STU FIIT

SLOVENSKÁ TECHNICKÁ UNIVERZITA V BRATISLAVE
FAKULTA INFORMATIKY A INFORMAČNÝCH TECHNOLÓGIÍ

Dátové štruktúry a algoritmy

Dynamické programovanie

21. 11. 2017

zimný semester 2017/2018

Dynamické programovanie (DP)

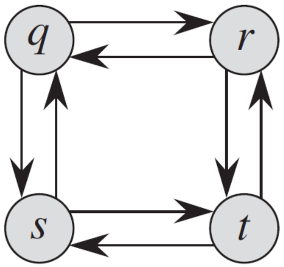
- Všeobecná metóda riešenia komplexných problémov rozdelením na jednoduchšie podproblémy
  - Vyriešiť každý podproblém najviac raz a zapamätať výsledok
  - Keď sa objaví rovnaký podproblém, využijeme už vypočítaný výsledok, čím ušetríme čas výpočtu na úkor použitej pamäte

- Dve hlavné charakteristiky
  1. Optimálna podštruktúra riešenia
  2. Prelínajúce sa podproblémy

DP – Optimálna podštruktúra

- Optimálna podštruktúra (riešenia)
  - Optimálne riešenie problému je možné efektívne zostrojiť z optimálnych riešení podproblémov

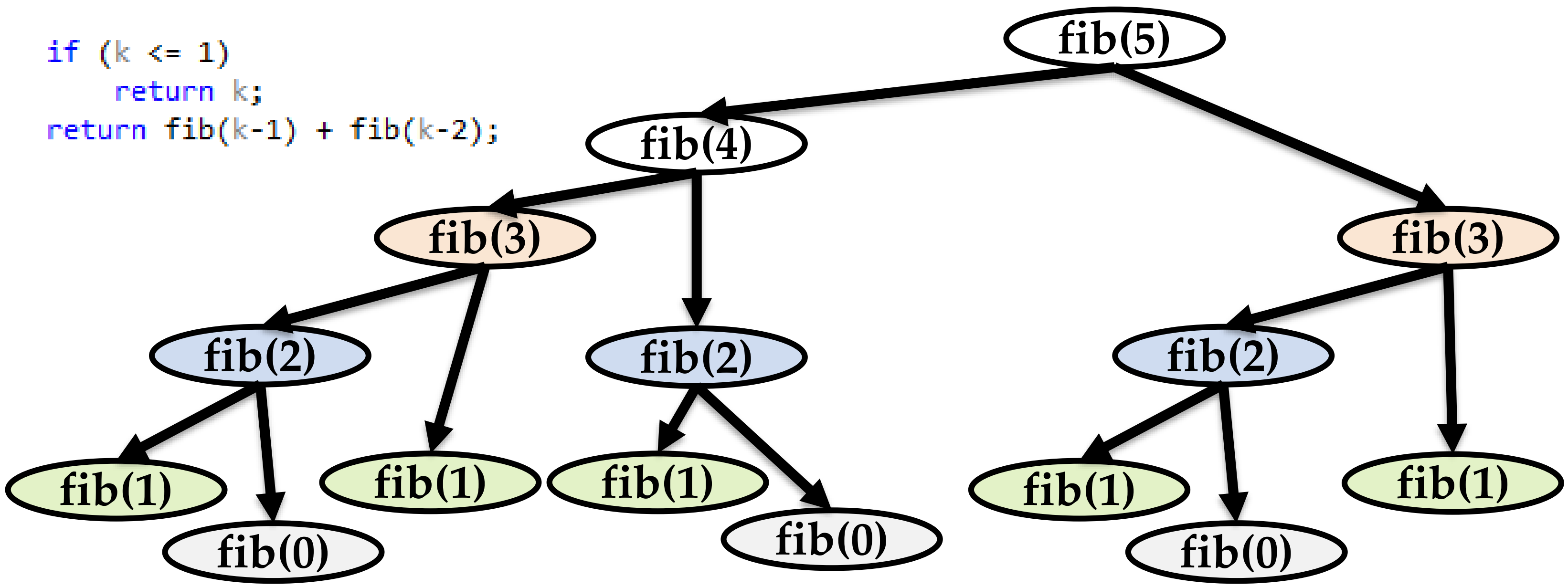
- Napr. **problém najkratšej u-v cesty v grafe <u>má</u> optimálnu podštruktúru**
  - Ak vrchol x leží na najkratšej u-v ceste, potom táto u-v cesta je spojenie najkratšej u-x cesty a x-v cesty.
- Naopak: **problém najdlhšej u-v cesty v grafe <u>nemá</u> optimálnu podštruktúru**:
  - Najdlhšia cesta q-t cesta (q,r,t) nie je kombinácia najdlhšej q-r cesty a r-t cesty. (najdlhšia q-r cesta je q,s,t,r).



DP – Prelínajúce sa podproblémy

- Optimálne riešenie problému je možné rozdeliť na menšie podproblémy, ktoré sa opakovane používajú viac krát, napr. rekurzívny výpočet Fibonacciho čísel:

```
// vypocitaj k-te fibonacciho cislo
int fib(int k)
{
    if (k <= 1)
        return k;
    return fib(k-1) + fib(k-2);
}
```



DP vs. Rozdeľuj a panuj (divide and conquer)

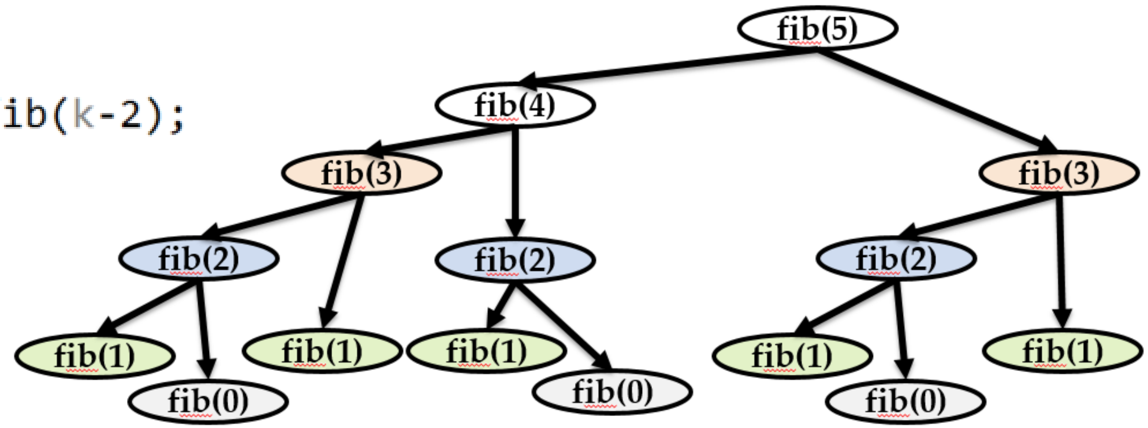
- Ak problém môžeme vyriešiť kombinovaním optimálnych riešení **neprelínajúcich sa** podproblémov:
- **Rozdeľuj a panuj** je algoritmická paradigma založená na rekurzii, ktorá sa môže rozvetvovať do viacerých vetiev:
  - **Rozdeľ** problém na dva alebo viac podproblémov, ktoré rekurzívne vyrieš (pokračuj až kým nebudú riešiteľné triviálne)
  - **Spoj** riešenia podproblémov do riešenia pôvodného problému
  - Napr. Usporadúvanie zlučovaním (Merge sort), Quick sort, Najbližšia dvojica bodov v rovine
- Zjednodušená verzia: Ak sa vnárame len do jednej vetvy **decrease and conquer** (zmenši a panuj)
  - Napr. binárne vyhľadávanie, Euklidov algoritmus (NSD)

