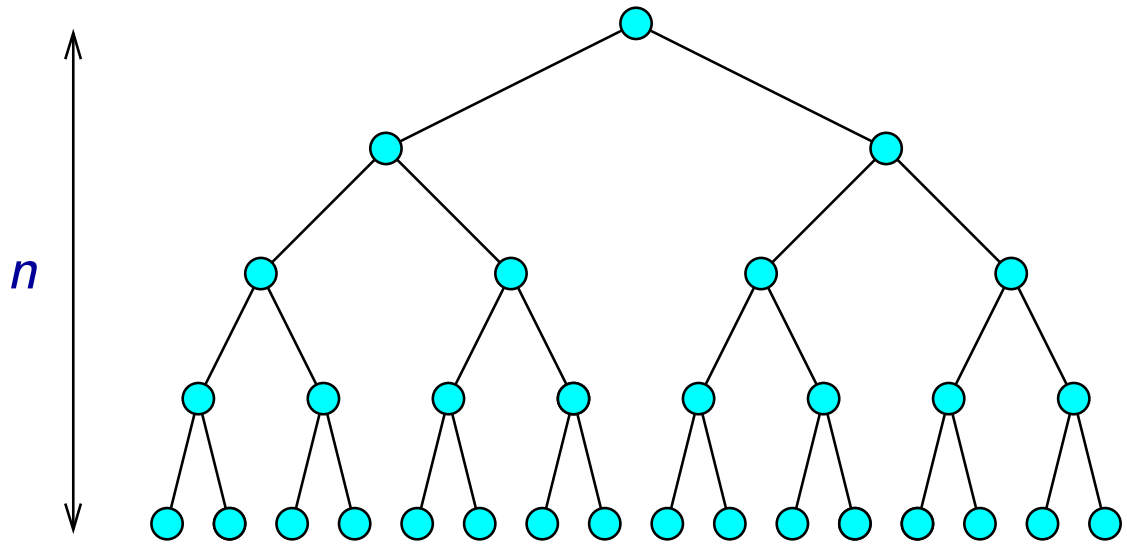
Rekurentní vztahy

Někdy je vhodné vyjadřovat hodnotu funkce pomocí **rekurentních vztahů**.

Rekurentní vztah je rovnice nebo nerovnice, kde je hodnota funkce pro větší argumenty vyjádřena pomocí hodnot, kterých funkce nabývá pro menší argumenty.

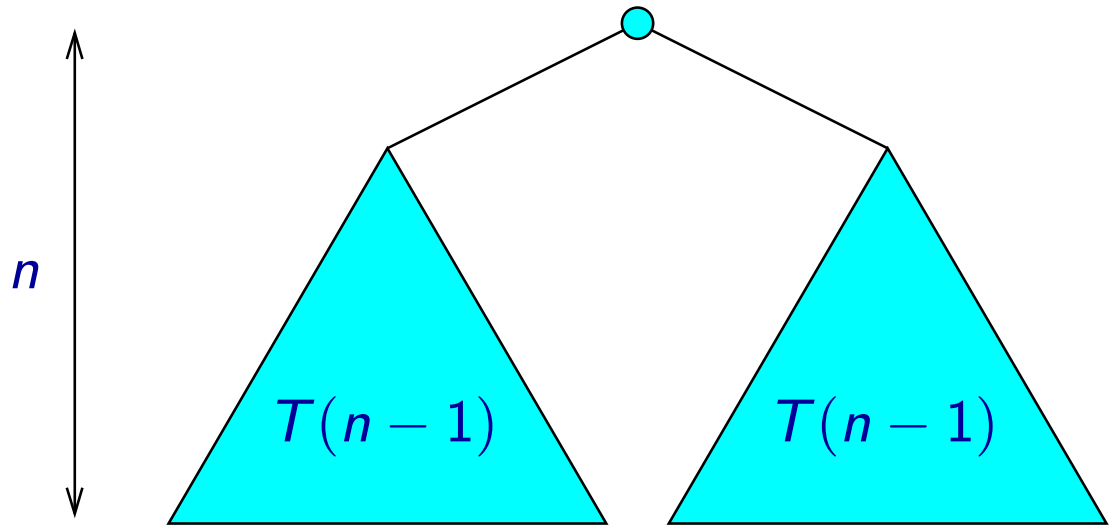
Příklad: Kolik vrcholů má úplný binární strom výšky n?



Rekurentní vztahy

$T(n)$ – počet vrcholů úplného binárního stromu výšky n

$$T(n) = \begin{cases} 1 & \text{pro } n = 0 \\ 2 \cdot T(n-1) + 1 & \text{pro } n > 0 \end{cases}$$



Rekurentní vztahy

$$T(n) = \begin{cases} 1 & \text{pro } n = 0 \\ 2 \cdot T(n-1) + 1 & \text{pro } n > 0 \end{cases}$$

$$T(0) = 1$$
$$T(1) = 2 \cdot T(0) + 1 = 2 \cdot 1 + 1 = 3$$
$$T(2) = 2 \cdot T(1) + 1 = 2 \cdot 3 + 1 = 7$$
$$T(3) = 2 \cdot T(2) + 1 = 2 \cdot 7 + 1 = 15$$
$$T(4) = 2 \cdot T(3) + 1 = 2 \cdot 15 + 1 = 31$$
$$T(5) = 2 \cdot T(4) + 1 = 2 \cdot 31 + 1 = 63$$
$$T(6) = 2 \cdot T(5) + 1 = 2 \cdot 63 + 1 = 127$$
$$T(7) = 2 \cdot T(6) + 1 = 2 \cdot 127 + 1 = 255$$
$$T(8) = 2 \cdot T(7) + 1 = 2 \cdot 255 + 1 = 511$$
$$T(9) = 2 \cdot T(8) + 1 = 2 \cdot 511 + 1 = 1023$$
$$T(10) = 2 \cdot T(9) + 1 = 2 \cdot 1023 + 1 = 2047$$

$\ldots$

Rekurentní vztahy

Pokud chceme vyjádřit hodnotu $T(n)$ nerekurentně, jednou z možností je použít tzv. **substituční metodu**:

1. Uhodnout správné řešení.
2. Ověřit jeho správnost pomocí matematické indukce.

Rekurentní vztahy

Příklad:

$$T(n) = \begin{cases} 1 & \text{pro } n = 0 \\ 2 \cdot T(n-1) + 1 & \text{pro } n > 0 \end{cases}$$

Uhodneme, že $T(n) = 2^{n+1} - 1$.

