Cvičení 12

Příklad 1: Podrobně zdůvodněte, proč pro funkce

$$f(n) = 5n^3 + 2n^2 - 9n + 13 \qquad\qquad g(n) = n^3$$

platí $f \in O(g)$ a $f \in \Omega(g)$.
(Z předchozího pak vyvoďte, že tedy také platí $g \in O(f)$, $g \in \Omega(f)$, $f \in \Theta(g)$ a $g \in \Theta(f)$.)

Řešení: Je třeba najít konstanty $c \in \mathbb{R}$ a $n_0 \in \mathbb{N}$ takové, že $c > 0$ a pro každé $n \geq n_0$ platí $f(n) \leq c \cdot g(n)$.
Snadno se ověří, že pro všechna $n \geq 1$ platí:

$$5n^3 \leq 5n^3 \qquad 2n^2 \leq 2n^3 \qquad -9n \leq 9n^3 \qquad 13 \leq 13n^3$$

Pokud tedy zvolíme $c = 5 + 2 + 9 + 13 = 29$ a $n_0 = 1$, tak pro všechna $n \geq n_0$ platí

$$f(n) = 5n^3 + 2n^2 - 9n + 13 \leq 5n^3 + 2n^3 + 9n^3 + 13n^3 = 29 \cdot n^3 = c \cdot g(n)\,.$$

Příklad 2: Následující funkce seřaďte podle rychlosti jejich růstu, tj. seřaďte je do posloupnosti $g_1, g_2, \ldots, g_{15}$, kde $g_1 \in O(g_2)$, $g_2 \in O(g_3)$, $\ldots$, $g_{14} \in O(g_{15})$. Uveďte také, pro které dvojice funkcí g_i a g_{i+1} v této posloupnosti platí $g_i \in \Theta(g_{i+1})$.

n^2	2^n	n	$\log_2 n$	n^n
$n!$	$\log_2(n^2)$	$(\log_2 n)^2$	n^3	$\sqrt{n}$
2^{2^n}	10^n	n^{1000}	$\sqrt[3]{n}$	$n \log_2 n$

Řešení: Funkce je možno seřadit následujícím způsobem:

g_1	g_2	g_3	g_4	g_5	g_6	g_7	g_8	g_9	g_{10}
$\log_2 n$	$\log_2(n^2)$	$(\log_2 n)^2$	$\sqrt[3]{n}$	$\sqrt{n}$	n	$n \log_2 n$	n^2	n^3	n^{1000}

g_{11}	g_{12}	g_{13}	g_{14}	g_{15}
2^n	10^n	$n!$	n^n	2^{2^n}

Jediné funkce, které rostou stejně rychle jsou funkce g_1 a g_2, kde platí $\log_2 n \in \Theta(\log_2(n^2))$ a také $\log_2(n^2) \in \Theta(\log_2 n)$. (To, že tyto vztahy platí, plyne z toho, že $\log_2(n^2) = 2 \log_2 n$.)
Pro všechny ostatní funkce platí, že v této posloupnosti každá následující funkce roste rychleji než předchozí, tj. pro $i \geq 2$ platí $g_i \in O(g_{i+1})$, ale $g_{i+1} \notin O(g_i)$, a tedy $g_i \notin \Theta(g_{i+1})$.

Příklad 3: Pro následující trojice funkcí f_1, f_2, f_3 určete, které vztahy tvaru $f_i \in O(f_j)$, $f_i \in \Omega(f_j)$ a $f_i \in \Theta(f_j)$ platí a které ne.

Řešení: Následující řešení jsou prezentovány ve formě tabulek, kde jednotlivá políčka těchto tabulek odpovídají vztahům uvedeným v následující tabulce. Kroužky označují vztahy, které platí, křížky vztahy, které neplatí.

$f_1 \in O(f_2)$	$f_2 \in O(f_1)$	$f_1 \in O(f_3)$	$f_3 \in O(f_1)$	$f_2 \in O(f_3)$	$f_3 \in O(f_2)$
$f_1 \in \Omega(f_2)$	$f_2 \in \Omega(f_1)$	$f_1 \in \Omega(f_3)$	$f_3 \in \Omega(f_1)$	$f_2 \in \Omega(f_3)$	$f_3 \in \Omega(f_2)$
$f_1 \in \Theta(f_2)$	$f_2 \in \Theta(f_1)$	$f_1 \in \Theta(f_3)$	$f_3 \in \Theta(f_1)$	$f_2 \in \Theta(f_3)$	$f_3 \in \Theta(f_2)$

a) $f_1(n) = 3n^2 + 5n - 1, \quad f_2(n) = 2n^3 - 15n - 183, \quad f_3(n) = (n+1)(n-1)$

