WYŻSZA SZKOŁA MENEDŻERSKA INSTYTUT NAUK O ZARZĄDZANIU I JAKOŚCI INFORMATYKA
Algorytmy i złożoność

Praca zaliczeniowa

Sortowanie 2

<u>Skład zespołu/Student</u> Tomasz Jan Oksiędzki	**Prowadzący zajęcia:** Marcin Paprzycki	**Semestr** **III**
	Grupa **49DR – A1**	**Studia** **Niestacjonarne**
Data wykonania **2021-01-16**	Data oddania: 2021-01-16	

Spis treści

Wprowadzenie

Przedmiotem tego opracowania jest przeprowadzenie empirycznego doświadczenia i przedstawienie wniosków w zakresie efektywności metody sortowania Quick Sort, w zależności od danych wejściowych oraz metody wyboru elementu pivot. Drugim aspektem, który będzie poruszony w opracowaniu jest porównanie wyników metody Quick Sort z dwoma dodatkowymi algorytmami wyszukiwania Merge Sort oraz Heap Sort, celem weryfikacji i sugestii na podstawie empirycznego doświadczenia, która metoda jest najbardziej efektywna w zależności od danych wejściowych. Analiza Quick Sortu będzie przedmiotem sekcji Eksperyment I zaś porównanie trzech metod zostanie opisane w sekcji Eksperyment II.

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:
  - Quick Sort z losowym elementem zbioru, jako pivot
  - Quick Sort z pierwszym elementem zbioru, jako pivot
  - Quick Sort z ostatnim elementem zbioru, jako pivot
- 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	PyCharm 2020.3.2 (Community Edition) Build #PC-203.6682.179, built on December 30, 2020 Runtime version: 11.0.9.1+11-b1145.63 amd64 VM: OpenJDK 64-Bit Server VM by JetBrains s.r.o. Windows 10 10.0 GC: ParNew, ConcurrentMarkSweep Memory: 974M Cores: 4

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

Opis algorytmów sortowania

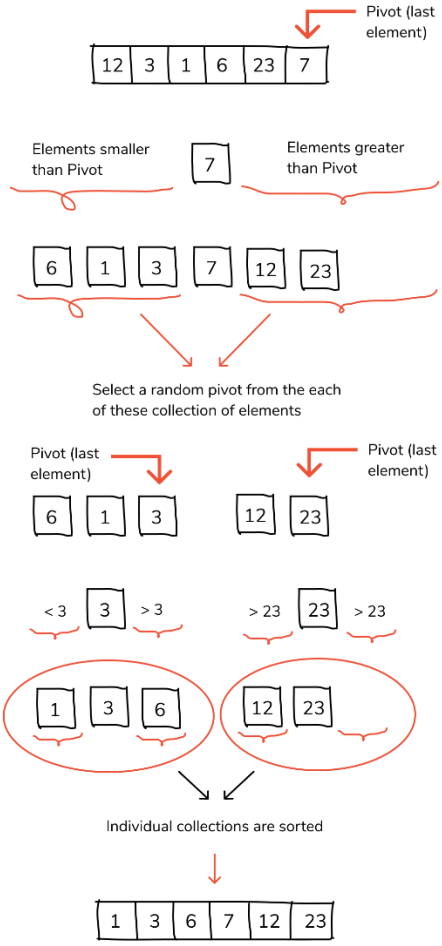
Na potrzeby eksperymentu wykorzystano trzy metody wyboru elementu pivot i algorytm Quick Sort, które zostaną przedstawione w bardziej szczegółowy sposób w dalszej części opracowania:

Quick Sort

Metoda sortowania Quick Sort jest tłumaczona na język polski, jako szybkie sortowanie, co jak najbardziej odpowiada jej charakterystyce. Jest to jedna z metod sortowania, która działa na zasadzie dziel i zwyciężaj (np. Merge Sort). Quick Sort, wynaleziony w 1962 roku[2], jest dość powszechnie stosowany ze względu na niską złożoność implementacji oraz efektywność czasową procedury sortowania. Średnia złożoność obliczeniowa algorytmu wynosi $O(n * \log(n))$[3], czyli porównywalna Merge Sort[4], jednakże w pesymistycznym wariancie jest ona równa $O(n^2)$[5].

[1] Opracowanie własne
[2] C.A.R. Hoare: Quicksort. Computer Journal, Vol. 5, 1, 10–15 (1962).
[3] https://eduinf.waw.pl/inf/alg/003_sort/0018.php
[4] https://eduinf.waw.pl/inf/alg/003_sort/0013.php
[5] http://www.algorytm.edu.pl/algorytmy-maturalne/quick-sort.html

Graficzny przykład wykorzystania metody Quick Sort:



Rysunek 1: Przykład sortowania z wykorzystaniem metody Quick Sort[6]

W źródłach dostępne są 4 metody wyboru elementu pivot[7]:

- Pierwszy element zbioru;
- Ostatni element zbioru;
- Losowy element zbioru;
- Element zbioru znajdujący się na medianie zbioru.

Do analizy wykorzystano trzy pierwsze metody wyboru pivot'u.

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

[6] https://www.faceprep.in/c/quick-sort-algorithm-in-c/
[7] https://www.geeksforgeeks.org/quick-sort/

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	Rodzaj pivot w QuickSort	Długość wektora, żeby sortowanie trwało ok 1 sekundy	Długość wektora, żeby sortowanie trwało ok 5 minut
Wektor danych losowych	Losowy element	130 000 elementów	24 000 000 elementów
	Pierwszy element	150 000 elementów	26 000 000 elementów
	Ostatni element	170 000 elementów	31 000 000 elementów
Wektor danych posortowanych rosnąco	Losowy element	130 000 elementów	24 500 000 elementów
	Pierwszy element	4 000 elementów	66 000 elementów
	Ostatni element	2 700 elementów	44 000 elementów
Wektor danych posortowanych malejąco	Losowy element	130 000 elementów	24 000 000 elementów
	Pierwszy element	2 700 elementów	50 000 elementów
	Ostatni element	3 000 elementów	54 000 elementów

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

Biorąc pod uwagę powyższe wyniki czasowe sortowania wektorów oraz uwzględniając czas potrzebny na przeprowadzenie eksperymentu, przyjęto do dalszego eksperymentu pięć punktów pomiarowych, które zostały wyznaczone na wektorze o długości 100 000 elementów, dzięki czemu będzie możliwe przeprowadzenie obserwacji na zauważalnie mierzalnych danych w racjonalnym czasie przeliczeniowym. W ten sposób wyznaczono przedziały przedstawione w poniższej tabeli:

Numer przedziału	Pierwszy element	Ostatni element
1	0	20 000
2	20 001	40 000
3	40 001	60 000
4	60 001	80 000
5	80 001	100 000

Tabela 3: Punkty pomiarowe i ich przedziały[9]

Procedura eksperymentu

Eksperyment polega na wykonaniu pięciokrotnego sortowania każdego z pięciu powyżej określonych odcinków każdym z 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 IDE opisane w podrozdziale Środowisko. Następnie na podstawie otrzymanych wyników policzono średni czas algorytmu $\bar{X}$, odchylenie standardowe σ oraz współczynnik zmienności V wykorzystując poniższe wzory[10]:

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

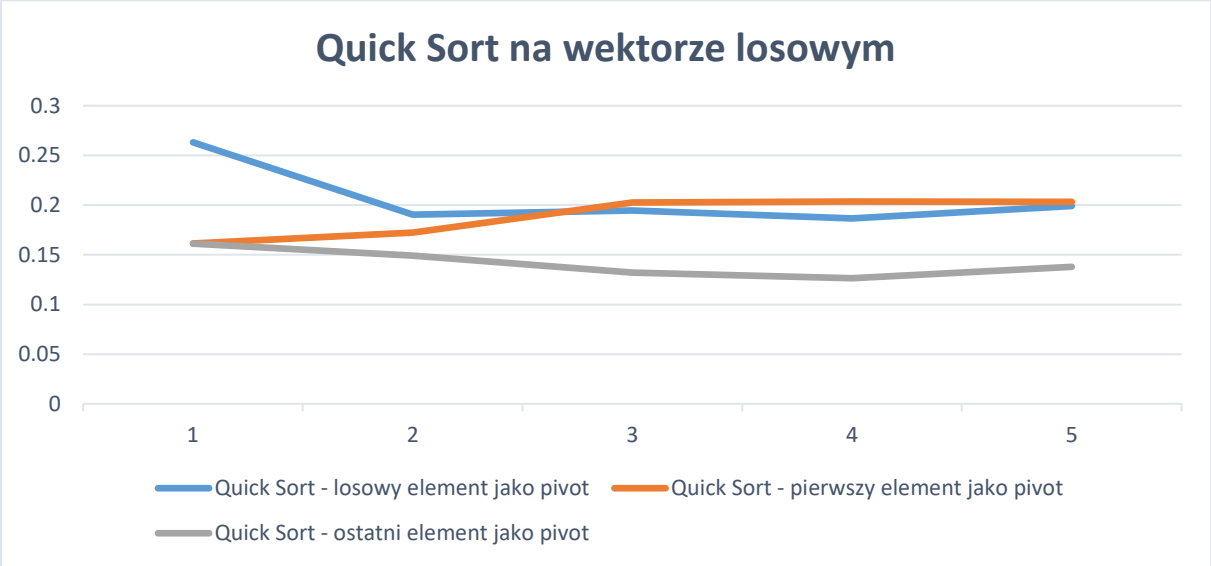
$$\sigma = \sqrt{\frac{\sum_{i=1}^{n}(x_i - \bar{X})^2}{n}}, gdzie\ n - liczba\ obserwacji, x_i - wynik\ obserwacji, \bar{X} - średnia$$

$$V = \frac{\sigma}{\bar{X}}$$

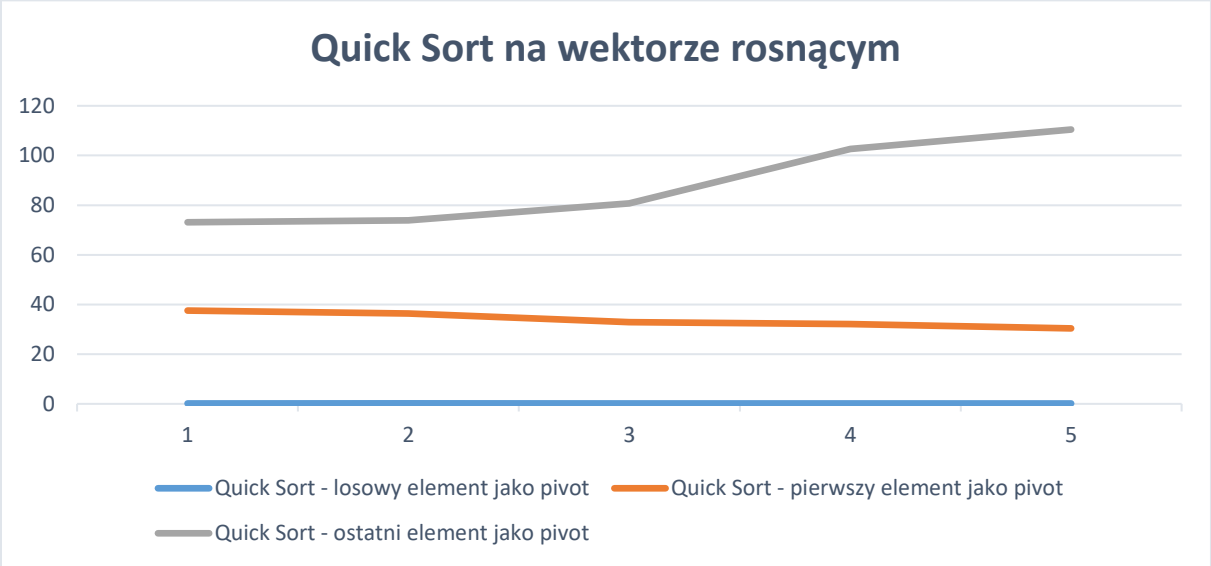
[8] Opracowanie własne
[9] Opracowanie własne
[10] Średnia arytmetyczna: https://www.matemaks.pl/srednia-arytmetyczna.html, odchylenie standardowe: https://www.matemaks.pl/odchylenie-standardowe.html

