Algorytmy Sortowania

Projekt wykonały:

Paulina Pogorzelska

Zuzanna Ławniczak

Grupa dziekańska: 3a

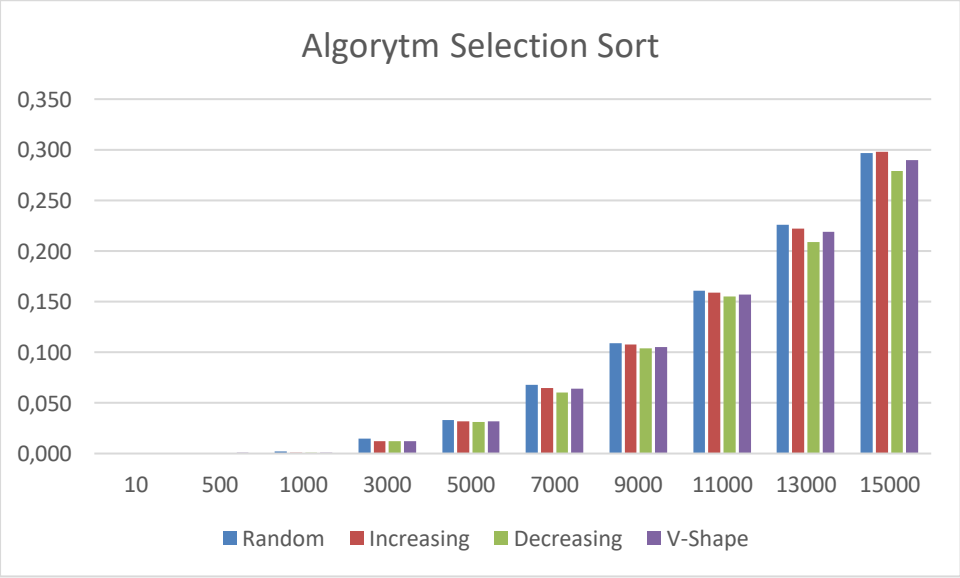
1.Opis Projektu

Przygotowane zostały generatory instancji wartości: losowych, rosnących, malejących, w postaci rozkładu v-kształtnego oraz implementacje algorytmów: Selection sort(roz.2) ,Insertion sort(roz.3), Quick sort w wersji z ostatnim elementem rozdzielającym(roz.4.1), Quick sort w wersji z losowym elementem rozdzielającym(roz.4.2), Heap sort(roz.5).

Następnie został zmierzony czas(wartości zawarte w tabelach) działania każdego z algorytmów dla każdego zbioru danych dla różnych uprzednio wybranych wielkości instancji: 10,500,1000,3000,5000,7000,9000,11000,13000,15000

2.Algorytm Selection Sort

Wielkość instancji	**Random**	**Increasing**	**Decreasing**	**V-Shape**
10	0,000	0,000	0,000	0,000
500	0,000	0,000	0,000	0,001
1000	0,002	0,001	0,001	0,001
3000	0,015	0,012	0,012	0,012
5000	0,033	0,032	0,031	0,032
7000	0,068	0,065	0,060	0,064
9000	0,109	0,108	0,104	0,105
11000	0,161	0,159	0,155	0,157
13000	0,226	0,222	0,209	0,219
15000	0,297	0,298	0,279	0,290

