

Metody Programowania - Projekt

Gabrysia Niedbała, Nadzeya Kaliada, Konrad Kasza

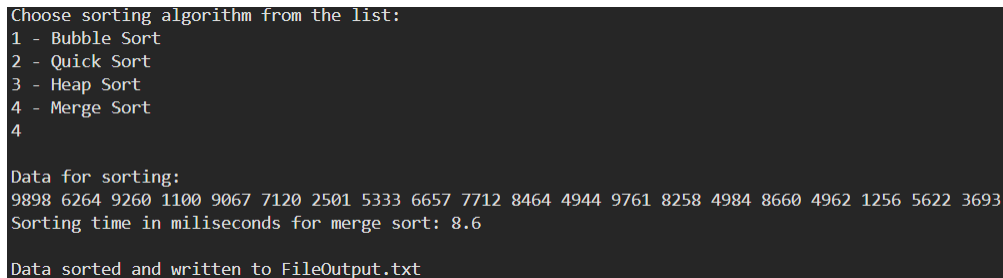
Sortowanie - <https://github.com/gabin0918/Sortowanie>

1 Wprowadzenie

Zadanie polegało na zaprogramowaniu programu umożliwiającego sortowanie danych wejściowych następującymi algorytmami:

- Sortowanie bąbelkowe - bubble sort
- Sortowanie szybkie - quick sort
- Sortowanie przez kopcowanie - heap sort
- Sortowanie przez scalanie - merge sort

Algorytmy zostały zaprogramowane w języku C++, a wspólna praca odbyła się poprzez system kontroli wersji GIT. Jego działanie odbywa się poprzez wprowadzenie rodzaju sortowania w konsoli, otworenie pliku wejściowego, odczytanie danych do tabeli i wykonanie sortowania wraz z zapisaniem danych wyjściowych do pliku `FileOutput.txt`.



```
Choose sorting algorithm from the list:
1 - Bubble Sort
2 - Quick Sort
3 - Heap Sort
4 - Merge Sort
4

Data for sorting:
9898 6264 9260 1100 9067 7120 2501 5333 6657 7712 8464 4944 9761 8258 4984 8660 4962 1256 5622 3693
Sorting time in milliseconds for merge sort: 8.6

Data sorted and written to FileOutput.txt
```

Zrzut ekranu 1 - przykład działania programu

Następnie ćwiczenie polegało na przeanalizowaniu działania programu w następujący sposób:

- Poprawność działania
 - Wykonanie unit testów w celu pokazania poprawności działania
 - Rozważenie wszystkich scenariuszy, typu: tablica pusta, posortowana, odwrotnie posortowana, pojedynczy element
 - Unit testy wykonaliśmy za pomocą biblioteki `Google test`
- Szybkość wykonywania operacji
 - Porównanie otrzymanych czasów wykonywania do oczekiwanych wartości wynikających z obliczonych złożoności czasowych
 - Wykonanie wykresów obrazujących czasy wykonywania w funkcji rozmiaru tablicy dla przypadków: średni, optymistyczny, pesymistyczny
- Wnioski dotyczące opracowanych wyników

2 Wykonanie kodu

BubbleSort

Funkcja sortowania bąbelkowego (bubble sort) działa na zasadzie iteracyjnego przechodzenia przez tablicę i porównywania sąsiednich elementów. W każdej iteracji sprawdzane są pary sąsiednich elementów, a jeśli są one w złej kolejności (względem porządku rosnącego lub malejącego, zależnie od parametru), są one zamieniane miejscami. Proces ten powtarza się aż do momentu, gdy tablica jest posortowana. Optymalizacja polega na zatrzymaniu algorytmu, jeśli w jednej iteracji nie wykonano żadnej zamiany elementów, co oznacza, że tablica jest już posortowana.

QuickSort

Algorytm sortowania szybki został zaimplementowany rekurencyjnie. Najpierw sprawdzana jest długość podtablicy, aby zakończyć działanie, jeśli jest ona mniejsza lub równa jednemu. Następnie wybierany jest `pivot`, a elementy tablicy są przestawiane tak, aby te mniejsze (lub większe, zależnie od kierunku sortowania) znajdowały się po jednej stronie `pivota`, a większe (lub mniejsze) po drugiej. Po dokonaniu przestawień tablica jest dzielona na dwie części, które są sortowane niezależnie za pomocą rekurencji. Sortowanie może być wykonane rosnąco lub malejąco, w zależności od wartości parametru sterującego.

