ALGORYTMY I STURKTURY DANYCH

**Specyfikacja poszczególnych algorytmów sortujących:**

1. **MergeSort (Sortowanie przez scalanie)**

**Złożoność:**

Czasowa: średnia, pesymistyczna, optymistyczna: O(n logn)

Pamięciowa: O(n)

**Idea:**

1) Łączenie dwóch posortowanych ciągów w jeden posortowany można wykonać w czasie O(n)

2) Ciągi jednoelementowe są posortowane

Za pomocą techniki „dziel i rządź” dzielimy ciąg na dwa podciągi równe (lub prawie równe), sortujemy każdy (stosując rekurencyjnie ta samą metodę sortowania), łączymy wyniki w ciąg posortowany

**Cechy zaimplementowanego algorytmu:**

- stabilność : TAK

- sortowanie w miejscu: NIE

- łatwy do zrównoleglenia (programowanie współbieżne)

1. **QuickSort (Sortowanie szybkie)**

**Złożoność:**

Czasowa: pesymistyczna: O(n2), średnia, optymistyczna: O(n log n)

Pamięciowa: O(log n) – (potrzebne miejsce na stosie wywołań )

**Idea:** Z tablicy wybiera się element rozdzielający (ang. pivot). Do jednej tablicy wstawia się elementy mniejsze równe pivotowi, do drugiej – większe od pivota. Element rozdzielający wstawiamy pomiędzy tabele i jest to jego ostateczne miejsce. Z każdą tablicą rekurencyjnie robimy to samo. Oczywiście tablice jednoelementowe są posortowane i następuje powrót z rekurencji.

**Cechy zaimplementowanego algorytmu:**

- stabilność: NIE

- sortowanie w miejscu: TAK

- dobrze działa na mniejszych listach/tablicach

- nie jest tak efektywny dla bardzo dużych list/tablic (MergeSort działa wówczas lepiej)

- zastosowane usprawnienie: losowe wybieranie *pivota*

1. **HeapSort (Sortowanie stogowe)**

**Złożoność:**

Czasowa: średnia, pesymistyczna: O(nlog n), optymistyczna: O(n)

Pamięciowa: O(1)

**Idea:**

1) Stworzenie kopca

2) Kopiec (tablica posortowana): zamiana korzenia z ostatnim elementem. Po zamianie ostatni element tablicy nie należy już do kopca. Natomiast trzeba naprawić kopiec od korzenia. Kolejne kroki wyglądają analogicznie.

