Algorytmy Sortowania

Jakob Wolitzki (grupa I2 136830)

Przebieg ćwiczenia:

I. 1. Porównaj szybkość działania 4 metod sortowania: BS, HS, CS, ShS dla tablicy liczb całkowitych generowanych losowo zgodnie z równomiernym rozkładem prawdopodobieństwa. Przedstaw wykres *t=f(n)* gdzie: *t* - czas sortowania; *n* - liczba elementów tablicy. Liczbę elementów należy dobrać w taki sposób, aby możliwe było wykonanie pomiarów. Wyniki przedstawić na jednym wykresie (przynajmniej 15 punktów pomiarowych).

2. Sformułuj wnioski odnośnie złożoności obliczeniowej badanych metod i ich związku z efektywnością sortowania oraz zajętością pamięciową każdej z nich.

II. 1. Dla różnych typów danych wejściowych porównaj efektywność działania 3 algorytmów sortowania. QS ze środkowym elementem wyboru, b) HS oraz MS. Zbadaj działanie dla następujących typów danych:

- losowy (rozkład jednorodny)
- ciąg stały (np. równy 0)
- ciąg rosnący (co 1)
- ciąg malejący (co 1)
- rosnąco-malejący
- malejąco- rosnący

Przedstaw wykresy *t=f(n)* gdzie: *t* - czas sortowania; *n* - liczba elementów tablicy dla różnych typów danych (2 kolejne typy na jednym wykresie - 6 charakterystyk). Liczbę elementów należy dobrać w taki sposób, aby możliwe było wykonanie pomiarów. Wyniki przedstawić na wykresach - jeden dla dwóch typów danych (przynajmniej 15 punktów pomiarowych).

2. Sformułować wnioski odnośnie złożoności obliczeniowej i efektywności wykonywania QS oraz zachowania się algorytmu w najgorszym przypadku i dla poszczególnych typów danych. Jaki wpływ ma mediana na czas sortowania QS? W jakim celu jest ona stosowana?

Ćwiczenie 1.

Celem zadania jest implementacja algorytmów sortowania: BS, HS, CS, ShS, testowanie oraz porównanie szybkości dla tablicy liczb całkowitych generowanych losowo zgodnie z równomiernym rozkładem prawdopodobieństwa. Następnie przedstawienie danych na wykresach i sformułowanie wniosków.

1.

Zacznijmy od opisania poszczególnych algorytmów sortujących:

- Bubble Sort (BS) jest jednym z najwolniejszych algorytmów sortujących, jego złożoność w średnim i najgorszym przypadku jest równa $O(n^2)$. Gdy ciąg jest posortowany na wejściu to złożoność wynosi O(n). Sortowanie odbywa się w miejscu, do jego działania nie jest potrzebna pamięć dodatkowa, elementy sortowane przechowywane są przez cały czas w tablicy wejściowej. Bubble Sort polega na porównywaniu sąsiednich par elementów i zamienianiu ich gdy element znajdujący się bliżej początku jest większy od elementu kolejnego. W kolejnej iteracji powtarzamy daną czynność do elementu n-1 (n to wielkość tablicy), gdyż mamy pewność, że wartość maksymalna znajduje się na końcu. W moim algorytmie użyłem również flagi, która pozwala na przerwanie sortowania, gdy ciąg jest posortowany (nie zostanie wykonana żadna zamiana elementów w danej iteracji).

