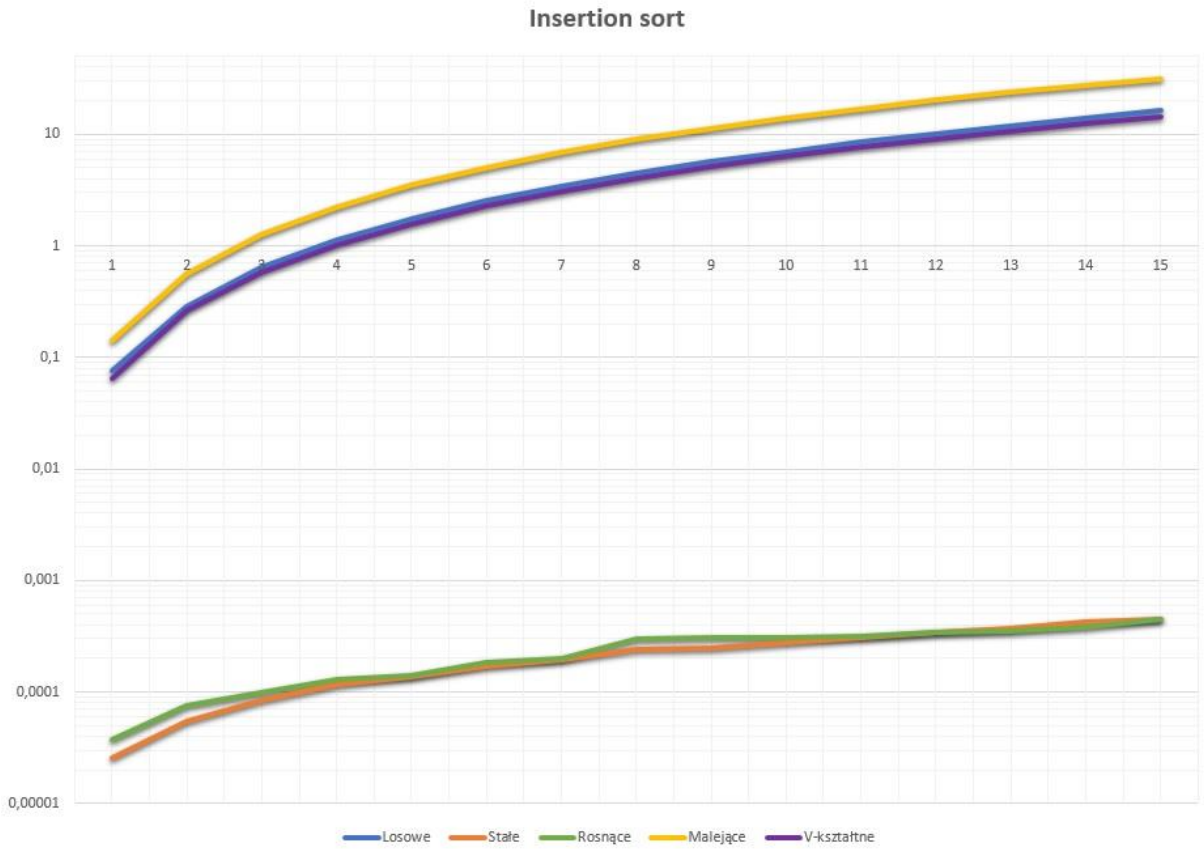
Zadanie 1. – algorytmy sortowania

Grupa I1, Jan Techner (132332), Sebastian Maciejewski (132275)

Wszystkie wykresy są zrealizowane w podobny sposób: na osi oX oznaczono czas działania algorytmów w sekundach, zaś na osi oY przedstawiono ilość danych (15 kroków po 10 000 liczb). Wszystkie pomiary wykonano dla przynajmniej 10 próbek danych (czas ich działania jest średnią czasów dla poszczególnych próbek). Na niektórych wykresach konieczne było zastosowanie skali logarytmicznej, aby zaprezentować na jednym wykresie dane ze stosunkowo szerokiego przedziału.

Na początek porównamy działanie omawianych algorytmów sortowania dla różnych typów danych (punkt drugi rzymski z zadania).