Wyniki z eksperymentu

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

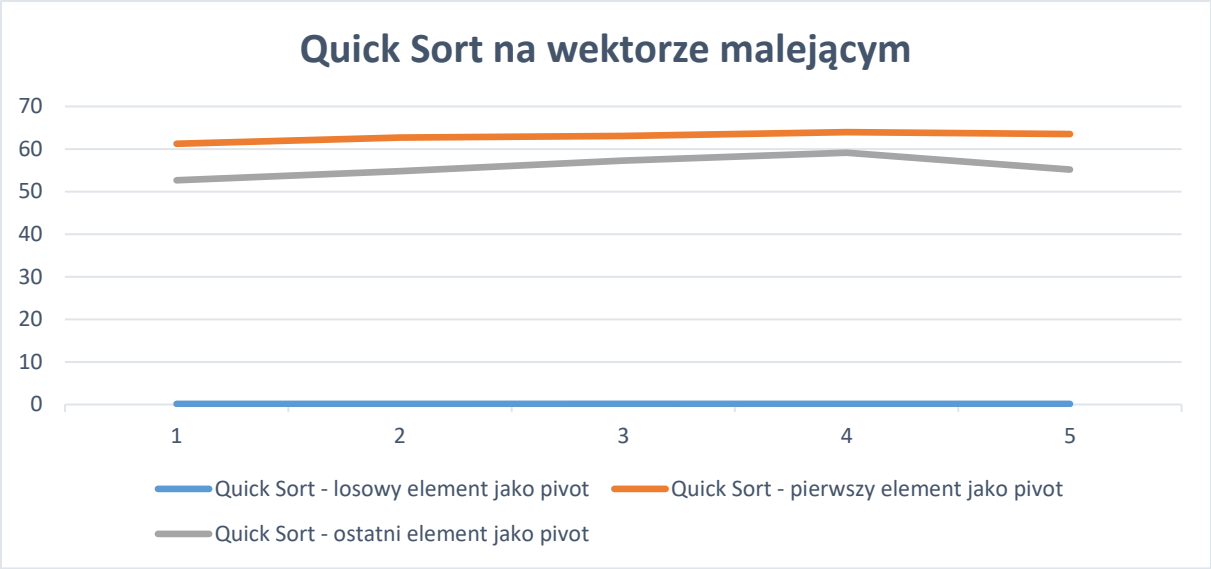


Rysunek 2: Średnie wyniki Quick Sort na danych losowych na odcinkach pomiarowych[11]



Rysunek 3: Średnie wyniki Quick Sort na danych posortowanych rosnąco na odcinkach pomiarowych [12]

[11] Opracowanie własne
[12] Opracowanie własne



Rysunek 4: Średnie wyniki Quick Sort na danych posortowanych malejąco na odcinkach pomiarowych [13]

Z powyższych wykresów można zauważyć, iż sortowanie za pomocą metody Quickort nie zwraca jednorodnych wyników. W zależności od wektora wejściowego oraz metody wyznaczania elementu pivot otrzymywane są zróżnicowane wyniki. Pierwszym spostrzeżeniem po analizie wykresów jest fakt, iż na wektorze losowym wyniki każdej z trzech metod wyznaczania elementu pivot są zbieżne i oscylują pomiędzy 0,1 oraz 0,3 sekundy, podczas gdy zróżnicowanie na wektorach posortowanych rosnąco oraz malejąco jest dużo większe. W obu przypadkach najkrótsze wyniki są osiągane z wykorzystaniem metody wyznaczania elementu pivot, jako losowy element, podczas, gdy zarówno, gdy przyjęto, jako pivot pierwszy lub ostatni element czas sortowania znacząco się wydłużał oraz wyniki były innego rzędu wielkości. W obu przypadkach Quick sort na danych posortowanych rosnąco oraz malejąco dla losowego elementu zbioru, jako pivot był porównywalnej sakli do czasu wyszukiwania na wektorze losowym, zaś pozostałe metody trwały od ok 30 sekund aż do ok 120 sekund. Na podstawie wyników graficznych można stwierdzić, iż najbardziej efektywną metodą sortowania Quick Sort jest metoda z wykorzystaniem elementu losowego zbioru, jako pivot. Zatem wyniki empiryczne potwierdzają wnioski płynące z opisu teoretycznego Quick Sort.

Wyniki finalne i wnioski

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

Dla losowego wektora wejściowego:

Rodzaj pivot w QuickSort	Średnia [s]	Odchylenie standardowe [s]	Współczynnik zmienności
Losowy element	0.206801	0.048241386	23%
Pierwszy element	0.188640	0.036148176	19%
Ostatni element	0.141321	0.029497003	21%

Tabela 4: Średnia, odchylenie standardowe oraz współczynnik zmienności w zależności od zastosowanej metody wyboru element pivot wektorze liczb losowych[14]

[13] Opracowanie własne
[14] Opracowanie własne

Dla posortowanego rosnąco wektora wejściowego:

Rodzaj pivot w QuickSort	Średnia [s]	Odchylenie standardowe [s]	Współczynnik zmienności
Losowy element	0.173919	0.030322373	17%
Pierwszy element	33.872545	2.965251862	9%
Ostatni element	88.147387	19.66682125	22%

Tabela 5: Średnia, odchylenie standardowe oraz współczynnik zmienności w zależności od zastosowanej metody wyboru element pivot wektorze liczb posortowanych rosnąco[15]

Dla posortowanego malejąco wektora wejściowego:

Rodzaj pivot w QuickSort	Średnia [s]	Odchylenie standardowe [s]	Współczynnik zmienności
Losowy element	0.147718	0.011351246	8%
Pierwszy element	62.899279	2.233791186	4%
Ostatni element	55.812223	3.424530034	6%

Tabela 6: Średnia, odchylenie standardowe oraz współczynnik zmienności w zależności od zastosowanej metody wyboru element pivot wektorze liczb posortowanych malejąco[16]

Na podstawie powyższych wyników można stwierdzić następujące:

- Najefektywniejszą metodą sortowania Quick Sort jest metoda z wyborem elementu pivot, jako losowy element zbioru – niezależnie od wektora wejściowego.
- W przypadku, gdy wektor wejściowy jest losowym wektorem, wtedy na podstawie powyższych wyników:
  - Najefektywniejszą metodą jest metoda Quick Sort z ostatnim elementem zbioru, jako pivot – średni czas sortowania wynosi 0,141321 sekundy;
  - Metodą sortowania, charakteryzującą się najmniejszą zmiennością absolutną jest Quick Sort z ostatnim elementem zbioru, jako pivot z odchyleniem standardowym równym 0,029497003 sekundy;
  - Metodą sortowania, charakteryzującą się najmniejszą zmiennością względną jest Quick Sort z pierwszym elementem zbioru, jako pivot z współczynnikiem zmienności równym 19%.
- W przypadku, gdy wektor wejściowy jest posortowanym rosnąco wektorem, wtedy na podstawie powyższych wyników:
  - Najefektywniejszą metodą jest metoda Quick Sort z losowym elementem zbioru, jako pivot – średni czas sortowania wynosi 0,173919 sekundy;
  - Metodą sortowania, charakteryzującą się najmniejszą zmiennością absolutną jest Quick Sort z losowym elementem zbioru, jako pivot z odchyleniem standardowym równym 0,030322373 sekundy;
  - Metodą sortowania, charakteryzującą się najmniejszą zmiennością względną jest Quick Sort z pierwszym elementem zbioru, jako pivot z współczynnikiem zmienności równym 9%, jednakże uwzględniając czas sortowania nie jest to sugerowany determinant wyboru metody sortowania.
- W przypadku, gdy wektor wejściowy jest posortowanym malejąco wektorem, wtedy na podstawie powyższych wyników:
  - Najefektywniejszą metodą jest metoda Quick Sort z losowym elementem zbioru, jako pivot – średni czas sortowania wynosi 0,147718 sekundy;

[15] Opracowanie własne
[16] Opracowanie własne

- Metodą sortowania, charakteryzującą się najmniejszą zmiennością absolutną jest Quick Sort z losowym elementem zbioru, jako pivot z odchyleniem standardowym równym 0,011351246 sekundy;
- Metodą sortowania, charakteryzującą się najmniejszą zmiennością względną jest Quick Sort z pierwszym elementem zbioru, jako pivot z współczynnikiem zmienności równym 4%, jednakże uwzględniając czas sortowania nie jest to sugerowany determinant wyboru metody sortowania.

Reasumując, spośród badanych metod sortowania Quick Sort na podstawie wyników empirycznych sugerowaną metodą do aplikowania w algorytmach sortujących, biorąc pod uwagę trzy atrybuty (średnia, odchylenie standardowe oraz współczynnik zmienności) jest metoda sortowania Quick Sort z losowym elementem zbioru, jako pivot. Przeciętnie zwraca ona najlepsze rezultaty.

Przeprowadzenie eksperymentu II

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.
- Porównanie pięciu algorytmy sortowania:
  - Quick Sort z losowym elementem zbioru, jako pivot
  - Quick Sort z pierwszym elementem zbioru, jako pivot
  - Quick Sort z ostatnim elementem zbioru, jako pivot
  - Merge Sort
  - Heap 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	PyCharm 2020.3.2 (Community Edition) Build #PC-203.6682.179, built on December 30, 2020 Runtime version: 11.0.9.1+11-b1145.63 amd64 VM: OpenJDK 64-Bit Server VM by JetBrains s.r.o. Windows 10 10.0 GC: ParNew, ConcurrentMarkSweep Memory: 974M Cores: 4

Tabela 7: Specyfikacja środowiska, na którym przeprowadzano eksperyment[17]

Przebieg eksperymentu

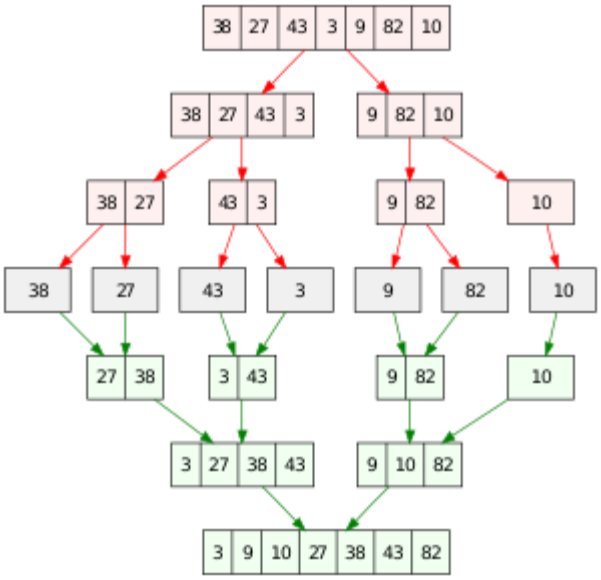
Celem eksperymentu jest porównanie i weryfikacja wyników otrzymanych podczas Eksperymentu I oraz wyników uzyskanych dwiema dodatkowymi metodami sortowania Merge Sort oraz Heap Sort. Analiza została wykonana na tych samych wektorach oraz punktach pomiarowych, które były wykorzystane do analizy Quick Sort w Eksperymencie I.

Opis algorytmów sortowania

Merge Sort

Merge Sort można przetłumaczyć, jako metodę sortowania przez złączanie, jest to bardzo efektywny algorytm sortowania 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.