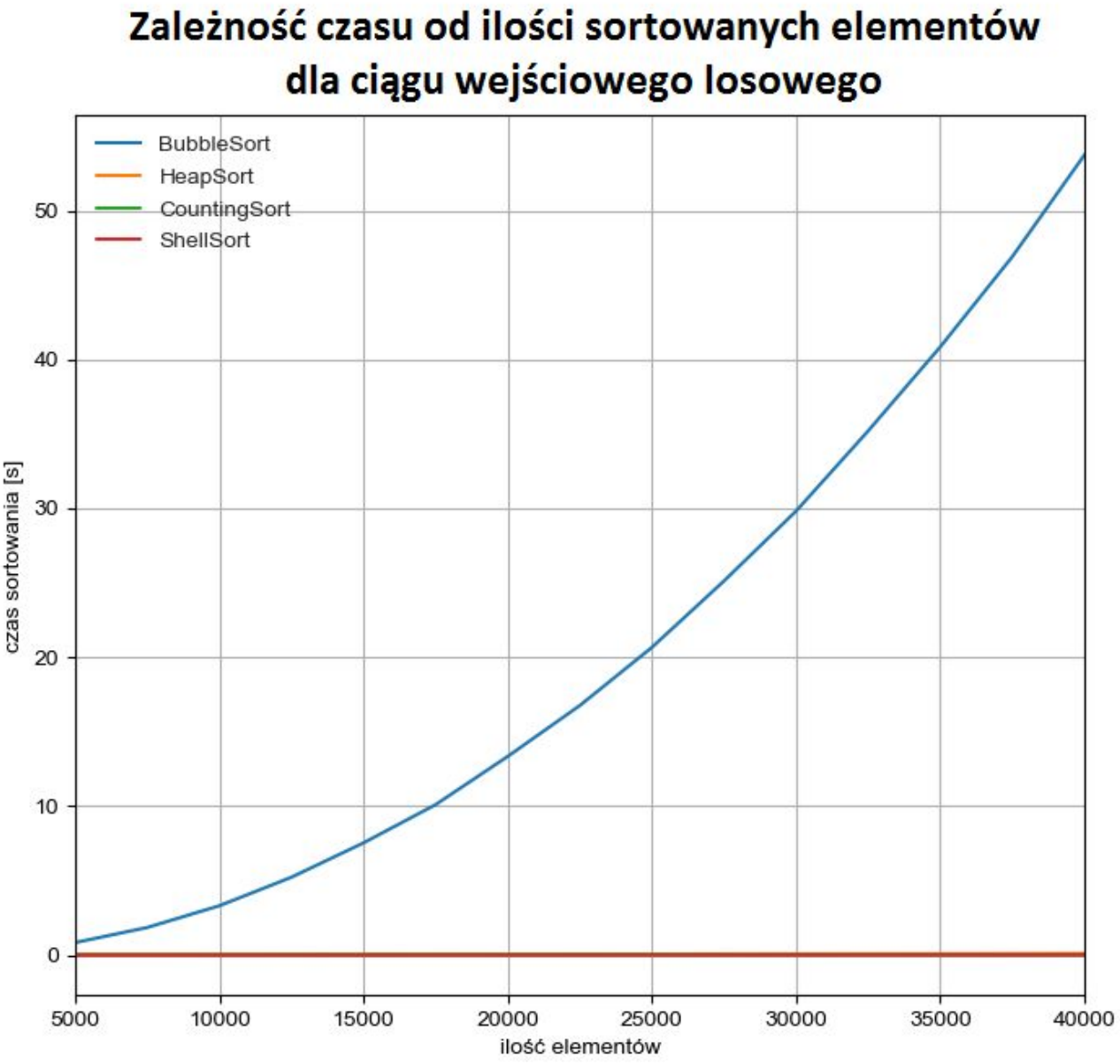
- Heap Sort (HS) posiada złożoność obliczeniową w każdym przypadku O(n logn). Złożoność pamięciowa wynosi O(n), nie tworzymy żadnej dodatkowej struktury przechowującej dane, sortowanie w miejscu. Algorytm można podzielić na dwa etapy. Najpierw z danych wejściowych tworzymy strukturę kopca, której złożoność wynosi O(n log n). Gdy nasze elementy są już ułożone w odpowiednim porządku, przechodzimy do etapu drugiego. Zdejmujemy korzeń kopca w danej iteracji, zamieniając go z

elementem ostatnim. Po każdej takiej iteracji, przywracamy strukturę kopca, złożoność tej operacji wynosi O(logn). Operację powtarzamy, aż do wyczerpania elementów w kopcu co daje nam całkowitą złożoność O(n log n). W wyniku wykonania tych dwóch etapów, nasza tablica wejściowa będzie posortowana.

