Algorytmy i struktury danych, rozdział I, część II – omówienie zadań

Dariusz Rzońca
Politechnika Rzeszowska
drzonca@kia.prz.edu.pl

1 Zadanie 4

Dany jest następujący algorytm:

```
MAX
1 czytaj x
2 max←x
3 dopóki x≠0
4 wykonuj { jeśli x>max
5           to max←x
6           czytaj x }
7 pisz max
```

a) Napisać program w języku maszyny RAM realizujący ten algorytm.

- Rozwiązanie z komentarzem:

Istnieje wiele metod tworzenia programów w języku maszyny RAM. Jedną z nich jest translacja programu w pseudojęzyku do kodu w języku maszyny RAM. Często jest to prostsze niż napisanie programu bezpośrednio w języku maszyny RAM, wyłącznie na podstawie specyfikacji. Postępowanie takie zapewnia także zgodność programu RAM z pseudojęzykiem. Tworząc program według tej metody działamy analogicznie jak hipotetyczny kompilator pseudojęzyka tłumaczący program do zbliżonego do asemblera języka maszyny RAM.

W pierwszym kroku należy każdej zmiennej z pseudojęzyka przypisać rejestr maszyny RAM. W naszym programie występują zmienne `x` i `max`. Przyjmijmy, że odpowiednikiem zmiennej `x` w maszynie RAM będzie rejestr r_1, a odpowiednikiem zmiennej `max` będzie rejestr r_2.

W kolejnym kroku tłumaczymy program w pseudojęzyku linia po linii. Pojedyncza linijka pseudojęzyka może być przetłumaczona na jedną bądź kilka komend maszyny RAM. Pierwsza linia naszego programu to `czytaj x`. Skoro przyjęliśmy, że odpowiednikiem zmiennej `x` będzie rejestr r_1 linię tę przetłumaczymy jako:

```
READ 1
```

Kolejna linia programu w pseudojęzyku zawiera przypisanie `max←x`. Odpowiednikiem w maszynie RAM będzie przypisanie do rejestru r_2 zawartości rejestru r_1. Nie ma pojedynczej instrukcji maszyny RAM realizującej takie przypisanie, należy tu posłużyć się sumatorem w celu przechowania pośredniej wartości. Finalnie linię tę przetłumaczymy jako sekwencję poleceń:

```
LOAD 1
STORE 2
```

W linii 3 programu w pseudojęzyku rozpoczyna się pętla. Typowo w językach wysokiego poziomu występują specjalizowane operatory pętli. W językach niskiego poziomu często musimy pętle realizować za pomocą instrukcji skoków i skoków warunkowych. Ważne jest by zapamiętać, że postępujemy tak wyłącznie wtedy, gdy wymusza to specyfika niskopoziomowego języka programowania. W językach wysokiego poziomu instrukcje skoków zazwyczaj nie występują, a nawet jeśli język na to pozwala, to nie należy ich używać. Stosowanie skoków narusza paradygmat programowania strukturalnego lub obiektowego, a także prowadzi do tworzenia bardzo nieczytelnego kodu. Jest to zdecydowanie zła praktyka.

Przypominam także, że podczas naszych zajęć obowiązuje nas składnia i semantyka pseudojęzyka zdefiniowana w skrypcie. W dialekcie tym występują operatory pętli, nie występują instrukcje skoków. Użycie instrukcji skoku do „ręcznego" zaimplementowania pętli w pseudokodzie, zamiast stosownego operatora pętli, będzie uznawane za błędne rozwiązanie danego zadania.

W języku maszyny RAM niestety nie mamy innej możliwości, musimy pętlę zaimplementować bazując na skokach. Przed każdym obiegiem pętli `dopóki` sprawdzany jest warunek pozostania w pętli. Jeżeli jest on spełniony to następuje jednokrotna iteracja instrukcji (prostej lub złożonej) następującej po słowie kluczowym `wykonuj`, a następnie powrót do sprawdzenia warunku. W języku maszyny RAM wygodniej jest przed każdą iteracją sprawdzić warunek przeciwny, a więc warunek opuszczenia pętli. Jeżeli będzie on spełniony to wykonany zostanie skok warunkowy poza pętlę, do kolejnej instrukcji (u nas odpowiednika linii 7 z pseudojęzyka). W przeciwnym wypadku wykonana zostanie pojedyncza iteracja pętli, zakończona skokiem (powrotem) do poprzedniego fragmentu programu sprawdzającego warunek.

