

# Dynamické programování

Programovací technika pro řešení takových optimalizačních úloh, kde úloha se dá rozložit na menší podúlohy, z jejichž optimálních řešení se skládá optimální řešení celé původní úlohy.

Slouží k tvorbě časově efektivních algoritmů, zpravidla za cenu jistého zvýšení paměťových nároků - čas i paměť polynomiální (nahradí backtracking s exponenciální časovou složitostí).

*Princip:*

Rozklad úlohy na menší podúlohy stejného charakteru, z jejichž řešení se sestaví řešení původní úlohy – podobné jako u techniky „Rozděl a panuj“.

Na rozdíl od techniky „Rozděl a panuj“ nemusejí být podúlohy navzájem nezávislé → širší možnosti použití.

*Typické úlohy:*

**1. najít nejlepší způsob, jak lze něco udělat**

- a) buď nám stačí optimální dosažitelná hodnota  
(např. délka nejdelší posloupnosti, nejmenší možný počet)
- b) nebo
  - nejprve určíme optimální dosažitelnou hodnotu
  - při jejím výpočtu si evidujeme, jak jsme ji získali  
(viz ukládání předchůdců při hledání nejkratší cesty v grafu)
  - nakonec „zpětným chodem“ zrekonstruujeme cestu k výsledku

*Příklad:*

- nejdelší rostoucí vybraná podposloupnost

## ***2. určit počet způsobů, jak lze něco udělat***

- postupujeme podobně jako v 1a)
- místo porovnávání dílčích hodnot je budeme sčítat

*Příklad:*

- počet různých cest z vrcholu A do vrcholu B  
v orientovaném acyklickém grafu

*Postup řešení – dvě možnosti:*

### **1. shora dolů**

rekurzivní rozklad problému shora, do pomocného pole si přitom průběžně ukládáme všechny již jednou spočítané hodnoty, aby se nepočítaly opakovaně znovu, až budou opět potřeba („chytrá rekurze“, „memoizace“)

### **2. zdola nahoru**

postupujeme naopak iteračně zdola od elementárních úloh ke složitějším, výsledky řešení všech podúloh si ukládáme (zpravidla do pole) pro další použití při řešení složitějších podúloh

*Příklad: Fibonacciho čísla (bylo již dříve)*

## Paměťové nároky:

Oproti primitivnímu rekurzivnímu řešení potřebujeme paměť navíc pro uložení výsledků všech dílčích podúloh

– při iteračním postupu zdola dokonce často i těch, které nakonec ani nebudeme potřebovat (ale předem nevíme, které dílčí výsledky později potřebovat budeme a které ne, takže si počítáme a ukládáme systematicky všechny).

Tyto dodatečné paměťové nároky mohou být velikosti  $O(N)$ , velmi často bývají  $O(N^2)$  – tedy dvourozměrné pole. V každém případě jsou polynomiální.

## Časová efektivita:

Díky ukládání výsledků všech menších podúloh máme zajištěno, že se žádná podúloha nebude počítat opakovaně vícekrát – může se jen vícekrát použít její výsledek uložený již jednou v poli. Proto nemusejí být podúlohy na sobě navzájem nezávislé a přesto je zaručena polynomiální časová složitost výpočtu.

Počet provedených operací =

*počet řešených podúloh*

(počet prvků pole určeného pro uložení všech dílčích výsledků)

x *náročnost řešení každé jednotlivé podúlohy*

(s využitím toho, že už známe výsledky podúloh menších)

*Příklad:*  $N^2$  (počítaných hodnot) x  $N$  (výpočet každé z nich) =  $N^3$

→ časová složitost  $O(N^3)$

# ***Příklady využití dynamického programování***

## **Fibonacciho čísla *(již známe)***

Určení  $N$ -tého Fibonacciho čísla rekurzivní funkcí

– chybně použitý princip „Rozděl a panuj“, vznikající podúlohy byly závislé → exponenciální časová složitost

