Proiectarea Algoritmilor

Curs 2 – Scheme de algoritmi – Greedy continuare + Programare dinamică

Bibliografie

- Cormen – Introducere în Algoritmi cap. Algoritmi Greedy (17) și Programare dinamică (16)
- Giumale – Introducere în Analiza Algoritmilor cap 4.4 ,4.5
- 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
- http://www.cse.ust.hk/~dekai/271/notes/L12/L12.pdf

Greedy - reminder

- Metodă de rezolvare eficientă a unor probleme de optimizare.
- Soluția trebuie să satisfacă un criteriu de optim global (greu de verificat) → optim local mai ușor de verificat.
- Se aleg soluții parțiale ce sunt îmbunătățite repetat pe baza criteriului de optim local până ce se obțin soluții finale.
- Soluțiile parțiale ce nu pot fi îmbunătățite sunt abandonate → proces de rezolvare irevocabil (fără reveniri)!

Greedy – schemă generală de rezolvare

- Schema generală de rezolvare a unei probleme folosind Greedy (programarea lacomă):

- Rezolvare_lacomă(Crit_optim, Problemă)
  - 1. sol_parțiale = sol_inițiale(Problemă); // determinarea soluțiilor parțiale
  - 2. sol_fin = Φ;
  - 3. **Cât timp** (sol_parțiale ≠ Φ)
  - 4. **Pentru fiecare** (s **în** sol_parțiale)
  - 5. **Dacă** (s este o soluție a problemei) { // dacă e soluție
  - 6. sol_fin = sol_fin U {s}; // finală se salvează
  - 7. sol_parțiale = sol_parțiale \ {s};
  - 8. } **Altfel** // se poate optimiza?
  - 9. **Dacă** (optimizare_posibilă (s, Crit_optim, Problemă))
  - 10. sol_parțiale = sol_parțiale \ {s} U optimizare(s,Crit_optim,Problemă) // da
  - 11. **Altfel** sol_parțiale = sol_parțiale \ {s}; // nu
  - 12. **Întoarce** sol_fin;

Comparație D&I și Greedy

- Tip abordare
  - D&I: top-down;
  - Greedy: top-down.
- Criteriu de optim
  - D&I: nu;
  - Greedy: da.

Arbori Huffman – Definiții (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 există mai mult de o cheie într-o frunză;
  - Toate nodurile interne au exact 2 copii;
  - Arcele sunt codificate cu 0 și 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 în 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 – Definiții (II)

- A – arborele de codificare a cheilor.
- lg_cod(k) – lungimea codului cheii k conform A.
- nivel(k,A) – nivelul pe care apare în 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 frecvență p este un arbore Huffman, iar codurile cheilor sunt coduri Huffman.

Arbori Huffman – algoritm de construcție (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 numită Arb.
- 2. Se aleg doi subarbori a şi b din Arb astfel încât a şi b au pondere minimă.

Arbori Huffman – algoritm de construcție (II)

- 3. Se construiește un arbore binar cu o rădăcina *r* care nu conține nici o cheie și cu descendenții a și b. Ponderea arborelui este definită ca w(r) = w(a) + w(b).
- 4. Arborii a și b sunt eliminați din Arb iar r este inserat în Arb.
- 5. Se repetă procesul de construcție descris de pașii 2-4 până când mulțimea Arb conține un singur arbore – Arborele Huffman pentru cheile K.

Arbori Huffman - pseudocod

- Huffman(K,p){
  - 1. Arb = {k $\in$ K | frunză(k, p(k))};
  - 2. **Cât timp** (card (Arb) > 1) // am mai mulți subarbori
  - 3. Fie a_1 și a_2 arbori din Arb a.i. $\forall a \in$ Arb $a \neq a_1$ si $a \neq a_2$, avem $w(a_1) \leq w(a)$ și $w(a_2) \leq w(a)$); // practic se extrage // de două ori minimul și se salvează în a_1 și a_2
  - 4. Arb = Arb \ {a_1, a_2} U nod_intern(a_1, a_2, $w(a_1) + w(a_2)$));
  - 5. **Dacă** (Arb = Φ)
  - 6. **Întoarce** arb_vid;
  - 6. **Altfel**
  - 7. fie A singurul arbore din mulțimea Arb;
  - 8. **Întoarce** A;

- Notații folosite:
  - a = frunză (k, p(k)) – subarbore cu un singur nod care conține cheia k, iar w(a) = p(k);
  - a = nod_intern(a_1, a_2, x) – subarbore format dintr-un nod intern cu descendenții a_1 și a_2 și w(a) = x.

Demonstrație (I)