Warunkiem przeciwnym do `x≠0` jest `x=0`. Skok warunkowy umożliwiający porównanie z wartością zero to `JZERO`. Aby go użyć musimy pobrać do sumatora wartość z rejestru r_1 (odpowiednika zmiennej `x`). Otrzymujemy więc dalszy fragment programu, będący tłumaczeniem linii 3 pseudojęzyka:

```
dopoki: LOAD 1
        JZERO koniec_dopoki
```

Uważny czytelnik zapewne dostrzeże, że po wykonaniu wcześniejszego fragmentu w sumatorze pozostała bieżąca wartość `x`, a więc komenda `LOAD 1` wydaje się tu nadmiarowa. Analiza dalszego fragmentu, w szczególności odpowiednika linii 6, uzasadni jej obecność.

W linii 4 pseudojęzyka występuje sprawdzenie warunku `x>max`. W języku maszyny RAM są dwa skoki warunkowe, `JZERO` i `JGTZ`, testujące czy wartość w sumatorze jest odpowiednio równa zero, bądź większa od zera. Należy więc odpowiednio przekształcić podane wyrażenie. Nierówność $x > max$ jest tożsama z nierównością $x - max > 0$. Obliczamy wyrażenie $x - max$ pamiętając, że odpowiednikiem `x` jest r_1, a odpowiednikiem `max` r_2. Otrzymujemy więc:

```
LOAD 1
SUB 2
JGTZ nowe_max
JUMP czytaj_x
```

Podobnie jak poprzednio fragment ten zaczyna się od załadowania wartości z rejestru r_1, która już jest w sumatorze. Tym razem faktycznie można tę instrukcję pominąć. Należy jednak uważać podczas wprowadzania podobnych optymalizacji, a w razie wątpliwości pozostawić instrukcję. Pamiętajmy, że pozostawienie w programie nadmiarowej, zbędnej instrukcji (a nawet dowolnej skończonej liczby takich instrukcji) nie zwiększy asymptotycznej złożoności programu.

Przypisanie w linii 5 pseudojęzyka jest analogiczne do przypisania z linii 2. Podobnie jak poprzednio otrzymujemy:

```
nowe_max: LOAD 1
          STORE 2
```

Wczytanie kolejnej wartości `x` (linia 6) realizujemy podobnie jak wczytywanie w linii 1. Otrzymujemy więc:

```
czytaj_x: READ 1
```

Zamykający nawias klamrowy w linii 6 kończy instrukcję złożoną, cyklicznie powtarzaną w pętli. Należy wrócić do sprawdzenia warunku:

```
JUMP dopoki
```

Linia 7 pseudojęzyka znajduje się poza pętlą. W linii tej wypisywana jest wartość zmiennej `max`. W języku maszyny RAM wypiszemy zawartość rejestru r_2.

```
koniec_dopoki: WRITE 2
```

Finalnie program maszyny RAM kończymy instrukcją:

```
HALT
```

Łącząc przedstawione uprzednio fragmenty otrzymujemy kompletny program:

```
        READ 1
        LOAD 1
        STORE 2
```

```
       dopoki: LOAD 1
               JZERO koniec_dopoki
               LOAD 1
               SUB 2
               JGTZ nowe_max
               JUMP czytaj_x
    nowe_max: LOAD 1
               STORE 2
    czytaj_x: READ 1
               JUMP dopoki
koniec_dopoki: WRITE 2
               HALT
```

b) Określić złożoność czasową i pamięciową programu RAM, przyjmując zarówno równomierne, jak i logarytmiczne kryterium wagowe.

- Rozwiązanie z komentarzem:

Złożoność obliczeniowa jest funkcją rozmiaru zadania, należy więc najpierw zastanowić się, co będzie tym rozmiarem w danym przypadku. Program, który analizujemy wczytuje n liczb (do momentu wczytania wartości zero), a następnie wypisuje największą z wczytanych liczb. Rozmiarem zadania będzie więc liczba wczytanych liczb.

Szacując złożoność pamięciową w równomiernym kryterium wagowym zakładamy, że każdy z rejestrów wykorzystuje jedną jednostkę pamięci. Złożoność pamięciowa w tym kryterium nie zależy więc od wartości przechowywanych w rejestrach, a jedynie od liczby rejestrów użytych w programie. Nie ma znaczenia czy w rejestrach przechowywane są duże czy też małe liczby, ważne tylko z ilu rejestrów program korzysta.

