x) do
        predq ← q
        q ← q->succ
    if (q ≠ NULL)
        then delete(q->latr) /* elimina toata lista */
             if (q = T[i])
                 then T[i] ← q->succ
                 else predq->succ ← q->succ
    delete(q)
end
```

Dispersie cu înlănțuire - numărul mediu de comparații

Teorema (1)

Numărul mediu de comparații pentru o căutare cu succes este aproximativ $1+\frac{\beta}{2}$, *unde* $\beta=\frac{\#S}{p}$ *este factorul de încărcare al tabelei.*

Demonstrație.

Metoda 1. Știm că o căutare cu succes într-o listă liniară de lungime m necesită în medie $\frac{m+1}{2}$ comparații. Lungimea medie a unei liste este β, iar probabilitatea ca elementul căutat să aparțină unei liste este $\frac{1}{p}$ (se presupune o dispersie uniformă). Numărul mediu de comparații pentru o căutare cu succes este:

$$1+\sum_{i=0}^{p-1}\frac{\beta+1}{2}\cdot\frac{1}{p}=1+\frac{\beta+1}{2}$$

Metoda 2. Presupunem pentru moment că inserările elementelor se fac la sfârșitul listelor. Căutarea pentru al i-lea element inserat necesită un număr de comparații egal cu 1 plus lungimea listei la sfârșitul căreia a fost adăugat. La momentul inserării elementului i, lungimea medie a listelor este $\frac{i-1}{p}$. Rezultă că numărul mediu de comparații pentru o căutare cu succes este:

$$\frac{1}{n}\sum_{i=1}^{n}\left(1+\frac{i-1}{p}\right)=1+\frac{\beta}{2}-\frac{1}{2p}$$

unde $n=\#S$

Dispersie cu adresare deschisă

- Pentru un nume $x \in \mathbb{U}$ se definește o secvență $h(x,i)$, $i = 0,1,2,\ldots$, de poziții în tabelă.
- Această secvență, numită și secvență de *încercări*, va fi cercetată ori de câte ori apare o coliziune. De exemplu, pentru operația de adăugare, se caută cel mai mic i pentru care locația de la adresa $h(x,i)$ este liberă.
- O metodă uzuală de definire a secvenței $h(x,i)$ constă dintr-o combinație liniară:

$$h(x,i) = (h_1(x) + c \cdot i) \bmod p$$

unde $h_1(x)$ este o funcție de dispersie, iar c o constantă.

- Se pot considera și forme pătratice:

$$h(x,i) = (h_1(x) + c_1 \cdot i + c_2 \cdot i^2) \bmod p$$

- O altă secvență de încercări des utilizată este următoarea:

$$h(x), h(x)-1, \ldots, 0, p-1, p-2, \ldots, h(x)+1$$

unde h este o funcție de dispersie.

Dispersie cu adresare deschisă - continuare

Teorema (2)

Pentru dispersia cu adresare deschisă, numărul mediu de încercări pentru o căutare fără succes este cel mult $\frac{1}{1-\beta}$ *(*$\beta < 1$*).*

Demonstrație.

Vom presupune că $h(x,0), h(x,1), h(x,2), \ldots, h(x,p-1)$ realizează o permutare a mulțimii $\{0,1,2,\ldots,p-1\}$. Aceasta asigură faptul că o inserare se realizează cu succes ori de câte ori există o poziție liberă în tabelă. Notăm cu X variabila aleatoare care întoarce numărul de încercări în poziții ocupate și cu p_i probabilitatea $Pr(X=i)$. Evident, dacă $i > n$ ($n = \#S$), atunci $p_i = 0$. Numărul mediu de încercări M în poziții ocupate este:

$$1+\sum_{i=0}^{n} i\cdot p_i = 1+\sum_{i=0}^{\infty} i\cdot p_i = 1+\sum_{i=0}^{\infty} i\cdot Pr(X=i) = 1+\sum_{i=0}^{\infty} i\cdot(Pr(X\geq i)-Pr(X\geq i+1)) =$$
$$1+\sum_{i=1}^{\infty} Pr(X\geq i) = 1+\sum_{i=1}^{\infty} q_i$$

Avem $q_1 = \frac{n}{p}$, $q_2 = \frac{n}{p}\cdot\frac{n-1}{p-1} \leq \left(\frac{n}{p}\right)^2, \ldots$ de unde rezultă $M = 1+\sum_{i=1}^{\infty}\beta^i = \frac{1}{1-\beta}$ □

Dispersie cu adresare deschisă - continuare

Corolar (Knuth, 1976)

Pentru dispersia cu adresare deschisă liniară, numărul mediu de încercări pentru o căutare cu succes este $\frac{1}{2}\left[1+\frac{1}{1-\beta}\right]$.

Arbori digitali (tries) - Introducere

- Până acum am studiat numai structuri de date pentru reprezentarea mulțimilor, astfel încât pentru operațiile de căutare, inserare și ștergere să existe algoritmi eficienți din punctul de vedere al complexității timp.
- În această secțiune vom studia o structură de date care, în anumite situații, reduce semnificativ spațiul de memorie necesar reprezentării unei mulțimi.
- Considerăm cazul unui dicționar. Se poate reduce spațiul necesar pentru memorarea dicționarului dacă pentru cuvintele cu aceleși prefix, acesta este reprezentat o singură dată.
- Definiția arborilor digitali are ca punct de plecare această idee.
- Un arbore digital este o structură de date care se bazează pe reprezentarea digitală (cu cifre) a elementelor din mulțimea univers.
- Denumirea de *tri* (în unele cărți *trie*) a fost dată de E. Fredkin (CACM, 3, 1960, pp. 490-500) și constituie o parte din expresia din limba engleză *information-re***tri***eval*.
- Un arbore digital este un arbore cu rădăcină ordonat k-ar (fiecare vârf are cel mult k succesori), unde k este numărul de cifre (litere dintr-un alfabet) necesare pentru reprezentarea elementelor mulțimii univers.
- Se presupune că toate elementele sunt reprezentate prin secvențe de cifre (litere) de aceeași lungime m. Astfel, mulțimea univers conține m^k elemente.

Exemplu de arbore digital

- Presupunem că elementele mulțimii univers sunt codificate prin secvențe de trei cifre din alfabetul $\{0,1,2\}$.
- Mulțimea de chei $S = \{121, 102, 211, 120, 210, 212\}$ este reprezentată prin arborele din figura 6.

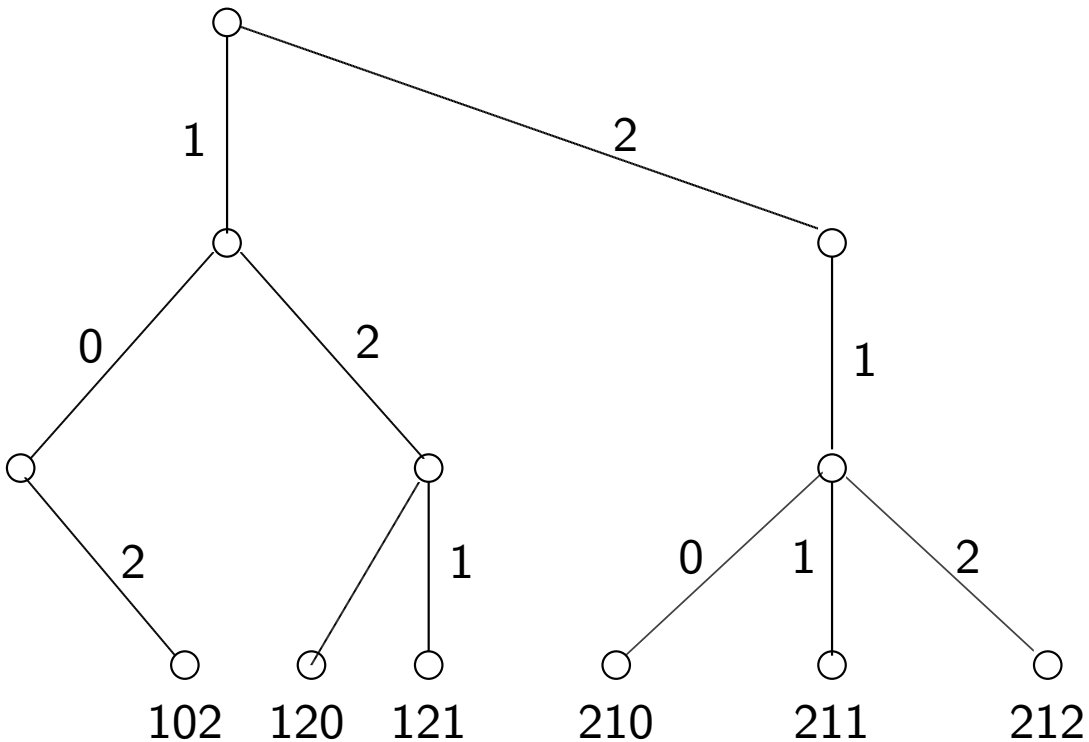


Figura 6 : Arbore digital

Exemplu de arbore digital - continuare

- Reprezentarea cuvintelor prin arbori digitali aduce o economie de memorie numai în cazul când există multe prefixe comune.
- În figura 7 este reprezentat un exemplu în care spațiul ocupat de arborele digital este mai mică decât cel ocupat de lista liniară a cuvintelor.

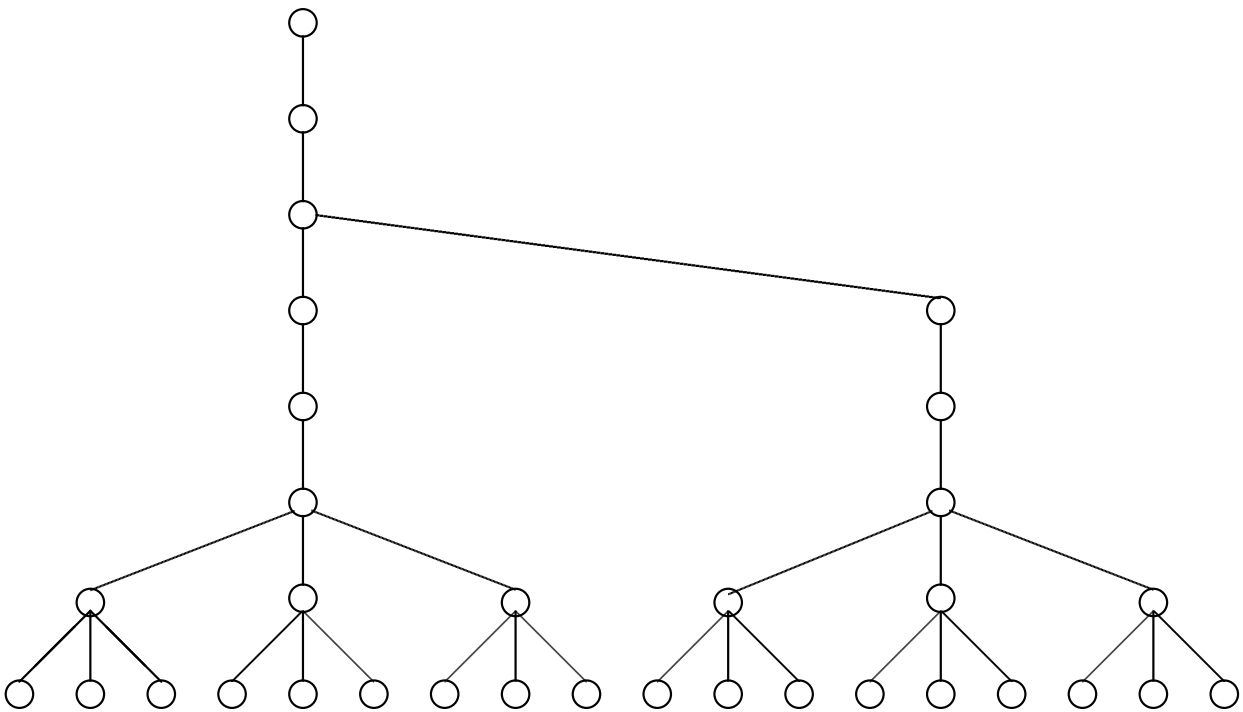


Figura 7 : Cazul când există multe prefixe comune

Structuri de date pentru reprezentarea arborilor digitali

- Un arbore tri poate fi reprezentat printr-o structură înlănțuită în care fiecare nod `v` are k câmpuri de legătură, $(\texttt{v.succ[j]} \mid 0 \leq j < k)$, ce memorează adresele fiilor lui v.
- Pentru simplitatea prezentării, presupunem că alfabetul este $\{0, \ldots, k-1\}$.
- Elementele din mulțimea S sunt chei, iar nodurile de pe frontieră memorează informațiile asociate acestor chei.

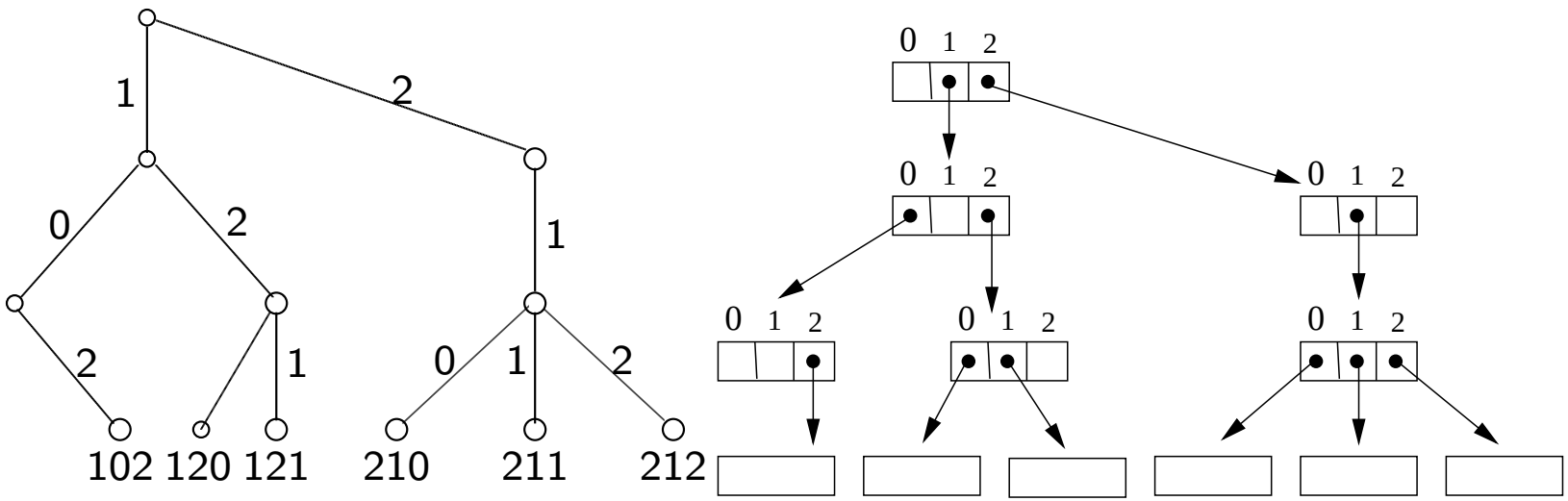


Figura 8 : Arborele tri din figura 6 și structura de date pentru reprezentare

Căutarea în arbori digitali

- Căutarea pentru un element a în structura `t` constă în încercarea de a parcurge drumul în arbore descris de secvența $(a[0], \ldots, a[m-1])$.
- Parcurgerea completă a drumului înseamnă căutare cu succes ($a \in S$), iar parcurgerea parțială are semnificația căutării fără succes.

```
function cautTri(a, m, t)
begin
    i ← 0
    p ← t
    while ((p ≠ NULL) and (i < m) do
        p ← p->succ[a[i]]
        i ← i+1
    return p
end
```

- Complexitatea timp pentru cazul cel mai nefavorabil este $O(m)$. De notat că aceasta nu depinde de numărul cuvintelor n.
- Pentru ca operația de căutare să fie mai eficientă decât căutarea binară trebuie ca $n > 2^m$.

Inserarea în arbori digitali

- Algoritmul care realizează operația de inserare a unui cuvânt a în structura `t` simulează parcurgerea drumului descris de secvența $(a[0], \ldots, a[m-1])$.
- Pentru acele componente $a[i]$ pentru care nu există noduri în `t` se va adăuga un nou nod ca succesor celui corespunzător lui $a[i-1]$.

Ștergerea în arbori digitali

- Considerăm un exemplu.
- Dacă din structura din figura 10 se șterge elementul reprezentat de 102, atunci este necesară și ștergerea a două noduri (cele corespunzătoare componentelor 0 și 2).
- Ștergerea elementului 210 presupune eliminarea unui singur nod (cel corespunzător lui 0).

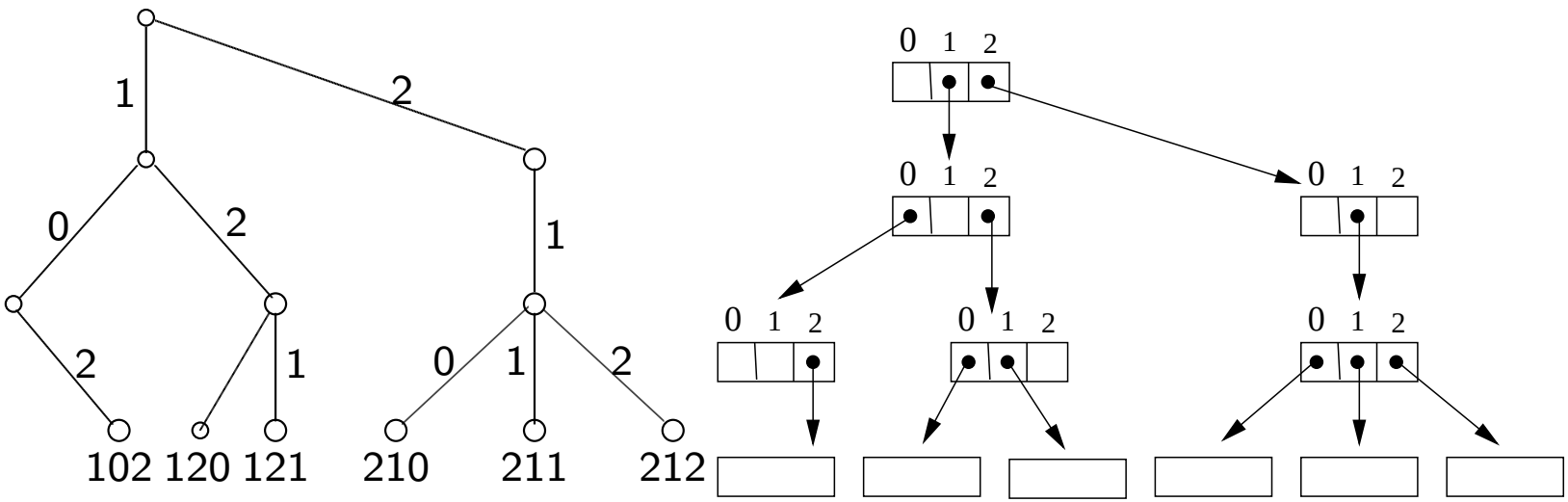


Figura 9 : Arborele tri din figura 6 și reprezentarea acestuia

Ștergerea în arbori digitali - continuare

- Un element care urmează a fi eliminat este împărțit în două:
  - un prefix care este comun și altor elemente care există în structură, și deci nu trebuie distrus și
  - un sufix care nu mai aparține niciunui element și deci poate fi șters.
- Strategia pe care o urmează algoritmul de ștergere este următoarea:
  1. se parcurge drumul descris de elementul *a* ce urmează a fi șters și se memorează acest drum într-o stivă;
  2. se parcurge acest drum înapoi, utilizând stiva, și dacă pentru un nod de pe acest drum toți succesorii sunt egali cu *null*, atunci se elimină acel nod.

Cazul cheilor cu lungimi diferite

- Există multe situații practice când cheile nu aceeași lungime.
- Exemplu: *un dicționar de sinonime*.
  - Cheile sunt cuvinte și, evident, acestea au lungimi diferite.
  - Este foarte posibil ca un cuvânt să fie prefix al altui cuvânt. În acest caz se pune problema evidențierii nodurilor care corespund cheilor.
- Soluția cea mai la îndemână este de a adăuga un nou câmp la structura unui nod.
- Noul câmp va avea următoarea semnificație:
  - dacă nodul corespunde unei chei, atunci câmpul va referi informația asociată cheii (de exemplu, lista sinonimelor);
  - altfel, câmpul va avea valoarea NULL.
- Algoritmii de căutare, inserare și eliminare se obțin din cei descriși în cazul cheilor de lungimi egale prin adăugarea operațiilor de testare și de actualizare a câmpului nou adăugat.

Exemplu de arbore digital ce memorează chei cu lungimi diferite

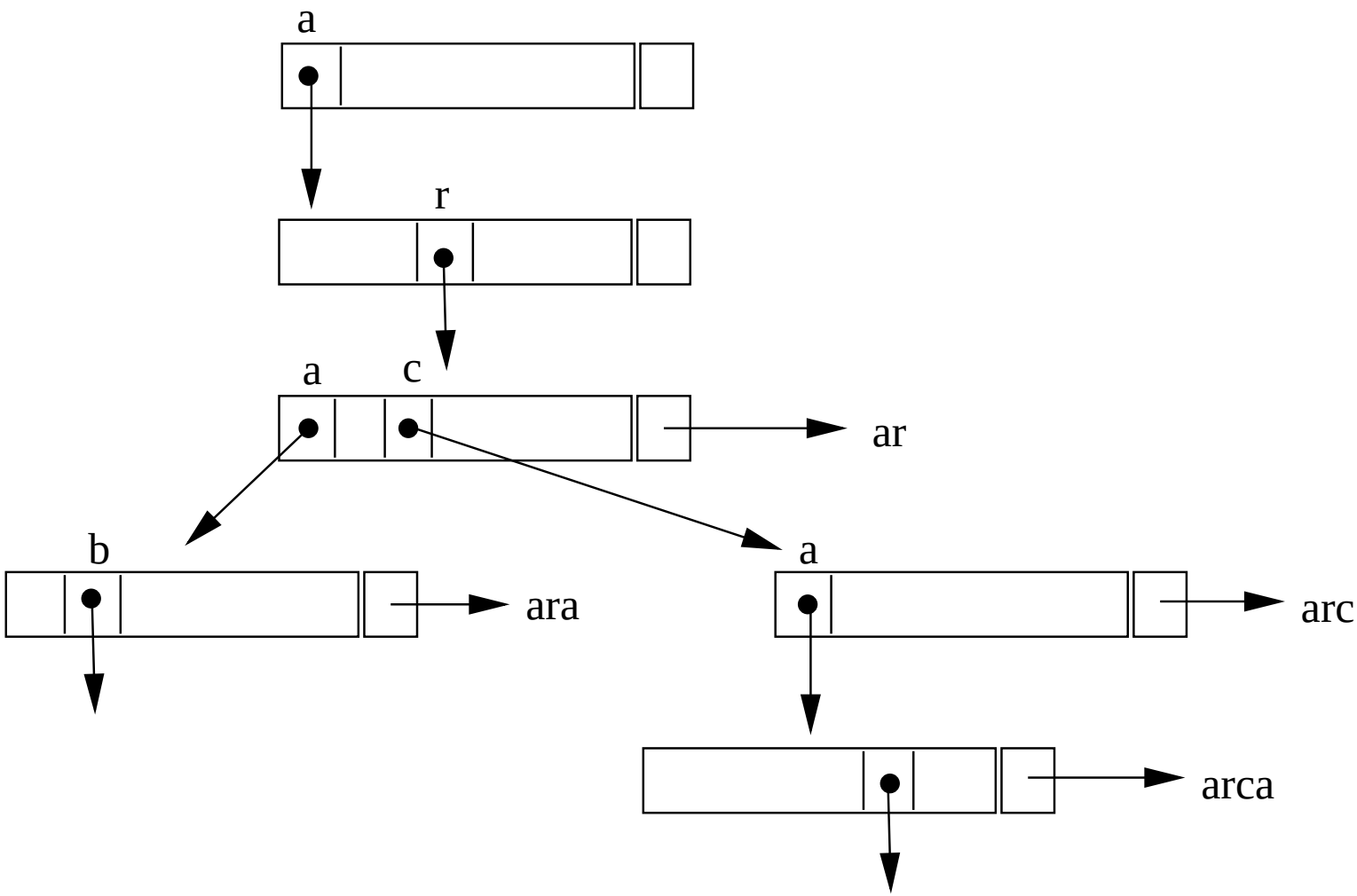


Figura 10 : Arbore digital ce memorează chei cu lungimi diferite

Proiectarea algoritmilor

Problema sortării

Mitică Craus

Cuprins

Sortare - Considerații generale

- Sortarea este una dintre problemele cele mai importante atât din punct de vedere practic, cât și teoretic.
- Cea mai generală formulare și mai des utilizată:

  Fie dată o secvență $(v_0, \ldots, v_{n-1})$ *cu componentele* v_i *dintr-o mulțime total ordonată. Problema sortării constă în determinarea unei permutări* π *astfel încât* $v_{\pi(0)} \leq v_{\pi(1)} \leq \cdots \leq v_{\pi(n-1)}$ *și în rearanjarea elementelor din secvență în ordinea dată de permutare.*

- O altă formulare:

  Fie dată o secvență de structuri statice $(R_0, \ldots, R_{n-1})$*, unde fiecare structură* R_i *are o valoare cheie* K_i*. Peste mulțimea cheilor* K_i *este definită o relație de ordine totală. Problema sortării constă în determinarea unei permutări* π *cu proprietatea* $K_{\pi(0)} \leq K_{\pi(1)} \leq \cdots \leq K_{\pi(n-1)}$ *și în rearanjarea structurilor în ordinea* $(R_{\pi(0)}, \ldots, R_{\pi(n-1)})$.

- Dacă secvența este memorată în memoria internă a calculatorului, atunci avem *sortare internă*.
- Dacă secvența este înregistrată într-un fișier memorat pe un suport extern, atunci avem *sortare externă*.
- În această lecție ne ocupăm numai de sortarea internă.

Considerații generale - contiunuare

- Vom simplifica formularea problemei presupunând că secvența dată este memorată într-un tablou unidimensional.
- Acum problema sortării poate fi formulată astfel:

  Intrare: *n și tabloul* $(\mathtt{a[i]} \mid i = 0, \ldots, n-1)$ *cu* $a[i] = v_i$, $i = 0, \ldots, n-1$.
  Ieșire: *tabloul* a *cu proprietățile:* $a[i] = w_i$ *pentru* $i = 0, \ldots, n-1$, $w_0 \leq \cdots \leq w_{n-1}$ *și* $(w_0, \ldots, w_{n-1})$ *este o permutare a secvenței* $(v_0, \ldots, v_{n-1})$*; convenim să notăm această proprietate prin* $(w_0, \ldots, w_{n-1}) = \mathrm{Perm}(v_0, \ldots, v_{n-1})$.

- Un algoritm de sortare este stabil dacă păstrează ordinea relativă inițială a elementelor egale.
- Există foarte mulți algoritmi care rezolvă problema sortării interne.
- Vom prezenta numai câteva metode pe care le considerăm cele mai semnificative.

Sortare bazată pe comparații

- Determinarea permutării se face comparând la fiecare moment două elemente $a[i]$ și $a[j]$ ale tabloului supus sortării.
- Scopul comparării poate fi diferit:
  - pentru a rearanja valorile celor două componente în ordinea firească (sortare prin interschimbare), sau
  - pentru a insera una dintre cele două valori într-o subsecvență ordonată deja (sortare prin inserție), sau
  - pentru a selecta o valoare ce va fi pusă pe poziția sa finală (sortare prin selecție).
- Decizia apartenenței unei metode la una dintre subclasele de mai sus are un anumit grad de subiectivitate.
- Exemplu: selecția naivă ar putea fi foarte bine considerată o metodă bazată pe interschimbare.

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 (*bubble-sort*) - 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 *buble-sort* - pseudocod

```
procedure Sort(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 *impar-par*

- Esența metodei *impar-par* constă în alternarea secvenței de operații `swap(a[i],a[i+1]), i=0,2,4,...` cu secvența `swap(a[i],a[i+1]), i=1,3,5,...`
- După cel mult $n/2$ execuții a fiecăreia din cele două secvențe de operații, elementele tabloului a vor fi sortate.
- Metoda nu conduce la un algoritm superior algoritmului `bubbleSort`.
- Algoritmul de sortare prin metoda *impar-par* are calitatea de a fi paralelizabil, datorită faptului că perechile care interacționează sunt disjuncte.

