Wybrane zadania z ćwiczeń 2015

Bartek Klin i Damian Niwiński

Uwaga. Ten tekst jest materiałem dydaktycznym opartym na wielu źródłach, których tu nie wymieniamy.

1 Obliczenia deterministyczne

1.1 Binarna reprezentacja par słów

Zadanie: Wymyślić efektywną reprezentację par ciągów binarnych jako ciągów binarnych.
Rozwiązanie: Propozycja wypracowana na ćwiczeniach

$$(u, v) \quad \mapsto \quad C(u)\, v, \tag{1}$$

gdzie

$$C(x_1 \dots x_n) \quad = \quad 0x_1 0x_2 \dots 0x_n 1. \tag{2}$$

Na przykład $(01, 1100) \mapsto 000111100$, $(\varepsilon, 000) \mapsto 1000$.

Uwaga: nie bardzo da się krócej. Przyjęcie $0x_1 0x_2 \dots 0x_{n-1} 1x_n$ zamiast funkcji C, nie działałoby dla $u = \varepsilon$.

Ale można spróbować upakować to bardziej. Niech $L(u)$ będzie binarną reprezentacją długości słowa u. Przypomnijmy, że długość binarnego zapisu liczby $n > 0$ jest[1] $\lfloor \log n \rfloor + 1$. Wtedy $|L(u)| = \lfloor \log_2 |u| \rfloor + 1$ dla u niepustych. Kolejne przybliżenia kodowania:

$$C(L(u))uv, \qquad C(L^2(u))L(u)uv, \qquad \dots$$

Kolejne iteracje przestają się opłacać jeśli $|L^i(u)| \leq 6$. Przerachujmy:

$$\begin{aligned} |u| &= 6 \\ |C(u)| &= 2 * 6 + 1 = 13 \\ |L(u)| &= \lfloor \log_2 |u| \rfloor = 3 \\ |C(L(u))u| &= 2 * 3 + 1 + 6 = 13 = |C(u)| \end{aligned}$$

W praktyce 3 iteracje wystarczą do reprezentowania ciągów dowolnej długości[2]. Możemy więc zastosować reprezentację z trzema iteracjami, przy czym można odpuścić sobie $C(-)$ i po prostu przeznaczyć pierwsze 6 bitów na pierwszy element reprezentacji:

$$L^3(u)L^2(u)L(u)uv$$

przy czym na $L^3(u)$ przeznaczamy 6 bitów. Gdybyśmy chcieli użyć dowolnej liczby iteracji, to na początku trzeba by zapisać liczbę iteracji, których użyliśmy.

[1] Dla liczby n o $k+1$ cyfrach, $n = a_k 2^k + a_{k-1} 2^{k-1} + \dots + a_1 2 + a_0$ (gdzie $a_k = 1$) mamy $2^k \leq n < 2^{k+1}$, zatem $k \leq \log n < k + 1$. Przyjmujemy tu oczywiście $\log_2 = \log$.

[2] Czytelnik może zechcieć porównać rząd wielkości $2^{2^{2^6}}$ z szacowaną liczbą atomów wszechświata $\approx 100^{39}$.

1.2 Krótka notacja dla $\mathbb{N}$

W standardowej reprezentacji liczb naturalnych, każda liczba n jest reprezentowana przez $\lfloor \log_2 n \rfloor + 1$ bitów. Może da się lepiej? Przecież w tej reprezentacji nie jest wykorzystywany żaden ciąg zaczynający się od 0...

Zadanie. Pokazać że dla każdej funkcji różnowartościowej $\alpha : \mathbb{N} \to \{0,1\}^*$ istnieje nieskończenie wiele $n \in \mathbb{N}$, takich że $|\alpha(n)| > \log_2 n$.

