Grupa 3A

Wydział Informatyki i Telekomunikacji

Politechnika Poznańska

Algorytmy i Struktury Danych
Prowadzący: dr inż. Tomasz Żok

Sprawozdanie do
Ćwiczenia 1 – Algorytmy sortowania

Autor:
Dominik Roszkowiak
Nr indeksu 145182

1.Implementacja wybranych metod sortowania.

Do zaimplementowania metod sortowania podanych w treści zadania użyto języka C++. Do mierzenia czasu działania poszczególnych algorytmów posłużono się funkcją clock z biblioteki ctime. Wybrano 15 punktów pomiaru czasu wykonywania algorytmów zaczynając od 10 tys. danych wejściowych, a na 150 tys. kończąc.

2. Tabele z Zależnością czasu t[S] od ilości sortowanych elementów n.

a) Dane wejściowe są uporządkowane **malejąco**

n	Insertion Sort t[S]	Selection Sort t[S]	Heap sort t[S]	Quick Sort* t[S]	Quick Sort** t[S]	Quick Sort*** t[S]
10000	0.218	0.328	0.457	0.230	0.001	0.552
20000	0.815	0.912	0.827	0.716	0.004	2
30000	0.001	1	1	1	0.005	5
40000	0.001	1	2	-	0.006	9
50000	0.001	3	3	-	0.008	14
60000	0.001	4	6	-	0.009	20
70000	0.001	6	8	-	0.012	27
80000	0.001	7	9	-	0.013	35
90000	0.001	10	11	-	0.015	45
100000	0.001	12	15	-	0.017	55
110000	0.001	14	19	-	0.019	67
120000	0.001	17	24	-	0.020	80
130000	0.001	20	31	-	0.023	94
140000	0.001	24	32	-	0.024	109
150000	0.001	28	33	-	0.026	120

* - metoda Quick Sort z podziałem według ostatniego elementu
* *- metoda Quick Sort z podziałem według ostatniego ostatniego
* **-Dodatkowo zaimplementowana iteracyjna metoda Quick Sort

b) Dane wejściowe są uporządkowane **rosnąco**

n	Insertion Sort t[S]	Selection Sort t[S]	Heap sort t[S]	Quick Sort* t[S]	Quick Sort** t[S]	Quick Sort*** t[S]
10000	0	0.001	0.001	0.001	0.001	0.458
20000	1	0.001	0.001	1	0.004	1
30000	2	1	1	2	0.005	3
40000	5	1	2	4	0.007	5
50000	7	2	3	-	0.009	8
60000	11	4	4	-	0.012	12
70000	15	5	6	-	0.013	16
80000	20	7	8	-	0.016	21
90000	25	9	10	-	0.019	27
100000	31	11	12	-	0.020	33
110000	38	13	15	-	0.021	40
120000	46	16	18	-	0.022	48
130000	54	19	21	-	0.026	56
140000	62	22	25	-	0.036	65
150000	72	23	28	-	0.080	73

* - metoda Quick Sort z podziałem według ostatniego elementu
* *- metoda Quick Sort z podziałem według ostatniego ostatniego
* **-Dodatkowo zaimplementowana iteracyjna metoda Quick Sort

c) Dane wejściowe są uporządkowane **losowo**

n	Insertion Sort t[S]	Selection Sort t[S]	Heap sort t[S]	Quick Sort* t[S]	Quick Sort** t[S]	Quick Sort*** t[S]
10000	0.0234	0.314	0.123	0.002	0.002	0.002
20000	0.837	0.918	0.512	0.003	0.004	0.005
30000	1	1	1	0.004	0.007	0.008
40000	2	2	2	0.006	0.010	0.10
50000	3	3	4	0.008	0.012	0.14
60000	5	4	6	0.010	0.014	0.18
70000	7	6	8	0.012	0.017	0.20
80000	10	8	10	0.013	0.020	0.22
90000	12	10	12	0.016	0.025	0.26
100000	15	12	15	0.018	0.026	0.28
110000	19	15	18	0.019	0.029	0.31
120000	22	18	21	0.020	0.034	0.34
130000	26	21	24	0.021	0.054	0.39
140000	30	24	27	0.023	0.059	0.41
150000	35	23	34	0.025	0.062	0.45

