Projektowanie Algorytmów i Metod Sztucznej Inteligencji

ALGORYTMY SORTOWANIA

Termin:
wtorek 15:15

Autor:
Szymon Zajda 248917

Prowadzący:
mgr inż. Marcin Ochman

14 kwietnia 2020

1. Wstęp

Celem zadania było napisanie i zaimplementowanie trzech wybranych algorytmów sortowania oraz przetestowanie ich pod względem efektywności. W tym wypadku są to algorytmy*: quicksort, mergesort, introsort.* Dla stu tablic typu całkowitoliczbowego o następujących rozmiarach: 10000, 50000, 100000, 500000, 1000000 miały zostać przeprowadzone testy według poniższych kryteriów:

- wszystkie elementy tablicy są losowe
- 25%, 50%, 75%, 95%, 99%, 99,7% początkowych elementów tablicy jest już posortowanych
- wszystkie elementy są posortowane, ale w odwrotnej kolejności.

2. Implementacja

Cały program składa się dwóch plików nagłówkowych oraz jednego pliku źródłowego. Implementacja tablicy oraz metod sortowania umieszczone zostały w pliku array.h. Obsługa timera, czyli pomiaru czasu wykonywania się algorytmów sortowania została zaimplementowana z w pliku timer.h. Do obsługi tego podprogramu wykorzystano bibliotekę <chrono>.

Inicjalizacja tablicy realizuje się dzięki metodzie *Array::SetArray().* W zależności od podania parametru funkcji otrzymujemy.

- liczba od 0 do 100 – ilość posortowania początkowych elementów tablicy wyrażone w procentach,
- liczba ujemna – tablica posortowana w odwrotnej kolejności.

Metoda *isSorted()* zwraca wartość *true* w przypadku prawidłowego posortowania tablicy, w przeciwnym wypadku wartość *false.*

Wszystkie algorytmy wywoływane są w pliku main.cpp, gdzie również odbywa się pomiar czasu sortowania. Stworzenie 100 tablic w tym wypadku 100 wskaźników umożliwia biblioteka <vector>.

3. Algorytmy sortujące

3.2. Quicksort (sortowanie szybkie)

Algorytm wykorzystuje technikę "dziel i zwyciężaj". Według ustalonego schematu wybierany jest jeden element w sortowanej tablicy, który będziemy nazywać pivot. Operacja znajdowania pivot'a zaimplementowana jest w metodzie *Array::partition*. Następnie elementy nie większe ustawiane są na lewo tej wartości, natomiast nie mniejsze na prawo. W ten sposób powstaną dwie części tablicy, gdzie w pierwszej części znajdują się elementy nie większe od drugiej. Następnie każdą z tych podtablic sortujemy osobno według tego samego schematu. W momencie gdy każda z najmniejszych tablic będzie już posortowana program składa je z powrotem w większą tablicę już posortowaną. Algorytm sortowania szybkiego znajduje się w metodzie *Array::quicksort.* Złożoność obliczeniowa algorytmu zależy od wyboru elementu osiowego. W najlepszym przypadku gdy tablice są zrównoważone złożoność takiego algorytmu jest równa $O(n \cdot \log n)$. W przypadku pesymistycznym złożoność taka wynosi $O(n^2)$.

3.3. Mergesort (sortowanie przez scalanie)

Ideą działania algorytmu jest dzielenie zbioru danych na mniejsze zbiory, aż do uzyskania n zbiorów jednoelementowych, następnie zbiory te są łączone w coraz większe zbiory posortowane, aż do uzyskania jednego, posortowanego zbioru. Metoda *Array::merge* dzieli oraz sortuje tablice a cały algorytm sortowania jest rekurencyjnie opisany w metodzie *Array::mergesort*. Sortowanie przez scalanie w odróżnieniu od sortowania szybkiego posiada stałą złożoność obliczeniową O(n·logn).

3.4. Introsort (sortowanie introspektywne)

Algorytm sortowania introspektywnego jest rodzajem hybrydy, czyli połączeniem sortowania szybkiego, sortowania przez kopcowanie oraz sortowania przez wstawianie. Algorytm dzieli tablice jak pozostałych algorytmach sortowania do pewnej ilości wywołań określonych przez stałą $2 \cdot \log_2$(rozmiar tablicy). Jeśli stała jest równa 0 kończy się dzielenie a następnie obsługiwane są dwa przypadki. Pierwszy, gdy algorytm dojdzie do maksymalnej głębokości wykonuje sortowanie przez kopcowanie o złożoności $O(n \cdot \log_2 n)$ metodo - *Array::heapsort.* Drugi, gdy podtablica zawiera mniej niż 16 elementów wykonuje sortowanie przez wstawianie *Array::insertionsort.* Jeśli stała jest większa od 0 wówczas introsort zachowuje się jak sortowanie szybkie. W najgorszym wypadku takie sortowanie będzie miało złożoność równą sortowaniu szybkiemu $O(n^2)$.

