Jakub Korytko (nr. albumu [redacted])

Informatyka niestacjonarna I stopnia

(rok 1, semestr 2, gr. ćw. 1)

Algorytmy sortowania

Algorytmy i Struktury Danych

Działanie programu

W niniejszej pracy porównamy znaną powszechnie złożoność algorytmów sortowania z własną ich implementacją. Porównania dokonamy na bazie programu konsolowego w języku C, który posiada menu wyboru algorytmu oraz opcję uruchomienia wszystkich algorytmów po kolei. Program działa w następujących krokach:

1. Podanie rozmiaru tablicy
2. Zatwierdzenie enterem
3. Zapoznanie się z wczytaną z pliku tablicą
4. Potwierdzenie enterem, że prawidłowa tablica została wczytana
5. Wybranie algorytmu sortowania (lub innej opcji menu)

Przed uruchomieniem programu należy:

1. Umieścić plik `data` (bez rozszerzenia) w katalogu głównym. Znajdować muszą się w nim liczby (z założenia losowe) rozdzielone spacjami. Do wygenerowania pliku można użyć narzędzia `tools/generate`.
    a. Aby użyć owego narzędzia, uprzednio należy uruchomić komendę `make generate` w katalogu głównym. Utworzy to plik `generate`[1] w folderze `tools/out`. Składnia polecenia (zakładając, że znajdujemy się w katalogu głównym) to:

[1] Na systemie operacyjnym Windows będzie to plik z rozszerzeniem „.exe”

`tools/out/generate <ilość_liczb> <nazwa_pliku>`

Oba argumenty są opcjonalne, domyślne „100000” oraz „data”. Na systemie operacyjnym Windows, konieczne może być zamknięcie polecenia (bez argumentów) w cudzysłowie.

2. Kiedy posiadamy już plik `data` w katalogu głównym, możemy skompilować program. (Dla jasności, było to możliwe wcześniej jednak nie miało większego sensu; bez pliku `data` program nie uruchomiłby się poprawnie). Uruchamiamy `make` w katalogu głównym.
3. Tworzy to plik `sort`[2] w katalogu głównym.

Plik służący do uruchomienia programu to plik `sort`[3]. Uruchamiamy go poprzez wywołanie jego nazwy, tj. komenda `sort` w katalogu głównym.

[2] Na systemie operacyjnym Windows będzie to plik z rozszerzeniem „.exe”
[3] j.w.

Aplikacja zawiera również program do generowania i zapisywania wyników czasu działania algorytmów do plików „*.txt” (dla 3 testów wymienionych w dalszej części dokumentu) dla podanych wielkości tablic. Jego również należy skompilować, poleceniem `make results` w katalogu głównym. Utworzy to plik `results`[4] w folderze `tools/out`. Składnia polecenia (zakładając, że znajdujemy się w katalogu głównym) to:

```
tools/out/results <ilość_wywołań>
<nazwa_pliku_z_liczbami> <indeksy_algorytmów_z_tablicy>
(oddzielone spacjami, między 0 a 5)>
```

`<ilość_wywołań>`: każde kolejne wywołanie uruchamia sortowanie dla ilości elementów określonych ciągiem: 1000, 2000, 5000, 10000, 20000, 50000, 100000, 200000, 500000, 1000000...

tj. 1 wywołanie - dla 1000 elementów,

2 wywołania - dla 1000 el., dla 2000 el.,

3 wywołania - dla 1000 el., dla 2000 el., dla 5000 el.,

…

Przy użyciu komendy dla N wywołań, zapisany do plików zostanie czas każdego wywołania algorytmów, dla każdego z 3 testów osobno.

[4] Na systemie operacyjnym Windows będzie to plik z rozszerzeniem „.exe”

`<indeksy_algorytmów_z_tablicy>`:

0 – Sortowanie przez wstawianie,

1 – Sortowanie przez selekcję,

2 – Sortowanie bąbelkowe,

3 – Sortowanie Quicksort,

4 – Sortowanie Shellsort,

5 – Sortowanie przez kopcowanie

Po uruchomieniu komendy utworzą się pliki z wynikami dostępne w katalogu `results`

Skoro działanie programu jest już jasne to na podstawie wykresów programu Microsoft Excel spróbujemy dokonać aproksymacji złożoności czasowych algorytmów. Dane do wykresów zostały wygenerowane poleceniem `tools/out/results` .

Badane algorytmy

Grupa I

- Sortowanie przez wstawianie
- Sortowanie przez selekcję
- Sortowanie bąbelkowe

Grupa II

- Sortowanie Quicksort
- Sortowanie Shella
- Sortowanie przez kopcowanie

Grupy te różnią się złożonością czasową ale również i trudnością implementacji. 1 grupa jest niezwykle prosta do zaimplementowania ale za to dość powolna (bardzo zła dla dużych rozmiarów tablic), 2 natomiast jest szybka ale trudniejsza w implementacji.

Aby lepiej zobrazować sobie różnice w złożoności implementacji, pokażemy owe algorytmy pseudokodem oraz na schematach blokowych.

Algorytmy grupy I – opis

– Sortowanie przez wstawianie

Algorytm sortowania przez wstawianie polega na kolejnym pobieraniu elementów i wstawianiu ich na właściwe miejsca w posortowanej już sekwencji elementów. Można to porównać do układania kart pobieranych z talii, gdzie najpierw bierzemy pierwszą kartę, a następnie pobieramy kolejne, aż wyczerpiemy całą talie. Każdą pobraną kartę porównujemy z kartami, które już mamy w ręce i szukamy dla niej odpowiedniego miejsca w sekwencji, tak aby zachować jej porządek. Jeśli nowy element jest większy od elementów w sekwencji, to umieszczamy go na końcu, w przeciwnym razie wstawiamy go na odpowiednie miejsce przed elementami już posortowanymi.

– Sortowanie przez selekcję

Algorytm sortowania przez selekcje polega na wyszukaniu elementu o najmniejszej wartości i zamianie go z elementem na pierwszej pozycji. W ten sposób umieszczamy element o najmniejszej wartości na swojej właściwej pozycji. Powtarzamy ten sam proces dla pozostałych elementów zbioru, aż wszystkie elementy zostaną posortowane.

– Sortowanie bąbelkowe

Algorytm sortowania bąbelkowego opiera się na porównywaniu sąsiadujących elementów i zamianie ich kolejności, jeśli nie spełniają kryterium porządkowego. Proces ten jest wykonywany cyklicznie aż do momentu, gdy cały zbiór zostanie posortowany.

Algorytmy grupy I – cechy

PLUSY

- Algorytmy te są bardzo proste do implementacji

- Można je z łatwością wyprowadzić samodzielnie na podstawie znajomości zasady działania

- Dla małej ilości danych spełniają swoją rolę

- Są dobrym przykładem do nauki koncepcji sortowania; algorytmy złożone takie jak sortowanie przez kopcowanie mogłyby utrudnić zrozumienie koncepcji osobom, które nie miały wcześniej styczności z sortowaniem

MINUSY

- Algorytmy te cechują się złożonością czasową $O(n^2)$

- Znane są o wiele lepsze algorytmy sortowania

- Są niepraktyczne dla dużych rozmiarów tablic

Algorytmy grupy I – pseudokod

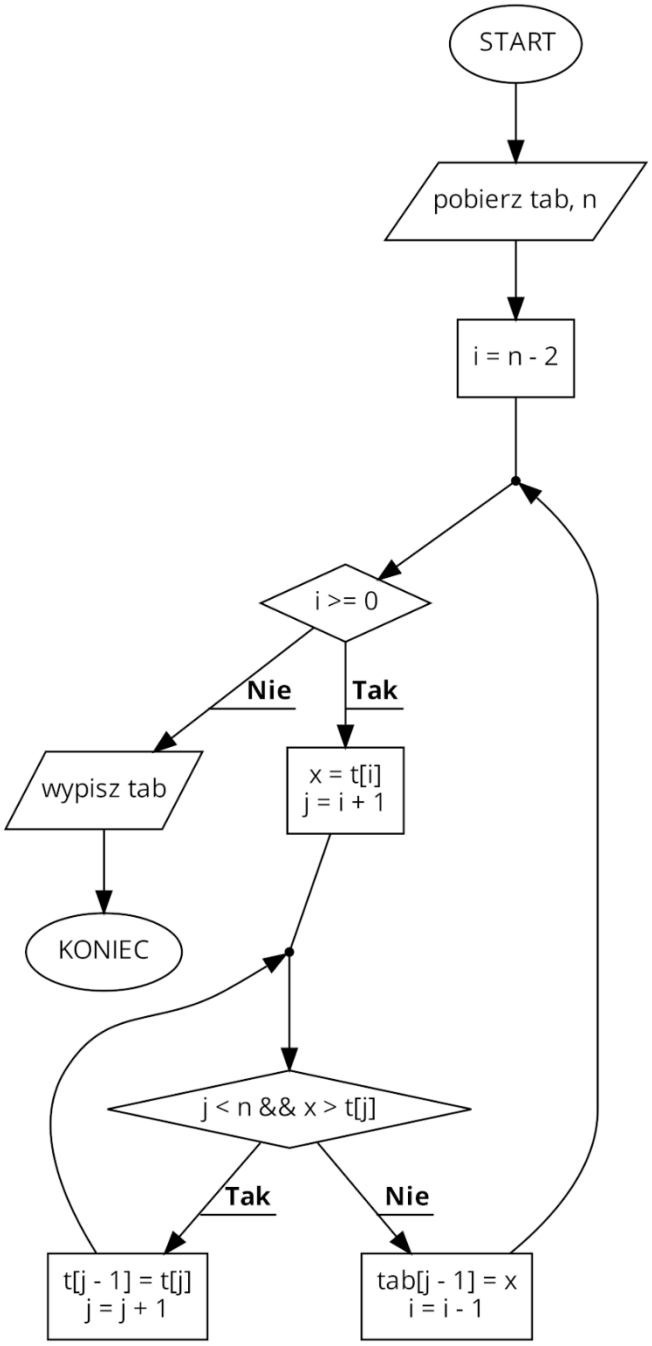
```
// Sortowanie przez wstawianie

Od i = długość tablicy - 2 do i >= 0 {

    Ustaw x = t[i] oraz j = i+1

    Dopóki j < długość tablicy oraz x > t[j] {
        t[j - 1] = t[j]
        Ustaw j = j + 1
    }

    Ustaw d[j - 1] = x
}

// Sortowanie przez selekcje

Od i = 0 do i < długość tablicy - 1 {
    Ustaw min = i

    Od j = i + 1 do j < długość tablicy {
        Jeśli t[j] < t[min] to ustaw min = j
    }

    Zamień t[min] z t[i] miejscami
}

// Sortowanie bąbelkowe

Od i = 0 do i < długość tablicy - 1 {
    Od i = 0 do i < długość tablicy - 1 {
        Jeśli t[j]>t[j+1], zamień je
    }
}
```

