Cvičení 13

Příklad 1: Podrobně zdůvodněte, proč pro funkce

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

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 $g \in \Theta(f)$.)

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}	$\sqrt[3]{n}$	$n \log_2 n$

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.

a) $f_1(n) = 3n^2 + 5n - 1$, $f_2(n) = 2n^3 - 15n - 183$, $f_3(n) = (n+1)(n-1)$
b) $f_1(n) = 4n^2 + n^2 \log_2 n$, $f_2(n) = \log_2^5 n$, $f_3(n) = 17n + 3$
c) $f_1(n) = n\sqrt[5]{n}$, $f_2(n) = n$, $f_3(n) = \sqrt{n}$
d) $f_1(n) = 2^n$, $f_2(n) = n^{1024}$, $f_3(n) = n!$
e) $f_1(n) = 2^n$, $f_2(n) = n^n$, $f_3(n) = n!$
f) $f_1(n) = 2^n$, $f_2(n) = n^n$, $f_3(n) = n^{\log_2 n}$
g) $f_1(n) = 10^n$, $f_2(n) = 2^n$, $f_3(n) = 2^{2^n}$
h) $f_1(n) = \log_{10}(n^2)$, $f_2(n) = \log_2 n$, $f_3(n) = \log_2(n^2)$
i) $f_1(n) = n + \sqrt{n} \cdot \log_2 n$, $f_2(n) = n \cdot \log_2 n$, $f_3(n) = \sqrt{n} \cdot \log_2^2 n$
j) $f_1(n) = 2^n$, $f_2(n) = 2^{\sqrt{n}}$, $f_3(n) = n!$
k) $f_1(n) = n/2048$, $f_2(n) = \sqrt{n} \cdot 3n$, $f_3(n) = n + n \cdot \log_2 n$
l) $f_1(n) = (\log_2 n)^n$, $f_2(n) = n^n$, $f_3(n) = 10^{\sqrt{n}}$

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.

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 1: Třídění přímým výběrem

```
1  SELECTION-SORT (A, n):
2  begin
3      i := n − 1
4      while i > 0 do
5          k = 0
6          for j := 1 to i do
7              if A[k] < A[j] then
8                  k := j
9              end
10         end
11         x := A[k];  A[k] := A[i];  A[i] := x
12         i := i − 1
13     end
14 end
```

Algoritmus 2: Binární vyhledávání

```
1  BSEARCH (x, A, n):
2  begin
3      l := 0
4      r := n
5      while l < r do
6          k := ⌊(l + r) / 2⌋
7          if A[k] < x then
8              l := k + 1
9          else
10             r := k
11         end
12     end
13     if l < n and A[l] = x then
14         return l
15     end
16     return NOTFOUND
17 end
```

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.

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 $O(n)$.

Příklad 8: Někdy je v algoritmech potřeba pracovat s poli, u kterých předem nevíme, kolik prvků do nich bude potřeba během výpočtu uložit. V takovém případě může být výhodné použít druh pole, jehož velikost se během výpočtu může dynamicky měnit — tento datový typ se většinou označuje jako *vector*.

Typická implementace tohoto datového typu vypadá tak, že se alokuje o něco větší pole, než je momentálně potřeba, přičemž se kromě tohoto pole a jeho délky navíc udržuje informace o počtu prvků, které jsou zatím použity. Když je potřeba přidat další prvek nebo prvky, použijí se dosud nepoužité buňky a jen se zvětší příslušný index. Pouze v případě, kdy je pole zcela zaplněno a není v něm dostatek volných buněk, alokuje se nové větší pole, do kterého se obsah původního pole překopíruje.

Pro jednoduchost se zaměřme jen na operaci APPEND, která přidá jeden nový prvek na konec pole. Tato operace je popsána Algoritmem 3. Proměnná *arr* je alokované pole, proměnná *allocated* udává délku tohoto alokovaného pole a proměnná *len* pak počet buněk, které jsou skutečně použity. (Předpokládá se, že vždy platí invariant $allocated \geq len$.) Pro jednoduchost berme tyto tři proměnné (*arr*, *allocated*, *len*) jako globální. Všechny ostatní proměnné jsou lokální.

V proceduře APPEND je použito několik podprogramů:

- MALLOC(*size*) — alokuje pole o *size* prvcích (pro jednoduchost zde neřešme ošetření případu, kdy tato alokace selže),
- FREE(*arr*) — uvolňuje paměť použitou pro pole *arr*,
- COPY(*dst*, *src*, *cnt*) — kopíruje *cnt* prvků z pole *src* do pole *dst*

Algoritmus 3: Přidání prvku na konec vektoru

```
1  APPEND (x):
2  begin
3      if allocated ≤ len then
4          s := NEW-SIZE(allocated)
5          if s < len + 1 then
6              s := len + 1
7          end
8          newarr := MALLOC (s)
9          COPY (newarr, arr, len)
10         FREE (arr)
11         arr := newarr
12         allocated := s
13     end
14     arr[len] := x
15     len := len + 1
16 end
```

U těchto tří podprogramů počítejte s tím, že jejich časová složitost je $O(n)$, kde n je počet prvků daného pole, resp. počet prvků, které je třeba překopírovat (u podprogramu COPY).

Funkce NEW-SIZE určuje, jaká má být velikost nově alokovaného pole v závislosti na velikosti dosud alokovaného pole.

