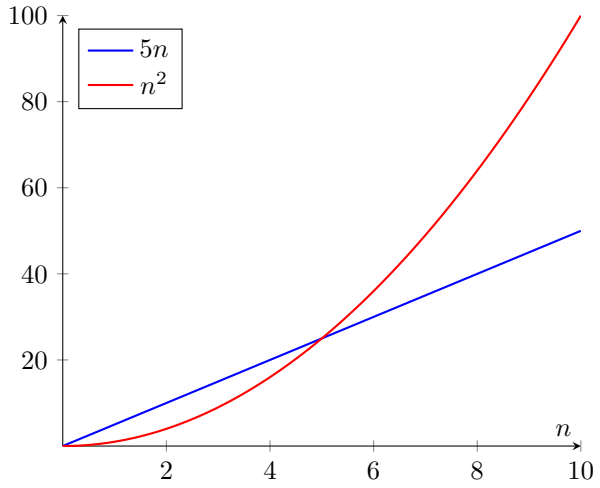
C2142: 2. cvičení

1. úkol: Zopakujte si formální definici asymptotické složitosti, tj. množin $\mathcal{O}(g)$, $\Omega(g)$, $\Theta(g)$. Vysvětlete význam konstant c a n_0.

Řešení:
Formálně lze definovat tyto množiny následovně:

$$\mathcal{O}(g) = \{f | \exists c \in \mathbb{R}^+, \exists n_0 \in \mathbb{N} : \forall n \in \mathbb{N}, n \geq n_0 : f(n) \leq c \cdot g(n)\}$$
$$\Omega(g) = \{f | \exists c \in \mathbb{R}^+, \exists n_0 \in \mathbb{N} : \forall n \in \mathbb{N}, n \geq n_0 : c \cdot g(n) \leq f(n)\}$$
$$\Theta(g) = \mathcal{O}(g) \cap \Omega(g)$$

• Konstanta c umožňuje ignorovat rozdíl daný pouze multiplikativní konstantou, tzn. že například funkce n^3, $15n^3$ nebo $0{,}10n^3$ všechny patří do třídy $\mathcal{O}(n^3)$. V rámci asymptotické složitosti tento rozdíl nepovažujeme za významný (ten je konstantní a nemění se s velikostí vstupu).
• Konstanta n_0 říká, že daná nerovnost musí platit až od určitého bodu. Uvažme například funkce $5n$ a n^2.



Názorně vidíme, že funkce n^2 je rychleji rostoucí, ale $5n \leq n^2$ obecně neplatí. Pro $n \geq 5$ už nerovnost splněna bude, takže můžeme psát $5n \in \mathcal{O}(n^2)$.

2. úkol: Ukažte, zdali funkce $f(n) = 3n^5 - 2$ patří do následujících množin:

- $\mathcal{O}(n^2)$
- $\mathcal{O}(n^5)$
- $\mathcal{O}(n^6)$

Řešení:
Pokud máme funkci zařadit do konkrétní množiny, je potřeba najít konstanty c a n_0, aby byla splněna definiční nerovnost, případně ukázat, že to nejde.
• Pro první příklad ukažme, že toho nelze docílit. Spočítejme podíl v následující limitě (limita pro $n \to \infty$ zaručuje, že $n \geq n_0$ pro libovolné n_0):

$$\lim_{n \to \infty} \frac{3n^5 - 2}{n^2} = \infty$$

Protože je výsledkem ∞, uvažovaná funkce (v čitateli) roste výrazně rychleji než n^2, takže nemůže patřit do množiny $\mathcal{O}(n^2)$.

• Ve druhém případě je triviálně nerovnost splněna volbou $c = 3, n_0 = 1$. Uvědomme si, že není potřeba najít nejmenší c a n_0. Definice zmiňuje, že tyto konstanty pouze *musí existovat*, takže stejně dobře funguje v tomto případě i volba $c = 100, n_0 = 16$ nebo $c = 2, n_0 = 123$ atd.