X

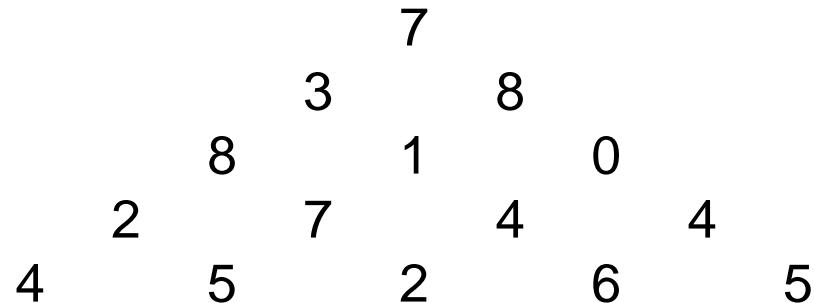
Určení  $N$ -tého Fibonacciho čísla rekurzivní funkcí doplněnou polem pro ukládání již spočítaných hodnot, takže každá hodnota se počítá jen jednou → lineární časová složitost

Určení  $N$ -tého Fibonacciho čísla postupným počítáním zdola v cyklu  
– nic se nepočítá opakovaně vícekrát → lineární časová složitost

# Trojúhelník z čísel

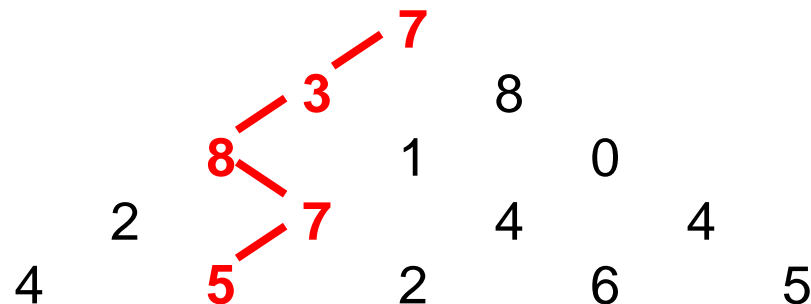
$N$  řádků, obsahují postupně  $1, 2, 3, \dots, N$  čísel – zadán na vstupu

*Příklad pro  $N = 5$ :*



*Úkol:* nalézt cestu z horního vrcholu na základnu s maximálním součtem čísel, smí se jít vždy jen šikmo směrem dolů.

*Řešení:*





## ***Postup řešení***

### *1. Neefektivní:*

- uložit všechna čísla do pole  $\rightarrow$  paměť  $O(N^2)$
- backtrackingem postupně zkoušet všechny cesty  $\rightarrow$  čas  $O(2^N)$

## 2. Dynamické programování:

- načíst a uložit vždy jen jeden řádek čísel, spočítat a uložit maximální součet čísel na cestě vedoucí do každého z nich

- paměť  $O(N)$

*některé z těchto údajů se ve výsledku neuplatní, ale ještě nevíme které, tak počítáme a ukládáme všechny!*

- další řádek spočítat vždy z předchozího, každé číslo má jen dva možné předchůdce a součty maximálních cest do nich vedoucích máme uloženy

- výsledkem je maximální spočítaná hodnota v posledním řádku

- počítáme tedy řádově  $N^2$  čísel, každé určíme v konstantním čase  
→ časová složitost  $O(N^2)$

### 3. Nalezení cesty:

- předchozí řešení určilo pouze hodnotu cesty s maximálním součtem čísel, neznáme zatím ale cestu samotnou
- pokud chceme určit i průběh cesty, musíme si do pole ukládat všechna čísla a s každým také informaci, zda jsme do něj přišli z předchozího řádku zleva nebo zprava → paměť  $O(N^2)$

*opět platí:*

*některé z těchto údajů se ve výsledku neuplatní, ale ještě nevíme které, tak počítáme a ukládáme všechny!*

