ZAAWANSOWANE PROGRAMOWANIE KOMPUTEROWE WNE (2022)

KRZYSZTOF ZIEMIAŃSKI

4. Sortowanie

Sortowanie to operacja polegająca na uporządkowaniu elementów tablicy w pewnej ustalonej kolejności, rosnącej lub malejącej. Na przykład tablica

$$\begin{bmatrix}5 & 13 & 6 & 28 & 3 & 11 & 2\end{bmatrix}$$

po posortowaniu w kolejności rosnącej będzie wyglądać tak

$$\begin{bmatrix}2 & 3 & 5 & 6 & 11 & 13 & 28\end{bmatrix}.$$

Istnieje wiele algorytmów sortowania różniących się złożonością (tj. szybkością działania), łatwością implementacji, wymaganiem (bądź nie) dodatkowej pamięci do działania, itp. Na tych zajęciach poznamy kilka podstawowych algorytmów sortowania. Na początek zajłatwiejszy do zaimplementowania.

4.1. **Sortowanie bąbelkowe.** Ten algorytm korzysta bezpośrednio z definicji: tablica `t` jest posortowana rosnąco jeśli `t[i]<=t[i+1]` dla wszystkich indeksów `i`. Próbujemy więc poprawić tablicę tam, gdzie ten warunek nie jest spełniony: po prostu zamieniamy ze sobą sąsiednie elementy:

```
void babelek(vector<int>& t) {
    for(int i=1; i<t.size(); i++) {
        if(t[i-1]>t[i])
            zamien(t[i-1], t[i]);
    }
}
```

Po uruchomieniu tej funkcji dla tablicy

$$\begin{bmatrix}5 & 13 & 6 & 28 & 3 & 11 & 2\end{bmatrix}$$

uzyskujemy

$$\begin{bmatrix}5 & 6 & 13 & 3 & 11 & 2 & 28\end{bmatrix}$$

czyli tablica nadal nie jest posortowana (choć na pewno największy element znalazł się na końcu). Jeśli uruchomimy funkcję `babelek` po raz drugi, również drugi największy element znajdzie się na swoim (przedostatnim) miejscu. Po wielokrotnym (wystarczy rozmiar tablicy minus jeden) wywołaniu tablica będzie posortowana.

Jest to algorytm najłatwiejszy do napisania. Zastanówmy sie nad szybkością jego działania. Dla tablicy o rozmiarze n pierwszy przebieg wymaga $n-1$ porównań i średnio $(n-1)/2$ zamian. W k–ty przebiegu nie musimy już zwracać uwagi na $k-1$ ostatnich elementów, wystarczy więc wykonać $n-k$ porównań i średnio $(n-k)/2$ zamian. Oczekiwana liczba zamian wynosi więc

$$\tfrac{1}{2}((n-1)+(n-2)+\cdots+3+2+1)\approx n^2/4,$$

czyli jest rzędu n^2. Mówimy wtedy o algorytmie o złożoności $O(n^2)$.

4.2. **Sortowanie przez wstawianie.** Ideą tego algorytmu jest sortowanie tablicy "od początku". W każdym kolejnym kroku startujemy od tablicy, w której początkowy k–elementowy fragment jest posortowany; naszym celem jest przedłużenie posortowanego fragmentu o 1.

Przykład: startujemy od tablicy

$$\begin{bmatrix}\mathbf{5} & 13 & 8 & 4 & 7 & 11 & 2\end{bmatrix}$$

Oczywiście jednoelementowy początek (oznaczony grubszą czcionką) jest posortowany. W pierwszym kroku nie musimy nic robić, tylko upewniamy się pierwsze dwa elementy stanowią ciąg posortowany; otrzymujemy

$$\begin{bmatrix}\mathbf{5} & \mathbf{13} & 8 & 4 & 7 & 11 & 2\end{bmatrix}$$

W kolejnym kroku musimy wstawić liczbę 8 na odpowiednie miejsce, tzn. pomiędzy 5 a 13:

$$\begin{bmatrix}\mathbf{5} & \mathbf{8} & \mathbf{13} & 4 & 7 & 11 & 2\end{bmatrix}$$