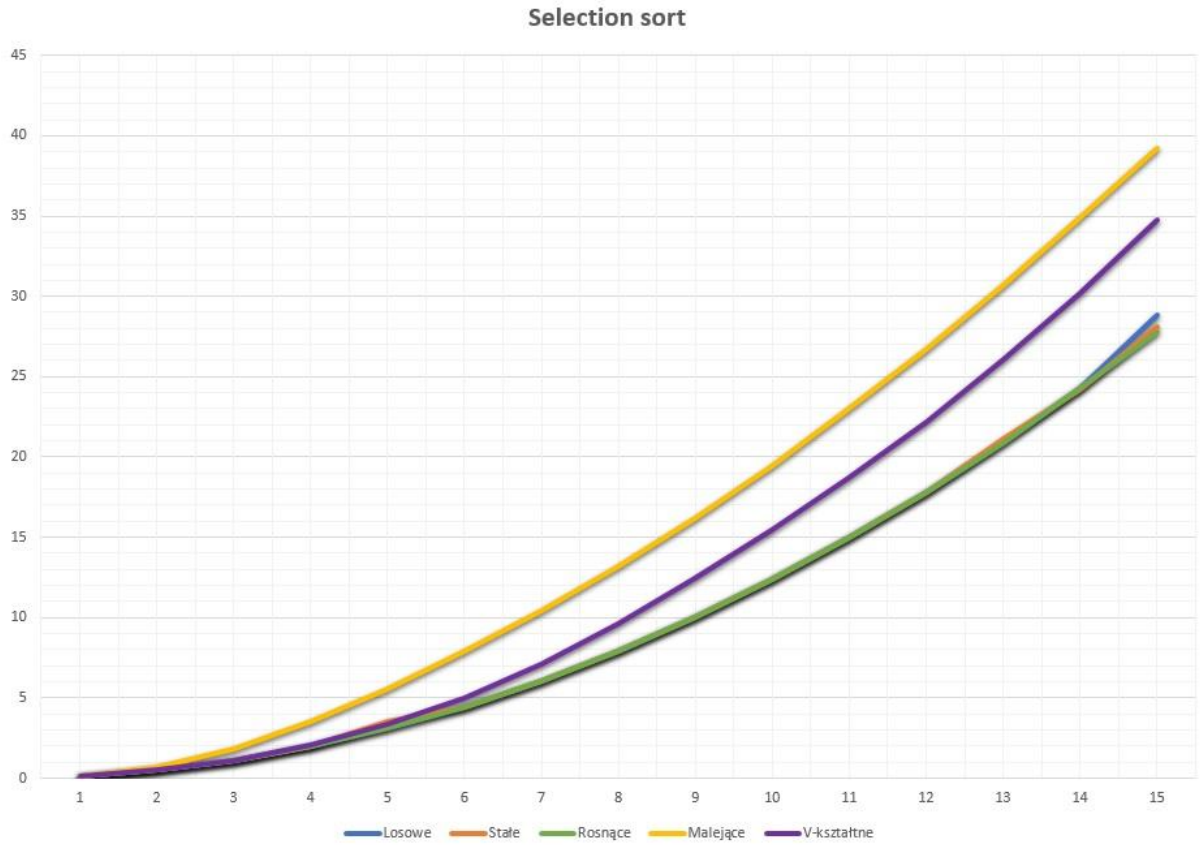


W przypadku algorytmu insertion sort zastosowaliśmy skalę logarytmiczną, gdyż różnica między seriami danych jest bardzo duża. Jak widać, algorytm zakończył się błyskawicznie dla danych stałych i posortowanych rosnąco, co wynika z jego konstrukcji – przebiega on zbiór danych w poszukiwaniu miejsc, w których element n jest mniejszy niż element pierwszy (w kolejnych przebiegach odpowiednio drugi, trzeci, itd.). W posortowanych już rosnąco danych, tak jak w danych stałych, algorytm nie znajduje żadnych elementów, które powinien zamienić miejscami, co za tym idzie całe jego działanie sprowadza się do pojedynczego przebiegu pętli, co skutkuje bardzo szybkim działaniem (jest to najlepszy przypadek danych dla tego algorytmu).

Z tego samego powodu, podanie danych posortowanych malejąco skutkuje bardzo wolnym działaniem, gdyż algorytm przy każdym kroku znajduje element mniejszy niż pierwszy, w tym

przypadku musi dokonać aż n^2 przebiegów (czyli dla jego złożoności obliczeniowej, $O(n^2)$, jest to najbardziej pesymistyczny przypadek).

Dla danych losowych i v-kształtnych algorytm zachowuje się podobnie. Jak widać z wykresu, są to przypadki, w których czas działania algorytmu dla 150 000 liczb jest ponad 15 sekund krótszy niż dla pesymistycznego przypadku, można zatem uznać te zestawy danych za przypadki średnie.

