7. prednáška (24.3.2025)

Dynamické programovanie

alebo

Bol raz jeden problém...

Čo už vieme

- Asymptotická **časová zložitosť** algoritmov
- Základné **údajové štruktúry**:
  - spájaný zoznam, strom, zásobník, rad
  - možnosti ich efektívnej implementácie
  - príklady ich aplikácie pri riešení rôznych úloh
- **Rekurzia** ako dôležitý programovací koncept na zápis algoritmov
- **Metódy** riešenia problémov (=úloh):
  - rozdeľuj a panuj
  - backtracking

Divide et impera

- **Rozdeľuj a panuj !**
  - armádna, politická a ekonomická stratégia
  - jedna zo základných stratégií riešenia problémov v informatike

- Princíp stratégie:
  - **rozdeliť** problém **na podproblémy**
  - vyriešiť podproblémy
  - ak treba, tak **z riešení podproblémov vytvoriť riešenie** pre pôvodný problém

Rozdeľuj a panuj

- Stratégiu **rozdeľuj a panuj** sme už používali:
  - QuickSort
    - pivotizácia **rozdelí problém** (postupnosť) na dva menšie problémy (postupnosti)
    - problémy vyriešime, ich vyriešením máme riešenie pre pôvodný problém
  - MergeSort
    - **rozdelíme problém** (postupnosť) na dva menšie podproblémy a vyriešime ich
    - z **riešení podproblémov vytvoríme riešenie** problému pomocou algoritmu na zlúčenie dvoch utriedených (usporiadaných) postupností

Rozdeľuj a panuj

- **Schéma** (ešte raz):
  - rozdeľ problém na menšie (jednoduchšie) podproblémy
  - vyrieš podproblémy
  - z riešení podproblémov vytvor riešenie pre pôvodný problém (skombinovanie riešení)
- **QuickSort**
  - zložité rozdeľovanie, jednoduché kombinovanie riešení
- **MergeSort**
  - jednoduché rozdeľovanie, zložité kombinovanie riešení

Rozdeľuj a panuj

- Ďalšie príklady, ktoré už poznáme:
  - konštrukcia stromu z **inorder** a **preorder** postupnosti
  - konštrukcia stromu zo zátvorkového popisu *((1) 3 (2))*

- Ďalšie úlohy:
  - vyhodnotenie aritmetického výrazu
  - nájdenie mediánu


BANK

Čo robia v banke?

- Banky môžu emitovať **dlhopisy** (cenné papiere)
  - dlhopis má svoju **nominálnu hodnotu** (napr.100 €)
  - každý druh emitovaného dlhopisu musí schváliť CDCP (centrálny depozitár CP) – banka necháva schvaľovať malý počet druhov dlhopisov
- **Príklad:** Banka sa rozhodne emitovať 3 druhy dlhopisov s nominálnymi hodnotami 100 €, 500 €, 800 €, t.j., musí požiadať CDCP o schválenie 3 druhov CP

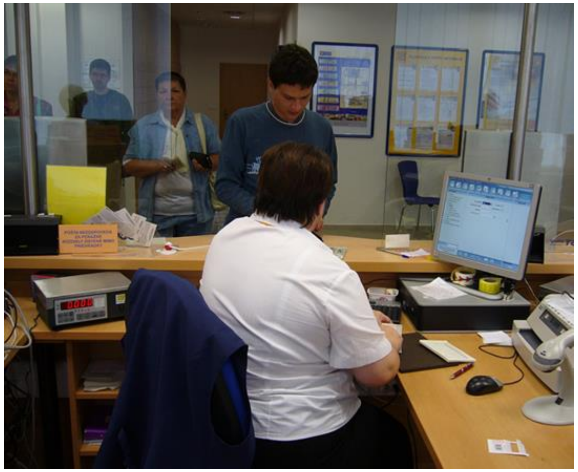
Šetrenie (Ako bojovať s krízou)

- Klient pri kúpe dostane od banky dlhopisy (cenné papiere) na špeciálnom papieri
- Špeciálny papier je **drahý**

- **Problém:**
  - Ak klient chce nakúpiť dlhopisy za 1000€, dlhopisy v akej nominálnej hodnote a v akom počte vydať, aby sa použilo **čo najmenej papiera**? (banka chce **ušetriť**, klient nechce skladovať veľa „papiera“)
  - Je vôbec možné kúpiť dlhopisy v požadovanej hodnote?

