Sortowanie przez scalanie

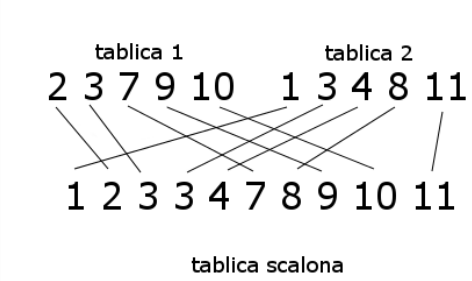
Sortowanie przez scalanie należy do algorytmów, które wykorzystują metodę "**dziel i zwyciężaj**". Złożoność algorytmu wynosi $n \cdot \log n$, a więc jest on znacznie wydajniejszy niż sortowanie bąbelkowe, przez wstawianie czy przez selekcję, gdzie złożoność jest kwadratowa. Żeby zrozumieć zasadę działania przyjrzyjmy się najpierw dwóm posortowanym tablicom:

tablica 1: **2 3 7 9 10**

tablica 2: **1 3 4 8 11**

Zauważmy, że możemy **liniowo** scalić te dwa ciągi liczb i uzyskać jedną posortowaną tablicę postępując ze schematem:

- ustawiamy liczniki na początki tablic posortowanych,
- następnie porównujemy elementy i mniejszy lub równy element wskakuje jako pierwszy w scalonej tablicy,
- zwiększamy licznik w tej tablicy, z której "zabraliśmy element",
- czynność powtarzamy aż do wyczerpania danych z obu tablic.



A więc najpierw musimy doprowadzić do sytuacji, gdzie będziemy mieli dwie posortowane tablice. Dzielimy nasz zbiór liczb na dwie części, następnie każdą z nich także dzielimy na dwie części, czynność powtarzamy do momentu otrzymania podtablic jednoelementowych (wykonujemy to rekurencyjnie). Ponieważ zbiór jednoelementowy jest już posortowany możemy przejść do scalania. W ten sposób powstają nam coraz to większe posortowane zbiory, aż w rezultacie otrzymamy oczekiwany efekt - ciąg posortowanych elementów.

Zalety algorytmu

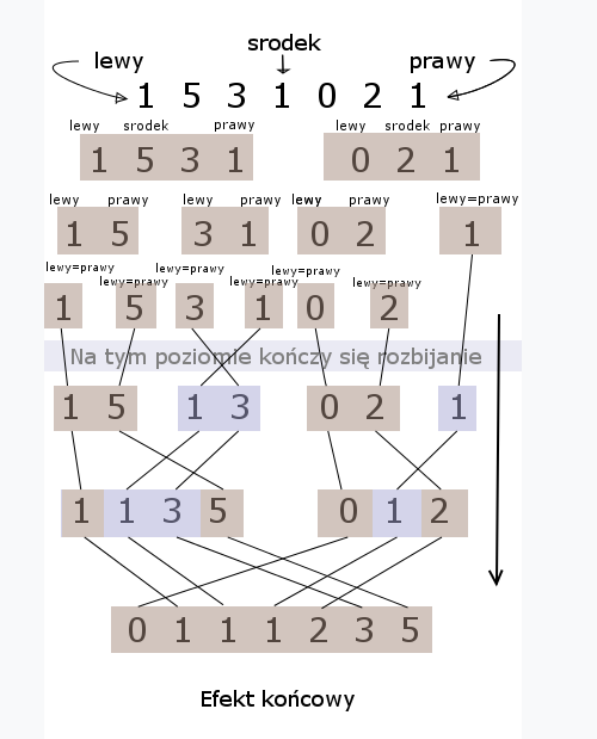
- prostota implementacji
- wydajność
- stabilność
- algorytm sortuje zbiór **n-elementowy** w czasie proporcjonalnym do liczby $n \log n$

- bez względu na rodzaj danych wejściowych

Wady algorytmu

- podczas scalania potrzebny jest dodatkowy obszar pamięci przechowujący kopie podtablic do scalenia

Schemat:



Implementacja algorytmu przez scalanie:

```
//algorytm.edu.pl
#include<iostream>
using namespace std;

int *pom; //tablica pomocnicza, potrzebna przy scalaniu

//scalenie posortowanych podtablic
void scal(int tab[], int lewy, int srodek, int prawy)
{
        int i = lewy, j = srodek + 1;

  //kopiujemy lewą i prawą część tablicy do tablicy pomocniczej
  for(int i = lewy;i<=prawy; i++)
    pom[i] = tab[i];

  //scalenie dwóch podtablic pomocniczych i zapisanie ich
  //we własciwej tablicy
  for(int k=lewy;k<=prawy;k++)
  if(i<=srodek)
    if(j <= prawy)
         if(pom[j]<pom[i])
             tab[k] = pom[j++];
         else
             tab[k] = pom[i++];
    else
        tab[k] = pom[i++];
  else
      tab[k] = pom[j++];
}

void sortowanie_przez_scalanie(int tab[],int lewy, int prawy)
```

```
{
        //gdy mamy jeden element, to jest on już posortowany
        if(prawy<=lewy) return;

        //znajdujemy srodek podtablicy
        int srodek = (prawy+lewy)/2;

        //dzielimy tablice na część lewą i prawa
        sortowanie_przez_scalanie(tab, lewy, srodek);
        sortowanie_przez_scalanie(tab, srodek+1, prawy);

        //scalamy dwie już posortowane tablice
        scal(tab, lewy, srodek, prawy);
}

int main()
{
        int *tab,
        n; //liczba elementów tablicy

        cin>>n;
        tab = new int[n]; //przydzielenie pamięci na tablicę liczb
        pom = new int[n]; //przydzielenie pamięci na tablicę pomocniczą

        //wczytanie elementów tablicy
        for(int i=0;i<n;i++)
                cin>>tab[i];

        //sortowanie wczytanej tablicy
        sortowanie_przez_scalanie(tab,0,n-1);

        //wypisanie wyników
        for(int i=0;i<n;i++)
                cout<<tab[i]<<" ";

        return 0;
}
```

Nieco szybsza wersja funkcji scalającej:

```
//scalenie posortowanych podtablic
void scal(int tab[], int lewy, int srodek, int prawy)
{
  int i, j;

  //zapisujemy lewą część podtablicy w tablicy pomocniczej
  for(i = srodek + 1; i>lewy; i--)
    pom[i-1] = tab[i-1];

  //zapisujemy prawą część podtablicy w tablicy pomocniczej
  for(j = srodek; j<prawy; j++)
    pom[prawy+srodek-j] = tab[j+1];

  //scalenie dwóch podtablic pomocniczych i zapisanie ich
  //we własciwej tablicy
  for(int k=lewy;k<=prawy;k++)
    if(pom[j]<pom[i])
      tab[k] = pom[j--];
    else
      tab[k] = pom[i++];
}
```

Inne rozwiązanie:

```
// Scalanie ciągów
// www.algorytm.org

#include <iostream>
using namespace std;

int main()
{
int tablica1[10] = { 1, 13, 43, 77, 88, 109, 121, 144, 165, 200 };
int tablica2[10] = { 0, 23, 44, 77, 99, 101, 111, 150, 162, 199 };
int tablica_wynikowa[20];
int i, j, k; // zmienne pomocnicze

// wyswietlenie wygenerowanych ciagow
cout << "Ciag pierwszy: ";
for (i = 0; i < 10;
i++)
cout << tablica1[i] << " ";
cout << endl;
cout << "Ciag drugi: ";
for (i = 0; i < 10; i++)
cout << tablica2[i] << " ";
cout << endl;

// scalanie ciagow
for (i = j = k = 0; i < 10 && j < 10; k++)
{
if (tablica1[i] < tablica2[j]) // sprawdzamy, ktora liczba jest mniejsza
{
tablica_wynikowa[k] = tablica1[i]; // mniejsza wpisujemy do tablicy
i++; // zwiekszamy indeks w tablicy, w ktorej byl mniejszy element
}
else
{
tablica_wynikowa[k] = tablica2[j];
j++;
}
}
// przepisanie reszty tablicy pierwszej lub drugiej w zaleznosci, od tego ktora juz przepisalismy w calosci do tablicy
wynikowej
if (i < 10)
for ( ; i < 10; i++, k++)
tablica_wynikowa[k] = tablica1[i];
else
for ( ; j < 10; j++, k++)
tablica_wynikowa[k] = tablica2[j];

// wypisanie tablicy wynikowej
cout << "Ciag posortowany: ";
for (i = 0; i < 20; i++)
cout << tablica_wynikowa[i] << " ";
cout << endl;

return 0;
}
```

Rekurencyjne sortowanie tablic:

1.
a) Napisz funkcję rekurencyjną realizującą sortowanie tablicy liczbowej przez scalanie: void ScalanieSort(int A[], int lewy, int prawy) gdzie lewy i prawy oznaczają indeksy ograniczające sortowany fragment tablicy A.
b) Napisz program, który posortuje niemalejąco tablicę n-elementową.
c) Uzupełnij program tak, aby podał liczbę rekurencyjnych wywołań funkcji ScalanieSort przy sortowaniu tablicy n-elementowej (pierwsze wywołanie, dla całej tablicy, ma postać: ScalanieSort(A, 0, n-1)).