* - metoda Quick Sort z podziałem według ostatniego elementu
* *- metoda Quick Sort z podziałem według ostatniego ostatniego
* **-Dodatkowo zaimplementowana iteracyjna metoda Quick Sort

d) Dane wejściowe są **v-kształtne**

n	Insertion Sort t[S]	Selection Sort t[S]	Heap sort t[S]	Quick Sort* t[S]	Quick Sort** t[S]	Quick Sort*** t[S]
10000	0.0234	0.314	0.123	0.002	0.002	0.002
20000	0.837	0.918	0.512	0.003	0.004	0.005
30000	1	1	1	0.004	0.007	0.008
40000	2	2	2	0.006	0.010	0.10
50000	3	3	4	0.008	0.012	0.14
60000	5	4	6	0.010	0.014	0.18
70000	7	6	8	0.012	0.017	0.20
80000	10	8	10	0.013	0.020	0.22
90000	12	10	12	0.016	0.025	0.26
100000	15	12	15	0.018	0.026	0.28
110000	19	15	18	0.019	0.029	0.31
120000	22	18	21	0.020	0.034	0.34
130000	26	21	24	0.021	0.054	0.39
140000	30	24	27	0.023	0.059	0.41
150000	35	23	34	0.025	0.062	0.45

* - metoda Quick Sort z podziałem według ostatniego elementu
* *- metoda Quick Sort z podziałem według ostatniego ostatniego
* **-Dodatkowo zaimplementowana iteracyjna metoda Quick Sort

3.

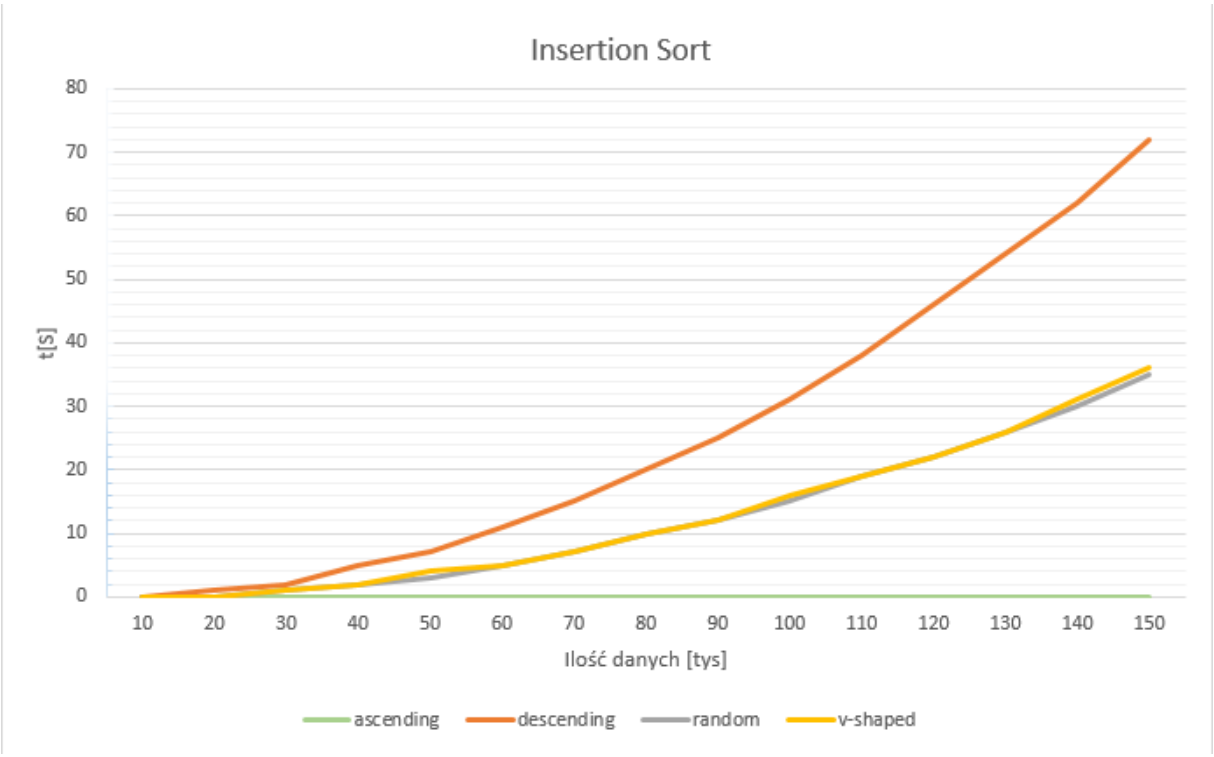
4. Klasa złożoności algorytmów - tabela

Klasa złożoności				Sortowanie	
Nazwa algorytmu	oprtymistyczna	typowa	pesymistyczna	**Stabilność**	**w miejscu**
Selectrion Sort	$O(n^2)$	$O(n^2)$	$O(n^2)$	NIE	TAK
Quick Sort	O(nlog n)	O(nlog n)	$O(n^2)$	NIE	TAK
Heap Sort	O(nlog n)	O(nlog n)	O(nlog n)	NIE	TAK
Insertion Sort	$O(n^2)$	$O(n^2)$	$O(n^2)$	TAK	TAK