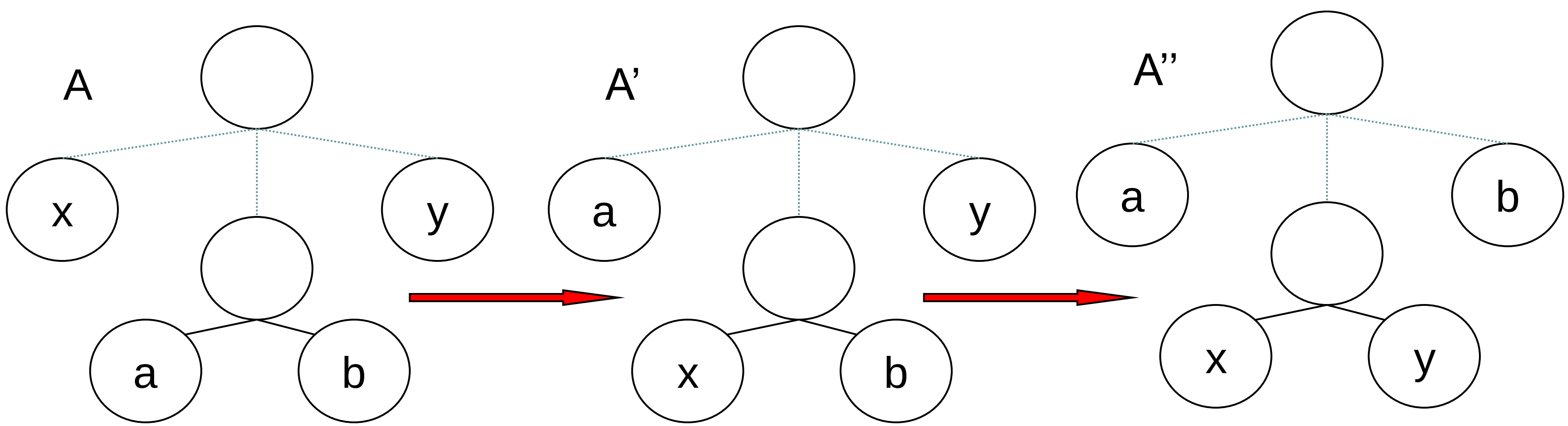
- Arborele de codificare construit trebuie să aibă cost minim pentru a fi arbore Huffman.

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

Demonstrație (II)

- Demonstrație 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

Demonstrație (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 înlocuirea subarborelui cu rădăcina *a* și frunzele x, y cu subarborele cu un singur nod care conţine frunza z, este un arbore Huffman cu cheile K'.

- Demonstrație:
  - 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} => construiește arborele Huffman(K, p).

Alt exemplu (I)

Problema rucsacului

Trebuie să umplem un rucsac de capacitate maximă 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) – "problema 0-1"

Alt exemplu (II)

- Varianta 1: Algoritm Greedy
  - sortăm obiectele după raportul v_i/m_i;
  - adăugăm fracţiuni din obiectul cu cea mai mare valoare per kg până epuizăm stocul şi apoi adăugăm fracţiuni din obiectul cu valoarea următoare.
  - Exemplu: M = 10; m_1 = 5 kg, v_1 = 10, m_2 = 8 kg, v_2 = 19, m_3 = 4 kg, v_3 = 4
  - Soluție: ($m_{2,}$, v_2) 8kg şi 2kg din (m_1,v_1) – valoarea totală: 19 + 2 * 10 / 5 = 23

- Varianta 2: Algoritmul Greedy nu funcţionează => contraexemplu
  - Exemplu: M = 10; m_1 = 5 kg, v_1 = 10, m_2 = 8 kg, v_2 = 19, m_3 = 4 kg, v_3 = 4
  - Rezultat corect – 2 obiecte (m_1,v_1) – valoarea totală: 20
  - Rezultat algoritm Greedy – 1 obiect (m_2,v_2) – valoarea totală: 19

Când funcţionează algoritmii Greedy? (I)

- Problema are proprietatea de substructură optimă
  - Soluţia problemei conţine soluţiile subproblemelor.
- Problema are proprietatea alegerii locale
  - Alegând soluţia optimă local se ajunge la soluţia optimă global.

Când funcţionează algoritmii Greedy? (II)

- Fie E o mulţime finită nevidă şi $I \subset P(E)$ a.i. $\emptyset \in I$, $\forall X \subseteq Y$ şi $Y \subseteq I \Rightarrow X \subseteq I$. Atunci spunem că (E,I) este un sistem accesibil.
- Submulţimile din I sunt numite submulţimi "independente".
- Exemple:
  - Ex1: $E = \{e_1, e_2, e_3\}$ si $I = \{\emptyset, \{e_1\}, \{e_2\}, \{e_3\}, \{e_1, e_2\}, \{e_2, e_3\}\}$ – 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.

Când funcţionează algoritmii Greedy? (III)

- Un sistem accesibil este un matroid dacă satisface proprietatea de interschimbare:

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

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

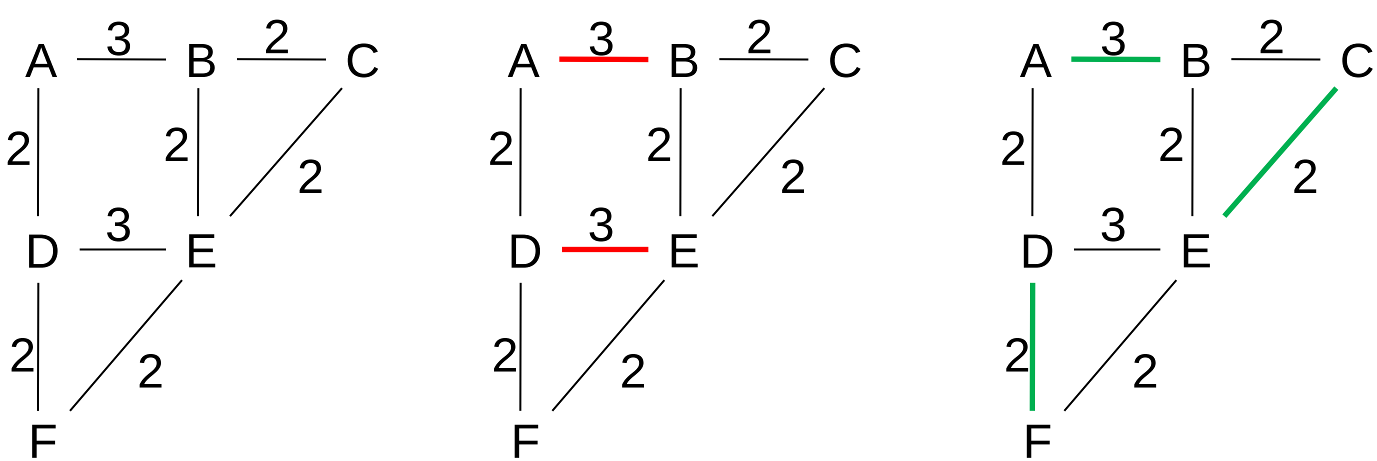
Verificăm exemplele

- Ex1: $I = \{\varnothing, \{e_1\}, \{e_2\}, \{e_3\}, \{e_1, e_2\}, \{e_2, e_3\}\}$

