Tehnici de dezvoltare a algoritmilor (III)

Dynamic Programming

Programarea dinamica

- Spatiu *vs* Timp
- Ia ce e mai bun din tehnicile
  - Greedy
    - eficienta, dar nu si optimalitate
  - “exhaustive” (backtracking)
    - optimalitate, dar nu eficienta
- Intuitie
  - cautare sistematica a tuturor posibilitatilor
  - stocare a rezultatelor pt evitarea recalcularilor
    - se stocheaza “consecintele” tuturor deciziilor posibile
- Nota: *Cere exercitiu mult pentru a fi aplicata cu usurinta*

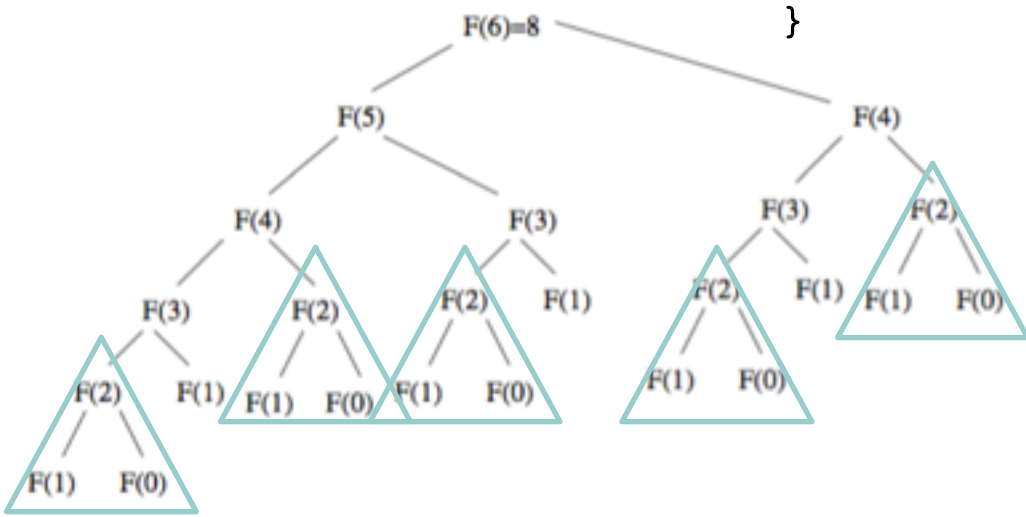
Programarea dinamica

- **<u>Definitie</u>**: Tehnica eficienta de implementare a algoritmilor recursivi, prin stocarea rezultatelor partiale
- Cand/Cum?
  - algoritmul recursiv reface aceleasi calcule de mai multe ori
  - stocare rezultate in tabel
  - trebuie sa avem o recurenta (recursivitate) corecta
- Potrivita in cazurile in care obiectele combinatoriale manifesta o ordonare stanga-dreapta a componentelor
  - siruri de caractere, arbori, poligoane, secvente de intregi

Sirul lui Fibonacci - varianta recursiva

- $F_n = F_{n-1} + F_{n-1}$, $F_0 = 1$, $F_1 = 1$
- $F_n = \{1,1,2,3,5,8,13,21,34,55,\ldots\}$
- Istoria sirului
- Pseudocod

```
long fib_r(int n)
{
  if (n == 0) return(0);
  if (n == 1) return(1);
  return(fib_r(n-1) + fib_r(n-2));
}
```