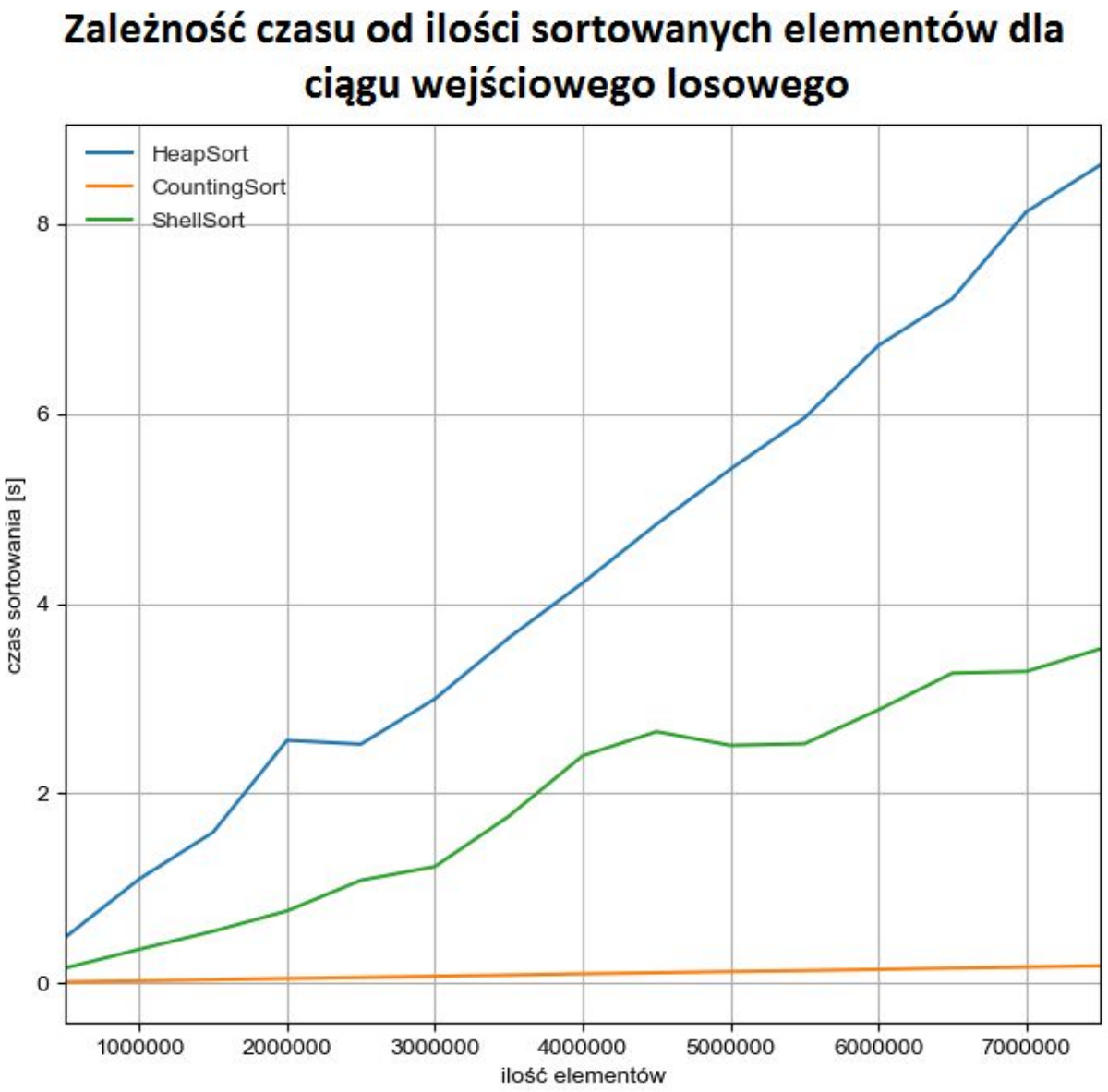
- Counting Sort (CS) polega na liczeniu ile razy dana liczba występuje w ciągu, który chcemy posortować. Jest bardzo szybki, jego złożoność obliczeniowa wynosi O(n+k), gdzie k jest równe maksymalnemu elementowi ciągu. Złożoność pamięciowa wynosi tyle samo co złożoność obliczeniowa, jest to spowodowane faktem, iż CS nie jest sortowaniem w miejscu. Potrzebna jest nam dodatkowa tablica przechowująca liczniki wystąpień liczb z ciągu wejściowego.

- Shell Sort (ShS) bazuje na algorytmie sortowania przez wstawianie, z różnicą, że sortowany ciąg jest dzielony na podzbiory, w których elementy są od siebie odległe o odstęp h. Każdy z podzbiorów jest sortowany przez wstawianie, a wartość h jest zmniejszana, aż osiągnie wartość równą 1. Wtedy sortujemy przez wstawianie, ale na uporządkowanym zbiorze, przez co algorytm jest bardzo efektywny. Złożoność obliczeniowa tego sortowania wynosi $O(n^{1,25})$. Sortowanie odbywa się w miejscu.

Teraz przedstawię wykresy otrzymane z przeprowadzonych testów:



Z powyższego wykresu wynika, że algorytm sortowania bąbelkowego jest najwolniejszy spośród badanych w tym ćwiczeniu. Posortowanie 40000 elementów losowych zajęło prawie 50 sekund Bubble sortowi, podczas gdy Counting Sort, Shell Sort oraz Heap Sort zrobiły to w czasie mniejszym od sekundy. Do dokładniejszej trzech szybszych sortów, postanowiłem stworzyć osobny wykres.



Z powyższego wykresu wynika iż algorytm sortowania stogowego jest najwolniejszy z pozostałych przedstawionych na wykresie dla danych generowanych losowo. Najszybszy okazał się Counting Sort. Tablica wypełniona 7 mln liczbami została posortowana przez algorytm Shella w czasie około 3.8 sekundy, podczas gdy Heap Sortowi zajęło to ponad 9 sekund. Counting Sort posortował tą tablice w mniej niż 0.25s.

