P O L I T E C H N I K A R Z E S Z O W S K A
im. Ignacego Łukasiewicza
WYDZIAŁ MATEMATYKI I FIZYKI STOSOWANEJ

STACH KACPER

Projekt #2 – porównanie algorytmów sortujących

kierunek studiów: Inżynieria i analiza danych

Rzeszów 2022

SPIS TREŚCI

1. Sortowanie grzebieniowe

1. Podstawy teoretyczne sortowania grzebieniowego

Sortowanie Grzebieniowe opiera się na idei sortowania bąbelkowego. Różnica jednak polega na wykonywaniu zamian w inny sposób - staramy się w każdej iteracji przesunąć wszystkie duże elementy na sam koniec. Można to porównać do czesania z grzebieniem. Początkowo rozczesujemy włosy grzebieniem z dużym odstępem pomiędzy ząbkami - w ten sposób będzie łatwiej i szybciej rozczesać pojedyncze grupki.

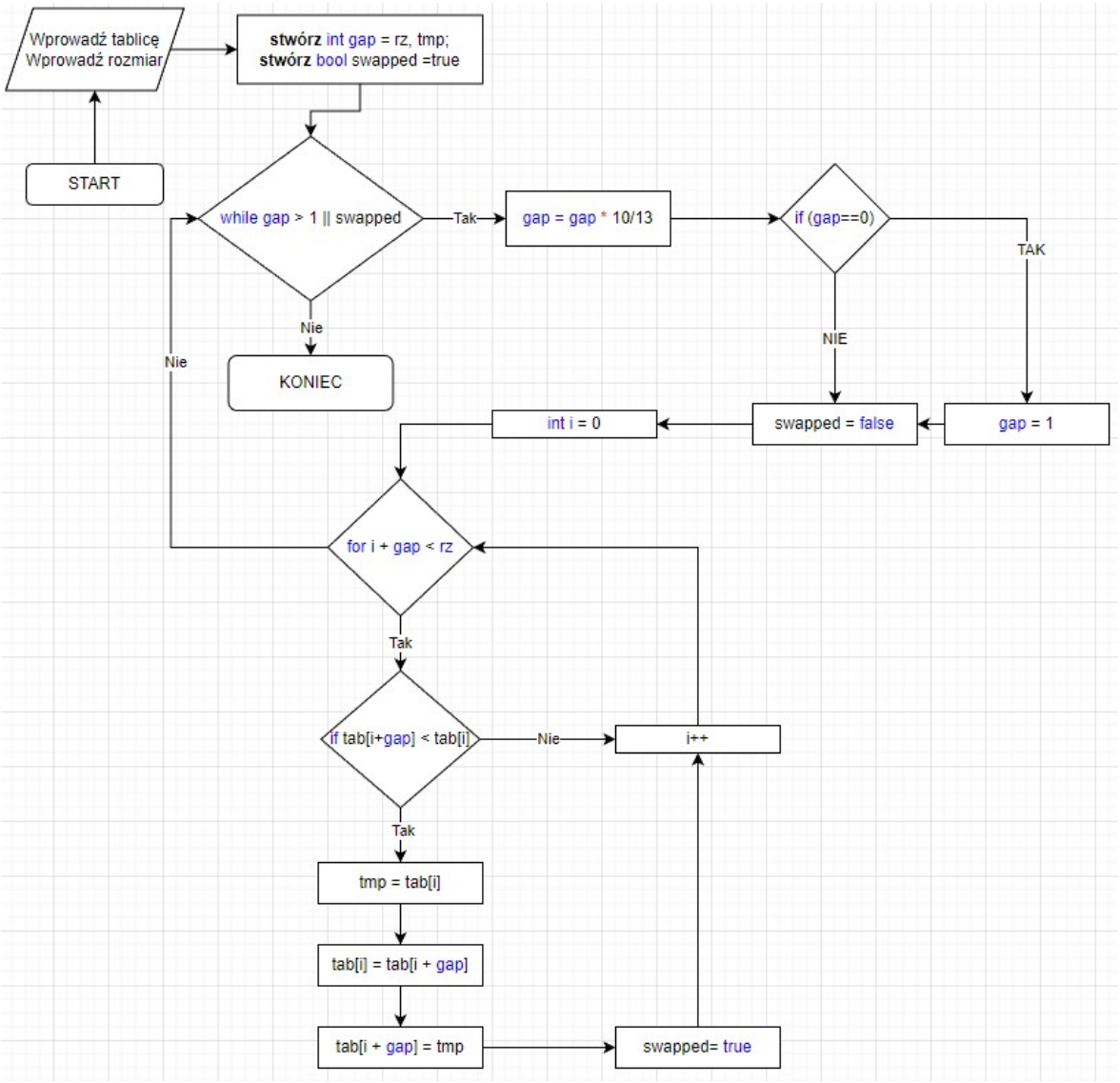
Algorytm sortowania Grzebieniowego zawiera element empiryczny - czyli wyliczony doświadczalnie. Jest to współczynnik *w:*=**1.3**. Nie ma on znaczenia dla poprawności posortowania listy, ale ma znaczący wpływ na szybkość wykonywania algorytmu. W ten sposób zwykłe sortowanie bąbelkowe poprawiamy prawdopodobnie do złożoności . Niemniej jest on prawdopodobnie wolniejszy od sortowania szybkiego.