Fie $Y = \{\{e_1\}, \{e_2\}, \{e_3\}, \{e_1, e_2\}\}$ și $X = \{\{e_1\}, \{e_3\}\}$

$\rightarrow Y \setminus X = \{\{e_2\}, \{e_1, e_2\}\} \rightarrow X \cup \{e_2\} \subseteq I \rightarrow$ matroid

- Ex4:



Problema rucsacului discretă (varianta 2)

- Exemplu: $M = 10$; $m_1 = 5$ kg, $v_1 = 10$, $m_2 = 8$ kg, $v_2 = 19$, $m_3 = 4$ kg, $v_3 = 4$

- Fie I = mulțimea subsoluțiilor și $X = \{m_2\}$ și $Y = \{m_1, m_1\}$ → $m_1 \in Y \setminus X$ dar $X \cup \{m_1\} \not\subseteq I$ → problema nu respectă proprietatea de matroid → problema nu se poate rezolva folosind tehnica Greedy!

Greedy – tema de gândire

- Se dă un număr natural n. Să se găsească cel mai mare subset S din {1, 2, ..., n} astfel încât nici un element din S să nu fie divizibil cu nici un alt element din S.
- În plus, dacă există mai multe subseturi maximale ce respectă proprietatea de mai sus, să se determine întotdeauna cel mai mic din punct de vedere lexicografic.
- S este lexicografic mai mic decât T dacă cel mai mic element care este membru din S sau T, dar nu din amândouă, face parte din S.

Programare dinamică

- Programare dinamică
  - Descriere generală
  - Algoritm generic
  - Caracteristici
- Exemplificare: Înmulțirea matricilor
- Exemplificare: Arbori optimi la căutare (AOC)
  - Definiții
  - Construcția AOC

Programare dinamică

- Descriere generală
  - Soluții optime construite iterativ asamblând soluții optime ale unor probleme similare de dimensiuni mai mici.

- Algoritmi “clasici”
  - Înmulţirea unui şir de matrici
  - AOC
  - Algoritmul Floyd-Warshall care determină drumurile de cost minim dintre toate perechile de noduri ale unui graf.
  - Numere catalane
  - Viterbi

Algoritm generic

- Programare dinamică (crit_optim, problema)
  - // fie problema$_0$ problema$_1$ … problema$_n$ astfel încât
  - // problema$_n$ = problema; problema$_i$ mai simplă decât problema$_{i+1}$
  - 1. Sol = soluții_inițiale(crit_optim, problema$_0$);
  - 2. **Pentru** *i* **de la** 1 **la** *n* // construcție soluții pentru
    // problema$_i$ folosind soluțiile problemelor precedente
  - 3. Sol$_i$ = calcul_soluții(Sol, Crit_optim, Problema$_i$);
    // determin soluția problemei$_i$
  - 4. Sol = Sol U Sol$_i$;
    // noile soluții se adaugă pentru a fi refolosite pe viitor
  - 5. s = soluție_pentru_problema$_n$(Sol);
    // selecție / construcție soluție finală
  - 6. **Întoarce** s;

Caracteristici

- O soluție optimă a unei probleme conține soluții optime ale subproblemelor.
- Decompozabilitatea recursivă a problemei P în subprobleme similare $P = P_n, P_{n-1}, \ldots P_0$ care acceptă soluții din ce în ce mai simple.
- Suprapunerea problemelor (soluția unei probleme P_i participă în procesul de construcție a soluțiilor mai multor probleme P_k de talie mai mare $k > i$) – memoizare (se foloseşte un tablou pentru salvarea soluţiilor subproblemelor cu scopul de a nu le recalcula).
- În general se folosește o abordare bottom-up, de la subprobleme la probleme.

Diferențe Greedy – Programare dinamică

Programare lacomă

- Sunt menținute doar soluțiile parțiale curente din care evoluează soluțiile parțiale următoare
- Soluțiile parțiale anterioare sunt eliminate
- Se poate obține o soluție neoptimă. (trebuie demonstrat că se poate aplica).

Programare dinamică

- La construcția unei soluții noi poate contribui orice altă soluție parțială generată anterior
- Se păstrează toate soluțiile parțiale
- Se obține soluția optimă.

Diferenţe divide et impera – programare dinamică

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 (memoizare)

Exemplu: Parantezarea matricilor (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:

  $$A_1 \times A_2 \times \ldots \times A_n \ ?$$

- 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, ..., A_n.
- Vector de dimensiuni: p_0, p_1, p_2, ... , 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.

- $\forall\ 1 \le i \le j \le n$:
  - Notăm $A_{i,\,j} = A_i \times \ldots \times A_j$. $A_{i,j}$ are p_{i-1} linii și 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 $i \leq k < j$ care asigură parantezarea optimă:

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

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

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 $A_{k+1,j}$ să aibă soluție optimă).

Substructura optimală

- Dacă ştim că alegerea optimală a soluţiei pentru problema $A_{i,j}$ implică folosirea subproblemelor ($A_{i,k}$ şi $A_{k+1,j}$) şi soluţia pentru $A_{i,j}$ este optimală, atunci şi soluţiile subproblemelor $A_{i,k}$ şi $A_{k+1,j}$ trebuie să fie optimale!

