WYŻSZA SZKOŁA MENEDŻERSKA

INSTYTUT NAUK O ZARZĄDZANIU I JAKOŚCI

INFORMATYKA

Algorytmy i złożoność

Praca zaliczeniowa

Sortowanie – zadanie 1

Skład zespołu/Student	**Prowadzący zajęcia:**	**Semestr**
Tomasz Jan Oksiędzki	Marcin Paprzycki	**III**
	Grupa	**Studia**
	49DR – A1	**Niestacjonarne**

Data wykonania	Data oddania:
2020-12-20	2020-12-20

Spis treści

Wprowadzenie

Przedmiotem tego opracowania jest analiza trzech rodzajów sortowanie i ich zachowania w zależności od zbioru danych wejściowych. Sortowanie, które będzie przedmiotem analizy jest bardzo kosztowną (jeżeli chodzi o złożoność algorytmiczną ale również i czasowo) operacją dość powszechnie wykorzystywaną w praktyce w różnych przedmiotach i zagadnieniach informatycznych. Ze względu na powyższy aspekt bardzo istotne jest dobranie efektywnej metody sortowania do danych tak, aby ograniczyć niepotrzebne zużycie zasobów.

Szczegółowym celem opracowania będzie sprawdzenia jak efektywnie sortowane są zbiory liczb losowych, posortowanych rosnąco oraz malejąco z zastosowaniem trzech algorytmów wyszukiwania: Selection sort, Insertion sort oraz Merge sort (w raporcie działanie opisywane jest jako eksperyment I). Dodatkowym aspektem jest sprawdzenie efektywności metod Insertion sort and Merge sort na posortowanym zbiorze poszerzonym o obserwacje losowe (w raporcie opisany jako eksperyment II).

Po zapoznaniu się z podstawowym opisem każdego z trzech algorytmów sortowania można oczekiwać, iż w eksperymencie I najbardziej efektywną metodą będzie w generalnie metoda merge sort. Aczkolwiek jak to się będzie przestawiało w zależności od danych wejściowych i długości analizowanych danych zostanie przedstawione w dalszej części raportu, zaś w eksperymencie II, można oczekiwać, iż wystarczy wydłużenie posortowanego wektora o niewielką część losowych liczb, co będzie skutkowało znacznym wydłużeniem czasu Insertion sort.

Przeprowadzenie eksperymentu I

Założenia

Do przeprowadzenia eksperymentu przyjęto następujące założenia:

- Zbiór danych składa się z losowo wygenerowanych nieposortowanych liczb zmiennoprzecinkowych z przedziału 0-1.
- Algorytmy sortujące mają za zadanie dokonać sortowania rosnąco zbioru wejściowego.
- Porównane zostaną trzy algorytmy sortowania:
  - Selection Sort
  - Insertion Sort
  - Merge Sort.
- Porównanie nastąpi na pięciu punktach pomiarowych wyznaczonych na podstawie analizy możliwej liczebności zbiorów wejściowych zapewniającej czasową mierzalność sortowania.
- Dla każdego z punktów pomiarowych zostanie przeprowadzonych 5 eksperymentów.

Eksperyment - opis

Środowisko

Element środowiska	Parametry
Procesor	Inter® Core™ i7-7600 CPU @ 2.80 GHz 2.90 GHZ
RAM	8 GB
System	Windows 10 Enterprise version 1809
Dysk	SSD Samsung PM961 256 GB M.2 2280 PCI-E x4 Gen3 NVMe (MZVLW256HEHP-00000)
Język programowania	Python 3.8.2 32-bit
IDE	Microsoft Visual Studio Community 2019 Version 16.8.2

Tabela 1: Specyfikacja środowiska, na którym przeprowadzano eksperyment[1]

Opis algorytmów sortowania

Na potrzeby eksperymentu wykorzystano trzy metody Selection Sort, Insertion Sort oraz Merge Sort, które zostaną przedstawione w bardziej szczegółowy sposób w dalszej części opracowania:

Selection sort

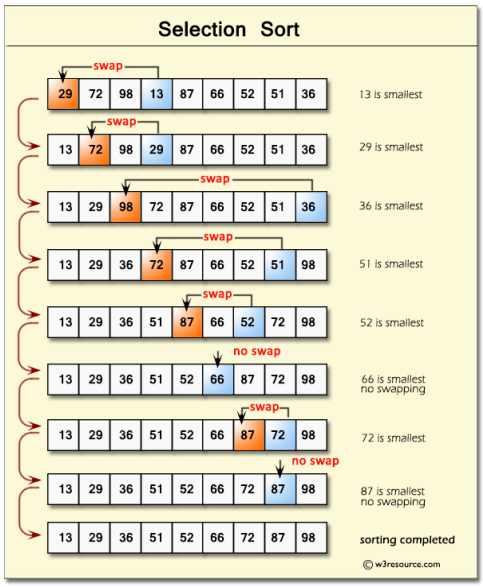
Selection sort[2] w tłumaczeniu określane jest poprzez sortowanie przez wybór. Jest to jedna z prostszych metod sortowania, która cechuje się złożonością $O(n^2)$, gdzie n określa liczbę elementów zbioru. Metoda sortowania przez wybieranie polega na wyszukaniu w zbiorze elementu, który powinien co do kolejności znajdować się na żądanej pozycji i zamienienie go z tym elementem, który obecnie ją zajmuje. Taką operację wykonuje się do momentu aż pozostały zbiór ma jeden element.

Przedstawiając tą metodę bardziej algorytmicznie:

1. Wyszukaj minimalną wartość zbioru spośród elementów od bieżącego do końca zbioru
2. Zamień wartość minimalną wyznaczoną w punkcie 1 z bieżącym elementem.

Metodę sortowania Selection Sort dla przejrzystości przedstawiamy graficznie:

[1] Opracowanie własne
[2] https://www.tutorialspoint.com/data_structures_algorithms/selection_sort_algorithm.htm



Rysunek 1: Przykład sortowania z wykorzystaniem metody Selection sort[3]

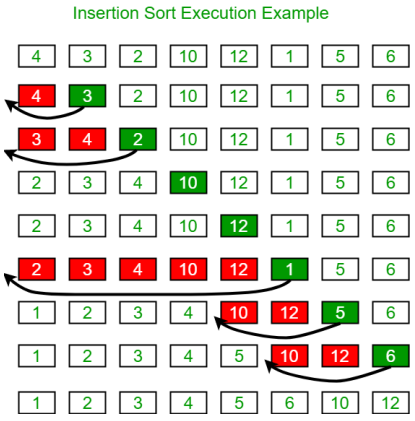
Insertion sort

Insertion sort[4] w tłumaczeniu na język polski to metoda sortowania przez wstawianie. Jej nazwa odnosi się do sposobu układania kart podczas partii gry – tzn. układanie każdej karty w odpowiednie miejsce względem „wartości". Sortowanie przez wstawianie cechuje się złożonością $O(n^2)$, gdzie n określa liczbę elementów zbioru, czyli analogiczną do Selection sort. Insertion sort jest efektywny na małych zbiorach danych, zaś wraz ze wzrostem liczebności zbioru jego efektywność maleje. Warto zauważyć, iż jest to bardzo efektywna metoda w przypadku wykorzystania na zbiorze już posortowanym, gdyż wtedy wymagane jest tylko $n - 1$ operacji porównania.

Przedstawiając tą metodę bardziej algorytmicznie:

1. Dzielimy zbiór na część posortowaną (na początku jest to pierwszy element zbioru) i nieposortowaną.
2. W każdym kolejnym kroku bierzemy pierwszy element z części nieposortowanej i wstawiamy we właściwe miejsce w części posortowanej (poprzez porównanie z ostatnim i poprzednimi elementami posortowanej części zbioru).

Metodę można również dla uproszczenia przedstawić graficznie:

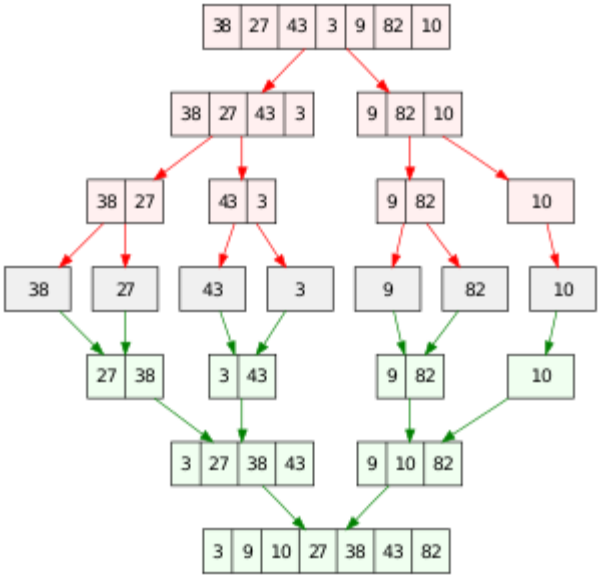


[3] https://www.w3resource.com/php-exercises/searching-and-sorting-algorithm/searching-and-sorting-algorithm-exercise-4.php
[4] http://www.algorytm.edu.pl/algorytmy-maturalne/sortowanie-przez-wstawianie.html

Rysunek 2: Przykład sortowania z wykorzystaniem metody Insertion sort[5].

Merge sort

Merge sort można przetłumaczyć, jako metodę sortowania przez złączanie, jest to w efektywniejszy algorytm sortowania od powyższych cechujący się złożonością czasową $n \log_2(n)$. Algorytm ten jest rekurencyjną metodą typu dziel i zwyciężaj, którego ideą jest podzielenie zbioru na mniejsze zbiory aż do uzyskania jednoelementowych zbiorów i następnie łączenie ich w posortowane liczniejsze zbiory.



Rysunek 3: Przykład sortowania z wykorzystaniem metody Merge sort[6].

Generowanie danych losowych

Jako dane do przeprowadzenia eksperymentu wykorzystano polecenie języka Python random.random(), które generuje liczby losowe z przedziału (0, 1) w formacie float.

Wyznaczenie punktów pomiarowych

Celem eksperymentu jest przeprowadzenie doświadczenia na określonych puntach pomiarowych i odpowiednich zbiorach. Do wyznaczenia pięciu punktów pomiarowych wyznaczono eksperymentalnie długości wektorów, dla których każda z trzech metod wyszukiwania trwa minimum jedną sekundę i około 5 minut dla każdego z trzech rodzajów danych wejściowych. Jak przedstawiają się wyniki eksperymentalnego wyznaczenia liczebności wektorów przedstawia poniższa tabela:

Rodzaj danych wejściowych	Metoda sortowania	Długość wektora, żeby sortowanie trwało ok 1 sekundy	Długość wektora, żeby sortowanie trwało ok 5 minut
Wektor danych losowych	Selection Sort	1 500 elementów	25 000 elementów
	Insertion Sort	1 600 elementów	26 00 elementów
	Merge Sort	24 000 elementów	4 500 000 elementów
Wektor danych posortowanych rosnąco	Selection Sort	1 700 elementów	26 000 elementów
	Insertion Sort	500 000 elementów	-[7]
	Merge Sort	24 000 elementów	4 700 000 elementów
	Selection Sort	1 600 elementów	23 000 elementów

[5] https://www.geeksforgeeks.org/insertion-sort/
[6] https://en.wikipedia.org/wiki/Merge_sort
[7] Przy 4 500 000 elementach czas sortowania Insertion Sort zajmował ok 11 sekund – nie weryfikowano dalej.

