Projektowanie Algorytmów i Metody Sztucznej Inteligencji				
Projekt 1	Prowadzący	Mgr inż. Marta Emirsajłow	Termin	Pt, 7:30 – 9:00
	Wykonał	Amadeusz Janiszyn 249013	Data	27.03.2020

1. Wprowadzenie

Celem projektu była implementacja i analiza wybranych algorytmów sortowania. Testowanie algorytmów zostało przeprowadzone dla 100 tablic o rozmiarach: *10 000, 50 000, 100 000, 500 000* oraz *1 000 000,* przy następujących przypadkach posortowania tablic: *wszystkie elementy losowe, 25%, 50%, 75%, 95%, 99%, 99,7% uporządkowania, tablica posortowana w odwrotnej kolejności.*

2. Badane algorytmy

- QuickSort (Sortowanie szybkie)
  Na początku wybierany jest tzw. element osiowy. Następnie tablica dzielona jest na dwie podtablice. Pierwsza z nich zawiera elementy mniejsze od elementu osiowego, druga elementy większe lub równe, element osiowy znajdzie się między nimi. Proces dzielenia powtarzany jest aż do uzyskania tablic jednoelementowych, nie wymagających sortowania. Właściwe sortowanie jest tu jakby ukryte w procesie przygotowania do sortowania. Wybór elementu osiowego wpływa na równomierność podziału na podtablice (najprostszy wariant – wybór pierwszego elementu tablicy – nie sprawdza się w przypadku, gdy tablica jest już prawie uporządkowana).
  Złożoność obliczeniowa:
  Przypadek najlepszy: $\boldsymbol{O(n\ log\ n)}$
  Średni przypadek: $\boldsymbol{O(n\ log\ n)}$
  Najgorszy przypadek: $\boldsymbol{O(n^2)}$

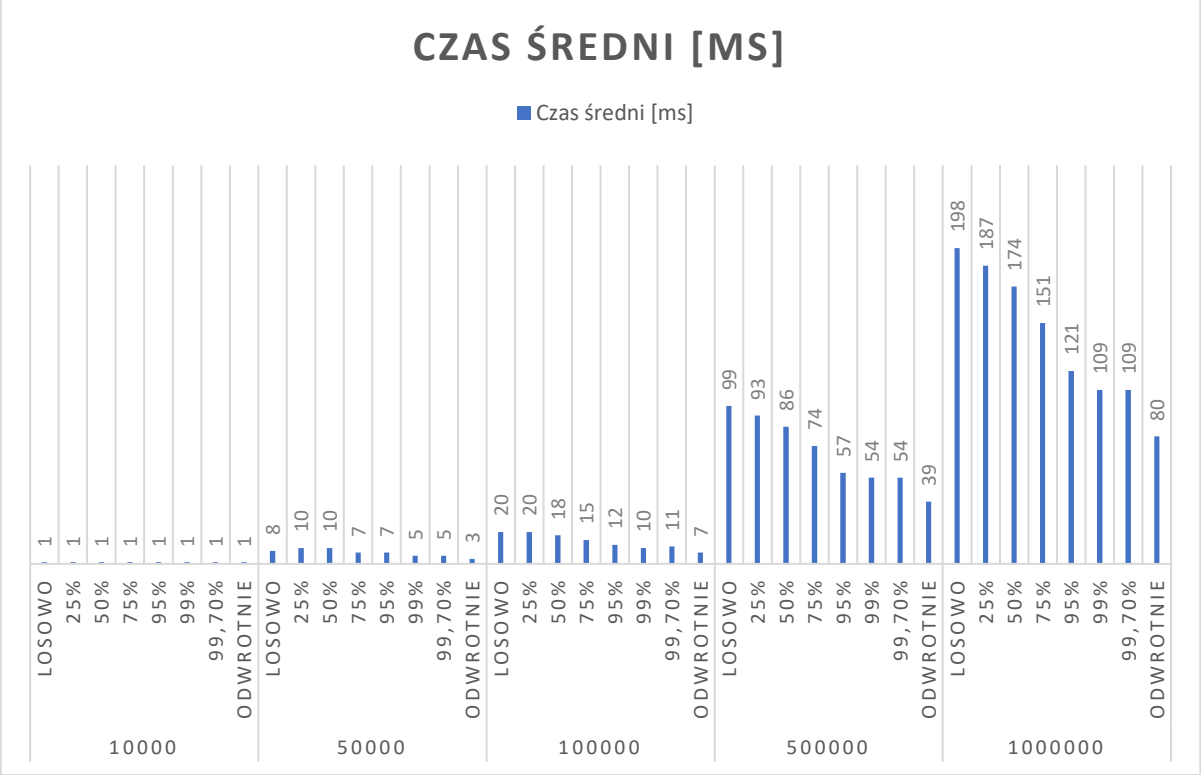
- MergeSort (Sortowanie przez scalanie)
  Sortowana tablica dzielona jest rekurencyjnie na dwie podtablice aż do uzyskania tablic jednoelementowych. Następnie podtablice te są scalane w odpowiedni sposób, dający w rezultacie tablicę posortowaną. Wykorzystana jest tu metoda podziału problemu na mniejsze, łatwiejsze do rozwiązania zadania („dziel i rządź").
  Złożoność obliczeniowa:
  Przypadek najlepszy: $\boldsymbol{O(n\ log\ n)}$
  Średni przypadek: $\boldsymbol{O(n\ log\ n)}$
  Najgorszy przypadek: $\boldsymbol{O(n\ log\ n)}$

