

Sprawozdanie z listy 1 (AiSD)

Szymon Wojtasik, nr albumu: 287306

Streszczenie

W tym sprawozdaniu omówimy 4 algorytmy sortujące. Każdy z nich jest inny, a szczególnie czasy ich wykonywania często się bardzo różnią. Z tych 4 algorytmów wyjmiemy sobie najciekawsze fragmenty kodu.

1 QuickSort

Algorytm bazuje na pivotach, czyli elementach odpowiadającym za porównanie czy dane elementy są mniejsze(równe) czy większe od pivotów. Najciekawszy dla mnie fragment kodu to ten poniższy:

```
1 for (int p = m; p <= n;) {
2     if (A[p] < y) {
3         swap(A[p], A[m]);
4         m++;
5         p++;
6     }
7     else if (A[p] > k) {
8         swap(A[p], A[n]);
9         n--;
10        continue;
11    }
12    else {
13        p++;
14    }
15}
16 swap(A[początek_2], A[m - 1]);
17 swap(A[koniec_2], A[n + 1]);
18 return {m - 1, n + 1};
19 }
```

W 2 linijce porównujemy element z mniejszym pivotem(na pierwszym miejscu) i jeżeli element jest mniejszy od pivota to zamieniamy go z A[m], gdzie m jest skorelowane z tym ile jest elementów mniejszych od pivota mniejszego. Zwiększamy p, ale póki co może być to nieoczywiste. Na razie zastanówmy się nad dalszym działaniem algorytmu.

Jeżeli pierwszy warunek nie jest prawdziwy to sprawdzamy czy element jest większy od większego pivota. W przypadku prawdy zmniejszamy indeks n, gdzie n jest skorelowany z tym ile elementów jest mniejszych od większego pivota.

W innym(jedynym) przypadku element jest pośrodku pivotów.

Pierwsza zagwostka:

Czemu zwiększamy tylko p na pierwszym i ostatnim warunku(gdzie p odzwierciedla ile elementów jest mniejszych od większego pivota)? Odpowiedź jest nieoczywista. Rozpatrzmy jakie mamy możliwości.

\Rightarrow Element jest mniejszy od większego pivota i większy od mniejszego. W takim przypadku zwiększamy p.

\Rightarrow Element jest mniejszy od mniejszego pivota. Teraz również zwiększamy p, ponieważ jest to oczywiście prawda, że ten element jest większy od większego pivota

\Rightarrow Jeżeli element jest większy od większego pivota to nie zwiększamy p, co myślę że jest logiczne.

Czemu w takim razie **continue**? Odpowiedź jest taka, że przecież po zamianie elementu, może się okazać że dalej będzie on większy od większego pivota, a przecież nie o to nam chodzi.

2 RadixSort

Ideą radix sort jest sortowanie poprzez sprawdzanie kolejnych cyfr liczby(jedności, dziesiątki, setki itd.).

```
1 for (int j = 1; j < d; j++) {  
2     C[j] += C[j-1];  
3 }  
4 for (int i = n - 1; i >= 0; i--) {  
5     B[C[f(A[i])] - 1] = A[i];  
6     C[f(A[i])]--;  
7 }
```

Po pętli wartość $C[j]$ pokazuje na cyfrę jedności tablicy B, którą ma zająć element z cyfrą jedności równą j.

Iterujemy od tyłu, aby zachować stabilność. Pętla przechodzi przez tablicę A od końca. Wstawiamy element $A[i]$ do tablicy B na cyfrę o jeden mniejszą(jak były jedności to teraz są dziesiątki) niż aktualna wartość w C, czyli $B[C[f(A[i])] - 1]$.

Algorytm jest stabilny, ponieważ po wstawieniu licznik $C[f(A[i])]$ jest zmniejszany o 1. Dzieje się tak, ponieważ zmniejszenie licznika daje to, że następny element w A, który będzie miał tę samą cyfrę trafi na cyfrę o jeden mniejszą w B. Daje to dużo, ponieważ w ten sposób zachowujemy pierwotną kolejność elementów o tych samych cyfrach(na przykład jedności). Bez tego RadixSort nie działałby tak dobrze