5. Różnice w prędkości działania algorytmu w zależności od danych wejściowych oraz ich ilości – wykresy liniowe

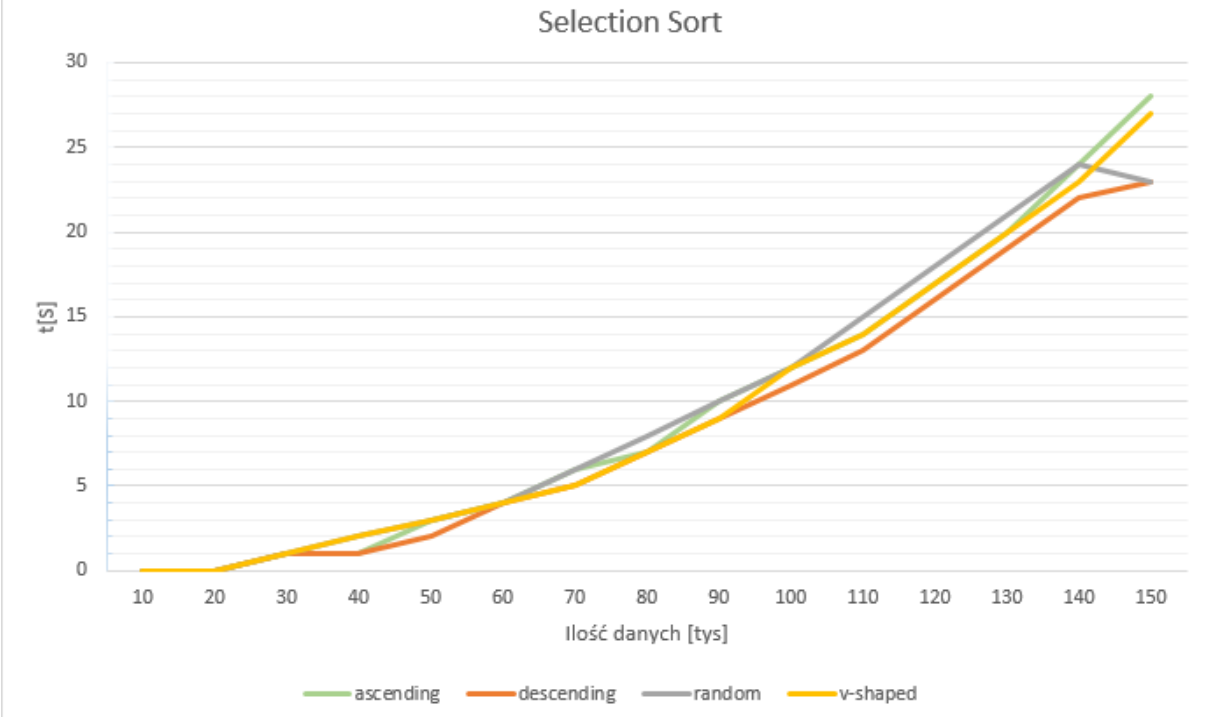
a) Insertion Sort

n[tys]	Increasing t[S]	Decreasing t[S]	Random t[S]	V-Shaped t[S]
10	0.001	0	0.001	0.001
20	0.001	1	0.001	0.001
30	0.001	2	1	1
40	0.001	5	2	2
50	0.001	7	3	4
60	0.001	11	5	5
70	0.001	15	7	7
80	0.001	20	10	10
90	0.001	25	12	12
100	0.001	31	15	16
110	0.001	38	19	19
120	0.001	46	22	22
130	0.001	54	26	26
140	0.001	62	30	31
150	0.001	72	35	36


Insertion Sort
80
70
60
50
40
30
20
10
0
t[s]
10
20
30
40
50
60
70
80
90
100
110
120
130
140
150
Ilość danych [tys]
ascending
descending
random
v-shaped

