Wstęp do programowania

Dr Anna Muranova

Semestr zimowy 2023/2024, UWM w Olsztynie

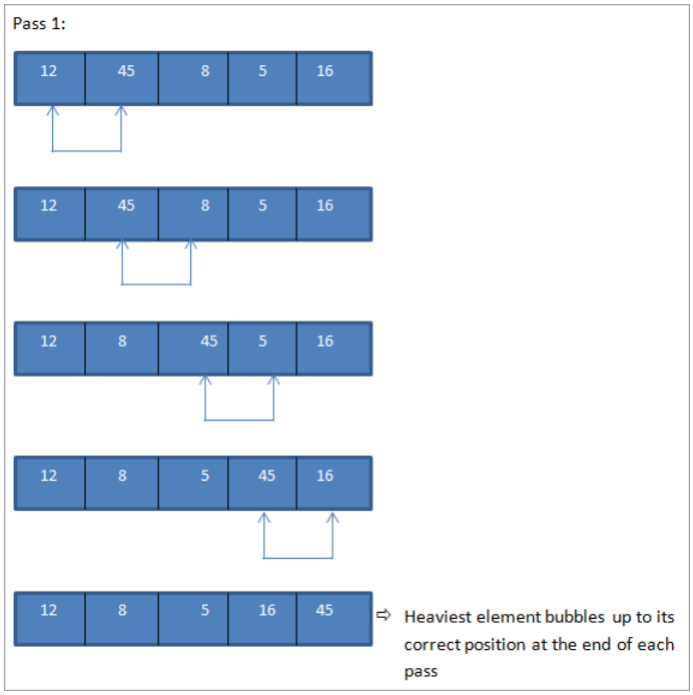
8. Zajęcie (sortowanie)

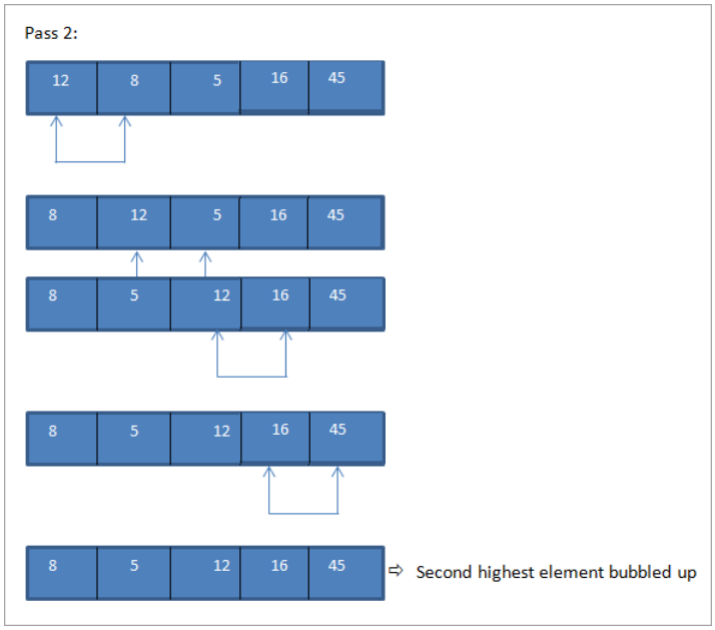
8.1 Algorytmy sortowania

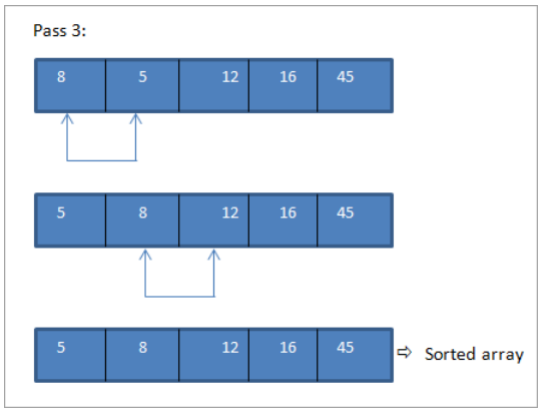
- Sortowanie bąbelkowe (Bubble sort)
- Sortowanie przez selekcję albo przez wybór (selection sort)
- Sortowanie przez wstawianie (insertion sort)
- Sortowanie szybkie (Quicksort)

8.1.1 Sortowanie bąbelkowe

W algorytmie tym w celu posortowania n elementów wykonujemy $n-1$ przebiegów. W każdym przebiegu zaczynamy od początku zbioru, bierzemy dwa elementy i zamieniamy je miejscami tak, aby większy był ‚wyżej' (lub z większym indeksem itp). W pierwszym kroku robimy to dla elementów 1 i 2, w drugim kroku dla 2 i 3 itd. Każdy przebieg ‚wyciąga' do góry kolejny element w kolejności, po wykonaniu wszystkich przebiegów nasz zbiór jest posortowany.