Príklad

- Emitované dlhopisy: 100€, 500€, 800€
- Suma: 1300 €
  - **2 x 500€, 3 x 100€** - 5 papierov
  - **1 x 800€, 1 x 500€** - 2 papiere
- Suma: 250 €
  - nemožno kúpiť cenné papiere



Potrebujeme softvér, ktorý nám pre zadanú sumu povie, či ju možno previesť na dlhopisy, a ak áno, tak povie, dlhopisy s akými hodnotami použiť.

IT oddelenie

„Greedy“ stratégie

- **„greedy“ stratégie** (založené na najlepšom okamžitom rozhodnutí) **nefungujú**:
  - kým sa dá, vyberať papier s najvyššou nominálnou hodnotou
    - emitované papiere: 100€, 500€, 800€
    - 1100€ = 800€ + 100€ + 100€ + 100€
    - 1100€ = 500€ + 500€ + 100€
  - nájsť papier s najvyššou nominálnou hodnotou, ktorá delí požadovanú sumu
    - emitované papiere: 100€, 500€, 800€
    - 1100€ = 11 x 100€

Formalizácia problému

- **n** typov dlhopisov s nominálnymi hodnotami: **h[0], h[1], …, h[n-1]**
- Zovšeobecnené označenie:
  - **R[m]** – minimálny počet dlhopisov potrebných na vyplatenie sumy **m**, ▼ ak také vyplatenie neexistuje

Špeciálna hodnota pre „nedá sa vyplatiť“

Pre začiatok nám stačí vypočítať, či sumu ide vyplatiť, a ak áno, tak koľko na to treba dlhopisov (cenných papierov).

- Okamžité pozorovanie:
  - ak `m = h[i]` pre nejaké *i*, potom `R[m] = 1`

Reinterpretácia pre fyzikov

- Rôzne typy rezistorov:
  - 10Ω, 22Ω, 47Ω, 100Ω, 270Ω
  - z každého typu si vieme objednať ľubovoľný počet

- Pri zapojení do série je výsledný odpor súčet odporov jednotlivých rezistorov.
- Problém: Aký najmenší počet rezistorov treba na sériové zapojenie so zadaným celkovým odporom?

Ako to vyriešiť?

Máme nový výpočtový
cluster za pár miliónov,
problém je zdá sa tažký,
ale bactracking to spraví.

Backtracking

Nominálne hodnoty dlhopisov

Suma: **3000**

0	1	2
100	500	800

Generujeme všetky možné vyplatenia sumy 3000, t.j., koľko akých dlhopisov použiť.

$$3000 = 2.100 + 4.500 + 1.800$$

0	1	2
2	**4**	**1**

Generujeme čísla: 0..Suma/800

Generujeme čísla: 0..Suma/100

Generujeme čísla: 0..Suma/500

Backtracking - zefektívnenie

Nominálne hodnoty dlhopisov

Suma: **3000**

0	1	2
100	500	800

$$3000 = 7.100 + ?.500 + ?.800$$

0	1	2
7		

Ostáva:
$3000 - 7.100 = 2300$

Generujeme čísla:
0..2300/500=0..4

Backtracking – vylepšenia

- Súčet počtu dlhopisov a zostávajúcu sumu vypočítavame priebežne
- Pri generovaní početností jednotlivých typov dlhopisov zohľadňujeme výšku zostávajúcej sumy
- Hodnoty dlhopisov v klesajúcom poradí

Zdá sa, že to funguje!

VIP klient

Nákup dhlopisov za **1 000 000** €

Kde to zlyhalo?

- Výpočet trvá pridlho pre veľké sumy
- Čas výpočtu zásadne rastie s počtom typov dlhopisov
  - Pri n typoch dlhopisov máme minimálne 2^n rôznych vyplatení

Analýza problému

Backtracking zlyhal, skúsme rozdeľuj a panuj.

Ako vieme rozložiť problém na menšie podproblémy?