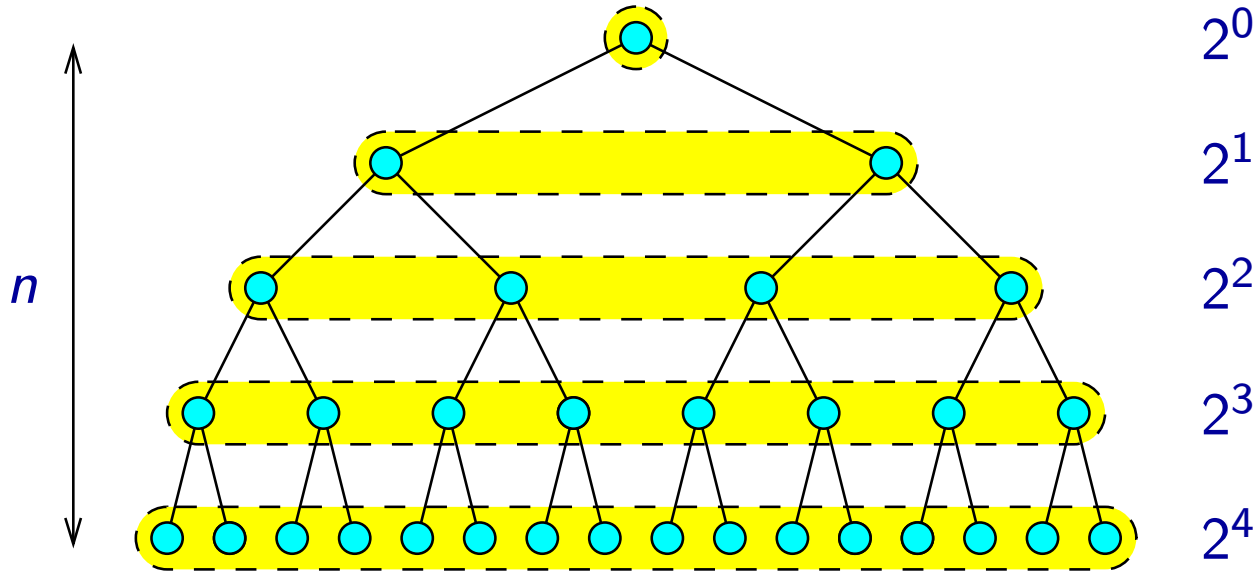
Indukcí ověříme, že tomu tak skutečně je:

- Báze ($i = 0$): $T(0) = 2^{0+1} - 1 = 1$
- Indukční krok ($i > 0$):

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

Rekurentní vztahy

Poznámka: Alternativní možností je spočítat počty vrcholů v jednotlivých „vrstvách“ stromu.



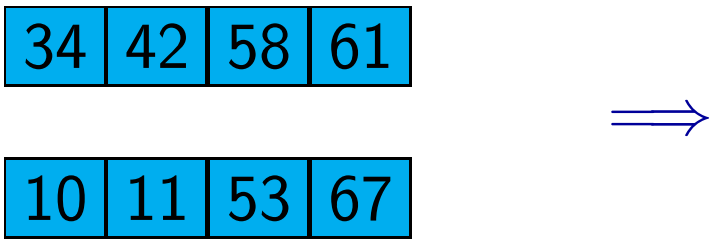
$$T(n) = \sum_{i=0}^{n} 2^i = \frac{2^{n+1} - 1}{2 - 1} = 2^{n+1} - 1$$

Rekurentní vztahy

Rekurentní vztahy se často používají při zkoumání časové složitosti rekurzivních 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.

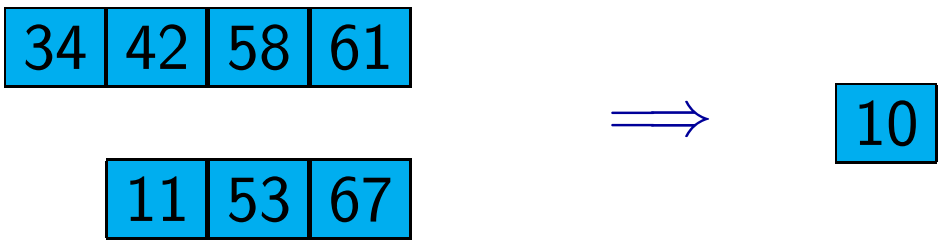


Rekurentní vztahy

Rekurentní vztahy se často používají při zkoumání časové složitosti rekurzivních 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.

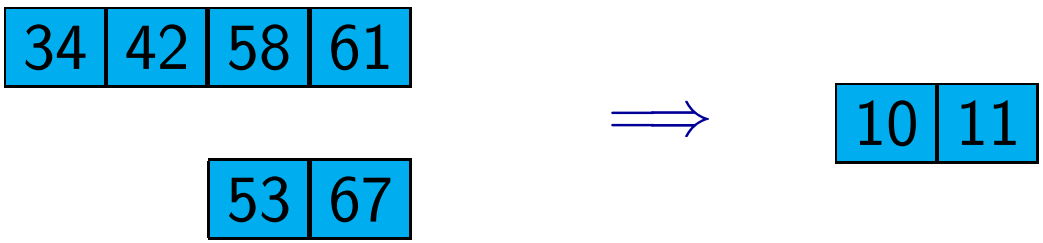


Rekurentní vztahy

Rekurentní vztahy se často používají při zkoumání časové složitosti rekurzivních 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.

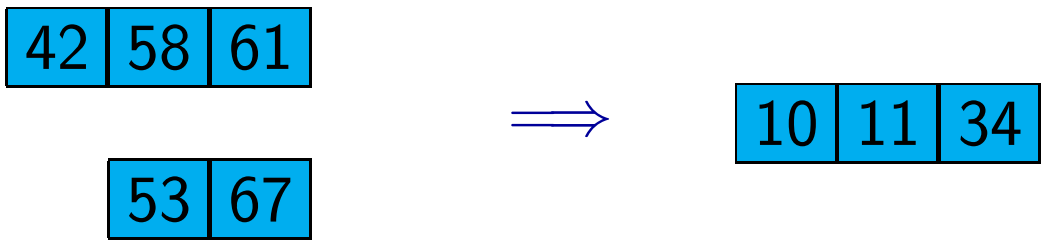


Rekurentní vztahy

Rekurentní vztahy se často používají při zkoumání časové složitosti rekurzivních 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.

