Rozwiązania kilku zadań z WDI

Antoni Kościelski

18 listopada 2015

1 Lista 3 zadanie 1 g)

Rozwiązując to zadanie należy sprawdzić, czy $\log n^n = O(\log n!)$ oraz czy $\log n! = O(\log n^n)$.

W związku z tym musimy ustalić związki między $n!$ i n^n. Najpierw zauważmy, że mamy

Fakt 1.1 Dla każdej dodatniej liczby naturalnej n zachodzi nierówność

$$n! \leqslant n^n.$$

Dowód. Jest to raczej oczywiste, iloczyn n liczb nie przekraczających n jest mniejszy bądź równy iloczynowi n liczb równych n, czyli n^n. Dowód indukcyjny tego faktu też nie powinien sprawiać kłopotów. □

Przedstawione rozumowanie można trochę wzmocnić i w ten sposób dowieść

Fakt 1.2 Dla każdej liczby naturalnej $n \geqslant 2$ mamy

$$n! \leqslant 2n^{n-2}, \text{ a także } n^2 \cdot n! \leqslant 2 \cdot n^n. \ \square$$

Tak można nawet dowieść

Fakt 1.3 Dla każdej liczby naturalnej $n \geqslant k$ mamy

$$n! \leqslant k! \cdot n^{n-k}, \text{ a także } n^k \cdot n! \leqslant k! \cdot n^n. \ \square$$

Fakt 1.1 właściwie stwierdza, że

$$n! = O(n^n).$$

Powyższa zależność oznacza, że dla pewnej liczby $c > 0$ i dla wszystkich dostatecznie dużych liczb naturalnych zachodzi nierówność

$$n! \leqslant c \cdot n^n.$$

Zgodnie z faktem 1.1, nierówność ta zachodzi dla stałej $c = 1$ i dla wszystkich dodatnich liczb naturalnych n. Tak więc $n!$ jest O-duże od n^n.

Nierówność z faktu 1.1 możemy zlogarytmować stronami. Wtedy będzie ona świadczyć o tym, że

$$\log n! = O(\log n^n).$$

Przy okazji zauważmy, że fakt 1.2 implikuje, że funkcja n^n nie jest O-duże od $n!$. Gdyby – przeciwnie – n^n było O-duże od $n!$, to dla pewnej stałej $c > 0$ i dla wszystkich dostatecznie dużych liczb naturalnych n mielibyśmy

$$n^2 \cdot n! \leqslant n^n \leqslant c \cdot n!.$$

Dzieląc wtedy skrajne nierówności stronami przez $n!$ otrzymalibyśmy, że nierówność

$$n^2 \leqslant c$$

zachodzi wszystkich dostatecznie dużych liczb, co oczywiście nie jest możliwe.

Teraz zajmiemy się drugą z interesujących nas relacji. Pokażemy, że jednak $\log n^n = O(\log n!)$, mimo że n^n nie jest O-duże od $n!$ i mimo tego, że relacje

$$n! = O(n^n) \text{ oraz } \log n! = O(\log n^n)$$

mają właściwie ten sam dowód. Najpierw musimy pokazać

Lemat 1.4 *Dla wszystkich liczb naturalnych $n \geqslant 1$ mamy*

$$n^n \leqslant (n!)^2.$$

Dowód. Lemat ten wynika z nierówności

$$n \leqslant x(n+1-x)$$

słusznej dla wszystkich liczb $x \in [1, n]$. O jej słuszności można przekonać się badając wielomian $x(n+1-x)$ lub w następujący sposób: $1 \leqslant n$ mnożymy przez (nieujemną!) liczbę $x-1$ i otrzymaną nierówność

$$x - 1 \leqslant n(x-1)$$

przekształcamy do wymaganej postaci

$$n \leqslant x(n+1-x).$$

Dalej otrzymane tak nierówności dla $x = 1, 2, \ldots n$ wystarczy pomnożyć przez siebie stronami:

$$n^n = \prod_{x=1}^{n} n \leqslant \prod_{x=1}^{n} x(n+1-x) = \left(\prod_{x=1}^{n} x\right) \cdot \prod_{x=1}^{n} (n+1-x) = (n!)^2$$

(w pierwszym iloczynie liczby od 1 do n są w zwykłej kolejności, w drugim – w odwrotnej). □

Jeżeli nierówność z lematu 1.4 zlogarytmujemy stronami, to otrzymamy nierówność, która świadczy o tym, że $\log n^n = O(\log n!)$.

