Prostorová (paměťová) složitost algoritmů

- Zatím jsme se zajímali o čas, který potřebujeme k výpočtu
- Někdy bývá kritickou velikost paměti potřebné k provedení výpočtu.

Množství paměti použité strojem $\mathcal{M}$ při provádění výpočtu nad vstupem w může být např.:

- maximální počet bitů nutných pro uložení všech dat v každé jednotlivé konfiguraci
- maximální počet paměťových buněk použitých během výpočtu

Definice

Prostorová složitost algoritmu Alg běžícího na stroji $\mathcal{M}$ je funkce $S : \mathbb{N} \to \mathbb{N}$, kde $S(n)$ udává maximální množství paměti použité strojem $\mathcal{M}$ při výpočtech nad vstupy velikosti n.

Prostorová (paměťová) složitost algoritmů

- Pro konkrétní problém můžeme mít dva algoritmy takové, že jeden má menší prostorovou složitost a druhý zase časovou složitost.
- Je-li časová složitost algoritmu v $O(f(n))$ je i prostorová v $O(f(n))$ (počet použitých paměťových buněk nemůže být větší než počet provedených operací, protože v každém kroku se použije jen nějaký omezený počet buněk (omezený nějakou konstantou nezávislou na velikosti vstupu).
- Prostorová složitost může být mnohdy o dost menší než časová složitost — například paměťová složitost algoritmu Insertion-Sort je $\Theta(n)$, zatímco časová $\Theta(n^2)$.

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.)

$O(n)$	$O(n \log n)$	$O(n^2)$	$O(n^3)$
$1\,000\,000 - 100\,000\,000$	$100\,000 - 1\,000\,000$	$1000 - 10\,000$	$100 - 1000$

$2^{O(n)}$	$O(n!)$
$20 - 30$	$10 - 15$

Složitost algoritmů

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.

Složitost algoritmů

- 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.

Složitost algoritmů

Polynom — funkce tvaru

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

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

Příklady polynomů:

$$4n^3 - 2n^2 + 8n + 13 \qquad 2n + 1 \qquad n^{100}$$

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:

$$O(n) \qquad O(n \log n) \qquad O(n^2) \qquad O(n^5) \qquad O(\sqrt{n}) \qquad O(n^{100})$$

Složitost algoritmů

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

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

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

Složitost algoritmů

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.

Složitost algoritmů

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).

Složitost algoritmů

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

Algoritmus 1: Násobení matic

```
1  MATRIX-MULT (A, B, C, n):
2  begin
3      for i := 1 to n do
4          for j := 1 to n do
5              x := 0
6              for k := 1 to n do
7                  x := x + A[i][k] * B[k][j]
8              end
9              C[i][j] := x
10         end
11     end
12 end
```

Složitost algoritmů

- 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 $a_0, a_1, \ldots, a_{n-1}$, kde

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

kde d je nějaká konstanta nezávistlá na i.

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

Součet aritmetické posloupnosti:

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

Složitost algoritmů

Příklad:

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

Konkrétně například pro $n = 100$ je

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

Poznámka: Stačí si umědomit, že

$$1 + 2 + \cdots + 100 \;=\; (1 + 100) + (2 + 99) + \cdots + (50 + 51),$$

kde sčítáme celkem 50 dvojic, kde součet každé dvojice je 101.

Složitost algoritmů

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

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

kde q je nějaká konstanta nezávistlá na i.

Poznámka: V geometrické posloupnosti tedy pro všechna i platí $a_{i+1} = a_i \cdot q$ for each i

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

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

Složitost algoritmů

Příklad:

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

Speciálně pro $q = 2$:

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

Složitost algoritmů

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

Tvrzení

Pro libovolná $a, b > 1$ a libovolné $n > 0$ platí

$$\log_a n = \frac{\log_b n}{\log_b a}$$

Důkaz: Z $n = a^{\log_a n}$ plyne $\log_b n = \log_b(a^{\log_a n})$.
Protože $\log_b(a^{\log_a n}) = \log_a n \cdot \log_b a$, dostáváme $\log_b n = \log_a n \cdot \log_b a$,
z čehož plyne výše uvedený závěr. □

Z toho důvodu se při použití asymptotické notace základ logaritmu obvykle vynechává: například místo $\Theta(n \log_2 n)$ můžeme napsat $\Theta(n \log n)$.

Složitost algoritmů

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

- Nějaká 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 $\Theta(\log n)$.

Složitost algoritmů

- 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

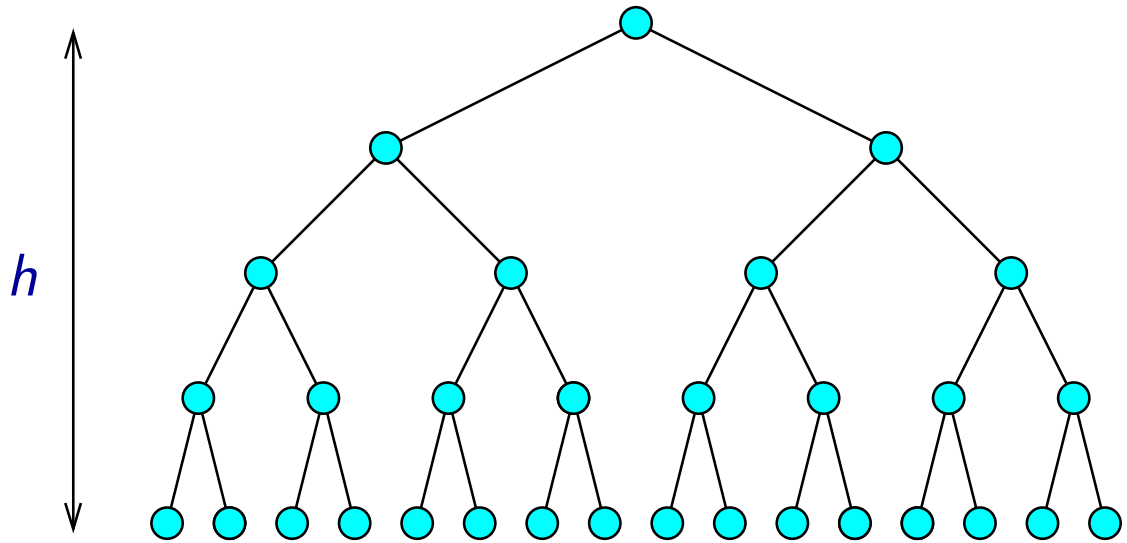
$$\lceil \log_2(x+1) \rceil.$$

- 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.

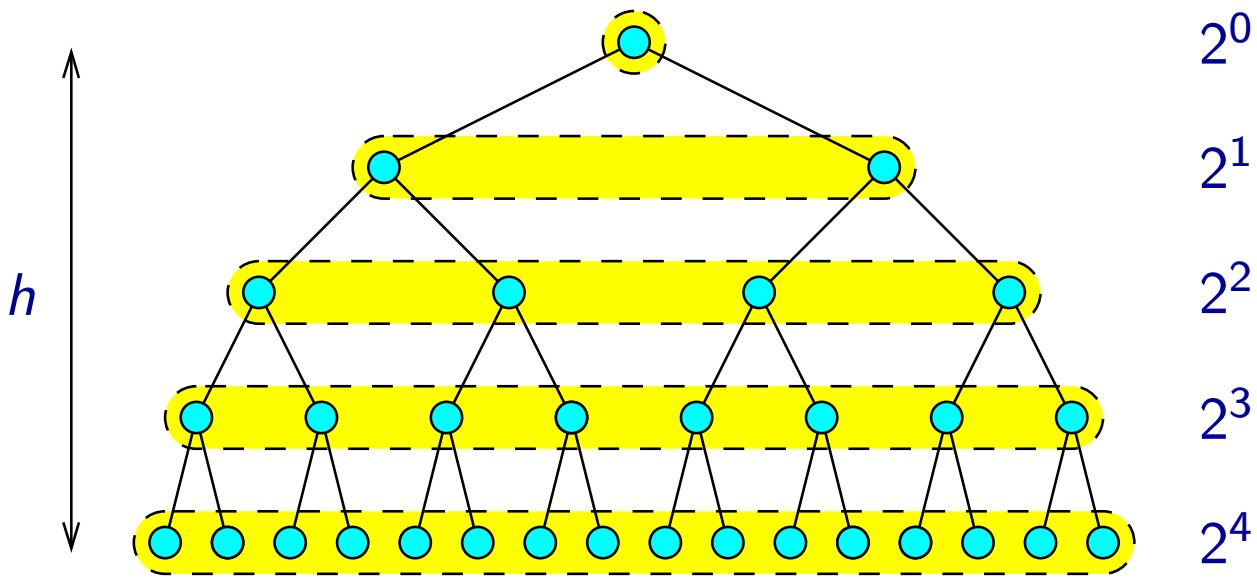
Složitost algoritmů

Dokonale vyvážený binární strom výšky h:



Složitost algoritmů

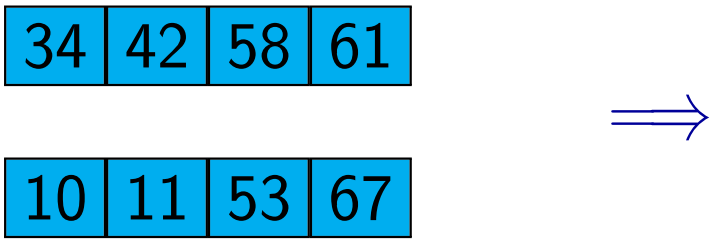
Dokonale vyvážený binární strom výšky h:



Složitost algoritmů

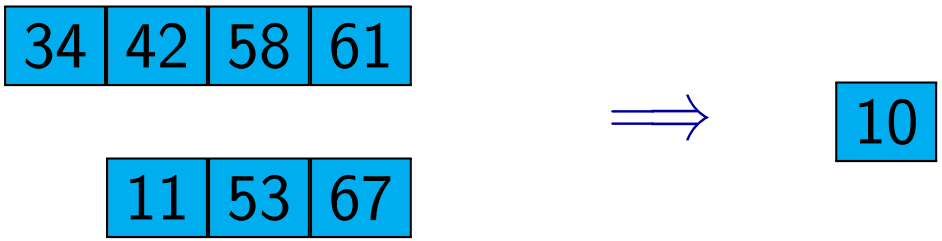
Příklad: Algoritmus MERGE-SORT.

