Wiktor Gonczaronek 127191

Poznań 27. 03. 2017 r.

Algorytmy i struktury danych

Badanie algorytmów sortowania

1 Cel badań

Celem badań była ocena wydajności następujących algorytmów: Insertion Sort, Selection Sort, Heap Sort, Merge Sort oraz dla Quick Sort w wersji iteracyjnej i rekurencyjnej.

2 Środowisko testowe

Badania były wykonywane na komputerze z procesorem i5 2,7 GHz, 8GB RAM i systemem operacyjnym OS X 10.12 z kernelem Darwin Kernel Version 16.4.0. Na czas trwania testów wyłączono dostęp do sieci. Algorytmy implementowano w języku Python w wersji 3.5.2.

3 Metodologia

Każdy algorytm był badany dla 15 rozmiarów tablic danych zaczynając od 125 elementów i zwiększajac co 125. Każdy rozmiar zawierał 15 różnych zestawów danych w różnych wersjach:

- rosnącej z fluktuacją o 5 jednostek względem linii trendu,
- malejącej z fluktuacją o 5 jednostek względem linii trendu,
- stałej z fluktuacją +/- 0,5 rozmiaru tablicy,
- v-kształnej z fluktuacją o 5 jednostek względem linii trendu
- stałej o wartości 0,5 rozmiaru tablicy z fluktuacją +/- 5 jednostek.

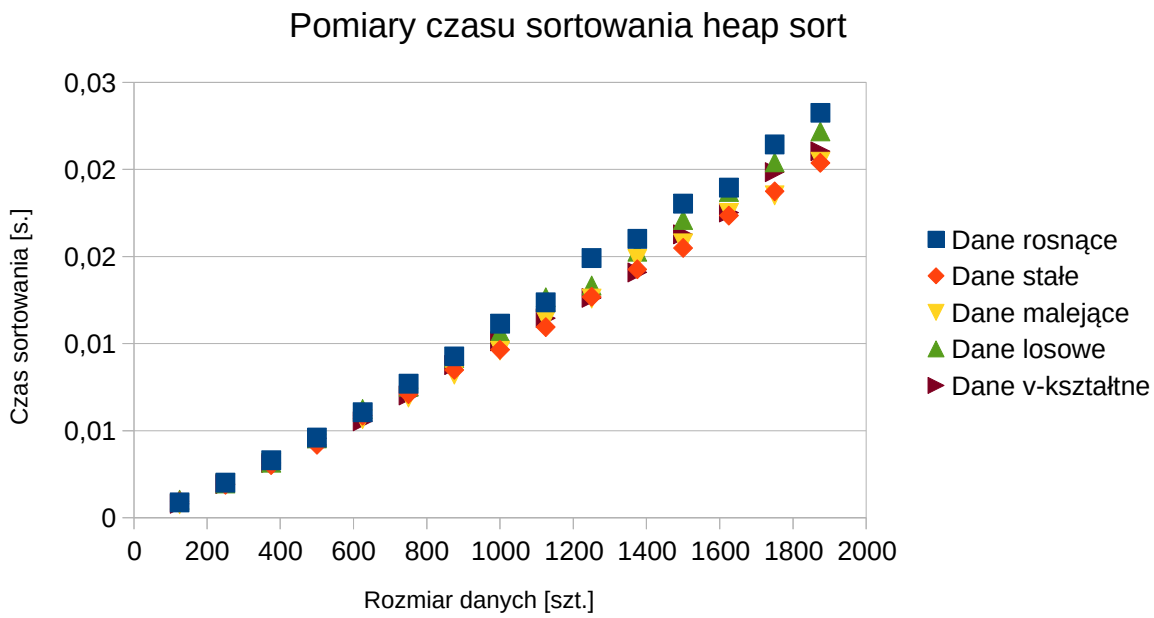
Z każdego zestawu wyciągnięto średnią w celu minimalizacji wpływu zdarzeń losowych.

Do pomiaru czasu wykorzystano funkcję *time()* z modułu *time*. W celu weryfikacji poprawności implementacji dla każdego algorytmu napisano test jednostkowy sprawdzający dla dużej (min. 1000) próby danych losowych czy każdy kolejny element jest nie mniejszy od poprzedniego.

