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

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

```
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
```

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

```
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
```

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 $(\ell+r)-2 < 2k \le \ell+r$. Protože $d' = r-k-1$, tak $k = r-d'-1$. Dosazením do předchozího vztahu dostáváme

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

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

b) Platí $A[k] \ge x$ 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 je pak dána celkovým počtem paměťových buněk nutných pro uložení těchto tří matic, tedy $\Theta(m \cdot n + n \cdot p + m \cdot p)$.

Algoritmus 3: Násobení matic

```
1  MATRIX-MULT (A, B, C, m, n, p):
2  begin
3      for i := 1 to m do
4          for j := 1 to p 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
```

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.

Příklad 8: Někdy je v algoritmech potřeba pracovat s poli, u kterých předem nevíme, kolik

Algoritmus 4:

```
1  Alg ():
2  begin
3      read n
4      for i := 1 to n do
5          A[i] := 0
6      end
7      for i := 1 to n do
8          read x
9          if x < 1 or x > n then
10             return False
11         else if A[x] ≠ 0 then
12             return False
13         end
14         A[x] := 1
15     end
16     return True
17 end
```

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 5. 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 $allocated \geq len$ always holds) 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*

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).

Algoritmus 5: 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
```

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 $O(1)$.)

Řešení:

a) Případ, kdy funkce NEW-SIZE(m) vrací hodnotu $m + 1$:

   V tomto případě se bude nové pole alokovat celkem n krát. V celkové době výpočtu bude dominovat čas, který se stráví těmito alokacemi a kopírováním ze staré kopie pole do nové.

   Postupně se při volání procedury COPY bude kopírovat 1 prvek, 2 prvky, atd., až nakonec $n - 1$ prvků. Celkový počet prvků, které se budou kopírovat, se dá tedy spočítat jako součet aritmetické řady

$$1 + 2 + \cdots + (n - 1) = \Theta(n^2)\,.$$

   Celková časová složitost je v tomto případě $\Theta(n^2)$.

b) funkce NEW-SIZE(m) vrací hodnotu $2 * m$:

K alokaci nového pole a kopírování obsahu starého pole bude docházet pouze v těch případech, kdy pole obsahuje $1, 2, 4, 8, \ldots$ prvků, tj. ve chvílích, kdy je počet prvků 2^i pro nějaké přirozené číslo i.

Pokud není aktuální počet prvků v poli mocninou dvojky, při přidání nového prvku bude v poli volné místo, takže se nový prvek rovnou zapíše do pole *arr* a hodnota proměnné *len* se zvětší o jedna. Celková doba přidání jednoho nového prvku je v těchto případech $\Theta(1)$. Celkový čas, který se stráví zpracováním těchto případů je tedy $O(n)$.

Zbývá tedy určit, kolik času se stráví alokací a kopírováním v případech, kdy je aktuální velikost pole mocninou dvojky. Je zřejmé, že tyto případy nastanou při postupném přidávání n prvků celkem k krát, kde $k = \lceil \log_2 n \rceil$. Počty prvků, které se v těchto případech kopírují, tvoří geometrickou posloupnost

$$2^0, 2^1, \ldots 2^{k-1}.$$

Pro jednoduchost předpokládejme, že $n = 2^k$. Součet této geometrické posloupnosti se spočte následujícím způsobem:

$$1 + 2 + 4 + 8 + \cdots + 2^{k-1} = \sum_{j=0}^{k-1} 2^j = \frac{2^k - 1}{2 - 1} = 2^{\log_2 n} - 1 = n - 1 = \Theta(n)$$

Pokud není n mocninou dvojky, bude tento součet o něco větší, ale je zřejmé, že nebude větší o víc než n. I tomto případě se kopíruje maximálně $2n - 1$ prvků a tedy počet kopírovaných prvků je vždy $\Theta(n)$.

Celkový čas, který se stráví všemi ostatními operacemi je rovněž $\Theta(n)$. Celková časová složitost je tedy $\Theta(n)$.

Příklad 9: Sekvence prvků může být 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 $O(1)$.

Řešení: Časová složitost jednotlivých operací je shrnuta v následující tabulce (jedná se o složitost v nejhorším případě) — sloupce odpovídají jednotlivým datovým strukturám a řádky jednotlivým operacím:

Operace	(a)	(b)	(c)	(d)
1.	$\Theta(1)$	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$
2.	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$
3.	$\Theta(1)$	$\Theta(n)$	$\Theta(1)$	$\Theta(1)$
4.	$\Theta(n)$	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$
5.	$\Theta(1)$	$\Theta(n)$	$\Theta(1)$	$\Theta(1)$
6.	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$	$\Theta(1)$
7.	$\Theta(n)$	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$
8.	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$	$\Theta(1)$

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 6.
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. Podprogram Process 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 $B[0], B[1], \ldots, B[\ell-1]$. 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 $O(n)$.)

Řešení: Představme si strom zachycující strukturu jednotlivých rekurzivních volání procedury Subsets. Jedná se o úplný binární strom výšky n. Listy odpovídají těm voláním, kde $k = n$ a kde se volá procedura Process. Těchto listů je celkem 2^n. Vrcholy stromu, které nejsou listy, odpovídají těm voláním, kdy se procedura Subsets dále dvakrát rekurzivně volá. Celkový

Algoritmus 6: 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
```

počet vrcholů stromu se dá spočítat jako součet následující geometrické posloupnosti (můžeme si představit, že počítáme počet vrcholů v každé jednotlivé „vrstvě“ stromu):