Postupnosť riešenia podproblémov (1 z 3)

- Principálne sú tri varianty, ako môžeme pristupovať ku riešeniu podproblémov:

1. Neuvažujeme prelínanie podproblémov
   Naivný (rozdeľuj a panuj)

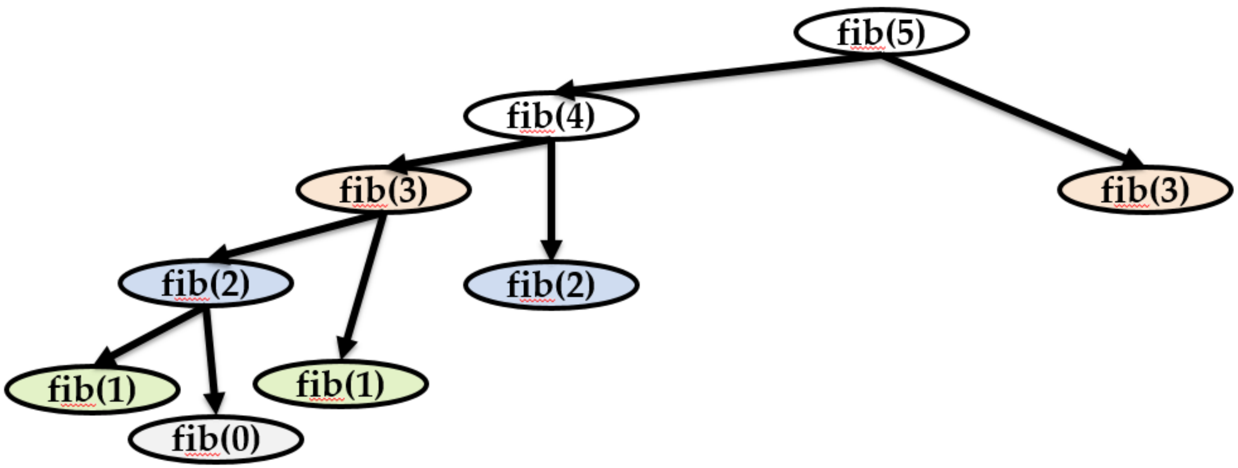
```
// divide-and-conquer
int fib(int k)
{
    if (k <= 1)
        return k;
    return fib(k-1) + fib(k-2);
}
```



Exponenciálna zložitosť $O(2^k)$

Postupnosť riešenia podproblémov (2 z 3)

2. Uvažujeme prelínanie podproblémov
   Zhora nadol (top-down)
   Zodpovedá výpočtu podľa rekurzívnej definície problému s priebežným pamätaním medzivýsledkov (memoization)



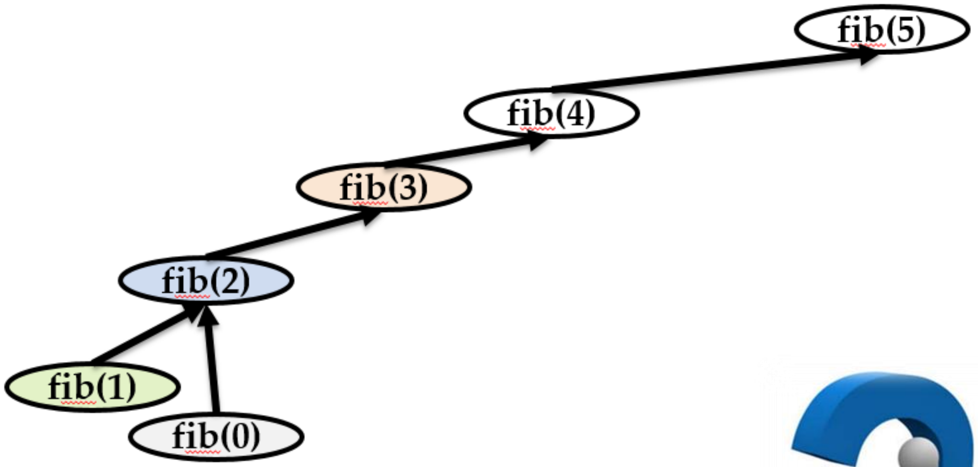

```
int f[100];

// top-down
int fib(int k)
{
    if (k <= 1)
        return k;
    if (f[k] > 0)
        return f[k];
    return f[k] = fib(k-1) + fib(k-2);
}
```

Čas / pamäť:
Lineárna zložitosť O(k)

Postupnosť riešenia podproblémov (3 z 3)

3. Uvažujeme prelínanie podproblémov
   Zdola nahor (bottom-up)
   Najskôr vyriešime (menšie) podproblémy, z ktorých potom zostrojíme riešenie (väčších) problémov

```
// bottom-up
int fib(int k)
{
    int i, a=0, b=1, next;
    if (k <= 1)
        return k;
    for (i = 0; i < k-1; i++)
    {
        next = a+b;
        a = b;
        b = next;
    }
    return b;
}
```



Čas: O(k)
Pamäť: O(1)

DP vs. Greedy (pažravé) algoritmy

- Greedy algoritmus sa v každom kroku rozhodne vybrať lokálne optimálne riešenie, a preto tento prístup **nemusí** viesť k (globálne) optimálnemu riešeniu
- Pre nejaký greedy algoritmus vždy vychádzame z toho, že to je „len“ aproximatívna heuristika – teda, že nemusí vrátiť správne riešenie, zaujíma nás:
  - nájsť kontrapríklad, v ktorom nenájde (globálne) optimálne riešenie
  - garancia nakoľko bude nájdené riešenie horšie ako (globálne) optimálne riešenie (aproximačný faktor $\boldsymbol{\rho}$ rho)
- Greedy algoritmus môže nájsť globálne optimálne riešenie v problémoch s optimálnou podštruktúrou, v ktorých vieme dokázať (globálnu) optimálnosť rozhodnutí v každom kroku
  - Napr. Kruskalov a Primov algoritmus pre najlacnejšiu kostru grafu

Úloha: Výdavok peňazí

- Daná je množina platidiel a suma, nájdite najmenší počet platidiel, ktorými môže pokladník túto sumu vyplatiť.
- Napr. Platidlá {1,4,5,15,20}, suma 23
  - **Greedy algoritmus** – zvoľ platidlo s najväčšou hodnotou neprevyšujúcou zostávajúcu sumu na vyplatenie

    Suma: 23 – platidlo 20

    Suma: 3 – platidlá 1, 1, 1

    Výsledok (štyri platidlá): **20+1+1+1**

Úloha: Výdavok peňazí (2)

- Daná je množina platidiel a suma, nájdite najmenší počet platidiel, ktorými môže pokladník túto sumu vyplatiť.
- Napr. Platidlá {1,4,5,15,20}, suma 23
  - **Dynamické programovanie** – pre každú menšiu sumu (podproblém) nájdi najmenší počet platidiel, ktorými je možné sumu vyplatiť; počet pre cieľovú sumu urči skombinovaním týchto výsledkov tak, že vyskúšaj použiť každé platidlo:

    Suma (počet): 22(3), 19(2), 18(4), 8(2), 3(3)

    Suma: 23 – platidlo 15

    Suma (počet): 7(3), 4(1), 3(3)

    Suma: 8 – platidlo 4

    Suma: 4 – platidlo 4

    Výsledok (tri platidlá): **15+4+4**

Úloha: Výdavok peňazí (3)

- Daná je množina platidiel a suma, nájdite najmenší počet platidiel, ktorými môže pokladník túto sumu vyplatiť.
- Greedy algoritmus určí správny výsledok ak pre všetky platidlá platí, že hodnota najbližšieho vyššieho platidla je aspoň dva krát vyššia
- Napr. Platidlá {1, 2, 5, 10} alebo {1, 2, 4, 8}
- Naopak: pre platidlá {1, 3, 4}
  sumu 6 vyplatíme ako 4+1+1
  pričom optimálne je 3+3 (dve platidlá)

Kroky k riešeniu dynamickým programovaním

- Pre dané zadanie úlohy:
    1. Definovať podproblémy
    2. Vyjadriť rekurentný vzťah medzi týmito podproblémami
    3. Vyriešiť základné prípady

- Identifikovať typ dynamického programovania:
    - 1D, 2D, ...
    - Intervalové
    - Stromové (výpočet nad štruktúrou stromu)
    - Výpočet nad (všetkými) podmnožinami

1D dynamické programovanie – Ukážka

- Podproblémy reprezentujeme v jednom rozmere
- Ukážková úloha:
  Dané je číslo N, nájdite **počet** spôsobov koľkými to môžeme zapísať ako súčet čísel 1, 3 a 4.
- Napr. pre N=5, výsledok je 6:

$$
\begin{aligned}
5 &= 1 + 1 + 1 + 1 + 1 \\
&= 1 + 1 + 3 \\
&= 1 + 3 + 1 \\
&= 3 + 1 + 1 \\
&= 1 + 4 \\
&= 4 + 1
\end{aligned}
$$

1D dynamické programovanie – Ukážka (2)

- Definovať podproblémy:
  - Označíme D_n počet spôsobov, koľkými môžeme n zapísať ako súčet 1, 3 a 4.
- Vyjadriť rekurentný vzťah medzi týmito podproblémami:
  - Uvažujme jedno možné vyjadrenie: $n = x_1 + x_2 + ... + x_{m-1} + x_m$
  - Ak $x_m = 1$, tak zvyšok musí mať súčet $n-1= x_1 + x_2 + ... + x_{m-1}$ preto, počet súčtov, ktoré končia $x_m=1$ je D_{n-1}
  - Analogicky pre $x_m = 3$ a $x_m = 4$.
  - Rekurentný vzťah je teda: $D_n = D_{n-1} + D_{n-3} + D_{n-4}$
- Vyriešiť základné prípady:
  - $D_0 = 1, D_n = 0$ pre $n < 0$, alebo
  - $D_0 = D_1 = D_2 = 1, D_3 = 2$

1D dynamické programovanie – Ukážka (3)

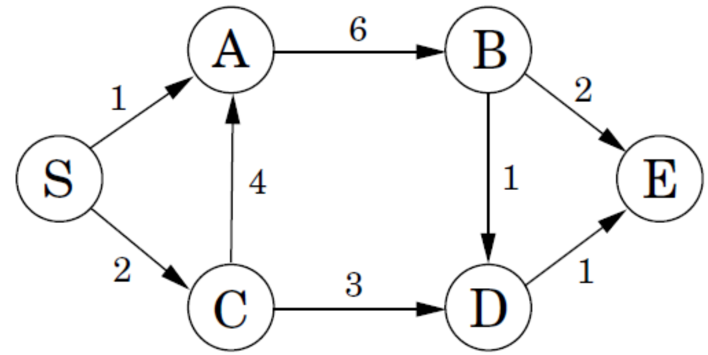
- Implementácia:

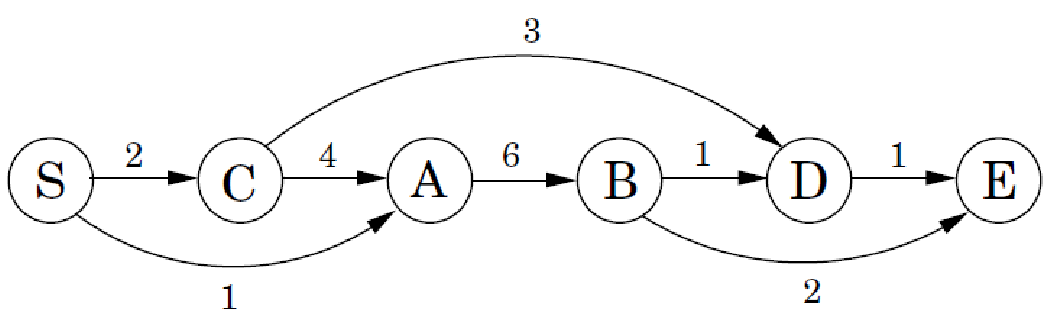
```
int d[100];

int pocet_suctov(int n)
{
    int i, d[100];
    d[0] = d[1] = d[2] = 1; d[3] = 2;
    for (i = 4; i <= n; i++)
        d[i] = d[i-1] + d[i-3] + d[i-4];
}
```

- Na zamyslenie:
  ako urýchliť tento výpočet pre veľké n? napr. n = 10^{20}

Dynamické programovanie ako cesta v DAGu

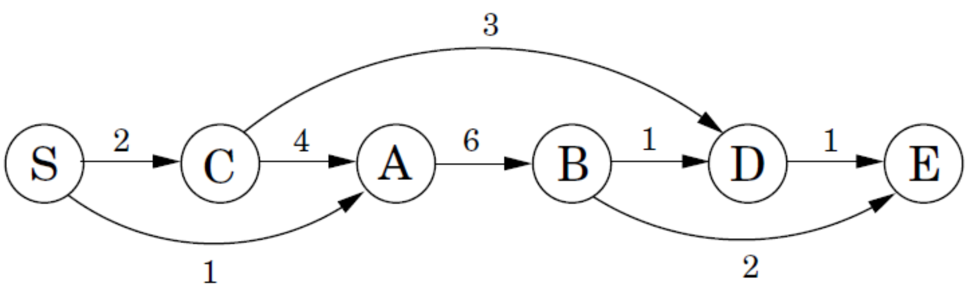
- Orientovaný acyklický graf (DAG)
- Nájsť v ňom najkratšiu cestu je veľmi jednoduché
- Prečo?
- lebo je možné ho linearizovať: vrcholy usporiadať do postupnosti tak, že hrany smerujú len do vrcholov neskôr v postupnosti (zľava doprava) (topologické usporiadanie)





Dynamické programovanie ako cesta v DAGu (2)

- Pre určenie najkratšej cesty (do vrcholu) stačí uvažovať najkratšie cesty do niektorého z predchádzajúcich susedov vrcholu
  Napr. najkratšia cesta do D, vedie alebo z B alebo z C:
  dist(D) = min{dist(B) + 1, dist(C) + 3}

