|  |
| --- |
| WSEI |
| Efektywność Algorytmów |
| Jakub Hojda – 14492 - lab1/2/GRYS |

# Opis Eksperymentu

Przeanalizowane zostaną cztery algorytmy sortowania:

* **Insertion Sort** – analizuje kolejne elementy tablicy i cofa je o jedną pozycję, dopóki nie znajdą się w odpowiedniej kolejności, względem przeanalizowanych już elementów;
* **Merge Sort** – dzieli tablicę na połowę i rekurencyjnie ją sortuje, po czym scala podtablice w całość;
* **Quick Sort Classical** – sortowanie w oparciu o punkty odniesienia (Pivoty), gdzie tablice są rekurencyjnie dzielone i sortowane wokół losowo wybieranych pivotów;
* **Quick Sort** – tak jak QSC, jednak pivoty nie są wybierane losowo, a na podstawie mediany z kilku elementów;

# Opis Implementacji

Napisany został kod C#, zawierający statyczne funkcje generujące 5 rodzajów tablic – losowo posortowaną, posortowaną, posortowaną od tyłu, prawie posortowaną oraz losowo posortowaną z wielokrotnie powtarzającymi się wartościami.

Generowanie tablic zasymulowano w trzech skalach: 10 elementów (mała), 1 000 elementów (średnia) oraz 100 000 elementów (duża).

Przed rozpoczęciem pomiarów, uruchamia się funkcja przygotowująca tablice do testów w określonym rozmiarze – dopiero po wygenerowaniu danych uruchamia się Benchmark, który sklonowawszy wcześniej przygotowaną tablicę, sortuje ją.

Funkcje odpowiedzialne za generowanie tablic i sortowanie umieszczone są w publicznych klasach statycznych nazwanych kolejno ‘ArrayGenerator’ i ‘SortingAlgorithms’. Wszystkie Benchmarki, Setup oraz zmienne związane z testami zgrupowane są w publicznej klasie ‘Benchmark’, wywoływanej w funkcji ‘Main’.