2. Kod funkcji combSort()

```
void combSort(int *tab, int rz)
{
    int gap = rz, tmp;
    bool swapped = true;
    while (gap > 1 || swapped)
     {
         // jeśli gap = 1 i nie dokonano zamiany - wyjście z petl
       gap = gap * 10 / 13;
       if(gap==0)
             gap=1;
       swapped = false;
       for ( int i = 0; i + gap < rz; ++i )
       {
           // wykonuj od 0 do ostatniego elementu tablicy
          if ( tab[i + gap] < tab[i] )
             {
             // porównanie elementów odległych o rozpiętość
             tmp = tab[i];
             // zamiana elementów
             tab[i] = tab[i + gap];
             tab[i + gap] = tmp;
             swapped = true;
            }
       }
    }
}
```

Rysunek 1 - kod funkcji combSort

3. Schemat blokowy sortowania grzebieniowego



Rysunek 2 - schemat blokowy sortowania grzebieniowego

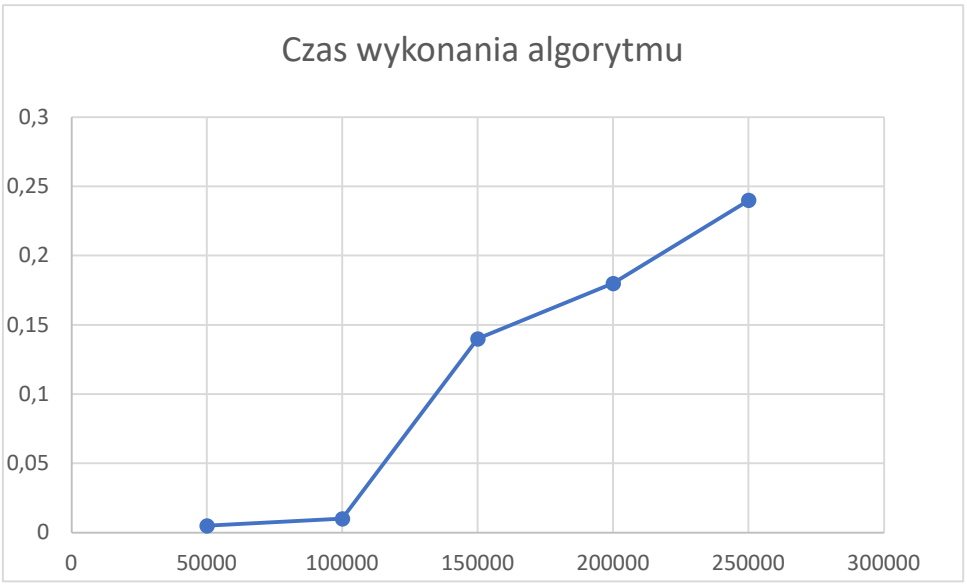
4. Pseudokod funkcji sortowania grzebieniowego

```
Pseudokod funkcji sortującej grzebieniowo

Funkcja void combSort()
stwórz zmienną int gap = rz, tmp;
stwórz bool swapped = true;

dopóki gap > 1 i swapped
{
    gap = gap * 10/13;
    jeśli gap ==0
        zamian gap na 1
    swapped = false;
    dla (int i = 0; i + gap <rz; ++i)
    {
        jeśli (tab[i +gap] < tab[i])
        stwórz zmienną tmp = tab[i];

        tab[i] = tab[i + gap];
        tab[i + gap] = tmp;
        swapped = true;
    }
}
```

Rysunek 3 - pseudokod sortowania grzebieniowego

5. Wykres złożoności obliczeniowej sortowania grzebieniowego.

Wykres przedstawia zależność czasu od ilości rekordów w tabeli sortowanej. Czas przedstawiony jest w sekundach.



Rysunek 4 - wykres sortowania grzebieniowego

1. Średnia złożoność obliczeniowa.