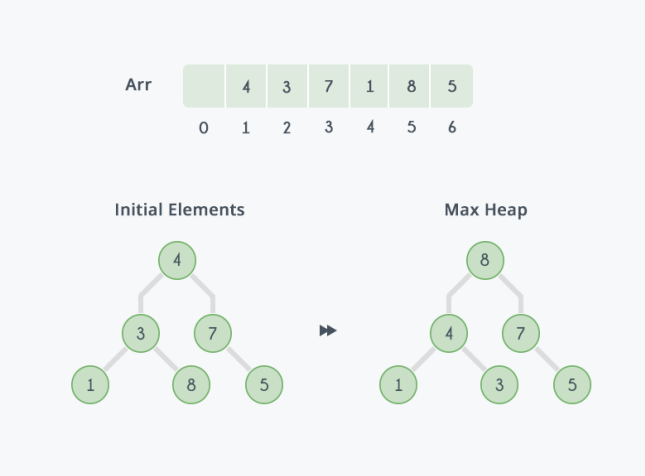
[17] Opracowanie własne



Rysunek 5: Przykład sortowania z wykorzystaniem metody Merge Sort[18].

Heap Sort

Heap Sort z języka angielskiego tłumaczone, jako sortowanie przez kopcowanie (lub sortowanie stogowe), jest to szybki oraz pamięcioooszczędny algorytm sortowania. Polega on na budowanie binarnego kopca oraz sortowaniu właściwym oraz cechuje się złożonością czasową $n(\log n)$. Przeważnie jest wolniejszy niż Quick Sort, jednakże ma przewagę w kontekście niższej złożoności pesymistycznej, również jest przeważnie wolniejszy niż Merge Sort, który ma prostszą definicję. Przykład graficzny przedstawia poniższy rysunek:



Rysunek 6: Przykład sortowania z wykorzystaniem metody Heap Sort[19].

Wyniki eksperymentu

Empiryczne wyniki Merge Sort oraz Heap Sort na danych oraz puntach pomiarowych zdefiniowanych i wykorzystanych podczas procedury Eksperymentu I przedstawiają się po uśrednieniu następująco:

[18] https://en.wikipedia.org/wiki/Merge_sort
[19] https://en.wikipedia.org/wiki/Merge_sort

Wektor wejściowy	Algorytm	Średnia [s]	Odchylenie standardowe [s]	Współczynnik zmienności
Losowy wektor	Merge Sort	0.189879	0.03374724	18%
Losowy wektor	Heap Sort	0.316062	0.081367066	26%
Posortowany rosnąco	Merge Sort	0.180282	0.038919318	22%
Posortowany rosnąco	Heap Sort	0.288774	0.032290955	11%
Posortowany malejąco	Merge Sort	0.172614	0.036970016	21%
Posortowany malejąco	Heap Sort	0.280223	0.04293542	15%

Tabela 8: Średnie, odchylenia standardowe oraz współczynniki zmienności dla metod Merge oraz Heap Sort[20]

Na podstawie powyższych wyników oraz wyników poprzedniego eksperymentu można ustawić zdefiniować następującą hierarchię kierując się średnim czasem sortowania:

- Dla wektora losowego:
  - najszybszą metodą okazał się algorytm Quick Sort z elementem pivot wyznaczonym, jako ostatni element zbioru;
  - najmniejszym odchyleniem standardowym cechuje się również algorytm Quick Sort z elementem pivot wyznaczonym, jako ostatni element zbioru;
  - najmniejszym współczynnikiem zmienności cechuje się algorytm Merge Sort.
- Dla wektora posortowanego rosnąco:
  - najszybszą metodą okazał się algorytm Quick Sort z elementem pivot wyznaczonym, jako losowy element zbioru;
  - najmniejszym odchyleniem standardowym cechuje się również algorytm Quick Sort z elementem pivot wyznaczonym, jako losowy element zbioru;
  - najmniejszym współczynnikiem zmienności jednakże cechuje się algorytm Quick Sort z elementem pivot wyznaczonym, jako pierwszy element zbioru.
- Dla wektora posortowanego malejąco:
  - najszybszą metodą okazał się algorytm Quick Sort z elementem pivot wyznaczonym, jako losowy element zbioru;
  - najmniejszym odchyleniem standardowym cechuje się również algorytm Quick Sort z elementem pivot wyznaczonym, jako losowy element zbioru;
  - najmniejszym współczynnikiem zmienności jednakże cechuje się algorytm Quick Sort z elementem pivot wyznaczonym, jako pierwszy element zbioru.

Zatem podstawowym wnioskiem z eksperymentu empirycznego jest fakt, iż najszybszą metodą sortowania (uwzględniając Quick Sort, Merge Sort oraz Heap Sort) jest metoda Quick Sort. Oba dodatkowe algorytmy cechują się trochę dłuższym czasem sortowania niż najszybszy algorytm równocześnie są szybsze niż pesymistyczne warianty algorytmu Quick Sort. Z tego należy wnioskować, iż w przypadku nieznajomości charakterystyki danych, lub niepewności, co do zastosowanego algorytmu Quick Sort, warto rozważyć skorzystanie z Merge Sort lub Heap Sort, równocześnie biorąc pod uwagę dodatkowy koszt, jakim jest czas sortowania.

[20] Opracowanie własne

Załączniki

Pełne wyniki eksperymentu I

Metoda sortowania	Wektor wejściowy	Odcinek pomiarowy	Wynik [s]
Quick Sort - Element losowy jako pivot	Wektor losowy	1	0.389004
Quick Sort - Element losowy jako pivot	Wektor losowy	1	0.275
Quick Sort - Element losowy jako pivot	Wektor losowy	1	0.222002
Quick Sort - Element losowy jako pivot	Wektor losowy	1	0.213967
Quick Sort - Element losowy jako pivot	Wektor losowy	1	0.215995
Quick Sort - Element losowy jako pivot	Wektor losowy	2	0.177996
Quick Sort - Element losowy jako pivot	Wektor losowy	2	0.203005
Quick Sort - Element losowy jako pivot	Wektor losowy	2	0.182995
Quick Sort - Element losowy jako pivot	Wektor losowy	2	0.21
Quick Sort - Element losowy jako pivot	Wektor losowy	2	0.178006
Quick Sort - Element losowy jako pivot	Wektor losowy	3	0.176002
Quick Sort - Element losowy jako pivot	Wektor losowy	3	0.220008
Quick Sort - Element losowy jako pivot	Wektor losowy	3	0.164001
Quick Sort - Element losowy jako pivot	Wektor losowy	3	0.211001
Quick Sort - Element losowy jako pivot	Wektor losowy	3	0.202005
Quick Sort - Element losowy jako pivot	Wektor losowy	4	0.160989
Quick Sort - Element losowy jako pivot	Wektor losowy	4	0.201999
Quick Sort - Element losowy jako pivot	Wektor losowy	4	0.242001
Quick Sort - Element losowy jako pivot	Wektor losowy	4	0.162037
Quick Sort - Element losowy jako pivot	Wektor losowy	4	0.165994
Quick Sort - Element losowy jako pivot	Wektor losowy	5	0.191
Quick Sort - Element losowy jako pivot	Wektor losowy	5	0.265003
Quick Sort - Element losowy jako pivot	Wektor losowy	5	0.214998
Quick Sort - Element losowy jako pivot	Wektor losowy	5	0.157001
Quick Sort - Element losowy jako pivot	Wektor losowy	5	0.168009
Quick Sort - Element losowy jako pivot	Wektor posortowany malejąco	1	0.157966
Quick Sort - Element losowy jako pivot	Wektor posortowany malejąco	1	0.175998
Quick Sort - Element losowy jako pivot	Wektor posortowany malejąco	1	0.141001
Quick Sort - Element losowy jako pivot	Wektor posortowany malejąco	1	0.136966
Quick Sort - Element losowy jako pivot	Wektor posortowany malejąco	1	0.137969
Quick Sort - Element losowy jako pivot	Wektor posortowany malejąco	2	0.142032
Quick Sort - Element losowy jako pivot	Wektor posortowany malejąco	2	0.134998
Quick Sort - Element losowy jako pivot	Wektor posortowany malejąco	2	0.139963

Quick Sort - Element losowy jako pivot	Wektor posortowany malejąco	2	0.137967
Quick Sort - Element losowy jako pivot	Wektor posortowany malejąco	2	0.140997
Quick Sort - Element losowy jako pivot	Wektor posortowany malejąco	3	0.174039
Quick Sort - Element losowy jako pivot	Wektor posortowany malejąco	3	0.144967
Quick Sort - Element losowy jako pivot	Wektor posortowany malejąco	3	0.155006
Quick Sort - Element losowy jako pivot	Wektor posortowany malejąco	3	0.14559
Quick Sort - Element losowy jako pivot	Wektor posortowany malejąco	3	0.147977
Quick Sort - Element losowy jako pivot	Wektor posortowany malejąco	4	0.154278
Quick Sort - Element losowy jako pivot	Wektor posortowany malejąco	4	0.150034
Quick Sort - Element losowy jako pivot	Wektor posortowany malejąco	4	0.137009
Quick Sort - Element losowy jako pivot	Wektor posortowany malejąco	4	0.137006
Quick Sort - Element losowy jako pivot	Wektor posortowany malejąco	4	0.16103
Quick Sort - Element losowy jako pivot	Wektor posortowany malejąco	5	0.165002
Quick Sort - Element losowy jako pivot	Wektor posortowany malejąco	5	0.152035
Quick Sort - Element losowy jako pivot	Wektor posortowany malejąco	5	0.141099
Quick Sort - Element losowy jako pivot	Wektor posortowany malejąco	5	0.142011
Quick Sort - Element losowy jako pivot	Wektor posortowany malejąco	5	0.140015
Quick Sort - Element losowy jako pivot	Wektor posortowany rosnąco	1	0.190961
Quick Sort - Element losowy jako pivot	Wektor posortowany rosnąco	1	0.152999
Quick Sort - Element losowy jako pivot	Wektor posortowany rosnąco	1	0.153999
Quick Sort - Element losowy jako pivot	Wektor posortowany rosnąco	1	0.162
Quick Sort - Element losowy jako pivot	Wektor posortowany rosnąco	1	0.158007
Quick Sort - Element losowy jako pivot	Wektor posortowany rosnąco	2	0.226003
Quick Sort - Element losowy jako pivot	Wektor posortowany rosnąco	2	0.272
Quick Sort - Element losowy jako pivot	Wektor posortowany rosnąco	2	0.228003
Quick Sort - Element losowy jako pivot	Wektor posortowany rosnąco	2	0.157004
Quick Sort - Element losowy jako pivot	Wektor posortowany rosnąco	2	0.168994
Quick Sort - Element losowy jako pivot	Wektor posortowany rosnąco	3	0.162998
Quick Sort - Element losowy jako pivot	Wektor posortowany rosnąco	3	0.200997
Quick Sort - Element losowy jako pivot	Wektor posortowany rosnąco	3	0.145005
Quick Sort - Element losowy jako pivot	Wektor posortowany rosnąco	3	0.179995
Quick Sort - Element losowy jako pivot	Wektor posortowany rosnąco	3	0.157003
Quick Sort - Element losowy jako pivot	Wektor posortowany rosnąco	4	0.177997
Quick Sort - Element losowy jako pivot	Wektor posortowany rosnąco	4	0.167007
Quick Sort - Element losowy jako pivot	Wektor posortowany rosnąco	4	0.159998
Quick Sort - Element losowy jako pivot	Wektor posortowany rosnąco	4	0.144993
Quick Sort - Element losowy jako pivot	Wektor posortowany rosnąco	4	0.201994