Wektor danych posortowanych malejąco	Insertion Sort	1 300 elementów	20 000 elementów
	Merge Sort	24 000 elementów	4 700 000 elementów

Tabela 2: Wyniki pomiarów długości wektorów wejściowych w zależności od metody sortowania[8]

Biorąc pod uwagę powyższe wyniki czasowe sortowania wektorów, przyjęto do dalszego eksperymentu pięciu punktów pomiarowych, które zostały wyznaczone na wektorze o długości 30000 elementów, dzięki czemu będzie możliwe przeprowadzenie obserwacji na zauważalnie mierzalnych danych. W ten sposób wyznaczono przedziały przedstawione w poniższej tabeli:

Numer przedziału	Pierwszy element	Ostatni element
1	0	6 000
2	6 001	12 000
3	12 001	18 000
4	18 001	24 000
5	24 001	30 000

Tabela 3: Punkty pomiarowe i ich przedziały[9]

Procedura eksperymentu

Eksperyment polega na wykonaniu pięciokrotnego sortowania każdego z pięciu powyżej określonych odcinków każdym z trzech opisanych algorytmów. Powyższa procedura zostanie uruchomiona na 3 rodzajach danych wejściowych – wektor liczb losowych, wektor liczb posortowanych rosnąco oraz wektor liczb posortowanych malejąco. Procedura została uruchomiona poprzez Microsoft Visual Studio Community 2019. Następnie na podstawie otrzymanych wyników policzono średni czas algorytmu $\bar{X}$ oraz odchylenie standardowe σ wykorzystując poniższe wzory[10]:

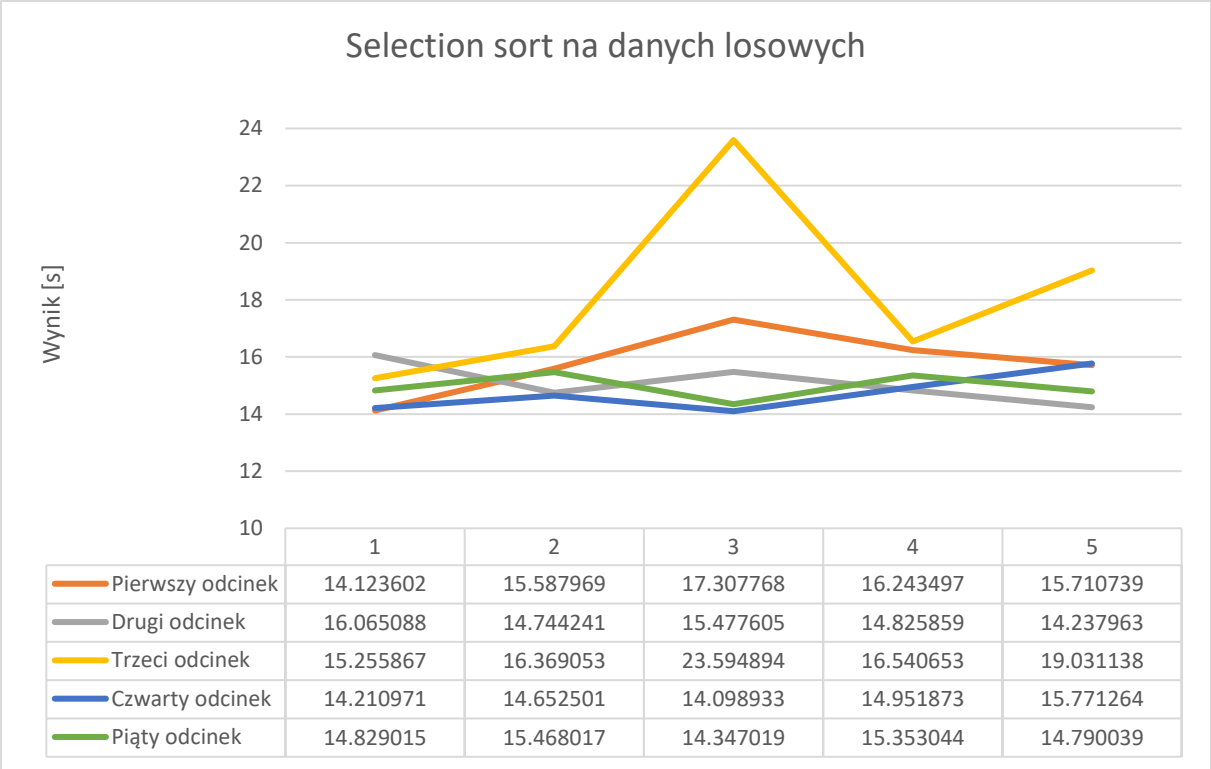
$$\bar{X} = \frac{\sum_{i=1}^{n} x_i}{n}, gdzie\ n - liczba\ obserwacji, x_i - wynik\ obserwacji$$

$$\sigma = \sqrt{\frac{\sum_{i=1}^{n}(x_i - \bar{X})^2}{n}}, gdzie\ n - liczba\ obserwacji, x_i - wynik\ obserwacji, \bar{X} - średnia$$

Wyniki z eksperymentu

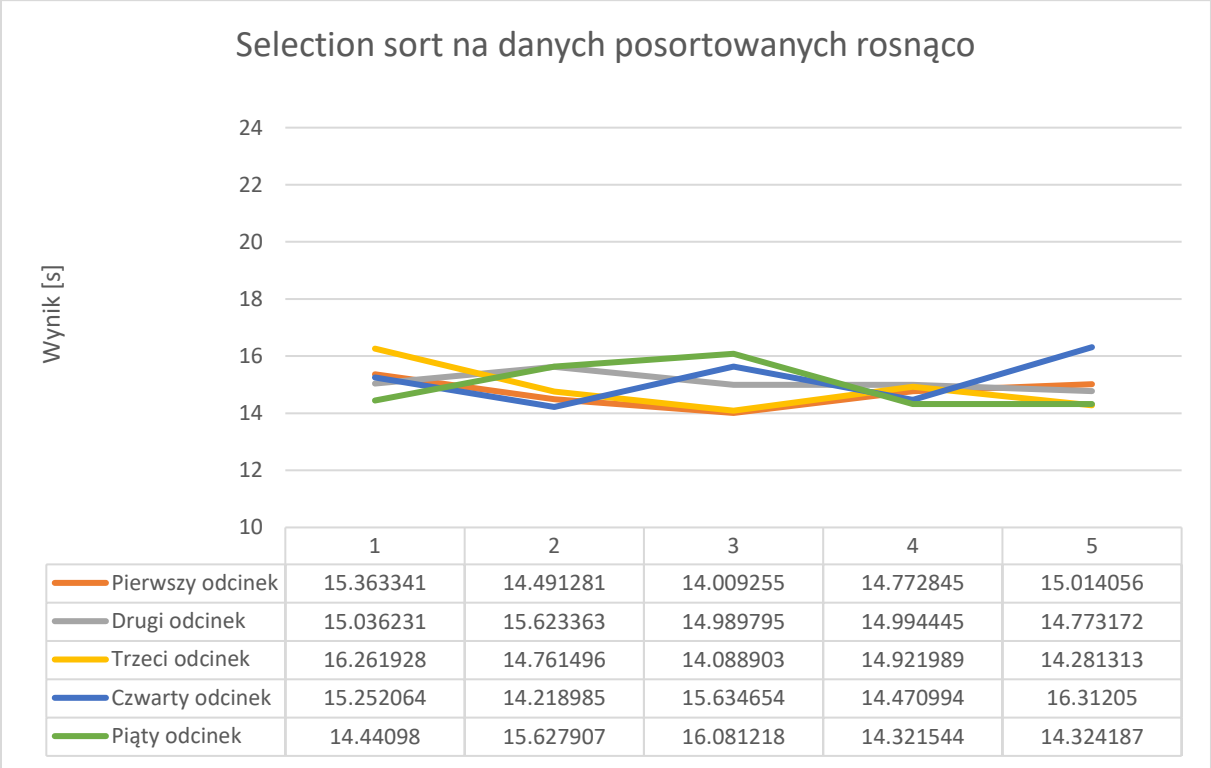
Pełne wyniki eksperymentu zostały dołączone do pracy pod postacią jednego z załączników. Wyniki czasowe konkretnych metod na poszególnych odcinach pomiarowych zostały przedstawione poniżej w formie graficznej:

[8] Opracowanie własne
[9] Opracowanie własne
[10] Średnia arytmetyczna: https://www.matemaks.pl/srednia-arytmetyczna.html, odchylenie standardowe: https://www.matemaks.pl/odchylenie-standardowe.html



	1	2	3	4	5
Pierwszy odcinek	14.123602	15.587969	17.307768	16.243497	15.710739
Drugi odcinek	16.065088	14.744241	15.477605	14.825859	14.237963
Trzeci odcinek	15.255867	16.369053	23.594894	16.540653	19.031138
Czwarty odcinek	14.210971	14.652501	14.098933	14.951873	15.771264
Piąty odcinek	14.829015	15.468017	14.347019	15.353044	14.790039

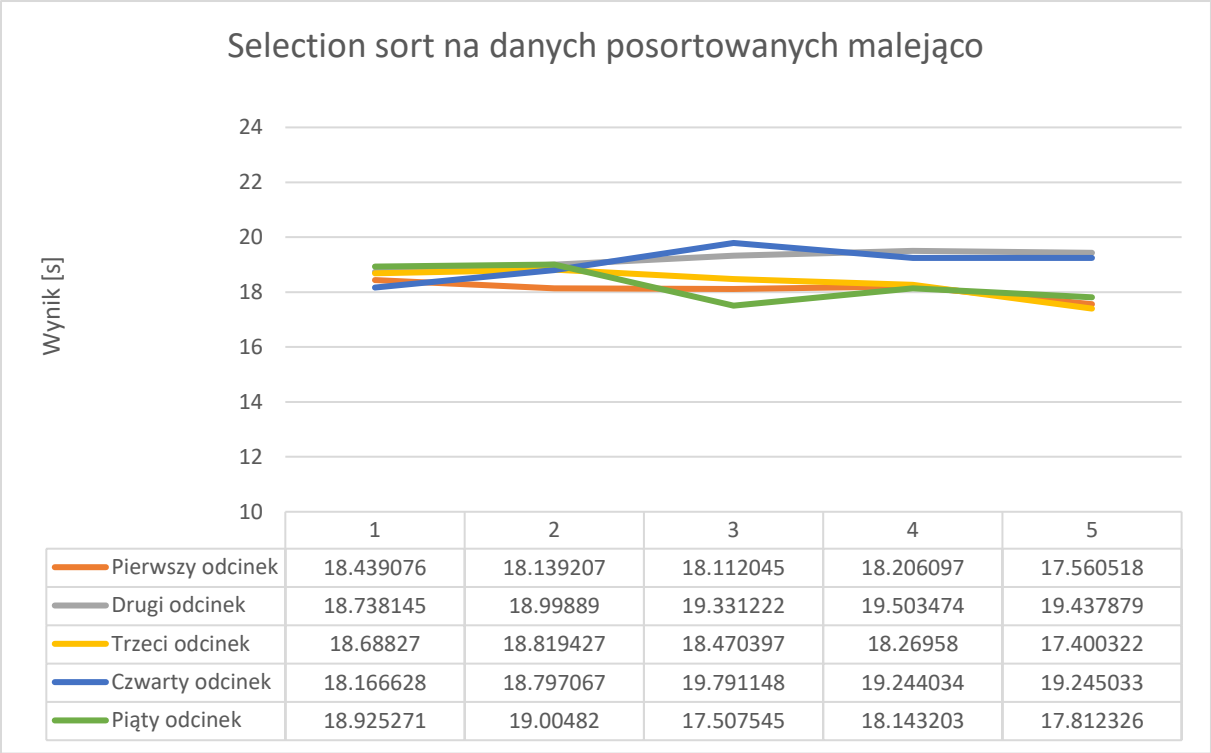
Rysunek 1: Wyniki Selection sort na danych losowych dla każdej z prób na odcinkach pomiarowych[11]