Uvažujme dvě možné implementace funkce NEW-SIZE:

a) funkce NEW-SIZE(m) vrací hodnotu $m + 1$,
b) funkce NEW-SIZE(m) vrací hodnotu $2 * m$.

Uvažujme nyní o algoritmu, který začne s prázdným polem a v cyklu do něj bude pomocí procedury APPEND postupně přidávat n prvků. (Řekněme například pro jednoduchost, že na začátku výpočtu mají proměnné *allocated* a *len* hodnoty 0 a pole *arr* obsahuje 0 prvků.) Jaká bude časová složitost daného algoritmu pro každou z výše uvedených variant funkce NEW-SIZE?

(Předpokládejte, že pokud nepočítáme čas strávený v proceduře APPEND, tak doba zpracování každého jednotlivého přidávaného prvku je $O(1)$.)

Příklad 9: Sekvence prvků může v paměti počítače reprezentována pomocí různých datových struktur. Příklady těchto datových struktur jsou například:

a) pole
b) jednosměrný seznam, kde máme ukazatel na první prvek seznamu
c) jednosměrný seznam, kde máme ukazatele na první a poslední prvek seznamu
d) obousměrný seznam, kde máme ukazatele na první a poslední prvek seznamu

Připomeňte si, jak tyto datové struktury vypadají a jak se s nimi pracuje.

Určete co nejpřesněji časovou složitost následujících operací na těchto datových strukturách (předpokládejte, že n udává celkový počet prvků v dané datové struktuře).

1. přečtení hodnoty prvku na pozici i, kde i může být libovolné číslo od 0 do $n-1$ (předpokládejte, že prvky jsou číslovány od 0),
2. přečtení hodnoty prvního prvku (tj. prvku na pozici 0),
3. přečtení hodnoty posledního prvku (tj. prvku na pozici $n-1$),
4. přidání prvku na začátek (a posunutí všech ostatních prvků o jednu pozici dále),
5. přidání prvku na konec,
6. přidání prvku před daný prvek (a posunutí všech ostatních prvků za ním o jednu pozici dále),
7. přidání prvku za daný prvek (a posunutí všech ostatních prvků za ním o jednu pozici dále),
8. odstranění zadaného prvku.

Poznámka: V bodech 6., 7. a 8. předpokládejte, že prvek, před/za který se přidává, nebo který se odstraňuje, je určen pomocí indexu v případě pole a pomocí ukazatele na tento prvek v případě seznamu.

Pro jednoduchost u polí předpokládejte, že zvětšení pole o jeden prvek je možné provést v čase $O(1)$.

Příklad 10: Řekněme, že máme dáno n prvků, které jsou uloženy v poli A, a chtěli bychom postupně provést nějakou operaci se všemi podmnožinami těchto n prvků.

Jednou možností, jak tyto podmnožiny generovat, je použít rekurzivní algoritmus. Příkladem takového algoritmu je Algoritmus 4.

Předpokládá se zde, že A a B jsou globální pole a n je globální proměnná obsahující jako hodnotu velikost obou těchto polí. Pole A obsahuje zadané prvky a jeho obsah se v průběhu výpočtu nemění. Do pole B algoritmus průběžně zapisuje jednotlivé podmnožiny, přičemž zpravování každé podmnožiny je provedeno podprogramem PROCESS, který jako parametr dostane číslo ℓ, které udává počet prvků v dané podmnožině, přičemž hodnoty těchto prvků jsou v dané chvíli zapsány v poli B jako prvky $B[0]$, $B[1]$, ..., $B[\ell-1]$. Proměnné k a ℓ, které představují parametry procedury SUBSETS, jsou v této proceduře lokální. Procedura SUBSETS se na začátku volá s nulovými hodnotami obou argumentů, tj. SUBSETS$(0,0)$.

Určete co nejpřesněji časovou a paměťovou složitost tohoto algoritmu. (Předpokládejte, že časová i paměťová složitost podprogramu PROCESS je v $O(n)$.)

Příklad 11: Uvažujme o následujících dvou variantách Euklidova algoritmu pro výpočet největšího společného dělitele dvou čísel popsaných Algoritmy 5 a 6.

Určete časovou složitost obou těchto algoritmů, přičemž jako velikost vstupu uvažujete celkový počet bitů čísel a a b. (Pro jednoduchost předpokládejte, že doba provedení každé jednotlivé aritmetické operace je $O(1)$.)

Algoritmus 4: Generování podmnožin

```
1  SUBSETS(k, ℓ):
2  begin
3      if k ≥ n then
4          PROCESS(ℓ)
5          return
6      end
7      SUBSETS(k + 1, ℓ)
8      B[ℓ] := A[k]
9      SUBSETS(k + 1, ℓ + 1)
10 end
```

Algoritmus 5: Euklidův algoritmus — neefektivní verze

```
1  EUCLID(a, b):
2  begin
3      if b = 0 then
4          return a
5      else if a ≥ b then
6          return EUCLID(b, a − b)
7      else
8          return EUCLID(b − a, a)
9      end
10 end
```

Algoritmus 6: Euklidův algoritmus — efektivnější varianta

```
1  EUCLID(a, b):
2  begin
3      while b ≠ 0 do
4          c := a mod b
5          a := b
6          b := c
7      end
8      return a
9  end
```