2 Zadanie 2 z listy 3

W tym zadaniu należało oszacować złożoność czasową oraz pamięciową podanego na wykładzie algorytmu znajdującego przedstawienie dwójkowe danej liczby naturalnej.

Z różnych powodów (na przykład po to, by nie odbierać osobom zainteresowanym możliwości samodzielnego przeanalizowania oryginalnego zadania), będziemy zajmować się podobnym, następującym algorytmem lub programem

```
i ← 0; {*1*}
dopóki a > 1 wykonuj {*2*}
   x_i ← a mod 2;
   a ← a div 2;
   i ← i+1 {*3*}
{*4*}
x_i ← a; {*5*}
dopóki i ⩾ 0 wykonuj
   write(x_i);
   i ← i-1
```

Algorytm z wykładu działa podobnie, ale w szczególny sposób znajduje „najmłodszą" cyfrę przedstawienia, a później wszystkie pozostałe. Podany program, na końcu w specjalny sposób ustala cyfrą „najstarszą" po uprzednim wyliczeniu w pętli wszystkich pozostałych cyfr.

Ponadto w przytoczonym programie zostało zaznaczone kilka miejsc przez umieszczenie w nich odpowiednich liczb (jako komentarzy). Jeżeli wykonanie programu będziemy sobie wyobrażać jak wędrówkę po jego instrukcjach (podobnie jak wędrówkę po schemacie blokowym), to po wykonaniu pierwszego przypisania, a przed rozpoczęciem wykonywania pierwszej instrukcji *dopóki* znajdziemy się w miejscu oznaczonym liczbą 1. W miejscu z 2 znajdziemy się bezpośrednio po wykonaniu z wynikiem pozytywnym testu z instrukcji *dopóki*, tuż przed rozpoczęciem wykonywania podstawień z pętli. Miejsce 3 to miejsce tuż po wykonaniu podstawień z pętli i przed kolejnym wykonaniem testu. Natychmiast po wykonaniu pierwszej instrukcji *dopóki* znajdziemy się w miejscu oznaczonym cyfrą 4.

Po zapoznaniu się z programem zauważmy jeszcze, że jeżeli nie ma potrzeby przechowywania w pamięci wyliczonego przedstawienia, to cztery ostatnie linijki można w nim zastąpić następującymi

```
write(a);
dopóki i > 0 wykonuj
   i ← i-1;
   write(x_i)
```

Przed szczegółową analizą programu wprowadzimy jeszcze potrzebne oznaczenia. Symbole a, i oraz x_i są używane w naszym programie jako zmienne. Będą używane także na oznaczenie wartości tych zmiennych, ustalonych w określonym, wynikającym z kontekstu momencie. W każdym wzorze te symbole powinny oznaczać wartości odpowiednich zmiennych określone w tym samym momencie. Będziemy też używać symbolu A oznaczającego początkową wartość zmiennej a, określoną (np. wczytaną za pomocą pominiętej instrukcji *Read*) na początku programu i przechowywaną w zmiennej a w momencie 1, a także symbolu I oznaczającego wartość zmiennej i w momencie oznaczonym cyfrą 4, a więc bezpośrednio po wykonaniu w całości pierwszej instrukcji *dopóki*.

Najpierw pokażemy (prezentując przy okazji pewną technikę), że jeżeli uruchomimy program z daną $A \geqslant 1$, to gdy znajdziemy się w miejscu 4 też będzie zachodzić nierówność $a \geqslant 1$.