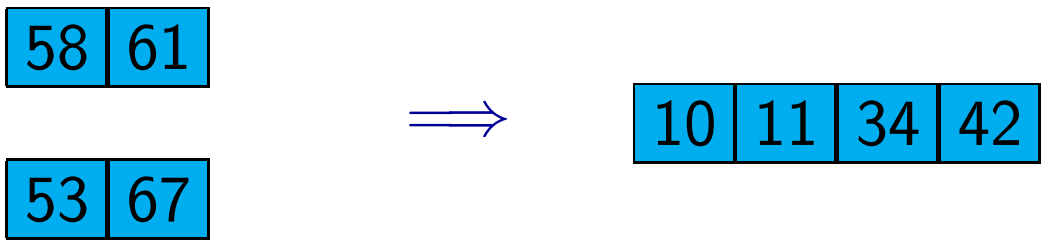


Rekurentní vztahy

Rekurentní vztahy se často používají při zkoumání časové složitosti rekurzivních 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.

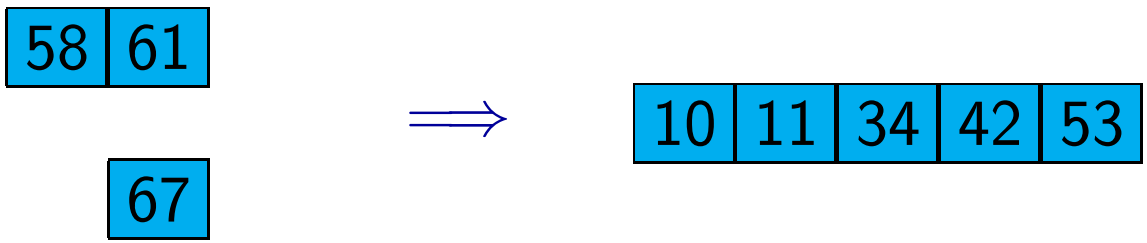


Rekurentní vztahy

Rekurentní vztahy se často používají při zkoumání časové složitosti rekurzivních 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.

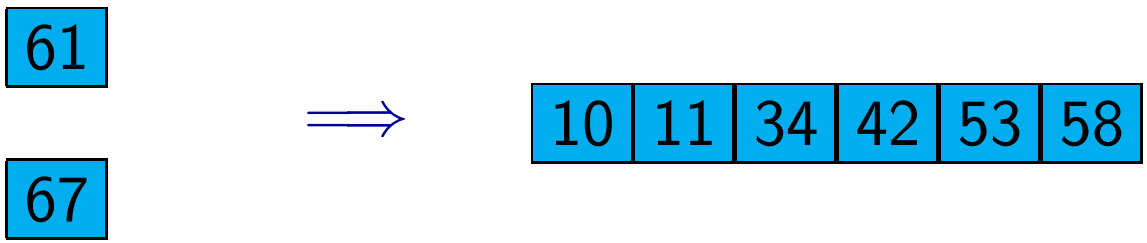


Rekurentní vztahy

Rekurentní vztahy se často používají při zkoumání časové složitosti rekurzivních 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.

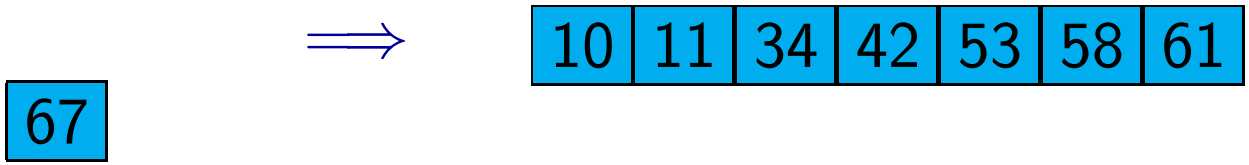


Rekurentní vztahy

Rekurentní vztahy se často používají při zkoumání časové složitosti rekurzivních 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.



Rekurentní vztahy

Rekurentní vztahy se často používají při zkoumání časové složitosti rekurzivních 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.

Rekurentní vztahy

MERGE-SORT(A, p, r)

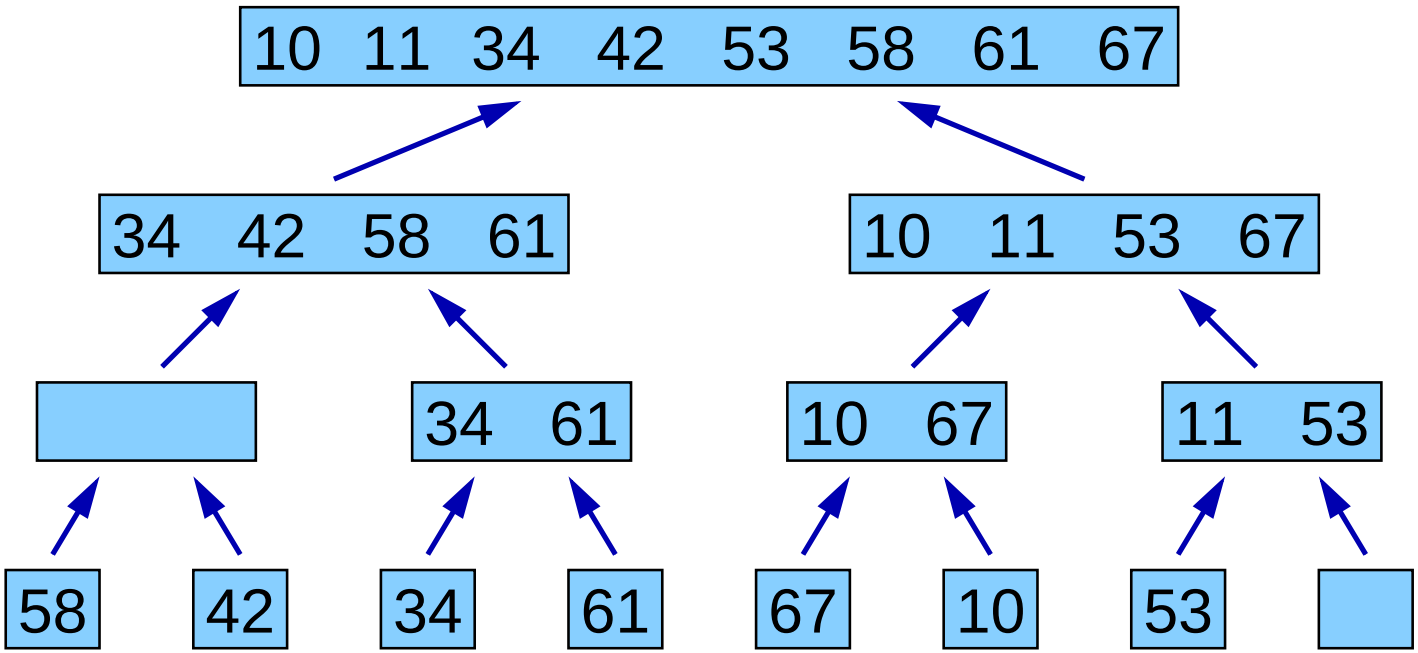
```
1  if r − p > 1
2      then q ← ⌊(p + r)/2⌋
3           MERGE-SORT(A, p, q)
4           MERGE-SORT(A, q, r)
5           MERGE(A, p, q, r)
```

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 $A[p \,..\, r-1]$.