- IntroSort (Sortowanie introspektywne)
  Jest to metoda hybrydowa, będąca połączeniem sortowania szybkiego i sortowania przez kopcowanie. Sortowanie introspektywne pozwala uniknąć najgorszego przypadku dla sortowania szybkiego (nierównomierny podział tablicy w przypadku, gdy jako element osiowy zostanie wybrany element najmniejszy lub największy).
  Złożoność obliczeniowa:
  Przypadek najlepszy: $\boldsymbol{O(n\ log\ n)}$
  Średni przypadek: $\boldsymbol{O(n\ log\ n)}$
  Najgorszy przypadek: $\boldsymbol{O(n\ log\ n)}$

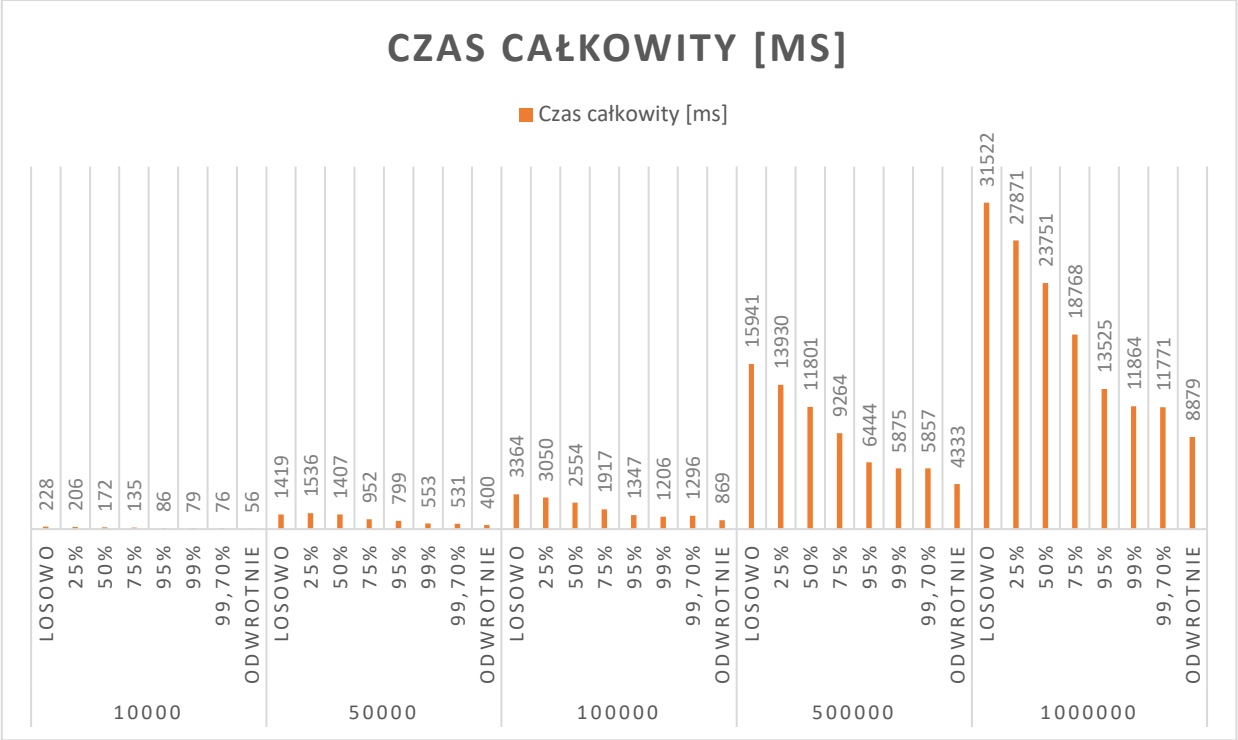
3. Quicksort

```cpp
/*
    Quicksort
*/

template <typename Item>
void Quicksort(Item* Arr, int left, int right) {
    if (right <= left) return;

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

    while (true) {
        while (pivot < Arr[--j]);
        while (pivot > Arr[++i]);

        if (i <= j) {
            Item temp = Arr[i];
            Arr[i] = Arr[j];
            Arr[j] = temp;
        }
        else break;
    }

    if (j > left) Quicksort(Arr, left, j);
    if (i < right) Quicksort(Arr, i, right);
}
```

Kod 1. Implementacja algorytmu Quicksort.



