SPRAWOZDANIE

Autor: Michał Deć

Opiekun pracy: **dr inż. prof. PRz Mariusz Borkowski**

Rzeszów, rok 2021

1. Wstęp

Program ma za zadanie zaimplementować dwa oddzielne algorytmy sortujące, takie jak sortowanie przez wstawianie oraz sortowanie grzebieniowe, a następnie porównać ich złożoność czasową oraz złożoność obliczeniową.

2. Opis podstaw zagadnienia

Program jest podzielony na pięć funkcji. Funkcja main zawiera tablicę wypełnioną liczbami pseudolosowymi, która jest przekazywana do funkcji zawierającej algorytm sortowania poprzez wstawianie oraz do funkcji z algorytmem sortującym grzebieniowo. Blok main wywołuje te dwie funkcje, a następnie porównuje czasy ich wykonywania dla danych próbek ilości elementów na podstawie, których można sporządzić ich wykresy złożoności czasowej.

3. Podział programu

Program podzielony jest na pięć funkcji:

- **int main**
- **void sortowanie_grzebieniowe**
- **void sortowanie_przez_wstawianie**
- **void init**
- **double getCurrentValue**

4. Ograniczenia programu

Tablica na podstawie, której przeprowadza się testy wypełniona jest liczbami pseudolosowymi od 0 do 100000. Program mierzący złożoność obliczeniową oraz czasową wykonywania się algorytmów nie uwzględnia liczb ujemnych lub zmiennoprzecinkowych.

5. Opis teoretyczny algorytmów sortujących

5.1 Sortowanie przez wstawianie

Jest to jeden z najprostszych algorytmów sortowania. Zasada działania polega na tym, że kolejno elementy wejściowe są ustawiane na odpowiednie miejsca docelowe. Typ tego sortowania jest wydajny dla zbiorów dla niewielkiej liczby elementów.

Zalety sortowania przez wstawianie:

- **Wydajny dla danych wstępnie posortowanych.**
- **Wydajny dla zbiorów o niewielkiej ilości elementów.**
- **Stabilny.**

5.2 Sortowanie grzebieniowe

Sortowanie to opiera się na idei sortowania bąbelkowego, z różnicą, że wszystkie duże elementy w każdej iteracji przesuwane są na sam koniec. Podczas każdej iteracji wyliczana jest rozpiętość, która początkowo odpowiada ilości elementów w tablicy. Na początku iteracji rozpiętość ta dzielona jest przez współczynnik wynoszący **1.3**. Pozostała część ułamkowa jest odrzucana. Podczas działania algorytmu porównywane są ze sobą pary elementów, które są odległe o rozpiętość. Gdy rozpiętość spada do 1 algorytm ten postępuje w identyczny sposób tak jak sortowanie bąbelkowe. Tylko wtedy metoda ta może określić, czy dane są już posortowane lub nie. Do sprawdzenia tego można posłużyć się dodatkową zmienną typu bool.

Główne cechy sortowania grzebieniowego:

- **Zawiera element empiryczny (współczynnik 1.3 wyznaczony doświadczalnie).**
- **Oparty na metodzie bubblesort (sortowanie bąbelkowe).**
- **Metoda ta potrzebuje dodatkowej zmiennej w każdej iteracji, która sprawdzi czy wykonała się jakakolwiek zmiana, gdy rozpiętość wynosi 1.**

6. Schemat blokowy oraz pseudokod

6.1 Sortowanie przez wstawianie

6.1.1 Pseudokod

Krok 1: Wczytaj liczby z tablicy oraz ich ilość.

Krok 2: Zadeklaruj zmienne ‘i’ oraz ‘j’ służące do sterowania pętlami oraz zmienną ‘x’ służącą do przechowywania wybranego elementu ze zbioru danych.

Krok 3: Sprawdź, czy ‘j’ jest większe lub równe 1, jeśli nie jest to zakończ sortowanie danych.

Krok 4: Przechowaj w zmiennej ‘x’ wybrany element ze zbioru danych oraz za ‘i’ przyjmij ‘j+1’.

Krok 5: Sprawdź czy zmienna ‘i’ jest mniejsza bądź równa liczbie elementów. Jeśli nie to przejdź do krok 9.

