Politechnika Wrocławska

| **Projektowanie Algorytmów i Metody Sztucznej Inteligencji** | |
|---|---|
| **Projekt I** | |
| Imię i Nazwisko: | Jakub Cichy |
| Numer indeksu: | 228976 |
| Termin zajęć: | Wt. 7:30 |
| Prowadzący kurs: | Dr. inż Łukasz Jeleń |

**Wstęp:**

Sortowanie to bardzo często występujący problem w informatyce. Jego celem jest uszeregowanie zbioru danych względem pewnej cechy pojedynczego elementu. Algorytmy sortowania klasyfikuje się według kilku cech. Podstawową cechą jest złożoność, która określa ilość potrzebnych zasobów. Dzielimy ją na dwa rodzaje.

Pierwszym rodzaj to czasowa złożoność obliczeniowa, czyli ilości operacji podstawowych wykonywanych przez algorytm. Przez operację podstawową rozumiemy operacje arytmetyczne, podstawianie czy też porównywanie. Pomiar czasu rzeczywistego jest nieuniwersalną miarą, ponieważ zależy od czynników takich jak użyty kompilator oraz moc obliczeniowa maszyny.

Drugi rodzaj to pamięciowa złożoność obliczeniowa. Jest ona miarą ilości wykorzystanej pamięci, czyli określa liczbę komórek pamięci, która będzie zajmowana w trakcie wykonywania algorytmu.

Złożoność jako cechę rozpatruje się dla przypadków najgorszych – tak zwana złożoność pesymistyczna, oraz złożoność oczekiwana, będąca uśrednieniem możliwych przypadków.

W ramach projektu zaimplementowałem trzy algorytmy, z czego dwa należą do grupy algorytmów niestabilnych. Sortowanie nazywamy stabilnym jeśli gwarantuje ono że elementy o tej samej wartości zachowają swoją kolejność ze zbioru wejściowego. Dosłownie, jeśli $x=y$, oraz $x$ wystąpiło przed $y$ w zbiorze nieposortowanym, to po sortowaniu nadal $x$ będzie znajdowało się przed y. Sortowanie niestabilne może zmienić taką kolejność.

Każdy algorytm został przetestowany na zbiorach o rozmiarach: *10000, 50000, 100000, 500000, 1000000*, oraz w każdym przypadku dla procentowo wyrażonej ilości posortowanych na wstępie elementów: *0%, 25%, 50%, 75%, 95%, 99%, 99,7%*. Każdy z przypadków został wykonany na stu tablicach co daje 3500 różnych tablic, ostateczne wyniki zostały uśrednione. Zastosowałem również funkcję, która po każdym posortowaniu sprawdza czy tablica rzeczywiście jest posortowana, jeśli nie - przerywa działanie programu.

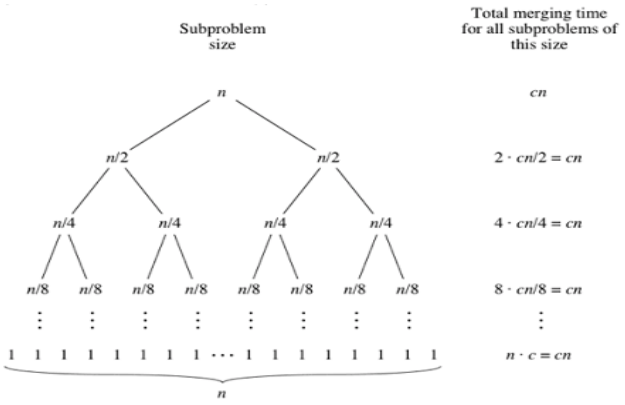
Opis algorytmów:

a) Sortowanie przez scalanie – merge sort.

Jest to algorytm rekurencyjny, stabilny, którego idea polega na dzieleniu n-elementowego zbioru na podzbiory aż do uzyskania n jednoelementowych podzbiorów, a następnie scalanie ich w uporządkowanej kolejności. Istotą jest tutaj procedura scalania. Wymaga ona wykorzystania dodatkowej pamięci, co zwiększa złożoność pamięciową algorytmu – wynosi ona *O(n)*.