Quick Sort - Element losowy jako pivot	Wektor posortowany rosnąco	5	0.139
Quick Sort - Element losowy jako pivot	Wektor posortowany rosnąco	5	0.162019
Quick Sort - Element losowy jako pivot	Wektor posortowany rosnąco	5	0.155999
Quick Sort - Element losowy jako pivot	Wektor posortowany rosnąco	5	0.164039
Quick Sort - Element losowy jako pivot	Wektor posortowany rosnąco	5	0.158967
Quick Sort - Ostatni element jako pivot	Wektor losowy	1	0.127006
Quick Sort - Ostatni element jako pivot	Wektor losowy	1	0.198005
Quick Sort - Ostatni element jako pivot	Wektor losowy	1	0.157004
Quick Sort - Ostatni element jako pivot	Wektor losowy	1	0.221002
Quick Sort - Ostatni element jako pivot	Wektor losowy	1	0.103036
Quick Sort - Ostatni element jako pivot	Wektor losowy	2	0.126967
Quick Sort - Ostatni element jako pivot	Wektor losowy	2	0.134996
Quick Sort - Ostatni element jako pivot	Wektor losowy	2	0.135003
Quick Sort - Ostatni element jako pivot	Wektor losowy	2	0.131006
Quick Sort - Ostatni element jako pivot	Wektor losowy	2	0.216996
Quick Sort - Ostatni element jako pivot	Wektor losowy	3	0.120005
Quick Sort - Ostatni element jako pivot	Wektor losowy	3	0.123001
Quick Sort - Ostatni element jako pivot	Wektor losowy	3	0.165999
Quick Sort - Ostatni element jako pivot	Wektor losowy	3	0.130989
Quick Sort - Ostatni element jako pivot	Wektor losowy	3	0.120001
Quick Sort - Ostatni element jako pivot	Wektor losowy	4	0.126002
Quick Sort - Ostatni element jako pivot	Wektor losowy	4	0.133999
Quick Sort - Ostatni element jako pivot	Wektor losowy	4	0.122998
Quick Sort - Ostatni element jako pivot	Wektor losowy	4	0.131005
Quick Sort - Ostatni element jako pivot	Wektor losowy	4	0.117988
Quick Sort - Ostatni element jako pivot	Wektor losowy	5	0.134002
Quick Sort - Ostatni element jako pivot	Wektor losowy	5	0.128006
Quick Sort - Ostatni element jako pivot	Wektor losowy	5	0.163999
Quick Sort - Ostatni element jako pivot	Wektor losowy	5	0.132
Quick Sort - Ostatni element jako pivot	Wektor losowy	5	0.132003
Quick Sort - Ostatni element jako pivot	Wektor posortowany malejąco	1	54.737504
Quick Sort - Ostatni element jako pivot	Wektor posortowany malejąco	1	53.946783
Quick Sort - Ostatni element jako pivot	Wektor posortowany malejąco	1	51.312294
Quick Sort - Ostatni element jako pivot	Wektor posortowany malejąco	1	50.200346
Quick Sort - Ostatni element jako pivot	Wektor posortowany malejąco	1	53.122183
Quick Sort - Ostatni element jako pivot	Wektor posortowany malejąco	2	56.126032
Quick Sort - Ostatni element jako pivot	Wektor posortowany malejąco	2	52.506054

Quick Sort - Ostatni element jako pivot	Wektor posortowany malejąco	2	57.488557
Quick Sort - Ostatni element jako pivot	Wektor posortowany malejąco	2	51.705458
Quick Sort - Ostatni element jako pivot	Wektor posortowany malejąco	2	56.210816
Quick Sort - Ostatni element jako pivot	Wektor posortowany malejąco	3	55.943095
Quick Sort - Ostatni element jako pivot	Wektor posortowany malejąco	3	56.741968
Quick Sort - Ostatni element jako pivot	Wektor posortowany malejąco	3	58.960263
Quick Sort - Ostatni element jako pivot	Wektor posortowany malejąco	3	57.828996
Quick Sort - Ostatni element jako pivot	Wektor posortowany malejąco	3	57.066825
Quick Sort - Ostatni element jako pivot	Wektor posortowany malejąco	4	55.539513
Quick Sort - Ostatni element jako pivot	Wektor posortowany malejąco	4	65.368161
Quick Sort - Ostatni element jako pivot	Wektor posortowany malejąco	4	58.449258
Quick Sort - Ostatni element jako pivot	Wektor posortowany malejąco	4	58.936409
Quick Sort - Ostatni element jako pivot	Wektor posortowany malejąco	4	57.395317
Quick Sort - Ostatni element jako pivot	Wektor posortowany malejąco	5	57.85312
Quick Sort - Ostatni element jako pivot	Wektor posortowany malejąco	5	59.324117
Quick Sort - Ostatni element jako pivot	Wektor posortowany malejąco	5	49.807533
Quick Sort - Ostatni element jako pivot	Wektor posortowany malejąco	5	57.100068
Quick Sort - Ostatni element jako pivot	Wektor posortowany malejąco	5	51.634893
Quick Sort - Ostatni element jako pivot	Wektor posortowany rosnąco	1	72.901865
Quick Sort - Ostatni element jako pivot	Wektor posortowany rosnąco	1	72.717183
Quick Sort - Ostatni element jako pivot	Wektor posortowany rosnąco	1	74.198299
Quick Sort - Ostatni element jako pivot	Wektor posortowany rosnąco	1	73.980759
Quick Sort - Ostatni element jako pivot	Wektor posortowany rosnąco	1	71.782067
Quick Sort - Ostatni element jako pivot	Wektor posortowany rosnąco	2	71.59586
Quick Sort - Ostatni element jako pivot	Wektor posortowany rosnąco	2	75.348225
Quick Sort - Ostatni element jako pivot	Wektor posortowany rosnąco	2	73.968154
Quick Sort - Ostatni element jako pivot	Wektor posortowany rosnąco	2	75.064364
Quick Sort - Ostatni element jako pivot	Wektor posortowany rosnąco	2	73.225272
Quick Sort - Ostatni element jako pivot	Wektor posortowany rosnąco	3	73.792211
Quick Sort - Ostatni element jako pivot	Wektor posortowany rosnąco	3	72.520641
Quick Sort - Ostatni element jako pivot	Wektor posortowany rosnąco	3	71.839774
Quick Sort - Ostatni element jako pivot	Wektor posortowany rosnąco	3	90.612478
Quick Sort - Ostatni element jako pivot	Wektor posortowany rosnąco	3	94.686365
Quick Sort - Ostatni element jako pivot	Wektor posortowany rosnąco	4	89.351006
Quick Sort - Ostatni element jako pivot	Wektor posortowany rosnąco	4	88.745482
Quick Sort - Ostatni element jako pivot	Wektor posortowany rosnąco	4	85.679684
Quick Sort - Ostatni element jako pivot	Wektor posortowany rosnąco	4	118.256279

Quick Sort - Ostatni element jako pivot	Wektor posortowany rosnąco	4	131.13024
Quick Sort - Ostatni element jako pivot	Wektor posortowany rosnąco	5	122.089967
Quick Sort - Ostatni element jako pivot	Wektor posortowany rosnąco	5	122.873488
Quick Sort - Ostatni element jako pivot	Wektor posortowany rosnąco	5	125.72949
Quick Sort - Ostatni element jako pivot	Wektor posortowany rosnąco	5	100.592076
Quick Sort - Ostatni element jako pivot	Wektor posortowany rosnąco	5	81.003441
Quick Sort - Pierwszy element jako pivot	Wektor losowy	1	0.165004
Quick Sort - Pierwszy element jako pivot	Wektor losowy	1	0.141998
Quick Sort - Pierwszy element jako pivot	Wektor losowy	1	0.196
Quick Sort - Pierwszy element jako pivot	Wektor losowy	1	0.155997
Quick Sort - Pierwszy element jako pivot	Wektor losowy	1	0.148
Quick Sort - Pierwszy element jako pivot	Wektor losowy	2	0.17599
Quick Sort - Pierwszy element jako pivot	Wektor losowy	2	0.178999
Quick Sort - Pierwszy element jako pivot	Wektor losowy	2	0.150001
Quick Sort - Pierwszy element jako pivot	Wektor losowy	2	0.181003
Quick Sort - Pierwszy element jako pivot	Wektor losowy	2	0.174999
Quick Sort - Pierwszy element jako pivot	Wektor losowy	3	0.242986
Quick Sort - Pierwszy element jako pivot	Wektor losowy	3	0.194011
Quick Sort - Pierwszy element jako pivot	Wektor losowy	3	0.188997
Quick Sort - Pierwszy element jako pivot	Wektor losowy	3	0.253002
Quick Sort - Pierwszy element jako pivot	Wektor losowy	3	0.134002
Quick Sort - Pierwszy element jako pivot	Wektor losowy	4	0.240998
Quick Sort - Pierwszy element jako pivot	Wektor losowy	4	0.214999
Quick Sort - Pierwszy element jako pivot	Wektor losowy	4	0.135003
Quick Sort - Pierwszy element jako pivot	Wektor losowy	4	0.221
Quick Sort - Pierwszy element jako pivot	Wektor losowy	4	0.205997
Quick Sort - Pierwszy element jako pivot	Wektor losowy	5	0.262006
Quick Sort - Pierwszy element jako pivot	Wektor losowy	5	0.203001
Quick Sort - Pierwszy element jako pivot	Wektor losowy	5	0.174998
Quick Sort - Pierwszy element jako pivot	Wektor losowy	5	0.159999
Quick Sort - Pierwszy element jako pivot	Wektor losowy	5	0.217003
Quick Sort - Pierwszy element jako pivot	Wektor posortowany malejąco	1	59.048925
Quick Sort - Pierwszy element jako pivot	Wektor posortowany malejąco	1	59.844497
Quick Sort - Pierwszy element jako pivot	Wektor posortowany malejąco	1	65.286864
Quick Sort - Pierwszy element jako pivot	Wektor posortowany malejąco	1	63.231309
Quick Sort - Pierwszy element jako pivot	Wektor posortowany malejąco	1	58.803458
Quick Sort - Pierwszy element jako pivot	Wektor posortowany malejąco	2	59.633496

Quick Sort - Pierwszy element jako pivot	Wektor posortowany malejąco	2	65.380559
Quick Sort - Pierwszy element jako pivot	Wektor posortowany malejąco	2	63.056684
Quick Sort - Pierwszy element jako pivot	Wektor posortowany malejąco	2	61.227858
Quick Sort - Pierwszy element jako pivot	Wektor posortowany malejąco	2	64.214327
Quick Sort - Pierwszy element jako pivot	Wektor posortowany malejąco	3	60.972375
Quick Sort - Pierwszy element jako pivot	Wektor posortowany malejąco	3	65.13325
Quick Sort - Pierwszy element jako pivot	Wektor posortowany malejąco	3	60.60819
Quick Sort - Pierwszy element jako pivot	Wektor posortowany malejąco	3	63.01405
Quick Sort - Pierwszy element jako pivot	Wektor posortowany malejąco	3	65.578426
Quick Sort - Pierwszy element jako pivot	Wektor posortowany malejąco	4	62.422531
Quick Sort - Pierwszy element jako pivot	Wektor posortowany malejąco	4	64.871969
Quick Sort - Pierwszy element jako pivot	Wektor posortowany malejąco	4	66.305241
Quick Sort - Pierwszy element jako pivot	Wektor posortowany malejąco	4	62.675355
Quick Sort - Pierwszy element jako pivot	Wektor posortowany malejąco	4	63.565496
Quick Sort - Pierwszy element jako pivot	Wektor posortowany malejąco	5	63.820262
Quick Sort - Pierwszy element jako pivot	Wektor posortowany malejąco	5	65.549069
Quick Sort - Pierwszy element jako pivot	Wektor posortowany malejąco	5	62.994331
Quick Sort - Pierwszy element jako pivot	Wektor posortowany malejąco	5	64.992643
Quick Sort - Pierwszy element jako pivot	Wektor posortowany malejąco	5	60.250808
Quick Sort - Pierwszy element jako pivot	Wektor posortowany rosnąco	1	36.720686
Quick Sort - Pierwszy element jako pivot	Wektor posortowany rosnąco	1	38.440158
Quick Sort - Pierwszy element jako pivot	Wektor posortowany rosnąco	1	37.915722
Quick Sort - Pierwszy element jako pivot	Wektor posortowany rosnąco	1	36.941141
Quick Sort - Pierwszy element jako pivot	Wektor posortowany rosnąco	1	37.787687
Quick Sort - Pierwszy element jako pivot	Wektor posortowany rosnąco	2	37.257185
Quick Sort - Pierwszy element jako pivot	Wektor posortowany rosnąco	2	38.600655
Quick Sort - Pierwszy element jako pivot	Wektor posortowany rosnąco	2	37.401648
Quick Sort - Pierwszy element jako pivot	Wektor posortowany rosnąco	2	32.207634
Quick Sort - Pierwszy element jako pivot	Wektor posortowany rosnąco	2	36.716159
Quick Sort - Pierwszy element jako pivot	Wektor posortowany rosnąco	3	33.834142
Quick Sort - Pierwszy element jako pivot	Wektor posortowany rosnąco	3	32.885145
Quick Sort - Pierwszy element jako pivot	Wektor posortowany rosnąco	3	32.347112
Quick Sort - Pierwszy element jako pivot	Wektor posortowany rosnąco	3	32.777621
Quick Sort - Pierwszy element jako pivot	Wektor posortowany rosnąco	3	32.634147
Quick Sort - Pierwszy element jako pivot	Wektor posortowany rosnąco	4	32.03463
Quick Sort - Pierwszy element jako pivot	Wektor posortowany rosnąco	4	32.917637
Quick Sort - Pierwszy element jako pivot	Wektor posortowany rosnąco	4	31.587613