- Demonstraţie: Folosind metoda cut-and-paste (metodă standard de demonstrare a substructurii optimale pentru problemele de programare dinamică).

- Observație: Nu toate problemele de optim posedă această proprietate! Ex: drumul maxim dintr-un graf orientat.

Definirea recursivă

- Folosind descompunerea în subprobleme, combinarea subproblemelor, alegerea optimală şi substructura optimală putem să rezolvăm problema prin programare dinamică.
- Următorul pas este să definim recursiv soluţia unei subprobleme.
- Vrem să găsim o formulă recursivă pentru m[i, j] şi s[i, j].

Definirea recursivă (II)

$$m[i,j] = \begin{cases} 0 & i = j, \\ \min_{i \le k < j} (m[i,k] + m[k+1,j] + p_{i-1} p_k p_j) & i < j \end{cases}$$

- Cazurile de bază sunt m[i, i]
- Noi vrem să calculăm m[1, n]
- Cum alegem s[i, j] ?
- Bottom-up de la cele mai mici subprobleme la cea iniţială.

Rezolvare bottom-up

$$m[1,2], m[2,3], m[3,4], \ldots, m[n-3, n-2], m[n-2, n-1], m[n-1, n]$$

$$m[1,3], m[2,4], m[3,5], \ldots, m[n-3, n-1], m[n-2, n]$$

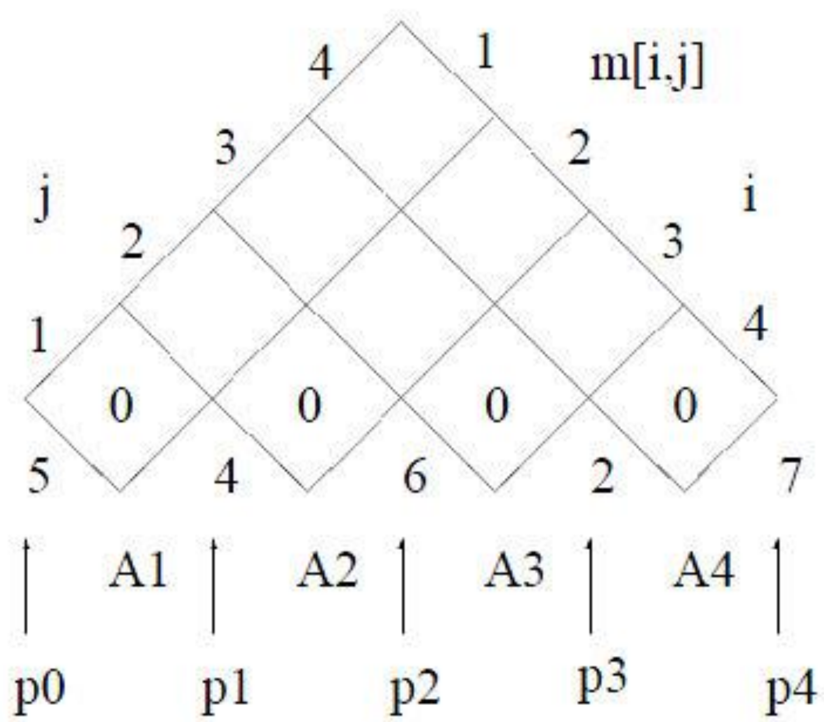
$$m[1,4], m[2,5], m[3,6], \ldots, m[n-3, n]$$

$$\vdots$$

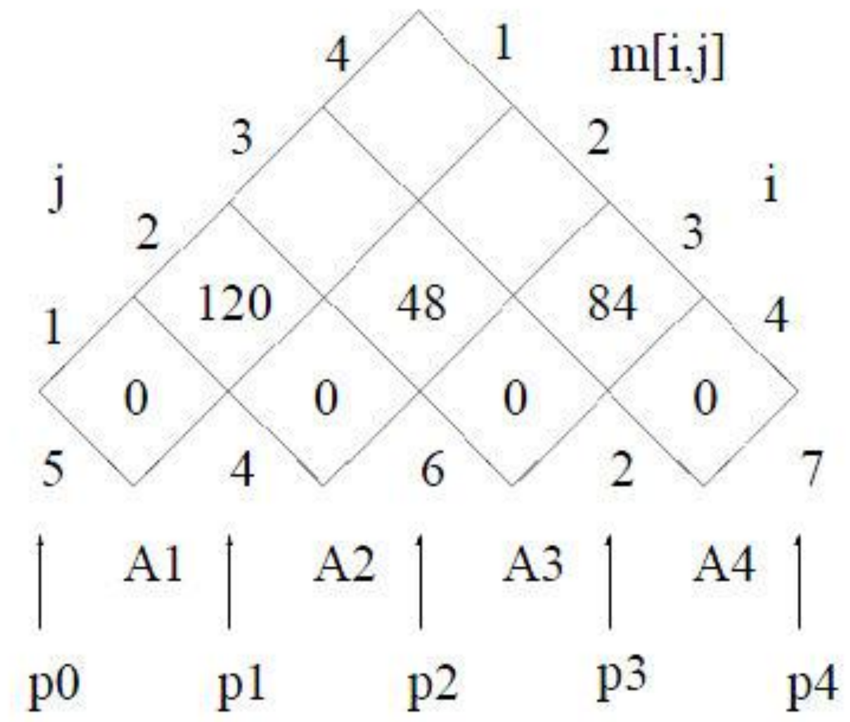
$$m[1, n-1], m[2, n]$$

$$m[1, n]$$

Rezolvare - iniţializare



Rezolvare – pas intermediar (I)