Podczas pojedynczej operacji scalania alokujemy pamięć na zbiór wynikowy a następnie porównujemy ze sobą elementy podzbiorów i w zbiorze wynikowym umieszczamy mniejszy z nich. Jeden zbiór może wyczerpać się wcześniej niż drugi, wtedy przepisujemy resztę elementów do zbioru wynikowego.

W wyznaczeniu pesymistycznej złożoności obliczeniowej pomaga przeanalizowanie drzewa binarnego reprezentującego rekurencyjne podziały dokonywane przez nasz algorytm. Bardzo czytelny jest poniższy diagram, który zaczerpnąłem ze strony *Khan Academy*:



Wysokość takiego drzewa binarnego jest równa $\log_2 n$, gdzie n to rozmiar sortowanego zbioru. Widzimy że scalenie każdego poziomu drzewa jest rzędu *O(n)* więc otrzymujemy całkowitą złożoność *O(nlog(n))*. Jest to złożoność również dla najlepszego przypadku ponieważ merge sort zawsze dzieli tablice na podzbiory które potem są scalane w liniowym.

b) Sortowanie szybkie – quick sort

Jest to rekurencyjny, niestabilny algorytm oparty o zasadę „dziel i zwyciężaj”. Sortowanie odbywa się poprzez podział na mniejsze pod problemy. Obywa się to poprzez zamianę elementów w zbiorze tak aby po jeden stronie znajdowały się elementy mniejsze od klucza, tak zwanego pivota, a po drugiej elementy większe. Następnie rekurencyjnie wywołujemy algorytm dla dwóch podzbiorów.

Przy złożoności obliczeniowej przypadek najbardziej pesymistyczny, mamy wtedy, gdy jako klucz wybierzemy największy element tablicy – musimy wykonać wtedy znacznie więcej zamian elementów. Jeżeli wybór takiego klucza powtórzy się w kolejnych wywołaniach to na koniec otrzymamy złożoność $O(n^2)$.

Przypadek optymalny mamy wtedy gdy przy każdym podziale dzielimy zbiór na podzbiory o zbliżonych rozmiarach. Podobnie jak w merge sort drzewo rekurencji ma wysokość $log_2 n$, a każdy poziom wywołania posiada złożoność *O(n)* co daje *O(nlog(n))*.

Złożoność pamięciowa zależy od głębokości rekurencji. W przypadku pesymistycznym jeśli jako klucz zostanie wybrany element największy, głębokość wyniesie n-1, a złożoność obliczeniowa będzie w *O(n)*. Głębokość rekurencji można jednak ograniczyć sprawdzając która część utworzonego podzbioru jest krótsza i ją porządkować najpierw, a dłuższą ponownie dzielić na tym samym poziomie.

c) Sortowanie przez kopcowanie – heap sort

Rekurencyjny, niestabilny algorytm, opierający się na strukturze kopca w którym węzeł nadrzędny jest większy lub równy węzłom potomnym. Jest algorytmem sortującym w miejscu.

W algorytmie na przemian tworzymy kopiec a następnie zamieniamy jego korzeń (czyli element największy) z elementem początkowym, stopniowo tworząc posortowany zbiór.

Zarówno pesymistyczna jak i optymistyczna złożoność obliczeniowa tego algorytmu wynosi *O(nlog(n))*. Jego złożoność pamięciowa jest rzędu *O(1)*.

**Przebieg eksperymentów:**

Napisałem program który wykonuje testy każdego z algorytmu w zadanych przypadkach. Uśrednione wyniki dla 100 sortowań każdego z przypadków testowych zapisywałem do pliku, po czym z agregowałem dane do tabel i wykresów. Każdorazowo program sprawdzał czy tablica jest rzeczywiście posortowana.