Analýza optimálneho riešenia

Minimálny počet dlhopisov pre sumu 8900

Minimálny počet dlhopisov pre sumu 8400

Pozorovanie

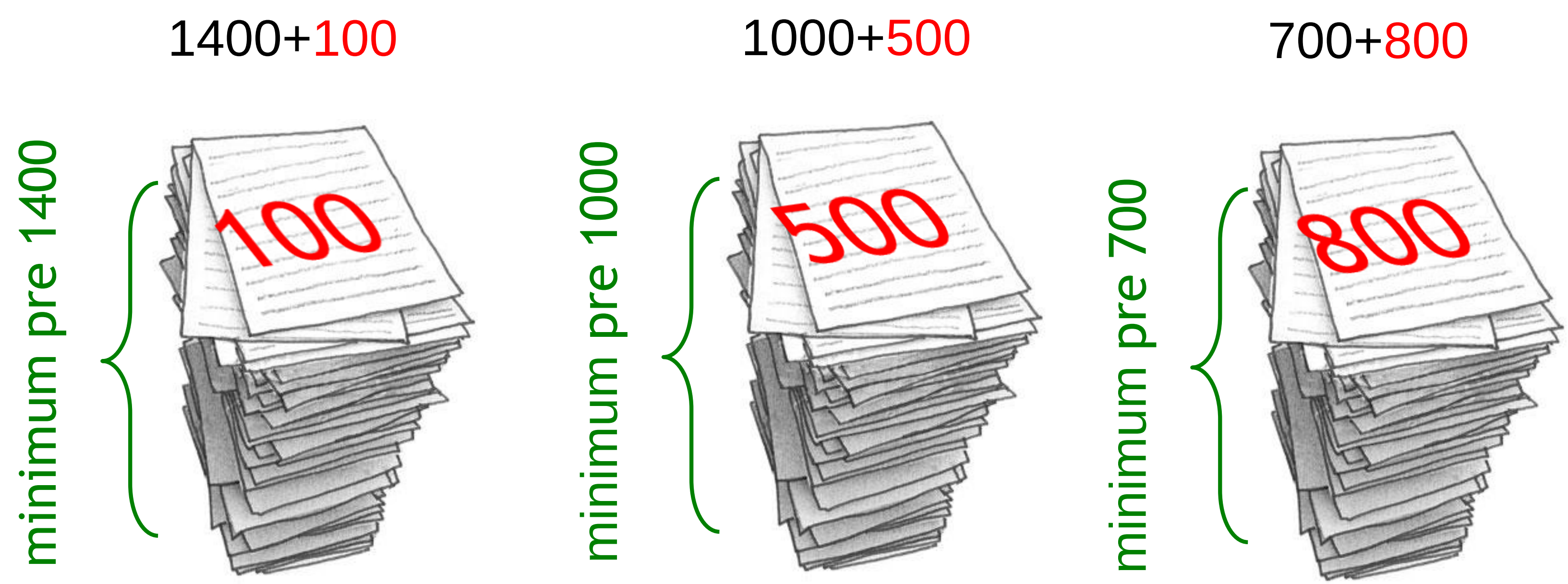
- Nech $d_1 + d_2 + ... + d_k = m$ je také rozdelenie sumy m na k dlhopisov, že k je najmenšie možné a $d_i \in \{h[0], h[1], ..., h[n-1]\}$ pre všetky i.
- Pozorovanie:
  - $d_1 + d_2 + ... + d_{k-1}$ je vyplatenie sumy $m-d_k$ s minimálnym počtom dlhopisov, t.j.

```
R[m] = R[m-dk]+1
```

A d_k je ???