# Wyniki Pomiarów

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Method** | **ArrSize** | **Mean** | **Error** | **StdDev** | **Median** |
| InsertionSort\_Random | 10 | 93.67 ns | 1.761 ns | 2.028 ns | 93.07 ns |
| InsertionSort\_Sorted | 10 | 74.32 ns | 0.610 ns | 0.509 ns | 74.10 ns |
| InsertionSort\_Reversed | 10 | 81.75 ns | 0.784 ns | 0.655 ns | 81.69 ns |
| InsertionSort\_Almost | 10 | 74.04 ns | 0.542 ns | 0.453 ns | 73.97 ns |
| InsertionSort\_FewUnique | 10 | 82.27 ns | 1.384 ns | 1.156 ns | 81.93 ns |
| MergeSort\_Random | 10 | 182.02 ns | 2.835 ns | 2.368 ns | 181.79 ns |
| MergeSort\_Sorted | 10 | 181.55 ns | 1.755 ns | 1.642 ns | 181.52 ns |
| MergeSort\_Reversed | 10 | 185.25 ns | 2.084 ns | 1.740 ns | 185.80 ns |
| MergeSort\_Almost | 10 | 179.77 ns | 2.841 ns | 2.657 ns | 178.98 ns |
| MergeSort\_FewUnique | 10 | 180.28 ns | 1.170 ns | 1.037 ns | 180.62 ns |
| QuickClassicalSort\_Random | 10 | 116.31 ns | 1.481 ns | 2.076 ns | 115.62 ns |
| QuickClassicalSort\_Sorted | 10 | 126.01 ns | 1.257 ns | 1.114 ns | 126.22 ns |
| QuickClassicalSort\_Reversed | 10 | 109.15 ns | 2.330 ns | 6.494 ns | 108.41 ns |
| QuickClassicalSort\_Almost | 10 | 127.02 ns | 3.218 ns | 9.337 ns | 125.44 ns |
| QuickClassicalSort\_FewUnique | 10 | 121.09 ns | 3.425 ns | 9.827 ns | 118.59 ns |
| QuickSort\_Random | 10 | 98.99 ns | 2.232 ns | 6.403 ns | 98.05 ns |
| QuickSort\_Sorted | 10 | 90.68 ns | 5.174 ns | 14.594 ns | 84.94 ns |
| QuickSort\_Reversed | 10 | 104.73 ns | 3.117 ns | 8.792 ns | 102.49 ns |
| QuickSort\_Almost | 10 | 91.42 ns | 2.737 ns | 7.984 ns | 90.11 ns |
| QuickSort\_FewUnique | 10 | 87.90 ns | 2.280 ns | 6.688 ns | 86.95 ns |
| InsertionSort\_Random | 1000 | 189,280.41 ns | 3,437.103 ns | 7,098.213 ns | 188,823.36 ns |
| InsertionSort\_Sorted | 1000 | 1,327.40 ns | 27.493 ns | 77.543 ns | 1,288.65 ns |
| InsertionSort\_Reversed | 1000 | 182,560.66 ns | 3,133.671 ns | 4,785.446 ns | 180,997.67 ns |
| InsertionSort\_Almost | 1000 | 28,743.47 ns | 564.869 ns | 627.850 ns | 28,658.73 ns |
| InsertionSort\_FewUnique | 1000 | 196,906.72 ns | 3,889.134 ns | 10,245.535 ns | 193,985.18 ns |
| MergeSort\_Random | 1000 | 55,910.61 ns | 1,113.481 ns | 2,813.909 ns | 55,673.60 ns |
| MergeSort\_Sorted | 1000 | 32,272.87 ns | 642.693 ns | 1,726.554 ns | 32,014.31 ns |
| MergeSort\_Reversed | 1000 | 55,347.90 ns | 1,024.922 ns | 2,475.303 ns | 54,208.04 ns |
| MergeSort\_Almost | 1000 | 31,426.89 ns | 414.633 ns | 346.237 ns | 31,506.17 ns |
| MergeSort\_FewUnique | 1000 | 55,599.61 ns | 1,277.831 ns | 3,707.219 ns | 54,383.33 ns |
| QuickClassicalSort\_Random | 1000 | 21,692.53 ns | 430.590 ns | 1,080.266 ns | 21,474.51 ns |
| QuickClassicalSort\_Sorted | 1000 | 272,135.89 ns | 5,401.157 ns | 11,968.591 ns | 268,830.42 ns |
| QuickClassicalSort\_Reversed | 1000 | 22,106.82 ns | 440.318 ns | 1,227.430 ns | 22,015.53 ns |
| QuickClassicalSort\_Almost | 1000 | 30,445.19 ns | 163.834 ns | 136.809 ns | 30,421.90 ns |
| QuickClassicalSort\_FewUnique | 1000 | 20,806.18 ns | 466.707 ns | 1,368.770 ns | 20,570.46 ns |
| QuickSort\_Random | 1000 | 7,827.31 ns | 184.595 ns | 544.283 ns | 7,673.21 ns |
| QuickSort\_Sorted | 1000 | 4,574.15 ns | 125.329 ns | 369.535 ns | 4,544.38 ns |
| QuickSort\_Reversed | 1000 | 7,812.12 ns | 155.395 ns | 406.643 ns | 7,775.25 ns |
| QuickSort\_Almost | 1000 | 6,223.21 ns | 123.805 ns | 310.602 ns | 6,205.42 ns |
| QuickSort\_FewUnique | 1000 | 7,845.71 ns | 156.636 ns | 431.421 ns | 7,755.25 ns |
| InsertionSort\_Random | 100000 | 1,807,789,585.37 ns | 35,790,714.453 ns | 64,538,040.878 ns | 1,792,670,300.00 ns |
| InsertionSort\_Sorted | 100000 | 163,314.50 ns | 3,388.014 ns | 9,666.195 ns | 161,941.05 ns |
| InsertionSort\_Reversed | 100000 | 1,845,630,675.41 ns | 36,665,400.138 ns | 82,759,918.351 ns | 1,836,217,200.00 ns |
| InsertionSort\_Almost | 100000 | 238,239,821.79 ns | 4,736,149.501 ns | 6,482,894.896 ns | 238,901,333.33 ns |
| InsertionSort\_FewUnique | 100000 | 1,852,190,301.89 ns | 36,969,816.813 ns | 77,169,774.865 ns | 1,851,959,900.00 ns |
| MergeSort\_Random | 100000 | 11,310,907.26 ns | 257,270.656 ns | 742,284.666 ns | 11,146,789.06 ns |
| MergeSort\_Sorted | 100000 | 4,808,529.20 ns | 95,325.907 ns | 226,552.103 ns | 4,769,932.81 ns |
| MergeSort\_Reversed | 100000 | 10,199,323.08 ns | 76,687.616 ns | 64,037.633 ns | 10,213,392.19 ns |
| MergeSort\_Almost | 100000 | 5,569,936.96 ns | 110,478.649 ns | 256,051.069 ns | 5,537,478.91 ns |
| MergeSort\_FewUnique | 100000 | 10,147,564.71 ns | 103,045.472 ns | 80,451.127 ns | 10,131,935.16 ns |
| QuickClassicalSort\_Random | 100000 | 6,332,596.27 ns | 19,022.402 ns | 15,884.567 ns | 6,340,924.22 ns |
| QuickClassicalSort\_Sorted | 100000 | NA | NA | NA | NA |
| QuickClassicalSort\_Reversed | 100000 | 6,286,985.49 ns | 15,513.919 ns | 12,954.825 ns | 6,283,446.48 ns |
| QuickClassicalSort\_Almost | 100000 | 5,108,979.12 ns | 39,639.160 ns | 33,100.494 ns | 5,113,930.08 ns |
| QuickClassicalSort\_FewUnique | 100000 | 5,923,131.81 ns | 17,261.592 ns | 15,301.952 ns | 5,925,954.30 ns |
| QuickSort\_Random | 100000 | 5,484,597.45 ns | 20,177.962 ns | 18,874.479 ns | 5,478,809.38 ns |
| QuickSort\_Sorted | 100000 | 822,071.64 ns | 2,140.305 ns | 1,787.252 ns | 822,590.09 ns |
| QuickSort\_Reversed | 100000 | 5,505,499.49 ns | 30,963.915 ns | 25,856.272 ns | 5,506,880.86 ns |
| QuickSort\_Almost | 100000 | 2,463,448.61 ns | 11,170.757 ns | 10,449.133 ns | 2,464,548.63 ns |
| QuickSort\_FewUnique | 100000 | 5,427,950.06 ns | 22,695.484 ns | 18,951.757 ns | 5,427,739.06 ns |