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.

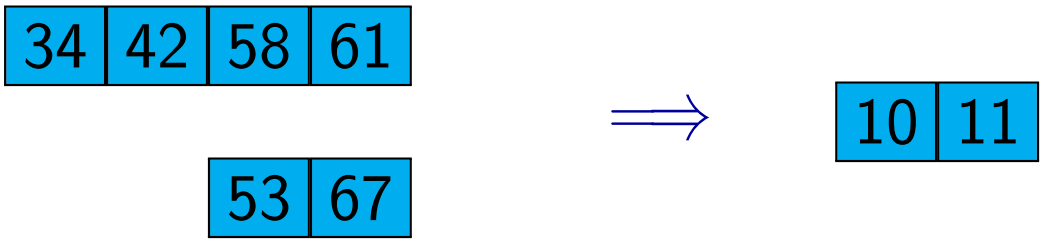
Hlavní myšlenka algoritmu: Dvě setříděné posloupnosti snadno spojíme do jediné setříděné posloupnosti.
Pokud mají obě posloupnosti dohromady n prvků, vyžaduje tato operace n kroků.



Složitost algoritmů

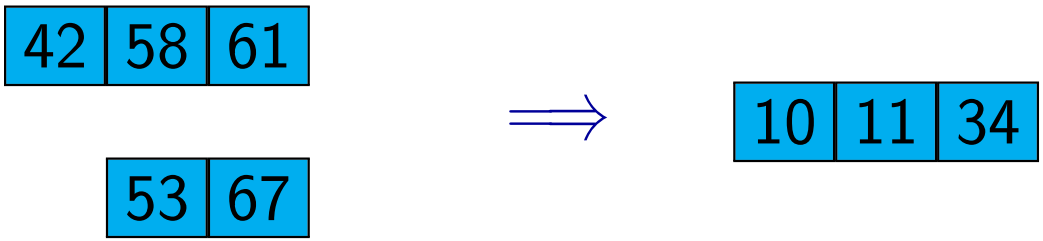
Příklad: Algoritmus MERGE-SORT.

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.

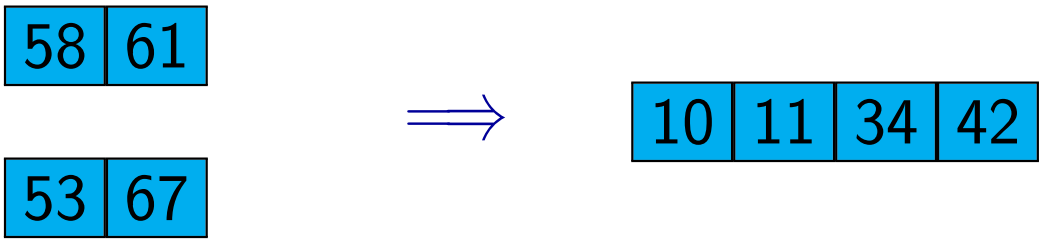
Hlavní myšlenka algoritmu: Dvě setříděné posloupnosti snadno spojíme do jediné setříděné posloupnosti.
Pokud mají obě posloupnosti dohromady n prvků, vyžaduje tato operace n kroků.



Složitost algoritmů

Příklad: Algoritmus MERGE-SORT.

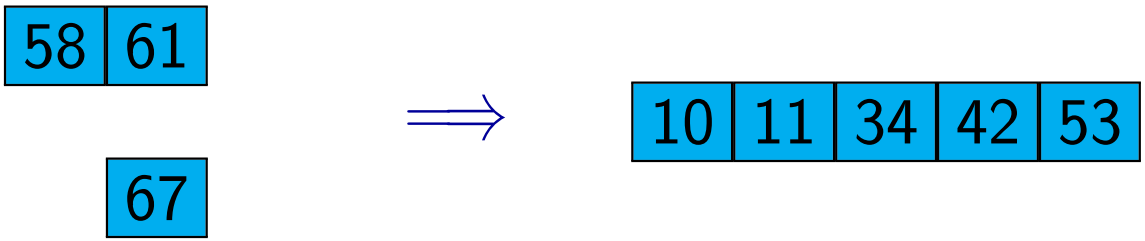
Hlavní myšlenka algoritmu: Dvě setříděné posloupnosti snadno spojíme do jediné setříděné posloupnosti.
Pokud mají obě posloupnosti dohromady n prvků, vyžaduje tato operace n kroků.



Složitost algoritmů

Příklad: Algoritmus MERGE-SORT.

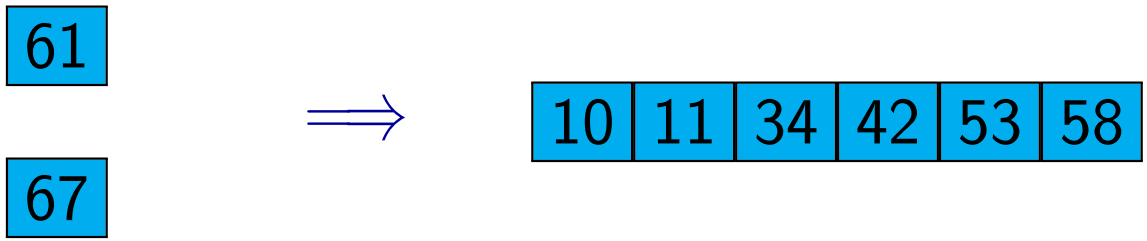
Hlavní myšlenka algoritmu: Dvě setříděné posloupnosti snadno spojíme do jediné setříděné posloupnosti.
Pokud mají obě posloupnosti dohromady n prvků, vyžaduje tato operace n kroků.



Složitost algoritmů

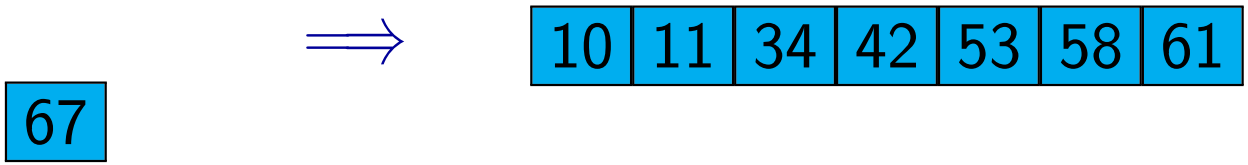
Příklad: Algoritmus MERGE-SORT.

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.



Složitost algoritmů

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.

Složitost algoritmů

Algoritmus 2: Merge sort

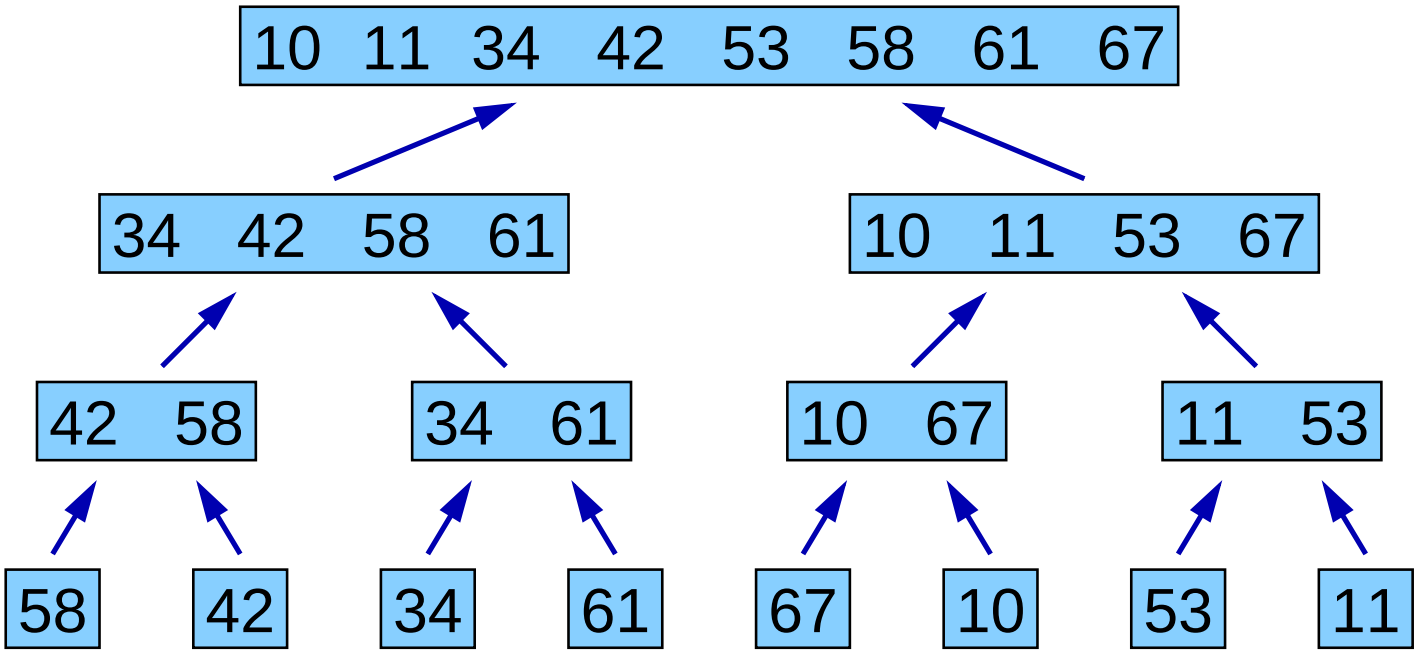
```
1 MERGE-SORT (A, p, r):
2 begin
3     if r − p > 1 then
4         q := ⌊(p + r) / 2⌋
5         MERGE-SORT(A, p, q)
6         MERGE-SORT(A, q, r)
7         MERGE(A, p, q, r)
8     end
9 end
```