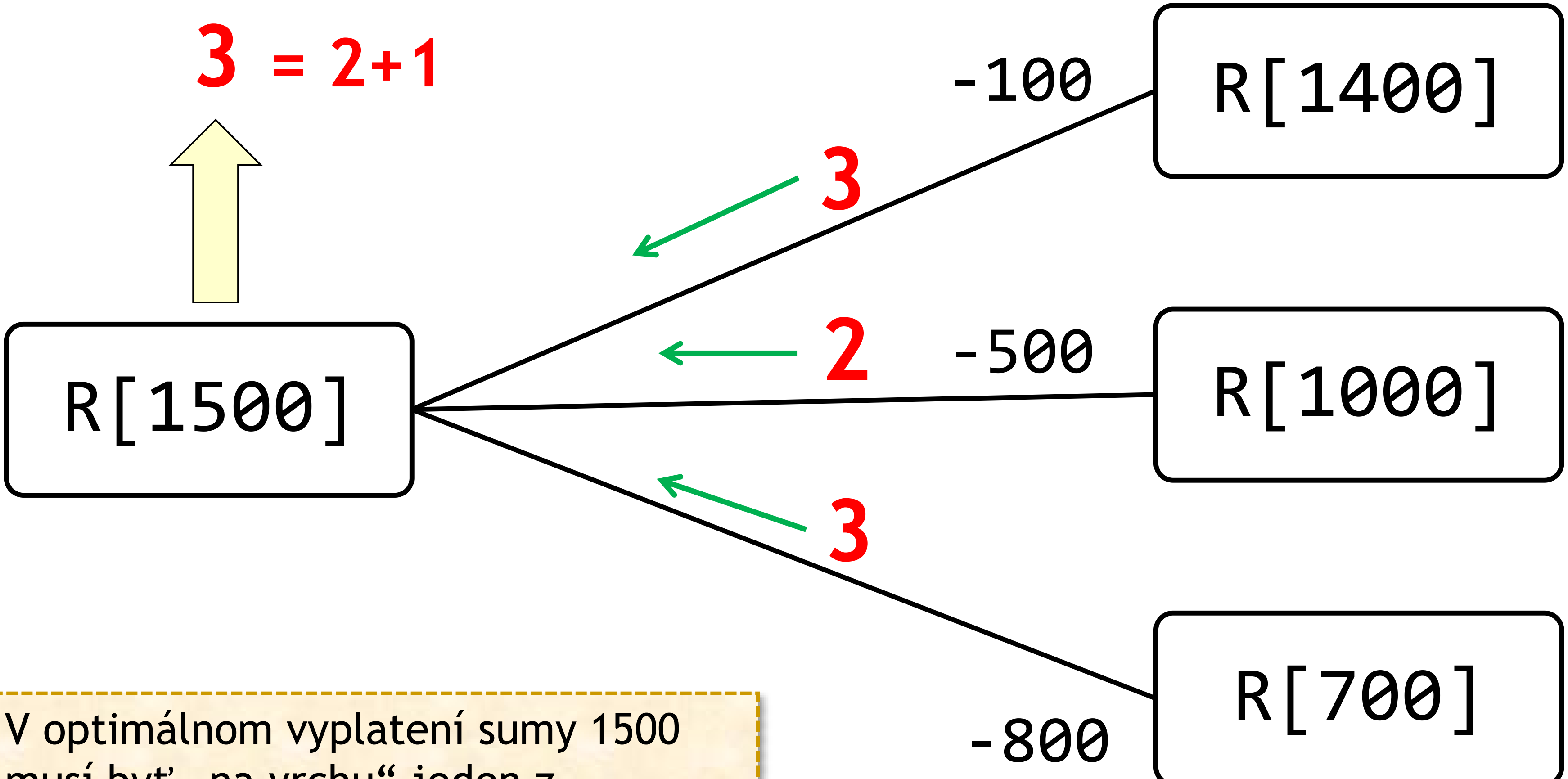
- Dôkaz (spor s optimalitou):
  - ak by $\boldsymbol{m-d_k}$ išlo vyplatiť menej než $\boldsymbol{k-1}$ dlhopismi, potom nám stačí k tomuto vyplateniu priložiť dlhopis $\boldsymbol{d_k}$ a máme vyplatenie sumy $\boldsymbol{m}$ menej ako $\boldsymbol{k}$ dlhopismi.

Čo vieme povedať o kôpkách?



- Jedna z kopiek je minimálna pre 1500
- Žiadna zo zvyšných kopiek nemá menej papierov ako minimálna (optimálna)

Optimálne riešenie



V optimálnom vyplatení sumy 1500 musí byť „na vrchu“ jeden z dhlopisov a po jeho odstránení nám ostane optimum pre zvyšnú sumu.

