Proiectarea Algoritmilor

Greedy

Greedy

- Metodă de rezolvare eficientă a unor probleme de optimizare
- Se pleacă de la o soluție parțială elementară
- Există un criteriu de optim local
- Se extind soluțiile partiale pînă ce se obtine o soluție finală – criteri de validare a soluției finale

Schema Greedy

Soluții-parțiale← {Soluție-parțială-elementară$_1$,
Soluție-parțială-elementară$_2$,...}

Repetă

Soluție-parțială←Alege-pentru-extindere (Soluții-parțiale,
Criteriu-de-optim)

Dacă Criteriu-de-finiș(Soluție-parțială)

atunci Întoarce Soluție-parțială

Soluții-parțiale← Soluții-parțiale U {Soluție-parțială}

Comparație D&I și Greedy

- D&I: top-down; Greedy: bottom-up (top-down?)
- Criteriu de optim? D&I: nu; Greedy: da

Discuție:

Când merge prost Greedy?

Exemplu (I)

- Problema rucsacului
  - Trebuie să umplem un rucsac de capacitate maxima M kg cu obiecte care au greutatea m_i şi valoarea v_i.
  - Putem alege mai multe obiecte din fiecare tip cu scopul de a maximiza valoarea obiectelor din rucsac.

Exemplu (II)

- Problema rucsacului
  - Trebuie să umplem un rucsac de capacitate maxima M kg cu obiecte care au greutatea m_i şi valoarea v_i.
  - Putem alege mai multe obiecte din fiecare tip cu scopul de a maximiza valoarea obiectelor din rucsac.
    - Varianta 1: putem alege fracţiuni de obiect – “problema continuă”
    - Varianta 2: nu putem alege decât obiecte întregi (număr natural de obiecte din fiecare tip), cu versiunea ”problema 0-1”

Exemplu (III)

- varianta 1: algoritm greedy
  - sortăm obiectele după raportul v_i/m_i
  - adăugăm obiectele cu cele mai mari valori per kg şi apoi adăugăm fracţiuni din următorul
  - Exemplu:
    - M=10kg
    - m_1=5kg,v_1=10, m_2=8kg, v_2=19, m_3=4kg, v_3=4
    - Soluţie: ($m_{2,}$, v_2) 8kg şi 2kg din (m_1,v_1)

Exemplu (IV)

- varianta 2: algoritmul greedy nu funcţionează
  - Contraexemplu:
    - rezultat corect - 2 obiecte (m_1,v_1)
    - rezultat alg. greedy – 1 obiect (m_2,v_2)

Arbori Huffman

Arbori Huffman

- Metoda de codificare folosita la compresia fișierelor
- Construcția unui astfel de arbore se realizează printr-un algoritm greedy
- Exemplu:
  - ana are mere – 12*8 biti=96 biti
  - a - 00; e -11; r - 010; ’ ‘ - 011; m - 100;
  - n – 101 –
  - 6*2+6*3=12+18=30 biti
  - Compresie de 30/96 ~ 66%

Arbori Huffman – Definitii (I)

- K – mulțimea de simboluri ce vor fi codificate

- Arbore de codificare a cheilor K este un arbore binar ordonat cu proprietățile:
  - Doar frunzele conțin cheile din K; nu exista mai mult de o cheie intr-o frunză
  - Toate nodurile interne au exact 2 copii
  - Arcele sunt codificate cu 0 si 1 (arcul din stânga unui nod – codificat cu 0)

- k = Codul unei chei – este șirul etichetelor de pe calea de la rădăcina arborelui la frunza care conține cheia k (k este din K).

- p(k) – frecvența de apariție a cheii k in textul ce trebuie comprimat.

- Ex pentru “ana are mere”:
  - p(a) = p(e) = 0.25; p(n) = p(m) = 0.083;p(r) = p( ) = 0.166

Arbori Huffman – Definitii (II)

- A – arborele de codificare a cheilor
- lg_cod(k) – lungimea codului cheii k conform A
- nivel(k,A) – nivelul pe care apare in A frunza ce conține cheia K
- Costul unui arbore de codificare A al unor chei K relativ la o frecventa p este:

$$Cost(A) = \sum_{k \in K} lg\_cod(k) * p(k) = \sum_{k \in K} nivel(k, A) * p(k)$$

- Un arbore de codificare cu cost minim al unor chei K, relativ la o frecventa p este un arbore Huffman, iar codurile cheilor sunt coduri Huffman.

Arbori Huffman – algoritm de constructie (I)

- 1. pentru fiecare k din K se construiește un arbore cu un singur nod care conține cheia k si este caracterizat de ponderea w = p(k). Subarborii construiți formează o mulțime numita Arb.

- 2. Se aleg doi subarbori a şi b din Arb astfel incât a şi b au pondere minima.

Arbori Huffman – algoritm de constructie (II)

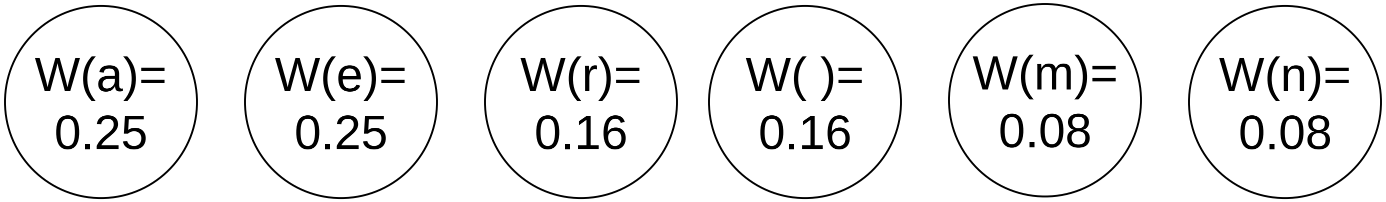
- 3. Se construieste un arbore binar cu o radacina *r* care nu contine nici o cheie si cu descendentii a si b. Ponderea arborelui este definita ca $w(r) = w(a) + w(b)$

- 4. Arborii a si b sunt eliminati din Arb iar r este inserat in Arb.