Kolejne kroki to:

$$\begin{bmatrix}\mathbf{4} & \mathbf{5} & \mathbf{8} & \mathbf{13} & 7 & 11 & 2\end{bmatrix}$$
$$\begin{bmatrix}\mathbf{4} & \mathbf{5} & \mathbf{7} & \mathbf{8} & \mathbf{13} & 11 & 2\end{bmatrix}$$
$$\begin{bmatrix}\mathbf{4} & \mathbf{5} & \mathbf{7} & \mathbf{8} & \mathbf{11} & \mathbf{13} & 2\end{bmatrix}$$
$$\begin{bmatrix}\mathbf{2} & \mathbf{4} & \mathbf{5} & \mathbf{7} & \mathbf{8} & \mathbf{11} & \mathbf{13}\end{bmatrix}$$

Teraz tablica jest posortowana.

Szybkość działania tego algorytmu jest zbliżona do szybkości sortowania bąbelkowego. W każdym kroku musimy znaleźć miejsce na które wstawiamy nowy element (co można zrobić szybko), ale przestawienie nowego elementu w k–tym kroku wymaga średnio $k/2$ przypisań, więc złożoność sortowania przez wstawianie wynosi $O(n^2)$. Gdyby jednak elementy tablicy dały się "rozsuwać", algorytm działałby szybciej. W dalszym ciągu kursu poznamy struktury danych, które to umożliwiają.

4.3. **Sortowanie szybkie.** Jest to algorytm, który jest najczęściej wykorzystywany w programach. Żeby posortować tablicę `t` o rozmiarze `n`, wykonujemy następujące kroki:

(1) Wybierz pewien element tablicy, np. pierwszy `t[0]` (tak jest najwygodniej),

(2) Przestaw elementy tak, aby te mniejsze lub równe `t[0]` znalazły się w początkowej części tablicy (na miejscach $0, \ldots, k-1$), te większe od `t[0]` na końcu (na miejscach $k+1, \ldots, n-1$), a `t[0]` pomiędzy nimi (na miejscu k),

(3) Posortuj fragment tablicy o indeksach $0, \ldots, k-1$.

(4) Posortuj fragment tablicy o indeksach $k+1, \ldots, n-1$.

Jak widzimy algorytm używa rekurencji. Pozostaje jeszcze wyjaśnić w jaki sposób odpowiednio szybko wykonać krok (2). W poniższym przykładzie liczby mniejsze od 5 (pierwszego elementu tablicy) są niebieskie, a te większe czerwone. Naszym celem jest je poprzestawiać tak, wszystkie niebieskie liczby znalazły się przed czerwonymi. Wybieramy dwa wskaźniki: L ustawiony na pierwszym kolorowym elemencie i P ustawiony na ostatnim:

$$\begin{bmatrix}5 & \overset{L}{1} & 6 & 9 & 3 & 11 & \overset{P}{2}\end{bmatrix}$$

Wskaźnik L będziemy przesuwać w prawo, a wskaźnik P w lewo. Musimy przestrzegać dwóch zasad:

- Wszystkie liczby na lewo od L są niebieskie.
- Wszystkie liczby na prawo od P są czerwone.

Naszym celem jest aby wskaźniki się wyminęły (tj. żeby $P < L$). Jeśli wskaźnik L pokazuje niebieski element (jak na przykładzie), przesuwamy go w prawo:

$$\begin{bmatrix} 5 & 1 & \overset{L}{6} & 9 & 3 & 11 & \overset{P}{2} \end{bmatrix}$$

Podobnie moglibyśmy przesunąć w lewo P, gdyby pokazywał czerwony element. Jeśli nie możemy przesunąć żadnego ze wskaźników, zamieniamy wskazywane elementy:

$$\begin{bmatrix} 5 & 1 & \overset{L}{2} & 9 & 3 & 11 & \overset{P}{6} \end{bmatrix}$$

