Projektowanie Algorytmów i Metody Sztucznej Inteligencji

Temat: Algorytmy sortujące

Nazwisko i Imię prowadzącego kurs: mgr inż. Marta Emirsajłow

Wykonawca:	
Imię i Nazwisko Nr indeksu, wydział	Jakub Kolasa 249012, W4
Termin zajęć	Pt, 13:15-15:00
Data oddania sprawdozdania	27.03.2020

1. Wprowadzenie.

Celem projektu jest przeprowadzenie analizy algorytmów sortujących. Algorytmy te sortują elementy tablicy rosnąco począwszy od indeksu zerowego. Główną badaną cechą algorytmów będzie ich czas działania w zależności od ilości elementów tablicy oraz od jej początkowego posortowania.

2. Badane algorytmy.

Badanymi algorytmami będą sortowanie przez kopcowanie, sortowanie przez scalanie, oraz sortowanie szybkie.

2.1. Sortowanie przez kopcowanie (*ang.* heapsort).

Działanie algorytmu Heapsort opiera się na strukturze kopca binarnego zupełnego. Jest to struktura danych o charakterystycznych właściwościach, które są wykorzystywane przez algorytm. Pierwszą z nich jest relacja mniejszości tzn. zasada według, której rodzic zawsze reprezentuje wyższą wartość, niż jego dzieci. W konsekwencji element o najwyższej wartości będzie znajdował się w korzeniu drzewa (*ang.* root). Drugą właściwością opisywanej struktury jest zupełność. Oznacza to, że każdy poziom kopca jest zapełniony maksymalnie od strony lewej począwszy od pierwszego poziomu. Dzięki tej zasadzie dowolną tablice z danymi można przedstawić jako kopiec. Aby odnaleźć dzieci rodzica należy posłużyć się wzorami: indeks lewego dziecka =2k+1, indeks prawego dziecka =2k+2, gdzie k - indeks rodzica.

Na początku algorytm sortuje dane w taki sposób, aby spełniały one relację mniejszości. Heapsort dzieli główny kopiec na mniejsze podkopce i sortuje cały kopiec począwszy od ostatniego rodzica. Następnie zamienia pierwszy największy element z ostatnim. Daje to gwarancję, że element o największej wartości znajduje się na samym końcu kopca. Ten element zostaje wyłączony i nie bierze się go pod uwagę w kolejnym strukturyzowaniu kopca. Cała opisana procedura jest powtarzana do momentu w którym tablica zostanie posortowana.

Złożoność czasowa algorytmu heapsort: $O(n \log n)$

Pesymistyczna złożoność czasowa algorytmu heapsort: $O(n \log n)$

Algorytm sortowania przez kopcowanie jest zatem algorytmem stabilnym gdyż niezależnie od początkowego posortowania tablicy jego złożoność obliczeniowa pozostanie taka sama.

Implementacja algorytmu heapsort w języku C++

```
void heapify(int Table[], int Size, int root){
    int left = 2*root + 1;
    int right = 2*root + 2;
    int N = root;
    if (left < Size && Table[left] > Table[N])      N = left;
    if (right < Size && Table[right] > Table[N]) N = right;
    if (N != root){
        swap(Table[root], Table[N]);
        heapify(Table, Size, N);
    }
}
```

```
void Heapsort(int Table[], int left, int right){
	int root;
	for (root = right/2-1; root >= 0; root--)	heapify(Table, right, root);
	for (root = right-1; root>=left; root--){
		swap(Table[left], Table[root]);
		heapify(Table, root, left);
	}
}
```

2.2. Sortowanie przez scalanie (*ang.* mergesort).

Działanie algorytmu mergesort opiera się na metodzie dziel i zwyciężaj. Jest to algorytm rekurencyjny. Dzieli on tablice na dwie części do momentu, aż nowe tablice będą jednoelementowe. Następnie elementy pochodzące z dwóch mniejszych tablic są z powrotem łączone w taki sposób, że elementy o wyższej wartości są wstawiane na prawo, a mniejsze na lewo. W ten sposób gdy rekurencja dobiegnie końca tablica będzie posortowana.