Rezolvare – pas intermediar (II)

$$m[i,j] = \begin{cases} 0 & i = j, \\ \min_{i \le k < j} \left(m[i,k] + m[k+1,j] + p_{i-1} p_k p_j \right) & i < j \end{cases}$$

$A_{1,3} = A_{1,2} * A_3 = A_1 * A_{2,3}$

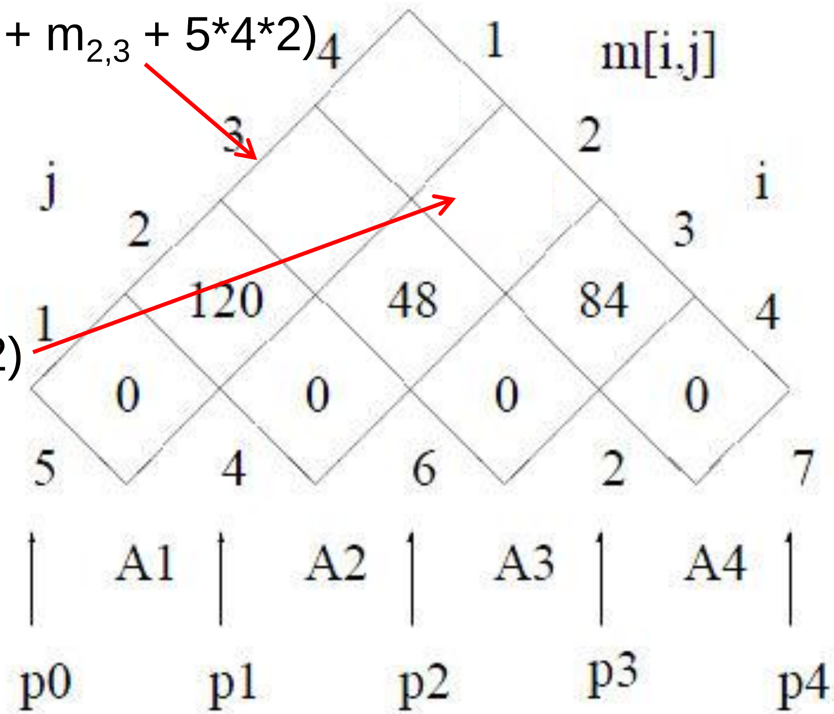
$m_{1,3} = \min(m_{1,2} + m_{3,3} + 5*6*2,\ m_{1,1} + m_{2,3} + 5*4*2)$

$s_{1,3} = 2$

$A_{2,4} = A_{2,3} * A_4 = A_2 * A_{3,4}$

$m_{2,4} = \min(m_{2,3} + 4*2*7,\ m_{3,4} + 4*6*2)$

$s_{2,4} = 2$



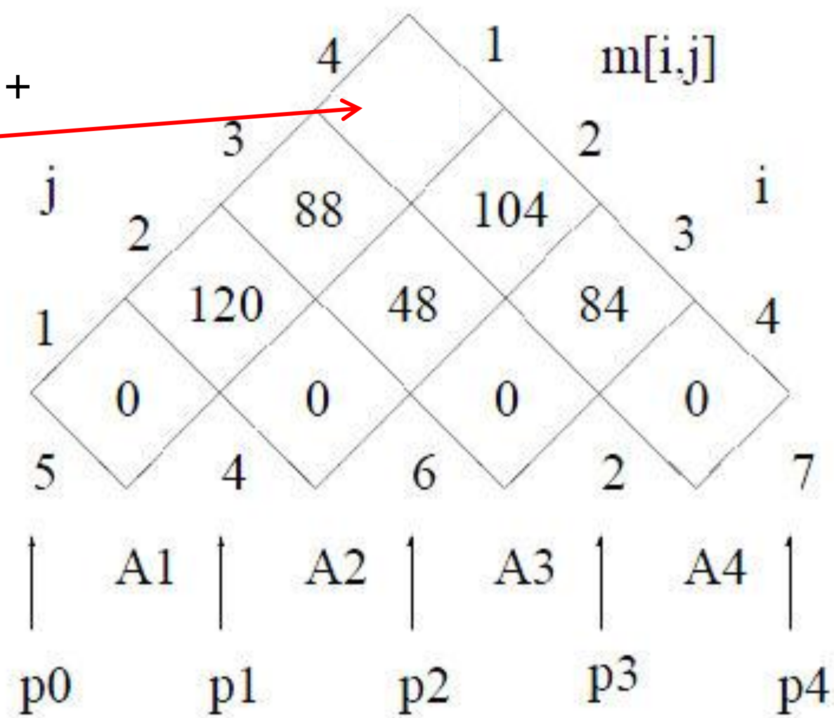
Rezolvare – final

$$m[i,j] = \begin{cases} 0 & i = j, \\ \min_{i \le k < j} (m[i,k] + m[k+1,j] + p_{i-1} p_k p_j) & i < j \end{cases}$$

$A_{1,4} = A_1 * A_{2,4} = A_{1,2} * A_{3,4} = A_{1,3} * A_4$

$m_{1,3} = \min(m_{2,4} + 5*4*7,\ m_{1,2} + m_{3,4} + 5*6*7,\ m_{1,3} + 5*2*7\ ) = 158$

$s_{1,4} = 1$



Parantezarea optimă este:

$(A_1(A_2A_3))A_4$

Numărul optim de operații

este: 158

Pseudocod

- Înmulțire_matrici (p, n)
  - **Pentru** *i* **de la** 1 **la** *n* // inițializare
    - m[i, i] = 0
  - **Pentru** *l* **de la** 2 **la** *n* // dimensiune problema
    - **Pentru** *i* **de la 1 la** *n – l + 1* // indice stânga
      - j = i + l – 1 // indice dreapta
      - m[i, j] = ∞ // pentru determinare minim
      - **Pentru** *k* **de la** *i* **la** *j – 1*
        - q = m[i, k] + m[k + 1, j] + p[i - 1] * p[k] * p[j]
        - **Dacă** q < m[i, j]
        - m[i, j] = q
        - s[i, j] = k
  - **Întoarce** *m* și *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: $O(n^3)$ este de obicei egală cu Ns x Na