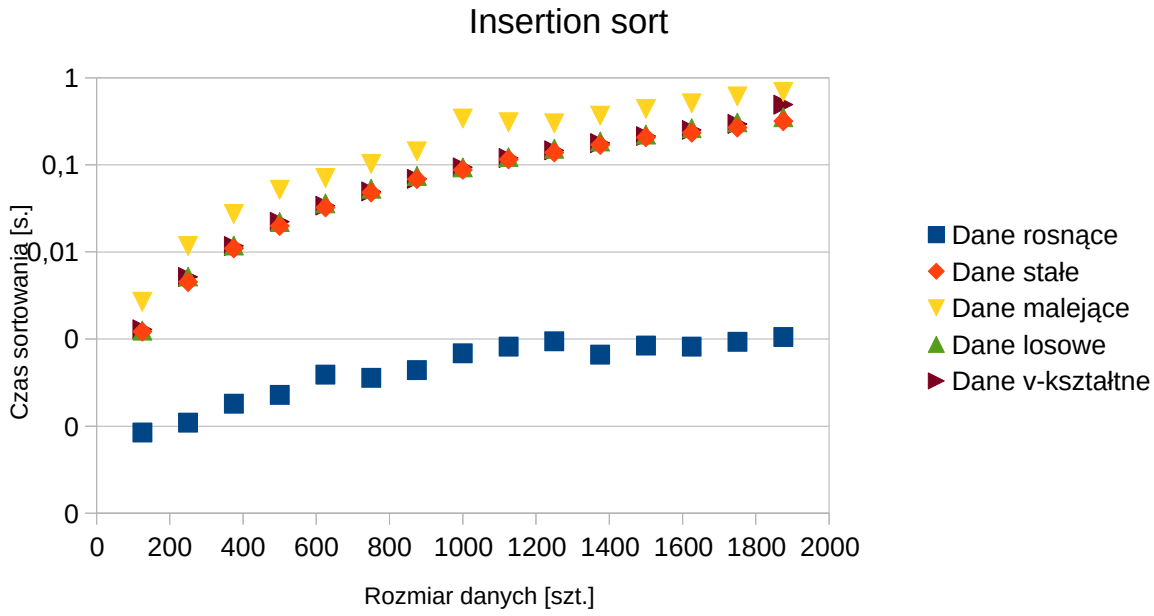
4 Porównanie działania algorytmów

4.1 Ocena efektywności działania algorytmu w zależności od kształtu danych

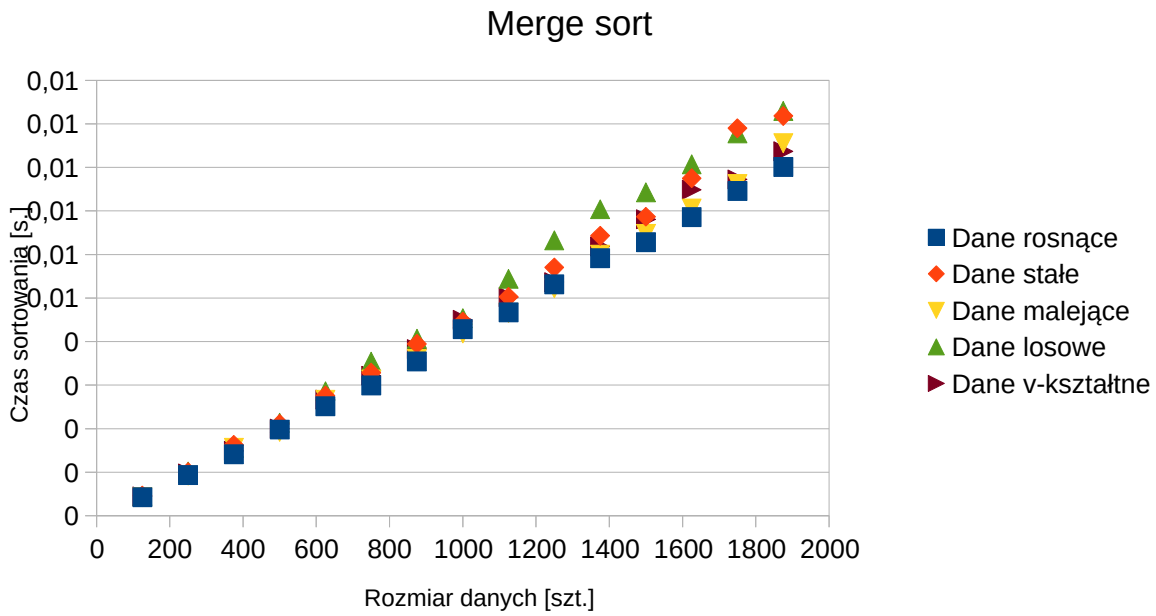
4.1.1 Heapsort



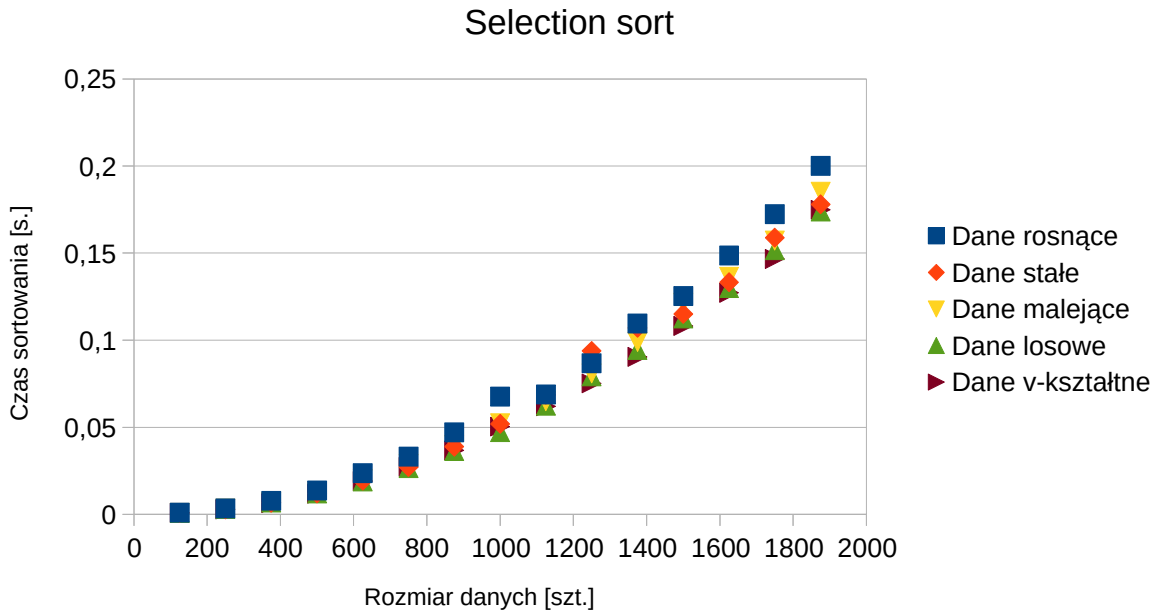
4.1.2 Insertion sort



4.1.3 Merge sort



4.1.4 Selection sort

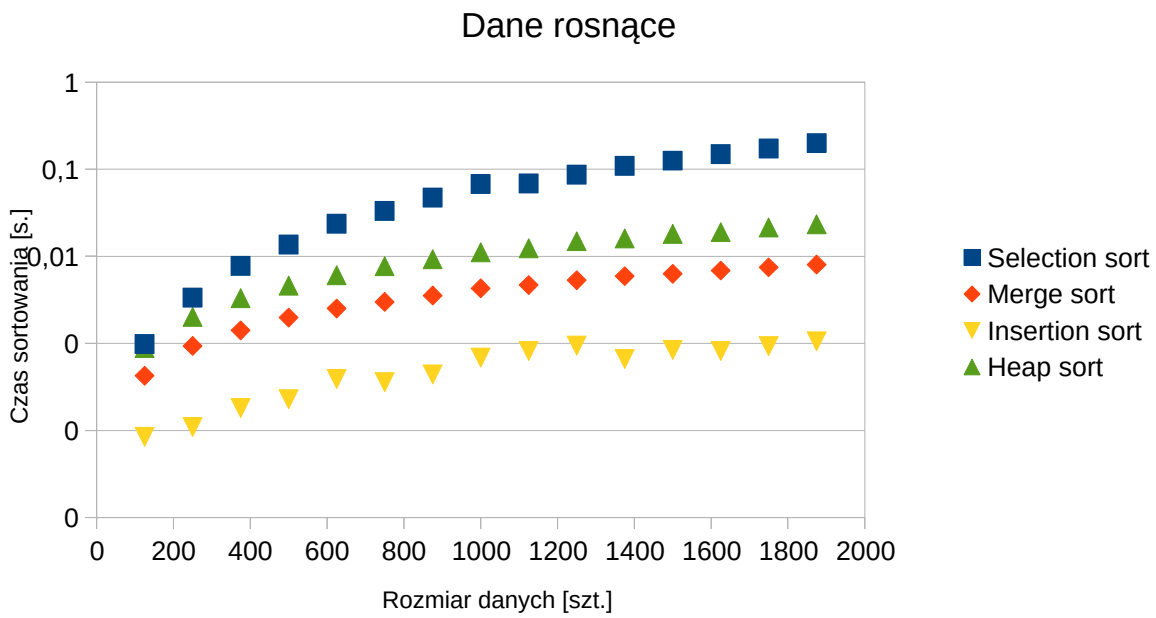


