Algorytmy sortowania

Maciej Kaszkowiak, 151856

1 Metodologia pomiaru wydajności algorytmów

Zmierzyłem czas wykonywania, liczbę porównań oraz zamian elementów dla następujących sześciu algorytmów:

- Insertion sort
- Bubble sort
- Selection sort
- Heap sort
- Quick sort
- Merge sort

Algorytmy zostały zaimplementowane w języku Python 3.8.10. Testy zostały uruchomione pod systemem Ubuntu 20.04.2 działającym na platformie WSL2.

Dane wejściowe zostały wygenerowane w następujące pięć sposobów:

- Rosnący ciąg liczb z przedziału <1, 10N>
- Malejący ciąg liczb z przedziału <1, 10N>
- V-kształtny ciąg liczb z przedziału <1, 10N> (malejące, a następnie rosnące)
- A-kształtny ciąg liczb z przedziału <1, 10N> (rosnące, a następnie malejące)
- Losowy ciąg liczb z przedziału <-N, N>

Dla każdej dwójki (algorytm, rodzaj danych) czas został zmierzony dla piętnastu różnych rozmiarów tablic, gdzie liczba elementów to następująco: 4, 7, 12, 21, 37, 64, 111, 194, 338, 588, 1024, 1783, 3104, 5405, 9410.

Pomiary zostały wykonane z dokładnością do mikrosekund oraz obejmowały wyłącznie czas sortowania tablicy[1]. Pominięto czas generowania danych wejściowych.

[1] Udostępniłem pod tym adresem dokładne dane wykorzystane do stworzenia wykresów.

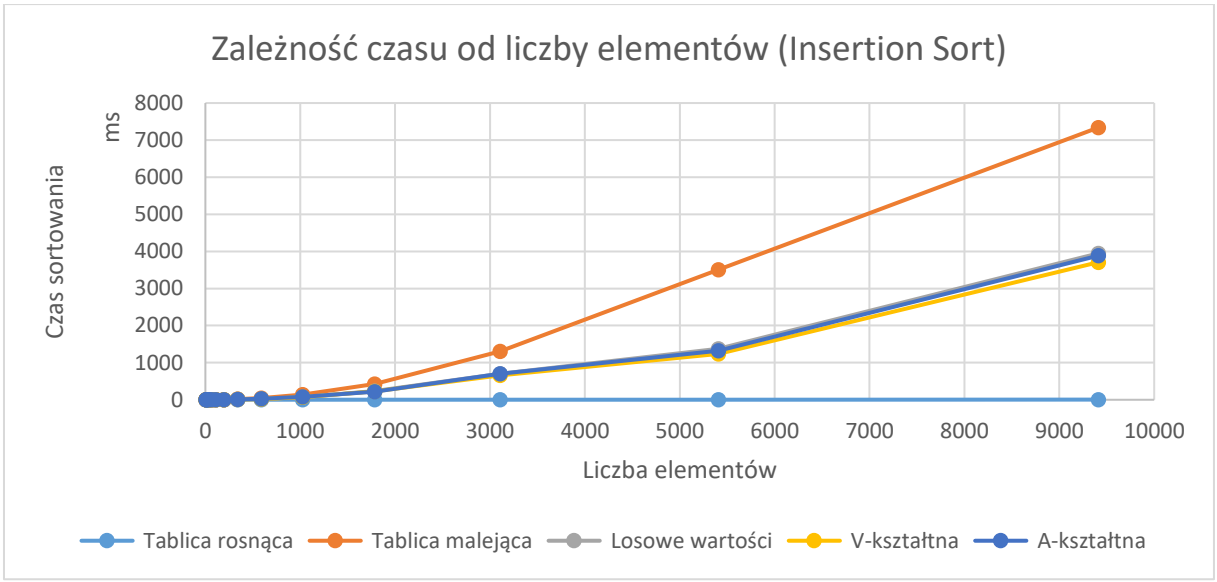
2 Wydajność algorytmów sortowania

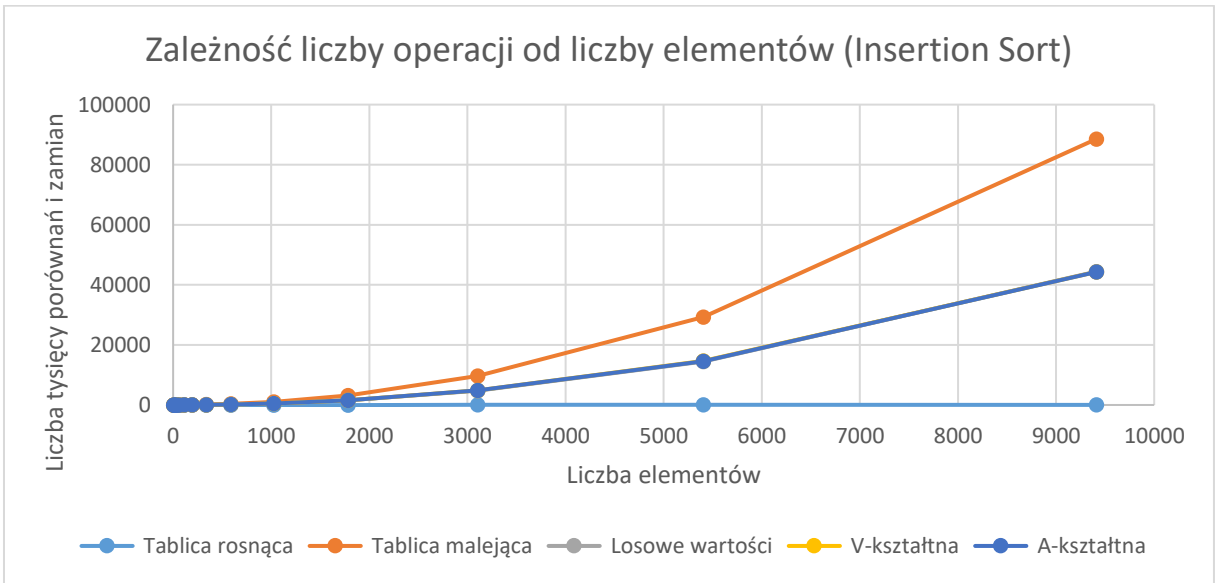
2.1 Insertion Sort

Insertion sort działa poprzez sortowanie lewej części tablicy. Co iterację posortowana część rośnie – pierwszy element z nieposortowanej części tablicy zostaje wstawiony na odpowiednie miejsce.