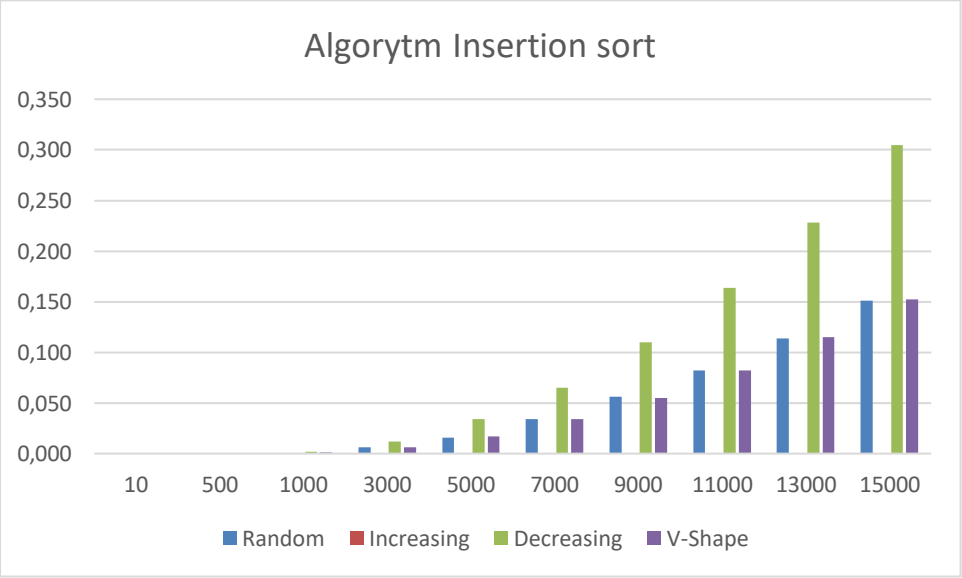


Złożoność obliczeniowa w najgorszym przypadku: $O(n^2)$

Jak można zauważyć w tym algorytmie rozkład danych nie ma większego znaczenia. Wynika to z tego, że algorytm Selection Sort dla danej długości ciągu zawsze (niezależnie od zawartości ciągu) wykonuje taką samą liczbę porównań - nawet dla ciągu wejściowego, który już jest posortowany. Dlatego też, złożoność obliczeniowa tego algorytmu jest taka sama w najlepszym jak i najgorszym przypadku.

3. Insertion Sort

Wielkość instacjii	**Random**	**Increasing**	**Decreasing**	**V-Shape**
10	0,000	0,000	0,000	0,000
500	0,000	0,000	0,000	0,000
1000	0,000	0,000	0,002	0,001
3000	0,006	0,000	0,012	0,006
5000	0,016	0,000	0,034	0,017
7000	0,034	0,000	0,065	0,034
9000	0,056	0,000	0,110	0,055
11000	0,082	0,000	0,164	0,082
13000	0,114	0,000	0,228	0,115
15000	0,151	0,000	0,305	0,152

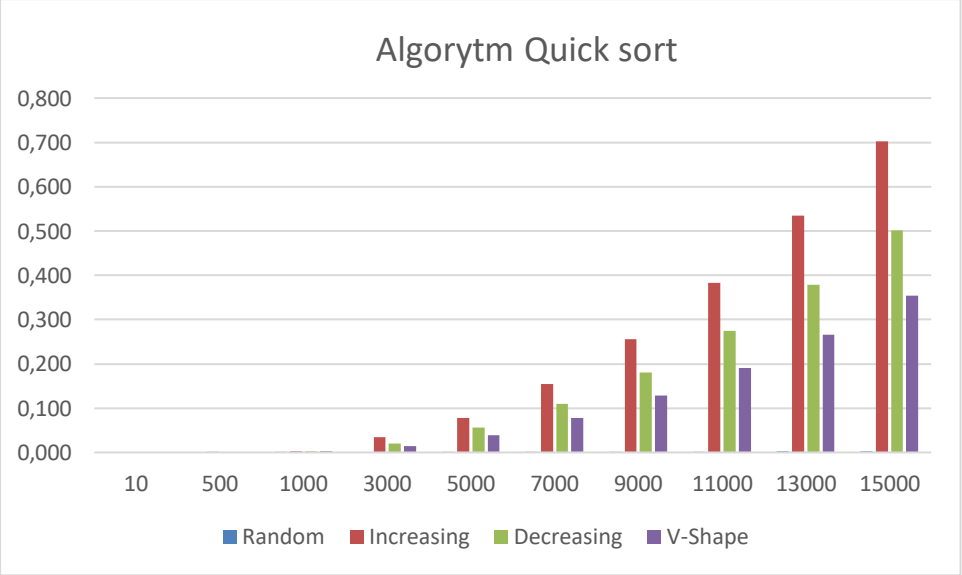


Złożoność obliczeniowa w najgorszym przypadku: $O(n^2)$

Jeśli dane wejściowe już są posortowane, algorytm działa bardzo szybko i wymaga tylko n-1 porównań. Jest on więc bardziej optymalny niż selection Sort, gdyż dostosowuje ilość pracy do stopnia posortowania danych. W przypadku losowych oraz v-kształtnych danych przy założeniu, że dane są liczbami naturalnymi od 1 do n i każda permutacja jest jednakowo prawdopodobna to algorytm ten jest 2 razy szybszy niż w przypadku najgorszym, ale wciąż ma kwadratową złożoność obliczeniową. Z tego wynika więc, że najlepszym przypadkiem będzie rozkład danych rosnący, środkowym będzie rozkład losowy i v-kształtny a najgorszym rozkład malejący.