Alt exemplu: Arbori optimi la căutare (AOC)

- Def 2.1: Fie K o mulțime de chei. Un arbore binar cu cheile K este un graf orientat și aciclic A = (V,E) a.î.:
  - 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.î. i-grad(r) = 0 și $\forall$u $\neq$ r, i-grad(u) = 1.
  - $\forall$u $\in$ V, e-grad(u) $\leq$ 2; S(u) / D(u) = succesorul stânga / dreapta.

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

Căutare într-un arbore de căutare

- Caută(elem, Arb)
  - **Dacă** Arb = null
    - **Întoarce** null
  - **Dacă** elem = Arb.val // valoarea din nodul crt.
    - **Întoarce** Arb
  - **Dacă** elem $\prec$ Arb.val
    - **Întoarce** Caută(elem, Arb.st)
  - **Întoarce** Caută(elem, Arb.dr)

Complexitate: $\Theta(\log n)$

Inserţie într-un arbore de căutare

- Inserare(elem, Arb)
  - **Dacă** Arb = vid // adaug cheia în arbore
    - nod_nou(elem, null, null)
  - **Dacă** elem = Arb.val // valoarea există deja
    - **Întoarce** Arb
  - **Dacă** elem $\prec$Arb.val
    - **Întoarce** Inserare(elem, Arb.st) // adaugă în stânga
    - **Întoarce** Inserare(elem, Arb.dr) // sau în dreapta

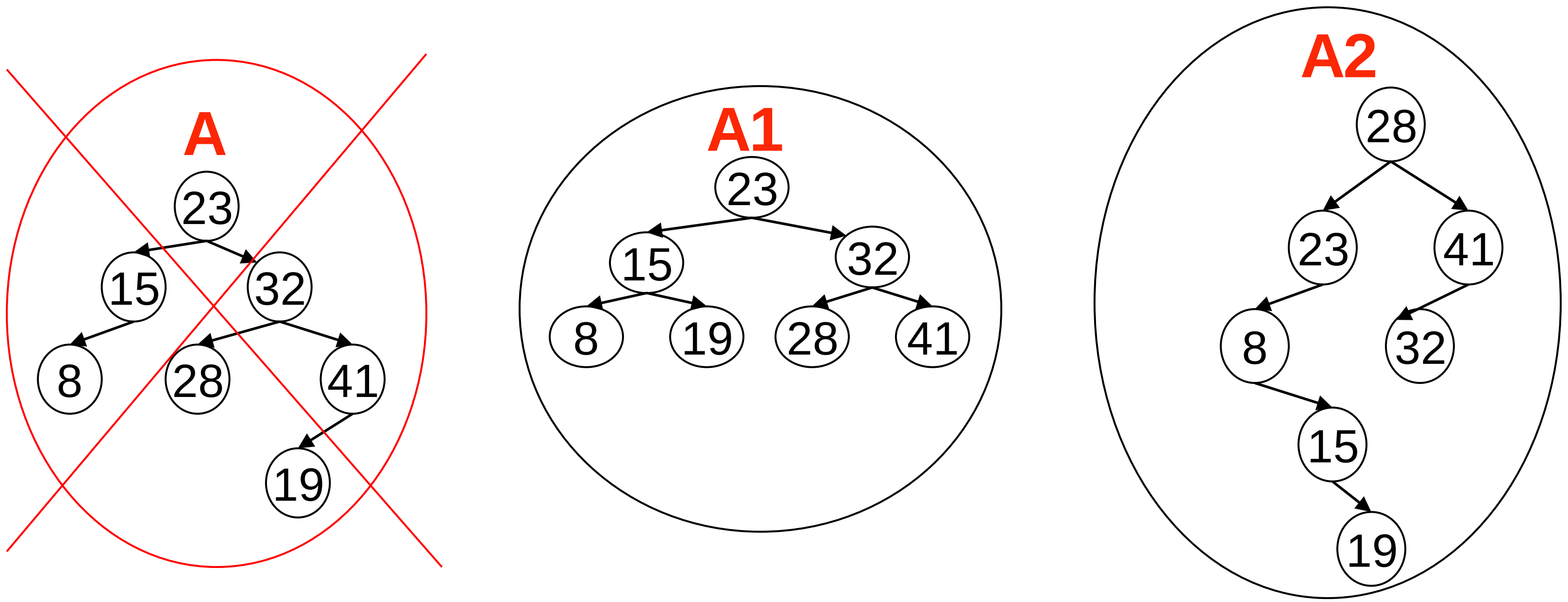
nod Stânga

nod Dreapta

Complexitate: Θ(logn)