• Do třetí množiny funkce $f(n)$ patří, protože už jsme ukázali, že patří do $\mathcal{O}(n^5)$ a zároveň platí $n^5 \leq n^6$ pro všechna $n > 1$.

3. úkol: Seřaďte následující funkce dle rychlosti růstu (ve smyslu asymptotické složitosti):

$$n^2 + \log n, (3/2)^n, n, \log(n!), 7n^5 - n^3 + n, 2^n, n \log n, n^n, 2^{\log_2 n}, n^2, \log n, n!, 6$$

Řešení:

$$\begin{array}{l} 6 \\ \log n \\ n, 2^{\log_2 n} \\ \log(n!), n \log n \\ n^2, n^2 + \log n \\ 7n^5 - n^3 + n \\ (3/2)^n \\ 2^n \\ n! \\ n^n \end{array}$$

Rozeberme některé zajímavé případy:

• 6 je konstanta, tato funkce tedy není ani rostoucí.

• $2^{\log_2 n} = n$, protože platí obecně $a^{\log_a n} = n$ (logaritmus a exponenciální funkce jsou vůči sobě inverzní).

• $\log(n!)$ a $n \log n$ patří do stejné složitostní třídy.
Z jedné strany platí:

$$\log(n!) = \log 1 + \log 2 + ... + \log n \leq \log n + \log n + ... + \log n = n \log n$$

Naopak lze ukázat i opačnou nerovnost (vynecháme prvních $n/2$ prvků):

$$\log(n!) = \log 1 + \log 2 + ... + \log n \geq \log(n/2) + \log(n/2 + 1) + ... + \log(n) \geq \\ \log(n/2) + \log(n/2) + ... + \log(n/2) = (n/2) \log(n/2)$$

Rozdíl je zde pouze v multiplikativní konstantě. Odtud můžeme říct, že $\mathcal{O}(\log(n!)) = \mathcal{O}(n \log n)$.

• Od určitého n_0 zjevně platí $\log n \leq n^2$, takže $n^2 + \log n \leq 2n^2$.

• Poznámka: Logaritmy o různých základech rostou asymptoticky stejně rychle. Připomeňme převodní vztah mezi logaritmy o různém základu.

$$\log_a n = \frac{\log_b n}{\log_b a} \to \log_b a \cdot \log_a n = \log_b n.$$

Zde $\log_b a$ je pouze multiplikativní konstanta. To je důvod, proč stačí vždy psát pouze $\mathcal{O}(\log n)$ a neuvádět základ logaritmu.

• Naopak exponenciální funkce rostou různě rychle v závislosti na základu. Ověřme výpočtem limity:

$$\lim_{n\to\infty} \frac{(3/2)^n}{2^n} = \lim_{n\to\infty} \left(\frac{3/2}{2/1}\right)^n = \lim_{n\to\infty} (3/4)^n = 0$$

Zde je vidět, že rozdíl není pouze multiplikativní konstanta. Co bychom dostali, kdybychom v limitě přehodili čitatele a jmenovatele?

• *Poznámka: Při určování složitosti algoritmů nejsou tyto výpočty nutné, zde pouze ukazují, proč některé vztahy platí. Pro praxi si stačí pamatovat základní řazení složitostních tříd z přednášky.*

4. úkol: Uvažme algoritmus, který pro vyřešení problému potřebuje provést 2^n operací. Na současném počítači lze za 1 den provést výpočet pro velikost problému $n = 36$. Kdybychom měli k dispozici 1000krát rychlejší počítač, jak velký problém bychom zvládli vyřešit ve stejném čase?

Řešení:

Jednoduchou trojčlenkou (1000krát rychlejší počítač je to samé jako původní počítač a čas 1000 dnů) lze dospět k řešení $n = 45$ (nebo 46 dle zaokrouhlení).

A co kdybychom měli milionkrát rychlejší počítač? Jaký je praktický důsledek u exponenciální složitosti?

5. úkol: Určete složitost následujícího algoritmu zapsaného v pseudokódu. Na vstupu uvažme pole celých čísel A o velikosti n.

```
1. S ← 0
2. for i ← 0 to n − 1 do
3.     S ← S + A[i]
4. end for
```

Řešení:

Při určování složitosti jednoduchého algoritmu můžeme postupovat *řádek po řádku* a určovat složitost každého z nich. Složitost celého algoritmu je pak jejich součet.