4.1 Quick Sort (Ostatnia wartość w tablicy jest elementem rozdzielającym)

Wielkość instancji	**Random**	**Increasing**	**Decreasing**	**V-Shape**
10	0,000	0,000	0,000	0,000
500	0,000	0,001	0,000	0,000
1000	0,001	0,003	0,003	0,002
3000	0,000	0,034	0,020	0,014
5000	0,001	0,078	0,056	0,039
7000	0,001	0,155	0,110	0,078
9000	0,001	0,256	0,180	0,128
11000	0,001	0,383	0,275	0,191
13000	0,002	0,534	0,378	0,266
15000	0,002	0,703	0,502	0,354

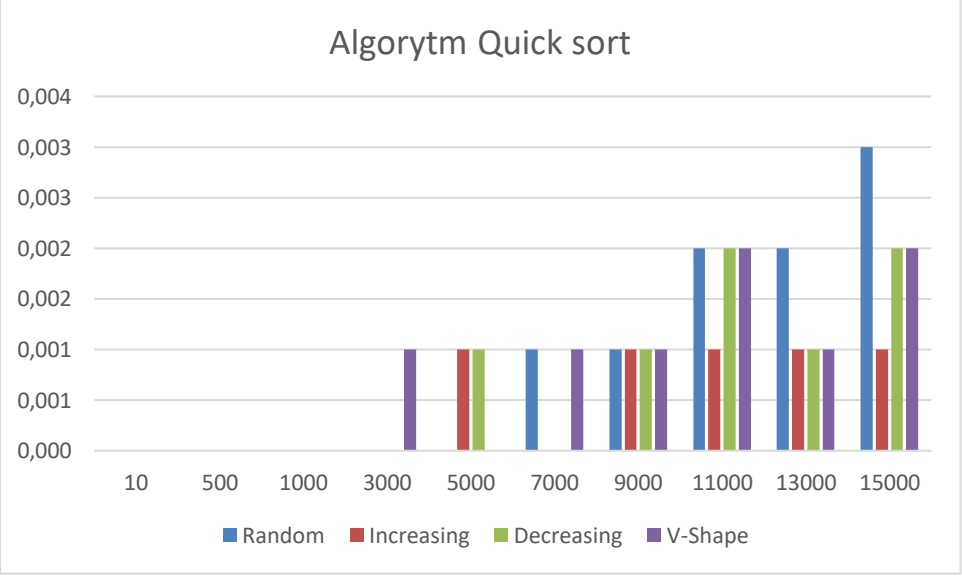


Złożoność obliczeniowa w najgorszym przypadku: $O(n^2)$

Operacja sortowania w Quick Sort odbywa się w czasie liniowym. O złożoności algorytmu decyduje więc to, ile nastąpi rekurencyjnych wywołań funkcji. Najlepszy przypadek szybkiego sortowania występuje, gdy podziały są możliwie równe: ich rozmiary są równe albo różnią się jednym elementem. Wówczas głębokość drzewa wywołań zależy logarytmicznie od liczby danych wejściowych, czyli optymistyczna złożoność czasowa algorytmu jest rzędu *O(n logn)*. W przypadku średnim również zachodzi taki rząd złożoności. Najoptymalniejszy będzie rozkład losowy. Po wykresie można zobaczyć, że dane v-kształtne są bardziej optymalne niż dane rosnące czy malejące(czyli wtedy gdy piwot za każdym razem znajdzie się na brzegu tabeli). Najgorszym przypadkiem będzie rozkład rosnący.