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

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

Složitost algoritmů

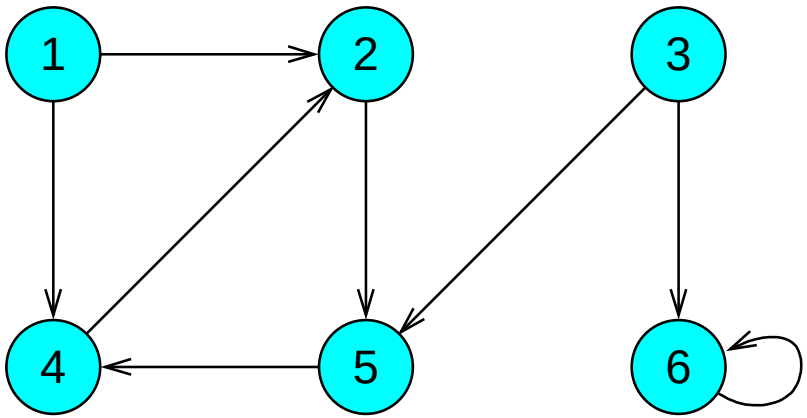
Vstup: 58, 42, 34, 61, 67, 10, 53, 11

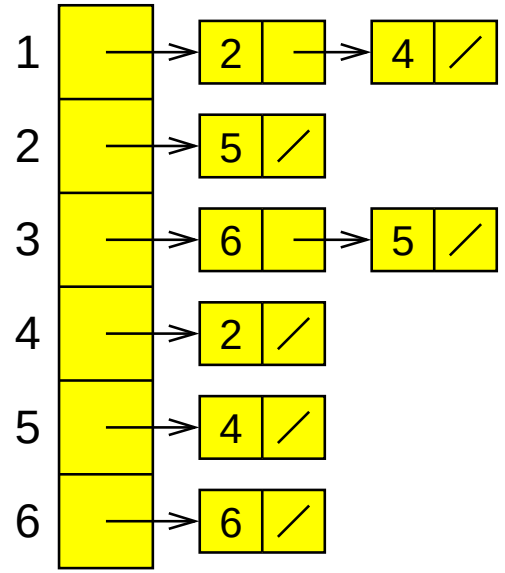


Strom rekurzivních volání má $\Theta(\log n)$ úrovní. Na každé úrovni se provede $\Theta(n)$ operací. Časová složitost algoritmu MERGE-SORT je $\Theta(n \log n)$.

Složitost algoritmů

Reprezentace grafu:

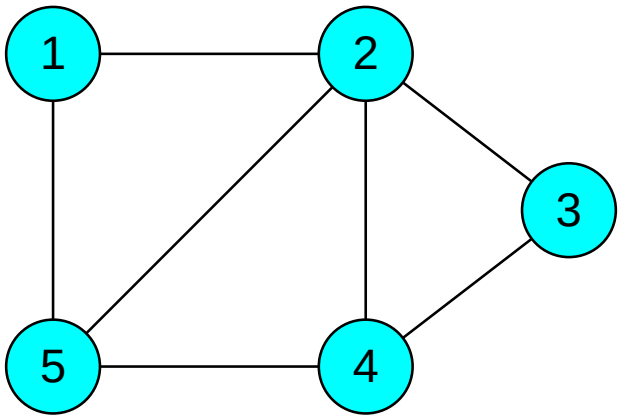


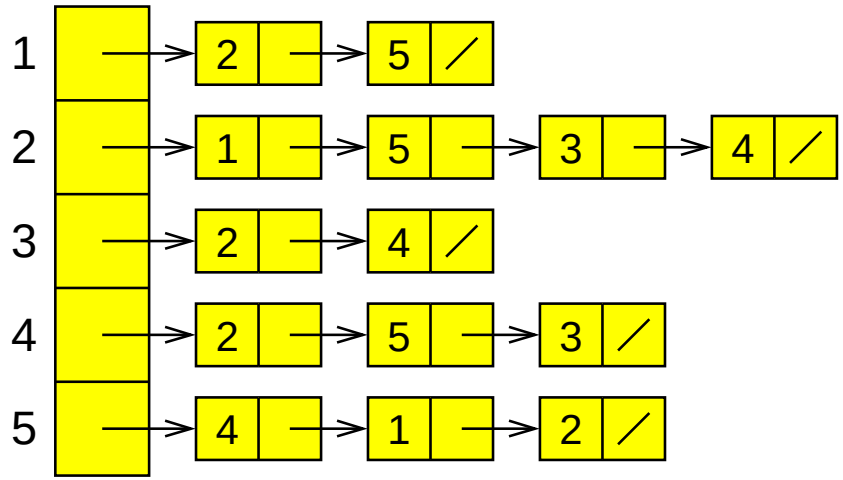


	1	2	3	4	5	6
1	0	1	0	1	0	0
2	0	0	0	0	1	0
3	0	0	0	0	1	1
4	0	1	0	0	0	0
5	0	0	0	1	0	0
6	0	0	0	0	0	1

Složitost algoritmů

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

Složitost algoritmů

Nalezení nejkratší cesty 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 $\Theta(n + m)$.

Složitost algoritmů

Algoritmus 3: Prohledávání do šířky

```
1  BFS (G, s):
2  begin
3      BFS-INIT(G, s)
4      ENQUEUE(Q, s)
5      while Q ≠ ∅ do
6          u := DEQUEUE(Q)
7          for each v ∈ edges[u] do
8              if color[v] = WHITE then
9                  color[v] := GRAY
10                 d[v] := d[u] + 1
11                 pred[v] := u
12                 ENQUEUE(Q, v)
13             end
14         end
15         color[u] := BLACK
16     end
17 end
```

Složitost algoritmů

Algoritmus 4: Prohledávání do šířky — inicializace — initialization

```
1  BFS-INIT (G, s):
2  begin
3      for each u ∈ V − {s} do
4          color[u] := WHITE
5          d[u] := ∞
6          pred[u] := NIL
7      end
8      color[s] := GRAY
9      d[u] := 0
10     pred[u] := NIL
11     Q := ∅
12 end
```

Složitost algoritmů

- Množina obsahující n prvků má 2^n podmnožin.

  Algoritmus řešící daný problém **hrubou silou**, kdy testuje nějakou podmínku pro všechny podmnožiny dané množiny.

- Pokud by stačilo omezit se na podmnožiny určité konkrétní velikosti k, těchto podmnožin je

$$\binom{n}{k}$$

  Pro některé hodnoty k celkový počet těchto podmnožin není o moc menší než 2^n:

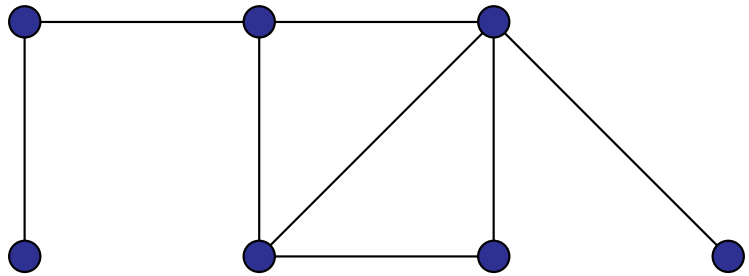
  Dá se například ukázat, že

$$\binom{n}{\lfloor n/2 \rfloor} \geq \frac{2^n}{n}.$$

Problém nezávislé množiny (IS — independent set)

Vstup: Neorientovaný graf G, číslo k.

Otázka: Existuje v grafu G nezávislá množina velikosti k?



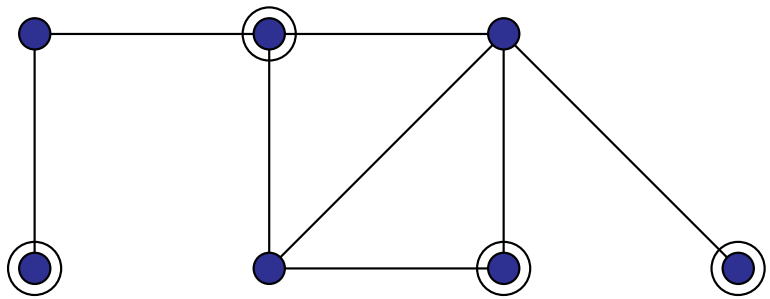
$k = 4$

Poznámka: **Nezávislá množina** v grafu je podmnožina vrcholů grafu taková, že žádné dva vrcholy z této podmnožiny nejsou spojeny hranou.

Problém nezávislé množiny (IS — independent set)

Vstup: Neorientovaný graf G, číslo k.

Otázka: Existuje v grafu G nezávislá množina velikosti k?

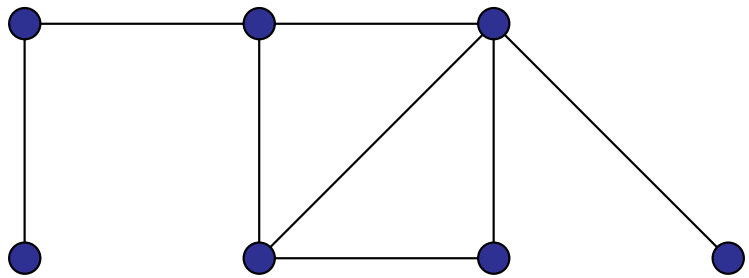


$k = 4$

Poznámka: **Nezávislá množina** v grafu je podmnožina vrcholů grafu taková, že žádné dva vrcholy z této podmnožiny nejsou spojeny hranou.