**Wyniki:**

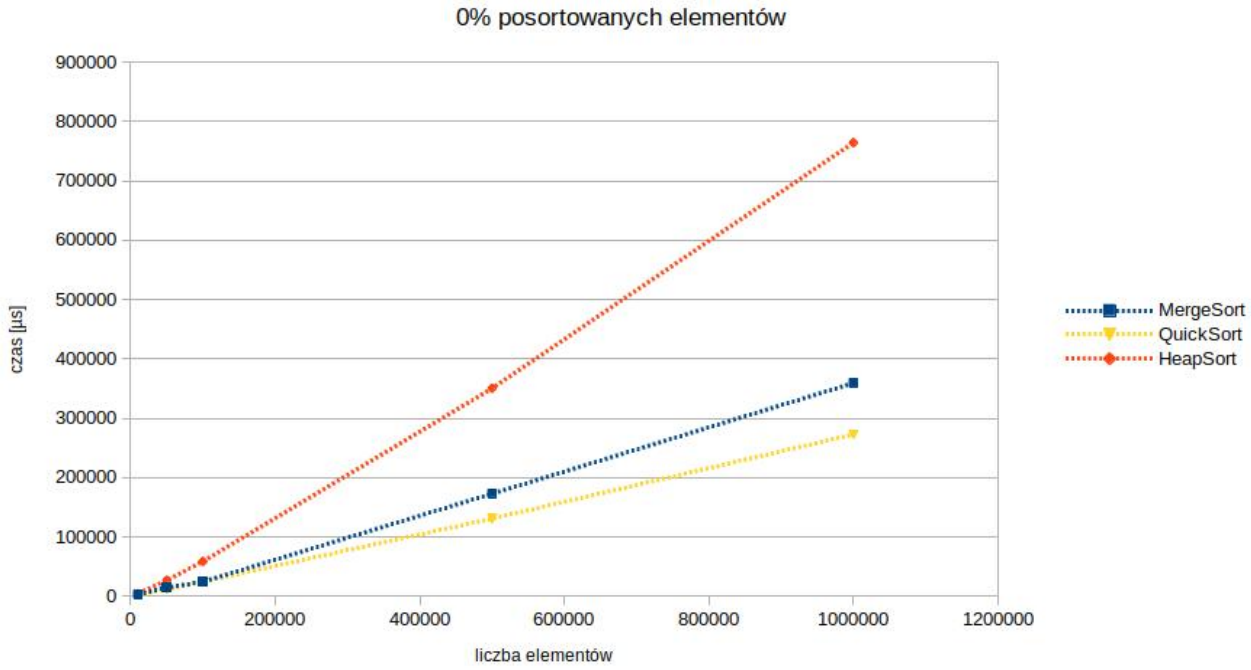
a) Poniższe tabele prezentują uśrednione pomiary czasów dla opisanych algorytmów:

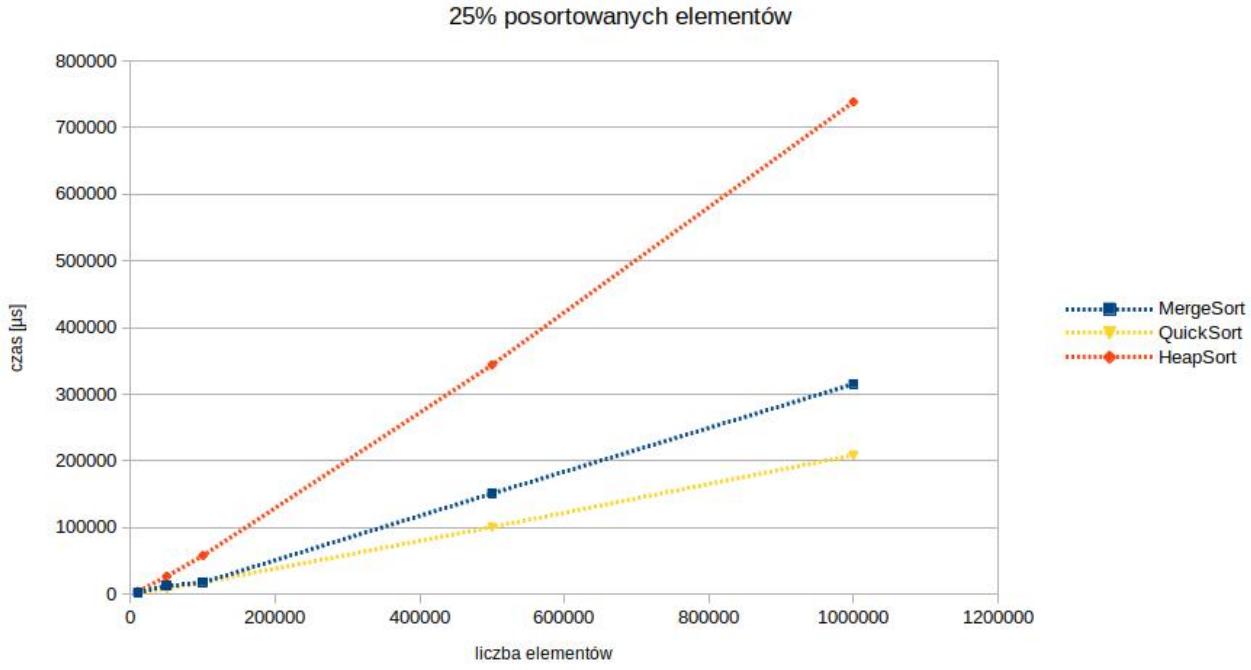
| n | 10000 | | | 50000 | | |
|---|---|---|---|---|---|---|
| | Heap Sort | Quick Sort | Merge Sort | Heap Sort | Quick Sort | Merge Sort |
| posortowane | t [µs] | | | t [µs] | | |
| 0,00% | 4613,73 | 1957,05 | 2661,37 | 27000 | 11441,7 | 14883 |
| 25,00% | 4509,36 | 1463,88 | 2342,35 | 27154,6 | 8646,42 | 13007,9 |
| 50,00% | 4474,66 | 1185,73 | 2169,96 | 26799,8 | 7302,11 | 12048,9 |
| 75,00% | 4329,66 | 853,31 | 1950,82 | 25287,4 | 4923,37 | 10918,5 |
| 95,00% | 4229,12 | 595,39 | 1810,05 | 24514,6 | 3420,78 | 10031,3 |
| 99,00% | 4250,93 | 531,13 | 1778,72 | 24603,8 | 3201,35 | 9925,22 |
| 99,70% | 4177,64 | 529,53 | 1769,95 | 24390,7 | 3101,43 | 9894,85 |