	1	2	3	4	5
Pierwszy odcinek	15.363341	14.491281	14.009255	14.772845	15.014056
Drugi odcinek	15.036231	15.623363	14.989795	14.994445	14.773172
Trzeci odcinek	16.261928	14.761496	14.088903	14.921989	14.281313
Czwarty odcinek	15.252064	14.218985	15.634654	14.470994	16.31205
Piąty odcinek	14.44098	15.627907	16.081218	14.321544	14.324187

Rysunek 2: Wyniki Selection sort na danych posortowanych rosnąco dla każdej z prób na odcinkach pomiarowych [12]

[11] Opracowanie własne
[12] Opracowanie własne



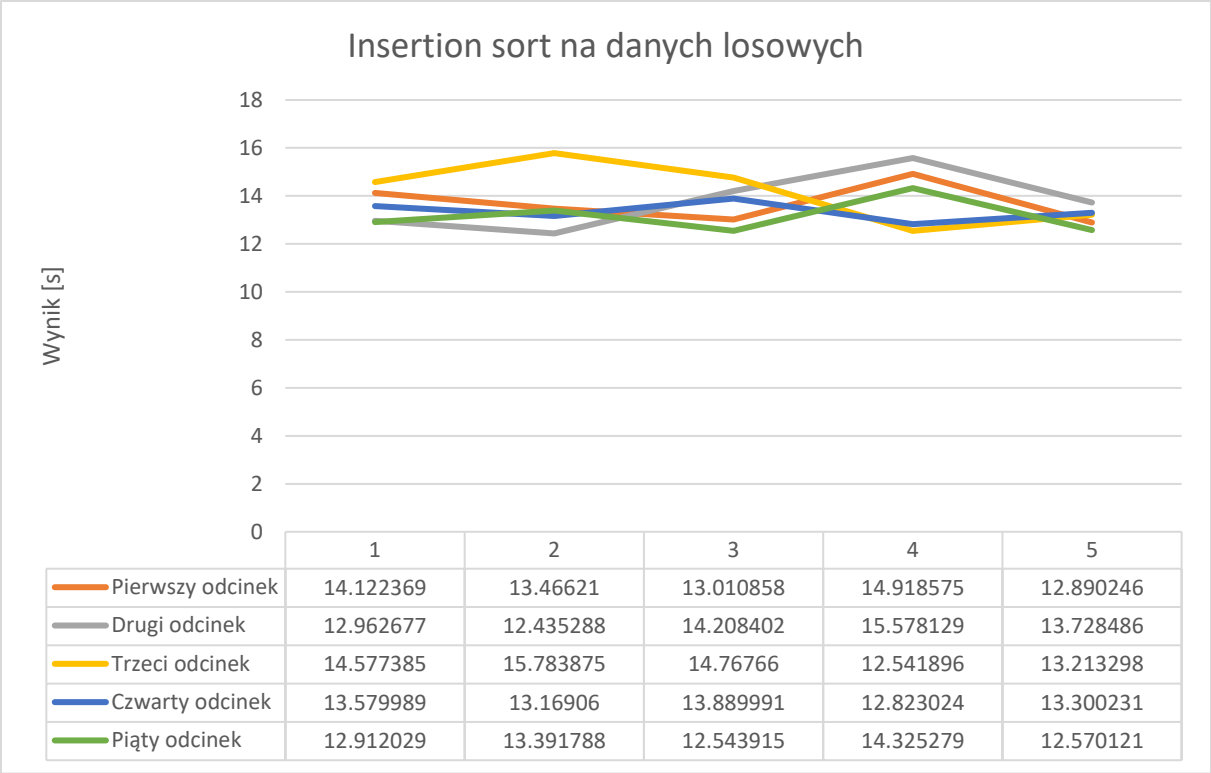
	1	2	3	4	5
Pierwszy odcinek	18.439076	18.139207	18.112045	18.206097	17.560518
Drugi odcinek	18.738145	18.99889	19.331222	19.503474	19.437879
Trzeci odcinek	18.68827	18.819427	18.470397	18.26958	17.400322
Czwarty odcinek	18.166628	18.797067	19.791148	19.244034	19.245033
Piąty odcinek	18.925271	19.00482	17.507545	18.143203	17.812326

Rysunek 3: Wyniki Selection sort na danych posortowanych malejąco dla każdej z prób na odcinkach pomiarowych [13]

Z powyższych wykresów można zauważyć, iż sortowanie za pomocą metody Selection sort na wszystkich wektorach i odcinkach trwa od ok 14 do 20 sekund (jest jedna obserwacja odstająca ok 23 sekundy, co jednakże może wskazywać na chwilowe obciążenie zasobów maszyny testowej) i zachowuje się w miarę stabilnie. Warte podkreślenia jest to, iż wyniki na wektorze posortowanym malejąco cechują się najwyższą wizualnie średnią, co jest zgodne z odczuciami, co do tej metody sortującej. Również widać, iż wyniki pomiarowe metody Selection sort na danych losowych cechują się największą zmiennością.

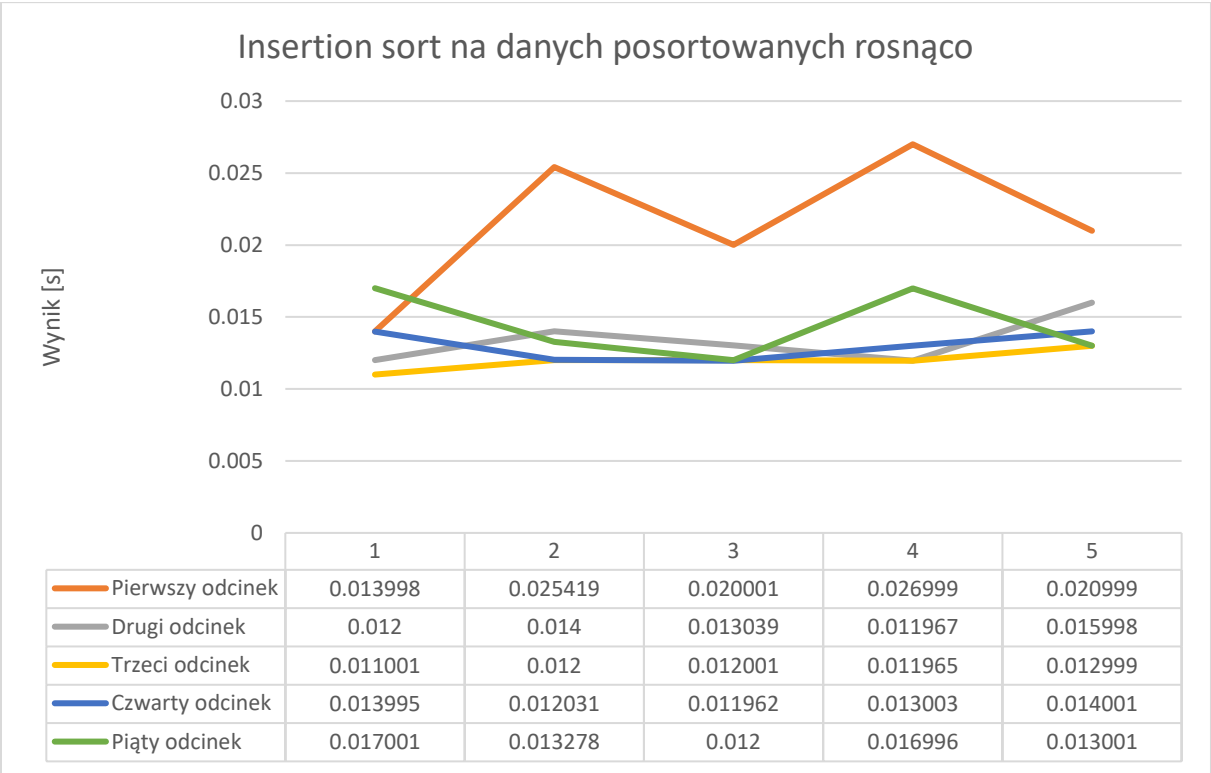
Poniżej alogiczne graficzne przedstawienie dla wyników metody Insertion sort (wartości osi pionowych pomiędzy wykresami nie są konsystentne):

[13] Opracowanie własne



	1	2	3	4	5
Pierwszy odcinek	14.122369	13.46621	13.010858	14.918575	12.890246
Drugi odcinek	12.962677	12.435288	14.208402	15.578129	13.728486
Trzeci odcinek	14.577385	15.783875	14.76766	12.541896	13.213298
Czwarty odcinek	13.579989	13.16906	13.889991	12.823024	13.300231
Piąty odcinek	12.912029	13.391788	12.543915	14.325279	12.570121