Poniższy dowód nie jest może najbardziej intuicyjny, ale wprowadza pewną pożyteczną ideę. Dla liczby naturalnej k, niech

$$n_k \quad = \quad \min\{n : |\alpha(n)| \geq k\}$$

Twierdzimy, że powyższe n_k spełniają pożądany warunek, tj. $|\alpha(n_k)| > \log_2 n_k$.

Istotnie, słów krótszych niż k (włącznie z pustym) jest

$$1 + 2 + 2^2 + \ldots + 2^{k-1} = 2^k - 1.$$

Z minimalności wynika, że słowo $\alpha(i)$ jest krótsze niż k, dla wszystkich i mniejszych niż n_k. Zatem (wobec różnowartościowości α) liczb mniejszych niż n_k jest nie więcej niż $2^k - 1$, czyli $n_k \leq 2^k - 1$. Jeśli $n_k > 0$, to mamy $\log_2 n_k < k$. Zarazem $|\alpha(n_k)| \geq k$, zatem

$$|\alpha(n_k)| \geq k > \log_2 n_k,$$

jak chcieliśmy. Jeśli $n_k = 0$, to przyjmujemy, że $\log_2 0 = -\infty$, więc nierówność jest trywialnie spełniona.

Uwaga. Intuicyjnie, liczby n_k są „trudne" dla notacji α, ponieważ wymagają stosunkowo „długiej" notacji. Ta idea leży u podstaw tzw. złożoności informacyjnej Kołmogorowa, gdzie funkcja α przyporządkowuje liczbie naturalnej najkrótszy program, który ją generuje (przy odpowiednio ścisłej definicji tych pojęć). Nierówność $|\alpha(n)| \geq \log n$ sugeruje, że nie ma istotnie prostszego programu niż $write(n)$, gdzie n jest dane binarnie. Dla porównania, liczba (10!)! ma w „sensownym" języku programowania program znacznie krótszy od siebie. Liczby spełniające wspomnianą nierówność Kołmogorow uważał za *losowe*.

1.3 Algorytm Euklidesa

Problem jest następujący.

Dane: liczby naturalne a i b w postaci binarnej.
Znaleźć: największy wspólny dzielnik NWD (a, b).

Interesuje nas czas rozwiązania tego problemu na maszynie Turinga. Naiwny algorytm wykonuje odejmowanie tak długo aż otrzyma 0; wynikiem jest ostatnia

wartość niezerowa. Czas działania jest tu proporcjonalny do $\max(a, b)$, a więc **wykładniczy** względem rozmiaru danych.

Szybsza wersja algorytmu Euklidesa wykorzystuje dzielenie *modulo*. Można opisać ją poniższym programem.

while $\min(a, b) > 0$ **do**
  if $b > a$ **then** $b := b \bmod a$
  else $a := a \bmod b$
Return $\max(a, b)$

Dowód poprawności pozostawiamy Czytelnikowi. Dla oszacowania złożoności zbadajmy najpierw, ile razy wykona się petla *while*. Bez zmniejszenia ogólności załóżmy, że na początku $a_1 \leq b_1$ i petla wykonała się co najmniej dwukrotnie. Wtedy kolejne wartości zmiennych a i b spełniają

$$\begin{array}{ccc} a_1 & \leq & b_1 \\ \| & & \\ a_2 & \geq & b_2 \\ & & \| \\ a_3 & \leq & b_3 \end{array}$$

Co więcej

$$b_1 = k \cdot a_1 + b_2 \leq k \cdot a_1 + a_1$$

(dla pewnego $k \geq 1$), a zatem $k \cdot a_1 \geq \frac{1}{2} b_1$, skąd

$$b_3 = b_2 \leq \frac{1}{2}\, b_1$$

czyli

$$\max(a_3, b_3) \leq \frac{1}{2} \max(a_1, b_1).$$

A zatem, po dwukrotnym wykonaniu pętli, $\max(a, b)$ zmniejsza się co najmniej o połowę. Stąd wniosek, że pętla może się wykonać co najwyżej $2 \cdot \log \max(a, b)$ razy.