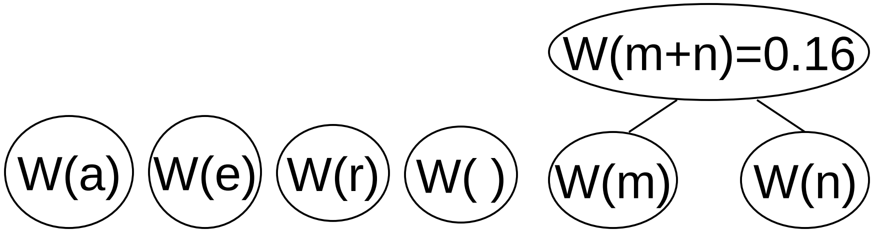
- 5. Se repeta procesul de constructie descris de pasii 2-4 pana cand multimea Arb contine un singur arbore – Arborele Huffman pentru cheile K

Arbori Huffman – Exemplu

- Text: ana are mere
- p(a) = p(e) = 0.25; p(n) = p(m) = 0.083; p(r) = p( ) = 0.166
- Pasul 1

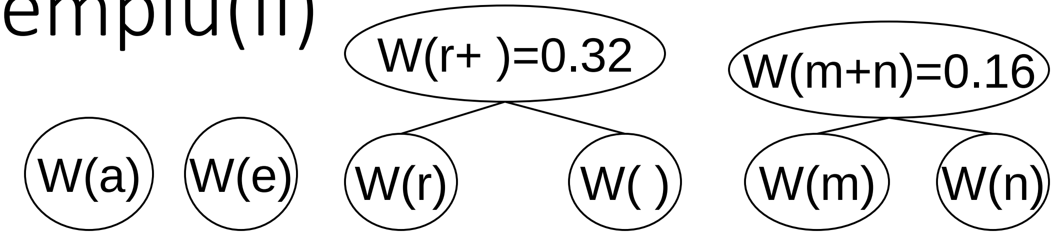


- Pasii 2-4

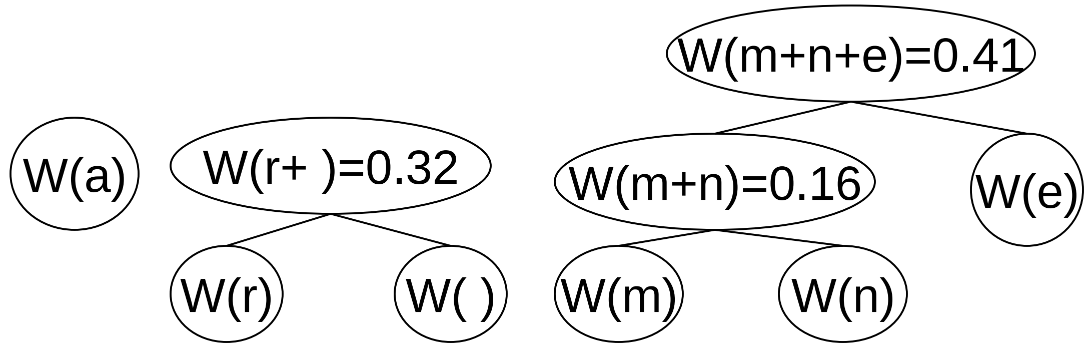


Arbori Huffman – Exemplu(II)

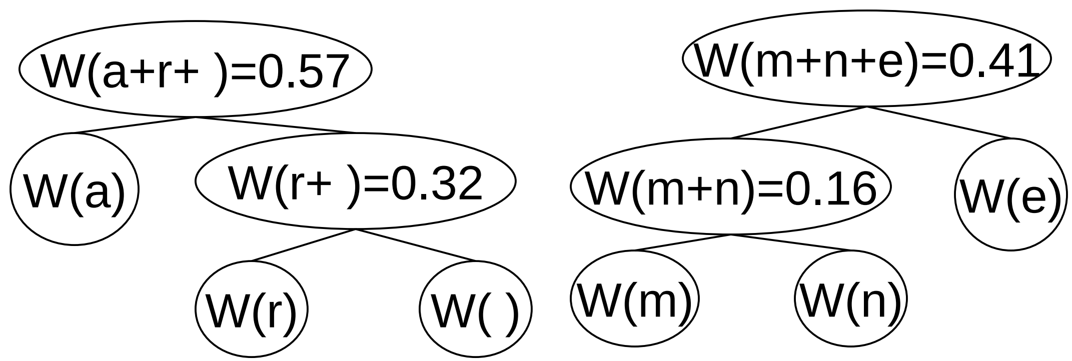
- Pasii 2-4 (II)



- Pasii 2-4 (III)

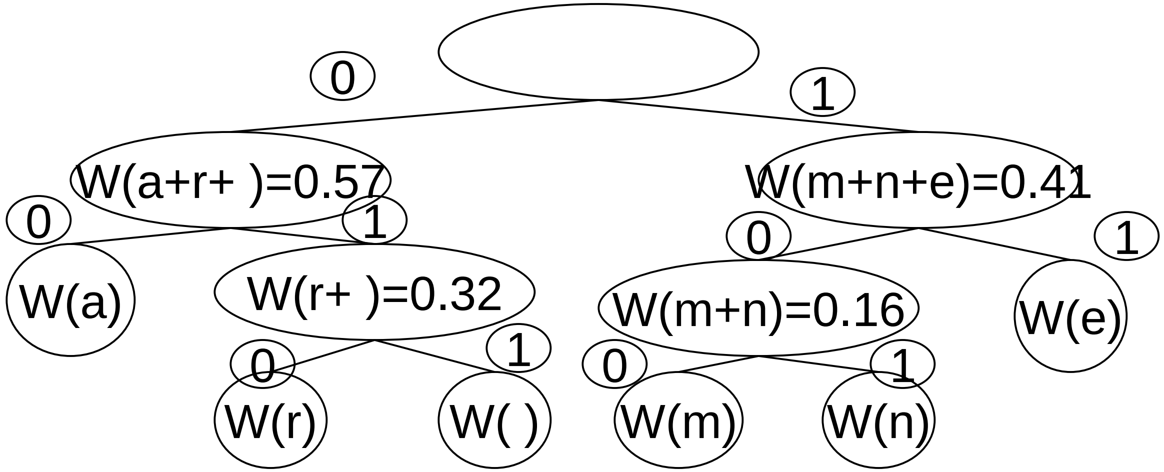


- Pasii 2-4 (IV)



Arbori Huffman – Exemplu (III)

- Pasii 2-4 (V)



- Codificare: a - 00; e -11; r - 010; ’ ‘ - 011; m - 100; n - 101;
- Cost(A) = 2 * 0.25 + 2 * 0.25 + 3 * 0.083 + 3 * 0.083 + 3 * 0.166 + 3 * 0.166 = 1 + 1.2 = 2.2 biti.