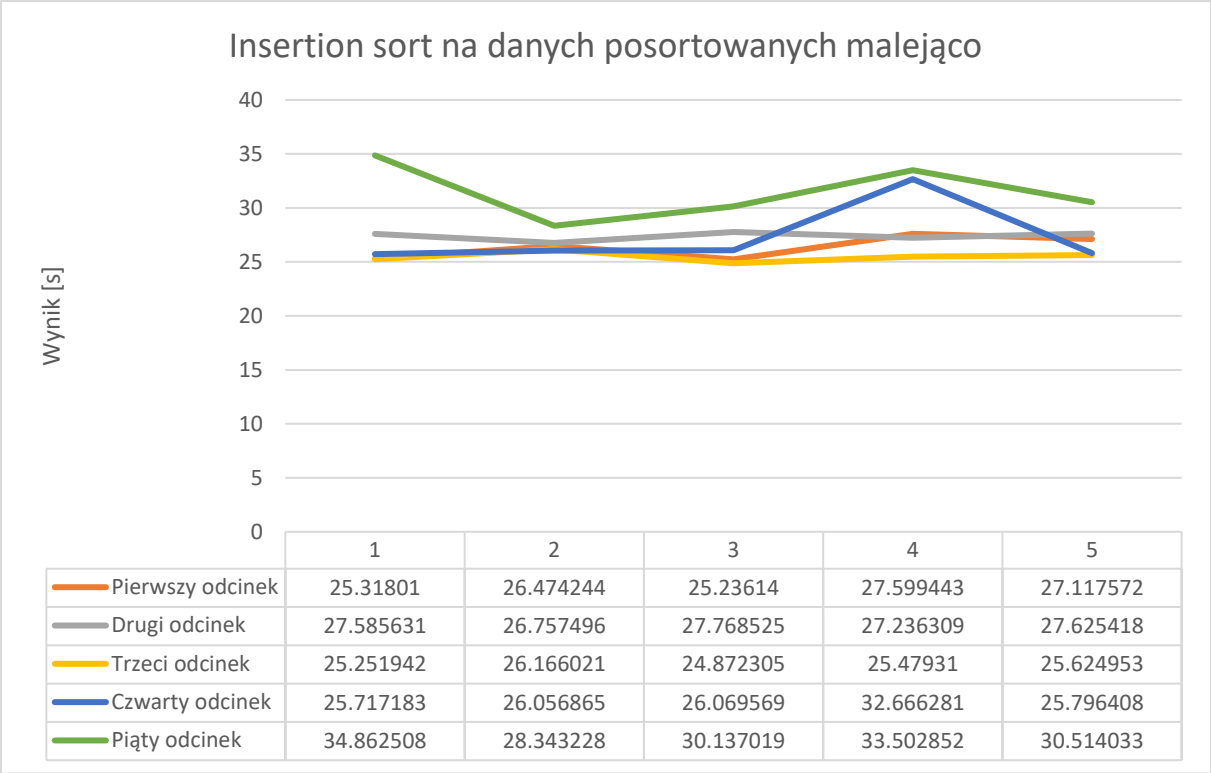
Rysunek 4: Wyniki Insertion sort na danych losowych dla każdej z prób na odcinkach pomiarowych[14]



	1	2	3	4	5
Pierwszy odcinek	0.013998	0.025419	0.020001	0.026999	0.020999
Drugi odcinek	0.012	0.014	0.013039	0.011967	0.015998
Trzeci odcinek	0.011001	0.012	0.012001	0.011965	0.012999
Czwarty odcinek	0.013995	0.012031	0.011962	0.013003	0.014001
Piąty odcinek	0.017001	0.013278	0.012	0.016996	0.013001

Rysunek 5: Wyniki Insertion sort na danych posortowanych rosnąco dla każdej z prób na odcinkach pomiarowych[15]

[14] Opracowanie własne
[15] Opracowanie własne

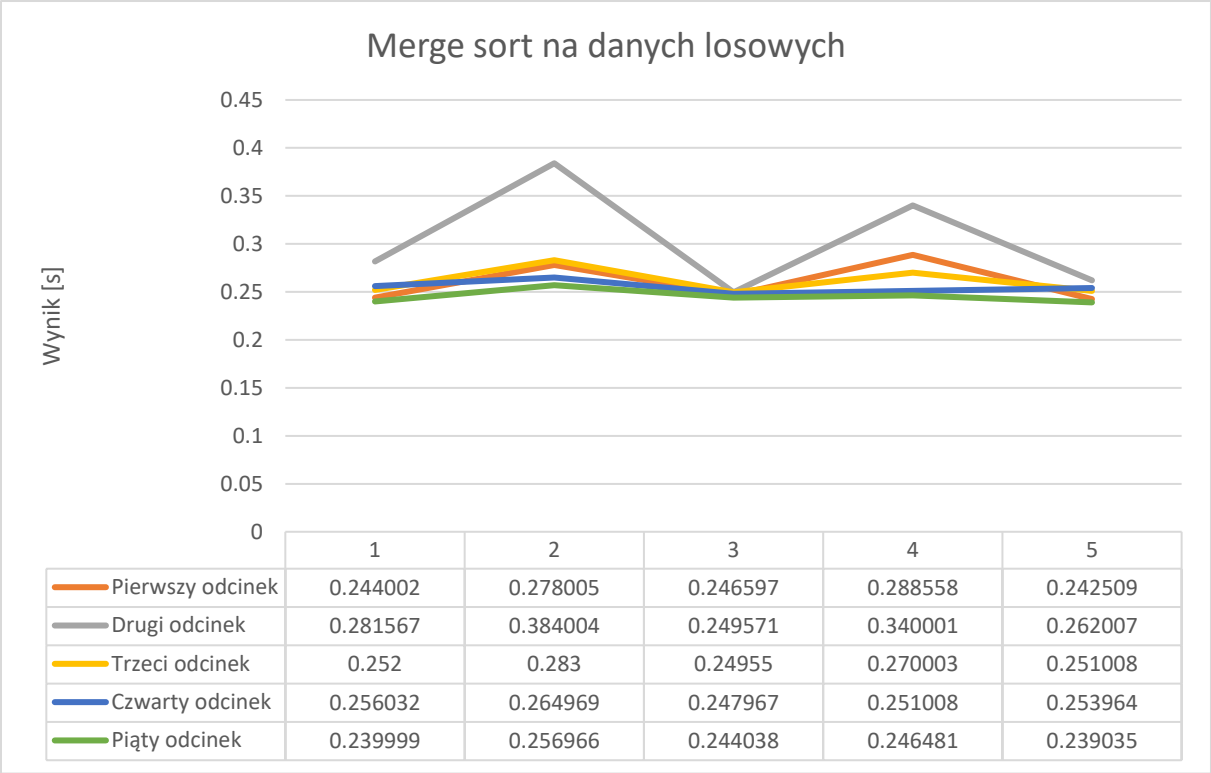


	1	2	3	4	5
Pierwszy odcinek	25.31801	26.474244	25.23614	27.599443	27.117572
Drugi odcinek	27.585631	26.757496	27.768525	27.236309	27.625418
Trzeci odcinek	25.251942	26.166021	24.872305	25.47931	25.624953
Czwarty odcinek	25.717183	26.056865	26.069569	32.666281	25.796408
Piąty odcinek	34.862508	28.343228	30.137019	33.502852	30.514033

Rysunek 6: Wyniki Insertion sort na danych posortowanych malejąco dla każdej z prób na odcinkach pomiarowych[16]

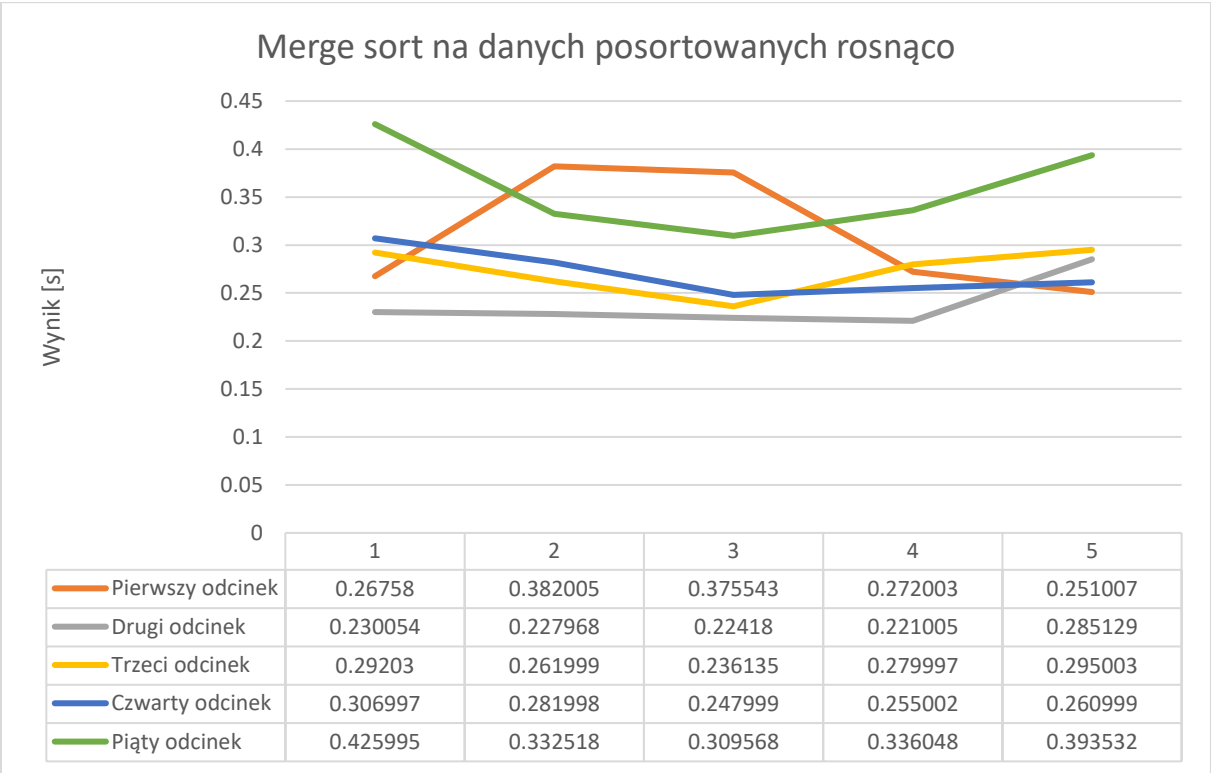
Na podstawie graficznych wyników sortowania metodą Insertion sort można zauważyć następującą prawidłowość, najszybsze rezultaty uzyskano na danych posortowanych rosnąco, więc empiria potwierdza opis metody. Najwolniej zaś przebiegało sortowanie zbioru posortowanego malejąco, zaś pomiędzy mini był zbiór losowy, który poniekąd może być wewnątrz losowy, ale częściowo posortowany. Warto zauważyć, iż pomiędzy zbiorem posortowanym rosnąco i malejąco jest różnica skali tzn. osiągane wyniki to rzędu 0,015 sekundy vs. 25 sekund. Uzyskane wyniki generalnie są stabilne wobec średniej, aczkolwiek przy zastosowaniu metody Insertion sort na danych posortowanych rosnąco na pierwszym odcinku zanotowano trochę odstające wyniki, co może skazywać na pobocznie uruchomiony jakiś dodatkowy proces na komputerze testowym.