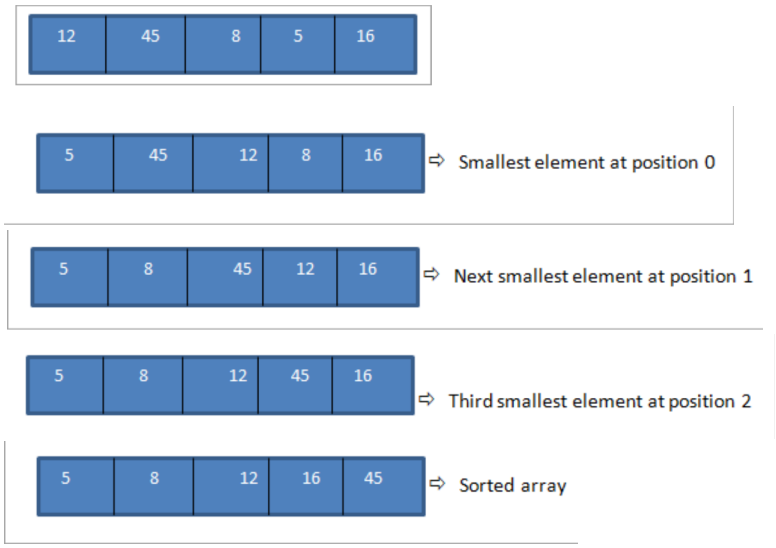



```
#include<iostream>
using namespace std;
int main()
{
    int i, j, temp;
    int a[10] = { 10,2,0,43,12, -3, 8,9,13, 1 };
    cout << "Input list ...\n";
    for (i = 0; i < 10; i++) {
        cout << a[i] << "\t";
    }
    cout << endl;
    for (i = 0; i < 10; i++) {
        for (j = 0; j < 9; j++)
        {
            if (a[j] > a[j+1]) {
                temp = a[j];
                a[j] = a[j+1];
                a[j+1] = temp;
            }
        }
        //for (int k = 0; k < 10; k++) {
```

```
        // cout << a[k] << "\t";
        //}
        //cout << endl;
}
    cout << "Sorted Element List ...\n";
    for (i = 0; i < 10; i++) {
        cout << a[i] << "\t";
    }
    return 0;
}
```

8.1.2 Sortowanie przez wybór

Strategia sortowania przez wybór jest bardzo prosta. Szukamy najmniejszego elementu w zbiorze i zamieniamy go z elementem stojącym na pozycji pierwszej. Następnie szukamy znowu elementu najmniejszego w zbiorze pominiętym o pierwszy element i wstawiamy go na pozycję drugą. Czynności powtarzamy do momentu otrzymania jednoelementowego podzbioru.



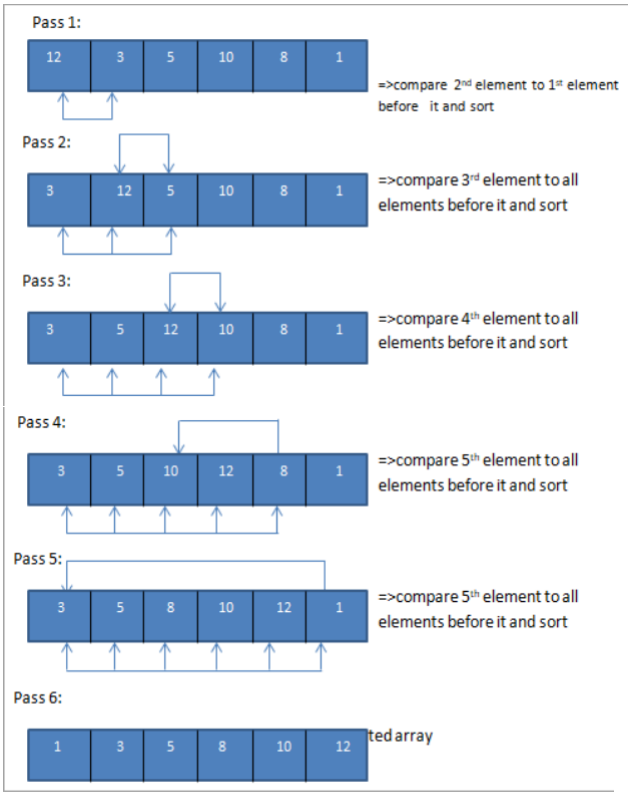

```
#include <iostream>
using namespace std;
int main() {
    int i, j, temp;
    int a[10] = { 10,2,0,43,12, -3, 8,9,13, 1 };
    for(i = 0; i < 10; i++) {
        for(j = i + 1; j < 10; j++) {
            if(a[j] < a[i]) {
                temp = a[i];
                a[i] = a[j];
                a[j] = temp;
                    }
                }
        //for (int k = 0; k < 10; k++) {
        //cout << a[k] << "\t";
```

```
        //}
        //cout << endl;
        }
    cout << "Sorted Element List . . . \ n";
    for(i = 0; i < 10; i++) {
        cout << a[i] << " \t ";
    }
    return 0;
}
```