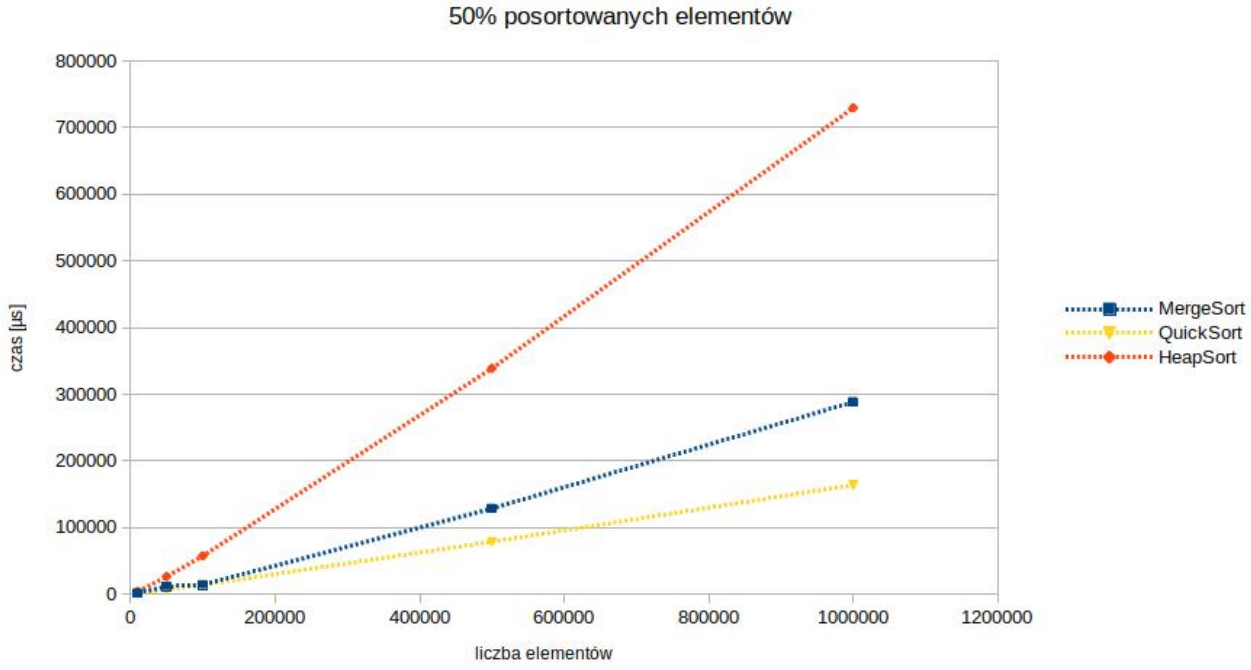
| n | 100000 | | | 500000 | | |
|---|---|---|---|---|---|---|
| | Heap Sort | Quick Sort | Merge Sort | Heap Sort | Quick Sort | Merge Sort |
| posortowane | t [µs] | | | t [µs] | | |
| 0,00% | 58821,9 | 24660,2 | 24660,2 | 350294 | 130894 | 172880 |
| 25,00% | 58240,9 | 17998,4 | 17998,4 | 344242 | 101052 | 151361 |
| 50,00% | 57659,9 | 14361,4 | 14361,4 | 339009 | 79065,7 | 128884 |
| 75,00% | 54872,8 | 10326,7 | 10326,7 | 317573 | 56528,3 | 124930 |
| 95,00% | 52553,3 | 7231,36 | 7231,36 | 300132 | 37753,8 | 113840 |
| 99,00% | 52252,5 | 6600,57 | 6600,57 | 298487 | 34783,7 | 111779 |
| 99,70% | 52036,1 | 6503,95 | 6503,95 | 298568 | 34401 | 111468 |