Řešení:

○	×	○	○	×	○
×	○	○	○	○	×
×	×	○	○	×	×

b) $f_1(n) = 4n^2 + n^2 \log_2 n, \quad f_2(n) = \log_2^5 n, \quad f_3(n) = 17n + 3$

Řešení:

×	○	×	○	○	×
○	×	○	×	×	○
×	×	×	×	×	×

c) $f_1(n) = n\sqrt[5]{n}, \quad f_2(n) = n, \quad f_3(n) = \sqrt{n}$

Řešení:

×	○	×	○	×	○
○	×	○	×	○	×
×	×	×	×	×	×

d) $f_1(n) = 2^n, \quad f_2(n) = n^{1024}, \quad f_3(n) = n!$

Řešení:

×	○	○	×	○	×
○	×	×	○	×	○
×	×	×	×	×	×

e) $f_1(n) = 2^n, \quad f_2(n) = n^n, \quad f_3(n) = n!$

Řešení:

○	×	○	×	×	○
×	○	×	○	○	×
×	×	×	×	×	×

f) $f_1(n) = 2^n, \quad f_2(n) = n^n, \quad f_3(n) = n^{\log_2 n}$

Řešení:

○	×	×	○	×	○
×	○	○	×	○	×
×	×	×	×	×	×

g) $f_1(n) = 10^n, \quad f_2(n) = 2^n, \quad f_3(n) = 2^{2^n}$

Řešení:

×	○	○	×	○	×
○	×	×	○	×	○
×	×	×	×	×	×

h) $f_1(n) = \log_{10}(n^2), \quad f_2(n) = \log_2 n, \quad f_3(n) = \log_2(n^2)$

Řešení:

○	○	○	○	○	○
○	○	○	○	○	○
○	○	○	○	○	○

i) $f_1(n) = n + \sqrt{n} \cdot \log_2 n$, $f_2(n) = n \cdot \log_2 n$, $f_3(n) = \sqrt{n} \cdot \log_2^2 n$

Řešení:

◯	×	×	◯	×	◯
×	◯	◯	×	◯	×
×	×	×	×	×	×

j) $f_1(n) = 2^n$, $f_2(n) = 2^{\sqrt{n}}$, $f_3(n) = n!$

Řešení:

×	◯	◯	×	◯	×
◯	×	×	◯	×	◯
×	×	×	×	×	×

k) $f_1(n) = n/2048$, $f_2(n) = \sqrt{n} \cdot 3n$, $f_3(n) = n + n \cdot \log_2 n$

Řešení:

◯	×	◯	×	×	◯
×	◯	×	◯	◯	×
×	×	×	×	×	×

l) $f_1(n) = (\log_2 n)^n$, $f_2(n) = n^n$, $f_3(n) = 10^{\sqrt{n}}$

Řešení:

◯	×	×	◯	×	◯
×	◯	◯	×	◯	×
×	×	×	×	×	×

Příklad 4: Určete co nejpřesněji časovou a paměťovou složitost Algoritmu 1.

Předpokládejte, že hodnota n udává počet prvků v poli A a že toto pole je indexováno od nuly.

Algoritmus 1: Třídění přímým výběrem

```
SELECTION-SORT(A, n):
    i := n − 1
    while i > 0 do
        k = 0
        for j := 1 to i do
            if A[k] < A[j] then
                k := j
        x := A[k];  A[k] := A[i];  A[i] := x
        i := i − 1
```

Řešení: Při průchodech cyklem **while** nabývá proměnná i postupně hodnot $n-1$, $n-2$,...,2,1. Cyklus **while** tedy proběhne vždy $(n-1)$ krát. V cyklu **for** nabývá proměnná j hodnot $1, 2, \ldots, i$. V jedné iteraci cyklu **while** se tedy tělo cyklu **for** provede i krát. Počet iterací cyklu **for** tedy závisí na aktuální hodnotě proměnné i, která se v průběhu výpočtu s každou iterací cyklu **while** mění.

Z kódu je snadno vidět, že z hlediska analýzy časové složitosti tohoto algoritmu je nejdůležitější určit celkový počet provedení těla cyklu **for**, neboť to jsou instrukce, které jsou během výpočtu prováděny nejčastěji. Doba, která se stráví prováděním jiných instrukcí, je pro velké hodnoty n zanedbatelná vůči času, který se stráví prováděním cyklu **for**.

