

19.04.2020

SPRAWOZDANIE

PAMSI

PROJEKT 1

ALGORYTMY SRTOWANIA

Prowadzący:

Mgr inż. Marta Emirsajłow

Dane studenta:

Krzysztof Ragan 249026

Termin zajęć:

Piątek 11:15

1. Zadanie

Celem zadania było zaimplementowanie wybranych algorytmów sortowania. Wybrano następujące algorytmy:

- Sortowanie przez scalanie;
- Sortowanie szybkie;
- Sortowanie introspektywne;

Oprócz implementacji przeprowadzono również badania algorytmów, czasu ich działania dla różnych rozmiarów tabel i różnych stopni ich posortowania.

2. Opis algorytmów

a) Sortowanie przez scalanie(Merge Sort)

Sposób działania algorytmu sortowania przez scalanie wykorzystuje zasadę „dziel i zwyciężaj”. Dzieli on problem na mniejsze podproblemy i wywołuje się rekurencyjnie również dla nich. Rekurencja trwa, aż podzielone części tablicy będą miały rozmiar 2. Wtedy zaczynamy proces scalania (Merge). Scalanie odbywa się na dwóch mniejszych, ale już posortowanych ciągach i tworzy jeden większy, również posortowany.

Dzielenie tablicy o długości n na połowy może odbyć się $\log_2 n$ razy. Scalanie na każdym poziomie zajmuje $O(n)$ operacji. Wynika z tego, że całkowita złożoność obliczeniowa algorytmu Merge Sort to $O(n \log n)$.

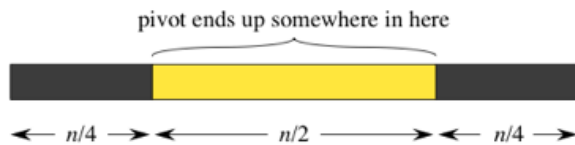
Sortowanie przez scalanie posiada taką samą złożoność obliczeniową zarówno dla najlepszego, średniego jak i najgorszego przypadku.

b) QuickSort (sortowanie szybkie)

Sortowanie szybkie polega na wyborze elementu rozdzielającego, według którego uporządkujemy pozostałe elementy. Wszystkie na lewo od pivotu będą od niego mniejsze, a te na prawo-większe. Następnie korzystamy z rekurencji, wywołując sortowanie szybkie dla lewej i prawej strony, pivotu zostawiając na swoim miejscu.

W przypadku najgorszym element rozdzielający znajduje się na początku lub końcu tablicy, przez co jedna z partycji ma wielkość 0, a druga jest tylko o jeden mniejsza od wyjściowej tablicy. Suma czasu potrzebnego na wykonanie tych operacji to $c(n+(n-1)+(n-2)+(n-3)+\dots+1)$. A zatem wykorzystując wzór na sumę ciągu arytmetycznego otrzymujemy $(n+1) * n^2$. Złożoność obliczeniowa jest zatem równa $O(n^2)$.

W przypadku średnim mamy do czynienia z czymś pomiędzy najlepszym (czyli podziałem na połowy), a najgorszym przypadkiem. Gdyby pivot łądownał zawsze w takim obszarze:



To znaczy, że w najgorszym przypadku podział nastąpiłby na partycje o $n/4$ i $3n/4$ elementach. Aby obliczyć ile wywołań rekurencji wystąpi dla dłuższej partycji należy przeanalizować problem następująco: idąc od dołu (gdzie rozmiar to 1) każdy wyższy poziom będzie miał rozmiar $4/3$ raza większy (bo niższy poziom to $3/4$ wyższego);

zatem ile razy należy powiększać $4/3$ raza wartość 1 aby otrzymać n ? Lub inaczej:

$$\left(\frac{4}{3}\right)^x = n;$$

Po przekształceniu $x = \log_{\frac{4}{3}} n$. Analogiczne rozumowanie można przeprowadzić dla krótszej partycji, jednakże nie jest to niezbędne. Ponadto:

$$\log_{\frac{4}{3}} n = \frac{\log_2 n}{\log_2 \frac{4}{3}};$$

Na każdym z poziomów następuje n operacji, zatem złożoność obliczeniowa wynosi $O(n \log n)$.

c) Sortowanie introspektywne (IntroSort)

Jest to algorytm hybrydowy, połączenie dwóch lub trzech algorytmów. Przedstawione później wyniki są wynikami działania algorytmu złożonego z trzech innych: QuickSorta, HeapSorta oraz InsertSorta.

