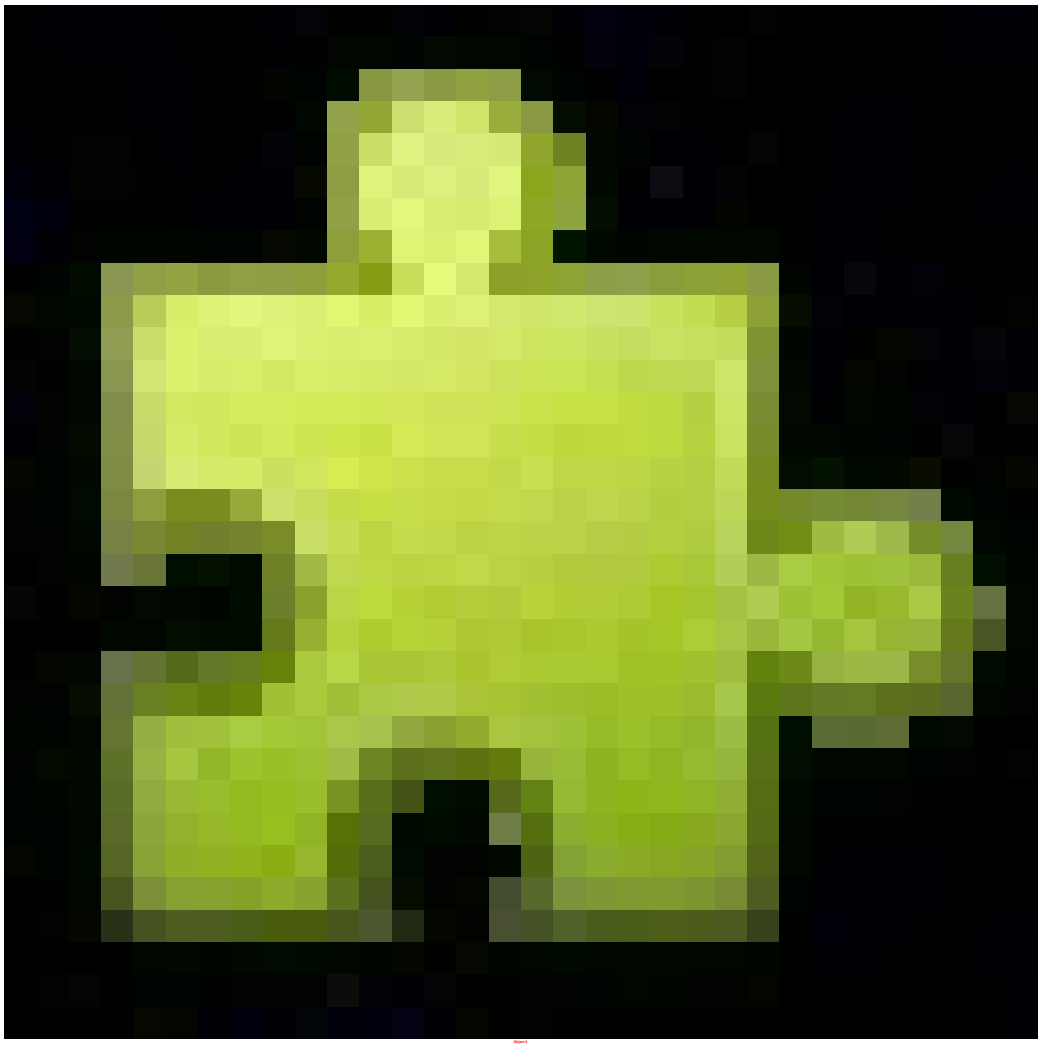
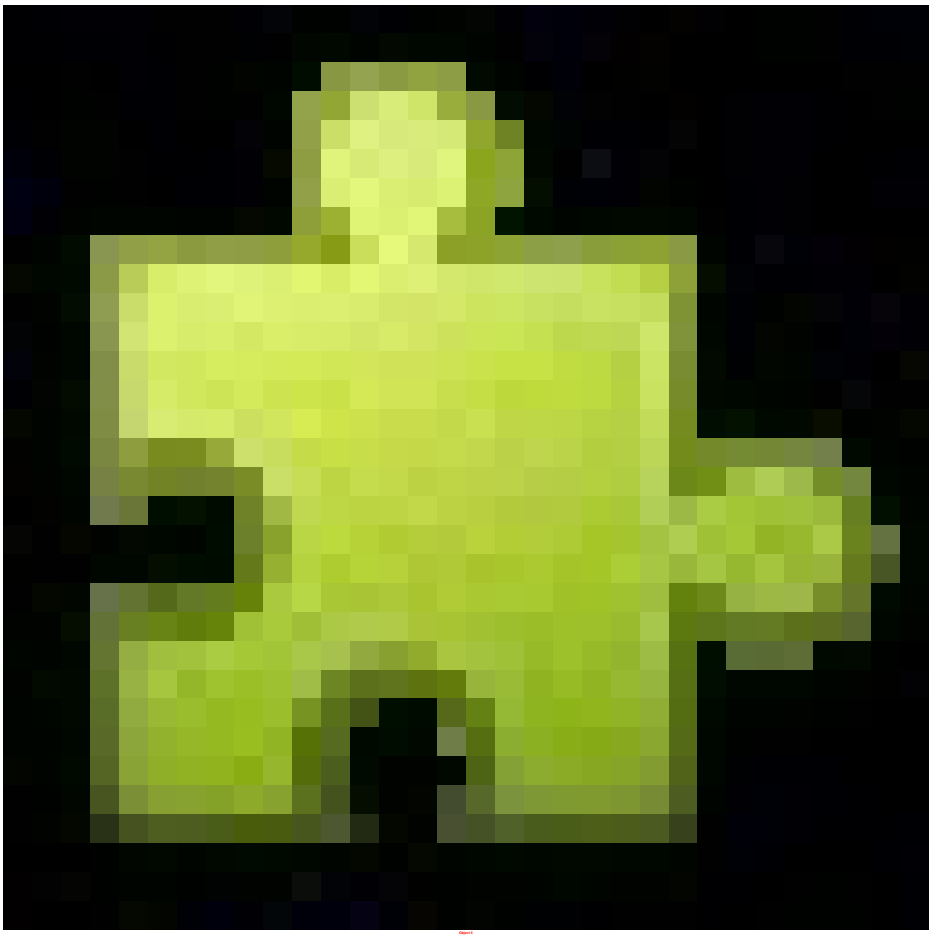
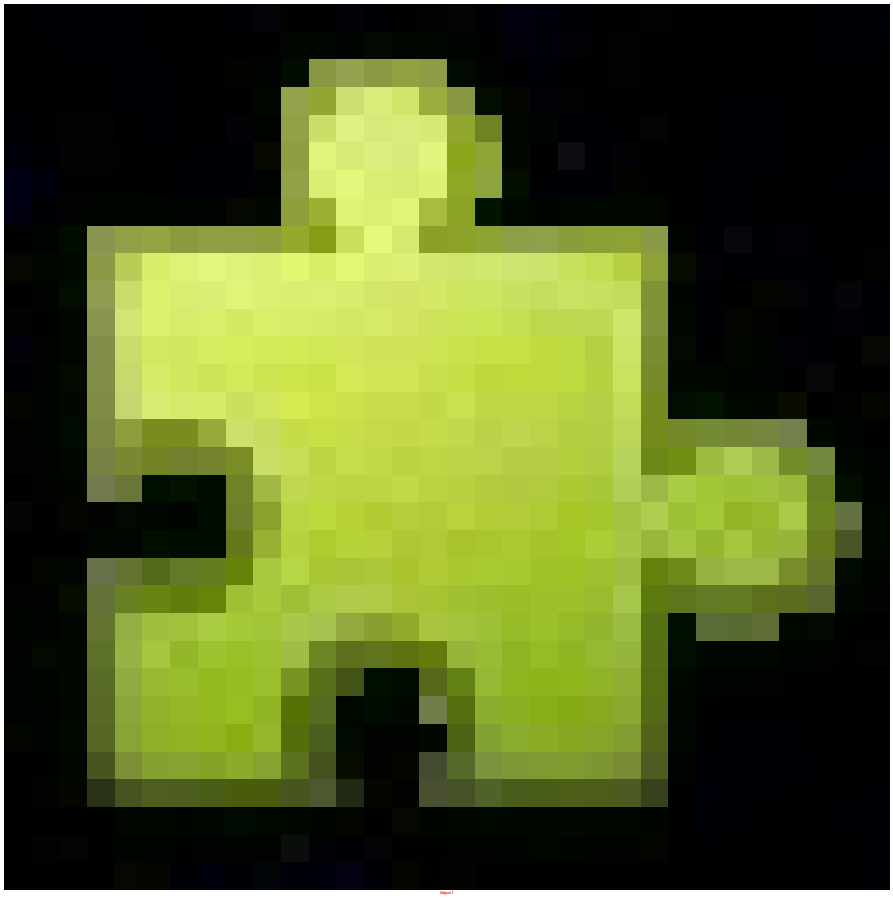
Hubert Cwajda, 141203
Michał Wiśniewski, 141335