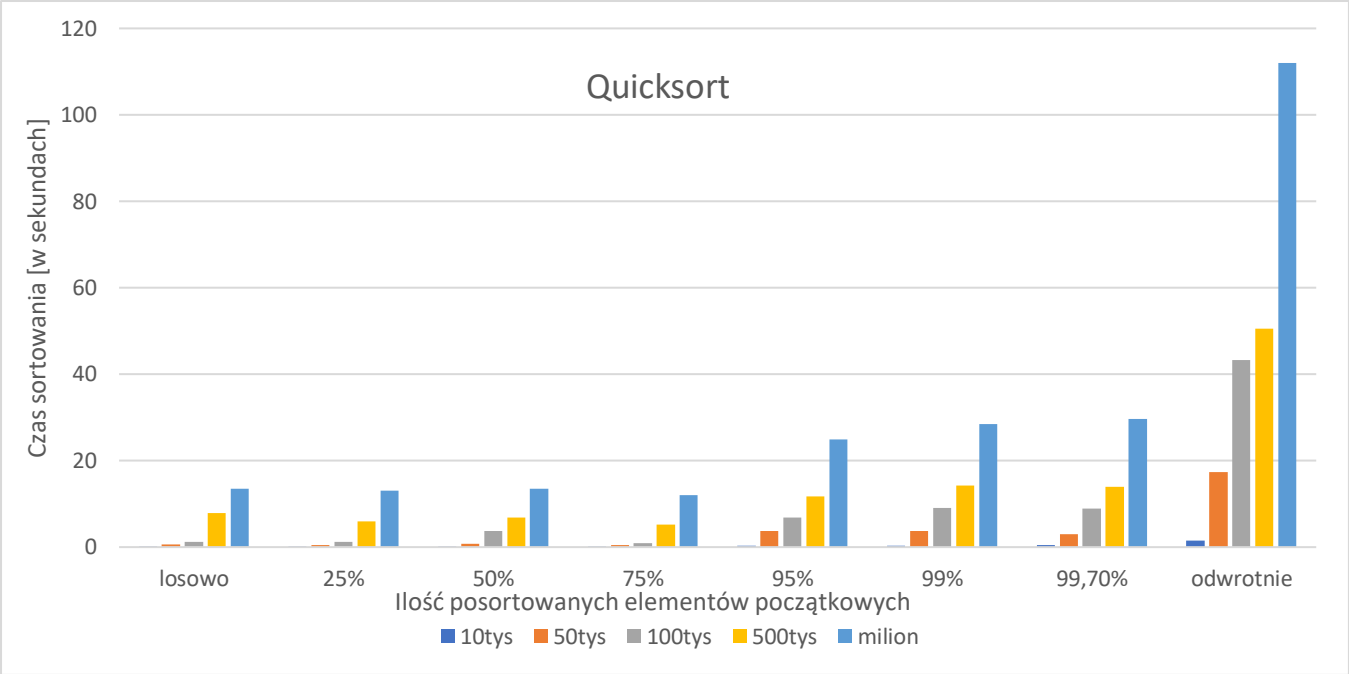
4. Przebieg testów oraz wyniki analiz

4.1. Analiza poszczególnych algorytmów

Poniżej zaprezentowane wykresy zawierają analizę dla poszczególnych algorytmów. Każdy z nich reprezentuje czas sortowania w zależności od procentowej zawartości elementów posortowanych.

