

# Příklady analýzy složitosti algoritmů

(a příklady technik používaných při návrhu efektivních algoritmů)

# Složitost algoritmů

Orientační typické hodnoty velikosti vstupu  $n$ , pro které algoritmus s danou časovou složitostí ještě většinou zvládne na „běžném PC“ spočítat výsledek ve zlomku sekundy nebo maximálně v řádu sekund.

(Závisí to samozřejmě výrazně na konkrétních detailech. Navíc se zde předpokládá, že v asymptotické notaci nejsou skryty nějaké velké konstanty.)

$O(n)$	$O(n \log n)$	$O(n^2)$	$O(n^3)$
1 000 000 – 100 000 000	100 000 – 1 000 000	1000 – 10 000	100 – 1000
	$2^{O(n)}$	$O(n!)$	
	20 – 30	10 – 15	

Při používání asymptotických odhadů časové složitosti algoritmů bychom si měli být vědomi některých úskalí:

- Asyptotické odhady se týkají pouze toho, jak roste čas s rostoucí velikostí vstupu.
- Neříkají nic o konkrétní době výpočtu. V asymptotické notaci mohou být skryty velké konstanty.
- Algoritmus, který má lepší asymptotickou časovou složitost než nějaký jiný algoritmus, může být ve skutečnosti rychlejší až pro nějaké hodně velké vstupy.
- Většinou analyzujeme složitost v nejhorším případě. Pro některé algoritmy může být doba výpočtu v nejhorším případě mnohem větší než doba výpočtu na „typických“ instancích.

- Můžeme si to ilustrovat na algoritmech pro třídění.

Algoritmus	Nejhorší případ	Průměrný případ
Bubblesort	$\Theta(n^2)$	$\Theta(n^2)$
Heapsort	$\Theta(n \log n)$	$\Theta(n \log n)$
Quicksort	$\Theta(n^2)$	$\Theta(n \log n)$

- Quicksort má horší asymptotickou složitost v nejhorším případě než Heapsort, stejnou asymptotickou složitost v průměrném případě a přesto je v praxi nejrychlejší.

**Polynom** — funkce tvaru

$$a_k n^k + a_{k-1} n^{k-1} + \dots + a_2 n^2 + a_1 n + a_0$$

kde  $a_0, a_1, \dots, a_k$  jsou konstanty.

Příklady polynomů:

$$4n^3 - 2n^2 + 8n + 13$$

$$2n + 1$$

$$n^{100}$$

Funkce  $f$  je **polynomiální**, jestliže je shora omezena nějakým polynomem, tj. jestliže existuje nějaká konstanta  $k$  taková, že  $f \in O(n^k)$ .

Polynomiální jsou například funkce, které patří do následujících tříd:

$$O(n)$$

$$O(n \log n)$$

$$O(n^2)$$

$$O(n^5)$$

$$O(\sqrt{n})$$

$$O(n^{100})$$

Funkce jako  $2^n$  nebo  $n!$  polynomiální nejsou — pro libovolně velkou konstantu  $k$  platí

$$2^n \in \Omega(n^k)$$

$$n! \in \Omega(n^k)$$

**Polynomiální algoritmus** — algoritmus, jehož časová složitost je polynomiální (tj. shora omezená nějakým polynomem)

Zhruba se dá říct, že:

- polynomiální algoritmy jsou efektivní algoritmy, které se dají prakticky použít i pro relativně velké vstupy
- algoritmy, které polynomiální nejsou, se dají použít jen pro poměrně malé vstupy

Rozdělení na polynomiální a nepolynomiální algoritmy je velmi hrubé — nelze kategoricky tvrdit, že polynomiální algoritmy jsou vždy prakticky použitelné a nepolynomiální naopak nikdy nejsou:

- algoritmus se složitostí  $\Theta(n^{100})$  pravděpodobně příliš prakticky použitelný nebude,
- některé algoritmy, které nejsou polynomiální, mohou fungovat efektivně pro velkou část vstupů, a složitost větší než polynomiální mají jen kvůli některým problematickým vstupům, na kterých může výpočet trvat velmi dlouhou dobu.

**Poznámka:** Polynomiální algoritmy, kde by konstanta v exponentu bylo nějaké velké číslo (např. algoritmy se složitostí  $\Theta(n^{100})$ ), se při řešení běžných algoritmických problémů prakticky nevyskytují.

Pro většinu běžných algoritmických problémů nastává jedna ze tří možností:

- Je znám polynomiální algoritmus se složitostí  $O(n^k)$ , kde  $k$  je nějaké velmi malé číslo (např. 5 a častěji třeba 3 a méně).
- Není znám žádný polynomiální algoritmus a nejlepší známé algoritmy mají složitosti jako třeba  $2^{\Theta(n)}$ ,  $\Theta(n!)$  nebo nějaké ještě větší.  
V některých případech může být znám i důkaz, že pro daný problém žádný polynomiální algoritmus neexistuje (tj. nedá se vytvořit).
- Není znám žádný algoritmus, který řeší daný problém (a případně je i dokázáno, že žádný takový algoritmus neexistuje).



Typický příklad polynomiálního algoritmu — násobení matic s časovou složitostí  $\Theta(n^3)$  a paměťovou složitostí  $\Theta(n^2)$ :

---

**Algoritmus:** Násobení matic

---

MATRIX-MULT ( $A, B, C, n$ ):

```
for  $i := 1$  to  $n$  do
  for  $j := 1$  to  $n$  do
     $x := 0$ 
    for  $k := 1$  to  $n$  do
       $x := x + A[i][k] * B[k][j]$ 
     $C[i][j] := x$ 
```

---

- Při hrubé analýze složitosti často stačí spočítat počet do sebe vnořených smyček — a tento počet pak udává stupeň polynomu

**Příklad:** Tři vnořené cykly při násobení matic — časová složitost algoritmu je  $O(n^3)$ .

- Pokud neprobíhají všechny smyčky např. od 0 do  $n$ , ale počet průchodů vnitřními smyčkami se při různých iteracích vnější smyčky mění, podrobnější analýza může být komplikovanější.

Většinou to pak vede na počítání součtů různých typů číselných řad (např. aritmetické, geometrické, apod.).

Často dá taková podrobnější analýza podobný výsledek jako hrubá analýza, mnohdy však může být složitost zjištěná touto podrobnější analýzou podstatně nižší než by vyplývalo z hrubého odhadu.

**Aritmetická posloupnost** — číselná řada  $a_0, a_1, \dots, a_{n-1}$ , kde

$$a_i = a_0 + i \cdot d,$$

kde  $d$  je nějaká konstanta nezávislá na  $i$ .

V aritmetické posloupnosti tedy pro všechna  $i$  platí  $a_{i+1} = a_i + d$ .

**Příklad:** Aritmetická posloupnost, kde  $a_0 = 1$ ,  $d = 1$  a  $n = 100$ :

$$1, 2, 3, 4, 5, 6, \dots, 96, 97, 98, 99, 100$$

**Součet aritmetické posloupnosti:**

$$\sum_{i=0}^{n-1} a_i = a_0 + a_1 + \dots + a_{n-1} = \frac{1}{2}n(a_0 + a_{n-1})$$

## Příklad:

$$1 + 2 + \dots + n = \frac{1}{2}n(n+1) = \frac{1}{2}n^2 + \frac{1}{2}n = \Theta(n^2)$$

Konkrétně například pro  $n = 100$  je

$$1 + 2 + \dots + 100 = 50 \cdot 101 = 5050.$$

**Důkaz:** Označme

$$s = \sum_{i=0}^{n-1} a_i = a_0 + a_1 + \cdots + a_{n-1}$$

$$2s = s + s$$

$$= (a_0 + a_1 + \cdots + a_{n-1}) + (a_0 + a_1 + \cdots + a_{n-1})$$

$$= (a_0 + a_1 + \cdots + a_{n-1}) + (a_{n-1} + a_{n-2} + \cdots + a_0)$$

$$= (a_0 + a_{n-1}) + (a_1 + a_{n-2}) + \cdots + (a_{n-1} + a_0)$$

$$= ((a_0 + 0 \cdot d) + (a_0 + (n-1) \cdot d)) + ((a_0 + 1 \cdot d) + (a_0 + (n-2) \cdot d)) + \cdots + ((a_0 + (n-1) \cdot d) + (a_0 + 0 \cdot d))$$

$$= n \cdot (a_0 + a_0 + (n-1) \cdot d)$$

$$= n \cdot (a_0 + a_{n-1})$$

**Příklad:**  $s = 1 + 2 + 3 + \dots + 99 + 100$

$$\begin{aligned}2s &= s + s \\&= (1 + 2 + \dots + 100) + (1 + 2 + \dots + 100) \\&= (1 + 2 + \dots + 100) + (100 + 99 + \dots + 1) \\&= (1 + 100) + (2 + 99) + (3 + 98) + \dots + (99 + 2) + (100 + 1) \\&= 100 \cdot (1 + 100) = 10100\end{aligned}$$

Takže

$$s = \frac{1}{2} \cdot 10100 = 5050$$

**Geometrická posloupnost** — číselná řada  $a_0, a_1, \dots, a_n$ , kde

$$a_i = a_0 \cdot q^i,$$

kde  $q$  je nějaká konstanta nezávislá na  $i$ .