Algoritmul de sortare *impar-par* - pseudocod

```
procedure imparparSort(a, n)
    for faza ← 1 to n do
        if faza este impara
            then for i ← 0 to n-2 step 2 do
                        swap(a[i], a[i+1])
        if faza este para
            then for i ← 1 to n-2 step 2 do
                        swap(a[i], a[i+1])
end
```

Exemplu de execuție a algoritmului de sortare *impar-par*

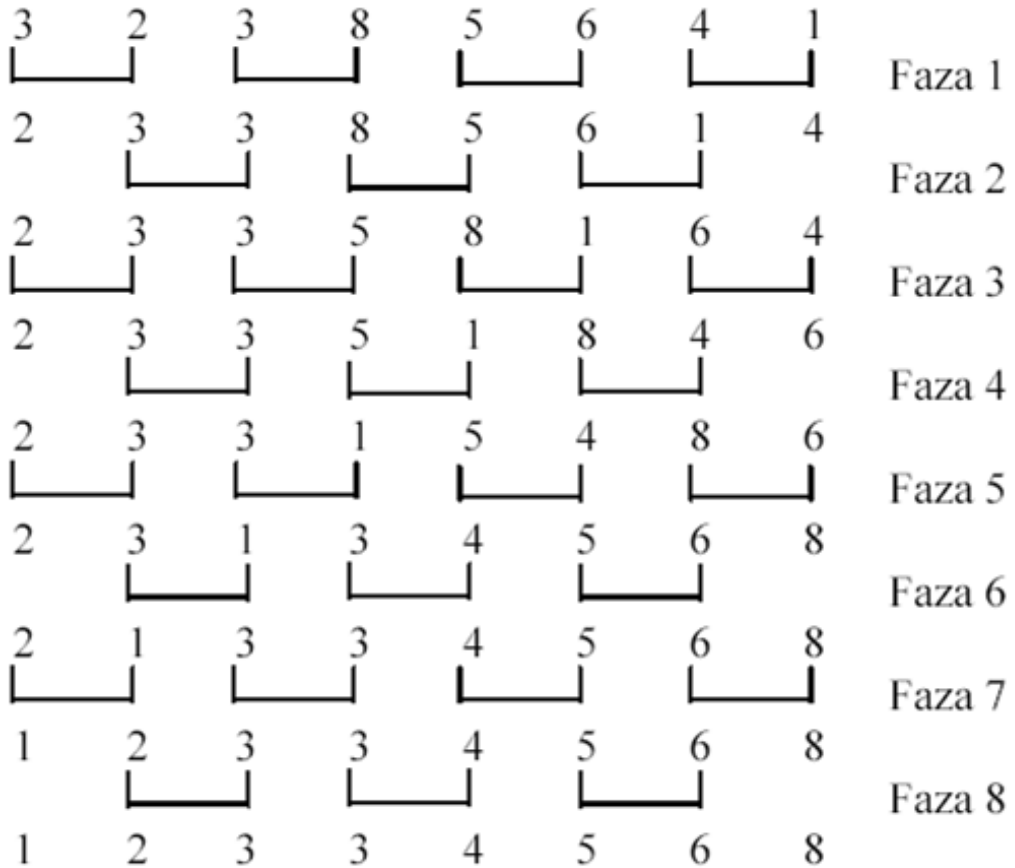


Figura 1 : Exemplu de execuție a algoritmului de sortare *impar-par* pentru $n = 8$

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 Sort(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 metoda lui Shell

- În algoritmul *insertionSort* elementele se deplasează numai cu câte o poziție o dată și prin urmare timpul mediu va fi proporțional cu n^2, deoarece fiecare element se deplasează în medie cu $n/3$ poziții în timpul procesului de sortare.
- Din acest motiv s-au căutat metode care să îmbunătățească inserția directă, prin mecanisme cu ajutorul cărora elementele fac salturi mai lungi, în loc de pași mici.
- O asemenea metodă a fost propusă în anul 1959 de Donald L. Shell, metodă pe care o vom mai numi *sortare cu micșorarea incrementului*.
- Exemplul următor ilustrează ideea generală care stă la baza metodei.

Sortare prin metoda lui Shell - exemplu

- Presupunem $n = 9$. Sunt executați următorii pași (figura 2):
  1. *Prima parcurgere a secvenței*. Se împart elementele în grupe de câte trei sau două elemente (valoarea incrementului $h_1 = 4$) care se sortează separat:

     $$(a[0], a[4], a[8]), \ldots, (a[3], a[7])$$

  2. *A doua parcurgere a secvenței*. Se grupează elementele în două grupe de câte trei elemente (valoarea incrementului $h_2 = 3$): $(a[0], a[3], a[6]) \ldots (a[2], a[5], a[8])$ și se sortează separat.
  3. *A treia trecere*. Acest pas termină sortarea prin considerarea unei singure grupe care conține toate elementele. În final, cele nouă elemente sunt sortate.

Sortare prin metoda lui Shell - exemplu (continuare)

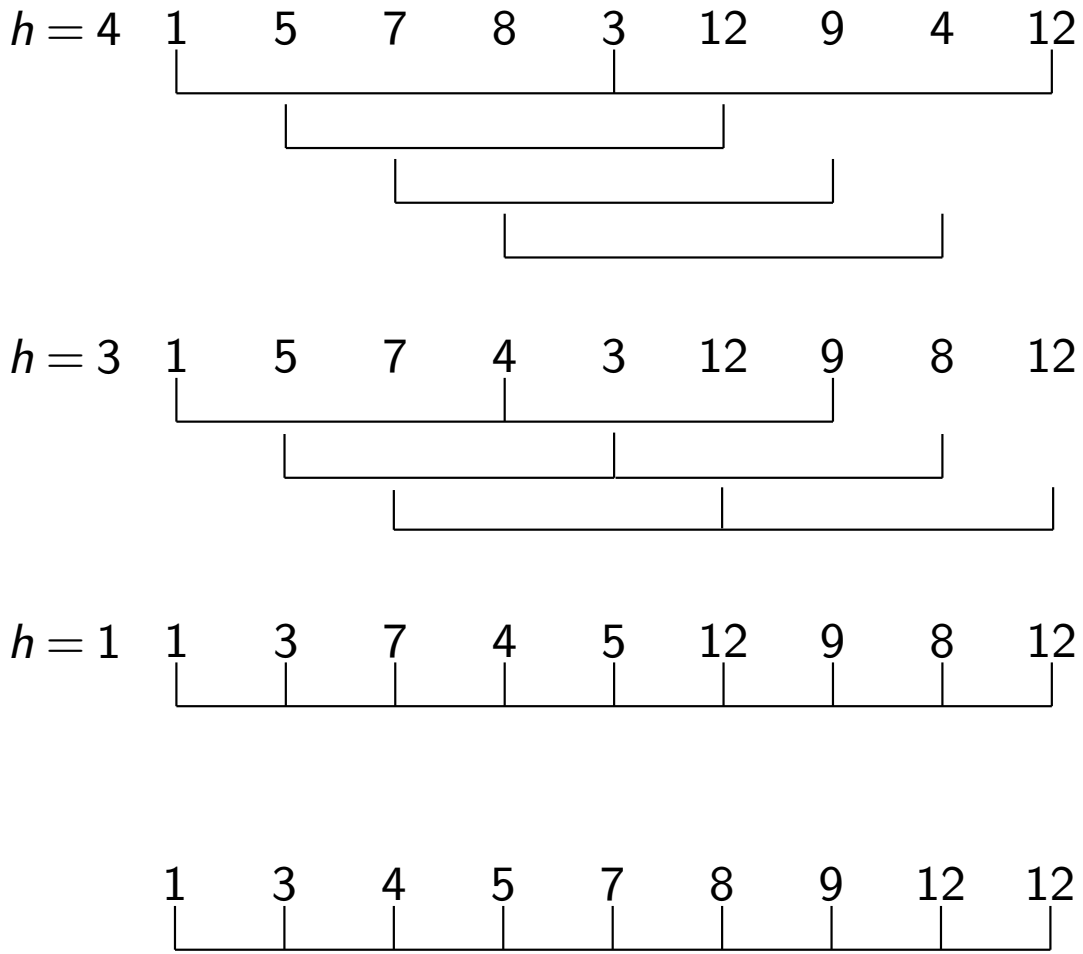


Figura 2 : Sortarea prin metoda lui Shell

Sortare prin metoda lui Shell - exemplu (continuare)

- Fiecare dintre procesele intermediare de sortare implică fie o sublistă nesortată de dimensiune relativ scurtă, fie una aproape sortată, astfel că inserția directă poate fi utilizată cu succes pentru fiecare operație de sortare.
- Prin inserțiie intermediare, elementele tind să conveargă rapid spre destinația lor finală.
- Secvența de incremente 4, 3, 1 nu este obligatorie; poate fi utilizată orice secvență $h_i > h_{i-1} > \cdots > h_0$, cu condiția ca ultimul increment h_0 să fie 1.

Algoritmul de sortare prin metoda lui Shell - pseudocod

- Presupunem că numărul de valori de incrementare este memorat de variabila `nIncr` și că acestea sunt memorate în tabloul $(\texttt{valIncr[h]} \mid 0 \leq k \leq nIncr - 1)$.

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

Evaluarea algoritmului - cazul cel mai favorabil și cazul cel mai nefavorabil

- Presupunem că elementele din secvență sunt diferite și dispuse aleator.
- Denumim operația de sortare corespunzătoare primei faze h_t-*sortare*, apoi h_{t-1}-*sortare* etc.
- O subsecvență pentru care $a[i] \leq a[i+h]$, pentru $0 \leq i \leq n-1-h$, este denumită *h-ordonată*.
- Considerăm pentru început cea mai simplă generalizare a inserției directe, și anume cazul când avem numai două valori de incrementare: $h_1 = 2$ și $h_0 = 1$.
- Cazul cel mai favorabil este obținut când secvența de intrare este ordonată crescător și sunt executate $2O(n) = O(n)$ comparații și nici o deplasare.
- Cazul cel mai nefavorabil, când secvența de intrare este ordonată descrescător, necesită $2O(1+2+3+(\frac{n}{2}-1))+O(1+2+3+(n-1)) = O(n^2)$ comparații și $2O(1+2+3+(\frac{n}{2}-1))+O(1+2+3+(n-1)) = O(n^2)$ deplasări.

Evaluarea algoritmului - cazul mediu

- În cea de-a doua parcurgere a tabloului a, avem o secvență 2-ordonată de elemente $a[0], a[1], \ldots, a[n-1]$.
- Este ușor de văzut că numărul de permutări $(i_0, i_1, \ldots i_{n-1})$ ale mulțimii $\{0, 1, \ldots, n-1\}$ cu proprietatea $i_k \leq i_{k+2}$, pentru $0 \leq k \leq n-3$, este $C_n^{[\frac{n}{2}]}$, deoarece obținem exact o permutare 2-ordonată pentru fiecare alegere de $[\frac{n}{2}]$ elemente care să fie puse pe poziții impare.

Evaluarea algoritmului - cazul mediu (continuare)

- Fiecare permutare 2-ordonată este egal posibilă după ce o subsecvență aleatoare a fost 2-ordonată.
- Determinăm numărul mediu de inversări între astfel de permutări.
- Fie A_n numărul total de inversări peste toate permutările 2-ordonate de $\{0, 1, \ldots, n-1\}$.
- Relațiile $A_1 = 0$, $A_2 = 1$, $A_3 = 2$ sunt evidente. Considerând cele șase cazuri 2-ordonate, pentru $n = 4$,

$$0\,1\,2\,3 \quad 0\,2\,1\,3 \quad 0\,1\,3\,2 \quad 1\,0\,2\,3 \quad 1\,0\,3\,2 \quad 2\,0\,3\,1$$

  vom găsi $A_4 = 0+1+1+2+3 = 8$.
- În urma calculelor, care sunt un pic dificile, se obține pentru A_n o formă destul de simplă:

$$A_n = \left[\frac{n}{2}\right] 2^{n-2}$$

- Numărul mediu de inversări într-o permutare aleatoare 2-ordonată este:

$$\frac{\left[\frac{n}{2}\right] 2^{n-2}}{C_n^{[\frac{n}{2}]}}$$

- După aproximarea lui Stirling, aceasta converge asimptotic către $\frac{\sqrt{\pi}}{128 n^{\frac{3}{2}}} \approx 0.15 n^{\frac{3}{2}}$.

Evaluarea algoritmului - cazul mediu (continuare)

Teorema (1)

Numărul mediu de inversări executate de algoritmul lui Shell pentru secvența de incremente $(2,1)$ *este* $O(n^{\frac{3}{2}})$.

Evaluarea algoritmului - continuare

Teorema (2)

Dacă $h \approx \left(\frac{16n}{\pi}\right)^{\frac{1}{3}}$, *atunci algoritmul lui Shell necesită timpul* $O(n^{\frac{5}{3}})$ *pentru cazul cel mai nefavorabil. (Knuth, 1976, pag. 89).*

Evaluarea algoritmului - continuare

Teorema (3)

Dacă secvența de incremente $h_{t-1}, \ldots, h_0$ *satisface condiția*

$$h_{s+1} \bmod h_s = 0 \text{ pentru } 0 \leq s < t-1,$$

atunci timpul de execuție pentru cazul cel mai nefavorabil este $O(n^2)$.

- O justificare intuitivă a teoremei de mai sus este următoarea:
  - Dacă $h_s = 2^s$, $0 \leq s \leq 3$, atunci o 8-sortare urmată de o 4-sortare, urmată de o 2-sortare nu permite nici o interacțiune între elementele de pe pozițiile pare și impare.
  - Fazei finale (1-sortaree) îi vor reveni $O(n^{\frac{3}{2}})$ inversări.
  - O 7-sortare urmată de o 5-sortare, urmată de o 3-sortare amestecă astfel lucrurile încât în faza finală (1-sortarea) nu vor fi mai mult de $2n$ inversări.

Teorema (4)

Timpul de execuție în cazul cel mai nefavorabil a algoritmului `ShellSort` *este* $O(n^{\frac{3}{2}})$, *atunci când* $h_s = 2^{s+1} - 1$, $0 \leq s \leq t-1, t = [\log_2 n] - 1$.

Sortare prin selecție

- Strategiile de sortare incluse în această clasă se bazează pe următoarea schemă :
  - La pasul curent se selectează un element din secvență și se plasează pe locul său final.
  - Procedeul continuă până când toate elementele sunt plasate pe locurile lor finale.
- După modul în care se face selectarea elementului curent, metoda poate fi mai mult sau mai puțin eficientă.

Sortare prin selecție naivă

- Este o metodă mai puțin 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 așezate elementele pe pozițiile 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 așezare este $0, 1, \ldots, n-1$.

Algoritmul de sortare prin selecție 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 execuție este $O(n^2)$ pentru toate cazurile, adică algoritmul `NaivSort` are timpul de execuție $\Theta(n^2)$.

Comparație între BubbleSort și NaivSort

.

- În sortarea prin metoda bulelor sunt efectuate mai puține comparații decât în cazul selecției naive,.
- Paradoxal, sortarea prin metoda bulelor este aproape de două ori mai lentă decât selecția naivă.
  - Explicație: În sortarea prin metoda bulelor sunt efectuate multe schimbări în timp ce selecția naivă implică o mișcare redusă a datelor.
- În tabelul din figura 3 sunt redați timpii de execuție (în sutimi de secunde) pentru cele două metode obținuți în urma a zece teste pentru $n = 1000$.

Nr. test	BubbleSort	NaivSort
1	71	33
2	77	27
3	77	28
4	94	38
5	82	27
6	77	28
7	83	32
8	71	33
9	71	39
10	72	33

Figura 3 : Compararea algoritmilor BubbleSort și NaivSort

Sortare prin selecție sistematică

- Se bazează pe structura de date de tip max-*heap*.
- Metoda de sortare prin selecție sistematică constă în parcurgerea a două etape:
  1. construirea pentru secvența curentă a proprietății MAX-HEAP(a);
  2. selectarea în mod repetat a elementului maximal din secvența curentă și refacerea proprietății MAX-HEAP pentru secvența rămasă.
- Definiție:

  a) Tabloul $(a[i] \mid i = 0, \ldots, n-1)$ are proprietatea MAX-HEAP, dacă $(\forall k)(1 \leq k < n \Rightarrow a\left[\frac{k-1}{2}\right] \geq a[k])$.
  - Notăm această proprietate prin MAX-HEAP(a).

  b) Tabloul a are proprietatea MAX-HEAP începând cu poziția ℓ dacă $(\forall k)(\ell \leq \frac{k-1}{2} < k < n \Rightarrow a\left[\frac{k-1}{2}\right] \geq a[k])$.
  - Notăm această proprietate cu MAX-HEAP(a, ℓ).

Etapa 1. Construirea pentru secvența curentă a proprietății MAX-HEAP(a)

- Tabloul a are lungimea n. Inițial, are loc MAX-HEAP(a, $\frac{n}{2}$).
- Dacă are loc MAX-HEAP(a, $\ell+1$) atunci se procedează la introducerea lui $a[\ell]$ în grămada deja construită a[$\ell+1$..n-1], astfel încât să obținem MAX-HEAP(a, ℓ).
- Procesul se repetă până când ℓ devine 0.

```
procedure intrInGr(a, n, ℓ)
    j ← ℓ
    esteHeap ← false
    while ((2*j+1 ≤ n-1) and not esteHeap)
        k ← j*2+1
        if ((k < n-1) and (a[k] < a[k+1])
            then k ← k+1
        if (a[j] < a[k])
            then swap(a[j], a[k])
            else esteHeap ← true
        j ← k
end
```

Etapa 2: Selectarea în mod repetat a elementului maximal din secvența curentă și refacerea proprietății MAX-HEAP pentru secvența rămasă

- Deoarece inițial avem MAX-HEAP(a), rezultă că pe primul loc se găsește elementul maximal din $a[0..n-1]$.
- Punem acest element la locul său final prin interschimbarea $a[0] \leftrightarrow a[n-1]$. Acum $a[0..n-2]$ are proprietatea MAX-HEAP începând cu 1.
- Refacem MAX-HEAP pentru această secventă prin introducerea lui $a[0]$ în grămada $a[0..n-2]$, după care punem pe locul său final cel de-al doilea element cel mai mare din secvența de sortat.
- Procedeul continuă până când toate elementele ajung pe locurile lor finale.

Algoritmul de sortare prin selecție sistematică - pseudocod

```
procedure heapSort(a,n)
    n1 ← ⌊(n−1)/2⌋
    for ℓ ← n1 downto 0 do
        intrInGr(a, n, ℓ)
    r ← n-1
    while (r ≥ 1) do
        swap(a[0], a[r])
        intrInGr(a, r, 0)
        r ← r-1
end
```

Evaluarea algoritmului

- Considerăm $n = 2^k - 1$.
- În faza de construire a proprietății MAX-HEAP pentru toată secvența de intrare sunt efectuate următoarele operații:
  - se construiesc vârfurile de pe nivelurile $k-2, k-3, \ldots$;
  - pentru construirea unui vârf de pe nivelul i se vizitează cel mult câte un vârf de pe nivelurile $i+1, \ldots, k-1$;
  - la vizitarea unui vârf sunt executate două comparații.

  Rezultă că numărul de comparații executate în prima etapă este cel mult:

$$\sum_{i=0}^{k-2} 2(k-i-1)2^i = (k-1)2 + (k-2)2^2 + \cdots + 1 \cdot 2^{k-1} = 2^{k+1} - 2(k+1)$$

- În etapa a II-a, dacă presupunem că $a[r]$ se găsește pe nivelul i, introducerea lui $a[0]$ în grămada $\mathtt{a[1..r]}$ necesită cel mult $2i$ comparații. Deoarece i ia succesiv valorile $k-1, k-2, \ldots, 1$, rezultă că în această etapă numărul total de comparatii este cel mult:

$$\sum_{i=1}^{k-1} 2i2^i = (k-2)2^{k+1} + 4$$

Evaluarea algoritmului - continuare

- Numărul total de comparații este cel mult

$$\begin{aligned} C(n) &= 2^{k+1} - 2(k+1) + (k-2)2^{k+1} + 4 \\ &= 2^{k+1}(k-1) - 2(k-1) \\ &= 2k(2^k - 1) - 2(2^k - 1) \\ &= 2n\log_2 n - 2n \end{aligned}$$

- Complexitatea algoritmului `heapSort` este $O(n \log n)$.

Sortare prin numărare clasică

- Presupunem că tipul `TElement`, peste care sunt definite secvențele supuse sortării, conține numai două elemente: 0 și 1.
- Sortarea unui tablou a : `array[1..n] of TElement` se face astfel:
  - Se determină numărul n_1 de elemente din a egale cu 0 și numărul n_2 de elemente egale cu 1. Aceasta se poate face în timpul $\Theta(n)$.
  - Apoi se pun în tablou n_1 elemente 0 urmate de n_2 elemente 1. Și cea de-a doua etapă necesită $\Theta(n)$ timp.
- Rezultă un algoritm de sortare cu timpul de execuție $\Theta(n)$.
- Obținerea unui algoritm de sortare cu timp de execuție liniar a fost posibilă datorită faptului că s-au cunoscut informații suplimentare despre elementele supuse sortării, anume că mulțimea univers conține numai două elemente.

Sortare prin numărare clasică - continuare

- Algoritmul de sortare prin numărare se utilizează atunci când mulțimea univers conține puține elemente.
- Presupunem că $\mathbb{U} = \{0, 1, \ldots, k-1\}$ cu relația de ordine naturală.
- În prima etapă a algoritmului va fi calculat vectorul p: $p[i]$ = numărul de elemente din a mai mici decât sau egale cu i, $i = 0, 1, \ldots k-1$.
- În etapa a doua a algoritmului tabloul a va fi parcurs de la dreapta la stânga și va fi păstrată invariantă următoarea proprietate:
  $\mathscr{P}$: $p[i]$ va fi poziția ultimului element i transferat din a în tabloul final b,
  $i = k-1, k-2, \ldots 0$.
- Propritetatea $\mathscr{P}$ poate fi păstrată prin decrementarea componentei din p imediat după ce elementul corespunzător din a a fost transferat.

Algoritmul de sortare prin numărare - pseudocod

```
procedure sortNumarare(a, b, n)
    for i ← 0 to k-1 do
        p[i] ← 0
    for j ← 0 to n-1 do
        p[a[j]] ← p[a[j]]+1
    for i ← 1 to k-1 do
        p[i] ← p[i-1] + p[i]
    for j ← n-1 downto 0 do
        i ← a[j]
        b[p[i]-1] ← i
        p[i] ← p[i]-1
end
```

Evaluarea algoritmului

- Determinarea tabloului `p` necesită $O(k+n)$ timp, iar transferul $O(n)$ timp.
- Algoritmul `sortNumarare` are complexitatea timp $O(n+k)$.
- În practică, algoritmul este aplicat pentru $k = O(n)$, caz în care rezultă un timp de execuție liniar a pentru `sortNumarare`.

Sortare prin metoda "bingo"

(K. Berman, J. Paul, "Algorithms: Sequential, Parallel and Distributed", Thomson Learning, 2005)

- Numărul elementelor secvenței de sortat este n.
- Numărul elementelor distincte este m, $1 \leq m \leq n$.
- Exemplu: sortarea unei liste de adrese poștale după codul poștal.
- Complexitatea: $O(nm)$.
- Daca $m \leq \log n$, algoritmul `bingoSort` este la fel de performant ca și `heapSort` și `mergeSort`.
- Ideea este de plasa, în fiecare iterație, una din valorile distincte pe pozițiile finale.
- Valoarea care este obiectul procesării într-o iterație se numeste "bingo".

Algoritmul de sortare prin metoda 'bingo" - pseudocod

```
procedure bingoSort(a, n)
    MaxMin(a,n,valMax,valMin)
    bingo ← valMin
    urm ← 0
    urmBingo ← valMax
    while (bingo<valMax) do
        posStart ← urm
        for i ← posStart to n-1 do
            if (A[i]=bingo)
                then interschimba(A[i],A[urm])
                    urm ← urm+1
                else if A[i]<urmBingo then urmBingo ← A[i]
        bingo ← urmBingo
        urmBingo ← valMax
end
```

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 4.

Sortarea cuvintelor - continuare

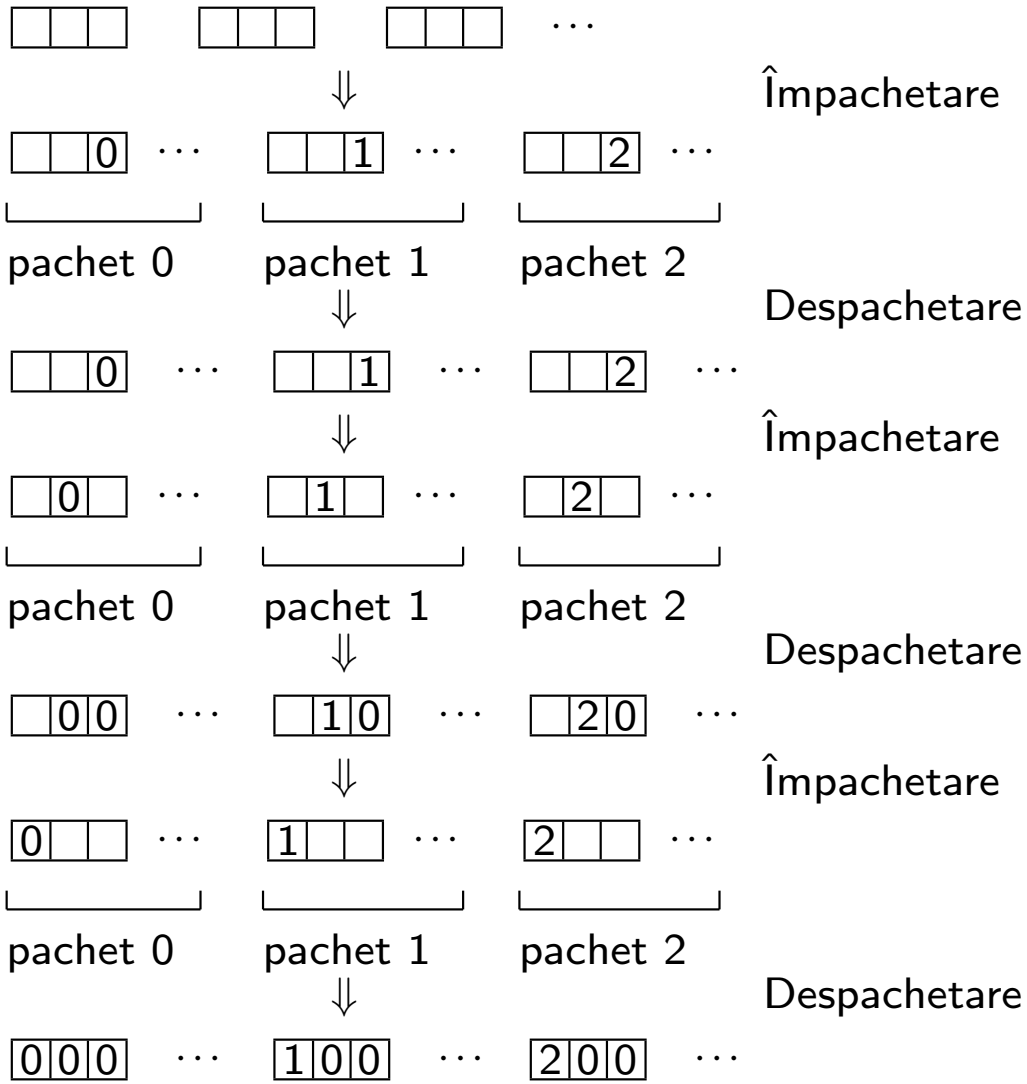


Figura 4 : Sortare prin distribuire

Algoritmul de sortare distribuire - 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 de sortare distribuire - 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 $O(m \cdot n)$.

Sortare topologică - considerații generale

- Se aplică la secvențe cu elemente din mulțimi parțial ordonale.
- Exemplu de relație de ordine este parțială: $a_1 < a_0, a_1 < a_2 < a_3$.
- Problema constă în a crea o listă liniară care să fie compatibilă cu relația de ordine, adică, dacă $a_i < a_j$, atunci a_i va precede pe a_j în lista finală.
- Pentru exemplul nostru, lista liniară finală va putea fi (a_1, a_0, a_2, a_3), sau (a_1, a_2, a_0, a_3), sau (a_1, a_2, a_3, a_0).

Definitia

Fie $(S, \leq)$ o mulțime parțial ordonată finită și $a = (a_0, a_1, \ldots, a_{n-1})$ o liniarizare a sa. Spunem că secvența a este *sortată topologic*, dacă $\forall i, j : a_i < a_j \Rightarrow i < j$.

Sortare topologică - considerații generale (continuare)

Teorema (1)

Orice mulțime parțial ordonată finită $(S, \leq)$ poate fi sortată topologic.

Demonstrație.

Vom extinde relația $\leq$ la o relație de ordine totală.
Fie elementele distincte $a, b \in S$ ce nu pot fi comparate, *i.e.*, nu are loc nici $a < b$ și nici $b < a$.
Extindem relația $<$, considerând $x < y$ pentru orice x cu $x \leq a$ și orice y cu $b \leq y$.
Procedeul continuă până când sunt eliminate toate perechile incomparabile.
O mulțime total ordonată poate fi sortată, iar secvența sortată respectă ordinea parțială. □

Grafuri și digrafuri

- Un *graf* este o pereche $G = (V, E)$, unde V este o mulțime ale cărei elemente sunt numite *vârfuri*, iar E este o mulțime de perechi neordonate $\{u, v\}$ de vârfuri, numite *muchii*.
- Considerăm numai cazul când V și E sunt finite.
- Dacă $e = \{u, v\}$ este o muchie, atunci u și v se numesc extremitățile muchiei e; mai spunem că e este *incidentă* în u și v sau că vârfurile u și v sunt *adiacente (vecine)*.
- Dacă G conține muchii de forma $\{u, u\}$, atunci o asemenea muchie se numește *buclă*, iar graful se numește *graf general* sau *pseudograf*.
- Un *digraf* este o pereche $D = (V, A)$, unde V este o mulțime de vârfuri, iar A este o mulțime de perechi ordonate (u, v) de vârfuri, numite arce.
- Considerăm numai cazul când V și A sunt finite.
- Dacă $a = (u, v)$ este un arc, atunci spunem că u este *extremitatea inițială* (*sursa*) a lui a și că v este *extremitatea finală* (*destinația*) lui a; mai spunem că u este un *predecesor imediat* al lui v și că v este un *succesor imediat* al lui u.
- Formulări echivalente: a este *incident din* u și *incident în (spre)* v, sau a este *incident cu* v *spre interior* și a este *incident cu* u *spre exterior*, sau a *pleacă din* u *și sosește în* v.