Złożoność czasowa wynosi O(n^2) w przypadku średnim, O(n^2) w przypadku pesymistycznym oraz O(n) w przypadku optymistycznym.

Możemy zaobserwować przypadek optymistyczny dla tablicy rosnącej (posortowanej) – w tym przypadku wystarczy wyłącznie jedna iteracja przez tablicę, aby zakończyć sortowanie.



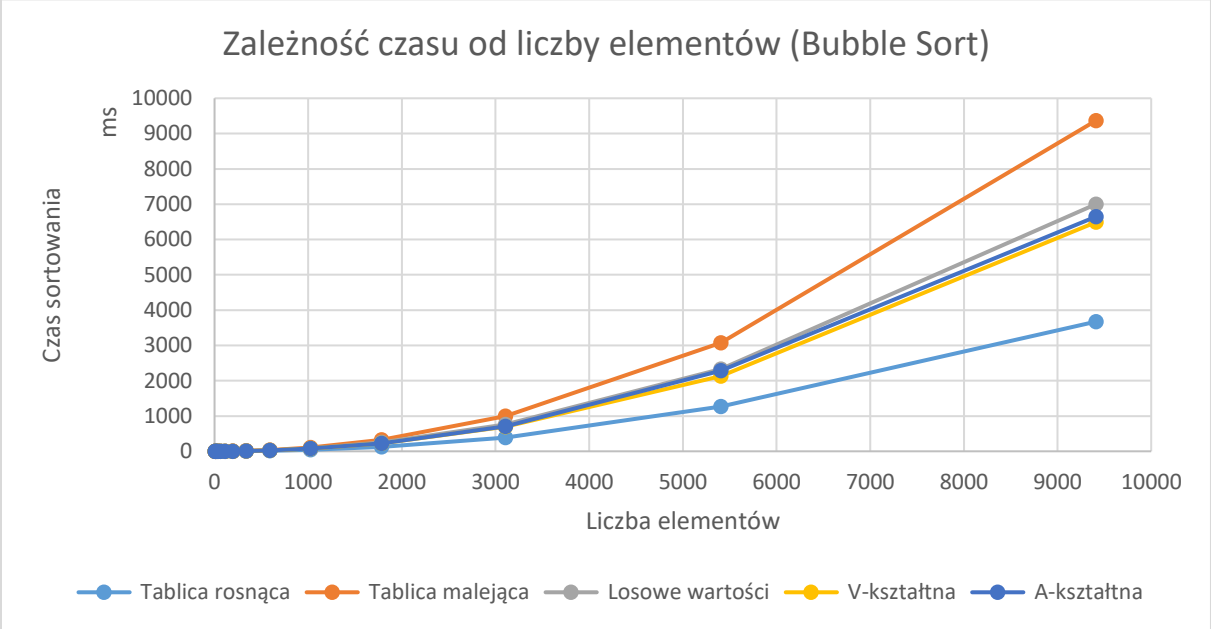


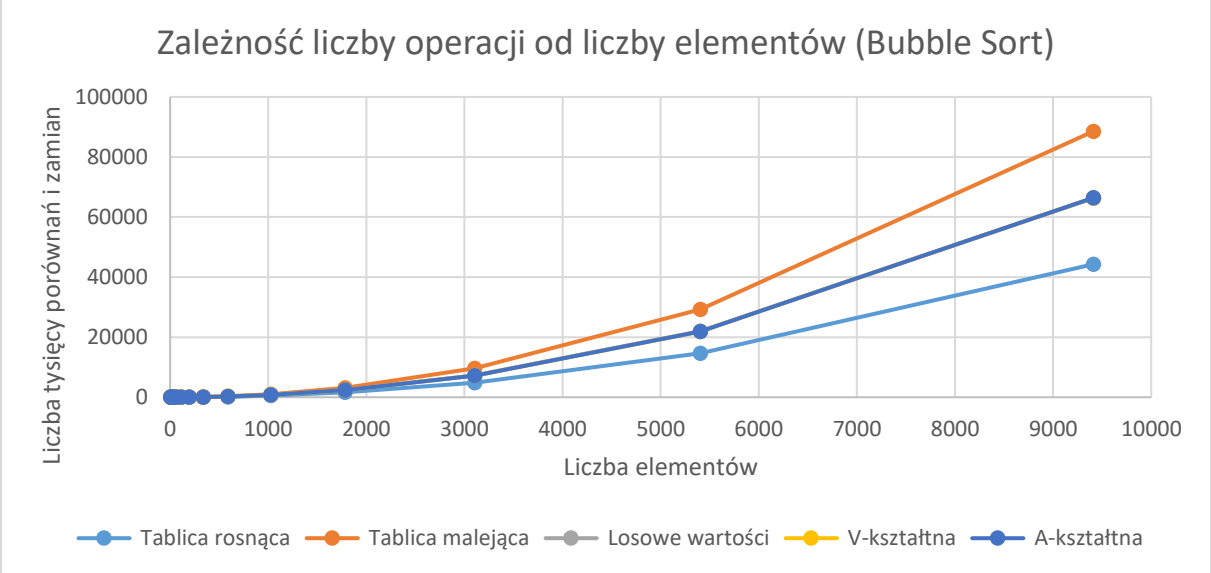
2.2 Bubble Sort

Bubble sort działa poprzez porównywanie i zamianę sąsiadujących ze sobą elementów aż do uzyskania posortowanej tablicy.

