AISD lista 0

Wiktor Kuchta

0/3

```
procedure BubbleSort(T[1..n])
    repeat
        swapped ← false
        for i ← 2 to n do
            if T[i − 1] > T[i] then
                swapped ← true
                T[i − 1] ↔ T[i]
            end if
        end for
    until not swapped
end procedure
```

Idea algorytmu: przechodzimy od lewej po tablicy i jeśli sąsiednie elementy są w złej kolejności, przestawiamy je. Takie przejścia powtarzamy, aż tablica będzie posortowana (nie będziemy musieli żadnej pary przestawić).

Algorytm zużywa $O(1)$ dodatkowej pamięci, tak samo jak SelectSort i InsertSort.

Kolejność dwóch wybranych elementów zmienia się w trakcie algorytmu wtedy i tylko wtedy, gdy element po lewej jest większy od elementu po prawej. W szczególności algorytm nie zmienia względnej kolejności elementów równych, czyli jest stabilny.

Gdy początkowo tablica jest posortowana rosnąco, to algorytm jej nie rusza i zatrzymuje się po jednym przejściu, więc wykonuje się w czasie $\Theta(n)$.

Zauważmy, że po i-tej iteracji zewnętrznej pętli i największych elementów jest na właściwym miejscu. Stąd możemy wywnioskować, że tych iteracji będzie co najwyżej n. Tak się dzieje w przypadku, gdy początkowo tablica jest posortowana malejąco. Jedna iteracja pętli zewnętrznej wykonuje się w $\Theta(n)$, więc w najgorszym przypadku algorytm wykonuje się w czasie $\Theta(n^2)$.

0/4

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 $\log_2 n$ iteracji wykonujemy ich dwie lub trzy, więc złożoność czasowa to $O(\log_2 n)$. W pamięci przechowujemy tylko dwie liczby, więc złożoność pamięciowa to $O(1)$.

W każdej iteracji wykonujemy operacje przesunięcia bitowego i dodawania na liczbach niewiększych niż nm, co przy logarytmicznym kryterium kosztów ma złożoność czasową $O(\log_2 nm)$. Zatem w tym przypadku złożoność czasowa algorytmu wynosi $O(\log_2 n \log_2 nm)$. Na przechowywanie liczb niewiększych niż nm potrzebujemy $\log_2 nm$ bitów pamięci, więc złożoność pamięciowa to $O(\log_2 nm)$.

0/5

Jeśli $x_n = a_k x_{n-1} + a_{k-1} x_{n-2} + \ldots + a_1 x_{n-k}$, to

$$\begin{pmatrix} x_{n-k-1} \\ x_{n-k-2} \\ \vdots \\ x_n \end{pmatrix} = \begin{pmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & \ddots & 0 \\ 0 & 0 & \ldots & 1 \\ a_1 & a_2 & \ldots & a_k \end{pmatrix} \begin{pmatrix} x_{n-k} \\ x_{n-k} \\ \vdots \\ x_{n-1} \end{pmatrix} = A \begin{pmatrix} x_{n-k} \\ x_{n-k} \\ \vdots \\ x_{n-1} \end{pmatrix}.$$

Jeśli $x_n = a_k x_{n-1} + a_{k-1} x_{n-2} + \ldots + a_1 x_{n-k} + w_0 n^0 + w_1 n^1 + \ldots + w_m n^m$, to

$$\begin{pmatrix} x_{n-k-1} \\ x_{n-k-2} \\ \vdots \\ x_n \\ (n+1)^0 \\ (n+1)^1 \\ \vdots \\ (n+1)^m \end{pmatrix} = \begin{pmatrix} A & W \\ 0 & P \end{pmatrix} \begin{pmatrix} x_{n-k} \\ x_{n-k} \\ \vdots \\ x_{n-1} \\ n^0 \\ n^1 \\ \vdots \\ n^m \end{pmatrix},$$

gdzie

$$W = \begin{pmatrix} 0 & 0 & 0 & 0 \\ \vdots & \vdots & \vdots & \vdots \\ w_0 & w_1 & \ldots & w_m \end{pmatrix},$$

a indeksując od zera $P_{ij} = \binom{i}{j}$ (korzystamy ze wzoru dwumianowego Newtona).

Teraz n-ty wyraz ciągu możemy obliczyć podnosząc macierz do potęgi i obliczając odpowiedni wiersz iloczynu z wyrazami początkowymi.

0/6

Zauważmy, że x (ostatni element, który został w A) jest równy wartości pewnego ponawiasowania wyrażenia

$$a_1 - a_2 - \ldots - a_n,$$

gdzie (a_n) to pewne wyliczenie elementów A. Czyli $x = \sum_{a \in A} (-1)^{\sigma_a} a$ dla pewnych $\sigma_a \in \{0, 1\}$, zatem

$$x \bmod 2 = \sum_{a \in A} (-1)^{\sigma_a} a \bmod 2 = \sum_{a \in A} a \bmod 2 = \bigoplus_{a \in A} (a \bmod 2).$$

Najoszczędniejszy algorytm obliczania $x \bmod 2$ trzyma jeden bit pamięci zainicjalizowany 0 i XORuje go z bitami parzystości liczb na wejściu.

0/7

Najpierw z wejścia w postaci listy krawędzi *rodzic→dziecko* musimy odczytać, który wierzchołek jest korzeniem. Możemy to zrobić tworząc tablicę przypisującą każdemu wierzchołkowi wartość logiczną mówiącą, czy ma pewnego ojca. Wtedy korzeń to jedyny wierzchołek, który nie ma ojca.

Zbudujmy tablicę trzymającą dla każdego wierzchołka u_i listę wierzchołków v_i, których obecność na ścieżce z u_i do korzenia mamy sprawdzić.

Zbudujmy dla każdego wierzchołka listę jego synów, aby móc wydajnie przeszukiwać drzewo w głąb. Przy przeszukiwaniu drzewa będziemy w tablicy trzymać dla

każdego wierzchołka wartość logiczną mówiącą, czy jest na obecnej ścieżce (na stosie wywołań DFSa).

Właściwe rozwiązanie problemu teraz polega na tym, że podczas przeszukiwania drzewa (rozpoczętego w korzeniu), odwiedzając każdy wierzchołek u_i, dla każdego z odpowiadających mu v_j odczytujemy bezpośrednio z tablicy odpowiedź, czy v_j jest na ścieżce z korzenia do u_i.

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.