**Cechy zaimplementowanego algorytmu:**

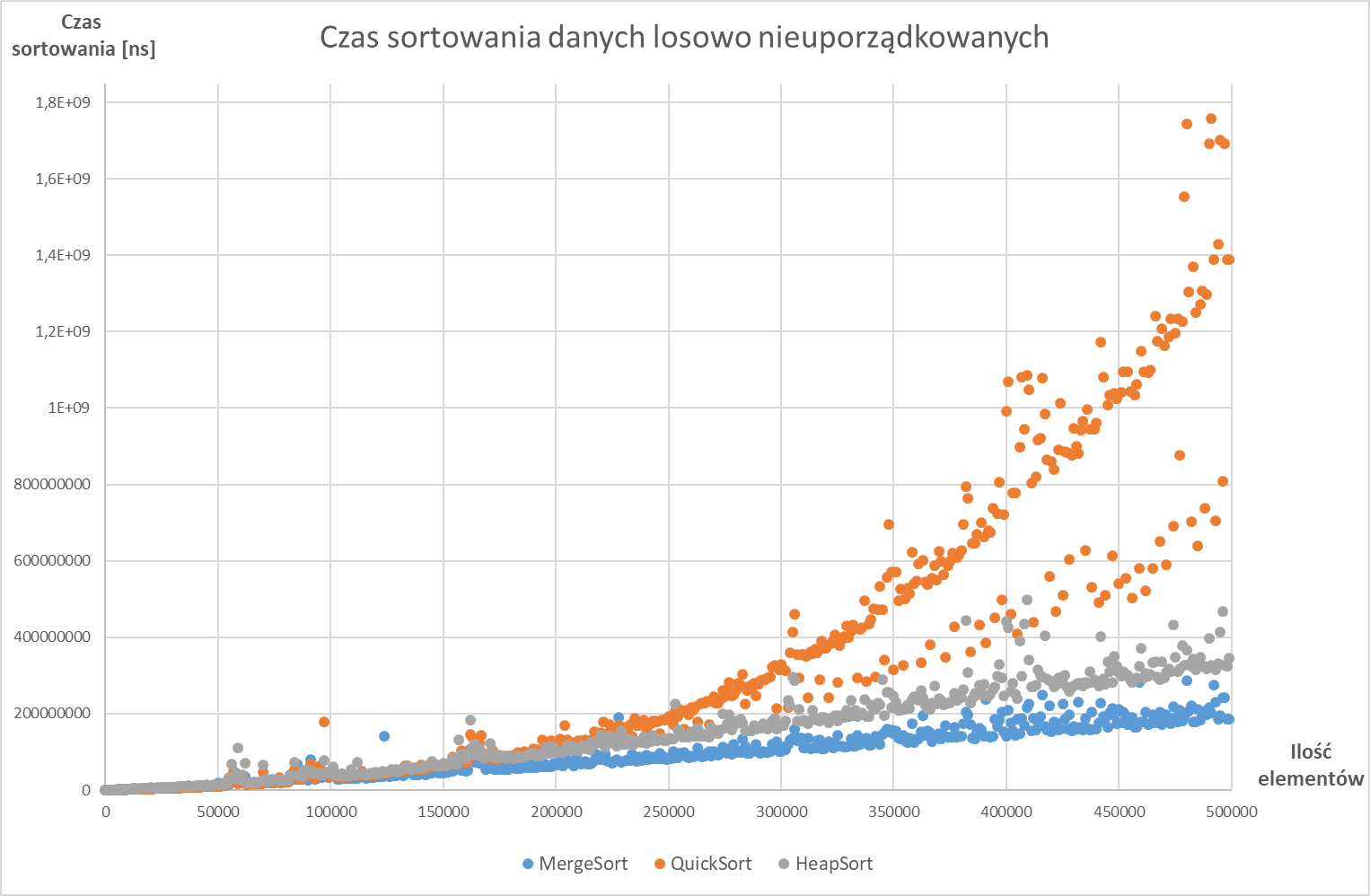
- w miejscu: TAK

- stabilność: NIE

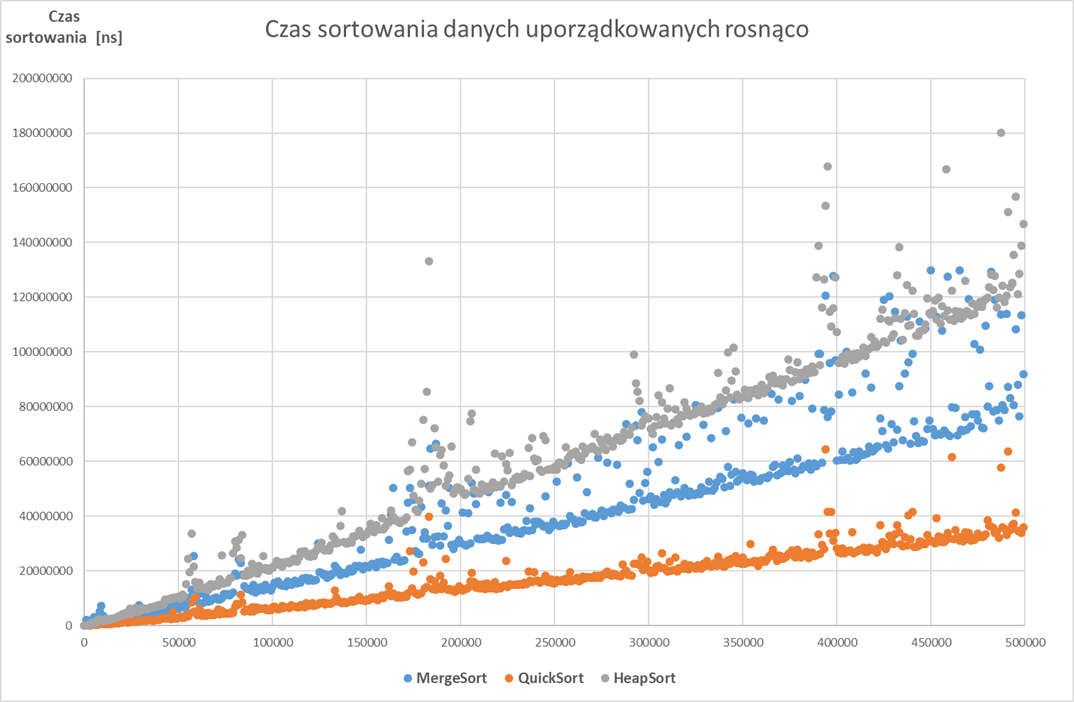
- dla mniejszych list: wolniejszy od quickSorta i MergeSorta

