ASD, wykład 08, 2.04.202

Sortowania kubełkowe i pozycyjne

$O(n \cdot \log_2(n))$ – dolne ograniczenie na ilość porównań w sortowaniu opartym o porównania.

Dla sortowania, które nie są oparte o porównania:

sortowanie przez zliczanie

— " — pozycyjne

Sortowanie przez zliczanie

Załóżmy, że sortujemy liczby naturalne z przedziału $[0, \ldots, N]$. $t[0..n]$

Możemy mieć dużą tablicę ale kluczy w tablicy mamy tylko $N+1$.

Wtedy możemy policzyć ile mamy elementów o danym kluczu — $O(n)$.

potem umieszczamy elementy na swoich miejscach — $O(n)$.

Sortujemy tablicę t[0..n-1] i mamy klucze [0,…,N]

```
Counting_sort(int *t, int n, int bound){ // bound=N= ograniczenie na wielkosc kluczy
        int counters[0..bound]={0,0,….,0};
        int whereToStart[0..bound]={0,0,….,0};//whereToStart[i] mówi gdzie w posortowanej tablicy zaczynaja się elementy o wartości i
        int pom[0,…,n-1];
        //Liczymy ile jest elementów o danym kluczu
        for(i=0;i<n;i++)
                counters[ t[i] ]++; //inkrementujemy licznik dla elementu t[i]
         //Teraz counters przechowuje liczności elementów w tablicy t

        // Wypełniamy whereToStart
         // whereToStart[0]==0
        for(i=1;i<=bound;i++)
                whereToStart[i]= whereToStart[i-1]+counters[i-1];
         //whereToStart przechowuje to co chcieliśmy wyżej

          //Umieszczamy elementy na właściwych pozycjach w tablicy pom
        for(i=0;i<n;i++){
           pom[ whereToStart[ t[i] ] ]  = t[i];
           whereToStart[ t[i] ] ++;
         }
       //przepisujemy wartości z powrotem do tablicy t
       t[0..n-1] = pom[0..n-1];
 }
```

Zauważmy, że sortowanie takie jak opisane na poprzednim slajdzie, jest stabilne, tzn.

Dla dowolnych 0<=i<j<n, jeśli element t[i] był równy elementowi t[j],
to w posortowanej tablicy element t[i] będzie przed elementem t[j].

Przykład

Max bound = 4

t[0..7] = 1, 3, 1, 0, 4, 0, 4, 1

counters = [2 3 0 1 2] — O(n)

0 1 2 3 4

whereTo Start [0..4]

0 1 2 3 4

whereTo Start [0, 2, 5, 5, 6] — O(N)

N = bound

1 3 6 7

2 4 8

5

i = 0, t[i] = 1, whereTS[1] = 2

i = 1, t[i] = 3, whereTS[3] = 5

i = 2, t[i] = 1, whereTS[1] = 3

pom = [0 0 1 1 1 3 4 4] — O(n)

0 1 2 3 4 5 6 7

Końcowy koszt to

$$O(n) + O(N)$$

Mamy szybkie sortowanie (ale N nie jest tak duże, że koszt czasowy algorytmu [illegible] pamięci dla N nie jest za duży – i wygenerowania jej

Gdzie są "ciemne strony" tego sortowania?

1) Musimy znać zakres danych (Min i Max)
2) Musimy alokować dużą tablicę counters.

Np. $t = [0, 1, 1000000000, 4]$

W tych potrzebujemy miliard liczników.

Wymóg pamięciowe mogą być nieakceptowalne.

~~Sam~~ Warto to rozważyć, gdy mamy ograniczenie na wielkość danych i nie jest ono zbyt duże.

Omyłkowe nazwy takiej macierzy powtarzają na indeksowanie nimi tablicy counters.

Sortowanie pozycyjne

Sortujemy liczby naturalne (całk. ≥ 0).