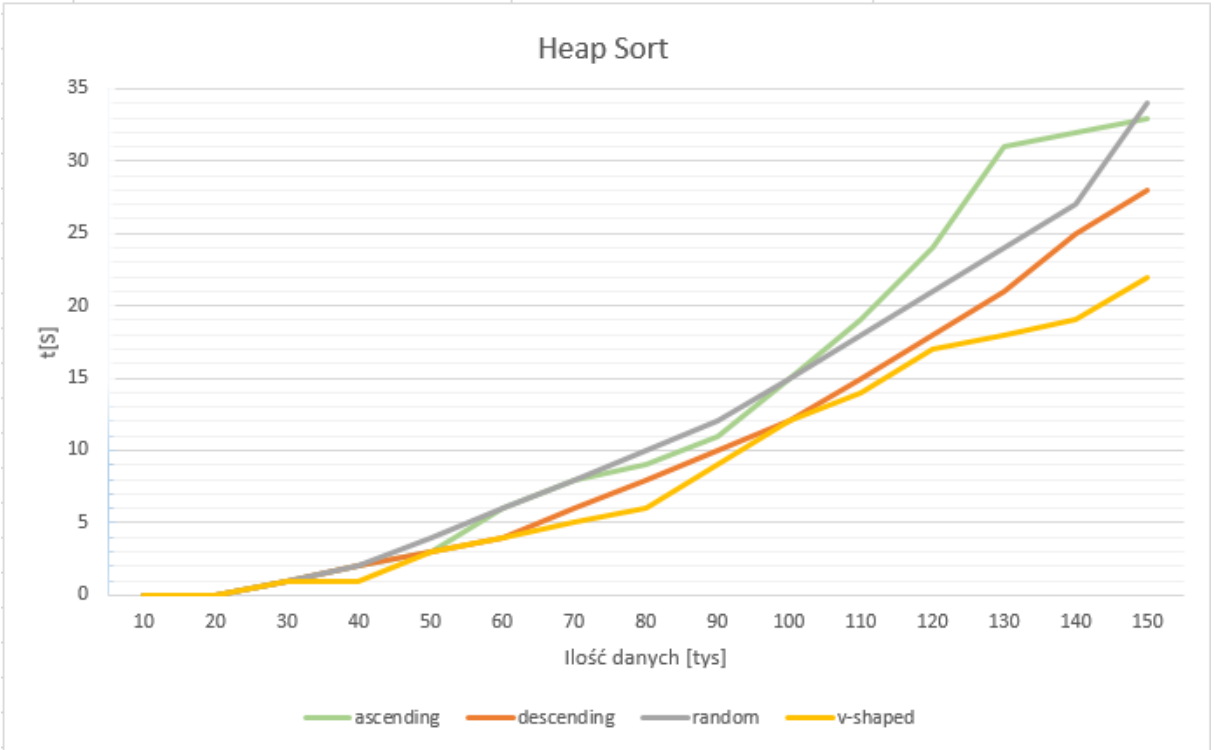
b) Selection Sort

n[tys]	Increasing t[S]	Decreasing t[S]	Random t[S]	V-Shaped t[S]
10	0.001	0.001	0.001	0.001
20	0.001	0.002	0.001	0.001
30	1	1	1	1
40	1	1	2	2
50	3	2	3	3
60	4	4	4	4
70	6	5	6	5
80	7	7	8	7
90	10	9	10	9
100	12	11	12	12
110	14	13	15	14
120	17	16	18	17
130	20	19	21	20
140	24	22	24	23
150	28	23	23	27