[16] Opracowanie własne



	1	2	3	4	5
Pierwszy odcinek	0.244002	0.278005	0.246597	0.288558	0.242509
Drugi odcinek	0.281567	0.384004	0.249571	0.340001	0.262007
Trzeci odcinek	0.252	0.283	0.24955	0.270003	0.251008
Czwarty odcinek	0.256032	0.264969	0.247967	0.251008	0.253964
Piąty odcinek	0.239999	0.256966	0.244038	0.246481	0.239035

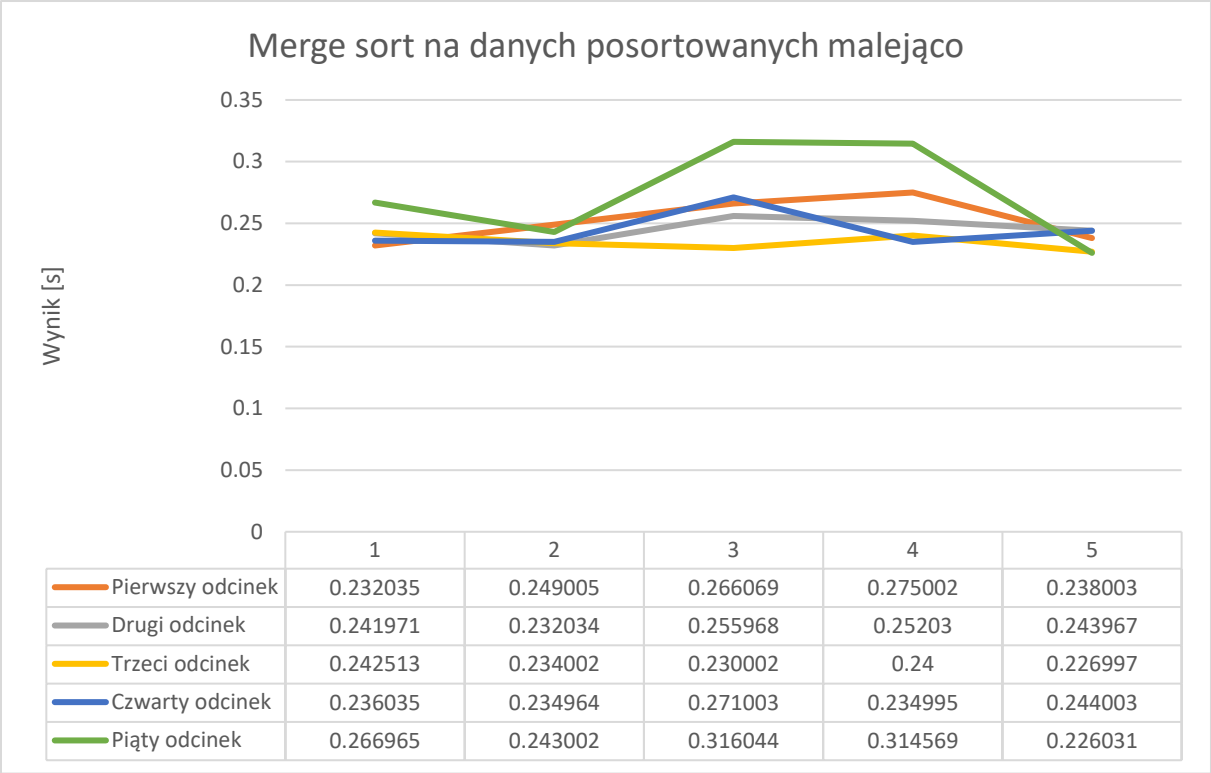
Rysunek 7: Wyniki Merge sort na danych losowych dla każdej z prób na odcinkach pomiarowych[17]



	1	2	3	4	5
Pierwszy odcinek	0.26758	0.382005	0.375543	0.272003	0.251007
Drugi odcinek	0.230054	0.227968	0.22418	0.221005	0.285129
Trzeci odcinek	0.29203	0.261999	0.236135	0.279997	0.295003
Czwarty odcinek	0.306997	0.281998	0.247999	0.255002	0.260999
Piąty odcinek	0.425995	0.332518	0.309568	0.336048	0.393532

Rysunek 8: Wyniki Merge sort na danych posortowanych rosnąco dla każdej z prób na odcinkach pomiarowych[18]

[17] Opracowanie własne
[18] Opracowanie własne



	1	2	3	4	5
Pierwszy odcinek	0.232035	0.249005	0.266069	0.275002	0.238003
Drugi odcinek	0.241971	0.232034	0.255968	0.25203	0.243967
Trzeci odcinek	0.242513	0.234002	0.230002	0.24	0.226997
Czwarty odcinek	0.236035	0.234964	0.271003	0.234995	0.244003
Piąty odcinek	0.266965	0.243002	0.316044	0.314569	0.226031

Rysunek 9: Wyniki Merge sort na danych posortowanych malejąco dla każdej z prób na odcinkach pomiarowych[19]

Uogólniając uzyskane wyniki sortowania metodą Merge sort oraz poprzednimi metodami można zaobserwować, iż jest to ogólnie najszybsza metoda, która w osiąga najbardziej spójne wyniki sortowania zbiorów wejściowych. Wyniki wyszły spójne wokół jednej średniej z drobnymi odchyleniami. Bez znajomości charakterystyki zbioru wejściowego, wydaje się, iż jest to najlepsza metoda do zastosowania.

Wyniki finalne i wnioski

Na podstawie pełnych wyników wyznaczono wartości średnie oraz odchylenia standardowe dla każdego z zastosowanych algorytmów. Wyniki przedstawiają się następująco:

Algorytm	Średnia [s]	Odchylenie standardowe [s]	Odchylenie standardowe [%]
Selection sort	15.74354448	1.940514987	12
Insertion sort	13.62843124	0.926541617	7
Merge sort	0.26491364	0.032313896	12

Tabela 4: Średnia oraz odchylenie standardowe w zależności od zastosowanej metody sortowania na wektorze liczb losowych[20]

Na podstawie powyższych wyników otrzymanych na wejściowym wektorze danych losowych można stwierdzić, iż najszybszą metodą z trzech analizowanych jest metoda sortowania Merge sort, która średnio trwa ok 0,26 sekundy, podczas gdy pozostałe dwie metody na tych samych danych trwają porównywalnie co do skali – Insertion sort 13,6 sekundy zaś Selection sort (najwolniejsza) 15,7 sekundy. Te wyniki potwierdzają opis teoretyczny z poprzednich części raportu, co do złożoności czasowej każdej z metod. Kierując się wartością bezwzględną odchylenia standardowego najbardziej dokładna wydaje się metoda Merge sort, jednakże procentowo zróżnicowanie jest najmniejsze dla wyników uzyskanych metodą Insertion sort. Zatem jeżeli badacza interesuje przeciętny czas sortowania

[19] Opracowanie własne
[20] Opracowanie własne

– sugerowane jest wykorzystanie metody Merge sort, podczas gdy, jeżeli dla badacza największa wartość dodana jest w zmniejszeniu zróżnicowania wyników, wtedy sugerowane jest wykorzystanie metody Insertion sort.

Algorytm	Średnia [s]	Odchylenie standardowe [s]	Odchylenie standardowe [%]
Selection sort	18.59006496	0.642819794	3
Insertion sort	27.5911706	2.659546209	10
Merge sort	0.24988836	0.023427071	9

Tabela 5: Średnia oraz odchylenie standardowe w zależności od zastosowanej metody sortowania na wektorze liczb posortowanych malejąco[21]

Na podstawie powyższych wyników widać, iż generalnie uzyskane wyniki są inne niż dla wektorów losowych. Dla wektorów posortowanych malejąco tak jak poprzednio najszybszą metodą jest metoda Merge sort, zaś najwolniejszą Insertion sort, te wyniki ponownie potwierdzają opis teoretyczny. Co do wartości bezwzględnych najmniejszym odchyleniem standardowym charakteryzuje się metoda Merge sort, zaś dla metody Selection sort odchylenie standardowe jest ok 30 razy większe, zaś dla Insertion sort ponad 100 krotnie więcej. W wartościach procentowych widać, iż najmniej zmienne wyniki uzyskano dla metody Selection sort, przy czym zarówno Insertion sort jak i Merge sort mają procentowe odchylenie ok 3 krotnie większe. Zatem na podstawie uzyskanej empirii dla badaczy, którym zależy na najkrótszym czasie sugerowane jest wykorzystanie metody Merge sort do sortowania wektora zawierającego dane posortowane malejąco, podczas gdy zależy na najmniejsze zmienności sugerowane jest wykorzystanie metody Selection sort.

[21] Opracowanie własne

Algorytm	Średnia [s]	Odchylenie standardowe [s]	Odchylenie standardowe [%]
Selection sort	14.96271984	0.654321561	4
Insertion sort	0.01486616	0.004163255	28
Merge sort	0.29009176	0.055212091	19

Tabela 6: Średnia oraz odchylenie standardowe w zależności od zastosowanej metody sortowania na wektorze liczb posortowanych rosnąco[22]

Powyższe wyniki prezentują średni czas oraz odchylenie standardowe dla każdej z trzech metod sortowania na wektorze zawierającym dane posortowane rosnąco. Zgodnie z opisem teoretycznym powinno się oczekiwać, iż najszybszą metodą będzie Insertion sort, co też znalazło potwierdzenie w wynikach empirycznych. Drugą co do średniego czasu przetwarzania jest metoda Merge sort, zaś trzecią (najwolniejszą) Selection sort. Jeżeli chodzi o zmienność, co do wartości – hierarchia odchyleń standardowych jest analogiczna jak średniego czasu sortowania, natomiast relatywnie najmniejsza zmienność jest dla metody najwolniejszej Selection sort, następnie Merge sort i Insertion sort.

Celem porównania efektywności każdej z trzech metod policzono również wyniki zagregowane, które są przedstawione w poniższej tabeli:

Algorytm	Średnia [s]	Odchylenie standardowe [s]
Selection sort	16.43210976	1.991394146
Insertion sort	13.74482267	11.37509394
Merge sort	0.26829792	0.042687731

Tabela 7: Średnia oraz odchylenie standardowe w zależności na podstawie wszystkich pomiarów[23]

Powyższe wyniki uśrednione na trzech rodzajach danych wejściowych (wektor losowy, wektor danych posortowanych rosnąco oraz wektor danych posortowanych malejąco) pokazuje, iż na testowanych danych najszybszą metodą oraz zarówno cechującą się najmniejszą zmiennością (wartość odchylenia standardowego) jest Merge sort, następnie drugą, co do szybkości jest metoda Insertion sort (około 50 razy dłużej), jednakże owa metoda cechuje się dość dużą zmiennością, co dobrze pokazano na wcześniejszych wynikach, gdzie Insertion sort jest bardzo szybką metodą na danych posortowanych losowo. Natomiast trzecią, co do przeciętnego czasu sortowania jest metoda Selection sort, jest ona trochę bardziej czasochłonna niż Insertion sort, jednakże cechuje się mniejszą zmiennością.

Na podstawie powyższych danych, widać, iż w kontekście niepewności lub niewiedzy odnośnie charakterystyki danych wejściowych najszybszą metodą sortowania okazała się metoda Merge sort, jednakże w przypadku, gdy dane wejściowe są posortowanie rosnąco, na podstawie badania sugerowane jest zastosować metodę Insertion sort, która empirycznie potwierdziła teorię, iż jest to bardzo szybka metoda na takich danych. Na podstawie eksperymentu, sugeruje się wpierw analizę danych zaś następnie dobranie odpowiednią metodę.

[22] Opracowanie własne
[23] Opracowanie własne

Przeprowadzenie eksperymentu II

Założenia

Do przeprowadzenia eksperymentu przyjęto następujące założenia:

- Zbiór danych początkowy składa się z losowo wygenerowanych liczb zmiennoprzecinkowych z przedziału 0-1, które zostały posortowane.
- Do powyższego zbioru będą dodawane dane losowe nieposortowane (liczby zmiennoprzecinkowe z przedziału 0-1).
- Porównane zostaną dwa algorytmy sortowania:
  - Insertion sort
  - Merge sort

Eksperyment – opis

Środowisko

Element środowiska	Parametry
Procesor	Inter® Core™ i7-7600 CPU @ 2.80 GHz 2.90 GHZ
RAM	8 GB
System	Windows 10 Enterprise version 1809
Dysk	SSD Samsung PM961 256 GB M.2 2280 PCI-E x4 Gen3 NVMe (MZVLW256HEHP-00000)
Język programowania	Python 3.8.2 32-bit
IDE	Microsoft Visual Studio Community 2019 Version 16.8.2

Tabela 8: Specyfikacja środowiska, na którym przeprowadzano eksperyment[24]

Przebieg eksperymentu

Celem realizacji eksperymentu jest sprawdzenie jak o ile obserwacji (losowych) należy rozszerzyć wektor danych posortowanych rosnąco tak, aby metoda Merge sort była metodą efektywniejszą niż metoda Insertion sort. Na podstawie pomiarów czasowych uzyskanych w toku prac nad eksperymentem I przyjęto, iż jako wektor bazowy wykorzysta się wektor o długości 100 000 liczb uprzednio posortowanych.