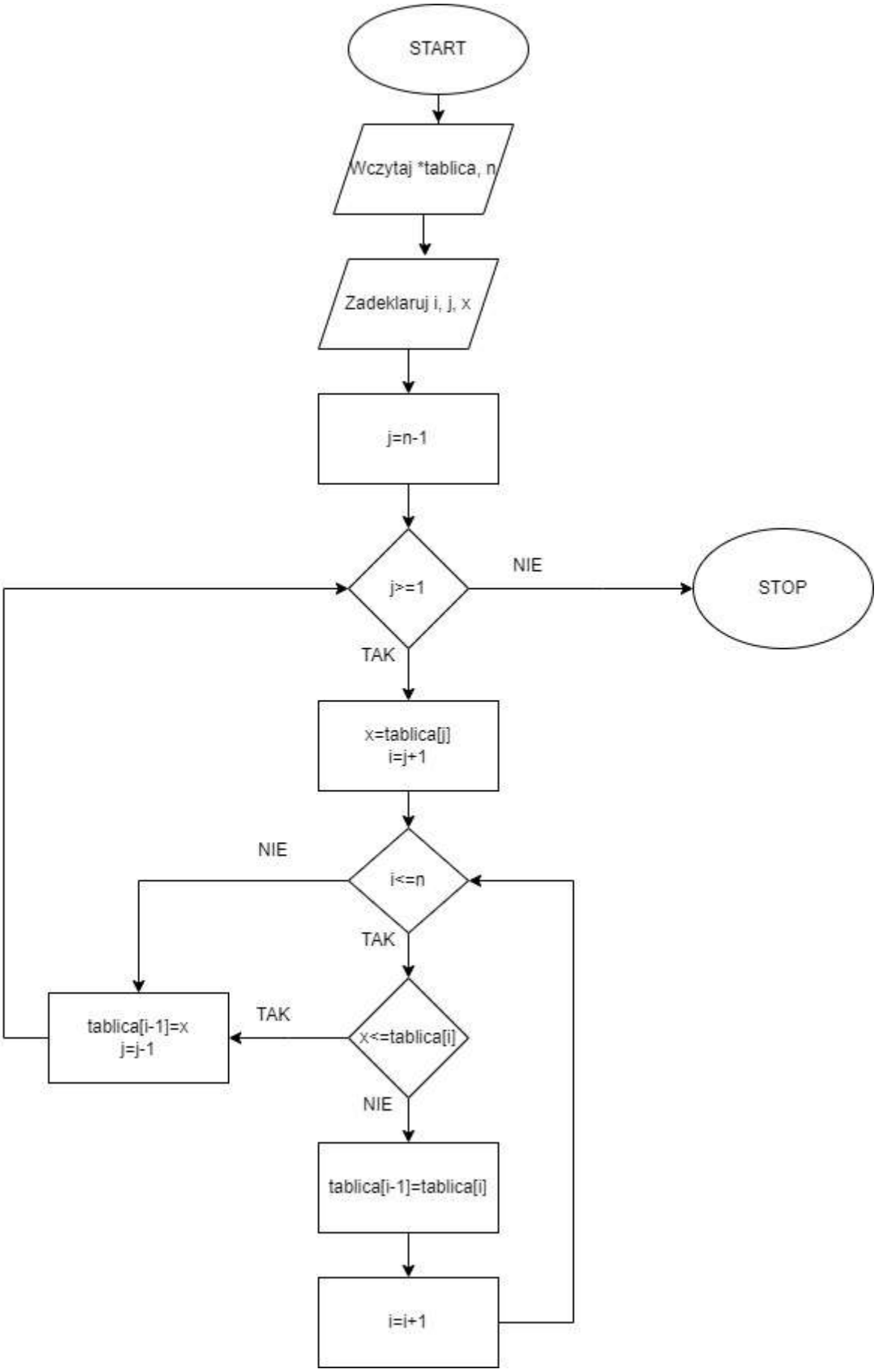
Krok 6: Sprawdź czy zmienna ‘x’ jest mniejsza bądź równa od elementu ze zbioru danych. Jeśli tak to przejdź do krok 9.

Krok 7: Do poprzedniego elementu z zbioru danych przypisz aktualny element.

Krok 8: Inkrementuj zmienną ‘i’ oraz wróć do do kroku 5.

Krok 9: Do poprzedniego elementu z zbioru danych przypisz zmienną ‘x’ oraz dekrementuj zmienną ‘j’, następnie wróć do krok 3.

6.1.2 Schemat blokowy


START
Wczytaj *tablica, n
Zadeklaruj i, j, x
j=n-1
j>=1
NIE
STOP
TAK
x=tablica[j]
i=j+1
NIE
i<=n
TAK
tablica[i-1]=x
j=j-1
TAK
x<=tablica[i]
NIE
tablica[i-1]=tablica[i]
i=i+1

6.2 Sortowanie grzebieniowe

6.2.1 Pseudokod

Krok 1: Wczytaj liczby z tablicy oraz ilość elementów.

Krok 2: Zadeklaruj zmienną ‘gap’ i przypisz do niej ilość elementów oraz zmienną typu bool ‘replace’ oraz przypisz jej wartość true.

Krok 3: Sprawdź czy zmienna ‘gap’ jest większa od 1 lub czy zmienna ‘replace’ jest prawdziwa. Jeśli nie to zakończ proces sortowania danych.

Krok 4: Podziel zmienną ‘gap’ przez 1,3.

Krok 5: Sprawdź czy zmienna ‘gap’ równa się zero. Jeśli tak to przypisz jej wartość 1 i wróć do krok 3. Jeśli nie to kontynuuj do krok 6.

Krok 6: Przypisz zmiennej ‘replace’ wartość false.