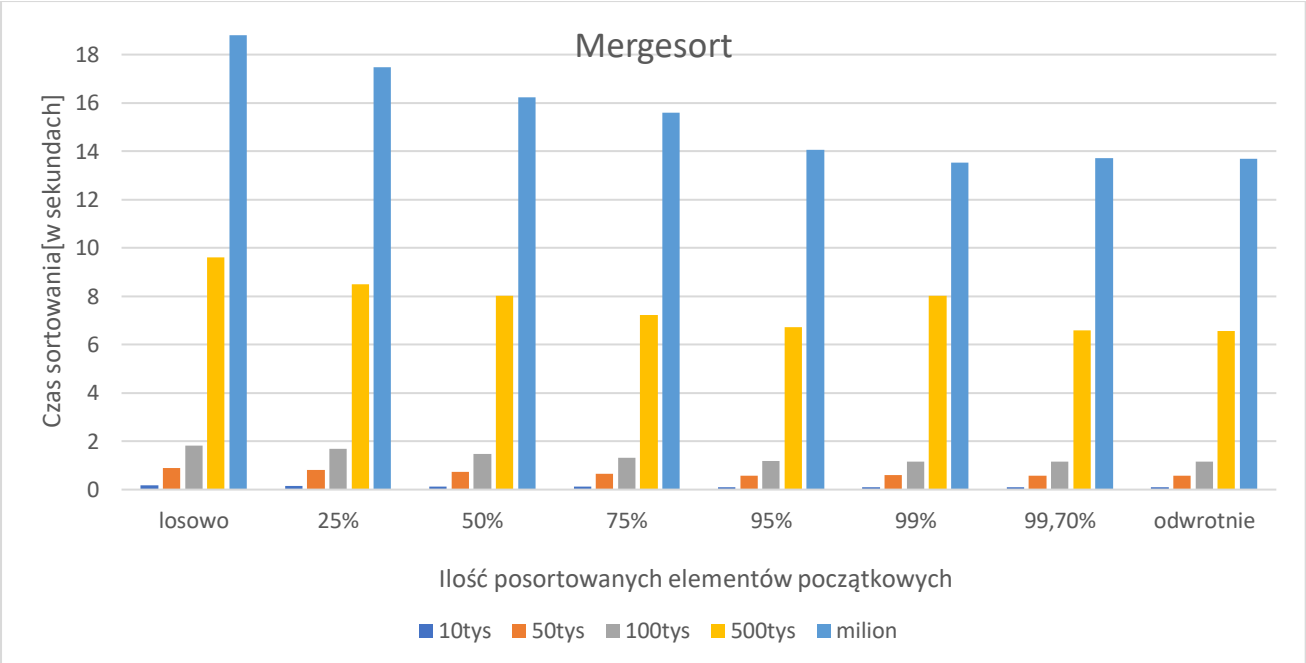
4.1.1. Quicksort(sortowanie szybkie)

Pojemność	losowo	25%	50%	75%	95%	99%	99,70%	odwrotnie
10tys	0,113182	0,0985985	0,092635	0,085971	0,314476	0,366005	0,416277	1,56368
50tys	0,574247	0,543601	0,760774	0,463457	3,69497	3,70188	3,06261	17,42
100tys	1,22908	1,15223	3,67035	0,979838	6,86698	9,06123	8,86831	43,2481
500tys	7,9109	5,97334	6,79626	5,27367	11,6600	14,187	13,9765	50,4994
milion	13,4435	13,0926	13,4552	12,0108	24,8526	28,4286	29,7164	111,982



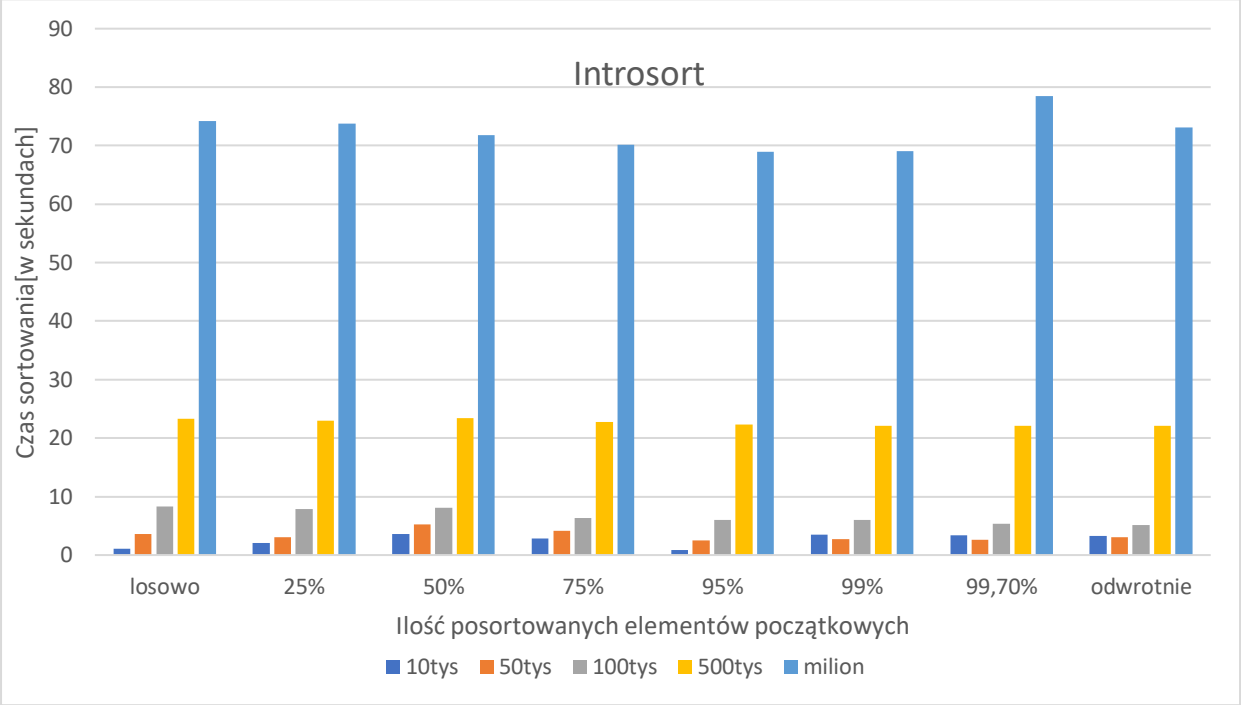
4.1.2. Mergesort(sortowanie przez scalanie)

