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 $O(1)$.)

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 $O(1)$.

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 $O(n)$.)

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)
```

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

Algoritmus 4: Euklidův algoritmus — efektivnější varianta

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

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]
```

b) Algoritmus 6 — podprogram PROC-B

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
```

c) Algoritmus 7 — podprogram PROC-C

d) Algoritmus 8 — podprogram PROC-D

e) Algoritmus 9 — podprogram PROC-E

f) Algoritmus 10 — podprogram PROC-F

Algoritmus 7:

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

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
```

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
```

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
```

Příklad 6: Připoměnte 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