W programie z poprzedniego podpunktu korzystamy z trzech rejestrów: r_0, r_1 i r_2. Jest to stała liczba rejestrów, nie zależy od rozmiaru zadania, a więc złożoność pamięciowa według kryterium równomiernego wynosi $\mathcal{O}(1)$.

Pomocnicze, inżynierskie spostrzeżenie jest takie, że o ile w programie RAM nie użyto adresacji pośredniej $(*i)$ podając rejestr w którym jest numer innego rejestru z którego korzystamy, to wszystkie numery użytych rejestrów są jawnie podane w programie. Skoro tak, to jest ich ograniczona liczba, niezależna od rozmiaru zadania, a złożoność wynosi $\mathcal{O}(1)$. Jedynie przy wykorzystaniu adresacji pośredniej możliwe jest napisanie programu maszyny RAM w którym liczba użytych rejestrów będzie zależeć od rozmiaru zadania.

Szacując złożoność czasową w równomiernym kryterium wagowym zakładamy, że czas potrzebny na wykonanie każdej z instrukcji maszyny RAM jest jednakowy. Co za tym idzie rozważamy jak zmienia się liczba wykonywanych instrukcji wraz ze wzrostem rozmiaru zadania. Zauważmy, że analizowany program składa się z pewnych operacji początkowych (przed pętlą), pewnych operacji końcowych (za pętlą) i operacji powtarzanych w pętli. Operacje początkowe i końcowe wykonywane są tylko raz, nie rzutują więc na złożoność czasową. Nale-

ży skupić się na operacjach powtarzanych w pętli. Można przyjąć, że pojedyncza iteracja pętli trwa pewną liczbę jednostek czasu. Pętla wykona się raz dla każdej wczytanej liczby, poza ostatnią liczbą równą zero, a więc przy n liczbach wykona się $n-1$ razy. Wraz ze wzrostem rozmiaru zadania n, dla dużych wartości n, liniowo rośnie liczba obiegów pętli, a więc w przybliżeniu liniowo rośnie czas potrzebny na wykonanie programu. Liniową złożoność zapisujemy jako $\mathcal{O}(n)$.

Przyjmując logarytmiczne kryterium wagowe zakładamy, że duże liczby zajmują więcej miejsca w pamięci, a operacje wykonywane na nich trwają dłużej. Wzrost ten ma w przybliżeniu charakter zbliżony do funkcji logarytmicznej o podstawie 2. Związek ten ma proste, intuicyjne wytłumaczenie. Architektura współczesnych komputerów bazuje na systemie binarnym. Oznacza to, że na zapisanie dwukrotnie większej liczby potrzebujemy tylko o jeden bit więcej. Oczywiście jest to duże przybliżenie, w praktyce nie możemy przydzielić do zmiennej dowolnej liczby bitów, występuje granulacja związana z rozmiarem słowa maszynowego i rozmiarami typów danych w docelowym języku programowania.

Szacując złożoność pamięciową w kryterium logarytmicznym należy obliczyć po wszystkich użytych rejestrach sumę $\sum_i l(x_i)$ wartości funkcji l (zdefiniowanej w skrypcie) dla największych wartości bezwzględnych, które mogą pojawić się w poszczególnych rejestrach. Jak wspomniano poprzednio w rozpatrywanym przykładzie korzystamy z trzech rejestrów: r_0, r_1 i r_2. Największa wartość jaka pojawi się w rejestrze r_2 to największa z wczytanych liczb, oznaczmy ją jako max. Ta sama wartość pojawi się w rejestrze r_1 podczas wczytywania z taśmy. W trakcie obliczeń zostanie pobrana także do sumatora r_0. Uzyskujemy więc sumę $l(max) + l(max) + l(max) = 3l(max)$.

W kolejnym kroku należy zastanowić się jak ta największa z wczytanych liczb (max) zależy od rozmiaru zadania (n), a więc liczby wczytanych liczb. Łatwo zauważyć, że nie zależy. Może być tak, że wczytujemy miliony liczb, ale ciągle niewielkich (np. na przemian 1 lub 2), może być i tak, że wśród zaledwie kilku wczytanych liczb są liczby bardzo duże. Skoro wartość max nie zależy od n, to możemy traktować ją jako pewną stałą względem n. Skoro $max = const.$ to także $l(max) = const.$, a więc obliczana suma jest stała. Oznacza to, że złożoność pamięciowa w kryterium logarytmicznym również jest stała, co zapisujemy $\mathcal{O}(1)$.