Pojemność	losowo	25%	50%	75%	95%	99%	99,70%	odwrotnie
10tys	0,1633	0,1461	0,1310	0,1144	0,1013	0,0992	0,0986	0,1001
50tys	0,8863	0,8114	0,7248	0,6439	0,5785	0,5933	0,5617	0,5627
100tys	1,8229	1,6754	1,4819	1,3203	1,1935	1,1663	1,1621	1,1628
500tys	9,6096	8,4921	8,0207	7,2272	6,7218	8,0087	6,5868	6,5735
milion	18,8081	17,4762	16,2293	15,5991	14,0656	13,5419	13,7185	13,6877



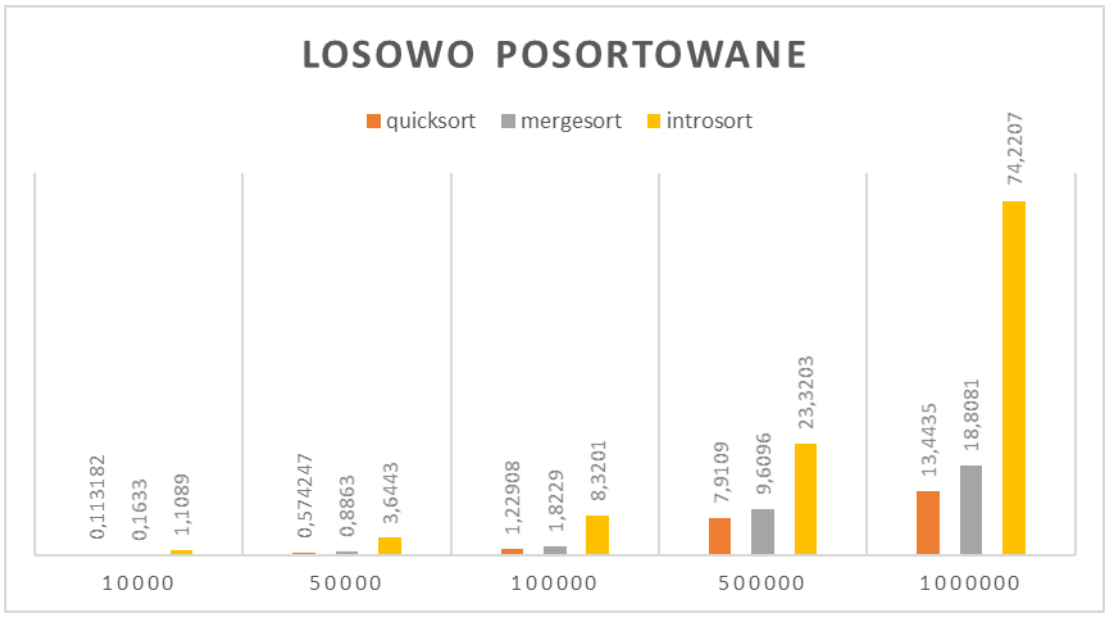
4.1.3. Introsort(sortowanie introspektywne)