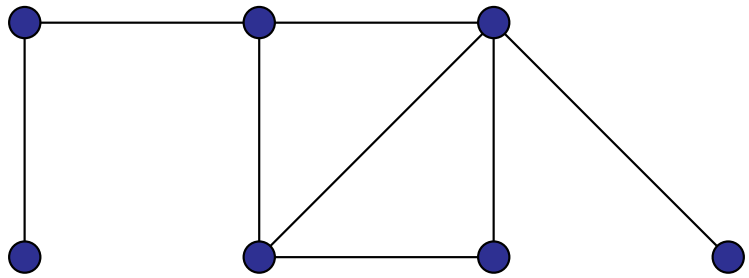
Složitost algoritmů

Příklad instance, kde je odpověď ANO:



$k = 4$

Příklad instance, kde je odpověď NE:



$k = 5$

Složitost algoritmů

Algoritmus řešící problém nezávislé množiny hrubou silou tím způsobem, že bude postupně testovat pro všechny k-prvkové podmnožiny n vrcholů, zda tvoří daná podmnožina nezávislou množinu, bude mít časovou složitost $2^{\Theta(n)}$.

Příklady algoritmických problémů

Na následujích slidech jsou uvedeny příklady několika typických algoritmických problémů, které:

- jsou algoritmicky řešitelné — většinou není těžké vymyslet algoritmus s exponenciální časovou složitostí, řešící daný problém
- není pro ně znám žádný algoritmus s polynomiální časovou složitostí
- na druhou stranu není ani dokázáno, že daný pro daný problém nemůže algoritmus s polynomiální časovou složitostí existovat
- vše jsou to příklady tzv. **NP-úplných** problémů

Problém SAT

SAT (splnitelnost booleovských formulí)

Vstup: Formule výrokové logiky φ.

Otázka: Je φ splnitelná?

Příklad:

Formule $\varphi_1 = p \wedge (\neg q \vee r)$ je splnitelná:
např. při ohodnocení v, kde $v(p) = 1$, $v(q) = 0$, $v(r) = 1$, platí $v \models \varphi_1$.

Formule $\varphi_2 = (p \wedge \neg p) \vee (\neg q \wedge r \wedge q)$ není splnitelná:
pro libovolné ohodnocení v platí $v \not\models \varphi_2$.

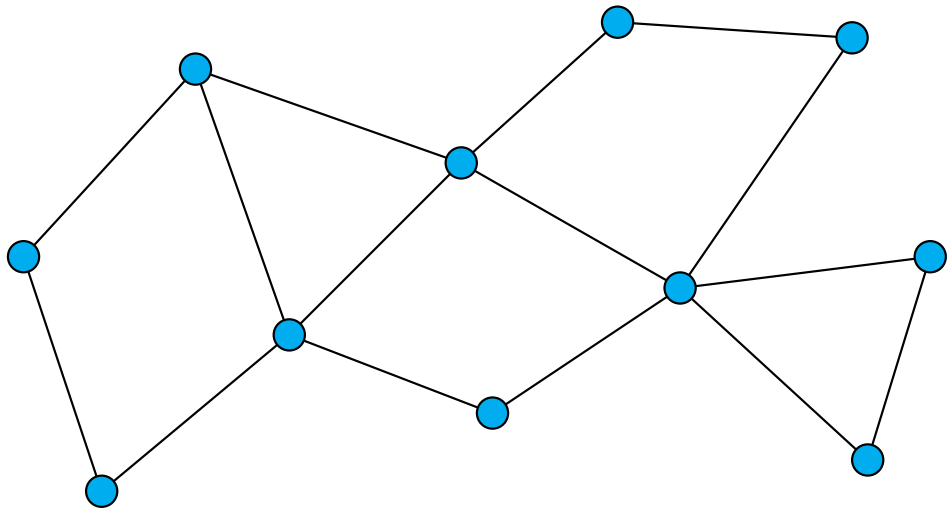
Barvení grafu

Barvení grafu

Vstup: Neorientovaný graf G, přirozené číslo k.

Otázka: Lze vrcholy grafu G obarvit k barvami tak, aby žádné dva vrcholy spojené hranou neměly stejnou barvu?

Příklad: $k = 3$



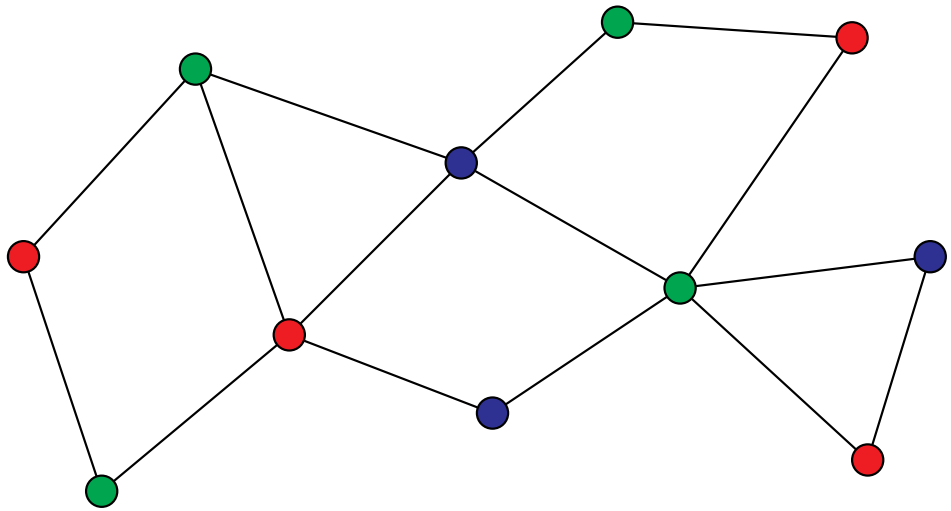
Barvení grafu

Barvení grafu

Vstup: Neorientovaný graf G, přirozené číslo k.

Otázka: Lze vrcholy grafu G obarvit k barvami tak, aby žádné dva vrcholy spojené hranou neměly stejnou barvu?

Příklad: $k = 3$



Odpověď: ANO

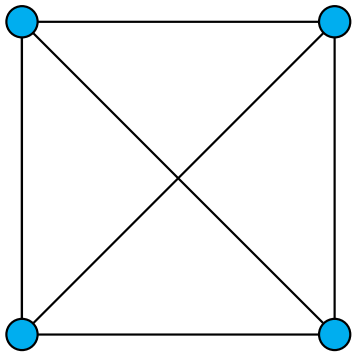
Barvení grafu

Barvení grafu

Vstup: Neorientovaný graf G, přirozené číslo k.

Otázka: Lze vrcholy grafu G obarvit k barvami tak, aby žádné dva vrcholy spojené hranou neměly stejnou barvu?

Příklad: $k = 3$



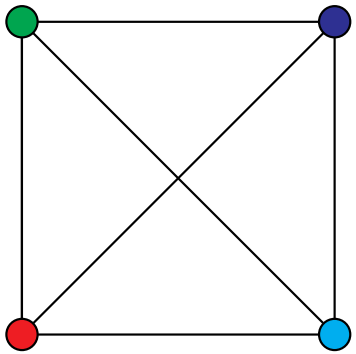
Barvení grafu

Barvení grafu

Vstup: Neorientovaný graf G, přirozené číslo k.

Otázka: Lze vrcholy grafu G obarvit k barvami tak, aby žádné dva vrcholy spojené hranou neměly stejnou barvu?

Příklad: $k = 3$



Odpověď: NE

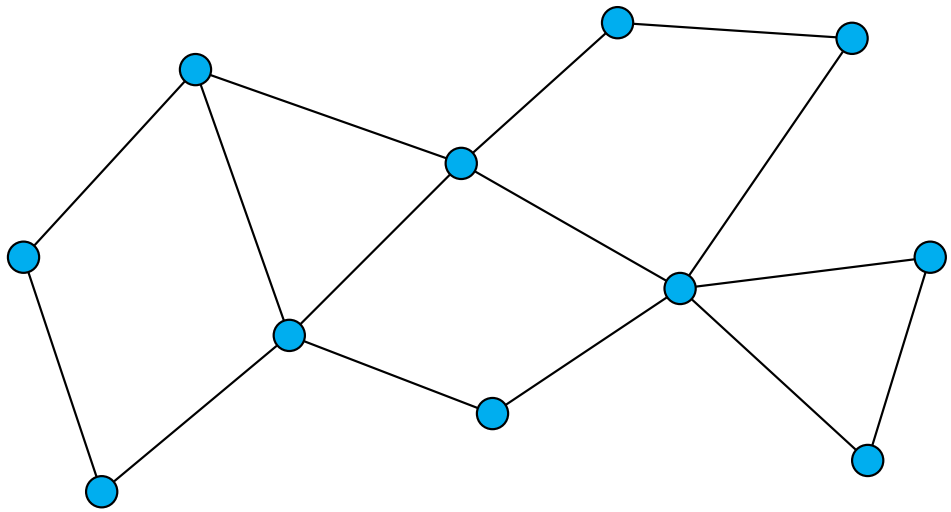
VC – Vrcholové pokrytí

VC – vrcholové pokrytí (vertex cover)

Vstup: Neorientovaný graf G a přirozené číslo k.

Otázka: Existuje v grafu G množina vrcholů velikosti k taková, že každá hrana má alespoň jeden svůj vrchol v této množině?