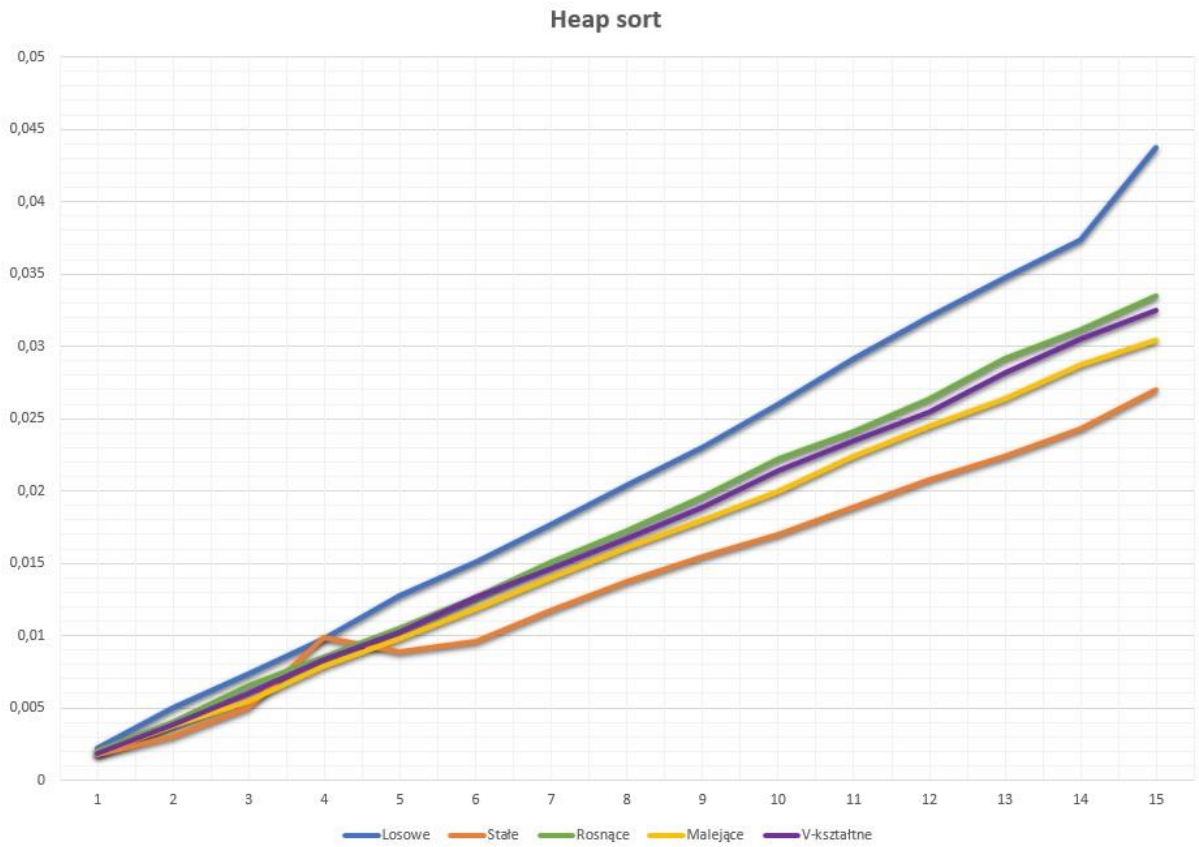


Z wykresu algorytmu selection sort na pierwszy rzut oka można wywnioskować jego złożoność obliczeniową, $O(n^2)$. Widać, że ten algorytm zachowuje się dla danych losowych, stałych i rosnących praktycznie tak samo. Jest to spowodowane tym, że dla n-tego elementu algorytm zawsze przebiegnie dane od n+1 do końca w poszukiwaniu elementów mniejszych od n-tego (czyli liczba porównań jest stała dla dowolnego zestawu danych). Następnie, jeżeli znajdzie takie elementy, zamienia je miejscami. Stąd szybkie działanie dla danych stałych i rosnących – tam algorytm nie musi dokonywać żadnych zamian miejscami, zaś dla danych losowych czas działania również jest nieduży, gdyż musi on dokonać tylko pewnej, losowej ale mniejszej od maksymalnej, liczby zamian miejscami. Można zatem stwierdzić, że dane posortowane rosnąco i dane stałe są optymistycznym przypadkiem dla tego algorytmu.

Po rozważeniu przebiegu algorytmu, od razu wiadomo dlaczego algorytm działa najwolniej dla danych posortowanych malejąco – jest to pesymistyczny przypadek, gdyż przy każdym przebiegu wewnętrznej pętli (od n do końca) znajduje on element mniejszy od n i musi dokonać jego zamiany.

Dla danych v-kształtnych można powiedzieć, że są średnim przypadkiem działania tego algorytmu, gdyż połowa zestawu danych jest posortowana malejąco (są pesymistycznym przypadkiem), zaś połowa jest już posortowana rosnąco i algorytm nie musi dokonywać tam aż tak wielu zamian

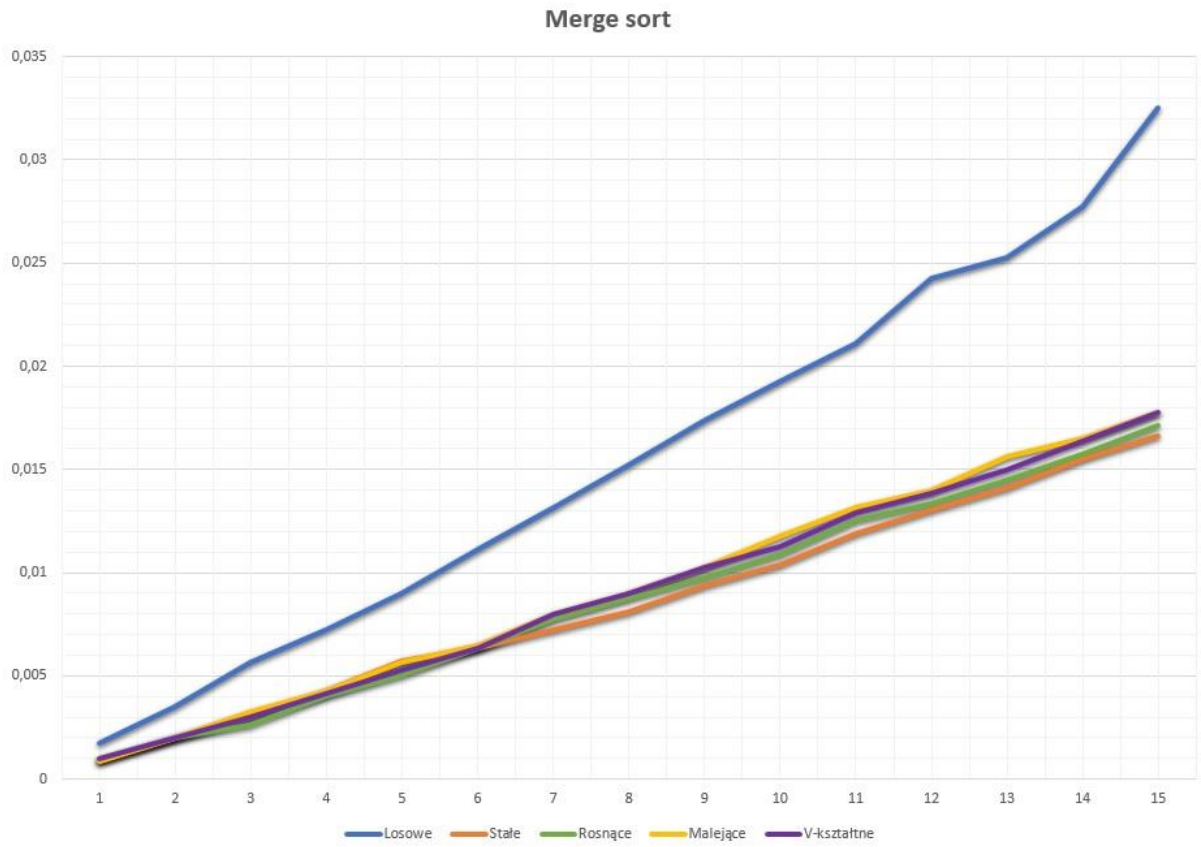
miejscami – stąd miejsce tej próbki na wykresie, gdzie uplasowała się ona między pesymistycznym i optymistycznym przypadkiem.