Celem hybrydyzacji algorytmu jest wykluczenie najgorszego przypadku dla algorytmu sortowania szybkiego. Wprowadza się licznik, który liczy głębokość wywołań rekurencyjnych algorytmu. Jeśli głębokość jest zbyt duża, zamiast kolejnego QuickSorta wykorzystuje się HeapSorta (sortowanie przez kopcowanie). Ponadto dla partycji o wielkości mniejszej niż 16 wykorzystuje się sortowanie przez wstawianie (Insert Sort).

Złożoność obliczeniowa algorytmu zarówno w najgorszym jak i średnim przypadku wynosi $O(n \log n)$.

3. Przebieg eksperymentu

Do implementacji algorytmów w języku C++ wykorzystano program Qt Creator 4.11.0. Zgodnie z instrukcją, pomiary czasu działania algorytmów wykonano dla różnych typów tabel oraz różnych ich rozmiarów. Poniżej przedstawiono tabele wyników (czas podano w milisekundach) oraz wykresy sporządzone na ich podstawie.

1) Sortowanie przez scalanie

```
template <class Item>
void merge(Item *tab, int left, int mean, int right, Item *temp)
{
    int i, j;
    for(i = mean + 1; i>left; i--)
        temp[i-1] = tab[i-1];
    for(j = mean; j<right; j++)
        temp[right+mean-j] = tab[j+1];
    for(int k=left;k<=right;k++)
        if(temp[j]<temp[i])
            tab[k] = temp[j--];
        else
            tab[k] = temp[i++];
}

template <class Item>
void mergeSortRange(Item *array,int left, int right, Item *temp)
{
    if(right<=left) return;

    int mean = (right+left)/2;

    mergeSortRange(array, left, mean, temp);
    mergeSortRange(array, mean+1, right, temp);

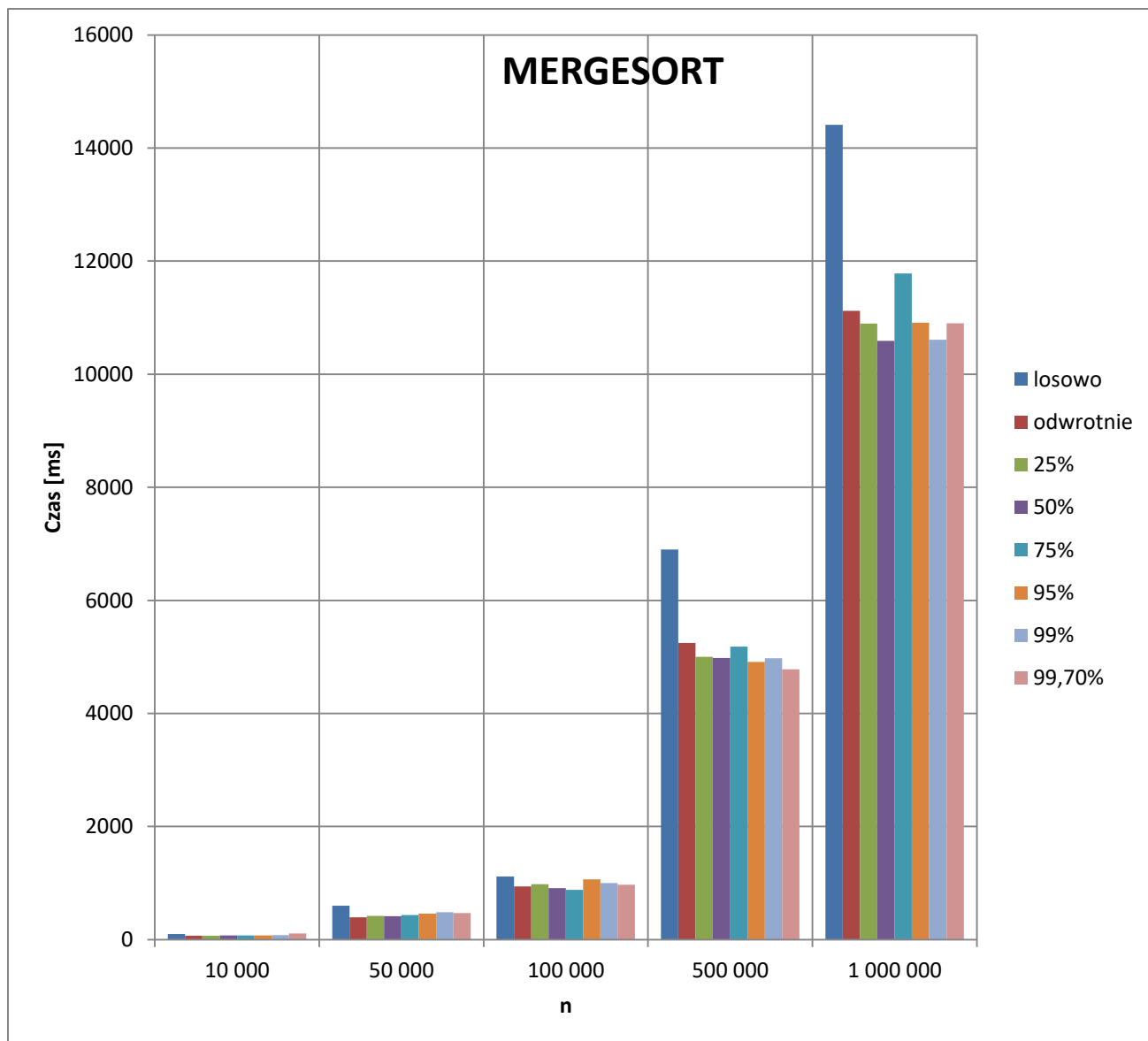
    merge(array, left, mean, right, temp);
}

template <class Item>
void mergeSort(Item *array,int size)
{
    Item *temp=new Item[size];
    mergeSortRange(array,0,size-1,temp);
    free(temp);
}
```