2.

Wnioski:

Algorytm sortowania bąbelkowego okazał się najwolniejszy spośród wszystkich badanych w tym zadaniu. Kształt jego wykresu bardzo zbliżony jest do jego złożoności obliczeniowej, która wynosi $O(n^2)$. Algorytm posiada najgorszą złożoność spośród wszystkich badanych.

Na podstawie przeprowadzonych badań, okazało się, że algorytm sortowania stogowego jest drugim najwolniej sortującym. Jego złożoność obliczeniowa wynosi O(n log n), co powoduje, że wykonuje mniej operacji niż Bubble Sort, ale więcej niż dwa pozostałe.

Algorytm sortowania Shella jest drugim najlepszym pod względem szybkości sortowania. Jego złożoność obliczeniowa wynosi $O(n^{1,25})$. Shell Sort wykonuje mniej operacji niż Heap Sort i Bubble Sort.

Algorytm sortowania przez zliczanie okazał się najszybszym spośród pozostałych trzech. Złożoność obliczeniowa wynosi O(n+k), gdzie n to liczba elementów do posortowania, a k to maksymalna liczba w ciągu. Counting Sort wygrywa pod względem szybkości z pozostałymi, jednak wykorzystuje więcej pamięci, gdyż do jego wykonania , potrzebna jest dodatkowa tablica, w której zostaną przechowane liczniki liczb. Przy dużych zakresach liczb, zostanie wykorzystane bardzo dużo pamięci, przez co może się okazać, że bardziej efektywnym sortowaniem było by jedne z pozostałych. Bubble Sort, Shell Sort i Heap Sort są algorytmami sortowania w miejscu, przez co nie potrzebują dodatkowych zasobów pamięci.

Ćwiczenie 2

Celem zadania jest implementacja algorytmów sortowania: QS,HS,MS, testowanie oraz porównanie szybkości dla tablicy liczb całkowitych generowanych na różne sposoby (szczegóły w treści). Następnie przedstawienie danych na wykresach i sformułowanie wniosków.

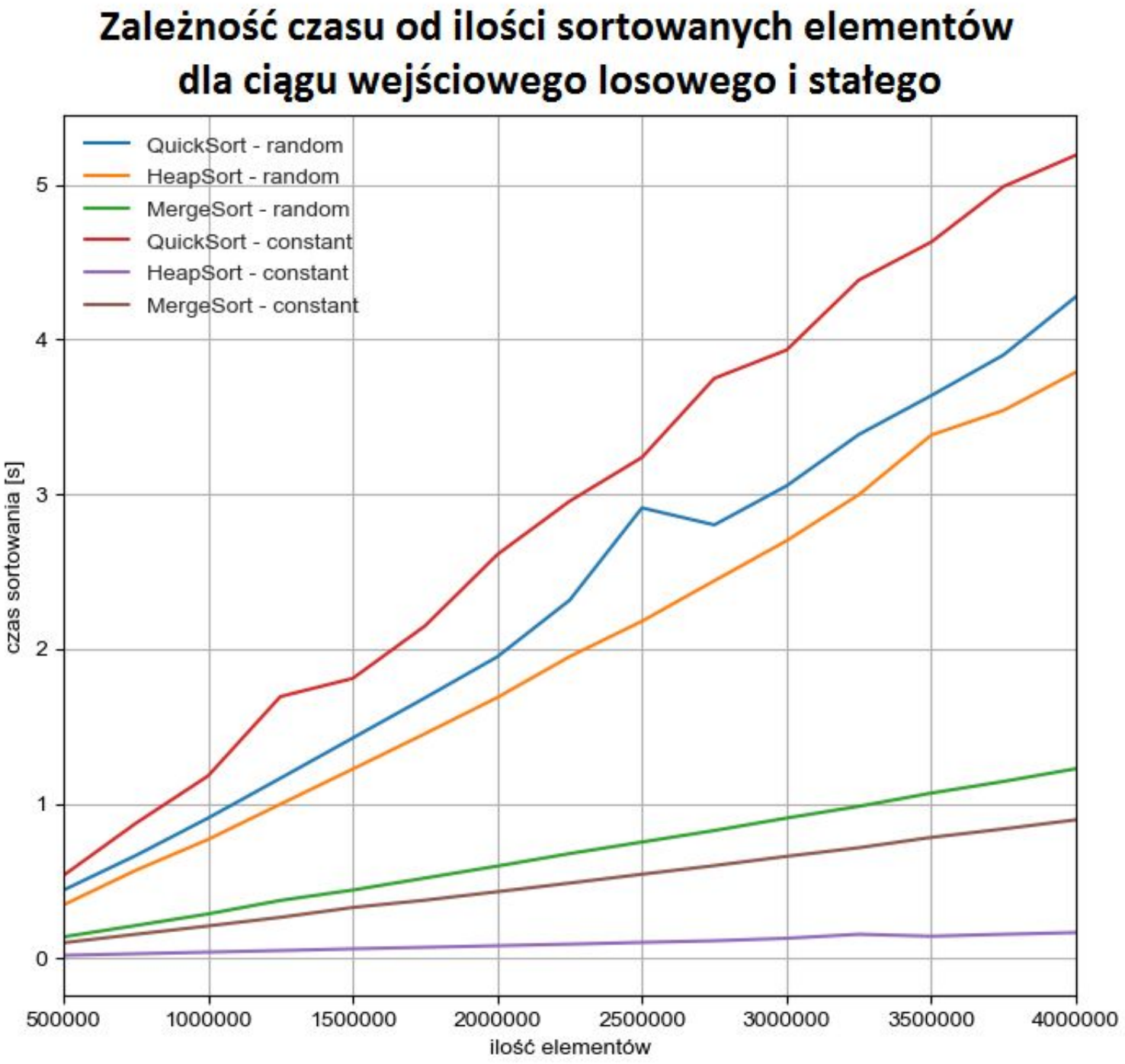
Zacznijmy od opisania poszczególnych algorytmów sortujących:

