Dynamické programování

Karel Richta a kol.

Katedra počítačů
Fakulta elektrotechnická
České vysoké učení technické v Praze

Datové struktury a algoritmy, B6B36DSA
2026

Evropský sociální fond
Praha & EU: Investujeme do vaší budoucnosti

Myšlenka dynamického programování

Definice funkce

$$f(x,y) = \begin{cases} 1 & (x = 0) \;||\; (y = 0) \\ 2 \cdot f(x, y-1) + f(x-1,y) & (x > 0) \;\&\&\; (y > 0) \end{cases}$$

Otázka

f(10,10) = ?

Program

```
int f(int x, int y) {
  if ( (x == 0) || (y == 0))
    return 1;
  return (2* f(x,y-1) + f(x-1,y));
}
```

```
print( f(10,10) );
```

Odpověď

f(10,10) = 127 574 017

Myšlenka dynamického programování

Jednoduchá analýza

```
int count = 0;

public static int f(int x, int y) {
  count++;
  if ( (x == 0) || (y == 0))
    return 1;
  return (2* f(x, y-1) + f(x-1,y));
}
```

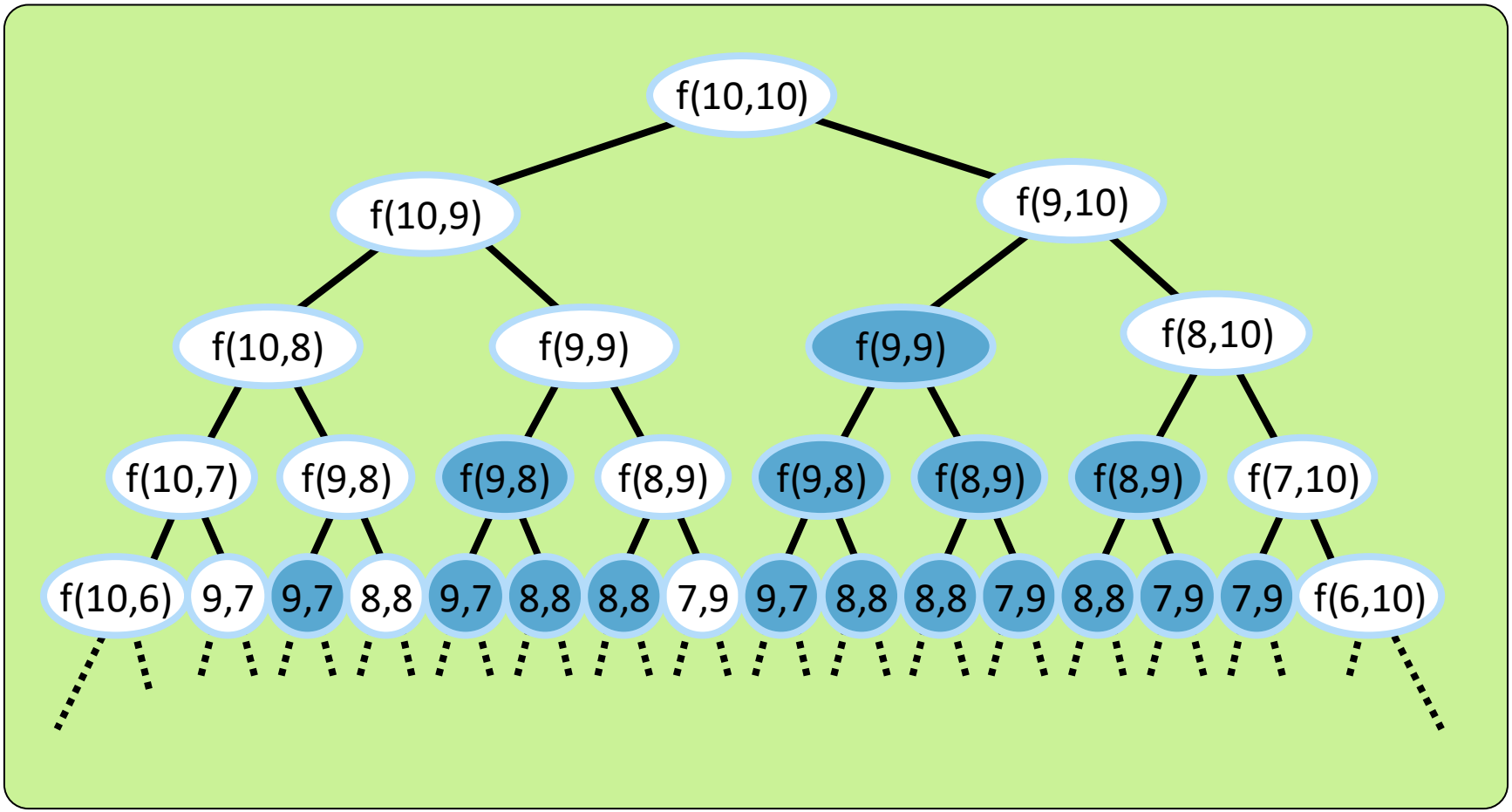
```
xyz = f(10,10);
print(count);
```

Výsledek analýzy

count = 369 511

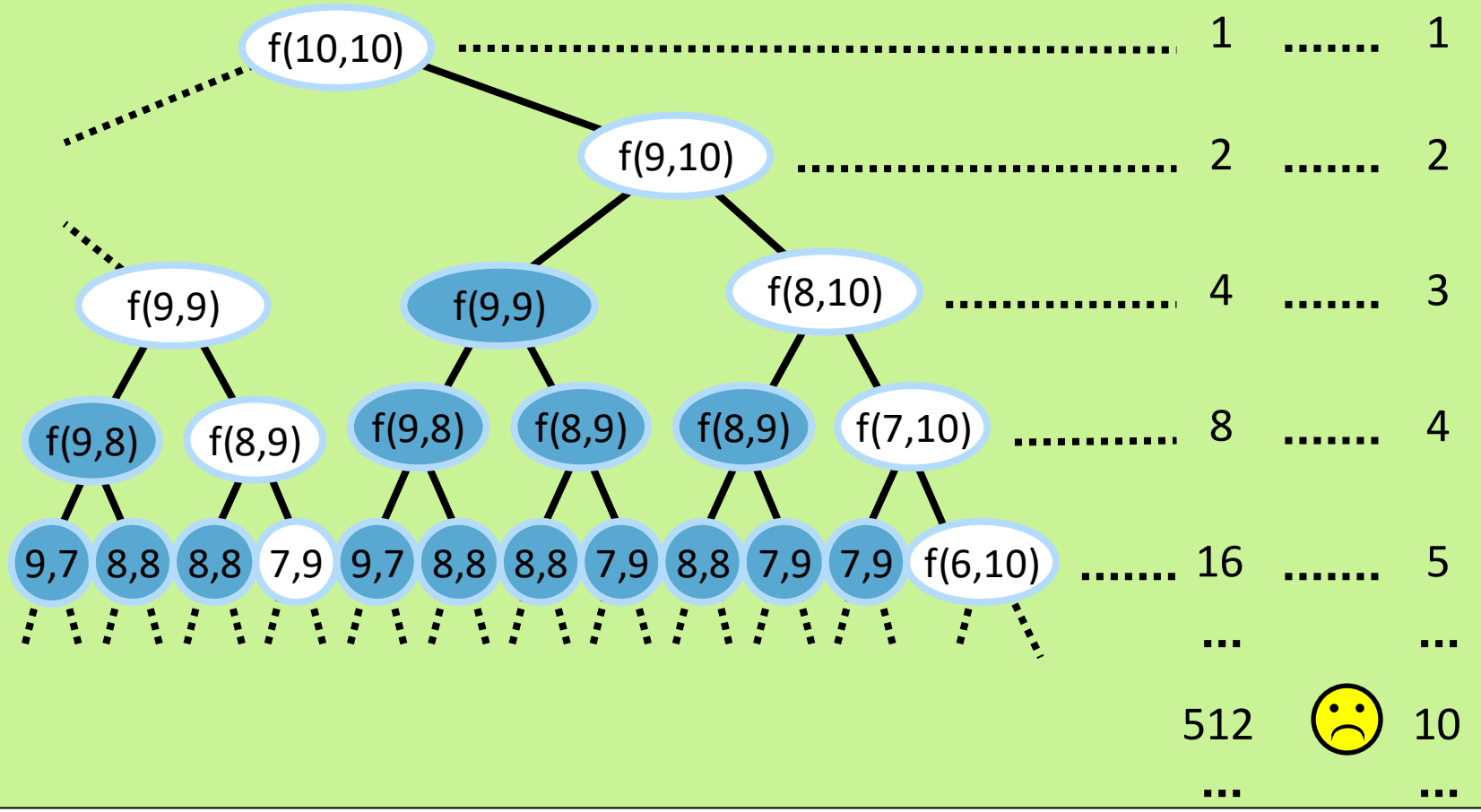
Myšlenka dynamického programování

Detailnější analýza – strom rekurzivního volání



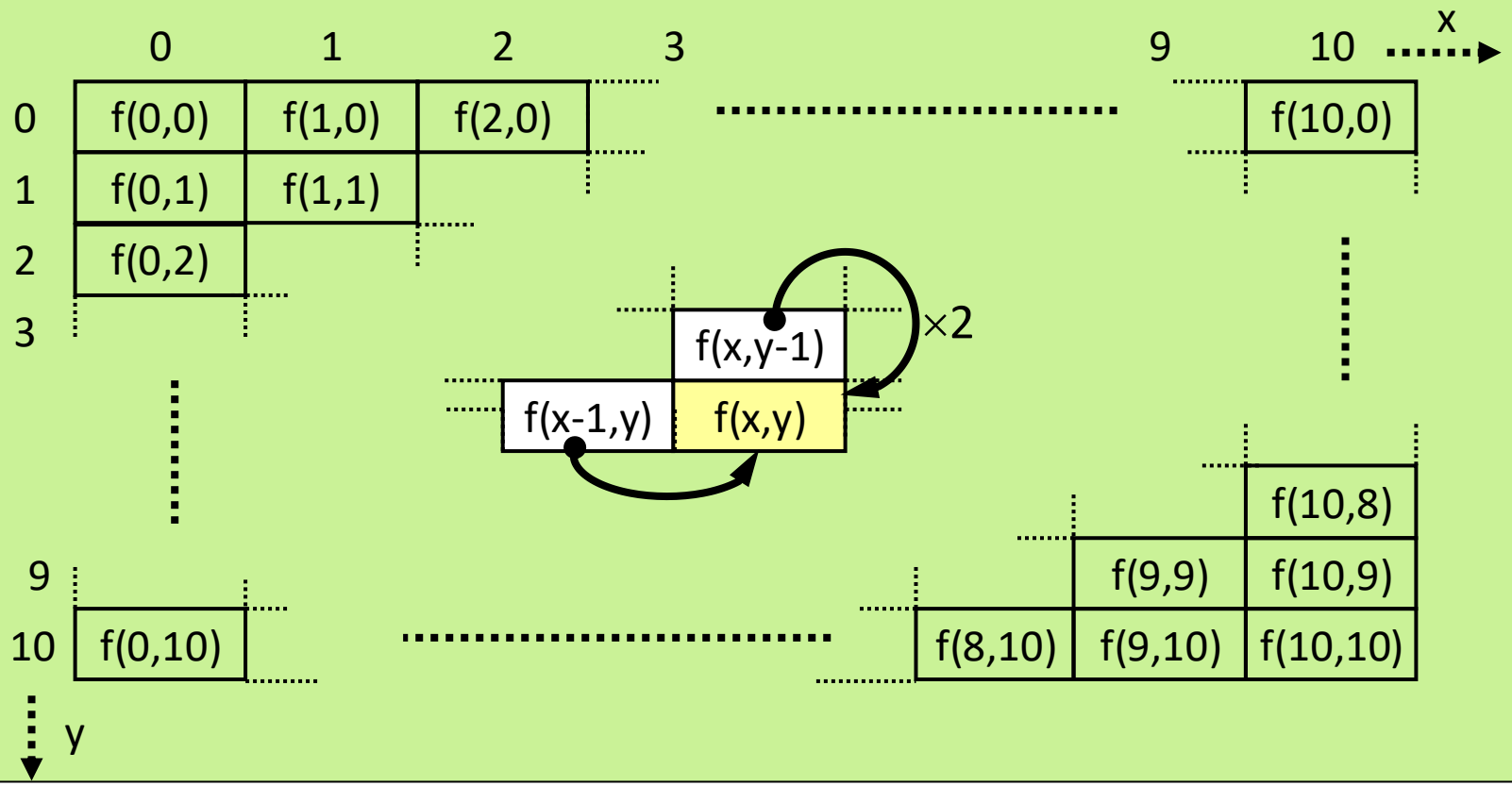
Myšlenka dynamického programování

Detailnější analýza pokračuje – efektivita rekurzivního volání



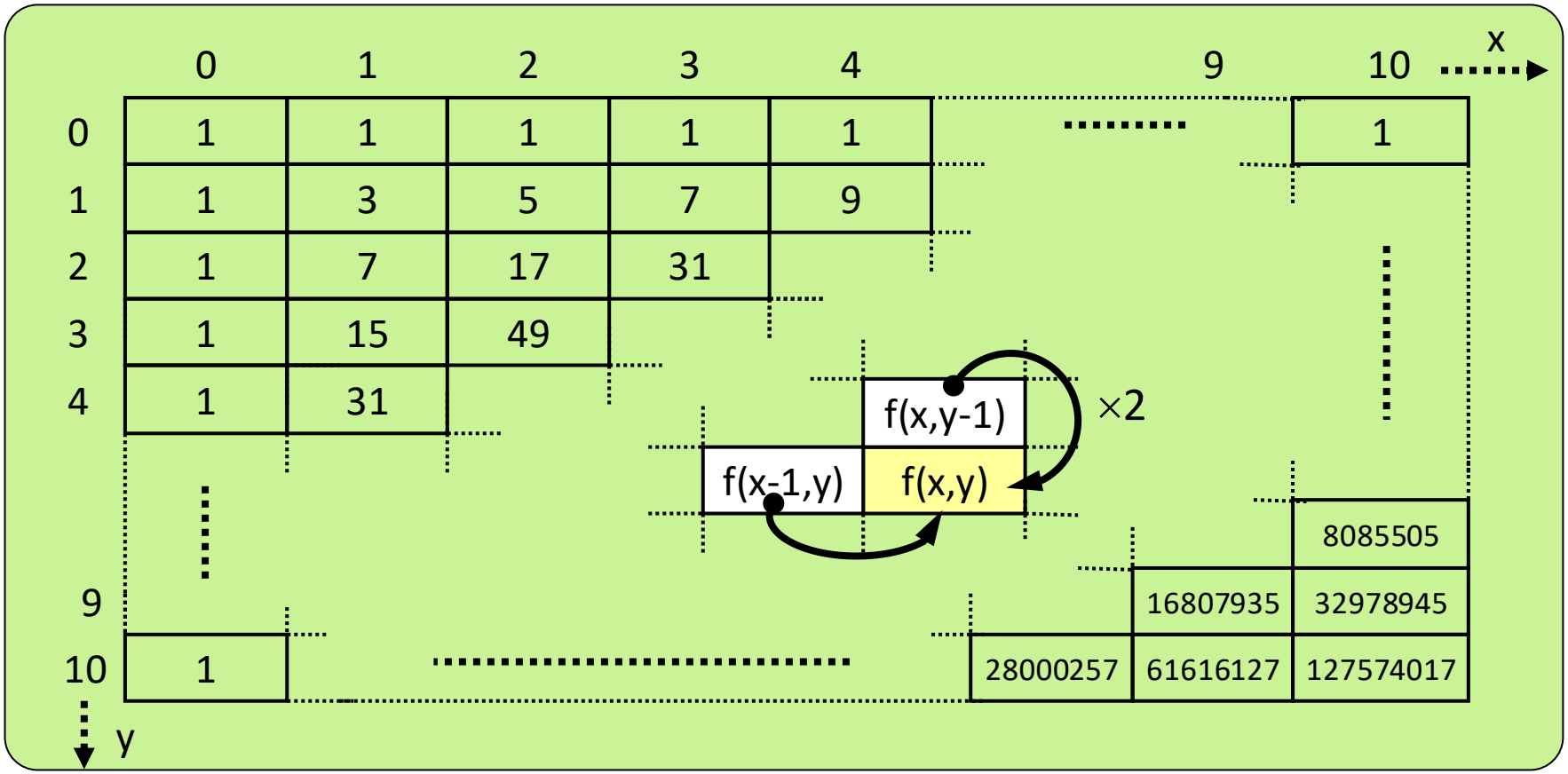
Myšlenka dynamického programování

$$f(x,y) = \begin{cases} 1 & (x = 0) \;||\; (y = 0) \\ 2 \cdot f(x, y-1) + f(x-1,y) & (x > 0) \;\&\&\; (y > 0) \end{cases}$$



Myšlenka dynamického programování

$$f(x,y) = \begin{cases} 1 & (x = 0) \;||\; (y = 0) \\ 2 \cdot f(x, y-1) + f(x-1,y) & (x > 0) \;\&\&\; (y > 0) \end{cases}$$



Myšlenka dynamického programování

Všechny hodnoty se předpočítají

```
static int dynArr [N+1][N+1];

void fillDynArr() {
  int xy, x, y;
  for (xy = 0; xy <= N; xy++)
      dynArr[0][xy] = dynArr[xy][0] = 1;

  for (y = 1; y <= N; y++)
    for (x = 1; x <= N; x++)
       dynArr[y][x] = 2*dynArr[y-1][x] + dynArr[y][x-1];