Kod 1. Implementacja algorytmu Mergesort.

Tabela 1. Wyniki pomiarów czasu algorytmu sortowania przez scalanie

n\rodzaj tabeli	losowo	odwrotnie	25%	50%	75%	95%	99%	99,70%
10 000	98	69	68	76	74	73	78	108
50 000	600	397	418	415	436	461	485	470
100 000	1118	943	980	911	882	1067	1003	972
500 000	6897	5246	5004	4980	5182	4911	4978	4782
1 000 000	14412	11123	10897	10592	11782	10910	10612	10900



Wykres 1. Wyniki pomiarów czasu algorytmu sortowania przez scalanie

2) Sortowanie szybkie

```
template <class Item>
void quickSortRange(Item *array, int left, int right)
{
    if(right <= left) return;

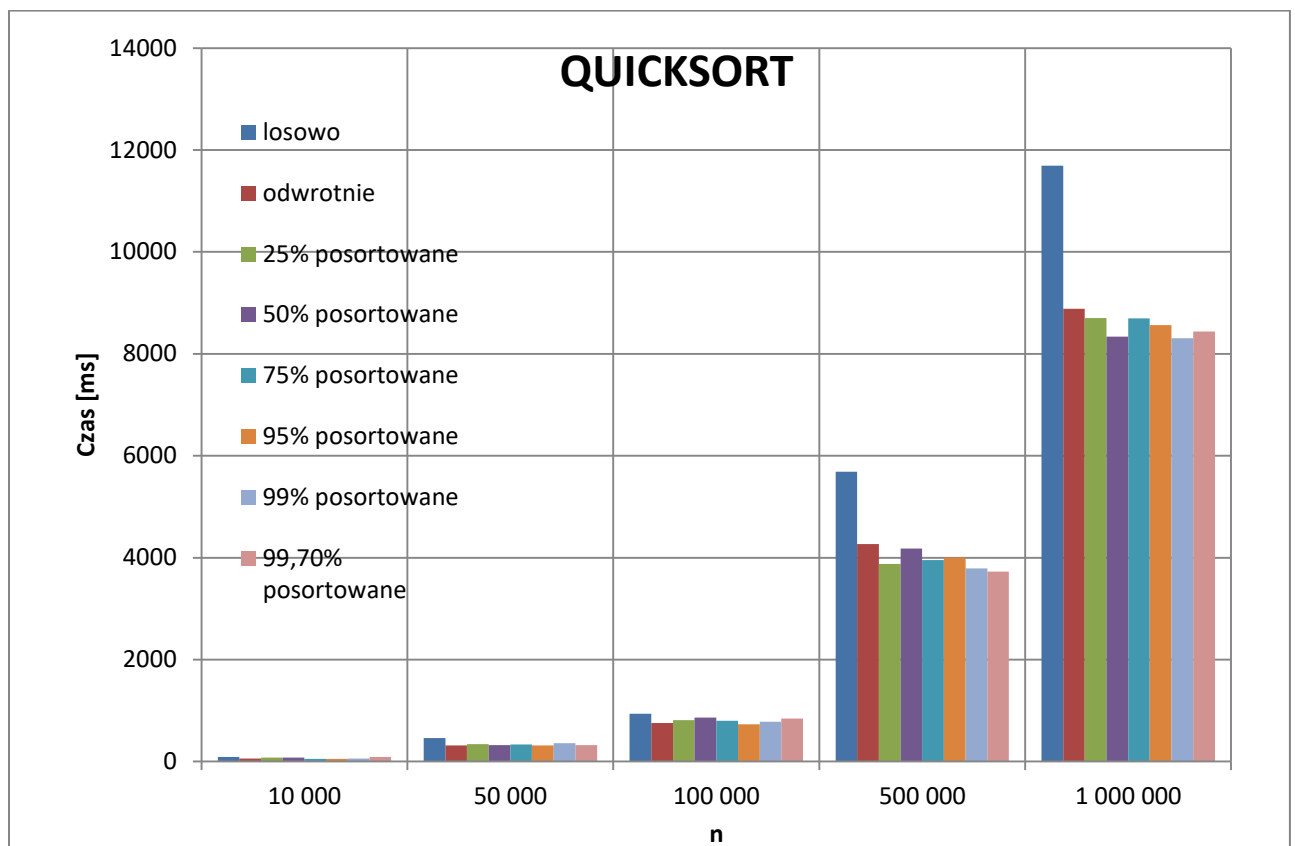
    int i = left - 1, j = right + 1,
        pivot = array[(left+right)/2];

    while(1)
    {
        while(pivot > array[++i]);
        while(pivot < array[--j]);
        if( i <= j)
            swap(array[i],array[j]);
        else
            break;
    }
    if(j > left)
        quickSortRange(array, left, j);
    if(i < right)
        quickSortRange(array, i, right);
}
#endif // QUICKSORT_H
template <class Item>
void quickSort(Item *array, int size)
{
    quickSortRange(array,0,size-1);
}
```

Kod 2. Implementacja algorytmu Quicksort.

Tabela 2. Wyniki pomiarów czasu algorytmu sortowania szybkiego