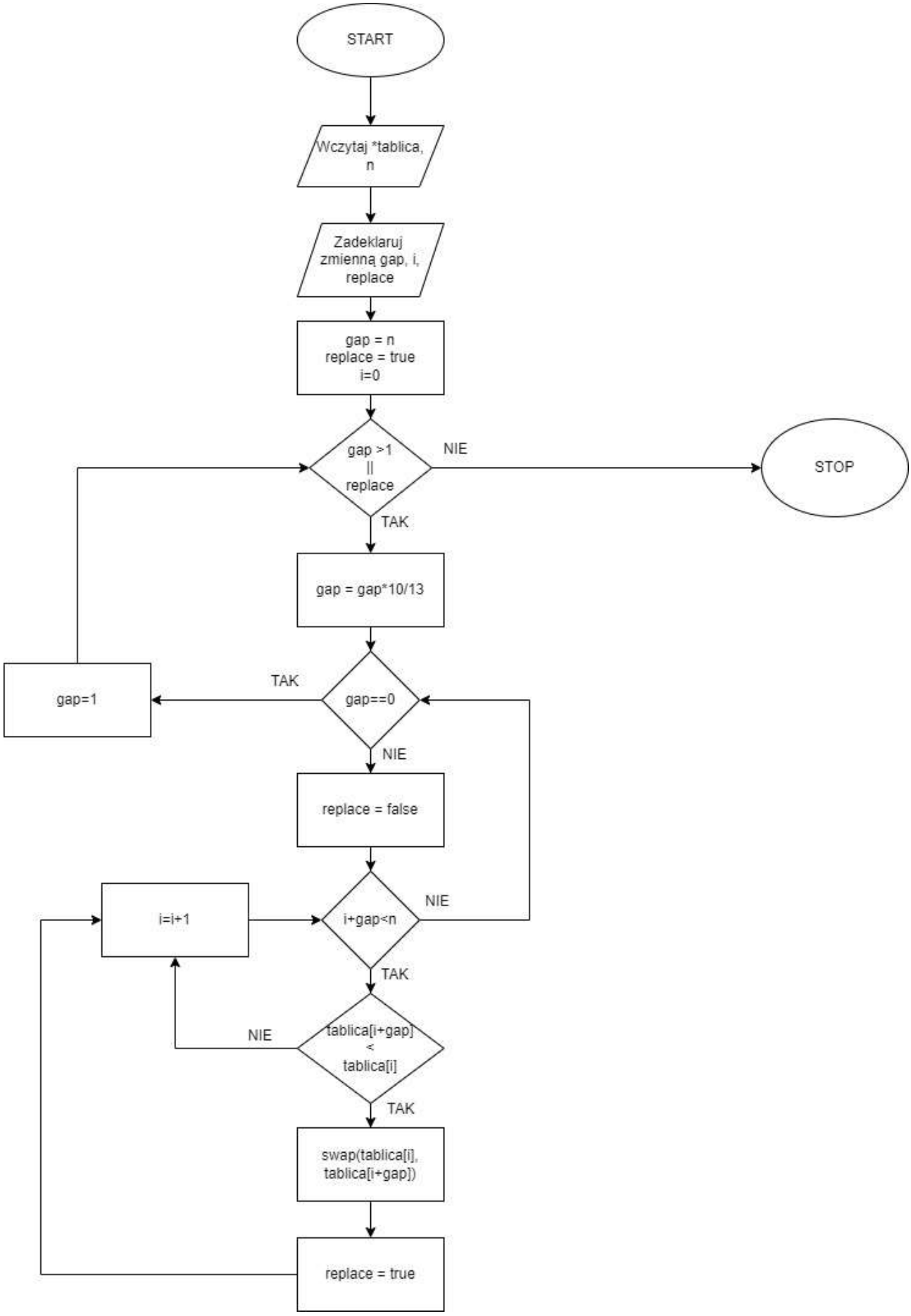
Krok 7: Zadeklaruj zmienną ‘i’ wynoszącą zero w pętli typu for i sprawdzaj czy ‘i + gap’ jest mniejsze od ilości elementów w zbiorze danych oraz inkrementuj zmienną ‘i’ po każdym przejściu pętli. Po wykonaniu się pętli wróć do krok 5.

Krok 8: Sprawdź czy wartość w zbiorze danych ‘i + gap’ jest mniejsza od aktualnej wartości w zbiorze danych. Jeśli nie to wróć do krok 7. Jeśli tak to kontynuuj do krok 9.

Krok 9: Zamień wartości w zbiorze danych ‘i + gap’ z aktualną wartością z tablicy.

Krok 10: Przypisz zmiennej ‘replace’ wartość true oraz wróć do krok 7.