Arbori Huffman – Construcție

```
Huffman(K,p){
    Arb = {k ∈ K | frunza(k, p(k))};
    while (card(Arb) > 1)
      fie a1 si a2 arbori din Arb a.i. ∀a ∈ Arb a ≠ a1 si a ≠ a2, avem
            w(a1) ≤ w(a) ∧ w(a2) ≤ w(a));     // practic se extrage de doua ori minimul si se
                                              // salveaza in a1 si a2
      Arb = Arb \ {a1, a2} U nod_intern(a1, a2, w(a1) + w(a2));
    if(Arb = Φ) return arb_vid;
      else
         fie A singurul arbore din multimea Arb;
         return A;
```

Notatii folosite:

a = frunza(k, $p(k)$) – subarbore cu un singur nod care contine cheia k, iar $w(a) = p(k)$;

a = nod_intern(a_1, a_2, x) – subarbore format dintr-un nod intern cu descendentii a_1 si a_2 si $w(a) = x$

Arbori Huffman - Decodificare

Se incarca arborele si se decodifica textul din fisier conform algoritmului:

```
Decodificare (in, out)
    A = restaurare_arbore (in)   // reconstruiesc arborele
    while(! terminare_cod(in))   // mai am caractere de citit
        nod = A                  // pornesc din radacina
        while (! frunza(nod))    // cat timp nu am determinat caracterul
            if (bit(in) = 1) nod = dreapta(nod)     // avansez in arbore
                else nod = stanga(nod)
        write(out, cheie(nod))   // am determinat caracterul si il scriu
```

Demonstrația corectitudinii aplicării schemei greedy (I)

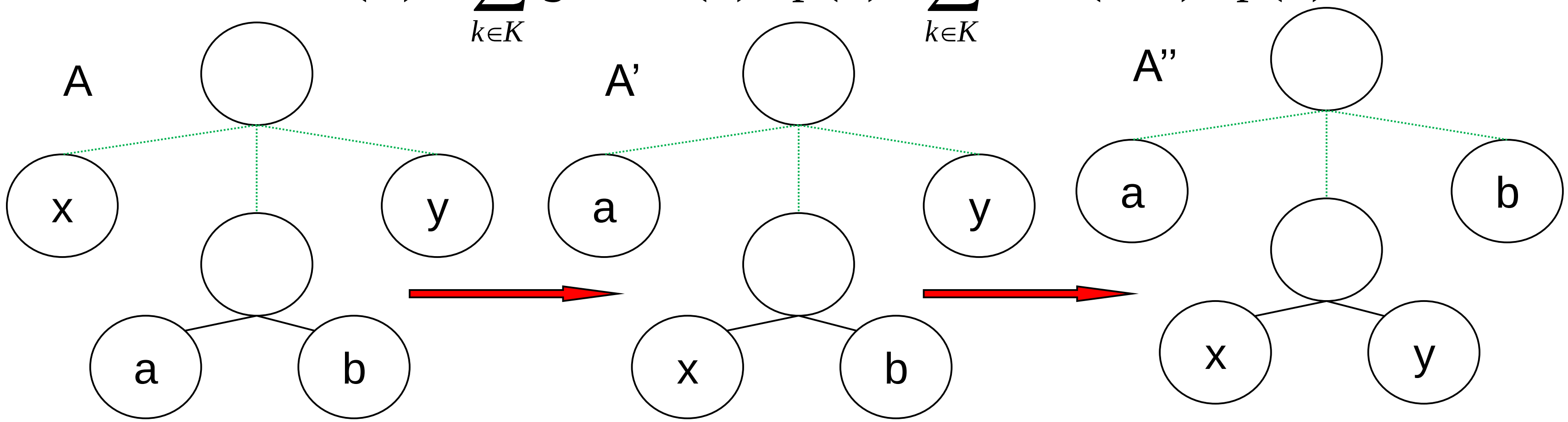
- Arborele de codificare construit trebuie să aibă cost minim pentru a fi arbore Huffman

- Lema 1. Fie K mulțimea cheilor dintr-un arbore de codificare, card(K) $\geq$ 2, x, y două chei cu pondere minimă. $\exists$ un arbore Huffman de înălțime h in care cheile x şi y apar pe nivelul h fiind descendente ale aceluiași nod intern.

Demonstrația corectitudinii aplicării schemei greedy (II)

- Demonstratie Lema 1:

$$Cost(A) = \sum_{k \in K} lg\_cod(k) * p(k) = \sum_{k \in K} nivel(k, A) * p(k)$$



- Se interschimbă a cu x şi b cu y şi din definiţia costului arborelui => cost(A'') ≤ cost(A') ≤ cost(A) => A'' arbore Huffman

Demonstratie (III)

- Lema 2. Fie A un arbore Huffman cu cheile K, iar x şi y două chei direct descendente ale aceluiaşi nod intern a. Fie K' = K \ {x,y} U {z} unde z este o cheie fictivă cu ponderea w(z) = w(x) + w(y). Atunci arborele A' rezultat din A prin inlocuirea subarborelui cu rădăcina a si frunzele x, y cu subarborele cu un singur nod care conţine frunza z, este un arbore Huffman cu cheile K'.

- Demonstratie
  - 1) analog Cost(A') ≤ Cost(A) (Cost(A) = Cost(A') + w(x) + w(y))
  - 2) pp există A'' a.i. Cost(A'') < Cost(A') =>
    - Cost(A'') < Cost(A) - (w(x) + w(y));
    - Cost(A'') + w(x) + w(y) < Cost(A); => A nu este Huffman (contradicţie)

Demonstrație (IV)

- Teoremă – Algoritmul Huffman construiește un arbore Huffman.

- Demonstrație prin inducție după numărul de chei din mulțimea K.

- $n \leq 2$ => evident

- $n > 2$
  - Ip. Inductivă: algoritmul Huffman construiește arbori Huffman pentru orice mulțime cu n-1 chei

  - Fie $K = \{k_1, k_2, \ldots , k_n\}$ a.i. $w(k_1) \leq w(k_2) \leq \ldots \leq w(k_n)$

Demonstrație (V)

- Cf. Lema 1, $\exists$ Un arbore Huffman unde cheile k_1, k_2 sunt pe același nivel şi descendente ale aceluiași nod.

- A_{n-1} – arborele cu n-1 chei $K' = K - \{k_1,k_2\} \cup z$ unde $w(z) = w(k_1) + w(k_2)$

- A_{n-1} rezultă din A_n prin modificările prezentate in Lema 2 => A_{n-1} este Huffman, şi cf. ipotezei inductive e construit prin algoritmul Huffman(K',p')

- => Algoritmul Huffman(K, p) construiește arborele format din k_1 si k_2 si apoi lucrează ca şi algoritmul Huffman(K', p') ce construiește A_{n-1} => construieste arborele Huffman(K, p)

Model abstract al algoritmilor greedy

- Fie E o mulţime finită nevidă şi I $\subset \mathcal{P}$(E) a.i. :
  - $\varnothing \in$ I,
  - $\forall$X $\subseteq$ Y şi Y $\in$ I=> X $\in$ I.
- Atunci spunem ca (E,I) este un sistem accesibil
- Submulţimile din I sunt denumite submulţimi independente:
- Exemple:
  - Ex1: E = {e_1, e_2, e_3} si I = {$\varnothing$, {e_1}, {e_2}, {e_3}, {e_1, e_2}, {e_2, e_3}} – mulțimile ce nu conțin e_1 si e_3.
  - Ex2: E – muchiile unui graf neorientat şi I mulţimea mulţimilor de muchii ce nu conţin un ciclu (mulțimea arborilor).
  - Ex3: E set de vectori dintr-un spaţiu vectorial, I mulţimea mulţimilor de vectori linear independenţi.
  - Ex4: E – muchiile unui graf neorientat şi I mulţimea mulţimilor de muchii în care oricare 2 muchii nu au un vârf comun.