}
```

Volání funkce

```
int f(int x,int y) {
  return dynArr[y][x];
}
```

Rozděl-a-panuj vs. dynamické programování

- Metoda **rozděl-a-panuj** je vhodná pro řešení úloh, které lze rozdělit na **nezávislé podúlohy**:
  - Rozděl úlohu na nezávislé podúlohy.
  - Rekurzivně vyřeš podúlohy.
  - Zkombinuj řešení podúloh do celkového výsledku úlohy.
- **Dynamické programování** (DP) je vhodné pro řešení úloh, ve kterých se podúlohy **překrývají**, t.j., **řeší se stejné podúlohy**.
- Pro takové úlohy není základní verze metody rozděl-a-panuj vhodná, protože sdílené podúlohy se **opakovaně** (redundantně) řeší znovu a znovu, což muže vést na **řádově vyšší složitost**.
- Metody DP si hodnoty dříve vyřešených podúloh **zapamatují** a při jejich znovuobjevení je využijí.

Význam termínu DP

- Termín ”**programování**” v sousloví ”**dynamické programování**” nemá standardní význam vytváření počítačového kódu.
- Je odvozen z termínu ”**matematické programování**”, což je synonymum pro optimalizaci.
- V této interpretaci, slovo ”**program**” znamená optimální či přijatelný plán (rozpis, rozvrh) pro provedení nějaké množiny souvisejících úkonů.
- Programování v tomto smyslu tedy znamená metodu nalezení takového plánu či rozvrhu akcí.

Dynamické programování pro řešení optimalizačních úloh

Definice optimalizační úlohy: Z více možných řešení hledáme řešení s optimální cenou (maximální či minimální hodnotou).

Vytvoření DP algoritmu pro řešení dané optimalizační úlohy se skládá ze 4 kroků:

- Charakterizuj **strukturu** optimálního řešení.
- Rekurzivně definuj **hodnotu** optimálního řešení.
- Vypočítej efektivně **hodnotu** optimálního řešení:
  - metodou shora dolů,
  - metodou zdola nahoru.
- Zrekonstruuj **strukturu** optimálního řešení z vypočtených hodnot.

- Poslední bod odpadá, pokud stačí pouze cena (hodnota) optimálního řešení.