Algorytmy grupy I – schematy blokowe

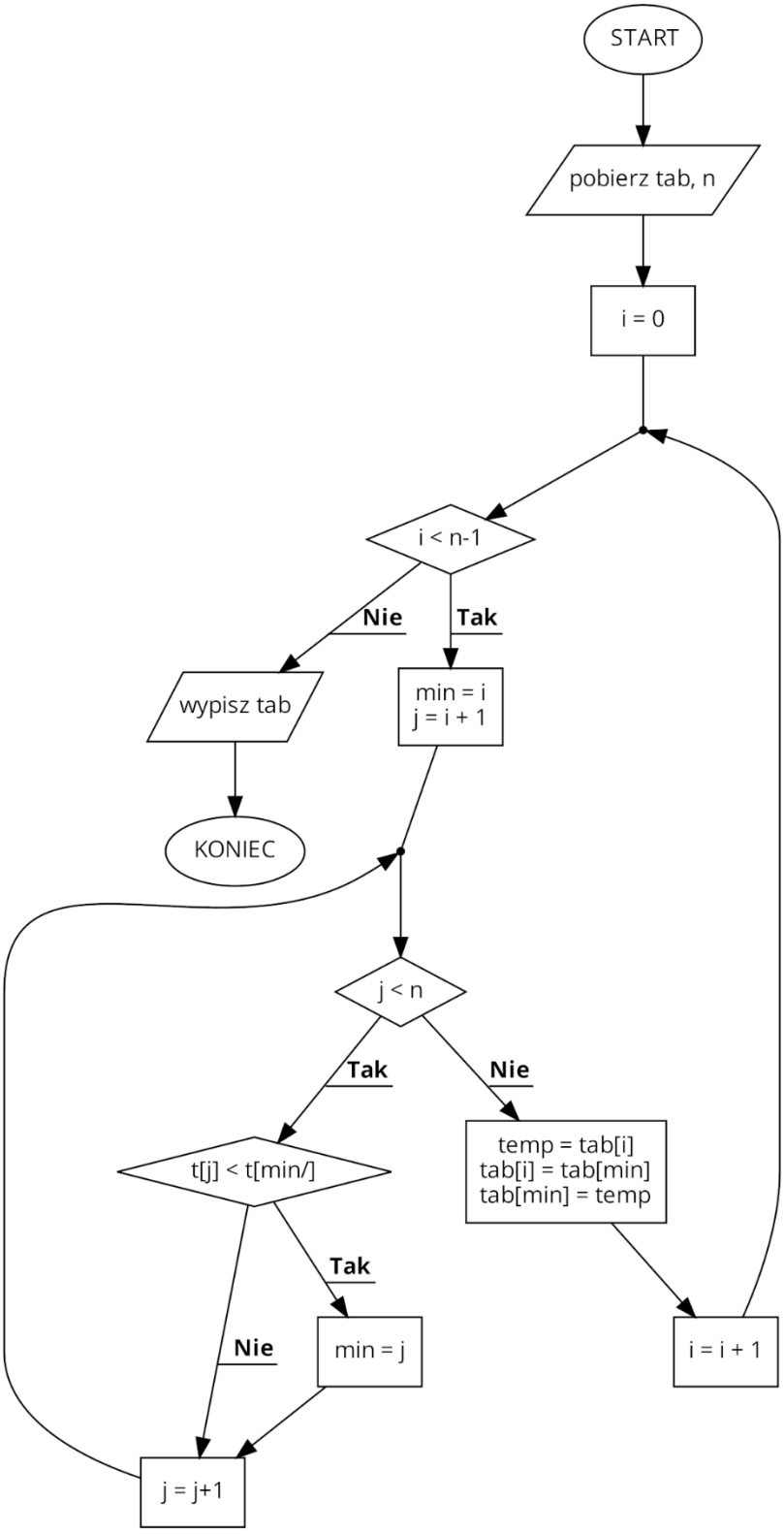
n: *rozmiar tablicy*

tab: *tablica z elementami do posortowania*

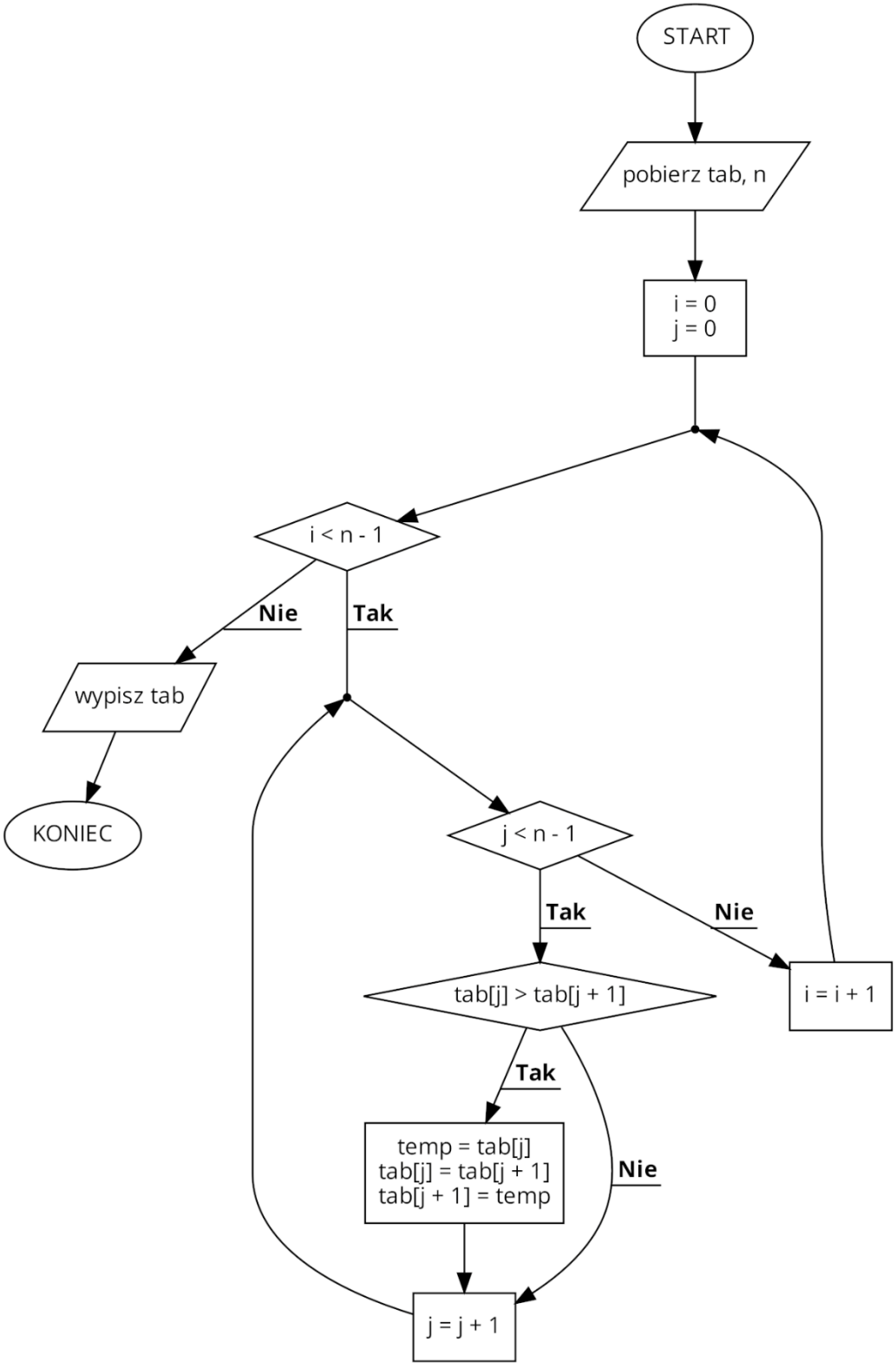
Sortowanie przez wstawianie



Sortowanie przez selekcje



Sortowanie bąbelkowe



Algorytmy grupy II – opis

- Sortowanie Quicksort

Algorytm sortowania szybkiego (ang. quicksort) polega na podziale sortowanego zbioru na dwie części w taki sposób, że wszystkie elementy w lewej części (zwanej lewą partycją) są mniejsze lub równe od wszystkich elementów w prawej części (zwanej prawą partycją). Następnie każdą z partycji sortuje się rekurencyjnie tym samym algorytmem. Po posortowaniu obu partycji, łączy się je w jeden zbiór posortowany.

- Sortowanie Shella

Algorytm sortowania Shell Sort polega na podziale zbioru na podzbiory i sortowaniu każdego z tych podzbiorów za pomocą algorytmu przez wstawianie z różnymi odstępami. Następnie odstęp jest zmniejszany, a operacje sortowania i zmniejszania odstępu są powtarzane aż do osiągnięcia odstępu równego 1, po czym sortujemy za pomocą algorytmu przez wstawianie.

- Sortowanie przez kopcowanie

Algorytm przez kopcowanie to metoda sortowania elementów w strukturze drzewiastej, zwanej kopcem. Kopiec jest drzewem, w którym każdy węzeł posiada wartość nie mniejszą niż wartość jego rodzica. Algorytm sortowania przez kopcowanie polega na budowie kopca z nieuporządkowanych elementów, następnie wyciąganiu elementu o największej wartości i umieszczaniu go na końcu tablicy, aż do momentu, gdy cała tablica zostanie posortowana.