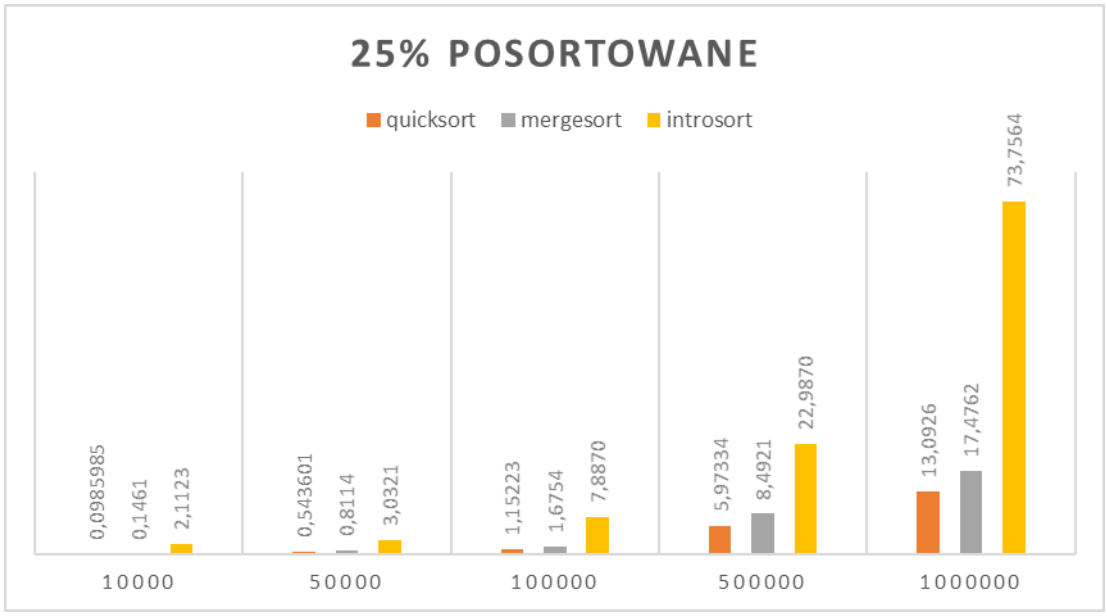
Pojemność	losowo	25%	50%	75%	95%	99%	99,70%	odwrotnie
10000	1,1089	2,1123	3,6425	2,7994	0,8856	3,5004	3,3286	3,2278
50000	3,6443	3,0321	5,2059	4,1154	2,5432	2,7765	2,6541	2,9987
100000	8,3201	7,8870	8,1021	6,3058	6,0189	5,9999	5,3324	5,1209
500000	23,3203	22,9870	23,4449	22,7654	22,3498	22,1109	22,0904	22,1040
1000000	74,2207	73,7564	71,7494	70,1124	69,0023	69,0983	78,4320	73,0777