V geometrické posloupnosti tedy pro všechna  $i$  platí  $a_{i+1} = a_i \cdot q$ .

**Příklad:** Geometrická posloupnost, kde  $a_0 = 1$ ,  $q = 2$  a  $n = 14$ :

1, 2, 4, 8, 16, 32, 64, 128, 256, 512, 1024, 2048, 4096, 8192, 16384

**Součet geometrické posloupnosti** (kde  $q \neq 1$ ):

$$\sum_{i=0}^n a_i = a_0 + a_1 + \dots + a_n = a_0 \frac{q^{n+1} - 1}{q - 1}$$

**Příklad:**

$$1 + q + q^2 + \dots + q^n = \frac{q^{n+1} - 1}{q - 1}$$

Speciálně pro  $q = 2$ :

$$1 + 2^1 + 2^2 + 2^3 + \dots + 2^n = \frac{2^{n+1} - 1}{2 - 1} = 2 \cdot 2^n - 1 = \Theta(2^n)$$



**Důkaz:** Označme

$$s = \sum_{i=0}^n a_i = a_0 + a_1 + \dots + a_n$$

$$s = a_0 \cdot q^0 + a_0 \cdot q^1 + \dots + a_0 \cdot q^n$$

$$s \cdot q = (a_0 \cdot q^0 + a_0 \cdot q^1 + \dots + a_0 \cdot q^n) \cdot q$$

$$= a_0 \cdot q^1 + a_0 \cdot q^2 + \dots + a_0 \cdot q^{n+1}$$

$$s \cdot q - s = a_0 \cdot q^{n+1} - a_0 \cdot q^0$$

$$s \cdot (q - 1) = a_0 \cdot (q^{n+1} - 1)$$

$$s = a_0 \cdot \frac{q^{n+1} - 1}{q - 1}$$

**Exponenciální** funkce: funkce tvaru  $c^n$ , kde  $c$  je konstanta —  
např. funkce  $2^n$

**Logaritmus** — inverzní funkce k exponenciální funkci: pro dané  $n$  je

$$\log_c n$$

taková hodnota  $x$ , že  $c^x = n$ .

# Složitost algoritmů

$n$	$2^n$
0	1
1	2
2	4
3	8
4	16
5	32
6	64
7	128
8	256
9	512
10	1024
11	2048
12	4096
13	8192
14	16384
15	32768
16	65536
17	131072
18	262144
19	524288
20	1048576

$n$	$\lceil \log_2 n \rceil$
0	—
1	0
2	1
3	2
4	2
5	3
6	3
7	3
8	3
9	4
10	4
11	4
12	4
13	4
14	4
15	4
16	4
17	5
18	5
19	5
20	5

$n$	$\log_2 n$
1	0
2	1
4	2
8	3
16	4
32	5
64	6
128	7
256	8
512	9
1024	10
2048	11
4096	12
8192	13
16384	14
32768	15
65536	16
131072	17
262144	18
524288	19
1048576	20

Příklady toho, kde se při analýze algoritmů objevují exponenciální funkce a logaritmy:

- Někaká hodnota se opakovaně zmenšuje na polovinu nebo naopak zdvojnásobuje.

Například u **binárního vyhledávání** (metodou půlení intervalu) se s každou iterací cyklu zmenšuje velikost intervalu na polovinu.

Předpokládejme, že pole má velikost  $n$ .

Jaká je minimální velikost pole  $n$ , při které se provede alespoň  $k$  iterací?

Odpověď:  $2^k$

Platí tedy  $k = \log_2(n)$ . Časová složitost algoritmu je pak  $\Theta(\log n)$ .

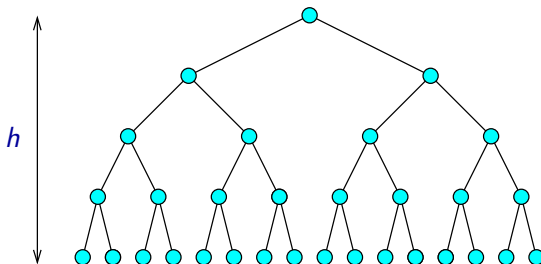
- Pomocí  $n$  bitů je možno reprezentovat čísla od 0 do  $2^n - 1$ .
- Minimální počet bitů potřebných pro uložení přirozeného čísla  $x$  reprezentovaného binárně je

$$\lceil \log_2(x + 1) \rceil.$$

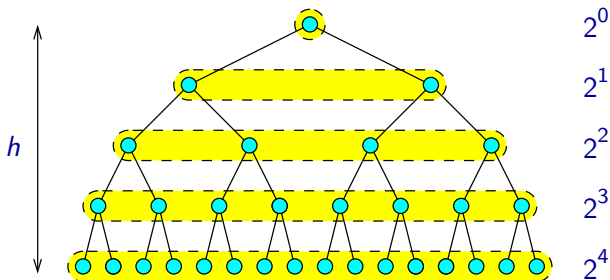
- Dokonale vyvážený binární strom o výšce  $h$  má  $2^{h+1} - 1$  vrcholů, z čehož  $2^h$  jsou listy.
- Dokonale vyvážený binární strom o  $n$  vrcholech má výšku zhruba  $\log_2 n$ .

Ilustrační příklad: Kdybychom nakreslili vyvážený strom o  $n = 1\,000\,000$  vrcholech tak, aby sousední vrcholy byly vzdáleny o 1 cm a výška každé vrstvy byla také 1 cm, měl by tento strom na šířku 10 km a na výšku zhruba 20 cm.

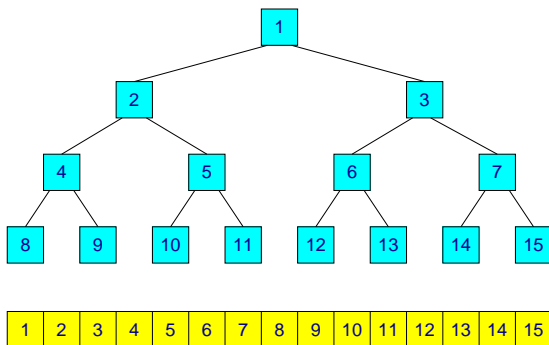
Dokonale vyvážený binární strom výšky  $h$ :



Dokonale vyvážený binární strom výšky  $h$ :

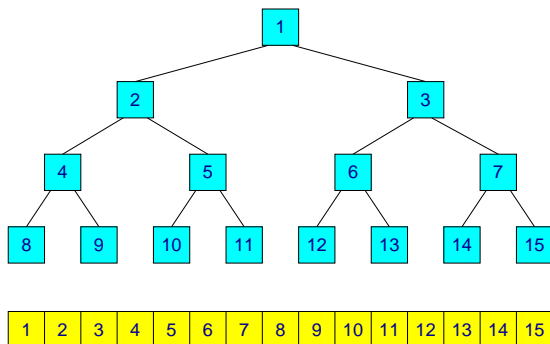


Efektivní uložení úplného binárního stromu v poli:





Efektivní uložení úplného binárního stromu v poli:



Potomci vrcholu s indexem  $i$  mají indexy  $2i$  a  $2i + 1$ .  
Rodič vrcholu s indexem  $i$  má index  $\lfloor i/2 \rfloor$ .