Celkový počet provedení těla cyklu **for** se dá přesně spočítat jako součet aritmetické řady:

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

Doba jednoho provedení těla cyklu **for** je $\Theta(1)$. Množství času, které se stráví prováděním těla cyklu **for** je tedy $\Theta(n^2)\cdot\Theta(1)=\Theta(n^2)$. Z toho vidíme, že celková časová složitost algoritmu je $\Theta(n^2)$.

Algoritmus pracuje s polem, které má celkem n buněk. Pro další proměnné postačuje $\Theta(1)$ paměťových buněk. Paměťová složitost algoritmu je tedy $\Theta(n)$.

Příklad 5: Určete co nejpřesněji časovou a paměťovou složitost Algoritmu 2 (připomeňte si tento algoritmus z minulého cvičení).

(Předpokládejte, že hodnota n udává počet prvků v poli A, že toto pole je indexováno od nuly, a že x je hodnota hledaného prvku.)

Algoritmus 2: Binární vyhledávání

```
BSearch(x, A, n):
    ℓ := 0
    r := n
    while ℓ < r do
        k := ⌊(ℓ + r) / 2⌋
        if A[k] < x then
            ℓ := k + 1
        else
            r := k
    if ℓ < n and A[ℓ] = x then
        return ℓ
    return NotFound
```

Řešení: Co se týká paměťové složitosti, algoritmus BSearch pracuje s polem o n prvcích. Paměť potřebná pro ostatní proměnné je vůči paměti potřebné pro toto pole zanedbatelná. Celková paměťová složitost algoritmu BSearch je tedy $\Theta(n)$.

Zaměřme se nyní na jeho časovou složitost.

Nejprve si všimněme, že s každou iterací cyklu se hodnota $r-\ell$ snižuje „zhruba“ na polovinu. Na začátku je $r-\ell=n$ a postupně se tato hodnota snižuje až na nulu, kdy výpočet končí. Pokud bychom pro jednoduchost předpokládali, že se tato hodnota snižuje vždy přesně na polovinu a že n je mocninou dvojky, tj. $2^i=n$ pro nějaké $i\in\mathbb{N}$, je vidět, že počet iterací, které se provedou, než rozdíl $r-\ell$ dosáhne hodnoty 1, je i. (Je také jasné, že pokud $r-\ell=1$,

v další iteraci bude $r - \ell = 0$ a výpočet končí.) Z $2^i = n$ vyplývá $i = \log_2 n$. Protože doba strávená jednou iterací cyklu je $\Theta(1)$, je zřejmé, že celková doba výpočtu bude $\Theta(\log n)$.

Tento výsledek bude platit i v případě, kdy n není mocninou dvojky. Intuitivně je asi jasné, že se v tomto případě provede nanejvýš jedna iterace cyklu navíc oproti situaci, kdybychom n nahradili nejbližší mocninou dvojky.

Ve skutečnosti není úplně pravda, že se hodnota $r - \ell$ snižuje vždy přesně na polovinu — např. u lichých hodnot dochází při dělení dvěma k zaokrouhlování, nová hodnota rozdílu by možná mohla být ve skutečtnosti o jedna větší nebo menší než přesná polovina hodnoty původního rozdílu, apod.

Přesné určení těchto detailů vyžaduje podrobnější analýzu. Ve skutečnosti se ukazuje, že výsledek takové podrobnější analýzy je pak většinou úplně stejný, jako při výše uvedené „hrubé“ analýze. Na těchto detailech tedy v naprosté většině případů nezáleží a pokud nám jde jen o asymptotický odhad, tak je můžeme ignorovat.

Pro ilustraci zde uveďme, jak by mohla vypadat podrobnější analýza, ze které je vidět, že časová složitost algoritmu BSearch skutečně vyjde $\Theta(\log n)$ i v případě, kdy tyto detaily bereme v úvahu.

Nejprve se zaměřme na přesné určení toho, jak se snižuje s každou iterací cyklu hodnota rozdílu $r - \ell$. Jako ℓ a r označme hodnoty těchto proměnných před provedením dané iterace cyklu, a jako ℓ' a r' označme hodnoty těchto proměnných po provedení této iterace. (Předpokládejme, že $\ell < r$, protože jinak by se žádná iterace neprovedla a výpočet by skončil.)