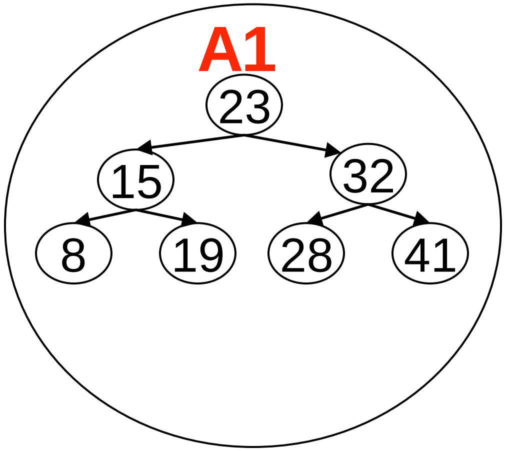
Exemplu de arbori de căutare

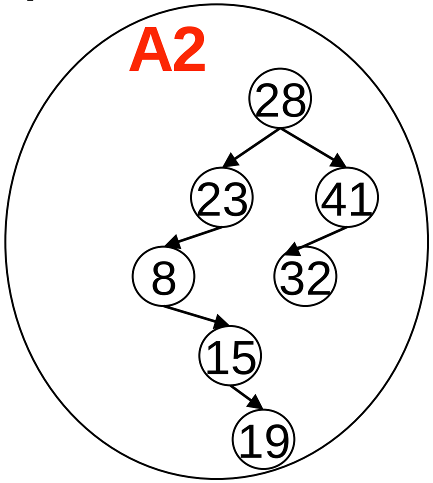
- Cu aceleaşi chei se pot construi arbori distincţi



Exemplu (I)

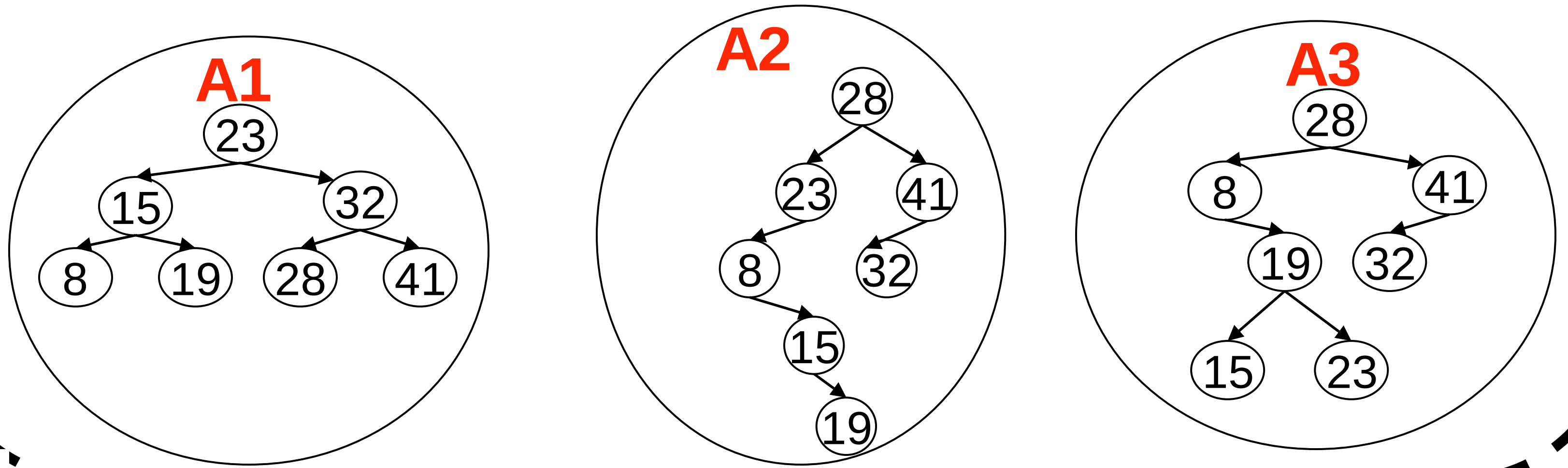
- Presupunem că elementele din A1 şi A2 au probabilităţi de căutare egale:
  - Numărul mediu de comparaţii pentru A1 va fi: (1 + 2 + 2 + 3 + 3 + 3 + 3) / 7 = 2.42
  - Numărul mediu de comparaţii pentru A2 va fi: (1 + 2 + 2 + 3 + 3 + 4 + 5) / 7 = 2.85





Exemplu (II)

- Presupunem că elementele au următoarele probabilităţi:
  - 8: 0.2; 15: 0.01; 19: 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+0.1*4+0.25*3+0.22*3=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*5=2.47



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 ajunge la o simplă listă cu n elemente.

Definiţie AOC

- Definiție: Fie A un arbore binar de căutare cu chei într-o mulțime K; fie $\{x_1, x_2, \dots x_n\}$ cheile conținute în A, iar $\{y_0, y_1, \dots y_n\}$ chei reprezentante ale cheilor din K ce nu sunt în A astfel încât: $y_{i-1} \prec x_i \prec y_i, i = \overline{1,n}$. Fie p_i, i = 1,n probabilitatea de a căuta cheia x_i și q_j, j = 0,n probabilitatea de a căuta o cheie reprezentată de y_j. Vom avea relația: $\sum_{i=1}^{n} p_i + \sum_{j=0}^{n} q_j = 1$. Se numește arbore de căutare probabilistică, un arbore cu costul:

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

- Definiție: Un arbore de căutare probabilistică având cost minim este un arbore optim la căutare (AOC).

Algoritm AOC naiv

- Generarea permutărilor $x_1, \ldots x_n$.
- Construcţia arborilor de căutare corespunzători.
- Calcularea costului pentru fiecare arbore.
- Alegerea arborelui de cost minim.
- Complexitate: $\Theta(n!)$ (deoarece sunt n! permutări).
- → căutăm altă variantă!!!

Construcţia AOC – Notaţii

- $A_{i,j}$ desemnează un AOC cu cheile $\{x_{i+1}, x_{i+2}, \dots x_j\}$ în noduri și cu cheile $\{y_i, y_{i+1}, \dots y_j\}$ în frunzele fictive.