6.2.2 Schemat blokowy


START
Wczytaj *tablica, n
Zadeklaruj zmienną gap, i, replace
gap = n
replace = true
i=0
gap >1 || replace
NIE
STOP
TAK
gap = gap*10/13
gap==0
TAK
gap=1
NIE
replace = false
i=i+1
i+gap<n
NIE
TAK
tablica[i+gap] < tablica[i]
NIE
TAK
swap(tablica[i], tablica[i+gap])
replace = true

7. Złożoność obliczeniowa algorytmów sortujących

7.1 Opis teoretyczny

Złożoność obliczeniowa ma na celu określenie potrzebnych zasobów komputera do rozwiązania danego problemu. Na cel testów mierzących procentowe wykorzystanie CPU podczas wykonywania się algorytmów sortujących wykorzystamy procesor typu Intel Core i7-4790k o taktowaniu wynoszącym 4.00GHz

7.2 Opis implementacji rozwiązania

Do przeprowadzenia testów wykorzystamy bibliotekę ‘psapi’. Funkcja **void init()** wywołana w bloku main pobiera informacje o systemie operacyjnym oraz o procesorze. Funkcja **double getCurrentValue** mierzy aktualne procentowe wykorzystanie procesora podczas wykonywania się algorytmu sortującego.

7.3 Testy wykorzystania procesora

7.3.1 Sortowanie przez wstawianie

Ilość elementów	Procentowe wykorzystanie procesora
5000	5
10000	7
20000	10
50000	14
75000	18
100000	23

7.3.2 Sortowanie grzebieniowe

Ilość elementów	Procentowe wykorzystanie procesora
5000	8
10000	11
20000	15
50000	19
75000	24
100000	29

7.4 Wniosek

Na podstawie powyższych dwóch tabel można stwierdzić, iż sortowanie grzebieniowe zużywa większe ilości zasobów podzespołów komputera, niż sortowanie przez wstawianie. Jest to spowodowane tym, że sortowanie grzebieniowe jest lepszą klasą złożoności czasowej, niż algorytm sortujący poprzez wstawianie. Z racji tego algorytm, który sortuje grzebieniowo potrzebuje większej mocy obliczeniowej oraz mocniej obciąża sprzęt komputerowy. Sortowanie grzebieniowe układa elementy dużo szybciej, niż sortowanie przez wstawianie, co przekłada się na większe zużycie zasobów komputera, w odróżnieniu od funkcji sortującej poprzez wstawianie.

8. Złożoność czasowa algorytmów sortujących

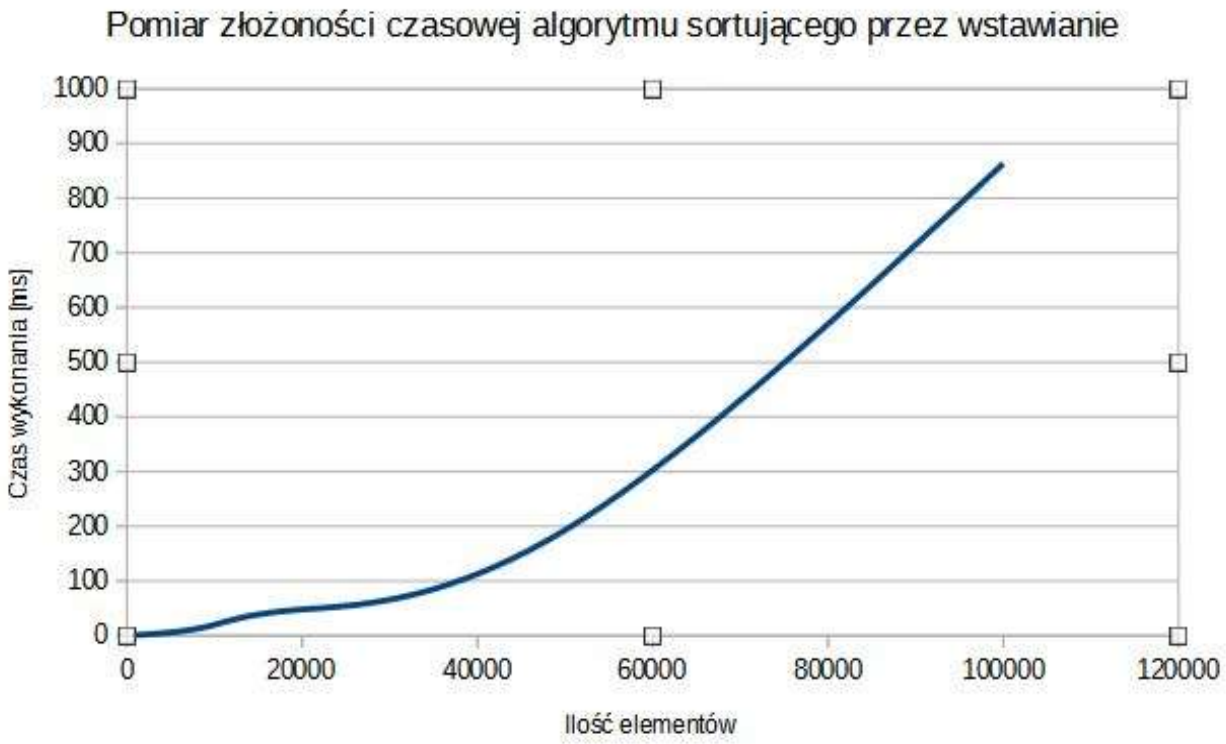
8.1 Wstęp

Złożoność czasowa algorytmów ma na celu opisać ilość potrzebnego czasu do zrealizowania danego problemu. Do pomiaru złożoności czasowej wykorzystamy algorytm sortujący grzebieniowo oraz poprzez wstawianie.