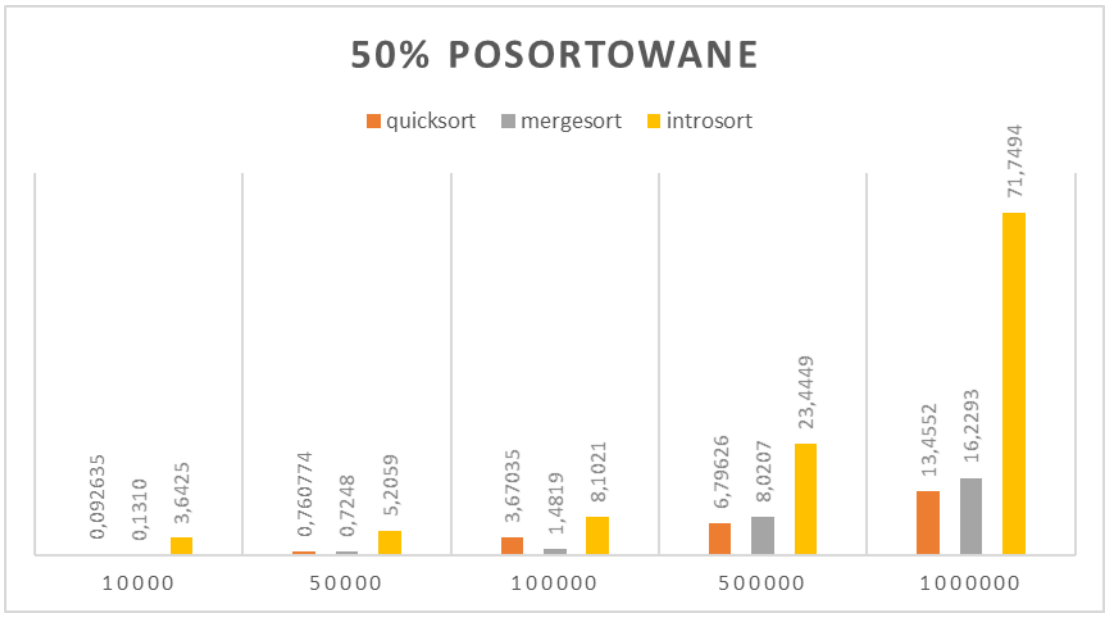


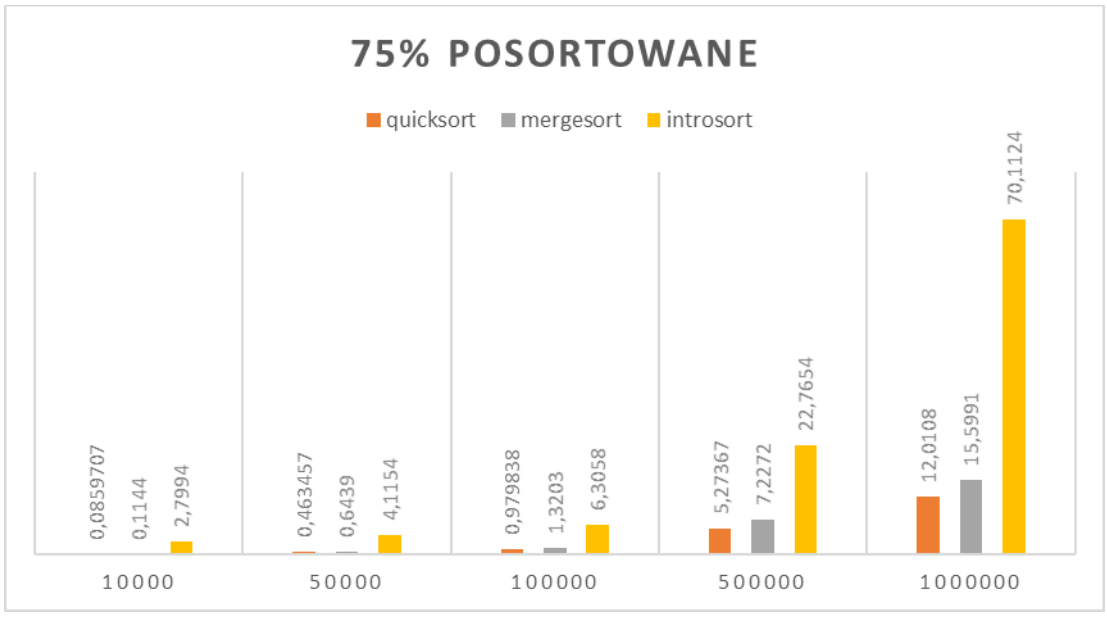
4.2. Analiza algorytmów względem wstępnego posortowania

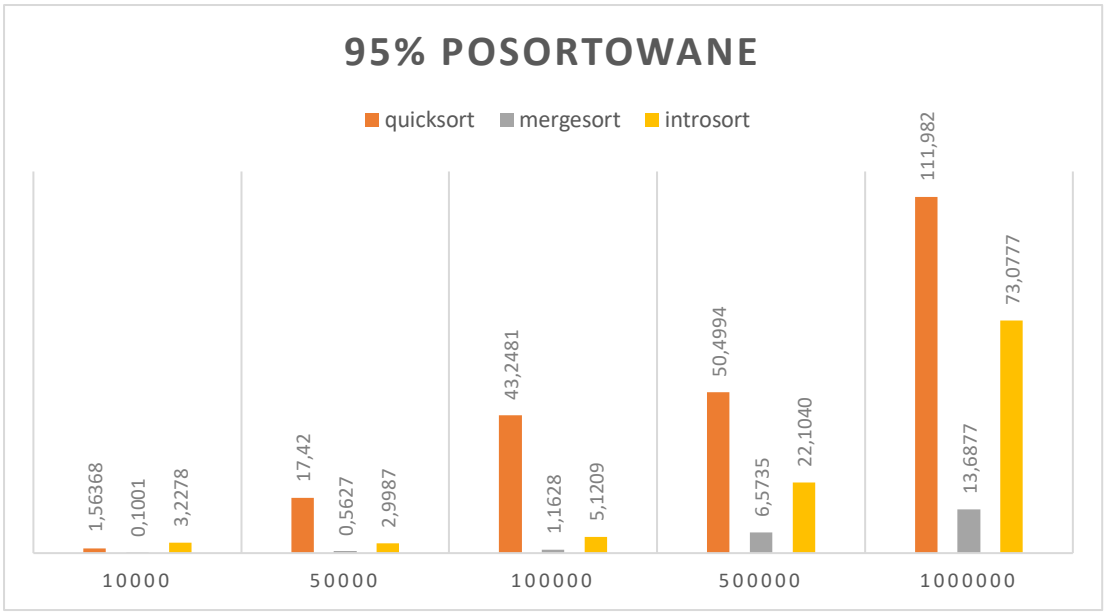
Poniższe wykresy ilustrują czas wykonywania się algorytmu względem wstępnego względnego posortowania.