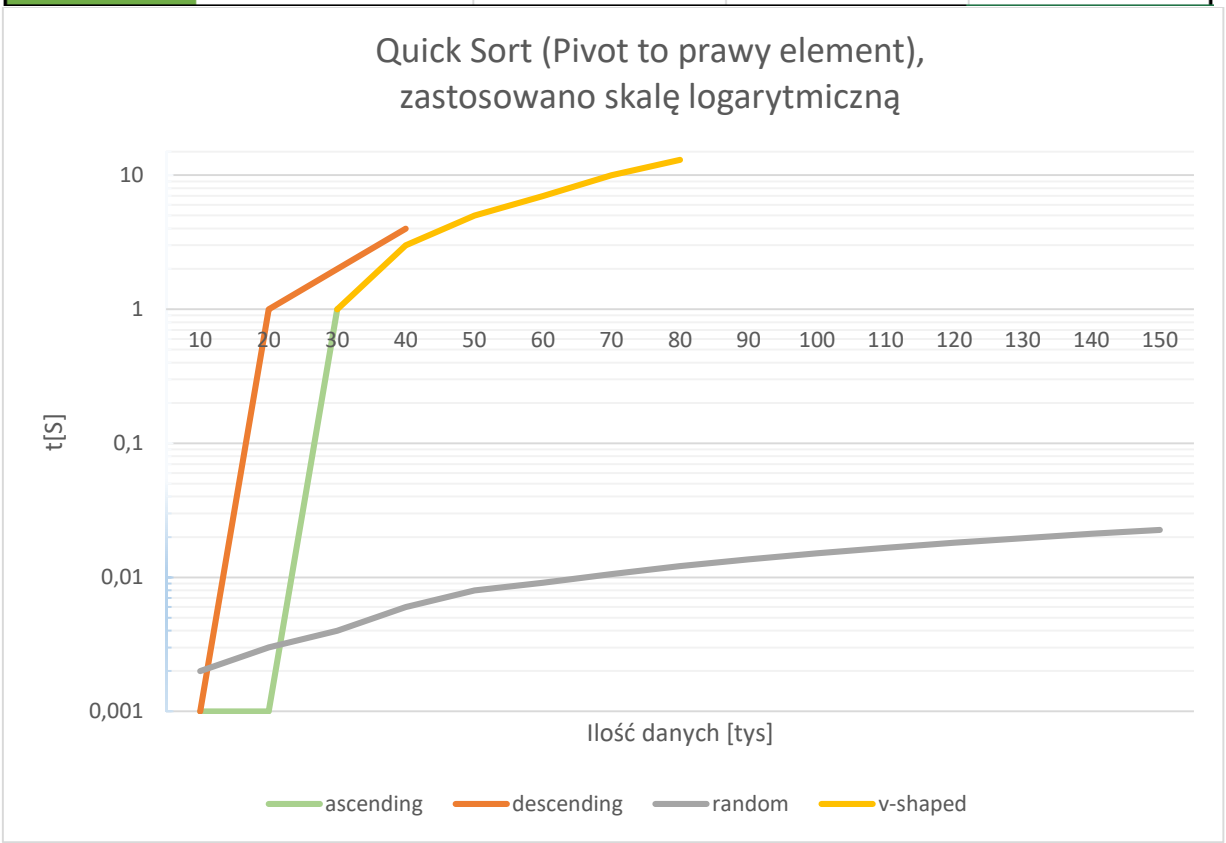
c) Heap Sort

n[tys]	Increasing t[S]	Decreasing t[S]	Random t[S]	V-Shaped t[S]
10	0.001	0.001	0.001	0.001
20	0.002	0.003	0.002	0.004
30	1	1	1	1
40	2	2	2	1
50	3	3	4	3
60	6	4	6	4
70	8	6	8	5
80	9	8	10	6
90	11	10	12	9
100	15	12	15	12
110	19	15	18	14
120	24	18	21	17
130	31	21	24	18
140	32	25	27	19
150	33	28	34	22



d) Quick Sort(Pivot to prawy element)