Średnią złożoność obliczeniowa algorytmu sortowania grzebieniowego można wyrazić wzorem:

$$\Omega(n^2/2^p)$$

2. Złożoność obliczeniowa w przypadku pesymistycznym.

Złożoność obliczeniową w przypadku pesymistycznym możemy wyrazić poniższym wzorem:

$$O(n^2)$$

3. Złożoność obliczeniowa w przypadku optymistycznym:

Złożoność obliczeniową w przypadku optymistycznym możemy wyrazić wzorem:

$$\theta(n \log n)$$

2. Sortowanie przez wstawianie

1. Podstawy teoretyczne sortowania przez wstawianie

Sortowanie przez wstawianie jest jednym z najprostszych algorytmów sortowania. Został on opracowany w sposób, który może odzwierciedlać np. ustawianie kart – kolejne elementy są ustawiane na odpowiednie miejsca docelowe.

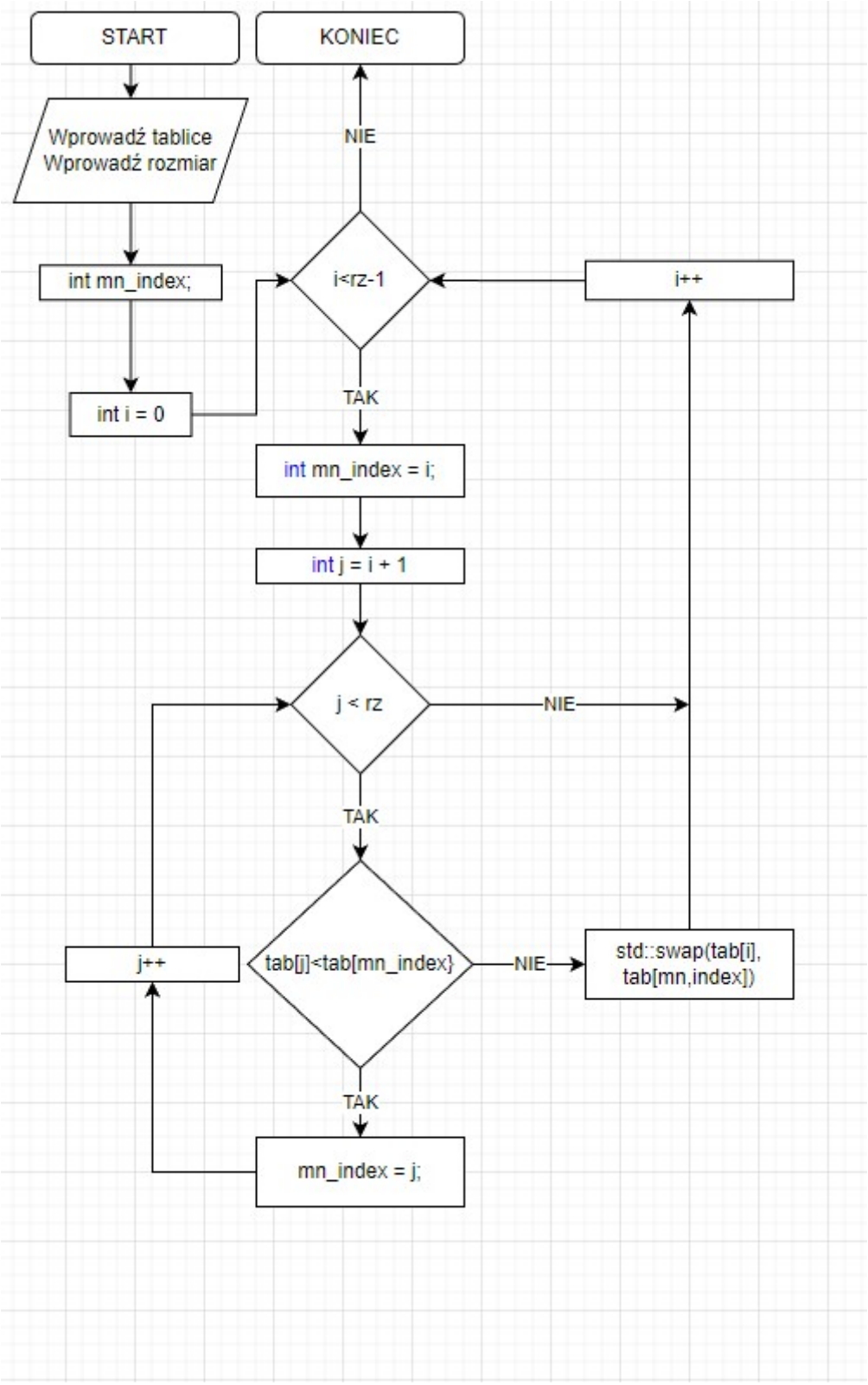
Cechuje go wysoka wydajność dla zbiorów o stosunkowo niewielkiej wielkości. Mimo to, jest znacznie mniej wydajny od innych algorytmów sortujących, chociażby wyżej przedstawionego sortowania grzebieniowego.