Złożoność czasowa algorytmu mergesort: $O(n \log n)$

Pesymistyczna złożoność czasowa algorytmu mergesort: $O(n \log n)$

Algorytm mergesort jest stabilny gdyż jego złożoność obliczeniowa nie może zostać pogorszona poprzez sortowanie nieprzychylnego algorytmowi zestawu liczb.

Implementacja algorytmu mergesort w języku C++

```
void Merge(int MainTable[], int left, int mid, int right){
	int leftTableSize = mid-left+1;
	int rightTableSize = right-mid;
	int LeftTable [leftTableSize];
	int RightTable[rightTableSize];
	int i,j;

	for(i = 0; i<leftTableSize; i++) LeftTable[i] = MainTable[left+i];
	for(j = 0; j<rightTableSize; j++) RightTable[j] = MainTable[mid+j+1];

	int position = left;
	for(i = 0, j = 0; i < leftTableSize &&	j < rightTableSize; position++){
		if(LeftTable[i] <= RightTable[j]){
			MainTable[position] = LeftTable[i];
			i++;
		}else{
			MainTable[position] = RightTable[j];
			j++;
		}
	}
	for(; i<leftTableSize; i++, position++) MainTable[position] = LeftTable[i];
	for(; j<rightTableSize; j++, position++) MainTable[position] = RightTable[j];
}
```

```
void MergeSort(int Table[], int left, int right){
	if(left < right){
		int mid = (right+left)/2;
		MergeSort(Table, left, mid);
		MergeSort(Table, mid+1, right);
		Merge(Table, left, mid, right);
	}
}
```

2.3. Sortowanie szybkie (*ang.* quicksort).

Quicksort tak samo jak mergesort opiera się na technice dziel i zwyciężaj. Jednak, że quicksort wykonuje sortowanie już w trakcie podziału, bez późniejszego etapu scalania. Dzielenie odbywa się poprzez wybrania pivota - elementu, który podzieli tablicę. Początkowa faza algorytmu polega na stworzeniu na lewym końcu tablicy ,,granicy". Gdy jakiś element przed którym znajduje się granica będzie mniejszy niż wcześniej wybrany pivot to zostanie on ,,przerzucony" na jej drugą stronę. Natomiast gdy element okaże się mniejszy to sprawdzane są kolejne elementy. Na koniec element pełniący rolę pivota zostaje wstawiony w miejsce w którym znajduje się granica. W tej sytuacji jest pewność, że elementy znajdujące się po jego lewej stronie są od niego mniejsze, a po prawej stronie są od niego większe. Następnie wywoływana jest funkcja quicksort dla tablic wytworzonych z podziału podziału przez pivota. Proces ten jest powtarzany do momentu w którym sortowane tablice są jednoelementowe.

Złożoność czasowa algorytmu quicksort: $O(n \log n)$

Pesymistyczna złożoność czasowa algorytmu quicksort: $O(\log n^2)$

Przypadek pesymistyczny (gdy pivotem zostanie najmniejszy lub największy element w tablicy) powoduje znaczny wzrost złożoności czasowej algorytmu. Sprawia to, że quicksort bez odpowiednich zabezpieczeń może zostać zdestabilizowany i w znaczny sposób wydłużyć proces sortowania. Rolę zabezpieczenia może pełnić wybranie na pivota mediany, lub przewidywanej średniej.

Implementacja algorytmu quicksort w języku C++

```
int Border(int Table[], int left, int right){
	int pivot = Table[(left+right)/2];
	while (true){
		while (Table[right] > pivot) right--;
		while (Table[left] < pivot) left++;
		if (left < right){ swap(Table[left],Table[right]); left++; right--;}
		else return right;
	}
}
```

```
void Quicksort(int Table[], int left, int right){
	if (left < right){
		int pivot = Border(Table,left,right);
		Quicksort(Table, left, pivot);
		Quicksort(Table, pivot+1, right);
	}
}
```