Już pierwszy rzut oka na wykres czasów działania algorytmu heap sort ukazuje nam dwie rzeczy: przede wszystkim znacznie niższe czasy działania niż czasy dwóch poprzednich analizowanych algorytmów (najwyżej 0,044 sekundy w porównaniu do 43 sekund dla algorytmu selection sort) i zupełnie inny kształt krzywych przedstawiających zależność czasu od ilości danych. Kształt tych krzywych (które w dobrym przybliżeniu przypominają proste) wynika ze złożoności obliczeniowej tego algorytmu – O(n*log(n)).

Widać, że czasy działania dla danych rosnących, malejących, v-kształtnych i stałych są podobne. Wynika to z faktu, iż częścią działania algorytmu jest budowanie stogu, który łatwiej zbudować i przeszukać, gdy dane są w jakimkolwiek porządku. Widać dodatkowo, że najszybciej posortowana została próbka stała, dla której poza łatwiejszym budowaniem i przeszukaniem stogu, dodatkowo nie trzeba wykonywać żadnych zamian miejscami elementów – jest to zatem najbardziej optymistyczny przypadek.

Nieco wolniej algorytm działa dla danych losowych, dla których poza tym, że trudniej mu zbudować i przeszukiwać stogi, to musi on zwykle wykonać dodatkowo dużą ilość zamian miejscami elementów. Można zatem powiedzieć, że losowa próbka danych jest dla tego algorytmu pesymistycznym przypadkiem.



W przypadku algorytmu merge sort widać, że krzywe na wykresie układają się w sposób podobny do wykresu dla algorytmu heap sort. Ukazuje to zależność między ich złożonością obliczeniową – złożoność tego algorytmu to także O(n*log(n)). Widać także, że algorytm ten, poza tym, że jest ogólnie najszybszy ze wszystkich porównywanych, to dodatkowo cechuje się, podobnie jak heap sort, stabilnymi wynikami dla danych uporządkowanych w jakikolwiek sposób. Wynika to z algorytmu scalania, który w programie pełni rolę kluczową dla złożoności czasowej. Dla danych stałych i rosnących scalanie polega tylko na złączaniu list, już bez konieczności wybierania elementów z poszczególnych ciągów, dlatego też są to przypadki optymistyczne. Dane malejące przy sortowaniu dają rezultat lepszy tylko od danych losowych, gdyż wymagają scalania "na krzyż" już posortowanych ciągów, co zajmuje trochę więcej czasu i jest to przypadek najgorszy z rozpatrywanych czterech typów danych. Dla danych V-kształtnych rezultat plasuje się pośrodku, ponieważ przy pierwszym podziale pierwsza część tablicy będzie malejąca a druga rosnąca, czyli będzie to przypadek średni, bazując na powyższych wnioskach.

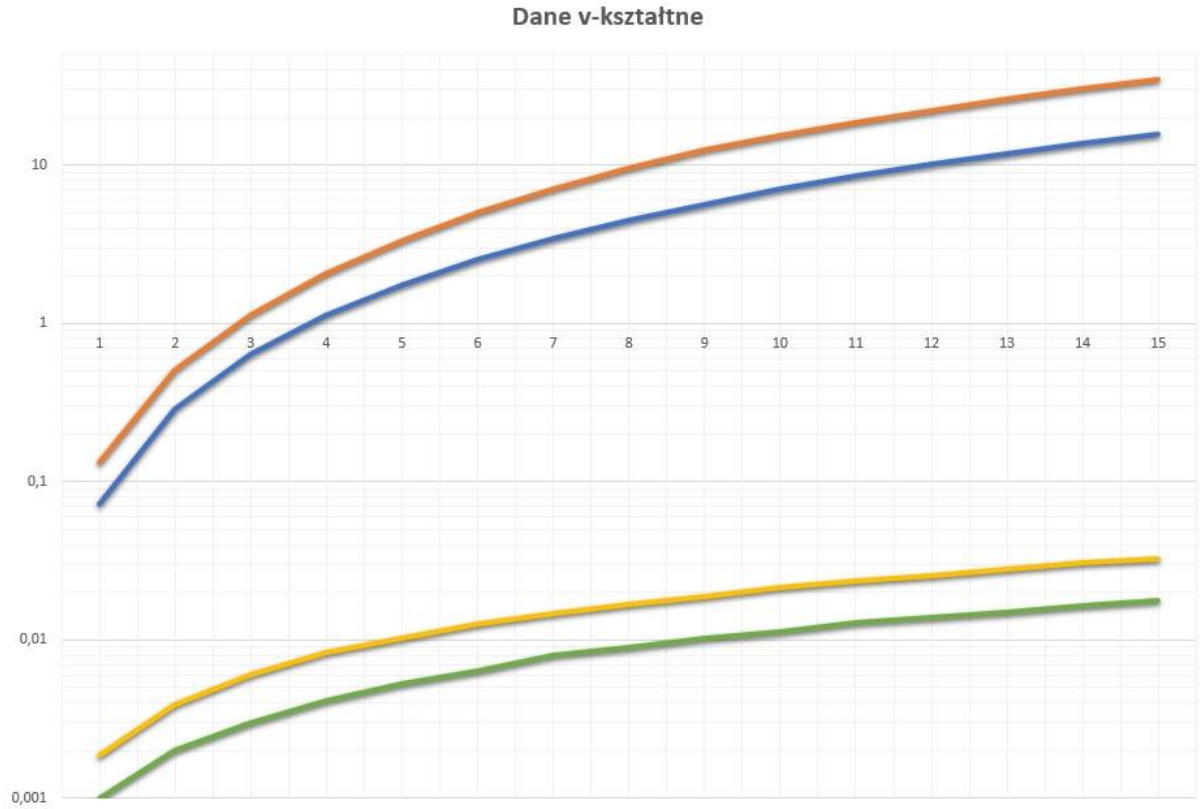
Szczególnie interesującym przypadkiem jest próbka danych losowych, która jako jedyna odstaje od reszty. Jest tak dlatego, że w tym przypadku nie ma żadnej regularności podczas scalania list i algorytm musi wybierać elementy do scalenia z obu ciągów, co jest trudniejsze i bardziej czasochłonne niż scalanie gotowych już posortowanych ciągów, jak ma to miejsce w przypadku pozostałych typów danych. Jest to zdecydowanie pesymistyczny typ danych dla sortowania przez scalanie.