Grafuri și digrafuri

- Orice graf $G = (V, E)$ poate fi reprezentat ca un digraf $D = (V, A)$ considerând pentru fiecare muchie $\{i, j\} \in E$ două arce $(i, j), (j, i) \in A$.
- Cu aceste reprezentări, operațiile tipului Graf pot fi exprimate cu ajutorul celor ale tipului Digraf. Astfel, inserarea/ștergerea unei muchii este echivalentă cu inserarea/ștergerea a două arce.

Reprezentarea digrafurilor prin matricea de adiacență

- Digraful D este reprezentat printr-o structură cu trei câmpuri:
  - `D.n` = numărul de vârfuri,
  - `D.m` = numărul de arce și
  - un tablou bidimensional `D.a` de dimensiune $n \times n$ astfel încât:

$$D.a[i,j] = \begin{cases} 0 & \text{dacă } (i,j) \notin A, \\ 1 & \text{dacă } (i,j) \in A. \end{cases}$$

  pentru $i,j = 0,\ldots,n-1$.

- Dacă `D` este reprezentarea unui graf, atunci matricea de adiacență este simetrică:

$$D.a[i,j] = D.a[j,i], \text{ pentru orice } i,j.$$

- În loc de valorile 0 și 1 se pot considera valorile booleene *false* și respectiv *true*.

Reprezentarea digrafurilor prin liste de adiacență

- Reprezentarea prin liste de adiacență exterioară:
  - Digraful D este reprezentat printr-o structură asemănătoare cu cea de la matricele de adiacență.
  - Matricea de adiacență este înlocuită cu un tablou unidimensional de n liste liniare, implementate prin liste simplu înlănțuite și notate cu `D.a[i]` pentru $i = 0, \ldots, n-1$.
  - Lista `D.a[i]` conține vârfurile destinatare ale arcelor care pleacă din i (= lista de adiacență exterioară).
- Reprezentarea prin liste de adiacență interioară:
  - Lista `D.a[i]` conține vârfurile surse ale arcelor care sosesc în i (= lista de adiacență interioară).

Exemplu de reprezentare a grafurilor prin liste de adiacență

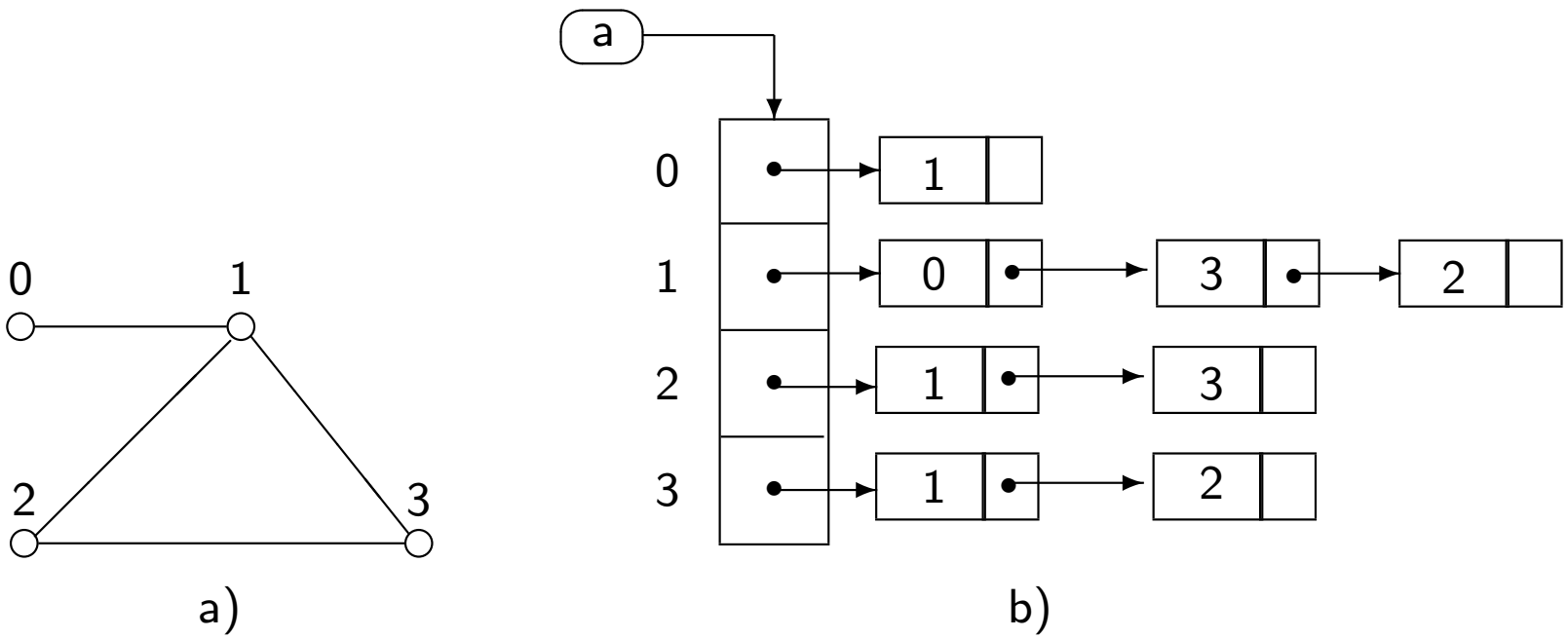


Figura 5 : Graf reprezentat prin liste de adiacență înlănțuite

Exemplu de reprezentare a digrafurilor prin liste de adiacență exterioară

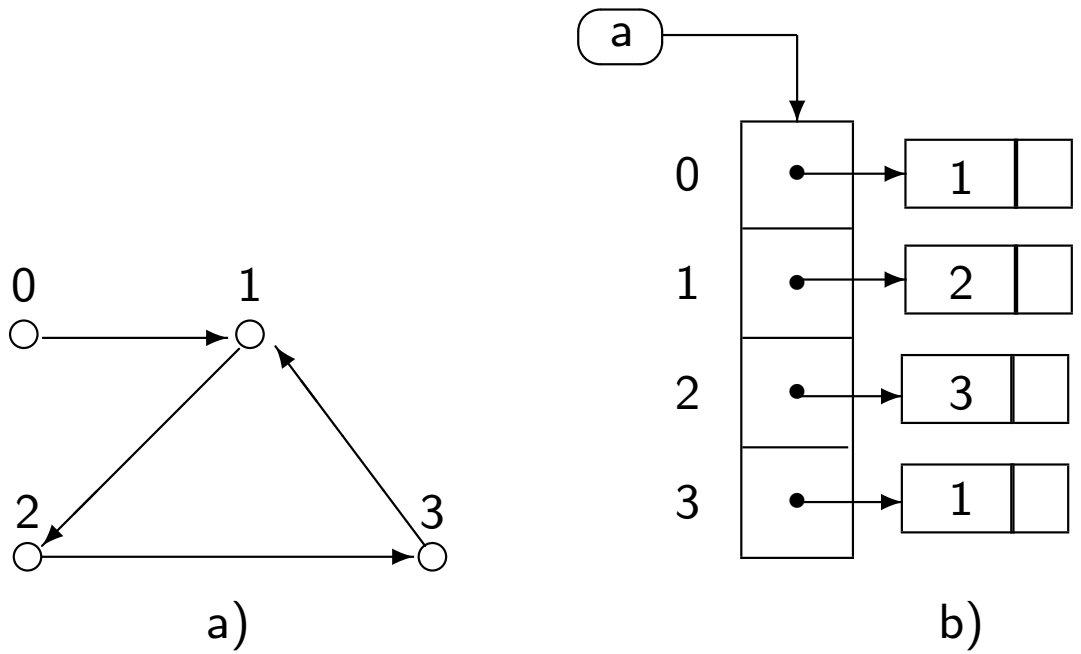


Figura 6 : Digraf reprezentat prin liste de adiacență exterioară înlănțuite

Explorarea DFS (Depth First Search) și BFS (Breadth First Search

- Explorarea DFS:

```
procedure DFS(D, i0, viziteaza(), S)
   for i ← 0 to D.n-1 do
       p[i]  ← D.a[i].prim
       S[i]  ← 0
   Sa ← stivaVida() /* Sa = mulţimea vârfurilor active */
   viziteaza(i0)
   S[i0] ← 1
   push(Sa, i0)
   while (not esteStivaVida(Sa)) do
      i ← top(Sa)
      if (p[i] = NULL)
         then pop(Sa)
         else j ← p[i]->elt
              p[i] ← p[i]->succ
              if S[j] = 0
                 then viziteaza(j)
                      S[j] ← 1
                      push(Sa, j)
end
```

- Explorarea BFS se obține prin reprezentarea mulțimii *Sa* printr-o coadă.

Exemplu de explorare

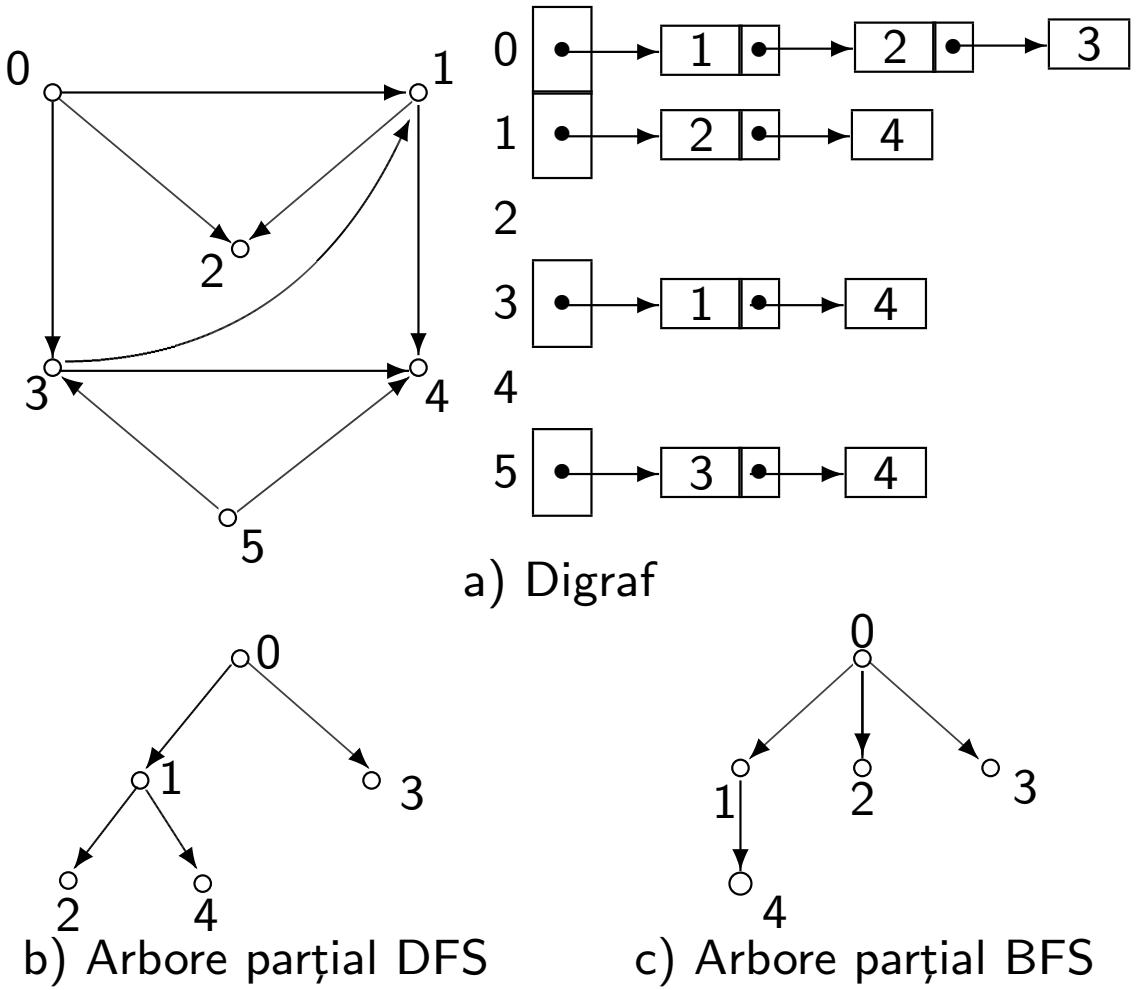


Figura 7 : Explorarea unui digraf

Mulțimi parțial ordonate finite și digrafuri aciclice

- Există o legătură strânsă între mulțimile parțial ordonate finite și digrafurile aciclice (digrafuri fără circuite numite pe scurt dag-uri).
- Orice mulțime parțial ordonată $(S, \leq)$ definește un dag $D = (S, A)$, unde există arc de la a la b, dacă $a < b$ și nu există $c \in S$ cu proprietatea $a < c < b$.
- Reciproc, orice dag $D = (V, A)$ definește o relație de ordine parțială $\leq$ peste V, dată prin: $u \leq v$, dacă există un drum de lungime ≥ 0 de la u la v.
- De fapt, $\leq$ este închiderea reflexivă și tranzitivă a lui A (se mai notează $\leq = A^*$).
- Sortarea topologică a unui dag constă într-o listă liniară a vârfurilor astfel încât dacă există arc de la u la v, atunci u precede pe v în listă, pentru oricare două vârfuri u și v.
- Vârfurile care candidează pentru primul loc în lista sortată topologic au proprietatea că nu există arce incidente spre interior (care sosesc în acel vârf) și se numesc *surse*.

Sortare topologică - metoda DFS

- Reamintim că în timpul explorării DFS, un vârf poate fi întâlnit de mai multe ori.
- Notăm cu $f[v]$ momentul când vârful v este întâlnit ultima dată (când lista de adiacență este epuizată).
- Descrierea algoritmului:
  1. Apelează algoritmul DFS pentru a determina momentele de terminare $f[v]$ pentru orice vârf v.
  2. De fiecare dată când un vârf este terminat este adăugat la începutul unei listei înlănțuite.
  3. Lista înlănțuită finală va conține o sortare topologică a vârfurilor.
- Corectitudinea algoritmului se bazează pe următorul rezultat.

Lema (1) (T. Cormen, C. Leiserson, R. Rivest, (2000), Introducere în algoritmi, Computer Libris Agora, Cluj)

Un digraf D este aciclic dacă și numai dacă explorarea DFS nu produce arce înapoi.

Sortare topologică - metoda BFS

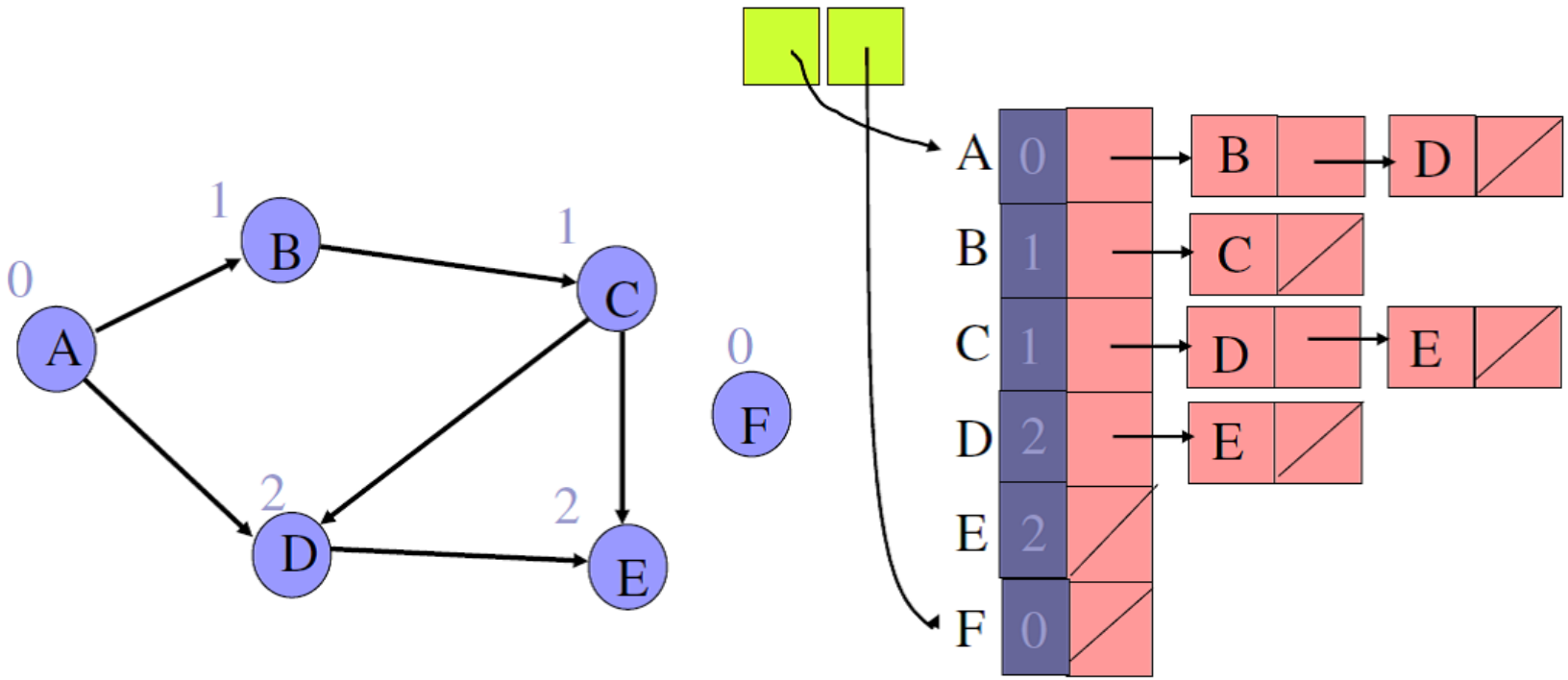
- Presupunem că pentru dag-ul D sunt create atât listele de adiacență interioară, cât și cele de adiacență exterioară.
- Listele de adiacență interioară vor fi utilizate la determinarea vârfurilor sursă (vârfuri fără predecesori); acestea au listele de adiacență interioară vide.
- Descrierea algoritmului:
  1. Inițializează coada cu vârfurile sursă.
  2. Extrage un vârf u din coadă pe care-l adaugă la lista sortată parțial.
  3. Elimină din reprezentarea (acum parțială) a lui D vârful u și toate arcele (u, v).
  4. Dacă pentru un astfel de arc lista de adiacență interioară a vârfului v devine vidă, atunci v va fi adăugat la coadă.
  5. Repetă pașii 2-4 până când coada devine vidă.

Sortare topologică - algoritm BFS

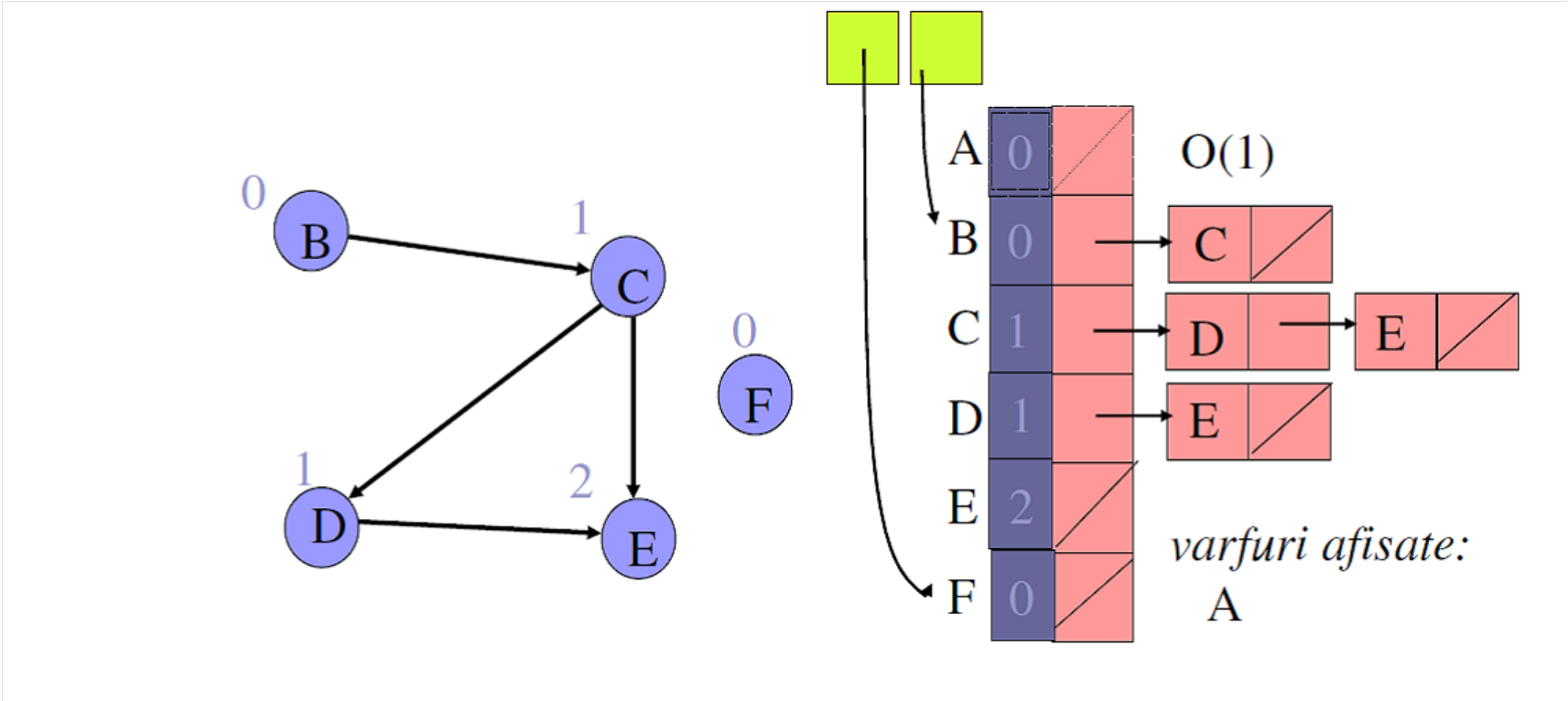
- Extindem structura `D`, care reprezintă digraful D, cu tabloul `np[1..n]`.
- `D.np[u]` conține numărul predecesorilor vârfului u.
- `L` este lista care conține varfurile digrafului D în ordine topologică.

```
procedure sortareTopologicaBFS(D,np)
   coadaVida(C) //initializeaza coada C
   // insereaza in C varfurile fara predecesori
   for u ← 0 to D.n-1 do
       if D.np[u]=0 then insereaza(C,u)
   // construieste lista varfurilor (afiseaza) in ordine topologica
   for k ← 0 to D.n-1 do
       if esteVida(C)
          then return ("Graful contine cicluri")
       u ← elimina(C)
       inserează(L, u) // inserarea se face la sfarsitul listei
       p ← D.a[u]
       while p≠NULL do
          v ← p->elt //v este un succesor imedial al lui u
          D.np[v] ← D.np[v]-1
          if D.np[v]=0
             then insereaza(C,v)
          p ← p->succ
end
```

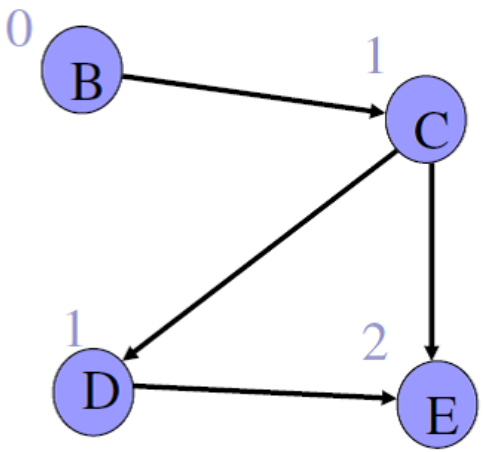
Exemplu de execuție a algoritmului de sortare toplogică prin metoda BFS

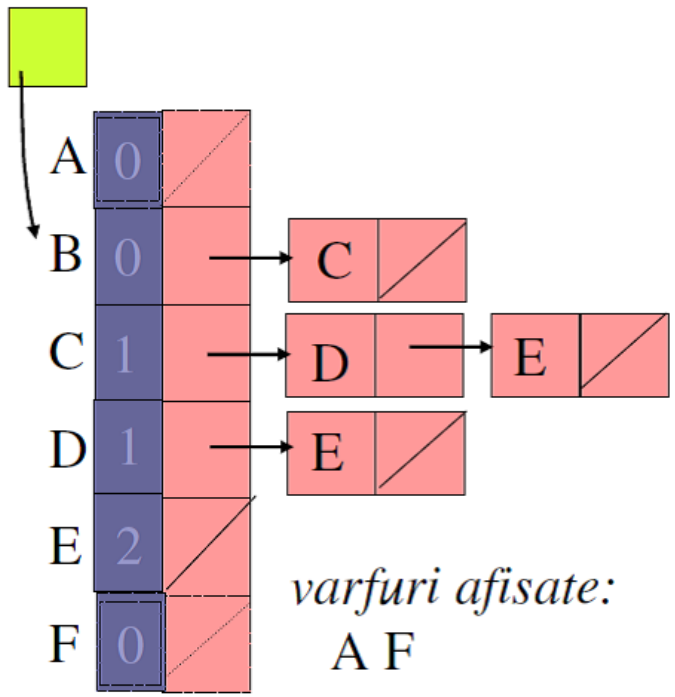


Exemplu de execuție a algoritmului de sortare toplogică prin metoda BFS

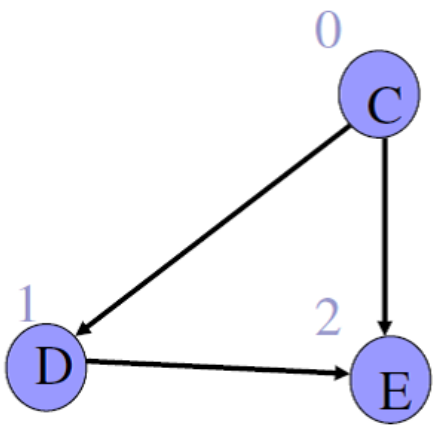


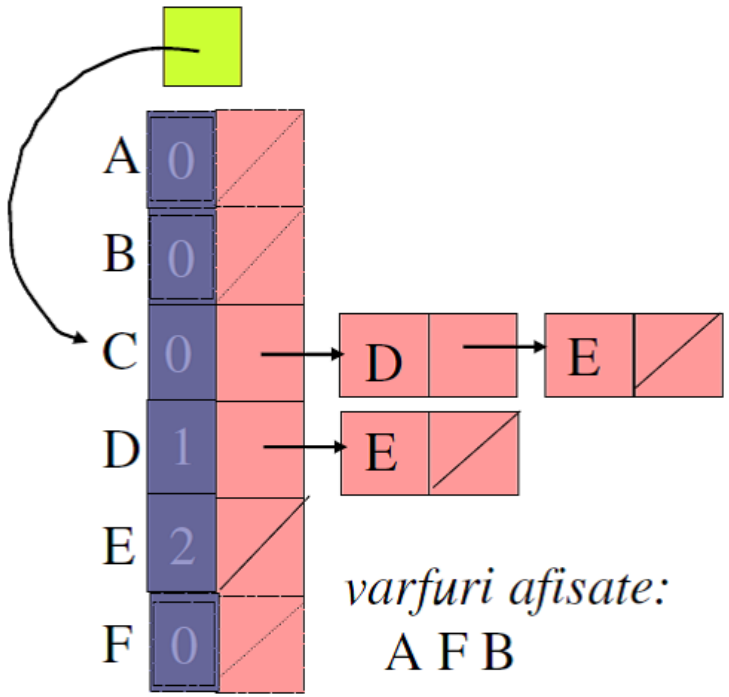
Exemplu de execuție a algoritmului de sortare toplogică prin metoda BFS



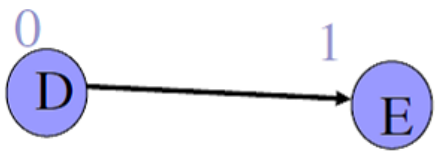


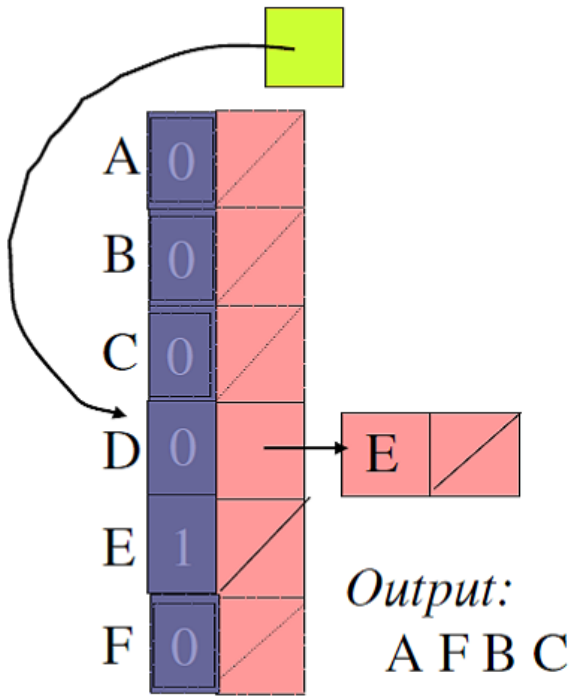
Exemplu de execuție a algoritmului de sortare toplogică prin metoda BFS



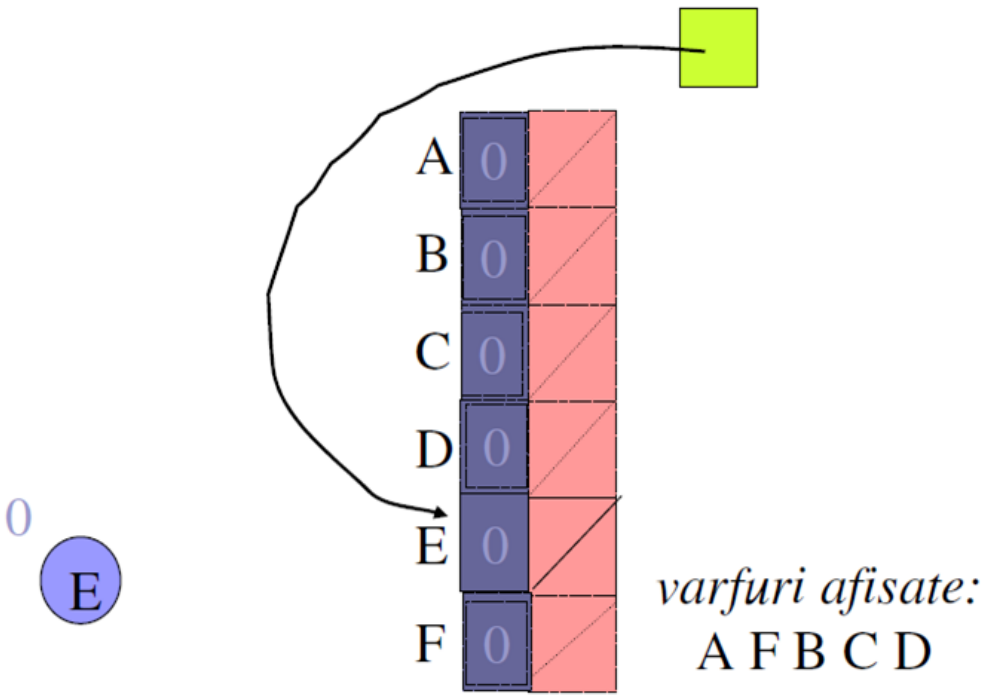


Exemplu de execuție a algoritmului de sortare toplogică prin metoda BFS



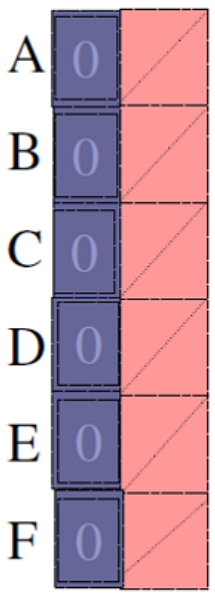


Exemplu de execuție a algoritmului de sortare toplogică prin metoda BFS



Exemplu de execuție a algoritmului de sortare toplogică prin metoda BFS

FINAL!



varfuri afisate:
A F B C D E

Complexitatea timp

- Compusă din timpul pentru:
  - identificarea vârfurilor fără predecesori: $O(n)$
  - ștergerea muchiilor: $O(m)$
  - afișarea vârfurilor: $O(n)$
- Timp total : $O(n+m)$

Proiectarea algoritmilor

Paradigma *divide-et-impera* **(în engleză** *divide-and-conquer***)**

Mitică Craus

Univeristatea Tehnică "Gheorghe Asachi" din Iași

Cuprins

Paradigma *divide-et-impera* - descriere

- Paradigma *divide-et-impera* (în engleză *divide-and-conquer*) constă în divizarea problemei inițiale în două sau mai multe subprobleme de dimensiuni mai mici, apoi rezolvarea în aceeași manieră (recursivă) a subproblemelor și combinarea soluțiilor acestora pentru a obține soluția problemei inițiale.
- Divizarea unei probleme se face până când se obțin subprobleme de dimensiuni mici ce pot fi rezolvate prin tehnici elementare.