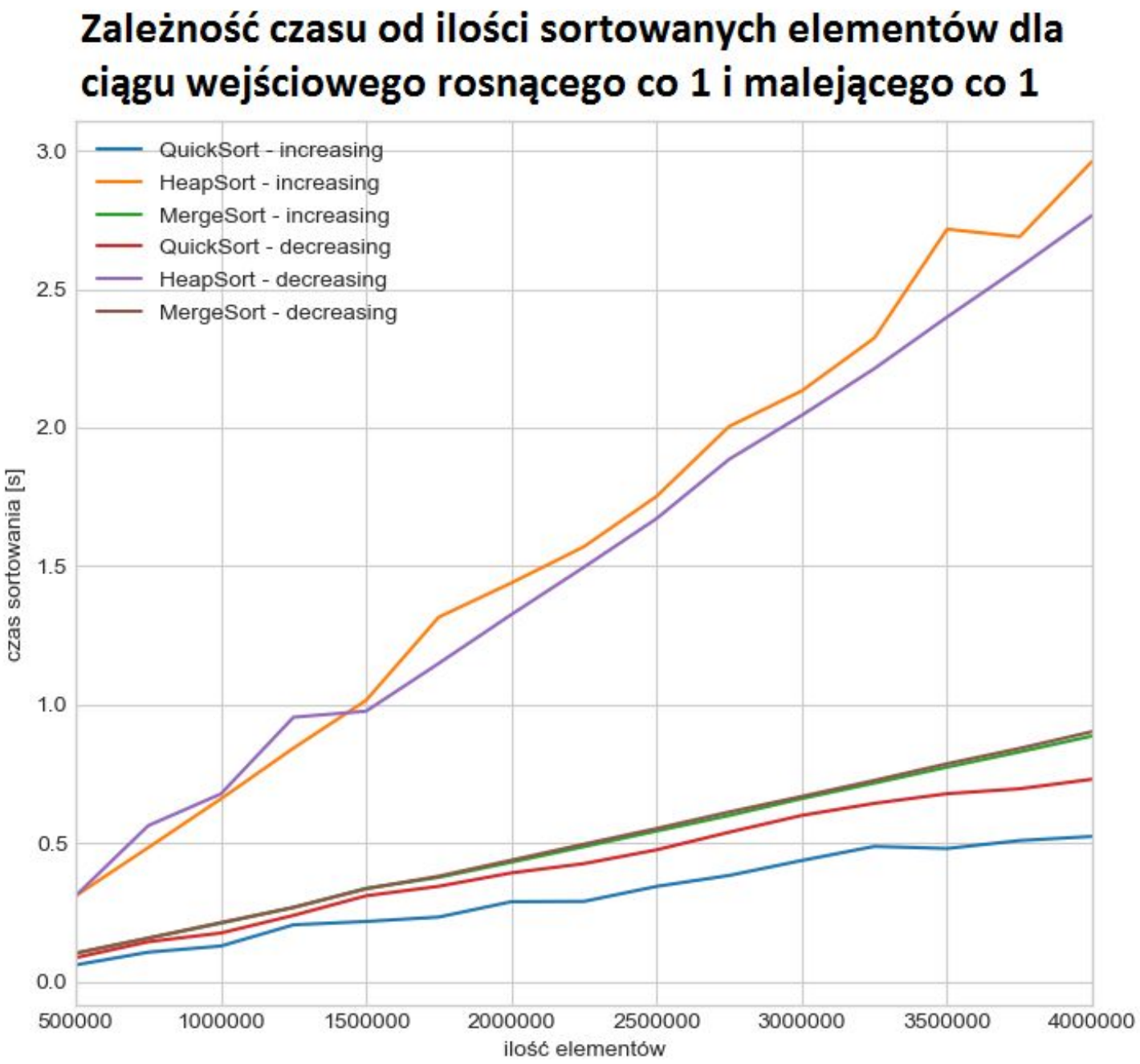
- Quick Sort (QS) , złożoność obliczeniowa algorytmu jest rzędu O(n log n), a w przypadku pesymistycznym $O(n^2)$. Działanie opiera się na zasadzie dziel i zwyciężaj. Algorytm działa rekurencyjnie. Najpierw dzielimy zbiór elementów na dwie części, tak aby wszystkie elementy znajdujące się na lewo od Pivota, który w tym przypadku jest elementem środkowym, były mniejsze lub równe od wszystkich z drugiej części. Następnie każdą z podzielonych części sortujemy rekurencyjnie tym samym sposobem. Na koniec łączymy posortowane części w posortowany ciąg. Sortowanie odbywa się w miejscu.