Wykres 1. Przedstawienie średniego czasu sortowania [w ms] dla zbioru tablic różnego rozmiaru, oraz sposobu uporządkowania.



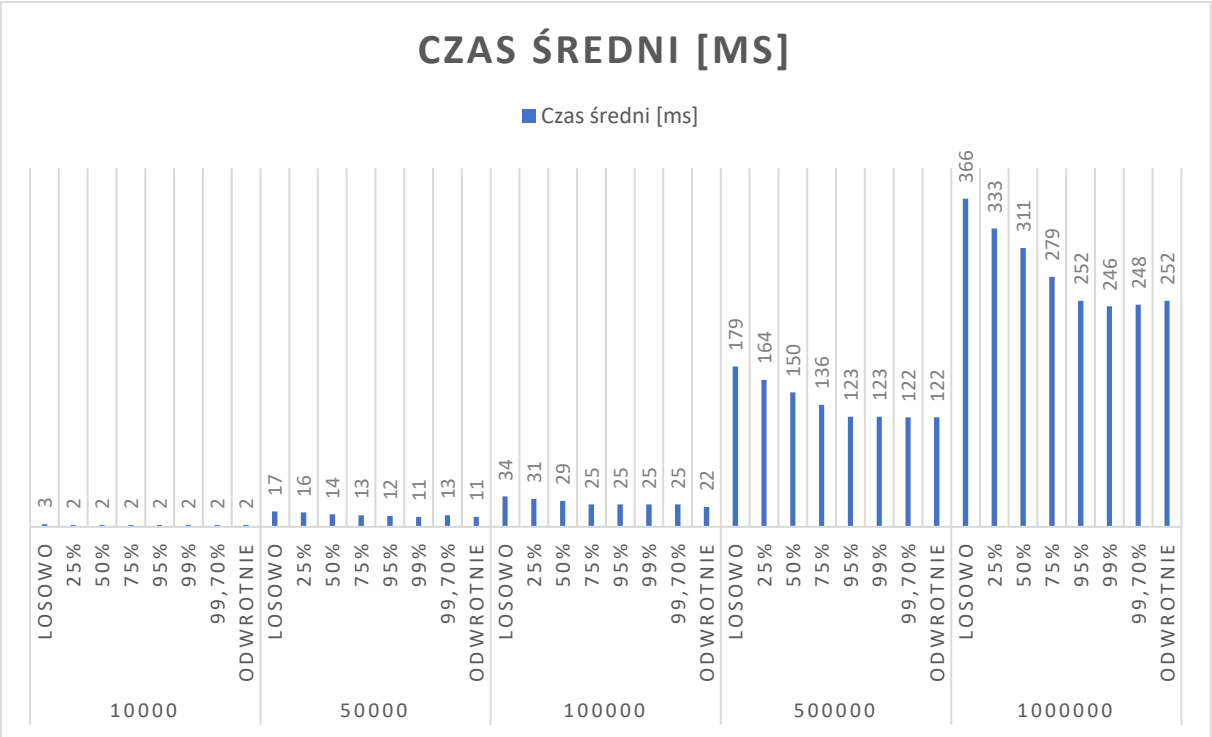
Wykres 2. Przedstawienie całkowitego czasu sortowania [w ms] dla zbioru 100 tablic różnych rozmiarów, oraz sposobu uporządkowania.

Dla najmniejszego rozmiaru tablic, nie można zaobserwować różnicy w średnim czasie, ponieważ jest on rzędu mikrosekund, natomiast można zauważyć różnicę w całkowitym czasie sortowania. Dla tablic 500 000 i 1 000 000 elementowych widać różnicę w średnim i całkowitym czasie sortowania. Uporządkowanie losowe przyczynia się do najdłuższego czasu sortowania, uporządkowanie odwrotne jest najszybciej sortowane przez algorytm. Dla tablic mniejszych można zauważyć podobieństwo do wniosków powyżej, choć nie w każdym przypadku.