Paradigma *divide-et-impera* - descriere (continuare)

```
procedure divideEtImpera(P, n, S)
    if (n ≤ n0)
        then rezolvă subproblema P prin tehnici elementare
        else împarte P in P_1,...,P_a de dimensiuni n_1,...,n_a
             divideEtImpera(P_1,n_1,S_1)
             ...
             divideEtImpera(P_a,n_a,S_a)
             combină S_1,...,S_a pentru a obţine S
end
```

Paradigma *divide-et-impera* - descriere (continuare)

- Deoarece descrierea strategiei are un caracter recursiv, aplicarea ei trebuie precedată de o generalizare de tipul problemă $\mapsto$ subproblemă prin care dimensiunea problemei devine o variabilă liberă.
- Vom presupune că dimensiunea n_i a subproblemei P_i satisface $n_i \leq \frac{n}{b}$, unde $b > 1$.
- În acest fel pasul de divizare reduce o subproblemă la altele de dimensiuni mai mici, ceea ce asigură proprietatea de terminare a subprogramului recursiv.
- De asemenea, descrierea recursivă ne va permite utilizarea inducției pentru demonstrarea corectitudinii.

Evaluarea timpului de execuție

- Presupunem că divizarea problemei în subprobleme și asamblarea soluțiilor necesită timpul $O(n^k)$.
- Timpul de execuție $T(n)$ a algoritmului `divideEtImpera` este dat de următoarea relație de recurență:

$$T(n) = \begin{cases} O(1) & \text{, dacă } n \leq n_0 \\ a \cdot T(\frac{n}{b}) + O(n^k) & \text{, dacă } n > n_0 \end{cases} \quad (1)$$

Evaluarea timpului de execuție - continuare

Teorema (1)

Dacă $n > n_0$ atunci:

$$T(n) = \begin{cases} O(n^{\log_b a}) & ,dacă\ a > b^k \\ O(n^k \log_b n) & ,dacă\ a = b^k \\ O(n^k) & ,dacă\ a < b^k \end{cases} \quad (2)$$

Demonstrație.

Fără să restrângem generalitatea presupunem $n = b^m \cdot n_0$. De asemenea mai presupunem că $T(n) = cn_0^k$ dacă $n \leq n_0$ și $T(n) = aT(\frac{n}{b}) + cn^k$ dacă $n > n_0$. Pentru $n > n_0$ avem:

$$\begin{aligned} T(n) &= aT(\tfrac{n}{b}) + cn^k \\ &= aT(b^{m-1}n_0) + cn^k \\ &= a(aT(b^{m-2}n_0) + c\left(\frac{n}{b}\right)^k) + cn^k \\ &= a^2T(b^{m-2}n_0) + c\left[a(\tfrac{n}{b})^k + n^k\right] \\ &= \cdots \end{aligned}$$

Evaluarea timpului de execuție - continuare

Demonstrație.

Partea a doua:

$$\begin{aligned} T(n) &= a^m T(n_0) + c\left[a^{m-1}\left(\frac{n}{b^{m-1}}\right)^k + \cdots + a\left(\frac{n}{b}\right)^k + n^k\right] \\ &= a^m c n_0^k + c\left[a^{m-1} b^k n_0^k + \cdots + a\left(b^{m-1}\right)^k n_0^k + \left(b^m\right)^k n_0^k\right] \\ &= c n_0^k a^m \left[1 + \frac{b^k}{a} + \cdots + \left(\frac{b^k}{a}\right)^m\right] \\ &= c a^m \sum_{i=0}^{m} \left(\frac{b^k}{a}\right)^i \end{aligned}$$

unde am renotat cn_0^k prin c.

Distingem cazurile:

1. $a > b^k$. Seria $\sum_{i=0}^{m} \left(\frac{b^k}{a}\right)^i$ este convergentă și deci șirul sumelor parțiale este convergent. De aici rezultă $T(n) = O(a^m) = O(a^{\log_b n}) = O(n^{\log_b a})$.
2. $a = b^k$. Rezultă $a^m = b^{km} = cn^k$ și de aici $T(n) = O(n^k m) = O(n^k \log_b n)$.
3. $a < b^k$. Avem $T(n) = O(a^m(\frac{b^k}{a})^m) = O(b^{km}) = O(n^k)$.

Sortare prin interclasare (Merge Sort) - descriere

- Considerăm cazul cândsecvența ce urmează a fi sortată este memorată într-un tablou unidimensional.
- Prin interclasarea a două secvențe sortate se obține o secvență sortată ce conține toate elementele secvențelor de intrare.
- Ideea este de a utiliza interclasarea în etapa de asamblare a soluțiilor:
  - În urma rezolvării recursive a subproblemelor rezultă secvențe ordonate și prin interclasarea lor obținem secevența finală sortată.
- Primul pas constă în generalizarea problemei.
  - Presupunem că se cere sortarea unei secvențe memorate într-un tablou a[p..q] cu p și q variabile libere, în loc de sortarea unei secvențe memorate într-un tablou a[0..n − 1] cu n variabilă legată (deoarece este dată de intrare).
- Divizarea problemei constă în împărțirea secevței de sortat în două subsecvențe a[p..m] și $a[m+1..q]$, de preferat de lungimi aproximativ egale (de exemplu $m = \left[\frac{p+q}{2}\right]$.
- Faza de combinare a soluțiilor constă în interclasarea celor două subsecvențe, după ce ele au fost sortate recursiv prin același procedeu.

Merge Sort - pseudocod

```
procedure mergeSort(a, p, q)
    if (p < q)
        then m ← ⌈(p+q)/2⌉
             mergeSort(a, p, m)
             mergeSort(a, m+1, q)
             interclasare(a, p, q, m, temp)
             for i ← p to q do
                 a[i] ← temp[i-p]
end
```

Merge Sort - pseudocod

```
procedure interclasare(a, p, q, m, temp)
    i ← p
    j ← m+1
    k ← -1
    while ((i ≤ m) and (j ≤ q)) do
        k ← k+1
        if (a[i] ≤ a[j])
            then temp[k] ← a[i]
                 i ← i+1
            else temp[k] ← a[j]
                 j ← j+1
    while (i ≤ m) do
        k ← k+1
        temp[k] ← a[i]
        i ← i+1
    while (j ≤ n) do
        k ← k+1
        temp[k] ← a[j]
        j ← j+1
end
```

Evaluarea algoritmului

- Divizare a problemei în subprobleme se face în timpul constant ($O(1)$).
- Asamblare a soluțiilor: Interclasarea a două secvențe ordonate crescător se face în timpul $O(m_1 + m_2)$, unde m_1 și m_2 sunt lungimile celor două secvențe.
- Complexitatea timp a algoritmului: Se aplică teorema 1 pentru $a = 2, b = 2, k = 1$. Rezultă pentru algoritmul `mergeSort` timpul $O(n \log_2 n)$.
- Complexitatea spațiu: Algoritmul utilizează $O(n + \log_2 n)$ memorie suplimentară (tabloul auxiliar și stiva cu apelurile recursive).

Algoritmul lui Batcher de sortare bitonică - descriere

- Autor: Batcher; Anul publicării: 1968.
- Operația de bază este sortarea unei secvențe bitone.
- Esența problemei sortării unei secvențe bitone este transformarea sortării unei secvențe de bitone lungime n în sortarea a doua secvențe bitone de dimensiune $\frac{n}{2}$.
- Pentru a sorta o secvență de n elemente, prin tehnica sortării unei secvențe bitone, trebuie să dispunem de o secvență bitonă formată din n elemente.
- Observații:
  - Două elemente formează o secvență bitonă.
  - Orice secvență nesortată este o concatenare de secvențe bitone de lungime 2.
- Ideea transformării unei secvențe oarecare în una bitonă: combinarea a două secvențe bitone de lungime $\frac{n}{2}$ pentru a obține o secvență bitonă de lungime n.
- Algoritmul este paralelizabil.

Secvențe bitone

- Secvența bitonă este o secvență de elemente $[a_0, a_1, \ldots, a_{n-1}]$ pentru care
  - există i astfel încât $[a_0, a_1, \ldots, a_i]$ este monoton crescătoare și $[a_{i+1}, \ldots, a_{n-1}]$ este monoton descrescătoare sau
  - există o permutare circulară astfel încât să fie satisfacută condiția anterioară.
- Exemple:
  - [1, 2, 4, 7, 6, 0]; întâi crește și apoi descrește; i = 3.
  - [8, 9, 2, 1, 0, 4]: după o permutare circulară la stânga cu 4 poziții rezultă [0, 4, 8, 9, 2, 1]; i = 3.
- Fie $S = [a_0, a_1, \ldots, a_{n-1}]$ o secventă bitonă,
  - $S_1 = [\min\{a_0, a_{\frac{n}{2}}\}, \min\{a_1, a_{\frac{n}{2}+1}\}, \ldots \min\{a_{\frac{n}{2}-1}, a_{n-1}\}]$ și
  - $S_2 = [\max\{a_0, a_{\frac{n}{2}}\}, \max\{a_1, a_{\frac{n}{2}+1}\}, \ldots \max\{a_{\frac{n}{2}-1}, a_{n-1}\}]$
- Secvențele S_1 si S_2 au proprietățile următoare:
  - Sunt bitone.
  - Fiecare element din S_1 este mai mic decit fiecare element din S_2.

Algoritmul lui Batcher de sortare bitonică - pseudocod

- *Notații*:
  - a[0..n − 1] este un tablou unidimensional de dimensiune n.
  - (a, i, d) definește segmentul a[i]..a[i + d − 1] din tabloul a.
  - s este un parametru binar care specifică ordinea crescătoare ($s = 0$) sau descrescătoare ($s = 1$) a cheilor de sortare.
  - compara_si_schimba(a[i], a[i + 1], s) desemnează sortarea a două elemente x si y ordinea indicată de parametrul s.
- *Premise*: Inițial, a[0..n − 1] conține secvența de sortat.
- *Apel*: `BatcherSort(a, 0, n, 0)` sau `BatcherSort(a, 0, n, 1)`.

```
procedure BatcherSort(a, i, d, s)
   if (d = 2)
      then (a[i], a[i + 1] ← compara_si_schimba(a[i], a[i + 1], s)
      else /* sortarea crescătoare a unei secvenţe S de lungime d/2 */
           BatcherSort(a, i, d/2, 0)
           /*sortarea descrescătoare a secvenţei S', care urmează lui S,
           de lungime d/2 */
           BatcherSort(a, i + d/2, d/2, 1)
           /*sortarea secvenţei bitone SS', de lungime d */
           sortare_secventa_bitona(a, i, d, s)
end
```

Sortarea unei secvențe bitone - pseudocod

- *Premise*: Inițial, segmentul a[i..i + d − 1] conține o secventă bitonă de lungime d;

```
procedure sortare_secventa_bitona(a, i, d, s)
   if (d = 2)
      then (a[i], a[i + 1]) ← compara_si_schimba(a[i], a[i + 1], s)
      else /*Construirea secvențelor bitone S1 și S2, de lungime d/2*/
           for j ← 0 to d/2-1 do
               (a[i+j], a[i+j+d/2]) ← compara_si_schimba(a[i+j], a[i+j+d/2], s)
           /*sortarea secvenţei bitone S1, de lungime d/2 */
           sortare_secventa_bitona(a, i, d/2, s)
           /*sortarea secvenţei bitone S2, de lungime d/2 */
           sortare_secventa_bitona(a, i + d/2, d/2, s)
end
```

Exemplu de sortare a unei secvențe bitone

Original sequence	3	5	8	9	10	12	14	20	95	90	60	40	35	23	18	0
1st Split	3	5	8	9	10	12	14	0	95	90	60	40	35	23	18	20
2nd Split	3	5	8	0	10	12	14	9	35	23	18	20	95	90	60	40
3rd Split	3	0	8	5	10	9	14	12	18	20	35	23	60	40	95	90
4th Split	0	3	5	8	9	10	12	14	18	20	23	35	40	60	90	95

Copyright ©1994 Benjamin/Cummings Publishing Co.

Figura 1 : Exemplu de sortare a unei secvențe bitone

Corectitudinea

Lema (2)

Dacă algoritmul lui Batcher sortează orice secvență de chei de sortare binare, atunci sortează orice secvență de chei de sortare numere reale oarecare.

Demonstrație.

Fie $f : \mathrm{R} \to \mathrm{R}$ o funcție monotonă. Astfel, $f(a_i) \leq f(a_j)$ dacă și numai dacă $a_i \leq a_j$. Evident, dacă algoritmul lui Batcher transformă secvența $[a_1, a_2, \ldots, a_n]$ în secvența $[b_1, b_2, \ldots, b_n]$, atunci va transforma secvența $[f(a_1), f(a_2), \ldots, f(a_n)]$ în secvența $[f(b_1), f(b_2), \ldots, f(b_n)]$. Astfel, dacă în secvența $[b_1, b_2, \ldots, b_n]$ există un indice i pentru care $b_i > b_{i+1}$, atunci în secvența $[f(b_1), f(b_2), \ldots, f(b_n)]$ vom avea $f(b_i) > f(b_{i+1})$.

Fie acum f o funcție monotonă definită astfel: $f(b_j) = \begin{cases} 0 & ,\text{dacă } b_j < b_i \\ 1 & ,\text{dacă } b_j \geq b_i \end{cases}$

În aceste condiții, secvența $[f(b_1), f(b_2), \ldots, f(b_n)]$ va fi o secventă binară nesortată deoarece $f(b_i) = 1$ și $f(b_{i+1}) = 0$. Rezultă că algoritmul lui Batcher eșuează în sortarea secvenței binare $[f(a_1), f(a_2), \ldots, f(a_n)]$. Deci, dacă algoritmul lui Batcher eșuează în sortarea unei secvențe de chei de sortare numere reale oarecare, atunci există o secvența binară care nu va fi sortată în urma aplicăriii algoritmului lui Batcher. □

Corectitudinea -continuare

Teorema (4)

Algoritmul lui Batcher sorteaza cele n elemente ale secvenței memorate în tabloul $A[0..n-1]$, *în ordinea crescătoare* $(s=0)$ *respectiv descrescătoare* $(s=1)$ *a cheilor de sortare.*

Demonstrație.

Este suficient să demonstrăm corectitudinea procedurii `sortare_secventa_bitona` pentru cazul binar (Lema 2). Procedăm prin inducție după lungimea d a secvențelor procesate.

Dacă $d=2$, evident procedura `sortare_secventa_bitona` transformă secvența inițială $S_{init}=A[i..i+d-1]$ într-o secvență sortată S_{fin}.

Vom demonstra că procedura `sortare_secventa_bitona` transformă o secvență binară S_{init} de tipul $0^r1^t0^v$ sau $1^r0^t1^v(r+t+v=d)$ într-o secvență S_{fin} sortată în ordinea indicată de valoarea lui $s,(\forall)d\geq 2$

Pasul paralel 6 transformă secvența S_{init} într-o secvență S_{temp} conform figurilor 2 și 3.

Se observă că în toate cazurile, secvența rezultată S_{temp}, este formată din două sub-secvențe bitone S^1_{temp} și S^2_{temp}, fiecare de lungime $\frac{d}{2}$. O secvență este de tipul S_{init} iar cealaltă secvență conține numai cifre 0 sau numai cifre 1. Dacă $s=0$, cheia maximă din S^1_{temp} este mai mică sau egală cu cheia minimă din S^2_{temp}. Dacă $s=1$, cheia minimă din S^1_{temp} este mai mare sau egală cu cheia maximă din S^2_{temp}.

Conform ipotezei de inducție, procedura `sortare_secventa_bitona` transformă secvențele S^1_{temp} și S^2_{temp} în două secvențe S^1_{fin} și S^2_{fin}, sortate crescător $(d=0)$ sau descrescător $(s=1)$. Dacă $s=0$, cheia maximă din S^1_{fin} este mai mică sau egală cu cheia minimă din S^2_{fin} deci secvența $S^1_{fin}S^2_{fin}$ este crescătoare. Daca $s=1$, cheia minimă din S^1_{fin} este mai mare sau egală cu cheia maximă din S^2_{fin} deci secvența $S^1_{fin}S^2_{fin}$ este descrescătoare. □

Corectitudinea -continuare

s	$p+q(p,q \leq \frac{d}{2})$	Secvența inițială	Secvența rezultată
0	$\leq \frac{d}{2}$	$0^{\frac{d}{2}-p}1^p1^q0^{\frac{d}{2}-q}$	$0^{\frac{d}{2}}1^q0^{\frac{d}{2}-(p+q)}1^p$
0	$\leq \frac{d}{2}$	$0^{\frac{d}{2}}0^k1^{p+q}0^m$	$0^{\frac{d}{2}}0^k1^{p+q}0^m$
0	$\leq \frac{d}{2}$	$0^k1^{p+q}0^m0^{\frac{d}{2}}$	$0^{\frac{d}{2}}0^k1^{p+q}0^m$
0	$\leq \frac{d}{2}$	$1^p0^{\frac{d}{2}-p}0^{\frac{d}{2}-q}1^q$	$0^{\frac{d}{2}}1^p0^{\frac{d}{2}-(p+q)}1^q$
0	$\leq \frac{d}{2}$	$1^{\frac{d}{2}}1^p0^{\frac{d}{2}-(p+q)}1^q$	$1^p0^{\frac{d}{2}-(p+q)}1^q1^{\frac{d}{2}}$
0	$\leq \frac{d}{2}$	$1^p0^{\frac{d}{2}-(p+q)}1^q1^{\frac{d}{2}}$	$1^p0^{\frac{d}{2}-(p+q)}1^q1^{\frac{d}{2}}$
0	$> \frac{d}{2}$	$0^{\frac{d}{2}-p}1^p1^q0^{\frac{d}{2}-q}$	$0^{\frac{d}{2}-p}1^{(p+q)-\frac{d}{2}}0^{\frac{d}{2}-q}1^{\frac{d}{2}}$
0	$> \frac{d}{2}$	$1^p0^{\frac{d}{2}-p}0^{\frac{d}{2}-q}1^q$	$0^{\frac{d}{2}-q}1^{(p+q)-\frac{d}{2}}0^{\frac{d}{2}-p}1^{\frac{d}{2}}$
1	$\leq \frac{d}{2}$	$0^{\frac{d}{2}-p}1^p1^q0^{\frac{d}{2}-q}$	$1^q0^{\frac{d}{2}-(p+q)}1^p0^{\frac{d}{2}}$
1	$\leq \frac{d}{2}$	$0^{\frac{d}{2}}0^k1^{p+q}0^m$	$0^k1^{p+q}0^m0^{\frac{d}{2}}$
1	$\leq \frac{d}{2}$	$0^k1^{p+q}0^m0^{\frac{d}{2}}$	$0^k1^{p+q}0^m0^{\frac{d}{2}}$
1	$\leq \frac{d}{2}$	$1^p0^{\frac{d}{2}-p}0^{\frac{d}{2}-q}1^q$	$1^p0^{\frac{d}{2}-(p+q)}1^q0^{\frac{d}{2}}$
1	$\leq \frac{d}{2}$	$1^{\frac{d}{2}}1^p0^{\frac{d}{2}-(p+q)}1^q$	$1^{\frac{d}{2}}1^p0^{\frac{d}{2}-(p+q)}1^q$
1	$\leq \frac{d}{2}$	$1^p0^{\frac{d}{2}-(p+q)}1^q1^{\frac{d}{2}}$	$1^{\frac{d}{2}}1^p0^{\frac{d}{2}-(p+q)}1^q$
1	$> \frac{d}{2}$	$0^{\frac{d}{2}-p}1^p1^q0^{\frac{d}{2}-q}$	$1^{\frac{d}{2}}0^{\frac{d}{2}-p}1^{(p+q)-\frac{d}{2}}0^{\frac{d}{2}-q}$
1	$> \frac{d}{2}$	$1^p0^{\frac{d}{2}-p}0^{\frac{d}{2}-q}1^q$	$1^{\frac{d}{2}}0^{\frac{d}{2}-q}1^{(p+q)-\frac{d}{2}}0^{\frac{d}{2}-p}$

Figura 2 : Corectitudinea

Corectitudinea -continuare

s=0

0000111111	1110000000	0000000000	1110111111
0000000000	0011111000	0000000000	0011111000
0011111000	0000000000	0000000000	0011111000
1111110000	0000000011	0000000000	1111110011
1111111111	1100000111	1100000111	1111111111
1100000111	1111111111	1100000111	1111111111
0011111111	1111000000	0011000000	1111111111
1111111000	0000001111	0000001000	1111111111

s=1

0000111111	1110000000	1110111111	0000000000
0000000000	0011111100	0011111100	0000000000
0011111100	0000000000	0011111100	0000000000
1111110000	0000000111	1111110111	0000000000
1111111111	1111000111	1111111111	1111000111
1111000111	1111111111	1111111111	1111000111
0111111111	1110000000	1111111111	0110000000
1111100000	0011111111	1111111111	0011100000

Figura 3 : Corectitudinea

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:

$$\forall i' : p \leq i' < i \Rightarrow a[i'] \leq x \tag{3}$$

și

$$\forall j' : j < j' \leq q \Rightarrow a[j'] \geq x \tag{4}$$

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 (3).
  2. $a[j] \geq x$. Transformarea $\texttt{j} \leftarrow \texttt{j} - 1$ păstrează proprietatea (4).
  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 (3) și (4).
- 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 quickSort(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
```

Evaluarea algoritmului în cazul cel mai nevaforabil

- 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".
- Așa cum vom vedea, 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.

Evaluarea algoritmului în cazul mediu

- Presupunem că $q+1-p=n$ (lungimea secvenței) și că probabilitatea ca pivotul x să fie al k-lea element este $\frac{1}{n}$ (fiecare element al tabloului poate fi pivot cu aceeași probabilitate $\frac{1}{n}$).
- Rezultă că probabilitatea obținerii subproblemelor de dimensiuni $k-p=i-1$ și $q-k=n-i$ este $\frac{1}{n}$.
- În procesul de partiționare, un element al tabloului (pivotul) este comparat cu toate celelalte, astfel că sunt necesare $n-1$ comparații.
- Acum numărul mediu de comparații se calculează prin formula:

$$T^{med}(n)=\begin{cases}(n-1)+\frac{1}{n}\sum_{i=1}^{n}(T^{med}(i-1)+T^{med}(n-i)) & ,\text{dacă } n\geq 1\\ 1 & ,\text{dacă } n=0\end{cases}$$

Evaluarea algoritmului în cazul mediu (continuare)

Rezolvăm recurența

$$T^{med}(n) = (n-1) + \frac{2}{n}(T^{med}(0) + \cdots + T^{med}(n-1))$$

- Înmulțim cu n:

$$nT^{med}(n) = n(n-1) + 2(T^{med}(0) + \cdots + T^{med}(n-1))$$

- Trecem pe n în $n-1$:

$$(n-1)T^{med}(n-1) = (n-1)(n-2) + 2(T^{med}(0) + \cdots + T^{med}(n-2))$$

- Scădem:

$$nT^{med}(n) = 2(n-1) + (n+1)T^{med}(n-1)$$

- Împărțim prin $n(n+1)$ și rezolvăm recurența obținută:

$$\begin{aligned}
\frac{T^{med}(n)}{n+1} &= \frac{T^{med}(n-1)}{n} + \frac{2}{n+1} - \frac{2}{n(n+1)} \\
&= \frac{T^{med}(n-2)}{n-1} + \frac{2}{n} + \frac{2}{n+1} - \left(\frac{2}{(n-1)n} + \frac{2}{n(n+1)}\right) \\
&= \cdots \\
&= \frac{T^{med}(0)}{1} + \frac{2}{1} + \cdots + \frac{2}{n+1} - \left(\frac{2}{1\cdot 2} + \cdots + \frac{2}{n(n+1)}\right) \\
&= 1 + 2\left(\frac{1}{1} + \frac{1}{2} + \cdots + \frac{1}{n+1}\right) - 2\left(\frac{1}{1\cdot 2} + \cdots + \frac{1}{n(n+1)}\right)
\end{aligned}$$

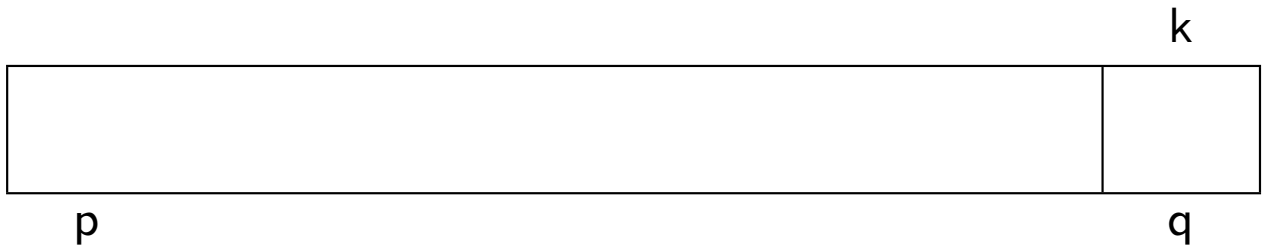
Evaluarea algoritmului în cazul mediu (continuare)

- Deoarece $1+\frac{1}{2}+\cdots+\frac{1}{n}=O(\log_2 n)$ și seria $\sum \frac{1}{k(k+1)}$ este convergentă (și deci șirul sumelor parțiale este mărginit), rezultă că $T(n)=O(n\log_2 n)$.
- *Concluzie* :

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 quickSort(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.

Transformata Fourier discretă - descriere

- Notăm cu $f(t)$ funcția care descrie semnalul în domeniul timp și cu $F(\nu)$ funcția care descrie semnalul în domeniul frecvenței.
- Se poate trece dintr-o descriere în alta prin *transformata Fourier*.
- Trecerea din domeniul timp în domeniul frecvență se face prin *transformata Fourier directă*:

$$F(\nu) = \mathscr{F}[f(t)] = \int_{-\infty}^{+\infty} f(t) \exp^{-2\pi i \nu t} \, dt, \tag{5}$$

iar trecerea de la domeniul frecvență la domeniul timp se face prin *transformata Fourier inversă*:

$$f(t) = \mathscr{F}^{-1}[F(\nu)] = \int_{-\infty}^{+\infty} F(\nu) \exp^{2\pi i \nu t} \, d\nu. \tag{6}$$

Transformata Fourier discretă - descriere (continuare)

- *Transformata Fourier discretă* se obține când se consideră $f(t)$ măsurată într-un număr finit de puncte: $t_0, \ldots, t_{n-1}$ cu $t_j = jT$, unde T este perioada de timp la care se fac măsurătorile.
- Notând $x_k = f(t_k)$, pentru $k = 0, \ldots, n-1$, *transformata Fourier discretă directă* este dată de relația:

$$\mathscr{F}[f(t)] = \sum_{k=0}^{n-1} x_k \exp^{-2\pi i \frac{j}{nT} kT} T = T \sum_{k=0}^{n-1} x_k \exp^{\frac{-2\pi ijk}{n}}. \tag{7}$$

- Notăm:

$$y_j = \sum_{k=0}^{n-1} x_k \exp^{\frac{-2\pi ijk}{n}}. \tag{8}$$

- *Transformata Fourier discretă inversă* este dată de relația:

$$x_k = \sum_{j=0}^{n-1} y_j T \exp^{\frac{2\pi ijk}{n}} \frac{1}{nT} = \frac{1}{n} \sum_{j=0}^{n-1} y_j \exp^{\frac{2\pi ijk}{n}} \tag{9}$$

Transformata Fourier discretă - descriere (continuare)

- Dacă se consideră polinomul de variabilă z:

$$X(z) = x_0 + x_1 z + \cdots x_{n-1} z^{n-1} \tag{10}$$

și rădăcinile de ordinul n ale unității:

$$\omega_j = \exp^{\frac{2\pi ij}{n}}, j = 0, 1, \ldots, n-1, \tag{11}$$

obținem

$$y_{n-j} = X(\omega_j), j = 0, 1, \ldots, n-1, \tag{12}$$

unde am considerat $y_n = y_0$.

Transformata Fourier discretă - descriere (continuare)

Rădăcinile de ordinul n ale unității satisfac următoarele proprietăți:

Lema (1)

Dacă n este par, $n = 2m$, și $j < m$, atunci $-\omega_j = \omega_{j+m}$.

Demonstrație.

Avem $\omega_{j+m}^2 = (\exp^{\frac{2\pi i(j+m)}{2m}})^2 = (\exp^{\frac{2\pi ij}{2m}})^2(\exp^{\frac{2\pi im}{2m}})^2 = (\exp^{\frac{2\pi ij}{2m}})^2 \exp^{\frac{2\pi i2m}{2m}} = \omega_j^2$.
Deoarece ω_j și ω_{j+m} sunt distincte, rezultă $\omega_j = -\omega_{j+m}$. □

Lema (1)

Dacă n este par, $n = 2m$, și $1 \leq j < m$, atunci ω_j^2 este de asemenea o rădăcină $\neq 1$.

Demonstrație.

Deoarece $\omega_j^n = 1$ rezultă $(\omega_j^2)^n = 1$. Pe de altă parte,
$\omega_j^2 = (\exp^{\frac{2\pi ij}{n}})^2 = \exp^{\frac{2\pi i(j+j)}{n}} = \omega_{j+j} \neq 1$ pentru că $2 \leq j + j \leq n - 2$. □

Algoritm pentru transformata Fourier discretă - descriere

- Reamintim că determinarea valorii unui polinom într-un punct se poate realiza prin n înmulțiri și n adunări, utilizând schema lui Horner, și numărul acestor operații este optim. Prin urmare, calculul direct al celor n componente ale transformatei Fourier necesită timpul $O(n^2)$.
- Utilizând strategia *divide-et-impera*, se poate obține un algoritm care necesită $O(n \log n)$ timp. Numim calculul dat de acest algoritm *transformata Fourier rapidă* și-l notăm FFT (Fast Fourier Transform).
- Considerăm cazul $n = 2^r$.
- Utilizăm proprietățile de mai sus ale rădăcinilor unității pentru a despărți suma din definiția lui $X(\omega_j)$ în două subsume: suma coeficienților pari și suma coeficienților impari.
- Avem:

$$\begin{aligned} X(\omega_j) &= \sum_{l=0}^{\frac{n}{2}-1} x_{2l}\omega_j^{2l} + \sum_{l=0}^{\frac{n}{2}-1} x_{2l+1}\omega_j^{2l+1} \\ &= \sum_{l=0}^{\frac{n}{2}-1} x_{2l}\omega_j^{2} + \omega_j \sum_{l=0}^{\frac{n}{2}-1} x_{2l+1}\omega_j^{2l} \end{aligned} \tag{13}$$

Algoritm pentru transformata Fourier discretă - descriere (continuare)

- Expresia anterioară corespunde scrierii polinomului $f(t)$ sub forma:

$$X(z) = (x_0 + x_2 z^2 + \cdots + x_{n-2} z^{n-2}) + z(x_1 + x_3 z^2 + \cdots + x_{n-1} z^{n-2}) = X1(z) + zX2(z), \tag{14}$$

  unde $X1(z)$ și $X2(z)$ sunt polinoame de grad $n-2$, *i.e.*, de grad mai mic cu 1 decât cel al lui $X(z)$.
- Evaluarea acestor polinoame se za face numai în punctele $\omega_j,\ j = 0, 1, \ldots, \frac{n}{2} - 1 = 2^{r-1} - 1$.
- Considerăm secvențele de numere $x_0, x_2, x_4, \ldots, x_{2^r-2}$, respectiz $x_1, x_3, x_5, \ldots, x_{2^r-1}$. Seriile Fourier ale acestor secvențe au sumele parțiale date de valorile polinoamelor

$$X'(z) = x_0 + x_2 z + x_4 z^2 + \cdots + x_{n-2} z^{\frac{n}{2}-1} \tag{15}$$

  și

$$X''(z) = x_1 + x_3 z + x_5 z^2 + \cdots + x_{n-1} z^{\frac{n}{2}-1} \tag{16}$$

  pentru rădăcinile de ordinul $\frac{n}{2}$ ale unității.
- Notăm cu $\omega'_j,\ j = 0, 1, \ldots, \frac{n}{2} - 1$ aceste rădăcini.
- Se pune problema dacă putem calcula $X(\omega_j)$, utilizând $X'(\omega'_j)$ și $X''(\omega'_j)$.

Algoritm pentru transformata Fourier discretă - descriere (continuare)

- Pentru $j < \frac{n}{2}$ avem:

$$\omega_j^{2l} = \exp^{\frac{2\pi ij2l}{2^r}} = \exp^{\frac{2\pi ijl}{2^{r-1}}} = \omega_j'^{l} \tag{17}$$

  iar pentru $j \geq \frac{n}{2}$ are loc:

$$\omega_j^{2l} = -\omega_{j-\frac{n}{2}}^{2l} = -\exp^{\frac{2\pi i(j-\frac{n}{2})2l}{2^r}} = \exp^{\frac{2\pi i(j-\frac{n}{2})l}{2^{r-1}}} = \omega'^{l}_{j-\frac{n}{2}} \tag{18}$$

- Aceste relații arată că valorile $X(\omega_j)$ pot fi calculate dacă se cunosc $X(\omega_j')$ și $X''(\omega_j')$:

$$X(\omega_j) = X'(\omega_j') + \omega_j X''(\omega_j'), \text{dacă } j < \frac{n}{2} \tag{19}$$

  și

$$X(\omega_j) = X'(\omega'_{j-\frac{n}{2}}) + \omega_j X''(\omega'_{j-\frac{n}{2}}), \text{dacă } j \geq \frac{n}{2} \tag{20}$$

Algoritm pentru transformata Fourier discretă - pseudocod

Notații:

- X este polinomul cu coeficienții $x_0, x_1, \ldots, x_{n-1}$.
- $\Omega[0..n-1] = (\omega_0, \omega_1, \ldots, \omega_{n-1})$ este un tablou de dimenisune n.
- $\Omega'[0..\lceil \frac{n}{2} \rceil - 1] = (\omega'_0, \omega'_1, \ldots, \omega'_{\lceil \frac{n}{2} \rceil - 1})$ este un tablou de dimenisune $\lceil \frac{n}{2} \rceil$.

Premise:

- Ω conține rădăcinile de ordinul n ale unității.
- Ω' conține rădăcinile de ordinul $\lceil \frac{n}{2} \rceil$ ale unității.

```
procedure FFT_Recursiv_1(X,n,Ω)
   if (n = 1)
      then X(ω_0) ← x_0
      else X' ← x_0 + x_2 z + ··· + x_{n-2} z^{n/2 - 1}
           X'' ← x_1 + x_3 z + ··· + x_{n-1} z^{n/2 - 1}
           FFT_Recursiv_1_1(X', n/2, Ω')
           FFT_Recursiv_1(X'', n/2, Ω')
           for j ← 0 to n/2 - 1 do
               X(ω_j) ← X'(ω'_j) + ω_j X''(ω'_j)
           for j ← n/2 to n - 1 do
               X(ω_j) ← X'(ω'_{j - n/2}) + ω_j X''(ω'_{j - n/2})
end
```

Algoritm îmbunătățit pentru transformata Fourier discretă - descriere

- În algoritmul de mai sus există dezavantajul că trebuie calculate atât rădăcinile unității de ordinul n, cât și cele de ordinul $\frac{n}{2}$.
- Acest dezavantaj poate fi eliminat dacă se schimbă instanța problemei.
- Plecând de la observația că dacă se cunoaște o rădăcină primitivă de ordinul n a lui 1, să zicem $\omega \neq 1$, atunci celelalte rădăcini sunt puteri ale lui ω.
- Deci putem lua $\omega_j = \omega^j$.
- Instanța problemei se modifică în felul următor: în loc să considerăm dat numai polinomul X, presupunem date polinomul X de grad $n-1$ și o rădăcină primitivă de ordinul n a unității.
- Ieșirea constă în determinarea valorilor $X(\omega^0), X(\omega^1), X(\omega^2), \ldots, X(\omega^{n-1})$.
- De asemenea, se ține cont de faptul că dacă ω este o rădăcină primitivă de ordinul n a unității, atunci ω^2 este o rădăcină primitivă de ordinul $\frac{n}{2}$ a unității.

Algoritm îmbunătățit pentru transformata Fourier discretă - pseudocod

Notații:

- X este polinomul cu coeficienții $x_0, x_1, \ldots, x_{n-1}$.
- ω este o rădăcină de ordinul n a unității.

```
procedure FFT_Recursiv_2(X, n, ω)
   if (n = 1)
      then X(ω^0) ← x_0
      else X' ← x_0 + x_2 z + ··· + x_{n-2} z^{n/2 - 1}
           X'' ← x_1 + x_3 z + ··· + x_{n-1} z^{n/2 - 1}
           FFT_Recursiv_2(X', n/2, ω^2)
           FFT_Recursiv_2(X'', n/2, ω^2)
           for j ← 0 to n - 1 do
               X(ω^j) ← X'(ω^{2(j mod n/2)}) + ω^j X''(ω^{2(j mod n/2)})
end
```

Exemplu de execuție

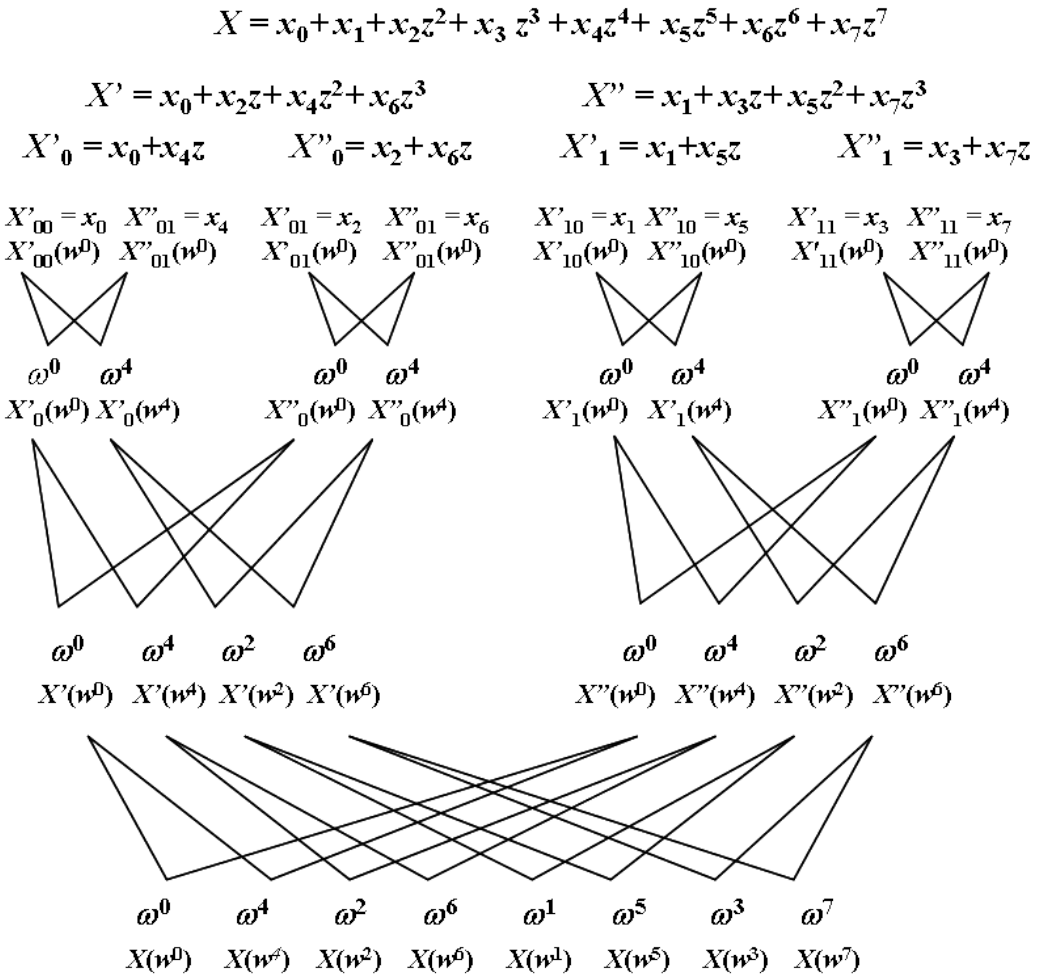


Figura 4.1 – Exemplu de execuție pentru $n = 8$

Înmulțirea numerelor întregi - algoritm *Divide_et_Impera* clasic

- Să presupunem că dorim să înmulțim două numere întregi x si y, formate din n cifre.
- Putem presupune ca x si y sunt pozitive. Algoritmul clasic necesită $O(n^2)$ operații.
- De exemplu, dacă $x = 61,438,521$ 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 $xy = x_s y_s 10^8 + (x_s y_d + x_d y_s)10^4 + x_d y_d$.
- Sunt necesare 4 înmultiri 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 $x_s y_d + x_d y_s = (x_s - x_d)(y_d - y_s) + x_s y_s + x_d y_d$
- Astfel, 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.
- Rezultă $T(n) = 3T(n/2) + O(n)$, adică $T(n) = O(n^{\log_2 3}) = O(n^{1.59})$.

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 ca 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 5 : 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 $T(n) = O(n^{\log 27}) = O(n^{2.81})$!!!

Comentarii bibliografice

Secțiunea *Transformata Fourier discretă* are la bază cărțile

1. *D. Lucanu & M. Craus, Proiectarea algoritmilor, Editura Polirom*, 2008;
2. *V. Kumar, A. Grama A. Gupta & G Karypis, Introduction to Parallel Computing: Design and Analysis of Algorithms, Addison Wesley*, 2003
   și ediția mai veche
   V. Kumar, A. Grama A. Gupta & G Karypis, Introduction to Parallel Computing: Design and Analysis of Algorithms, Benjamin-Cummings, 1994.

Secțiunile *Înmulțirea numerelor întregi* și *Înmulțirea matricelor* au la bază cartea

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

Proiectarea algoritmilor

Paradigma *greedy*

Mitică Craus

Univeristatea Tehnică "Gheorghe Asachi" din Iași

Cuprins

Paradigma *greedy* - descriere

1. Fie S o mulțime de date și $\mathscr{C}$ un tip de date cu proprietățile:
   a) obiectele din $\mathscr{C}$ reprezintă submulțimi ale lui S;
   b) operațiile includ inserararea $(X \cup \{x\})$ și eliminarea $(X \setminus \{x\})$.
2. *Clasa de probleme* la care se aplică include probleme de optim.

   Intrare: S;
   Ieșire: O submulțime maximală B din $\mathscr{C}$ care optimizează o funcție f cu valori reale.

Proprietăți ale paradigmei *greedy*

a) *Proprietatea de alegere locală*. Ș
   - Soluția problemei se obține făcând alegeri optime locale (de aici și denumirea de *greedy*=„lacom”).
   - O alegere optimă locală poate depinde de alegerile de până atunci, dar nu și de cele viitoare.
   - Alegerile optime locale nu asigură automat că soluția finală realizează optimul global, adică constituie o soluție a problemei. Trebuie demonstrat acest fapt. De regulă, aceste demonstrații nu sunt foarte simple. Acesta este un inconvenient major al metodei *greedy*.
   - Algoritmii sunt relativ simpli, dar demonstrarea faptului că aceștia rezolvă într-adevăr problema de optim asociată este deseori dificilă.

b) *Proprietatea de substructură optimă*:
   - Soluția optimă a problemi conține soluțiile optime ale subproblemelor.

Modelul matematic al paradigmei *greedy*

- Fie S o mulțime finită de intrări și $\mathscr{C}$ o colecție de submulțimi ale lui S. Spunem că $\mathscr{C}$ este *accesibilă* dacă satisface *axioma de accesibilitate*:

$$(\forall X \in \mathscr{C}) X \neq \emptyset \Rightarrow (\exists x \in X) X \setminus \{x\} \in \mathscr{C} \tag{1}$$

- Dacă $\mathscr{C}$ este accesibilă, atunci perechea $(S, \mathscr{C})$ se numește *sistem accesibil*.
- O submulțime $X \in \mathscr{C}$ se numește *bază* dacă este maximală, *i.e.*, nu există $x \in S \setminus X$ cu $X \cup \{x\} \in \mathscr{C}$.
- O submulțime $X \in \mathscr{C}$ care nu este bază se numește *extensibilă*. Cu alte cuvinte, dacă X este extensibilă, atunci există $y \in S \setminus X$ astfel încât $X \cup \{y\} \in \mathscr{C}$.
- Clasa de probleme pentru care se pot defini algoritmi *greedy* este definită de următoarea schemă:

  Se consideră date un sistem accesibil $(S, \mathscr{C})$ și o *funcție obiectiv* $f : \mathscr{C} \to \mathbb{R}$.
  Problema constă în determinarea unei baze $B \in \mathscr{C}$ care satisface:

$$f(B) = \text{optim}\{f(X) \mid X \text{ bază în } \mathscr{C}\}$$

- În general, prin optim vom înțelege minim sau maxim.
- Strategia *greedy* constă în găsirea unui criteriu de selecție a elementelor din S care candidează la formarea bazei optime (care dă optimul pentru funcția obiectiv), numit *alegere greedy* sau *alegere a optimului local*.
- Formal, optimul local are o următoarea definiție [2]:

$$f(X \cup \{x\}) = \text{optim}\{f(X \cup \{y\}) \mid y \in S \setminus X, X \cup \{y\} \in \mathscr{C}\} \tag{2}$$

Prototip algoritm *greedy* - pseudocod

```
procedure greedy(S, B)
    S1 ← S
    B ← ∅
    while (B este extensibilă) do
        alege un optim local x din S1 conform cu (2)
        S1 ← S1 \ {x}
        B ← B ∪ {x}
end
```

- Din păcate, numai condiția de accesibilitate nu asigură întotdeauna existența unui criteriu de alegere locală care să conducă la determinarea unei baze optime.
- Pentru anumite probleme, putem proiecta algoritmi *greedy* care nu furnizează soluția optimă, ci o bază pentru care funcția obiectiv poate avea valori apropiate de cea optimă.

Analiza paradigmei *greedy*

- Presupunem că pasul de alegere greedy selectează elemente x în timpul $O(k^p)$ unde $k = \#S_1$ și că testarea condiției "B este extensibilă" se face în timpul $O(\ell^q)$ cu $\ell = \#B$; $k + \ell \leq n$.
- Presupunem costul operațiilor $S_1 \setminus \{x\}$ și $B \cup \{x\}$. egal cu $O(1)$.
- Deoarece pasul de alegere este executat de n ori rezultă că metoda are complexitatea timp

$$\begin{aligned} T(n) &= O(n^p + 1^q) + \cdots + O(1^p + n^q) \\ &= O(1^p + \cdots + n^p + 1^q + \cdots + n^q) \\ &= O(n^{p+1} + n^{q+1}) = O(n^{\max(p+1, q+1)}) \end{aligned}$$

- Preprocesarea intrărilor poate conduce la o reducere considerabilă a complexității metodei.

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:

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

- 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 2).

Arbori binari ponderați pe frontieră - exemplu

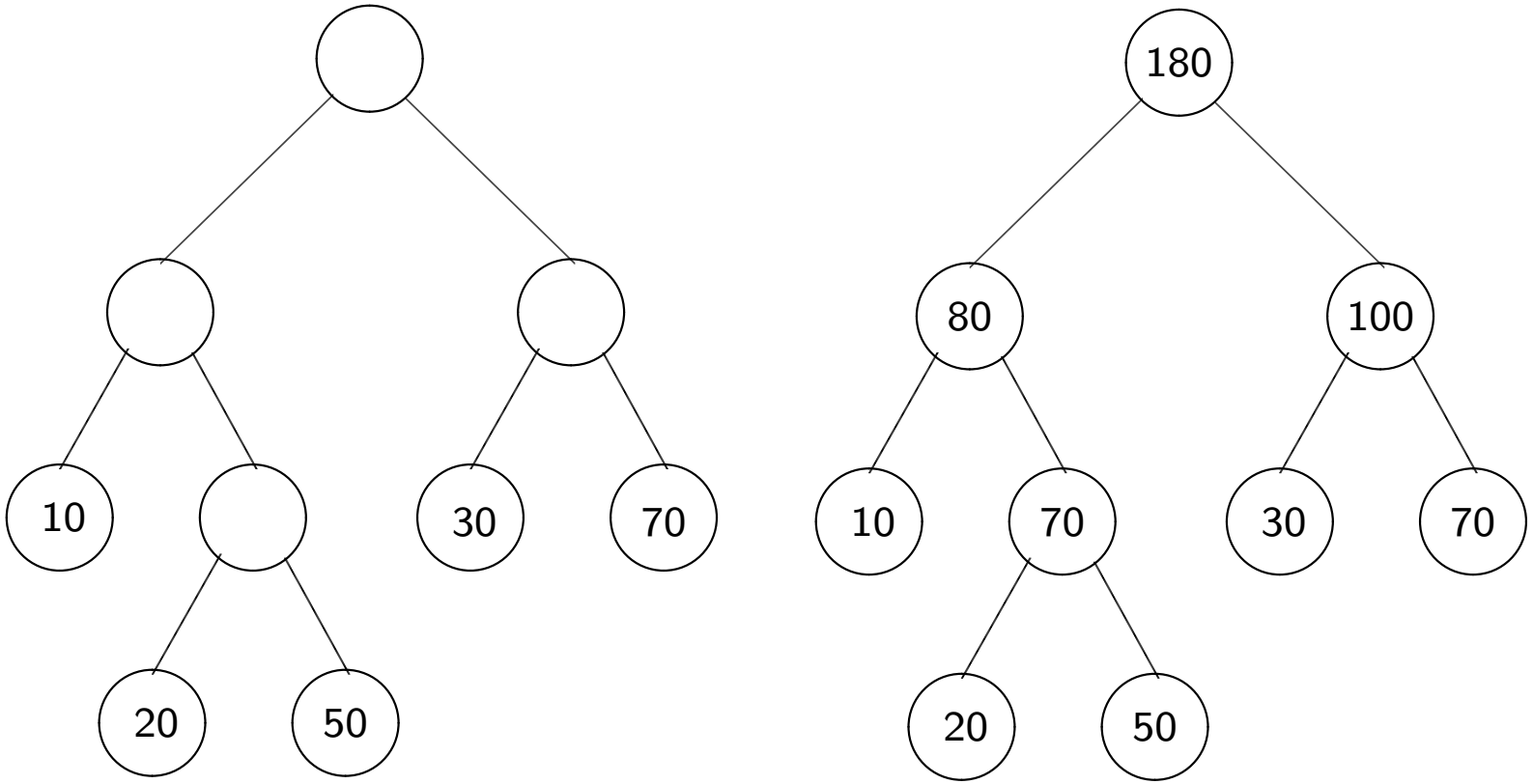


Figura 1 : Arbore ponderat pe frontieră, înainte și după modificare

Lungimea externă ponderată

Lema (1)

Fie t un arbore binar ponderat pe frontieră.
Atunci

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

Demonstrație.

Se procedează prin inducție după n, numărul de vârfuri de pe frontiera lui t.
Baza inducției. Presupunem $n = 2$. Relația este evidentă.
Pasul inductiv. Presupunem că t are $n+1$ vârfuri pe frontieră. Fie v_1 și v_2 două vârfuri de pe frontieră cu același predecesor imediat (tată) v_3. Avem $d_{v_1} = d_{v_2} = d$ și $info(v_3) = info(v_1) + info(v_2)$. Considerăm arborele t' obținut din t prin eliminarea vârfurilor v_1 și v_2. Acum vârful v_3 se află pe frontiera lui t'. Conform ipotezei inductive avem:

$$\mathrm{LEP}(t') = \sum_{v \text{ intern în } t'} info(v) \tag{3}$$

□

Lungimea externă ponderată (continuare)

Demonstrație.

Utilizăm (3) pentru a calcula lungimea externă ponderată a lui t:

$$
\begin{aligned}
\mathrm{LEP}(t) &= \sum_{v \text{ pe frontiera lui } t} d_v \cdot info(v) \\
&= \sum_{v \text{ pe frontiera lui } t, v \neq v_1, v_2} d_v \cdot info(v) + d \cdot info(v_1) + d \cdot info(v_2) \\
&= \sum_{v \text{ pe frontiera lui } t, v \neq v_1, v_2} d_v \cdot info(v) + (d-1)(info(v_1)+info(v_2)) + info(v_1) + info(v_2) \\
&= \sum_{v \text{ pe frontiera lui } t, v \neq v_1, v_2} d_v \cdot info(v) + (d-1) info(v_3) + info(v_3) \\
&= \sum_{v \text{ pe frontiera lui } t'} d_v \cdot info(v) + info(v_3) \\
&= \mathrm{LEP}(t') + info(v_3) \\
&= \sum_{v \text{ intern în } t'} info(v) + info(v_3) \\
&= \sum_{v \text{ intern în } t} info(v)
\end{aligned}
$$

S-a ținut cont de faptul că interiorul lui t este format din interiorul lui t' la care se adaugă v_3.

□

Lungimea externă ponderată - exemplu

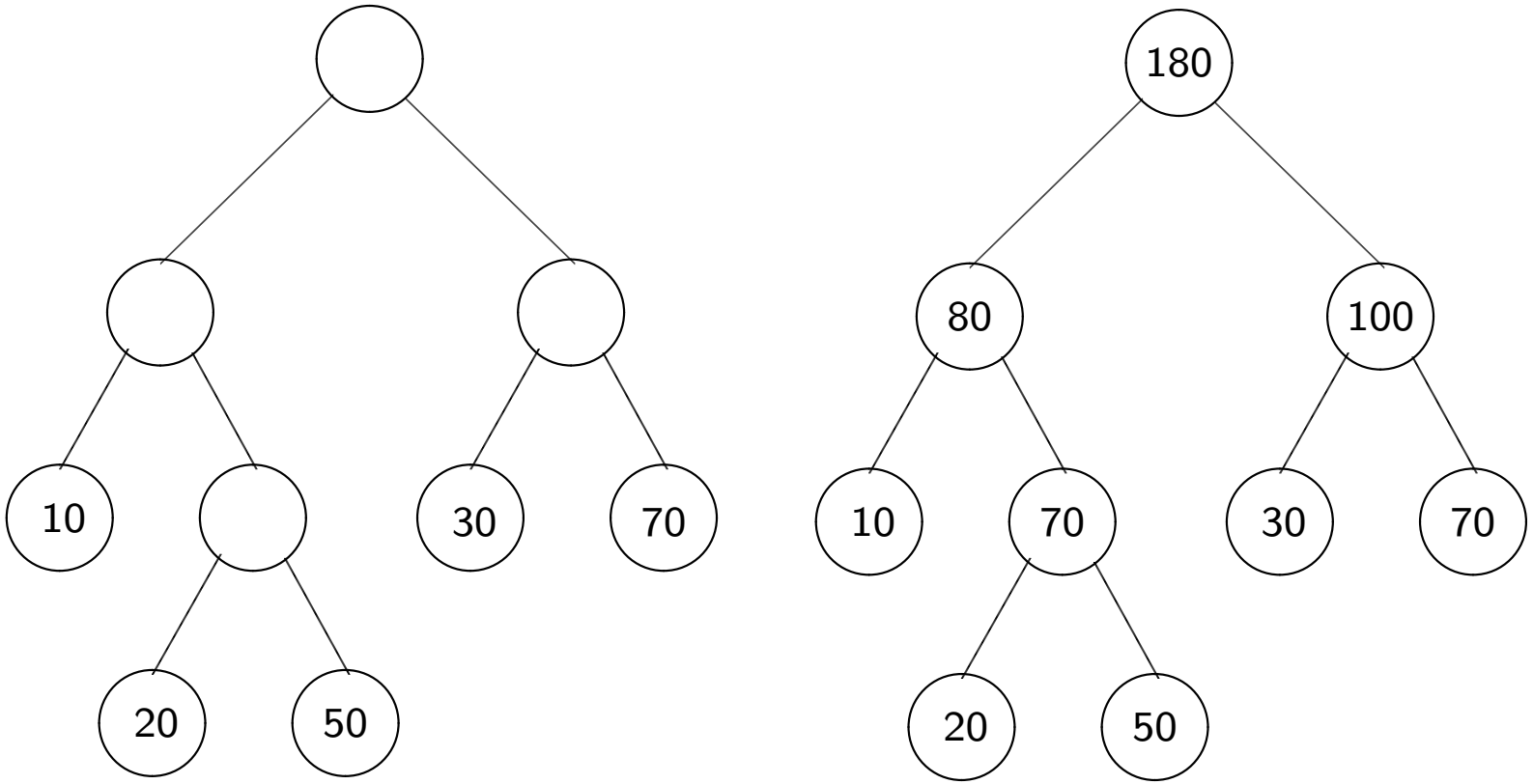


Figura 2 : Arbore ponderat pe frontieră, înainte și după modificare

- Lungimea externă ponderată a arborelui t este egală cu $80+70+180+100 = 10\cdot 2+20\cdot 3+50\cdot 3+30\cdot 2+70\cdot 2 = 430.$

Lungimea externă ponderată minimă

- Fie dată $x = (x_0, \ldots, x_{n-1})$ o secvență (listă liniară) de numere. Problema constă în determinarea unui arbore binar ponderat pe frontieră care are ca informații în cele n vârfuri de pe frontieră numerele $x_0, \ldots, x_{n-1}$ și cu lungimea externă ponderată minimă.
- O metodă total ineficientă ar putea fi generarea tuturor arborilor binari cu n vârfuri pe frontieră etichetate cu elementele secvenței x și alegerea unuia cu lungimea externă ponderată minimă.

Lungimea externă ponderată minimă (continuare)

Notăm cu $\mathscr{T}(x)$ mulțimea arborilor binari care au ca informații în vârfurile de pe frontieră numerele din secvența x.

Lema (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 $d_{v_1} \geq d_{v_2}$.

Demonstrație.

Presupunem $d_{v_1} < d_{v_2}$. Notăm $d_1 = d_{v_1}$ și $d_2 = d_{v_2}$. Fie t' arborele obținut din t prin interschimbarea vârfurilor v_1 și v_2. Avem:

$$\begin{aligned}
\mathrm{LEP}(t') &= \sum_{v \text{ pe frontiera lui } t'} d_v \cdot info(v) \\
&= \sum_{v \text{ pe frontiera lui } t', v \neq v_1, v_2} d_v \cdot info(v) + d_1 \cdot info(v_2) + d_2 \cdot info(v_1) \\
&= \sum_{v \text{ pe frontiera lui } t} d_v \cdot info(v) - d_1 \cdot info(v_1) - d_2 \cdot info(v_2) + d_1 \cdot info(v_2) + \\
&\qquad d_2 \cdot info(v_1) \\
&= \mathrm{LEP}(t) - (d_1 - d_2) \cdot (info(v_1) - info(v_2)) \\
&< \mathrm{LEP}(t)
\end{aligned}$$

Contradicție: s-a obținut un arbore cu lungime externă ponderată mai mică. Rezultă $d_1 \geq d_2$.

Lungimea externă ponderată minimă (continuare)

Lema (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.

Demonstrație.

Fie t un arbore cu LEP minimă. Fie v_i vârful etichetat cu x_i ($info(v_i) = x_i$) și d_i distanța de la rădăcină la vârful v_i, $i = 0, \ldots, n-1$.
Deoarece $x_i \leq x_{i+1}$ rezultă, conform lemei 2, $d_i \geq d_{i+1}$ (în caz de egalitate $x_i = x_{i+1}$ considerăm pe locul i vârful mai depărtat de rădăcină).
Fie v_i vârful frate al vârfului v_0. Avem $d_1 \geq d_i$ (deoarece $x_1 \leq x_i$) și $d_1 \leq d_0 = d_i$ (deoarece $x_1 \geq x_0$ și v_0 și v_i sunt vârfuri frate) care implică $d_1 = d_i$.
În arborele t interschimbăm vârfurile v_1 și v_i și obținem un arbore t' care satisface concluzia lemei. □

Algoritm pentru calcularea lungimii externă ponderate minime - 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 calcularea lungimii externă ponderate minime - 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
```