Rekurentní vztahy

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



Rekurentní vztahy

Dobu výpočtu $T(n)$ algoritmu MERGE-SORT pro vstup velikosti n můžeme vyjádřit jako:

$$T(n) = \begin{cases} \Theta(1) & \text{pro } n = 1 \\ 2T(n/2) + \Theta(n) & \text{pro } n > 1 \end{cases}$$

Poznámka: $\Theta(1)$ označuje množinu všech funkcí jejichž hodnota je omezena konstantou.

Poznámka: Přesněji můžeme $T(n)$ vyjádřit jako

$$T(n) = \begin{cases} \Theta(1) & \text{pro } n = 1 \\ T(\lceil n/2 \rceil) + T(\lfloor n/2 \rfloor) + \Theta(n) & \text{pro } n > 1 \end{cases}$$

Rekurentní vztahy

Pokud bychom znali hodnoty všech konstant a přesné hodnoty funkcí, mohli bychom spočítat přesnou hodnotu $T(n)$ pro libovolné n.

Příklad:

$$T(n) = \begin{cases} 4 & \text{pro } n = 1 \\ T(\lceil n/2 \rceil) + T(\lfloor n/2 \rfloor) + 5n + 11 & \text{pro } n > 1 \end{cases}$$

$$\begin{aligned}
T(1) &= 4 \\
T(2) &= T(\lceil 2/2 \rceil) + T(\lfloor 2/2 \rfloor) + 5 \cdot 2 + 11 = T(1) + T(1) + 21 = 29 \\
T(3) &= T(\lceil 3/2 \rceil) + T(\lfloor 3/2 \rfloor) + 5 \cdot 3 + 11 = T(2) + T(1) + 26 = 59 \\
T(4) &= T(\lceil 4/2 \rceil) + T(\lfloor 4/2 \rfloor) + 5 \cdot 4 + 11 = T(2) + T(2) + 31 = 89 \\
T(5) &= T(\lceil 5/2 \rceil) + T(\lfloor 5/2 \rfloor) + 5 \cdot 5 + 11 = T(3) + T(2) + 36 = 124 \\
T(6) &= T(\lceil 6/2 \rceil) + T(\lfloor 6/2 \rfloor) + 5 \cdot 6 + 11 = T(3) + T(3) + 41 = 159 \\
T(7) &= T(\lceil 7/2 \rceil) + T(\lfloor 7/2 \rfloor) + 5 \cdot 7 + 11 = T(4) + T(3) + 46 = 194 \\
&\dots
\end{aligned}$$

Rekurentní vztahy

- Přesné hodnoty funkcí většinou neznáme, místo nich známe jen asymptotické odhady.
- V tom případě bude výsledkem také jen asymptotický odhad.
- I když všechny přesné hodnoty známe, může nám stačit jen asymptotický odhad, pokud by přesné odvození bylo příliš komplikované.

Rekurentní vztahy

Příklad: $T(n) = \begin{cases} \Theta(1) & \text{pro } n = 1 \\ 2T(\lfloor n/2 \rfloor) + n & \text{pro } n > 1 \end{cases}$

Uhodneme, že řešením je $T(n) = O(n \log n)$, a dokážeme, že $T(n) \leq cn \lg n$ pro nějakou vhodně zvolenou konstantu c:

$$\begin{aligned} T(n) &\leq 2(c\lfloor n/2 \rfloor \lg(\lfloor n/2 \rfloor)) + n \\ &\leq cn \lg(n/2) + n \\ &= cn \lg n - cn \lg 2 + n \\ &= cn \lg n - cn + n \\ &\leq cn \lg n \end{aligned}$$

kde poslední nerovnost platí pro libovolné $c \geq 1$.

Poznámka: $\lg n$ označuje $\log_2 n$.

Rekurentní vztahy

Zbývá ověřit, že platí báze indukce.

Předpokládejme na chvíli pro jednoduchost, že $T(1) = 1$.

Všimněte si, že vztah $T(n) \leq cn \lg n$ neplatí pro $n = 1$, ať už zvolíme jakoukoliv hodnotu $c \geq 1$ (protože $\lg 1 = 0$), a báze indukce tedy neplatí!

Řešení: Pokud chceme odvodit pouze asymptotický odhad, stačí ukázat, že $T(n) \leq cn \lg n$ platí pro $n \geq n_0$, kde n_0 je nějaká vhodně zvolená konstanta.

Zvolíme-li například $n_0 = 2$, stačí ukázat, že pro nějaké c platí pro libovolné $n \geq 2$ vztah $T(n) \leq cn \lg n$:

- Báze: $T(2) \leq c2 \lg 2$, $T(3) \leq c3 \lg 3$
  Platí pro $c = 2$, neboť $T(2) = 4$ a $T(3) = 5$
- Indukční krok: pro $n \geq 4$ hodnota $T(n)$ nezávisí (přímo) na $T(1)$.

Rekurentní vztahy

Při používání asymptotické notace je třeba dávat pozor, protože snadno můžeme udělat chybu při zanedbávání konstant.

Příklad:

$$T(n) = \begin{cases} \Theta(1) & \text{pro } n = 1 \\ 2T(\lfloor n/2 \rfloor) + n & \text{pro } n > 1 \end{cases}$$

Uhodneme, že $T(n) = O(n)$ a tedy $T(n) \leq cn$ pro nějakou konstantu c.

$$\begin{aligned} T(n) &\leq 2(c\lfloor n/2 \rfloor) + n \\ &\leq cn + n \\ &= O(n) \qquad \Longleftarrow \text{Chyba!} \end{aligned}$$

Výše uvedené odvození je špatně, protože jsme nedokázali, že $T(n) \leq cn$ platí pro tutéž konstantu c, která byla použita v indukčním předpokladu.

Rekurentní vztahy

Poznámka:
Nadále budeme předpokládat, že pro „malé“ hodnoty n je funkce $T(n)$ omezena nějakou konstantou, tj. že existují nějaké konstanty c, d takové, že

$$n \leq d \quad \Rightarrow \quad T(n) \leq c$$

Tento předpoklad nebudeme nadále explicitně uvádět.

Pokud se například hodnota $T(1)$ změní, výsledná hodnota $T(n)$ se většinou změní nanejvýš o konstantu, ale asymptotický růst funkce $T(n)$ se nezmění.