Jednoduché aritmetické nebo logické operace mají konstantní složitost, protože se přímo mapují na instrukce procesoru. U každého řádku můžeme tedy napsat jeho složitost.

Připomeňme, že pokud se v kódu vyskytuje výraz $A[i]$, reálně je nutné určit konkrétní adresu prvku výpočtem (viz minulý týden). To má ale zjevně konstantní složitost, protože použitý vzorec obsahuje stejný počet operací nezávisle na velikosti pole ani na hodnotě i.

Určeme tedy složitost jednotlivých řádků (nezávisle na ostatních):

```
1. S ← 0                                  O(1)
2. for i ← 0 to n − 1 do                  O(n)
3.     S ← S + A[i]                       O(1)
4. end for
5. return S                               O(1)
```

Lze vidět, že třetí řádek se vykonává v cyklu, tzn. přes všechny iterace je celkový počet operací daný třetím řádkem $\mathcal{O}(n)$.

Celková složitost algoritmu je tedy rovna součtu přes všechny řádky (výpočty probíhají za sebou), tj. $\mathcal{O}(n)$.

Poznámka: O tomto algoritmu bychom určitě mohli říct, že patří i do třídy třeba $\mathcal{O}(2^n)$, protože počet jeho kroků *roste nejvýše tak rychle jako* 2^n. Vždy ale udáváme ten nejtěsnější odhad.

Poznámka: Obecně platí, že u určování složitosti algoritmu se stačí zaměřit na cykly a rekurzivní volání funkcí. Všechny ostatní operace mají konstantní složitost. Pozor však na skryté cykly. Např. zjistit délku textového řetězce může mít lineární složitost, pokud tuto délku nemáme někde uloženu, protože je pak nutné projít řetězec znak po znaku.

Výzva v návaznosti na předchozí tvrzení! Zkuste napsat program (v libovolném programovacím jazyce), který neobsahuje cykly ani rekurzivní volání funkce, jehož výpočet bude trvat netriviálně dlouho, třeba alespoň sekundu.

6. úkol: Určete složitost následujícího algoritmu zapsaného v pseudokódu. Na vstupu uvažme pole celých čísel A o velikosti n.

```
1. for i ← 0 to 9999 do
2.     if i < n then
3.         print(A[i])
4.     end if
5. end for
```

Řešení:

Nejprve řekněme, že funkci *print* budeme považovat při výpisu jednoho čísla za konstantní. Tento algoritmus má ale i celkově konstantní složitost, tj. $\mathcal{O}(1)$.

Proč? Počet volání funkce *print* je omezen velikostí vstupu n, takže by se mohlo zdát, že je složitost *lineární*. Celkový počet iterací je ale i tak shora omezen konstantou (číslem 10000).

7. úkol: Určete složitost následujícího algoritmu zapsaného v pseudokódu. Na vstupu uvažme pole celých čísel A o velikosti n.

```
1. for i ← 0 to n − 1 do
2.     for j ← 0 to n − 1 do
3.         print(A[i] · A[j])
4.     end for
5. end for
```

Řešení:

Algoritmus obsahuje dva vnořené *for* cykly, každý má n iterací, uvnitř je konstantní operace. Jednotlivé složitosti tedy násobíme. Celkově dostáváme kvadratický algoritmus, tj. $\mathcal{O}(n^2)$.