Analiza algoritmui pentru calcularea lungimii externă ponderate minime

- Pasul de alegere *greedy* constă în
  1. selectarea a doi arbori cu etichetele din rădăcină minime și
  2. construirea unui nou arbore ce va avea rădăcina etichetată cu suma etichetelor din rădăcinile celor doi arbori și pe cei doi arbori ca subarbori ai rădăcinii (figura 3).

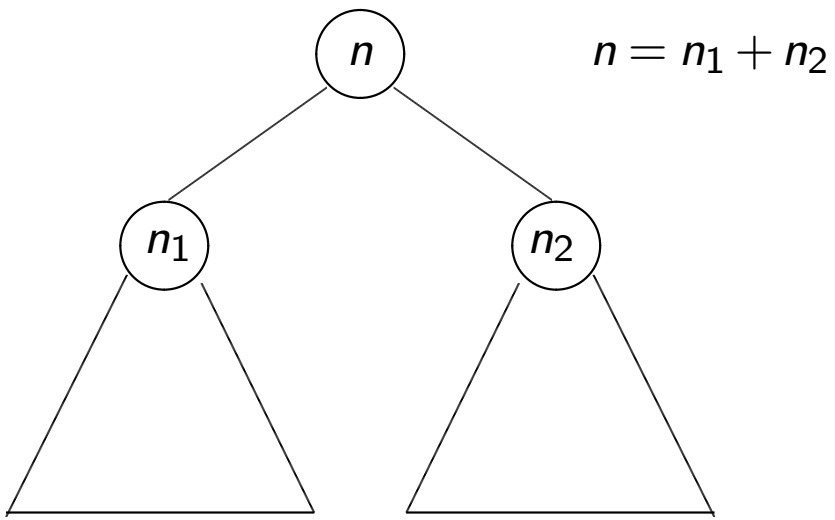


Figura 3 : Pasul de alegere *greedy*

Analiza algoritmui pentru calcularea lungimii externă ponderate minime

Teorema (1)

Fie $t^m(x)$ *unicul element din mulțimea calculată de schema procedurală* `lep`. *Arborele* $t^m(x)$ *are proprietatea:*

$$\mathrm{LEP}(t^m(x)) = \min\{\mathrm{LEP}(t) | t \in \mathscr{T}(x)\} \tag{4}$$

Demonstrație.

Consecință a Lemei 3. □

Implementarea algoritmui pentru calcularea lungimii externă ponderate minime

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.

Codificare de lungimea medie minimă.

- Fie n mesaje $M_0, \ldots, M_{n-1}$ recepționate cu frecvențele $f_0, \ldots, f_{n-1}$.
- Mesajele sunt codificate cu șiruri (cuvinte) construite peste alfabetul $\{0, 1\}$ cu proprietatea că pentru orice $i \neq j$, codul mesajului M_i nu este un prefix al codului lui M_j. O astfel de codificare se numește *independentă de prefix* („prefix-free").
- Notăm cu d_i lungimea codului mesajului M_i. *Lungimea medie* a codului este $\sum_{i=0}^{n-1} f_i \cdot d_i$.
- Problema constă în determinarea unei codificări cu lungimea medie minimă.
- Unei codificări îi putem asocia un arbore binar cu proprietățile următoare:
  - Mesajele corespund nodurilor de pe frontieră.
  - Muchiile (*tata, fiu-stânga*) sunt etichetate cu 0;
  - Muchiile (*tata, fiu-dreapta*) sunt etichetate cu 1.
  - Nodurile de pe frontiera arborelui sunt etichetate cu frecvențele mesajelor corespunzătoare.
- Drumul de la rădăcină la un nod de pe frontieră descrie codul mesajului asociat acestui nod.
- Determinarea unui cod optim coincide cu determinarea unui arbore ponderat pe frontieră optim.

Coduri Huffman - exemplu

- Codurile Huffman pot fi utilizate la scrierea comprimată a textelor.
- Considerăm textul HARABABURA.
- Mesajele sunt literele din text, iar frecvențele sunt date de numărul de apariții ale fiecărei litere în text (Figura 4a).

Literă	Frecvență
H	1
A	4
R	2
B	3
U	1

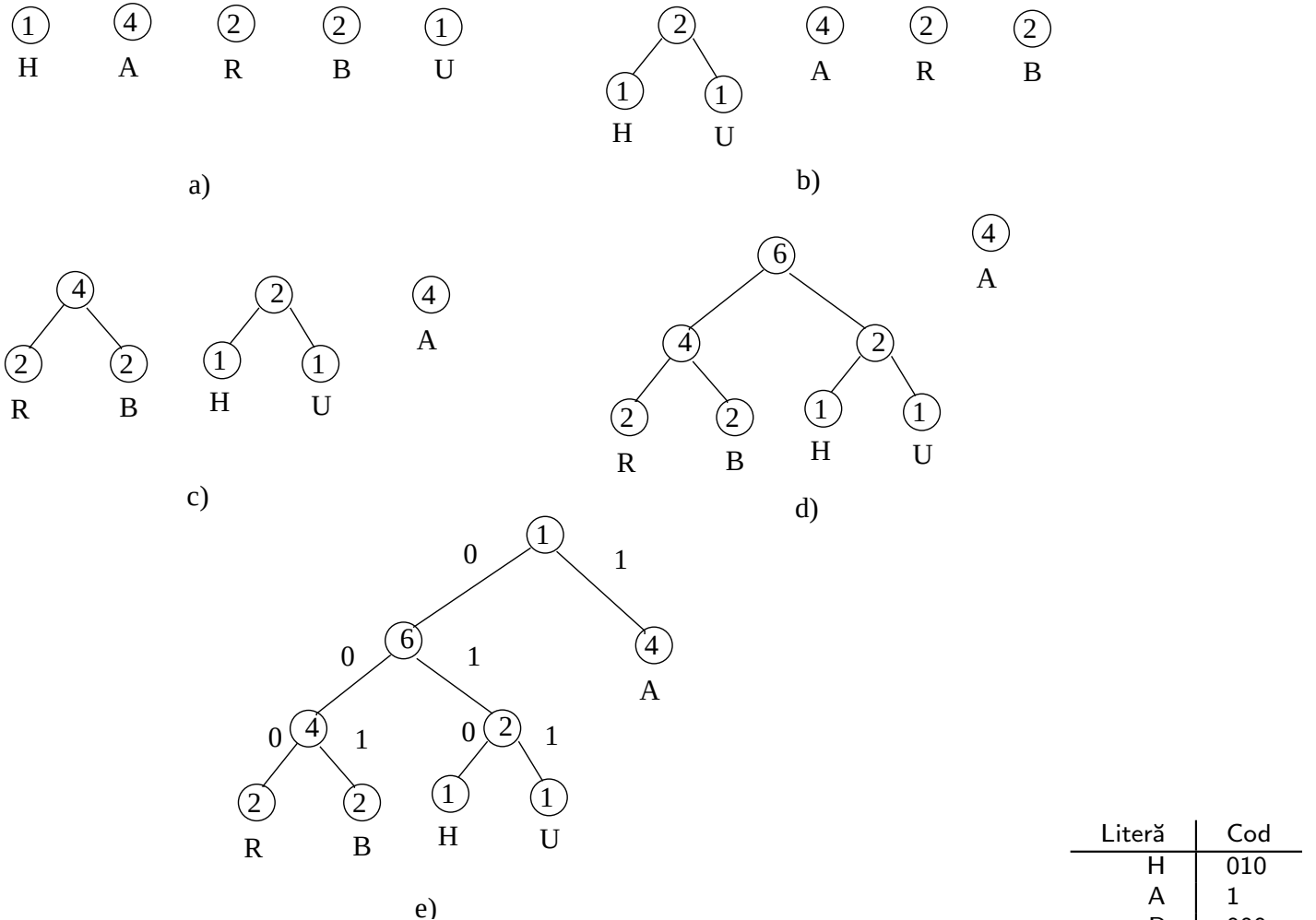
a)

Literă	Cod
H	010
A	1
R	000
B	001
U	011

b)

Figura 4 : Codificarea caracterelor din textul HARABABURA

Construcția arborelui Huffman - exemplu



Literă	Cod
H	010
A	1
R	000
B	001
U	011

Figura 5 : Construcția arborelui Huffman pentru HARABABURA

Algoritm de construcție a arborelui Huffman optim - descriere

- Presupunem că intrarea este memorată într-un tabel `T` de structuri cu două câmpuri:
  - `T[i].mes` reprezintă mesajul i;
  - `T[i].f` reprezintă frecvența mesajului i.
- Pentru implementare recomandăm reprezentarea arborilor prin tablouri.
- Notăm cu `H` tabloul ce reprezentă arborele Huffman.
- 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 mesajului corespunzător.
- Notăm cu $val(i)$ funcția care intoarce informația din nodul i, calculată ca mai sus.
- Tabloul `H`, care în final va memora arborele Huffman corespunzător codurilor optime, va memora pe parcursul construcției acestuia colecțiile intermediare de arbori.

Algoritm de construcție a arborelui Huffman optim - descriere (continuare)

- În timpul execuției algoritmului de construcție a arborelui, `H` este compus din trei părți (Figura 6):

  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.

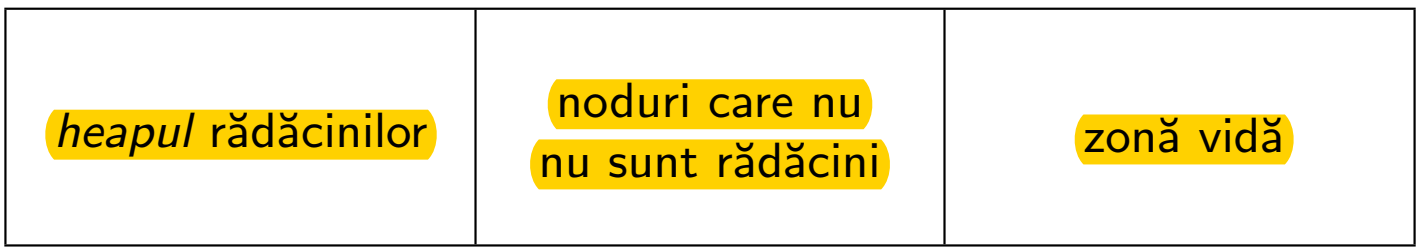


Figura 6 : Organizarea tabloului `H`

Algoritm de construcție a arborelui Huffman optim (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] ← H[1]
   k ← k + 1
   ```
   b) mutarea ultimului element din heap pe prima poziție:
   ```
   H[1] ← H[m]
   m ← m - 1
   ```
   c) refacerea min-*heapului*.
2. Mutarea rădăcinii cu informația cea mai mică pe prima poziție liberă din zona a treia, fără a o elimina din min-*heap*:
   ```
   H[k] ← H[1]
   k ← k + 1
   ```
3. Construirea noii rădăcini și memorarea acesteia pe prima poziție în min-*heap* (în locul celei mutate mai sus).
4. Refacerea min-*heapului*.

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

Interclasarea optimală a unei mulțimi de secvențe sortate - descriere

- 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 optimală a unei mulțimi de secvențe sortate - descriere (continuare)

- Considerăm problema interclasării a două secvențe sortate:

  Fie date două secvențe sortate $x = (x_0, \ldots, x_{m-1})$ și $y = (y_0, \ldots, y_{n-1})$ ce conțin elemente dintr-o mulțime total ordonată. Să se construiască o secvență sortată $z = (z_0, \ldots, z_{n+m-1})$ care să conțină cele $m+n$ 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 algoritm este cel mult $m+n-1$, iar numărul de elemente transferate este $m+n$.
- 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:
    $b_0 = \text{merge}(a_0, a_1)$
    $b_1 = \text{merge}(b_0, a_2)$
    $b_2 = \text{merge}(a_3, a_4)$
    $b = \text{merge}(b_1, b_2)$
  - 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 LEP a arborelui asociat.
- Așadar, alegerea optimă corespunde arborelui cu LEP minimă.

```
procedure interclOpt(x, n)
begin
    B ← {a_0, ..., a_{n-1}}
    while (#B > 1) do
        alege x1, x2 din B cu lungimi minime
        intercl2(x1, x2, y)
        B ← (B \ {x1,x2}) ∪ {y}
end
```

Interclasarea optimală a unei mulțimi de secvențe sortate - exemplu

- 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:
  $b_0 = merge(a_0, a_4)$
  $b_1 = merge(a_3, b_0)$
  $b_2 = merge(a_1, a_2)$
  $b = merge(b_1, b_2)$
- Numărul de comparații este $50 + 90 + 130 + 220 = 490$.

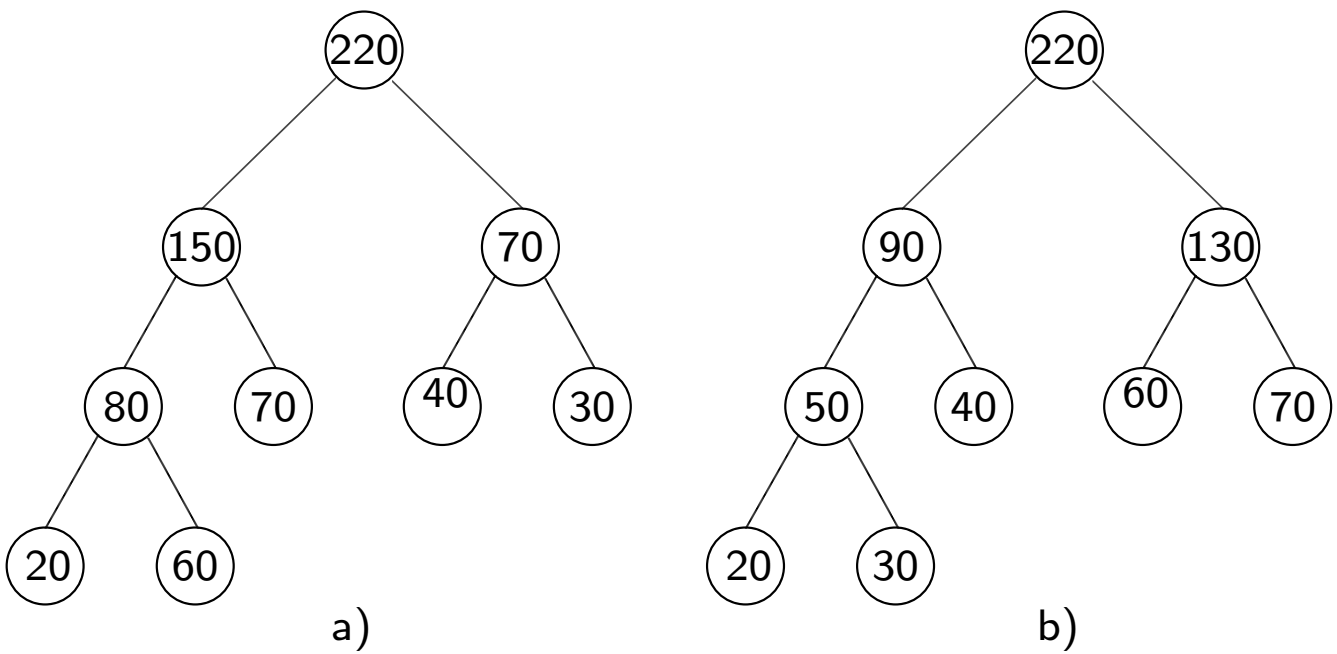


Figura 7 : Arbori asociați algoritmilor de interclasare

Problema rucsacului

- Se consideră un rucsac de capacitate M și n obiecte notate cu $0, 1, \ldots, n-1$ de dimensiuni (greutăți) $w_0, w_1, \ldots, w_{n-1}$.
- Dacă în rucsac se pune o parte fracționară x_i din obiectul i, $0 \leq x_i \leq 1$, atunci se obține un profit $p_i \cdot x_i$ ($p_i > 0$).
- Umplerea rucsacului cu fracțiunile (cantitățile) $x_0, \ldots, x_{n-1}$ aduce profitul total $\sum_{i=0}^{n-1} p_i x_i$.
- Problema constă în a determina părțile fracționare $x_0, \ldots x_{n-1}$ care aduc un profit total maxim.
- Problema poate fi formulată ca o problemă de optim, în modul următor:
  - Funcția obiectiv:

$$\max \sum_{i=0}^{n-1} p_i x_i$$

  - Restricții:

$$\sum_{i=0}^{n-1} w_i x_i \leq M, 0 \leq x_i \leq 1, i = 0, \ldots, n-1$$

- Dacă $\sum_{i=0}^{n-1} w_i \leq M$, atunci profitul maxim se obține când $x_i = 1, 0 \leq i \leq n-1$.
- Vom presupune că $\sum_{i=0}^{n-1} w_i > M$. Fracțiunile x_i nu pot fi toate egale cu 1.
- Rucsacul poate fi umplut exact, *i.e.*, putem alege x_i astfel încât $\sum_{i=0}^{n-1} w_i x_i = M$.

Problema rucsacului - soluția I

- În fiecare pas se introduce în rucsac obiectul care aduce profit maxim.
- În ultimul pas, dacă obiectul nu încape în totalitate, se introduce numai acea parte fracționară a sa, care umple exact rucsacul.