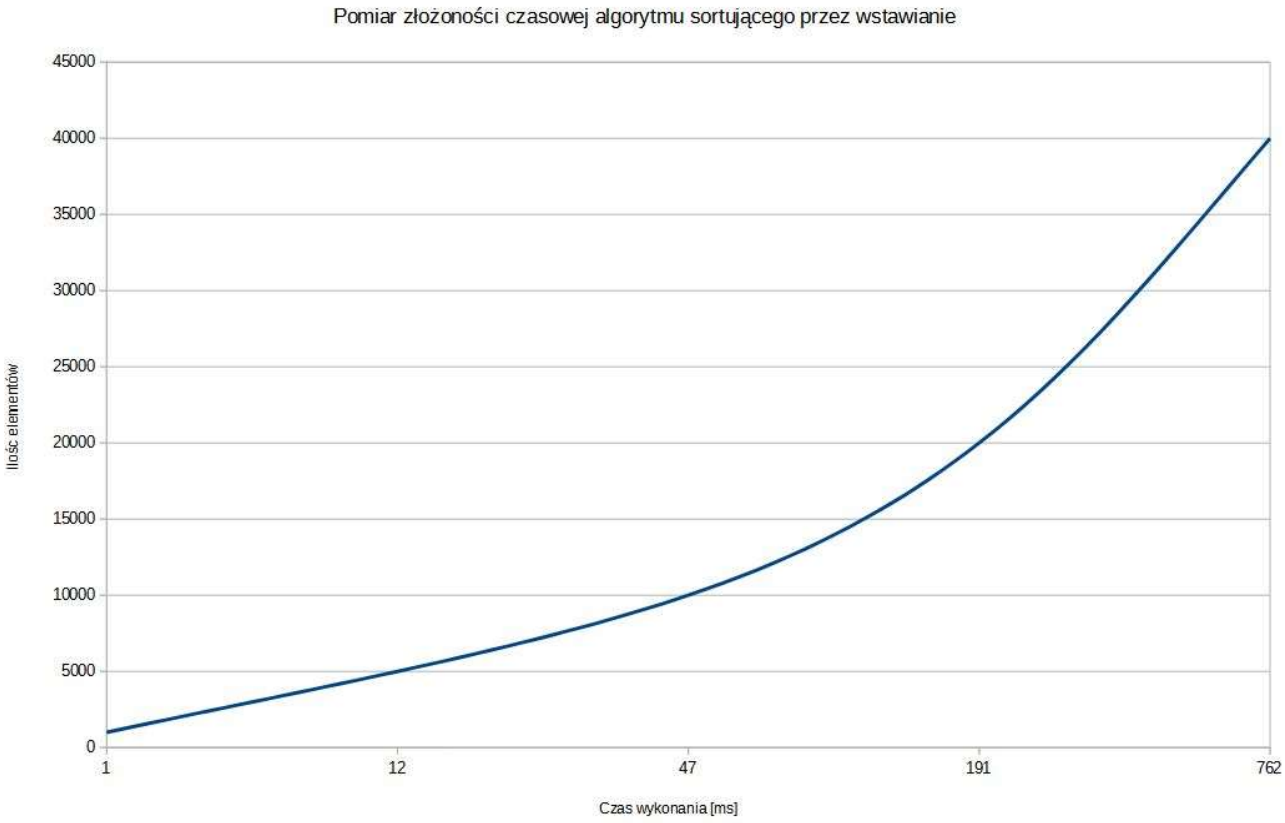
8.2 Sortowanie przez wstawianie

8.2.1 Wykres pomiaru złożoności czasowej

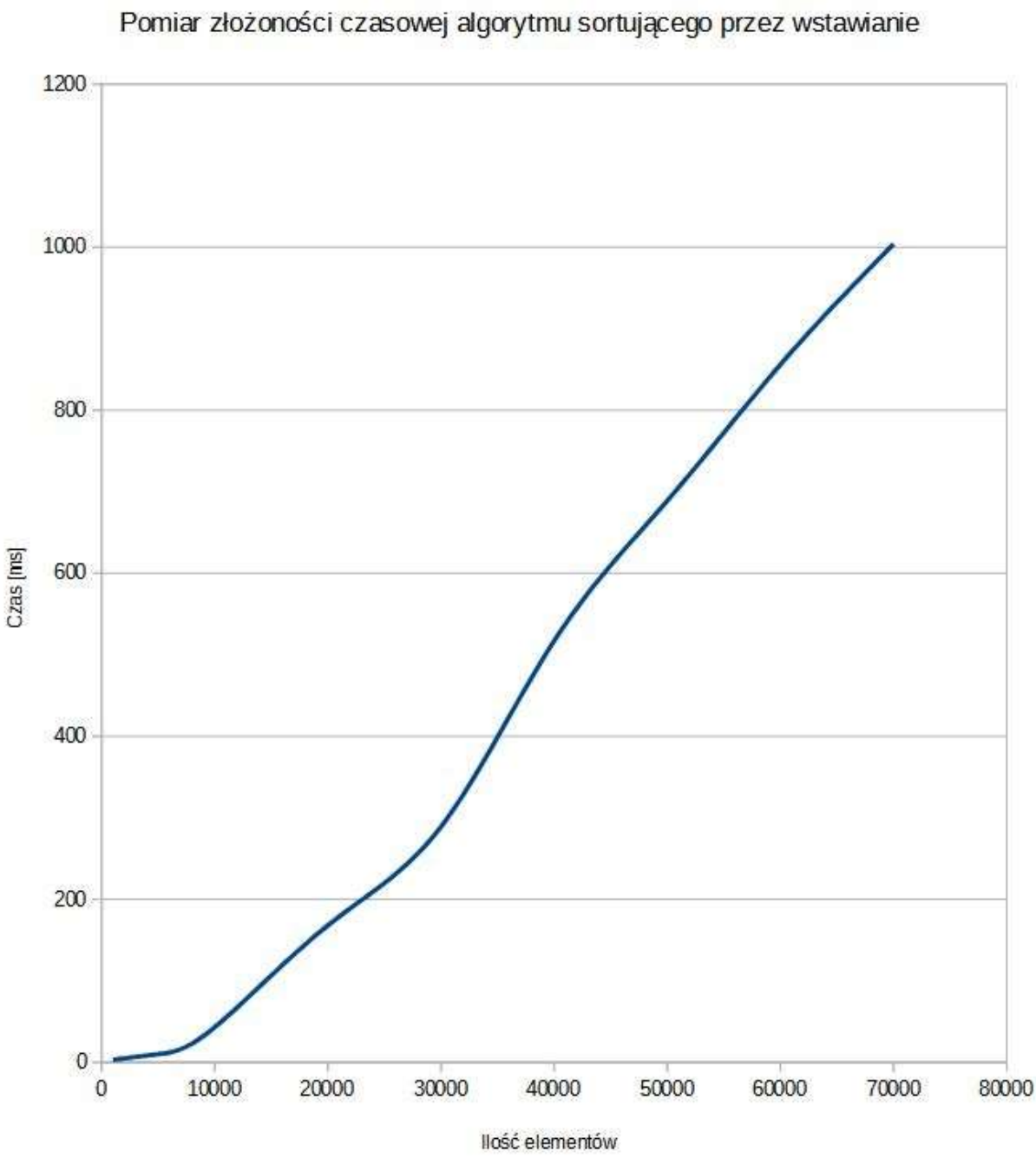
- **Przypadek optymistyczny**



- **Przypadek pesymistyczny**



- **Przypadek oczekiwany**



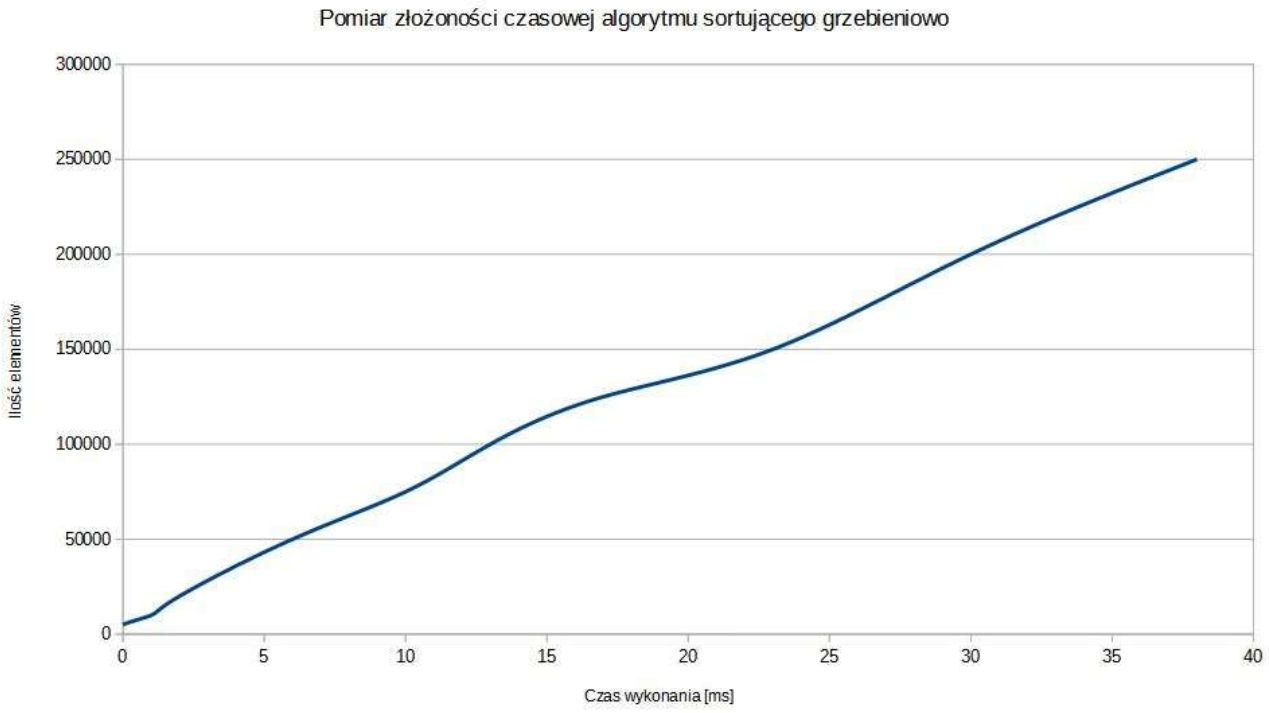
8.2.2 Wniosek

Sortowanie przez wstawianie jest algorytmem o złożoności czasowej $O(n^2)$. Algorytm ten ma złożoność kwadratową. W przypadku sortowania przez wstawianie bardzo ważne jest wstępne posortowanie danych oraz ich ilość. Algorytm ten nadaje się, jeśli mamy optymistyczną wersję, gdy ilość danych sięga rzędu 5000 oraz ich rozpiętość nie jest wielka. W przypadku pesymistycznej wersji, gdy mamy do czynienia z większymi zbiorami danych typu 50000 oraz ich rozpiętość jest wielka czas wykonania sortowania wynosi już w przybliżeniu około sekundy i cały czas rośnie. Algorytm ten przestaje być wydajny dla dużych zbiorów danych o większej rozpiętości. Dla sortowania przez wstawianie należy pamiętać o tym, że najważniejsze jest wstępne posortowanie danych oraz ich rozpiętość.