- Heap Sort został opisany w zadaniu pierwszym.

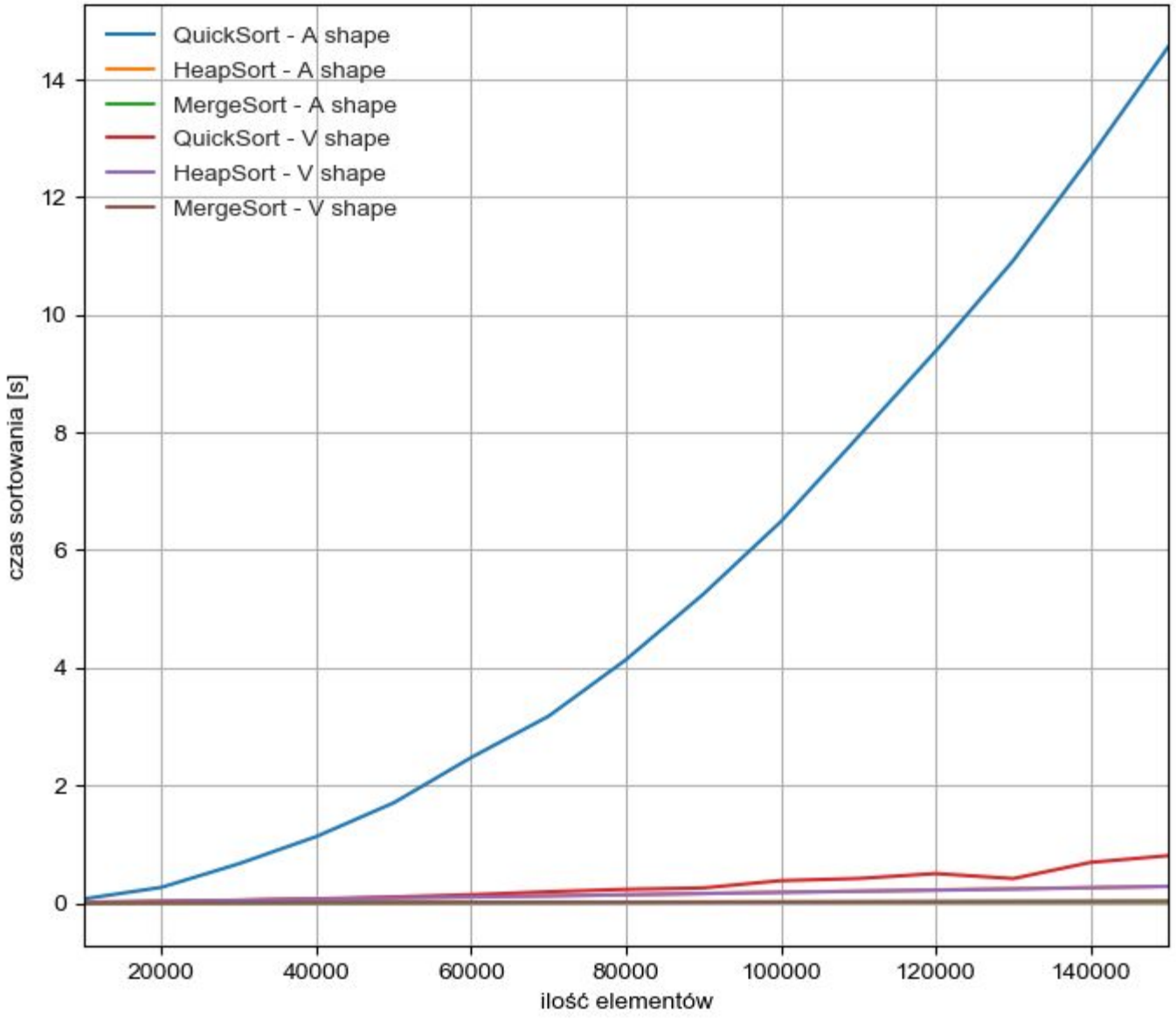
- Merge Sort (MS), złożoność obliczeniowa algorytmu wynosi w każdym przypadku O(n log n). Nie wykonuje się tu sortowanie w miejscu. Sortowanie przez scalanie posiada większą złożoność pamięciową, gdyż do jego wykonania potrzebna jest dodatkowa, pomocnicza struktura danych. Algorytm polega na dzieleniu ciągu elementów na mniejsze zbiory, aż do uzyskania n jednoelementowych. Następnie zbiory te są łączone w coraz większe, posortowane, dopóki nie uzyskamy n elementowego posortowanego ciągu.