- průběh cesty poté snadno zrekonstruujeme „zpětným chodem“ zdola nahoru (jdeme po předchůdcích, podobně jako při určování nejkratší cesty při procházení do šířky)
- časová složitost zůstává  $O(N^2)$

#### 4. Jiný postup dynamickým programováním – zespodu:

- čteme řádky čísel postupně zdola, pro každé z čísel můžeme spočítat a uložit maximální součet čísel na cestě vedoucí z tohoto čísla až dolů

*některé z těchto údajů se opět ve výsledku neuplatní, ale ještě nevíme které, tak počítáme a ukládáme všechny!*

- řádek spočítáme vždy z řádku následujícího pod ním, každé číslo má jen dva možné následníky a součty maximálních cest z nich vedoucích máme uloženy

- výsledkem je spočítaná hodnota v prvním řádku

- počítáme tedy řádově  $N^2$  čísel, každé určíme v konstantním čase  
→ opět časová složitost  $O(N^2)$

- lze opět doplnit nalezení cesty

## Arpád (počet různých cest minimální délky)

- ve městě je pravoúhlá síť ulic velikosti  $N \times M$ , dvojce sousedních ulic mají stejné vzájemné vzdálenosti (jednotkové)
- některé křižovatky jsou rozkopené a proto neprůjezdné (seznam zadán na vstupu)
- Arpád bydlí na křižovatce  $[1, 1]$ , jezdí do práce na křižovatku  $[N, M]$
- Arpád jezdí na poslední chvíli, takže musí jet po trase minimální délky (což je délka  $M+N-2$ )
- úkol: určit, kolik různých cest má Arpád na výběr

## ***Řešení:***

označme  $T[i, j]$  počet různých cest z křižovatky  $[1, 1]$  na  $[i, j]$   
hledáme  $T[N, M]$

základní pozorování: platí vztah  $T[i, j] = T[i-1, j] + T[i, j-1]$

(ještě ošetřit okraje a rozkopené křižovatky! – ale to je snadné)

### *1. primitivní řešení rekurzí*

voláme rekurzivní funkci  $R(N, M)$ , který implementuje výše uvedený vztah

→ opakovaná volání funkce  $R$  se stejnými parametry,  
exponenciální časová složitost

## *2. dynamické programování shora dolů*

voláme rekurzivní funkci  $R(N, M)$ , který implementuje výše uvedený vztah a využívá dvojrozměrné pole  $T$  velikosti  $N \times M$  již spočítaných hodnot

→ nejsou opakované výpočty,  
časová složitost v nejhorším případě  $O(N.M)$

## *3. dynamické programování zdola nahoru*

po řádcích vyplňujeme dvojrozměrné pole  $T$  velikosti  $N \times M$  známých hodnot podle výše uvedeného vztahu

→ nejsou opakované výpočty,  
časová složitost v každém případě  $O(N.M)$

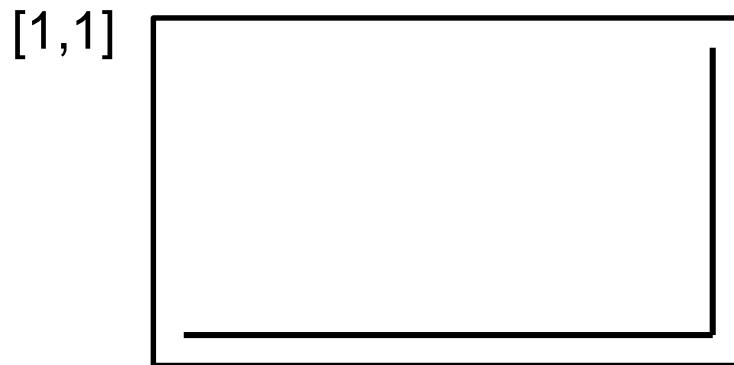
Nevýhoda postupu shora dolů:

opakovaná volání rekurzivní funkce (režie na rekurzi nás něco stojí)

Výhoda postupu shora dolů:

nepočítáme nepotřebné hodnoty, takže výpočet může být rychlejší,  
v případě na obrázku dole dokonce jen  $O(N+M)$

*čára na obrázku vyznačuje, kde jsou rozkopené křížovatky*





# Délka nejdelší rostoucí vybrané podposloupnosti

## 1. hladově

- lineární průchod s postupným prodlužováním → výsledek špatně!