n\rodzaj tabeli	losowo	odwrotnie	25%	50%	75%	95%	99%	99,70%
10 000	91	56	79	78	55	52	59	89
50 000	459	319	341	325	332	315	357	320
100 000	936	757	811	863	799	732	782	844
500 000	5689	4268	3876	4178	3956	4001	3789	3727
1 000 000	11690	8887	8702	8340	8694	8561	8305	8441



Wykres 2. Wyniki pomiarów czasu algorytmu sortowania szybkiego

3) Sortowanie introspektywne

```
template <class Item>
void exchange (Item *array, long i, long j)
{
    Item temp;
    temp=array[i];
    array[i]=array[j];
    array[j]=temp;
}
template <class Item>
void heapify (Item *array, long i, long N)
{
    long j;
    while (i<=N/2)
    {
        j=2*i;
        if (j+1<=N && array[j+1]>array[j])
            j=j+1;
        if (array[i]<array[j])
            exchange(array,i,j);
        else break;
        i=j;
    }
}
template <class Item>
void insertionSort (Item *array, long N)
{
    long i, j;
    Item temp;
    for (i=1; i<N; ++i)
    {
        temp=array[i];
        for (j=i; j>0 && temp<array[j-1]; --j)
            array[j]=array[j-1];
        array[j]=temp;
    }
}

template <class Item>
void heapSort (Item *array, long N)
{
    long i;
    for (i=N/2; i>0; --i)
        heapify(array-1,i,N);
    for (i=N-1; i>0; --i)
    {
        exchange(array,0,i);
        heapify(array-1,1,i);
    }
}
```