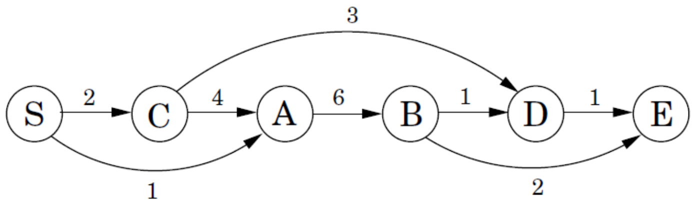



```
dist[v] = ∞, pre v ∈ V(G)
dist[s] = 0
foreach (v ∈ V(G)−{s} v topologickom usporiadaní) do
  dist[v] = min{dist[u] + w(e_{u,v}) : e_{u,v} ∈ E(G)}
```

Dynamické programovanie ako cesta v DAGu (3)

```
dist[v] = ∞, pre v ∈ V(G)
dist[s] = 0
foreach (v ∈ V(G)−{s} v topologickom usporiadaní) do
  dist[v] = min{dist[u] + w(e_{u,v}) : e_{u,v} ∈ E(G)}
```

- Množina podproblémov: {dist[u] : u ∈ V(G) }
- Základné prípady: dist[s] = 0



- Postupujeme od najmenšieho (základného) podproblému k „väčším" podproblémom (vrcholom, ktoré sú ďalej v topologickom usporiadaní)
- **Dynamické programovanie – implicitný DAG:**
  - **Vrcholy sú podproblémy**
  - **Hrany sú závislosti medzi podproblémami**

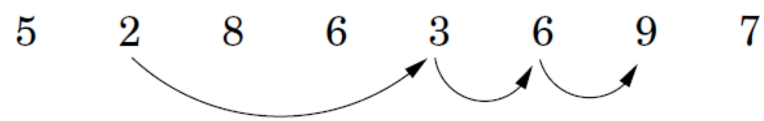
Najdlhšia vybraná rastúca podpostupnosť

- Daná je postupnosť N čísel $a_1, a_2, \ldots, a_N$. Podpostupnosť dĺžky K <u>vybraná</u> z danej postupnosti je podmnožina čísel so zachovaním poradia tvaru a_{i1}, a_{i2}, … a_{iK}, kde $1 \leq i1 < i2 < \ldots < iK \leq N$. Úloha je nájsť takúto rastúcu podpostupnosť najväčšej dĺžky.

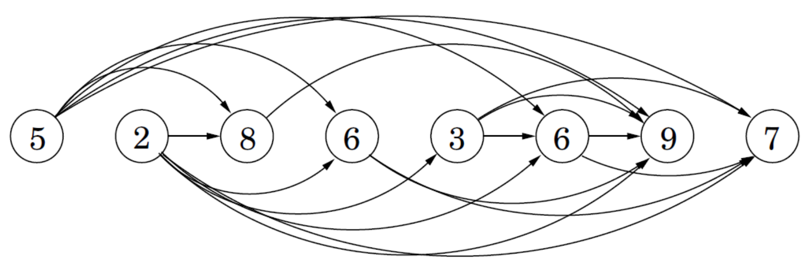
- Napr. pre čísla 5, 2, 8, 6, 3, 6, 9, 7 to je 2, 3, 6, 9:

$$5 \quad 2 \quad 8 \quad 6 \quad 3 \quad 6 \quad 9 \quad 7$$

Najdlhšia vybraná rastúca podpostupnosť (2)

- Napr. 

- Vytvorme graf možných prechodov v nejakom možnom riešení (vybranej rastúcej podpostupnosti):

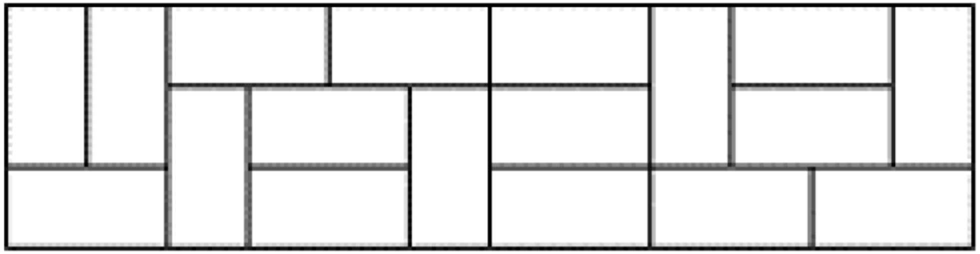


- Nájdeme najdlhšiu cestu (v tomto DAGu):

```
for (j = 1,2, ..., N) do
  L[j] = 1 + max{L[i] : e_{i,j} ∈ E(G)}
return max{L[j]:j ∈ V(G)}
```

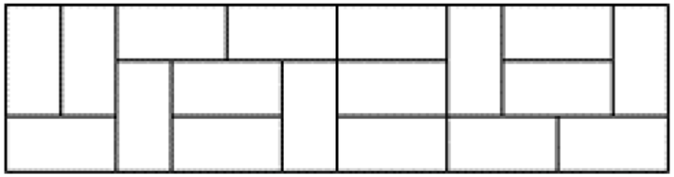
Kachličkovanie

- Dané je N, nájdite počet spôsobov, koľkými je možné vykachličkovať chodbu 3xN kachličkami veľkosti 2x1. Napr. pre N=12 jedno možné riešenie:



- Ideme nato...
- Definovať podproblémy:
  - Označíme D_n počet spôsobov, koľkými môžeme vykachličkovať chodbu veľkosti 3xN.
- Rekurentný vzťah medzi týmito podproblémami:
  - Nedá sa ...

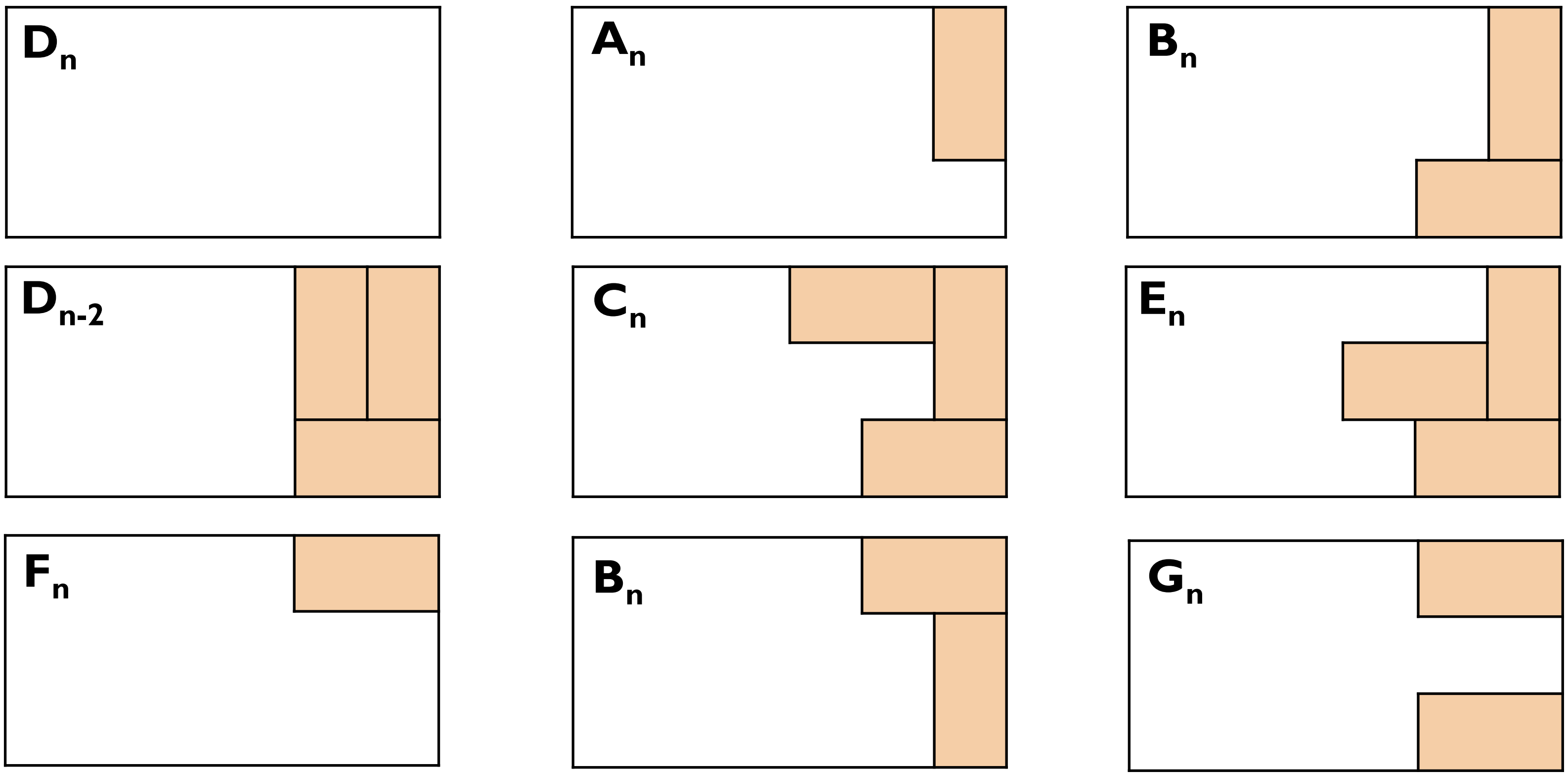
Kachličkovanie (2)