HeapSort

Algorytm sortowania przez kopcowanie jest realizowany w dwóch głównych fazach. W pierwszej fazie budowany jest kopiec z tablicy wejściowej poprzez wywołanie funkcji `heapify` dla odpowiednich węzłów kopca. Następnie w drugiej fazie elementy kopca są sukcesywnie usuwane, a na ich miejsce przesuwany jest ostatni element kopca, przy jednoczesnym przywracaniu właściwości kopca za pomocą funkcji `heapify`. Proces ten powtarzany jest aż do posortowania całej tablicy. Odpowiednie zachowanie funkcji `heapify`, zależne od parametru `isIncreasing`, pozwala na sortowanie rosnące lub malejące.

MergeSort

Algorytm sortowania przez scalanie polega na podziale tablicy na mniejsze części, a następnie scalaniu ich w odpowiedniej kolejności. W pierwszej fazie funkcja `mergeSort` dzieli tablicę na dwie części, aż do momentu, gdy każda z części zawiera tylko jeden element lub brak elementów. Następnie funkcja `merge` łączy te części w jedną tablicę, przy jednoczesnym zachowaniu porządku elementów, zgodnie z parametrem `isIncreasing`. Proces ten powtarzany jest rekurencyjnie aż do momentu, gdy cała tablica jest posortowana.

3 Poprawność działania

Przetestowaliśmy wynik sortowania dla tabeli wyjściowej i porównaliśmy ją z oczekiwaną tabelą wyjściową za pomocą następujących testów, na przykładzie testów sortowania bąbelkowego. Poniżej zestawienia wyjściowych tabeli i oczekiwanych, które zostały użyte w tych testach:

BubbleSort

```
1 int arr [] = { 5, 3, 8, 4, 2 };
2 int expected [] = { 2, 3, 4, 5, 8 };
```

BubbleSortEmpty

```
1 int* arr = nullptr;  
2 EXPECT_EQ(arr, nullptr);
```

BubbleSortSingleElement

```
1 int arr[] = { 7 };  
2 int expected[] = { 7 };
```

BubbleSortAlreadySorted

```
1 int arr[] = { 8, 9, 11, 16, 23 };  
2 int expected[] = { 8, 9, 11, 16, 23 };
```

BubbleSortReversed

```
1 int arr[] = { 19, 12, 6, 2, 1 };  
2 int expected[] = { 1, 2, 6, 12, 19 };
```

Wyniki unit testów

SortingTest (20)	1 ms
BubbleSort	< 1 ms
BubbleSortAlreadySorted	< 1 ms
BubbleSortEmpty	< 1 ms
BubbleSortReversed	< 1 ms
BubbleSortSingleElement	< 1 ms
HeapSort	< 1 ms
HeapSortAlreadySorted	< 1 ms
HeapSortEmpty	< 1 ms
HeapSortReversed	< 1 ms
HeapSortSingleElement	< 1 ms
MergeSortAlreadySorted	< 1 ms
MergeSortEmpty	< 1 ms
MergeSortReversed	< 1 ms
MergeSortSingleElement	< 1 ms
QuickSort	< 1 ms
QuickSortAlreadySorted	< 1 ms
QuickSortEmpty	< 1 ms
QuickSortReversed	< 1 ms
QuickSortSingleElement	< 1 ms
MergeSort	1 ms

Wszystkie testy zakończyły się pozytywnie, co oznacza, że algorytmy działają zgodnie z oczekiwaniami i poprawnie.

4 Pomiar czasu i wnioski do ćwiczenia

Bardzo ważną cechą algorytmu jest jego efektywność czasowa. Wraz ze wzrostem liczby elementów tablicy, czas się wydłuża, a zależność tę określamy za pomocą notacji "Big O". Dla każdego algorytmu znaleźliśmy złożoność czasową:

Bąbelkowe sortowanie

Zagnieżdżone pętle powodują w najgorszym przypadku złożoność $O(n^2)$. W przeciętnym przypadku algorytm będzie również musiał wykonać wiele porównań i zamian, co sprawia, że jego złożoność czasowa jest nadal kwadratowa. W najlepszym, gdy tablica jest posortowana, algorytm jedynie przechodzi przez tablicę, dzięki czemu otrzymujemy $O(n)$.