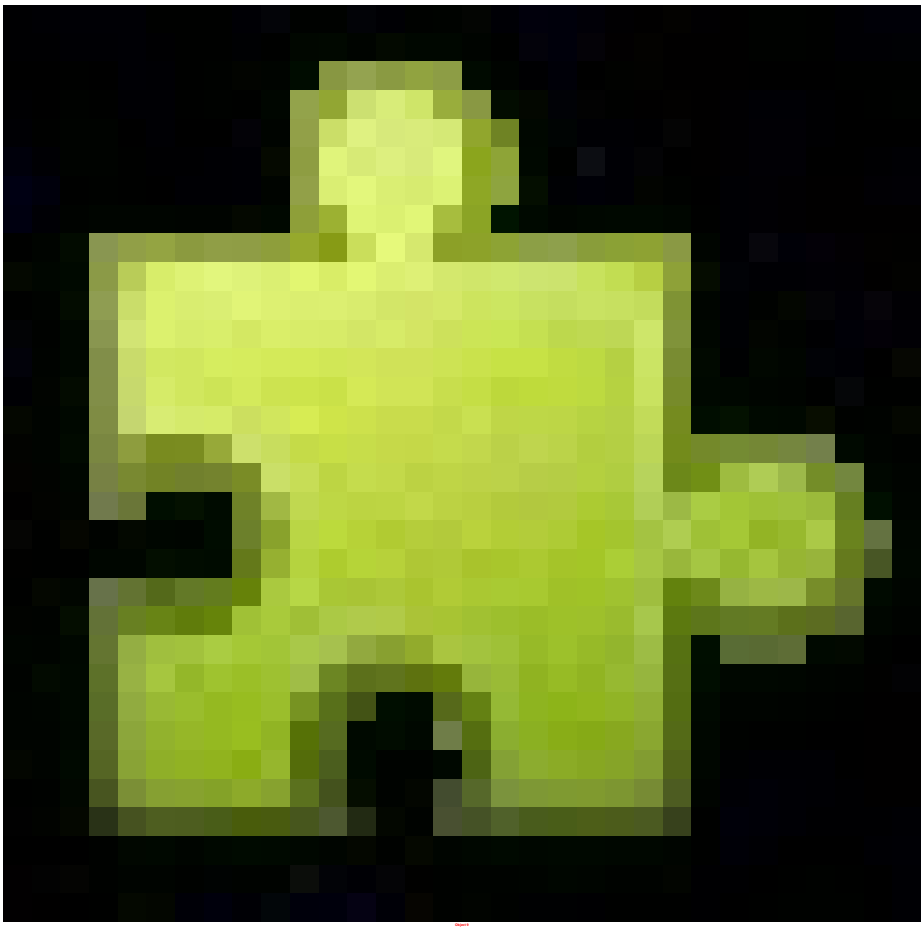
Algorytmy i Struktury Danych - Sprawozdanie 1: Sortowanie