Teraz przedstawię wykresy otrzymane z przeprowadzonych testów:

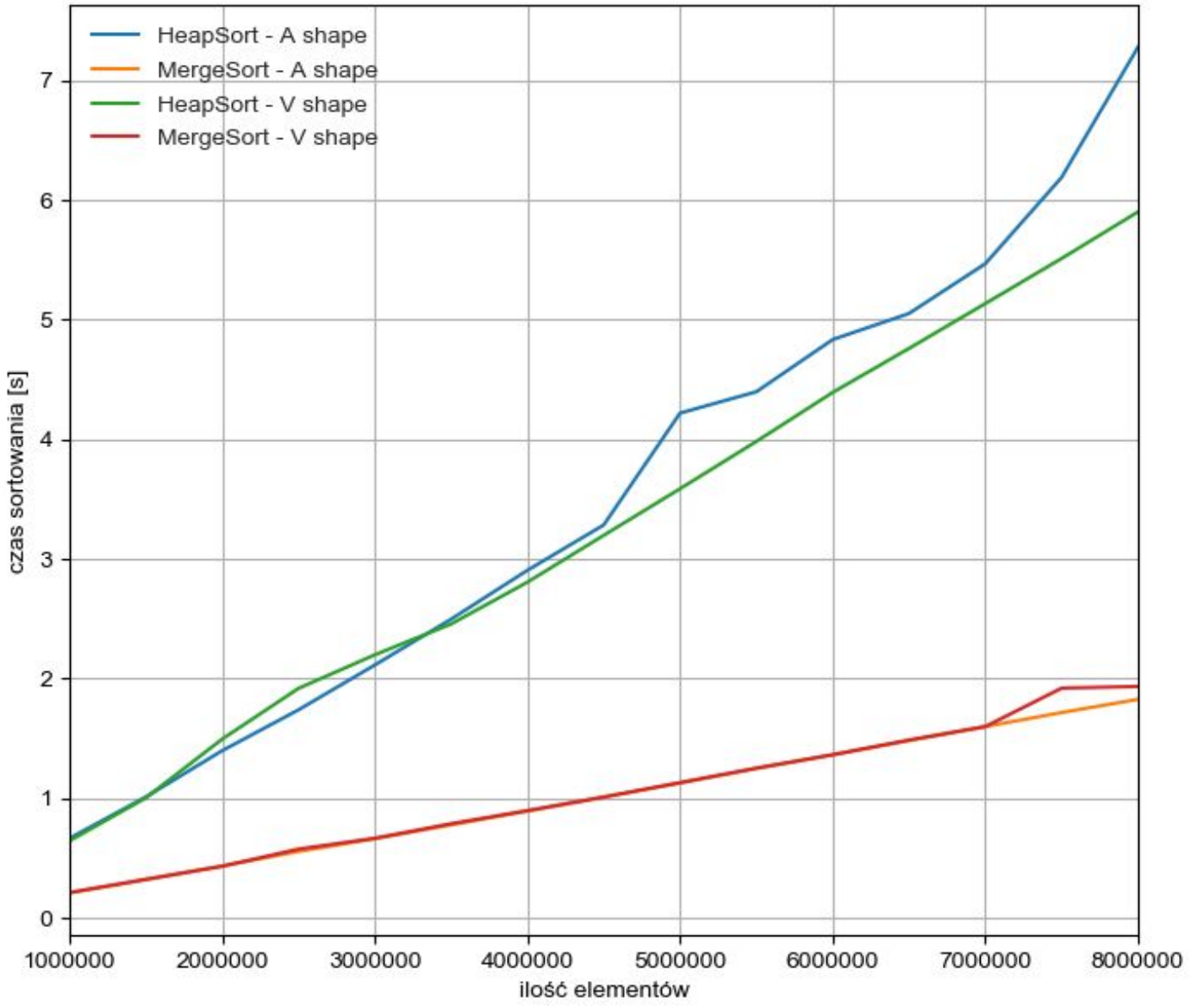

Zależność czasu od ilości sortowanych elementów
dla ciągu wejściowego losowego i stałego
QuickSort - random
HeapSort - random
MergeSort - random
QuickSort - constant
HeapSort - constant
MergeSort - constant
czas sortowania [s]
0
1
2
3
4
5
500000
1000000
1500000
2000000
2500000
3000000
3500000
4000000
ilość elementów


Zależność czasu od ilości sortowanych elementów dla ciągu wejściowego rosnącego co 1 i malejącego co 1
QuickSort - increasing
HeapSort - increasing
MergeSort - increasing
QuickSort - decreasing
HeapSort - decreasing
MergeSort - decreasing
czas sortowania [s]
3.0
2.5
2.0
1.5
1.0
0.5
0.0
500000
1000000
1500000
2000000
2500000
3000000
3500000
4000000
ilość elementów

Zależność czasu od ilości sortowanych elementów dla ciągu wejściowego rosnąco-malejącego (A) i malejąco-rosnącego (V)



Zależność czasu od ilości sortowanych elementów dla ciągu wejściowego rosnąco-malejącego (A) i malejąco-rosnącego (V)



2. Wnioski

Z pierwszego wykresu wynika, że Quick Sort zarówno dla danych losowych jak i stałych jest wolniejszy od pozostałych umieszczonych na wykresie. Algorytm sortowania przez podział okazał się szybszy od Quick Sorta pomimo takiej samej złożoności obliczeniowej O(nlogn). Liczba porównań Merge Sorta bardziej zbliżona jest do wartości $n\log_2 n$, gdyż prawdopodobieństwo wyboru niekorzystnych elementów osiowych jest bardzo małe, dlatego podczas podziałów na dwie części, występuje mniej więcej równa ilość elementów. Najszybszy okazał się Heap Sort dla stałych danych. Jego złożoność obliczeniowa również wynosi O(nlogn). Stało się tak, gdyż dla ciągów stałych w wyniku przesiewania nie są przenoszone żadne elementy w ciągu i odbudowa kopca jest bardzo prosta.

