Cvičení 10

Příklad 1: 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 1. 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.

Algoritmus 1: Přidání prvku na konec vektoru

```
APPEND(x):
    if allocated ≤ len then
        s := NEW-SIZE(allocated)
        if s < len + 1 then
            s := len + 1
        newarr := MALLOC(s)
        COPY(newarr, arr, len)
        FREE(arr)
        arr := newarr
        allocated := s
    arr[len] := x
    len := len + 1
```

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).

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 2: 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 3: 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 2.

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 $B[0]$, $B[1]$, ..., $B[\ell-1]$. 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$(0, 0)$.

Algoritmus 2: Generování podmnožin

```
SUBSETS(k, ℓ):
    if k ≥ n then
        PROCESS(ℓ)
        return
    SUBSETS(k + 1, ℓ)
    B[ℓ] := A[k]
    SUBSETS(k + 1, ℓ + 1)
```

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 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 4: Uvažujme o následujících dvou variantách Euklidova algoritmu pro výpočet největšího společného dělitele dvou čísel popsaných Algoritmy 3 a 4.

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 3: Euklidův algoritmus — neefektivní verze

```
EUCLID (a, b):
    if b = 0 then
        return a
    else if a ≥ b then
        return EUCLID(b, a − b)
    else
        return EUCLID(b − a, a)
```

Algoritmus 4: Euklidův algoritmus — efektivnější varianta

```
EUCLID (a, b):
    while b ≠ 0 do
        c := a mod b
        a := b
        b := c
    return a
```

Řešení:

a) Algoritmus 3: 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 4:

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 rigorózním 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.)

Příklad 5: Určete co nejpřesněji časové složitosti následujících podprogramů. Výsledky vyjádřete v asymptotické notaci pomocí Θ.

Poznámka: Jako velikost vstupu uvažujte hodnotu n. Můžete předpokládat, že hodnoty všech proměnných jsou již načteny v paměti.

a) Algoritmus 5 — podprogram Proc-A

Algoritmus 5:

```
Proc-A (A, b, n):
    for i := 1 to n * n do
        for j := 1 to i do
            A[i][j] := A[i][j] + b[j]
```

Řešení: Vnější cyklus se provede n^2-krát. Počet iterací vnitřního cyklu však závisí na proměnné i vnějšího cyklu. Celkový počet průchodů vnitřním cyklem je

$$\sum_{i=1}^{n^2} i = 1 + 2 + 3 + \cdots + (n^2 - 1) + n^2 = (1 + n^2)\frac{n^2}{2} = \frac{n^4}{2} + \frac{n^2}{2} = \Theta(n^4)$$

Časová složitost tohoto podprogramu je tedy $\Theta(n^4)$.

b) Algoritmus 6 — podprogram Proc-B

Řešení:

$$\sum_{i=1}^{n} i^2 = \Theta(n^3)$$

Algoritmus 6:

```
PROC-B (R, d, n):
    x := 0
    for i := 1 to n do
        j := i * i
        while j > 0 do
            if d[j] < R[i][j] then
                R[i][j] := x − 1
                x := d[j]
            j := j − 1
```

Podrobnější zdůvodnění:

$$\sum_{i=1}^{n} i^2 \leq \sum_{i=1}^{n} n^2 = n \cdot n^2 = n^3$$

$$\sum_{i=1}^{n} i^2 \geq \sum_{i=\lceil \frac{n}{2} \rceil}^{n} i^2 \geq \sum_{i=\lceil \frac{n}{2} \rceil}^{n} \left(\frac{n}{2}\right)^2 \geq \frac{n}{2} \cdot \left(\frac{n}{2}\right)^2 = \frac{1}{8} n^3$$

c) Algoritmus 7 — podprogram PROC-C

Algoritmus 7:

```
PROC-C (Q, n):
    i := 1
    while i < n do
        Q[i] := Q[i] + i
        i := i + i
```

Řešení: $\Theta(\log n)$

d) Algoritmus 8 — podprogram PROC-D

Algoritmus 8:

```
PROC-D (E, S, n):
    i := 1;  j := 1
    while i < n do
        E[i][j] := E[i][j] mod S[i]
        i := i + j
        j := j + 1
```

Řešení: Potřebujeme zjistit pročet průchodu cyklem. Nutně to bude takové největší číslo k, pro které platí $1 + 2 + \cdots + k \leq n$. Protože

$$1 + 2 + \cdots + k = (1 + k) \cdot \frac{k}{2} \approx \frac{1}{2}k^2$$

Z rovnosti $\frac{1}{2}k^2 \approx n$ dostáváme, že hodnota k je přibližně rovna $\sqrt{2n}$, tj. že počet průchodů cyklem (a celková složitost) je $\Theta(\sqrt{n})$.