Příklad: Řetězové násobení matic

Definice:

- Je dána posloupnost matic $A_1, A_2, \ldots, A_n$, matice A_i má rozměry $d_{i-1} \times d_i$. **Řetězové násobení matic** (ŘNM) je úkol vypočítat **součin** $A = A_1 \times A_2 \times \cdots \times A_n$ s **minimální** aritmetickou složitostí (= cenou řešení).
- Jinými slovy, úkolem je najít takové **uzávorkování pořadí** násobení matic, které vede na vynásobení s nejmenším počtem aritmetických operací.
- Předpokládáme klasické násobení matic "řádky krát sloupce". Složitost součinu $A_i \times A_{i+1}$ aproximujeme výrazem $d_{i-1} \cdot d_i \cdot d_{i+1}$ (zanedbáváme konstanty a nižší členy).

Vlastnosti ŘNM

Poznámky:

- Násobení matic je **asociativní**, proto **jakékoli** uzávorkování dá správný výsledek A.
- Různé pořadí násobení má dramatický vliv na aritmetickou složitost výpočtu.
- Přitom kombinatorická složitost ŘNM je exponenciální, takže řešení hrubou silou, zkoušením všech možností, není myslitelná.

Příklad:

- Uvažujme $n = 3$ a $d_0 = 5$, $d_1 = 50$, $d_2 = 10$ a $d_3 = 30$.
- Pak uzávorkování $(A1 \times A2) \times A3$ má aritmetickou složitost $5 \cdot 50 \cdot 10 + 5 \cdot 10 \cdot 30 = 2500 + 1500 = 4000$, zatímco:
- uzávorkování $A1 \times (A2 \times A3)$ má složitost $50 \cdot 10 \cdot 30 + 5 \cdot 50 \cdot 30 = 15000 + 7500 = 22500$!!!

Počet různých uzávorkování

- Označme *Z*(n) počet všech různých způsobů, jak uzávorkovat součin $A_1 \times A_2 \times \cdots \times A_n$.
- Posloupnost matic můžeme rozdělit na dvě části hranicí mezi A_k a A_{k+1} pro libovolné k = 1, 2, . . . , n−1 a pak uzávorkovat rekurzívně a nezávisle na sobě obě vzniklé podposloupnosti $A_1, \ldots, A_k$ a $A_{k+1}, \ldots, A_n$.
- Proto platí rekurentní vztah:

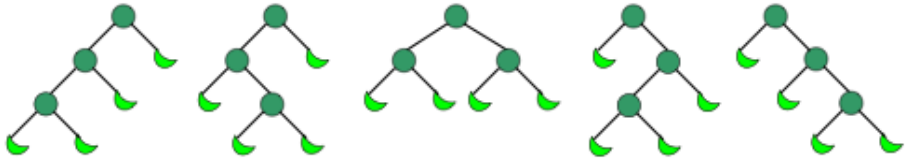
$$Z(n) = \begin{cases} 1 & \text{if } n = 1, \\ \sum_{k=1}^{n-1} Z(k)Z(n-k) & \text{if } n \geq 2. \end{cases}$$

- Řešením jsou tzv. Catalanova čísla: Z(n) = C(n − 1):

$$C(n) = \frac{1}{n+1}\binom{2n}{n} = \Omega\left(4^n/n^{\frac{3}{2}}\right)$$

Příklady použití Catalanových čísel

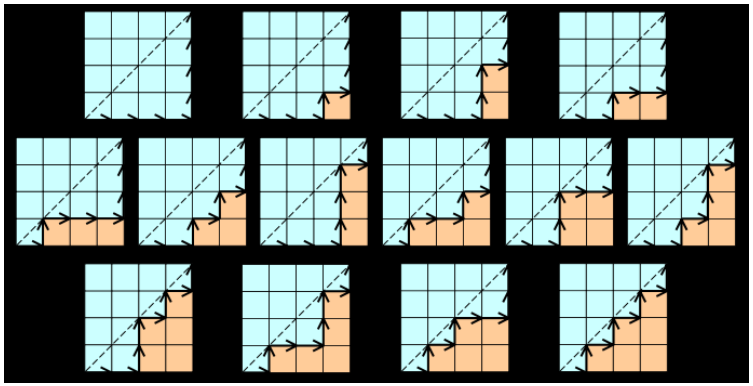
- Počet kořenových binárních stromů s n listy je C(n-1).



- Počet korektních uzávorkování 2n závorek je C(n).

((())) ()(()) ()()() (())() (()())

- Počet způsobů, jak se v mřížce n×n dostat z levého dolního do pravého horního rohu, aniž bychom překročili diagonálu nebo se vraceli, je C(n).



První krok DP: Charakterizování optimálního závorkování

- Označme $A_{i\ldots j}$ matici vzniklou vynásobením $A_i A_{i+1} \ldots A_j$, $i \leq j$.
- Pro každé optimální závorkování existuje **hranice** k, $1 \leq k < n$, taková, že rekurzivně se zkonstruuje závorkování pro výpočet $A_{1\ldots k}$, pak závorkování pro výpočet $A_{k+1\ldots n}$, a výslednou $A = A_{1\ldots n}$ získáme vynásobením $A_{1\ldots k} A_{k+1\ldots n}$.
- Aritmetická složitost ŘNM je tedy složitost výpočtu $A_{1\ldots k}$ + složitost výpočtu $A_{k+1\ldots n}$ + složitost násobení 2 matic $A_{1\ldots k} A_{k+1\ldots n}$.

Lemma

Nechť k je hranice optimálního závorkování (s minimální aritmetickou složitostí). Pak závorkování pro výpočet $A_{1\ldots k}$ i závorkování pro výpočet $A_{k+1\ldots n}$ musejí být optimální.

- **Důkaz.** Důkaz sporem. Pokud by např. první nebylo, existovalo by závorkování pro $A_{1\ldots k}$ s nižší cenou a tím by i výsledná cena byla nižší, což je spor, protože dané závorkování pro $A_{1\ldots n}$ je optimální. ■

Druhý krok DP: Rekurzivní definice ceny optimálního závorkování

- Nechť $m[i,j]$ je minimální **cena závorkování** (=nejmenší aritmetická složitost násobení) $A_i A_{i+1} \dots A_j$, $i \leq j$. Pak platí:

$$m[i,j] = \begin{cases} 0 & \text{if } i = j, \\ \min_{i \leq k < j} \{m[i,k] + m[k+1,j] + d_{i-1} d_k d_j\} & \text{if } i < j. \end{cases} \quad (1)$$

- Označme $h[i,j]$ právě takové k, které v (1) dává minimum, čili

$$m[i,j] = m[i,h[i,j]] + m[h[i,j]+1,j] + d_{i-1} d_{h[i,j]} d_j.$$

- Pak cena optimálního závorkování je

$$m[1,n] = m[1,h[1,n]] + m[h[1,n]+1,n] + d_0 d_{h[1,n]} d_n.$$

Třetí krok DP: Algoritmus výpočtu ceny optimálního závorkování

Přímý rekurzivní algoritmus $\textsc{RecRNM}(d, 1, n)$ realizující výpočet $m[1, n]$ podle (1) nelze použít, protože má **exponenciální složitost**!!!

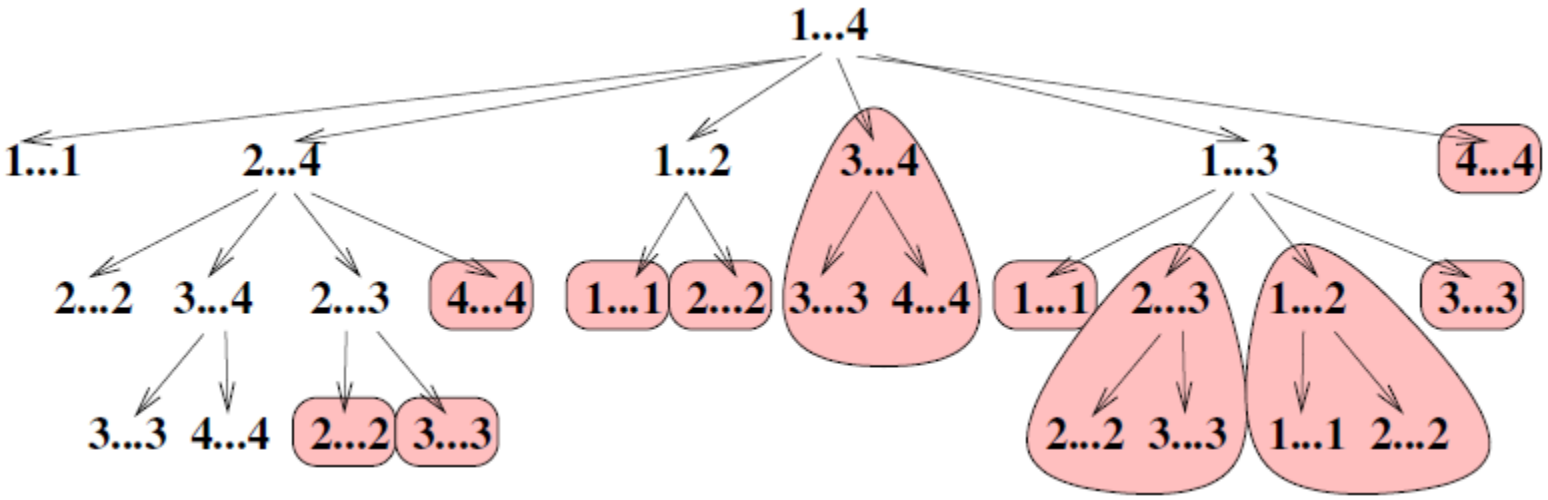
```
procedure RecRNM(d, i, j)
{
(1)     if (i = j) then return(0);
(2)     m[i, j] ← +∞;
(3)     for (k ← i to j − 1)
(4)         do {q ← RecRNM(d, i, k) + RecRNM(d, k + 1, j)
(5)                 +d[i − 1] · d[k] · d[j];
(6)             if (q < m[i, j]) then { m[i, j] ← q; h[i, j] ← k}};
(7)     return(m[i, j], h[i, j]);
}
```

Strom rekurzivních volání RecRNM

- Důvod exponenciální složitosti je podobný jako u rekurzivního výpočtu v úvodu.
- Celkový počet hodnot m[*,*], které je třeba vypočítat pro zjištění hodnoty m[1, n], je pouze počet všech dvojic i, j takových, že $1 \leq i \leq j \leq n$, což je

$$\binom{n}{2} + n = \binom{n+1}{2} \doteq \frac{n^2}{2}$$

- Dochází však masivně k opakovanému výpočtu stejných hodnot, viz příklad stromu rekurzivních volání RecRNM(1, 4).