- używany do tworzenia kolejek priorytetowych

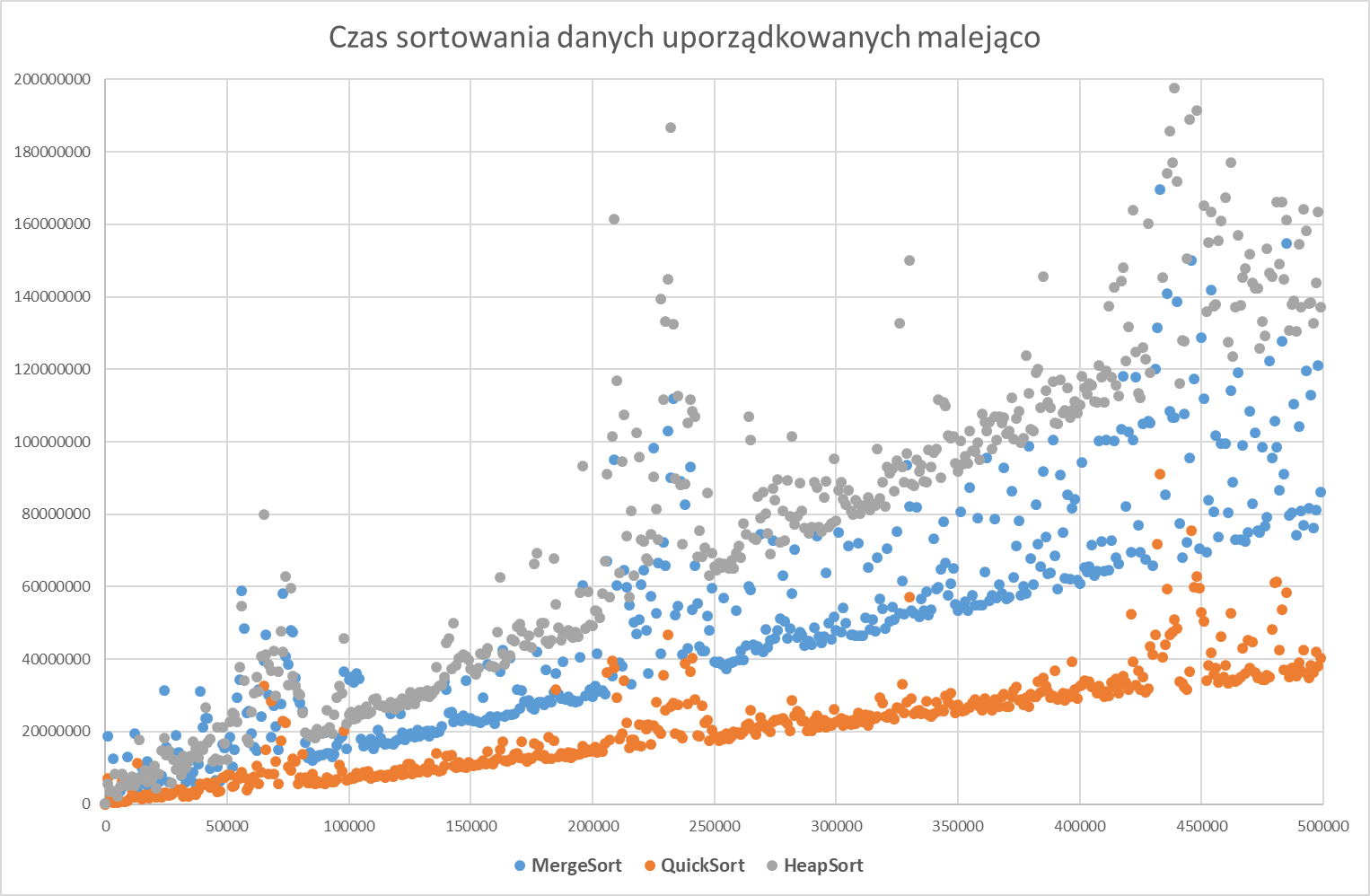
**Porównanie wydajności czasowej algorytmów:**



***Wykres1****: Dane losowe (pseudolosowe)*



***Wykres2:*** *Dane uporządkowane rosnąco (już posortowane)*

**

***Wykres3:*** *Dane uporządkowane malejąco (już posortowane)*

**Tabelaryczne zestawienie wyników sortowania:**





***Tabele1****: Zestawienie czasów sortowania dla poszczególnych ilości elementów*

***Wykresy4****: Porównanie złożoności czasowej dla poszczególnych implementacji sortowania dla jednorazowego przesortowania*

**Uzasadnienie wyników i wnioski:**

Spośród wybranych implementacji sortowania dla bardzo dużych list/tablic, najlepiej radzi sobie Merge*Sort*. Dla mniejszych list/tablic (ale nadal dość znaczących rozmiarów) najlepszym wyborem będzie implementacja algorytmu sortowania quickSort. Pomimo w ogólnym przypadku gorszej nieco wydajności algorytmu sortowania quickSort, jest on uznawany za najlepszy ogólny algorytm sortowania. Wpływają na to następujące aspekty:

1. Wymagane miejsce: mergeSort wymaga dużo miejsca w pamięci w porównaniu do quickSorta
2. Pesymistyczny przypadek quickSorta może zostać wyeliminowany łatwo poprzez użycie losowo wybieranego pivota, albo poprzez medianę z trzech środkowych elementów losowanej listy

Tak jak wcześniej wspomniano, mergeSort znajduje swoje zastosowanie przy sortowaniu naprawdę dużych ilości danych. W pozostałych (nie licząc bardzo małych list) quickSort uzyskuje najlepszą złożoność czasową. HeapSort jest mniej wydajny niż mergeSort czy quickSort i zwykle używa się go raczej do tworzenia kolejek priorytetowych (*priority queue*)