Złożoność czasowa wynosi O(n^2) w przypadku średnim, O(n^2) w przypadku pesymistycznym oraz O(n) w przypadku optymistycznym.

Możemy zaobserwować, że przypadek optymistyczny nie pokrywa się z testami, które wykazują złożoność wykładniczą w każdym przypadku. Jest to spowodowane prostszą implementacją algorytmu bubble sort. W kodzie nie wykorzystałem mechanizmu flagi, która przerywałaby wykonywanie skryptu po zaobserwowaniu, że tablica jest posortowana. Drastycznie zmniejszyłoby to czas wykonywania dla tablicy rosnącej.



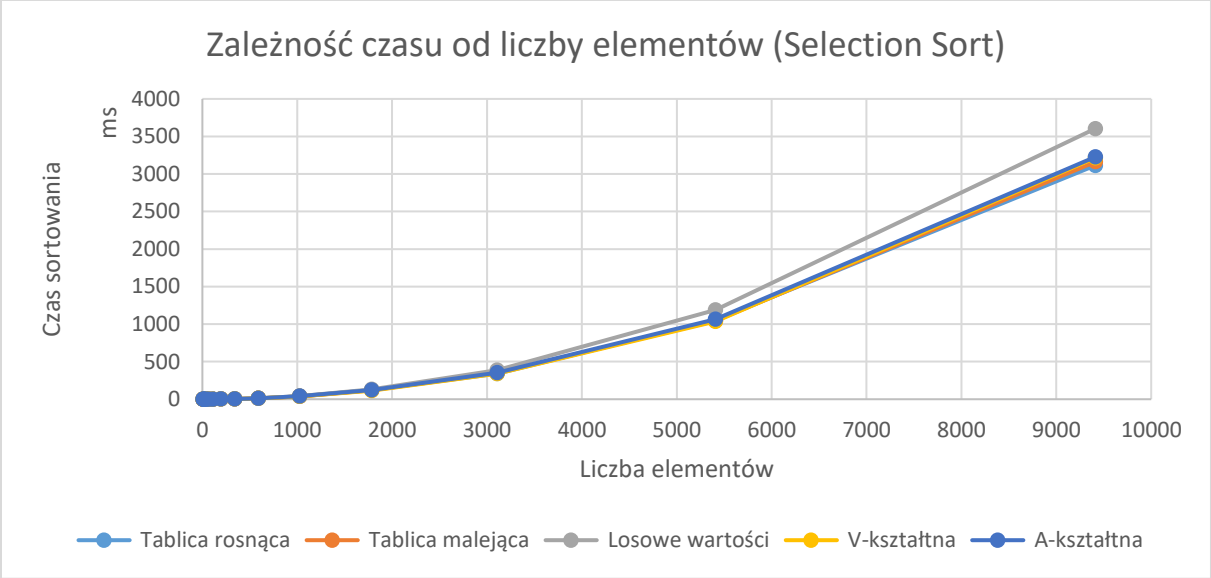


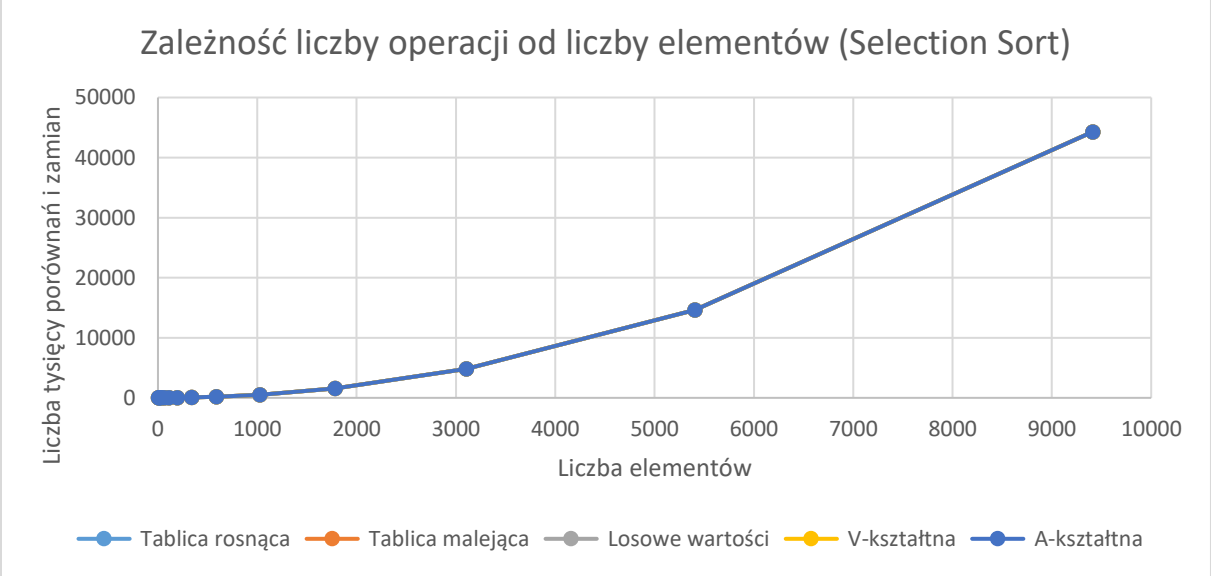
2.3 Selection Sort

Selection sort polega na wybieraniu co iterację najmniejszego elementu oraz wstawianie go na początek tablicy.

Złożoność czasowa wynosi O(n^2) w przypadku średnim, O(n^2) w przypadku pesymistycznym oraz O(n^2) w przypadku optymistycznym.

Możemy zaobserwować, że wydajność jest niemal identyczna bez względu na kształt danych.