Příklad: $k = 6$



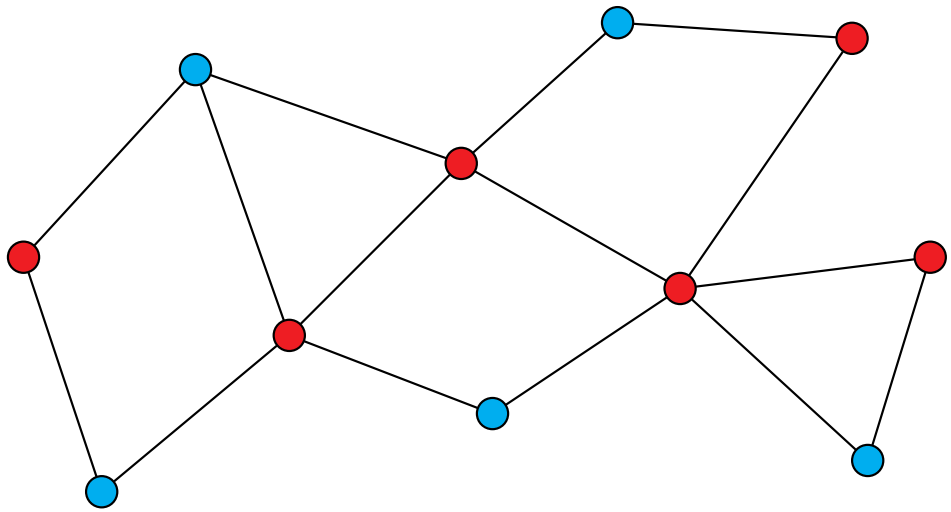
VC – Vrcholové pokrytí

VC – vrcholové pokrytí (vertex cover)

Vstup: Neorientovaný graf G a přirozené číslo k.

Otázka: Existuje v grafu G množina vrcholů velikosti k taková, že každá hrana má alespoň jeden svůj vrchol v této množině?

Příklad: $k = 6$



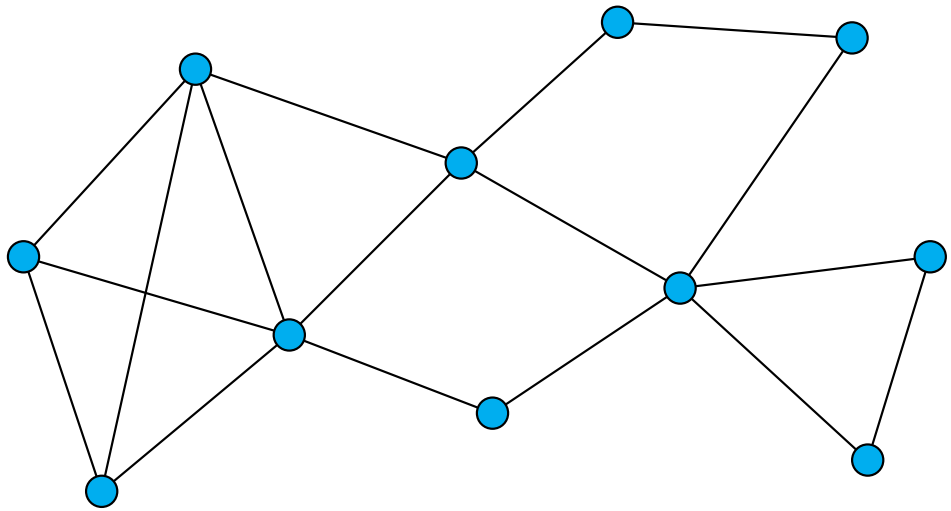
Odpověď: Ano

CLIQUE – problém kliky

Vstup: Neorientovaný graf G a přirozené číslo k.

Otázka: Existuje v grafu G množina vrcholů velikosti k taková, že každé dva vrcholy této množiny jsou spojeny hranou?

Příklad: $k = 4$

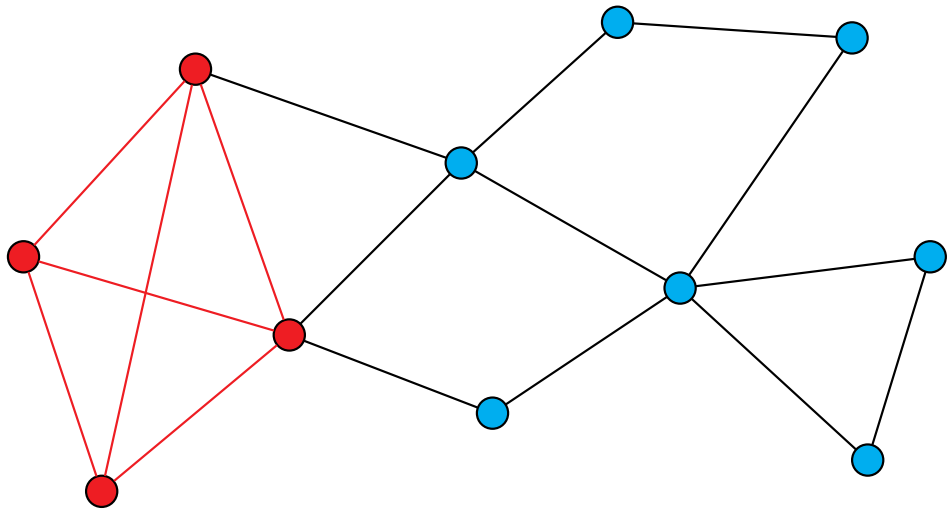


CLIQUE – problém kliky

Vstup: Neorientovaný graf G a přirozené číslo k.

Otázka: Existuje v grafu G množina vrcholů velikosti k taková, že každé dva vrcholy této množiny jsou spojeny hranou?

Příklad: $k = 4$



Odpověď: ANO

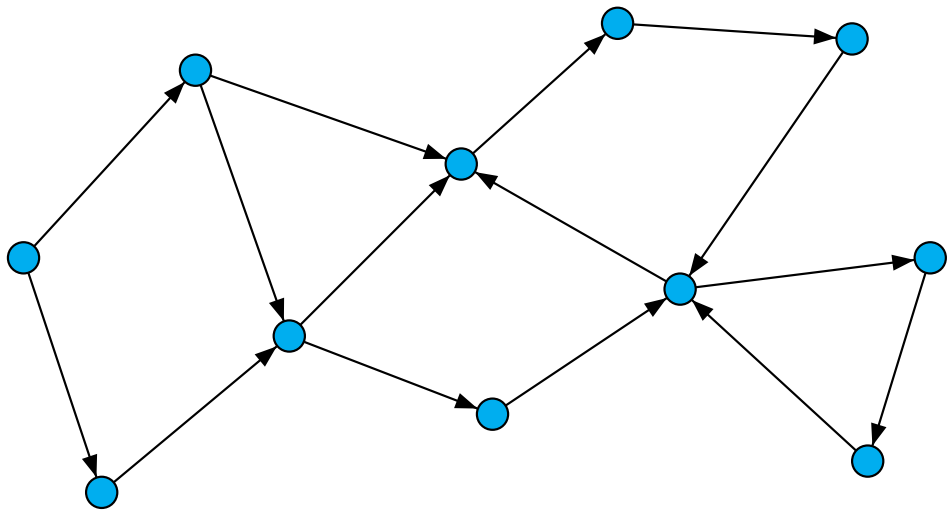
Hamiltonovský cyklus

HC – Problém „Hamiltonovský cyklus“

Vstup: Orientovaný graf G.

Otázka: Existuje v grafu G Hamiltonovský cyklus (orientovaný cyklus procházející každým vrcholem právě jednou)?

Příklad:



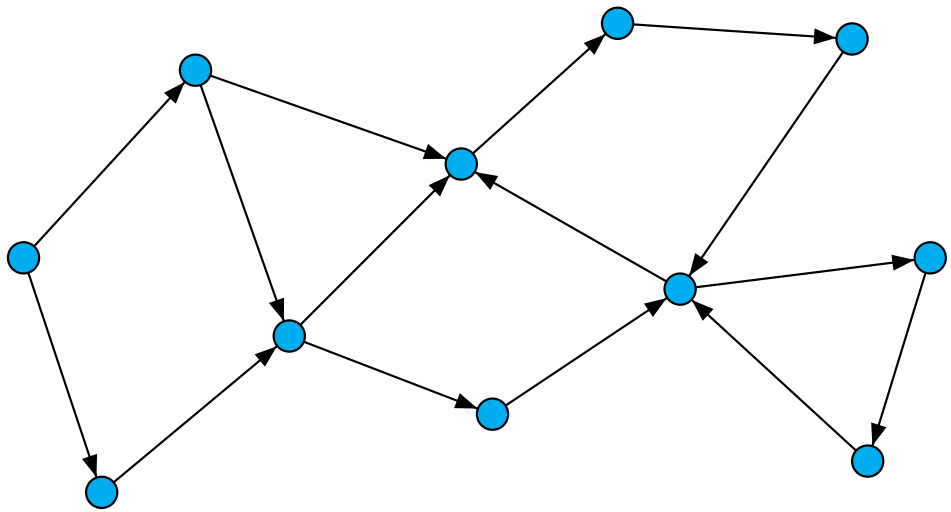
Hamiltonovský cyklus

HC – Problém „Hamiltonovský cyklus“

Vstup: Orientovaný graf G.

Otázka: Existuje v grafu G Hamiltonovský cyklus (orientovaný cyklus procházející každým vrcholem právě jednou)?