Dále položme $d = r - \ell$, $d' = r' - \ell'$ a $k = \lfloor (\ell + r)/2 \rfloor$. Chtěli bychom co nejpřesněji vyjádřit vztah mezi hodnotami d a d'.

Je potřeba rozebrat dva případy, podle toho, zda je nebo není splněna podmínka $A[k] < x$:

a) Platí $A[k] < x$ a provede se přiřazení $\ell := k + 1$:

   V tomto případě je $\ell' = k + 1$ a $r' = r$, takže $d' = r' - \ell' = r - k - 1$.

   Z $k = \lfloor (\ell + r)/2 \rfloor$ vyplývá, že $(\ell + r) - 2 < 2k \leq \ell + r$. Protože $d' = r - k - 1$, tak $k = r - d' - 1$. Dosazením do předchozího vztahu dostáváme

   $$(\ell + r) - 2 < 2r - 2d' - 2 \leq \ell + r\,.$$

   Odečtením hodnoty $2r - 2$ ode všech členů těchto nerovností dotáváme $\ell - r < -2d' \leq \ell - r + 2$, z čehož plyne $r - \ell > 2d' \geq (r - \ell) - 2$. Protože $d = r - \ell$, tak dostáváme $d > 2d' \geq d - 2$. Tento vztah se dá přepsat jako $d - 1 \geq 2d' > (d - 1) - 2$, z čehož plyne, že $d' = \lfloor (d - 1)/2 \rfloor$.

b) Platí $A[k] \geq x$ a provede se přiřazení $r := k$:

   V tomto případě je $\ell' = \ell$ a $r' = k$, takže $d' = r' - \ell' = k - \ell$.

   Podobně jako v předchozím případě z $k = \lfloor (\ell + r)/2 \rfloor$ vyplývá, že $(\ell + r) - 2 < 2k \leq \ell + r$, a protože $d' = k - \ell$, tak $k = d' + \ell$, takže

   $$(\ell + r) - 2 < 2d' + 2\ell \leq \ell + r\,.$$

   Odečtením hodnoty 2ℓ dostáváme $r - \ell - 2 < 2d' \leq r - \ell$. Protože $d = r - \ell$, tak platí $d - 2 < 2d' \leq d$, z čehož vyplývá, že $d' = \lfloor d/2 \rfloor$.

Z předchozí analýzy vidíme, že v závislosti na tom, kterou větví příkazu **if** se v dané iteraci jde, tak je buď $d' = \lfloor (d-1)/2 \rfloor$ nebo $d' = \lfloor d/2 \rfloor$. Z hlediska doby výpočtu je horší ten druhý případ, tj. případ, kdy platí $A[k] \geq x$ a provede se přiřazení $r := k$.

Není těžké si rozmyslet, že pro každé n existují vstupy, kdy v každé iteraci cyklu nastává tento druhý případ (tj. přiřazení $r := k$) — stačí, aby pro všechny prvky v poli platilo $A[i] \geq x$. Pro takové vstupy pak platí, že v každé iteraci je $d' = \lfloor d/2 \rfloor$.

Zaměřme se tedy na tyto nejhorší případy. Pokud si zkusíme spočítat přesný počet iterací v těchto nejhorších případech pro některé malé hodnoty d, asi dojdeme k hypotéze, že pro tyto nejhorší případy platí následující (předpokládáme $i > 0$):

$$\textit{ve zbytku výpočtu algoritmus provede } i \textit{ iterací} \quad \textit{právě tehdy, když} \quad 2^{i-1} \leq d < 2^i$$

Toto tvrzení můžeme ověřit indukcí. Pro $i = 1$ určitě platí, protože v tomto případě musí být $d = 1$ (pro $d > 1$ se v nejhorším případě provedou alespoň dvě iterace cyklu).

Předpokládejme nyní, že toto tvrzení platí pro i a ukažme, že pak platí i pro $i+1$. Algoritmus provede $i+1$ iterací právě tehdy, když po provedení jedné iterace provede i iterací, což podle indukčního předpokladu platí právě tehdy, když $2^{i-1} \leq d' < 2^i$. Protože $d' = \lfloor d/2 \rfloor$, platí tento vztah právě pro ta d, kde $2^i \leq d < 2^{i+1}$. Tím je důkaz hotov.