Szacując złożoność czasową w kryterium logarytmicznym w pierwszej kolejności należy znaleźć operację dominującą, rzutującą na charakter złożoności całego programu. Będzie to operacja która wykonuje się najczęściej (jest wewnątrz pętli) i za każdym razem trwa możliwie najdłużej (operuje na największych liczbach). W naszym przypadku za operację dominującą przyjmiemy instrukcję `SUB 2`. Czas potrzebny na jednokrotne wykonanie tej instrukcji, zgodnie z wagą logarytmiczną podaną w skrypcie, wynosi $l(c(0)) + l(2) + l(c(2))$. Instrukcja wykona się $n-1$ razy (raz dla każdej wczytanej liczby). Przyjmując przypadek pesymi-

styczny, gdzie za każdym razem wczytujemy wartość max otrzymujemy:

$$\begin{aligned}&\sum_{i=1}^{n-1}(l(c(0))+l(2)+l(c(2)))=\\&=\sum_{i=1}^{n-1}(l(max)+l(2)+l(max))=\\&=(n-1)(2l(max)+l(2))\end{aligned}\tag{1}$$

Zauważając, że podobnie jak poprzednio $max = const.$ i zaniedbując stałe otrzymujemy złożoność $\mathcal{O}(n)$.

Finalnie otrzymaliśmy następujące złożoności:

	Złożoność pamięciowa	Złożoność czasowa
Kryterium równomierne	$\mathcal{O}(1)$	$\mathcal{O}(n)$
Kryterium logarytmiczne	$\mathcal{O}(1)$	$\mathcal{O}(n)$

Zauważmy, że uzyskane złożoności w kryterium logarytmicznym są jednakowe jak w kryterium równomiernym. Jest to rzadka zbieżność, zazwyczaj jest inaczej. Zawdzięczamy to temu, że w analizowanym programie największa wartość zmiennej nie zależy od rozmiaru zadania.

2 Zadanie 5

Zapisać w pseudojęzyku algorytm, którego zadaniem jest sprawdzenie, czy na wejściu pojawiła się jednakowa liczba jedynek i dwójek. Po przeczytaniu 0, algorytm wypisuje 1, jeśli liczba jedynek i dwójek była jednakowa i zatrzymuje się. Zakładamy, że oprócz 0, 1 i 2 na wejściu nie mogą pojawić się żadne inne symbole.

- Przykładowe rozwiązanie z komentarzem:

```
JEDYNKI_I_DWOJKI
1 licznik_1←0
2 licznik_2←0
3 czytaj x
4 dopóki x≠0
5 wykonuj { jeśli x=1
6           to licznik_1←licznik_1+1
7           inaczej licznik_2←licznik_2+1
8           czytaj x }
9 jeśli licznik_1=licznik_2 to pisz "1"
```

Koncepcja przedstawionego przykładowego rozwiązania bazuje na dwóch licznikach, zliczających osobno jedynki i dwójki. W każdej iteracji identyfikujemy wczytaną liczbę, zwiększamy odpowiedni licznik i wczytujemy kolejną liczbę.

Postępujemy tak do wczytania symbolu 0. Na końcu porównujemy oba liczniki – jeśli mają jednakowe wartości to znaczy, że wczytano tyle samo jedynek co i dwójek.

Inna możliwość konstrukcji programu rozwiązującego to zadanie wykorzystuje zaledwie jeden licznik, w którym podczas pracy, na bieżąco, zliczamy o ile więcej dotychczas wczytano jedynek niż dwójek. Każdorazowo po wczytaniu jedynki inkrementujemy licznik, po wczytaniu dwójki – dekrementujemy. Wartość dodatnia oznacza, że jedynek było więcej, ujemna, że dwójek było więcej, równa zeru, że liczba jedynek i dwójek była jednakowa. Dla tak przygotowanego programu szczególnie interesująca jest analiza złożoności obliczeniowej, zarówno pesymistycznej jak też oczekiwanej. Napisanie takiego programu i przeprowadzenie stosownej analizy pozostawiam dociekliwemu czytelnikowi.

a) Napisać program w języku maszyny RAM realizujący ten algorytm.