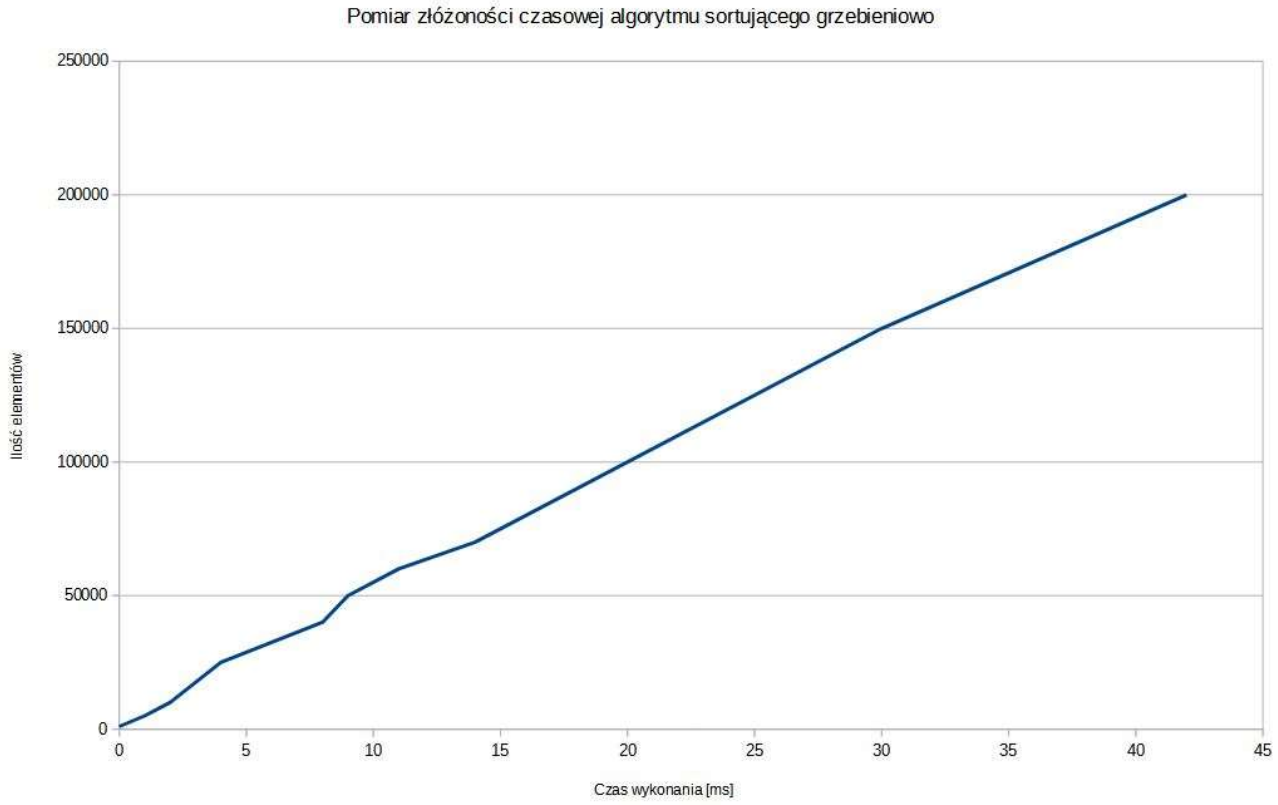
8.3 Sortowanie grzebieniowe

8.3.1 Wykres pomiaru złożoności czasowej

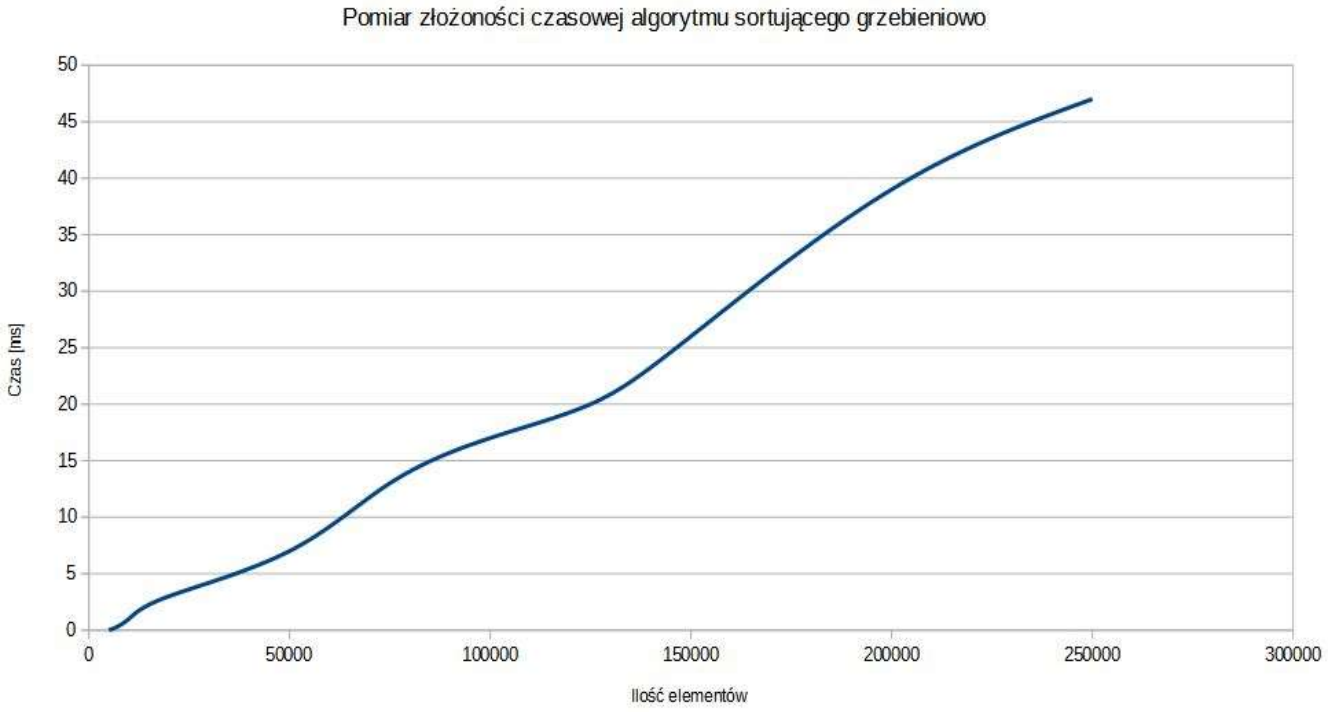
- **Przypadek optymistyczny**



- **Przypadek pesymistyczny**



- **Przypadek oczekiwany**



8.3.2 Wniosek

Sortowanie grzebieniowe jest algorytmem o złożoności czasowej O(n*log(n)). Algorytm ten ma złożoność czasową liniowo-logarytmiczną. Czas rozwiązywania problemu jest wprost proporcjonalny do iloczynu wielkości danych wejściowych i ich logarytmu. Dla optymistycznej wersji, gdy rozpiętość danych nie jest wielka czas ich układania podczas sortowania grzebieniowego wynosi zaledwie kilka milisekund. Nawet w przypadku pesymistycznej wersji dla wielkich zbiorów danych rzędu ćwierć miliona oraz przy dużej rozpiętości danych czas sortowania wynosi tylko kilkadziesiąt milisekund.

9. Podsumowanie

Program ma na celu mierzyć czas wykonywania się dwóch różnych algorytmów sortujących. Dodatkowo w programie zaimplementowana jest funkcja mierząca procentowe wykorzystanie procesora podczas przebiegu sortowania danych w przypadku obu algorytmów. Można zauważyć, że sortowanie grzebieniowe jest lepszą klasą złożoności czasowej, niż sortowanie przez wstawianie. Wynika z tego, że sortowanie grzebieniowe nadaje się lepiej do bardzo dużych ilości nieposortowanych danych. Niestety przekłada się to na większe zużycie zasobów komputera w przypadku sortowania grzebieniowego, niż przez wstawianie.