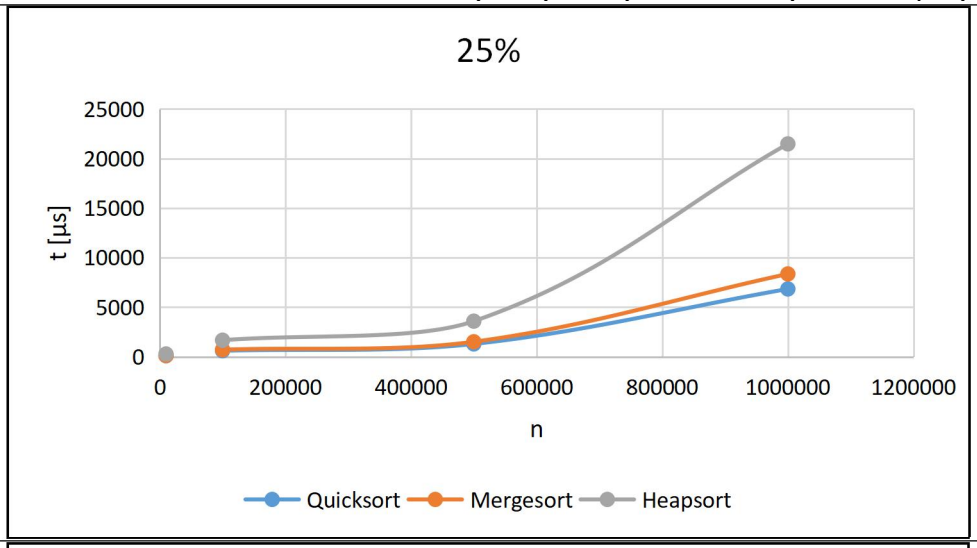
3. Przebieg eksperymentu.

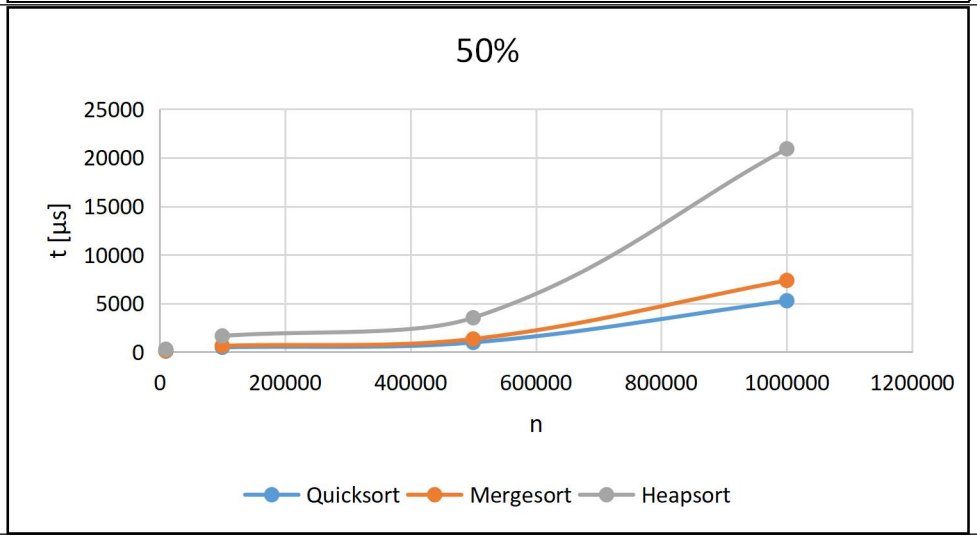
W celu wykonania eksperymentu stworzono klasę Sorter *(folder Sortowanie)*, której dynamicznie tworzone obiekty wykonywały sortowania na wygenerowanych tablicach o konkretnym poziomie początkowego posortowania, a następnie pomiary czasowe zapisywały do pliku w taki sposób, aby można je było odczytać za pomocą programu Excel. Każdy z algorytmów sortował ten sam zestaw danych.