Eksperyment będzie odbywał się iteracyjnie poprzez dodanie na końcu wektora wielokrotności 10 liczb nieposortowanych oraz pomiaru czasów sortowania takiego wektora. W ten sposób przeprowadzono 11 iteracji jak poniżej:

Numer iteracji	Liczba danych losowych na końcu wektora	Łączna długość wektora
1	0	100000
2	10	100010
3	20	100020
4	30	100030
5	40	100040
6	50	100050
7	60	100060
8	70	100070
9	80	100080
10	90	100090
11	100	100100

Tabela 9: Specyfika iteracji[25]

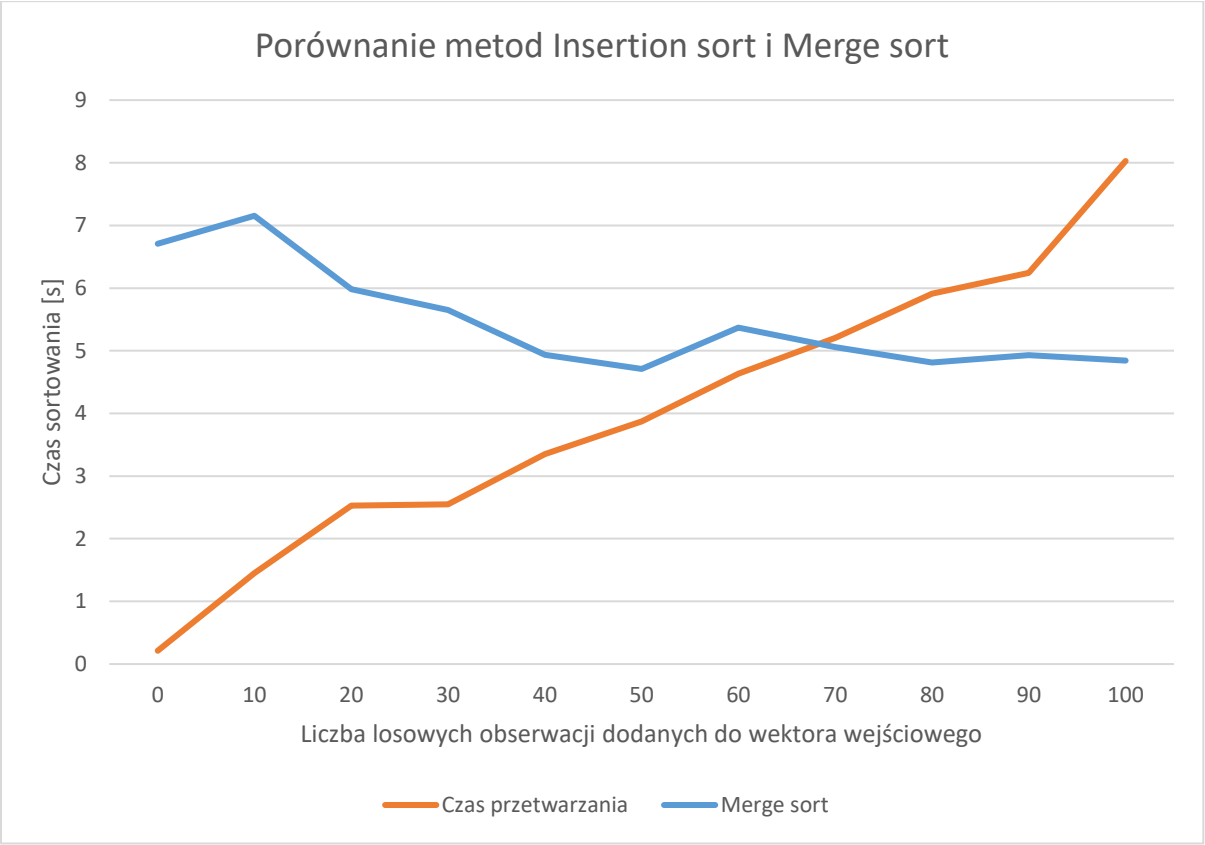
[24] Opracowanie własne
[25] Opracowanie własne

Wyniki eksperymentu

Na podstawie przeprowadzonego eksperymentu okazało się, iż:

- Dodanie nawet niewielu danych losowych do wektora bazowego skutkowało znacznym wydłużeniem metody Insertion sort, tzn. dodając do wektora bazowego liczby losowe o długości równej 0,01%[26] wektora wejściowego nastąpiło wydłużenie czasu przetwarzania 5,8 raza
- Po wydłużeni wektora bazowego o 0,08%[27] pomiary wykazały, iż sortowanie metodą Merge sort odbywa się bardziej efektywnie niż metodą Insertion sort.

Graficzna prezentacja porównania obu metod przedstawia się następująco:



Rysunek 10: Porównanie czasu sortowania obu metod w zależności od liczby dodanych danych losowych do wektora bazowego posortowanego rosnąco[28]

Na podstawie powyższych wyników wykazano, iż metoda Insertion sort jest bardzo wrażliwa w aspekcie czasu sortowania na postać wektora wejściowego, jeżeli jest on losowy w drobnym ułamku, taka sytuacja ma znaczący wpływ na efektywność metody Insertion sort i powoduje wydłużenie czasu sortowania. Dodatkowym wnioskiem wynikającym z eksperymentu, jest to, iż gdy dane są choć w drobnej części losowe (empirycznie 0,08%[29] długości wektora posortowanego) wtedy też szybszą metodą sortowania okazuje się metoda Merge sort.

Zatem podstawowym wnioskiem z badania empirycznego jest, iż w przypadku braku pewności odnośnie postaci i charakterystyki wektora wejściowego ze względu na czas sortowania i efektywność sugerowane jest korzystanie z metody Merge sort.

[26] Opracowanie własne na podstawie pełnych wyników
[27] Opracowanie własne na podstawie pełnych wyników
[28] Opracowanie własne
[29] Opracowanie własne na podstawie pełnych wyników

Załączniki

Pełne wyniki eksperymentu I

Metoda sortowania	Wektor wejściowy	Odcinek pomiarowy	Wynik [s]
Selection Sort	losowy	1	14.123602
Selection Sort	losowy	1	15.587969
Selection Sort	losowy	1	17.307768
Selection Sort	losowy	1	16.243497
Selection Sort	losowy	1	15.710739
Insertion Sort	losowy	1	14.122369
Insertion Sort	losowy	1	13.466210
Insertion Sort	losowy	1	13.010858
Insertion Sort	losowy	1	14.918575
Insertion Sort	losowy	1	12.890246
Merge Sort	losowy	1	0.244002
Merge Sort	losowy	1	0.278005
Merge Sort	losowy	1	0.246597
Merge Sort	losowy	1	0.288558
Merge Sort	losowy	1	0.242509
Selection Sort	posortowany rosnąco	1	15.363341
Selection Sort	posortowany rosnąco	1	14.491281
Selection Sort	posortowany rosnąco	1	14.009255
Selection Sort	posortowany rosnąco	1	14.772845
Selection Sort	posortowany rosnąco	1	15.014056
Insertion Sort	posortowany rosnąco	1	0.013998
Insertion Sort	posortowany rosnąco	1	0.025419
Insertion Sort	posortowany rosnąco	1	0.020001
Insertion Sort	posortowany rosnąco	1	0.026999
Insertion Sort	posortowany rosnąco	1	0.020999
Merge Sort	posortowany rosnąco	1	0.267580
Merge Sort	posortowany rosnąco	1	0.382005
Merge Sort	posortowany rosnąco	1	0.375543
Merge Sort	posortowany rosnąco	1	0.272003
Merge Sort	posortowany rosnąco	1	0.251007
Selection Sort	posortowany malejąco	1	18.439076
Selection Sort	posortowany malejąco	1	18.139207
Selection Sort	posortowany malejąco	1	18.112045
Selection Sort	posortowany malejąco	1	18.206097
Selection Sort	posortowany malejąco	1	17.560518
Insertion Sort	posortowany malejąco	1	25.318010
Insertion Sort	posortowany malejąco	1	26.474244
Insertion Sort	posortowany malejąco	1	25.236140
Insertion Sort	posortowany malejąco	1	27.599443
Insertion Sort	posortowany malejąco	1	27.117572

Merge Sort	posortowany malejąco	1	0.232035
Merge Sort	posortowany malejąco	1	0.249005
Merge Sort	posortowany malejąco	1	0.266069
Merge Sort	posortowany malejąco	1	0.275002
Merge Sort	posortowany malejąco	1	0.238003
Selection Sort	losowy	2	16.065088
Selection Sort	losowy	2	14.744241
Selection Sort	losowy	2	15.477605
Selection Sort	losowy	2	14.825859
Selection Sort	losowy	2	14.237963
Insertion Sort	losowy	2	12.962677
Insertion Sort	losowy	2	12.435288
Insertion Sort	losowy	2	14.208402
Insertion Sort	losowy	2	15.578129
Insertion Sort	losowy	2	13.728486
Merge Sort	losowy	2	0.281567
Merge Sort	losowy	2	0.384004
Merge Sort	losowy	2	0.249571
Merge Sort	losowy	2	0.340001
Merge Sort	losowy	2	0.262007
Selection Sort	posortowany rosnąco	2	15.036231
Selection Sort	posortowany rosnąco	2	15.623363
Selection Sort	posortowany rosnąco	2	14.989795
Selection Sort	posortowany rosnąco	2	14.994445
Selection Sort	posortowany rosnąco	2	14.773172
Insertion Sort	posortowany rosnąco	2	0.012000
Insertion Sort	posortowany rosnąco	2	0.014000
Insertion Sort	posortowany rosnąco	2	0.013039
Insertion Sort	posortowany rosnąco	2	0.011967
Insertion Sort	posortowany rosnąco	2	0.015998
Merge Sort	posortowany rosnąco	2	0.230054
Merge Sort	posortowany rosnąco	2	0.227968
Merge Sort	posortowany rosnąco	2	0.224180
Merge Sort	posortowany rosnąco	2	0.221005
Merge Sort	posortowany rosnąco	2	0.285129
Selection Sort	posortowany malejąco	2	18.738145
Selection Sort	posortowany malejąco	2	18.998890
Selection Sort	posortowany malejąco	2	19.331222
Selection Sort	posortowany malejąco	2	19.503474
Selection Sort	posortowany malejąco	2	19.437879
Insertion Sort	posortowany malejąco	2	27.585631
Insertion Sort	posortowany malejąco	2	26.757496
Insertion Sort	posortowany malejąco	2	27.768525
Insertion Sort	posortowany malejąco	2	27.236309
Insertion Sort	posortowany malejąco	2	27.625418