Algorytmy grupy II – cechy

PLUSY

- Algorytmy te cechują się złożonością czasową $O(n * \log n)$ (z wyjątkiem algorytmu sortowania Shella, jednak mimo wszystko jest on najszybszym algorytmem z algorytmów o klasie $O(n^2)$

- Dzięki prędkości ich działania można posortować duże tablice danych

- Są to bardzo szybkie algorytmy

MINUSY

- Są to algorytmy stosunkowo trudne do zrozumienia (tak jak przy algorytmach grupy I wystarczy zwykła logika do zobrazowania sobie ich działania, nie jest to tak oczywiste np. w przypadku algorytmu przez kopcowanie, który wymaga wiedzy informatycznej – kopce, drzewa binarne itp.)

- Algorytm sortowania szybkiego oraz Shella może degradować się do klasy $O(n^2)$

- Algorytmy te są trudne do implementacji

Algorytmy grupy II – pseudokod

```
// Sortowanie szybkie

Funkcja(tab, lewy, prawy):

    Ustaw i = lewy - 1, j = prawy + 1
    Ustaw piwot = tab[Całkowita z (lewy + prawy) / 2]

    Dopóki (prawda):
    {
        Dopóki (tab[++i] < piwot)
        Dopóki (tab[--j] > piwot)

        Jeżeli i <= j: zamień tab[i] z tab[j]
        W przeciwnym razie: break
    }

    Jeżeli lewy < j to Funkcja(tab, lewy, j)
    Jeżeli prawy > i to Funkcja(tab, i, prawy)

Funkcja(tab, 0, n-1)
```

```
// Sortowanie Shella

Od h = 1 do h < długość tablicy, 3 * h + 1
h = Liczba całkowita z h / 9
Jeśli h == 0 to ustaw h = h + 1

Dopóki h != 0: {
    Od i = długość tablicy - h - 1 do i >= 0: {
        Ustaw x = t[i], j = i + h

        Dopóki j < długość tablicy oraz x > t[j]: {
            Ustaw t[j - h] = t[j] oraz j = j + h
        }
        Ustaw t[j - h] = x
    }
    Ustaw h = Liczba całkowita z h / 3
}
```

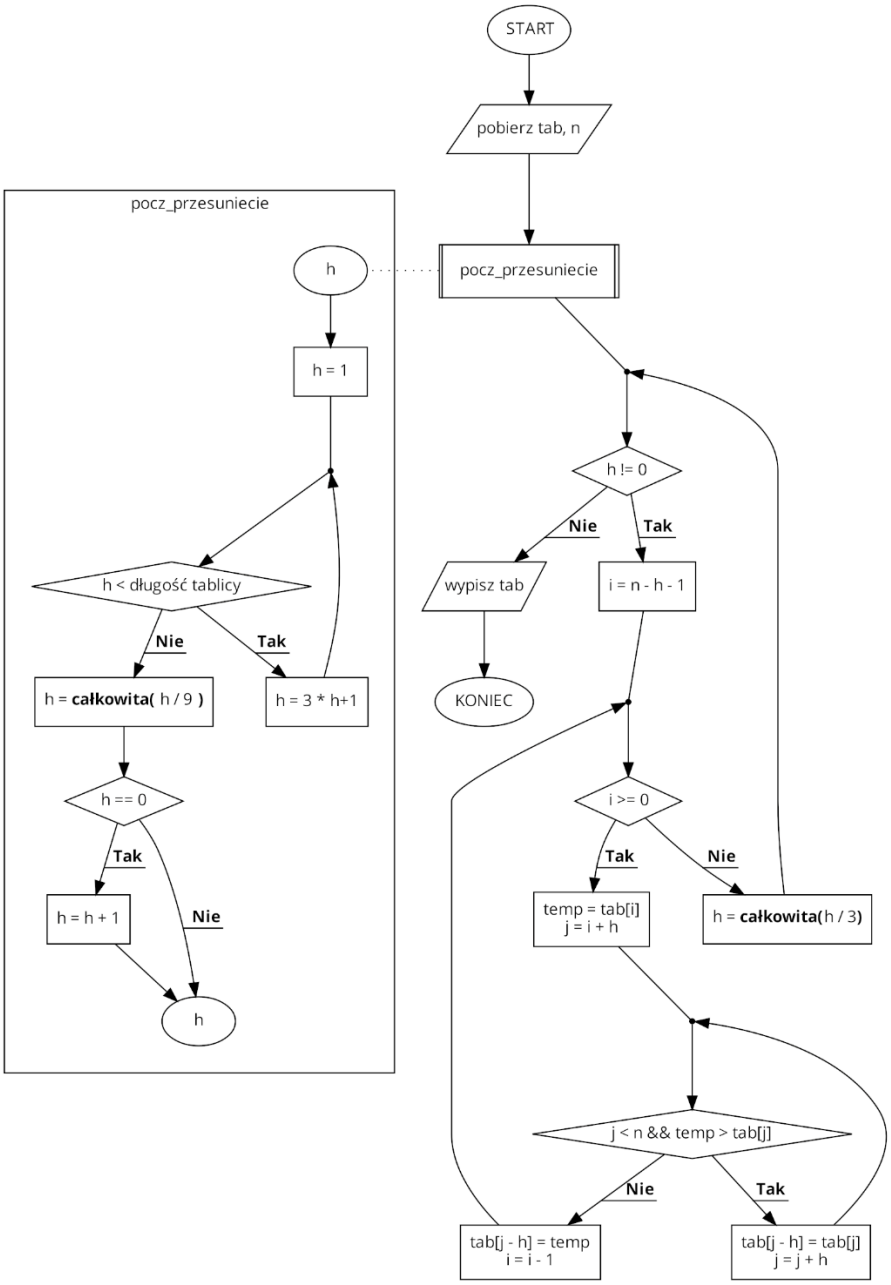
```
// Sortowanie przez kopcowanie

// Budowanie kopca

Od i=2 do i<= długość tablicy:
    Ustaw j=i, Całkowita k = j / 2, x = tab[i]
    Dopóki k>0 oraz tab[k] < x: {
        Ustaw tab[j] = tab[k],
        j = k, k = Całkowita j / 2
    }
    tab[j] = x
}

// Rozbiór kopca

Od i = długość tablicy do i > 1: {

    Zamieniamy tab[1] z tab[i]
    Ustaw j = 1, k = 2

    Dopóki k < i: {
        Jeżeli k + 1 < i oraz tab[k + 1] > tab[k]:
            m = k + 1
        W innym wypadku:
            m = k

    Jeżeli tab[m] <= tab[j], koniec pętli (break)
    Zamień tab[j] z tab[m]
    Ustaw j = m, k = j + j
    }
}
```