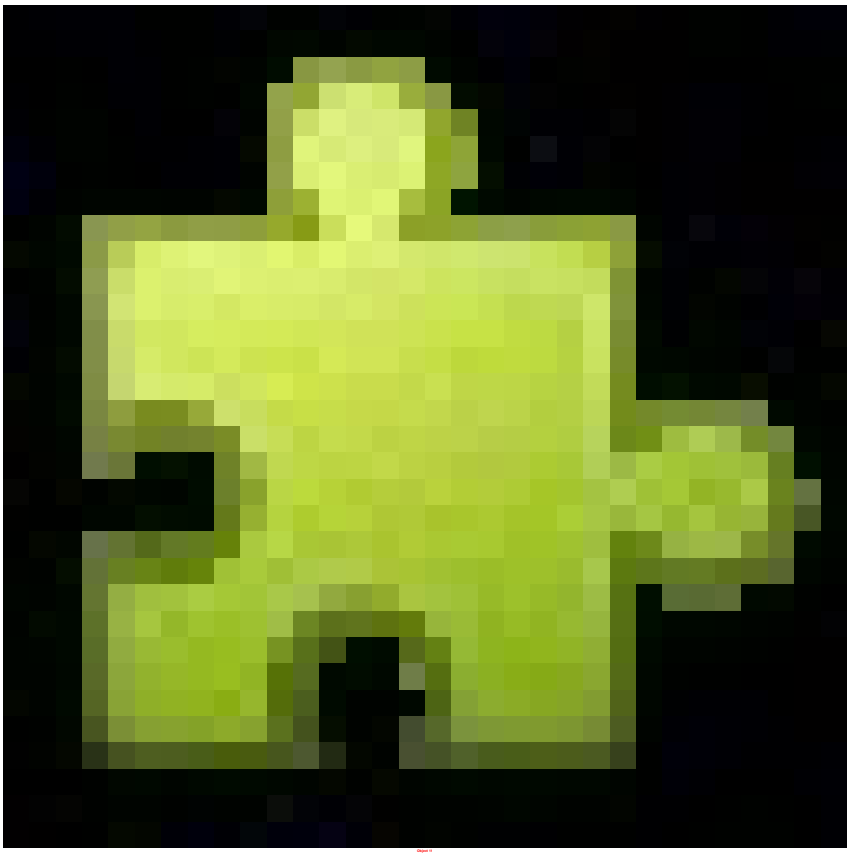
I. Porównanie szybkości działania 4 metod sortowania: Insertion Sort, Selection Sort, Heap Sort, Merge Sort dla tablicy liczb całkowitych generowanych w postaci: losowej, rosnącej, malejącej, stałej, v-kształtnej.