Szybkie sortowanie

Algorytm ma średnią złożoność czasową $O(n \log n)$ ze względu na podział tablicy i rekurencyjne wywołania dla podtablic. W najgorszym przypadku może obniżyć się do $O(n^2)$ przy złym wyborze pivotu, ale średnio działa z złożonością $O(n \log n)$.

Kopcowe sortowanie

Funkcja heapify ma złożoność czasową $O(\log n)$ i jest wywoływana n razy w funkcji heapSort. W związku z tym całkowita złożoność czasowa algorytmu wynosi $O(n \log n)$.

Sortowanie przez scalanie

Algorytm sortowania przez scalanie ma złożoność czasową $O(n \log n)$ zarówno w średnim, jak i w najgorszym przypadku. Funkcja scalania łączy dwie posortowane podtablice w czasie liniowym, a funkcja mergeSort rekurencyjnie dzieli tablicę na połowy, aż do osiągnięcia podtablic jednoelementowych, co skutkuje logarytmiczną liczbą poziomów w drzewie rekurencji.

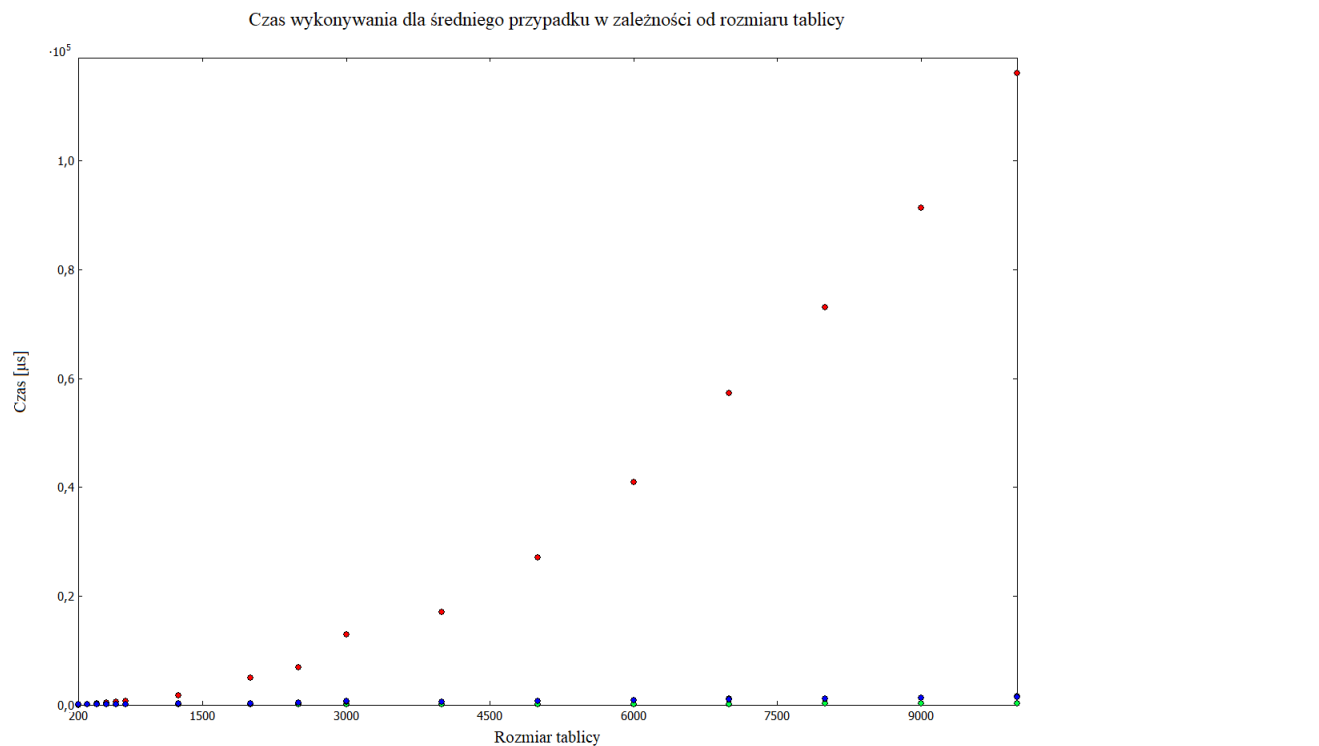
Podsumowując:

	Średni przypadek	Optymistyczny	Pesymistyczny
<i>BubbleSort</i>	$O(n^2)$	$O(n)$	$O(n^2)$
<i>QuickSort</i>	$O(n \cdot \log(n))$	$O(n \cdot \log(n))$	$O(n^2)$
<i>HeapSort</i>	$O(n \cdot \log(n))$	$O(n \cdot \log(n))$	$O(n \cdot \log(n))$
<i>MergeSort</i>	$O(n \cdot \log(n))$	$O(n \cdot \log(n))$	$O(n \cdot \log(n))$

Tabela 1 - złożoności czasowe algorytmów

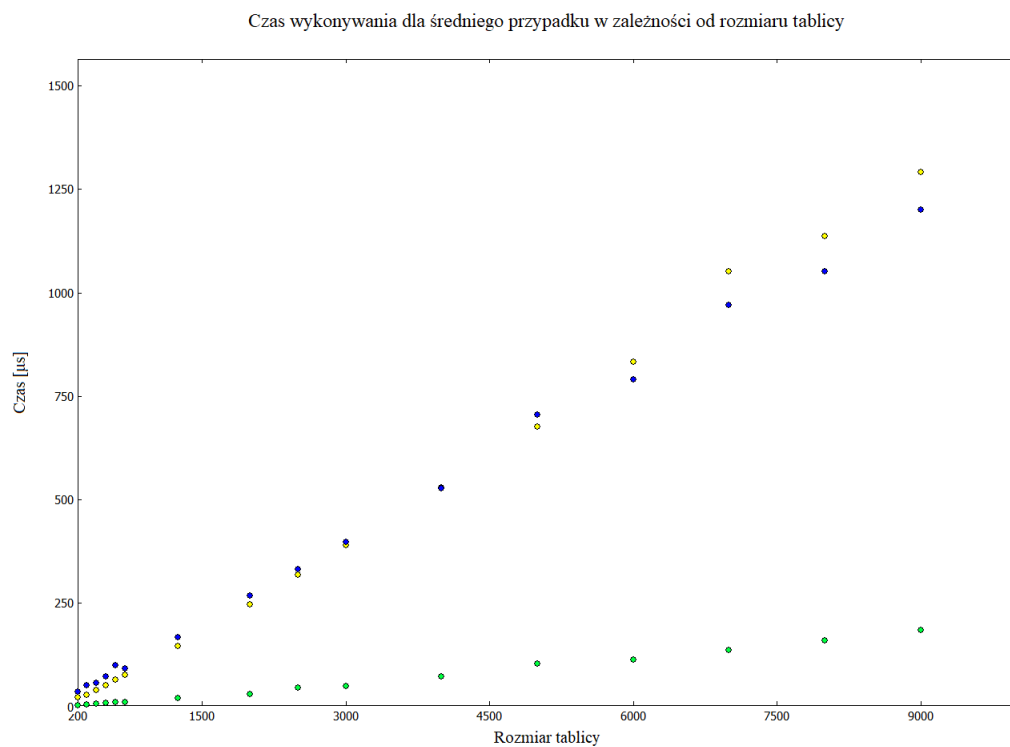
Jednakże, aby zreprodukować najgorszy przypadek dla algorytmu quicksort, należy zapewnić, że pivot w każdym kroku rekurencyjnym wybierany jest w taki sposób, że dzieli tablicę na bardzo nierówne części, co skutkuje maksymalną głębokością rekurencji. Najgorszy przypadek występuje, gdy pivot jest zawsze najniższym lub najwyższym elementem tablicy. W naszym algorytmie, pivot jest wybierany jako $[(n-1)/2]$ element tablicy, co mocno redukuje podatność na ten przypadek i utrudnia znalezienie przykładu obrazującego zmiany w złożoności czasowej dla tablicy o dużym rozmiarze, stąd nie byliśmy w stanie tego dokonać. Tablica o małym rozmiarze nie jest odpowiednia do porównywania czasu wykonywania, ponieważ przy czasie rzędu kilkudziesięciu mikrosekund istnieje wiele czynników niezależnych od nas, które mogą ten czas mocno zaburzyć. Zauważyliśmy to, sprawdzając kilkakrotnie działanie algorytmów w tych samych warunkach, zdarzały się przypadki, gdzie sortowanie tej samej tabeli w dwóch różnych próbach dawało wyniki różniące się o kilkanaście procent, stąd warto w niektórych przypadkach czas wykonywania obliczać jako uśrednioną wartość z kilku pomiarów.