Model al algoritmilor greedy (II)

- Un sistem accesibil este un matroid daca satisface proprietatea de interschimb:

  $X, Y \in I$ şi $|X| < |Y| \Rightarrow \exists e \in Y \setminus X$ a.i. $X \cup \{e\} \in I$

- Teorema. Pentru orice subset accesibil (E,I) algoritmul Greedy rezolvă problema de optimizare dacă şi numai dacă (E,I) este matroid.

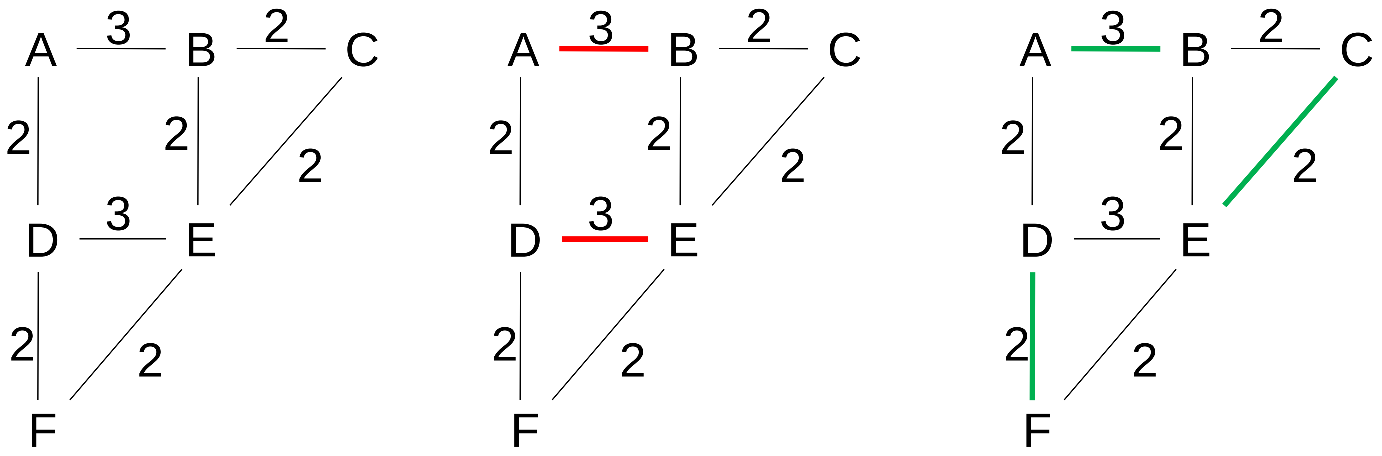
Verificăm exemplele

- Ex1: I = {∅, {e_1}, {e_2}, {e_3}, {e_1, e_2}, {e_2, e_3}}

Fie Y = {{e_1}, {e_2}, {e_3}, {e_1, e_2}} si X = {{e_1}, {e_3}}

→Y \ X = {{e_2}, {e_1, e_2}} → X $\cup$ {e_2} $\in$ I → matroid

- Ex4:



Algoritmul Greedy

Algoritmul generic Greedy devine:

$X = \emptyset$

sortează elementele din E în ordinea *descrescătoare* a ponderii

pentru fiecare element $e \in E$ (sortat) *repetă*

$X = X \cup \{e\}$ dacă şi numai dacă $(X \cup \{e\}) \in I$

întoarce X

Programare dinamica

Programare dinamică

- Programare dinamica
  - Descriere generala
  - Algoritm generic
  - Caracteristici
- Arbori optimi la cautare (AOC)
  - Definitii
  - Constructia AOC

Programare dinamică

- Descriere generală
  - Soluţii optime construite iterativ asamblând soluţii optime ale unor subprobleme (probleme similare de dimensiune mai mică)
- Algoritmi "clasici"
  - Problema rucsacului discretă (0-1)
  - Algoritmul Floyd-Warshall – determină drumurile de cost minim dintre toate perechile de noduri ale unui graf
  - AOC
  - Înmulţirea unui şir de matrici
  - Numere catalane
  - Viterbi
  - Distanta de editare

Algoritm generic

Soluții-parțiale^{0} $\leftarrow$ {Soluție-parțială-elementară$_{1}$, Soluție-parțială-elementară$_{2}$,...}

Pentru i=1 ***la*** n ***repetă***

Soluții-parțiale^{i} =combină(Soluții-parțiale$^{j<i}$,Criteriu-de-optim)

Întoarce Soluții-parțiale^{n}

Caracteristici

- O solutie optima a unei probleme contine solutii optime ale subproblemelor
- O solutie recursiva contine un numar mic de subprobleme distincte ce se repeta de multe ori

Diferente greedy – programare dinamică

Greedy

- Sunt mentinute solutiile partiale curente din care evolueaza solutiile partiale urmatoare

- Solutiile partiale anterioare sunt eliminate (ele sunt deja inglobate in solutiilor curente)

Programare dinamică

- Se pastreaza toate solutiile partiale

- La constructia unei solutii noi poate contribui orice alta solutie partiala generata anterior

Diferenţe programare dinamică – divide et impera

Divide et impera

- abordare top-down – problema este descompusă în subprobleme care sunt rezolvate independent
- putem rezolva aceeaşi problemă de mai multe ori (dezavantaj potenţial foarte mare)

Programare dinamică

- abordare bottom-up - se porneşte de la sub-soluţii elementare şi se combină sub-soluţiile mai simple în sub-soluţii mai complicate, pe baza criteriului de optim
- se evită calculul repetat al aceleiaşi subprobleme prin memorarea rezultatelor intermediare

Arbori optimi la căutare

- Def 2.1: Fie K o multime de chei. Un arbore binar cu cheile K este un graf orientat si aciclic A=(V,E) a.i.:
  - Fiecare nod contine o singura cheie $k(u) \in K$ iar cheile din noduri sunt distincte
  - Exista $!r \in V$ a.i. i-grad(r)=0 si $\forall u \neq r$ i-grad(u)=1
  - $\forall u \in V$ e-grad(u)≤2; S(u)= succesorul stanga si D(u)=succesorul dreapta

Arbori optimi la căutare

- Def 2.1: Fie K o mulțime de chei. Un arbore binar cu cheile K este un graf orientat si aciclic A = (V,E) a.i.:
  - Fiecare nod $u \in V$ conține o singură cheie $k(u) \in K$ iar cheile din noduri sunt distincte.
  - Există un nod unic $r \in V$ a.i. i-grad(r) = 0 si $\forall u \neq r$, i-grad(u) = 1.
  - $\forall u \in V$, e-grad(u) ≤ 2; S(u) / D(u) = subarbore stânga / dreapta.