**Halda (heap)** — úplný binární strom uložený v poli  $A$  výše uvedeným způsobem, kde navíc pro každé  $i = 1, 2, \dots, n$  platí:

- pokud  $2i \leq n$ , pak  $A[i] \leq A[2i]$
- pokud  $2i + 1 \leq n$ , pak  $A[i] \leq A[2i + 1]$

Příklady využití haldy:

- třídící algoritmus **HeapSort**
- efektivní implementace **prioritní fronty** — umožňuje provádět většinu operací na této frontě s časovou složitostí v  $O(\log n)$ , kde  $n$  je počet prvků ve frontě

---

**Algoritmus:** Vytvoření haldy z neseříděného pole

---

CREATE-HEAP ( $A, n$ ):

```
 $i := \lfloor n/2 \rfloor$   
while  $i \geq 1$  do  
   $j := i$   
   $x := A[j]$   
  while  $2 * j \leq n$  do  
     $k := 2 * j$   
    if  $k + 1 \leq n$  and  $A[k + 1] < A[k]$  then  
       $k := k + 1$   
    if  $x \leq A[k]$  then break  
     $A[j] := A[k]$   
     $j := k$   
   $A[j] := x$   
   $i := i - 1$ 
```

## Časová složitost algoritmu `CREATE-HEAP`:

- Rychlou a hrubou analýzou lehce zjistíme, že tato složitost je v  $O(n \log n)$  a v  $\Omega(n)$ :
  - Vnější cyklus se provede vždy  $\lfloor n/2 \rfloor$  krát — počet průchodů je tedy v  $\Theta(n)$ .
  - Počet průchodů vnitřním cyklem v rámci jedné iterace vnějšího cyklu je očividně v  $O(\log n)$ .
- Daleko méně zřejmé je, že celkový počet průchodů vnitřním cyklem (tj. dohromady přes všechny iterace vnějšího cyklu) je ve skutečnosti v  $O(n)$ .

Celkově tedy dostáváme:

Časová složitost algoritmu `CREATE-HEAP` je v  $\Theta(n)$ .

# Složitost algoritmů

Zdůvodnění toho, proč je počet průchodů vnitřním cyklem v  $O(n)$ :

Předpokládejme pro jednoduchost, že všechny větve stromu jsou stejně dlouhé a mají délku  $h$  — platí tedy  $n = 2^{h+1} - 1$ .

Označme  $C_i$ , kde  $0 \leq i < h$ , celkový počet průchodů vnitřním cyklem, kdy je na začátku cyklu hodnota  $j$  v  $i$ -té vrstvě stromu (vrstvy jsou číslovány odshora  $0, 1, 2, \dots$ ).

Zjevně je celkový počet průchodů  $s$  dán vztahem

$$s = C_{h-1} + C_{h-2} + \dots + C_0 = \sum_{i=0}^{h-1} C_i$$

Hodnotu  $C_i$  spočítáme jako celkový počet vrcholů ve vstvách  $0, 1, \dots, i$ :

$$C_i = 2^0 + 2^1 + \dots + 2^i = \sum_{k=0}^i 2^k = \frac{2^{i+1} - 1}{2 - 1} = 2^{i+1} - 1$$

Celkový součet pak spočítáme následovně:

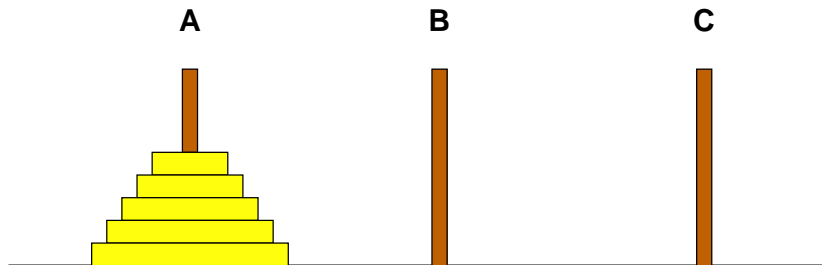
$$\begin{aligned} s &= \sum_{i=0}^{h-1} C_i = \sum_{i=0}^{h-1} (2^{i+1} - 1) = 2 \cdot \left( \sum_{i=0}^{h-1} 2^i \right) - \left( \sum_{i=0}^{h-1} 1 \right) \\ &= 2 \cdot \frac{2^h - 1}{2 - 1} - h = 2^{h+1} - 2 - h = n - 1 - h = O(n) \end{aligned}$$

**Rekurzivní algoritmus** je algoritmus, který převede řešení původního problému na řešení několika podobných problémů pro menší instance.

Obecné schéma rekurzivních algoritmů:

- Pokud se jedná o elementární případ, vyřeš ho přímo a vrať výsledek.
- V opačném případě vytvoř instance podproblémů.
- Zavolej sám sebe pro každou z těchto instancí.
- Z výsledků pro jednotlivé podproblémy slož řešení původního problému a vrať ho jako výsledek.

**Poznámka:** Instance podproblémů musí vždy být v nějakém smyslu menší než instance původního problému. Často (ne však vždy) se zmenšuje velikost instance.



**Úkol:** Přemístit disky z A na B, přičemž:

- V jednom okamžiku je možné přesouvat jen jeden disk.
- Není dovoleno položit větší disk na menší.