Rekurentní vztahy

Lemma

Nechť $a_1, \ldots, a_k, c$ *jsou kladné konstanty takové, že* $a_1 + \ldots + a_k < 1$ *a pro funkci* $T : \mathbb{N} \to \mathbb{N}$ *platí*

$$T(n) \leq T(\lceil a_1 n \rceil) + T(\lceil a_2 n \rceil) + \ldots + T(\lceil a_k n \rceil) + cn\,.$$

Pak $T(n) = O(n)$.

Důkaz: Zvolme $\varepsilon > 0$ takové, že $a_1 + \ldots + a_k < 1 - 2\varepsilon$. Pak pro nějaké dostatečně velké n_0 platí pro všechna $n \geq n_0$

$$\lceil a_1 n \rceil + \ldots + \lceil a_k n \rceil \leq (1 - \varepsilon) n$$

Chceme ukázat, že pro nějaké vhodně zvolené d platí pro všechna $n \geq 1$

$$T(n) \leq dn$$

Rekurentní vztahy

- Báze: Pro $n \leq n_0$ stačí, aby pro d platilo

$$d > \max\{T(n)/n \mid 1 \leq n \leq n_0\}$$

- Indukční krok:

$$\begin{aligned} T(n) &\leq T(\lceil a_1 n \rceil) + \ldots + T(\lceil a_k n \rceil) + cn \\ &\leq d \cdot \lceil a_1 n \rceil + \ldots + d \cdot \lceil a_k n \rceil + cn \\ &= d \cdot (\lceil a_1 n \rceil + \ldots + \lceil a_k n \rceil) + cn \\ &\leq d \cdot (1 - \varepsilon) n + cn \\ &= dn - d\varepsilon n + cn \\ &= dn - (d\varepsilon - c) n \\ &\leq dn \end{aligned}$$

Poslední nerovnost platí, pokud d zvolíme tak, že $d\varepsilon > c$.

Rekurentní vztahy

Lemma

Nechť $k \geq 2$ *a* $a_1, \ldots, a_k, c$ *jsou kladné konstanty takové, že* $a_1 + \ldots + a_k = 1$ *a pro funkci* $T : \mathbb{N} \to \mathbb{N}$ *platí*

$$T(n) \leq T(\lceil a_1 n \rceil) + T(\lceil a_2 n \rceil) + \ldots + T(\lceil a_k n \rceil) + cn\,.$$

Pak $T(n) = O(n \log n)$.

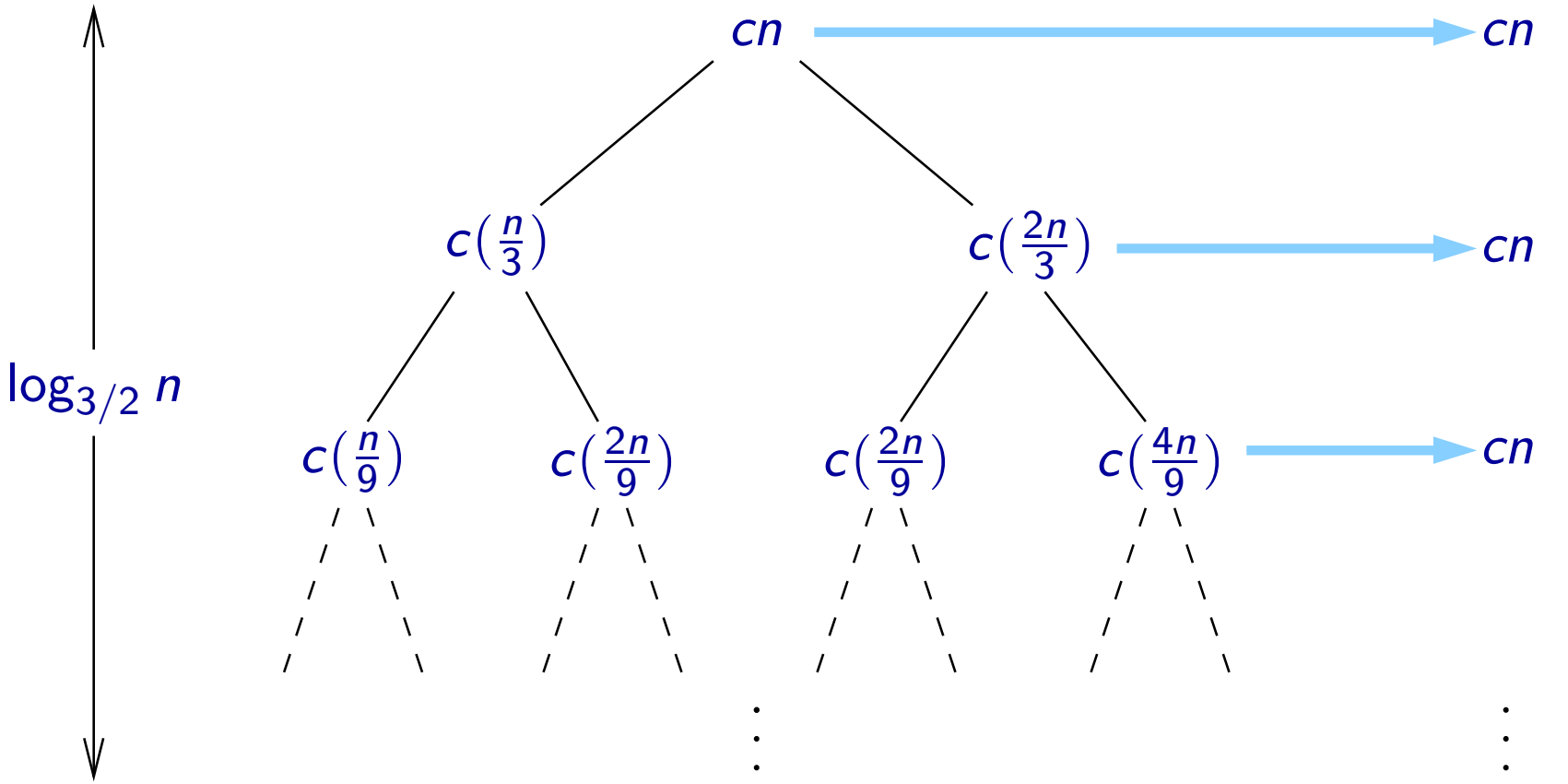
Důkaz by byl podobný jako v předchozím případě (o něco komplikovanější).

Ideu budeme ilustrovat na příkladě

$$T(n) = T(n/3) + T(2n/3) + cn$$

Ukážeme, že $T(n) = O(n \log n)$. Pro jednoduchost zanedbáme zaokrouhlování.