- Definovať podproblémy:
  - Označíme D_n počet spôsobov, koľkými môžeme vykachličkovať chodbu veľkosti 3xN.
- Rekurentný vzťah medzi týmito podproblémami:
  - Nedá sa vyjadriť jednoducho, pretože hodnoty D_n nie sú v jednoduchom vzájomnom vzťahu (priložením jednej kachličky nedostaneme chodbu 3xM pre M < N).
- Všimnime si, že potrebujeme také podproblémy, ktoré majú **jednoduchý** vzájomný vzťah.
  Jednoduchý = reprezentuje jeden krok riešenia úlohy
- Začneme teda uvažovaním, čo sa stane, keď na chodbe 3xN priložíme <u>jednu</u> kachličku.

Kachličkovanie (3)

- Začneme teda uvažovaním, čo sa stane, keď na chodbe 3xN priložíme <u>jednu</u> kachličku:



- Rekurentný vzťah: skúsim priložiť jednu kachličku sprava

2D dynamické programovanie – LCS

- Dané sú dva reťazce znakov x a y, nájdi dĺžku najdlhšej spoločnej vybranej podpostupnosti (LCS) oboch reťazcov.
- Napr.
  x : A**BC**BD**AB**
  y : **B**D**CAB**C
  Najdlhšia je BCAB, dĺžka 4.

2D dynamické programovanie – LCS

- Dané sú dva reťazce znakov x a y, nájdi dĺžku najdlhšej spoločnej vybranej podpostupnosti (LCS) oboch reťazcov.
- Definovať podproblémy:
  - Označíme $D_{i,j}$ dĺžku LCS reťazcov $x_{1,i}$ a $y_{1,j}$
- Rekurentný vzťah medzi podproblémami:
  - Ak $x_i = y_j$ tak oba prispievajú do LCS:
    $D_{i,j} = D_{i-1,j-1} + 1$
  - Inak, alebo x_i alebo y_j neprispieva do LCS,
    teda jedno môžeme odstrániť:
    $D_{i,j} = \max\{D_{i-1,j} , D_{i,j-1}\}$
- Základné prípady:
  - $D_{i,0} = D_{0,j} = 0$

2D dynamické programovanie – LCS

- Implementácia je priamy prepis:

```
for(i = 0; i <= n; i++)
  D[i][0] = 0;
for(j = 0; j <= m; j++)
  D[0][j] = 0;

for(i = 1; i <= n; i++)
  for(j = 1; j <= m; j++)
    if(x[i] == y[j])
      D[i][j] = D[i-1][j-1] + 1;
     else
      D[i][j] = max(D[i-1][j], D[i][j-1]);
```

- Zložitosť: O(nm)
- Ako zistím konkrétnu najdlhšiu postupnosť?
  - Spätné hrany (ako pri hľadaní v DAGu)

LCS – pamäťová zložitosť

- O(nm)
- Ako by sa to dalo zlepšiť?
- Pri výpočte si potrebujeme pamätať len predposledný a posledný riadok: teda zložitosť môžeme znížiť na O(m)
- **Ako potom nájdeme konkrétnu postupnosť?** (ak si pamätáme len posledné dva riadky výpočtu, tak nevieme spätne vystopovať rozhodnutia – neviem spätne určiť tú konkrétnu najdlhšiu spoločnú podpostupnosť)

LCS – pamäťová zložitosť (2)

- Ako vyzerá DAG pre túto úlohu

		A	B	C	D	A
	0	0	0	0	0	0
A	0	1	1	1	1	1
C	0	1	1	2	2	2
B	0	1	2	2	2	2
D	0	1	2	2	3	3
E	0	1	2	2	3	3
A	0	1	2	2	3	4

LCS - "ACDA"

- Ak si zapamätáme len posledné dva riadky, nevieme zrekonštruovať riešenie...

LCS – Hirschbergov algoritmus

- Každé riešenie prechádza stredovým riadkom: políčkom (n/2,**k**) pre nejaké **k**

		A	B	C	D	A
	0	0	0	0	0	0
A	0	1	1	1	1	1
C	0	1	1	2	2	2
B	0	1	2	2	2	2
D	0	1	2	2	3	3
E	0	1	2	2	3	3
A	0	1	2	2	3	4

LCS - "ACDA"

- Modifikujeme algoritmus, tak, aby okrem dĺžky vrátil aj **k** (teda cez ktoré políčko v strednom riadku prechádzalo)

LCS – Hirschbergov algoritmus (2)

- Každé riešenie prechádza stredovým riadkom: políčkom (n/2,**k**) pre nejaké **k**
- Modifikujeme algoritmus, tak, aby okrem dĺžky vrátil aj **k** (teda cez ktoré políčko v strednom riadku prechádzalo)
- Takouto modifikáciou dokážeme určiť konkrétne riešenie v čase O(nm) za použitia O(m) pamäte.
  - V prvom kroku riešime problém veľkosti n x m, nájdeme k: zložitosť T(n*m) = c*n*m operácií
  - V druhom kroku riešime problémy n/2 x **k** a n/2 x (m-**k**), čiže časová zložitosť T(n/2 * **k** + n/2 * (m-**k**)) = T(n/2 * m)
  - V treťom kroku zložitosť: T(n/4 * m), atď.
  - Celkovo: T(m * (n+n/2+n/4+…+1)) = T(m * 2n) = **O(nm) čas**

Problém batohu (Knapsack)

- Počas zát'ahu nájde zlodej viac proviantu, ako očakával. Batoh odnesie najviac W kg, môže si vybrat' z N vecí (pre každú vieme cenu v_i a hmotnost' w_i) Zlodej chce v batohu odniest' čo najcennejšie veci.
- Nosnost' batohu W. Dané máme N predmetov, pre každý: v_i cenu (value) a hmotnost' w_i: (weight)
- Úloha:

maximalizuj: $$\sum_{i \in T} v_i$$

aby sa zmestili do batohu: $$\sum_{i \in T} w_i \leq W.$$

Vo všeobecnosti je tento problém v NP.