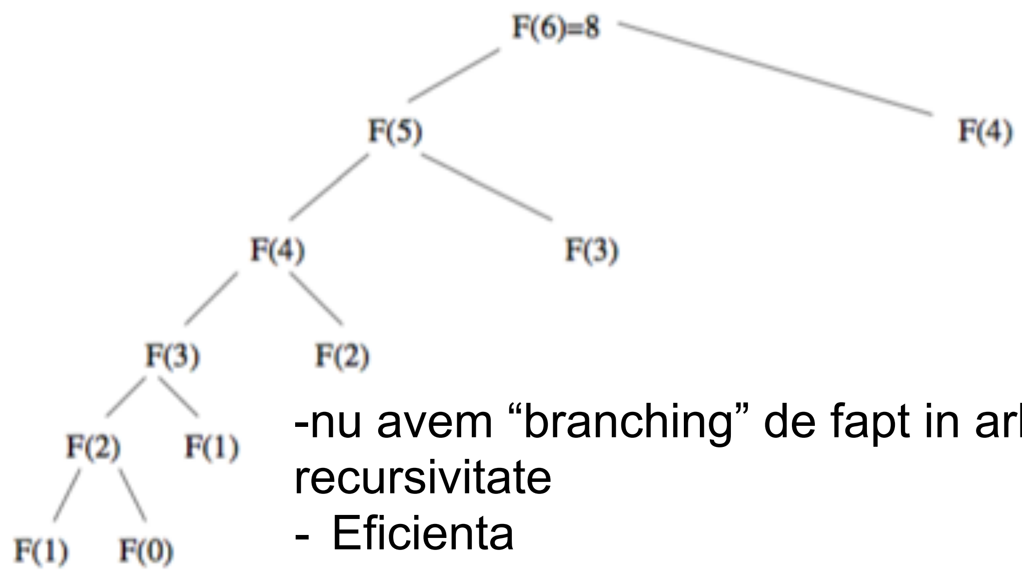
$$F_{n-1} / F_n \approx \phi = \frac{(1+\sqrt{5})}{2} \approx 1.61803$$

Sirul lui Fibonacci - PD cu “memoization” (caching, memorare)

```
#define MAXN     45   /* largest interesting n */
#define UNKNOWN -1   /* contents denote an empty cell */
long f[MAXN+1];      /* array for caching computed fib values */

long fib_c_driver(int n)
{
  int i; /* counter */

  f[0] = 0;
  f[1] = 1;
  for (i=2; i<=n; i++)
    f[i] = UNKNOWN;
  return(fib_c(n));
}
```



-nu avem “branching” de fapt in arborele de recursivitate

- Eficienta
  - timp?
  - memorie

```
long fib_c(int n)
{

  if (f[n] == UNKNOWN)
          f[n] = fib_c(n-1) + fib_c(n-2);
  return(f[n]);
}
```

!!! Memorarea are sens cand spatiul parametrilor diferiti are dimensiune rezonabila! (e.g. nu e utila la quicksort, DFS..)

Sirul lui Fibonacci - PD bottom up

```
long fib_dp(int n)
{
  int i;  /* counter */
  long f[MAXN+1]; /* array to cache computed
  fib values */

  f[0] = 0;
  f[1] = 1;
  for (i=2; i<=n; i++)
    f[i] = f[i-1]+f[i-2];
  return(f[n]);
}
```

- Inversand ordinea de evaluare, nu mai avem deloc apeluri recursive
- Dar tot memorie *O(n)*

Sirul lui Fibonacci - PD bottom-up “forgetting”

```
long fib_ultimate(int n)
{
  int i;                    /* counter */
  long back2=1, back1=1;    /* last two values of f[n] */
  long next;                /* placeholder for sum */
  if (n == 0) return (0);
  for (i=2; i<n; i++) {
        next = back1+back2;
        back2 = back1;
        back1 = next;
  }
  return(back1+back2);
}
```

- Obs: avem nevoie doar de ultimele doua valori calculate
- Memorie *O(1)*
- *Ordinea de evaluare*

Programare dinamica

- Algoritmii *Divide-and-conquer* partitioneaza pb. in sub-probleme ***independente***, pe care le rezolva recursiv si apoi combina solutiile pentru a genera solutia pentru problema originala
- Programarea dinamica:
  - Aplicabila cand sub-problemele nu sunt independente
  - Fiecare *<u>subproblema este rezolvata doar</u> o* data, rezultatul este stocat intr-o tabela pentru a evita re-calcularea
    - Consecinta: nr. relativ redus de sub-probleme pentru ca tabela sa fie calculabila
  - Permite transformarea unui algoritm exponential intr-unul polinomial
  - ***Numele 'Programare Dinamica' se refera la calcularea tabelei; trebuie gasita totusi si forma ei!***