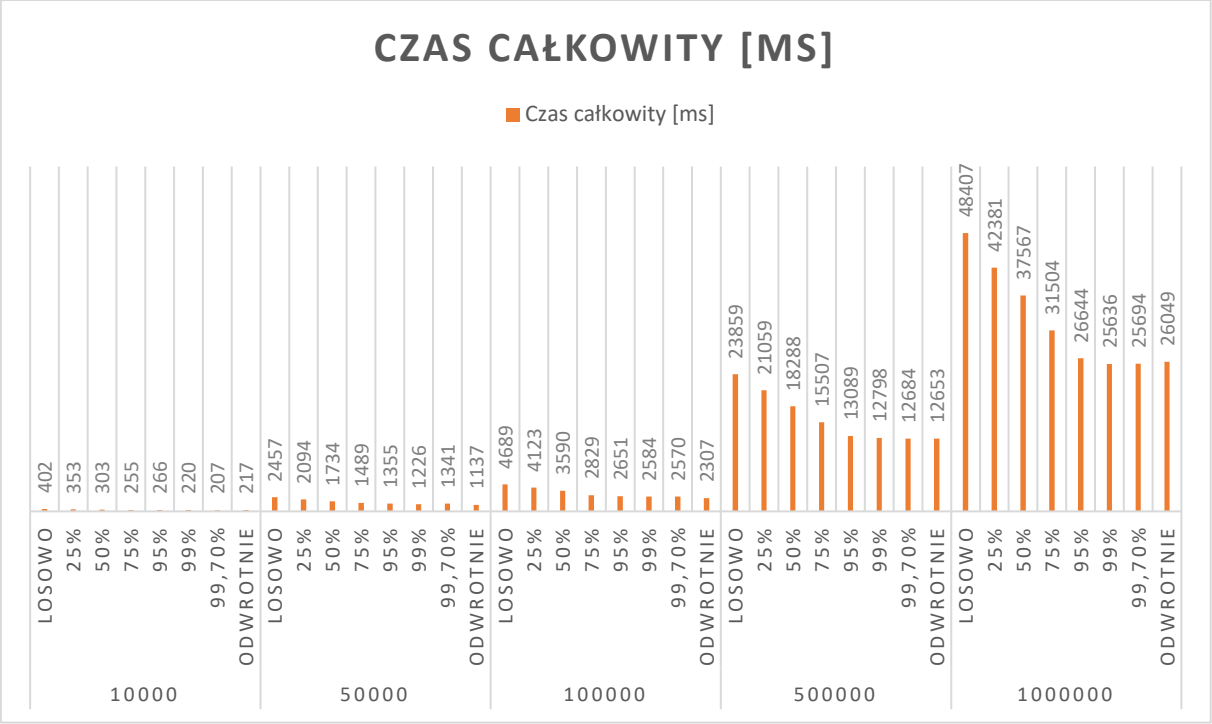
4. Mergesort

```cpp
/*
    Mergesort
*/

template <typename Item>
void Merging (Item* Arr, Item * tArr, int left, int mid, int right){
    int i = left, j = mid + 1;

    for (int i = left; i <= right; i++) {
        tArr[i] = Arr[i];
    }

    for (int k = left; k <= right; k++) {
        if (i <= mid) {
            if (j <= right) {
                if (tArr[j] < tArr[i]) Arr[k] = tArr[j++];
                else Arr[k] = tArr[i++];
            }
            else Arr[k] = tArr[i++];
        }
        else Arr[k] = tArr[j++];
    }
}

template <typename Item>
void Mergesort(Item* Arr, Item* tArr, int left, int right) {
    if (right <= left) return;

    int mid = (left + right) / 2;
    Mergesort(Arr, tArr, left, mid);
    Mergesort(Arr, tArr, mid, right);
    Merging(Arr, tArr, left, mid, right);
}
```

Kod 2. Implementacja algorytmu Mergesort.



Wykres 3. Przedstawienie średniego czasu sortowania [w ms] dla zbioru tablic różnego rozmiaru, oraz sposobu uporządkowania.



Wykres 4. Przedstawienie całkowitego czasu sortowania [w ms] dla zbioru 100 tablic różnych rozmiarów, oraz sposobu uporządkowania.