2. Kod funkcji selection_sort()

```
void selection_sort(int *tab,int rz)
//n - ilość elementów do posortowania
{
int mn_index;
//zmienna pomocnicza przechowująca indeks komórki
//z minimalną wartością
  for(int i=0;i<rz-1;i++)
  {
    mn_index = i;
    for(int j=i+1;j<rz;j++)
    //petla wyszukuje najmniejszy
    //element w podzbiorze nieposortowanym
    if(tab[j]<tab[mn_index])
      mn_index = j;

    //zamiana elementu najmniejszego
    //w podzbiorze z pierwszą pozycją nieposortowaną
    std::swap(tab[i], tab[mn_index]);
  }
}
```

Rysunek 5 - kod funkcji selection_sort

3. Schemat blokowy sortowania przez wstawianie



Rysunek 6 - schemat blokowy sortowania przez wstawianie

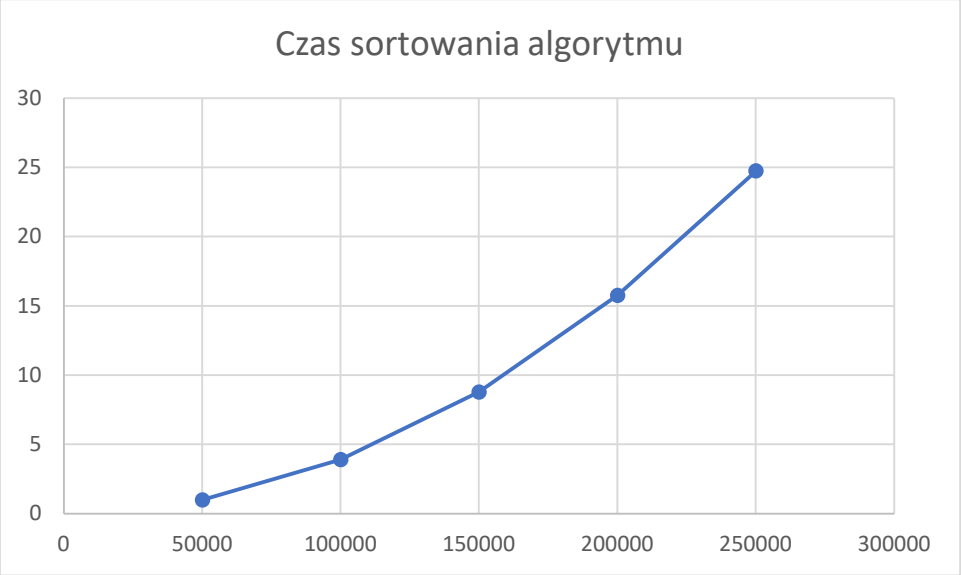
4. Pseudokod algorytmu sortowania przez wstawianie

```
Funkcja selection_sort Wprowadz int *tab, int rz
{
    wprowadz int mn_index;

    dla int i = 0; i<rz-1;i++
        {
            przypisz i do mn_index;
            dla int j=i+1;j<rz;j++

            jesli tab[j]>tab[mn_index
                przypisz j do mn_index
        }
        zamien najmniejszy element tab[i],
        z  tab[mn_index]
```

Rysunek 7 - pseudokod sortowania przez wstawianie

5. Wykres złożoności czasowej sortowania przez wybieranie

Wykres przedstawia zależność czasu od ilości rekordów w tabeli sortowanej. Czas przedstawiony jest w sekundach.



Rysunek 8 - wykres sortowania przez wstawianie

1. Średnia złożoność obliczeniowa.

Średnią złożoność obliczeniowa algorytmu sortowania przez wstawianie można wyrazić wzorem:

$$O(n^2)$$

2. Złożoność obliczeniowa w przypadku pesymistycznym.

Złożoność obliczeniową w przypadku pesymistycznym możemy wyrazić poniższym wzorem:

$$O(n)$$

3. Złożoność obliczeniowa w przypadku optymistycznym:

Złożoność obliczeniową w przypadku optymistycznym możemy wyrazić wzorem:

$$O(n^2)$$

3. Wnioski

Analizując obie metody sortowania, można dojść do następujących wniosków:

- Metoda sortowania grzebieniowego jest znacznie wydajniejsza od metody sortowania poprzez wstawianie,
- Metoda sortowania przez wstawianie znacznie traci na efektywności, gdy operujemy na dużych ilościach danych.
- Udało się poprawnie sprawdzić złożoność czasową algorytmów na podstawie działania programu
- Udało się poprawnie zaimplementować odczyt z pliku oraz zapis do pliku tekstowego.

SPIS RYSUNKÓW