- Def 2.2: Fie K o mulțime de chei peste care exista o relație de ordine $\prec$. Un arbore binar de căutare satisface:
  - $\forall u,v,w \in V$ avem ($v \in S(u) \Rightarrow$ cheie(v) $\prec$ cheie(u)) $\wedge$ ($w \in D(u) \Rightarrow$ cheie(u) $\prec$ cheie(w))

Căutare

```
Caută(elem, Arb)
    dacă Arb = null
        Întoarce null
    dacă elem = Arb.val // valoarea din nodul crt.
        Întoarce Arb
    dacă elem   Arb.val
        Întoarce Caută(elem, Arb.st)
    Întoarce Caută(elem, Arb.dr)
```

$\prec$

Complexitate: Θ(logn)

Inserţie în arbore de căutare

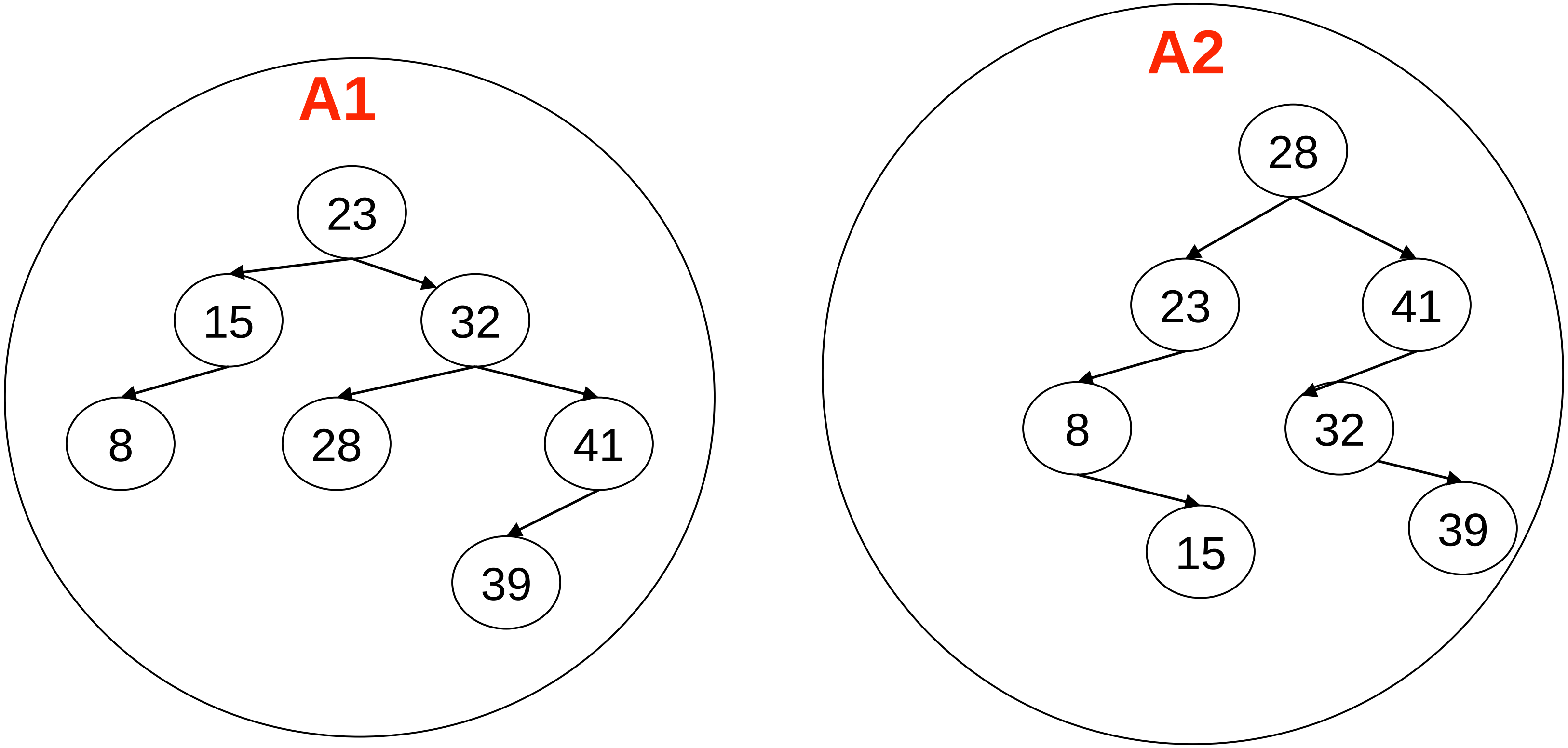
```
Inserare(elem, Arb)
    daca Arb=vid
        nod_nou(elem, null, null)
    daca elem=Arb.val
        Întoarce Arb
    daca elem<Arb.val
        Întoarce nod_nou(Arb.val, Inserare(elem, Arb.st), Arb.dr)
        Întoarce nod_nou(Arb.val, Arb.st, Inserare(elem, Arb.dr))
```

nod Stanga

nod Dreapta

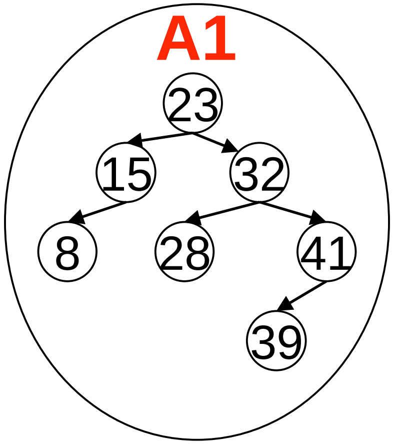
Exemplu de arbori de căutare

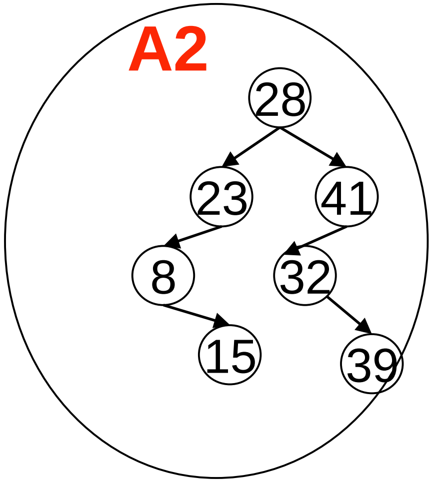
- Cu aceleaşi chei se pot construi arbori distincţi



Exemplu (I)