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

Čas strávený v proceduře SUBSETS při každém jednotlivém volání této procedury je $\Theta(1)$ (pokud nepočítáme čas, který se stráví v podprogramech, které jsou z ní volány). Celkový čas, který se stráví v proceduře SUBSETS je tedy $2^{n+1} \cdot \Theta(1) = \Theta(2^n)$.

Pokud předpokládáme, že čas strávený v jednom volání procedury PROCESS je $O(n)$, celkový čas strávený prováděním procedury PROCESS je

$$2^n \cdot O(n) = O(2^n \cdot n) = O(2^n \cdot 2^{\log_2 n}) = O(2^{n+\log_2 n}) = 2^{O(n)}$$

Celková doba výpočtu je pak součtem dob strávených v procedurách SUBSETS a PROCESS.

Nejprve si všimněme, že celková doba výpočtu je v $2^{O(n)}$. (Čas strávený v proceduře SUBSETS je v $\Theta(2^n)$, takže je i v $2^{O(n)}$, a jak jsme viděli, čas strávený v proceduře PROCESS je v $2^{O(n)}$.)

Na druhou stranu, čas strávený v proceduře SUBSETS je v $\Theta(2^n)$, takže celková doba výpočtu je určitě v $\Omega(2^n)$ a tedy i v $2^{\Omega(n)}$.

Z toho vidíme, že časová složitost celého algoritmu je $2^{\Theta(n)}$.

Co se týká paměťové složitosti, hloubka rekurzivních volání je n a množství paměti potřebné pro uložení lokálních proměnných procedury SUBSETS při jednom volání je $\Theta(1)$. Celkově je tedy množství paměti, která je potřeba pro proceduru SUBSETS, v $\Theta(n)$. Když k tomu připočteme $O(n)$ paměti pro proceduru PROCESS a $\Theta(n)$ paměti pro globální pole A a B, dostáváme celkovou paměťovou složitost $\Theta(n)$.

Příklad 11: Uvažujme o následujících dvou variantách Euklidova algoritmu pro výpočet největšího společného dělitele dvou čísel popsaných Algoritmy 7 a 8.

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 7: 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 8: 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
```

Řešení:

a) Algoritmus 7: S každým rekurzivním voláním se jedno z čísel a, b zmenší aspoň o 1. Nemůže tedy nastat více než $2 \cdot 2^k = 2^{k+1}$ rekurzivních volání. Každé volání trvá konstantní čas. Na druhou stranu si lze snadno představit zadání, při kterém bude 2^k iterací skutečně nutných: $a = 1$, $b = 2^k$. Celkem tedy je složitost algoritmu $\Theta(2^k)$.

b) Algoritmus 8:

Pokud $a \geq b$, tak číslo (zbytek) $c = a \bmod b$ je vždy méně než polovinou hodnoty b. Proto se v každé iteraci našeho algoritmu (možná mimo první) jedno z čísel a, b zmenší na méně než polovinu, neboli z jeho binárního zápisu ubude aspoň jedna číslice. Pokud na začátku měly a a b jen ℓ bitů, algoritmus musí skončit po méně než 2ℓ iteracích. Každá tato iterace trvá konstantní čas, takže celkem máme horní odhad $O(\ell)$, což je mnohem lepší než v předchozím případě.

Pro dolní odhad bychom měli najít dvojici čísel a, b, pro které trvá běh algoritmu co nejdéle. (Sice můžeme zjednodušeně říci, že potřebujeme aspoň přečíst ℓ bitů vstupu, ale to není úplně dostačující k rigoróznímu argumentu, neboť v zadání zanedbáváme délku zápisu vzhledem k aritmetickým operacím.) Není to nyní zase tak jednoduché, ale po pár pokusech asi přijdete na to, že nejlepší je volit dva po sobě jdoucí členy Fibonaciho posloupnosti ($a_0 = a_1 = 1$, $a_{n+1} = a_n + a_{n-1}$). Například $a = 13$ a $b = 8$ dá 6 iterací cyklu. Celkem v tomto případě počet iterací vyjde $\Theta(\ell)$, takže to je i nejhorší složitost našeho algoritmu. (Všimněte si, že ve Fibbonaciho posloupnosti platí pro všechna n, že $a_{n+1} \leq 2a_n$, takže n-té Fibonaciho číslo má určitě méně než n bitů. Pokud bychom měli n-bitové Fibonaciho číslo, provede se minimálně n iterací cyklu.)