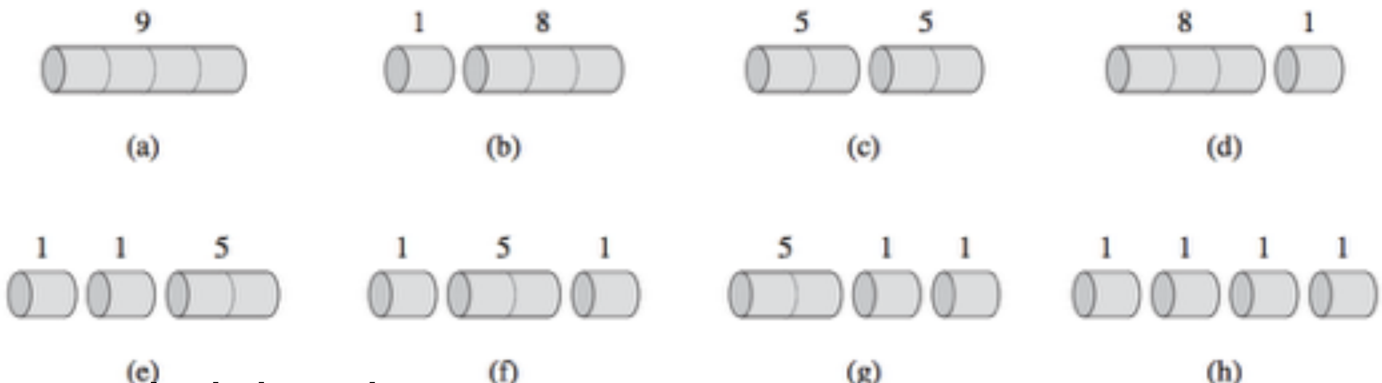
Programare Dinamica in Probleme de Optimizare

- Problemele de optimizare:
  - mai multe solutii posibile
  - fiecare solutie are o “calitate”; sarcina este gasirea solutiei de cea mai buna calitate - optima - minimizare/maximizare a unei valori ce cuantifica calitatea
- Dezvoltarea unui algoritm DP - 4 pasi:
  1. Caracterizam structura solutiei optime
  2. Definim recursiv valoarea solutiei optime
  3. Calculam valoarea solutiei optime intr-o maniera de jos in sus (*bottom-up)*
  4. Construim solutia optima

Taierea tijei (Rod cutting)

- Pt. fiecare lungime i se cunoaste $p_{i,}$ o taiere nu costa nimic
- Se cere determinarea venitului maxim obtinut prin taierea tijei

length i	1	2	3	4	5	6	7	8	9	10
price p_i	1	5	8	9	10	17	17	20	24	30



Structura solutiei optime:

$n = i_1 + i_2 + \ldots + i_k$

$r_n = p_{i1} + p_{i2} + \ldots + p_{ik}$

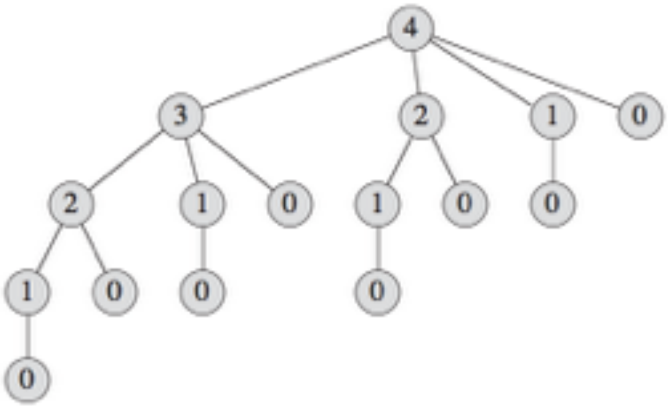
2^{n-1} *solutii*

$\boldsymbol{r_n = \max(p_n, r_1 + r_{n-1}, r_2 + r_{n-2}, \ldots, r_{n-1} + r_1)}$

Taierea Tijei (Rod cutting)

- Problema tijei manifesta *sub-structura optima*
  - *Avem de rezolvat probleme de acelasi tip, dar de dimensiune mai mica; partitiile pot fi considerate independente*
  - ***Solutia optima incorporeaza solutii optime ale sub-probl. ce se pot rezolva independent***
- Descompunere alternativa:
  - *$n = i_k + n-i_k$ - doar a doua parte se mai poate imparti; prima bucata si descompunerea restului*
  - $\boldsymbol{r_n = \max(p_i + r_{n-i}), i= 1 \text{ to } n}$
  - *Solutia optima incorporeaza doar solutie* ***unei*** *sub-probleme, nu a doua !*

Taierea Tijei



```
CUT-ROD(p, n)
if n == 0
    return 0
q = −∞
for i = 1 to n
    q = max(q, p[i] + CUT-ROD(p, n − i))
return q
```