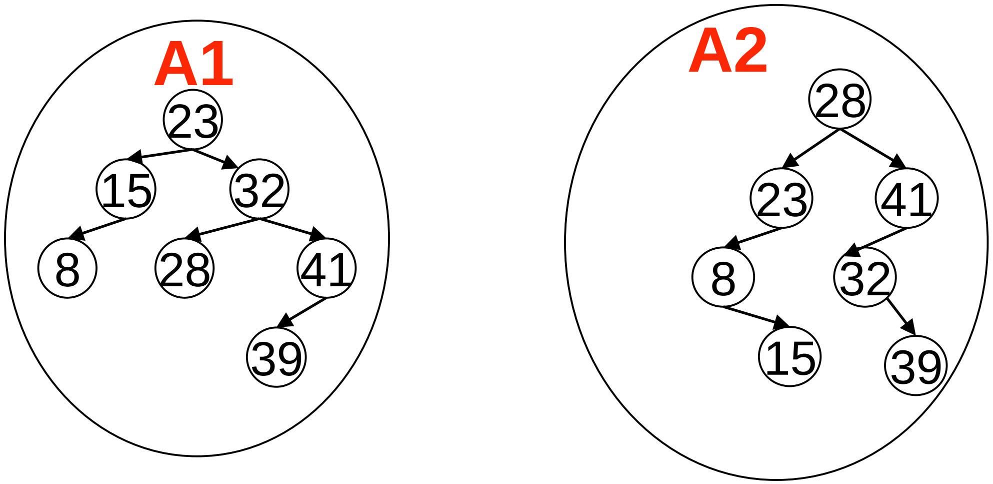
- presupunem cheile din A1 şi A2 au probabilităţi de căutare egale
  - numărul de comparaţii pentru A1 va fi (1+2+2+3+3+3+4)/7=2.42
  - numărul mediu de comparaţii pentru A2 va fi (1+2+2+3+3+4+4)/7=2.71





Exemplu (II)

- presupunem că elementele au următoarele probabilităţi:
  - 8:0.2; 15:0.01; 39:0.1; 23:0.02; 28:0.25; 32:0.2; 41:0.22;
  - numărul mediu de comparaţii pentru A1:
    - 0.02*1+0.01*2+0.2*2+0.2*3+0.25*3+0.22*3+0.1*4=2.85
  - numărul mediu de comparaţii pentru A2:
    - 0.25*1+0.02*2+0.22*2+0.2*3+0.2*3+0.01*4+0.1*4=2.37



Probleme

costul căutării depinde de frecvenţa cu care este căutat fiecare termen

=>ne dorim ca termenii cei mai des căutaţi să fie cât mai aproape de vârful arborelui pentru a micşora numărul de apeluri recursive

dacă arborele este construit prin sosirea aleatorie a cheilor putem avea o simplă listă cu n elemente

Definiţie AOC

- **Definitie:** Fie A un arbore binar de cautare cu chei intr-o multime K, fie $\{x_1, x_2, \ldots x_n\}$ cheile continute in A, iar $\{y_0, y_1, \ldots y_n\}$ chei reprezentante ale cheilor din K care nu sunt in A astfel incat:

$$y_{i-1} \prec x_i \prec y_i, i = \overline{1,n}$$

- Fie p_i, $i = 1,n$ probabilitatea de a cauta cheia x_i si q_j, $j = 0,n$ probabilitatea de a cauta o cheie reprezentata de y_j. Vom avea relatia:

$$\sum_{i=1}^{n} p_i + \sum_{j=0}^{n} q_j = 1$$

- A- arbore de căutare probabilistică cu costul:

$$Cost(A) = \sum_{i=1}^{n} (nivel(x_i, A) + 1) * p_i + \sum_{j=0}^{n} nivel(y_j, A) * q_j$$

- **Definitie:** Un arbore de cautare probabilistica avand cost minim este un ***arbore optim la cautare*** *(AOC)*.

Algoritm AOC naiv

- generarea permutărilor $x_1,\ldots,x_n$
- construcţia arborilor de căutare corespunzători
- calcularea costului pentru fiecare arbore
- complexitate: $\Theta(n!)$ (deoarece sunt n! permutări)
- =>căutăm altă variantă

Construcţia AOC – Notaţii

- Ai,j desemneaza un AOC cu cheile {xi+1, xi+2, … xj} in noduri si cu cheile {yi, yi+1, … yj} in frunzele fictive
- Ci,j = Cost (Ai,j)
- Ri,j este indicele α al cheii xα din radacina arborelui Ai,j
- $$w_{i,j} = \sum_{k=i+1}^{j} p_k + \sum_{k=i}^{j} q_k$$
- **<u>Observatie</u>**: A0,n este chiar arborele A, C0,n = Cost (A) iar w0,n = 1

Construcţia AOC - Demonstraţie

- **<u>Lemă:</u>** Pentru orice $0 \leq i \leq j \leq n$ există relaţiile:
  - Ci,j = 0 , daca i = j
  - $C_{i,j} = \min_{i \leq \alpha \leq j} \{C_{i,\alpha-1} + C_{\alpha,j}\} + w_{i,j}$

- Ci,j depinde de indicele α al nodului rădăcină

- dacă Ci,α-1şi Cα,j sunt minime (costurile unor AOC) => Ci,j este minim

Construcţia AOC

- 1. In etapa d, d = 1, 2, … n se calculeaza costurile si indicele cheilor din radacina arborilor AOC Ai,i+d, I = 0, n-d cu d noduri si d+1 frunze fictive
- Arborele Ai,i+d contine in noduri cheile {xi+1, xi+2, … xi+d}, iar in frunzele fictive sunt cheile {yi, yi+1, … yi+d}. Calculul este efectuat pe baza rezultatelor obtinute in etapele anterioare
- Conform lemei avem

$$C_{i,i+d} = \min_{i \le \alpha \le i+d} \{C_{i,\alpha-1} + C_{\alpha,i+d}\} + w_{i,i+d}$$

- radacina Ai,i+d are indicele Ri,j = α care minimizeaza Ci,i+d.
- 2. Pentru d = n, C0,n corespunde arborelui AOC A0,n cu cheile {x1, x2, … xn} in noduri si cheile {y0, y1, … yn} in frunzele fictive

Algoritm AOC

```
AOC(x, p, q, n){
    // initializare costuri AOC vid Ai,i
    for( i = 0; i ≤ n ; i++)
          {Ci,i = 0, Ri,i = 0, wii = qi}
    for( d = 1; d ≤ n ; d++){
          for( i = 0; i ≤ n-d ; i++){
                    // calcul indice radacina si cost pentru Ai,i+d
                    j = i + d,          Ci,j = ∞,          wi,j = wi,j-1 + pj + qj
                    // ciclul critic – operatii intensive
                    for (α = i + 1;  α ≤ j; α++)
                              if (Ci,α-1  + Cα,j < Ci,j)
                                        { Ci,j  = Ci,α-1  + Cα,j ; Ri,j = α}
                                        Ci,j  = Ci,j  + wi,j
                                        }
// constructie efectiva arbore A0,n cunoscand indicii
    return gen_AOC(C, R, x, 0, n)
}
```

AOC – Corectitudine (I)

- **Teorema:** Algoritmul AOC construieste un arbore AOC A cu cheile x = {x1, x2, … xn} conform probabilitatilor de cautare pi, i = 1,n si qj, j = 0,n