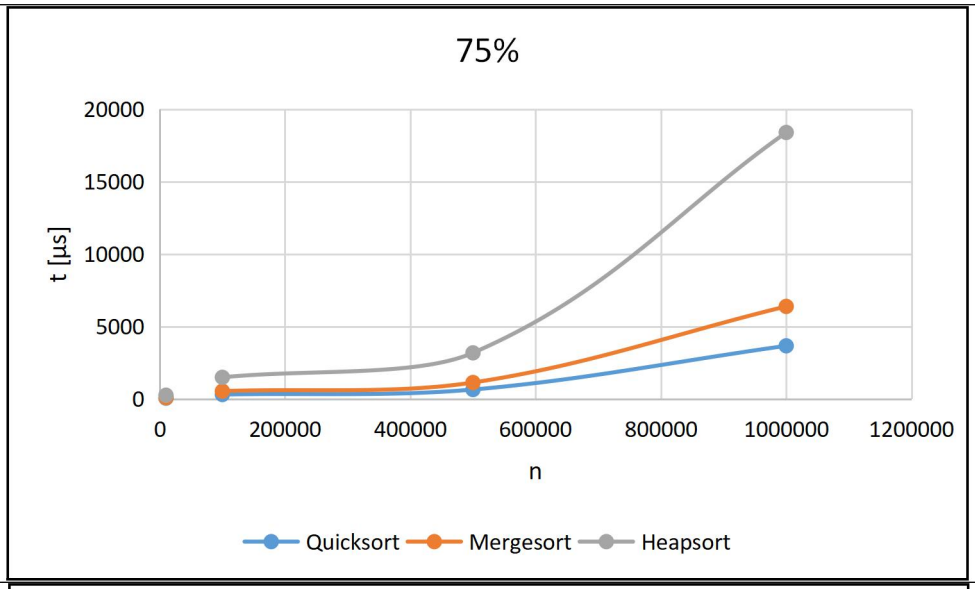
Aby upewnić się, że algorytmy prawidłowo sortują stworzono prosty program, który po wykonaniu sortowania analizował wyjściową tablicę. *(folder Testowanie)*

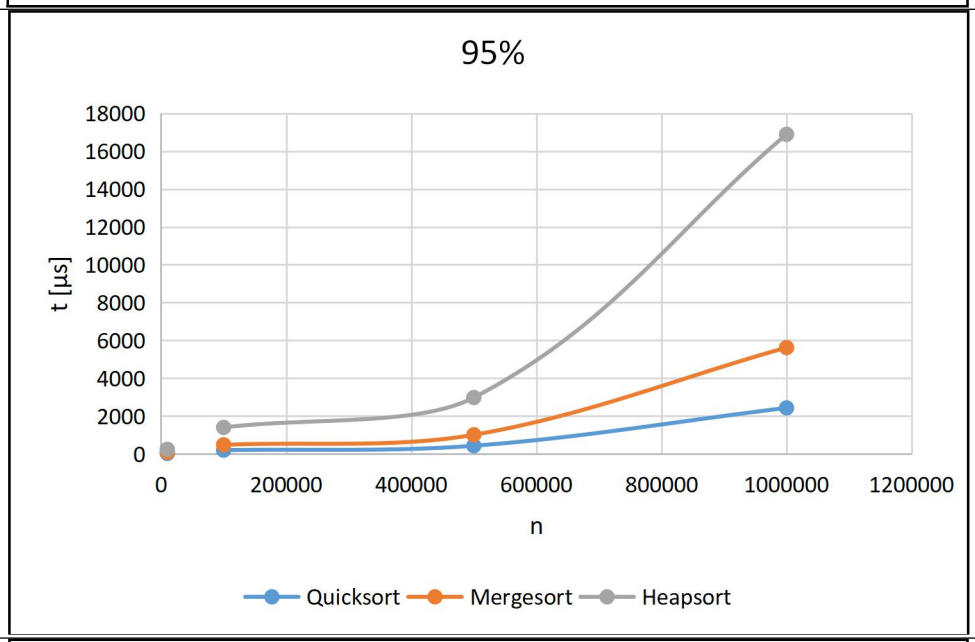
Otrzymane wyniki umieszczono w tabeli *(strona 9)*, a następnie na ich podstawie utworzono wykresy. W tytule wykresów znajduje się informacja na temat ich początkowego posortowania.