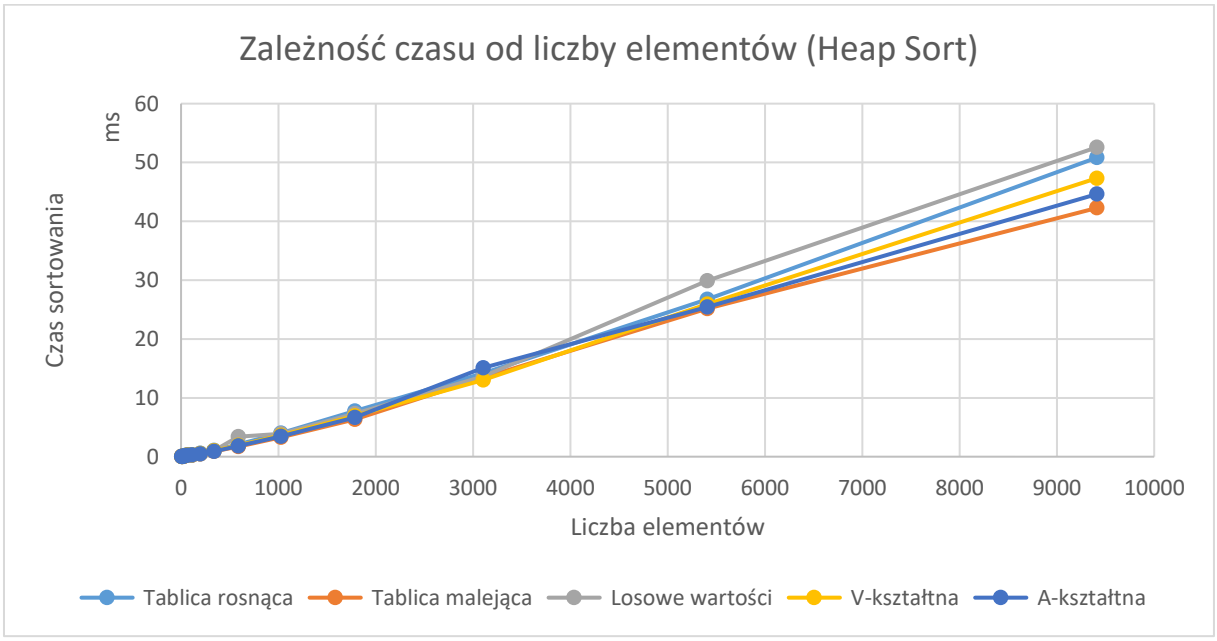


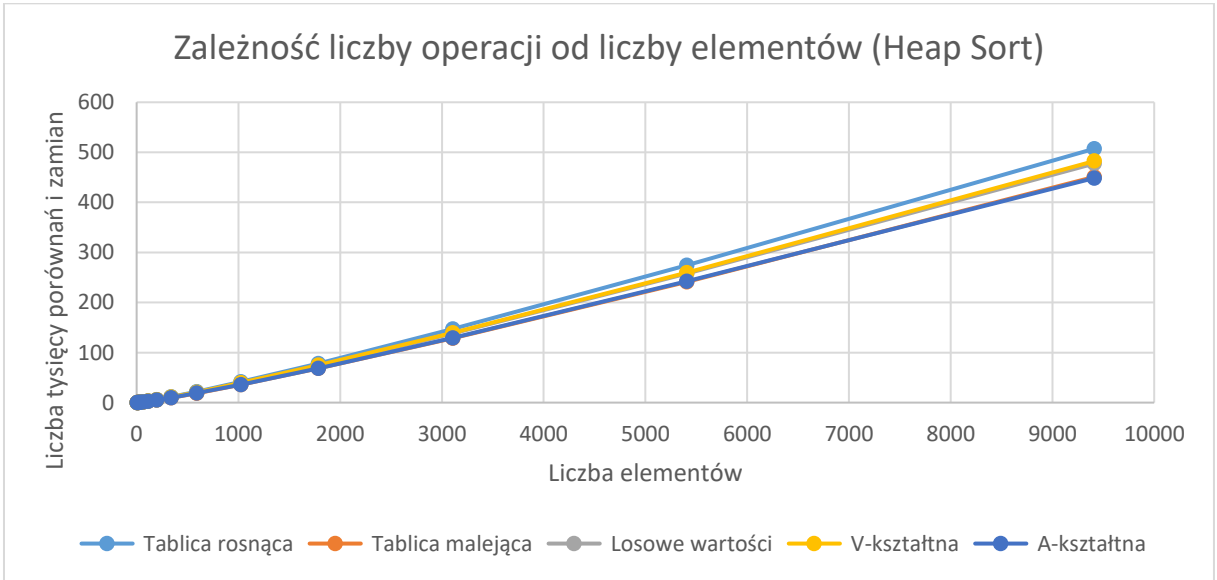
2.4 Heap Sort

Heap sort działa poprzez utworzenie w pierwszym kroku kopca, a następnie zmniejszanie go kolejno o najmniejsze elementy.

Złożoność czasowa wynosi O(n log n) w przypadku średnim, O(n log n) w przypadku pesymistycznym oraz O(n log n) w przypadku optymistycznym.

Możemy zaobserwować, że algorytm jest wydajny bez względu na kształt danych.