Skoro już zbadaliśmy zależności czasu działania od zestawu danych wejściowych dla wszystkich algorytmów, możemy zająć się porównaniem ich ze sobą wzajemnie. Podobnie jak wykres dla insertion sort, wykresy porównujące różne algorytmy są przedstawione przy pomocy skali logarytmicznej.

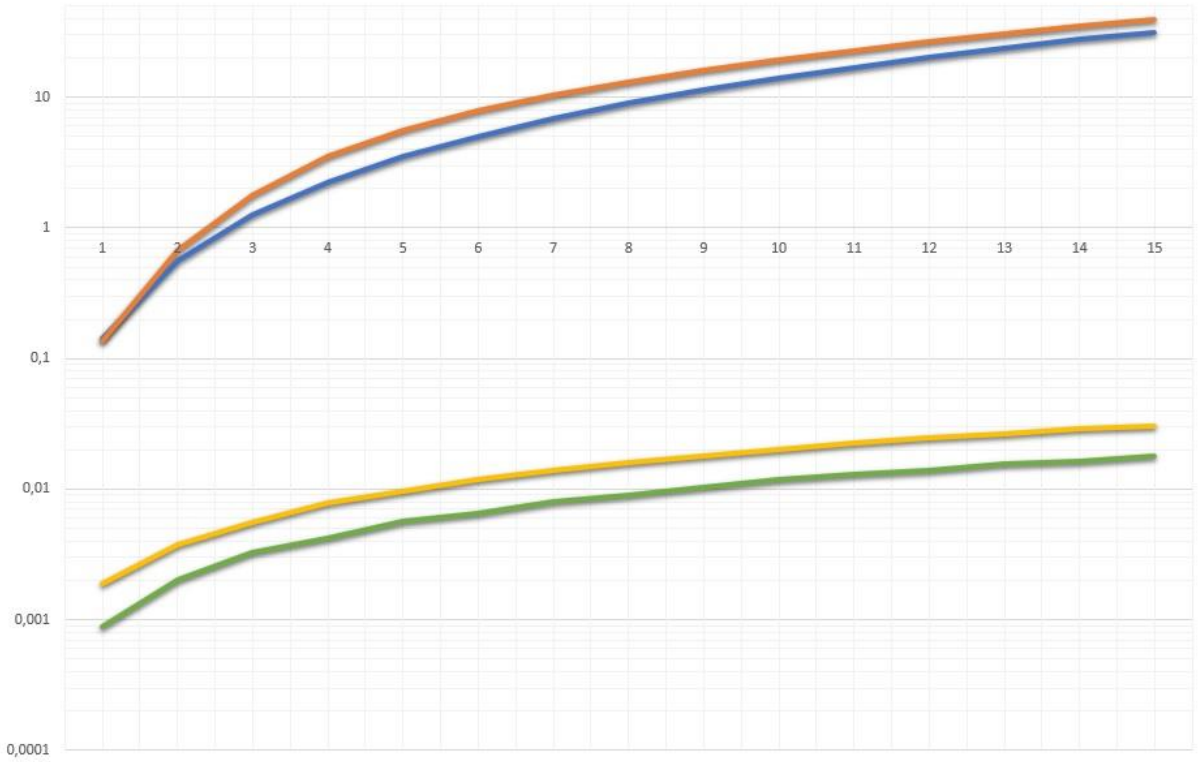
Algorytmy oznaczone są następującymi kolorami

Dla porządku, wykresy są umieszczone obok siebie, następnie opisane są pod spodem.