4.2 Quick Sort(Losowa wartość z tablicy jest elementem rozdzielającym)

Wielkość instancji	**Random**	**Increasing**	**Decreasing**	**V-Shape**
10	0,000	0,000	0,000	0,000
500	0,000	0,000	0,000	0,000
1000	0,000	0,000	0,000	0,000
3000	0,000	0,000	0,000	0,001
5000	0,000	0,001	0,001	0,000
7000	0,001	0,000	0,000	0,001
9000	0,001	0,001	0,001	0,001
11000	0,002	0,001	0,002	0,002
13000	0,002	0,001	0,001	0,001
15000	0,003	0,001	0,002	0,002

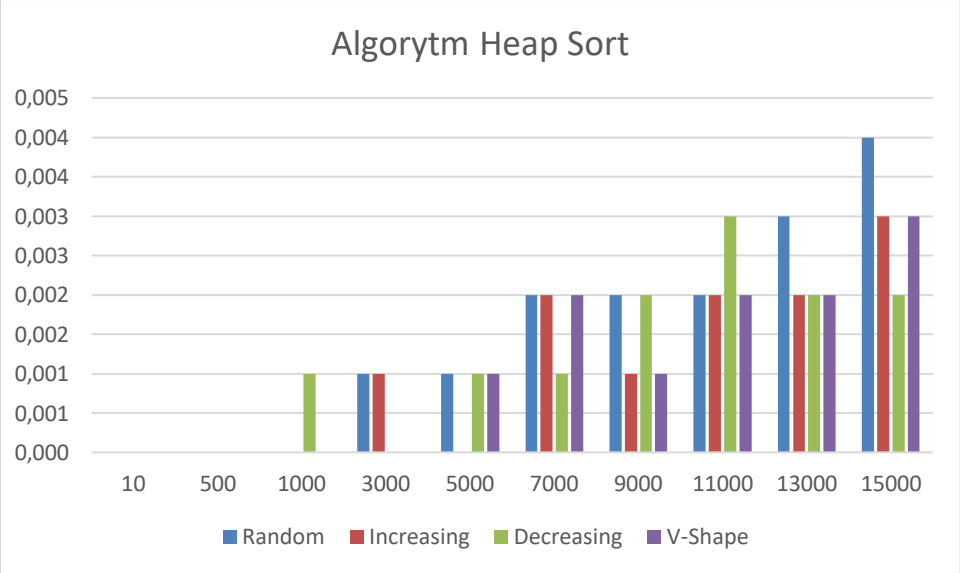


Złożoność obliczeniowa w najgorszym przypadku: $O(n^2)$

W tym przypadku, najgorsze działanie algorytmu wynika z generatora liczb losowych. Zadziała więc on źle gdy generator utworzy pechową permutacje. Dlatego złożoność obliczeniowa tego algorytmu w najgorszym przypadku jest taka sama jak w poprzednim algorytmie (3.1) gdzie działo się to gdy dane były rosnące. Tutaj typ danych wejściowych nie powoduje pesymistycznego działania algorytmu. Najlepszym dla tego algorytmu rozkładem danych będzie rozkład rosnący. Otrzymane wyniki w tabeli ukazują, że ten algorytm jest najszybszym algorytmem sortującym.