„Magický vzorec“

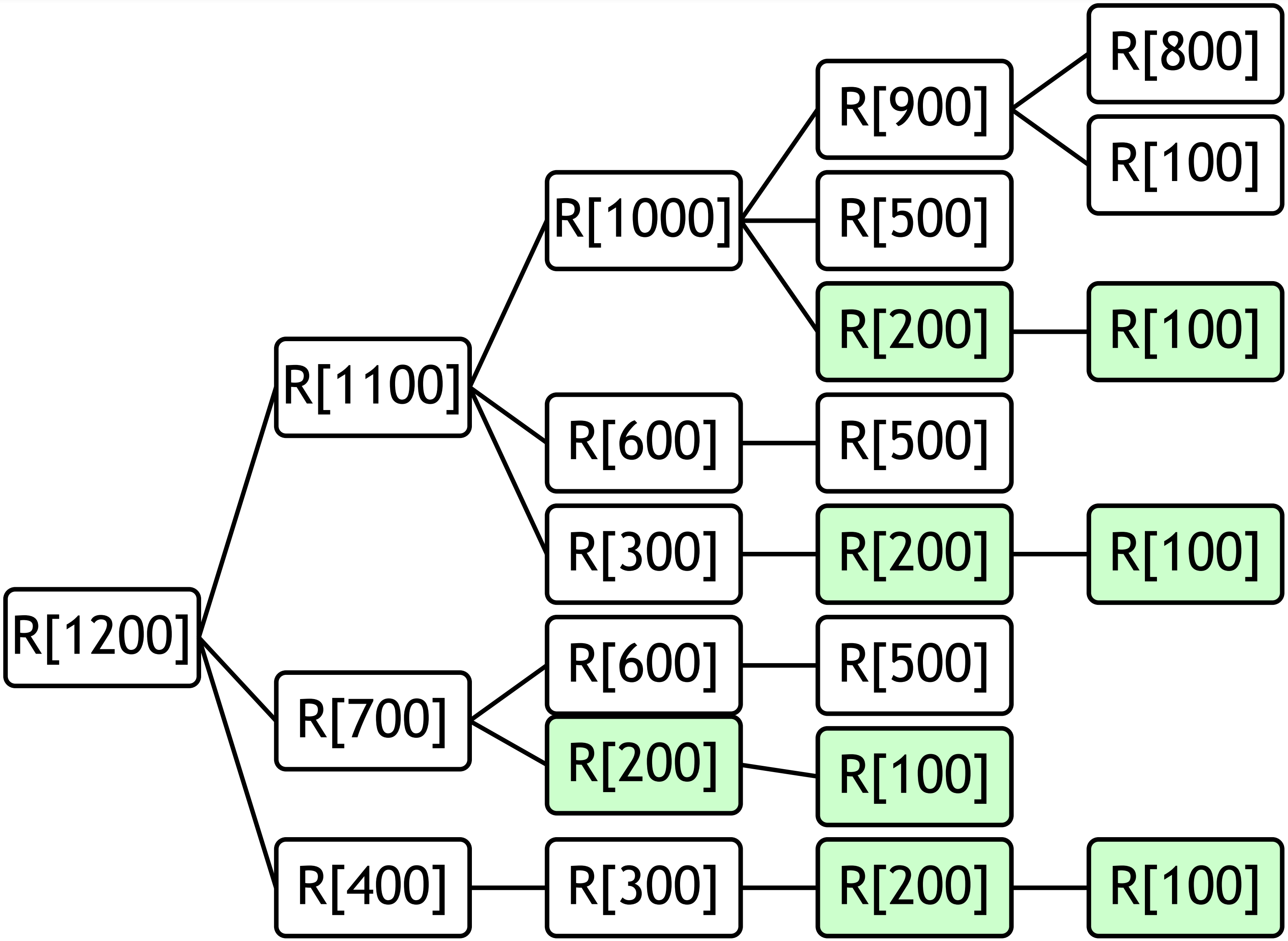
- **Rekurzívny „vzorec“** vyjadrujúci optimálne riešenie na základe optimálnych riešení menších problémov (vyplatenia menších súm)
  - `R[0] = 0`
  - `R[m] = 1 + min {R[m – h[i]]|i = 0..n-1}`
    - ak pre niektoré `i` je `R[m-h[i]]` rôzne od ▼
  - `R[m] = „nedá sa“ (▼), inak`
- Programujeme (rekurzívne) a testujeme ...

Efektívnosť?

- Nájdenie riešenia je **extrémne pomalé** aj pre nízke sumy a malé počty nominálnych hodnôt dlhopisov.
  - Nepraktické riešenie pre prax...

Analyzujme strom volaní a priebeh výpočtu...

Strom volaní



Dá sa s tým niečo spraviť?

- **Pozorovanie:** Zbytočne **opakované výpočty** – ak raz vypočítame, že na sumu 200€ treba 2 cenné papiere, tak to tak bude stále (pri rovnakých nominálnych hodnotách dlhopisov)

Ako zabrániť opakovaným výpočtom toho istého?

- Zavedenie „cache" – po vypočítaní medzivýsledku, si ho **uložíme**, aby sme ho už opäť nemuseli počítať.