Jeżeli algorytm został uruchomiony z daną $\geqslant 1$, to gdy rozpocznie się wykonywanie pierwszej instrukcji *dopóki* zmienna a będzie mieć wartość $\geqslant 1$. Po wykonaniu testu, czy $a > 1$, albo od razu znajdziemy się w punkcie 4 z wartością $a \geqslant 1$ (wykonanie testu nie zmienia wartości a), albo też znajdziemy się w punkcie 2. W tym drugim przypadku, wynik testu musiał być pozytywny, a więc mamy $a > 1$, albo inaczej: $a \geqslant 2$. Dla takich a mamy a div $2 \geqslant 1$. Następnie zostaną wykonane trzy instrukcje, w tym instrukcja, która przypisze zmiennej a nową wartość a div $2 \geqslant 1$. W dalszym ciągu wartość zmiennej a będzie więc $\geqslant 1$ i w miejscu 3 dalej będzie zachodzić nierówność $a \geqslant 1$.

Własność taka, jak $a \geqslant 1$, która zachodzi w miejscu 3 zawsze wtedy, gdy zachodziła w miejscu 2 (przynajniej po pozytywnym wyniku testu), nazywa się niezmiennikiem (odpowiedniej) instrukcji *dopóki*. Instrukcja *dopóki* ma tę własność, że jej następnik jest prawdziwy tuż przed jej wykonaniem, pozostaje prawdziwy właściwie stale podczas jej wykonywania i zachodzi także po zakończeniu jej wykonywania. Oznacza to, że po uruchomieniu analizowanego programu z daną $A \geqslant 1$ w miejscu 4 zachodzi nierówność $a \geqslant 1$.

Instrukcja *dopóki* ma jeszcze jedną własność. Instrukcja ta jest wykonywana tak długo, dopóki testowany w niej warunek okazuje się prawdziwy. Jeżeli przestaje być wykonywana, to znaczy, że jej warunek przestał być prawdziwy. Wobec tego, po zakończeniu wykonywania pierwszej instrukcji *dopóki* z rozważanego programu, a więc w miejscu 4, nie jest spełniony warunek $a < 1$, czyli zachodzi także nierówność $a \leqslant 1$.

Podsumowując, z dotychczasowych rozważań wynika, nasz program po uruchomieniu z $A \geqslant 1$ powoduje, że po wykonaniu dwóch pierwszych instrukcji, w miejscu 4, zmienna a ma wartość 1. Można też łatwo zobaczyć, że w tym samym momencie pracy, po uruchomieniu z daną $A = 0$ zmienna a ma wartość 0. Tak się złożyło (być może przypadkiem, na razie nie widać istotnych powodów), że w miejscu 4 wartość zmiennej a jest także wartością pierwszej („najstarszej") cyfry z przedstawienia dwójkowego danej A.

Pierwsza instrukcja *dopóki* ma jeszcze jeden niezmiennik. Jest nim równość

$$A = a \cdot 2^i + \sum_{j=0}^{i-1} x_j \cdot 2^j. \tag{1}$$

Aby przekonać się, że faktycznie jest to niezmiennik, nadajmy jej nieco inną postać

$$A = a \cdot 2^i + \sum_{j=0}^{i-1} x_j \cdot 2^j = ((\underbrace{a \text{ div } 2}_{a}) \cdot 2 + (\underbrace{a \text{ mod } 2}_{x_{(i+1)-1}})) \cdot 2^{\overbrace{i+1}^{i} - 1} + \sum_{j=0}^{(i+1)-2} x_j \cdot 2^j.$$

Jeżeli teraz uwzględnimy zmiany wartości zmiennych dokonywane podczas wykonywania podstawień z pętli *dopóki*, między punktami 2 i 3, to otrzymamy

$$A = (a \cdot 2 + x_{i-1}) \cdot 2^{i-1} + \sum_{j=0}^{i-2} x_j \cdot 2^j = a \cdot 2^i + \sum_{j=0}^{i-1} x_j \cdot 2^j.$$

Nietrudno zauważyć, że równość (1) zachodzi przed wykonaniem pierwszej instrukcji *dopóki* (wtedy $i = 0$, sumujemy zero składników, taka suma jest równa 0). Ponieważ jest to niezmiennik, więc również zachodzi po wykonaniu instrukcji *dopóki*, w miejscu 4. Wtedy w miejscu 5 mamy (oprócz równości $i = I$)

$$A = a \cdot 2^i + \sum_{j=0}^{i-1} x_j \cdot 2^j = x_i \cdot 2^i + \sum_{j=0}^{i-1} x_j \cdot 2^j = \sum_{j=0}^{i} x_j \cdot 2^j = \sum_{j=0}^{I} x_j \cdot 2^j.$$