4.1.5 Wnioski

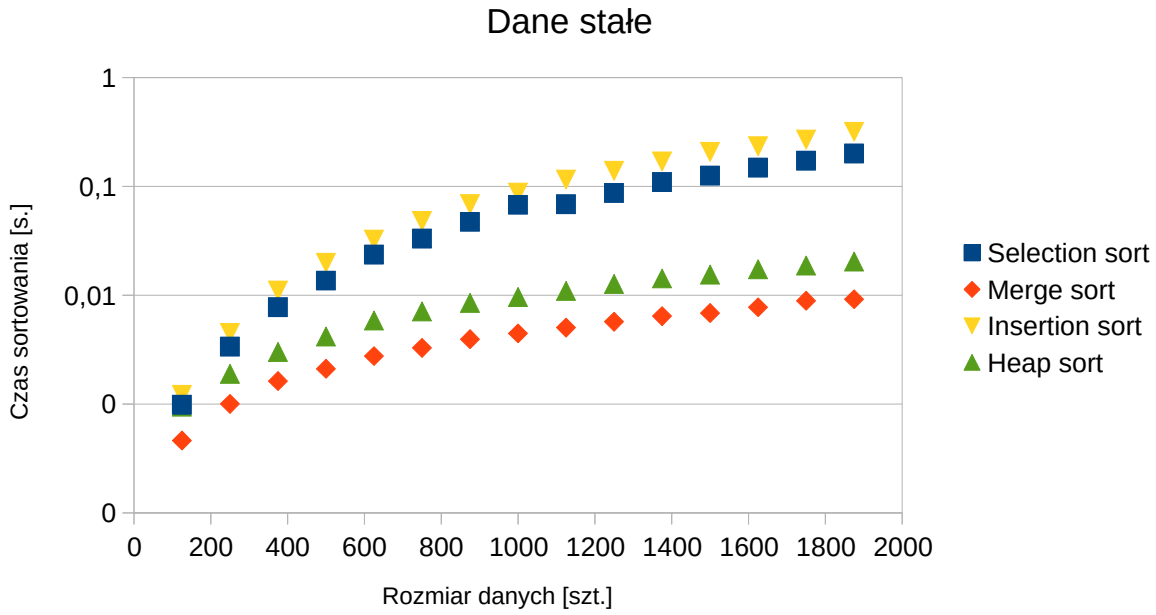
Algorytmy zachowały się zgodnie z oczekiwaniami. Merge sort i heap sort ze względu na stałe czasy podziału imergowania oraz budowy kopca osiągnłęy neizależnie od typu danych czas O(n log(n)). W przypadku insertion sort nastąpiła pewna anomalia dla danych malejących (jeden punkt mocno odstaje), co nie wpłynęło na całość oceny. Wyraźnie zaznaczył się przypadek optymistyczny z prawie posortowanymi danymi, w którym algorytm szybciej dochodzi do końca działania. Selection sort osiągnął czasu $O(n^2)$, co można byłoby poprawić np. przez wprowadzenie flagi sygnalizującej wprowadzenie zmian w wyszukiwaniu maksymalnego elementu podobnie jak ma to miejsce w bubble sort.