Časová složitost RecRNM(d, 1, n)

Lemma

Časová složitost $t(n)$ *běhu* $\textsc{RecRNM}(d, 1, n)$ *je* $\Omega(2^{n-1})$.

Důkaz. (Substituční metodou.)

- Předpokládejme, že kód na řádku (1) a (6) trvá čas aspoň 1.
- Pak triviálně $t(1) \geq 1$.
- Dále pro $n > 1$ je $t(n) \geq 1 + \sum_{k=1}^{n-1}(t(k) + t(n-k) + 1)$.
- Díky symetrii sčítanců to je totéž jako $t(n) \geq n + 2\sum_{i=1}^{n-1} t(i)$.
- Ukažme indukcí, že $t(n) \geq 2^{n-1}$.
  - Indukční základ: $T(1) \geq 1 = 2^0$ triviálně z předpokladů.
  - Indukční krok pro $n \geq 2$:
    $t(n) \geq n + 2\sum_{i=1}^{n-1} 2^{i-1} = n + 2(2^{n-1} - 1) = n + 2^n - 2 \geq 2^{n-1}$. ■

Třetí krok DP algoritmu: Efektivní varianta 1.

Nerekurzivní algoritmus výpočtu ceny optim. závorkování zdola nahoru.

Předpoklady:

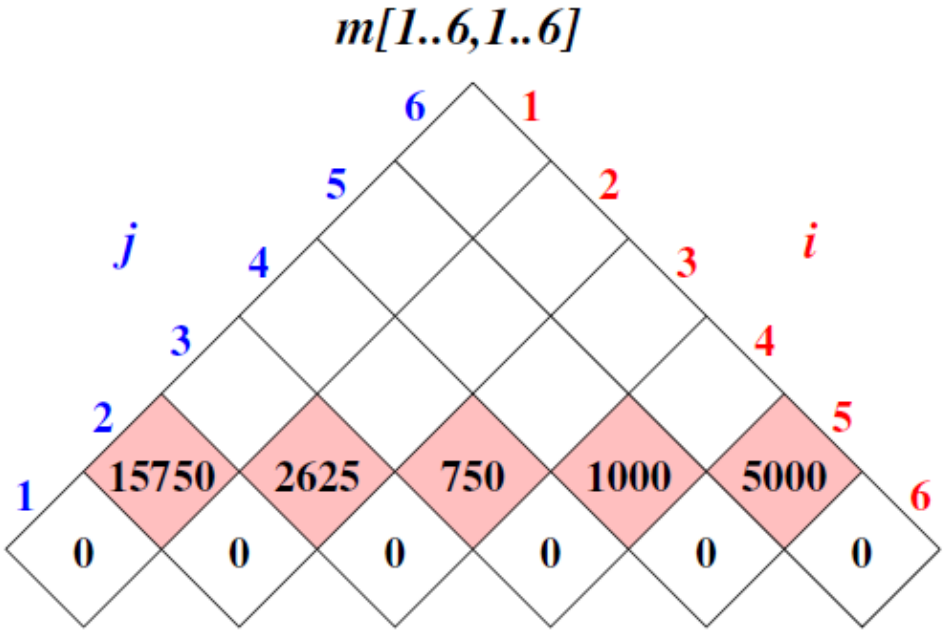
- Vstupem je pole d[0, . . . , n] rozměrů matic.
- Výstupem jsou 2 pole:
  - m[1, . . . , n, 1, . . . , n], kam se ukládají ceny optimálního závorkování pro výpočet $A_{i...j}$,
  - h[1 . . . n − 1, 2 . . . n], kam se ukládají příslušné hodnoty hranic optimálního závorkování.

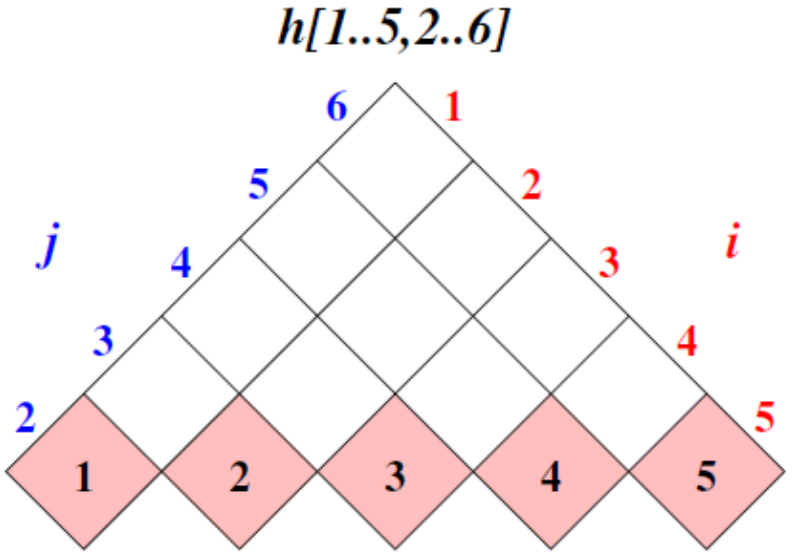
Princip:

- Systematické zaplňování tabulky zdola nahoru pro všechny dvojice i, j, kde $1 \leq i \leq j \leq n$ ($\binom{n+1}{2}$ hodnot).

```
procedure BottomUpRNM(n, d, m, h)
{
(1)for (i ← 1 to n) do m[i, i] ← 0;
(2)for (l ← 2 to n)
(3)       for (i ← 1 to n − l + 1)
(4)             do { j ← i + l − 1;
(5)                  m[i, j] ← +∞;
(6)                  for (k ← i to j − 1)
(7)                        do { q ← m[i, k] + m[k + 1, j] + d[i − 1] · d[k] · d[j];
(8)                             if (q < m[i, j])
(9)                             then { m[i, j] ← q;
(10)                                   h[i, j] ← k};
(11)                 }       };
(12)return(m[1 . . . n, 1 . . . n], h[1 . . . n − 1, 2 . . . n]);
}
```

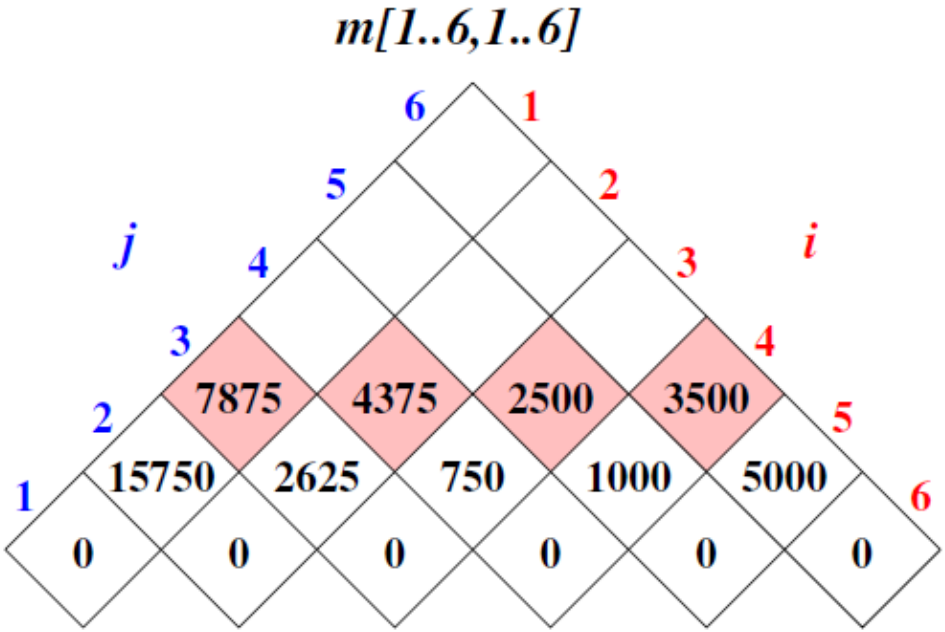
Příklad běhu BottomUpRNM(6, d,m, h)

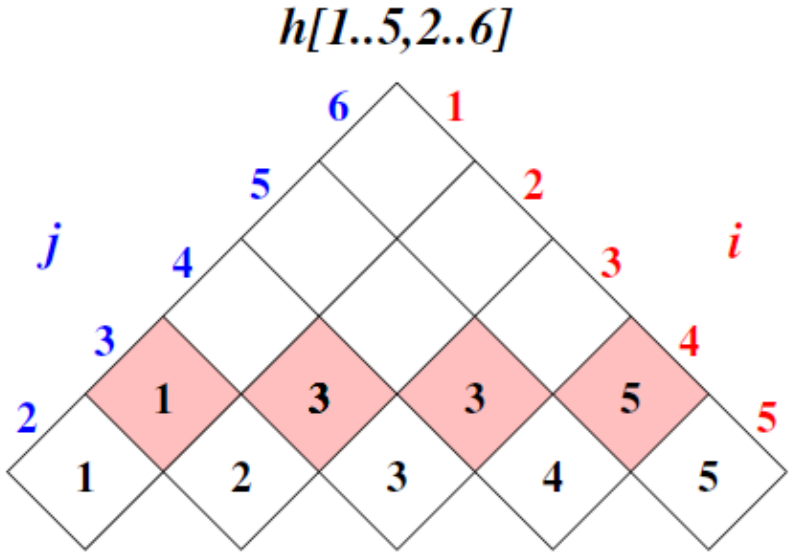




A_1	A_2	A_3	A_4	A_5	A_6
30×35	35×15	15×5	5×10	10×20	20×25

Příklad běhu BottomUpRNM(6, d,m, h)