Protože na začátku výpočtu je $d = n$, z předchozího vyplývá, že algoritmus provede i iterací v nejhorším případě právě pro ty vstupy, kde $2^{i-1} \leq n < 2^i$. Z toho plyne, že $i-1 \leq \log_2 n < i$, což můžeme přepsat jako $(\log_2 n) - 1 < i - 1 \leq \log_2 n$, z čehož vyplývá, že $i - 1 = \lfloor \log_2 n \rfloor$, a tedy $i = \lfloor \log_2 n \rfloor + 1$.

Pro vstup velikosti n tedy provede algoritmus v nejhorším případě přesně $\lfloor \log_2 n \rfloor + 1$ iterací, z čehož vyplývá, že časová složitost algoritmu je $\Theta(\log n)$.

Příklad 6: Popište pseudokódem libovolný algoritmus pro řešení následujícího problému a určete co nejpřesněji jeho časovou a paměťovou složitost. (Co je vhodné v případě tohoto problému považovat za velikost vstupu?)

VSTUP: Matice A, B, jejichž prvky jsou celá čísla.

VÝSTUP: Matice $A \cdot B$.

Poznámka: V algoritmu se nemusíte zabývat načítáním vstupu a výpisem výstupu.

Nepředpokládejte, že matice A a B musí být čtvercové, můžete však předpokládat, že jejich rozměry jsou takové, aby se obě matice daly vynásobit, tj. že velikost matice A je $m \times n$ a velikost matice B je $n \times p$, kde m, n a p jsou nějaká přirozená čísla.

Řešení: Algoritmus 3 je standardní jednoduchý algoritmus pro násobení matic. Vstup je představován maticemi A (velikosti $m \times n$) a B (velikosti $n \times p$), výstup je pak zapsán do matice C (velikosti $m \times p$).

Je zřejmé, že nejvíce času se při vykonávání tohoto algoritmu stráví prováděním těla nejvnitřnějšího cyklu **for** a že toto tělo se provede celkem $m \cdot n \cdot p$ krát.

Časová složitost tohoto algoritmu je tedy $\Theta(m \cdot n \cdot p)$.

Z hlediska paměťových nároků tohoto algoritmu je nejvíce paměti potřeba pro uložení matic A, B a C, paměť pro ostatní proměnné je vůči tomu zanedbatelná. Paměťová složitost algoritmu

Algoritmus 3: Násobení matic

```
MATRIX-MULT(A, B, C, m, n, p):
    for i := 1 to m do
        for j := 1 to p do
            x := 0
            for k := 1 to n do
                x := x + A[i][k] * B[k][j]
            C[i][j] := x
```

je pak dána celkovým počtem paměťových buněk nutných pro uložení těchto tří matic, tedy $\Theta(m \cdot n + n \cdot p + m \cdot p)$.

Příklad 7: Navrhněte (nějaký) algoritmus řešící následující problém:

VSTUP: Číslo n a sekvence čísel $a_1, a_2, \ldots, a_n$, kde pro všechna $i = 1, 2, \ldots, n$ platí $a_i \in \{1, 2, \ldots, n\}$.

OTÁZKA: Je v sekvenci $a_1, a_2, \ldots, a_n$ každé $x \in \{1, 2, \ldots, n\}$ obsaženo právě jednou?

Analyzujte časovou složitost vašeho algoritmu. Pokud je větší než $O(n)$, zkuste navrhnout algoritmus s časovou složitostí $O(n)$.

Řešení: Algoritmus 4 je příkladem algoritmu s časovou i paměťovou složitostí $\Theta(n)$, který řeší tento problém.

Hlavní myšlenkou, na které je tento algoritmus založen, je, že si v n-prvkovém poli A pamatuje, které hodnoty v intervalu 1 až n již byly načteny. Pokud je na vstupu nějaké číslo, které není z intervalu 1 až n, nebo nějaké číslo, které již bylo dříve načteno, je jasné, že příslušená podmínka není splněna a algoritmus může vrátit FALSE.

Pokud algoritmus načetl n čísel, přičemž všechna tato čísla byla z intervalu 1 až n a žádné číslo se neopakovalo, je také jasné, že žádné z čísel z tohoto intervalu nemohlo chybět. Každé číslo z intervalu 1 až n se tedy v tomto případě muselo vyskytnout na vstupu právě jednou, a je tedy v pořádku, že v tomto případě vrátí algoritmus výsledek TRUE.

Algoritmus 4:

```
ALG ():
    read n
    for i := 1 to n do
        A[i] := 0
    for i := 1 to n do
        read x
        if x < 1 or x > n then
            return FALSE
        else if A[x] ≠ 0 then
            return FALSE
        A[x] := 1
    return TRUE
```