Quick Sort - Pierwszy element jako pivot	Wektor posortowany rosnąco	4	32.132107
Quick Sort - Pierwszy element jako pivot	Wektor posortowany rosnąco	4	31.744112
Quick Sort - Pierwszy element jako pivot	Wektor posortowany rosnąco	5	32.385103
Quick Sort - Pierwszy element jako pivot	Wektor posortowany rosnąco	5	30.155781
Quick Sort - Pierwszy element jako pivot	Wektor posortowany rosnąco	5	30.424092
Quick Sort - Pierwszy element jako pivot	Wektor posortowany rosnąco	5	28.245046
Quick Sort - Pierwszy element jako pivot	Wektor posortowany rosnąco	5	30.72067

Pełne wyniki eksperymentu II

Metoda sortowania	Wektor wejściowy	Odcinek pomiarowy	Wynik [s]
Heap Sort	Wektor losowy	1	0.270004
Heap Sort	Wektor losowy	1	0.261116
Heap Sort	Wektor losowy	1	0.247001
Heap Sort	Wektor losowy	1	0.466976
Heap Sort	Wektor losowy	1	0.29004
Heap Sort	Wektor losowy	2	0.260975
Heap Sort	Wektor losowy	2	0.402002
Heap Sort	Wektor losowy	2	0.287082
Heap Sort	Wektor losowy	2	0.307011
Heap Sort	Wektor losowy	2	0.389006
Heap Sort	Wektor losowy	3	0.382069
Heap Sort	Wektor losowy	3	0.312098
Heap Sort	Wektor losowy	3	0.244981
Heap Sort	Wektor losowy	3	0.275003
Heap Sort	Wektor losowy	3	0.605054
Heap Sort	Wektor losowy	4	0.24905
Heap Sort	Wektor losowy	4	0.301008
Heap Sort	Wektor losowy	4	0.301999
Heap Sort	Wektor losowy	4	0.26201
Heap Sort	Wektor losowy	4	0.246001
Heap Sort	Wektor losowy	5	0.276011
Heap Sort	Wektor losowy	5	0.318006
Heap Sort	Wektor losowy	5	0.378007
Heap Sort	Wektor losowy	5	0.303002
Heap Sort	Wektor losowy	5	0.266042
Heap Sort	Wektor posortowany malejąco	1	0.230971
Heap Sort	Wektor posortowany malejąco	1	0.247004
Heap Sort	Wektor posortowany malejąco	1	0.257038
Heap Sort	Wektor posortowany malejąco	1	0.234007
Heap Sort	Wektor posortowany malejąco	1	0.334972
Heap Sort	Wektor posortowany malejąco	2	0.343008
Heap Sort	Wektor posortowany malejąco	2	0.343521
Heap Sort	Wektor posortowany malejąco	2	0.330043
Heap Sort	Wektor posortowany malejąco	2	0.282972

Heap Sort	Wektor posortowany malejąco	2	0.35093
Heap Sort	Wektor posortowany malejąco	3	0.277009
Heap Sort	Wektor posortowany malejąco	3	0.275
Heap Sort	Wektor posortowany malejąco	3	0.249039
Heap Sort	Wektor posortowany malejąco	3	0.360046
Heap Sort	Wektor posortowany malejąco	3	0.230002
Heap Sort	Wektor posortowany malejąco	4	0.312973
Heap Sort	Wektor posortowany malejąco	4	0.262998
Heap Sort	Wektor posortowany malejąco	4	0.265008
Heap Sort	Wektor posortowany malejąco	4	0.331
Heap Sort	Wektor posortowany malejąco	4	0.227039
Heap Sort	Wektor posortowany malejąco	5	0.233003
Heap Sort	Wektor posortowany malejąco	5	0.264971
Heap Sort	Wektor posortowany malejąco	5	0.259005
Heap Sort	Wektor posortowany malejąco	5	0.248006
Heap Sort	Wektor posortowany malejąco	5	0.256007
Heap Sort	Wektor posortowany rosnąco	1	0.336
Heap Sort	Wektor posortowany rosnąco	1	0.368974
Heap Sort	Wektor posortowany rosnąco	1	0.291003
Heap Sort	Wektor posortowany rosnąco	1	0.334044
Heap Sort	Wektor posortowany rosnąco	1	0.35104
Heap Sort	Wektor posortowany rosnąco	2	0.299975
Heap Sort	Wektor posortowany rosnąco	2	0.272968
Heap Sort	Wektor posortowany rosnąco	2	0.334096
Heap Sort	Wektor posortowany rosnąco	2	0.269969
Heap Sort	Wektor posortowany rosnąco	2	0.249044
Heap Sort	Wektor posortowany rosnąco	3	0.267971
Heap Sort	Wektor posortowany rosnąco	3	0.288002
Heap Sort	Wektor posortowany rosnąco	3	0.276566
Heap Sort	Wektor posortowany rosnąco	3	0.263975
Heap Sort	Wektor posortowany rosnąco	3	0.283002
Heap Sort	Wektor posortowany rosnąco	4	0.297003
Heap Sort	Wektor posortowany rosnąco	4	0.256043
Heap Sort	Wektor posortowany rosnąco	4	0.262046
Heap Sort	Wektor posortowany rosnąco	4	0.313979
Heap Sort	Wektor posortowany rosnąco	4	0.283003
Heap Sort	Wektor posortowany rosnąco	5	0.264006
Heap Sort	Wektor posortowany rosnąco	5	0.263007
Heap Sort	Wektor posortowany rosnąco	5	0.259044
Heap Sort	Wektor posortowany rosnąco	5	0.27904
Heap Sort	Wektor posortowany rosnąco	5	0.255552
Merge Sort	Wektor losowy	1	0.185007
Merge Sort	Wektor losowy	1	0.188001
Merge Sort	Wektor losowy	1	0.163337
Merge Sort	Wektor losowy	1	0.194376

Merge Sort	Wektor losowy	1	0.170053
Merge Sort	Wektor losowy	2	0.186047
Merge Sort	Wektor losowy	2	0.194006
Merge Sort	Wektor losowy	2	0.174973
Merge Sort	Wektor losowy	2	0.154035
Merge Sort	Wektor losowy	2	0.170004
Merge Sort	Wektor losowy	3	0.158002
Merge Sort	Wektor losowy	3	0.165005
Merge Sort	Wektor losowy	3	0.203004
Merge Sort	Wektor losowy	3	0.17803
Merge Sort	Wektor losowy	3	0.171004
Merge Sort	Wektor losowy	4	0.162963
Merge Sort	Wektor losowy	4	0.173032
Merge Sort	Wektor losowy	4	0.177047
Merge Sort	Wektor losowy	4	0.198001
Merge Sort	Wektor losowy	4	0.213997
Merge Sort	Wektor losowy	5	0.295005
Merge Sort	Wektor losowy	5	0.172005
Merge Sort	Wektor losowy	5	0.279
Merge Sort	Wektor losowy	5	0.234006
Merge Sort	Wektor losowy	5	0.187038
Merge Sort	Wektor posortowany malejąco	1	0.151634
Merge Sort	Wektor posortowany malejąco	1	0.140058
Merge Sort	Wektor posortowany malejąco	1	0.136955
Merge Sort	Wektor posortowany malejąco	1	0.152077
Merge Sort	Wektor posortowany malejąco	1	0.154067
Merge Sort	Wektor posortowany malejąco	2	0.195
Merge Sort	Wektor posortowany malejąco	2	0.153996
Merge Sort	Wektor posortowany malejąco	2	0.158998
Merge Sort	Wektor posortowany malejąco	2	0.145969
Merge Sort	Wektor posortowany malejąco	2	0.13908
Merge Sort	Wektor posortowany malejąco	3	0.139035
Merge Sort	Wektor posortowany malejąco	3	0.164014
Merge Sort	Wektor posortowany malejąco	3	0.17202
Merge Sort	Wektor posortowany malejąco	3	0.142037
Merge Sort	Wektor posortowany malejąco	3	0.14238
Merge Sort	Wektor posortowany malejąco	4	0.154001
Merge Sort	Wektor posortowany malejąco	4	0.140007
Merge Sort	Wektor posortowany malejąco	4	0.140007
Merge Sort	Wektor posortowany malejąco	4	0.217002
Merge Sort	Wektor posortowany malejąco	4	0.222008
Merge Sort	Wektor posortowany malejąco	5	0.248006
Merge Sort	Wektor posortowany malejąco	5	0.225997
Merge Sort	Wektor posortowany malejąco	5	0.212003
Merge Sort	Wektor posortowany malejąco	5	0.228005

Merge Sort	Wektor posortowany malejąco	5	0.241005
Merge Sort	Wektor posortowany rosnąco	1	0.139004
Merge Sort	Wektor posortowany rosnąco	1	0.231001
Merge Sort	Wektor posortowany rosnąco	1	0.143004
Merge Sort	Wektor posortowany rosnąco	1	0.166971
Merge Sort	Wektor posortowany rosnąco	1	0.241003
Merge Sort	Wektor posortowany rosnąco	2	0.128039
Merge Sort	Wektor posortowany rosnąco	2	0.135995
Merge Sort	Wektor posortowany rosnąco	2	0.140576
Merge Sort	Wektor posortowany rosnąco	2	0.232972
Merge Sort	Wektor posortowany rosnąco	2	0.244004
Merge Sort	Wektor posortowany rosnąco	3	0.19101
Merge Sort	Wektor posortowany rosnąco	3	0.230004
Merge Sort	Wektor posortowany rosnąco	3	0.170005
Merge Sort	Wektor posortowany rosnąco	3	0.196039
Merge Sort	Wektor posortowany rosnąco	3	0.134392
Merge Sort	Wektor posortowany rosnąco	4	0.199008
Merge Sort	Wektor posortowany rosnąco	4	0.187002
Merge Sort	Wektor posortowany rosnąco	4	0.145039
Merge Sort	Wektor posortowany rosnąco	4	0.143149
Merge Sort	Wektor posortowany rosnąco	4	0.221004
Merge Sort	Wektor posortowany rosnąco	5	0.187003
Merge Sort	Wektor posortowany rosnąco	5	0.184037
Merge Sort	Wektor posortowany rosnąco	5	0.137065
Merge Sort	Wektor posortowany rosnąco	5	0.231099
Merge Sort	Wektor posortowany rosnąco	5	0.148625
Heap Sort	Wektor losowy	1	0.270004
Heap Sort	Wektor losowy	1	0.261116
Heap Sort	Wektor losowy	1	0.247001
Heap Sort	Wektor losowy	1	0.466976
Heap Sort	Wektor losowy	1	0.29004
Heap Sort	Wektor losowy	2	0.260975
Heap Sort	Wektor losowy	2	0.402002
Heap Sort	Wektor losowy	2	0.287082
Heap Sort	Wektor losowy	2	0.307011
Heap Sort	Wektor losowy	2	0.389006
Heap Sort	Wektor losowy	3	0.382069
Heap Sort	Wektor losowy	3	0.312098
Heap Sort	Wektor losowy	3	0.244981
Heap Sort	Wektor losowy	3	0.275003
Heap Sort	Wektor losowy	3	0.605054
Heap Sort	Wektor losowy	4	0.24905
Heap Sort	Wektor losowy	4	0.301008
Heap Sort	Wektor losowy	4	0.301999
Heap Sort	Wektor losowy	4	0.26201