Versiunea PD cu "memoizare" top-down

```
MEMOIZED-CUT-ROD-AUX(p, n, r)
if r[n] ≥ 0
    return r[n]
if n == 0
    q = 0
else q = −∞
    for i = 1 to n
        q = max(q, p[i] + MEMOIZED-CUT-ROD-AUX(p, n − i, r))
r[n] = q
return q
```

```
MEMOIZED-CUT-ROD(p, n)
let r[0..n] be a new array
for i = 0 to n
    r[i] = −∞
return MEMOIZED-CUT-ROD-AUX(p, n, r)
```

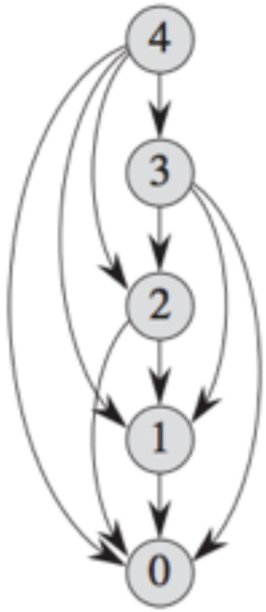
Taierea Tijei - versiunea PD bottom-up

```
BOTTOM-UP-CUT-ROD(p, n)
let r[0..n] be a new array
r[0] = 0
for j = 1 to n
    q = -∞
    for i = 1 to j
        q = max(q, p[i] + r[j - i])
    r[j] = q
return r[n]
```

- Ordonarea naturala a problemelor! Rezolvam problemele in ordinea “dimensiunii” lor
- Atat versiunea TD, cat si cea BU, au timp O(n^2)

Taierea Tijei - Graful sub-problemelor

- Multimea de sub-probleme implicate
- Interactiunea dintre ele
- Abordarea BU considera probl. in *ordinea invers topologica* asa cum apar ele in graf
- Abordarea TD cu “memoizare” considera probl. in ordinea data de *parcurgerea in adancime*
- Dimensiunea lui ne poate ajuta la estimarea complexitatii in timp (de regula |V+E|



Taierea Tijei - Reconstructia solutiei

- salvarea *deciziei* care a dus la acea valoare
  - dimensiune primei partitii a tijei, s_j

```
EXTENDED-BOTTOM-UP-CUT-ROD(p, n)
let r[0..n] and s[0..n] be new arrays
r[0] = 0
for j = 1 to n
    q = −∞
    for i = 1 to j
        if q < p[i] + r[j − i]
            q = p[i] + r[j − i]
            s[j] = i
    r[j] = q
return r and s
```

```
PRINT-CUT-ROD-SOLUTION(p, n)
(r, s) = EXTENDED-BOTTOM-UP-CUT-ROD(p, n)
while n > 0
    print s[n]
    n = n − s[n]
```

i	0	1	2	3	4	5	6	7	8	9	10
$r[i]$	0	1	5	8	10	13	17	18	22	25	30
$s[i]$	0	1	2	3	2	2	6	1	2	3	10

LCS - Longest Common Sub-sequence (Sub-secventa comuna de lungime maxima)

- Spell-checking, bio-informatica: secvente de DNA
  - **similaritatea**:
    - S1 = ACCGGTCGAGTGCGCGGAAGCCGGCCGAA
    - S2 = GTCGTTCGGAATGCCGTTGCTCTGTAAA
  - gasim S3 astfel incat bazele apar si in S1, si in S2, in aceeasi ordine, dar nu neaparat consecutiv
    - S3 = GTCGTCGGAAGCCGGCCGAA
- Se defineste ca **sub-secventa** a sirului $X = <x1, x2, \ldots ,xm>$ sirul $Z = <z1, z2, \ldots ,zk>$, unde exista o secventa crescatoare de indici $<i_1, i_2, \ldots, i_k>$ in X a.i. $\forall j = 1,2,\ldots,k$ avem $x_{i_j} = z_j$
- E.g. $Z=<B; C; D; B>$ e sub-secventa al lui $X=<A; B; C; B; D; A; B>$, cu indicii $<2; 3; 5; 7>$
- LCS - se dau 2 siruri, X si Y, si se doreste gasirea sub-secv. comune de lungime maxima