Otrzymaliśmy więc pewien związek między między liczbami A, I oraz wartościami $x_I, x_{I-1}, \ldots, x_0$. Wynika z niego, że wartości $x_I, x_{I-1}, \ldots, x_0$ są kolejnymi cyframi przedstawienia dwójkowego liczby A, które w dodatku ma $I + 1$ cyfr. Z wcześnieszych rozważań wynika także, że jest przedstawienie najkrótsze z możliwych. W ten sposób w pierwszym rzędzie uzasadniliśmy poprawność rozważanego algorytmu, a także znane twierdzenie matematyczne o istnieniu przedstawień dwójkowych wszystkich liczb naturalnych.

Jeżeli już mamy ten wzór, to możemy z niego wyprowadzić związki między liczbą i długością jej przedstawienia dwójkowego. Dla liczb $A > 0$ mamy oczywiście $x_I = 1$ oraz

$$2^I \leqslant A = \sum_{j=0}^{I} x_j \cdot 2^j \leqslant \sum_{j=0}^{I} 2^j = 2^{I+1} - 1 < 2^{I+1}.$$

Logarytmując tę nierówność stronami dostajemy

$$I \leqslant \log_2 A < I + 1.$$

Stąd otrzymujemy związek między liczbą i długością jej standardowego przedstawienia dwójkowego: długość przedstawienia dwójkowego dodatniej liczby naturalnej A (oznaczana często symbolem $| A |_2$) jest częścią całkowitą $\log_2 A$ powiększoną o 1, czyli

$$| A |_2 = \text{długość przedstawienia dwójkowego } A = I + 1 = \lfloor \log_2 A \rfloor + 1.$$

Najwyższy czas zająć się złożonością naszego programu, a to wymaga innego spojrzenia na wzór podany nieco wyżej. Można go także odczytać jako stwierdzenie, że po uruchomieniu naszego programu z daną $A > 0$, w chwili wykonania go do miejsca 4, zmienna i ma wartość $I = \lfloor \log_2 A \rfloor$.

Znając I łatwo można wyliczyć wartości dwóch funkcji $T(A)$ oraz $M(A)$ takich, że

$$T(A) = \text{liczba przypisań i testów wykonanych po uruchomieniu programu dla danej } A,$$

$$M(A) = \text{liczba zmiennych wykorzystywanych po uruchomieniu programu dla danej } A.$$

Łatwiej wyliczyć drugą z tych funkcji. Oczywiście, podczas pracy algorytmu korzystamy i definiujemy wartości zmiennych a, i oraz $x_0, x_1, \ldots, x_I$ i pamiętamy maksymalnie $I + 2$ liczby. Stąd

$$M(A) = I + 3 = \lfloor \log_2 A \rfloor + 3$$

dla $A > 0$ i $M(0) = 3$. Aby wyliczyć $T(A)$ należy zauważyć, że zmienna i w programie odgraywa rolę licznika pętli właściwie obu instrukcji *dopóki*. Pętla w pierwszej z tych instrukcji jest więc wykonywana I, a cała ta instrukcji wymaga wykonania $4 \cdot I + 1$ przypisań i testów, w tym testu kończącego. Podobnie, druga instrukcja *dopóki* wymaga wykonaia $3 \cdot I + 4$ instrukcji prostych. Uwzględniając dwa dodatkowe przypisania dostajemy

$$T(A) = 4 \cdot I + 1 + 3 \cdot I + 4 + 2 = 7 \cdot I + 7 = 7 \cdot \lfloor \log_2 A \rfloor + 7.$$

Teraz możemy zająć się złożonością programu. Wymaga to sprecyzowania zarówno sposobu rozumienia złożoności czasowej i pamięciowej, jak i rozmiaru danych. Pojęcie elementarnej czynności, zliczanej podczas ustalania złożoności czasowej, rozumianej wyżej jako test lub przypisanie wykonane podczas realizacji algorytmu, dobrze oddaje intuicje i może być uznane za określone w sposób standardowy. Oceniając złożoność czasową podobne rezultaty otrzymalibyśmy rysując schemat blokowy i zliczając wykonywane czynności, nieco inne, choć podobne, po sformalizowaniu algorytmu na maszynie RAM i zliczając wykonywane rozkazy. Podobne uwagi możnaby odnieść do złożoności pamięciowej. Dalej zajmiemy się więc rozmiarem danych.