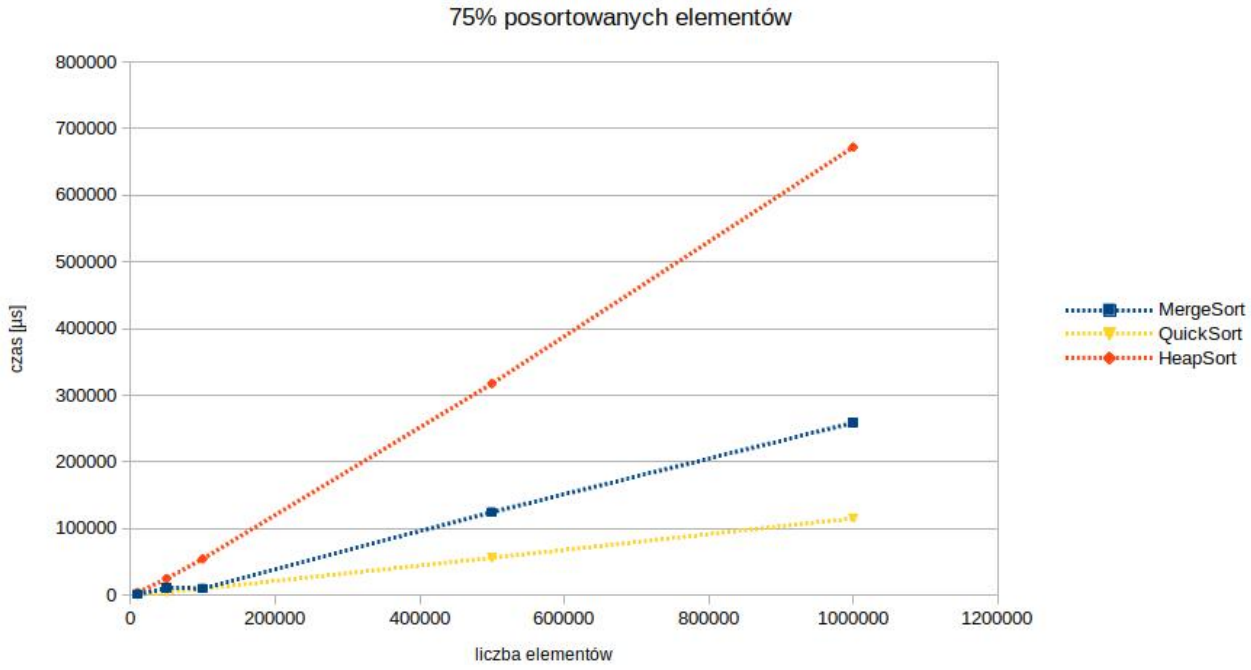
| n | 1000000 | | |
|---|---|---|---|
| | Heap Sort | Quick Sort | Merge Sort |
| posortowane | t [µs] | | |
| 0,00% | 764256 | 272683 | 359293 |
| 25,00% | 738572 | 208842 | 315459 |
| 50,00% | 729613 | 164242 | 288684 |
| 75,00% | 672298 | 115780 | 258886 |
| 95,00% | 634558 | 78493,7 | 235465 |
| 99,00% | 629538 | 71710,1 | 231785 |
| 99,70% | 628032 | 70589,1 | 231520 |