Pozostaje oszacować czas obliczenia operacji $A \bmod B$.

Załóżmy, że

$$A = \alpha_k \cdot 2^k + \alpha_{k-1} \cdot 2^{k-1} + \ldots + \alpha_1 \cdot 2 + \alpha_0$$

a zatem A dane jest jako ciag bitów $\alpha_k \alpha_{k-1} \ldots \alpha_1 \alpha_0$.

Rozważmy następujący program, implementujacy szkolny algorytm dzielenia.

$c := 0$
For $i = k$ **downto** 0 **do**
$c := (2c + \alpha_i) \bmod B$
Return c

W programie tym nadal wykonujemy operację $d \bmod B$, dla argumentu $d = 2c+\alpha_i$, gdzie $i = k, k-1, \ldots, 1, 0$. Zauważmy jednak, że tym razem zachowujemy własność

$$d < 2 \cdot B$$

więc wynik może być obliczony bardzo łatwo:

if $d < B$ **then return** d **else return** $d - B$

Koszt porównania i odejmowania można łatwo oszacować przez $\mathcal{O}(\log B)$. A zatem koszt operacji $A \bmod B$ jest $\mathcal{O}(\log A \cdot \log B)$. Wszystko razem daje nam oszacowanie algorytmu Euklidesa

$$\mathcal{O}(\log \max(a, b) \cdot \log a \cdot \log b) = \mathcal{O}(n^3)$$

gdzie n jest rozmiarem danych.

Można podać jeszcze szybszą, nieco trikową, wersję algorytmu.

$d := 1$
while $\min(a, b) > 0$ **do**
  if $2|a \wedge 2|b$ **then** $a := \frac{a}{2};\ b := \frac{b}{2};\ d := 2 \cdot d$
  else
    if $2|a$ **then** $a := \frac{a}{2}$;
    if $2|b$ **then** $b := \frac{b}{2}$
    else
    if $b \geq a$ **then** $b := b - a$ **else** $a := a - b$

Dowód poprawności pozostawiamy Czytelnikowi, zajmiemy się złożonością. Oczywiście każda instrukcja w pętli wykonuje się w czasie $\max(\log a, \log b)$. Ponadto w każdym przypadku z wyjątkiem ostatniego przynajmniej jedna z liczb zmniejsza się o połowę. W ostatnim przypadku wykonujemy odejmowanie na liczbach nieparzystych, rezultat jest więc (znowu) liczbą parzystą. To daje oszacowanie na liczbę wykonać pętli *while*: $2 \cdot (\log a + \log b)$ i w rezultacie cały algorytm ma złożoność $\mathcal{O}(n^2)$, gdzie n jest rozmiarem danych.

1.4 Rozpoznawanie palindromów na jednej taśmie

Ten przykład pokaże, że dodanie drugiej taśmy w maszynie Turinga może kwadratowo przyśpieszyć obliczenie. Nietrudno jest zbudować maszynę Turinga z dwiema taśmami, która rozpoznaje palindromy w czasie liniowym. Pokażemy, że jakakolwiek maszyna Turinga M z jedną taśmą wykonuje co najmniej cn^2 kroków obliczenia na wejściu o długości n dla dostatecznie dużych n i dla pewnej stałej c (zależnej od tej maszyny).

Pokażemy oszacowanie dla palindromów postaci $w0^{|w|}w^R$, gdzie $w \in (0+1)^*$.

Użyteczne jest pojęcie **historii lokalnej**; po angielsku używa się też określenia *crossing sequence*. Dla danej komórki taśmy maszyny Turinga rejestrujemy ciąg w stanów, w których maszyna kolejno

– opuszcza tę komórkę idąc w prawo,

– powraca do tej komórki przychodząc z prawej strony.