Heap Sort	Wektor losowy	4	0.246001
Heap Sort	Wektor losowy	5	0.276011
Heap Sort	Wektor losowy	5	0.318006
Heap Sort	Wektor losowy	5	0.378007
Heap Sort	Wektor losowy	5	0.303002
Heap Sort	Wektor losowy	5	0.266042
Heap Sort	Wektor posortowany malejąco	1	0.230971
Heap Sort	Wektor posortowany malejąco	1	0.247004
Heap Sort	Wektor posortowany malejąco	1	0.257038
Heap Sort	Wektor posortowany malejąco	1	0.234007
Heap Sort	Wektor posortowany malejąco	1	0.334972
Heap Sort	Wektor posortowany malejąco	2	0.343008
Heap Sort	Wektor posortowany malejąco	2	0.343521
Heap Sort	Wektor posortowany malejąco	2	0.330043
Heap Sort	Wektor posortowany malejąco	2	0.282972
Heap Sort	Wektor posortowany malejąco	2	0.35093
Heap Sort	Wektor posortowany malejąco	3	0.277009
Heap Sort	Wektor posortowany malejąco	3	0.275
Heap Sort	Wektor posortowany malejąco	3	0.249039
Heap Sort	Wektor posortowany malejąco	3	0.360046
Heap Sort	Wektor posortowany malejąco	3	0.230002
Heap Sort	Wektor posortowany malejąco	4	0.312973
Heap Sort	Wektor posortowany malejąco	4	0.262998
Heap Sort	Wektor posortowany malejąco	4	0.265008
Heap Sort	Wektor posortowany malejąco	4	0.331
Heap Sort	Wektor posortowany malejąco	4	0.227039
Heap Sort	Wektor posortowany malejąco	5	0.233003
Heap Sort	Wektor posortowany malejąco	5	0.264971
Heap Sort	Wektor posortowany malejąco	5	0.259005
Heap Sort	Wektor posortowany malejąco	5	0.248006
Heap Sort	Wektor posortowany malejąco	5	0.256007
Heap Sort	Wektor posortowany rosnąco	1	0.336
Heap Sort	Wektor posortowany rosnąco	1	0.368974
Heap Sort	Wektor posortowany rosnąco	1	0.291003
Heap Sort	Wektor posortowany rosnąco	1	0.334044
Heap Sort	Wektor posortowany rosnąco	1	0.35104
Heap Sort	Wektor posortowany rosnąco	2	0.299975
Heap Sort	Wektor posortowany rosnąco	2	0.272968
Heap Sort	Wektor posortowany rosnąco	2	0.334096
Heap Sort	Wektor posortowany rosnąco	2	0.269969
Heap Sort	Wektor posortowany rosnąco	2	0.249044
Heap Sort	Wektor posortowany rosnąco	3	0.267971
Heap Sort	Wektor posortowany rosnąco	3	0.288002
Heap Sort	Wektor posortowany rosnąco	3	0.276566
Heap Sort	Wektor posortowany rosnąco	3	0.263975

Heap Sort	Wektor posortowany rosnąco	3	0.283002
Heap Sort	Wektor posortowany rosnąco	4	0.297003
Heap Sort	Wektor posortowany rosnąco	4	0.256043
Heap Sort	Wektor posortowany rosnąco	4	0.262046
Heap Sort	Wektor posortowany rosnąco	4	0.313979
Heap Sort	Wektor posortowany rosnąco	4	0.283003
Heap Sort	Wektor posortowany rosnąco	5	0.264006
Heap Sort	Wektor posortowany rosnąco	5	0.263007
Heap Sort	Wektor posortowany rosnąco	5	0.259044
Heap Sort	Wektor posortowany rosnąco	5	0.27904
Heap Sort	Wektor posortowany rosnąco	5	0.255552