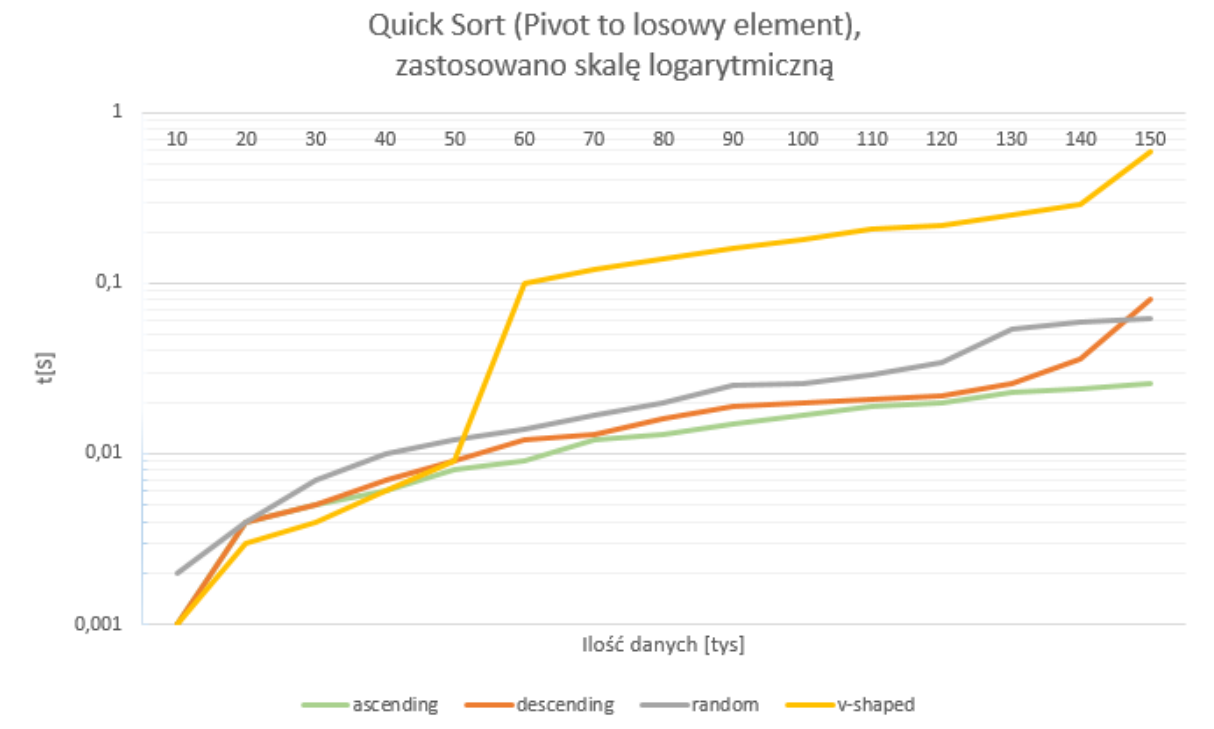
n[tys]	Increasing t[S]	Decreasing t[S]	Random t[S]	V-Shaped t[S]
10	0.001	0.001	0.002	0.219
20	0.001	1	0.003	0.918
30	1	2	0.004	1
40	-	4	0.006	3
50	-	-	0.008	5
60	-	-	0.010	7
70	-	-	0.012	10
80	-	-	0.013	13
90	-	-	0.016	-
100	-	-	0.018	-
110	-	-	0.019	-
120	-	-	0.020	-
130	-	-	0.021	-
140	-	-	0.023	-
150	-	-	0.025	-



Warto zauważyć, że stos został przepełniony w aż 3 na 4 prezentowane przypadki; W przypadku danych rosnących wystarczyło 30 tys. danych wejściowych, aby osiągnąć przepełnienie.

e) Quick Sort(pivot to losowy element)