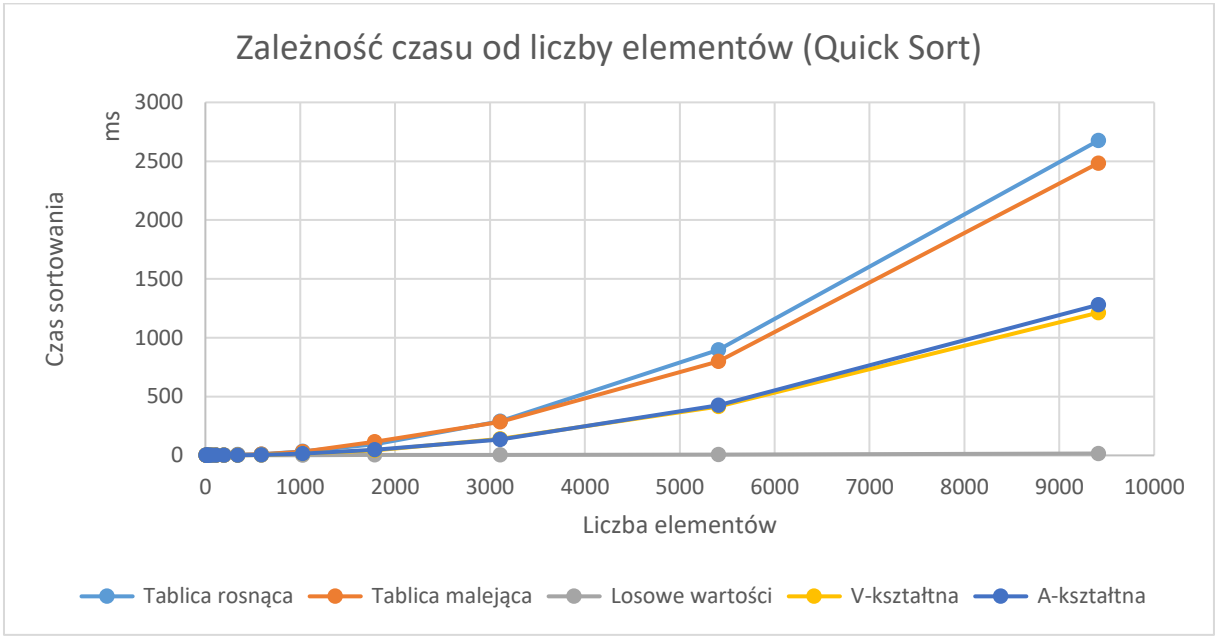


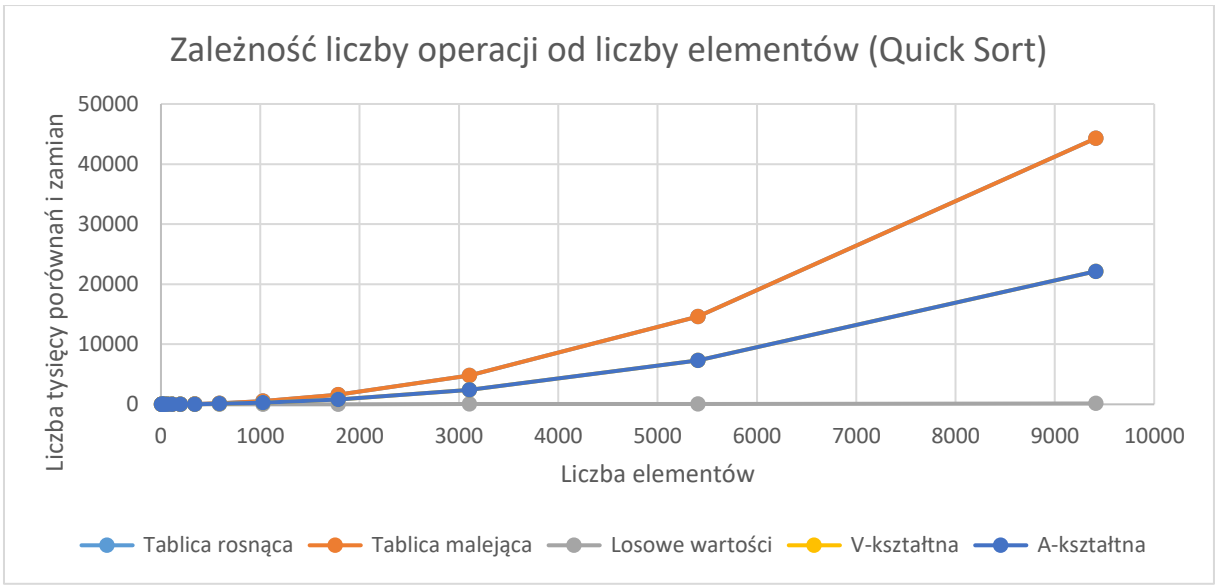


2.5 Quick Sort

Złożoność czasowa wynosi O(n log n) w przypadku średnim, O(n^2) w przypadku pesymistycznym oraz O(n log n) w przypadku optymistycznym.

Quick sort został zaimplementowany w oparciu o rekurencję. Warto wspomnieć, że zgodnie z wymaganiami ostatni element zawsze zostaje wybrany jako pivot. Wybór pozycji pivotu znacznie wpływa na zależność pomiędzy wydajnością algorytmu Quick sort a kształtem danych. Obecna implementacja sprawia, że wydajność degraduje do O(n^2) dla tablic rosnących oraz malejących. Tablice V-kształtne oraz A-kształtne również ulegają spadkowi wydajności. Jest to wywołane przez niezbalansowaną liczbę elementów mniejszych / większych od pivota – wybieranie skrajnego elementu w posortowanym ciągu zmaksymalizuje liczbę wywołań rekursywnej funkcji Quick sort.