Intuicyjnie, rejestrujemy, co się dzieje nad prawą ścianką komórki. Możemy założyć, że maszyna wykonuje jedynie ruchy w prawo lub w lewo (a nie w miejscu); ewentualne doprowadzenie jej do takiej postaci co najwyżej podwoi czas obliczenia.

Zauważmy, że sumując wysokości historii lokalnych, jakie powstają w obliczeniu maszyny dla danego słowa wejściowego, otrzymamy czas działania maszyny na tym wejściu. Kluczowa jest następująca obserwacja.

Rozważamy wejścia postaci $w0^n w^R$, gdzie $|w| = n$. Wybierzmy dowolną komórkę w „środkowej części", tj. o numerze $i \in [n+1, ..., 2n]$. Dwa **różne słowa** w i v muszą mieć w tym punkcie **różne historie** lokalne. Istotnie, w przeciwnym razie moglibyśmy złożyć ze sobą część obliczenia maszyny M na słowie $w0^n w^R$, która „dzieje się" na lewo od komórki i z częścią obliczenia maszyny na słowie $v0^n v^R$ na prawo od komórki i i uzyskać w ten sposób akceptujące obliczenie na słowie $w0^n v^R$, którego nie powinno być.

Dalej pozostaje rachunek. Zauważmy, że dla danej liczby k, liczbę historii lokalnych o wysokości mniejszej niż k można oszacować przez $1 + d + d^2 + \ldots + d^{k-1} < d^k$, dla pewnego $d \geq 2$. A zatem zbiór tych w, które w danej komórce i (w części środkowej) mają historię lokalną krótszą niż k, ma co najwyżej d^k elementów. Dla różnych i te zbiory mogą być oczywiście różne. Możemy jednak tak dobrać stałą c, że dla $k = c \cdot n$ nawet suma tych wszystkich zbiorów nie obejmie wszystkich słów długości n. Wystarczy, że

$$\begin{aligned} n \cdot d^{cn} &< 2^n, \text{ czyli} \\ \log_2 n + \log_2 d \cdot c \cdot n &< n. \end{aligned}$$

A zatem istnieje takie słowo $w \in (0+1)^n$, że w obliczeniu maszyny M na tym słowie wysokość lokalnej historii jest w **każdym** punkcie części środkowej co najmniej $c \cdot n$. Jeśli zsumujemy wysokości wszystkich historii lokalnych w części środkowej, otrzymamy, że czas obliczenia maszyny na tym słowie jest co najmniej $c \cdot n^2$.

Uwaga. Dla niektórych języków można uzyskać lepszy czas. Na przykład język $0^n 1^n$ można rozpoznać na jednotaśmowej maszynie Turinga w czasie $\mathcal{O}(n \log n)$.

Wsakzówka. Maszyna może inkrementować licznik binarny, znajdując w ten sposób liczbę zer, a następnie go dekrementować w celu porównania z liczbą jedynek. Licznik ten nie może jednak pozostawać w miejscu (bo koszt powrotów do niego byłby za duży), lecz powinien być systematycznie przesuwany w prawo. Na jedno przesunięcie wystarczy $\mathcal{O}(\log n)$ kroków.

1.5 Ewaluacja formuł w pamięci logarytmicznej

Jako rozgrzewkę zobaczmy jak w pamięci logarytmicznej (klasa **L**) można rozpoznać poprawne wyrażenia nawiasowe, czyli słowa generowane gramatyką $X \to \varepsilon \mid XX \mid (X)$. Wystarczy sukcesywnie obliczać różnicę liczby lewych i prawych

nawiasów, która na końcu powinnna osiągnąć zero, nigdy wcześniej nie spadając poniżej zera.