5. Heap Sort

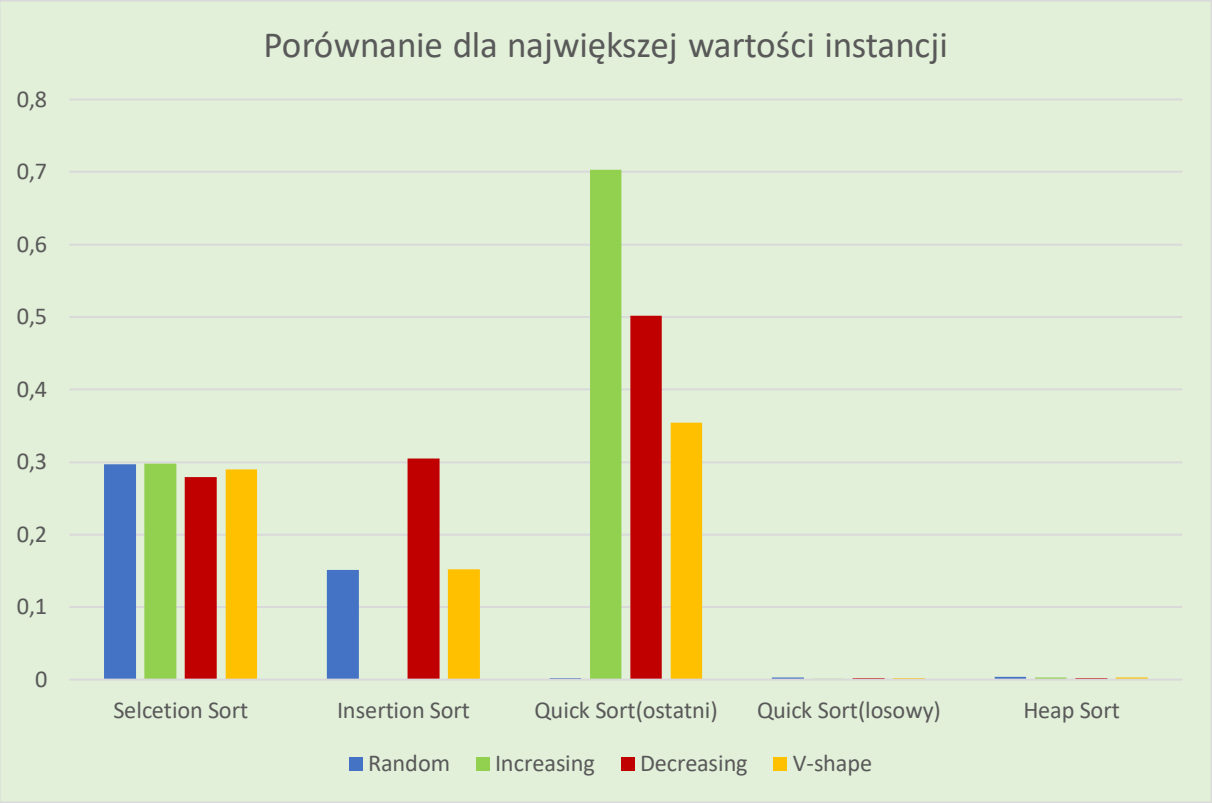
Wielkość instancji	**Random**	**Increasing**	**Decreasing**	**V-Shape**
10	0,000	0,000	0,000	0,000
500	0,000	0,000	0,000	0,000
1000	0,000	0,000	0,001	0,000
3000	0,001	0,001	0,000	0,000
5000	0,001	0,000	0,001	0,001
7000	0,002	0,002	0,001	0,002
9000	0,002	0,001	0,002	0,001
11000	0,002	0,002	0,003	0,002
13000	0,003	0,002	0,002	0,002
15000	0,004	0,003	0,002	0,003



Złożoność obliczeniowa w najgorszym przypadku: $O(n\log_n)$

Czasy podane w tabeli niewiele różnią się od siebie. Świadczy to o tym, że ten algorytm jest mało czuły na postać danych wejściowych. Złożoność obliczeniowa dla każdego przypadku jest tej samej klasy $O(n\log_n)$.

6.Wnioski



Wybór algorytmu sortującego musi być dopasowany w zależności od przewidywań naszych danych, ich ilości i układu. Algorytmy **Selection Sort** i **Insertion Sort** mogą być dobrą opcją w przypadku **małych ilości danych** lub dla Insertion Sort gdy będą one rosnące- mimo że są one dużo mniej wydajne niż pozostałe opcje przy małych ilościach danych różnica nie będzie tak wielka a ich duża przewagą pozostaje **łatwość implementacji**. Najoptymalniejszym wyborem algorytmu będą algorytmy **Heap Sort** i **Quick Sort** ze zwróceniem uwagi na implementacje w wersji z losowym elementem rozdzielającym w przeciwnym przypadku Quick Sort może okazać się najgorszym wyborem. **Heap Sort** mimo nieznaczenie gorszych czasów niż Quick Sort w wariancie z losowym elementem rozdzielającym posiada tę **przewagę** że dla każdego z przypadków będzie posiadał tę samą złożoność obliczeniową natomiast Quick Sort w wariancie z losowym elementem rozdzielającym, jeśli dane okażą się posortowane a wylosowany element rozdzielający będzie ostatnim może się okazać bardzo nieoptymalnym wyborem.