Kod 3. Implementacja algorytmu Introsort.


```

template <class Item>
void medianOfThree (Item *array, long &L, long &R)
{
    if (array[++L-1]>array[--R])
        exchange(array,L-1,R);
    if (array[L-1]>array[R/2])
        exchange(array,L-1,R/2);
    if (array[R/2]>array[R])
        exchange(array,R/2,R);
    exchange(array,R/2,R-1);
}
template <class Item>
long partition (Item *array, long L, long R)
{
    long i, j;
    if (R>=3)
        medianOfThree(array,L,R);
    for (i=L, j=R-2; ; )
    {
        for ( ; array[i]<array[R-1]; ++i);
        for ( ; j>=L && array[j]>array[R-1]; --j);
        if (i<j)
            exchange(array,i++,j--);
        else break;
    }
    exchange(array,i,R-1);
    return i;
}
template <class Item>
void introSort (Item *array, long N, int M)
{
    long i;
    if (M<=0)
    {
        heapSort(array,N);
        return;
    }
    i=partition(array,0,N);
    if (i>9)
        introSort(array,i,M-1);
    if (N-1-i>9)
        introSort(array+i+1,N-1-i,M-1);
}

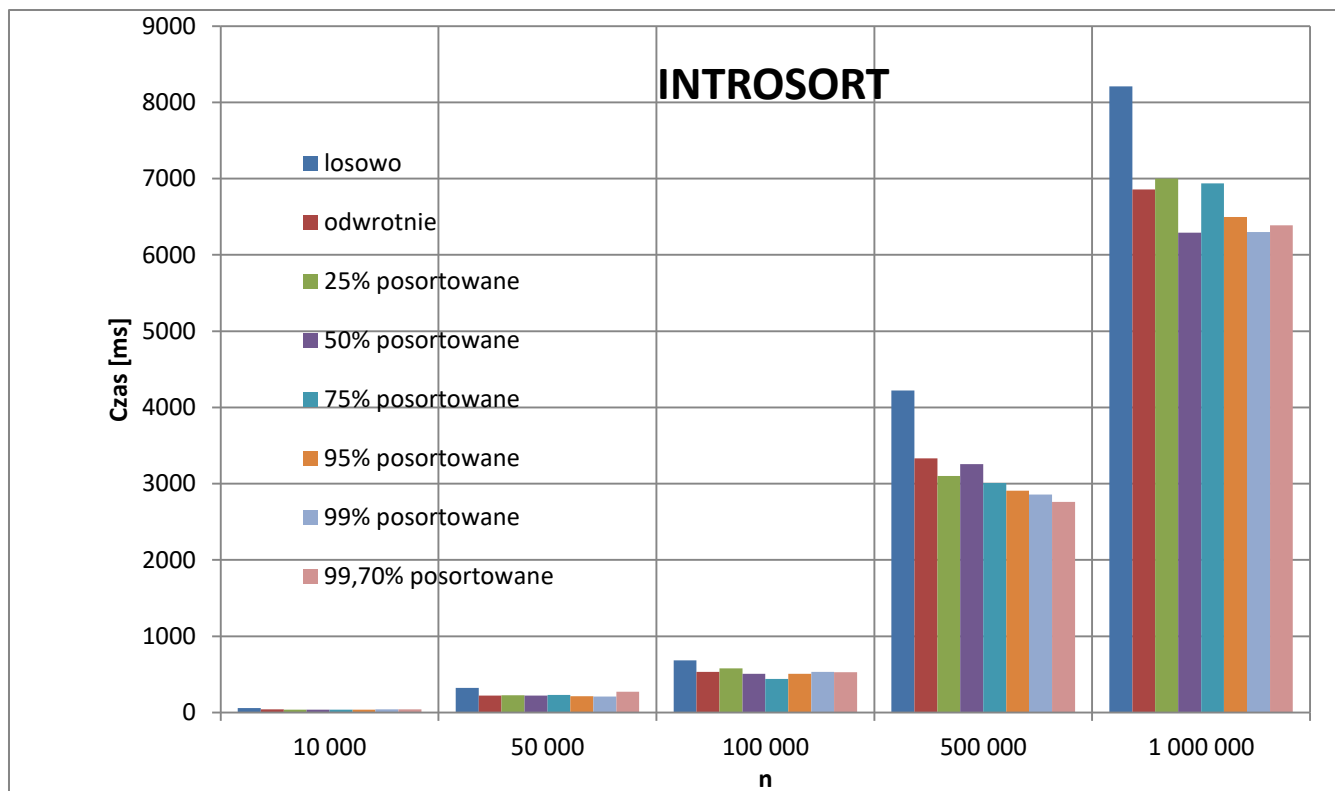
template <class Item>
void introspectiveSort (Item *array, long N)
{
    introSort(array,N,static_cast<int>(floor(2*log(N)/M_LN2)));
    insertionSort(array,N);
}

```

Kod 4. Implementacja algorytmu Introsort.