Algorytmy grupy II – schematy blokowe

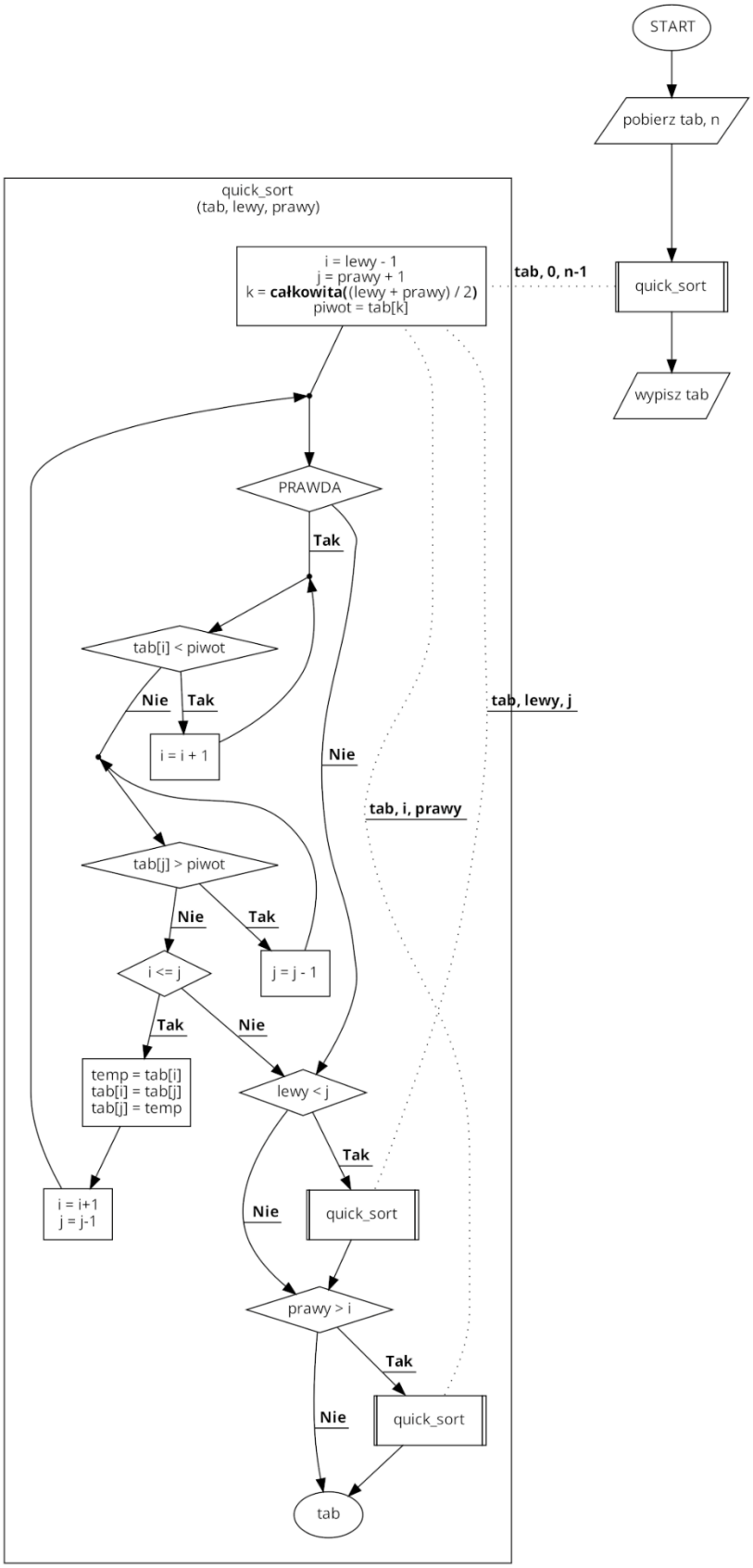
n: *rozmiar tablicy*

tab: *tablica z elementami do posortowania*

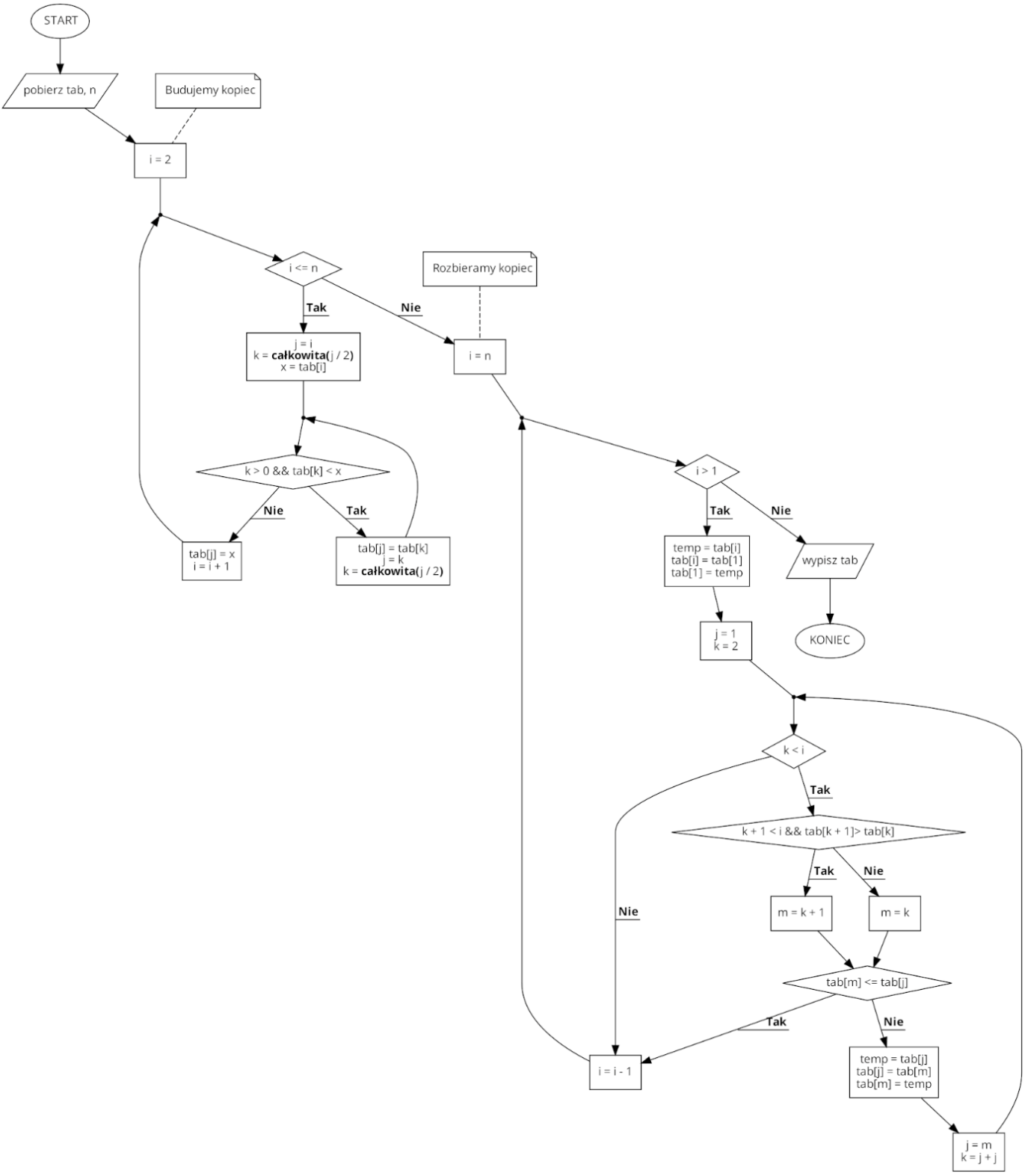
Sortowanie Shella



Sortowanie Quicksort



Sortowanie przez kopcowanie



Testy algorytmów

Przeprowadzimy 3 testy dla każdej grupy algorytmów, następnie spróbujemy dokonać aproksymacji złożoności czasowej algorytmów oraz porównamy ich wykresy zbiorcze.

Test1 - Dla danych wygenerowanych losowo

Test2 - Dla danych posortowanych w kolejności odwrotnej (malejąco)

Test3 - Dla danych posortowanych właściwie (rosnąco)

Czas podawany jest w mikrosekundach (aby otrzymać sekundy należy pomnożyć czasy przez 10^{-6})

Wyniki testów (tabela)

Tabela wyników dla wszystkich 3 testów na podstawie której stworzymy wykresy.

Test 1			
N	Sortowanie bąbelkowe	Sortowanie przez wstawianie	Sortowanie przez selekcje
1000	5000	1000	4000
2000	17000	2000	5000
5000	156000	19000	39000
10000	578000	89000	161000
20000	2257000	304000	640000
50000	14318000	1938000	4288000
100000	69017000	13257000	16594000
200000	255626000	37141000	75299000

N	Sortowanie przez kopcowanie	Sortowanie Quicksort	Sortowanie Shella
1000	<1	<1	<1
2000	<1	<1	<1
5000	<1	<1	<1
10000	2000	2000	2000
20000	5000	3000	3000
50000	13000	4000	10000
100000	22000	8000	16000
200000	55000	18000	48000
500000	148000	79000	111000
1000000	342000	100000	200000

Test 2			
N	Sortowanie bąbelkowe	Sortowanie przez wstawianie	Sortowanie przez selekcje
1000	9000	2000	1000
2000	59000	5000	6000
5000	185000	55000	47000
10000	515000	192000	183000
20000	1770000	778000	777000
50000	15920000	4874000	4697000
100000	64649000	24089000	17894000
200000	189851000	86573000	69549000

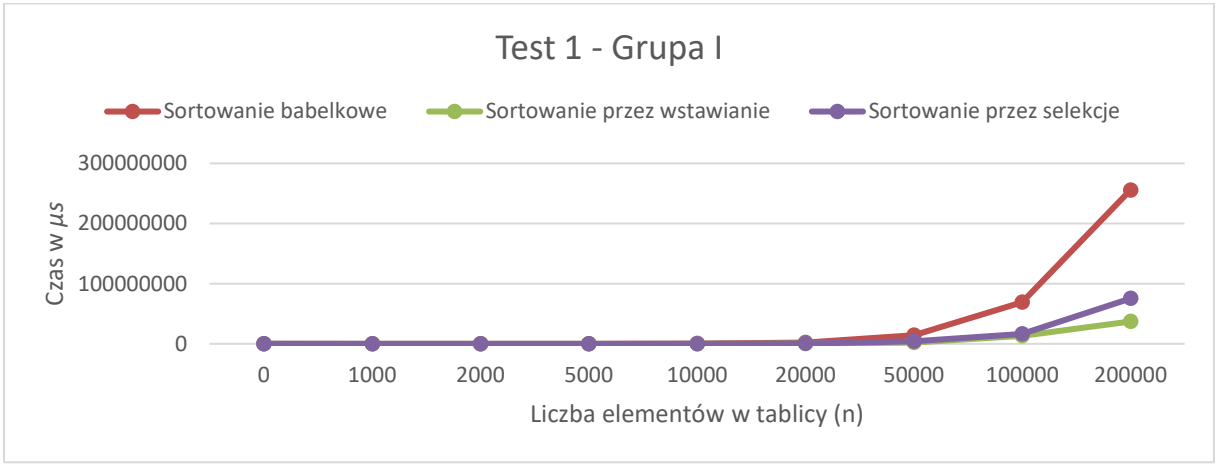
N	Sortowanie przez kopcowanie	Sortowanie Quicksort	Sortowanie Shella
1000	<1	<1	<1
2000	<1	<1	<1
5000	2000	<1	<1
10000	2000	1000	<1
20000	5000	1000	1000
50000	9000	3000	2000
100000	21000	6000	8000
200000	49000	13000	16000
500000	126000	37000	44000
1000000	210000	77000	74000