LCS - Longest Common Sub-sequence

(1) Proprietatea de **sub-structura optima**

- Prefixul *i* al unui sir $X = \langle x1, x2, \ldots, xm \rangle$:
  - $Xi = \langle x1, x2, \ldots, xi \rangle$
- **Teorema - Sub-structura optima a LCS**: Fie $X = \langle x1, x2, \ldots, xm \rangle$ si $Y = \langle y1, y2, \ldots, yn \rangle$ siruri, si $Z = \langle z1, z2, \ldots, zk \rangle$ orice LCS a lui X si Y
  - Daca $x_m = y_n$, atunci $z_k = x_m = y_n$, si Z_{k-1} este LCS a lui X_{m-1} si Y_{n-1}
  - Daca $x_m != y_n$, atunci $z_k != x_m$ => Z este LCS a lui X_{m-1} si Y
  - Daca $x_m != y_n$, atunci $z_k != y_n$ => Z este LCS a lui X si Y_{n-1}

LCS - Longest Common Sub-sequence

- **Teorema - demonstratie**
  - (1) daca z_k != x_m, putem adauga x_m=y_n la Z si sa obtinem o secventa comuna de lungime k+1; Z_{k-1} e sir comun de lungime k-1 a lui X_{m-1} si Y_{n-1} (red. absurd)
  - (2) daca z_k != x_m, atunci Z este sir comun al lui X_{m-1} si Y. Daca ar exista W - sir comun al lui X_{m-1} si Y de lungime > k, W ar fi si sir comun al lui X_m si Y (contrad.)
  - (3) simetric cu (2)

LCS - Longest Common Sub-sequence

(2) Definim recursiv valoarea solutiei optime

- avem de examinat 1 sau 2 sub-probleme pt a gasi LCS pt X si Y
  - $x_m=y_n$ - o problema; $x_m!=y_n$ - 2 probleme

$$c[i,j]=\begin{cases}0 & \text{if } i=0 \text{ or } j=0\\ c[i-1,j-1]+1 & \text{if } i,j>0 \text{ and } x_i=y_j\\ \max\{c[i,j-1],c[i-1,j]\} & \text{if } i,j>0 \text{ and } x_i\neq y_j\end{cases}$$

  - Conditiile din problema restrang sub-problemele (cate sub-rpobleme avem in total?)

LCS - Longest Common Sub-sequence

(3) Calculam lungimea LCS

```
LCS-LENGTH(X, Y)
m = X.length
n = Y.length
let b[1..m, 1..n] and c[0..m, 0..n] be new tables
for i = 1 to m
    c[i, 0] = 0
for j = 0 to n
    c[0, j] = 0
for i = 1 to m
    for j = 1 to n
        if x_i == y_j
            c[i, j] = c[i − 1, j − 1] + 1
            b[i, j] = "↖"
        elseif c[i − 1, j] ≥ c[i, j − 1]
            c[i, j] = c[i − 1, j]
            b[i, j] = "↑"
        else c[i, j] = c[i, j − 1]
            b[i, j] = "←"
return c and b
```

O(mn)

	j	0	1	2	3	4	5	6
i		y_j	B	D	C	A	B	A
0	x_i	0	0	0	0	0	0	0
1	A	0	↑0	↑0	↑0	↖1	←1	↖1
2	B	0	↖1	←1	←1	↑1	↖2	←2
3	C	0	↑1	↑1	↖2	←2	↑2	↑2
4	B	0	↖1	↑1	↑2	↑2	↖3	←3
5	D	0	↑1	↖2	↑2	↑2	↑3	↑3
6	A	0	↑1	↑2	↑2	↖3	↑3	↖4
7	B	0	↖1	↑2	↑2	↑3	↖4	↑4

LCS - Longest Common Sub-sequence

- (4) Construirea LCS

```
PRINT-LCS(b, X, i, j)
if i == 0 or j == 0
    return
if b[i, j] == "↖"
    PRINT-LCS(b, X, i − 1, j − 1)
    print x_i
elseif b[i, j] == "↑"
    PRINT-LCS(b, X, i − 1, j)
else PRINT-LCS(b, X, i, j − 1)
```

- O(m+n)
- Nu avem de fapt nevoie de ***b*** pt a calcula LCS
- Putem calcula eficient bazandu-me doar pe ***c***

Arbori Binari de Cautare Optimi

- Utilitate
  - translatoare (e.g. Engleza-Franceza)
  - parte a compilatorului care cauta cuvinte rezervate
- Fapt:
  - cuvintele sunt cautate cu anumite frecvente (*cunoscute*)
- Problema:
  - Cum sa organizam un ABC astfel incat sa minimizam numarul de noduri vizitate in toate cautarile, data fiind frecventa lor de aparitie