$n = 1 :$

$A \rightarrow B$

$n = 1 :$

$A \rightarrow B$

$n = 2 :$

$A \rightarrow C$

$A \rightarrow B$

$C \rightarrow B$

$n = 1 :$

$A \rightarrow B$

$n = 2 :$

$A \rightarrow C$

$A \rightarrow B$

$C \rightarrow B$

$n = 3 :$

$A \rightarrow B$

$A \rightarrow C$

$B \rightarrow C$

$A \rightarrow B$

$C \rightarrow A$

$C \rightarrow B$

$A \rightarrow B$

# Hanojské věže

$n = 1 :$

$A \rightarrow B$

$n = 2 :$

$A \rightarrow C$

$A \rightarrow B$

$C \rightarrow B$

$n = 3 :$

$A \rightarrow B$

$A \rightarrow C$

$B \rightarrow C$

$A \rightarrow B$

$C \rightarrow A$

$C \rightarrow B$

$A \rightarrow B$

$n = 4 :$

$A \rightarrow C$

$A \rightarrow B$

$C \rightarrow B$

$A \rightarrow C$

$B \rightarrow A$

$B \rightarrow C$

$A \rightarrow C$

$A \rightarrow B$

$C \rightarrow B$

$C \rightarrow A$

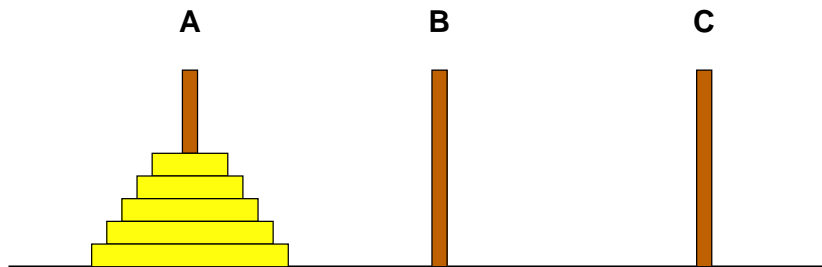
$B \rightarrow A$

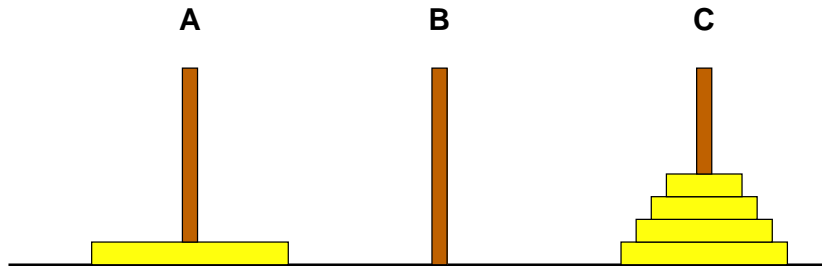
$C \rightarrow B$

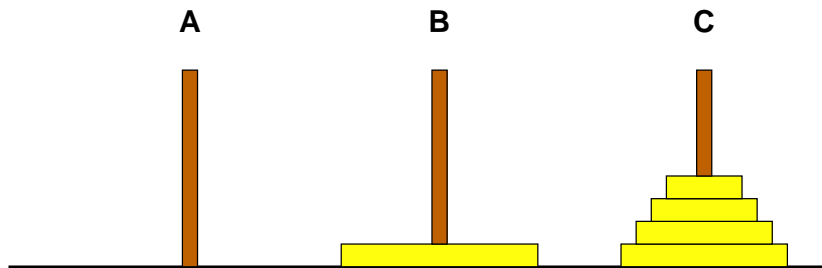
$A \rightarrow C$

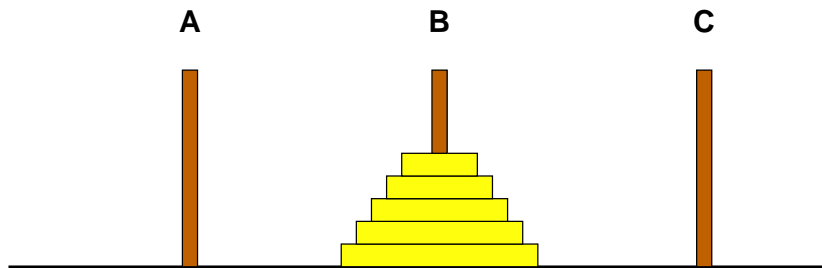
$A \rightarrow B$

$C \rightarrow B$











---

## Algoritmus: Hanojské věže

---

HANOI( $n$ ,  $src$ ,  $dst$ ,  $tmp$ ):

```
  if  $n = 0$  then return  
  HANOI( $n - 1$ ,  $src$ ,  $tmp$ ,  $dst$ )  
  print( $src$ , "→",  $dst$ )  
  HANOI( $n - 1$ ,  $tmp$ ,  $dst$ ,  $src$ )
```

MAIN( $n$ ):

```
  HANOI( $n$ , "A", "B", "C")
```

---

Označme  $P(n)$  počet tahů, které provede algoritmus pro  $n$  disků.

## Tvrzení

$$P(n) = 2^n - 1.$$

## Důkaz:

- Pro  $n = 0$ :  $P(n) = 0 = 2^0 - 1$
- Pro  $n > 0$ : Předpokládáme, že  $P(n - 1) = 2^{n-1} - 1$ .

$$\begin{aligned}P(n) &= 2P(n - 1) + 1 = \\&= 2(2^{n-1} - 1) + 1 = \\&= 2 \cdot 2^{n-1} - 2 + 1 = \\&= 2^n - 1\end{aligned}$$

## Tvrzení

Pro přesun  $n$  disků je třeba minimálně  $2^n - 1$  tahů.

## Důkaz:

Indukcí.

Uvedený algoritmus nalezne tedy optimální řešení.

## Poznámka

**Otázka:** Jak dlouho by trvalo přesunutí 64 disků, pokud by přesunutí jednoho disku trvalo 1 s?

## Tvrzení

Pro přesun  $n$  disků je třeba minimálně  $2^n - 1$  tahů.

## Důkaz:

Indukcí.

Uvedený algoritmus nalezne tedy optimální řešení.

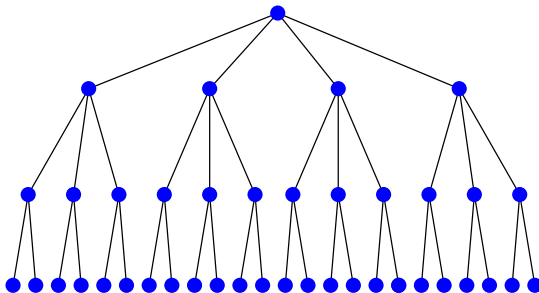
## Poznámka

**Otázka:** Jak dlouho by trvalo přesunutí 64 disků, pokud by přesunutí jednoho disku trvalo 1 s?

**Odpověď:** 18446744073709551615 s, tj. asi 585 miliard let.

Výpočet rekurzivního algoritmu je možné znázornit jako strom:

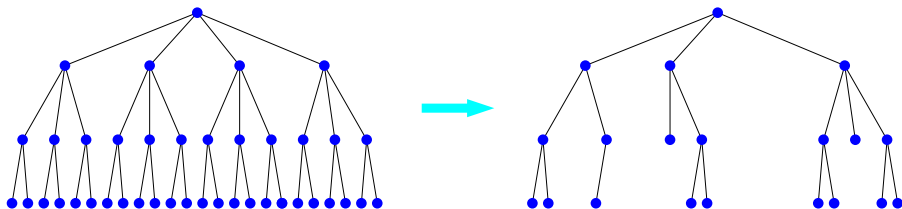
- vrcholy stromu odpovídají jednotlivým podproblémům
- kořen je původní problém
- potomci vrcholu odpovídají podproblémům daného problému



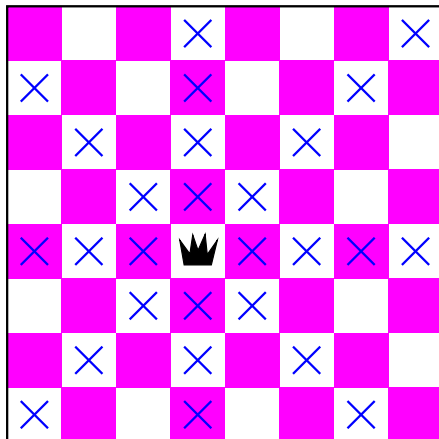
# Pruning

Přístup, kdy do rekurzivním algoritmu doplníme nějaké vhodné testy, které způsobí, že se rekurzivní procedura nebude volat pro některé podproblémy, u kterých je jisté, že jejich vyřešení nepovede k řešení celého problému, se nazývá **pruning**.

Čím více větví stromu tímto způsobem odstraníme, tím lépe.



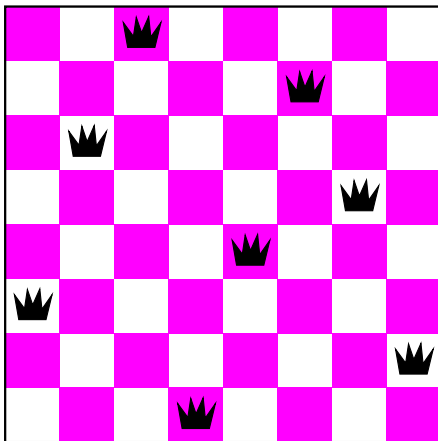
# Problém osmi dam



**Problém:** Najít všechny možnosti, jak rozmístit  $n$  dam na šachovnici velikosti  $n \times n$  tak, aby se žádné dvě dámy navzájem neohrožovaly.

# Problém osmi dam

Jedno z možných řešení.





# Problém osmi dam

Vstupem je číslo  $n$ .

Algoritmus bude používat následující pole:

- $Y$  — s indexy  $0 \dots n - 1$ , hodnota  $Y[i]$  udává pro dámu ve sloupci  $i$  číslo řádku, na kterém se nachází
- $A$  — s indexy  $0 \dots n - 1$ , booleovská hodnota  $A[j]$  udává, jestli je řádek  $j$  obsazený
- $B$  — s indexy  $0 \dots 2n - 2$ , booleovská hodnota  $B[k]$  udává, jestli je diagonála  $k$  ve směru ↗ obsazená
- $C$  — s indexy  $-(n - 1) \dots +(n - 1)$ , booleovská hodnota  $C[k]$  udává, jestli je diagonála  $k$  ve směru ↘ obsazená

Prohledávání se spustí zavoláním `SEARCH(0)`.