Analizując wykres z rosnącymi i malejącymi ciągami, widzimy że Quick Sort błyskawicznie poradził sobie z ciągiem rosnącym. Wynika to z faktu, iż ciąg rosnący o 1 stanowi najlepszy przypadek dla tego algorytmu, gdyż nie następują żadne zmiany w trakcie działania algorytmu. W przypadku sortowania ciągu malejącego co 1, QS ze środkowym Pivotem, już w pierwszym kroku otrzymuje prawie posortowany ciąg. Sortowanie ciągu malejącego o 1 zajmuje nieznacznie większy okres czasu. Najdłużej zajmuje posortowanie tego typu danych Heap Sortowi, gdyż jest to bardzo niekorzystny przypadek dla tego algorytmu, ze względu na budowę kopca.

Dla danych wejściowych A i V kształtnych stworzyłem dwa osobne wykresy. Na pierwszym uwzględnione zostały wszystkie trzy algorytmy. Widzimy od razu, że Quick Sort nie radzi sobie z danymi rosnąco-malejącymi oraz malejąco-rosnącymi. Merge Sort i Heap Sort o wiele szybciej sortują dane tych typów. Dodatkowo widać na wykresie, że Quick Sort dla elementów rosnąco-malejących zbliżony jest do paraboli. Posortowanie tych elementów jest jego najgorszym przypadkiem, przy którym złożoność osiąga $O(n^2)$. Spowodowane jest to faktem, iż za każdym razem wybierany element osiowy jest największym (w przypadku A kształtnego) lub najmniejszym (w

przypadku V kształtnego) elementem całego ciągu. W wyniku czego, w każdym kroku segment n obiektów zostaje podzielony na dwa podzbiory: jeden o n-1 elementach i drugi jednoelementowy. W każdym kroku zostaje wykonana jedna zamiana, w wyniku czego ilość wszystkich podziałów zbliżona jest do n zamiast $\log_2 n$.