Teraz znowu możemy przesuwać wskaźniki:

$$\begin{bmatrix} 5 & 1 & 2 & \overset{L}{9} & 3 & 11 & \overset{P}{6} \end{bmatrix}$$

$$\begin{bmatrix} 5 & 1 & 2 & \overset{L}{9} & 3 & \overset{P}{11} & 6 \end{bmatrix}$$

$$\begin{bmatrix} 5 & 1 & 2 & \overset{L}{9} & \overset{P}{3} & 11 & 6 \end{bmatrix}$$

I znów zamieniamy

$$\begin{bmatrix} 5 & 1 & 2 & \overset{L}{3} & \overset{P}{9} & 11 & 6 \end{bmatrix}$$

przesuwamy

$$\begin{bmatrix} 5 & 1 & 2 & \overset{P}{3} & \overset{L}{9} & 11 & 6 \end{bmatrix}$$

Cel został osiągniety. Teraz wstawiamy pierwszy element pomiędzy niebieskie i czerwone

$$\begin{bmatrix} 3 & 1 & 2 & \overset{P}{5} & \overset{L}{9} & 11 & 6 \end{bmatrix}$$

i możemy przejść do punktów (3) i (4) i posortować poszczególne fragmenty tablicy.

Zazwyczaj ten algorytm (zgodnie z nazwą) działa szybko. Jeśli mamy szczęście i początkowy element tablicy jest "średni", tzn. podobna liczba elementów jest od niego większa i mniejsza, to liczba operacji $F(n)$ potrzebna na posortowanie tablic o długości n dana jest przybliżonym wzorem rekurencyjnym

$$F(n) = n + F(n/2) + F(n/2).$$

Poszczególne składniki to liczby operacji potrzebne na wykonanie punktów (2), (3) i (4). Można sprawdzić, że wówczas $F(n) \sim O(n \log_2 n)$, a więc złożoność jest istotnie niższa niż w przypadku dwóch pozostałych algorytmów. Jeśli jednak mamy pecha i zawsze pierwsza liczba jest największa (lub najmniejsza), sortowanie szybkie działa równie wolno jak sortowanie bąbelkowe.

4.4. **Sortowanie przez łączenie.** Ten sposób również wykorzystuje rekurencję. Żeby posortować tablicę `t` o rozmiarze n:

(1) Dzielimy tablicę na dwie równe (lub prawie równe) części.

(2) Sortujemy lewą część.

(3) Sortujemy prawą część.

(4) Łączymy (scalamy) posortowane części.

Trzeba jeszcze opisać jest operację scalania. Potrzebna będzie nam dodatkowa tablica `s`. Załóżmy, że po wykonaniu kroków (2) i (3) tablica `t` wygląda jak poniżej.

$$\texttt{t}: \begin{bmatrix} \overset{L}{1} & 6 & 8 & 9 & \overset{P}{2} & 4 & 7 \end{bmatrix}$$
$$\texttt{s}: \begin{bmatrix} \overset{I}{?} & ? & ? & ? & ? & ? & ? \end{bmatrix}$$

Użyjemy trzech wskaźników: L przebiega lewą stronę `t`, P przebiega prawą stronę `t`, a I przebiega `s`. Postępujemy następująco:

(1) Jeśli `t[`L`]<=t[`P`]` lub przepisaliśmy już wszystkie elementy z prawej części, to przepisujemy `t[`L`]` do tablicy `s` w miejsce wskazywane przez wskaźnik I; następnie zwiększamy L i I.

(2) Jeśli `t[`L`]>t[`P`]` lub przepisaliśmy już wszystkie elementy z lewej części, to przepisujemy `t[`P`]` do tablicy `s` w miejsce wskazywane przez wskaźnik I; następnie zwiększamy P i I.

(3) Jeśli jeszcze nie przepisaliśmy wszystkiego przechodzimy to punktu (1).