---

**Algoritmus:** Rozmístění  $n$  dam na šachovnici

---

SEARCH ( $k$ ):

```
if  $k = n$  then
  PRINT-SOLUTION ()
  return
for  $i := 0$  to  $n - 1$  do
   $i_2 := k + i$ ;  $i_3 := k - i$ 
  if  $\neg A[i]$  and  $\neg B[i_2]$  and  $\neg C[i_3]$  then
     $Y[k] := i$ 
     $A[i] := \text{TRUE}$ ;  $B[i_2] := \text{TRUE}$ ;  $C[i_3] := \text{TRUE}$ 
    SEARCH ( $k + 1$ )
     $A[i] := \text{FALSE}$ ;  $B[i_2] := \text{FALSE}$ ;  $C[i_3] := \text{FALSE}$ 
```

---

# Problém osmi dam

	0	1	2	3
0	0	1	2	3
1	1	2	3	4
2	2	3	4	5
3	3	4	5	6

$x+y$

	0	1	2	3
0	0	1	2	3
1	-1	0	1	2
2	-2	-1	0	1
3	-3	-2	-1	0

$x-y$

- Uvedený algoritmus pro rozmístění  $n$  dam na šachovnici je příkladem **prohledávání s návratem (backtracking)**.
- I při použití pruningu má tento typ algoritmů většinou exponenciální složitost.
- Obecně rekurzivní algoritmy, které převádí řešení problému velikosti  $n$  na dva nebo více problémů velikosti  $n - 1$ , mívají většinou exponenciální složitost.
- Pokud rekurzivní algoritmus převádí řešení problému velikosti  $n$  na řešení problémů velikosti  $n/2$ , složitost může být (a často bývá) polynomiální.

Tento postup může někdy vést k řešením, která mohou být efektivnější než nějaké přímočaré řešení.

**Příklad:** Algoritmus **MERGE-SORT**.

Hlavní myšlenka algoritmu: Dvě setříděné posloupnosti snadno spojíme do jediné setříděné posloupnosti.

Pokud mají obě posloupnosti dohromady  $n$  prvků, vyžaduje tato operace  $n$  kroků.

34	42	58	61
----	----	----	----

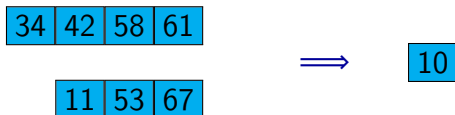


10	11	53	67
----	----	----	----

**Příklad:** Algoritmus **MERGE-SORT**.

Hlavní myšlenka algoritmu: Dvě setříděné posloupnosti snadno spojíme do jediné setříděné posloupnosti.

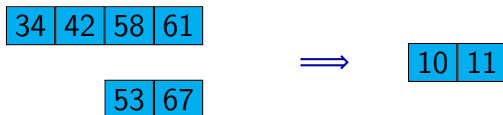
Pokud mají obě posloupnosti dohromady  $n$  prvků, vyžaduje tato operace  $n$  kroků.



**Příklad:** Algoritmus **MERGE-SORT**.

Hlavní myšlenka algoritmu: Dvě setříděné posloupnosti snadno spojíme do jediné setříděné posloupnosti.

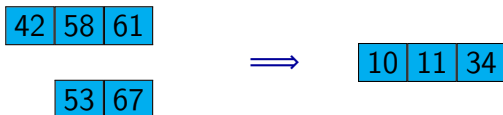
Pokud mají obě posloupnosti dohromady  $n$  prvků, vyžaduje tato operace  $n$  kroků.



**Příklad:** Algoritmus **MERGE-SORT**.

Hlavní myšlenka algoritmu: Dvě setříděné posloupnosti snadno spojíme do jediné setříděné posloupnosti.

Pokud mají obě posloupnosti dohromady  $n$  prvků, vyžaduje tato operace  $n$  kroků.

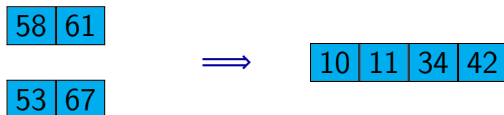




**Příklad:** Algoritmus **MERGE-SORT**.

Hlavní myšlenka algoritmu: Dvě setříděné posloupnosti snadno spojíme do jediné setříděné posloupnosti.

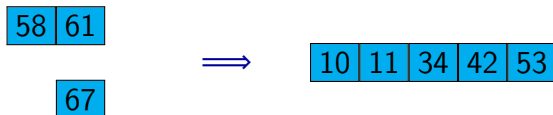
Pokud mají obě posloupnosti dohromady  $n$  prvků, vyžaduje tato operace  $n$  kroků.



**Příklad:** Algoritmus **MERGE-SORT**.

Hlavní myšlenka algoritmu: Dvě setříděné posloupnosti snadno spojíme do jediné setříděné posloupnosti.

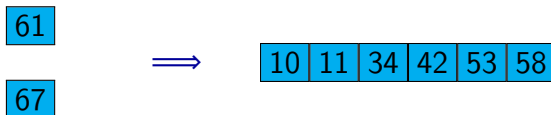
Pokud mají obě posloupnosti dohromady  $n$  prvků, vyžaduje tato operace  $n$  kroků.



**Příklad:** Algoritmus **MERGE-SORT**.

Hlavní myšlenka algoritmu: Dvě setříděné posloupnosti snadno spojíme do jediné setříděné posloupnosti.

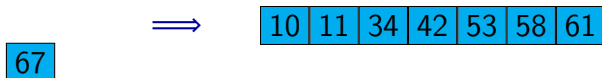
Pokud mají obě posloupnosti dohromady  $n$  prvků, vyžaduje tato operace  $n$  kroků.



**Příklad:** Algoritmus **MERGE-SORT**.

Hlavní myšlenka algoritmu: Dvě setříděné posloupnosti snadno spojíme do jediné setříděné posloupnosti.

Pokud mají obě posloupnosti dohromady  $n$  prvků, vyžaduje tato operace  $n$  kroků.



**Příklad:** Algoritmus **MERGE-SORT**.

Hlavní myšlenka algoritmu: Dvě setříděné posloupnosti snadno spojíme do jediné setříděné posloupnosti.

Pokud mají obě posloupnosti dohromady  $n$  prvků, vyžaduje tato operace  $n$  kroků.

⇒ 

10	11	34	42	53	58	61	67
----	----	----	----	----	----	----	----

---

**Algoritmus:** Merge sort

---

MERGE-SORT ( $A, p, r$ ):

**if**  $r - p > 1$  **then**

$q := \lfloor (p + r) / 2 \rfloor$

    MERGE-SORT( $A, p, q$ )

    MERGE-SORT( $A, q, r$ )

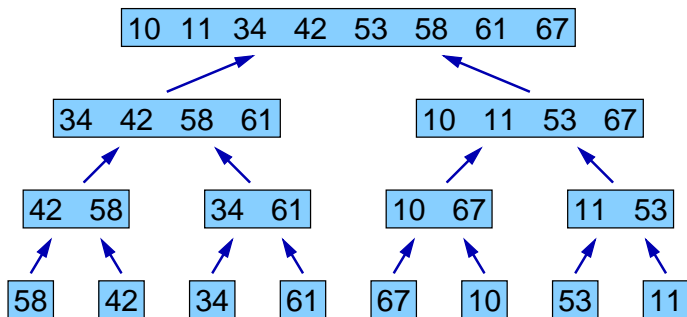
    MERGE( $A, p, q, r$ )

---

Pro setřídění pole  $A$ , které obsahuje prvky  $A[0], A[1], \dots, A[n - 1]$ , zavoláme  $\text{MERGE-SORT}(A, 0, n)$ .

**Poznámka:** Procedura  $\text{MERGE}(A, p, q, r)$  spojí setříděné posloupnosti uložené v  $A[p \dots q - 1]$  a  $A[q \dots r - 1]$  do jedné posloupnosti uložené v  $A[p \dots r - 1]$ .

**Vstup:** 58, 42, 34, 61, 67, 10, 53, 11



Strom rekurzivních volání má  $\Theta(\log n)$  úrovní. Na každé úrovni se provede  $\Theta(n)$  operací. Časová složitost algoritmu **MERGE-SORT** je  $\Theta(n \log n)$ .