- Rozwiązanie:

Przyjmijmy r_1 – `licznik_1`, r_2 – `licznik_2`, r_3 – `x`. Tłumacząc program w pseudojęzyku na język maszyny RAM podobnie jak poprzednio otrzymujemy:

```
                READ 3
        dopoki: LOAD 3
                JZERO koniec_dopoki
                SUB =1
                JZERO jedynka
                LOAD 2
                ADD =1
                STORE 2
                JUMP czytaj_x
       jedynka: LOAD 1
                ADD =1
                STORE 1
     czytaj_x: READ 3
                JUMP dopoki
koniec_dopoki: LOAD 1
                SUB 2
                JZERO pisz_1
                HALT
        pisz_1: WRITE =1
                HALT
```

b) Określić złożoność czasową i pamięciową programu RAM.

- Rozwiązanie z komentarzem:

Rozumując podobnie jak w poprzednim zadaniu łatwo określić złożoności w równomiernym kryterium wagowym. Wynoszą one odpowiednio: złożoność

pamięciowa – $\mathcal{O}(1)$, złożoność czasowa – $\mathcal{O}(n)$. Złożoności w kryterium logarytmicznym wymagają dłuższej analizy.

Przyjmijmy przypadek pesymistyczny w którym wczytujemy same jedynki. Wczytamy $n-1$ jedynek i finalnie zero. Ta liczba jedynek znajdzie się podczas działania programu w rejestrze r_1 (liczniku jedynek) i w sumatorze. W pozostałych rejestrach będą niewielkie stałe. Otrzymujemy więc:

$$\begin{aligned}\sum_i l(x_i) &= l(c(0)) + l(c(1)) + l(c(2)) + l(c(3)) = \\ &= l(n-1) + l(n-1) + l(0) + l(1) = \\ &= 2l(n-1) + l(0) + l(1)\end{aligned} \tag{1}$$

Pomijając stałe i przybliżając funkcję l funkcją logarytmiczną otrzymujemy złożoność pamięciową w kryterium logarytmicznym równą $\mathcal{O}(\lg n)$.

Szacując złożoność czasową w kryterium logarytmicznym przyjmijmy ten sam przypadek pesymistyczny co poprzednio. Wówczas zawsze wykona się skok do etykiety `jedynka:`. Jako operację dominującą przyjmijmy instrukcję `ADD =1` z kolejnej linii. Zauważmy, że nie możemy jako operację dominującą przyjąć żadnej z poprzednich, występujących przed skokiem, instrukcji, gdyż działają one na małych liczbach. Dopiero w fragmencie programu inkrementującym licznik jedynek pracujemy na dużych liczbach, rosnących wraz ze wzrostem rozmiaru zadania. Można więc jako operację dominującą przyjąć dowolną z następujących po sobie `LOAD 1`, `ADD =1`, `STORE 1`.

Czas potrzebny na wykonanie wybranej operacji w $n-1$ obiegach pętli wynosi:

$$\sum_{i=1}^{n-1} \left(l(c(0)) + l(1)\right) \tag{2}$$

W i-tym obiegu pętli w sumatorze mamy wartość $i-1$, a więc otrzymujemy:

$$\sum_{i=1}^{n-1} \left(l(c(0)) + l(1)\right) = \sum_{i=1}^{n-1} \left(l(i-1) + l(1)\right) \tag{3}$$

Nie możemy, w przeciwieństwie do poprzedniego zadania, zastąpić sumy mnożeniem, gdyż wyrażenie pod sumą zmienia się w każdym kroku sumy (składnik zależy od i). Możemy jednak, korzystając z faktu, że funkcja logarytmiczna jest monotonicznie rosnąca, a więc największą wartość ma dla największego argumentu, ograniczyć wyrażenie pod sumą z góry, przyjmując w każdym kroku $i = n-1$.

$$\sum_{i=1}^{n-1} \left(l(i-1) + l(1)\right) \leqslant \sum_{i=1}^{n-1} \left(l(n-1-1) + l(1)\right) \tag{4}$$

Wynikową sumę możemy zastąpić mnożeniem.

$$\sum_{i=1}^{n-1} \left(l(n-1-1) + l(1)\right) = (n-1)\left(l(n-2) + l(1)\right) \tag{5}$$

Finalnie otrzymujemy złożoność czasową w kryterium logarytmicznym równą $\mathcal{O}(n \lg n)$. Wynikowa tabela złożoności:

	Złożoność pamięciowa	Złożoność czasowa
Kryterium równomierne	$\mathcal{O}(1)$	$\mathcal{O}(n)$
Kryterium logarytmiczne	$\mathcal{O}(\lg n)$	$\mathcal{O}(n \lg n)$