Oto przebieg scalania na przykładzie:

Krok 1

$$\texttt{t}: \begin{bmatrix} 1 & \overset{L}{6} & 8 & 9 & \overset{P}{2} & 4 & 7 \end{bmatrix}$$
$$\texttt{s}: \begin{bmatrix} 1 & \overset{I}{?} & ? & ? & ? & ? & ? \end{bmatrix}$$

Krok 2

$$\texttt{t}: \begin{bmatrix} 1 & \overset{L}{6} & 8 & 9 & 2 & \overset{P}{4} & 7 \end{bmatrix}$$
$$\texttt{s}: \begin{bmatrix} 1 & 2 & \overset{I}{?} & ? & ? & ? & ? \end{bmatrix}$$

Krok 3

$$\texttt{t}: \begin{bmatrix} 1 & \overset{L}{6} & 8 & 9 & 2 & 4 & \overset{P}{7} \end{bmatrix}$$
$$\texttt{s}: \begin{bmatrix} 1 & 2 & 4 & \overset{I}{?} & ? & ? & ? \end{bmatrix}$$

Krok 4

$$\texttt{t}: \begin{bmatrix} 1 & 6 & \overset{L}{8} & 9 & 2 & 4 & \overset{P}{7} \end{bmatrix}$$
$$\texttt{s}: \begin{bmatrix} 1 & 2 & 4 & 6 & \overset{I}{?} & ? & ? \end{bmatrix}$$

Krok 5

$$\texttt{t}: \left[1 \quad 6 \quad \overset{L}{8} \quad 9 \quad 2 \quad 4 \quad 7\right]$$

$$\texttt{s}: \left[1 \quad 2 \quad 4 \quad 6 \quad 7 \quad \overset{I}{?} \quad ?\right]$$

Krok 6

$$\texttt{t}: \left[1 \quad 6 \quad 8 \quad \overset{L}{9} \quad 2 \quad 4 \quad 7\right]$$

$$\texttt{s}: \left[1 \quad 2 \quad 4 \quad 6 \quad 7 \quad 8 \quad \overset{I}{?}\right]$$

Krok 7

$$\texttt{t}: \left[1 \quad 6 \quad 8 \quad 9 \quad \overset{L}{2} \quad 4 \quad 7\right]$$

$$\texttt{s}: \left[1 \quad 2 \quad 4 \quad 6 \quad 7 \quad 8 \quad 9\right]$$

Elementy są posortowane, ale znajdują się w innej tablicy niż oryginalna (choć oczywiście możemy je przepisać z powrotem). Sortowanie przez łączenie wymaga użycia dodatkowej pamięci i jest to jego wadą. Jego oczekiwana (i pesymistyczna zarazem) złożoność jest dana wzorem takim jak w przypadku "optymistycznej" wersji sortowania szybkiego:

$$F(n) = \overset{\text{Sortowanie lewej części}}{F(n/2)} + \overset{\text{Sortowanie prawej części}}{F(n/2)} + \overset{\text{Scalanie}}{n}$$

czyli $F(n) \sim O(n \log_2 n)$.

4.5. **Zadania.**

(1) Napisać funkcję

```
int sortowanie_wstawianie(vector<int>& t);
```

która sortuje przez wstawianie tablicę `t`. Wynik jest liczbą dokonanych operacji (przypisań).

(2) Napisać funkcję

```
int sortowanie_scalanie(vector<int>& t);
```

która sortuje przez łączenie tablicę `t`. Wynik jest liczbą dokonanych operacji.

(3) Można rozważać inny warianty sortowania bąbelkowego. Np. kiedy sortujemy tablicę o rozmiarze 100, w pierwszym przebiegu porównujemy elementy odległe o 20, w drugim o 19, itd.; dopiero w końcowych przebiegach porównujemy sąsiednie elementy. Napisać funkcję (bądź funkcje) która sortuje tablice używając zmodyfikowanego algorytmu bąbelkowego. Następnie przeanalizować jaki dobór "odstępów" prowadzi do najszybszego działania.