Příklad:



Odpověď: NE

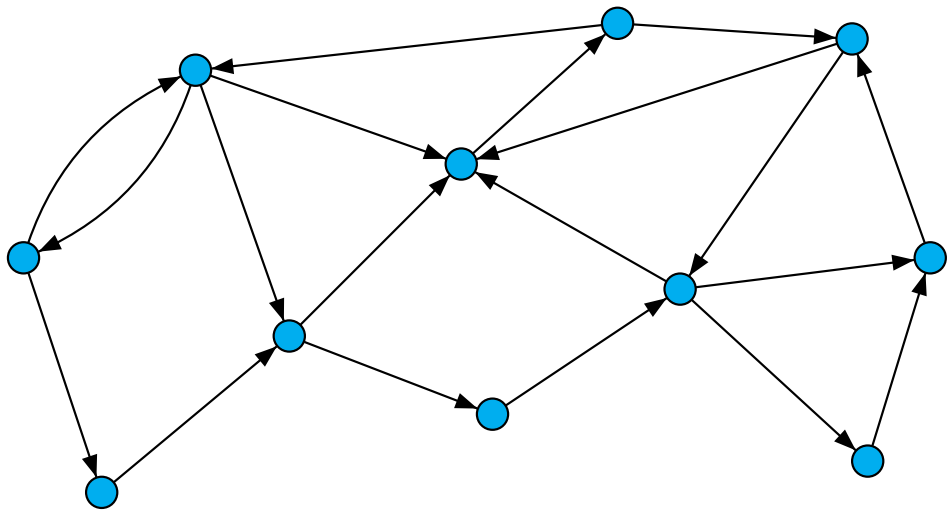
Hamiltonovský cyklus

HC – Problém „Hamiltonovský cyklus“

Vstup: Orientovaný graf G.

Otázka: Existuje v grafu G Hamiltonovský cyklus (orientovaný cyklus procházející každým vrcholem právě jednou)?

Příklad:



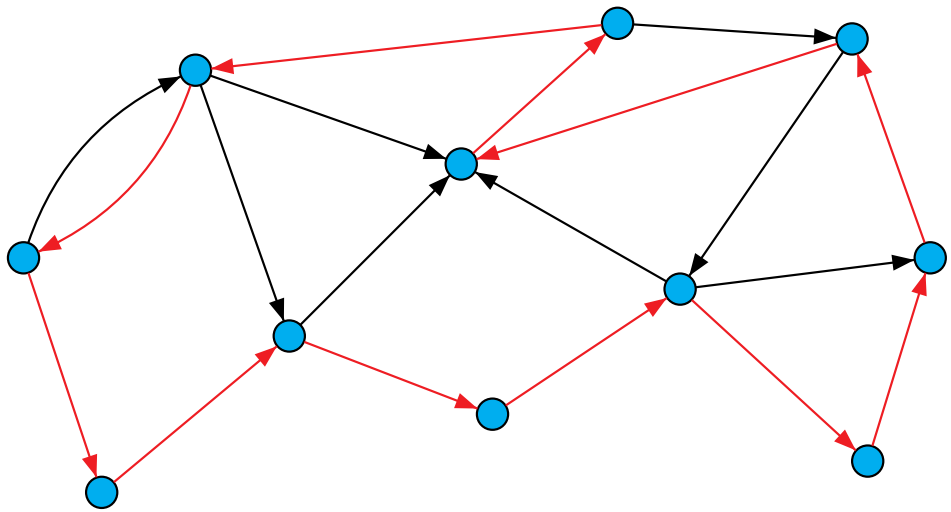
Hamiltonovský cyklus

HC – Problém „Hamiltonovský cyklus“

Vstup: Orientovaný graf G.

Otázka: Existuje v grafu G Hamiltonovský cyklus (orientovaný cyklus procházející každým vrcholem právě jednou)?

Příklad:



Odpověď: Ano

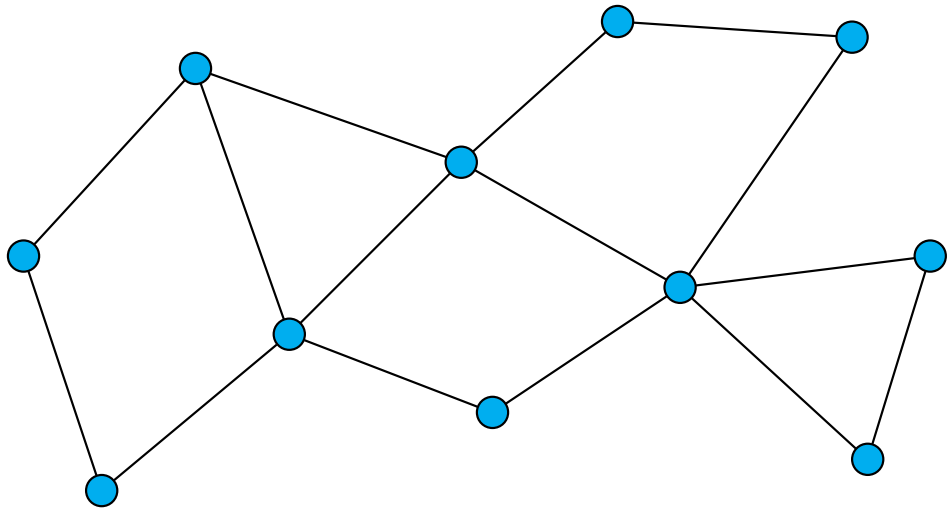
Hamiltonovská kružnice

HK – Problém „Hamiltonovská kružnice“

Vstup: Neorientovaný graf G.

Otázka: Existuje v grafu G Hamiltonovská kružnice (neorientovaný cyklus procházející každým vrcholem právě jednou)?

Příklad:



Odpověď: NE

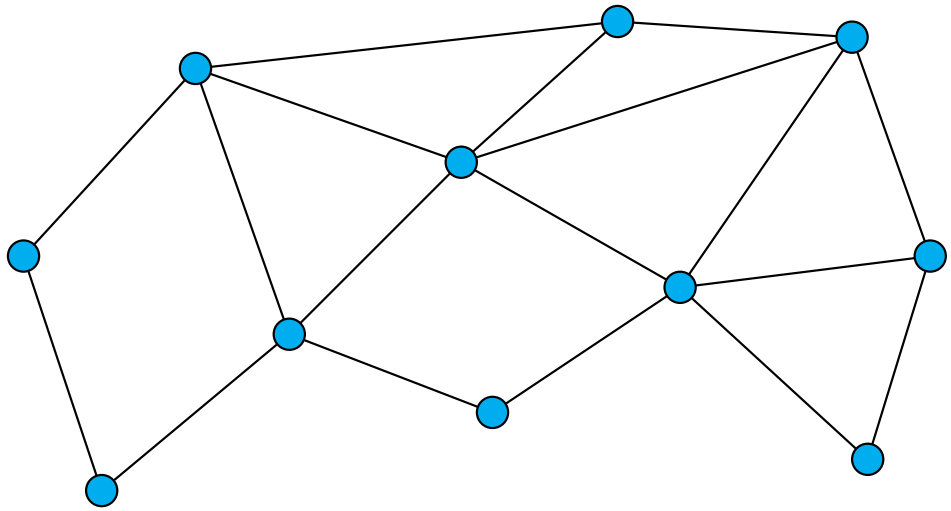
Hamiltonovská kružnice

HK – Problém „Hamiltonovská kružnice“

Vstup: Neorientovaný graf G.

Otázka: Existuje v grafu G Hamiltonovská kružnice (neorientovaný cyklus procházející každým vrcholem právě jednou)?

Příklad:



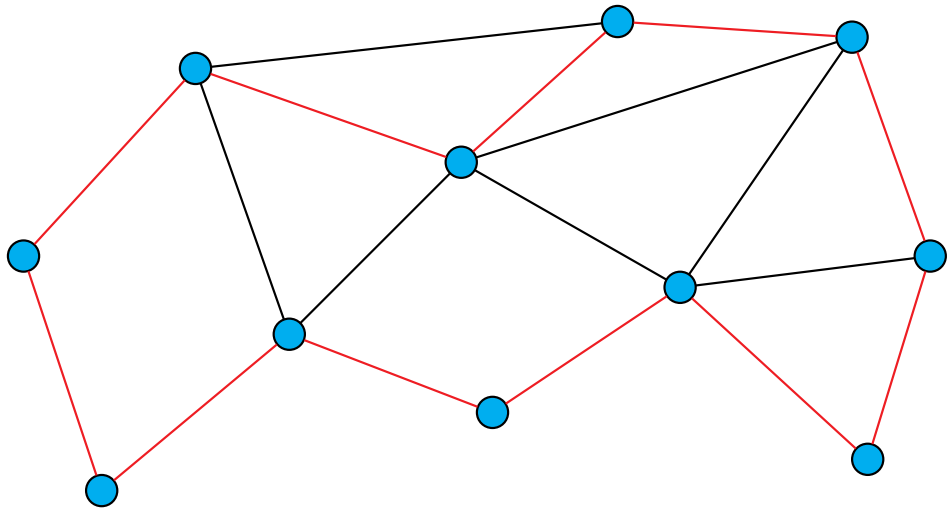
Hamiltonovská kružnice

HK – Problém „Hamiltonovská kružnice“

Vstup: Neorientovaný graf G.

Otázka: Existuje v grafu G Hamiltonovská kružnice (neorientovaný cyklus procházející každým vrcholem právě jednou)?