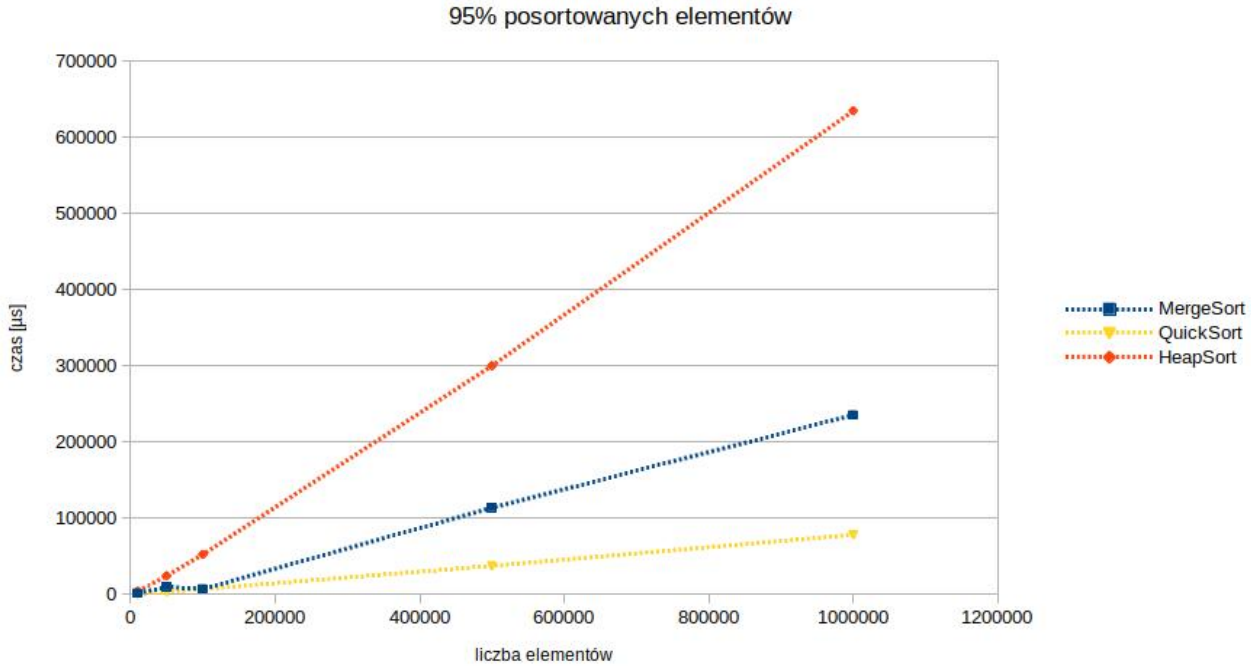
**b) Wykresy:**

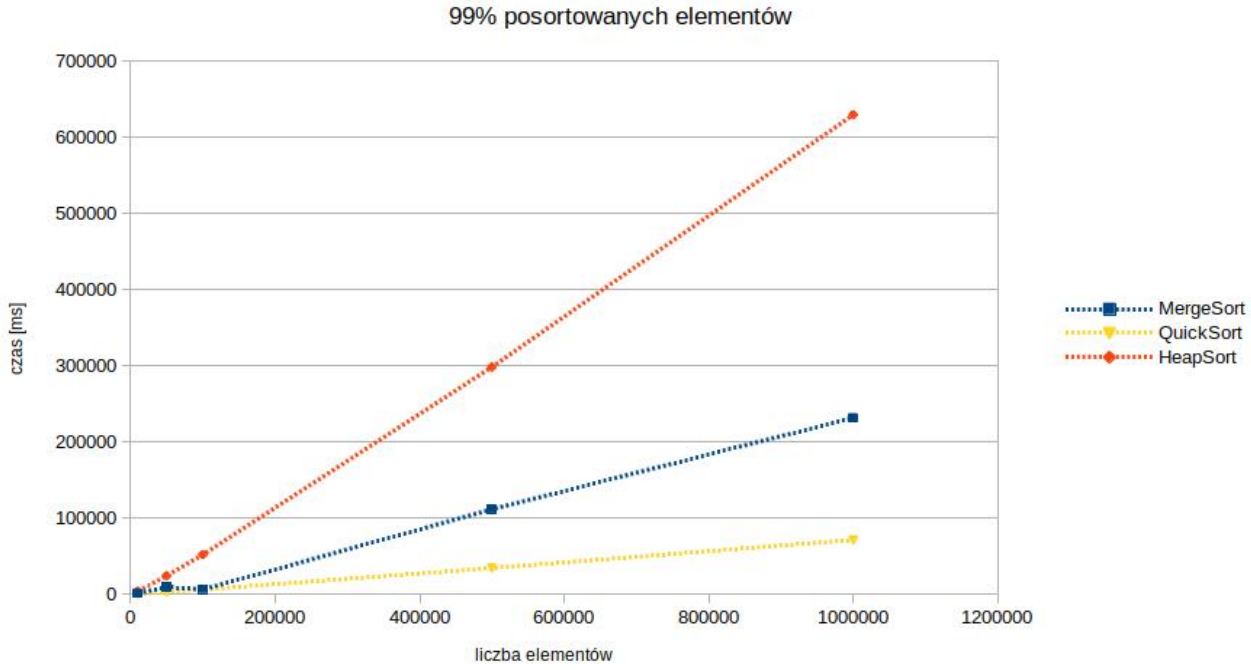

0% posortowanych elementów
900000
800000
700000
600000
500000
400000
300000
200000
100000
0
czas [µs]
0
200000
400000
600000
800000
1000000
1200000
liczba elementów
MergeSort
QuickSort
HeapSort