Sprawdziliśmy, jak wygląda czas wykonywania dla różnych rozmiarów tablic, jako wartości używając losowo wygenerowane liczby, w zakresie od 0 do 10 000. Z pomiarów zrobiliśmy wykres, na którym oznaczyliśmy kolorem czerwonym sortowanie **bąbelkowe**, zielonym - **szybkie**, zaś żółtym i niebieskim - kolejno sortowanie **przez kopcowanie** i **poprzez scalanie**.



Wykres 1 - porównanie wszystkich algorytmów sortowania, widoczna ogromna nieefektywność bubble sort.

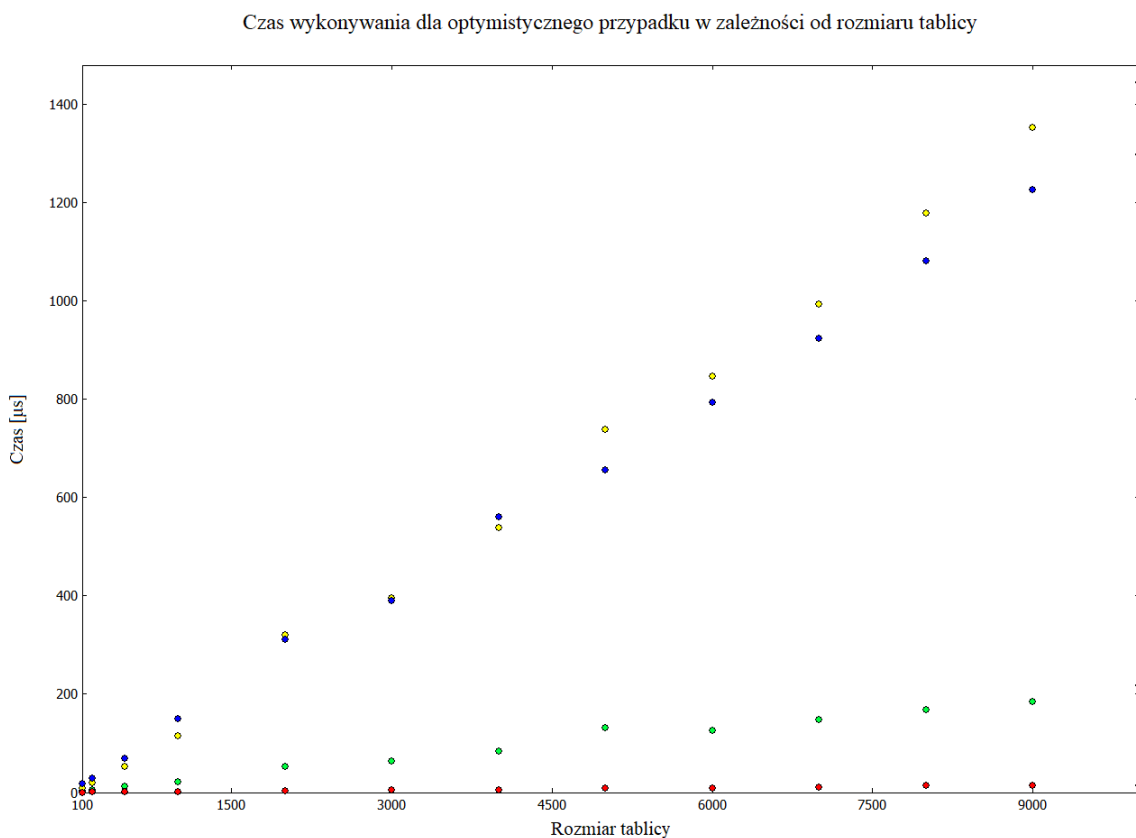
Ponieważ sortowanie bąbelkowe okazało się zbyt nieefektywne czasowo, by porównywać na wykresie z pozostałymi algorytmami, poniżej znajduje się wykres z wykluczeniem tego czasochłonnego sortowania:



Oczywistym wnioskiem jest to, że sortowanie szybkie osiągnęło najlepszy czas, znacząco przewyższając efektywnością pozostałe algorytmy. Mergesort i heapsort są bardzo podobne w tej kwestii - oba cechują się zależnością logarytmiczno-liniową. Efektywność quicksorta możemy uzasadnić ze względu na fakt, że quicksort jest algorytmem "in place", co oznacza, że działa na oryginalnej tablicy danych bez potrzeby dodatkowej pamięci. W przeciwieństwie do tego, mergesort i heapsort wymagają dodatkowej pamięci, co może wpływać na ich wydajność w sytuacjach, gdy pamięć jest ograniczona lub gdy mamy do czynienia z dużymi zestawami danych. Quicksort wykorzystuje lokalność danych, co oznacza, że ma tendencję do lepszego wykorzystania pamięci podręcznej procesora. Dzięki temu operacje odczytu/zapisu na tablicy danych są bardziej wydajne w quicksort niż w mergesort lub heapsort, co może przyczynić się do lepszej wydajności.

Wybór najlepszego algorytmu sortowania może również zależeć od charakterystyki danych. Dla pewnych typów danych quicksort może być bardziej skuteczny, podczas gdy dla innych mergesort lub heapsort może być lepszym wyborem. Na przykład mergesort jest stabilny i działa równie dobrze na danych o różnej charakterystyce, podczas gdy quicksort może mieć gorszą wydajność w przypadku danych już wstępnie posortowanych.

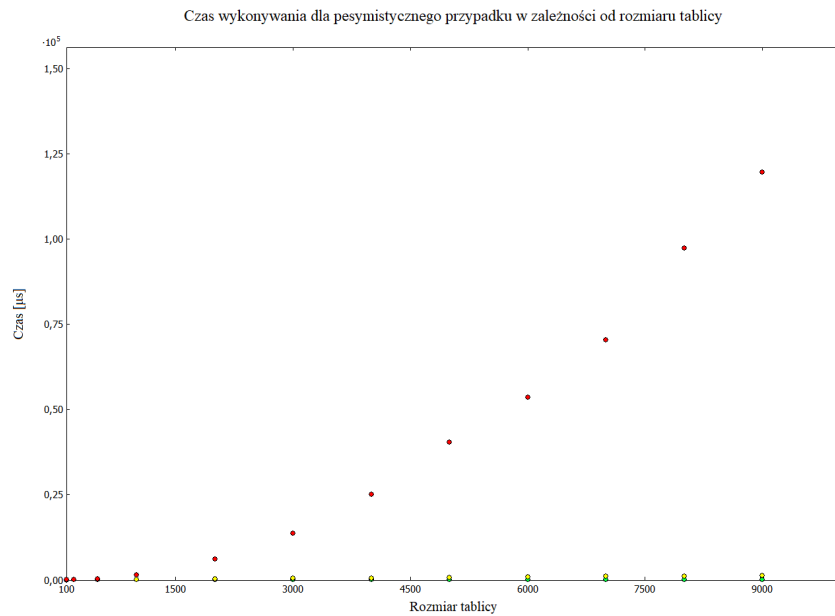
Sprawdziliśmy przypadek, gdzie tablica jest posortowana i sporządziliśmy analogiczny wykres:



Wykres 3 - przypadek, gdzie tablica była **posortowana** przed rozpoczęciem algorytmu