Ukladanie medzivýsledkov

- Vytvoríme dostatočne veľké pole a inicializujeme ho na **dohodnutú hodnotu** indikujúcu, že pre danú sumu výsledok ešte nebol vypočítaný.

```
int[] cache = new int[suma+1];
Arrays.fill(cache, NEBOLO_POCITANE);
```

Dohodnutá hodnota = „ešte nemáme vypočítané“

- Zmena v algoritme:
  - prv než začneme počítať, **overíme, či už nemáme výsledok** vypočítaný

```
if (cache[suma] != NEBOLO_POCITANE)
    return cache[suma];
```

Časová zložitosť

- Suma **m**, **n** nominálnych hodnôt dlhopisov:
  - skúmaných je vyplatenie nanajvýš **m** ($O(m)$) rôznych súm
  - na zistenie vyplatenia jednej hodnoty treba čas $O(n)$, ak nezarátavame čas riešenia podproblémov
  - vyplatenie každej sumy je počítané **nanajvýš raz** (vďaka uchovaniu medzivýsledkov)
- Celkový čas: $O(m.n)$
  - „pseudopolynomiálny čas“
- Celková pamäť: $O(m+n)$

Ako to celé dopadlo?

Prístup zdola-nahor

- Ukladanie medzivýsledkov – **memoizácia**
  - prístup **zhora-nadol** (od problému k podproblémom)
- „cache“ pamäť (pole medzivýsledkov) môžeme **systematicky** vyplniť počnúc najmenšími hodnotami
  - prístup **zdola-nahor**:
    - z riešení problémov **budujeme** riešenia väčších problémov
    - **začíname triviálnymi prípadmi**
  - možno vyplníme trochu viac, než treba...

Príklad vypĺňania

- Nominálne hodnoty dlhopisov: 1€, 3€

0	1	2	3	4	5	6	7
0	1	2	1	2	3	2	?

2 = 1 + min{1, 3}

- Nominálne hodnoty dlhopisov: 2€, 5€

0	1	2	3	4	5	6	7
0	▼	1	▼	2	1	3	?

Ako nájsť vyplatenie?

- Algoritmus vypočíta minimálny počet dlhopisov potrebných na vyplatenie sumy (optimálne riešenie), ale nenájde, aké dlhopisy máme použiť
- **Riešenie:**
  - Pre každú sumu si budeme pamätať akým dlhopisom sme našli optimálne vyplatenie danej sumy
  - **Spätným prechodom** zrekonštruujeme optimálne riešenie

Ako nájsť vyplatenie?

- Nominálne hodnoty dlhopisov: 1€, 3€

0	1	1	3	1	3	3	?
0	1	2	3	4	5	6	7
0	1	2	1	2	3	2	?

$2 = 1 + min\{1, 3\}$

Hodnota dlhopisu, ktorý je v nejakom optimálnom vyplatení danej sumy.

- Nominálne hodnoty dlhopisov: 2€, 5€

0	0	2	0	2	5	2	?
0	1	2	3	4	5	6	7
0	▼	1	▼	2	1	3	?

Dynamické programovanie

- Formalizácia riešenia, definovanie podproblémov, charakterizovanie štruktúry optimálneho riešenia
- **„Magický vzorec“** na výpočet optimálneho riešenia z optimálnych riešení podproblémov
- Skonštruovanie riešení z riešení podproblémov prístupom **zdola-nahor** („vyplnenie tabuľky“)
- **Nájdenie hodnoty** optimálneho riešenia
- Zrekonštruovanie optimálneho riešenia (spätným prechodom)

Nič nové ...

- Postup, ktorý sme použili pri riešení „bankového problému“ už poznáme:
  - **Fibonacciho čísla** F(n) = F(n-1) + F(n-2) sme riešili aj vyplňaním poľa
  - Fibonacciho čísla ide riešiť aj bez poľa s použitím 3 premenných
    - **Idea:** na vyplnenie políčka poľa nám stačia len dve predchádzajúce hodnoty
    - Podobnú „fintu“ ide často použiť aj pri dynamickom programovaní – v bankovom probléme potrebujeme ***max{h[i]}*** predchádzajúcich políčok

Kedy dynamické programovanie?

- **Optimálna podštruktúra**
  - optimálne riešenie je možné získať z optimálnych riešení podproblémov
- **Prekrývajúce sa podproblémy**
  - celkový počet rôznych podproblémov musí byť relatívne malý
- **Prístup zdola-nahor**
  - od malých problémov k väčším...

ak nie sú otázky...

Ďakujem za pozornosť!