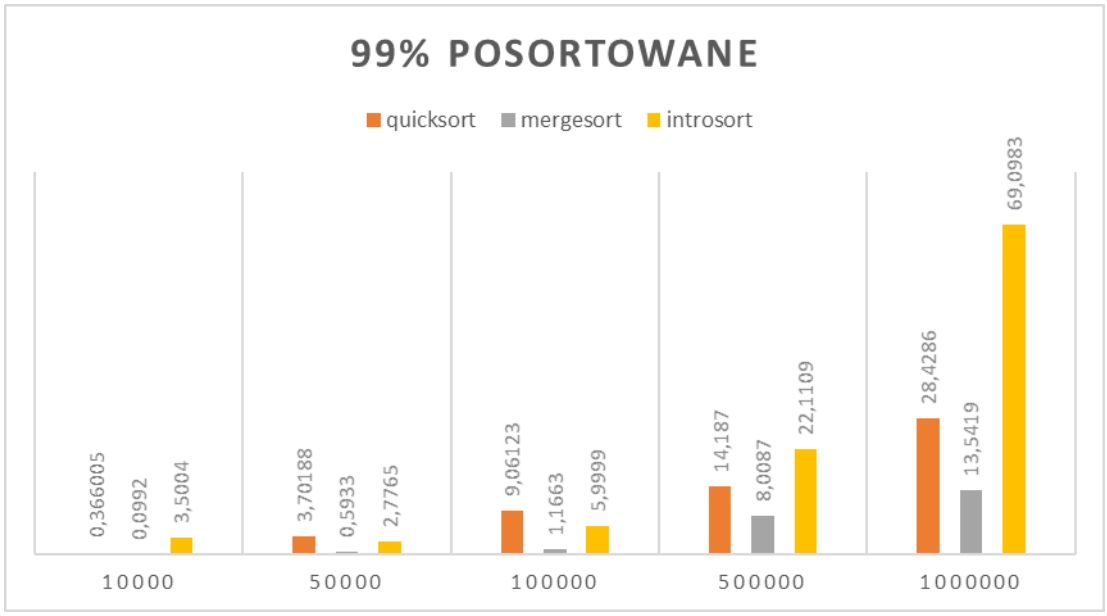

LOSOWO POSORTOWANE
quicksort
mergesort
introsort
0,113182
0,1633
1,1089
0,574247
0,8863
3,6443
1,22908
1,8229
8,3201
7,9109
9,6096
23,3203
13,4435
18,8081
74,2207
10000
50000
100000
500000
1000000

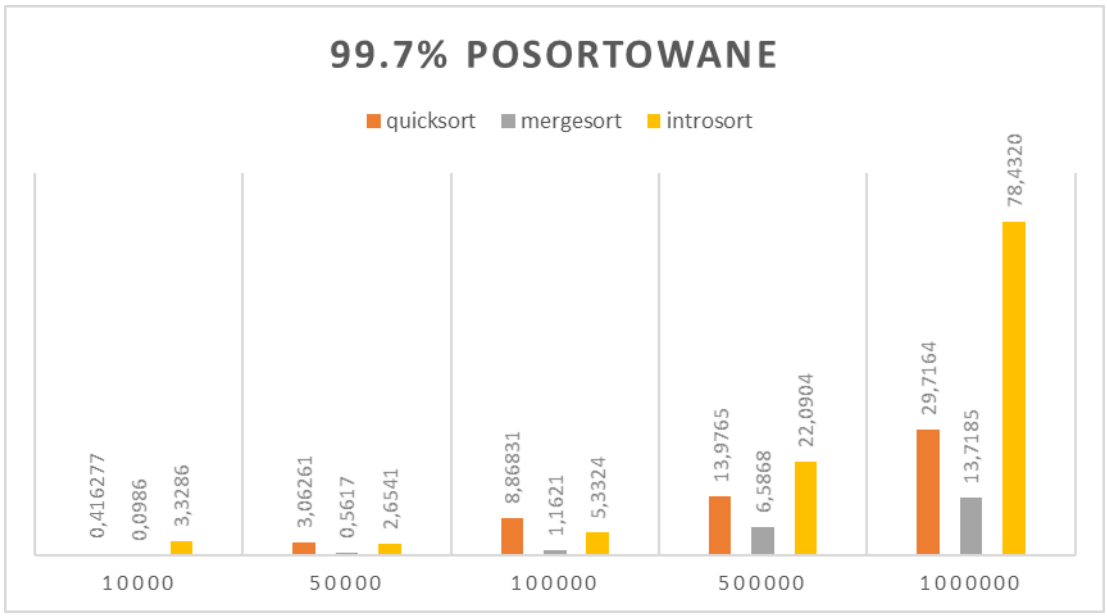

25% POSORTOWANE
quicksort
mergesort
introsort
0,0985985
0,1461
2,1123
0,543601
0,8114
3,0321
1,15223
1,6754
7,8870
5,97334
8,4921
22,9870
13,0926
17,4762
73,7564
10000
50000
100000
500000
1000000