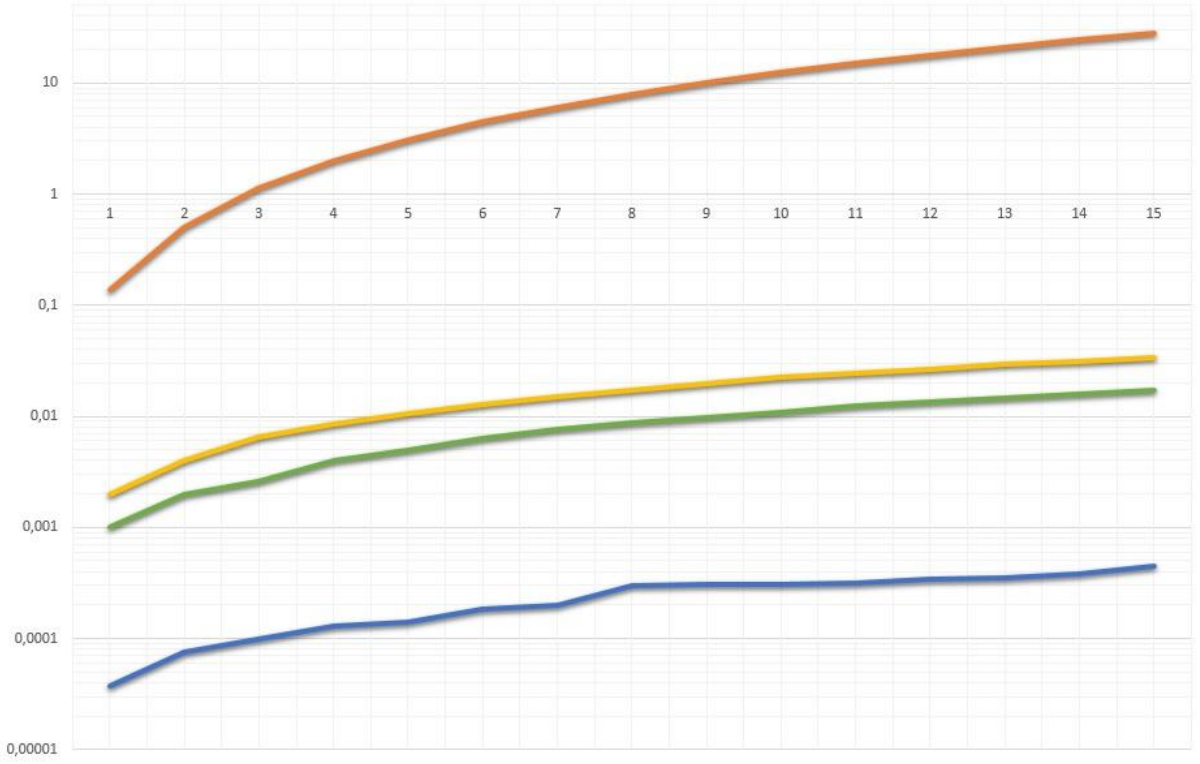

Dane losowe
10
1
0,1
0,01
0,001
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15

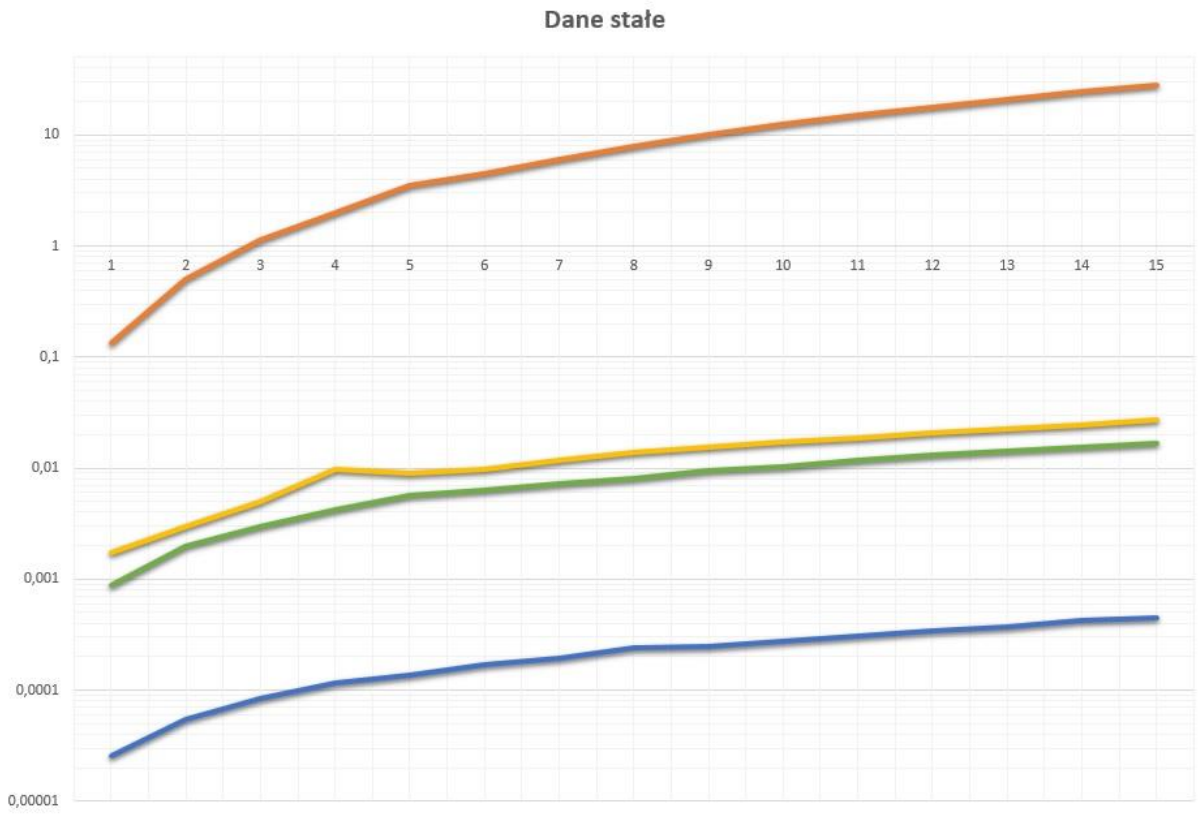

Dane v-kształtne
10
1
0,1
0,01
0,001
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15