Najdlhšia vybraná rastúca podpostupnosť

Najdlhšia vybraná rastúca podpostupnosť

- Postupnosť n čísel:

5	3	1	4	6	2	8	7

- Ako vybrať čo najviac čísel tak, aby vybrané čísla tvorili **rastúcu podpostupnosť**?

5	**3**	1	**4**	**6**	2	**8**	7

3 < 4 < 6 < 8

- Iné riešenie:

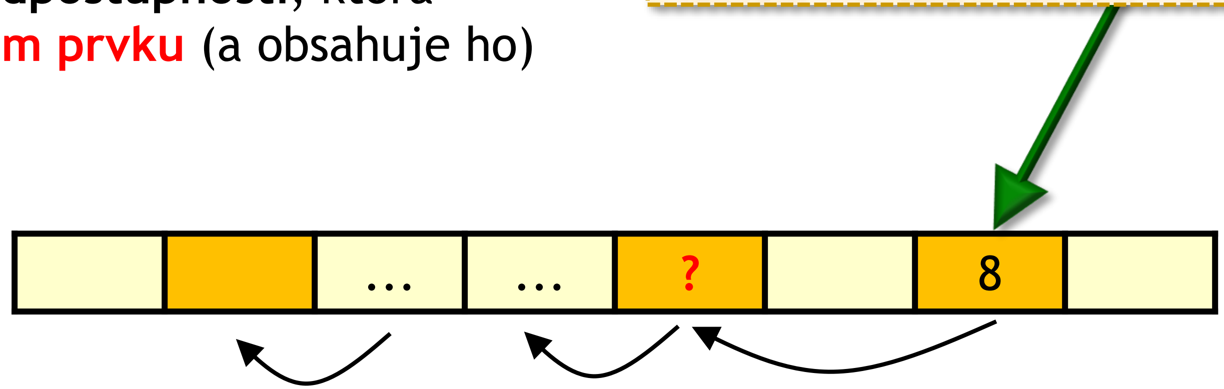
5	3	**1**	**4**	**6**	2	8	**7**

- *Existuje riešenie, kde vyberieme viac ako 4 čísla?*

Charakterizovanie štruktúry

Uvažujme **najdlhšiu vybranú rastúcu podpostupnosť**, ktorá končí vo zvolenom prvku postupnosti (a obsahuje ho)

D[i] – dĺžka **najdlhšej vybranej rastúcej podpostupnosti**, ktorá **končí v i-tom prvku** (a obsahuje ho)

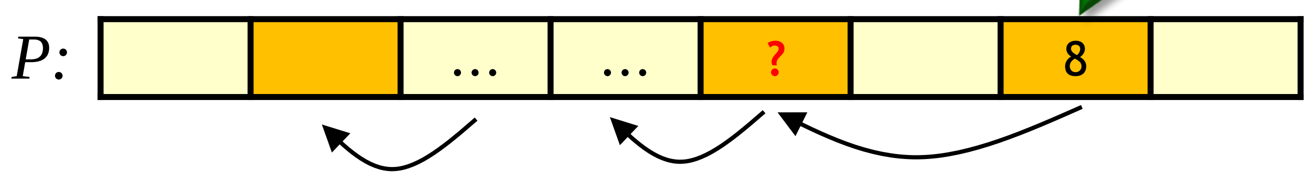


Optimálne riešenie predĺžuje o jeden prvok (8) nejakú **najdlhšiu vybranú rastúcu podpostupnosť**, ktorá končí v nejakom **prvku**, ktorý je **naľavo od 8** a je **menší ako 8**.

„Magický vzorec“

Nevieme, ktorú podpostupnosť predlžujeme, ale určite je to nejaká, ktorá končí v nejakom menšom prvku naľavo...

D[i] – dĺžka **najdlhšej vybranej rastúcej podpostupnosti**, ktorá **končí v i-tom prvku** (a obsahuje ho)



$$D[i] = 1 + \max \underbrace{\{0, D[j]: 0 \leq j < i \text{ a } P[j] < P[i]\}}$$

Skúšame predĺžiť každú **vhodnú** podpostupnosť končiacu **naľavo od P[i]** a vyberieme **maximum**.