4.2 Ocena wpływu doboru algorytmu do typu danych na czas sortowania

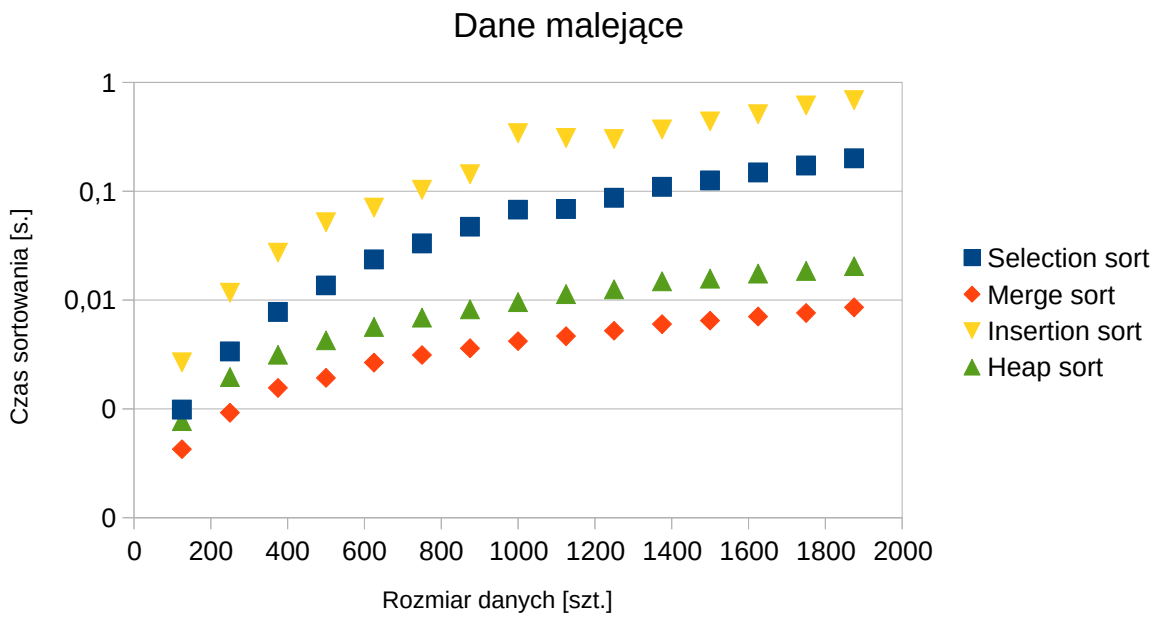
4.2.1 Dane rosnące



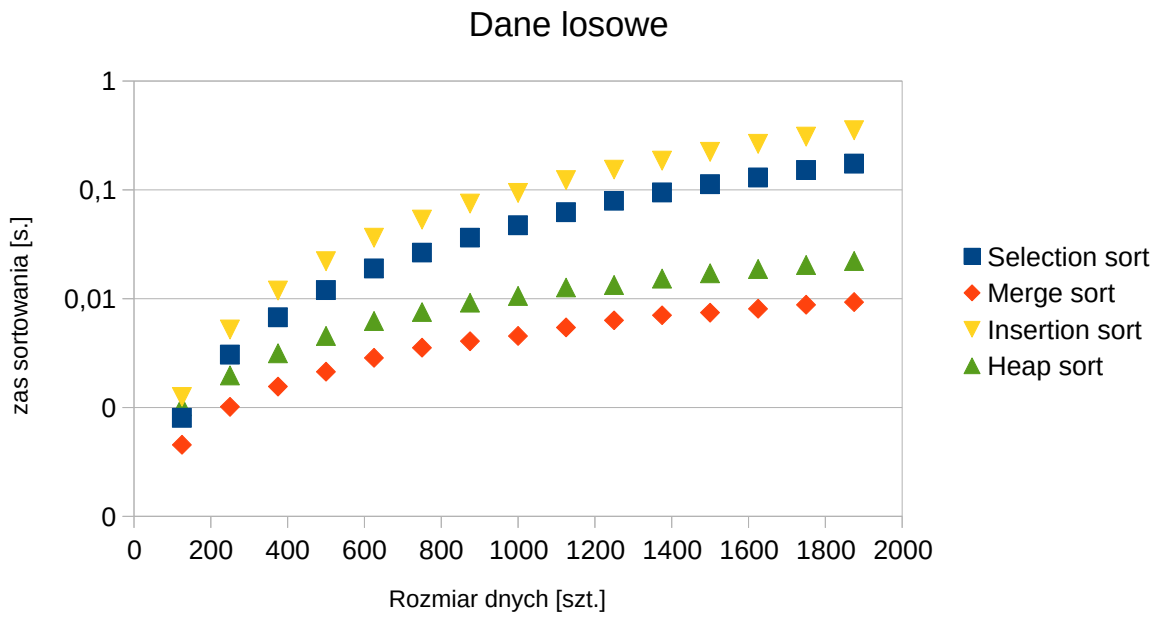
4.2.2 Dane stałe



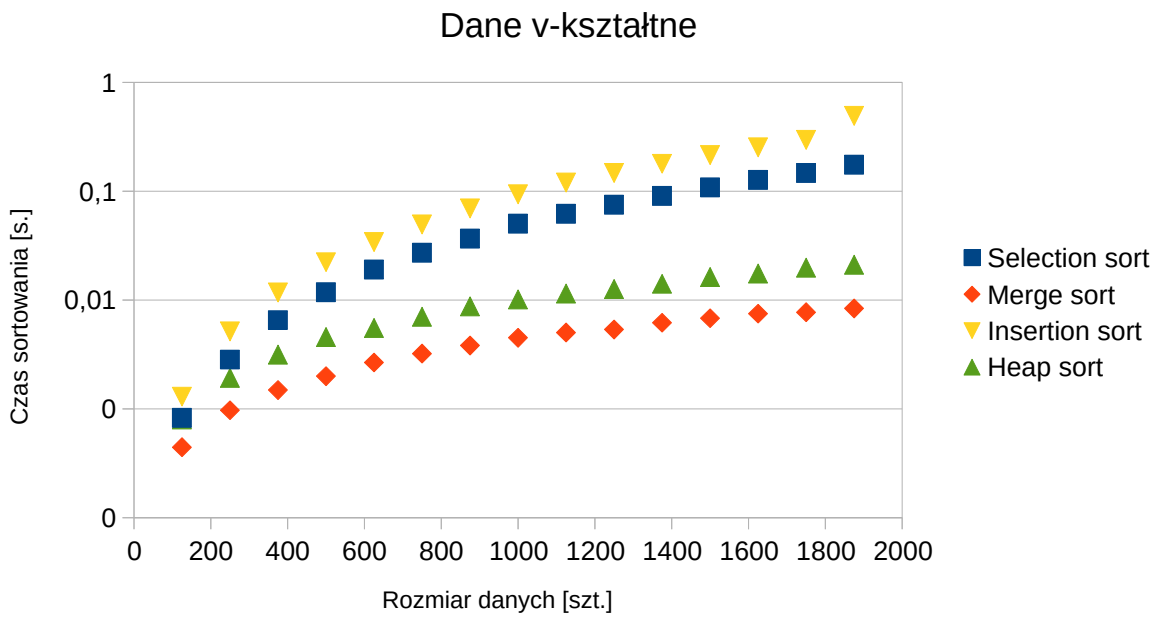
4.2.3 Dane malejące



4.2.4 Dane losowe



4.2.5 Dane v-kształtne

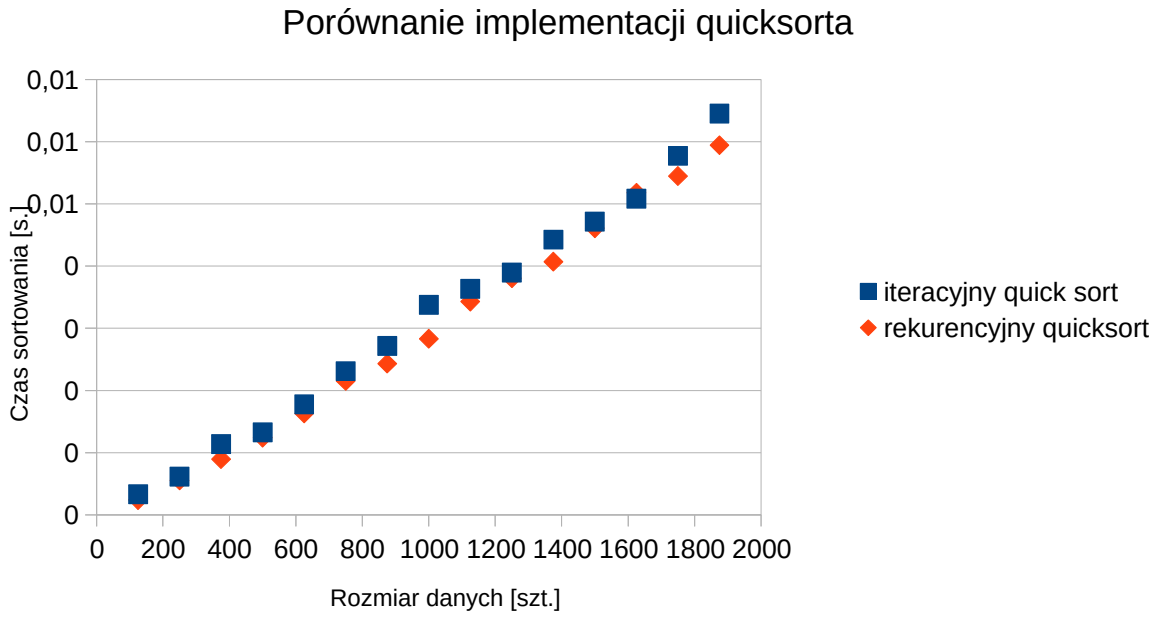


4.2.6 Wnioski

Nie licząc danych rosnących wszystkie wykresy kształtowały się podobnie. Merge sort i heap sort osiągały bardzo podobne wyniki zważywszy na to, że ze względu na ich specyfikę oba wymagają takiej samej liczby kroków niezależnie od rozmiaru danych. Insertion sort ze względu na preferencje dla danych posortowanych okazał się najlepszy w tym przypadku osiągając wyraźnie lepsze rezultaty od reszty.

4.3 Porównanie iteracyjnej i rekurencyjnej wersji quicksorta

Obie wersje korzystały z tego samego sposobu zmiany danych i tego samego sposobu wyboru pivota (był to skrajny prawy element).



Jak widać nie można odnotować większych różnic w obu implementacjach, dla innych sposobów doboru pivota należałoby oczekiwać podobnych rezultatów. Implementacja stosu z wykorzystaniem mechanizmów wbudowanych w Pythona mogła mieć tu znaczenie przez to, że operacje były zoptymalizowane podobnie jak dla wywołań rekurencyjnych. Aby to zweryfikować należałoby przeprowadzić badania na większej próbce danych. Próba dokonania tego skutkowała przerywaniem programu przez rzucenie wyjątku przez zbyt głębokie wejście w stos w przypadku implementacji rekurencyjnej. Nie wiadomo, czy problemem była implementacja czy też błąd był związany z czynnikami niezależnymi (np. ograniczeniami systemu).