## Master theorem

Předpokládejme, že  $a \geq 1$  a  $b > 1$  jsou konstanty, že  $f(n)$  je funkce a že funkce  $T(n)$  je definována rekurentním předpisem

$$T(n) = a \cdot T(n/b) + f(n)$$

(kde  $n/b$  může být buď  $\lfloor n/b \rfloor$  nebo  $\lceil n/b \rceil$ ). Pak platí:

- a Pokud  $f(n) \in O(n^{\log_b a - \epsilon})$  pro nějakou konstantu  $\epsilon > 0$ , pak  $T(n) = \Theta(n^{\log_b a})$ .
- b Pokud  $f(n) \in \Theta(n^{\log_b a})$ , pak  $T(n) = \Theta(n^{\log_b a} \log n)$ .
- c Pokud  $f(n) \in \Omega(n^{\log_b a + \epsilon})$  pro nějakou konstantu  $\epsilon > 0$  a pokud  $a \cdot f(n/b) \leq c \cdot f(n)$  pro nějakou konstantu  $c < 1$  a všechna dostatečně velká  $n$ , pak  $T(n) = \Theta(f(n))$ .



# The master theorem

Master theorem je možné použít pro analýzu složitosti libovolného rekurzivního algoritmu, kde:

- Řešení jednoho podproblému velikosti  $n$ , kde  $n > 1$ , se převede na řešení  $a$  podproblémů, z nichž každý má velikost  $n/b$ .
- Doba, která stráví řešením jednoho podproblému velikosti  $n$ , nepočítaje v to dobu, která se stráví v rekurzivních voláních, je určená funkcí  $f(n)$ .

**Příklad:** Algoritmus **MERGE-SORT**:  $a = 2$ ,  $b = 2$ ,  $f(n) \in \Theta(n)$

(v rámci jednoho volání — dva podproblémy, každý velikosti  $n/2$ , spojení dvou setříděných sekvencí v čase  $\Theta(n)$ )

Platí  $f(n) \in \Theta(n^{\log_b a}) = \Theta(n)$ , takže

$$T(n) \in \Theta(n^{\log_b a} \log n) = \Theta(n \log n).$$

# The master theorem

**Příklad:** Násobení čtvercových matic  $A$  a  $B$  velikosti  $n \times n$  rekurzivním způsobem:

- Pro  $n = 1$  se výsledek spočítá přímo.
- Pro  $n > 1$  se každá z matic  $A$  a  $B$  rozloží na čtyři podmatice velikosti  $(n/2) \times (n/2)$ .
- Výsledek se poskládá pomocí sčítání a násobení těchto osmi menších matic. Pro násobení těchto menších matic se funkce zavolá rekurzivně.
- Přímočarý způsob vyžaduje 8 násobení matic velikosti  $(n/2) \times (n/2)$ .

Máme tedy  $a = 8$ ,  $b = 2$ ,  $f(n) \in \Theta(n^2)$ .

Platí  $f(n) \in O(n^{\log_b a - \epsilon})$ , protože  $n^2 \in O(n^{\log_2 8 - \epsilon}) = O(n^{3 - \epsilon})$  platí např. pro  $\epsilon = 1$ .

Takže  $T(n) \in \Theta(n^{\log_b a}) = \Theta(n^{\log_2 8}) = \Theta(n^3)$ .

Tento postup tedy není lepší než standardní jednoduchý algoritmus pro násobení matic.

# The master theorem

Existuje však chytrý způsob, jak výše uvedené provést komplikovanějším způsobem tak, že v rámci jednoho rekurzivního volání postačí rekurzivně volat funkci 7 krát  
(za cenu většího počtu sčítání a odčítání).

Jedná se o tzv. **Strassenův algoritmus**.

Zde je  $a = 7$ ,  $b = 2$  a  $f(n) \in \Theta(n^2)$ .

Opět platí  $f(n) \in O(n^{\log_b a - \epsilon})$ , protože  $n^2 \in O(n^{\log_2 7 - \epsilon})$  platí např. pro  $\epsilon = 0.5$ .

( $\log_2 7$  je přibližně 2.80735)

Takže  $T(n) \in \Theta(n^{\log_b a}) = \Theta(n^{\log_2 7})$  a tedy  $T(n) \in O(n^{2.81})$ .

## Důkaz master theoremu:

Pro jednoduchost se omezíme jen na případy, kdy  $f(n) = n^c$  pro nějakou konstantu  $c > 0$ .

Rovněž pro jednoduchost předpokládejme, že  $n$  je mocninou čísla  $b$ , ať nemusíme řešit zaokrouhlování.

Představme si strom rekurzivních volání pro instanci velikosti  $n$ :

- Výška stromu je  $\log_b n$ .
- Počty vrcholů na jednotlivých úrovních jsou  $a^0, a^1, \dots, a^{\log_b n}$
- Čas, který se stráví v jednom vrcholu na úrovni  $i$  je

$$f\left(\frac{n}{b^i}\right) = \left(\frac{n}{b^i}\right)^c$$

# The master theorem

Platí tedy

$$T(n) = \sum_{i=0}^{\log_b n} a^i \cdot f\left(\frac{n}{b^i}\right) = \sum_{i=0}^{\log_b n} a^i \cdot \left(\frac{n}{b^i}\right)^c = n^c \cdot \sum_{i=0}^{\log_b n} \left(\frac{a}{b^c}\right)^i$$

Označme  $q = a/b^c$ . Je třeba rozlišit tři případy:

- $q > 1$  — tj. když platí  $a > b^c$ , neboli  $c < \log_b a$
- $q = 1$  — tj. když platí  $a = b^c$ , neboli  $c = \log_b a$
- $q < 1$  — tj. když platí  $a < b^c$ , neboli  $c > \log_b a$

# The master theorem

Případ  $q > 1$  — tj. když platí  $a > b^c$ , neboli  $c < \log_b a$ :

$$T(n) = n^c \cdot \sum_{i=0}^{\log_b n} \left(\frac{a}{b^c}\right)^i = n^c \cdot \frac{q^{\log_b n + 1} - 1}{q - 1} \in \Theta(n^c \cdot q^{\log_b n})$$

Platí

$$\begin{aligned} n^c \cdot q^{\log_b n} &= n^c \cdot \left(\frac{a}{b^c}\right)^{\log_b n} = n^c \cdot n^{\log_b \left(\frac{a}{b^c}\right)} \\ &= n^c \cdot n^{\log_b a - \log_b(b^c)} = n^{c + \log_b a - c} = n^{\log_b a} \end{aligned}$$

Platí tedy  $T(n) \in \Theta(n^{\log_b a})$ .

**Poznámka:** Počet listů stromů (tj. podproblémů velikosti 1) je  $a^{\log_b n} = n^{\log_b a}$ .

Většina času se tedy tráví řešením těchto elementárních případů.

# The master theorem

Případ  $q = 1$  — tj. když platí  $a = b^c$ , neboli  $c = \log_b a$ :

$$T(n) = n^c \cdot \sum_{i=0}^{\log_b n} \left(\frac{a}{b^c}\right)^i = n^c \cdot \sum_{i=0}^{\log_b n} 1 = n^c \cdot (\log_b n + 1) \in \Theta(n^{\log_b a} \log n)$$

**Poznámka:** V každé vrstvě stromu se stráví zhruba stejný čas  $\Theta(n^{\log_b a})$ .  
Vrstev je celkem  $\Theta(\log n)$ .

# The master theorem

Případ  $q < 1$  — tj. když platí  $a < b^c$ , neboli  $c > \log_b a$ :

$$T(n) = n^c \cdot \sum_{i=0}^{\log_b n} \left(\frac{a}{b^c}\right)^i < n^c \cdot \sum_{i=0}^{\infty} \left(\frac{a}{b^c}\right)^i = n^c \cdot \frac{1}{1-q} \in O(n^c)$$

protože pro  $q$ , kde  $0 < q < 1$ , platí

$$\sum_{i=0}^{\infty} q^i = \lim_{z \rightarrow \infty} \sum_{i=0}^z q^i = \lim_{z \rightarrow \infty} \frac{q^{z+1} - 1}{q - 1} = \frac{1}{1 - q}$$

Zjevně platí  $T(n) \in \Omega(n^c)$  (protože už v samotném kořeni se stráví čas  $\Theta(n^c)$ ), takže celkově platí  $T(n) \in \Theta(n^c)$ .