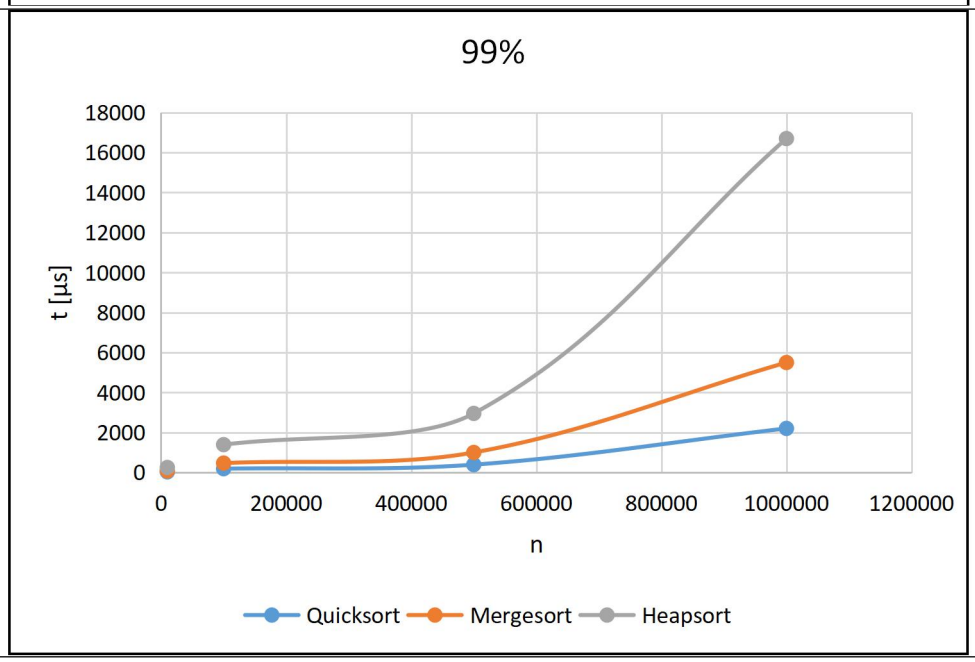
Wykresy otrzymane w wyniku eksperymntu

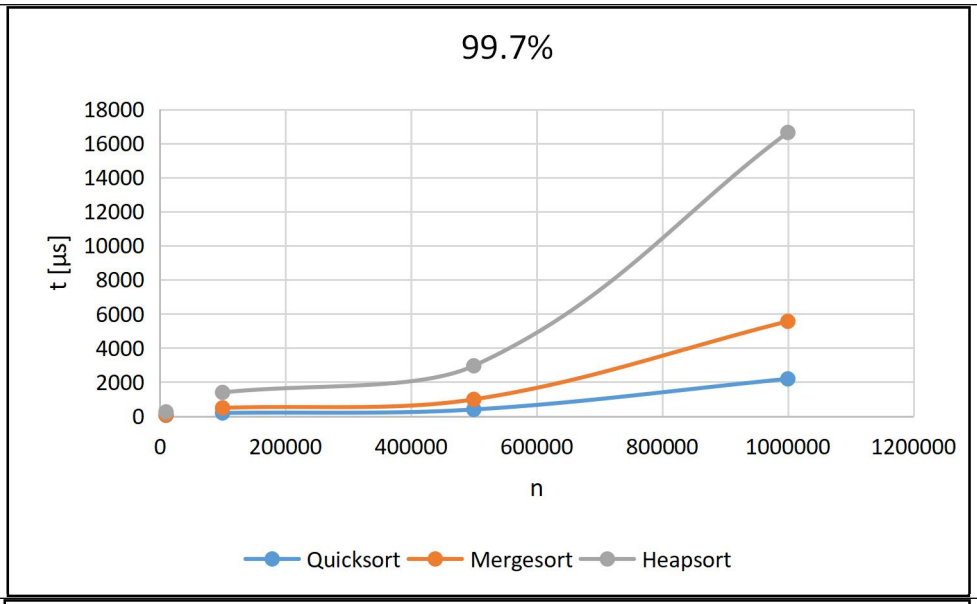


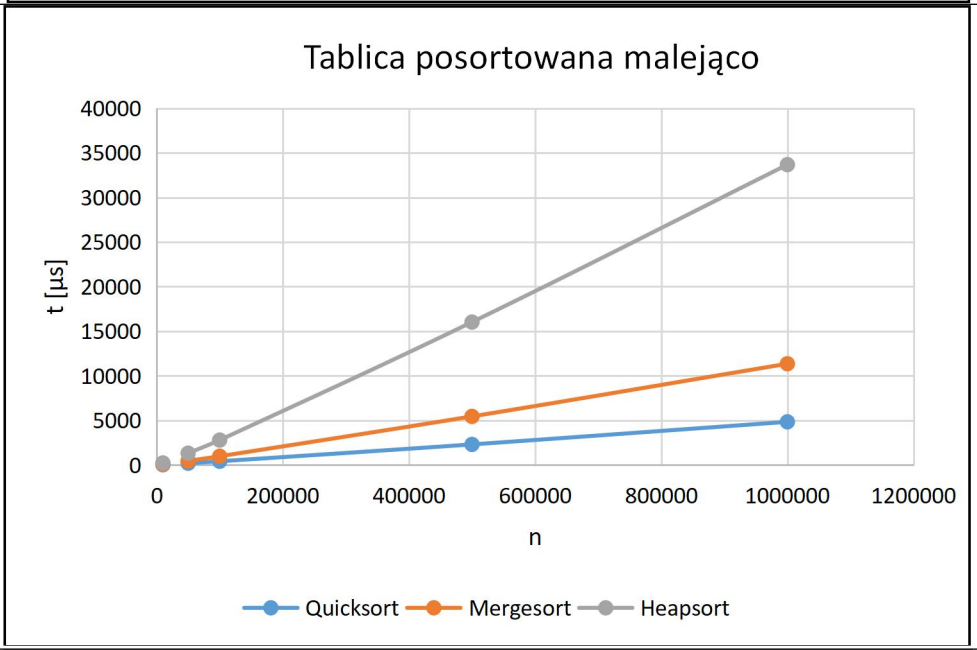


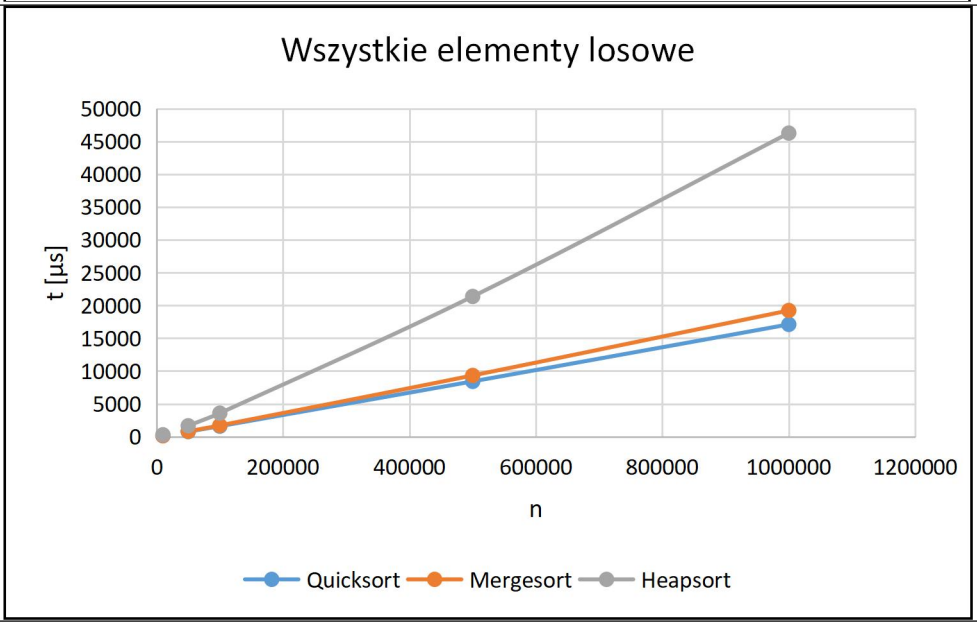

75%
20000
15000
10000
5000
0
t [µs]
0
200000
400000
600000
800000
1000000
1200000
n
Quicksort
Mergesort
Heapsort


95%
18000
16000
14000
12000
10000
8000
6000
4000
2000
0
t [µs]
0
200000
400000
600000
800000
1000000
1200000
n
Quicksort
Mergesort
Heapsort