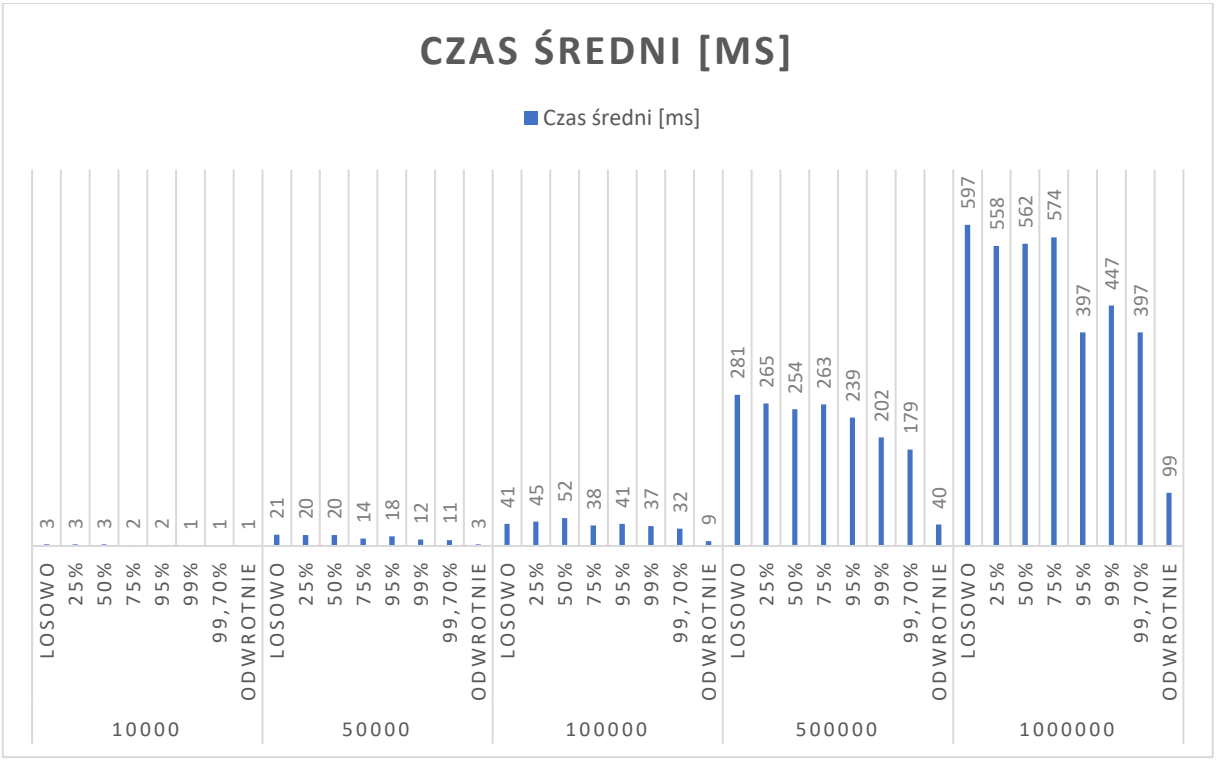
Algorytm Mergesort jest wolniejszy od Quicksort. Jednak można zauważyć, że dla tablicy posortowanej odwrotnie znów czas sortowania, średni jak i ten całkowity, jest najmniejszy.