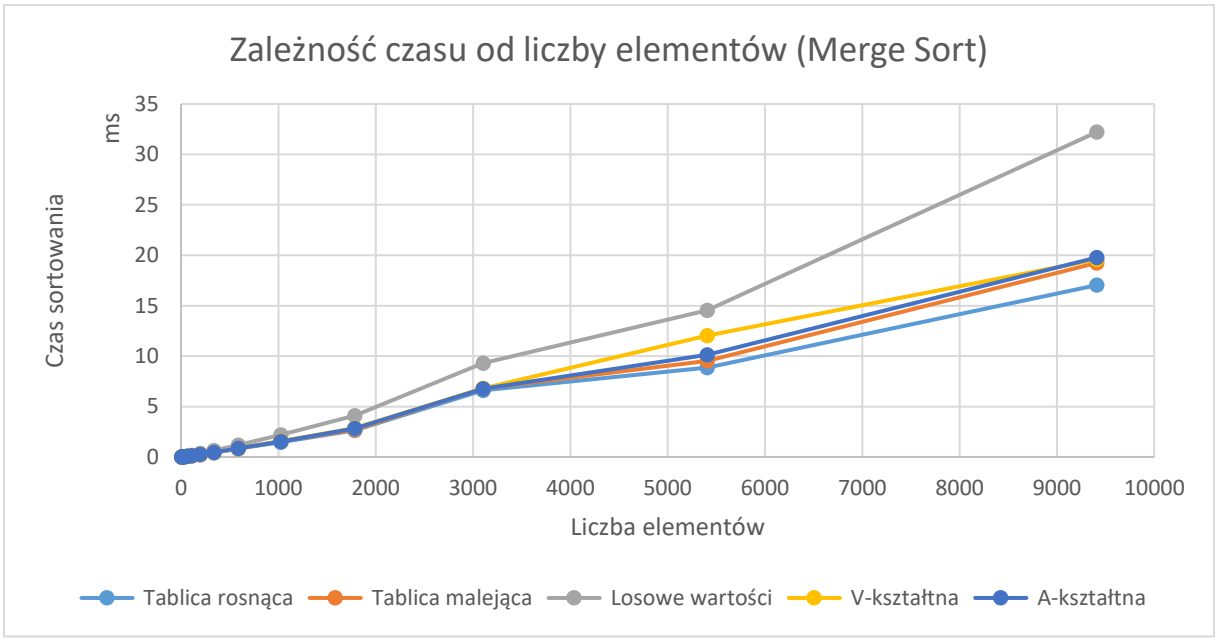


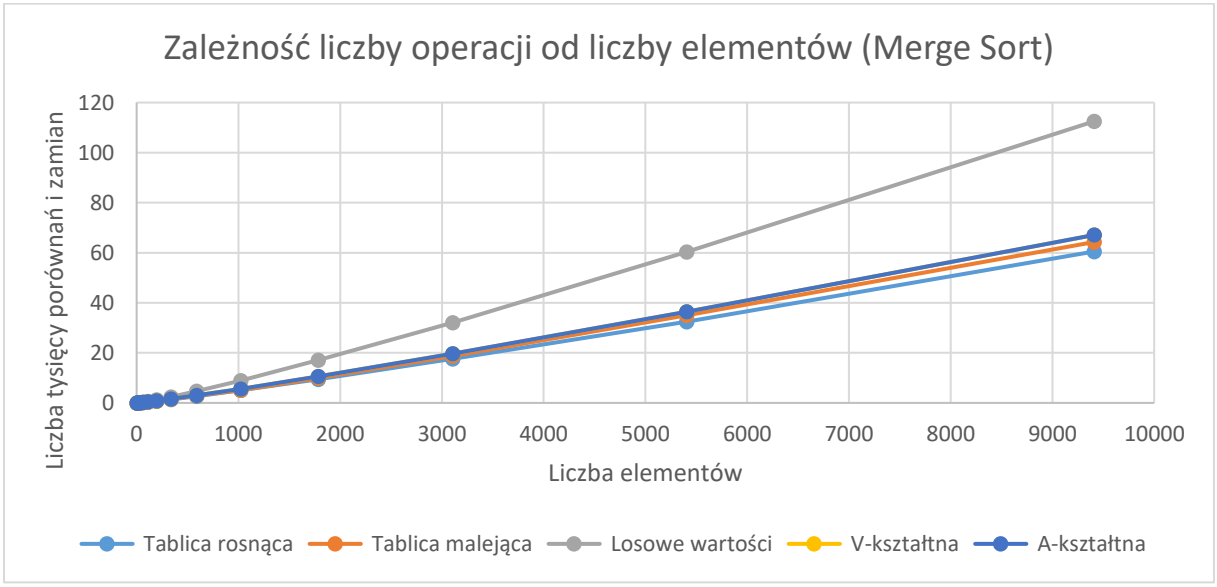


2.6 Merge Sort

Złożoność czasowa wynosi O(n log n) w przypadku średnim, O(n log n) w przypadku pesymistycznym oraz O(n log n) w przypadku optymistycznym.

Możemy zaobserwować, że algorytm jest wydajny bez względu na kształt danych. Minimalnie niższa wydajność algorytmu dla danych losowych (bez posortowanych ciągów) może zostać wytłumaczona implementacją. Algorytm wykorzystuje w ciele funkcji łączącej dwie tablice odmienne funkcje w zależności od miejsca występowania. Dane bez posortowanych ciągów będą dłużej wykorzystywały mniej wydajną funkcję pop. Również implikuje to liczba operacji – większa liczba porównań wywołana jest przez mniejszą liczbę podciągów, które natrafiają na warunek brzegowy w trakcie funkcji merge.