Rekurentní vztahy



Celkem: $O(n \log n)$

Rekurentní vztahy

Věta (Master theorem)

Nechť $a \geq 1$ a $b > 1$ jsou konstanty, $f : \mathbb{N} \to \mathbb{N}$ je funkce a funkce $T : \mathbb{N} \to \mathbb{N}$ je definována vztahem

$$T(n) = aT(n/b) + f(n)$$

Pak platí:

- Je-li $f(n) = O(n^c)$, kde $c < \log_b a$, pak $T(n) = \Theta(n^{\log_b a})$.
- Je-li $f(n) = \Theta(n^{\log_b a})$, pak $T(n) = \Theta(n^{\log_b a} \log n)$.
- Je-li $f(n) = \Theta(n^c)$, kde $c > \log_b a$, pak $T(n) = \Theta(n^c)$.

Poznámka: Zápis n/b ve výše uvedené větě, může znamenat jak $\lfloor n/b \rfloor$, tak $\lceil n/b \rceil$.

Složitost problémů

Složitost problémů

- Ukazuje se, že různé (algoritmické) problémy jsou různě těžké.
- Obtížnější jsou ty problémy, k jejichž řešení potřebujeme více času a paměti.
- Obtížnost problémů chceme nějak posuzovat, a to jak
  - absolutně – kolik času a kolik paměti potřebujeme k jejich řešení, tak
  - relativně – o kolik je jejich řešení obtížnější nebo naopak jednodušší oproti jiným problémům.
- Proč se u některých problémů nedaří nalézt efektivní algoritmy? Může vůbec nějaký efektivní algoritmus pro daný problém existovat?
- Kde přesně jsou limity toho, co můžeme prakticky zvládnout?

Složitost problémů

Je potřeba odlišovat **složitost algoritmu** a **složitost problému**.

Pokud například zkoumáme časovou složitost v nejhorším případě, mohli bychom neformálně říct:

- **složitost algoritmu** – funkce, která vyjadřuje, kolik kroků maximálně udělá daný algoritmus pro vstup velikosti n
- **složitost problému** – jaká je časová složitost „nejoptimálnějšího“ algoritmu, který řeší daný problém

Zavedení pojmu „složitost problému“ ve výše uvedeném smyslu naráží na značné technické obtíže. Pojem „složitost problému“ se tedy jako takový nedefinuje, ale obchází se zavedením tzv. **tříd složitosti**.

Pokud pak hovoříme o složitosti problému, máme tím na mysli to, do kterých tříd složitosti daný problém patří resp. nepatří.

Třídy složitosti

Třídy složitosti jsou podmnožiny množiny všech (algoritmických) **problémů**.

Daná konkrétní třída složitosti je vždy charakterizována nějakou vlastností, kterou mají problémy do ní patřící.

Typickým příkladem takové vlastnosti je vlastnost, že pro daný problém existuje nějaký algoritmus s určitým omezením (např. časové nebo prostorové složitosti):

- Do dané třídy pak patří všechny problémy, pro které takovýto algoritmus existuje.
- Naopak do ní nepatří problémy, pro které žádný takový algoritmus neexistuje.

Třídy složitosti

Definice

Pro libovolnou funkci $f : \mathbb{N} \to \mathbb{N}$ definujeme třídu $\mathcal{T}(f(n))$ jako třídu obsahující právě ty problémy, pro něž existuje algoritmus s časovou složitostí $O(f(n))$.

Příklad:

- $\mathcal{T}(n)$ – třída všech problémů pro něž existuje algoritmus s časovou složitostí $O(n)$
- $\mathcal{T}(n^2)$ – třída všech problémů pro než existuje algoritmus s časovou složitostí $O(n^2)$
- $\mathcal{T}(n \log n)$ – třída všech problémů pro než existuje algoritmus s časovou složitostí $O(n \log n)$

Třídy složitosti

Definice

Pro libovolnou funkci $f : \mathbb{N} \to \mathbb{N}$ definujeme třídu $\mathcal{S}(f(n))$ jako třídu obsahující právě ty problémy, pro něž existuje algoritmus s prostorovou složitostí $O(f(n))$.

Příklad:

- $\mathcal{S}(n)$ – třída všech problémů pro něž existuje algoritmus s prostorovou složitostí $O(n)$
- $\mathcal{S}(n^2)$ – třída všech problémů pro než existuje algoritmus s prostorovou složitostí $O(n^2)$
- $\mathcal{S}(n \log n)$ – třída všech problémů pro než existuje algoritmus s prostorovou složitostí $O(n \log n)$

Třídy složitosti

Poznámka:

Všimněte si, že u tříd $\mathcal{T}(f)$ a $\mathcal{S}(f)$ může to, které problémy do dané třídy patří, záviset na použitém výpočetním modelu (zda je to stroj RAM, jednopáskový Turingův stroj, vícepáskový Turingův stroj, ...).

Třídy složitosti

Pomocí tříd $\mathcal{T}(f(n))$ a $\mathcal{S}(f(n))$ můžeme definovat třídy PTIME a PSPACE jako

$$\text{PTIME} = \bigcup_{k \geq 0} \mathcal{T}(n^k) \qquad \text{PSPACE} = \bigcup_{k \geq 0} \mathcal{S}(n^k)$$

- PTIME je třída všech problémů, pro které existuje algoritmus s polynomiální časovou složitostí, tj. s časovou složitostí $O(n^k)$, kde k je nějaká konstanta.
- PSPACE je třída všech problémů, pro které existuje algoritmus s polynomiální prostorovou složitostí, tj. s prostorovou složitostí $O(n^k)$, kde k je nějaká konstanta.

Třídy složitosti

Poznámka: Vzhledem k tomu, že všechny (rozumné) výpočetní modely jsou schopné se navzájem simulovat tak, že při dané simulaci nevzroste počet kroků ani množství použité paměti víc než polynomiálně, není definice tříd PTIME a PSPACE závislá na použitém výpočetním modelu. Pro jejich zadefinování můžeme použít kterýkoliv výpočetní model.

Říkáme, že tyto třídy jsou **robustní** – jejich definice nezávisí na použitém výpočetním modelu.

Třídy složitosti

Analogicky můžeme zavést další třídy:

EXPTIME – množina všech problémů, pro které existuje algoritmus s časovou složitostí $2^{O(n^k)}$, kde k je nějaká konstanta

EXPSPACE – množina všech problémů, pro které existuje algoritmus s prostorovou složitostí $2^{O(n^k)}$, kde k je nějaká konstanta

LOGSPACE – množina všech problémů, pro které existuje algoritmus s prostorovou složitostí $O(\log n)$

Poznámka: Místo $2^{O(n^k)}$ bychom mohli psát také $O(c^{n^k})$, kde c a k jsou nějaké konstanty.