5. Introsort

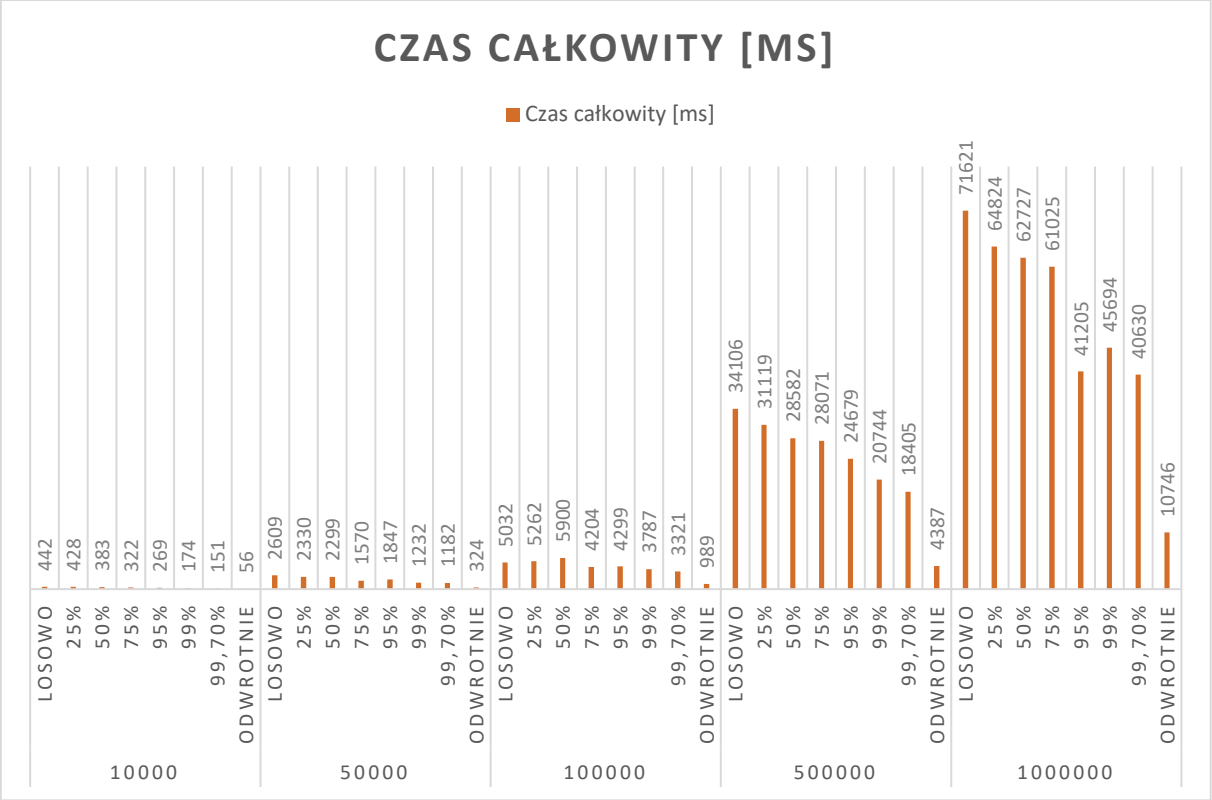
```cpp
/*
    INSERTIONSORT
*/

template <typename Item>
void Insertionsort(Item* Arr, int left, int right) {
    right++;
    int i = left;
    while (i < right) {
        int j = i;
        while (j > 0 && Arr[j - 1] > Arr[j]) {
            Swap(Arr, j, j - 1);
            --j;
        }
        ++i;
    }
}

/*
    HEAPSORT
*/

template <typename Item>
void Heapify(Item* Arr, int size, int root) {
    int largest = root;
    int left = 2 * root + 1;
    int right = 2 * root + 2;

    if (left < size && Arr[left] > Arr[largest]) largest = left;
    if (right < size && Arr[right] > Arr[largest]) largest = right;
    if (largest != root) {
        Swap(Arr, root, largest);
        Heapify(Arr, size, largest);
    }
}
```

```cpp
template <typename Item>
void Heapsort(Item* Arr, int left, int right) {
    ++right;

    Item* temp = new Item[right - left];
    for (int i = 0; i < right - left; i++) {
        temp[i] = Arr[i + left];
    }

    for (int i = (right - left) / 2 - 1; i >= 0; --i) {
        Heapify(temp, (right - left), i);
    }

    for (int i = (right - left) - 1; i >= 0; --i) {
        Swap(temp, 0, i);
        Heapify(temp, i, 0);
    }

    for (int i = 0; i < right - left; i++) {
        Arr[i + left] = temp[i];
    }

    delete[] temp;
}

/*
    Partition method
*/

template <typename Item>
int Partition(Item* Arr, int left, int right) {
    Item pivot = Arr[(left + right) / 2];

    int i = left - 1;
    int j = right + 1;

    while (1) {
        do ++i;
        while (Arr[i] < pivot);

        do --j;
        while (Arr[j] > pivot);

        if (i >= j) return j;
        Swap(Arr, i, j);
    }
}
```

```cpp
/*
    INTROSORT
*/

template <typename Item>
void IntroProcedure(Item* Arr, int left, int right, int maxdepth) {
    int size = right - left + 1;

    if (size <= 16) Insertionsort(Arr, left, right);
    else if (maxdepth == 0) Heapsort(Arr, left, right);
    else if (left < right) {
        int pivot = Partition(Arr, left, right);
        IntroProcedure(Arr, left, pivot, --maxdepth);
        IntroProcedure(Arr, pivot + 1, right, --maxdepth);
    }
}

template <typename Item>
void Introsort(Item* Arr, int left, int right) {
    int maxdepth = (int)log(right - left + 1) * 2;
    IntroProcedure(Arr, left, right, maxdepth);
}
```