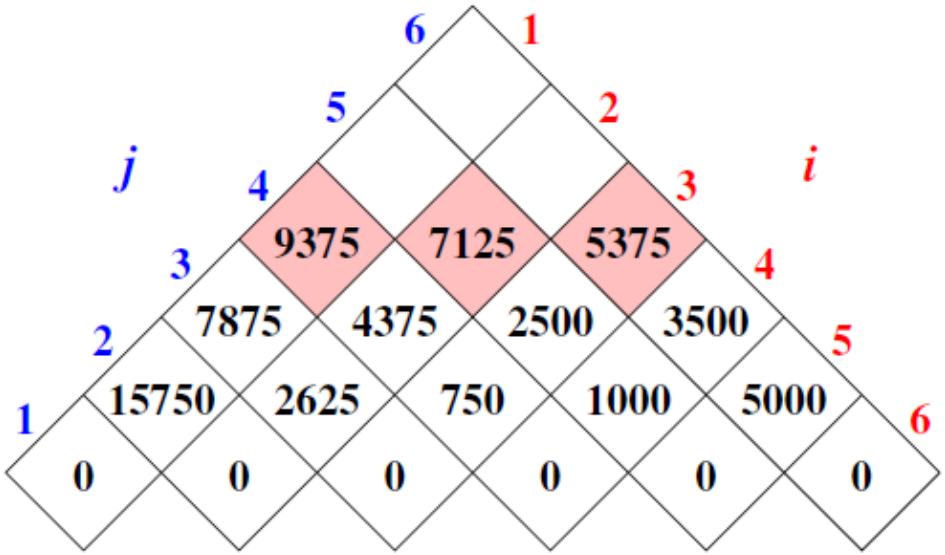




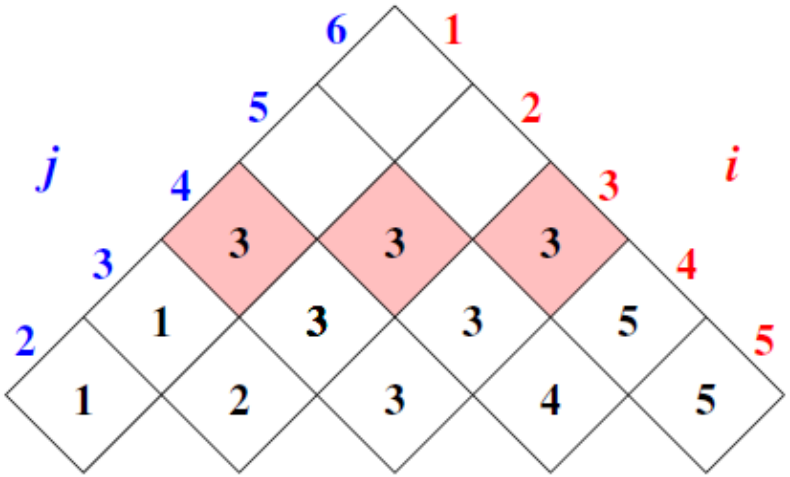
A_1	A_2	A_3	A_4	A_5	A_6
30×35	35×15	15×5	5×10	10×20	20×25

Příklad běhu BottomUpRNM(6, d,m, h)

m[1..6,1..6]

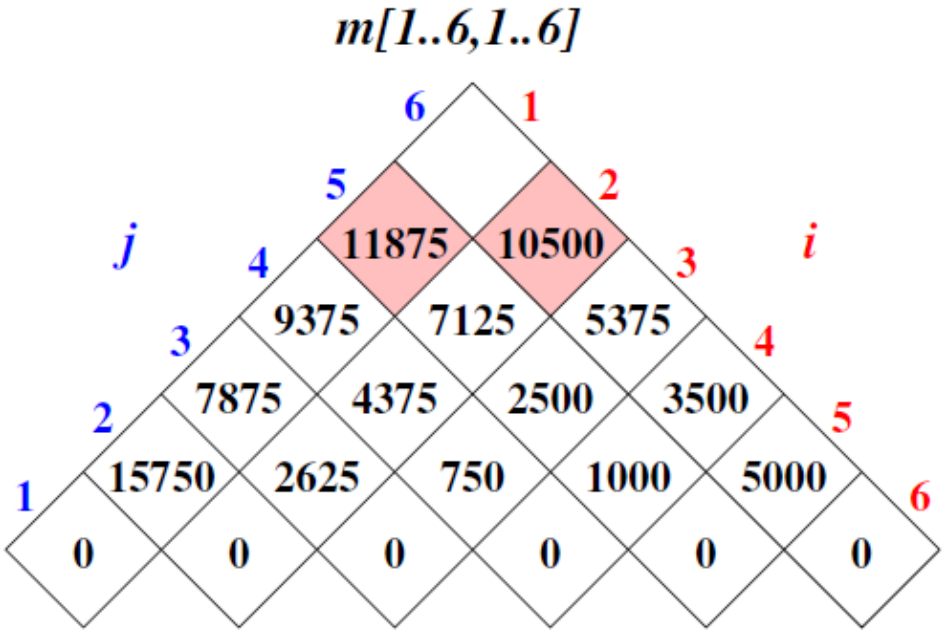


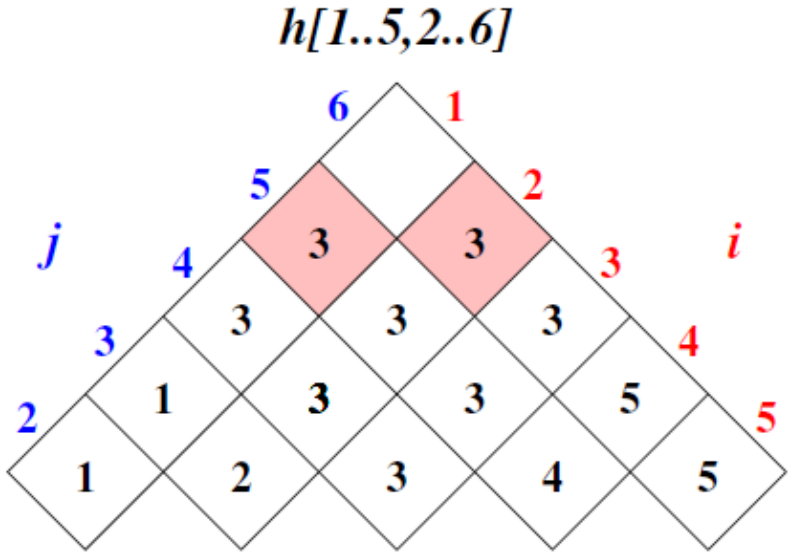
h[1..5,2..6]



A_1	A_2	A_3	A_4	A_5	A_6
30×35	35×15	15×5	5×10	10×20	20×25

Příklad běhu BottomUpRNM(6, d,m, h)

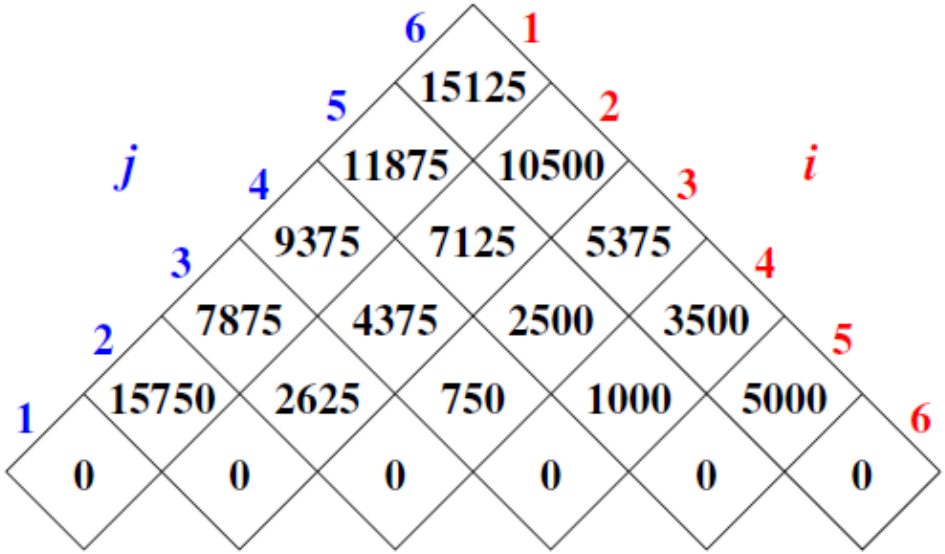




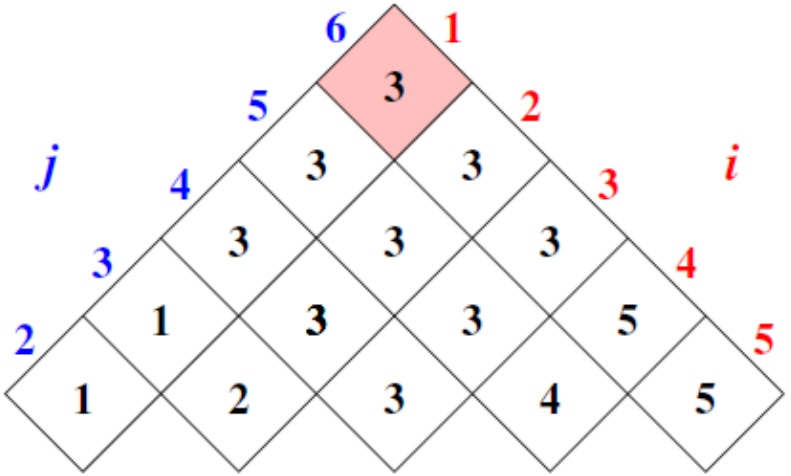
A_1	A_2	A_3	A_4	A_5	A_6
30×35	35×15	15×5	5×10	10×20	20×25

Příklad běhu BottomUpRNM(6, d,m, h)

m[1..6,1..6]



h[1..5,2..6]



A_1	A_2	A_3	A_4	A_5	A_6
30×35	35×15	15×5	5×10	10×20	20×25

Třetí krok DP algoritmu: Varianta 2

Rekurzivní algoritmus (shora dolu) výpočtu ceny optimálního závorkování s tabelací

Princip tabelace (anglicky memoization):

- Algoritmus také vytváří tabulku s hodnotami řešení podúloh.
- Každá položka tabulky je také inicializována speciální hodnotou **`Neurceno`**.
- Řídící struktura pro zaplňování však není záplavové plnění zdola nahoru jako ve variantě 1, ale pořadí vyvolávané rekurzívním algoritmem.
- První rekurzívně vyvolané řešení dané podúlohy provede výpočet a výsledek uloží do příslušného místa v tabulce.
- Následné opakované volání řešení téže podúlohy pouze přečte tuto hodnotu z tabulky.