Třídy složitosti

Při definici třídy LOGSPACE musíme přesněji specifikovat, co považujeme za prostorovou složitost algoritmu.

Uvažujeme například Turingův stroj, který pracuje se třemi páskami:

- **Vstupní páskou**, na které je na začátku výpočtu zapsán vstup. Z této pásky je možno pouze číst.
- **Pracovní páskou**, která je na začátku výpočtu prázdná. Z této pásky je možno číst i na ni zapisovat.
- **Výstupní páskou**, která je také na začátku výpočtu prázdná a na kterou je možno pouze zapisovat.

Množství použité paměti je pak definováno, jako počet použitých políček na pracovní pásce.

Třídy složitosti

Další příklady tříd složitosti:

2-EXPTIME – množina všech problémů, pro které existuje algoritmus s časovou složitostí $2^{2^{O(n^k)}}$, kde k je nějaká konstanta

2-EXPSPACE – množina všech problémů, pro které existuje algoritmus s prostorovou složitostí $2^{2^{O(n^k)}}$, kde k je nějaká konstanta

ELEMENTARY – množina všech problémů, pro které existuje algoritmus s časovou (či prostorovou) složitostí

$$2^{2^{2^{\cdot^{\cdot^{\cdot^{2^{2^{O(n^k)}}}}}}}}}$$

kde k je konstanta a počet exponentů je omezen konstantou.

Vztahy mezi třídami složitosti

Pokud Turingův stroj provede m kroků, tak použije maximálně m políček na pásce.

Pokud tedy existuje pro nějaký problém algoritmus s časovou složitostí $O(f(n))$, má tento algoritmus paměťovou složitost (nejvýše) $O(f(n))$.

Je tedy zřejmé, že platí následující vztah.

Pozorování

Pro libovolnou funkci $f : \mathbb{N} \to \mathbb{N}$ platí $\mathcal{T}(f(n)) \subseteq \mathcal{S}(f(n))$.

Poznámka: Analogicky bychom mohli argumentovat například pro stroj RAM.

Vztahy mezi třídami složitosti

Z předchozího okamžitě plyne:

$$\begin{aligned} \text{PTIME} &\subseteq \text{PSPACE} \\ \text{EXPTIME} &\subseteq \text{EXPSPACE} \\ \text{2-EXPTIME} &\subseteq \text{2-EXPSPACE} \\ &\vdots \end{aligned}$$

Vzhledem k tomu, že polynomiální funkce rostou pomaleji než exponenciální, zjevně platí:

$$\text{PTIME} \subseteq \text{EXPTIME} \subseteq \text{2-EXPTIME} \subseteq \cdots$$

$$\text{LOGSPACE} \subseteq \text{PSPACE} \subseteq \text{EXPSPACE} \subseteq \text{2-EXPSPACE} \subseteq \cdots$$

Vztahy mezi třídami složitosti

Věta

Pro libovolnou prostorově zkonstruovatelnou funkci $f : \mathbb{N} \to \mathbb{N}$ existuje problém, který je možné řešit algoritmem s prostorovou složitostí $O(f(n))$, ale není možné řešit žádným algoritmem s prostorovou složitostí $o(f(n))$.

Důkaz tohoto tvrzení je netriviální a značně přesahuje rozsah tohoto předmětu.

Poznámka: Funkce $f : \mathbb{N} \to \mathbb{N}$, kde $f(n) \geq \log n$, se nazývá prostorově zkonstruovatelná, jestliže existuje algoritmus s prostorovou složitostí $O(f(n))$, který pro vstup w, kde $|w| = n$, vytvoří binární reprezentaci hodnoty $f(n)$.

Prakticky všechny „rozumné“ funkce mají tuto vlastnost.

Vztahy mezi třídami složitosti

Důsledek

Pro libovolné dvě funkce $f_1, f_2 : \mathbb{N} \to \mathbb{N}$ takové, že $f_1(n)$ je v $o(f_2(n))$ a $f_2(n)$ je prostorově zkonstruovatelná, platí, že $\mathcal{S}(f_1(n)) \subsetneq \mathcal{S}(f_2(n))$.

Z tohoto tvrzení okamžitě plynou následující důsledky:

- Pro libovolná dvě reálná čísla $0 \leq \epsilon_1 < \epsilon_2$ platí

$$\mathcal{S}(n^{\epsilon_1}) \subsetneq \mathcal{S}(n^{\epsilon_2})$$

- LOGSPACE $\subsetneq$ PSPACE
- PSPACE $\subsetneq$ EXPSPACE
- EXPSPACE $\subsetneq$ 2-EXPSPACE

Vztahy mezi třídami složitosti

Věta

Pro libovolnou časově zkonstruovatelnou funkci $t : \mathbb{N} \to \mathbb{N}$ existuje problém, který je možné řešit algoritmem s časovou složitostí $O(t(n))$, ale není možné řešit žádným algoritmem s časovou složitostí $o(t(n)/\log t(n))$.

Poznámka: Funkce $t : \mathbb{N} \to \mathbb{N}$, kde $t(n) \geq \log n$, se nazývá časově zkonstruovatelná, jestliže existuje algoritmus s časovou složitostí $O(t(n))$, který pro vstup w, kde $|w| = n$, vytvoří binární reprezentaci hodnoty $t(n)$.

Důsledek

Pro libovolné dvě funkce $t_1, t_2 : \mathbb{N} \to \mathbb{N}$ takové, že $t_1(n)$ je v $o(t_2(n)/\log t_2(n))$ a $t_2(n)$ je časově zkonstruovatelná, platí, že $\mathcal{T}(f_1(n)) \subsetneq \mathcal{T}(f_2(n))$.

Vztahy mezi třídami složitosti

Z předchozího plynou následující důsledky:

- Pro libovolná dvě reálná čísla $0 \leq \epsilon_1 < \epsilon_2$ platí

$$\mathcal{T}(n^{\epsilon_1}) \subsetneq \mathcal{T}(n^{\epsilon_2})$$

- PTIME $\subsetneq$ EXPTIME
- EXPTIME $\subsetneq$ 2-EXPTIME

Vztahy mezi třídami složitosti

Při zkoumání vztahů mezi třídami složitosti se ukazuje jako užitečný pojem **konfigurace**.

Konfigurací budeme rozumět celkový stav, ve kterém se během jednoho kroku nachází stroj, provádějící nějaký daný algoritmus.