Příklad:



Odpověď: Ano

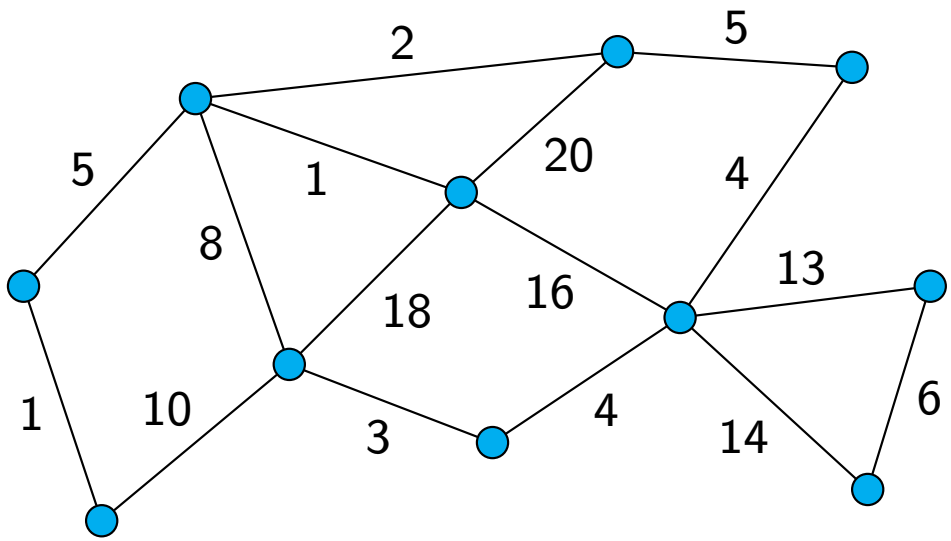
Problém obchodního cestujícího

TSP - Problém „obchodního cestujícího“

Vstup: Neorientovaný graf G s hranami ohodnocenými přirozenými čísly a číslo k.

Otázka: Existuje v grafu G uzavřená cesta procházející všemi vrcholy takový, že součet délek hran na této cestě (včetně opakovaných) je maximálně k?

Příklad: $k = 70$



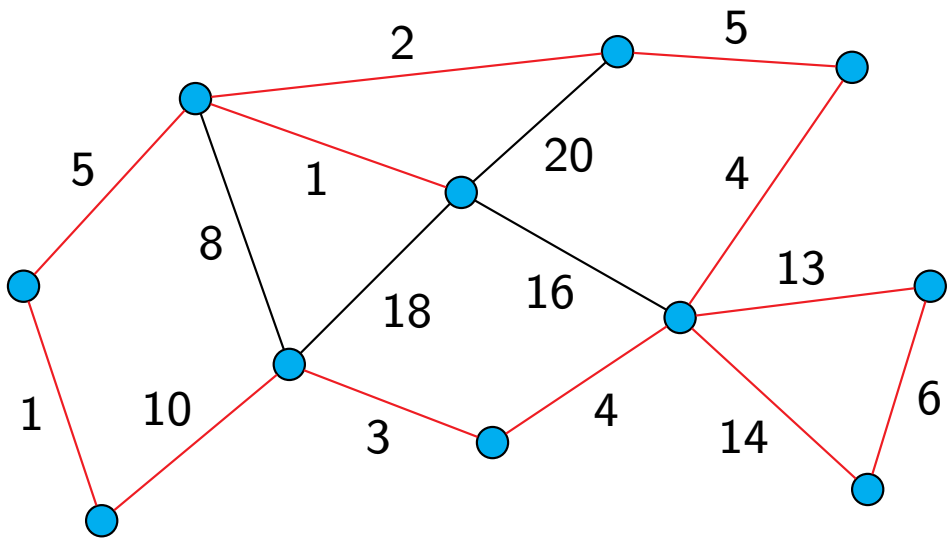
Problém obchodního cestujícího

TSP - Problém „obchodního cestujícího“

Vstup: Neorientovaný graf G s hranami ohodnocenými přirozenými čísly a číslo k.

Otázka: Existuje v grafu G uzavřená cesta procházející všemi vrcholy takový, že součet délek hran na této cestě (včetně opakovaných) je maximálně k?

Příklad: $k = 70$



Odpověď: Ano, protože byl nalezen sled se součtem 69.

SUBSET-SUM

Problém SUBSET-SUM

Vstup: Sekvence přirozených čísel $a_1, a_2, \ldots, a_n$ a přirozené číslo s.

Otázka: Existuje množina $I \subseteq \{1, 2, \ldots, n\}$ taková, že $\sum_{i \in I} a_i = s$?

Jinak řečeno, ptáme se zda z dané (multi)množiny čísel je možné vybrat podmnožinu, jejíž součet je s.

Příklad: Pro vstup tvořený čísly $3, 5, 2, 3, 7$ a číslem $s = 15$ je odpověď Ano, neboť $3 + 5 + 7 = 15$.

Pro vstup tvořený čísly $3, 5, 2, 3, 7$ a číslem $s = 16$ je odpověď Ne, neboť žádná podmnožina těchto čísel nedává součet 16.

SUBSET-SUM

Poznámka:
Pořadí čísel $a_1, a_2, \ldots, a_n$ na vstupu není důležité.

Všimněte si však určitého rozdílu oproti tomu, kdybychom problém formulovali tak, že vstupem je množina $\{a_1, a_2, \ldots, a_n\}$ a číslo s — v množině se čísla neopakují, zatímco v sekvenci se může totéž číslo vyskytnout vícekrát.

SUBSET-SUM

Problém SUBSET-SUM je speciálním případem **problému batohu** (knapsack problem):

Knapsack problem

Vstup: Sekvence dvojic přirozených čísel $(a_1, b_1), (a_2, b_2), \ldots, (a_n, b_n)$ a dvě přirozená čísla s a t.

Otázka: Existuje množina $I \subseteq \{1, 2, \ldots, n\}$ taková, že $\sum_{i \in I} a_i \leq s$ a $\sum_{i \in I} b_i \geq t$?

SUBSET-SUM

Neformálně můžeme problém batohu formulovat takto:

Máme n předmětů, kde i-tý předmět váží a_i gramů a má cenu b_i Kč. Do batohu se vejdou předměty o maximální celkové váze s gramů.

Otázka zní, zda můžeme z předmětů vybrat podmnožinu, která by vážila maximálně s gramů a měla celkovou cenu alespoň t Kč.

Poznámka:
Zde jsme problém batohu formulovali jako rozhodovací problém.

Běžnější je formulovat tento problém jako optimalizační problém, kde je cílem najít takovou množinu $I \subseteq \{1, 2, \ldots, n\}$, kde hodnota $\sum_{i \in I} b_i$ je maximální, přičemž ovšem musí být dodržena podmínka $\sum_{i \in I} a_i \leq s$, tj. vybrat předměty s maximální celkovou cenou tak, aby nebyla překročena kapacita batohu.

SUBSET-SUM

To, že SUBSET-SUM je speciálním případem problému batohu, vidíme z následující jednoduché konstrukce:

Řekněme, že $a_1, a_2, \ldots, a_n$, s_1 je instance problému SUBSET-SUM. Je očividné, že pro instanci problému batohu, kde máme sekvenci $(a_1, a_1), (a_2, a_2), \ldots, (a_n, a_n)$, $s = s_1$ a $t = s_1$, je odpověď stejná jako pro původní instanci SUBSET-SUM.

SUBSET-SUM

Pokud chceme studovat složitost problémů jako jsou SUBSET-SUM nebo problém batohu, je dobré si nejprve ujasnit, co považujeme za velikost vstupu.

Asi nejpřirozenější je definovat velikost vstupu jako celkový počet bitů, který potřebujeme k zápisu instance.

Musíme však určit, jakým způsobem jsou na vstupu zadána přirozená čísla – zda binárně (případně v jiné číselné soustavě o základu alespoň 2, např. desítkové nebo šestnáctkové) nebo unárně.

- Pokud počítáme velikost vstupu jako celkový počet bitů při použití **binárního** zápisu čísel, tak pro problém SUBSET-SUM není znám polynomiální algoritmus.
- Pokud počítáme velikost vstupu jako celkový počet bitů při použití **unárního** zápisu, tak existuje pro problém SUBSET-SUM algoritmus s polynomiální časovou složitostí.

ILP – celočíselné lineární programování

Problém ILP (celočíselné lineární programování)

Vstup: Celočíselná matice A a celočíselný vektor b.

Otázka: Existuje celočíselný vektor x, takový že $Ax \leq b$?

Příklad instance problému:

$$A = \begin{pmatrix} 3 & -2 & 5 \\ 1 & 0 & 1 \\ 2 & 1 & 0 \end{pmatrix} \qquad b = \begin{pmatrix} 8 \\ -3 \\ 5 \end{pmatrix}$$

Ptáme se tedy, zda existuje celočíselné řešení následující soustavy nerovnic:

$$\begin{aligned} 3x_1 - 2x_2 + 5x_3 &\leq 8 \\ x_1 + x_3 &\leq -3 \\ 2x_1 + x_2 &\leq 5 \end{aligned}$$

ILP – celočíselné lineární programování

Jedním z řešení soustavy

$$\begin{aligned} 3x_1 - 2x_2 + 5x_3 &\leq 8 \\ x_1 + x_3 &\leq -3 \\ 2x_1 + x_2 &\leq 5 \end{aligned}$$

je například $x_1 = -4$, $x_2 = 1$, $x_3 = 1$, tj.

$$x = \begin{pmatrix} -4 \\ 1 \\ 1 \end{pmatrix}$$

neboť

$$\begin{aligned} 3 \cdot (-4) - 2 \cdot 1 + 5 \cdot 1 &= -9 \leq 8 \\ -4 + 1 &= -3 \leq -3 \\ 2 \cdot (-4) + 1 &= -7 \leq 5 \end{aligned}$$

Pro tuto instanci je tedy odpověď Ano.

ILP – celočíselné lineární programování

Poznámka: Analogický problém, kdy se pro danou soustavu lineálních nerovnic ptáme, zda existuje její řešení v oboru **reálných** čísel, je možné řešit v polynomiálním čase.

Příklady algoritmických problémů

Na následujích slidech jsou uvedeny příklady několika typických algoritmických problémů, které:

- **nejsou algoritmicky řešitelné**
- pro tyto problémy se dá dokázat, že pro pro ně nemohou existovat algoritmy, které by je řešily

Poznámka: Připomeňme, že algoritmus řeší daný problém, jestliže se pro každý vstup po konečném počtu kroků zastaví a vydá správný výstup.