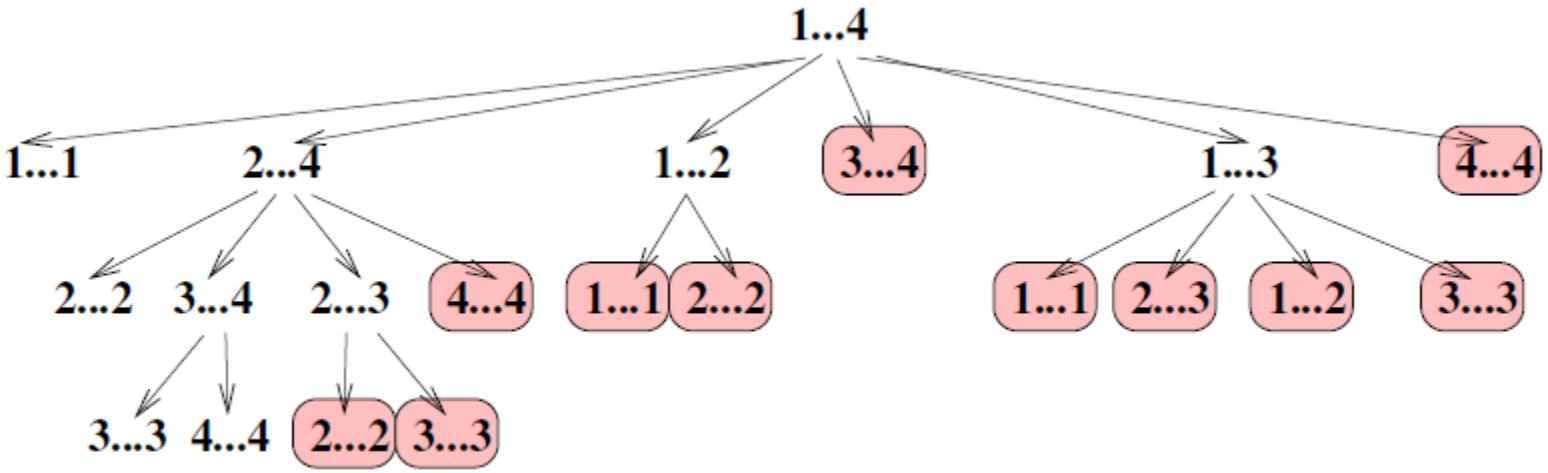
Rekurzivní algoritmus s tabelací

```
procedure MemRecRNM(n, d) {
(1)   for (i ← 1 to n)
(2)      for (j ← i to n) do m[i, j] ← Neurceno.
(3)   return(LookUpRNM(d, 1, n))}

procedure LookUpRNM(d, i, j) {
(1)   if (m[i, j] ≠ Neurceno)
(2)      then return(m[i, j]);
(3)   if (i = j)
(4)      then m[i, j] ← 0
(5)      else for (k ← i to j − 1)
(6)              do {q ← LookUpRNM(d, i, k)
(7)                 +LookUpRNM(d, k + 1, j) + d[i − 1] · d[k] · d[j];
(8)                 if (q < m[i, j]) then {m[i, j] ← q; h[i, j] ← k}};
(9)   return(m[i, j])}
```

Strom rekurzivních volání MemRecRNM

- Zakroužkované uzly stromu rekurzivních volání se nepočítají, ale pouze načtou z tabulky.



Srovnání MemRecRNM a BottomUpRNM

Lemma:

- Oba algoritmy mají složitost $O(n^3)$.

Srovnání:

- Algoritmus MemRecRNM je svojí podstatou metoda shora dolů, kdežto BottomUpRNM je metoda zdola nahoru.
- Pokud logika výpočtu vyžaduje, aby každá podúloha byla řešena aspoň jednou, pak BottomUpRNM je rychlejší o multiplikativní konstantu.
- Pravidelnost přístupu do tabulky pak uspoří čas i díky menším výpadkům cache a efektivnějším tokům dat mezi pamětí a procesorem.
- Pokud rekurzivní sestup umožní, že některé podúlohy se vůbec nevyvolají, pak může být rychlejší MemRecRNM.

Čtvrtý krok DP algoritmu: Konstrukce optimálního řešení úlohy ŘNM

- Tabulka h[1 . . . n − 1, 2 . . . n] obsahuje hodnoty hranic pro optimální závorkování při ŘNM.
- Díky Lemmatu o možném uzávorkování je konstrukce optimálního řešení triviální - rekurzivním sestupem.
- Na nejvyšší úrovni to je h[1, n]: Nejprve rekurzívně optimálně vypočteme $A_1 \times \ldots \times A_{h[1,n]}$, pak $A_{h[1,n]+1} \times \ldots \times A_n$ a výsledné matice nakonec vynásobíme.

Čtvrtý krok DP algoritmu: Konstrukce optimálního řešení úlohy ŘNM

- Úloha ŘNM je pak vyřešena voláním $\mathrm{RNM}(A, n, h, 1, n)$.
- Vstupní data: posloupnost matic $A_1, A_2, \ldots, A_n$ a tabulka hranic $h[1 \ldots n-1, 2 \ldots n]$.

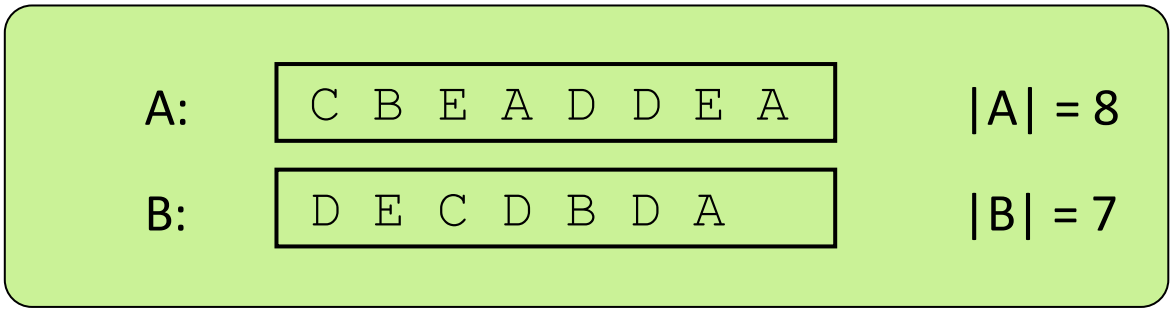
procedure $\mathrm{RNM}(A, n, h, i, j)$
{
(1) **if** $(i < j)$
(2) **then** { $X \leftarrow \mathrm{RNM}(A, n, h, i, h[i, j])$;
(3) $Y \leftarrow \mathrm{RNM}(A, n, h, h[i, j] + 1, j)$;
(4) **return**(XY)}
(5) **else return**(A_i);
}

DYNAMICKÉ PROGRAMOVÁNÍ:

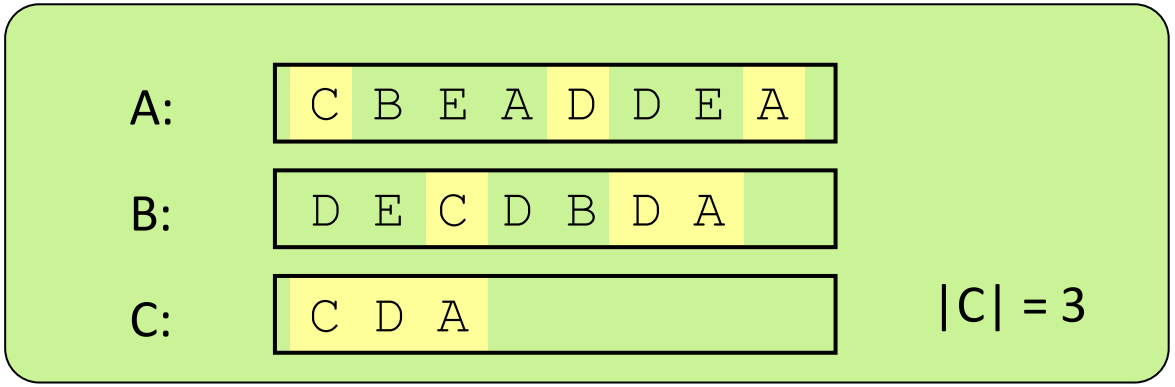
NEJDELŠÍ SPOLEČNÁ PODPOSLOUPNOST

Nejdelší společná podposloupnost

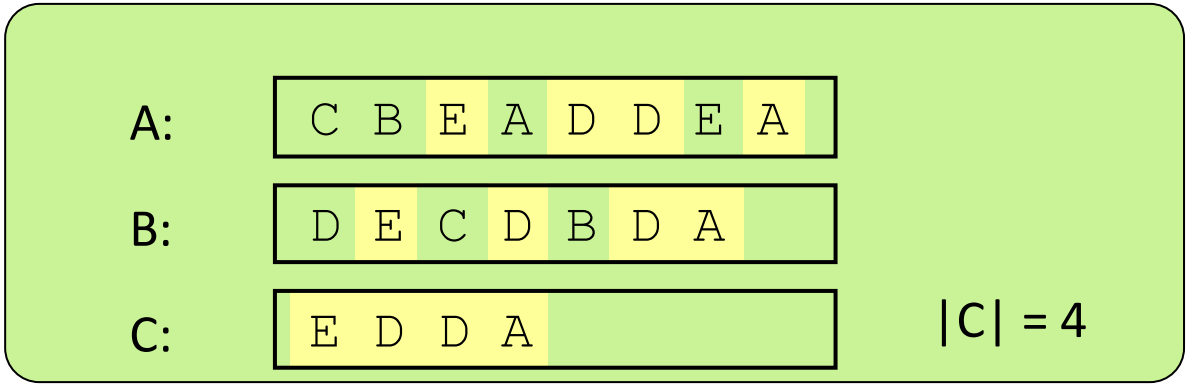
Dvě posloupnosti



Společná podposloupnost



Nejdelší společná podposloupnost (NSP, angl: LCS)



Nejdelší společná podposloupnost

A_n: $(a_1, a_2, ..., a_n)$

B_m: $(b_1, b_2, ..., b_m)$

C_k: $(c_1, c_2, ..., c_k)$

.....