**Poznámka:** Většina času se v tomto případě stráví v kořeni stromu.



## Definice

Slovo  $v = a_1 a_2 \dots a_n$  nad abecedou  $\Sigma$  (kde všechna  $a_i \in \Sigma$ ) je **podsekvencí** slova  $w \in \Sigma^*$ , jestliže existují slova  $u_0, u_1, \dots, u_n \in \Sigma^*$  taková, že

$$w = u_0 a_1 u_1 a_2 u_2 \cdots u_{n-1} a_n u_n$$

**Příklad:** Slovo  $bcba$  je podsekvencí slova  $abc b d c a b$ .

**Nejdelší společná podsekvence (longest common subsequence)** slov  $u$  a  $v$  je nejdelší slovo  $w$ , které je podsekvencí slova  $u$  a zároveň podsekvencí slova  $v$ .

**Příklad:** Nejdelší společnou podsekvencí slov  $abc b d a b$  a  $b d c a b a$  je slovo  $bcba$ .

**Poznámka:** Nejdelší společná podsekvence vždy existuje, ale ne vždy je dána jednoznačně — např. pro  $aaabb$  a  $bbbaa$  je to  $aa$  i  $bb$ .

## Problém: Nejdelší společná podsekvence

**Vstup:** Slova  $u$  a  $v$  nad abecedou  $\Sigma$ .

**Výstup:** Nejdelší slovo  $w$ , které je podsekvencí slova  $u$  a zároveň podsekvencí slova  $v$ .

Předpokládejme, že:

- slova  $u$  a  $v$  jsou uložena v polích  $A$  a  $B$  indexovaných od jedné
- hodnoty  $m$  a  $n$  udávají délku slov  $u$  a  $v$

Tj. pokud  $u = a_1a_2\cdots a_m$  a  $v = b_1b_2\cdots b_n$ , tak:

- prvky  $A[1], A[2], \dots, A[m]$  obsahují symboly  $a_1, a_2, \dots, a_m$
- prvky  $B[1], B[2], \dots, B[n]$  obsahují symboly  $b_1, b_2, \dots, b_n$

# Dynamické programování

Zaměříme se nejprve na problém, zjistit pro daná slova  $u$  a  $v$ , jaká je délka jejich nejdelší společné podsekvence.

Můžeme řešit rekurzivně podproblémy následujícího typu:

- $\text{LCS-LEN}(i, j)$  — pro dané  $i$  a  $j$ , kde  $0 \leq i \leq m$  a  $0 \leq j \leq n$ , vrátí délku nejdelší společné podsekvence prefixu slova  $u$  délky  $i$  a prefixu slova  $v$  délky  $j$ .

Tj.  $\text{LCS-LEN}(i, j)$  vrátí délku nejdelší společné podsekvence slov uložených v

$$A[1], A[2], \dots, A[i] \quad \text{a} \quad B[1], B[2], \dots, B[j]$$

Délku nejdelší společné podsekvence slov  $u$  a  $v$  pak můžeme zjistit pomocí  $\text{LCS-LEN}(m, n)$ .

Rekurzivní řešení:

$$\text{LCS-LEN}(i, j) = \begin{cases} 0 & \text{pokud } i = 0 \text{ nebo } j = 0 \\ \text{LCS-LEN}(i - 1, j - 1) + 1 & \text{pokud } A[i] = B[j] \\ \max(\text{LCS-LEN}(i - 1, j), \\ \text{LCS-LEN}(i, j - 1)) & \text{pokud } A[i] \neq B[j] \end{cases}$$

Rekurzivní řešení:

$$\text{LCS-LEN}(i, j) = \begin{cases} 0 & \text{pokud } i = 0 \text{ nebo } j = 0 \\ \text{LCS-LEN}(i-1, j-1) + 1 & \text{pokud } A[i] = B[j] \\ \max(\text{LCS-LEN}(i-1, j), \\ \text{LCS-LEN}(i, j-1)) & \text{pokud } A[i] \neq B[j] \end{cases}$$

Toto řešení má očividně exponenciální časovou složitost.

Ve skutečnosti potřebujeme řešit jen  $(m+1) \cdot (n+1)$  **různých** podproblémů (protože  $i \in \{0, 1, \dots, m\}$  a  $j \in \{0, 1, \dots, n\}$ ).

Výsledky řešení jednotlivých podproblémů si můžeme ukládat do tabulky a nemusíme je řešit opakovaně.

---

**Algoritmus:** Nalezení nejdelší společné podsekvence — vyplnění tabulky

---

LCS-COMP ( $A, m, B, n$ ):

```
for  $i := 0$  to  $m$  do  $C[i][0] := 0$ 
for  $j := 1$  to  $n$  do  $C[0][j] := 0$ 
for  $i := 1$  to  $m$  do
  for  $j := 1$  to  $n$  do
    if  $A[i] = B[j]$  then
      |  $C[i][j] := C[i-1][j-1] + 1$ ;  $D[i][j] := "↖"$ 
    else
      | if  $C[i-1][j] \leq C[i][j-1]$  then
      | |  $C[i][j] := C[i-1][j]$ ;  $D[i][j] := "↑"$ 
      | else
      | |  $C[i][j] := C[i][j-1]$ ;  $D[i][j] := "←"$ 
```

---

Složitost algoritmu je  $O(m \cdot n)$ .

# Dynamické programování

<i>i</i>	<i>j</i>	0	1	2	3	4	5	6
			<i>b</i>	<i>d</i>	<i>c</i>	<i>a</i>	<i>b</i>	<i>a</i>
0		0	0	0	0	0	0	0
1	<i>a</i>	0	↑	↑	↑	↖	←1	↖
2	<i>b</i>	0	↖	←1	←1	↑	↖	←2
3	<i>c</i>	0	↑	↑	↖	←2	↑	↑
4	<i>b</i>	0	↖	↑	↑	↑	↖	←3
5	<i>d</i>	0	↑	↖	↑	↑	↑	↑
6	<i>a</i>	0	↑	↑	↑	↖	↑	↖
7	<i>b</i>	0	↖	↑	↑	↑	↖	↑

## Dynamické programování:

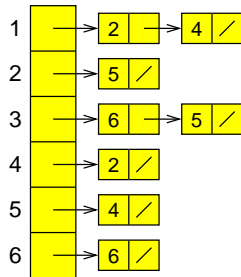
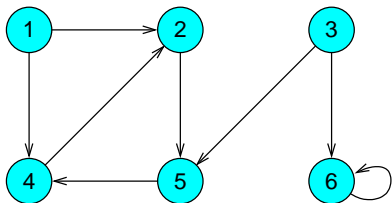
- Pokud máme rekurzivní řešení, kde se opakovaně mnohokrát řeší stejné instance podproblémů, je vhodné ukládat řešení podproblémů do nějaké datové struktury.
- Jedna možnost je ponechat rekurzivní řešení a značit si, které podproblémy již byly vyřešeny.
- Jiná možnost je systematické řešení všech podproblémů ve vhodném pořadí (od nejmenších po největší).

Při řešení daného podproblému jsou rekurzivní volání nahrazena přechtením již dříve uložených řešení.