Problém batohu – Variant s opakovaním

- Predpokladajme, že **zlodej je v supermarkete** a môže **z každej veci zobrať neobmedzené množstvo**.
- Dynamické programovanie – spôsob riešenia cez menšie podproblémy:
- Zmenšiť môžeme alebo **nosnosť batohu** (riešime úlohu pre $w \leq W$) alebo **počet predmetov** (riešime úlohu pre predmety 1, 2, ..., j pre $j \leq N$)
- Uvažujme prvú možnosť, podproblémy:
  K(w) = najväčšia hodnota predmetov, ktoré môžeme odniesť v batohu s nosnosťou w

Problém batohu – Variant s opakovaním (2)

- Definovať podproblémy:
  K(w) = najväčšia hodnota predmetov, ktoré môžeme odniesť v batohu s nosnosťou w
- Rekurentný vzťah medzi podproblémami:
  Ak optimálne riešenie pre K(w) obsahuje predmet i, tak odstránením predmetu z batohu dostaneme optimálne riešenie K(w-w_i)+v_i
  - **platí pre nejaké i, my ho nepoznáme, preto vyskúšame všetky možnosti:**

$$K(w) = \max_{i:w_i \leq w} \{K(w - w_i) + v_i\}$$

- Základný prípad: **Nosnosť batohu 0: K(0) = 0**

Problém batohu – Variant s opakovaním (3)

- Implementácia je priamy prepis:

```
K(0) = 0
for (w = 1,…,W) do
  K(w) = max{K(w − wᵢ) + vᵢ : wᵢ ≤ w}
return K(W)
```

- Vypĺňame jednorozmernú tabuľku dĺžky W+1 zľava doprava
- Zodpovedá to DAGu (medzi podproblémami), úloha je hľadanie najdlhšej cesty. Skúste si to nakresliť!
- Zložitosť: Pre každú položku skúšame všetky predmety: O(N)
- Celkový čas O(NW) – pseudopolynomiálna zložitosť:
  - Polynomiálna vzhľadom na číselnú hodnotu na vstupe
  - Exponenciálna vzhľadom na veľkosť zápisu (počet bitov)

Problém batohu – Variant bez opakovania

- **Každý predmet môže zlodej zobrať najviac raz**.
- Takéto podproblémy K(w) sú tuto nepoužiteľné:
  - Pretože, ak je aj hodnota $K(w - w_i)$ vysoká, nevieme z toho určiť či i-ty predmet už je použitý v tomto riešení alebo nie...
  - Musíme teda zjemniť definíciu podproblémov, aby zahŕňala informáciu o tom, ktoré predmety sú použité:
  - Označme K(w,j) = **najväčšia hodnota, ktoré môžeme odniesť v batohu s nosnosťou w, vybratím niektorých spomedzi predmetov 1, 2, ..., j**
  - Hľadáme hodnotu K(W,N).

Problém batohu – Variant bez opakovania (2)

- Definícia podproblémov:
  - Označme K(w,j) = **najväčšia hodnota, ktoré môžeme odniesť v batohu s nosnosťou w, vybratím niektorých spomedzi predmetov 1, 2, ..., j**
  - Hľadáme hodnotu K(W,N).
- Rekurentný vzťah medzi podproblémami:
  - **Predmet j je alebo potrebný v optimálnom riešení alebo nie je potrebný:**

$$K(w,j) = \max\{K(w-w_j, j-1) + v_j, K(w, j-1)\}$$

- Základné prípady:
  - **Žiadne predmety: K(w,0)=0 pre w=1,2,...,W**
  - **Nosnosť batohu 0: K(0,j) = 0 pre j=1,2,...,N**

Problém batohu – Variant bez opakovania (3)

- Implementácia je priamy prepis:

```
K(w,0) = K(0,j) = 0 pre všetky w a j
for (j = 1,…,N) do
  for (w = 1,…,W) do
    if (wj > w) then
      K(w,j) = K(w,j-1)
     else
      K(w,j) = max{K(w,j-1), K(w-wj,j-1)+vj}
return K(W,N)
```

- Vypĺňame dvojrozmernú tabuľku:
  W+1 riadkov, N+1 stĺpcov
- Každé políčko trvá O(1)
- Celková zložitosť: O(NW)

Intervalové dynamické programovanie

- Najkratšie doplnenie na palindróm:
  Daný je reťazec $x = x_{1..N}$, nájdi najmenší počet znakov, ktoré je potrebné pridať, aby vznikol z reťazca x palindróm.
- Napr.
  x: Ab3bd
  vložením dvoch znakov môže vzniknúť
  dAb3b**A**d alebo A**d**b3bd**A**

Najkratšie doplnenie na palindróm

- Definovať podproblémy:
  - Označme D_{ij} najmenší počet znakov, ktoré je potrebné pridať do reťazca $x_{i..j}$, aby bol palindróm
- Rekurentný vzťah medzi podproblémami:
  - Uvažujme najkratší palindróm $y_{1..k}$, ktorý obsahuje $x_{i..j}$
  - Platí $y_1 = x_i$ alebo $y_k = x_j$ (prečo?)
  - Kratší palindróm $y_{2..k-1}$ je optimálne riešenie pre $x_{i..j-1}$ alebo $x_{i+1..j}$ alebo $x_{i+1..j-1}$ (ak $y_1 = x_i$ a zároveň $y_k = x_j$)

$$D_{ij} = \begin{cases} 1 + \min\{D_{i+1,j}, D_{i,j-1}\} & x_i \neq x_j \\ D_{i+1,j-1} & x_i = x_j \end{cases}$$

- Základné prípady: $D_{ii} = D_{i,i-1} = 0$ pre všetky $i = 1, ..., N$

Najkratšie doplnenie na palindróm (2)

- Označme D_{ij} najmenší počet znakov, ktoré je potrebné pridať do reťazca $x_{i..j}$, aby bol palindróm

$$D_{ij} = \begin{cases} 1 + \min\{D_{i+1,j}, D_{i,j-1}\} & x_i \neq x_j \\ D_{i+1,j-1} & x_i = x_j \end{cases}$$

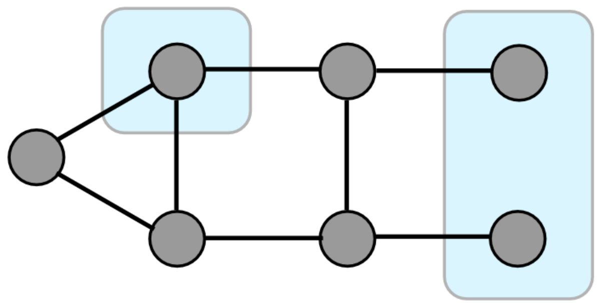
- Základné prípady: $D_{ii} = D_{i,i-1} = 0$ pre všetky i = 1, ..., N
- Ako vypĺňať položky D?
  - **Máme tam aj väčšie indexy i+1 aj menšie (j-1) indexy!**
  - **To môže byť vážna prekážka! Preto:**
    Hodnoty D musia byť vypĺňané v rastúcom poradí j-i
- Implementácia ako cvičenie…

Najkratšie doplnenie na palindróm (3)

- Alternatívne riešenie (pre Najkratšie doplnenie na palindróm): Uvažujme otočené slovo x^R
  Výsledok je: $N – LCS(x, x^R)$.
- Prečo?
  Skúste si nakresliť.