Arbori Binari de Cautare Optimi

- **Arbore Binar de Cautare Optim:**
  - secventa $K = <k_1, k_2, \ldots, k_n>$ chei distincte, sortate
  - fiecare cheie k_i are asociata o probabilitate de cautare p_i
  - $n+1$ chei *dummy* - nu apar in arbore: $d_0, d_1, \ldots, d_n$: d_0 repr. $cheile < k_1$, dn repr. $cheile > k_n$
  - fiecare cheie d_i are asociata o probabilitate de cautare q_i
  - nodurile k_i - interne, nodurile d_i - frunze

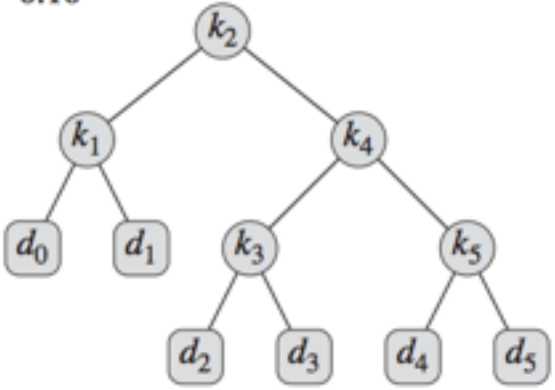
$$\sum_{i=1}^{n} p_i + \sum_{i=0}^{n} q_i = 1$$

Arbori Binari de Cautare Optimi

- Putem estima costul mediu (asteptat) de cautare pe un ABC astfel construit

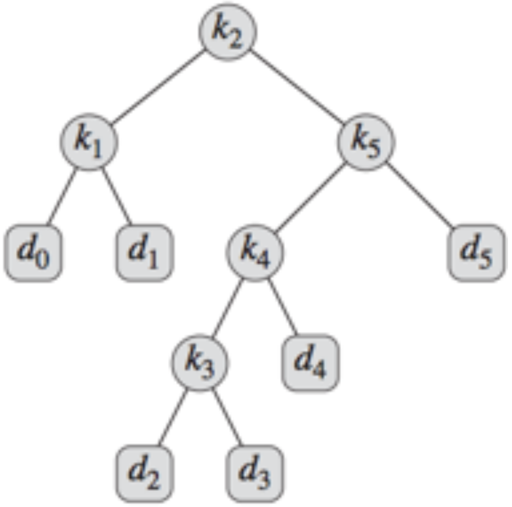
$$\begin{aligned}\mathrm{E}\left[\text{search cost in } T\right] &= \sum_{i=1}^{n}(\operatorname{depth}_T(k_i)+1)\cdot p_i + \sum_{i=0}^{n}(\operatorname{depth}_T(d_i)+1)\cdot q_i \\ &= 1 + \sum_{i=1}^{n}\operatorname{depth}_T(k_i)\cdot p_i + \sum_{i=0}^{n}\operatorname{depth}_T(d_i)\cdot q_i\ ,\end{aligned}$$

i	0	1	2	3	4	5
p_i		0.15	0.10	0.05	0.10	0.20
q_i	0.05	0.10	0.05	0.05	0.05	0.10



E[cost] = 2.8

E[cost] = 2.75



Arbori Binari de Cautare Optimi

- (1) Structura unui ABC Optim
  - orice sub-arbore din ABC trebuie sa contina chei intr-un domeniu continuu $k_i \dots k_j$, pt. $1 \leq i \leq j \leq n$
  - sub-arborele care contine $k_i \dots k_j$ trebuie sa contina si cheile dummy $d_i \dots d_j$
- Sub-structura optima
  - daca un ABC optim T are sub-arborele T' cu cheile $k_i \dots k_j$
  - T' trebuie sa fie si el optim pentru sub-problema cu cheile $k_i \dots k_j$ si cheile dummy $d_i \dots d_j$
- Arborii vizi:
  - daca selectam k_i ca radacina pt. $k_i \dots k_j$, sub-arb stang va contine d_i
  - similar si pt k_j