Wejście
Pierwszy wiersz standardowego wejścia zawiera jedną liczbę całkowitą dodatnią Z określającą liczbę testów. Pierwszy wiersz każdego testu zawiera liczbę całkowitą dodatnią ni (1 <= ni <= 1000) określającą liczbę elementów ciągu do posortowania. Elementy te, liczby całkowite z przedziału [-10^6 ; 10^6] oddzielone pojedynczymi spacjami, podane są w drugim wierszu testu.

Wyjście
Standardowe wyjście powinno zawierać po dwa wiersze odpowiedzi dla każdego testu. Pierwszy wiersz powinien zawierać oddzielone pojedynczymi spacjami posortowane niemalejąco liczby podane na wejściu. Drugi wiersz powinien zawierać informację o liczbie wywołań funkcji ScalanieSort w poniższym formacie:
Liczba wywolan funkcji ScalanieSort = xx

Przykład
Dla danych:
2
10
8 3 4 2 9 0 1 7 6 5
6
123 88 331 647 1967 -753

Poprawny wynik może mieć postać
0 1 2 3 4 5 6 7 8 9
Liczba wywolan funkcji ScalanieSort = 19
-753 88 123 331 647 1967
Liczba wywolan funkcji ScalanieSort = 11

2.
a) Napisz funkcję rekurencyjną realizującą sortowanie szybkie:
void QuickSort(int A[], int lewy, int prawy)
gdzie lewy i prawy oznaczają indeksy ograniczające sortowany fragment tablicy A.
b) Napisz program, który posortuje niemalejąco tablicę n-elementową.
c) Uzupełnij program tak, aby podał liczbę rekurencyjnych wywołań funkcji QuickSort przy sortowaniu tablicy n-elementowej (pierwsze wywołanie, dla całej tablicy, ma postać: QuickSort(A, 0, n-1)).

Wejście
Pierwszy wiersz standardowego wejścia zawiera jedną liczbę całkowitą dodatnią Z określającą liczbę testów. Pierwszy wiersz każdego testu zawiera liczbę całkowitą dodatnią ni (1 <= ni <= 1000) określającą liczbę elementów ciągu do posortowania. Elementy te, liczby całkowite z przedziału [-10^6 ; 10^6] oddzielone pojedynczymi spacjami, podane są w drugim wierszu testu.

Wyjście
Standardowe wyjście powinno zawierać po dwa wiersze odpowiedzi dla każdego testu. Pierwszy wiersz powinien zawierać oddzielone pojedynczymi spacjami posortowane niemalejąco liczby podane na wejściu. Drugi wiersz powinien zawierać informację o liczbie wywołań funkcji QuickSort w poniższym formacie:
Liczba wywolan funkcji QuickSort = xx

Przykład
Dla danych:
2
10

8 3 4 2 9 0 1 7 6 5
6
123 88 331 647 1967 -753

Poprawny wynik może mieć postać
0 1 2 3 4 5 6 7 8 9
Liczba wywołań funkcji QuickSort = 19
-753 88 123 331 647 1967
Liczba wywolan funkcji QuickSort = 11

```
#include <iostream>
#include <iomanip>
#include <cstdlib>

using namespace std;

const int N = 20;

//int d[N];

int d[N] = { 1, 13, 43, 77, 88, 109, 121, 144, 165, 200, 0, 23, 44, 77, 99, 101,
111, 150, 162, 199 };

void Sortuj_szybko(int lewy, int prawy)
{
    int i, j, piwot;

    i = (lewy + prawy) / 2;
    piwot = d[i]; d[i] = d[prawy];
    for (j = i = lewy; i < prawy; i++)
        if (d[i] < piwot)
        {
            swap(d[i], d[j]);
            j++;
        }
    d[prawy] = d[j]; d[j] = piwot;
    if (lewy < j - 1)  Sortuj_szybko(lewy, j - 1);
    if (j + 1 < prawy) Sortuj_szybko(j + 1, prawy);
}

int main()
{
    int i;
    cout << " Sortowanie szybkie \nPrzed sortowaniem:\n";

    for (i = 0; i < N; i++) cout << setw(4) << d[i];

    Sortuj_szybko(0, N - 1);

    cout << "\n Po sortowaniu:\n\n";
    for (i = 0; i < N; i++) cout << setw(4) << d[i];
    cout << endl;
    return 0;
}
```