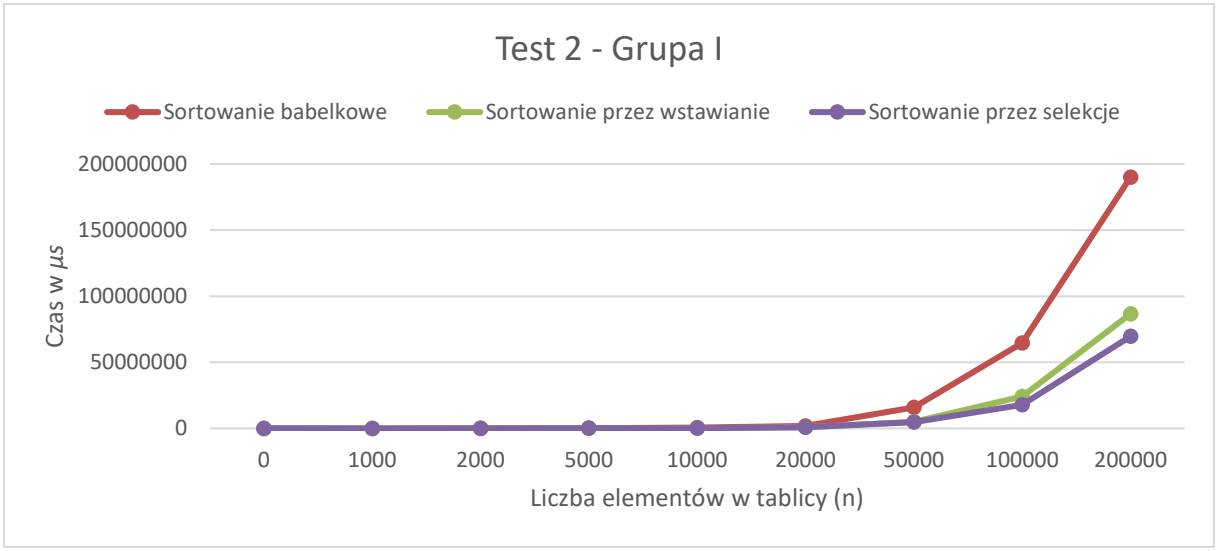
Test 3

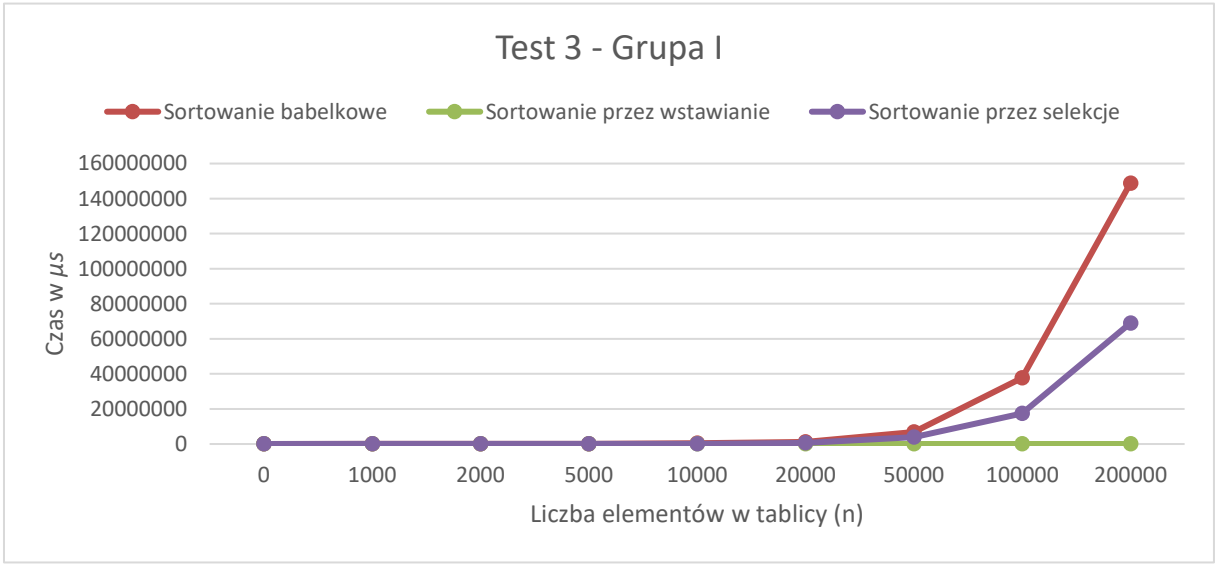
N	Sortowanie bąbelkowe	Sortowanie przez wstawianie	Sortowanie przez selekcje
1000	4000	<1	2000
2000	16000	<1	6000
5000	80000	<1	39000
10000	364000	<1	157000
20000	1165000	<1	653000
50000	6880000	<1	3985000
100000	37649000	<1	17378000
200000	148781000	1000	69007000

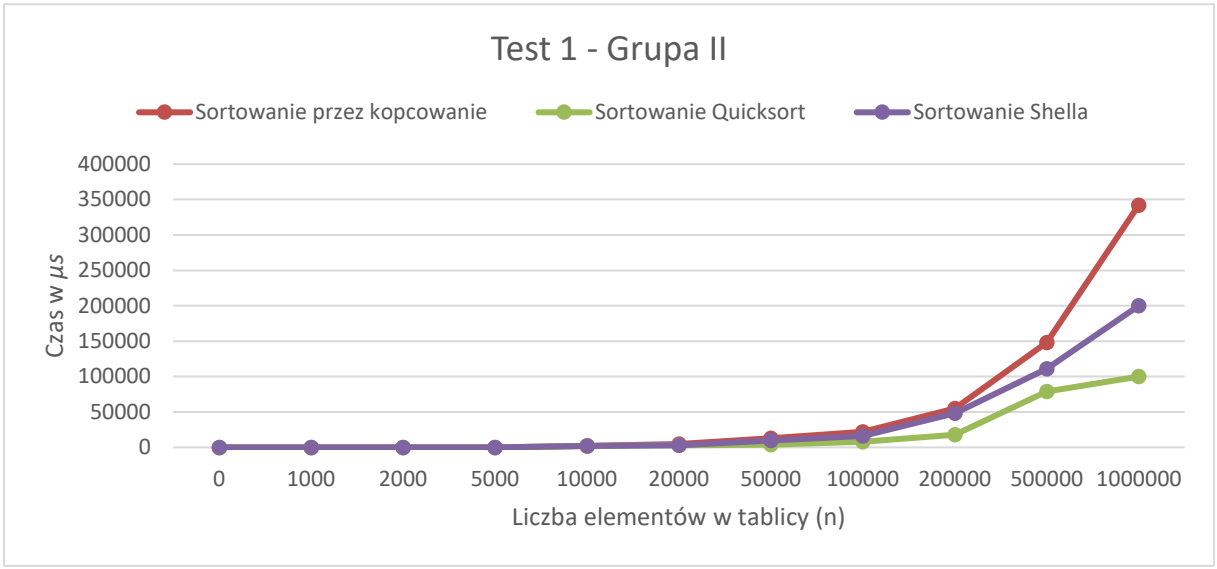
N	Sortowanie przez kopcowanie	Sortowanie Quicksort	Sortowanie Shella
1000	<1	<1	<1
2000	<1	<1	<1
5000	1000	1000	<1
10000	2000	1000	<1
20000	3000	2000	1000
50000	7000	3000	2000
100000	19000	6000	5000
200000	48000	19000	12000
500000	100000	43000	30000
1000000	195000	67000	58000

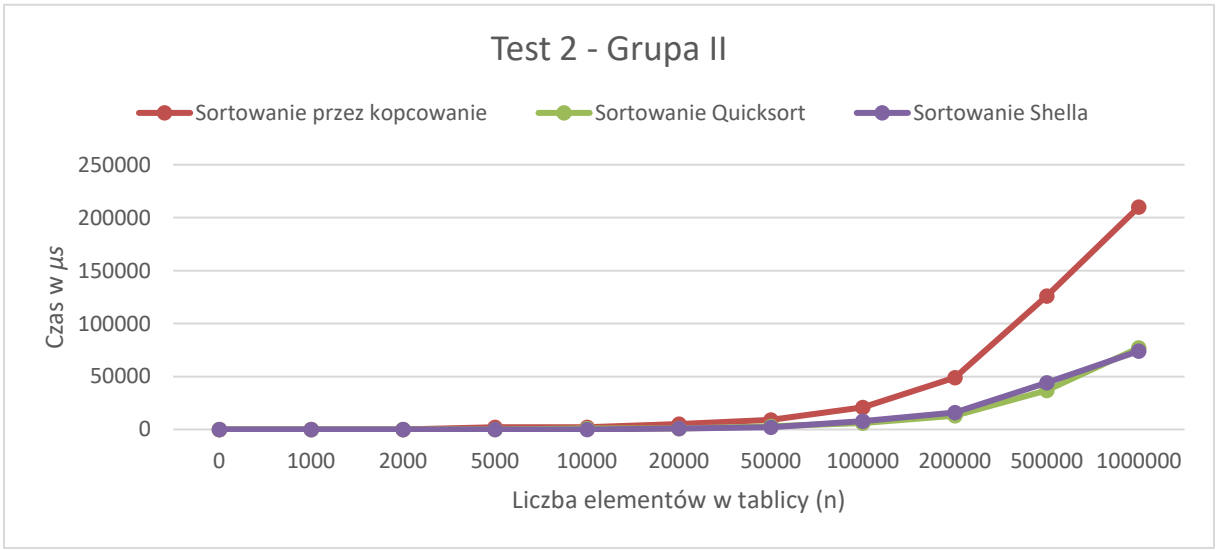
Wykresy

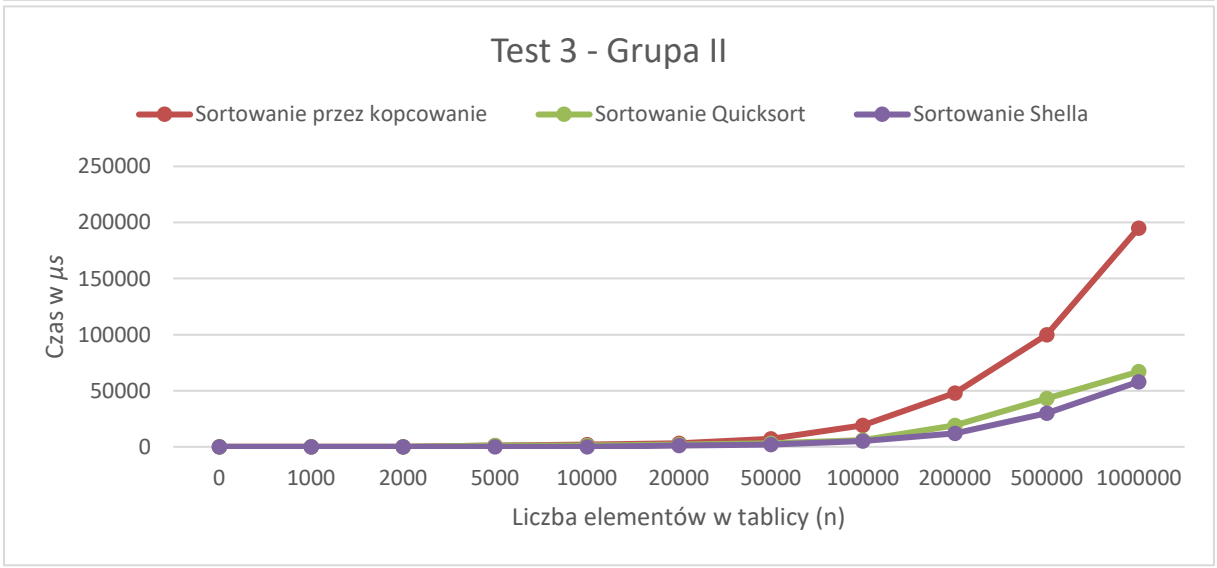

Test 1 - Grupa I
Sortowanie babelkowe
Sortowanie przez wstawianie
Sortowanie przez selekcje
300000000
200000000
100000000
0
Czas w µs
0
1000
2000
5000
10000
20000
50000
100000
200000
Liczba elementów w tablicy (n)