Wnioski:

- Algorytmy Selection Sort oraz Insertion Sort cechują się kwadratową złożonością obliczeniową w przeciętnym i najgorszym przypadku. Z tego powodu są z reguły znacznie wolniejsze w sortowaniu losowych ciągów od algorytmów Heap i Merge Sort, których ta złożoność obliczeniowa wynosi $n \log n$. W związku z nią, te drugie są zdecydowanie najlepszym wyborem jeżeli zależy nam głównie na czasie wykonania sortowania. To samo zachodzi dla ciągów uporządkowanych malejąco oraz V-kształtnych.

- Złożoność obliczeniowa algorytmu Selection Sort wynosi zawsze O(n^2). Z tego powodu dla danych każdego rodzaju osiąga najgorsze wyniki. Dodatkowo jest algorytmem niestabilnym, tj. w wyniku sortowania może zmienić kolejność elementów o tym samym kluczu. W przypadku algorytmów stabilnych mamy pewność co do jego zachowania. Zaletą sortowania przez wybieranie jest natomiast jego złożoność pamięciowa *O(1)*, ponieważ algorytm ten nie potrzebuje pamięci pomocniczej.

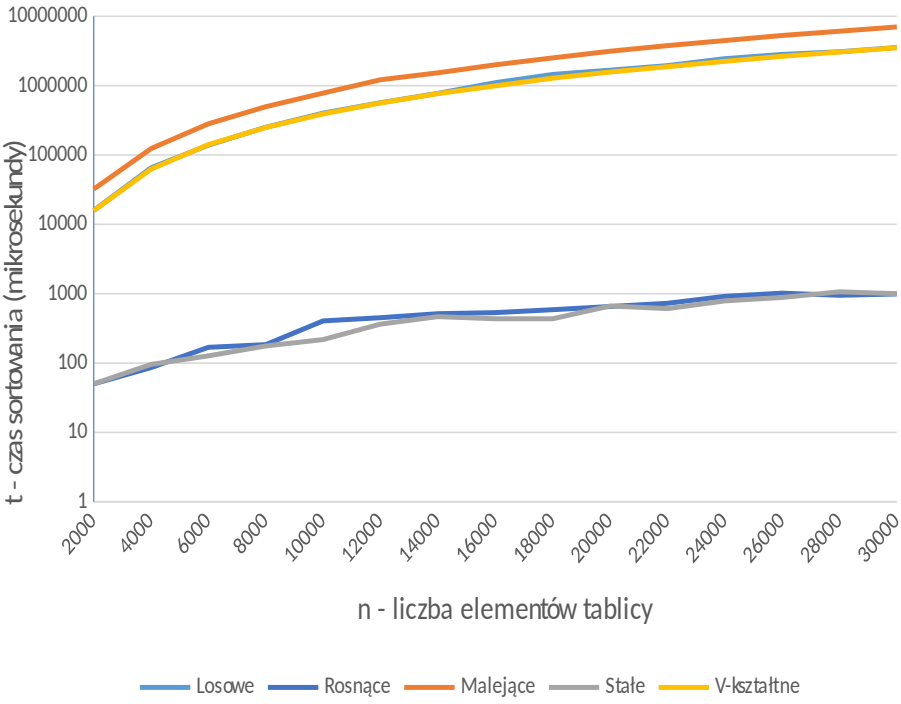
- Alogrytm Insertion Sort w optymistycznym przypadku ma złożoność czasową *n*, która zachodzi przy nie wykonywaniu pętli *while (i > 0) i (A[i] > key)*, a więc dla danych wstępnie posortowanych (właściwie), czyli ciągów uporządkowanych rosnąco, a także stałych. Dlatego przy sortowaniu tablic liczb tej postaci sortowanie przez scalanie jest wyraźnie najszybszą metodą spośród czterech. Sortowanie przez wstawianie cechuje się również stabilnością i niską złożonością pamięciową *O(1)*, gdyż tak jak Selection Sort, potrzebuje jedynie stałej ilości pamięci komputera, w której wykonywane są wszystkie potrzebne obliczenia.

- Algorytm Merge Sort ma złożoność obliczeniową $n \log n$ dla wszystkich przypadków, dlatego osiąga takie same wyniki dla każdej postaci danych wejściowych. Jest również algorytmem stabilnym, jego wadą jest natomiast złożoność pamięciowa, ponieważ nie dochodzi do sortowania w miejscu i wymagane jest $O(n)$ pamięci pomocniczej.

