AISD lista 1

Wiktor Kuchta

0/4

---

procedure MULT(a, b)
    $sum \leftarrow 0$
    while $a > 0$ **do**
        if $a \bmod 2 = 1$ **then**
            $sum \leftarrow sum + b$
        end if
        $a \leftarrow a \div 2$
        $b \leftarrow 2 * b$
    end while
    return sum
end procedure

Wartość obliczaną przez algorytm mnożenia rosyjskich chłopów dla liczb a i b możemy zapisać

$$f(a, b) = \begin{cases} 0 & \text{jeśli } a = 0 \\ (a \bmod 2)b + f(\lfloor \frac{a}{2} \rfloor, 2b) & \text{jeśli } a \neq 0 \end{cases}$$

Tezę $f(a, b) = ab$ możemy udowodnić poprzez indukcję względem a. Przypadek bazowy $f(0, b) = 0 = 0b$ jest oczywiście prawdziwy. Jeśli teza jest prawdziwa dla lewych argumentów mniejszych od $a > 0$, to w szczególności zachodzi równość $f(\lfloor \frac{a}{2} \rfloor, 2b) = \lfloor \frac{a}{2} \rfloor 2b$. Zauważmy, że wtedy

$$\begin{aligned} ab &= \left(\frac{a}{2}\right) 2b = \left(\left\lfloor \frac{a}{2} \right\rfloor + \left\{\frac{a}{2}\right\}\right) 2b = \left\lfloor \frac{a}{2} \right\rfloor 2b + \left\{\frac{a}{2}\right\} 2b \\ &= f\left(\left\lfloor \frac{a}{2} \right\rfloor, 2b\right) + \left(\frac{a \bmod 2}{2}\right) 2b = f\left(\left\lfloor \frac{a}{2} \right\rfloor, 2b\right) + (a \bmod 2)b, \end{aligned}$$

co kończy dowód.

Przy jednorodnym kryterium kosztów, zakładamy, że operacje arytmetyczne i bitowe zajmują czas $O(1)$ i w każdej z $\log_2 a$ iteracji wykonujemy ich dwie lub trzy, więc złożoność czasowa to $O(\log_2 a)$. W pamięci przechowujemy tylko dwie liczby, więc złożoność pamięciowa to $O(1)$.

Zauważmy, że w każdej iteracji liczba bitów a i liczba bitów b sumują się do $\log_2 ab$, więc przy logarytmicznym kryterium kosztów algorytm ma złożoność pamięciową $\Theta(\log_2 ab)$.

W każdej iteracji wykonujemy operacje przesunięcia bitowego na a i b, więc sumaryczny koszt tych operacji to zawsze $\Theta(\log_2 ab)$. Wewnątrz instrukcji warunkowej możemy jeszcze dodawać liczbę b. Ta liczba jest zawsze przesunięciem bitowym w lewo początkowej wartości b, więc koszt jej dodania to $\Theta(\log_2 b)$. Jeśli to dodawanie jest wykonywane w każdej iteracji, to kosztuje ono $O(\log_2 a \log_2 b)$, zatem przy logarytmicznym kryterium kosztów otrzymujemy złożoność czasową $O(\log_2 ab)$.

0/8

Obliczenia możemy wykonywać w pierścieniu $\mathbb{Z}[x]/(x^3, m)$.

$$((ax^2 + bx + c) - 2)^2 = (2(c-2)a + b^2)x^2 + 2(c-2)bx + (c-2)^2$$

Problem możemy sprowadzić do obliczenia n-tej iteracji funkcji

$$\Phi \begin{pmatrix} a \\ b \\ c \end{pmatrix} = \begin{pmatrix} 2(c-2)a + b^2 \\ 2(c-2)b \\ (c-2)^2 \end{pmatrix}$$

na argumencie $x_0 = \left(\begin{smallmatrix} 0 \\ 1 \\ 0 \end{smallmatrix}\right)$ (odpowiadającym wielomianowi x).

Zauważmy, że trzecia współrzędna $\Phi^n(x_0)$ dla $n > 0$ będzie zawsze równa 4, a co za tym idzie, druga współrzędna będzie czterokrotnością drugiej współrzędnej argumentu. Druga współrzędna $\Phi(x_0)$ to -4, więc druga współrzędna $\Phi^n(x_0)$ to -4^n. Zatem problem się upraszcza do obliczenia $a_n = \varphi^n(0)$, gdzie

$$\varphi(a) = 4a + ((-4)^n)^2 = 4a + 16^n.$$

Obliczanie tych iteracji możemy zapisać w postaci macierzowej jako

$$\begin{pmatrix} a_n \\ 16^n \end{pmatrix} = \begin{pmatrix} 4 & 1 \\ 0 & 16 \end{pmatrix}^n \begin{pmatrix} a_0 \\ 16^0 \end{pmatrix} = \begin{pmatrix} 4 & 1 \\ 0 & 16 \end{pmatrix}^n \begin{pmatrix} 0 \\ 1 \end{pmatrix},$$

czyli wystarczy obliczyć lewy górny róg n-tej potęgi macierzy.

1/1

```
data Tree = Leaf | Branch Tree Tree

nodeCount Leaf         = 1
nodeCount (Branch l r) = 1 + nodeCount l + nodeCount r

-- liczy średnicę i wysokość drzewa
go Leaf         = (0, 0)
go (Branch l r) =
  let ((ld, lh), (rd, rh)) = (go l, go r) in
  (maximum [lh + rh + 2, ld, rd], maximum [lh, rh] + 1)

diameter t = fst (go t)
```

1/3

Do problemu wystarczy użyć algorytmu Kahna zmodyfikowanego tak, by używał kolejki priorytetowej.

$L \leftarrow []$
$Q \leftarrow$ kolejka priorytetowa wierzchołków bez krawędzi wchodzących
while Q **is not empty do**
    $v \leftarrow$ *pop-min*(Q)
    push(L, v)
    for $e = (v, u)$ **in** *outcoming-edges*(v) **do**
        remove-from-graph(e)
        if *incoming-edges*(u) **is empty then**
            insert(Q, u)
        end if
    end for
end while
return L