Sporadycznie, głownie w rozważaniach bardzo abstrakcyjnych i mało praktycznych, przyjmuje się, że rozmiar liczby naturalnej A jest równy A. Takie rozumienie rozmiaru ma pewną zaletę: dla danych ustalonego rozmiaru działanie algorytmu jest ściśle określone. Nie jest jednak zgodne z pewnymi intuicjami, na przykład przeświadczeniem, że algorytmy o złożoności liniowej powinny być w rzeczywistości wykonywane w krótkim czasie, bez specjalnego oczekiwania na wynik obliczeń. Dla takiego rozumienia rozmiaru danych bez trudu określamy (dla $n > 0$) złożoność czasowę t_1 i pamięciową m_1 naszego algorytmu:

$$t_1(n) = T(n) = 7 \cdot \lfloor \log_2 n \rfloor + 7 \quad \text{oraz} \quad m_1(n) = M(n) = \lfloor \log_2 n \rfloor + 3.$$

Lepiej przyjąć, że rozmiarem liczby naturalnej jest długość jej zwykłego przedstawienia dwójkowego. W tym przypadku też łatwo znaleźć złożoność czasową t_2 i pamięciową naszego programu. Mamy bowiem

$$t_2(n) = \max\{T(A) \mid A \text{ jest rozmiaru } n\} = \max\{7 \cdot \lfloor \log_2 A \rfloor + 7 \mid n = \lfloor \log_2 A \rfloor + 1\} = 7 \cdot n$$

oraz

$$m_2(n) = \max\{M(A) \mid A \text{ jest rozmiaru } n\} = \max\{\lfloor \log_2 A \rfloor + 3 \mid n = \lfloor \log_2 A \rfloor + 1\} = n + 2.$$

Tym razem udało się uzyskać eleganckie wzory, ponieważ zarówno rozmiar danych, jak i opis działania algorytmu zależały od logarytmu dwójkowego wprowadzonej liczby.

Najbardziej naturalne wydaje się uznanie za rozmiar liczby naturalnej długości jej przedstawienia dziesiętnego. Wtedy dla dodatnich A mamy

$$\text{rozmar liczby } A = |\, A \,|_{10} = \lfloor \log_{10} A \rfloor + 1$$

i musimy znaleźć związek między logarytmami dziesiętnym i dwójkowym. Zauważmy, że

$$\log_2 10 \cdot \lfloor \log_{10} A \rfloor \leqslant \log_2 10 \cdot \log_{10} A = \log_2 A < \log_2 10 \cdot (\lfloor \log_{10} A \rfloor + 1),$$

a więc

$$\log_2 10 \cdot \lfloor \log_{10} A \rfloor - 1 \leqslant \lfloor \log_2 A \rfloor < \log_2 10 \cdot (\lfloor \log_{10} A \rfloor + 1).$$

Nierówność ta wyraża też związek między długościami przedstawień dziesiętnego $|A|_{10}$ i dwójkowego $|\, A \,|_2$ liczby A, a mianowicie

$$\log_2 10 \cdot |\, A \,|_{10} - \log_2 10 \leqslant |\, A \,|_2 < \log_2 10 \cdot |\, A \,|_{10} + 1.$$

Podane oszacowania są dość dokładne, skrajne ograniczenia różnią się o $1+\log_2 10 \approx 4,322$. Długość przedstawienia dwójkowego oscyluje między podanymi ograniczeniami, bywa blisko zarówno dolnego, jaki górnego. Proszę sprawdzić (wystarczy tylko umieć operować logarytmami), że na przykład, dla $A = 2^{93}$ mamy

$$|\, A \,|_2 = 94,\ |\, A \,|_{10} = 28 \ \text{ oraz } \ \log_2 10 \cdot |\, A \,|_{10} + 1 \approx 94,01399,$$

a dla $A = 10^{31}$

$$|\, A \,|_2 = 103,\ |\, A \,|_{10} = 32 \ \text{ oraz } \ \log_2 10 \cdot |\, A \,|_{10} - \log_2 10 \approx 102,9798.$$