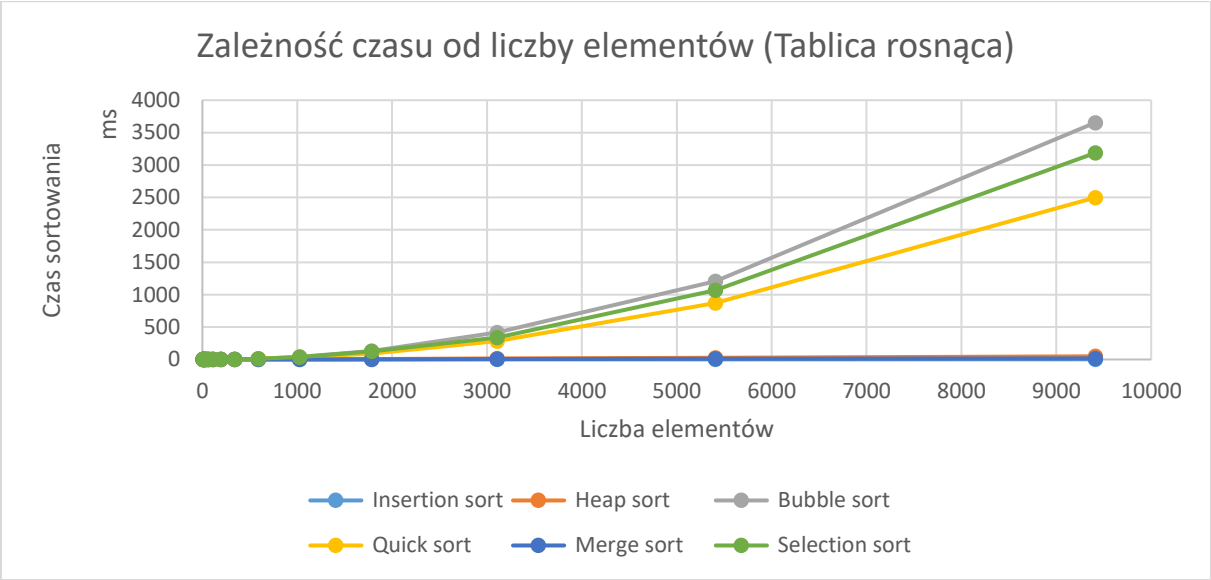


3 Porównanie algorytmów sortowania dla poszczególnych typów danych

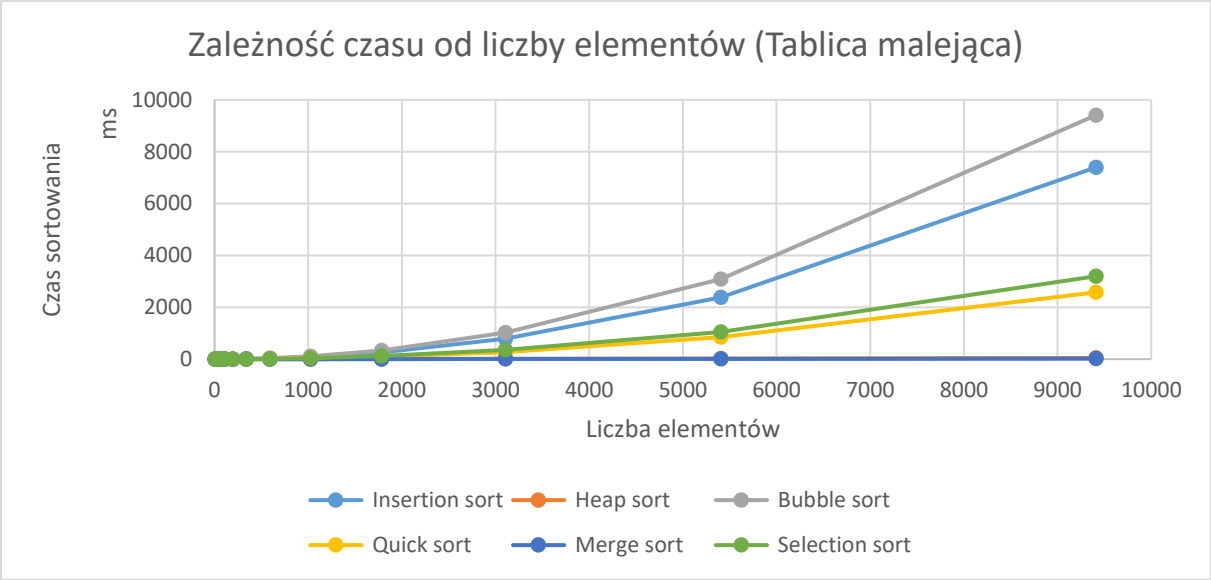
Porównanie algorytmów dla poszczególnych typów danych nie tworzy dodatkowych wniosków – możemy zauważyć dokładnie te same schematy co wcześniej:

- dobrą, powtarzalną wydajność Heap oraz Merge sorta;
- niską wydajność Bubble, Insertion oraz Selection sorta (poza przypadkiem optymistycznym);
- mieszaną wydajność Quick sorta przy obecnej implementacji.