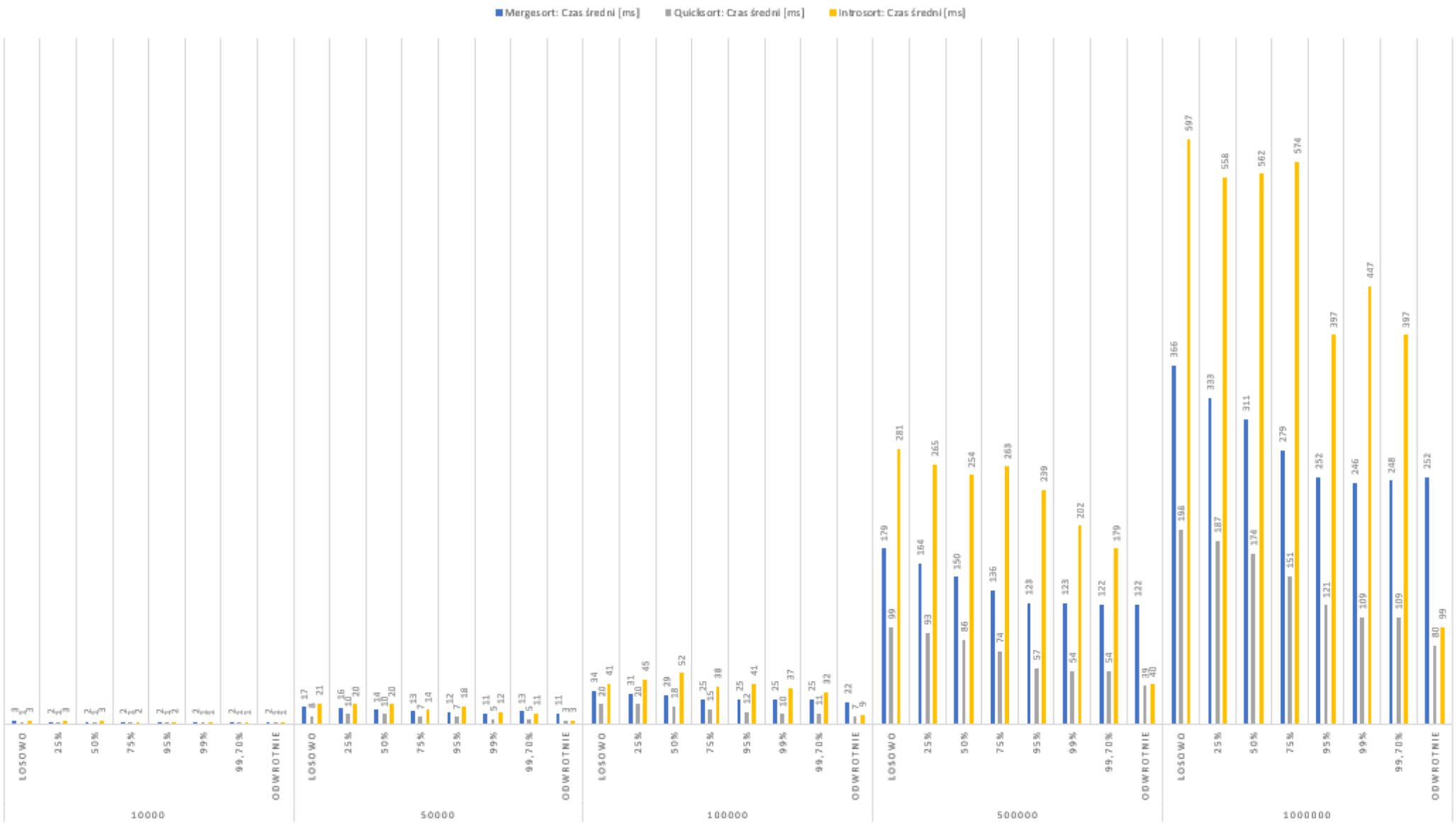
Wykres 5. Przedstawienie średniego czasu sortowania [w ms] dla zbioru tablic różnego rozmiaru, oraz sposobu uporządkowania.