Kod programu

Poniżej załączono kod programu, którego rozwój jest widoczny poprzez platformę github.com: https://github.com/oksiedz/Python/tree/master/Projects/Project3.

```
import random
import datetime


# ##############################Quick sort code
# This function is same in both iterative and recursive
def partition_end(arr, start, stop):
	index = start - 1
	x = arr[stop]

	for j in range(start, stop):
		if arr[j] <= x:
			# increment index of smaller element
			index = index + 1
			arr[index], arr[j] = arr[j], arr[index]

	arr[index + 1], arr[stop] = arr[stop], arr[index + 1]
	return index + 1


def partition_start(arr, start, stop):
	pivot = start
	index = start + 1
	for j in range(start + 1, stop + 1):
		if arr[j] <= arr[pivot]:
			arr[index], arr[j] = arr[j], arr[index]
			index = index + 1
	arr[pivot], arr[index - 1] = arr[index - 1], arr[pivot]
	return index - 1


def partition_rand(arr, start, stop):
	rand_pivot = random.randrange(start, stop)
	arr[start], arr[rand_pivot] = arr[rand_pivot], arr[start]
	return partition_start(arr, start, stop)


# Function to do Quick sort
# arr[] --> Array to be sorted,
# start --> Starting index,
# stop --> Ending index
def quick_sort_iterative(arr, start, stop, mode):

	# Create an auxiliary stack
	size = stop - start + 1
	stack = [0] * size
	# initialize top of stack
	top = -1
```

```
		# push initial values of l and h to stack
		top = top + 1
		stack[top] = start
		top = top + 1
		stack[top] = stop

		# Keep popping from stack while is not empty
		while top >= 0:

			# Pop h and l
			stop = stack[top]
			top = top - 1
			start = stack[top]
			top = top - 1

			# Set pivot element at its correct position in
			# sorted array
			p = 0
			if mode == 1:
				p = partition_rand(arr, start, stop)
			if mode == 2:
				p = partition_start(arr, start, stop)
			if mode == 3:
				p = partition_end(arr, start, stop)

			# If there are elements on left side of pivot,
			# then push left side to stack
			if p - 1 > start:
				top = top + 1
				stack[top] = start
				top = top + 1
				stack[top] = p - 1

			# If there are elements on right side of pivot,
			# then push right side to stack
			if p + 1 < stop:
				top = top + 1
				stack[top] = p + 1
				top = top + 1
				stack[top] = stop


def quick_sort_engine(input_array, q_s_mode=1, input_type="Z", if_save=0, save_results=0, measure_point=0):
		a = []
		for i1 in range(0, len(input_array)):
			a.append(input_array[i1])
		print("Quicksort Start: " + str(input_type))
		st = datetime.datetime.now()
		quick_sort_iterative(input_array, 0, len(a) - 1, q_s_mode)
		et = datetime.datetime.now()
		print("Quicksort End: " + str(input_type))
		if if_save == 1:
			for i2 in range(len(input_array)):
				sortedArrayAsc.append(input_array[i2])
		if save_results == 1:
			resultsList.append("Q;" + str(q_s_mode) + ";"+str(input_type) + ";" + str(measure_point) + ";" + str(et - st))


# ##############################Merge Sort code
def merge_sort(array):
		if len(array) > 1:
			mid = len(array)//2
			left_half = array[:mid]
			right_half = array[mid:]

			merge_sort(left_half)
			merge_sort(right_half)

			i1 = 0
			j = 0
			k = 0
			while i1 < len(left_half) and j < len(right_half):
				if left_half[i1] <= right_half[j]:
					array[k] = left_half[i1]
					i1 = i1 + 1
				else:
```

```
                                        array[k] = right_half[j]
                                        j = j + 1
                                k = k + 1

                        while i1 < len(left_half):
                                array[k] = left_half[i1]
                                i1 = i1 + 1
                                k = k + 1

                        while j < len(right_half):
                                array[k] = right_half[j]
                                j = j + 1
                                k = k + 1


def merge_sort_engine(input_array, input_type="Z", if_save=0, save_results=0, measure_point=0):
        a = []
        for i1 in range(0, len(input_array)):
                a.append(input_array[i1])
        print("Start - Merge sort")
        start_time = datetime.datetime.now()
        merge_sort(a)
        end_time = datetime.datetime.now()
        print("End - Merge sort")
        if if_save == 1:
                for i2 in range(len(a)):
                        sortedArrayAsc.append(a[i2])
        if save_results == 1:
                resultsList.append("M;"+str(input_type)+";"+str(measure_point)+";"+str(end_time-start_time))


# ##############################HeapSort
def heapify(arr, n, iterator):
        largest = iterator
        left = 2 * iterator + 1
        right = 2 * iterator + 2

        if left < n and arr[largest] < arr[left]:
                largest = left
        if right < n and arr[largest] < arr[right]:
                largest = right
        if largest != iterator:
                arr[iterator], arr[largest] = arr[largest], arr[iterator]
                heapify(arr, n, largest)


def heap_sort(arr):
        n = len(arr)

        for i1 in range(n//2 - 1, -1, -1):
                heapify(arr, n, i1)

        for i2 in range(n-1, 0, -1):
                arr[i2], arr[0] = arr[0], arr[i2]
                heapify(arr, i2, 0)


def heap_sort_engine(input_array, input_type="Z", if_save=0, save_results=0, measure_point=0):
        a = []
        for i1 in range(0, len(input_array)):
                a.append(input_array[i1])
        print("Start - Heap sort")
        start_time = datetime.datetime.now()
        heap_sort(a)
        end_time = datetime.datetime.now()
        print("End - Heap sort")
        if if_save == 1:
                for i2 in range(len(a)):
                        sortedArrayAsc.append(a[i2])
        if save_results == 1:
                resultsList.append("H;"+str(input_type)+";"+str(measure_point)+";"+str(end_time - start_time))


# ##############################Working code
resultsList = []
sortedArrayAsc = []
```

```
sortedArrayDesc = []
array_input = []
random_array = []
noOfGeneratedNumber = 100000

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

# print("Input array is:")
# print(random_array)

array_input = list(random_array)
quick_sort_engine(input_array=array_input, q_s_mode=1, input_type="R", if_save=1, save_results=0, measure_point=0)
print("reverse sorted Start")
sortedArrayDesc = list(reversed(sortedArrayAsc))
print("reverse sorted End")

loop_start = 0
loop_end = 5
print("Quick sort calculation")
print("Start - random array with random pivot")
print("Start: First loop")
start_number = 0
end_number = 20000
for i in range(loop_start, loop_end):
	array_input = list(random_array[start_number:end_number])
	quick_sort_engine(input_array=array_input, q_s_mode=1, input_type="R", if_save=0, save_results=1, measure_point=1)
print("End: First loop")
print("Start: Second loop")
start_number = 20001
end_number = 40000
for i in range(loop_start, loop_end):
	array_input = list(random_array[start_number:end_number])
	quick_sort_engine(input_array=array_input, q_s_mode=1, input_type="R", if_save=0, save_results=1, measure_point=2)
print("End: Second loop")
print("Start: Third loop")
start_number = 40001
end_number = 60000
for i in range(loop_start, loop_end):
	array_input = list(random_array[start_number:end_number])
	quick_sort_engine(input_array=array_input, q_s_mode=1, input_type="R", if_save=0, save_results=1, measure_point=3)
print("End: Third loop")
print("Start: Fourth loop")
start_number = 60001
end_number = 80000
for i in range(loop_start, loop_end):
	array_input = list(random_array[start_number:end_number])
	quick_sort_engine(input_array=array_input, q_s_mode=1, input_type="R", if_save=0, save_results=1, measure_point=4)
print("End: Fourth loop")
print("Start: Fifth loop")
start_number = 80001
end_number = 100000
for i in range(loop_start, loop_end):
	array_input = list(random_array[start_number:end_number])
	quick_sort_engine(input_array=array_input, q_s_mode=1, input_type="R", if_save=0, save_results=1, measure_point=5)
print("End: Fifth loop")
print("Start - random array with first pivot")
print("Start: First loop")
start_number = 0
end_number = 20000
for i in range(loop_start, loop_end):
	array_input = list(random_array[start_number:end_number])
	quick_sort_engine(input_array=array_input, q_s_mode=2, input_type="R", if_save=0, save_results=1, measure_point=1)
print("End: First loop")
print("Start: Second loop")
start_number = 20001
end_number = 40000
for i in range(loop_start, loop_end):
	array_input = list(random_array[start_number:end_number])
	quick_sort_engine(input_array=array_input, q_s_mode=2, input_type="R", if_save=0, save_results=1, measure_point=2)
print("End: Second loop")
print("Start: Third loop")
start_number = 40001
end_number = 60000
```

```
for i in range(loop_start, loop_end):
		array_input = list(random_array[start_number:end_number])
		quick_sort_engine(input_array=array_input, q_s_mode=2, input_type="R", if_save=0, save_results=1, measure_point=3)
print("End: Third loop")
print("Start: Fourth loop")
start_number = 60001
end_number = 80000
for i in range(loop_start, loop_end):
		array_input = list(random_array[start_number:end_number])
		quick_sort_engine(input_array=array_input, q_s_mode=2, input_type="R", if_save=0, save_results=1, measure_point=4)
print("End: Fourth loop")
print("Start: Fifth loop")
start_number = 80001
end_number = 100000
for i in range(loop_start, loop_end):
		array_input = list(random_array[start_number:end_number])
		quick_sort_engine(input_array=array_input, q_s_mode=2, input_type="R", if_save=0, save_results=1, measure_point=5)
print("End: Fifth loop")
print("Start - random array with end pivot")
print("Start: First loop")
start_number = 0
end_number = 20000
for i in range(loop_start, loop_end):
		array_input = list(random_array[start_number:end_number])
		quick_sort_engine(input_array=array_input, q_s_mode=3, input_type="R", if_save=0, save_results=1, measure_point=1)
print("End: First loop")
print("Start: Second loop")
start_number = 20001
end_number = 40000
for i in range(loop_start, loop_end):
		array_input = list(random_array[start_number:end_number])
		quick_sort_engine(input_array=array_input, q_s_mode=3, input_type="R", if_save=0, save_results=1, measure_point=2)
print("End: Second loop")
print("Start: Third loop")
start_number = 40001
end_number = 60000
for i in range(loop_start, loop_end):
		array_input = list(random_array[start_number:end_number])
		quick_sort_engine(input_array=array_input, q_s_mode=3, input_type="R", if_save=0, save_results=1, measure_point=3)
print("End: Third loop")
print("Start: Fourth loop")
start_number = 60001
end_number = 80000
for i in range(loop_start, loop_end):
		array_input = list(random_array[start_number:end_number])
		quick_sort_engine(input_array=array_input, q_s_mode=3, input_type="R", if_save=0, save_results=1, measure_point=4)
print("End: Fourth loop")
print("Start: Fifth loop")
start_number = 80001
end_number = 100000
for i in range(loop_start, loop_end):
		array_input = list(random_array[start_number:end_number])
		quick_sort_engine(input_array=array_input, q_s_mode=3, input_type="R", if_save=0, save_results=1, measure_point=5)
print("End: Fifth loop")

print("Start - sorted ASC array with random pivot")
print("Start: First loop")
start_number = 0
end_number = 20000
for i in range(loop_start, loop_end):
		array_input = list(sortedArrayAsc[start_number:end_number])
		quick_sort_engine(input_array=array_input, q_s_mode=1, input_type="A", if_save=0, save_results=1, measure_point=1)
print("End: First loop")
print("Start: Second loop")
start_number = 20001
end_number = 40000
for i in range(loop_start, loop_end):
		array_input = list(sortedArrayAsc[start_number:end_number])
		quick_sort_engine(input_array=array_input, q_s_mode=1, input_type="A", if_save=0, save_results=1, measure_point=2)
print("End: Second loop")
print("Start: Third loop")
start_number = 40001
end_number = 60000
for i in range(loop_start, loop_end):
		array_input = list(sortedArrayAsc[start_number:end_number])
		quick_sort_engine(input_array=array_input, q_s_mode=1, input_type="A", if_save=0, save_results=1, measure_point=3)
```

```
print("End: Third loop")
print("Start: Fourth loop")
start_number = 60001
end_number = 80000
for i in range(loop_start, loop_end):
        array_input = list(sortedArrayAsc[start_number:end_number])
        quick_sort_engine(input_array=array_input, q_s_mode=1, input_type="A", if_save=0, save_results=1, measure_point=4)
print("End: Fourth loop")
print("Start: Fifth loop")
start_number = 80001
end_number = 100000
for i in range(loop_start, loop_end):
        array_input = list(sortedArrayAsc[start_number:end_number])
        quick_sort_engine(input_array=array_input, q_s_mode=1, input_type="A", if_save=0, save_results=1, measure_point=5)
print("End: Fifth loop")
print("Start - sorted ASC array with first pivot")
print("Start: First loop")
start_number = 0
end_number = 20000
for i in range(loop_start, loop_end):
        array_input = list(sortedArrayAsc[start_number:end_number])
        quick_sort_engine(input_array=array_input, q_s_mode=2, input_type="A", if_save=0, save_results=1, measure_point=1)
print("End: First loop")
print("Start: Second loop")
start_number = 20001
end_number = 40000
for i in range(loop_start, loop_end):
        array_input = list(sortedArrayAsc[start_number:end_number])
        quick_sort_engine(input_array=array_input, q_s_mode=2, input_type="A", if_save=0, save_results=1, measure_point=2)
print("End: Second loop")
print("Start: Third loop")
start_number = 40001
end_number = 60000
for i in range(loop_start, loop_end):
        array_input = list(sortedArrayAsc[start_number:end_number])
        quick_sort_engine(input_array=array_input, q_s_mode=2, input_type="A", if_save=0, save_results=1, measure_point=3)
print("End: Third loop")
print("Start: Fourth loop")
start_number = 60001
end_number = 80000
for i in range(loop_start, loop_end):
        array_input = list(sortedArrayAsc[start_number:end_number])
        quick_sort_engine(input_array=array_input, q_s_mode=2, input_type="A", if_save=0, save_results=1, measure_point=4)
print("End: Fourth loop")
print("Start: Fifth loop")
start_number = 80001
end_number = 100000
for i in range(loop_start, loop_end):
        array_input = list(sortedArrayAsc[start_number:end_number])
        quick_sort_engine(input_array=array_input, q_s_mode=2, input_type="A", if_save=0, save_results=1, measure_point=5)
print("End: Fifth loop")
print("Start - sorted asc array with end pivot")
print("Start: First loop")
start_number = 0
end_number = 20000
for i in range(loop_start, loop_end):
        array_input = list(sortedArrayAsc[start_number:end_number])
        quick_sort_engine(input_array=array_input, q_s_mode=3, input_type="A", if_save=0, save_results=1, measure_point=1)
print("End: First loop")
print("Start: Second loop")
start_number = 20001
end_number = 40000
for i in range(loop_start, loop_end):
        array_input = list(sortedArrayAsc[start_number:end_number])
        quick_sort_engine(input_array=array_input, q_s_mode=3, input_type="A", if_save=0, save_results=1, measure_point=2)
print("End: Second loop")
print("Start: Third loop")
start_number = 40001