- $C_{i,j}$ = Cost ($A_{i,j}$). $Cost(A_{ij}) = \sum_{k=i+1}^{j}(nivel(x_k, A_{ij})+1) * p_k + \sum_{k=i}^{j} nivel(y_k, A_{ij}) * q_k$

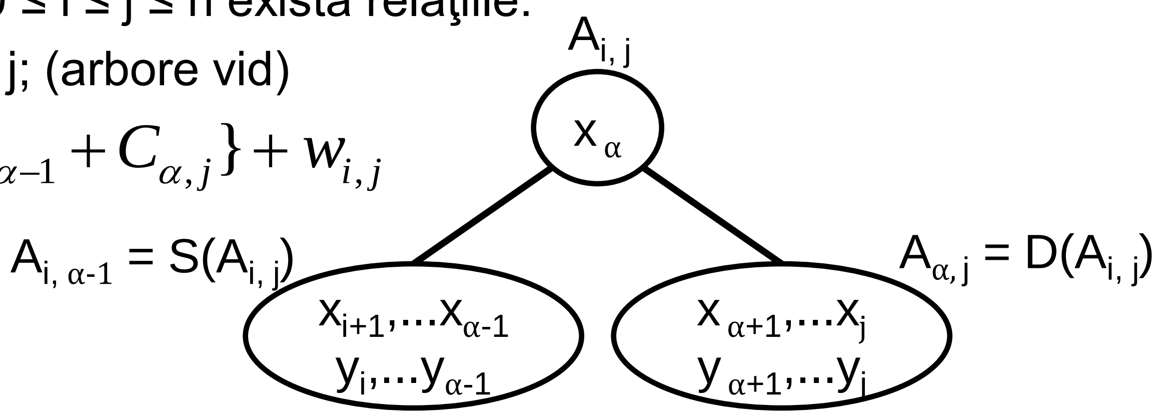
- $R_{i,j}$ este indicele α al cheii x_α din rădăcina arborelui $A_{i,j}$.

$$w_{i,j} = \sum_{k=i+1}^{j} p_k + \sum_{k=i}^{j} q_k \quad w_{i,j} = \sum_{k=i+1}^{j-1} p_k + p_j + \sum_{k=i}^{j-1} q_k + q_j = w_{i,j-1} + p_j + q_j$$

- Observație: $A_{0,n}$ este chiar arborele A, $C_{0,n}$ = Cost (A) iar $w_{0,n} = 1$.

Construcţia AOC - Demonstraţie

- Lemă: Pentru orice $0 \le i \le j \le n$ există relaţiile:
  - $C_{i,j} = 0$, dacă $i = j$; (arbore vid)
  - $$C_{i,j} = \min_{i<\alpha\le j}\{C_{i,\alpha-1} + C_{\alpha,j}\} + w_{i,j}$$



- Demonstraţie:

$$Cost(A_{ij}) = \sum_{k=i+1}^{j}(nivel(x_k, A_{ij}) + 1) * p_k + \sum_{k=i}^{j} nivel(y_k, A_{ij}) * q_k$$

→ $$C_{i,j} = C_{i,\alpha-1} + C_{\alpha,j} + w_{i,j}$$

- $C_{i,j}$ depinde de indicele α al nodului rădăcină.
- dacă $C_{i,\alpha-1}$ şi $C_{\alpha,j}$ sunt minime (costurile unor AOC) → $C_{i,j}$ este minim.

Construcţia AOC

- 1. În etapa d, d = 1, 2, … n se calculează costurile și indicele cheilor din rădăcina arborilor AOC $A_{i,\ i+d}$, i = 0, n-d cu d noduri și d + 1 frunze fictive.
- Arborele $A_{i,\ i+d}$ conține în noduri cheile $\{x_{i+1}, x_{i+2}, \dots x_{i+d}\}$, iar în frunzele fictive sunt cheile $\{y_i, y_{i+1}, \dots y_{i+d}\}$. Calculul este efectuat pe baza rezultatelor obținute în etapele anterioare.
- Conform lemei avem $$C_{i,i+d} = \min_{i<\alpha\le i+d}\{C_{i,\alpha-1} + C_{\alpha,i+d}\} + w_{i,i+d}$$
- Rădăcina $A_{i,\ i+d}$ are indicele $R_{i,\ j} = \alpha$ care minimizează $C_{i,\ i+d}$.
- 2. Pentru d = n, $C_{0,\ n}$ corespunde arborelui AOC $A_{0,\ n}$ cu cheile $\{x_1, x_2, \dots x_n\}$ în noduri si cheile $\{y_0, y_1, \dots y_n\}$ în frunzele fictive.

Algoritm AOC

- AOC(x, p, q, n)
- **Pentru** *i* **de la** 0 **la** *n*
- $\{C_{i,i} = 0, R_{i,i} = 0, w_{i,i} = q_i\}$ // inițializare costuri AOC vid $A_{i,i}$
- **Pentru** *d* **de la** 1 **la** *n*
- **Pentru** *i* **de la** 0 **la** *n-d* // calcul indice rădăcină și cost pentru $A_{i,i+d}$
- $j = i + d$, $C_{i,j} = \infty$, $w_{i,j} = w_{i,j-1} + p_j + q_j$
- **Pentru** *α* **de la** *i* + *1* **la** *j* // ciclul critic – operații intensive
- **Dacă** $(C_{i,\alpha-1} + C_{\alpha,j} < C_{i,j})$ // cost mai mic?
- $\{C_{i,j} = C_{i,\alpha-1} + C_{\alpha,j}$; $R_{i,j} = \alpha\}$ // update
- $C_{i,j} = C_{i,j} + w_{i,j}$ // update
- **Întoarce** gen_AOC(C, R, x, 0, n) // construcție efectivă arbore $A_{0,n}$
- // cunoscând indicii

Complexitate???

ÎNTREBĂRI?