Przytoczone własności przedstawień świadczą o tym, że dla rozważanego teraz rozmiaru danych trudno wyrazić złożoności czasową t_3 i pamięciową m_3 ładnymi wzorami, ale z powyższych nierówności można wyprowadzić pewne oszacowania, na przykład

$$\begin{aligned} &7 \cdot \log_2 10 \cdot |\, A \,|_{10} - 7 \cdot \log_2 10 \leqslant \\ \leqslant\ t_3(n) = \max\{T(A) \mid |\, A \,|_{10} = n\} &= \max\{7 \cdot \lfloor \log_2 A \rfloor + 7 \mid n = \lfloor \log_{10} A \rfloor + 1\} < \\ &< 7 \cdot \log_2 10 \cdot n + 1. \end{aligned}$$

Tego typu oszacowania są w złożoności obliczeniowej zwykle wystarczające. Często bardzo trudno uzyskać coś więcej. Wynika z nich w szczególności, że $t_2(n) = O(t_3(n))$ oraz $t_3(n) = O(t_2(n))$, a także, że obie funkcje są O-duże od n. Podobne rezultaty można otrzymać dla złożoności pamięciowej. Można też spróbować dowieść równość

$$t_3(n) = t_2(\lfloor n \cdot \log_2 10 \rfloor + 1).$$

3 Zadanie 7 z listy 2

Tym razem należy podać schemat blokowy i program w kodzie RAM dla zadania opisanego następującą specyfikacją:

Wejście: liczba naturalna n i n elementowy ciąg liczb $x_1, x_2, \ldots, x_n$,

Wyjście: wczytany ciąg wypisany w odwrotnej kolejności $x_n, \ldots, x_2, x_1$.

Zamiast schematu blokowego podam algorytm rozwiązujący ten problem w formie „listy kroków" przypominającej program w prostym języku programowania. Zadanie to można rozwiązać w następujący sposób:

```
read(n);                (1)
i ← 0;                  (2)
dopóki i < n wykonuj    (3)
   read(x_i);           (4)
   i ← i+1              (5)
dopóki i > 0 wykonuj    (6)
   i ← i-1;             (7)
   write(x_i)           (8)
```

Podany program mona przetłumaczyć na program dla maszyny RAM wzorując się może na działaniu kompilatora. Przyjmujemy, że zmienna n odpowiada komórce o numerze (adresie) 1, zmienna i – komórce o numerze 2, zmienna x_0 – o numerze 3, zmienna x_1 – o numerze 2 itd. Właściwie będziemy posługiwać się tablicą x. Proste kompilatory mogą utożsamiać tablicę z adresem jej pierwszej komórki i wymagać numerowania jej elementów od 0. Wtedy łatwo znaleźć adres i-tego elementu tablicy: wystarczy adres tablicy powiększyć o indeks elementu.

```
   READ 1     (1)  wczytanie wartości zmiennej n
   LOAD =0    (2)  nadanie początkowej wartości zmiennej i
   STORE 2
e1:LOAD 1     (3)  testujemy, czy i < n obliczając n-i
   SUB 2
   JZERO e2        wyjście z pętli, jeżeli n - i = 0
   LOAD 2     (4)
   ADD =3
   READ ^0
   LOAD 2     (5)  zwiększamy i o 1
   ADD =1
   STORE 2
   JUMP e1         powrót na początek pętli
e2:LOAD 2     (6)  druga pętla, w akumulatorze indeks pierwszej 'wolnej' komórki
   JZERO e3        testujemy dodatniość i
   SUB =1     (7)  i to indeks ostatniej 'zapisanej' komórki
   STORE 2
   ADD =3     (8)  obliczamy adres ostatniej 'zapisanej' komórki
   WRITE ^0
   JZERO e2        powrót na początek drugiej pętli
e3:
```