Merge Sort	posortowany malejąco	2	0.241971
Merge Sort	posortowany malejąco	2	0.232034
Merge Sort	posortowany malejąco	2	0.255968
Merge Sort	posortowany malejąco	2	0.252030
Merge Sort	posortowany malejąco	2	0.243967
Selection Sort	losowy	3	15.255867
Selection Sort	losowy	3	16.369053
Selection Sort	losowy	3	23.594894
Selection Sort	losowy	3	16.540653
Selection Sort	losowy	3	19.031138
Insertion Sort	losowy	3	14.577385
Insertion Sort	losowy	3	15.783875
Insertion Sort	losowy	3	14.767660
Insertion Sort	losowy	3	12.541896
Insertion Sort	losowy	3	13.213298
Merge Sort	losowy	3	0.252000
Merge Sort	losowy	3	0.283000
Merge Sort	losowy	3	0.249550
Merge Sort	losowy	3	0.270003
Merge Sort	losowy	3	0.251008
Selection Sort	posortowany rosnąco	3	16.261928
Selection Sort	posortowany rosnąco	3	14.761496
Selection Sort	posortowany rosnąco	3	14.088903
Selection Sort	posortowany rosnąco	3	14.921989
Selection Sort	posortowany rosnąco	3	14.281313
Insertion Sort	posortowany rosnąco	3	0.011001
Insertion Sort	posortowany rosnąco	3	0.012000
Insertion Sort	posortowany rosnąco	3	0.012001
Insertion Sort	posortowany rosnąco	3	0.011965
Insertion Sort	posortowany rosnąco	3	0.012999
Merge Sort	posortowany rosnąco	3	00.292030
Merge Sort	posortowany rosnąco	3	00.261999
Merge Sort	posortowany rosnąco	3	00.236135
Merge Sort	posortowany rosnąco	3	00.279997
Merge Sort	posortowany rosnąco	3	00.295003
Selection Sort	posortowany malejąco	3	18.688270
Selection Sort	posortowany malejąco	3	18.819427
Selection Sort	posortowany malejąco	3	18.470397
Selection Sort	posortowany malejąco	3	18.269580
Selection Sort	posortowany malejąco	3	17.400322
Insertion Sort	posortowany malejąco	3	25.251942
Insertion Sort	posortowany malejąco	3	26.166021
Insertion Sort	posortowany malejąco	3	24.872305
Insertion Sort	posortowany malejąco	3	25.479310
Insertion Sort	posortowany malejąco	3	25.624953

Merge Sort	posortowany malejąco	3	0.242513
Merge Sort	posortowany malejąco	3	0.234002
Merge Sort	posortowany malejąco	3	0.230002
Merge Sort	posortowany malejąco	3	0.240000
Merge Sort	posortowany malejąco	3	0.226997
Selection Sort	losowy	4	14.210971
Selection Sort	losowy	4	14.652501
Selection Sort	losowy	4	14.098933
Selection Sort	losowy	4	14.951873
Selection Sort	losowy	4	15.771264
Insertion Sort	losowy	4	13.579989
Insertion Sort	losowy	4	13.169060
Insertion Sort	losowy	4	13.889991
Insertion Sort	losowy	4	12.823024
Insertion Sort	losowy	4	13.300231
Merge Sort	losowy	4	0.256032
Merge Sort	losowy	4	0.264969
Merge Sort	losowy	4	0.247967
Merge Sort	losowy	4	0.251008
Merge Sort	losowy	4	0.253964
Selection Sort	posortowany rosnąco	4	15.252064
Selection Sort	posortowany rosnąco	4	14.218985
Selection Sort	posortowany rosnąco	4	15.634654
Selection Sort	posortowany rosnąco	4	14.470994
Selection Sort	posortowany rosnąco	4	16.312050
Insertion Sort	posortowany rosnąco	4	0.013995
Insertion Sort	posortowany rosnąco	4	0.012031
Insertion Sort	posortowany rosnąco	4	0.011962
Insertion Sort	posortowany rosnąco	4	0.013003
Insertion Sort	posortowany rosnąco	4	0.014001
Merge Sort	posortowany rosnąco	4	0.306997
Merge Sort	posortowany rosnąco	4	0.281998
Merge Sort	posortowany rosnąco	4	0.247999
Merge Sort	posortowany rosnąco	4	0.255002
Merge Sort	posortowany rosnąco	4	0.260999
Selection Sort	posortowany malejąco	4	18.166628
Selection Sort	posortowany malejąco	4	18.797067
Selection Sort	posortowany malejąco	4	19.791148
Selection Sort	posortowany malejąco	4	19.244034
Selection Sort	posortowany malejąco	4	19.245033
Insertion Sort	posortowany malejąco	4	25.717183
Insertion Sort	posortowany malejąco	4	26.056865
Insertion Sort	posortowany malejąco	4	26.069569
Insertion Sort	posortowany malejąco	4	32.666281
Insertion Sort	posortowany malejąco	4	25.796408

Merge Sort	posortowany malejąco	4	0.236035
Merge Sort	posortowany malejąco	4	0.234964
Merge Sort	posortowany malejąco	4	0.271003
Merge Sort	posortowany malejąco	4	0.234995
Merge Sort	posortowany malejąco	4	0.244003
Selection Sort	losowy	5	14.829015
Selection Sort	losowy	5	15.468017
Selection Sort	losowy	5	14.347019
Selection Sort	losowy	5	15.353044
Selection Sort	losowy	5	14.790039
Insertion Sort	losowy	5	12.912029
Insertion Sort	losowy	5	13.391788
Insertion Sort	losowy	5	12.543915
Insertion Sort	losowy	5	14.325279
Insertion Sort	losowy	5	12.570121
Merge Sort	losowy	5	0.239999
Merge Sort	losowy	5	0.256966
Merge Sort	losowy	5	0.244038
Merge Sort	losowy	5	0.246481
Merge Sort	losowy	5	0.239035
Selection Sort	posortowany rosnąco	5	14.440980
Selection Sort	posortowany rosnąco	5	15.627907
Selection Sort	posortowany rosnąco	5	16.081218
Selection Sort	posortowany rosnąco	5	14.321544
Selection Sort	posortowany rosnąco	5	14.324187
Insertion Sort	posortowany rosnąco	5	0.017001
Insertion Sort	posortowany rosnąco	5	0.013278
Insertion Sort	posortowany rosnąco	5	0.012000
Insertion Sort	posortowany rosnąco	5	0.016996
Insertion Sort	posortowany rosnąco	5	0.013001
Merge Sort	posortowany rosnąco	5	0.425995
Merge Sort	posortowany rosnąco	5	0.332518
Merge Sort	posortowany rosnąco	5	0.309568
Merge Sort	posortowany rosnąco	5	0.336048
Merge Sort	posortowany rosnąco	5	0.393532
Selection Sort	posortowany malejąco	5	18.925271
Selection Sort	posortowany malejąco	5	19.004820
Selection Sort	posortowany malejąco	5	17.507545
Selection Sort	posortowany malejąco	5	18.143203
Selection Sort	posortowany malejąco	5	17.812326
Insertion Sort	posortowany malejąco	5	34.862508
Insertion Sort	posortowany malejąco	5	28.343228
Insertion Sort	posortowany malejąco	5	30.137019
Insertion Sort	posortowany malejąco	5	33.502852
Insertion Sort	posortowany malejąco	5	30.514033

Merge Sort	posortowany malejąco	5	0.266965
Merge Sort	posortowany malejąco	5	0.243002
Merge Sort	posortowany malejąco	5	0.316044
Merge Sort	posortowany malejąco	5	0.314569
Merge Sort	posortowany malejąco	5	0.226031

Pełne wyniki eksperymentu II

Iteracja	Długość wektora	Ilość liczb losowych	Insertion sort [s]	Merge sort [s]
1	100000	0	0.211999	6.705953
2	100010	10	1.452524	7.154041
3	100020	20	2.53015	5.983948
4	100030	30	2.55055	5.649789
5	100040	40	3.348721	4.937344
6	100050	50	3.874134	4.709515
7	100060	60	4.634713	5.36809
8	100070	70	5.205103	5.059897
9	100080	80	5.910178	4.811486
10	100090	90	6.241485	4.92967
11	100100	100	8.028307	4.844697