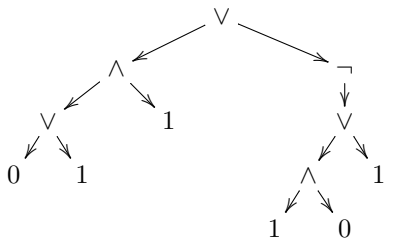
Teraz rozważamy formuły logiczne bez zmiennych z negacją oraz operatorami binarnymi $\vee$ i $\wedge$ zapisanymi infiksowo, z pełnym nawiasowaniem. Formalnie, formuły generowane są gramatyką

$$F \to 0 \,|\, 1 \,|\, (\neg F) \,|\, (F \vee F) \,|\, (F \wedge F)$$

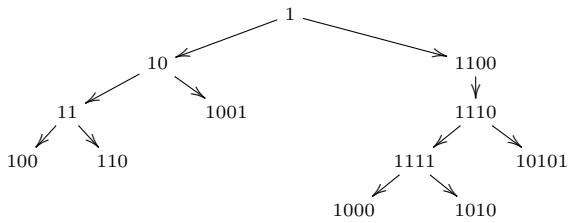
Istnieje naturalna reprezentacja formuł jako drzew. Na przykład formułę

$$(((0 \vee 1) \wedge 1) \vee (\neg\,((1 \wedge 0) \vee 1)))$$

można reprezentować jako drzewo



Węzły drzewa odpowiadają podformułom wyjściowej formuły. Z każdą taką podformułą wiążemy także jej **adres**, a mianowicie pozycję jej otwierającego nawiasu. Na przykład podformuła $(1 \wedge 0)$ „zaczyna się”[3] na pozycji 15; jej adresem (w postaci binarnej) jest więc 1111. W ten sposób mamy naturalną odpowiedniość między węzłami drzewa a liczbami binarnymi będącymi adresami podformuł. W naszym przykładzie wyglądałoby to następująco.



Nietrudno jest pokazać, że w pamięci logarytmicznej można „nawigować” pomiędzy węzłami drzewa, w szczególności

- sprawdzić czy aktualny węzeł jest lewym czy prawym synem, czy jest liściem lub korzeniem,
- znaleźć lewego syna, prawego syna, lub ojca aktualnego węzła,
- odczytać przy pomocy taśmy wejściowej etykietę węzła (stałą lub spójnik logiczny).

[3]Uwaga: różne wystąpienia tej samej formuły traktujemy jako różne podformuły.

Przy użyciu tych procedur, można obliczyć wartość formuły logicznej, przy czym poza tymi procedurami wystarczy stała pamięć do przechowywania wartości logicznej $\top$ lub $\bot$, przy czym początkowo wartość ta jest nieustalona. Chodzimy po drzewie podobnie jak w algorytmie *DFS*, pamiętając wartość logiczną i to, skąd przyszliśmy:

- jeśli przyszliśmy od ojca lub jesteśmy w korzeniu i nie mamy wartości, idziemy do lewego syna (w przypadku węzła $\neg$, do jedynego syna),
- ilekroć idziemy do syna, zapominamy wartość,
- jeśli jesteśmy w liściu, wartość logiczna przyjmuje wartość liścia i idziemy do ojca,
- jeśli przyszliśmy z lewego syna z wartością $\top$ i jesteśmy węźle $\wedge$, idziemy do prawego syna,
- jeśli tak samo ale z wartością $\bot$, to idziemy do ojca z wartością $\bot$,
- analogicznie dla węzłów $\vee$,
- jeśli przyszliśmy z prawego syna, to idziemy do ojca z tą samą wartością,
- jeśli przyszliśmy z jedynego syna (w przypadku węzła $\neg$) to zmieniamy wartość na przeciwną i idziemy do ojca,
- jeśli z powyższych instrukcji wynika, że mamy iść do ojca, ale jesteśmy w korzeniu, to kończymy obliczenie i aktualna wartość logiczna jest wartością formuły.

Niezmiennikiem powyższych działań jest, że jeśli wartość logiczna jest określona, to jest to wartość podformuły widocznej z aktualnie odwiedzanego węzła drzewa.

Aktualizowano 13.04.2015.