- U Turingova stroje je konfigurace dána stavem jeho řídící jednotky, obsahem pásky (resp. pásek) a pozicí hlavy (resp. hlav).
- U stroje RAM je konfigurace dána obsahem paměti, obsahem všech registrů (včetně IP) a pozicemi čtecí a zapisovací hlavy.

Vztahy mezi třídami složitosti

Mělo by být jasné, že konfigurace (resp. jejich popisy) můžeme zapisovat jako slova v nějaké abecedě.

Navíc můžeme konfigurace zapisovat tak, že délka těchto slov bude zhruba stejná jako množství paměti použité algoritmem (tj. počet políček na pásce použitých Turingovým stojem, počet bitů paměti použitých strojem RAM apod.).

Poznámka: Pokud máme abecedu Σ, kde $|\Sigma| = c$ tak:

- Počet slov délky n je c^n, tj. $2^{\Theta(n)}$.
- Počet slov délky nejvýše n je

$$\sum_{i=0}^{n} c^n = \frac{c^{n+1} - 1}{c - 1}$$

tj. také $2^{\Theta(n)}$.

Vztahy mezi třídami složitosti

Je jasné, že během výpočtu korektního algoritmu se žádná konfigurace nemůže zopakovat, protože jinak by se algoritmus zacyklil a běžel by donekonečna.

Pokud tedy víme, že paměťová složitost nějakého algoritmu je $O(f(n))$, znamená to, že počet různých konfigurací dosažitelných během výpočtu je $2^{O(f(n))}$.

Protože se konfigurace během žádného výpočtu neopakují, je i časová složitost daného algoritmu maximálně $2^{O(f(n))}$.

Pozorování

Pro libovolnou funkci $f : \mathbb{N} \to \mathbb{N}$ platí, že $\mathcal{S}(f(n)) \subseteq \mathcal{T}(2^{f(n)})$.

Vztahy mezi třídami složitosti

Z předchozího plynou následující důsledky:

$$\begin{aligned} \text{LOGSPACE} &\subseteq \text{PTIME} \\ \text{PSPACE} &\subseteq \text{EXPTIME} \\ \text{EXPSPACE} &\subseteq \text{2-EXPTIME} \\ &\vdots \end{aligned}$$

Vztahy mezi třídami složitosti

Shrnutí:

$$\text{LOGSPACE} \subseteq \text{PTIME} \subseteq \text{PSPACE} \subseteq \text{EXPTIME} \subseteq \text{EXPSPACE} \subseteq$$
$$\subseteq \text{2-EXPTIME} \subseteq \text{2-EXPSPACE} \subseteq \cdots \subseteq \text{ELEMENTARY}$$

Navíc je známo, že:

- $\text{PTIME} \subsetneq \text{EXPTIME} \subsetneq \text{2-EXPTIME} \subsetneq \cdots$
- $\text{LOGSPACE} \subsetneq \text{PSPACE} \subsetneq \text{EXPSPACE} \subsetneq \text{2-EXPSPACE} \subsetneq \cdots$

Horní a dolní odhady složitosti problémů

Horním odhadem složitosti problému rozumíme to, že složitost problému není vyšší než nějaká uvedená.

Většinou je to formulováno tak, že daný problém patří do nějaké určité třídy složitosti.

Příklady tvrzení, které se týkají horních odhadů složitosti:

- Problém nalezení nejkratší cesty v grafu je v PTIME.
- Problém ekvivalence dvou regulárních výrazů je v EXPSPACE.

Pokud chceme zjistit nějaký horní odhad složitosti problému, stačí ukázat, že existuje algoritmus s danou složitostí.

Horní a dolní odhady složitosti problémů

Dolním odhadem složitosti problému rozumíme to, že složitost problému je alespoň taková jako nějaká uvedená.

Většinou je to formulováno tak, že daný problém nepatří do nějaké určité třídy složitosti.

Obecně je zjišťování (netriviálních) dolních odhadů složitosti problémů mnohem obtížnější než zjišťování horních odhadů.

Pro odvození dolního odhadu musíme totiž ukázat, že neexistuje žádný algoritmus, který by řešil daný problém a přitom měl danou složitost.

Dolní odhad pro problém třídění

Problém „Třídění“

Vstup: Posloupnost prvků $a_1, a_2, \ldots, a_n$.

Výstup: Prvky $a_1, a_2, \ldots, a_n$ setříděné od nejmenšího po největší.

Ukážeme, že každý algoritmus, který řeší problém třídění a na prvcích tříděné posloupnosti používá pouze operaci porovnávání (tj. nezkoumá obsah těchto prvků), má složitost $\Omega(n \log n)$.

Dolní odhad pro problém třídění

Jestliže algoritmus nezkoumá obsah jednotlivých prvků jinak než jejich porovnáváním, je zřejmé, že pro činnost algoritmu je důležité pouze relativní pořadí prvků na vstupu a nikoliv jejich konkrétní hodnoty.

Můžeme se tedy omezit na případ, kdy vstupem jsou pouze permutace množiny $\{1, 2, \ldots, n\}$.

Budeme sledovat během výpočtu hodnotu registru IP stroje RAM, tj. která instrukce se v kterém kroku provádí.

Pokud předložíme stroji RAM dvě různé permutace, je jasné, že od nějakého okamžiku se budou hodnoty registru IP lišit (v důsledku porovnání dvou prvků a následného provedení nebo naopak neprovedení podmíněného skoku), protože pokud by se v obou výpočtech prováděly přesně stejné operace, v jednom z nich bychom dostali chybný výstup.

Dolní odhad pro problém třídění

Všechny možné výpočty nad vstupy velikosti n si můžeme představit jako strom, kde jednotlivé větve odpovídají jednotlivým výpočtům.

- Tento strom musí mít nejméně $n!$ listů, protože existuje $n!$ různých permutací n prvků.
- Výška tohoto stromu t (tj. délka jeho nejdelší větve) odpovídá časové složitosti algoritmu.
- Předpokládáme-li, že každý vrchol má nanejvýš c potomků, pak počet listů stromu je nanejvýš c^t.

Z toho plyne, že

$$c^t \geq n!$$

neboli

$$t \geq \log_c n!$$

Dolní odhad pro problém třídění

Vzhledem k tomu, že $\log_c n! = \Theta(n \log n)$, dostáváme požadovaný výsledek.

Poznámka: K dokázání toho, že $\log_c n! = \Theta(n \log n)$, stačí využít faktu, že $n! \leq n^n \leq (n!)^2$, z čehož plyne

$$\log n! \leq \log n^n \leq \log(n!)^2$$

$$\log n! \leq n \log n \leq 2 \log n!$$

$$\frac{1}{2} n \log n \leq \log n! \leq n \log n$$