Maximálna nezávislá množina

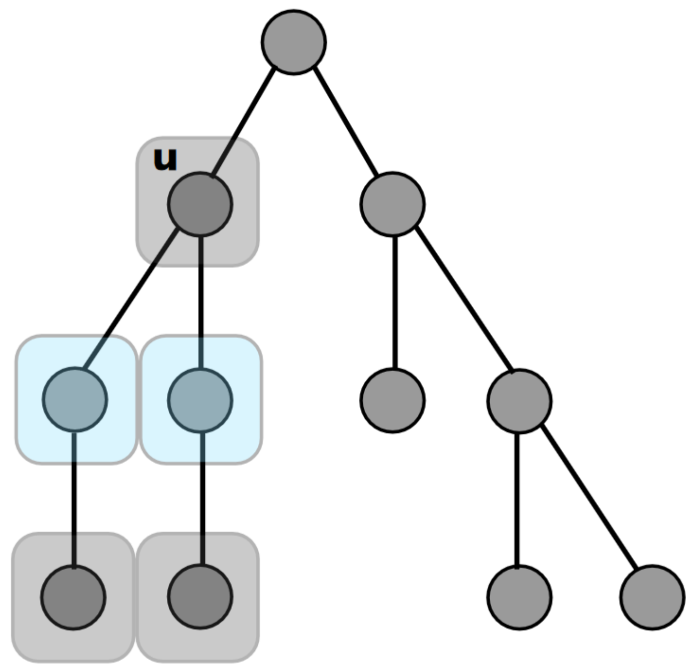
- Nezávislá množina je taká podmnožina vrcholov grafu, že medzi nimi nejde hrana.



- Vo všeobecných grafoch je to NP
- Skúsme to vyriešiť na stromoch...

Maximálna nezávislá množina stromu

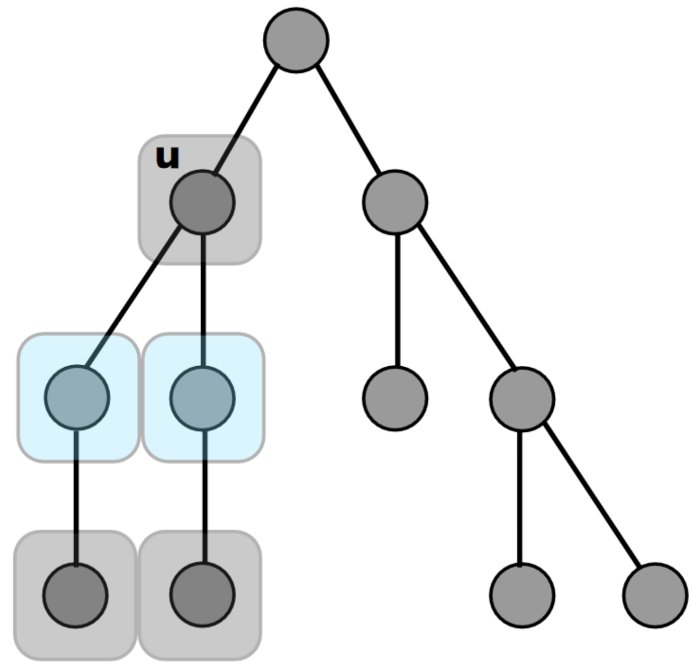
- Pre daný strom ofarbi čo najviac vrcholov na čierno tak, aby susedné (spojené hranou) nemali rovnakú farbu
- Hľadáme maximálnu nezávislú množinu stromu
- Uvažujme nejaký vrchol u
  Nastávajú dva prípady:
  - Zahrnieme u, ale nezahrnieme jeho priamych nasledovníkov
  - Do maximálnej nezávislej množiny nezahrnieme u



Maximálna nezávislá množina stromu (2)

- Definovať podproblémy:
  - Označíme I(u) veľkosť maximálnej nezávislej množiny podstromu s koreňom u
  - Hľadáme hodnotu I(root)
- Rekurzívny vzťah:
  - Zahrnieme u, ale nemôžeme deti (zahrnieme pradeti)
  - Nezahrnieme u, môžeme deti u

$$I(u) = \max\left\{1 + \sum_{\text{pradeti w vrcholu u}} I(w),\ \sum_{\text{deti w vrcholu u}} I(w)\right\}$$



- Základné prípady: listové vrcholy: I(list) = 1

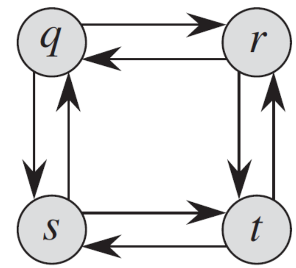
Zložitosť: O(N) každú hranu preskúmame najviac dva krát

Maximálna nezávislá množina stromu (3)

- Definovať podproblémy (alternatíva):
  - Označíme B_v optimálne riešenie pre podstrom s koreňom v, ktorý zafarbíme na čierno
  - Označíme W_v optimálne riešenie pre podstrom s koreňom v, ktorý nezafarbíme na čierno
  - Hľadáme $\max\{B_r, W_r\}$, kde r je koreň stromu (ľubovoľný)
- Rekurzívny vzťah:
  - Keď pre vrchol v zvolíme farbu (čiernu/bielu), tak podstromy môžeme riešiť ďalej už nezávislo
  - Ak v zafarbíme na čierno, deti nemôžeme: $B_v = 1 + \sum_{u \in \mathrm{children}(v)} W_u$
  - Ak v nezafarbíme, tak deti môžu mať ľubovoľnú farbu: $W_v = 1 + \sum_{u \in \mathrm{children}(v)} \max\{B_u, W_u\}$
- Základné prípady: listy

Najdlhšia cesta medzi každou dvojicou vrcholov

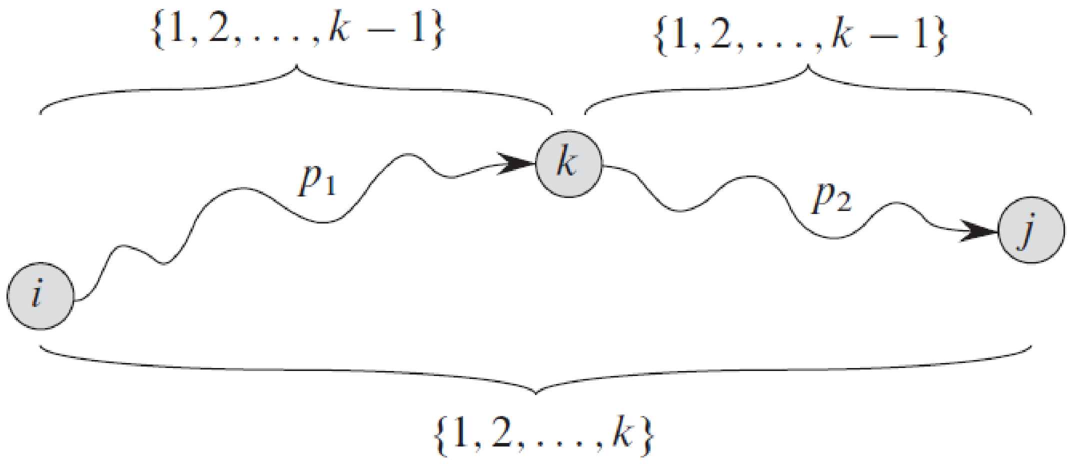
- **Problém najdlhšej u-v cesty v grafe <u>nemá</u> optimálnu podštruktúru**:
  - Najdlhšia cesta q-t cesta (q,r,t) nie je kombinácia najdlhšej q-r cesty a r-t cesty. (najdlhšia q-r cesta je q,s,t,r).