8. úkol: Určete složitost následujícího algoritmu zapsaného v pseudokódu. Na vstupu uvažme pole celých čísel A o velikosti n.

```
1. for i ← 0 to n − 1 do
2.     for j ← i to n − 1 do
3.         print(A[i] · A[j])
4.     end for
5. end for
```

Řešení:

Algoritmus opět obsahuje dva vnořené *for* cykly, počet iterací vnitřního je však závislý na aktuální hodnotě i. Kolikrát tedy bude proveden příkaz na třetím řádku? V první iteraci vnějšího cyklu platí, že $i = 0$, takže vnitřní cyklus bude mít n iterací. V druhé iteraci vnějšího cyklu je $i = 1$, takže vnitřní cyklus bude mít o jednu méně, tedy $n - 1$, další zase o jednu méně atd. Celkový počet iterací vnitřního cyklu tedy v součtu (přes všechny iterace vnějšího cyklu) odpovídá obecně výrazu:

$$n + (n-1) + (n-2) + ... + 2 + 1$$

Jaká je hodnota tohoto výrazu? Jedná se o jednoduchou aritmetickou posloupnost (s diferencí 1), součet tedy odpovídá výrazu:

$$(n+1)\frac{n}{2} = \frac{n^2+n}{2} \in \mathcal{O}(n^2)$$

Složitost uvedeného algoritmu je tedy kvadratická.

9. úkol: Určete složitost následujícího algoritmu zapsaného v pseudokódu. Na vstupu uvažme dvě přirozená čísla y, z. Výraz $\lfloor x \rfloor$ značí dolní celou část, takže např. $\lfloor 4{,}5 \rfloor = 4$.

```
1. x ← 0
2. while z > 0 do
3.     if z je liché: then
4.         x ← x + y
5.     end if
6.     y ← 2y
7.     z ← ⌊z/2⌋
8. end while
9. return x
```

Řešení:

Tento příklad je od předešlých odlišný tím, že obsahuje *while* cyklus, který nemá předem stanovený počet iterací. Všechny ostatní operace mají konstatní složitost (Jak ověříme, že je číslo liché v konstatním čase?), zajímá nás tedy jen počet iterací cyklu.

V cyklu je podmínka $z > 0$, kde z je přirozené číslo na vstupu. Musíme tedy najít, kde se z v kódu mění. Řádek 3 sice obsahuje z, ale jeho hodnota se pouze „čte“, takže ten počet iterací neovlivní. Řádek 7 pak jako jediný mění hodnotu z.

Při každé iteraci cyklu se hodnota z zmenší na polovinu (a případná desetinná část, pokud bylo z liché, se ořízne). Jakmile bude platit, že $z = 1$, tak v další iteraci po provedení řádku 7 se hodnota změní na 0 a cyklus končí.

Počet iterací tedy zjistíme, pokud odpovíme na otázku: „Kolikrát musím nějaké přirozené číslo vydělit dvěma (a případně oříznout desetinnou část), abych získal číslo 1?“

Předpokládejme nyní, že je naše zadané číslo mocninou dvojky. Můžeme ho tedy zapsat ve tvaru:

$$z = 2^k$$

Jak dlouho můžeme takové číslo dělit dvěma?

$$\begin{aligned} 2^k/2 &= 2^{k-1} \\ 2^{k-1}/2 &= 2^{k-2} \\ 2^{k-2}/2 &= 2^{k-3} \\ &\vdots \\ 2^2/2 &= 2^1 = 2 \\ 2^1/2 &= 2^0 = 1 \end{aligned}$$

Kolik takových dělení jsme udělali? Původní číslo bylo ve tvaru 2^k, poslední pak 2^0. Celkem jsme tedy provedli k takových dělení. Nyní potřebujeme už jen dát do vztahu k a původní číslo 2^k. Zjevně platí:

$$k = \log_2 (2^k)$$

Celková složitost algoritmu je tedy $\mathcal{O}(\log z)$.

Poznámka: Všimněte si, že složitost vůbec nezávisí na vstupním parametru y.

Poznámka: Pro jednoduchost jsme volili z jako mocninu dvojky. Toto odvození je samozřejmě ale platné pro libovolné přirozené z. Stačí zvolit výraz 2^k jako nejbližší větší mocninu dvojky a dosáhneme téhož výsledku.

Otázka: Co je výsledkem (x) tohoto algoritmu pro konkrétní vstupní parametry y a z?

10. úkol (bez řešení): Zkuste se zamyslet, kde se objevují (i mimo informatiku) problémy, které mají konstatní, lineární, kvadratickou nebo exponenciální složitost.