Arbori Binari de Cautare Optimi

- (2) Definim recursiv valoarea solutiei optime
- Domeniul sub-problemelor: gasirea ABC optim continand cheile $\boldsymbol{k_i \ldots k_j}$, $\boldsymbol{i} \geq 1,\ \boldsymbol{j} \leq \boldsymbol{n}, \boldsymbol{j} \geq \boldsymbol{i}-1$
- Cazuri
  - $\boldsymbol{j} = \boldsymbol{i}-1$ (doar dummy $\boldsymbol{d_{i-1}}$): costul asteptat: $\boldsymbol{e}[\boldsymbol{i,i}-1] = \boldsymbol{q_{i-1}}$
  - $\boldsymbol{j} \geq \boldsymbol{i}$: trebuie sa alegem $\boldsymbol{k_r}$ din $\boldsymbol{k_i \ldots k_j}$ si sa constr. un ABC optim din $\boldsymbol{k_i \ldots k_{r-1}}$ ca sub-arbore stang, respectiv un ABC optim cu $\boldsymbol{k_{r+1} \ldots k_j}$ ca sub-arbore drept
    - cand un ABC optim devine sub-arb. ii creste costul cu suma tuturor probabilitatilor din el (verificati relatia asta!)

$$w(i,j) = \sum_{l=i}^{j} p_l + \sum_{l=i-1}^{j} q_l$$

Arbori Binari de Cautare Optimi

- (2) Definim recursiv valoarea solutiei optime (contd)
- daca k_r este radacina ABC optim construit din cheile $k_i \ldots k_j$, avem:

$$e[i,j] = p_r + (e[i,r-1] + w(i,r-1)) + (e[r+1,j] + w(r+1,j))$$

where

$$w(i,j) = w(i,r-1) + p_r + w(r+1,j)$$

thus

$$e[i,j] = e[i,r-1] + + e[r+1,j] + w(i,j)$$

Recurenta

$$e[i,j] = \begin{cases} q_{i-1} & \text{if } j = i-1 , \\ \min\limits_{i \le r \le j} \{e[i,r-1] + e[r+1,j] + w(i,j)\} & \text{if } i \le j . \end{cases}$$