Pro následující problémy tedy platí, že pro každý algoritmus, který by se je pokoušel řešit, se dá najít příklad vstupu, pro který:

- se algoritmus nikdy nezastaví, nebo
- vydá chybný výstup

Poznámka: V případě, že se jedná o rozhodovací problémy, se problémům, které nejsou algoritmicky řešitelné, říká **nerozhodnutelné problémy**.

Příklady nerozhodnutelných problémů

S jedním příkladem nerozhodnutelného problému už jsme se setkali:

Problém

Vstup: Bezkontextové gramatiky G_1 a G_2.

Otázka: Je $L(G_1) = L(G_2)$?

případně

Problém

Vstup: Bezkontextová gramatika G generující jazyk nad abecedou Σ.

Otázka: Je $L(G) = \Sigma^*$?

Další nerozhodnutelné problémy

Problém

Vstup: Bezkontextové gramatiky G_1 a G_2.

Otázka: Je $L(G_1) \cap L(G_2) = \emptyset$?

Problém

Vstup: Bezkontextová gramatika G.

Otázka: Je G nejednoznačná?

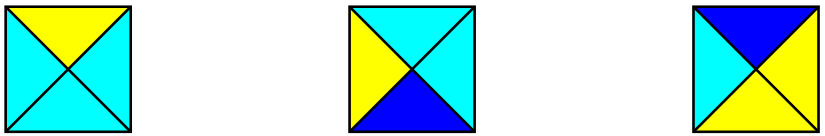
Další nerozhodnutelné problémy

Nerozhodnutelná je celá řada problémů, které se týkají ověřování chování programů:

- Vydá daný program pro nějaký vstup odpověď ANO?
- Zastaví se daný program pro libovolný vstup?
- Dávají dva dané programy pro stejné vstupy stejný výstup?
- ...

Další nerozhodnutelné problémy

Vstupem je množina typů kachliček, jako třeba:

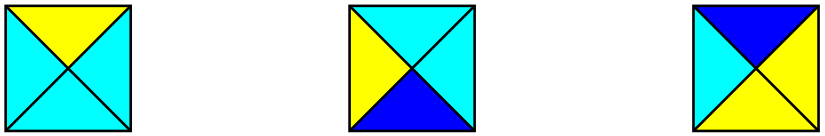


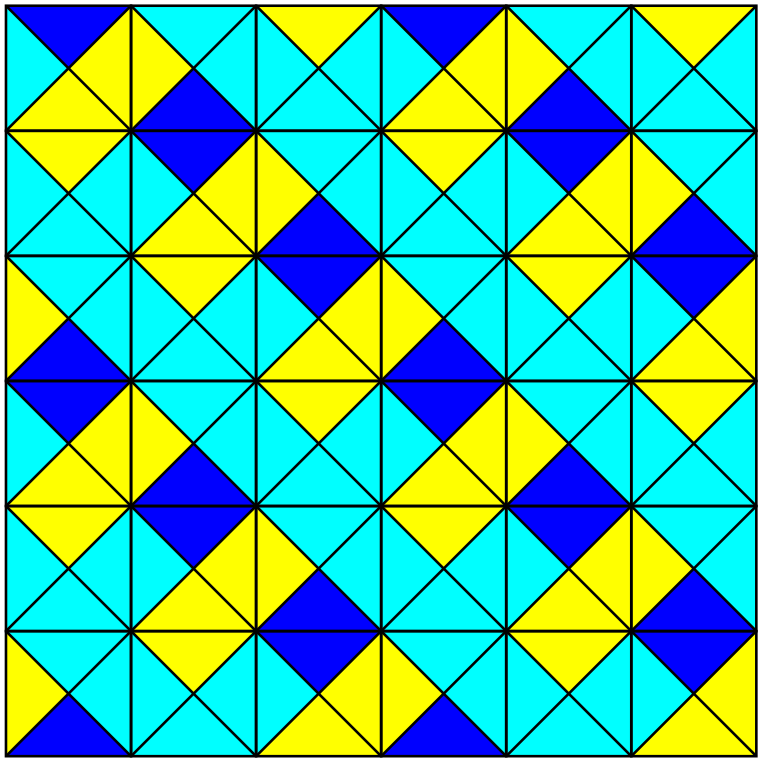
Otázka je, zda je možné použitím daných typů kachliček pokrýt každou libovolně velkou konečnou plochu tak, aby všechny kachličky spolu sousedily stejnými barvami.

Poznámka: Můžeme předpokládat, že máme v zásobě neomezené množství kachliček všech typů.

Kachličky není dovoleno otáčet.

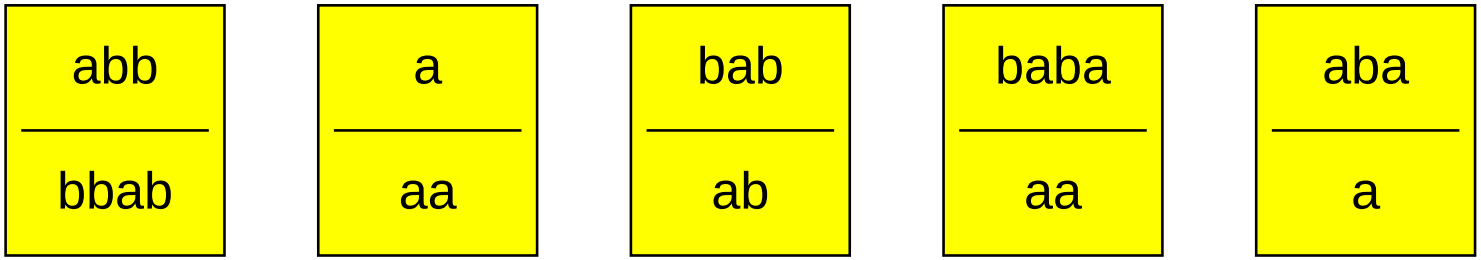
Další nerozhodnutelné problémy



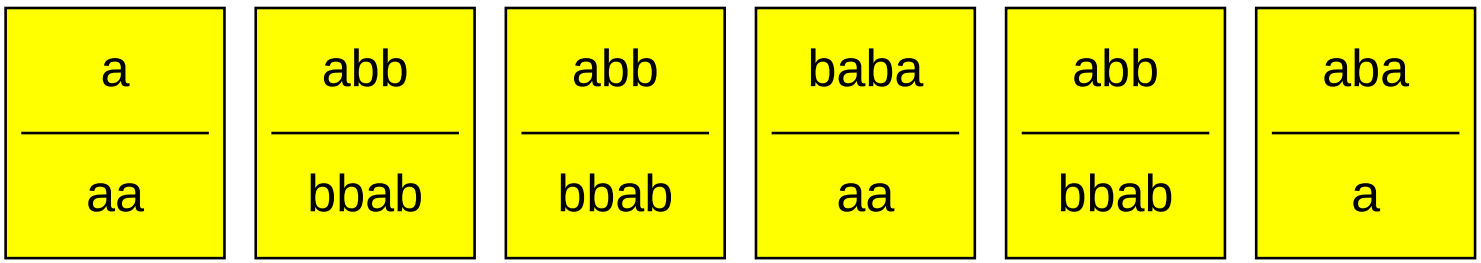


Další nerozhodnutelné problémy

Vstupem je množina typů kartiček, jako třeba:



Otázka je, zda je možné z těchto typů kartiček vytvořit neprázdnou konečnou posloupnost, kde zřetězením slov nahoře i dole vznikne totéž slovo. Každý typ kartičky je možné používat opakovaně.



Nahoře i dole vznikne slovo aabbabbbabaabbaba.

Další nerozhodnutelné problémy

Problém

Vstup: Uzavřená formule predikátové logiky, ve které mohou být použity jako funkční symboly $+$ a $*$ a celočíselné konstanty a jako predikátové symboly $=$ a $<$.

Otázka: Je daná formule pravdivá v oboru přirozených čísel (při přirozené interpretaci všech funkčních a predikátových symbolů)?

Příklad vstupu:

$$\forall x \exists y \forall z ((x * y = z) \wedge (y + 5 = x))$$

Poznámka: Úzce souvisí s Gödelovou větou o neúplnosti.

Další nerozhodnutelné problémy

Je zajímavé, že analogický problém, kde ale místo přirozených čísel uvažujeme čísla **reálná**, je algoritmicky rozhodnutelný (i když popis daného algoritmu a důkaz jeho korektnosti jsou značně netriviální).

Rovněž, pokud uvažujeme přirozená nebo celá čísla a stejné formule jako v předchozím případě, ale s tím, že v nich nesmí být použit funkční symbol $*$ (násobení), tak je problém algoritmicky rozhodnutelný.

Další nerozhodnutelné problémy

Pokud můžeme používat $*$, je ve skutečnosti nerozhodnutelný už velmi omezený případ:

Desátý Hilbertův problém

Vstup: Polynom $f(x_1, x_2, \ldots, x_n)$ vytvořený z proměnných $x_1, x_2, \ldots, x_n$ a celočíselných konstant.

Otázka: Existují přirozená čísla $x_1, x_2, \ldots, x_n$ taková, že $f(x_1, x_2, \ldots, x_n) = 0$?

Příklad vstupu: $5x^2y - 8yz + 3z^2 - 15$

Tj. ptáme se, zda

$$\exists x \exists y \exists z (5 * x * x * y + (-8) * y * z + 3 * z * z + (-15) = 0)$$

platí v oboru přirozených čísel.

Další nerozhodnutelné problémy

Také následující problém je algoritmicky nerozhodnutelný:

Problém

Vstup: Uzavřená formule φ predikátové logiky.

Otázka: Platí $\models \varphi$?

Poznámka: Zápis $\models \varphi$ znamená, že formule φ je logicky platná, tj. pravdivá v každé interpretaci.