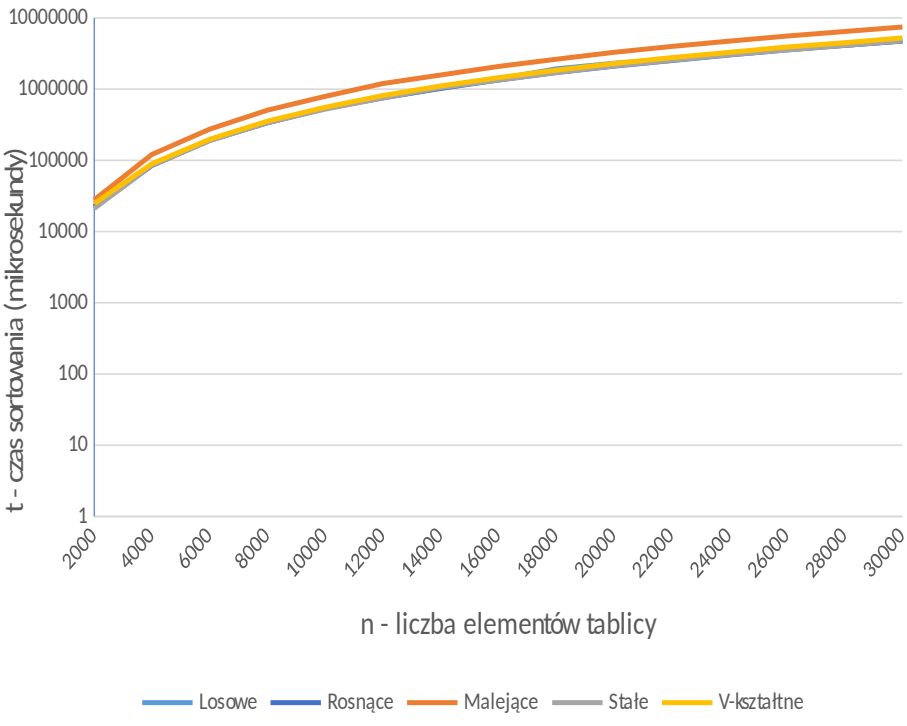
- Algorytm Heapsort cechuje się tą samą złożonością obliczeniową co Merge Sort, więc zwykle osiąga niemal identyczne wyniki (minimalne różnice mogą wynikać z implementacji). Jedynym wyjątkiem, który widać na wykresie, jest stały ciąg liczb, w przypadku którym złożoność obliczeniowa algorytmu wynosi jedynie *n*, ponieważ maksimum będące równe każdej innej liczbie nie wywołuje kolejny raz funkcji *heapify* i czas pochłania jedynie zbudowanie kopca. Przez ten fakt Heapsort nie jest jednak algorytmem stabilnym, w przeciwieństwie do sortowania przez scalanie. Generalną zaletą sortowania stogowego jest natomiast złożoność pamięciowa, jaką przyjmuje się za *O(1)*, ponieważ jest algorytmem sortowania w miejscu i sam w sobie nie wymaga przechowywania danych w dodatkowych strukturach.

II. Porównanie efektywności działania algorytmów sortowania w zależności od typów danych wejściowych (tablice liczb całkowitych w postaciach: losowej, rosnącej, malejącej, stałej, v-kształtnej).

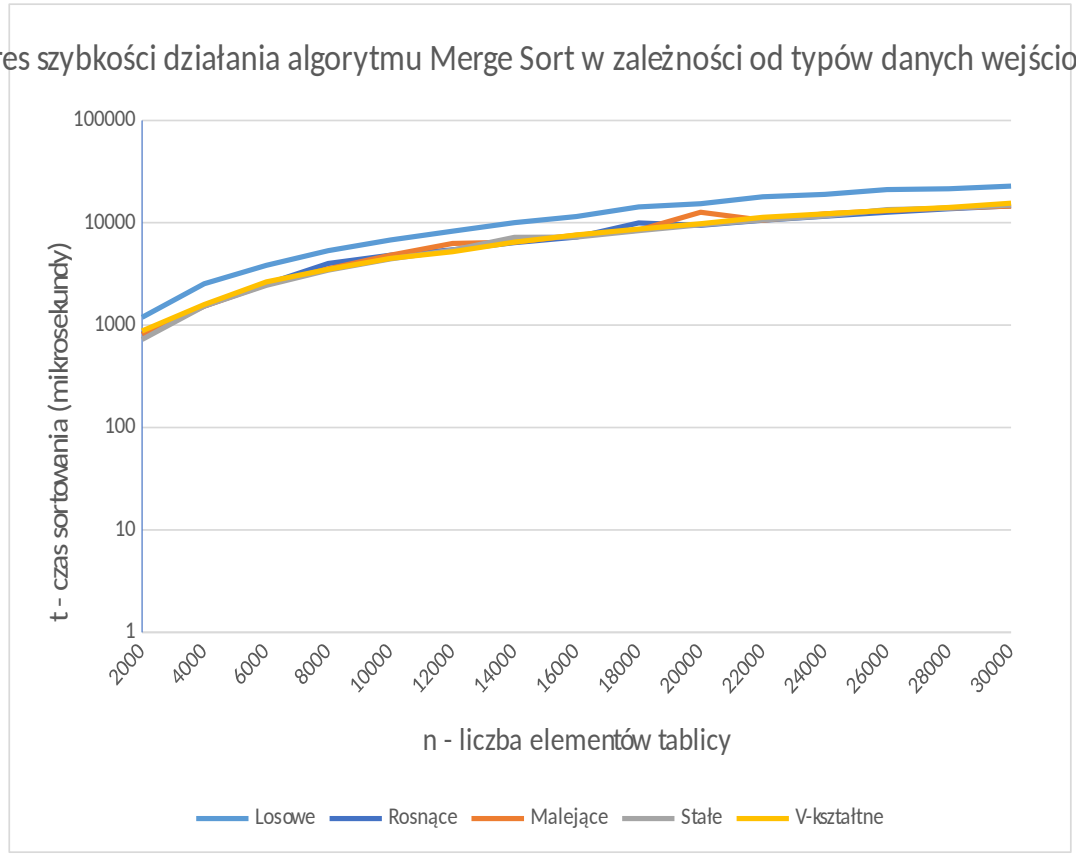
Wykres szybkości działania algorytmu Insertion Sort w zależności od typów danych wejściowych