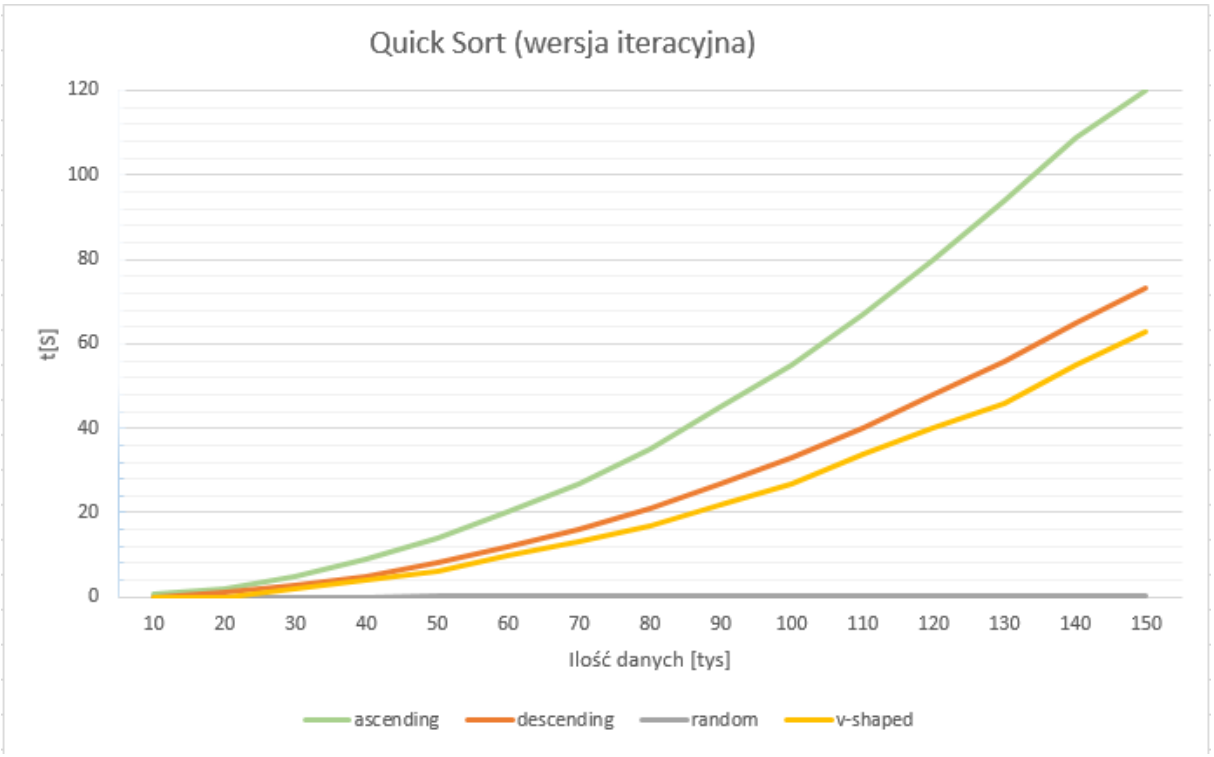
n[tys]	Increasing t[S]	Decreasing t[S]	Random t[S]	V-Shaped t[S]
10	0.001	0.001	0.002	0.001
20	0.004	0.004	0.004	0.003
30	0.005	0.005	0.007	0.004
40	0.006	0.007	0.010	0.006
50	0.008	0.009	0.012	0.009
60	0.009	0.012	0.014	0.11
70	0.012	0.013	0.017	0.13
80	0.013	0.016	0.020	0.14
90	0.015	0.019	0.025	0.16
100	0.017	0.020	0.026	0.18
110	0.019	0.021	0.029	0.21
120	0.020	0.022	0.034	0.22
130	0.023	0.026	0.054	0.25
140	0.024	0.036	0.059	0.29
150	0.026	0.080	0.062	0.59



Warto zauważyć, że ani razu nie trafiono na przypadek pesymistyczny, bardzo rzadko zdarza się, aby w wersji quick sorta z losowo dobieranym pivotem taki przypadek uzyskać, ponieważ zależy to nie tylko od danych wejściowych ale również od funkcji, która losuje pozycję pivota.

f) Quick Sort(wersja iteracyjna)*

n[tys]	Increasing t[S]	Decreasing t[S]	Random t[S]	V-Shaped t[S]
10	0.552	0.458	0.002	0.372
20	2	1	0.005	1.128
30	5	3	0.008	2
40	9	5	0.10	4
50	14	8	0.14	6
60	20	12	0.18	10
70	27	16	0.20	13
80	35	21	0.22	17
90	45	27	0.26	22
100	55	33	0.28	27
110	67	40	0.31	34
120	80	48	0.34	40
130	94	56	0.39	46
140	109	65	0.41	55
150	120	73	0.45	63



*wersja algorytmu dodana dodatkowo do sprawozdania
Warto zauważać, że wersja iteracyjna w przeciwieństwie do rekurencyjnej z pivotem na ostatnim elemencie nie powoduje przepełnienia stosu nawet przy 150 tys. danych wejściowych.

7.Wnioski:

a) Na efektywność algorytmów sortowania wpływa liczba sortowanych elementów, zakres elementów, złożoność obliczeniowa oraz rozkład. Podane wyżej algorytmy można podzielić ze względu na wrażliwość na dane wejściowe, stabilność i złożoność obliczeniową.

b) Insertion Sort okazał się bardzo efektywny w stosunku do danych uporządkowawnych malejąco, Nawet dla dużej wejściowej ilości danych. Niestety algorytm ten dla pozostałych typów danych wejściowych nie jest już tak skuteczny i jego czas działania był o wiele dłuższy od np. algorytmu Heap Sort. W związku z powyższym algorytm jest wart rozważenia tylko wtedy, gdy sortowane dane będą wersja optymistyczną.
Przypadkiem pesymistycznym są dane wejściowe uporządkowane malejąco
Przypadkiem średnim są losowe dane oraz rozkład V-kształtny złożoność obliczeniowa wynosi wtedy O(n2).
Przypadkiem optymistycznym są dane uporządkowane rosnąco.