Poznámka: Alternativně můžeme najít největší k takové, že $(1 + k) \cdot \frac{k}{2} \leq n$, jako jedno z řešení kvadratické rovnice

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

Řešení této kvadratické rovnice vypadá takto

$$k = \frac{-\frac{1}{2} \pm \sqrt{\left(\frac{1}{2}\right)^2 + 4 \cdot \frac{1}{2} \cdot n}}{2 \cdot \frac{1}{2}} = -\frac{1}{2} \pm \sqrt{\frac{1}{4} + 2n}$$

Vzhledem k tomu, že hledáme maximální hodnotu k, která je nezáporná a která je celým číslem, řešením bude hodnota $-\frac{1}{2} + \sqrt{\frac{1}{4} + 2n}$ zaokrouhlená na nejbližší menší celé číslo, tj. přibližně $\sqrt{2n}$.

e) Algoritmus 9 — podprogram Proc-E

Algoritmus 9:

```
Proc-E (A, B, n):
    s := 1
    while s ≤ n do
        i := 0
        while i < n do
            A[i] := A[B[i]] * s
            i := i + s
        s := s * 2
```

Řešení: Při průchodech vnějším cyklem nabývá proměnná s postupně hodnot $1, 2, 4, 8 \ldots$ Počet průchodů vnějším cyklem je tedy zhruba $\lg n$ (pozn.: $\lg n = \log_2 n$). Při každém průchodu vnějším cyklem se vnitřní cyklus provede zhruba n/s krát. Celkový počet průchodů vnitřním cyklem je tedy zhruba

$$\frac{n}{1} + \frac{n}{2} + \frac{n}{4} + \frac{n}{8} + \cdots + \frac{n}{n} \approx 1 + 2 + 4 + 8 + \cdots + 2^{\lg n} = \sum_{j=1}^{\lg n} 2^j =$$

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

To nám dává horní odhad časové složitosti $O(n)$. Na druhou stranu, při prvním průchodu vnějším cyklem se vnitřní cyklus provede n krát, což nám dává spodní odhad. Časová složitost tohoto podprogramu je tedy $\Theta(n)$.

f) Algoritmus 10 — podprogram Proc-F

Algoritmus 10:

```
Proc-F (A, n):
    s := 1
    while s ≤ n do
        i := 0
        while i < n do
            A[i] := A[i] + s
            i := i + s
        s := s + 1
```

Řešení: Proměnná s nabývá při průchodech vnějším cyklem postupně hodnot $1, 2, \ldots, n$, přičemž podobně jako v předchozím případě se při každém průchodu vnějším cyklem vnitřní cyklus provede zhruba n/s krát. Celková počet průchodů vnitřním cyklem (a celková složitost podprogramu) je tedy

$$\sum_{k=1}^{n} \frac{n}{k} = n \cdot \sum_{k=1}^{n} \frac{1}{k} = n \cdot \Theta(\log n) = \Theta(n \log n)$$

Poznámka: Využije se vztah pro součet harmonické řady

$$\ln(n+1) \leq \sum_{k=1}^{n} \frac{1}{k} \leq \ln n + 1\,.$$

Příklad 6: Připomeňte si pro každý z následujících problémů, co je jeho vstupem a jaká je otázka. Poté pro každý z těchto problémů navrhněte nějaký algoritmus, který ho řeší. Jaká je výpočetní složitost vámi navržených algoritmů?

a) SAT
b) 3-SAT
c) Problém nezávislé množiny (IS)
d) Problém kliky (CLIQUE)
e) Problém vrcholového pokrytí (VC)
f) Problém Hamiltonovského cyklu (HC)
g) Problém Hamiltonovské kružnice (HK)
h) Problém obchodního cestujícího (TSP)
i) Problém obarvení (vrcholů) grafu k barvami
j) SUBSET-SUM