Tabela 3. Wyniki pomiarów czasu algorytmu sortowania introspektywnego

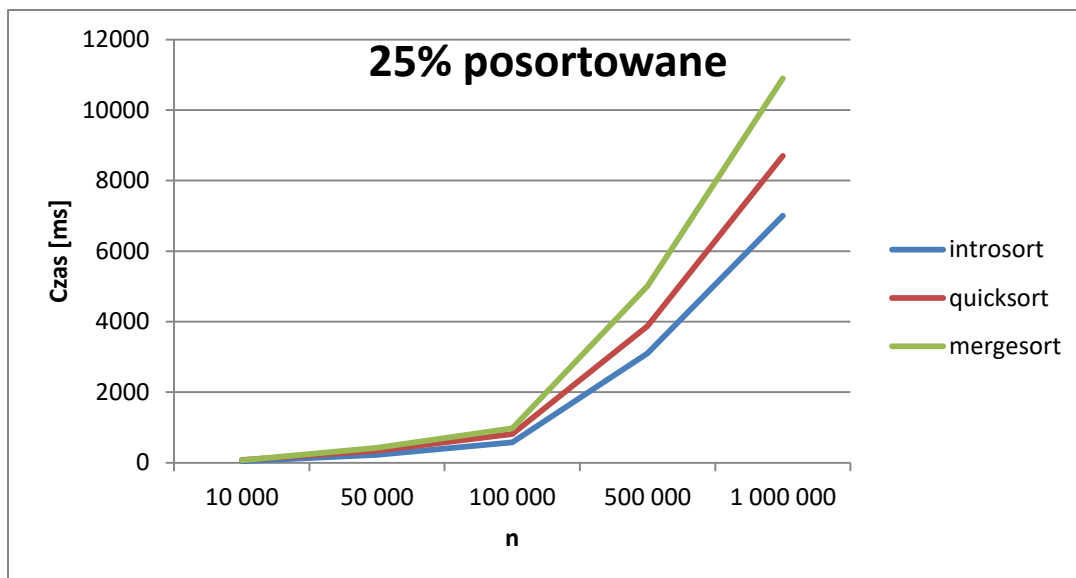
n\rodzaj tabeli	losowo	odwrotnie	25%	50%	75%	95%	99%	99,70%
10 000	56	42	38	38	36	37	39	41
50 000	323	222	227	220	231	212	209	272
100 000	683	532	579	508	440	505	534	529
500 000	4221	3333	3100	3256	3007	2908	2857	2760
1 000 000	8211	6857	7001	6293	6939	6499	6301	6389



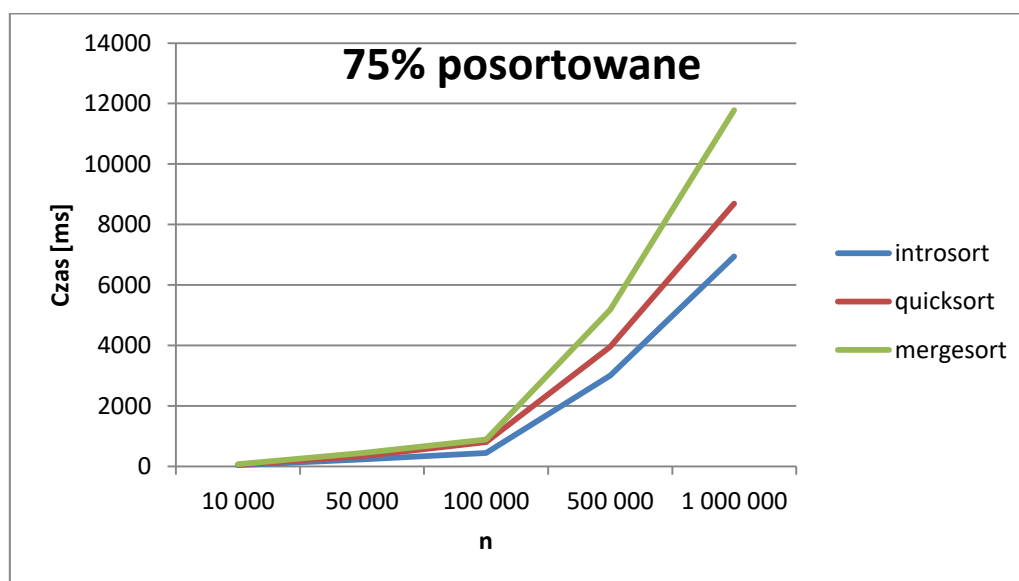
Wykres 3. Wyniki pomiarów czasu algorytmu sortowania introspektywnego