$C_k = LCS(A_n, B_m)$

	1	2	3	4	5	6	7	8
A_8:	C	B	E	A	D	D	E	A
B_7:	D	E	C	D	B	D	A	
C_4:	E	D	D	A				

Rekurzivní pravidla:

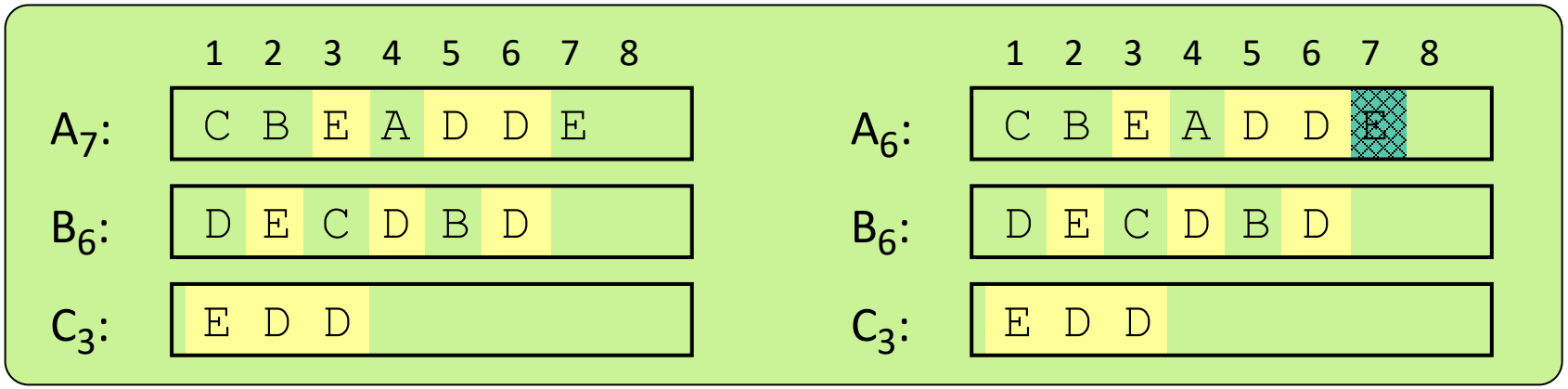
$(a_n = b_m) \Rightarrow (c_k = a_n = b_m) \ \& \ (C_{k-1} = LCS (A_{n-1}, B_{m-1}))$

	1	2	3	4	5	6	7	8
A_8:	C	B	E	A	D	D	E	A
B_7:	D	E	C	D	B	D	A	
C_4:	E	D	D	A				

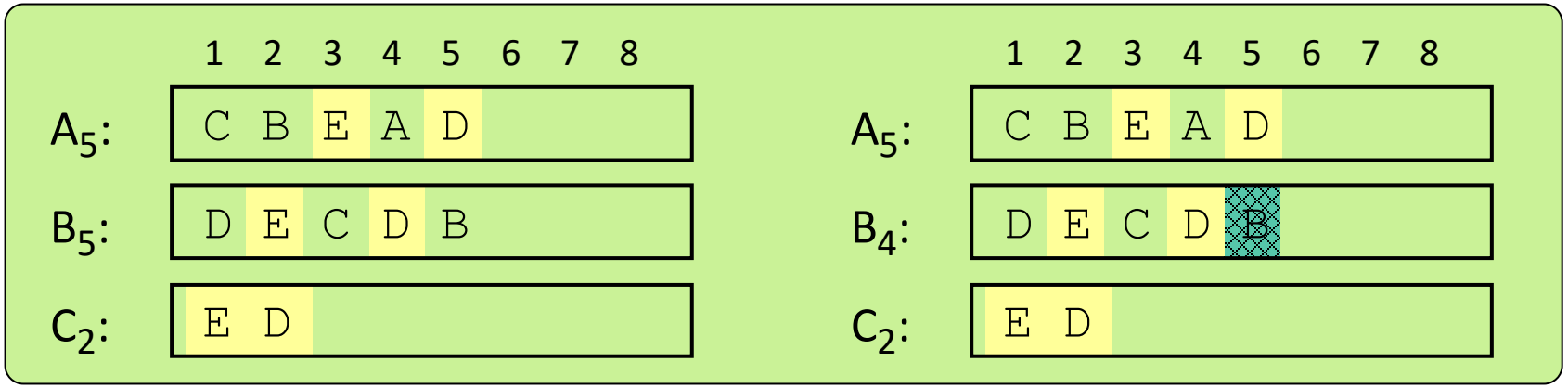
	1	2	3	4	5	6	7	8
A_7:	C	B	E	A	D	D	E	A
B_6:	D	E	C	D	B	D	A	
C_3:	E	D	D	A				

Nejdelší společná podposloupnost

$(a_n != b_m) \ \& \ (c_k != a_n) \Longrightarrow (C_k = LCS (A_{n-1}, B_m))$



$(a_n != b_m) \ \& \ (c_k != b_m) \Longrightarrow (C_k = LCS (A_n, B_{m-1}))$

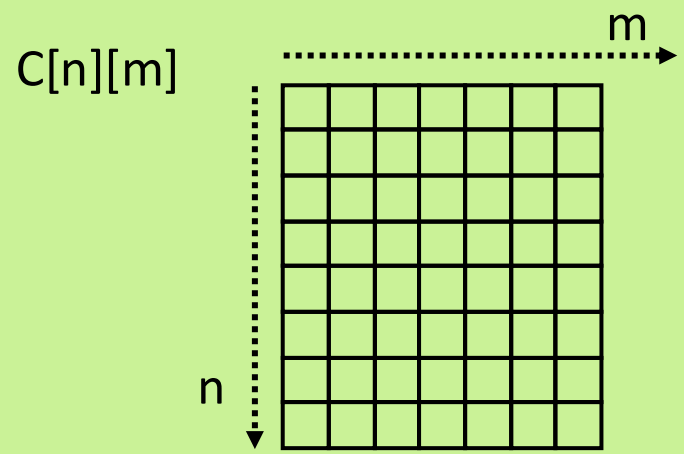


Nejdelší společná podposloupnost

Rekurzivní funkce – délka NSP

$$C(n,m) = \begin{cases} 0 & n = 0 \text{ or } m = 0 \\ C(n-1, m-1) + 1 & n > 0, m > 0, a_n = b_m \\ \max\{ C(n-1, m), C(n, m-1) \} & n > 0, m > 0, a_n \neq b_m \end{cases}$$

Strategie dynamického programování




```
for (a=1; a<=n; a++)
  for (b=1; b<=m; b++)
    C[a][b] = .... ;
```

Nejdelší společná podposloupnost

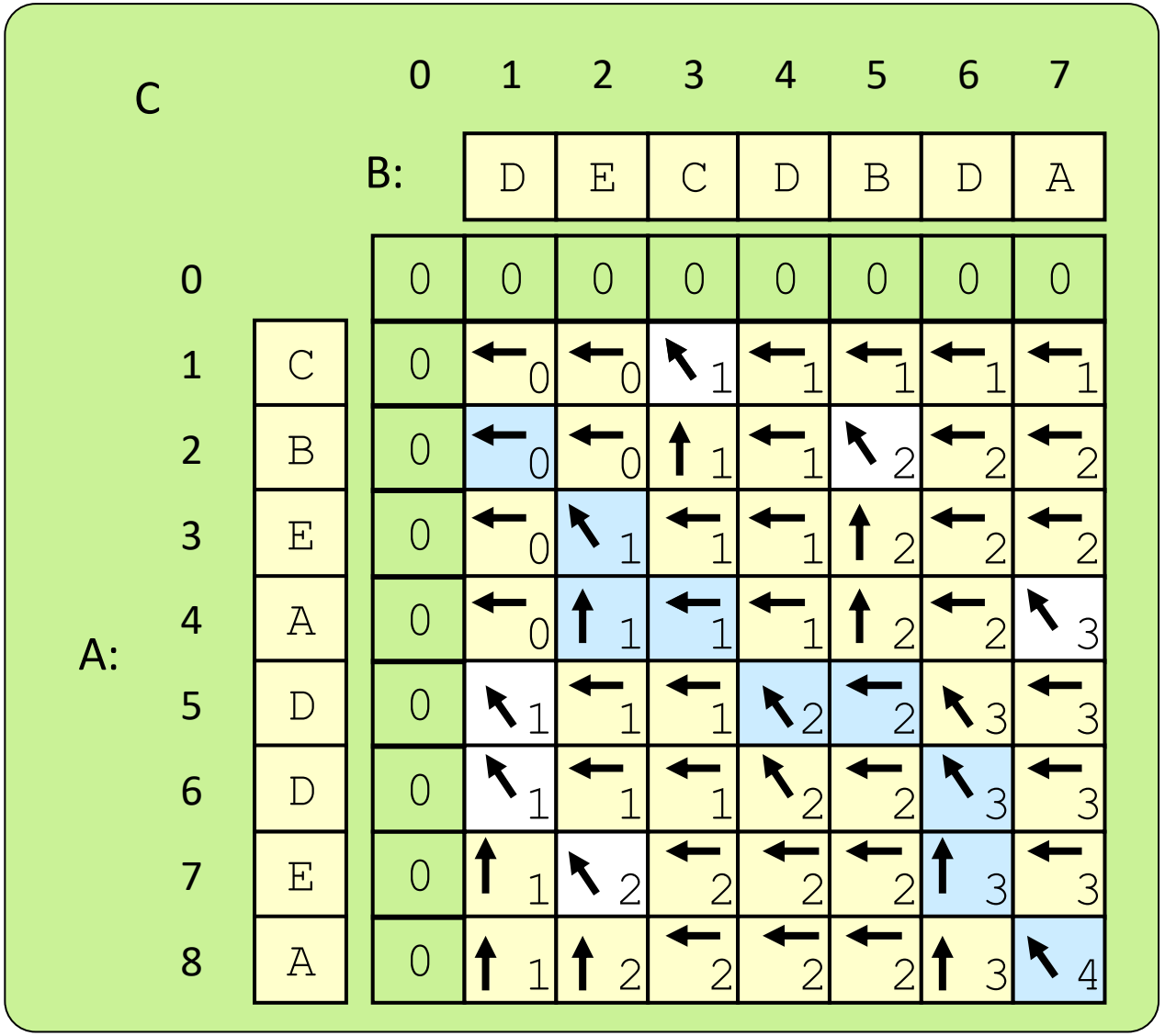
Konstrukce pole pro NSP

```
void findLCS() {
  int a, b;
  for (a=1; a<=n; a++)
    for (b=1; b <= m; b++)
      if (A[a] == B[b]) {
         C[a][b] = C[a-1][b-1]+1;
         arrows[a][b] = DIAG;
         }
      else
        if (C[a-1][b] > C[a][b-1]) {
           C[a][b] = C[a-1][b];
           arrows[a][b] = UP;
        }
        else {
          C[a][b] = C[a][b-1];
          arrows[a][b] = LEFT;
 }      }
```