# Wnioski

Bazując na powyższych pomiarach można jednoznacznie stwierdzić, że w przypadku tablic średnich i dużych rozmiarów, QuickSort jest uniwersalnie najszybszym algorytmem. Warto jednak zauważyć, że w przypadku małych tablic, nieznacznie lepiej wypada InsertionSort, który przy losowych tablicach o rozmiarze 10, jest średnio o 5 ns szybszy.

Tablice już posortowane (dowolnego rozmiaru), są zawsze najszybciej obsługiwane przez InsertionSort – w skali dużej, algorytm ten wykorzystuje tylko ~20% czasu, jaki na przetworzenie takiej tablicy potrzebuje drugi najszybszy algorytm.

MergeSort w skali dużej, jest w stanie obsłużyć tablicę prawie posortowaną, nieznacznie wolniej niż QuickSort, jednak fakt, że w skali średniej wypada gorzej, i w skali małej najgorzej, może sugerować że w skali ogromnych tablic, może osiągać lepsze wyniki.

InsertionSort jest nieoptymalny w skali dużej, jako że potrzebuje na obsłużenie losowo wygenerowanej tablicy potrzebuje ponad 100 razy więcej czasu niż MergeSort i ponad 200 razy więcej niż QuickSort.

Wszystkie testy wskazują że QuickSort jest bezpośrednim ulepszeniem QuickSortClassical.