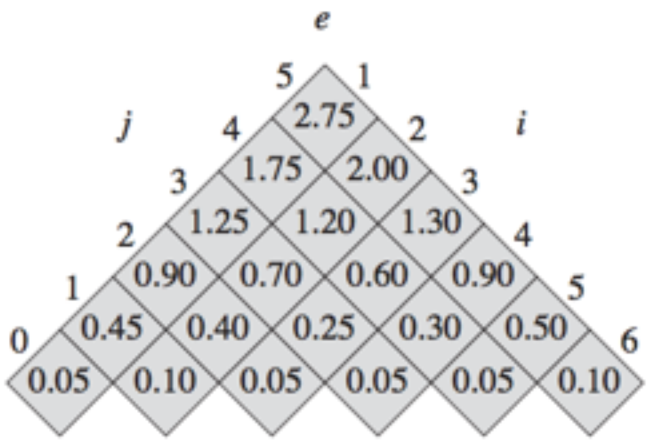
Arbori Binari de Cautare Optimi

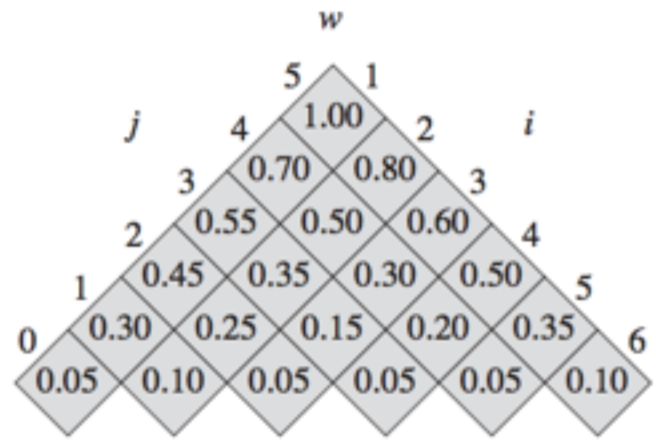
- (3) Calcularea costului ABC optim

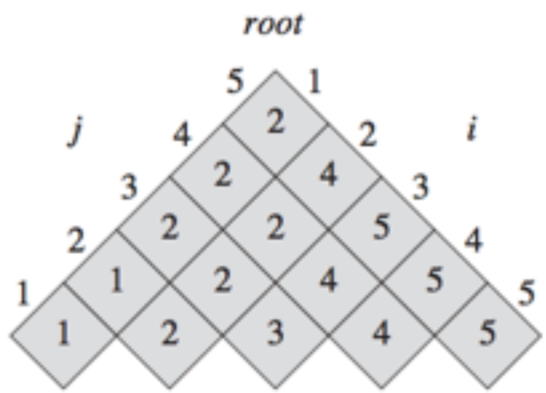
```
OPTIMAL-BST(p, q, n)
let e[1..n+1, 0..n], w[1..n+1, 0..n],
        and root[1..n, 1..n] be new tables
for i = 1 to n + 1
    e[i, i-1] = q_{i-1}
    w[i, i-1] = q_{i-1}
for l = 1 to n
    for i = 1 to n - l + 1
        j = i + l - 1
        e[i, j] = ∞
        w[i, j] = w[i, j-1] + p_j + q_j
        for r = i to j
            t = e[i, r-1] + e[r+1, j] + w[i, j]
            if t < e[i, j]
                e[i, j] = t
                root[i, j] = r
return e and root
```

Arbori Binari de Cautare Optimi

i	0	1	2	3	4	5
p_i		0.15	0.10	0.05	0.10	0.20
q_i	0.05	0.10	0.05	0.05	0.05	0.10







Elementele Programarii Dinamine (recap)

- (1) Demonstram ca avem **substructura optima** – sol. optima contine in ea solutiile optime la sub-probleme
  - Gasirea solutia unei probleme:
    - Efectuarea unei *alegeri* dintr-o serie de variante posibile (trebuie sa investigam care sunt acelea)
    - Rezolvarea *uneia sau mai multor* sub-probleme care sunt rezultatul unei alegeri (caracterizarea spatiului sub-problemelor)
  - Demonstram ca *solutiile* sub-problemelor trebuie sa fie optime *optimal* pentru ca intreaga solutie sa fie optima (reducere la absurd - “cut-and-paste”)

Elementele Programarii Dinamine (recap)

- (2) Scrierea unei *recurente* pt. val. solutiei optime
  - $M_{\text{opt}} = \min_{\text{over all choices } k}$ **{($\sum M_{\text{opt}}$ toate sub-pb, rezultand din sel. *k*) + (costul asociat pentru a face selectia *k*)}**
  - Demonstratie ca numarul de sub-probleme diferite este marginit superior de o functie polinomiala
- (3) Calcularea valorii solutiei optime intr-o maniera *bottom-up*, pt. a avea rezultatele necesare pre-calculate (sau *top-down* cu "*memoizare*")
  - Verificare daca se poate *reduce cerintele de memorie*, prin "uitarea" solutiilor la sub-probleme care nu mai sunt utilizate
- (4) Construirea solutiei optime din informatie pre-calculata (inregistram pe parcurs secventa de alegeri care ne duc la solutia optima)

Greedy vs PD

- PD:
  - Selectia la fiecare pas depinde de solutiile la sub-probleme
  - Versiune TD sau BU - cea BU e de regula mai eficienta
  - multe probleme sunt repetate in rezolvarea problemelor mai mari (e.g. taierea tijei)
    - aceasta repetitie faciliteaza eficientizarea la varianta BU
- Greedy
  - Face intai selectia, apoi se concentreaza pe sub-pb
  - Cea mai buna alegere nu depinde de solutiile la sub-pb
    - doar de cele mai bune selectii de pana atunci
- PD+Greedy: sub-structura [optima]
  - solutia problemei contine in ea solutiile [optime] ale sub-problemelor

Greedy vs PD

- **PD**

//T - table of values of best sol. of problems of sizes Smallest to P

```
for i in smallest subproblem to P loop
  T(i) := MAX of:
    T(j) + cost of choice that changes subproblem j into problem i
    T(k) + cost of choice that changes subproblem k into problem i
     ... as many subproblems as needed
end loop
Result is T(P)
```

- **Greedy**

```
tempP = P // tempP is the remaining subproblem
   while tempP not empty loop
       in subproblem tempP, decide greedy choice C
       Add value of C to solution
       tempP := subproblem tempP reduced based on choice C
   end loop
```

Bibliografie

- S. Skiena: The Algorithm Design Manual, cap 8.
  - Coeficienti binomiali
  - Edit distance (generalizare la LCS)
  - Cateva *war stories* interesante
- Th. Cormen et al.: Introduction to Algorithms, cap. 15
  - Matrix chain multiplication