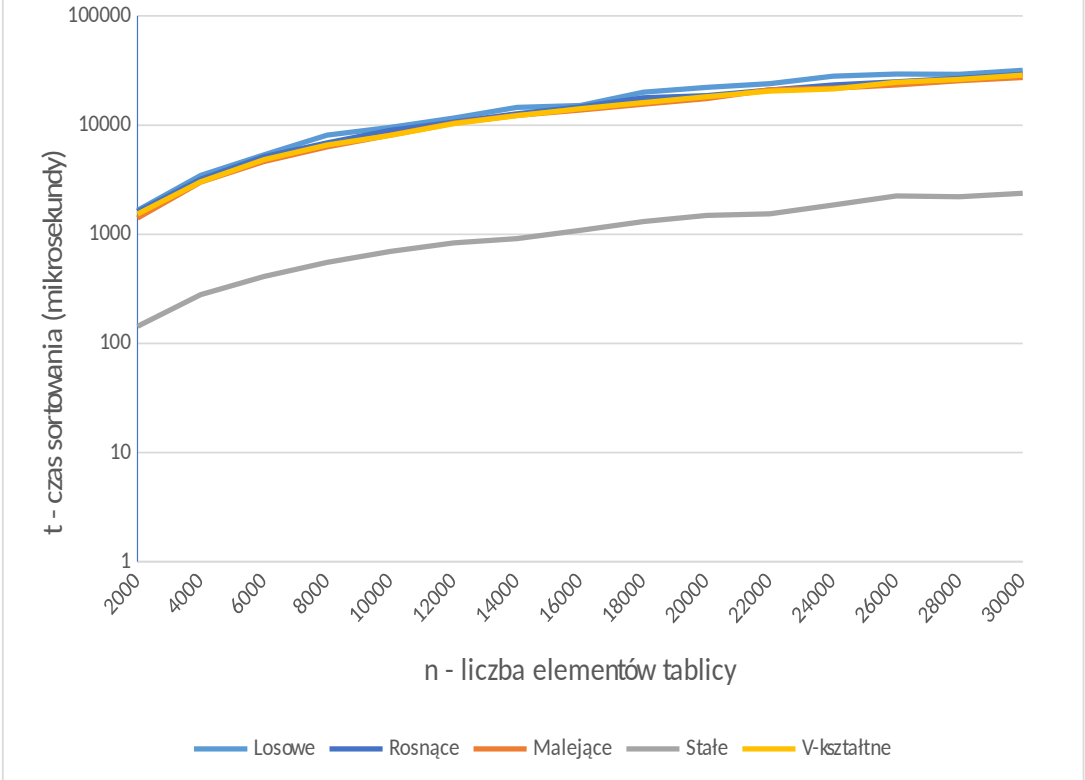
Wykres szybkości działania algorytmu Selection Sort w zależności od typów danych wejściowych



Wykres szybkości działania algorytmu Merge Sort w zależności od typów danych wejściowych



Wykres szybkości działania algorytmu Heap Sort w zależności od typów danych wejściowych



Wnioski:

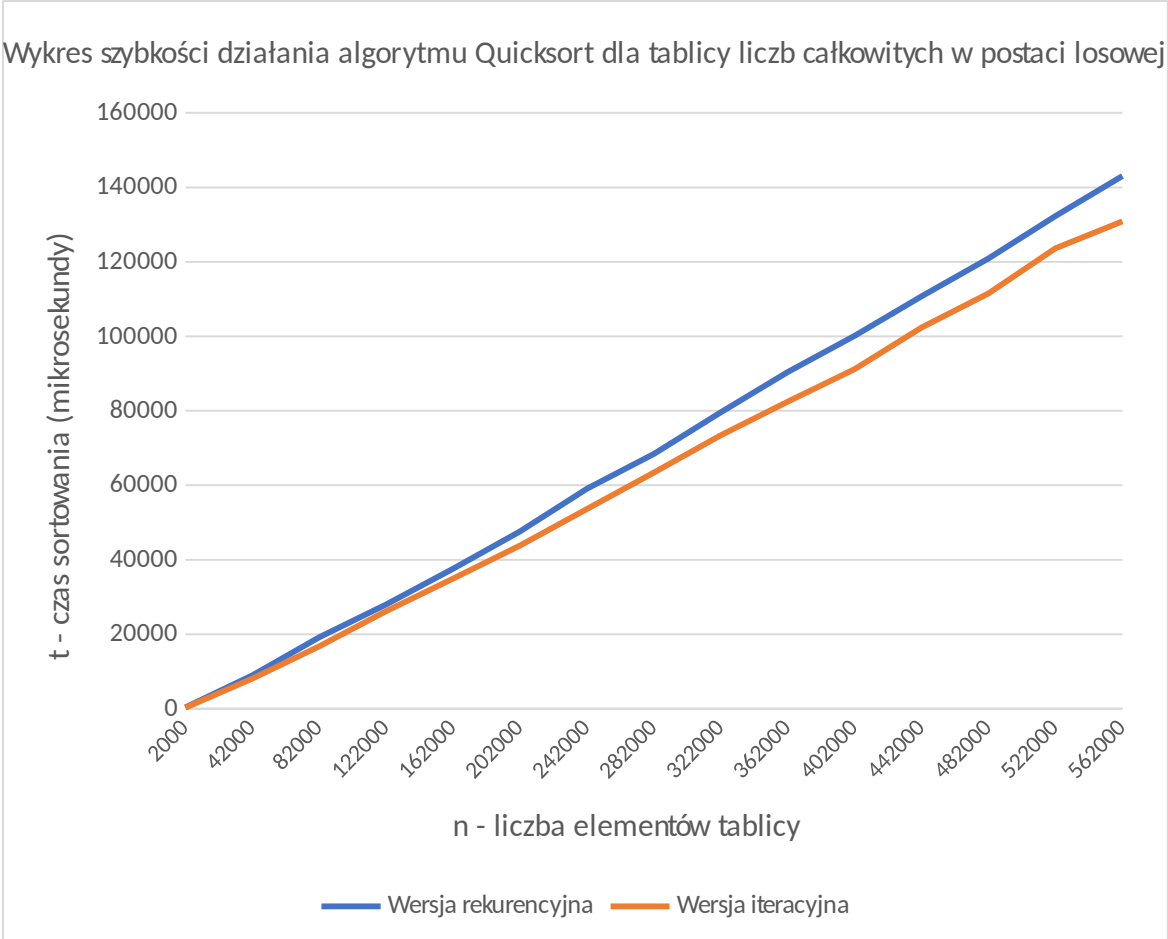
- Selection Sort osiąga identyczne czasy sortowań dla każdego rodzaju ciągu. Jest to wynikiem tego, że żadna z jego obu pętli nie zależy od danych wejściowych, więc algorytm zawsze będzie przechodził przez obie z nich w całości. Dlatego jego złożoność obliczeniowa to zawsze $O(n^2)$.

- Insertion Sort osiąga zdecydowanie lepsze wyniki w sortowaniu szczególnych ciągów postaci stałej oraz malejącej. Dla takich danych jest najefektywniejszym algorytmem spośród czterech. W tych przypadkach w każdej iteracji dochodzi jedynie do porównania elementu z kluczem i nie wykonuje się druga pętla. Dlatego też najgorszym możliwym scenariuszem dla sortowania przez wstawianie jest ciąg posortowany malejąco, ponieważ każda iteracja wewnętrznej pętli będzie efektowała przesunięciem całej posortowanej części w celu poprawnego wstawienia nowego elementu.

- Algorytm Merge Sort osiąga te same wyniki dla wszystkich rodzajów ciągów wejściowych, ponieważ stosuje metodę dziel i zwyciężaj, więc niezależnie od danych wejściowych w każdym kroku dokonuje podziału danych na pół. Czas t posortowania wszystkich podzielonych części będzie wynosił $2t(\frac{n}{2})$, a ich scalania $O(n)$, co po dodaniu i przekształceniu daje złożoność obliczeniową $O(n \log n)$).

- Algorytm Heap Sort osiąga te same wyniki dla wszystkich ciągów liczb za wyjątkiem ciągu liczb w postaci stałej. W tym przypadku maksimum jest równe każdej pozostałej liczbie i porównanie ich nie wywołuje kolejny raz funkcji heapify, co znacząco zmniejsza złożoność czasową do $O(n)$. W normalnych przypadkach dochodzi jednak do kolejnych wywołań, które zajmują dodatkowe $O(\log n)$) czasu, co po zsumowaniu daje złożoność $O(n \log n)$).