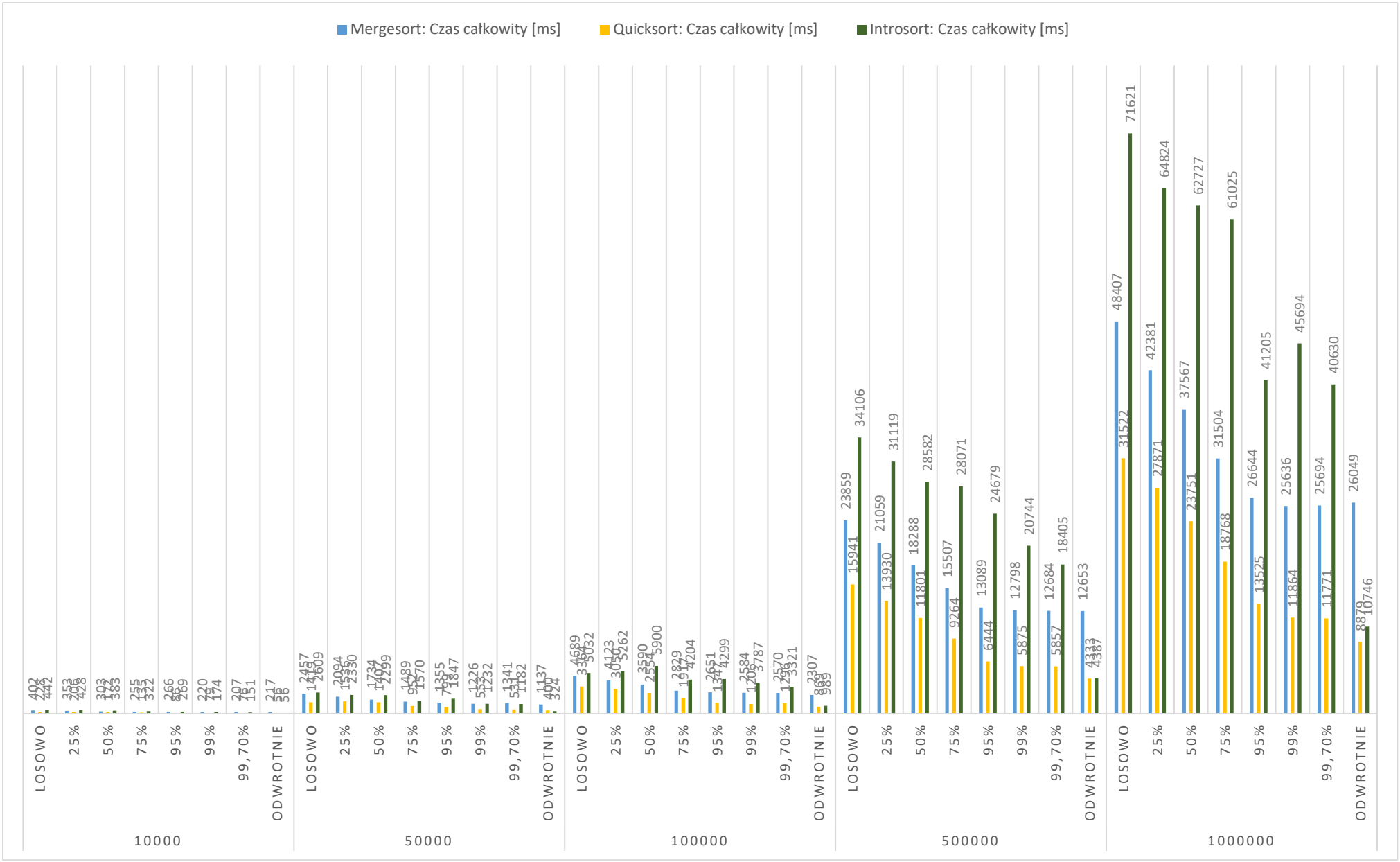
Wykres 6. Przedstawienie całkowitego czasu sortowania [w ms] dla zbioru 100 tablic różnych rozmiarów, oraz sposobu uporządkowania

Algorytm Introsort powinien być szybszy od algorytmu Quicksort, ponieważ dla każdego przypadku jego złożoność obliczeniowa jest taka sama. Na wykresach można zaważyć bardzo dużą rozbieżność między przypadkiem posortowania w 99,7 % a posortowaną odwrotnie tablicą.

6. Porównanie wyników



Wykres 7. Porównanie średniego czasu sortowania dla użytych algorytmów.



Wykres 8. Porównanie całkowitego czasu sortowania dla użytych algorytmów.

7. Wnioski

W algorytmie Introsort, który jest połączeniem 2 algorytmów sortowania, w zależności od głębokości i rozmiaru podtablic, wykorzystuje różne algorytmy sortujące. W tym przypadku, podczas implementacji musiały zajść pewne błędy, których wynik można zauważyć na wykresie. Introsort powinien być szybszy w każdym przypadku od algorytmu Quicksort. Algorytm sortowania Introspektywnego, jest skomplikowanym algorytmem. W trakcie implementacji można popełnić wiele błędów, które zwiększają czas obliczeniowy.

Program który napisałem, posiada funkcję, która automatycznie przekazuje rozmiar tablic oraz stopień posortowania, żeby użytkownik nie musiał tego ręcznie wpisywać. Zaimplementowałem również zapisywanie czasu do pliku; Program w trakcie działania sprawdza czy dana tablica została dobrze posortowana, i po tym sprawdzeniu, automatycznie tworzy nowy plik i zapisuje do niego średni i całkowity czas sortowania oraz zapisuje plik zgodnie ze schematem: *Nazwa_algorytmu*.*Rodzaj_inicjalizacji_tablicy*.*Rozmiar_tablicy.txt.*