Złożoność obliczeniowa tego algorytmu dla każdego przypadku wynosi O(n2).

c) Selection Sort jest algorytmem mało wrażliwym na rodzaj posortowania danych wejściowych. Algorytm dla losowego, malejącego, rosnącego i v-kształtnego układu elementów posiada podobny czas porządkowania danych. **Wykonuje taką samą liczbę porównań niezależnie od tego czy ciąg danych jest posortowany czy też nie. Złożoność obliczeniowa tego algorytmu jest taka sama dla najgorszego i najlepszego przypadku i wynosi O(n2).**

e) Heap Sort jest algorytmem sortowania przez kopcowanie. Jest on bardziej rozbudowany od Quick Sort, Jedyną przewagą Heap Sort nad szybkim sortowaniem jest lepszy pesymistyczny przypadek. **Klasa złożoności tego algorytmu dla każdego przypadku wynosi O(n2).**

f) Algorytm rekurencyjny Quick Sort powoduje przepełnienie stosu i program powyżej 40 tysięcy danych wejściowych zawiesza się. Sytuacja ta miała miejsce podczas sortowania danych uporządkowanych rosnąco albo malejąco, ponieważ dla wersji algorytmu z pivotem na końcu jest to najgorszy przypadek. Ciekawostką może być, że ilość danych wejściowych po których następuje przepełnienie stosu różni się w zależności od języka programowania np. W języku python przepełnienie stosu na mojej platformie nastąpiło już po tysiącu danych wejściowych.

g) Algorytm Quick Sort w wersji z pivotem wybieranym losowo ma znacznie mniejsze szanse na natrafienie na przypadek pesymistyczny. W przypadku sortowania danych ułożonych rosnąco albo malejąco jest on znacznie szybszym wyborem od swojego odpowiednika z pivotem na końcu. Jednak w przypadku danych wejściowych dobranych losowo algorytm z pivotem dobranym losowo jest zawsze wolniejszy, ponieważ losowanie pivota wiąże się z dodatkowymi operacjami do wykonania. Sytuację taką widać w punkcie 2.C, gdzie algorytm z elementem rozdzielającym jest około dwa razy wolniejszy.

h) Przypadkiem pesymistycznym dla algorytmu Quick Sort z prawym elementem rozdzielającym są dane uporządkowane rosnąo. Jego klasa złożoności wynosi wtedy O(n2). Przypadkiem optymistycznym jest

rozkład losowy. Klasa złożoności dla przypadku średniego i optymistycznego wynosi O(nlog n).

i) **W przypadku algorytmu Quick Sort z losowym elementem rozdzielającym klasy złożoności dla przypadku optymistycznego, typowego i pesymistycznego pozostają takie same, jak w przypadku Quick Sort z ostatnim elementem rozdzielającym.** Zaletą wersji z losowym pivotem jest mała szansa na otrzymanie przypadku pesymistycznego, ponieważ uzyskanie takiego przypadku w dużej mierze zależy od generatora liczb losowych. Przypadek pesymistyczny może powstać np. gdy generator liczb za każdym razem ustawi pivot na końcu, a dane wejściowe będą rosnące, lecz szansa na otrzymanie takiego przypadku jest minimalna.

j) **Heap Sort – Algorytm, którego klasa złożoności dla każdego przypadku wynosi O(nlog n),** w związku z czym nie jest on wrażliwy na dane wejściowe. Algorytm dużo szybszy od IS i SS, praktycznie nie posiada pesymistycznego przypadku, w czym przewyższa każdą wersję QS.

K) Algorytmy należące do wolnych np. Selection Sort i Insertion Sort nie są dobrą opcją w przypadku dużej ilości danych wejściowych, lecz ze względu na ich prostszą implementację można je rozważyć w przypadku małej ilości danych wejściowych.

L) Najoptymalniejszym wyborem do sortowania dużej ilości danych wejściowych jest algorytm Quick Sort z losowo wybieranym elementem rozdzielającym, ponieważ szansa trafienia na jego pesymistyczne dane wejściowe jest bardzi niewielka. Jeśli mamy pewność, że dane wejściowe nie będą posortowane rosnąco, to warto użyć Quick Sort z pivotem po prawej stronie. W przypadku gdy zależy nam zarówno na tym, aby rodzaj posortowania danych wejściowych miał mały wpływ na nasz algorytm, jak i na prędkości działania należałoby użyc Heap Sort, ponieważ posiada on tę samą klasę złożoności dla każdego przypadku.