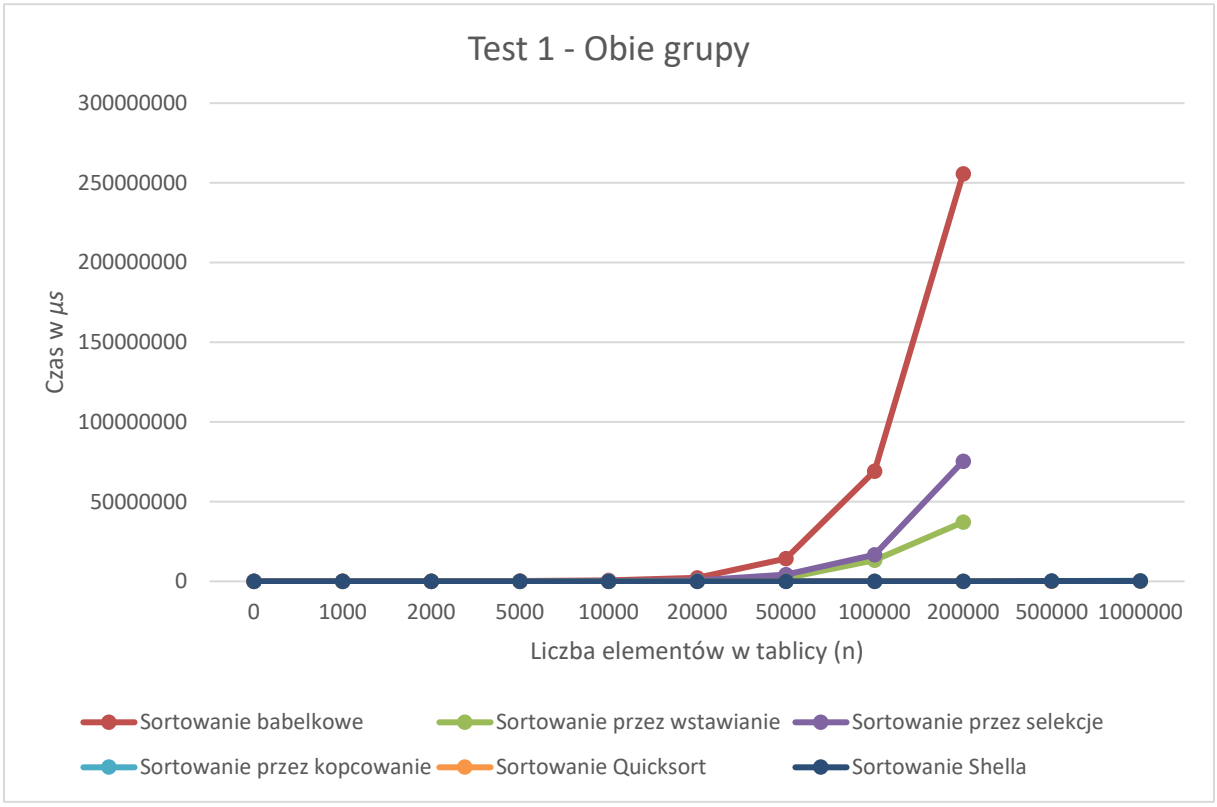

Test 2 - Grupa I
Sortowanie babelkowe
Sortowanie przez wstawianie
Sortowanie przez selekcje
200000000
150000000
100000000
50000000
0
Czas w µs
0
1000
2000
5000
10000
20000
50000
100000
200000
Liczba elementów w tablicy (n)

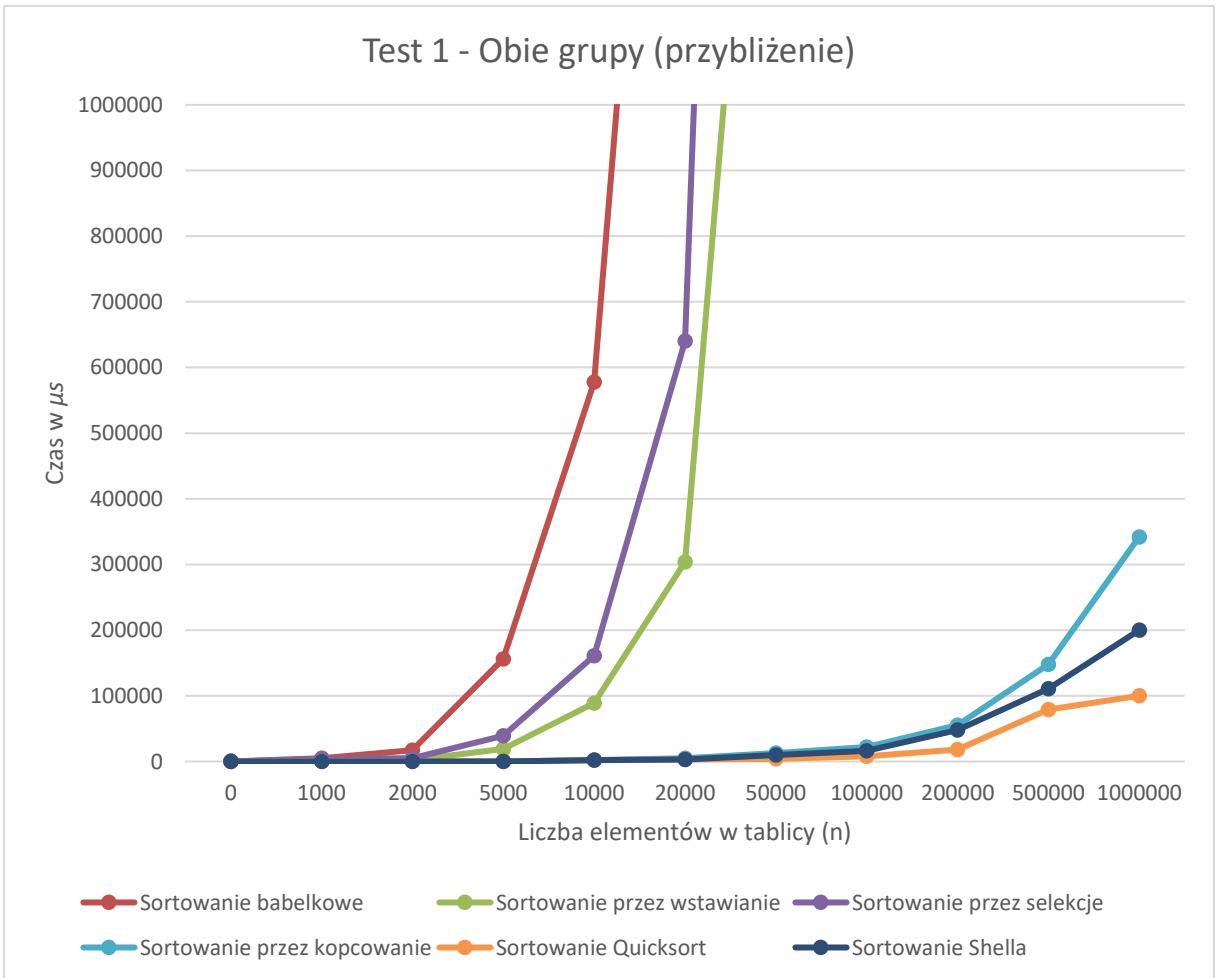

Test 3 - Grupa I
Sortowanie babelkowe
Sortowanie przez wstawianie
Sortowanie przez selekcje
160000000
140000000
120000000
100000000
80000000
60000000
40000000
20000000
0
Czas w µs
0
1000
2000
5000
10000
20000
50000
100000
200000
Liczba elementów w tablicy (n)

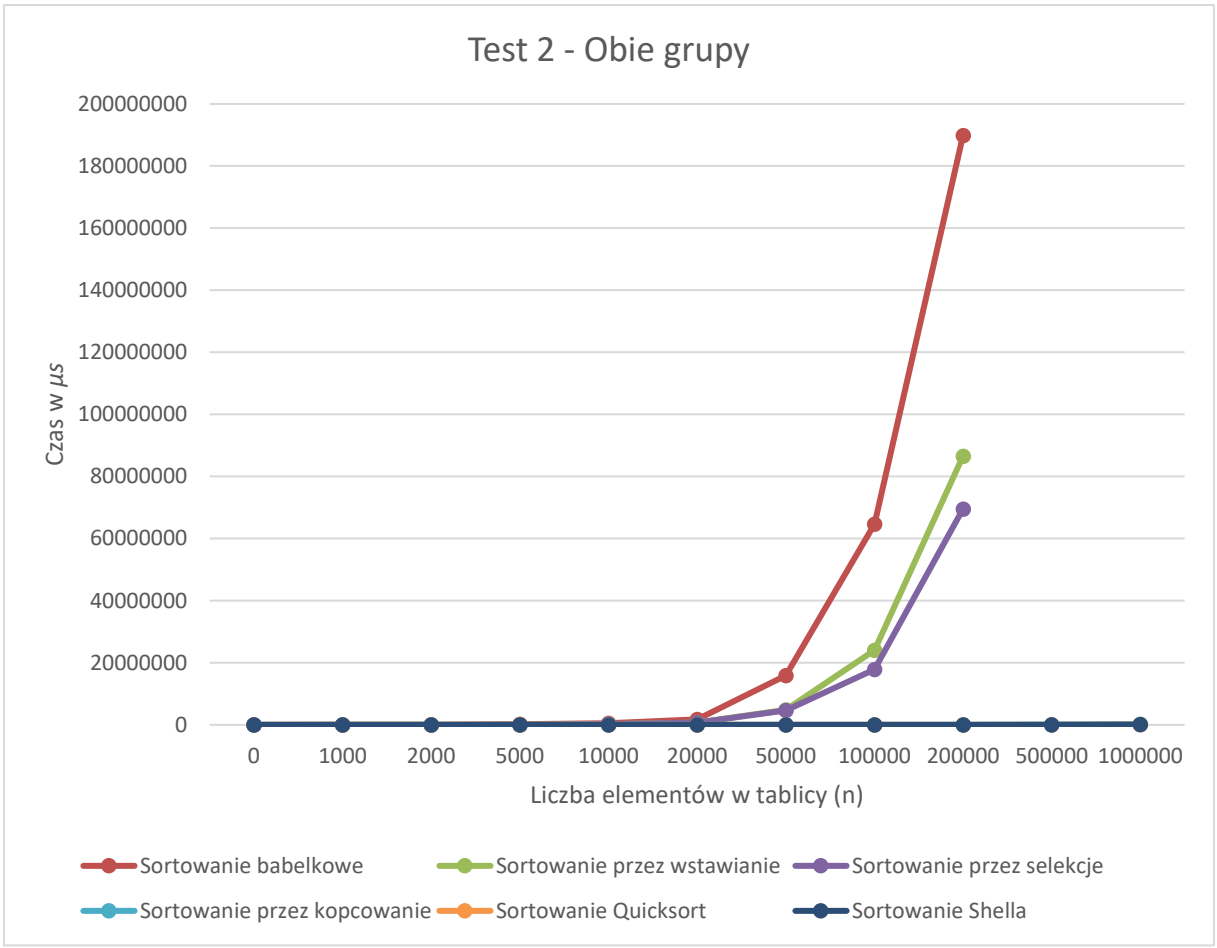

Test 1 - Grupa II
Sortowanie przez kopcowanie
Sortowanie Quicksort
Sortowanie Shella
400000
350000
300000
250000
200000
150000
100000
50000
0
Czas w µs
0
1000
2000
5000
10000
20000
50000
100000
200000
500000
1000000
Liczba elementów w tablicy (n)