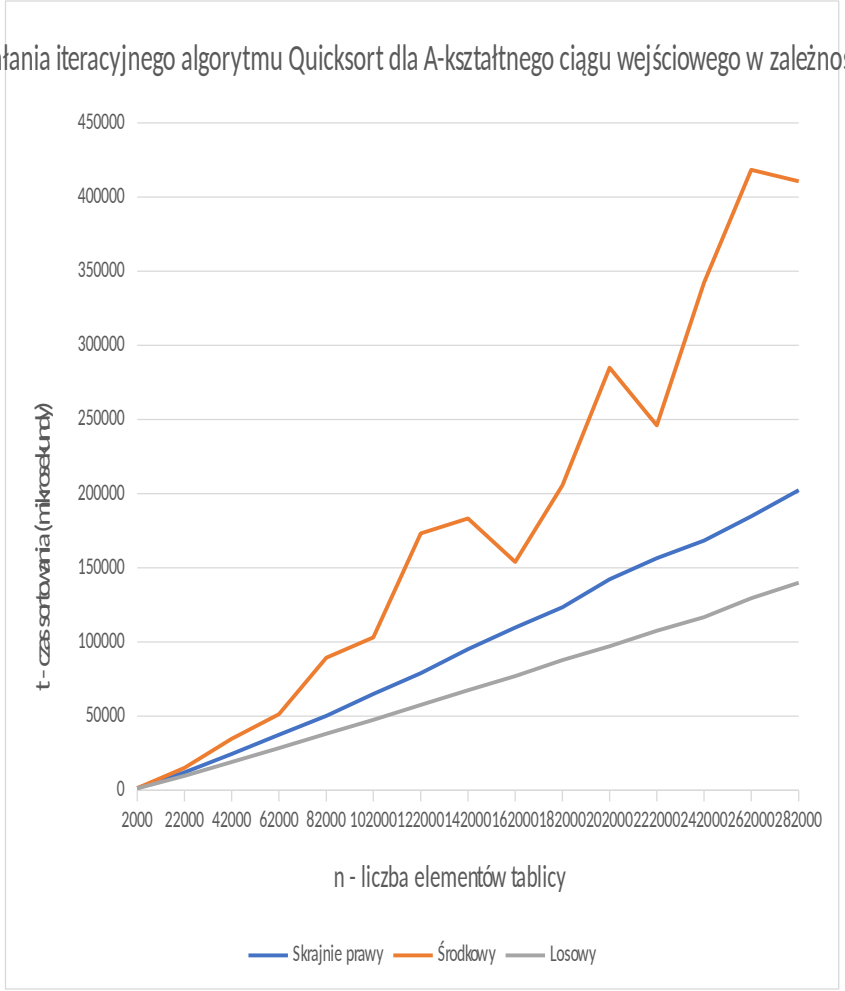
III. Porównanie szybkości działania algorytmu Quicksort w wersji rekurencyjnej oraz iteracyjnej dla tablicy liczb całkowitych w postaci losowej.



Wnioski:

- Obie wersje algorytmu mają taką samą złożoność obliczeniową. Widoczna różnica w szybkości działania wynika z wyższej złożoności pamięciowej rekurencyjnej wersji algorytmu. Powodem tej różnicy jest zasada z jaką realizowane są funkcje zagnieżdżone. W momencie kiedy funkcja wywołuje inną funkcję następuje włożenie jej wraz z wszystkimi zmiennymi lokalnymi na stos. W ten sposób jeśli funkcja jest rekurencyjną, to wraz z każdym następnym jej wywołaniem tworzony jest nowy zestaw zmiennych lokalnych, pochłania to pamięć oraz czas potrzebny na alokację. W przypadku Quicksort tych wywołań może być aż N. Wersja iteracyjna algorytmu jest wolna od tego problemu, cały czas operuje na jednym zestawie zmiennych.

Porównanie efektywności działania iteracyjnego algorytmu Quicksort dla A-kształtnego ciągu wejściowego w zależności od wyboru klucza do porównania: skrajnie prawego, środkowego co do położenia, losowo wybranego.



Wnioski:

- Najgorsze wyniki Quicksort osiąga przy wyborze klucza środkowego. Jest to spowodowane A-kształtnym uporządkowaniem ciągu liczb. Wybierany klucz zawsze będzie bowiem największą liczbą, co jest możliwie najmniej optymalną opcją, ponieważ wszystkie porównywane liczby będą trafiały na jedną stronę (za wyjątkiem ewentualnych równych liczb), co powoduje że algorytm będzie sortował „po jednej liczbie” i jego złożoność czasowa równa się wtedy *O(n)+O(n-1)+...+O(1),* czyli nieefektywne *O(n^2).*

- Lepsze wyniki algorytm osiąga dla skrajnie prawego klucza, chociaż wydaje się to również niezbyt optymalnym wyborem, ponieważ tam znajdują się z kolei najmniejsze elementu ciągu postaci A-kształtnej. Jest jednak spora szansa, że na samym początku ciągu znajdą się liczby mniejsze lub równe wybranemu elementowi, więc nie będzie tak często dochodziło do jednoelementowych

fragmentów danych, co praktycznie zawsze zachodzi przy wyborze klucza środkowego.

- Najlepsze wyniki spośród trzech badanych sposobów Quicksort osiąga dla klucza losowego, ponieważ małe jest wtedy prawdopodobieństwo trafienia na najmniej optymalny klucz, a z kolei większe na bardziej optymalne podziały ciągu w porównaniu do poprzednich dwóch sposobów.