Zdola-nahor

$$D[i] = 1 + \max \{0, D[j]: 0 \leq j < i \text{ a } P[j] < P[i]\}$$

Na určenie **D[i]** treba mať vypočítané **D[j]** pre **j < i**. Preto D budeme počítať postupne D[0], D[1], D[2], ...

```
int[] d = new int[p.length];

for (int i = 0; i < p.length; i++) {
    d[i] = 1;
    for (int j = 0; j < i; j++)
        if (p[j] < p[i])
            d[i] = Math.max(d[i], d[j] + 1);
}
```

Hodnota optimálneho riešenia

- Nevieme, v ktorom prvku postupnosti končí **globálne najdlhšia vybraná rastúca podpostupnosť**...
- Vyberieme tú najdlhšiu:

```
int najdlhsia = 0;
for (int i=0; i<d.length; i++)
    najdlhsia = Math.max(najdlhsia, d[i]);
```

Ako rýchlo zrekonštruovať optimálne riešenie? Akú pomocnú informáciu potrebujeme uchovať pri výpočte poľa d?

Najdlhšia spoločná podpostupnosť

LCS

- **LCS (Longest common subsequence)** – najdlhšia spoločná podpostupnosť
  - 2 postupnosti (môžu byť aj rôznej dĺžky)
  - hľadáme najdlhšiu spoločnú podpostupnosť

- Použitie: hľadanie rozdielov v textoch (v Linuxe príkaz **diff**), hľadanie génových mutácií

Formalizácia, podproblém

- Postupnosti (pozor, indexujeme od 1):
  - $a_1, a_2, \ldots, a_n$
  - $b_1, b_2, \ldots, b_m$
- Podproblém:
  - **LCS(i, j)** – dĺžka najdlhšej spoločnej podpostupnosti z postupností $a_1, a_2, \ldots, a_i$ a $b_1, b_2, \ldots, b_j$
- Náš cieľ: **LCS(n, m)**
- Triviálne fakty: *LCS(0, ?) = 0, LCS(?, 0) = 0*

„Magický vzorec“ pre LCS

- Zoberme si nejaké optimálne riešenie pre LCS(i, j), potom platí jedno z nasledujúceho:
  - a_i je v optimálnom riešení a b_j nie je
    - LCS(i, j) = LCS(i, j-1)
  - a_i nie je v optimálnom riešení a b_j je
    - LCS(i, j) = LCS(i-1, j)
  - ani a_i a ani b_j nie sú v optimálnom riešení
    - LCS(i, j) = LCS(i-1, j-1) a $a_i != b_j$
  - a_i aj b_j sú v optimálnom riešení
    - LCS(i, j) = 1 + LCS(i-1, j-1) a $a_i = b_j$

Ide ukázať sporom

„Magický vzorec" pre LCS

- Ak $a_i=b_j$, potom **LCS(i, j) = 1 + LCS(i-1, j-1)**
  - Ak a_i a b_j sú v nejakej optimálnej podpostupnosti, potom nie je čo riešiť a *LCS(i, j) = 1 + LCS(i-1, j-1)*
  - Ak len jedno z a_i a b_j nie je v optimálnej postupnosti, tak existuje optimálna postupnosť, kde sú obe (prečo?)
  - Ak a_i a b_j nie sú spoločne v nejakej optimálnej podpostupnosti, potom ich môžeme pridať na koniec nejakej optimálnej podpostupnosti, čím ju predlžíme (spor s optimalitou)
- Ak $a_i!=b_j$, potom **LCS(i-1, j-1) <= LCS(i, j-1)** a **LCS(i-1, j-1) <= LCS(i-1, j) <= LCS(i-1, j-1)+1**

„Magický vzorec" pre LCS

- LCS(0, j) = 0
- LCS(i, 0) = 0
- Ak a_i=b_j, potom
  - LCS(i, j) = 1 + LCS(i-1, j-1)
- Ak a_i!=b_j, potom
  - LCS(i, j) = max{LCS(i-1, j), LCS(i, j-1)}

Algoritmus

```
public static int vypocitajLCS(String s1, String s2) {
   int[][] lcs = new int[s1.length()+1][s2.length()+1];
   for (int i = 1; i < s1.length()+1; i++)
      for (int j = 1; j < s2.length()+1; j++)
         if (s1.charAt(i-1) == s2.charAt(j-1))
            lcs[i][j] = 1 + lcs[i-1][j-1];
         else
            lcs[i][j] = Math.max(lcs[i-1][j], lcs[i][j-1]);

    return lcs[s1.length()][s2.length()];
}
```

Časová zložitosť: $O(m.n)$
Pamäťová zložitosť: $O(m.n)$

Príklad tabuľky

	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

Spätným prechodom a poznačením si, ako sa dosiahlo optimum, vieme zrekonštruovať LCS