- **Demonstratie:** prin inductie dupa etapa de calcul al costurilor arborilor cu d noduri

- Caz de baza: d = 0. Costurile Ci,i ale arborilor vizi Ai,i, i = 0,n sunt 0, asa cum sunt initializate de algoritm

AOC – Corectitudine (II)

- <u>Pas de inductie:</u> d ≥ 1.
- Ip. ind. pentru orice d’ < d, algoritmul AOC calculeaza costurile Ci,i+d’ si indicii Ri,i+d’, ai radacinilor unor AOC Ai,i+d’, i = 0,n-d’ cu cheile {xi+1, xi+2, … xi+d’}. Trebuie sa aratam ca valorile Ci,i+d si Ri,i+d corespund unor AOC Ai,i+d, i = 0,n-d cu cheile {xi+1, xi+2, … xi+d}.
- Pentru d si i fixate, algoritmul calculeaza unde costurile Ci,α-1 si Cα,i+d corespund unor arbori cu un numar de noduri d’= α – 1 – i in cazul Ci,α-1 si d’= 1 + d – α in cazul Cα,i+d.
- 0 ≤ d’ ≤ d – 1 → aceste valori au fost deja calculate in etapele d’ < d si conform ipotezei inductive → sunt costuri si indici ai radacinilor unor AOC.
- Conform Lemei anterioare, Ci,j este costul unui AOC. Conform algoritmului → radacina acestui arbore are indicele r = Ri,j, iar cheile sunt {xi+1, xi+2, … xr-1} {xr} {xr+1, xr+2, … xj} = {xi+1, xi+2, … xj}.
- Pentru d = n, costul C0,n corespunde unui AOC A0,n cu cheile x si cu radacina de indice R0,n.

Exemplu de Programare Dinamică: Înmulțirea unui șir de matrice (Chain Matrix Multiplication)

- Se dă un şir de matrice: A_1, A_2, ..., A_n.

- Care este numărul minim de înmulţiri de scalari pentru a calcula produsul:

$$A_1 \times A_2 \times \dots \times A_n \ ?$$

- Să se determine una dintre parantezările care minimizează numărul de înmulţiri de scalari.

Înmulţirea matricilor

- A(p, q) x B (q, r) => pqr înmulţiri de scalari.
- Dar înmulţirea matricilor este asociativă (deşi nu este comutativă).
- A(p, q) x B (q, r) x C(r, s)

  (AB)C => pqr + prs înmulţiri

  A(BC) => qrs + pqs înmulţiri
- Ex: p = 5, q = 4, r = 6, s = 2

  (AB)C => 180 înmulţiri

  A(BC) => 88 înmulţiri
- Concluzie: Parantezarea este foarte importantă!

Soluţia banală

- Matrici: $A_1, A_2, \ldots, A_n$.
- Vector de dimensiuni: $p_0, p_1, p_2, \ldots, p_n$.
- $A_i(p_{i-1}, p_i) \rightarrow A_1(p_0, p_1), A_2(p_1, p_2), \ldots$
- Dacă folosim căutare exhaustivă şi vrem să construim toate parantezările posibile pentru a determina minimul: $\Omega(4^n / n^{3/2})$.
- Vrem o soluţie polinomială folosind P.D.

Descompunere în subprobleme

- Încercăm să definim subprobleme identice cu problema originală, dar de dimensiune mai mică.

- $\forall\ 1 \leq i \leq j \leq n$:
  - Notăm $A_{i,\,j} = A_i \times \ldots \times A_j$. $A_{i,j}$ are p_{i-1} linii si p_j coloane: $A_{i,j}(p_{i-1}, p_j)$
  - m[i, j] = numărul optim de înmulţiri pentru a rezolva subproblema $A_{i,j}$
  - s[i, j] = poziţia primei paranteze pentru subproblema $A_{i,j}$
  - Care e parantezarea optimă pentru $A_{i,\,j}$?

- Problema iniţială: $A_{1,n}$

Combinarea subproblemelor

- Pentru a rezolva $A_{i,j}$
  - trebuie găsit acel indice $i \le k < j$ care asigură parantezarea optimă:

$$A_{i,\,j} = (A_i \text{ x...x } A_k) \text{ x } (A_{k+1} \text{ x...x } A_j)$$

$$A_{i,\,j} = A_{i,\,k} \text{ x } A_{k+1,\,j}$$

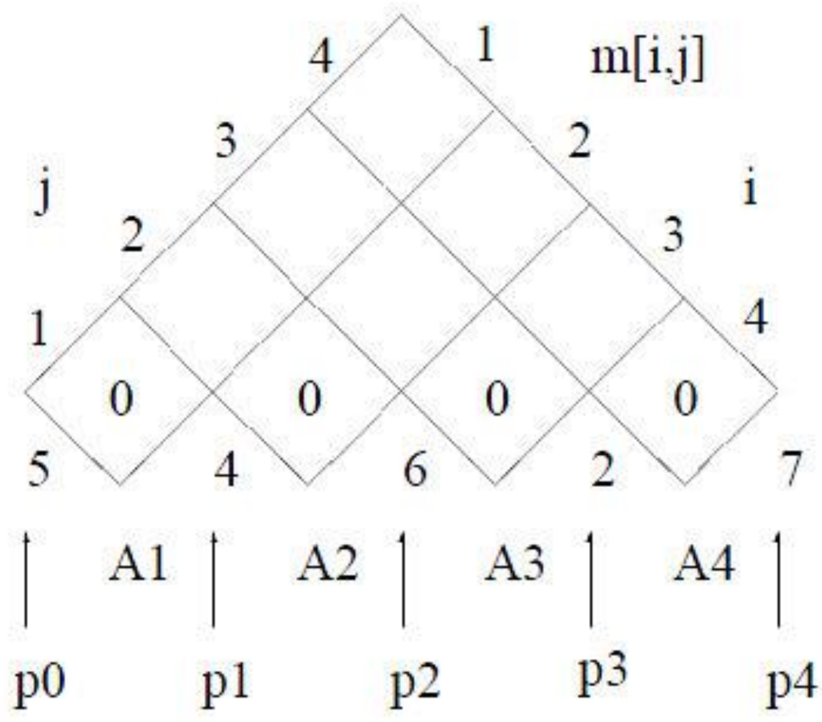
Alegerea optimală

- Căutăm optimul dintre toate variantele posibile de alegere ($i \leq k < j$)