## 2. primitivní řešení rekurzí

- zkouší všechny vybrané podposloupnosti → exponenciální složitost

## 3. dynamické programování (zdola)

- pole  $T[i]$  – délka maximální rostoucí podposloupnosti vybrané z počátečního úseku  $A[1], A[2], \dots, A[i]$  délky  $i$  a zakončené prvkem  $A[i]$
  - postupně počítáme  $T[1], T[2], \dots, T[N]$
  - hledaným výsledkem je maximum z hodnot v poli  $T$
  - každý prvek  $T[i]$  určíme v čase  $O(N)$  na základě už známých hodnot uložených v poli  $T$
- celková časová složitost  $O(N^2)$

```
a = [8, 1, 2, 9, 3, 6, 7]

t = [1]*len(a)
max = 1

for i in range(1, len(a)):
    for j in range(i):
        if a[j] < a[i] and t[j] + 1 > t[i]:
            t[i] = t[j] + 1
    if t[i] > max:
        max = t[i]

print(max)
```

## Rozšíření: určení prvků nejdelší rostoucí vybrané podposloupnosti

```
a = [8, 1, 2, 9, 3, 6, 7]
```

```
t = [1]*len(a)    # hodnoty délek
```

```
p = [0]*len(a)    # předchůdci
```

```
max = 1           # maximální délka
```

```
k = 0             # kde jsme získali maximální délku
```

```
for i in range(1, len(a)):
```

```
    for j in range(i):
```

```
        if a[j] < a[i] and t[j] + 1 > t[i]:
```

```
            t[i] = t[j] + 1
```

```
            p[i] = j
```

```
if t[i] > max:
```

```
    max = t[i]
```

```
    k = i
```

```
print(max)    # délka max. rostoucí vybrané podposloupnosti

print(k, end = ' ')
while p[k] != 0:
    k = p[k]
    print(k, end = ' ')
print()
# vypisuje ODZADU indexy všech prvků vybrané podposloupnosti
```

#### 4. poznámka

jde to i lépe – s časovou složitostí  $O(N \log N)$

idea:

- definujeme hodnoty  $M[j]$  = minimální dosud známá hodnota posledního prvku vybrané rostoucí posloupnosti délky  $j$
- přepočítáváme je postupně pro  $A[1], A[2], \dots, A[M]$   
(rozmyslete sami nebo viz cvičení)

# Délka nejdelší společné vybrané podposloupnosti

jsou dány posloupnosti  $A[1..M]$ ,  $B[1..M]$

$T[i, j]$  – délka nejdelší společné podposloupnosti vybrané z počátečních úseků  $A[1]$ ,  $A[2]$ , ...,  $A[i]$  resp.  $B[1]$ ,  $B[2]$ , ...,  $B[j]$   
hledáme  $T[N, M]$

platí:     když  $i = 0$  nebo  $j = 0$ , pak  $T[i, j] = 0$   
             jinak když  $A[i] = B[j]$ , pak  $T[i, j] = T[i-1, j-1] + 1$   
             jinak  $T[i, j] = \max(T[i, j-1], T[i-1, j])$

*dynamické programování (zdola)*: tabulku  $T$  vyplňujeme po řádcích, každou hodnotu  $T[i, j]$  spočítáme v konstantním čase  
→ celková časová složitost  $O(N.M)$

*poznámka*: v případě postupu rekurzivně shora může být časová složitost algoritmu  $O(N)$  ... pokud jsou obě posloupnosti shodné