Kod programu

Poniżej załączono kod programu, którego rozwój jest widoczny poprzez platformę github.com: https://github.com/oksiedz/Python/tree/master/Projects/Project2.

```
import random
import datetime

#Arrays to be used - randomArray - array with random numbers, sortedArrayAsc - array with sorted ascending numbers, sortedArrayDesc - array with sorted ascending numbers
randomArray = []
sortedArrayAsc = []
sortedArrayDesc = []

#variables explanation:
#inputArray - parameter for input array to be sorted
#ifSave = parameter set 1 if as an output there should be saved the sorted array
#inputType = parameter defining type of input array - if it's random (R), sorted ASC (A) or sorted DESC (D)
resultsList = []

#noOfGeneratedNumber - how many items contain array of random numbers
noOfGeneratedNumber = 30000

print("Start of random array generation")
for i in range(0, noOfGeneratedNumber):
        randomArray.append(float(random.random()))
print("End of random array generation")

#print("random array:")
```

```
#for i in range(len(randomArray)):
#    print(randomArray[i])

#Selection sort
def selectionSort(inputArray, inputType = "Z",measurePoint = 0):
        #Array to be sorted
        A = []
        for i in range(0, len(inputArray)):
                A.append(inputArray[i])
        #print("Start - Selection sort")
        startTime = datetime.datetime.now()
        for i in range(len(A)):
                #Find the min value in the remaining not sorted part of array
                min_idx = i
                for j in range(i+1, len(A)):
                        if A[min_idx] > A[j]:
                                min_idx = j
                #Swap the minimum with the first array element
                A[i], A[min_idx] = A[min_idx], A[i]
        endTime = datetime.datetime.now()
#       print("End - Selection sort")
        resultsList.append("S;"+str(inputType)+";"+str(measurePoint)+";"+str(endTime-startTime))

#Insertion sort
def insertionSort(inputArray, inputType = "Z", measurePoint = 0):
#       print("Start - Insertion sort")
        A = []
        for i in range(0, len(inputArray)):
                A.append(inputArray[i])
        startTime = datetime.datetime.now()
        for i in range(1, len(A)):
                key = A[i]
                #Moving elements of A[0..i-1], which are greater that key, to one position ahead of
their current position
                j = i-1
                while j >=0 and key < A[j] :
                                A[j+1] = A[j]
                                j -= 1
                A[j+1] = key
        endTime = datetime.datetime.now()
#       print("End - Insertion sort")
        resultsList.append("I;"+str(inputType)+";"+str(measurePoint)+";"+str(endTime-startTime))

def mergeSortEngine(alist):
        if len(alist)>1:
                mid = len(alist)//2
                lefthalf = alist[:mid]
                righthalf = alist[mid:]

                mergeSortEngine(lefthalf)
                mergeSortEngine(righthalf)

                i=0
                j=0
                k=0
```

```
                while i < len(lefthalf) and j < len(righthalf):
                        if lefthalf[i] <= righthalf[j]:
                                alist[k]=lefthalf[i]
                                i=i+1
                        else:
                                alist[k]=righthalf[j]
                                j=j+1
                        k=k+1

                while i < len(lefthalf):
                        alist[k]=lefthalf[i]
                        i=i+1
                        k=k+1

                while j < len(righthalf):
                        alist[k]=righthalf[j]
                        j=j+1
                        k=k+1

def mergeSort(inputArray, inputType = "Z", ifSave = 0, saveResults = 0, measurePoint = 0):
        A = []
        for i in range(0, len(inputArray)):
                A.append(inputArray[i])
        #print("Start - Merge sort")
        startTime = datetime.datetime.now()
        mergeSortEngine(A)
        endTime = datetime.datetime.now()
        #print("End - Merge sort")
        if (ifSave == 1):
                for i in range(len(A)):
                        sortedArrayAsc.append(A[i])
        if (saveResults == 1):
                resultsList.append("M;"+str(inputType)+";"+str(measurePoint)+";"+str(endTime-
startTime))


##Section used to determine times and produce sorted array ASC and DESC
#print("Start - random sorting")
#selectionSort(randomArray, "R")
#insertionSort(randomArray, "R")
print("Start - generation of sorted array ASC")
mergeSort(randomArray, "R", 1, 0, 0)
#print("End - random sorting")
print("End - generation of sorted array ASC")

###Section reversing sorted array
print("Start - Sorting Desc")
reverseStart = datetime.datetime.now()
#for i in range(0, len(sortedArrayAsc)):
#       sortedArrayDesc.append(list(reversed(sortedArrayAsc))[i])
sortedArrayDesc = list(reversed(sortedArrayAsc))
reverseEnd = datetime.datetime.now()
print("End - Sorting Desc, lasted: "+str(reverseEnd - reverseStart))

#print("sortowany DESC")
```

```
#print(sortedArrayDesc)
#sortedArrayDesc = list(reversed(sortedArrayAsc))

#print("Start - sortedASC sorting")
#selectionSort(sortedArrayAsc, "A")
#insertionSort(sortedArrayAsc, "A")
#mergeSort(sortedArrayAsc, "A", 0)
#print("End - sortedASC sorting")
#print("Start - sortedDESC sorting")
#selectionSort(sortedArrayDesc, "D")
#insertionSort(sortedArrayDesc, "D")
#mergeSort(sortedArrayDesc, "D", 0)
#print("End - sortedDESC sorting")

#print("random array:")
#for i in range(0, len(randomArray)):
#	print(randomArray[i])

#print("sorted ASC array:")
#for i in range(0, len(sortedArrayAsc)):
#	print(sortedArrayAsc[i])

#print("sorted DESC array:")
#for i in range(0, len(sortedArrayDesc)):
#	print(sortedArrayDesc[i])

##Section of calculations of times:
loopStart = 0
loopEnd = 5
startNumber = 0
endNumber = 6000
print("Start: First loop")
print("Start - random array")
for i in range(loopStart,loopEnd):
	selectionSort(randomArray[startNumber:endNumber], "R", 1)
for i in range(loopStart,loopEnd):
	insertionSort(randomArray[startNumber:endNumber], "R", 1)
for i in range(loopStart,loopEnd):
	mergeSort(randomArray[startNumber:endNumber], "R", 0, 1, 1)
print("End - random array")
print("Start - ASC array")
for i in range(loopStart,loopEnd):
	selectionSort(sortedArrayAsc[startNumber:endNumber], "A", 1)
for i in range(loopStart,loopEnd):
	insertionSort(sortedArrayAsc[startNumber:endNumber], "A", 1)
for i in range(loopStart,loopEnd):
	mergeSort(sortedArrayAsc[startNumber:endNumber], "A", 0, 1, 1)
print("End - ASC array")
print("Start - DESC array")
for i in range(loopStart,loopEnd):
	selectionSort(sortedArrayDesc[startNumber:endNumber], "D", 1)
for i in range(loopStart,loopEnd):
	insertionSort(sortedArrayDesc[startNumber:endNumber], "D", 1)
for i in range(loopStart,loopEnd):
	mergeSort(sortedArrayDesc[startNumber:endNumber], "D", 0, 1, 1)
```

```
print("End - DESC array")
print("End: First loop")
startNumber = 6001
endNumber = 12000
print("Start: Second loop")
for i in range(loopStart,loopEnd):
        selectionSort(randomArray[startNumber:endNumber], "R", 2)
for i in range(loopStart,loopEnd):
        insertionSort(randomArray[startNumber:endNumber], "R", 2)
for i in range(loopStart,loopEnd):
        mergeSort(randomArray[startNumber:endNumber], "R", 0, 1, 2)
print("End - random array")
print("Start - ASC array")
for i in range(loopStart,loopEnd):
        selectionSort(sortedArrayAsc[startNumber:endNumber], "A", 2)
for i in range(loopStart,loopEnd):
        insertionSort(sortedArrayAsc[startNumber:endNumber], "A", 2)
for i in range(loopStart,loopEnd):
        mergeSort(sortedArrayAsc[startNumber:endNumber], "A", 0, 1, 2)
print("End - ASC array")
print("Start - DESC array")
for i in range(loopStart,loopEnd):
        selectionSort(sortedArrayDesc[startNumber:endNumber], "D", 2)
for i in range(loopStart,loopEnd):
        insertionSort(sortedArrayDesc[startNumber:endNumber], "D", 2)
for i in range(loopStart,loopEnd):
        mergeSort(sortedArrayDesc[startNumber:endNumber], "D", 0, 1, 2)
print("End - DESC array")
print("End: Second loop")
startNumber = 12001
endNumber = 18000
print("Start: Third loop")
print("Start - random array")
for i in range(loopStart,loopEnd):
        selectionSort(randomArray[startNumber:endNumber], "R", 3)
for i in range(loopStart,loopEnd):
        insertionSort(randomArray[startNumber:endNumber], "R", 3)
for i in range(loopStart,loopEnd):
        mergeSort(randomArray[startNumber:endNumber], "R", 0, 1, 3)
print("End - random array")
print("Start - ASC array")
for i in range(loopStart,loopEnd):
        selectionSort(sortedArrayAsc[startNumber:endNumber], "A", 3)
for i in range(loopStart,loopEnd):
        insertionSort(sortedArrayAsc[startNumber:endNumber], "A", 3)
for i in range(loopStart,loopEnd):
        mergeSort(sortedArrayAsc[startNumber:endNumber], "A", 0, 1, 3)
print("End - ASC array")
print("Start - DESC array")
for i in range(loopStart,loopEnd):
        selectionSort(sortedArrayDesc[startNumber:endNumber], "D", 3)
for i in range(loopStart,loopEnd):
        insertionSort(sortedArrayDesc[startNumber:endNumber], "D", 3)
for i in range(loopStart,loopEnd):
        mergeSort(sortedArrayDesc[startNumber:endNumber], "D", 0, 1, 3)
```

```
print("End - DESC array")
print("End: Third loop")
startNumber = 18001
endNumber = 24000
print("Start: Fourth loop")
print("Start - random array")
for i in range(loopStart,loopEnd):
        selectionSort(randomArray[startNumber:endNumber], "R", 4)
for i in range(loopStart,loopEnd):
        insertionSort(randomArray[startNumber:endNumber], "R", 4)
for i in range(loopStart,loopEnd):
        mergeSort(randomArray[startNumber:endNumber], "R", 0, 1, 4)
print("End - random array")
print("Start - ASC array")
for i in range(loopStart,loopEnd):
        selectionSort(sortedArrayAsc[startNumber:endNumber], "A", 4)
for i in range(loopStart,loopEnd):
        insertionSort(sortedArrayAsc[startNumber:endNumber], "A", 4)
for i in range(loopStart,loopEnd):
        mergeSort(sortedArrayAsc[startNumber:endNumber], "A", 0, 1, 4)
print("End - ASC array")
print("Start - DESC array")
for i in range(loopStart,loopEnd):
        selectionSort(sortedArrayDesc[startNumber:endNumber], "D", 4)
for i in range(loopStart,loopEnd):
        insertionSort(sortedArrayDesc[startNumber:endNumber], "D", 4)
for i in range(loopStart,loopEnd):
        mergeSort(sortedArrayDesc[startNumber:endNumber], "D", 0, 1, 4)
print("End - DESC array")
print("End: Fourth loop")
startNumber = 24001
endNumber = 30000
print("Start: Fifth loop")
print("Start - random array")
for i in range(loopStart,loopEnd):
        selectionSort(randomArray[startNumber:endNumber], "R", 5)
for i in range(loopStart,loopEnd):
        insertionSort(randomArray[startNumber:endNumber], "R", 5)
for i in range(loopStart,loopEnd):
        mergeSort(randomArray[startNumber:endNumber], "R", 0, 1, 5)
print("End - random array")
print("Start - ASC array")
for i in range(loopStart,loopEnd):
        selectionSort(sortedArrayAsc[startNumber:endNumber], "A", 5)
for i in range(loopStart,loopEnd):
        insertionSort(sortedArrayAsc[startNumber:endNumber], "A", 5)
for i in range(loopStart,loopEnd):
        mergeSort(sortedArrayAsc[startNumber:endNumber], "A", 0, 1, 5)
print("End - ASC array")
print("Start - DESC array")
for i in range(loopStart,loopEnd):
        selectionSort(sortedArrayDesc[startNumber:endNumber], "D", 5)
for i in range(loopStart,loopEnd):
        insertionSort(sortedArrayDesc[startNumber:endNumber], "D", 5)
for i in range(loopStart,loopEnd):
```

```
        mergeSort(sortedArrayDesc[startNumber:endNumber], "D", 0, 1, 5)
print("End - DESC array")
print("End: Fifth loop")

print("time results")
for i in range(0, len(resultsList)):
        print(resultsList[i])

##to write down results of general tests
with open('testing results.txt', 'w') as f:
   for item in resultsList:
      f.write("%s\n" % item)




##Section of comparison of performance between Insertion sort and merge sort on sorted array
enriched with some random numbers

noOfGeneratedNumber = 100000
randomArray = []
sortedArrayAsc = []

print("Start of random array generation")
for i in range(0, noOfGeneratedNumber):
   randomArray.append(float(random.random()))
print("End of random array generation")

print("Start - generation of sorted array ASC")
mergeSort(randomArray, "R", 1, 0, 0)
#print("End - random sorting")
print("End - generation of sorted array ASC")


resultsList = []
sortedArray1 = sortedArrayAsc[0:100000]

print("Start - append the first array")
for i in range(0, 10000):
   sortedArray1.append(float(random.random()))
print("End - append the first array")

print("Time measures")
resultsList.append("Results of performance comparison")
sortedLength = noOfGeneratedNumber
step = 10
print("Start - first loops")
for i in (0,1,2,3,4,5,6,7,8,9,10):
        resultsList.append("array 10 000 with additional " + str(i)+" random numbers at the end of
array")
        print(sortedLength+i*step);
        insertionSort(sortedArray1[0:sortedLength+i*step]);
        mergeSort(sortedArray1[0:sortedLength+i*step],saveResults = 1);
print("End - first loops")
```

```
##results of simulations of performance of insertion sort and merge sort
with open('testing results2.txt', 'w') as f:
    for item in resultsList:
        f.write("%s\n" % item)

for i in range(0, len(resultsList)):
            print(resultsList[i])
```