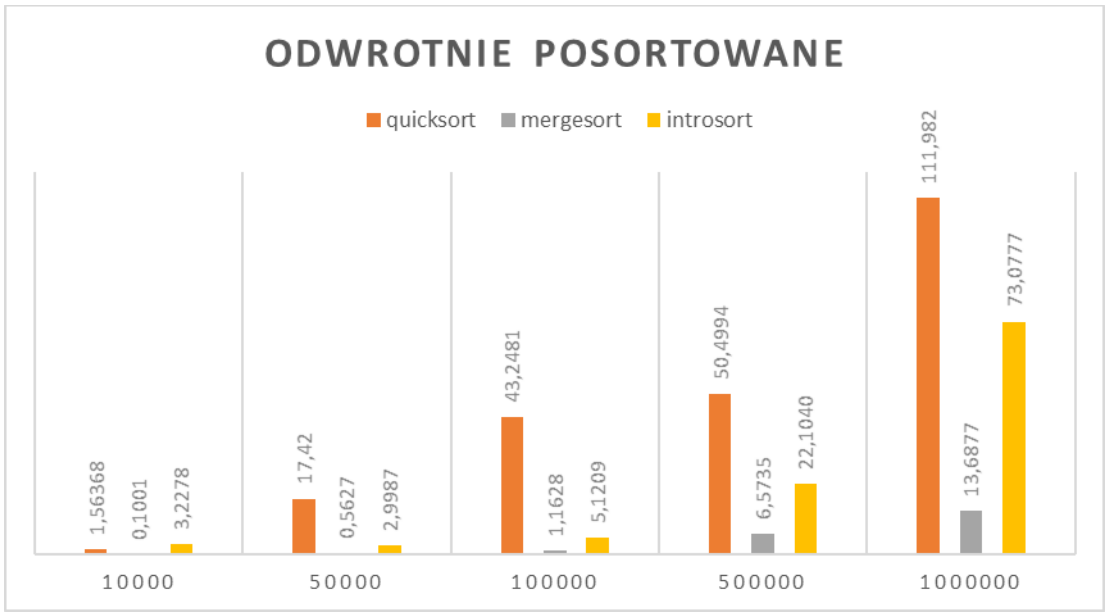

50% POSORTOWANE
quicksort
mergesort
introsort
0,092635
0,1310
3,6425
0,760774
0,7248
5,2059
3,67035
1,4819
8,1021
6,79626
8,0207
23,4449
13,4552
16,2293
71,7494
10000
50000
100000
500000
1000000


75% POSORTOWANE
quicksort
mergesort
introsort
0,0859707
0,1144
2,7994
0,463457
0,6439
4,1154
0,979838
1,3203
6,3058
5,27367
7,2272
22,7654
12,0108
15,5991
70,1124
10000
50000
100000
500000
1000000


95% POSORTOWANE
quicksort
mergesort
introsort
1,56368
0,1001
3,2278
17,42
0,5627
2,9987
43,2481
1,1628
5,1209
50,4994
6,5735
22,1040
111,982
13,6877
73,0777
10000
50000
100000
500000
1000000


99% POSORTOWANE
quicksort
mergesort
introsort
0,366005
0,0992
3,5004
3,70188
0,5933
2,7765
9,06123
1,1663
5,9999
14,187
8,0087
22,1109
28,4286
13,5419
69,0983
10000
50000
100000
500000
1000000





5. Wnioski

Sortowanie szybkie jest najbardziej optymalne dla losowych elementów tablicy, wówczas jest najdalej od pesymistycznego przypadku.

Sortowanie przez scalanie wykazuje te bardzo podobne czasy dla różnych stopni wstępnego sortowania. To świadczy o stabilności i niezawodności tego algorytmu.

Pomimo swej hybrydowej struktury algorytm introspektywny nie okazał się tak efektywny jak oczekiwano. Duża rozbieżność pomiędzy oczekiwaniami a realnymi wynikami może być spowodowane nieoptymalną implementacją tego algorytmu w programie.