I załóżmy, że długość rozwinięcia binarnego tych liczb jest $\leq N$.

np. $73 = 64 + 0\cdot 32 + 0\cdot 16 + 1\cdot 8 + 0\cdot 4 + 0\cdot 2 + 1$

$= (1001001)_2$

$6\ 5\ 4\ 3\ 2\ 1\ 0$

bity numerujemy - bit zerowy to najmniej znaczący bit przy 2^0

skrajnie lewy bit - najbardziej znaczący

Żeby napisać sortowanie przez zliczanie potrzebujemy ~~#~~ stabilnego sortowania tablic zero-jedynkowych w czasie liniowym (np. counting sort).

~~Wiedz~~

Def Sortowanie jest stabilne jeśli dla każdego wejścia $t[0 \ldots n]$ i dla dowolnych i, j $0 \leq i \leq j \leq n$ jeśli $a = t[i] = t[j] = b$, to w posortowanej tablicy element a ten jest przed b.

(cyk.

Dla dowolnej tablicy $t[0..n]$

i dla dowolnych i, j $0 \le i < j \le n$

jeśli $a = t[i] = t[j] = b$,

to a jest przed b w tablicy posortowanej.

Sortowanie stabilne nie zmienia kolejności równych sobie elementów.

Niech stable-01-sort(int t, int n, int i)

to:
- sortowanie stabilne
- w czasie liniowym
- sortuje liczby w tablicy t względem i-tego bitu.

Wg. mamy napisać takie sortowanie w oparciu o counting sort.

Sortowanie pozycyjne - kod

(radix sort)

```
radixSort(int t, int n, int bitsize){
// sortujemy t[0...n], liczby w t >=0,
//          oraz maja bity od 0 do bitsize
    for(i=0; i<=bitsize; i++)
        stable-01-sort(t, n, i);
}
```

Sortujemy w zasadzie liczby unsigned int.

Przykład

i = 0

	t[6]							
9		1	0	0	1	1	1	0	0
12		1	1	0	0	1	0	1	0
10		1	0	1	0	1	0	0	1
7		0	1	1	1	0	1	1	1
5		0	1	0	1	0	1	0	1
9		1	0	0	1	1	0	0	1
6	t[6]:	0	1	1	0				

12

i = 1

1	1	0	0
1	0	0	1
0	1	0	1
1	0	0	1
1	0	1	0
0	1	1	0
0	1	1	1

i = 2

1	0	0	1
1	0	0	1
1	0	1	0
1	1	0	0
0	1	0	1
0	1	1	0
0	1	1	1

i = 3

0	1	0	1	: 5
0	1	1	0	: 6
0	1	1	1	: 7
1	0	0	1	: 9
1	0	0	1	: 9
1	0	1	0	10
1	1	0	0	12

9	12	12	9	5
12	10	9	9	6
10	6	5	10	7
7	9	9	12	9
5	9	10	5	9
9	7	6	6	10
6	5	7	7	12
	9			

Po i-tym przebiegu pętli radix-sort
liczby w tablicy są posortowane
dla części rozumianej co do bitu i do bitu 0

Uwaga

W praktyce nie sortujemy po bitach
tylko po kilkuwartościowych bitowych cyfrach.

Dzielimy teraz tak sortowane ciągi znaków,
np. uzupełniamy te ciągi ~~spacją~~ symbolem $-\infty$
na ~~końcu~~ ~~bardziej~~ najmniejszych miejscach (tych po prawej)
i otrzymamy porządek leksykograficzny.

Złożoność radix-sort.

Jeśli wielkości liczb jest ograniczona i mają najwyżej c symboli zapisu (bitów)

to mamy złożoność $c \cdot n$, gdzie n to wielkość tablicy.

Ale jeśli wielkości naszych liczb jest rzędu n, to ich zapis binarny ma długość $\log_2(n)$ i mamy złożoność $O(n \cdot \log_2(n))$.

a b c

a b a

0

a b c

a b c

1

a b c

a b c

2

a a c a

a a b a a b d

a b c ∞

a b c d