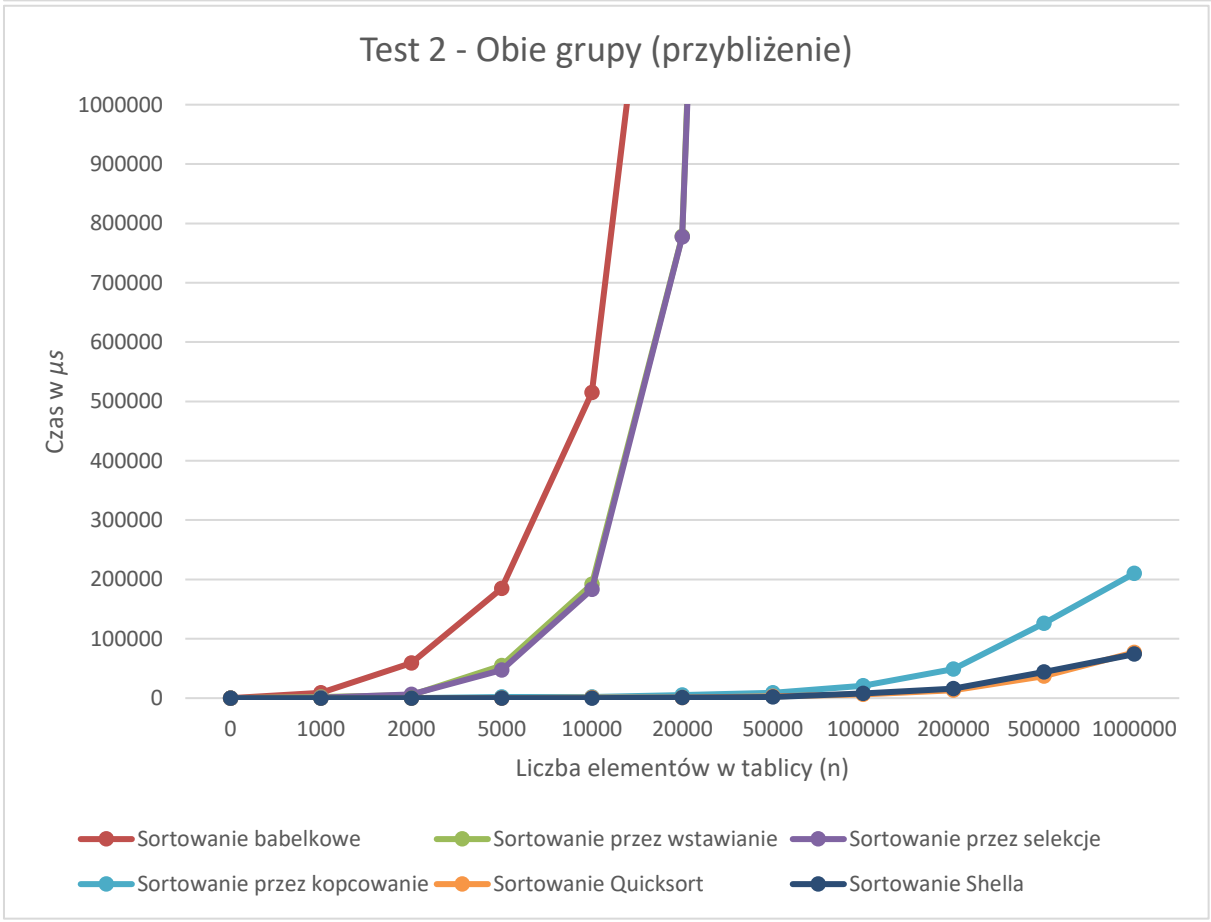

Test 2 - Grupa II
Sortowanie przez kopcowanie
Sortowanie Quicksort
Sortowanie Shella
250000
200000
150000
100000
50000
0
Czas w µs
0
1000
2000
5000
10000
20000
50000
100000
200000
500000
1000000
Liczba elementów w tablicy (n)

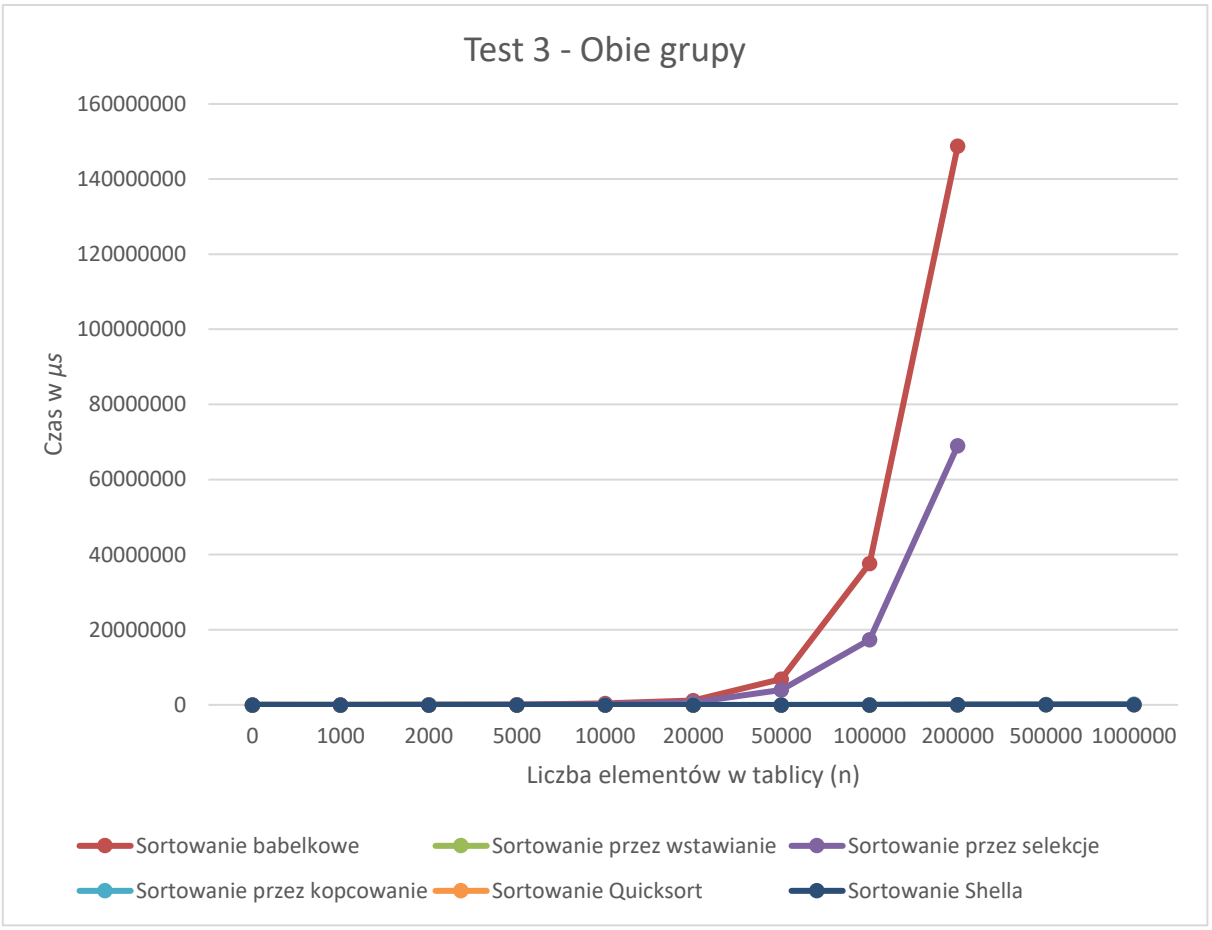

Test 3 - Grupa II
Sortowanie przez kopcowanie
Sortowanie Quicksort
Sortowanie Shella
250000
200000
150000
100000
50000
0
Czas w µs
0
1000
2000
5000
10000
20000
50000
100000
200000
500000
1000000
Liczba elementów w tablicy (n)