```
procedure rucsac_1(w, p, x, n)
    S ← {0,...,n-1}
    for i ← 0 to n-1 do
        x[i] ← 0
        C ← 0
    while ((C < M) and (S ≠ ∅)) do
*:      alege i ∈ S care maximizează profitul peste S
        S ← S \ {i}
        if (C + w[i] ≤ M)
           then C ← C + w[i]
                x[i] ← 1
           else C ← M
                x[i] ← (M - C) / w[i]
end
```

Problema rucsacului - soluția I (continuare)

- Procedura `rucsac_1` are dezavantajul că nu determină întodeauna optimul.
- Presupunem $n = 3, M = 10$, iar dimensiunile și profiturile obiectelor date de următorul tabel:

	0	1	2
w_i	6	4	8
p_i	3	4	6

- Algoritmul `rucsac_1` va determina soluția $x = (0, \frac{1}{2}, 1)$ care produce profitul $\sum p_i x_i = \frac{1}{2} \cdot 4 + 1 \cdot 6 = 8$.
- Se observă că vectorul $x' = (0, 1, \frac{3}{4})$ produce un profit mai bun: $\sum p_i x'_i = 1 \cdot 4 + \frac{3}{4} \cdot 6 = \frac{17}{2} > 8$.

Problema rucsacului - soluția a II-a

- La fiecare pas va fi introdus în rucsac obiectul care aduce profit maxim pe unitatea de capacitate (greutate) utilizată, adică obiecul care maximizează fracția $\frac{p_i}{w_i}$ peste mulțimea obiectelor neintroduse încă.
- Algoritmul corespunzător acestei strategii se obține din `rucsac_1` prin înlocuirea liniei $*$: cu alege `i` $\in$ `S` care maximizează profitul pe unitatea de greutate peste S
- Startegia a II-a determină soluția optimă (cu profit maxim).

Problema rucsacului - corectitudinea soluției II

Teorema (2)

Procedura `rucsac_2` *determină soluția optimă (cu profit maxim).*

Demonstrație.

Presupunem $\frac{p_0}{w_0} \geq \cdots \geq \frac{p_{n-1}}{w_{n-1}}$. Fie $x = (x_0, \cdots, x_{n-1})$ soluția generată de procedura `Rucsac_2`.
Dacă $x_i = 1, 0 \leq i < n$, atunci este evident că această soluție este optimă.
Altfel, fie j primul indice pentru care $x_j \neq 1$.
Din algoritm, se observă că $x_i = 1$ pentru orice $0 \leq i < j$ și $x_i = 0$ pentru $i > j$.
Fie $y = (y_0 \cdots y_{n-1})$ o soluție optimă (care maximizează profitul).
Avem $\sum_{i=0}^{n-1} y_i w_i = M$.
Fie k cel mai mic indice pentru care $x_k \neq y_k$.
Există următoarele posibilități:

i) $k < j$. Rezultă $x_k = 1$, iar $y_k \neq x_k$ implică $y_k < x_k$.

ii) $k = j$. Deoarece $\sum x_i \cdot w_i = M$ și $x_i = y_i, 1 \leq i < j$, rezultă că $y_k < x_k$ (altfel $\sum y_i \cdot w_i > M$. Contradicție).

iii) $k > j$. Rezultă $\sum_{i=0}^{n-1} y_i \cdot w_i > \sum_{i=0}^{j} x_i \cdot w_i = M$. Contradicție.

Problema rucsacului - corectitudinea soluției II (continuare)

Demonstrație.

Toate situațiile conduc la concluzia $y_k < x_k$ și $k \leq j$.
Mărim y_k cu diferența până la x_k și scoatem această diferență din secvența $(y_{k+1}, \ldots, y_{n-1})$, astfel încât capacitatea utilizată să rămînă tot M.
Rezultă o nouă soluție $z = (z_0, \ldots, z_{n-1})$ care satisface:

$$z_i = x_i,\ 0 \leq i \leq k$$

$$\sum_{k<i\leq n-1} (y_i - z_i) \cdot w_i = (x_k - y_k) \cdot w_k$$

Avem:

$$\begin{aligned}
\sum_{i=0}^{n-1} z_i \cdot p_i &= \sum_{i=0}^{n-1} y_i \cdot p_i + \sum_{0\leq i<k} z_i \cdot p_i + z_k \cdot p_k + \sum_{k<i<n} z_i p_i - \sum_{0\leq i<k} y_i p_i - y_k \cdot p_k - \\
&\quad - \sum_{k<i<n} y_i p_i \\
&= \sum_{i=0}^{n-1} y_i \cdot p_i + (z_k - y_k) \cdot p_k \cdot \frac{w_k}{w_k} - \sum_{k<i<n} (y_i - z_i) \cdot p_i \cdot \frac{w_i}{w_i} \\
&\geq \sum_{i=0}^{n-1} y_i \cdot p_i + (z_k - y_k) \cdot w_k \frac{p_k}{w_k} - \sum_{k<i<n} (y_i - z_i) \cdot w_i \cdot \frac{p_k}{w_k} \\
&= \sum_{i=0}^{n-1} y_i \cdot p_i
\end{aligned}$$

Problema rucsacului - corectitudinea soluției II (continuare)

Demonstrație.

Deoarece y este soluție optimă, rezultă $\sum_{i=0}^{n-1} z_i p_i = \sum_{i=0}^{n-1} y_i p_i$.
Soluția z are următoarele două proprietăți:

- este optimă, și
- coincide cu x pe primele k poziții (y coincidea cu x numai pe primele $k-1$ poziții).

Procedeul de mai sus este repetat (considerând z în loc de y) până când se obține o soluție optimă care coincide cu x. □

Problema rucsacului - complexitatea

- Timpul de execuție al algoritmului `rucsac_2` este $O(n \log n)$.
  - Dacă intrările satisfac $\frac{p_0}{w_0} \geq \cdots \geq \frac{p_{n-1}}{w_{n-1}}$, atunci algoritmul `rucsac_2` necesită timpul $O(n)$.
  - Timpul de preprocesare (ordonare) este $O(n \log n)$.

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 *programare dinamică*

Mitică Craus

Univeristatea Tehnică "Gheorghe Asachi" din Iași

Cuprins

Paradigma *programare dinamică* - descriere

- *Clasa de probleme* la care se aplică este formată din probleme de optim.
- Etape:
  1. Definirea noțiunii de *stare* (care este de fapt o subproblemă) și asocierea funcției obiectiv pentru stare (subproblemă).
  2. Definirea unei relații de tranziție între stări.
     - O relație $s \to s'$, unde s și s' sunt stări, va fi numită *decizie*.
     - O *politică* este o secvență de decizii consecutive, adică o secvență de forma $s_0 \to s_1 \to \cdots s_n$.
  3. Aplicarea *Principiului de optim* (*PO*) pentru obține o relație de recurență.
     - *O subpolitică a unei politici optimale este la rândul ei optimală*.
     - Deoarece este posibil ca *PO* să nu aibă loc, rezultă că trebuie verificată validitatea relației de recurență.
  4. *Calculul recurenței* rezolvând subproblemele de la mic la mare și memorând valorile obținute într-un tablou.
     - Nu se recomandă scrierea unui program recursiv care să calculeze valorile optime.
     - Dacă în procesul de descompunere problemă $\mapsto$ subproblemă, o anumită subproblemă apare de mai multe ori, ea va fi calculată de algoritmul recursiv de câte ori apare.
  5. Extragerea soluției optime din tablou utilizând proprietatea de *substructură optimă a soluției*, care afirmă că *soluția optimă a problemei include soluțiile optime ale subproblemelor sale*.
     - Proprietatea de substructură optimă este echivalentă cu principiul de optim.

Aplicarea paradigmei *programare dinamică*

- *Direct*:
  1. Se definește noțiunea de *stare* (de cele mai multe ori ca fiind o subproblemă);
  2. Se aplică principiul de optim pentru a deduce relațiile de recurență de tip 3
  3. Se stabilesc strategiile de calcul al valorilor și soluțiilor optime.
- *Prin comparare*:
  1. Se observă că problema este asemănătoare cu una dintre problemele cunoscute;
  2. Se încearcă aplicarea strategiei în aceeași manieră ca în cazul problemei corespunzătoare.

Modelul matematic al paradigmei *programare dinamică*

- Problemele ale căror soluții se pot obține prin programarea dinamică sunt probleme de optim.
- Prototipul clasic unei probleme de optim:
  - Să se determine:

$$\text{optim } R(x_0, \ldots, x_{m-1}) \tag{1}$$

    după $x_0, \ldots, x_{m-1}$, în condițiile în care acestea satisfac restricții de forma:

$$g(x_0, \ldots, x_{m-1}) \ ? \ 0 \tag{2}$$

    unde $? \in \{<, \leq, =, \geq, >\}$.
  - Prin *optim* înțelegem *min* sau *max*, iar ecuația 1 se mai numește și *funcție obiectiv*.
- Prototipul furnizat de digrafurile ponderate:
  - $R(x_0, \ldots, x_{m-1})$ exprimă suma ponderilor arcelor $x_0, \ldots, x_{m-1}$;
  - Restricțiile impun ca $x_0, \ldots, x_{m-1}$ să fie drum sau circuit cu anumite proprietăți.

Modelul matematic al paradigmei *programare dinamică* (continuare)

- Paradigma programării dinamice propune găsirea valorii optime prin luarea unui șir de decizii $(d_1, \ldots, d_n)$, numit și *politică*, unde decizia d_i transformă starea (problemei) s_{i-1} în starea s_i, aplicând *principiul de optim*:
  - Secvența de decizii optime (politica optimă) care corespunde stării s_0 are proprietatea că după luarea primei decizii, care transformă starea s_0 în starea s_1, secvența de decizii (politica) rămasă este optimă pentru starea s_1.
- Prin stare a problemei înțelegem o subproblemă.
- Unei stări s îi asociem o valoare z și definim $f(z)$, astfel încât, dacă starea s corespunde problemei inițiale, atunci:

$$f(z) = \text{optim } R(x_0, \ldots, x_{m-1})$$

Modelul matematic al paradigmei *programare dinamică* (continuare)

- Principiul de optim conduce la obținerea unei ecuații funcționale de forma:

$$f(z) = \underset{y}{\text{optim}} \left[H(z, y, f(T(z, y))) \right] \tag{3}$$

unde:

  - s și s' sunt două stări cu proprietatea că una se obține din cealaltă aplicând decizia d;
  - z este valoarea asociată stării s;
  - $T(z, y)$ calculează valoarea stării s', iar
  - H exprimă algoritmul de calcul al valorii $f(z)$ dat de decizia d.

- Relația 3 poate fi interpretată astfel:
  - Dintre toate deciziile care se pot lua în starea s (sau care conduc la starea s'), se alege una care dă valoarea optimă în condițiile în care politica aplicată în continuare (sau până atunci) este și ea optimă.
- Relația 3 poate fi dovedită utilizând inducția și reducerea la absurd.
- Deoarece este posibil ca principiul de optim să nu aibă loc pentru anumite formulări, este necesară verificarea sa pentru problema supusă rezolvării.
- Rezolvarea ecuațiilor recurente 3 conduce la determinarea unui șir de decizii ce în final constituie politica optimă prin care se determină valoarea funcției obiectiv.

Implementarea paradigmei *programare dinamică*

- Principul de optim implică proprietatea de substructură optimă a soluției:
  - *Soluția optimă a problemei include soluțiile optime ale subproblemelor sale*.
- Proprietatea de substructură optimă a soluției este folosită pentru determinarea soluțiilor corespunzătoare stărilor optime.
- În general, programele care implementează modelul dat de programarea dinamică au două părți:
  1. În prima parte, se determină valorile optime date de șirul de decizii optime, prin rezolvarea ecuațiilor 3.
  2. În partea a doua se construiesc soluțiile (valorile x_i care dau optimul) corespunzătoare stărilor optime, pe baza valorilor calculate în prima parte, utilizând proprietatea de substructură optimă.

Implementarea paradigmei *programare dinamică* (continuare)

- Nu se recomandă scrierea unui program recursiv care să calculeze valorile optime.
- Dacă în procesul de descompunere problemă $\mapsto$ subproblemă, o anumită subproblemă apare de mai multe ori, ea va fi calculată ori de de câte ori apare.
- Este mult mai convenabil ca valorile optime corespunzătoare subproblemelor să fie memorate într-un tablou și apoi combinate pe baza ecuațiilor 3 pentru a obține valoarea optimă a unei supraprobleme.
- În acest fel, orice subproblemă este rezolvată o singură dată (aici este una dintre diferențele dintre programarea dinamică și strategia *divide-et-impera* unde o aceeași subproblemă poate fi calculată de mai multe ori), iar determinarea valorilor optime se face de la subproblemele mai mici la cele mai mari (*bottom-up*).

Analiza paradigmei *programare dinamică*

- Complexitatea algoritmului care calculează valorile optime depinde direct de tipul de recursivitate implicat de recurențele rezultate prin aplicarea principiului de optim.
- Dacă rezultă o recursivitate liniară atunci valorile optime ale subproblemelor pot fi calculate în timp liniar și memorate într-un tablou unidimensional.
- În cazul recursiei în cascadă lucrurile sunt mai complicate.
  - Dacă adâncimea arborelui corespunzător apelurilor recursive este n atunci rezultă un număr de 2^n de subprobleme de rezolvat.
  - În unele cazuri, o redefinire a noțiunii de stare poate conduce la obținerea unei recursii liniare obținându-se astfel o reducere drastică a complexității – de la exponențial la polinomial.
  - O reducere la complexitatea polinomială (de cele mai multe ori pătratică) se obține atunci când se poate aplica metoda de derecursivare cu memorarea rezultatelor pentru subprobleme în tabele,
- Pentru problemele *NP*-dificile nu se poate obține o reducere la complexitatea polinomială
  - Pentru aceste cazuri se caută un mod de memorare cât mai compactă a valorilor optime pentru subprobleme și metode cât mai eficiente pentru calculul acestora astfel încât, pentru instanțele de dimensiuni rezonabile, determinarea soluției să se facă într-un timp acceptabil.

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 ℓ 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 ℓ_{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, ℓ) *este un digraf ponderat fără circuite negative*.
- Fie ρ 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 ρ.
- Dacă vârful k nu aparține lui ρ, atunci politica obținerii lui ρ 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 ρ, notăm cu ρ_1 subdrumul lui ρ de la i la k și cu ρ_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 ρ_1 și ρ_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 ℓ_{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 (1)

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 ℓ_{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
            ℓ^init_ij = { 0    , i = j
                        { ℓ_ij  , ⟨i,j⟩ ∈ A
                        { ∞    , altfel
            P^init_ij = { i    , i ≠ j, ⟨i,j⟩ ∈ A
                        { -1   , altfel
    for i ← 0 to n-1 do
        for j ← 0 to n-1 do
            ℓ^0_ij = min{ℓ^init_ij, ℓ^init_i0 + ℓ^init_0j}
            P^0_ij = { P^init_ij  , ℓ^0_ij = ℓ^init_ij
                     { P^init_0j  , ℓ^0_ij = ℓ^init_i0 + ℓ^init_0j
    for k ← 1 to n-1 do
        for i ← 0 to n-1 do
            for j ← 0 to n-1 do
                ℓ^k_ij = min{ℓ^(k-1)_ij, ℓ^(k-1)_ik + ℓ^(k-1)_kj}
                P^k_ij = { P^(k-1)_ij  , ℓ^k_ij = ℓ^(k-1)_ij
                         { P^(k-1)_kj  , ℓ^k_ij = ℓ^(k-1)_ik + ℓ^(k-1)_kj
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 ℓ^k și ℓ^{k-1} pot fi memorate de același tablou bidimensional `G.L`.
- Simbolul ∞ 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.

Arbori binari de căutare optimali - descriere

- Fie $A = (a_1, a_2, \ldots, a_n)$ o secvență de chei sortată crescător din care se construiește un arbore binar de căutare.
- Se consideră că operația de căutare se execută cu anumite frecvențe:
  - Notăm cu p_i probabilitatea de a fi căutat elementul $a_i, i = 1, \ldots, n$ și cu P secvența $(p_1, p_2, \ldots, p_n)$.
  - Notăm cu q_i probabilitatea de a fi căutat un element x cu proprietatea $a_i < x < a_{i+1}, i = 0, \ldots, n$, unde am presupus $a_0 = -\infty, a_{n+1} = +\infty$.
  - Secvența $(q_0, q_1, \ldots, q_n)$ va fi notată cu Q.
- În aceste condiții avem:

  $\sum_{i=1}^{n} p_i$ – probabilitatea căutărilor terminate cu succes;

  $\sum_{i=0}^{n} q_i$ – probabilitatea căutărilor terminate fără succes și

  $\sum_{i=1}^{n} p_i + \sum_{i=0}^{n} q_i = 1$.
- Pentru P și Q date, *arborele binar de căutare optimal* este cel pentru care operația de căutare se efectuează în timp mediu minim.

Arbori binari de căutare optimali - modelul matematic

- Pentru a pune în evidență timpul mediu se consideră arborele binar de căutare completat cu o serie de noduri pendante corespunzătoare intervalelor de insucces.
- În figura 1, se prezintă un arbore binar de căutare și arborele completat cu nodurile pendante.
- Nodul E_i corespunde tuturor căutărilor fără succes pentru valori cuprinse în intervalul (a_i, a_{i+1}).

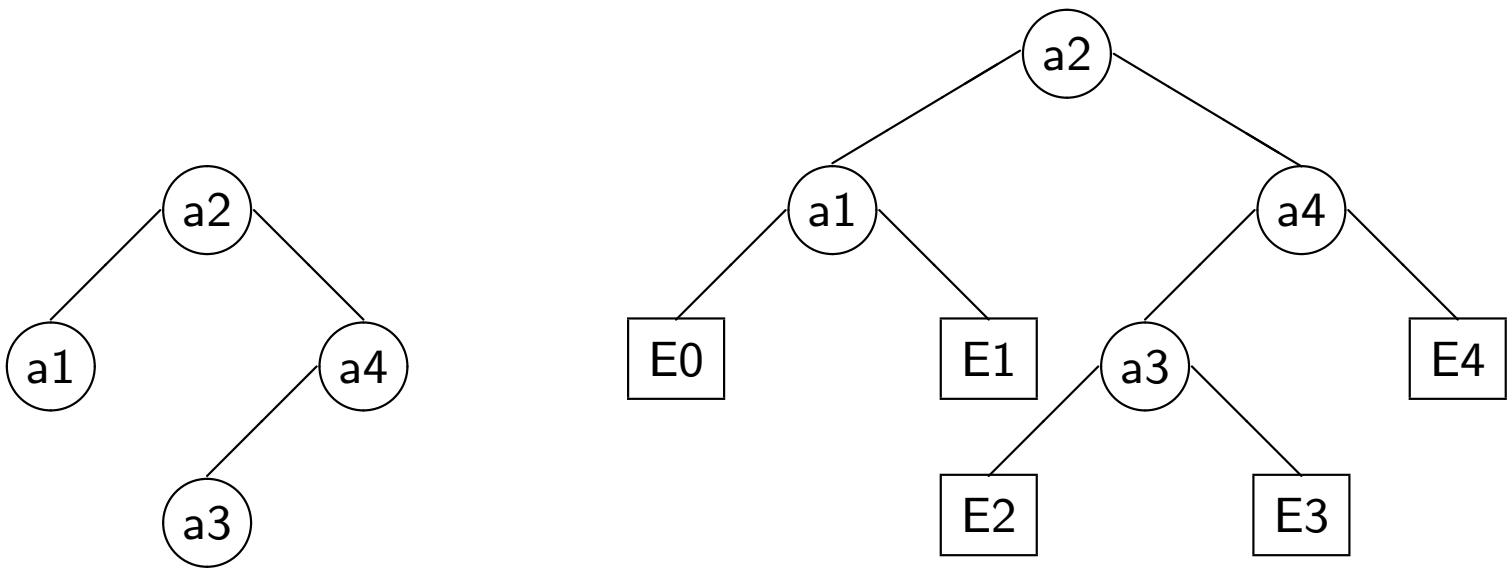


Figura 1: Arbore binar de căutare completat cu noduri pendante

Arbori binari de căutare optimali - modelul matematic (continuare)

- Timpul unei operații de căutare pentru un element x este dat de adâncimea nodului de valoare x sau de adâncimea pseudonodului corespunzător intervalului care-l conține pe x.
- Costul unui arbore binar de căutare se definește ca timpul mediu al unei căutări pentru o valoare x, adică:

$$\text{cost}(T) = \sum_{i=1}^{n} p_i * \text{nivel}(a_i) + \sum_{i=0}^{n} q_i * (\text{nivel}(E_i) - 1)$$

- Prin `nivel`(a) se notează nivelul nodului a și reprezintă numărul de comparații efectuate de funcția de căutare pe drumul de la rădăcină până la nodul a.
- Ponderând `nivel`(a) cu probabilitatea de a fi căutat nodul a și efectuând suma pentru toate nodurile se obține timpul mediu de căutare cu succes a unui nod.
- În mod similar, se calculează timpul mediu de căutare pentru insucces, cu observația că din `nivel`(E_i) se scade valoarea 1, deoarece decizia de apartenență la un interval nu se face la nivelul pseudonodului, ci la nivelul părintelui său.

Arbori binari de căutare optimali - modelul matematic (continuare)

- Dată fiind secvența de valori A, se pot construi o mulțime de arbori binari de căutare cu cheile nodurilor din A.
- Un arbore binar de căutare optimal este arborele de cost minim.
- Construcția unui astfel de arbore are la bază modelul programării dinamice.
- Programarea dinamică reduce numărul de încercări eliminând o serie de secvențe care nu pot fi optimale și aceasta apelând la principiul optimalității.
- Din punctul de vedere al programării dinamice, construirea unui arbore constă într-un șir de decizii privind nodul care se alege ca rădăcină la fiecare pas.
- Fie $c_{i,j}$ costul unui arbore binar de căutare optimal care are vârfurile din secvența $(a_{i+1}, a_{i+2}, \ldots, a_j)$ și pseudovârfurile din secvența $(e_i, e_{i+1}, \ldots, e_j)$.

Arbori binari de căutare optimali - modelul matematic (continuare)

Lema

Matricea $C = (c_{i,j})_{i,j=0,\ldots,n}$ *poate fi calculată cu ajutorul recurenței*

$$c_{i,j} = \begin{cases} \min_{i+1 \le k \le j}\{c_{i,k-1} + c_{k,j} + w_{i,j}\}, & i \neq j \\ 0, & i = j. \end{cases} \quad \text{unde } w_{i,j} = q_i + \sum_{l=i+1}^{j} (p_l + q_l)\} \quad (4)$$

Arbori binari de căutare optimali - modelul matematic (continuare)

Demonstrație.

Fie $(a_1, a_2, \ldots, a_r, \ldots, a_n)$ secvența de noduri sortată crescător. Presupunem că în primul pas se alege ca rădăcină nodul a_r.
Aplicând principiul optimalității, subarborele stâng L este optimal și construit cu secvențele $(a_1, a_2, \ldots, a_{r-1})$ și $(e_0, e_1, \ldots, e_{r-1})$:

$$\text{cost}(L) = \sum_{i=1}^{r-1} p_i * \text{nivel}(a_i) + \sum_{i=0}^{r-1} q_i * (\text{nivel}(e_i) - 1)$$

Analog, subarborele drept R este optimal și construit cu secvențele $(a_{r+1}, a_{r+2}, \ldots, a_n)$ și $(e_r, e_{r+1}, \ldots, e_n)$:

$$\text{cost}(R) = \sum_{i=r+1}^{n} p_i * \text{nivel}(_i) + \sum_{i=r}^{n} q_i * (\text{nivel}(e_i) - 1)$$

Valorile `nivel()` sunt considerate în subarborii L, R. □

Arbori binari de căutare optimali - modelul matematic (continuare)

Demonstrație.

Costul asociat arborelui T devine:

$$\text{cost}(T) = p_r + \text{cost}(L) + \text{cost}(R) + \sum_{i=1}^{r-1} p_i + \sum_{i=0}^{r-1} q_i + \sum_{i=r+1}^{n} p_i + \sum_{i=r}^{n} q_i.$$

Sumele $\sum_{i=1}^{r-1} p_i + \sum_{i=0}^{r-1} q_i + \sum_{i=r+1}^{n} p_i + \sum_{i=r}^{n} q_i$ reprezintă valorile suplimentare care apar datorită faptului că T introduce un nivel nou.

$$\text{cost}(T) = p_r + \text{cost}(L) + \text{cost}(R) + q_0 + \sum_{i=1}^{r-1} (p_i + q_i) + q_r + \sum_{i=r+1}^{n} (p_i + q_i)$$

Dacă se utilizează notația $w_{i,j} = q_i + \sum_{l=i+1}^{j} (p_l + q_l)$, rezultă

$$\text{cost}(T) = p_r + \text{cost}(L) + \text{cost}(R) + w_{0,r-1} + w_{r,n} = \text{cost}(L) + \text{cost}(R) + w_{0,n}$$

În urma aplicării definiției lui $c_{i,j}$ se obține:

$$\text{cost}(T) = c_{0,n},\ \text{cost}(L) = c_{0,r-1} \text{ and } \text{cost}(R) = c_{r,n}$$

$$c_{0,n} = c_{0,r-1} + c_{r,n} + w_{0,n} = \min_{1 \le k \le n} \{c_{0,k-1} + c_{k,n} + w_{0,n}\}$$

Prin generalizare rezultă relatia (4).

Arbori binari de căutare optimali - modelul matematic (continuare)

Recurența (4) este utilizată pentru calcularea matricei de valori optime, care la rândul ei, va fi la baza construirii unui arbore binar de căutare optimal.

Teorema

Arborele binar de căutare construit pe baza matricei C este optimal.

Demonstrație.

Relația (4) asigură faptul că $c_{0,n}$ este minimum între toate costurile arborilor binari de căutare construiți cu secvența $(a_1, a_2, \ldots, a_n)$. Astfel, arborele binar de căutare construit pe baza matricei C are cost minim deci este optimal. □

Arbori binari de căutare optimali - algoritm (descriere)

- Matricea C se poate calcula aplicând relația de recurență (4) pentru $i-j=1$, apoi $j-i=2,\ldots,j-i=n$.
- Pentru a pregăti informațiile necesare construirii efective a arborelui binar de căutare optimal, vor fi calculate și valorile $r_{i,j}$, unde $r_{i,j}$ semnifică indicele nodului rădăcină al subarborelui optimal format din secvența $(a_{i+1},a_{i+2},\ldots,a_j)$.
- Valorile inițiale sunt $c_{i,i}=0, w_{i,i}=q_i,\ 0\leq i\leq n$.
- În plus, se folosește relația $w_{i,j}=p_j+q_j+w_{i,j-1}$.

Arbori binari de căutare optimali - algoritm(pseudocod)

```
procedure abc_opt(C,R,W,P,Q,n)
      for i ← 1 to n-1 do
        C[i,i] ← 0; R[i,i]← 0; W[i,i]← Q[i]
        C[i,i+1]← Q[i]+P[i+1]+Q[i+1]
        R[i,i+1]← i+1
        W[i,i+1]← Q[i]+P[i+1]+Q[i+1]
      W[n,n]← Q[n]; R[n,n]← 0; C[n,n]← 0
      for d ← 2 to n do
        for i ← 0 to n-d do
          j ← i+d
          W[i,j]←P[j]+Q[j]+W[i,j-1]
          fie r indicele pentru care se obţine valoarea minimă pentru
              {C[i,k-1]+C[k,j]/k=i+1,i+2,...,j}
          C[i,j]←C[i,r-1]+C[r,j]+W[i,j]
          R[i,j]←r
     end
```

Evaluare. Complexitatea timpului de execuție este $O(n^3)$.

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 $\text{RUCSAC}(n, M)$):

- 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:

$$f_j(X) = \begin{cases} -\infty, & \text{dacă } X < 0 \\ 0, & \text{dacă } j = 0 \text{ şi } X \geq 0 \\ \max\{f_{j-1}(X), f_{j-1}(X-w_j) + p_j\}, & \text{dacă } j > 0 \text{ şi } X \geq 0 \end{cases} \quad (5)$$

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

Problema rucsacului, varianta discretă - model matematic (continuare)

- Din relația (5) 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 (5) 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 (5)≡(6)

$$f_j(X) = \begin{cases} -\infty, & \text{dacă } X < 0 \\ 0, & \text{dacă } j = 0 \text{ şi } X \geq 0 \\ \max\{f_{j-1}(X), f_{j-1}(X - w_j) + p_j\}, & \text{dacă } j > 0 \text{ şi } X \geq 0 \end{cases} \quad (6)$$

- 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

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

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

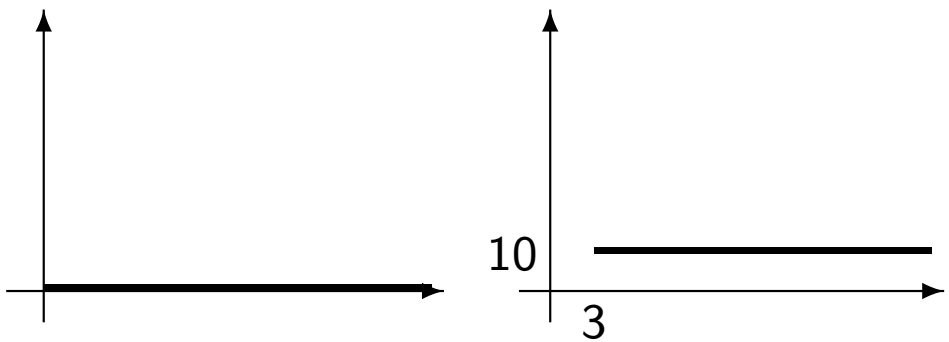


Figura 2: Funcțiile f_0 și g_0

Problema rucsacului, varianta discretă - exemplu

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

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

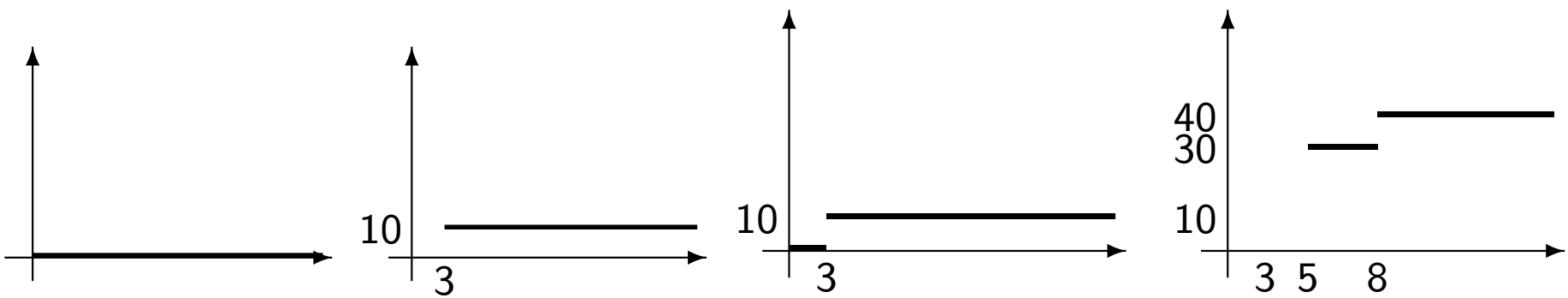


Figura 3: Funcțiile f_0 și g_0; Funcțiile f_1 și g_1

Problema rucsacului, varianta discretă - exemplu

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

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

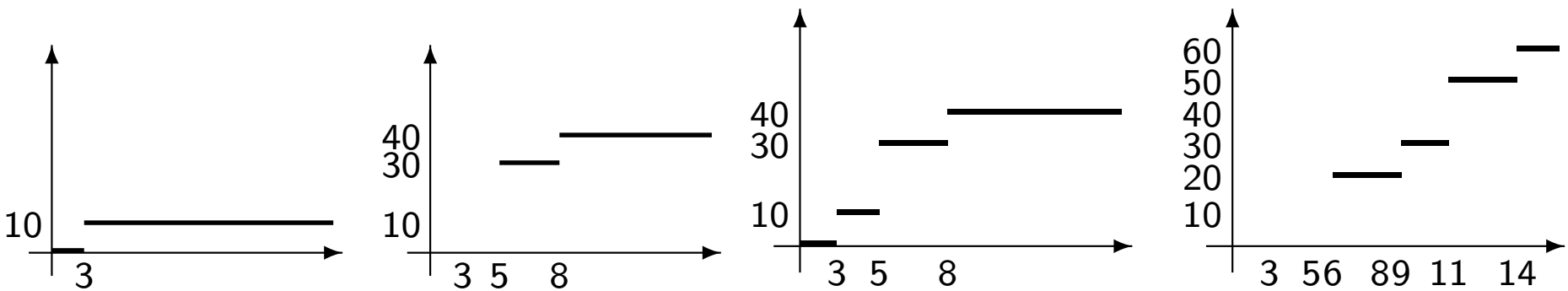


Figura 4: Funcțiile f_1 și g_1; Funcțiile f_2 și g_2

Problema rucsacului, varianta discretă - exemplu

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

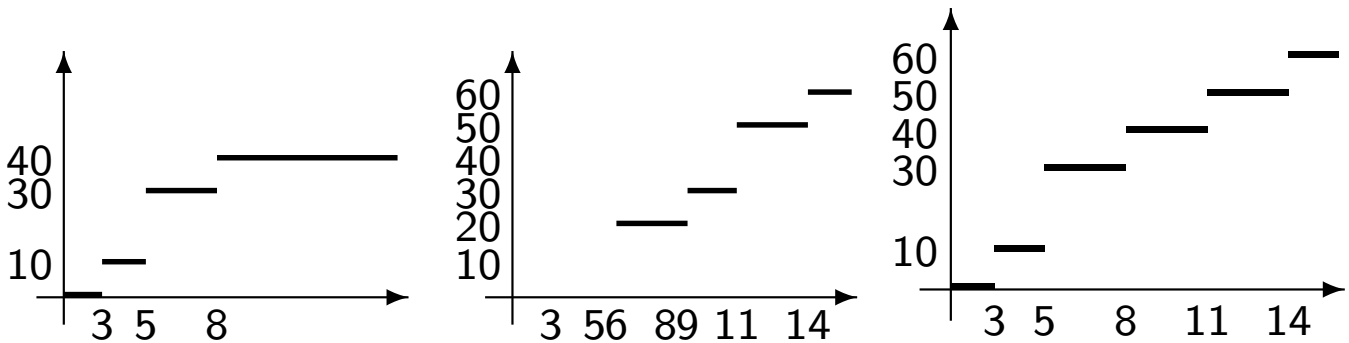


Figura 5: Funcțiile f_2 și g_2; Funcția f_3

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(Si,Ti)` 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
```

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

Backtracking și *Branch-and-Bound*

Mitică Craus

Univeristatea Tehnică "Gheorghe Asachi" din Iași

Cuprins

Paradigmele *Backtracking* și *Branch-and-Bound* - descriere

- *Backtracking* și *Branch-and-Bound* sunt două strategii care îmbunătățesc căutarea exhaustivă.
- Un algoritm de căutare exhaustivă este definit după următoarea schemă:
  1. Se definește spațiul *soluțiilor candidat (fezabile)* $\mathbb{U}$.
  2. Cu ajutorul unui algoritm de enumerare se selectează acele soluții candidat care sunt soluții ale problemei.
- În contrast cu căutarea exhaustivă, paradigmele *backtracking* și *branch-and-bound* propun o căutare sistematică în spațiul soluțiilor.

Paradigmele *Backtracking* și *Branch-and-Bound* - descriere (continuare)

1. Spațiul soluțiilor este organizat ca un arbore cu rădăcină astfel încât există o corespondență bijectivă între vârfurile de pe frontiera arborelui și soluțiile candidat.
   - Soluția candidat este descrisă de drumul de la rădăcină la vârful de pe frontieră corespunzător.
   - Un exemplu de organizare a spațiului soluțiilor ca arbore este reprezentat în figura 1.
2. Căutarea sistematică se realizează cu ajutorul unui algoritm de explorare sistematică a acestui arbore.
3. Vârfurile arborelui sunt clasificate astfel:
   a) un vârf *viabil* este un vârf pentru care sunt șanse să se găsească o soluție a problemei explorând subarborele cu rădăcina în acel vârf;
   b) un vârf *activ (live)* este un vârf care a fost vizitat cel puțin o dată de algoritmul de explorare și urmează să mai fie vizitat cel puțin încă o dată;
   c) un vârf activ după ce a fost vizitat ultima dată de algoritmul de explorare devine *inactiv (death)*.
4. Sunt explorați numai subarborii cu rădăcini viabile.
   - În felul acesta se evită procesarea inutilă a subarborilor pentru care suntem siguri că nu conțin soluții.
5. Cele două paradigme, *backtracking* și *branch-and-bound*, diferă doar prin modul în care explorează lista vârfurilor viabile din arbore.