99%
18000
16000
14000
12000
10000
8000
6000
4000
2000
0
t [µs]
0
200000
400000
600000
800000
1000000
1200000
n
Quicksort
Mergesort
Heapsort


99.7%
18000
16000
14000
12000
10000
8000
6000
4000
2000
0
t [µs]
0
200000
400000
600000
800000
1000000
1200000
n
Quicksort
Mergesort
Heapsort


Tablica posortowana malejąco
40000
35000
30000
25000
20000
15000
10000
5000
0
t [µs]
0
200000
400000
600000
800000
1000000
1200000
n
Quicksort
Mergesort
Heapsort


Wszystkie elementy losowe
50000
45000
40000
35000
30000
25000
20000
15000
10000
5000
0
t [µs]
0
200000
400000
600000
800000
1000000
1200000
n
Quicksort
Mergesort
Heapsort

4. Analiza wyników oraz wnioski końcowe.

Zgodnie z przewidywaniami wszystkie trzy algorytmy sortujące w przybliżeniu wykazały proporcjonalną zależność od złożonosci obliczeniowej. Dla wszystkich algorytmów wynosiła ona $O(n \log n)$. Wykresy przedstawiające sortowania w przypadku wszystkich elementów losowych, oraz tablicy posortowanej malejąco na pierwszy rzut oka wydają się być liniowe. Jest to spowodowane przeskalowaniem wykresu przez duże różnice w czasie trwania poszczególnych algorytmów.

Eksperyment pokazał przewagę algorytmu szybkiego sortowania nad resztą. W tym przypadku początkowe umieszczenie pivota w środku tablicy zdołało zabezpieczyć sortowanie przed pesysmistycznym przypadkiem.

Sortowanie przez scalanie działa porównywalnie szybko do quicksorta dla wszystkich elementów losowych. Jednak w miarę zwiększania początkowego posortowania tablicy czas działania algorytmu mergesort zaczyna coraz bardziej odstawać od czasu quicksorta. Ostatecznie dla początkowego posortowania równeg 99.7% oraz dla miliona elementów quicksort działa, aż trzykrotnie szybciej od mergesorta.

Najgorsze czasy w eksperymencie uzyskało sortowanie przez kopcowanie. Mimo to warto zauważyć, że czas działania dla każdego przypadku był niemal identyczny. Pokazuje to bardzo pozytywną cechę algorytmu heapsort jaką jest jego mocna stabilność.

5. Źródła.

http://lukasz.jelen.staff.iiar.pwr.edu.pl/styled-2/page-2/index.php'=
https://www.geeksforgeeks.org/sorting-algorithms/
http://www.algorytm.org/algorytmy-sortowania/
https://www.softwaretestinghelp.com/quick-sort/

Tabela z wynikami eksperymentów

Początkowo posortowanych elementów	Rozmiar tablicy	Czas sortownia Quicksort [µs]	Czas sortowana Mergesort [µs]	Czas sortowania Heapsort [µs]
Wszystkie elementy losowe	10000	140	140	281
	50000	765	812	1655
	100000	1593	1702	3577
	500000	8419	9325	21369
	1000000	17105	19229	46270
25%	10000	109	125	281
	50000	609	718	1671
	100000	1296	1515	3592
	500000	6842	8357	21479
	1000000	13934	17261	46504
50%	10000	78	109	265
	50000	468	624	1640
	100000	968	1327	3514
	500000	5264	7357	20916
	1000000	10778	15277	45020
75%	10000	62	93	265
	50000	312	546	1499
	100000	656	1140	3186
	500000	3671	6389	18386
	1000000	7591	13309	39115
95%	10000	31	93	249
	50000	203	484	1405
	100000	437	1015	2983
	500000	2436	5623	16886
	1000000	5092	11747	35538
99%	10000	31	78	249
	50000	187	468	1390
	100000	390	999	2952
	500000	2202	5498	16699
	1000000	4608	11466	35007
99.7%	10000	46	78	249
	50000	187	484	1390
	100000	390	984	2952
	500000	2186	5561	16636
	1000000	4545	11419	34897
Tablica posortowna malejąco	10000	31	93	234
	50000	203	468	1327
	100000	421	984	2796
	500000	2311	5451	16027
	1000000	4842	11356	33679