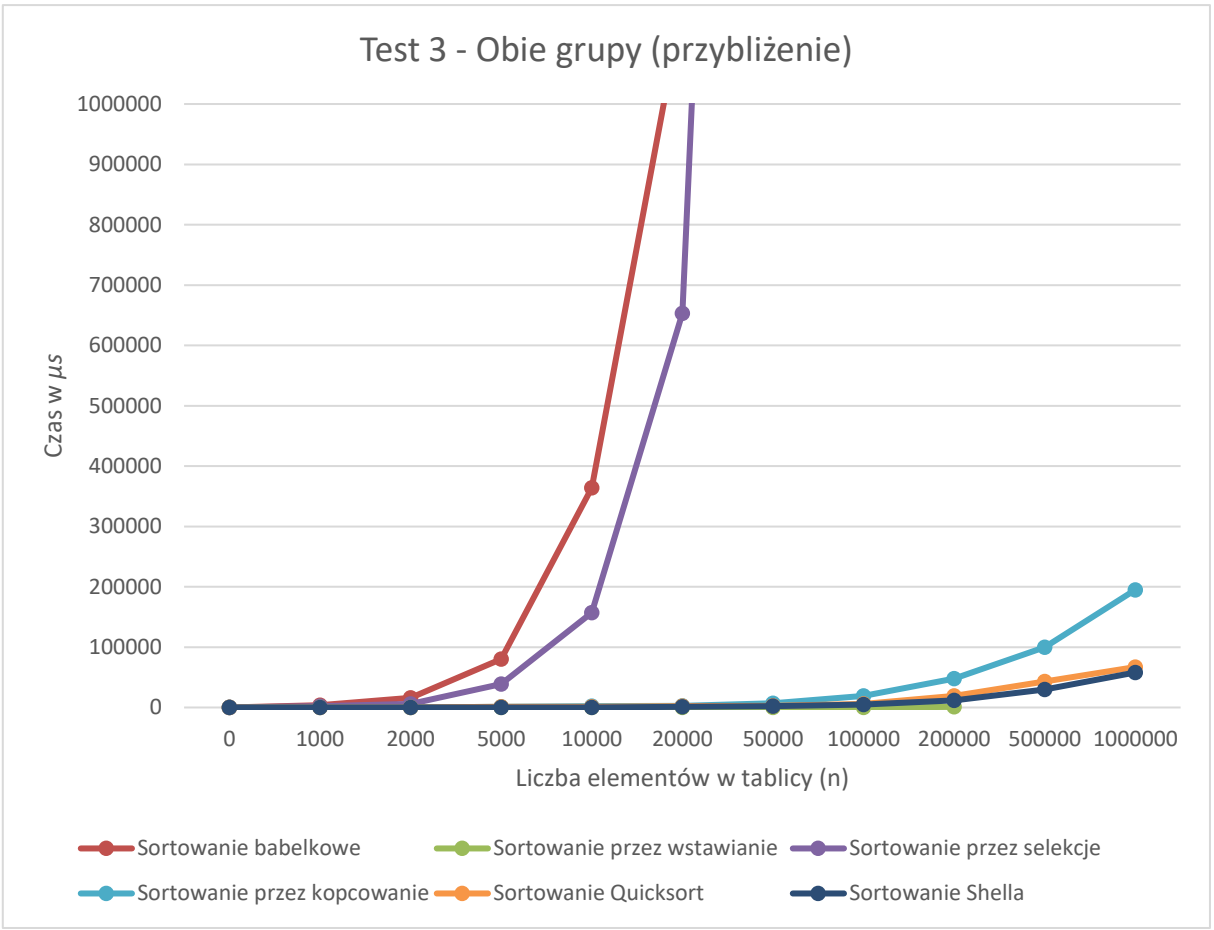

Test 1 - Obie grupy
300000000
250000000
200000000
150000000
100000000
50000000
0
Czas w µs
0
1000
2000
5000
10000
20000
50000
100000
200000
500000
1000000
Liczba elementów w tablicy (n)
Sortowanie babelkowe
Sortowanie przez wstawianie
Sortowanie przez selekcje
Sortowanie przez kopcowanie
Sortowanie Quicksort
Sortowanie Shella


Test 1 - Obie grupy (przybliżenie)
1000000
900000
800000
700000
600000
500000
400000
300000
200000
100000
0
Czas w µs
0
1000
2000
5000
10000
20000
50000
100000
200000
500000
1000000
Liczba elementów w tablicy (n)
Sortowanie babelkowe
Sortowanie przez wstawianie
Sortowanie przez selekcje
Sortowanie przez kopcowanie
Sortowanie Quicksort
Sortowanie Shella


Test 2 - Obie grupy
200000000
180000000
160000000
140000000
120000000
100000000
80000000
60000000
40000000
20000000
0
Czas w µs
0
1000
2000
5000
10000
20000
50000
100000
200000
500000
1000000
Liczba elementów w tablicy (n)
Sortowanie babelkowe
Sortowanie przez wstawianie
Sortowanie przez selekcje
Sortowanie przez kopcowanie
Sortowanie Quicksort
Sortowanie Shella


Test 2 - Obie grupy (przybliżenie)
1000000
900000
800000
700000
600000
500000
400000
300000
200000
100000
0
Czas w µs
0
1000
2000
5000
10000
20000
50000
100000
200000
500000
1000000
Liczba elementów w tablicy (n)
Sortowanie babelkowe
Sortowanie przez wstawianie
Sortowanie przez selekcje
Sortowanie przez kopcowanie
Sortowanie Quicksort
Sortowanie Shella


Test 3 - Obie grupy
160000000
140000000
120000000
100000000
80000000
60000000
40000000
20000000
0
Czas w µs
0
1000
2000
5000
10000
20000
50000
100000
200000
500000
1000000
Liczba elementów w tablicy (n)
Sortowanie babelkowe
Sortowanie przez wstawianie
Sortowanie przez selekcje
Sortowanie przez kopcowanie
Sortowanie Quicksort
Sortowanie Shella


Test 3 - Obie grupy (przybliżenie)
1000000
900000
800000
700000
600000
500000
400000
300000
200000
100000
0
Czas w µs
0
1000
2000
5000
10000
20000
50000
100000
200000
500000
1000000
Liczba elementów w tablicy (n)
Sortowanie babelkowe
Sortowanie przez wstawianie
Sortowanie przez selekcje
Sortowanie przez kopcowanie
Sortowanie Quicksort
Sortowanie Shella

Analiza wyników oraz wnioski

Grupa I:

Po szybkości rośnięcia wykresów z łatwością można stwierdzić że funkcje te są postaci $An^2 + Bn + C$ -> ich złożoność czasowa to $O(n^2)$. Jednak aby tego dowieść, przeanalizujmy przykładowe wyniki tych funkcji i porównajmy do wzrostu klasy $O(n^2)$ aby stwierdzić czy nasze założenie jest zgodne z prawdą:

Sortowanie bąbelkowe:

Dla n = 1000: 5000 µs

Dla n = 2000: 17000 µs -> wzrost czasu o 3.4 raza (wolniej niż $O(n^2)$)

Dla n = 5000: 156000 µs -> wzrost czasu o ok. 9.2 raza (szybciej niż $O(n^2)$)

Dla n = 10000: 578000 µs -> wzrost czasu o ok. 3.7 raza (wolniej niż $O(n^2)$)

Sortowanie przez wstawianie:

Dla n = 1000: 1000 µs

Dla n = 2000: 2000 µs -> wzrost czasu o 2 razy (dużo wolniej niż $O(n^2)$)

Dla n = 5000: 19000 µs -> wzrost czasu o ok. 9.5 raza (szybciej niż $O(n^2)$)

Dla n = 10000: 89000 µs -> wzrost czasu o ok. 4.7 raza (szybciej niż $O(n^2)$)

Sortowanie przez selekcje:

Dla n = 1000: 4000 µs

Dla n = 2000: 5000 µs -> wzrost czasu o 1.25 raza (dużo wolniej niż $O(n^2)$)

Dla n = 5000: 39000 µs -> wzrost czasu o ok. 7.8 raza (szybciej niż $O(n^2)$)

Dla n = 10000: 161000 µs -> wzrost czasu o ok. 4.12 raza (nieznacznie szybciej niż $O(n^2)$)

Można zauważyć tu pewną prawidłowość, algorytmy te wykonują się szybciej lub wolniej ze względu na różne czynniki (chociażby na szybkość maszyny czy stan posortowania tablicy). Jednak mniej więcej wyrównują się do poziomu $O(n^2)$, należy pamiętać również, że w klasie ignorujemy stałe.

Z tego względu można stwierdzić, że **klasa złożoności algorytmów grupy I wynosi $\boldsymbol{O(n^2)}$** i pokrywa się ona z ogólnodostępną wiedzą. Występuje to dla wszystkich algorytmów, jednakże w przypadku danych posortowanych **klasa złożoności czasowej algorytmu sortowania przez wstawianie dąży do $\boldsymbol{O(n)}$.**

Grupa II:

Po szybkości rośnięcia wykresów grupy 2 wcale nie jest tak łatwo stwierdzić ich klasy. Widzimy jednak z wykresów zbiorowych, że **są one o wiele szybsze niż algorytmy grupy I.** Bazując na powszechnej wiedzy, algorytm Shella posiada mniej więcej złożoność $O(n^{\frac{3}{4}})$ a reszta algorytmów $O(n * log\, n)$. Jednak algorytm Quicksort jest w stanie zdegradować się do klasy $O(n^2)$ a algorytm Shella w najlepszym przypadku do $O(n * log\, n)$ oraz $O(n^2)$ w najgorszym. Mamy tu dość duży rozstrzał, ale możemy mniej więcej porównać je do znanych złożoności. Spróbujmy więc mniej więcej określić klasy złożonościowe tych algorytmów.

Sortowanie przez kopcowanie:

Dla n = 1000: 2000 µs

Dla n = 2000: 5000 µs -> wzrost czasu o 2.5 raza (szybciej niż $O(n * \log n)$)

Dla n = 5000: 13000 µs -> wzrost czasu o ok. 2.6 raza (nieznacznie szybciej niż $O(n * \log n)$)

Dla n = 10000: 22000 µs -> wzrost czasu o ok. 1.7 raza (wolniej niż $O(n * \log n)$)

Sortowanie Shella:

Dla n = 1000: 2000 µs

Dla n = 2000: 3000 µs -> wzrost czasu o 1.5 razy (wolniej niż $O(n^{\frac{3}{4}})$)

Dla n = 5000: 4000 µs -> wzrost czasu o ok. 1.3 raza (wolniej niż $O(n^{\frac{3}{4}})$)

Dla n = 10000: 8000 µs -> wzrost czasu o ok. 2 razy (szybciej niż $O(n^{\frac{3}{4}})$)

Sortowanie Quicksort:

Dla n = 1000: 2000 µs

Dla n = 2000: 3000 µs -> wzrost czasu o 1.5 raza (wolniej niż $O(n * \log n)$)

Dla n = 5000: 10000 µs -> wzrost czasu o ok. 3.3 raza (tyle samo co $O(n * \log n)$)

Dla n = 10000: 16000 µs -> wzrost czasu o ok. 1.6 raza (wolniej niż $O(n * \log n)$)

Tak jak w przypadku poprzedniej grupy, szybkość czasów waha się. Jednak mniej więcej wyrównują się do znanych nam klas złożoności.

Z tego względu można stwierdzić, że **klasa złożoności algorytmów grupy II wynosi $\boldsymbol{O(n * log\ n)}$** z wyjątkiem algorytmu Shella, który wynosi mniej więcej $O(n^{\frac{3}{4}})$.

Należy jednak pamiętać, że są to jedynie aproksymacje. Wszystko zależy od środowiska oraz od operacji, które wprowadzają stałe do równania, a te są ignorowane przez klasę złożoności. Jednak daje nam to pewien poglqd i zrozumienie złożoności algorytmów sortowania.