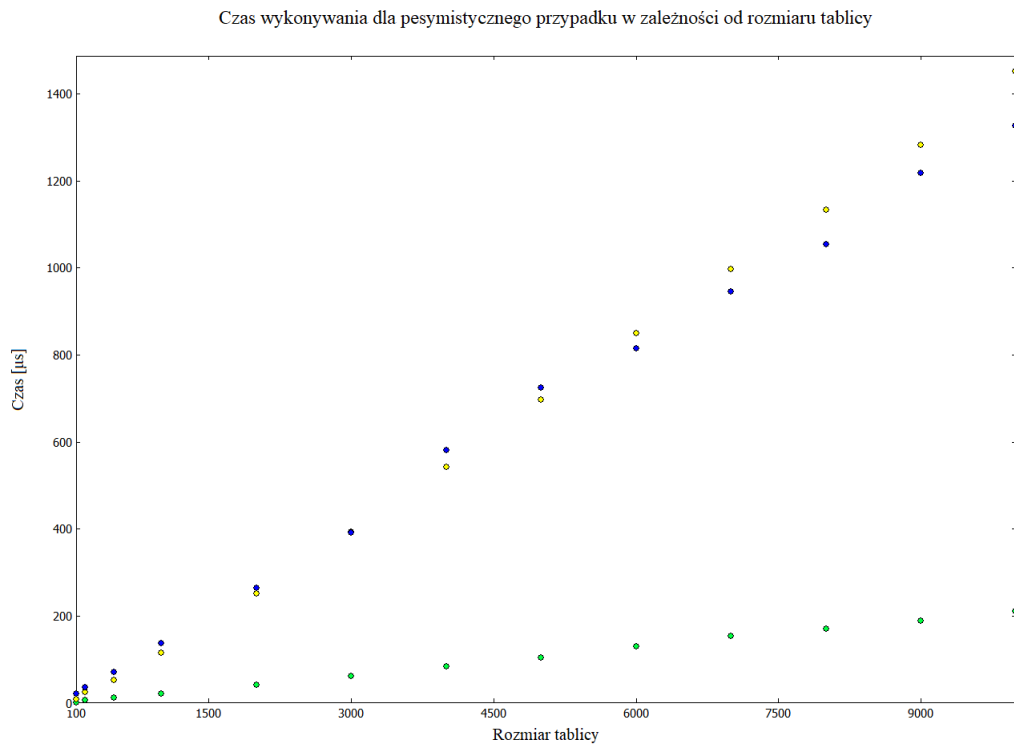
Sortowanie bąbelkowe było bardzo szybkie - oczekiwaliśmy tego, ponieważ nie wymagało ono żadnych dodatkowych procedur oprócz porównania wszystkich kolejnych elementów tablicy. Po iteracji przez tablicę, zwracana była bez żadnych dodatkowo obciążających czasowo operacji. Ponownie, mergesort i heapsort wykazały zbliżoną zależność i są kilkakrotnie mniej efektywne od quicksorta. W obu przypadkach operacje sortowania nie korzystają z lokalności danych w sposób optymalny. Mergesort działa na zasadzie dzielenia i łączenia list, podczas gdy heapsort przeprowadza operacje na kopcu, co może prowadzić do bardziej losowego dostępu do pamięci, szczególnie w przypadku dużych zestawów danych.

Ponownie przetestowaliśmy sortowanie dla tabeli posortowanej **odwrotnie** - to znaczy, że element ostatni jest najmniejszy dla sortowania niemalejącego, bądź podobnie pierwszy jest największy dla sortowania nierosnącego. Całość tablicy jest posortowana w przeciwny sposób, niż pożądany.



Sortowanie dla tablicy odwrotnie posortowanej

Ponownie, sortowanie bąbelkowe swoim czasem wykonywania ogromnie przewyższa pozostałe algorytmy i jest nieporównywalnie mniej efektywnie - względem pozostałych algorytmów czas jego wykonywania wydaje się rosnąć eksponencjalnie. Sprawdziliśmy wykres wykluczający bubblesorta:



Wykres wygląda podobnie do poprzedniego - to znaczy - quicksort efektywniejszy od pozostałych (kilkukrotnie), a mergesort i heapsort bardzo podobne względem siebie. Sam czas wykonywania również jest bardzo podobny, co sugeruje, że niezależnie od stanu tablicy - ich efektywność pozostaje taka sama. Jest to zgodne z tabelką 1, gdzie w każdym przypadku złożoność wynosi $O(n \log n)$.

Sam quicksort rzeczywiście wykazuje zależność logarytmiczno-liniową i jest ona w każdym przetestowanym przypadku korzystniejsza czasowo od pozostałych algorytmów (za wyjątkiem bubble sorta przy posortowanej tablicy). Z naszych doświadczeń wynika, że ten algorytm najlepiej się sprawdził do sortowania i wybranie środkowego elementu jako pivota zredukowało podatność na znaczące spowolnienie przy przypadkach tablicy posortowanej malejąco lub rosnąco. Ćwiczenie ponadto nauczyło nas korzystać z biblioteki Google test, która pozwoliła prosto i efektywnie przeprowadzić unit testy w celu sprawdzenia poprawności działania algorytmów. Wykazaliśmy empirycznie, że optymalizowanie algorytmów prowadzi do znacznej poprawy działania, w szczególności porównując pozostałe algorytmy do prostego sortowania bąbelkowego.