Organizare a spațiului soluțiilor ca arbore (exemplu)

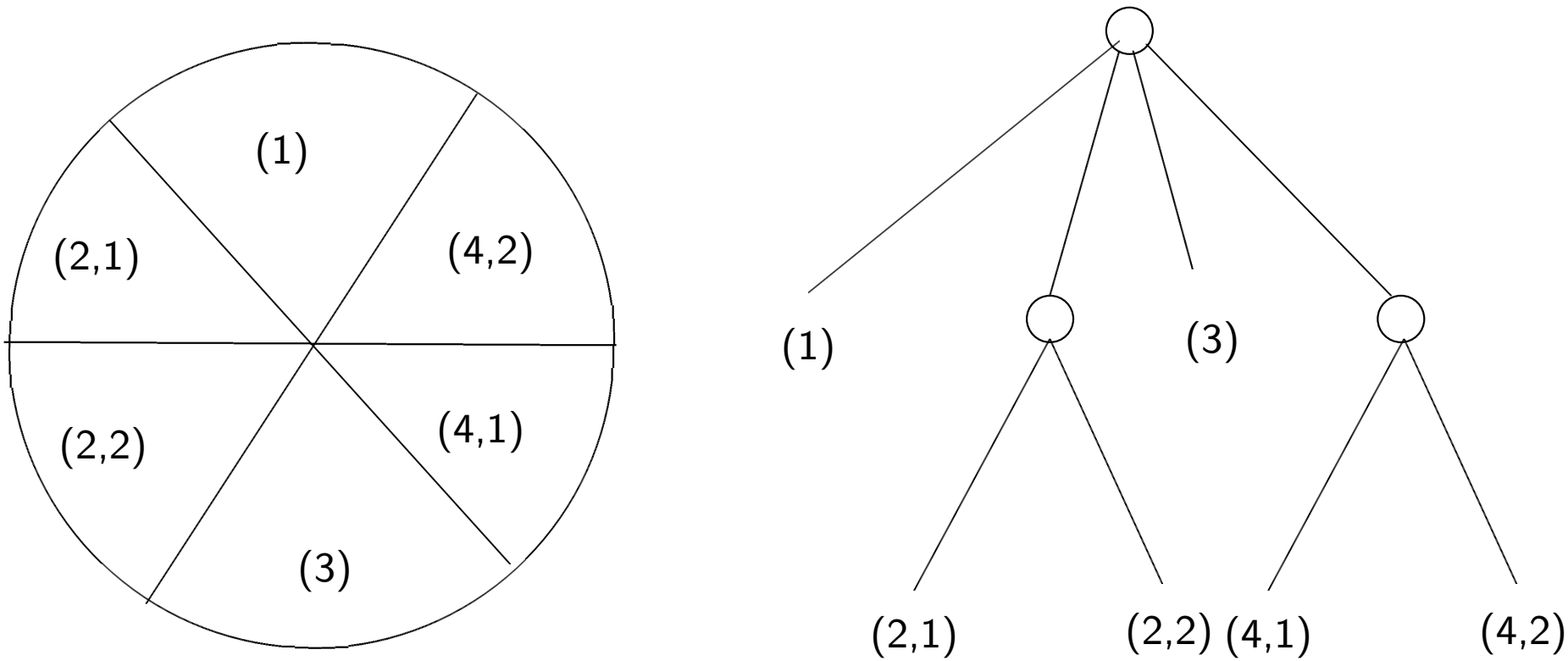


Figura 1 : Spațiului soluțiilor ca arbore

Spațiul soluțiilor candidat

- Metodele *backtracking* și *branch-and-bound* rezolvă o problemă prin căutarea sistematică a soluției în spațiul soluțiilor candidat.
- Mulțimea elementelor produsului cartezian, submulțimile unei mulțimi, mulțimea permutărilor unei mulțimi și drumurile într-un graf sunt spațiile de căutare cele mai utilizate.
- Presupunem că o soluție candidat este soluție a problemei dacă satisface o anumită condiție, notată cu ST ($eSTe$), ce poate fi testată în timp polinomial.
- Putem privi ST ca o funcție $ST : \mathbb{U} \rightarrow Boolean$.

Paradigma *Backtracking* - descriere

1. Se definește o funcție criteriu prin care se stabilește dacă un vârf este viabil sau nu.
2. Arborele este explorat prin algoritmul DFS.
3. Fie $x^k = (x_0, ..., x_k)$ secvența care descrie drumul de la rădăcina la vârful curent:
   - 3.1 dacă vârful curent este pe frontieră, atunci se verifică dacă x^k este soluție;
   - 3.2 dacă vârful curent nu este pe frontieră, se alege următorul succesor viabil (dacă există).

- Dacă $x^k = (x_0, ..., x_k)$ este secvența care descrie drumul de la rădăcina la vârful curent, se notează cu $T(x^k)$ mulțimea tuturor valorilor posibile pentru x_{k+1}, astfel încât secvența $x^{k+1} = (x_0, x_1, ..., x_k, x_{k+1})$ descrie de asemenea un drum de la rădăcină către o frunză.

Modelul general de algoritm *bactracking* - varianta nerecursivă

```
procedure backtracking(n)
   k ← 0
   while (k ≥ 0) do
      if (∃ y∈ T(x^k) neîncercat and viabil(y))
         then x_{k+1} ← y
              if ST(x^{k+1})
                 then scrie(x^{k+1})
                 else k ← k+1
         else k ← k-1
end
```

Modelul general de algoritm *bactracking* varianta recursivă

```
procedure backtrackingRec(x^k)
    k ← 0
    for each y∈ T(x^k) neîncercat and viabil(y) do
        x_{k+1} ← y
        if ST(x^{k+1})
            then scrie(x^{k+1})
            else backtrackingRec(x^{k+1})
end
```

Modelul general de algoritm *bactracking* în cazul în care spațiul soluțiilor candidat este produsul cartezian $\{0,1,\ldots,m-1\}^n$ - varianta nerecursivă

```
procedure backtrack(n, m)
   k ← 0
   x[0] ← -1
   while (k ≥ 0) do
      if (x[k] < m-1)
         then repeat
                 x[k] ← x[k]+1
              until(viabil(x, k) or (x[k]=m-1))
              if (viabil(x, k))
                 then if ((k = n-1) and ST(x))
                         then scrieElement(x,n)
                         else k ← k+1
                              x[k] ← -1
         else k ← k-1
end
```

Modelul general de algoritm *bactracking* în cazul în care spațiul soluțiilor candidat este produsul cartezian $\{0, 1, \ldots, m-1\}^n$ - varianta recursivă

```
procedure backtrackRec(x, k)
   for j ← 0 to m-1 do
       x[k] ← j
       if ((k = n-1) and ST(x))
          then scrieElement(x, n)
          else if (viabil(x, k))
                  then backtrackRec(x, k+1)
end
```

Problema celor n regine - descriere

- Se consideră o tablă de șah de dimensiune $n \times n$ și n regine.
- Problema constă în așezarea pe tabla de șah a celor n regine, astfel încât să nu se captureze una pe alta.
- Dacă există mai multe asemenea așezări, atunci se vor determina toate.

Problema celor n regine - modelul matematic

- Pozițiile de pe tabla de șah sunt identificate prin perechi (i,j) cu $1 \leq i,j \leq n$. De exemplu, pentru $n = 4$, tabla este cea reprezentată în figura 2.

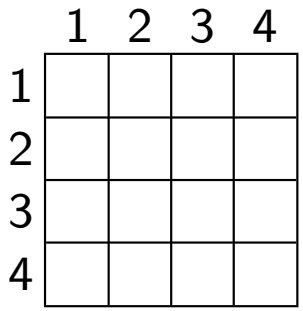


Figura 2 : Tabla de șah 4×4

- O soluție poate fi reprezentată printr-o funcție $Q : \{1,\ldots,n\} \times \{1,\ldots n\} \rightarrow \{false, true\}$ cu semnificația: $Q_{i,j} = Q(i,j) = true$ dacă și numai dacă pe poziția (i,j) se află o regină.
- Se observă că spațiul soluțiilor potențiale conține 2^{n^2} elemente.
- O reducere substanțială a acestuia se obține dacă se reprezintă soluțiile printr-o funcție $s : \{1,\ldots,n\} \rightarrow \{1,\ldots,n\}$ cu semnificația: $s_i = s(i) = j \Leftrightarrow Q(i,j) = true$.
- În acest caz, spațiul soluțiilor potențiale conține n^n elemente. De exemplu, pentru $n = 8$ avem $2^{n^2} = 2^{64}$, iar $8^8 = 2^{24}$.

Problema celor n regine - modelul matematic (continuare)

- Alegerea reprezentării pentru soluțiile potențiale este foarte importantă pentru metoda backtracking: o alegere bună poate conduce la redecerea substanțială a dimensiunii spațiului acestor soluții.
- Cele n^n elemente ale spațiului soluțiilor potențiale pot fi reprezentate printr-un arbore cu n nivele în care fiecare vârf are exact n succesori imediați.
- Pentru cazul $n = 4$, arborele este cel reprezentat în figura 3.

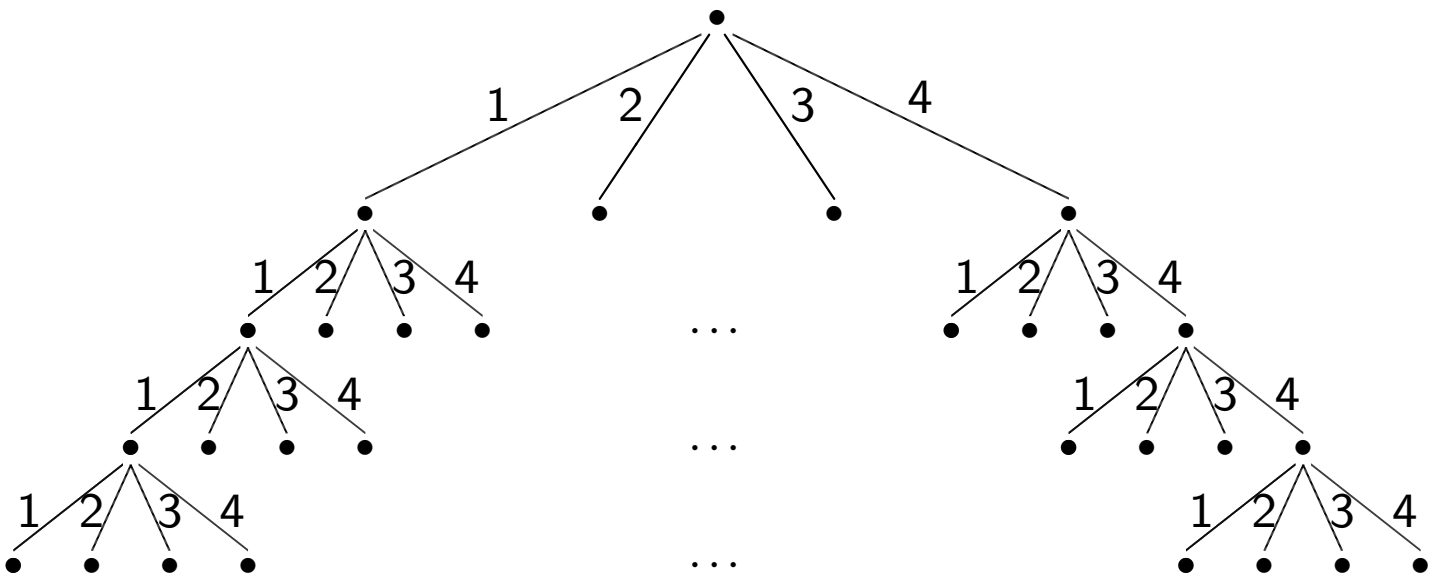


Figura 3 : 4 regine – reprezentarea spațiului soluțiilor potențiale

Problema celor n regine - modelul matematic (continuare)

- În continuare vom căuta criteriul de mărginire prin care să eliminăm acei subarbori care nu conțin vârfuri-soluție.
- Să observăm că s_k candidează la soluție, dacă așezând cea de-a k-a regină pe poziția (k, s_k), ea nu este atacată de și nu atacă nici una dintre cele $k-1$ regine așezate pe liniile $1, \ldots, k-1$.
- Ecuațiile celor patru direcții pe care se poate deplasa o regină sunt date de:
  Regina de pe poziția (i,j) *se poate deplasa pe poziția* (k,ℓ) *dacă și numai dacă*
  - $i = k$ (deplasare pe orizontală) sau
  - $j = \ell$ (deplasare pe verticală) sau
  - $i - j = k - \ell$ (deplasare pe o diagonală principală) sau
  - $i + j = k + \ell$ (deplasare pe o diagonală secundară).
- Considerând $(i,j) = (i, s_i)$, $(k,\ell) = (k, s_k)$ și $\neg = NOT$ obținem condiția care reprezintă criteriul de mărginire:

$$C(s_1, \ldots, s_k) \quad = \quad \forall i (1 \leq i \leq k-1): \quad \neg Q(i, s_k) \wedge \neg (i - s_i = k - s_k \vee i + s_i = k + s_k))$$

- Ținând cont de:

$$i - s_i = k - s_k \Rightarrow s_i = s_k - k + i \text{ și } i + s_i = k + s_k \Rightarrow s_i = s_k + k - i$$

obținem forma echivalentă:

$$C(s_1, \ldots, s_k) = \forall i (1 \leq i \leq k-1): \neg Q(i, s_k) \wedge \neg Q(i, s_k + k - i) \wedge \neg Q(i, s_k - k + i)$$

Problema celor n regine - modelul matematic (continuare)

- Pentru a ne face o idee cât de mult taie C din arborele spațiului soluțiilor potențiale, prezentăm arborele parțial obținut în urma aplicării criteriului de mărginire pentru cazul $n = 4$ în figura 4.

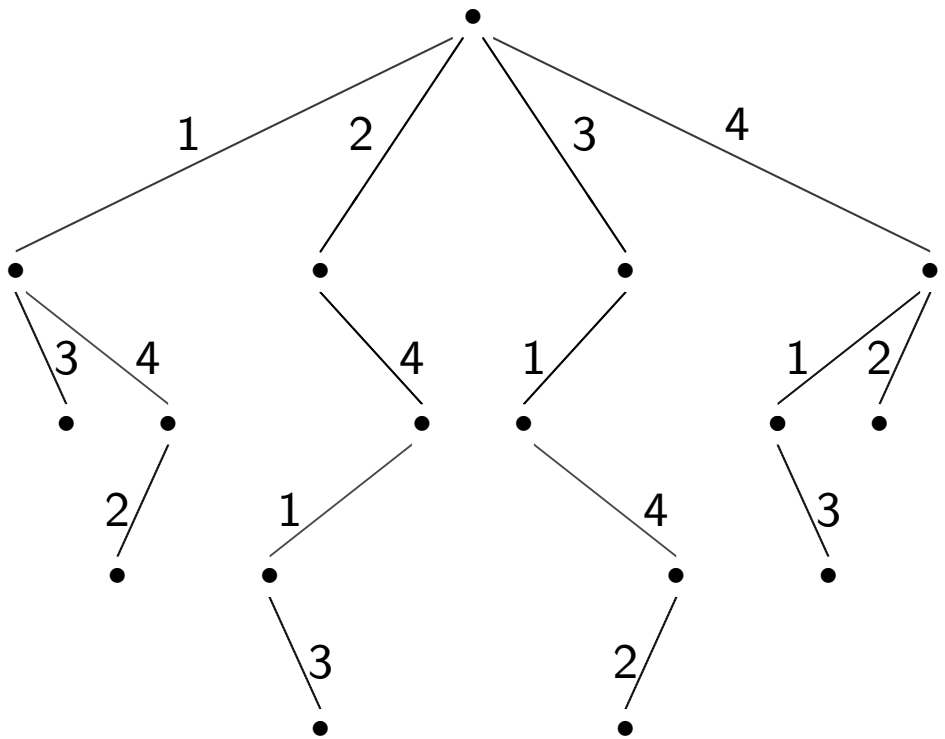


Figura 4 : 4 regine – arborele parțial

Problema celor n regine - algoritm

- Scrieți pseudocodul algoritmului pentru problema celor n regine, folosind varianta recursivă a modelului general de algoritm *bactracking*, în cazul în care spațiul soluțiilor candidat este produsul cartezian $\{0, 1, \ldots, m-1\}^n$

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 5.

Problema submulțimilor de sumă dată - modelul matematic (continuare)

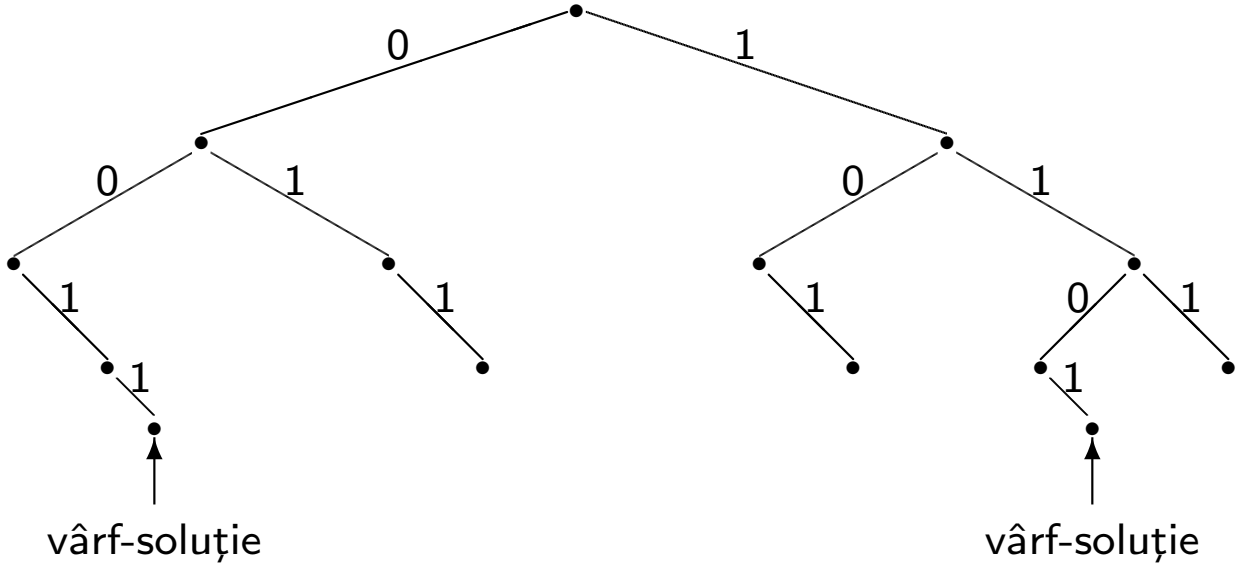


Figura 5 : Arbore parțial pentru submulțimi de sumă dată

- 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 $k + 1 \leq n$.

Paradigma *Branch-and-bound* - descriere

- Presupunem că spațiul soluțiilor candidat (fezabile) este S și, ca și în cazul paradigmei *backtracking*, este organizat ca un arbore.
- Rădăcina arborelui corespunde problemei inițiale (căutării în S), iar fiecare vârf intern corespunde unei subprobleme a problemei inițiale.
- *Branch-and-bound* dezvoltă acest arbore conform următoarelor reguli:
  1. Fiecare vârf viabil este explorat o singură dată.
  2. Când un vârf viabil este explorat, toți fiii viabili sunt generați și memorați într-o *structură de date de așteptare* pe care o notăm cu A.
     - Inițial, A conține doar rădăcina.
  3. Următorul vârf explorat este ales din structura de așteptare A.

Paradigma *Branch-and-bound* - implementarea structurii de așteptare

- *Coadă*.
  - Arborele este explorat la fel ca în cazul algoritmului BFS.
- *Heap*.
  - Se aplică pentru problemele de optim: pentru probleme de minimizare se alege min-*heap*, iar pentru probleme de maximizare se alege max-*heap*.
  - Se asociază o funcție *predictor* (valoare de cost aproximativă) pentru fiecare vârf și valoarea acesteia va constitui cheia în *heap*.
- *Stivă*.
  - Arborele este explorat în lățime, dar într-o manieră diferită de BFS.

Paradigma *Branch-and-bound* - Exemplu de implementare a structurii de așteptare ca stivă

- Se consideră arborele spațiului de soluții din figura 6
- În cazul cozii ordinea explorării vârfurilor este (1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11), iar în cazul stivei ordinea este (1, 4, 10, 3, 9, 8, 7, 2, 6, 11, 5).
- Deși pare paradoxal, principiul de bază al metodei *branch-and-bound* este respectat și în cazul stivei: când un vârf viabil este explorat, toți fiii viabili sunt generați și memorați.
- În momentul în care este explorat vârful 1, sunt generate și memorate în stivă vârfurile 2, 3 și 4.
- Următorul vârf explorat este 4, situat în vârful stivei.
- Procesul continuă cu acest vârf.

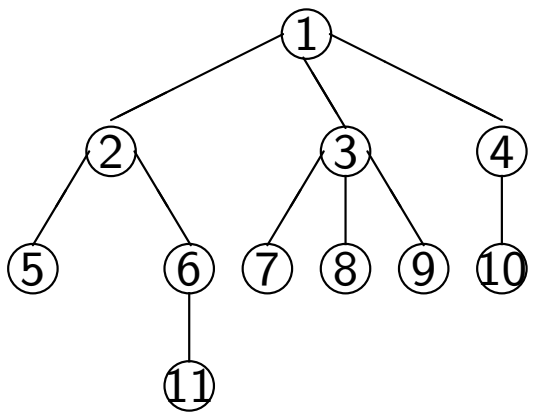


Figura 6 : Arbore de soluții pentru *branch-and-bound*

Modelul general de algoritm *branch-and-bound* - varianta nerecursivă

- *Branch-and-Bound* este termenul generic pentru toate tehnicile care au în comun:
  - Vârful curent explorat este complet expandat (se generează toți fiii și se memorează în structura de așteptare),
  - Se alege pentru explorare, după o strategie oarecare, un nou vârf.

```
procedure BranchAndBound()
    fie r rădăcina
    A ← {r}
    while A ≠ ∅ do
          selectează v din A
          A ← A \ {v}
          if v este pe frontieră
             then return solutie(v)
             else A ← A ∪{v_i | 1 ≤ i ≤ k, v_i fiu viabil al lui v}
    throw 'Nu există soluţie.'
end
```

Branch-and-Bound pentru probleme de optim

- Se aplică problemelor de optim pentru care nu există algoritmi *greedy* sau bazați pe programare dinamică.
- În general, sunt probleme $\mathbb{NP}$-dificile.
- Se poate aplica și problemelor care nu sunt de optim, dar care pot fi formulate ca probleme de optim prin adăugarea unei funcții de cost care să selecteze soluția (soluțiile) dorite; un astfel de exemplu este dat în secțiunea 4.
- Reamintim că fiecare vârf v descrie o *soluție potențială* x^k (drumul de la rădăcină la v); prin $c(v)$ notăm valoarea $c(x^k)$.
- Mulțimea soluțiilor potențiale o include pe cea a soluțiilor candidat; presupunem că funcția c este definită peste soluțiile potențiale.

Branch-and-bound cu cost minim

1. Se consideră o *funcție de predicție* c^* definită pentru fiecare vârf v din arbore, care trebuie să satisfacă următoarele proprietăți:
   a) $c^*(v) \leq c(v)$ pentru fiecare vârf v;
   b) dacă v este pe frontieră, atunci $c^*(v) = c(v)$;
   c) dacă w este fiu al lui v, atunci $c^*(v) \leq c^*(w)$; prin tranzitivitate, obținem $c^*(v) \leq c^*(w)$ pentru orice descendent w al lui v.
2. Structură de așteptare A este un min-heap pentru ca la fiecare pas să fie ales elementul cu $c^*(v)$ minim.

Modelul general de algoritm *branch-and-bound* pentru probleme de optim

```
procedure BBcostMin()
    fie r rădăcina
    calculează c*(r)
    A ← {(r, c*(r))}
    while (A ≠ ∅) do
            selectează v din A cu c*(v) minim
            A ← A \ {(v,c*(v)}
            if v este pe frontieră
                then return solutie(v)
                else fie v_1,...,v_k fiii viabili ai lui v
                     calculează c*(v_1),...,c*(v_k)
                     A ← A ∪{(v_i,c*(v_i)) | 1 ≤ i ≤ k}
    throw 'Nu există soluţie.'
end
```

Operațiile peste A trebuie citite ca operații peste un min-heap.

Structura standard a funcției c^*

- Structura standard a funcției c^* este

$$c^*(v) = f(h(v)) + g^*(v)$$

unde:

  - $h(v)$ este costul drumului de la rădăcină la v (de exemplu, lungimea drumului);
  - f este o funcție crescătoare;
  - $g^*(v)$ este costul subestimat al obținerii unei soluții pornind din v; $g^*(v)$ va constitui componenta optimistă a costului $c^*(v)$.

- Calculul lui $g^*(v)$ nu este întodeauna simplu. Nu se poate da o formulă generală pentru această funcție.

Corectitudinea algoritmului `BBcostMin()`

Teorema

Dacă v este vârful calculat de `BBCostMin(r,c*)`, *atunci drumul de la r la v reprezintă soluția de cost minim, adică pentru orice alt vârf s de pe frontieră, avem $c(s) \geq c(v)$.*

Demonstrație.

Vârful v este primul vârf de pe frontieră extras din structura de așteptare A, deci celelalte vârfuri de pe frontieră se află în structura de așteptare A sau sunt descendenți ai unor vârfuri din această structură. Din proprietătile funcției c^* și ale lui v rezultă:

- $c(v) = c^*(v) \leq c^*(w)$, pentru orice vârf w din structura de așteptare,
- $c^*(w) \leq c^*(s) = c(s)$, pentru orice vârf s de pe frontieră, descendent al unui vârf w din structura de așteptare.

În consecință, pentru orice vârf s de pe frontieră, $c(s) \geq c(v)$. □

"Perspico" - descrierea problemei

- Considerăm următoarea variantă simplificată a jocului *Perspico*.
- Fie o rețea formată din 3×3 pătrate, numite *locații*.
- Opt din cel nouă locații sunt ocupate de piese etichetate cu litere de la A la H, de dimensiuni potrivite astfel încât să poată fi deplasate pe orizontală sau verticală atunci când există o locație vecină liberă.
- O așezare a piesleor în care primele opt locații sunt ocupate în ordinea A, B, C, D, E, F, H se numește *configurație finală* și o notăm cu C_f (a se vedea figura 7).
- O *mutare* constă în deplasarea unei piese în locația liberă atunci când este posibil.
- Problema este următoarea:
  1. dată o configurație C, să se decidă dacă există o listă de mutări care să permită obținerea lui C_f din C;
  2. în cazul în care există, să se determine o astfel listă de mutări.

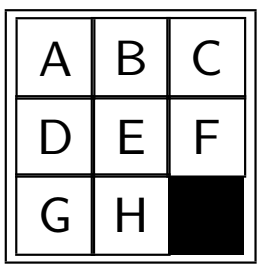


Figura 7 : Jocul *Perspico*

"Perspico" - modelul matematic

- Spațiul soluțiilor conține 9! combinații (∞ dacă nu se verifică ciclitățile).
- În plus, interesează modul de obținere a soluțiilor, adică drumul de la configurația C la configurația C_f.
- Dată o configurație finală C, se poate decide dacă există o listă de mutări care să permită obținerea lui C_f din C.
- Se definește funcția injectivă *poz*:

$$poz_C : \{A,B,C,...,L\} \to \{1,2,3,...,8,9\}$$

unde $poz_C(X) = i$ semnifică faptul că piesa notată cu X se află pe poziția i. Locația liberă a fost asociată cu piesa simbolică L.

- În figura 8, $poz_C(D) = 2$, $poz_C(L) = 8$.

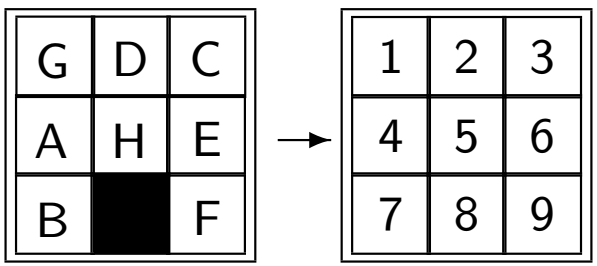


Figura 8 : Configurația definită prin funcția *poz*

"Perspico" - modelul matematic (continuare)

- Pentru o configurație C și pentru fiecare piesă de valoare X se definește

$$less_C(X) = \#\{Y | 1 \leq Y \leq L, Y < X, poz(Y) > poz(X)\}.$$

- Peste mulțimea {A,B,C,...,H,L}, se consideră ordinea alfabetică.
- În exemplul anterior avem $less_C(H) = 3$, $less_C(L) = 1$.

Teorema

Fie C o configurație, astfel încât $i, j \in \{1,2,3\}$ desemnează linia și coloana unde este plasată locația liberă, piesa L, și

$$l(C) = \begin{cases} 0, & \text{dacă } i+j \text{ este par;} \\ 1, & \text{dacă } i+j \text{ este impar.} \end{cases}$$

Atunci nu există un șir de transformări până la configurația finală C_f dacă:

$$S(C) = \sum_{X=A}^{L} less_C(X) + l(C) \text{ este număr impar.}$$

"Perspico" - implementare

- Pentru a rezolva problema determinării listei de mutări care să permită obținerea lui C_f din C, dacă această există, poate fi aplicat algoritmul *branch-and-bound* cu cost minim.
- În acest caz $c^*(v) = f(v) + g^*(v)$, unde:
  - $f(v) = \text{nivel}(v)$,
  - $g^*(v) =$ numărul de pieselor care nu sunt pe pozițiile definite de C_f.