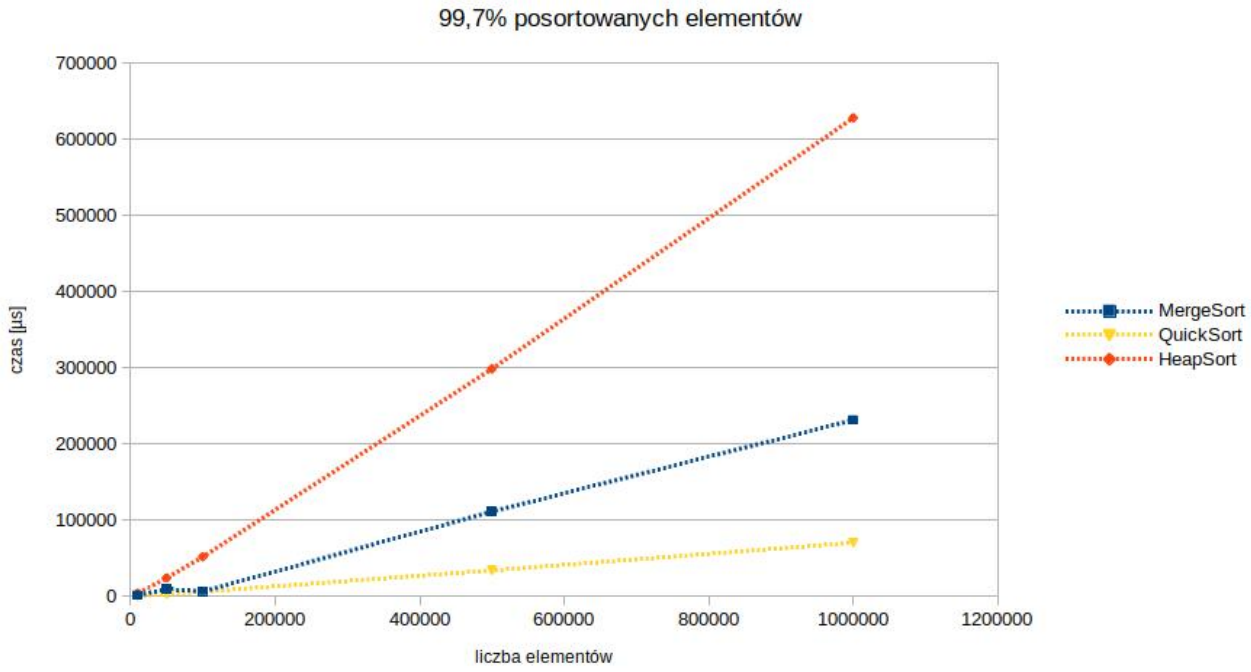

25% posortowanych elementów
800000
700000
600000
500000
400000
300000
200000
100000
0
czas [µs]
0
200000
400000
600000
800000
1000000
1200000
liczba elementów
MergeSort
QuickSort
HeapSort


50% posortowanych elementów
800000
700000
600000
500000
400000
300000
200000
100000
0
czas [µs]
0
200000
400000
600000
800000
1000000
1200000
liczba elementów
MergeSort
QuickSort
HeapSort


75% posortowanych elementów
800000
700000
600000
500000
400000
300000
200000
100000
0
czas [µs]
0
200000
400000
600000
800000
1000000
1200000
liczba elementów
MergeSort
QuickSort
HeapSort


95% posortowanych elementów
700000
600000
500000
400000
300000
200000
100000
0
czas [µs]
0
200000
400000
600000
800000
1000000
1200000
liczba elementów
MergeSort
QuickSort
HeapSort


99% posortowanych elementów
700000
600000
500000
400000
300000
200000
100000
0
czas [ms]
0
200000
400000
600000
800000
1000000
1200000
liczba elementów
MergeSort
QuickSort
HeapSort




**Podsumowanie i wnioski:**

Analizując przebiegi nasuwają się wnioski. Wszystkie przetestowane algorytmy należą do grupy tych wydajniejszych, przez co odczuwalne różnice w ich działaniu są niewielkie. Różnicę widać porównując je np. z Intro sortem. Który już dla tablic o rozmiarach 50000 działa bardzo wolno w porównaniu z powyższymi, o kilka rzędów.

Z wykresów widzimy że quick sort jest najszybszy – to dzięki zastosowanej implementacji, która unika pesymistycznej złożoności obliczeniowej $O(n^2)$ oraz poprawia złożoność pamięciową poprzez ograniczenie rekurencji.

Eksperymenty na algorytmach sortowania w największym stopniu zależą od maszyny na której je wykonujemy oraz implementacji. Moja pierwsza implementacja sortowania szybkiego była bardzo niewydajna, ponieważ jako klucz brałem wartość brzegową dla każdego wywołania, obliczenia trwały bardzo długo, aż ostatecznie przy tablicach o rozmiarze 500000, program się wysypał. Po zmianie implementacji na wersję optymalną sortowanie szybkie okazało się najszybszym z wszystkich trzech.

**Literatura:**


1. https://www.khanacademy.org/computing/computer-science/algorithms/merge-sort/a/analysis-of-merge-sort
2. Jerzy Walaszek, Algorytmy Sortujące, Sortowanie przez scalanie
3. Jerzy Walaszek, Algorytmy Sortujące, Sortowanie szybkie
4. Jerzy Walaszek, Algorytmy Sortujące, Sortowanie stogowe
5. https://softwareengineering.stackexchange.com/questions/297160/why-is-mergesort-olog-n
6. https://www.geeksforgeeks.org/quick-sort/
7. https://www.geeksforgeeks.org/quicksort-using-random-pivoting/
8. https://www.geeksforgeeks.org/heap-sort/
9. Kanał youtube kakaboc, https://www.youtube.com/channel/UCjyj5FeDqH1Du89xJpBLUPA


Mój kod na github: https://github.com/KubaC123/PAMSI/tree/master/SortAlgorithms