8.1.3 Sortowanie przez wstawianie (insertion sort)

Schemat działania algorytmu:

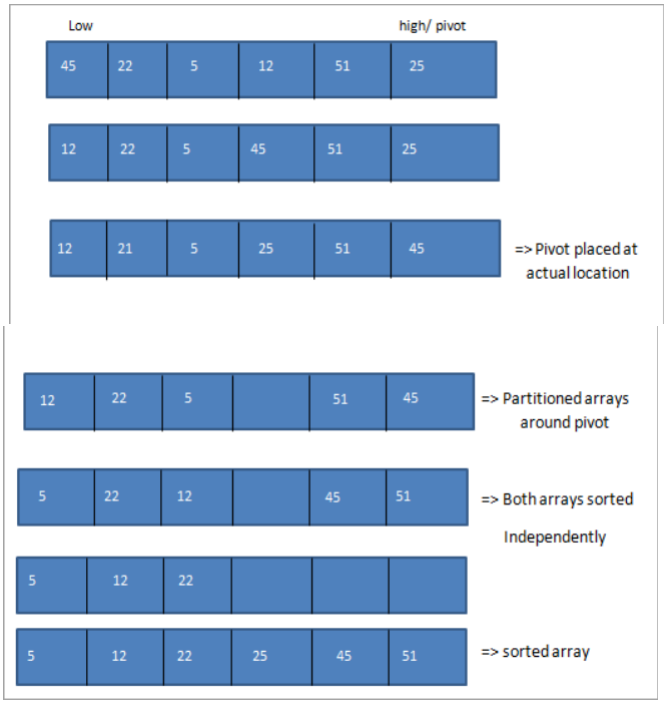
- Posortuj pierwsze k elementowi tablicy.
- Zmień pozycje $(k+1)$-ego elementu tak, zęby mieć pierwsze $k+1$ elementowi posortowane.
- Rob dopóki wszystkie n elementów nie będą posortowany.



8.1.4 Szybkie sortowanie

Algorytm ten jest najpopularniejszy, jeśli chodzi o ilość zastosowań, ponieważ w ogólnym i uśrednionym przypadku jest on najbardziej optymalny, a jednocześnie prosty w implementacji. Jego wadą jest negatywny przypadek, kiedy to złożoność wynosi n^2 .

Algorytm jest rekursywny, tj. odwołuje się do samego siebie i jest przykładem podejścia dziel i zwyciężaj (divide and conquer). W każdym kroku ze zbioru elementów wybieramy dowolny element (nazywany pivot; może to być po prostu pierwszy z brzegu lub losowy, dla nieuporządkowanych danych nie ma to znaczenia) a następnie dzielimy pozostałe elementy na dwie grupy — te mniejsze od wybranego elementu (‚przed nim') oraz większe od niego (‚za nim'). Każdą z tych grup sortujemy w ten sam sposób.




```
#include <iostream>
using namespace std;

void quickSort(int arr[], int start, int end)
{
    // base case
    if (start >= end)
    return;

    //find proper pivot index
    int pivotIndex = end;
    int count = 0;
    for (int i = start; i <= end; i++)
        if (arr[i] > arr[pivotIndex])
            count++;
    pivotIndex = end - count;

    //put pivot on its place
    int temp = arr[end];
    arr[end] = arr[pivotIndex];
    arr[pivotIndex] = temp;
```

```
    //organize elements around pivot
    int i = start, j = end;

    while (i < pivotIndex && j > pivotIndex) {

        while (arr[i] <= arr[pivotIndex]) i++;


        while (arr[j] >= arr[pivotIndex]) j--;


        if (i < pivotIndex && j > pivotIndex) {
            temp = arr[j];
            arr[j] = arr[i];
            arr[i] = temp;
            i++;
            j--;

        }
    }



    // Sorting the left part
    quickSort(arr, start, pivotIndex - 1);

    // Sorting the right part
    quickSort(arr, pivotIndex + 1, end);
}


int main()
{

    int arr[10] = { 9, 3, 4, 2, 1, 8 , -3, 5, 7, 8};
    int n = 10;

    quickSort(arr, 0, n - 1);


    for (int i = 0; i < n; i++) {
        cout << arr[i] << " ";
    }

    return 0;
}
```

8.2 Ćwiczenia

W nagłówku można używać tylko:

```
#include <iostream>
using namespace std;
```

Ćwiczenie 8.1. Przeczytaj i zrozumiej podany algorytmy i kody sortowania.

(a) Zmień kod sortowania bąbelkowego tak, żeby nie sprawdzał już posortowane elementy.

(b) Zmień kod sortowania przez wybór tak, żeby zamieniał największy element z ostatnim, drugi największy z przedostatnim i td.

Ćwiczenie 8.2. Zaimplementuj sortowanie przez wstawianie.

Ćwiczenie 8.3. Zmień kod szybkiego sortowanie tak, żeby pivotem był pierwszy element każdej części.