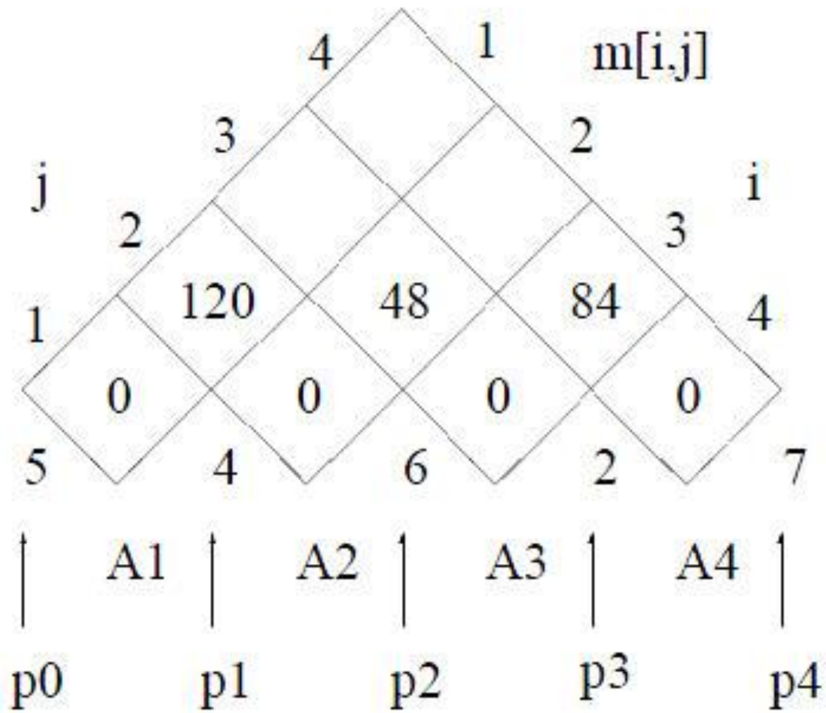
- Pentru aceasta, trebuie însă ca şi subproblemele folosite să aibă soluţie optimală

  (adică $A_{i, k}$ şi $A_{k+1, j}$)

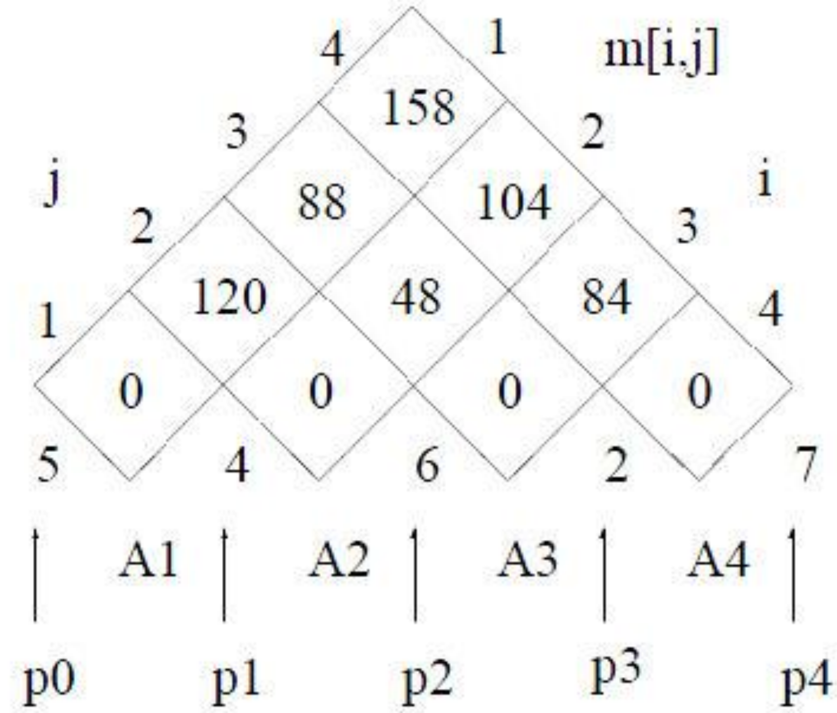
Rezolvare - iniţializare



Rezolvare – pas intermediar



Rezolvare – final



Pseudocod

```
Matrix-Chain(p, n)
{   for (i = 1 to n) m[i, i] = 0;
    for (l = 2 to n)
    {
        for (i = 1 to n - l + 1)
        {
            j = i + l - 1;
            m[i, j] = ∞;
            for (k = i to j - 1)
            {
                q = m[i, k] + m[k + 1, j] + p[i - 1] * p[k] * p[j];
                if (q < m[i, j])
                {
                    m[i, j] = q;
                    s[i, j] = k;
                }
            }
        }
    }
    return m and s;
}
```

Complexitate

- Spaţială: $\Theta(n^2)$
  - Pentru memorarea soluţiilor subproblemelor

- Temporală: $O(n^3)$
  - Ns: Număr total de subprobleme: $O(n^2)$
  - Na: Număr total de alegeri la fiecare pas: $O(n)$
  - Complexitatea este de obicei egala cu Ns x Na

Concluzii

- Caracteristici ale P.D.
  - Substructura optimală
  - Suprapunerea problemelor
- Substructura optimală
  - O alegere (un criteriu de alegere)
  - Una sau mai multe subprobleme ce rezultă din alegerea facută
  - Considerând că la pasul curent construim o soluţie optimală pentru problemă, trebuie sa arătăm că şi soluţiile subproblemelor folosite pentru rezolvarea sa sunt la rândul lor optimale
- Este foarte important spaţiul ales pentru reprezentarea subproblemelor:
  - De ce nu am folosit pentru AOC un spaţiu A1,j ?
  - Incercand sa determinam r optim intre 1 si j, am obtine doua subprobleme A1,r şi Ar+1,j, care nu pot fi reprezentate în acest spaţiu => trebuie sa permitem ca ambii indici sa varieze Ai,j

Concluzii (II)

- Câte subprobleme sunt folosite în soluţia optimală ?
  - AOC: 2 subprobleme
- Câte variante de ales avem de făcut pentru determinarea alegerii optimale ?
  - AOC: j-i+1 candidaţi pentru rădăcină
- Informal, complexitatea = #total subprobleme x #alegeri
  - AOC:
    - O primă aproximație: $O(n^2) * O(n) = O(n^3)$
    - Se poate arăta însă că este $O(n^2)$

Concluzii (III)

- Observatie! Nu toate problemele de optimizare posedă substructură optimală!
  - Ex: drumul cel mai lung in grafuri
- Suprapunerea problemelor
  - *Memoizare*
  - Se foloseşte un tablou pentru salvarea soluţiilor subproblemelor pentru a nu le recalcula (în special când folosim varianta recursivă a P.D.)
  - De obicei, construim soluţiile direct *bottom-up*, de la subprobleme la probleme

Bibliografie

- http://www.cs.umass.edu/~barring/cs611/lecture/4.pdf
- http://thor.info.uaic.ro/~dlucanu/cursuri/tpaa/resurse/Curs6.pps
- http://www.math.fau.edu/locke/Greedy.htm
- http://en.wikipedia.org/wiki/Greedoid
- Cormen – Introducere în Algoritmi cap. 15,16
- Giumale C. – Introducere în Analiza Algoritmilor - Algoritm de construcţie AOC + Demonstraţie