Dane malejące



Dane rosnące



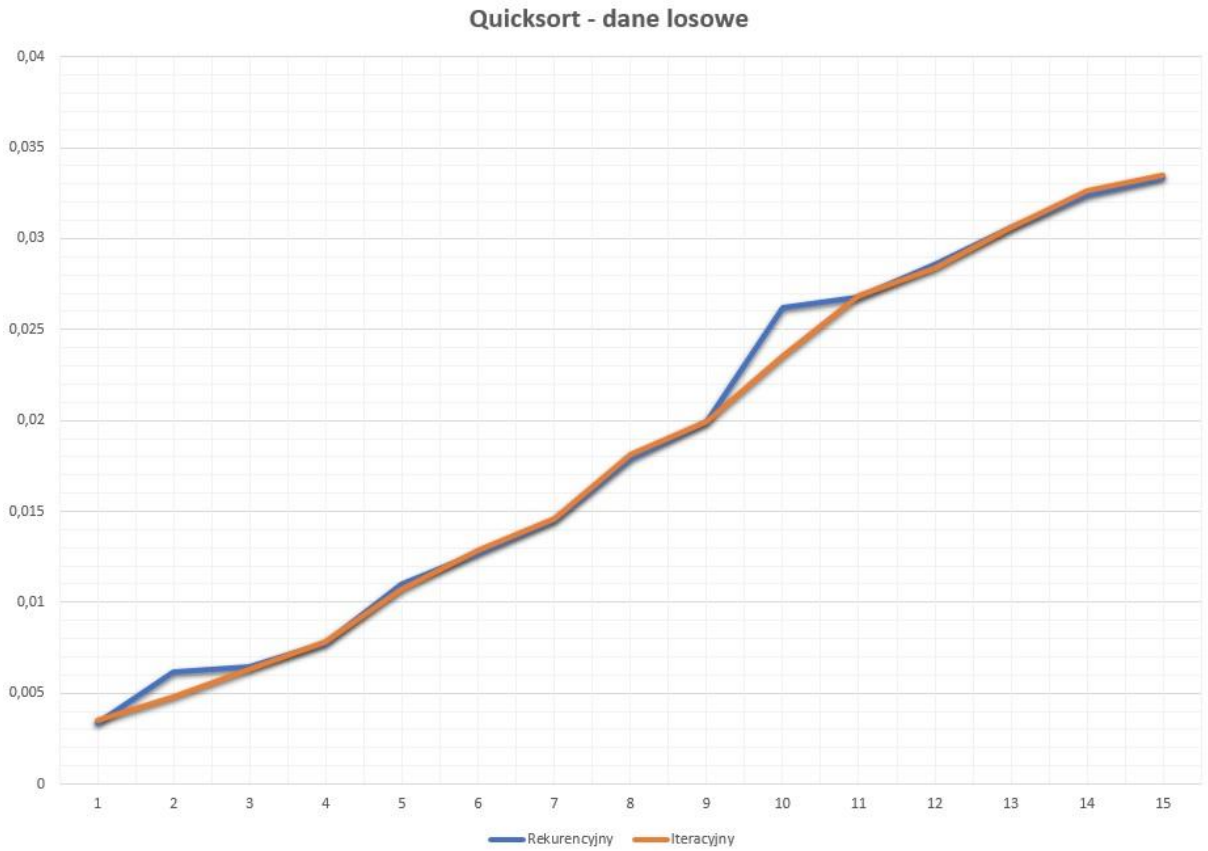


Jak widać, w 3 z 5 przypadków danych wejściowych algorytmy zachowują się w sposób podobny. Wykresy dla próbki losowej i v-kształtnej wyglądają niemalże identycznie, co ukazuje, że te zestawy danych są przypadkiem średnim dla wszystkich algorytmów, przy czym dla danych v-kształtnych czasy działania algorytmów heap sort i merge sort są nieco krótsze niż dla danych losowych. Wynika to z częściowego uporządkowania v-kształtnego zestawu danych, z czego te algorytmy mogą czerpać korzyści (zgodnie z tym, co napisaliśmy wcześniej).

Dla malejącego zestawu danych widać wyraźnie zmniejszoną różnicę pomiędzy selection sort i insertion sort, co wynika z faktu, iż insertion sort musi wykonać największą możliwą liczbę operacji, gdyż dane posortowane malejąco są najgorszym przypadkiem dla tego algorytmu. Wywnioskować można, że generalnie dane posortowane malejąco są najgorszym przypadkiem dla algorytmów selection sort i insertion sort, ale nie dla dwóch pozostałych, co ponownie wynika z opisanych wcześniej własności algorytmów merge sort i heap sort oraz ich tolerancji dla danych, które są w jakikolwiek sposób uporządkowane.