4. Sprawdzenie poprawności eksperymentu

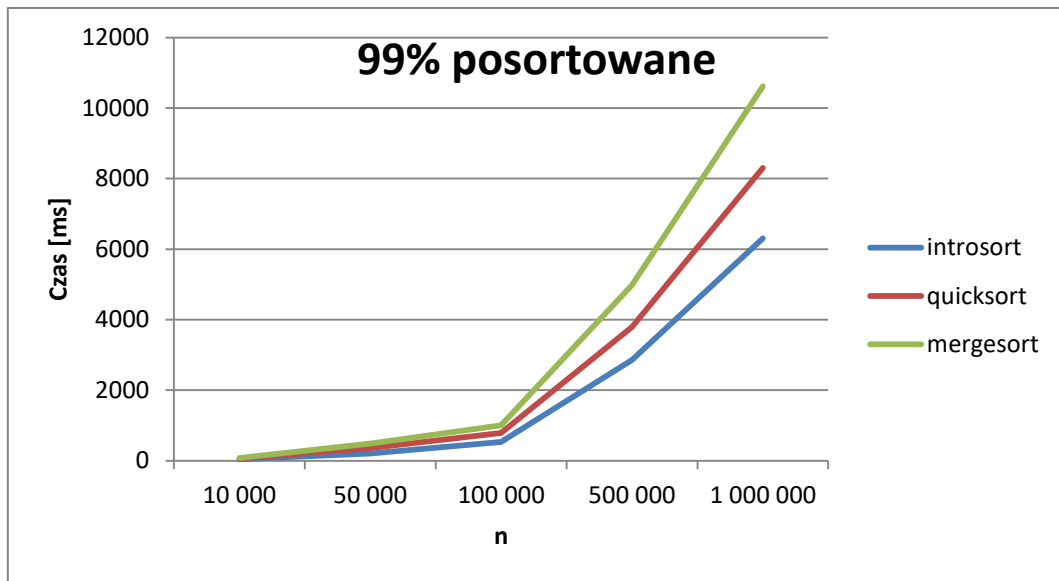
Dla wybranych przypadków posortowania tablic porównano wszystkie 3 typy sortowania:



Wykres 4. Porównanie 3 algorytmów dla przypadku, kiedy 25% początkowych elementów tablicy jest już posortowane



Wykres 5. Porównanie 3 algorytmów dla przypadku, kiedy 75% początkowych elementów tablicy jest już posortowane



Wykres 6. Porównanie 3 algorytmów dla przypadku, kiedy 99% początkowych elementów tablicy jest już posortowane

5. Wnioski

Zgodnie z przewidywaniami najskuteczniejszym typem sortowania okazał się algorytm sortowania introspektywnego, którego złożoność obliczeniowa dla najgorszego przypadku wynosi $n \log n$. Wykazał się on we wszystkich testach najkrótszym czasem sortowania każdej wielkości tablicy.

Drugi najbardziej optymalny algorytm to quicksort, który jest częścią sortowania introspektywnego. Dlatego czas jego działania jest dłuższy. Można to też zauważyć po jego złożoności obliczeniowej, która dla najgorszego przypadku wynosi n^2 .

Najgorszym w eksperymencie okazał się algorytm sortowania przez scalanie, który wykazuje się tą samą złożonością obliczeniową dla każdego przypadku i wynosi ona $n \log n$.

6. Źródła

https://pl.wikipedia.org/wiki/Sortowanie_introspektywne

https://pl.wikipedia.org/wiki/Sortowanie_szybkie

https://pl.wikipedia.org/wiki/Sortowanie_przez_kopcowanie

<http://www.algorytm.org/algorytmy-sortowania/sortowanie-szybkie-quicksort.html>

<http://www.algorytm.edu.pl/algorytmy-maturalne/sortowanie-przez-scalanie.html>