Nejdelší společná podposloupnost

Pole NSP pro

“CBEADDEA” a “DECDBDA”



Nejdelší společná podposloupnost

Výpis NSP -- rekurzivně :)

```
void outLCS(int a, int b) {
if ((a == 0 ) || (b == 0)) return;

if (arrows[a][b] == DIAG) {
    outLCS(a-1, b-1);       // recursion ...
    print(A[a]);            //... reverses the sequence!
  }
else
    if (arrows[a][b] == UP)
        outLCS(a-1,b);
    else
        outLCS(a,b-1);
}
```

DYNAMICKÉ PROGRAMOVÁNÍ:

ŘEZÁNÍ TYČÍ

Problém řezání tyčí

- Firma Serling Enterprises se zabývá prodejem železných tyčí. Nakupuje tyče určité délky, řeže je na menší délky a prodává. Předpokládáme, že řez má zanedbatelnou cenu. Chceme navrhnout optimální sadu řezů.
- Jako vstup uvažujeme řadu cen p_i, které si firma účtuje za tyč délky i (1,2,…). Problém je nalézt pro tyč délky n optimální řezy tak, aby zisk byl co největší. Pozn.: pokud by cena p_n byla dostatečně vysoká, bude optimální vůbec neřezat.
- Př.: Uvažme tabulku cen:

délka i	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
cena p_i	1	5	8	9	10	11	17	20	24	30

- Pro tyč délky 4 je optimální jeden řez na dvě tyče délky 2.
- Tyč délky 2 se řezat nevyplatí.

Problém řezání tyčí (pokr.)

- Pokud tyč délky n rozřízneme na k kusů, pak $n=i_1+i_2+\ldots+i_k$.
- Označme si r_n zisk z takového řezu: $r_n=p_{i1}+p_{i2}+\ldots+p_{ik}$.

l	p_i	r_i	řešení
1	1	1	1 = 1 (bez řezu)
2	5	5	2 = 2 (bez řezu)
3	8	8	3 = 3 (bez řezu)
4	9	10	4 = 2 + 2
5	10	13	5 = 2 + 3
6	11	17	6 = 6 (bez řezu)
7	17	18	7 = 1 + 6 nebo 7 = 2 + 2 + 3
8	20	22	8 = 2 + 6
9	24	25	9 = 3 + 6
10	30	30	(bez řezu)

Problém řezání tyčí (pokr.)

- Zisk lze spočítat z předchozích hodnot dle vztahu:
  $r_n = \max(p_n, r_1+r_{n-1}, r_2+r_{n-2}, \ldots, r_{n-1}+r_1)$.
- Abychom mohli spočítat řešení problém rozsahu n, musíme spočítat řešení problémů velikosti n-1, n-2, atd. Navíc, pokud rozřízneme tyč na dva kusy, můžeme uvažovat řešení obou částí jako nezávislá, ze kterých pak určíme optimální řešení celku.
- Výše uvedenou rovnost lze ještě zjednodušit (uvažujeme $r_0 = 0$):

  $$r_n = \max_{1 \le i \le n} (pi + r_{n-i})$$

Rekurzivní řešení

```
CUT-ROD(p, n)
1  if n == 0
2       return 0
3  q = -∞
4  for i = 1 to n
5       q = max(q, p[i] + CUT-ROD(p, n - i))
6  return q
```

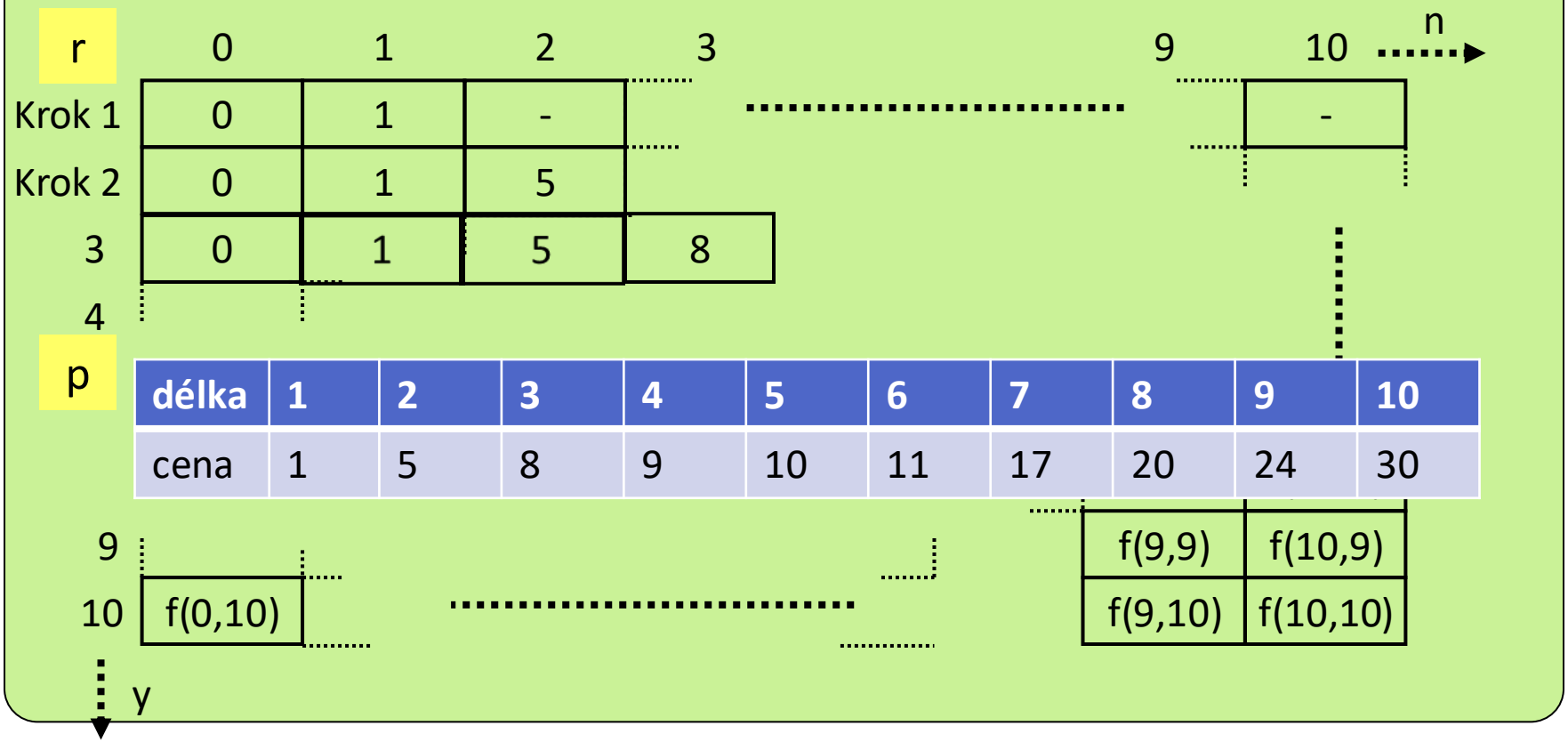
$$T(n) = 1 + \sum_{j=0}^{n-1} T(j) .$$

$$T(n) = 2^n$$

```
public static int cut_rod(int p[], int n) {
      int q;
      if (n == 0) return 0;
      q = MIN_VALUE;
      for (int i = 1; i < n; i++){
         q = Math.max(q, p[i]+cut_rod(p,n-i));
      };
      return q;
   };
```

Řešení dynamickým programováním

$$r_n = \max_{1 \le i \le n} (pi + r_{n-i})$$



Řešení pomocí dynamického programování

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

$\Theta(n^2)$.

```
public static int bottom_up_cut_rod(int p[], int n) {
    int q = 0; int r[] = new int[n];
    r[0] = 0;
    for (int j = 1; j < n; j++){
       q = MIN_VALUE;
       for (int i = 1; i < j; i++){
          q = Math.max(q, p[i]+r[j-i]);
       };
       r[j] = q;
    };
    return q;
 };
```

Příklady úloh vhodných pro řešení metodou DP

- Floydův algoritmus hledání nejkratších cest v grafech.
- Optimální plánování výpočetních úloh, zadaných časovými intervaly, ve kterých mohou běžet.
- Výpočet Levenshteinovy vzdálenosti mezi 2 textovými řetězci.
- Určení způsobu, jak nejlépe daný řetězec vygenerovat pomocí dané bezkontextové gramatiky.
- Hledání nejdelšího společného podřetězce dvou řetězců.
- Optimální triangularizace konvexního mnohoúhelníku.
- Problém obchodního cestujícího:
  - Obecný případ v exponenciálním, ale o(n!) čase.
  - Spec. případy v polynomiálním čase.
- . . .

The End