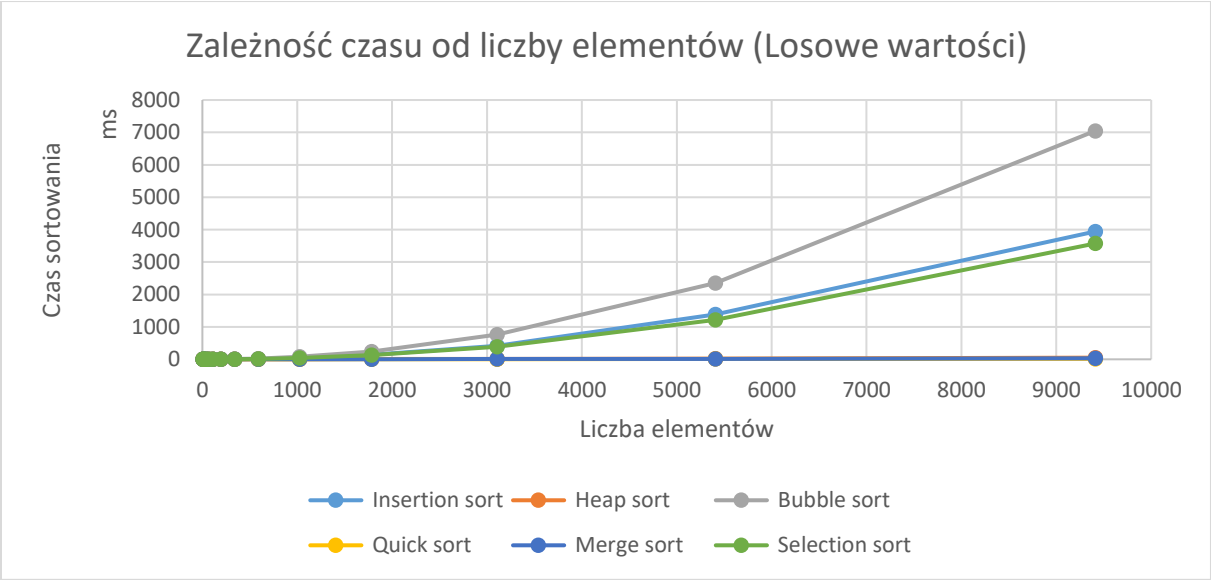
3.1 Tablica rosnąca



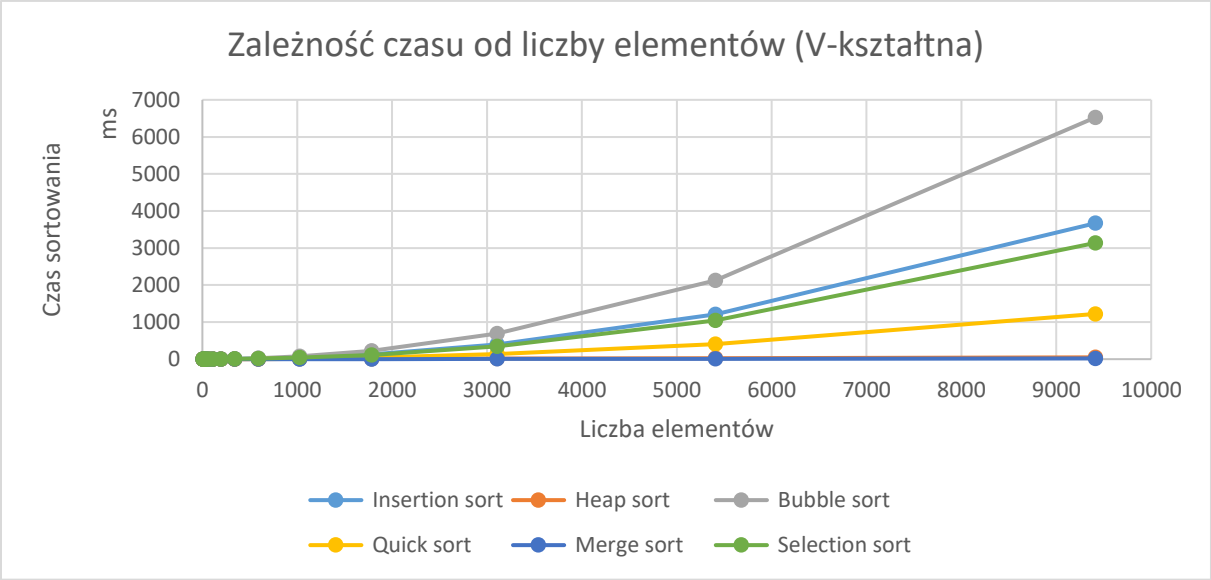
3.2 Tablica malejąca



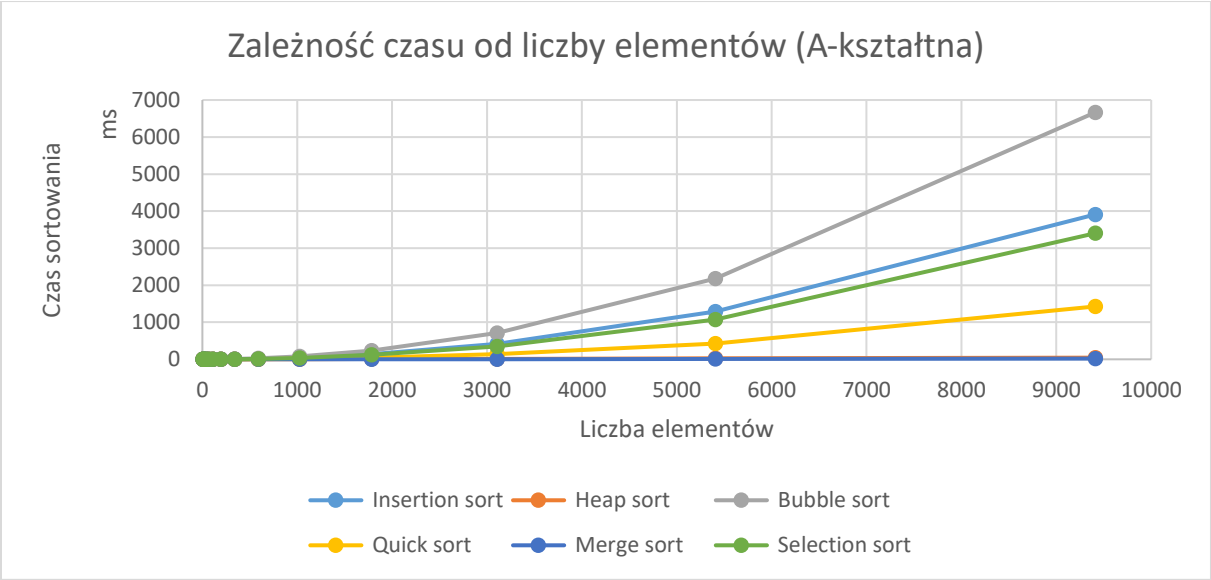
3.3 Losowe wartości



3.4 Dane V-kształtne



3.5 Dane A-kształtne



4 Wnioski

Na podstawie przeprowadzonych testów możemy zauważyć, że algorytmy różnią się czasem wykonywania nawet przy identycznej złożoności czasowej w notacji duże O. Rozpatrując algorytmy o średnim czasie wykonywania O(n log n) warto zauważyć, że wyłącznie wydajność Quick sort uległa degradacji do O(n^2) przy posortowanych podciągach danych. Wydajność Merge oraz Heap sort okazała się zbliżona – Merge sort wykazał dwukrotnie szybszy czas wykonywania, jednak może to być kwestia wyłącznie implementacji. Spośród przedstawionych algorytmów wskazałbym Merge sort jako zwycięzcę – algorytm jest szybki dla każdego rodzaju danych oraz prosty w implementacji.

5 Spis treści