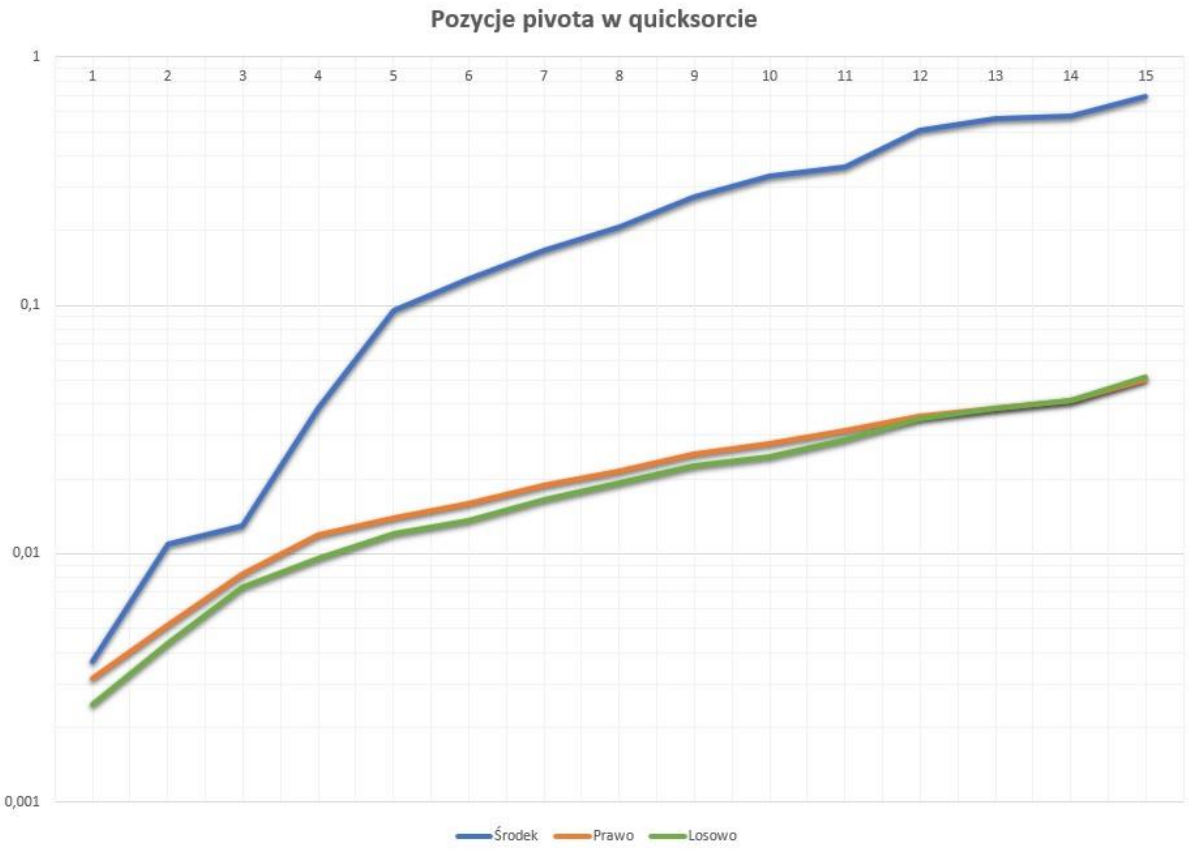
Zdziwienie mogą wywoływać na pierwszy rzut oka wykresy dla danych stałych i rosnących, gdzie widać, że algorytm insertion sort, mający złożoność $O(n^2)$, działa znacznie szybciej niż algorytmy o lepszej złożoności obliczeniowej. Ten zaskakujący wynik jest oczywiście efektem tego, o czym pisaliśmy już w analizie algorytmu inserion sort – dla danych stałych i rosnących całe jego działanie sprowadza się do jednorazowego przebiegu pętlą po wszystkich danych, bez wykonywania żadnych zamian miejscami elementów. Pozostałe algorytmy zachowują się dla tych dwóch zestawów danych w sposób podobny jak dla innych próbek.

Przejdźmy teraz do analizy algorytmu quicksort.



Z wykresu można zauważyć, że czasy działania obu wersji algorytmu są bardzo do siebie zbliżone, aczkolwiek wersja iteracyjna jest minimalnie szybsza od wersji rekurencyjnej. Wynika to z implementacji stosu w obu wersjach, ponieważ w obu wersjach korzystaliśmy z tej samej funkcji do wybierania pivota i przenoszenia liczb na odpowiednią stronę. Podczas wykonywania wersji rekurencyjnej algorytm korzysta z systemowej implementacji stosu, natomiast w wersji iteracyjnej algorytm korzysta z implementacji napisanej przez nas. Ta druga okazała się zauważalnie, aczkolwiek nieznacznie szybsza, także w obu przypadkach złożoność obliczeniowa to O(n*log(n)) (dane losowe są przeciętnym przypadkiem dla tego algorytmu), co znajduje swoje potwierdzenie w kształcie krzywych na wykresie.

Porównując obie wersje quicksorta - iteracyjną i rekurencyjną - można zauważyć ich podobną złożoność obliczeniową : w optymistycznym przypadku jest to O (n*log(n)) a w pesymistycznym aż O(n^2). Złożoność czasowa tych algorytmów jest bardzo podobna i różnice są praktycznie niezauważalne. Natomiast porównując złożoność pamięciową iteracyjna wersja wypada zdecydowanie lepiej niż rekurencyjna, ponieważ głęboka rekurencja zużywa bardzo dużo pamięci.



Do następnego opisu użyliśmy rekurencyjnej wersji algorytmu.

Z wykresu można odczytać, że metoda wybierania środkowego pivota daje zdecydowanie najgorszy rezultat, ponieważ dla ciągu A-kształtnego wybierane są na początku największe elementy, przez co rekurencja przebiega bardzo nieoptymalnie. Metoda wybierania prawego pivota dla danego ciągu działa zdecydowanie szybciej, porównywalnie do metody wybierania losowego pivota, ponieważ prawdopodobieństwo wybrania mediany z danego ciągu jest większe.

Złożoność obliczeniowa w dużej mierze zależy od wyboru klucza, jakim będziemy się posługiwali wybierając pivot. Generalnie dużo szybciej działają metody, w których prawdopodobieństwo znalezienia mediany ciągu w każdym kroku rekurencji jest największe. Najbardziej opytmalny przypadek jest wtedy, gdy algorytm trafi medianę w każdym kroku. Natomiast najbardziej pesymistyczny jest wtedy, gdy algorytm będzie trafiał w najmniejsze lub największe wartości w rozpatrywanym przedziale. Dlatego też bardzo dobrze działa losowe wybieranie piwota, gdyż istnieje wtedy duże prawdopodobieństwo trafienia mediany. Zdecydowanie gorzej działają metody, w których wybiera się skrajne lub środkowe pivoty. Jednakże wszystko zależy od rodzaju danych do posortowania, gdyż od tego zależy jaki pivot będzie wybrany w danej metodzie.