end_number = 60000
for i in range(loop_start, loop_end):
        array_input = list(sortedArrayAsc[start_number:end_number])
        quick_sort_engine(input_array=array_input, q_s_mode=3, input_type="A", if_save=0, save_results=1, measure_point=3)
print("End: Third loop")
print("Start: Fourth loop")
start_number = 60001
end_number = 80000
```

```
for i in range(loop_start, loop_end):
		array_input = list(sortedArrayAsc[start_number:end_number])
		quick_sort_engine(input_array=array_input, q_s_mode=3, input_type="A", if_save=0, save_results=1, measure_point=4)
print("End: Fourth loop")
print("Start: Fifth loop")
start_number = 80001
end_number = 100000
for i in range(loop_start, loop_end):
		array_input = list(sortedArrayAsc[start_number:end_number])
		quick_sort_engine(input_array=array_input, q_s_mode=3, input_type="A", if_save=0, save_results=1, measure_point=5)
print("End: Fifth loop")

print("Start - sorted DSC array with random pivot")
print("Start: First loop")
start_number = 0
end_number = 20000
for i in range(loop_start, loop_end):
		array_input = list(sortedArrayDesc[start_number:end_number])
		quick_sort_engine(input_array=array_input, q_s_mode=1, input_type="D", if_save=0, save_results=1, measure_point=1)
print("End: First loop")
print("Start: Second loop")
start_number = 20001
end_number = 40000
for i in range(loop_start, loop_end):
		array_input = list(sortedArrayDesc[start_number:end_number])
		quick_sort_engine(input_array=array_input, q_s_mode=1, input_type="D", if_save=0, save_results=1, measure_point=2)
print("End: Second loop")
print("Start: Third loop")
start_number = 40001
end_number = 60000
for i in range(loop_start, loop_end):
		array_input = list(sortedArrayDesc[start_number:end_number])
		quick_sort_engine(input_array=array_input, q_s_mode=1, input_type="D", if_save=0, save_results=1, measure_point=3)
print("End: Third loop")
print("Start: Fourth loop")
start_number = 60001
end_number = 80000
for i in range(loop_start, loop_end):
		array_input = list(sortedArrayDesc[start_number:end_number])
		quick_sort_engine(input_array=array_input, q_s_mode=1, input_type="D", if_save=0, save_results=1, measure_point=4)
print("End: Fourth loop")
print("Start: Fifth loop")
start_number = 80001
end_number = 100000
for i in range(loop_start, loop_end):
		array_input = list(sortedArrayDesc[start_number:end_number])
		quick_sort_engine(input_array=array_input, q_s_mode=1, input_type="D", if_save=0, save_results=1, measure_point=5)
print("End: Fifth loop")
print("Start - sorted DSC array with first pivot")
print("Start: First loop")
start_number = 0
end_number = 20000
for i in range(loop_start, loop_end):
		array_input = list(sortedArrayDesc[start_number:end_number])
		quick_sort_engine(input_array=array_input, q_s_mode=2, input_type="D", if_save=0, save_results=1, measure_point=1)
print("End: First loop")
print("Start: Second loop")
start_number = 20001
end_number = 40000
for i in range(loop_start, loop_end):
		array_input = list(sortedArrayDesc[start_number:end_number])
		quick_sort_engine(input_array=array_input, q_s_mode=2, input_type="D", if_save=0, save_results=1, measure_point=2)
print("End: Second loop")
print("Start: Third loop")
start_number = 40001
end_number = 60000
for i in range(loop_start, loop_end):
		array_input = list(sortedArrayDesc[start_number:end_number])
		quick_sort_engine(input_array=array_input, q_s_mode=2, input_type="D", if_save=0, save_results=1, measure_point=3)
print("End: Third loop")
print("Start: Fourth loop")
start_number = 60001
end_number = 80000
for i in range(loop_start, loop_end):
		array_input = list(sortedArrayDesc[start_number:end_number])
		quick_sort_engine(input_array=array_input, q_s_mode=2, input_type="D", if_save=0, save_results=1, measure_point=4)
```

```
print("End: Fourth loop")
print("Start: Fifth loop")
start_number = 80001
end_number = 100000
for i in range(loop_start, loop_end):
	array_input = list(sortedArrayDesc[start_number:end_number])
	quick_sort_engine(input_array=array_input, q_s_mode=2, input_type="D", if_save=0, save_results=1, measure_point=5)
print("End: Fifth loop")
print("Start - sorted DSC array with end pivot")
print("Start: First loop")
start_number = 0
end_number = 20000
for i in range(loop_start, loop_end):
	array_input = list(sortedArrayDesc[start_number:end_number])
	quick_sort_engine(input_array=array_input, q_s_mode=3, input_type="D", if_save=0, save_results=1, measure_point=1)
print("End: First loop")
print("Start: Second loop")
start_number = 20001
end_number = 40000
for i in range(loop_start, loop_end):
	array_input = list(sortedArrayDesc[start_number:end_number])
	quick_sort_engine(input_array=array_input, q_s_mode=3, input_type="D", if_save=0, save_results=1, measure_point=2)
print("End: Second loop")
print("Start: Third loop")
start_number = 40001
end_number = 60000
for i in range(loop_start, loop_end):
	array_input = list(sortedArrayDesc[start_number:end_number])
	quick_sort_engine(input_array=array_input, q_s_mode=3, input_type="D", if_save=0, save_results=1, measure_point=3)
print("End: Third loop")
print("Start: Fourth loop")
start_number = 60001
end_number = 80000
for i in range(loop_start, loop_end):
	array_input = list(sortedArrayDesc[start_number:end_number])
	quick_sort_engine(input_array=array_input, q_s_mode=3, input_type="D", if_save=0, save_results=1, measure_point=4)
print("End: Fourth loop")
print("Start: Fifth loop")
start_number = 80001
end_number = 100000
for i in range(loop_start, loop_end):
	array_input = list(sortedArrayDesc[start_number:end_number])
	quick_sort_engine(input_array=array_input, q_s_mode=3, input_type="D", if_save=0, save_results=1, measure_point=5)
print("End: Fifth loop")

print("Merge sort calculation")
print("Start - random array")
print("Start: First loop")
start_number = 0
end_number = 20000
for i in range(loop_start, loop_end):
	array_input = list(random_array[start_number:end_number])
	merge_sort_engine(input_array=array_input, input_type="R", if_save=0, save_results=1, measure_point=1)
print("End: First loop")
print("Start: Second loop")
start_number = 20001
end_number = 40000
for i in range(loop_start, loop_end):
	array_input = list(random_array[start_number:end_number])
	merge_sort_engine(input_array=array_input, input_type="R", if_save=0, save_results=1, measure_point=2)
print("End: Second loop")
print("Start: Third loop")
start_number = 40001
end_number = 60000
for i in range(loop_start, loop_end):
	array_input = list(random_array[start_number:end_number])
	merge_sort_engine(input_array=array_input, input_type="R", if_save=0, save_results=1, measure_point=3)
print("End: Third loop")
print("Start: Fourth loop")
start_number = 60001
end_number = 80000
for i in range(loop_start, loop_end):
	array_input = list(random_array[start_number:end_number])
	merge_sort_engine(input_array=array_input, input_type="R", if_save=0, save_results=1, measure_point=4)
print("End: Fourth loop")
print("Start: Fifth loop")
```

```
start_number = 80001
end_number = 100000
for i in range(loop_start, loop_end):
        array_input = list(random_array[start_number:end_number])
        merge_sort_engine(input_array=array_input, input_type="R", if_save=0, save_results=1, measure_point=5)
print("End: Fifth loop")

print("Start - sorted ASC array")
print("Start: First loop")
start_number = 0
end_number = 20000
for i in range(loop_start, loop_end):
        array_input = list(sortedArrayAsc[start_number:end_number])
        merge_sort_engine(input_array=array_input, input_type="A", if_save=0, save_results=1, measure_point=1)
print("End: First loop")
print("Start: Second loop")
start_number = 20001
end_number = 40000
for i in range(loop_start, loop_end):
        array_input = list(sortedArrayAsc[start_number:end_number])
        merge_sort_engine(input_array=array_input, input_type="A", if_save=0, save_results=1, measure_point=2)
print("End: Second loop")
print("Start: Third loop")
start_number = 40001
end_number = 60000
for i in range(loop_start, loop_end):
        array_input = list(sortedArrayAsc[start_number:end_number])
        merge_sort_engine(input_array=array_input, input_type="A", if_save=0, save_results=1, measure_point=3)
print("End: Third loop")
print("Start: Fourth loop")
start_number = 60001
end_number = 80000
for i in range(loop_start, loop_end):
        array_input = list(sortedArrayAsc[start_number:end_number])
        merge_sort_engine(input_array=array_input, input_type="A", if_save=0, save_results=1, measure_point=4)
print("End: Fourth loop")
print("Start: Fifth loop")
start_number = 80001
end_number = 100000
for i in range(loop_start, loop_end):
        array_input = list(sortedArrayAsc[start_number:end_number])
        merge_sort_engine(input_array=array_input, input_type="A", if_save=0, save_results=1, measure_point=5)
print("End: Fifth loop")

print("Start - sorted DSC array")
print("Start: First loop")
start_number = 0
end_number = 20000
for i in range(loop_start, loop_end):
        array_input = list(sortedArrayDesc[start_number:end_number])
        merge_sort_engine(input_array=array_input, input_type="D", if_save=0, save_results=1, measure_point=1)
print("End: First loop")
print("Start: Second loop")
start_number = 20001
end_number = 40000
for i in range(loop_start, loop_end):
        array_input = list(sortedArrayDesc[start_number:end_number])
        merge_sort_engine(input_array=array_input, input_type="D", if_save=0, save_results=1, measure_point=2)
print("End: Second loop")
print("Start: Third loop")
start_number = 40001
end_number = 60000
for i in range(loop_start, loop_end):
        array_input = list(sortedArrayDesc[start_number:end_number])
        merge_sort_engine(input_array=array_input, input_type="D", if_save=0, save_results=1, measure_point=3)
print("End: Third loop")
print("Start: Fourth loop")
start_number = 60001
end_number = 80000
for i in range(loop_start, loop_end):
        array_input = list(sortedArrayDesc[start_number:end_number])
        merge_sort_engine(input_array=array_input, input_type="D", if_save=0, save_results=1, measure_point=4)
print("End: Fourth loop")
print("Start: Fifth loop")
start_number = 80001
end_number = 100000
```

```
for i in range(loop_start, loop_end):
		array_input = list(sortedArrayDesc[start_number:end_number])
		merge_sort_engine(input_array=array_input, input_type="D", if_save=0, save_results=1, measure_point=5)
print("End: Fifth loop")



print("Heap sort calculation")
print("Start - random array")
print("Start: First loop")
start_number = 0
end_number = 20000
for i in range(loop_start, loop_end):
		array_input = list(random_array[start_number:end_number])
		heap_sort_engine(input_array=array_input, input_type="R", if_save=0, save_results=1, measure_point=1)
print("End: First loop")
print("Start: Second loop")
start_number = 20001
end_number = 40000
for i in range(loop_start, loop_end):
		array_input = list(random_array[start_number:end_number])
		heap_sort_engine(input_array=array_input, input_type="R", if_save=0, save_results=1, measure_point=2)
print("End: Second loop")
print("Start: Third loop")
start_number = 40001
end_number = 60000
for i in range(loop_start, loop_end):
		array_input = list(random_array[start_number:end_number])
		heap_sort_engine(input_array=array_input, input_type="R", if_save=0, save_results=1, measure_point=3)
print("End: Third loop")
print("Start: Fourth loop")
start_number = 60001
end_number = 80000
for i in range(loop_start, loop_end):
		array_input = list(random_array[start_number:end_number])
		heap_sort_engine(input_array=array_input, input_type="R", if_save=0, save_results=1, measure_point=4)
print("End: Fourth loop")
print("Start: Fifth loop")
start_number = 80001
end_number = 100000
for i in range(loop_start, loop_end):
		array_input = list(random_array[start_number:end_number])
		heap_sort_engine(input_array=array_input, input_type="R", if_save=0, save_results=1, measure_point=5)
print("End: Fifth loop")

print("Start - sorted ASC array")
print("Start: First loop")
start_number = 0
end_number = 20000
for i in range(loop_start, loop_end):
		array_input = list(sortedArrayAsc[start_number:end_number])
		heap_sort_engine(input_array=array_input, input_type="A", if_save=0, save_results=1, measure_point=1)
print("End: First loop")
print("Start: Second loop")
start_number = 20001
end_number = 40000
for i in range(loop_start, loop_end):
		array_input = list(sortedArrayAsc[start_number:end_number])
		heap_sort_engine(input_array=array_input, input_type="A", if_save=0, save_results=1, measure_point=2)
print("End: Second loop")
print("Start: Third loop")
start_number = 40001
end_number = 60000
for i in range(loop_start, loop_end):
		array_input = list(sortedArrayAsc[start_number:end_number])
		heap_sort_engine(input_array=array_input, input_type="A", if_save=0, save_results=1, measure_point=3)
print("End: Third loop")
print("Start: Fourth loop")
start_number = 60001
end_number = 80000
for i in range(loop_start, loop_end):
		array_input = list(sortedArrayAsc[start_number:end_number])
		heap_sort_engine(input_array=array_input, input_type="A", if_save=0, save_results=1, measure_point=4)
print("End: Fourth loop")
print("Start: Fifth loop")
start_number = 80001
```

```
end_number = 100000
for i in range(loop_start, loop_end):
        array_input = list(sortedArrayAsc[start_number:end_number])
        heap_sort_engine(input_array=array_input, input_type="A", if_save=0, save_results=1, measure_point=5)
print("End: Fifth loop")

print("Start - sorted DSC array")
print("Start: First loop")
start_number = 0
end_number = 20000
for i in range(loop_start, loop_end):
        array_input = list(sortedArrayDesc[start_number:end_number])
        heap_sort_engine(input_array=array_input, input_type="D", if_save=0, save_results=1, measure_point=1)
print("End: First loop")
print("Start: Second loop")
start_number = 20001
end_number = 40000
for i in range(loop_start, loop_end):
        array_input = list(sortedArrayDesc[start_number:end_number])
        heap_sort_engine(input_array=array_input, input_type="D", if_save=0, save_results=1, measure_point=2)
print("End: Second loop")
print("Start: Third loop")
start_number = 40001
end_number = 60000
for i in range(loop_start, loop_end):
        array_input = list(sortedArrayDesc[start_number:end_number])
        heap_sort_engine(input_array=array_input, input_type="D", if_save=0, save_results=1, measure_point=3)
print("End: Third loop")
print("Start: Fourth loop")
start_number = 60001
end_number = 80000
for i in range(loop_start, loop_end):
        array_input = list(sortedArrayDesc[start_number:end_number])
        heap_sort_engine(input_array=array_input, input_type="D", if_save=0, save_results=1, measure_point=4)
print("End: Fourth loop")
print("Start: Fifth loop")
start_number = 80001
end_number = 100000
for i in range(loop_start, loop_end):
        array_input = list(sortedArrayDesc[start_number:end_number])
        heap_sort_engine(input_array=array_input, input_type="D", if_save=0, save_results=1, measure_point=5)
print("End: Fifth loop")



# print ("Sorted array is:")
# print(sortedArrayAsc)
# print(randomArray)

print("Results:")
for i in resultsList:
        print(i)
```