- Skúsme sa rozpamätať ako pracuje Floyd-Warshallov algoritmus pre najkratšie cesty medzi každou dvojicou a premyslieť, prečo nie je dobrý...

Floydov algoritmus

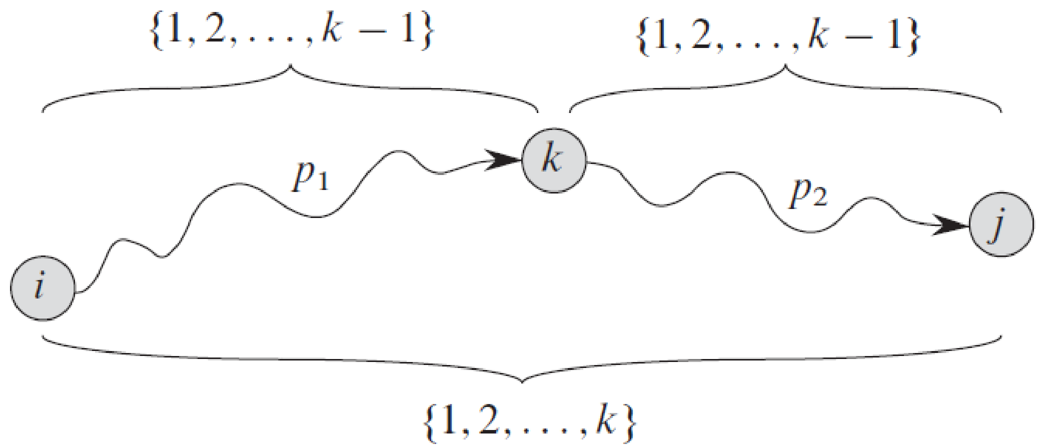
- Dĺžky najkratších ciest v kroku k určíme induktívne z dĺžok najkratších ciest z kroku k-1 nasledovne:



- Ak vrchol k **nie je** na najkratšej ceste idúcej cez vrcholy {1,...,k}, tak sa použije najkratšia cesta idúca cez {1,...,k-1}
- Ak vrchol k **je** na najkratšej ceste idúcej cez {1,...,k}, a podcesty p_1 a p_2 už môžu obsahovať len {1,...,k-1}.

Najdlhšia cesta – môžeme spojiť cesty?

- (Opakovanie: cesta je sled s neopakujúcimi sa vrcholmi)
- Najdlhšia i-k cesta, môže používať niektoré vrcholy, ktoré sú na najdlhšej k-j ceste, a teda **najdlhšia i-j cesta teda nie je len „obyčajné“ spojenie najdlhších ciest i-k a k-j** ...



- **Vždy** môžeme vykľučkovať z tohto problému tak, že do definície podproblému zahrnieme viac informácií
- Ako?

Najdlhšia cesta – môžeme spojiť cesty?

- **Vždy** môžeme vykľučkovať z tohto problému tak, že do definície podproblému **zahrnieme viac informácií**
- **Nebudeme hľadať riešenie problému $D_{i,j}$ = dĺžka najdlhšej i-j cesty (pre všetky dvojice i a j), ale:**

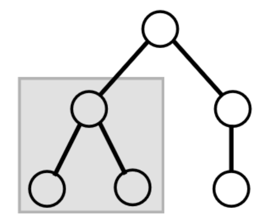
  budeme hľadať riešenie problému pre i, j a S:
  $D_{i,j,S}$ = dĺžka najdlhšej i-j cesty, ktorá používa len vrcholy z množiny S
- Teraz už môžeme kombináciou pre riešenia menších podproblémov zostrojiť riešenie pre väčší problém: **pretože vieme zaručiť, že budeme spájať len také cesty, ktoré nepoužívajú rovnaké vrcholy**

DP nad podmnožinami – Najdlhšia cesta

- Definícia podproblémov:
  - **$D_{i,j,S}$ = dĺžka najdlhšej i-j cesty, ktorá používa len vrcholy z množiny S**
- Rekurentný vzťah podproblémov:
  - **$D_{i,j,S} = \max \{D_{u,x,A} + D_{x,v,B}\}$ pre všetky $x \in S$, a všetky partície $S-\{x\} = A \cup B$, $A \cap B = \emptyset$.**
- Základné prípady: **$D_{v,v,\{v\}} = 0$**
- Zložitosť: počet podproblémov ($N^2 2^N$) je veľký, navyše pre každý z nich musíme nájsť všetky možné partície a skúsiť každé x... je to veľa!
- Ale vážne – nie je to až tak veľa: je to **rýchlejšie ako N!**

Podproblémy pre rôzne typické DP

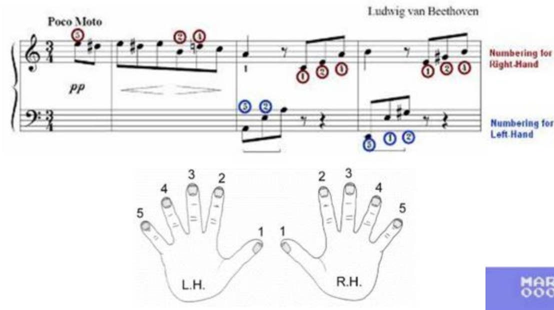
- **1D**: Vstup je $x_1, x_2, \ldots, x_N$: podproblém je $x_1, x_2, ..., x_i$
  počet podproblémov je lineárny
- **2D**: Vstup je $x_1, x_2, \ldots, x_N$ a $y_1, y_2, \ldots, y_M$:
  podproblém je $x_1, x_2, ..., x_i$ a $y_1, y_2, ..., y_j$
  počet podproblémov je N*M
- **Interval**: Vstup je $x_1, x_2, \ldots, x_N$: podproblém je $x_i, x_{i+1}, ..., x_j$
  počet podproblémov je $O(N^2)$
- **Stromové**: Vstup je (zakorenený) strom:
  podproblém je zakorenený podstrom.
  počet podproblémov N (počet vrcholov)



- **Podmnožiny**: Vstup je N prvkov: podproblém
  je podmnožina vstupu. počet podproblémov 2^N

Ďalšie problémy pre DP

- Prstoklad na klavíri:
  ktorými prstami zahrať
  ktoré noty, aby to bolo čo
  najmenej náročné



- Plošinovky: maximalizuj bodový zisk pre daný level (mapa, objekty, akcie)

- Zarovnanie textu pre tlač (TeX)
  - Minimalizácia štvorcov dĺžok voľného miesta na konci riadkov
- Rozloženie elementov na webovú stránku
- Balenie množiny produktov do balíčkov
- Rozpoznávanie hovorené slova:
  Viterbiho algoritmus: najpravdepodobnejší prechod
  pravdepodobnostnom modeli