# Reprezentace grafů

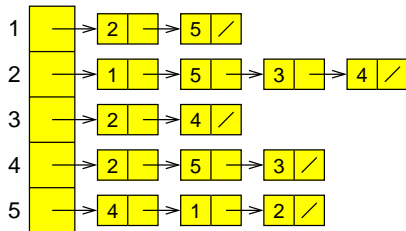
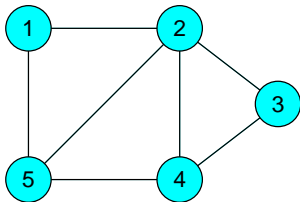
Reprezentace grafu:



	1	2	3	4	5	6
1	0	1	0	1	0	0
2	0	0	0	0	1	0
3	0	0	0	0	1	1
4	0	1	0	0	0	0
5	0	0	0	1	0	0
6	0	0	0	0	0	1

# Reprezentace grafů

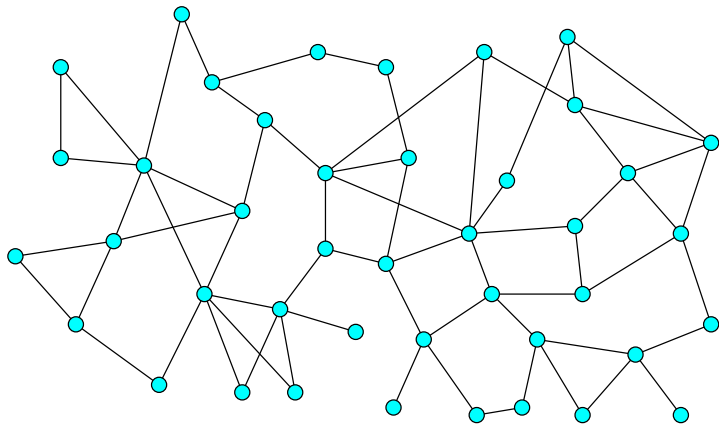
Reprezentace grafu:



	1	2	3	4	5
1	0	1	0	0	1
2	1	0	1	1	1
3	0	1	0	1	0
4	0	1	1	0	1
5	1	1	0	1	0

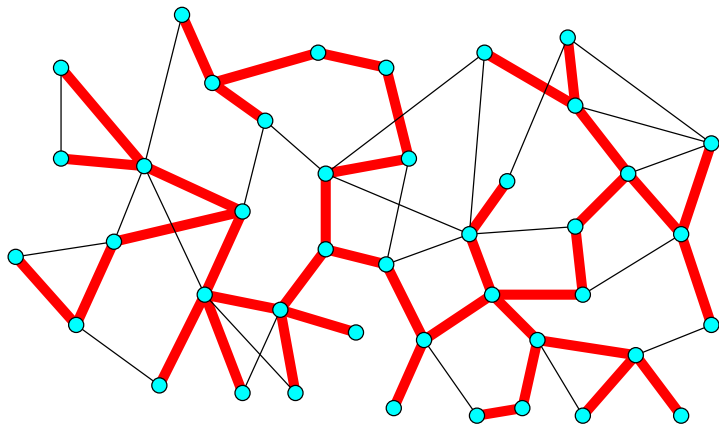
# Minimální kostra grafu

**Kostra grafu** — souvislý podgraf grafu, který obsahuje všechny vrcholy a neobsahuje žádné cykly



# Minimální kostra grafu

**Kostra grafu** — souvislý podgraf grafu, který obsahuje všechny vrcholy a neobsahuje žádné cykly



# Minimální kostra grafu

Uvažujme neorientovaný graf  $G = (V, E)$ , kde navíc máme dány váhy hran, tj. funkci  $w : E \rightarrow \mathbb{R}_+$ , přiřazující každé hraně její váhu.

Pokud  $T$  je podmnožina hran (tj.  $T \subseteq E$ ), můžeme funkci  $w$  rozšířit na tuto podmnožinu:

$$w(T) = \sum_{e \in T} w(e)$$

**Kostra** grafu  $G$  je dána takovou množinou hran  $T$  (kde  $T \subseteq E$ ), která splňuje to, že graf  $(V, T)$  je souvislý a neobsahuje žádný cyklus.

**Minimální kostra**  $T$  je taková kostra grafu  $G$ , kde pro libovolnou jinou kostru grafu  $T'$  platí

$$w(T) \leq w(T')$$

## Problém: Minimální kostra grafu

**Vstup:** Souvislý neorientovaný graf  $G = (V, E)$  s ohodnocením hran  $w : E \rightarrow \mathbb{R}_+$ .

**Výstup:** Některá minimální kostra grafu  $G$ .

Algoritmus, který by systematicky zkoušel všechny možné kostry, má očividně exponenciální složitost.

Efektivní algoritmy pro tento problém jsou založeny na tzv. **greedy** (**hltařím, hladověm**) řešení:

- na základě nějakého lokálního kritéria vybrat z mnoha možností jen jednu a nezkoušet všechny možnosti
  - z mnoha hran, které je možné do kostry přidat, vybrat vždy hranu s co nejmenší vahou, která nevytvoří cyklus, a tu přidat

## Kruskalův algoritmus:

- Setřídít hrany podle váhy od nejmenší po největší.

- $T := \emptyset$

- Probírat hrany v daném setříděném pořadí.

Pro každou hranu  $e$  otestovat, zda jejím přidáním do  $T$  nevznikne cyklus, pokud ne, nastavit

$$T := T \cup \{e\}$$

- Vrátit  $T$  jako výsledek.

Výpočetní složitost závisí na tom, jak je konkrétně implementováno testování toho, že nevznikne cyklus:

- Přímočaré řešení má složitost  $O(n \cdot m)$ .
- Existuje efektivní řešení se složitostí  $O(m \log n)$ .

U **greedy algoritmů** je obecně většinou nesložitější částí návrhu algoritmu **důkaz korektnosti** — zdůvodnění toho lokálně optimální volby skutečně vždy vedou ke globálně optimálnímu řešení.

U Kruskalova algoritmu může být důkaz založen na tom, že se udržuje následující invariant:

Aktuální množina  $T$  je podmnožinou hran nějaké minimální kostry  $T_0$ .



# Minimální kostra grafu

- Řekněme, že  $T$  je podmnožinou minimální kostry  $T_0$ , a algoritmus v následujícím kroku přidá hranu  $e$  takovou, že  $T \cup \{e\}$  není podmnožinou hran žádné minimální kostry.
- Přidáním hrany  $e$  do  $T_0$  vznikne cyklus.  
Tento cyklus musí obsahovat nějakou hranu  $e'$  takovou, že  $e' \notin T$  (jinak by přidáním  $e$  do  $T$  vznikl cyklus).  
Navíc musí platit  $w(e) \leq w(e')$  (jinak by algoritmus nevybral hranu  $e$ ).
- Množina  $T'_0 = (T_0 - \{e'\}) \cup \{e\}$  je rovněž kostra.  
Navíc zjevně platí  $w(T'_0) \leq w(T_0)$ , takže musí platit  $w(T'_0) = w(T_0)$  (jinak by kostra  $T_0$  nebyla minimální).
- Kostra  $T'_0$  je tedy minimální a platí  $T \cup \{e\} \subseteq T'_0$ , což je ve sporu s předpokladem, že  $T \cup \{e\}$  není podmnožinou hran žádné minimální kostry.

Nalezení nejkratší cesty v grafu, kde hrany nejsou ohodnoceny:

- Algoritmus pro **prohledávání grafu do šířky**
- Vstupem je graf  $G$  (s množinou vrcholů  $V$ ) a počáteční vrchol  $s$ .
- Algoritmus pro všechny vrcholy najde nejkratší cestu z vrcholu  $s$ .
- Pro graf, který má  $n$  vrcholů a  $m$  hran je doba výpočtu tohoto algoritmu  $\Theta(n + m)$ .

---

**Algoritmus:** Prohledávání do šířky

---

BFS ( $G, s$ ):

BFS-INIT( $G, s$ )

ENQUEUE( $Q, s$ )

**while**  $Q \neq \emptyset$  **do**

$u :=$  DEQUEUE( $Q$ )

**for each**  $v \in \text{edges}[u]$  **do**

**if**  $\text{color}[v] = \text{WHITE}$  **then**

$\text{color}[v] := \text{GRAY}$

$d[v] := d[u] + 1$

$\text{pred}[v] := u$

            ENQUEUE( $Q, v$ )

$\text{color}[u] := \text{BLACK}$

---

---

**Algoritmus:** Prohledávání do šířky — inicializace

---

BFS-INIT ( $G, s$ ):

**for** each  $u \in V - \{s\}$  **do**

$color[u] := \text{WHITE}$

$d[u] := \infty$

$pred[u] := \text{NIL}$

$color[s] := \text{GRAY}$

$d[s] := 0$

$pred[s] := \text{NIL}$

$Q := \emptyset$

---