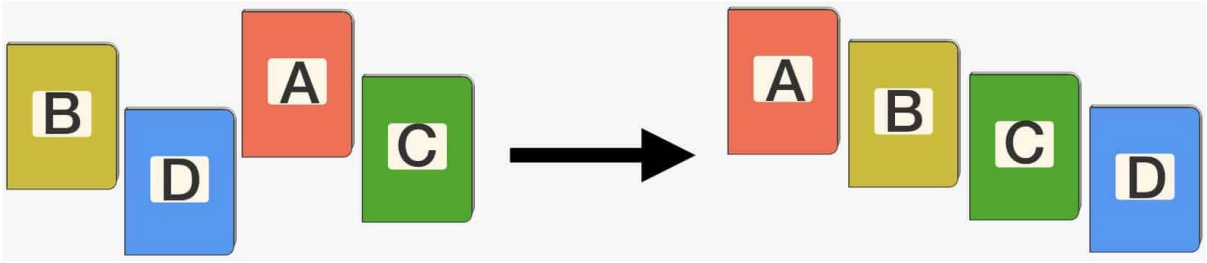
Projektowanie i analiza algorytmów	
Kierunek *Informatyczne Systemy Automatyki*	**Termin** *czwartek 11:15*
Imię, nazwisko, numer albumu *Iwo Chwiszczuk 280043*	**Data** *13 kwietnia 2025*
Link do projektu `https://github.com/iwonieevo/Sorting-algorithms`	

Raport

Implementacja i Analiza Algorytmów Sortowania



Spis treści

Spis tabel

Spis wykresów

1 Streszczenie

Celem projektu była implementacja wybranych algorytmów sortowania oraz analiza ich złożoności czasowej. Zbadano trzy metody: MergeSort (sortowanie przez scalanie), QuickSort (sortowanie szybkie) oraz IntroSort (sortowanie introspektywne). Najbardziej wydajnym algorytmem spośród badanych okazał się IntroSort. QuickSort może osiągać wysoką wydajność dla określonych danych, lecz jego czas wykonania bywa niestabilny. MergeSort jest zazwyczaj wolniejszy, ale zapewnia stabilność wykonania oraz stabilność sortowania (zachowanie kolejności elementów równych).

2 Wstęp

2.1 Opis problemu sortowania

Sortowanie to jedno z fundamentalnych zagadnień informatyki, które ma kluczowe znaczenie w optymalizacji wielu procesów obliczeniowych. Polega na uporządkowaniu zbioru danych według określonego kryterium, np. wartości liczbowych lub kolejności alfabetycznej. Jest to kluczowy problem występujący w wielu obszarach, takich jak bazy danych, algorytmy wyszukiwania czy przetwarzanie dużych zbiorów danych.

2.2 Klasyfikacja algorytmów sortowania

Algorytmy sortowania można klasyfikować według różnych kryteriów, m.in.:

- **Złożoność obliczeniowa** – określa liczbę operacji wykonywanych przez algorytm w zależności od liczby elementów wejściowych. Rozróżnia się przypadek optymistyczny, pesymistyczny oraz średni.
- **Zużycie pamięci** – określa ilość dodatkowej pamięci wymaganej do wykonania algorytmu w zależności od rozmiaru danych wejściowych.
- **Rekurencyjność** – algorytm może być implementowany w sposób rekurencyjny lub iteracyjny.
- **Stabilność** – algorytm jest stabilny, jeśli zachowuje kolejność względną elementów o równych wartościach klucza.

Istnieją również inne klasyfikacje, jednak w tym projekcie analiza skupia się głównie na złożoności obliczeniowej, a w szczególności na czasie wykonania badanych algorytmów.

2.3 Badane algorytmy sortowania

2.3.1 MergeSort

- MergeSort to algorytm sortowania wykorzystujący strategię dziel i rządź *(divide and conquer)*. Polega na rekurencyjnym (lub iteracyjnym) dzieleniu tablicy na mniejsze podtablice, sortowaniu ich, a następnie scalaniu (`merge`) w sposób uporządkowany.
- Pesymistyczna, średnia oraz optymistyczna złożoność wynosi $O(n \log_2 n)$ - algorytm zawsze dzieli sortowaną tablicę na $\log_2 n$ poziomów i na każdym wykonuje $O(n)$ operacji scalania.
- Merge Sort wymaga dodatkowej pamięci o złożoności $O(n)$, ponieważ przechowuje tymczasowe kopie scalanych podtablic. W wersji rekurencyjnej dodatkowo wykorzystywany jest stos rekurencyjny o złożoności $O(\log_2 n)$, ponieważ głębokość rekurencji odpowiada liczbie poziomów podziału tablicy. W wersji iteracyjnej zużycie dodatkowej pamięci jest ograniczone do pamięci wymaganej na tymczasowe kopie podtablic.
- Algorytm jest stabilny, czyli elementy o jednakowej wartości zachowują swoją pierwotną kolejność. W ramach projektu zaimplementowano iteracyjną wersję MergeSort, która eliminuje stos rekurencyjny poprzez iteracyjne scalanie coraz większych fragmentów tablicy, co może poprawić wydajność w praktycznych zastosowaniach.

2.3.2 QuickSort

- QuickSort to algorytm sortowania wykorzystujący strategię dziel i rządź. Polega na rekurencyjnym (lub iteracyjnym) podziale tablicy na mniejsze podtablice względem wybranego elementu zwanego pivotem, a następnie sortowaniu każdej z nich oddzielnie. Podział odbywa się w taki sposób, że elementy mniejsze od pivota są umieszczane po jego lewej stronie, a większe po prawej. Proces ten jest kontynuowany, aż podtablice będą miały tylko jeden element.

- Pesymistyczna złożoność QuickSort wynosi $O(n^2)$, co występuje, gdy pivot dzieli tablicę w sposób bardzo nierównomierny (np. zawsze wybierany jest największy lub najmniejszy element). W średnim i optymistycznym przypadku algorytm działa w czasie $O(n \log_2 n)$, co jest związane z równomiernym podziałem tablicy.

- QuickSort działa w miejscu (in-place), co oznacza, że nie wymaga dodatkowej pamięci poza stosową rekurencją lub pomocniczymi strukturami dla implementacji iteracyjnej. Złożoność pamięciowa wynosi $O(\log_2 n)$ w typowych przypadkach, ponieważ głębokość rekurencji (lub liczba przechowywanych zakresów w wersji iteracyjnej) jest ograniczona do liczby poziomów podziału. W pesymistycznym przypadku może wzrosnąć do $O(n)$, jeśli podziały są skrajnie nierównomierne.

- Algorytm nie jest stabilny, ponieważ podczas podziału tablicy może zmieniać względną kolejność elementów o tej samej wartości.

- W ramach projektu zaimplementowano iteracyjną wersję QuickSort, eliminującą stos rekurencyjny poprzez zarządzanie zakresami podtablic w pętli. Jako pivot wybrano medianę trzech elementów (pierwszego, środkowego i ostatniego), co poprawia równomierność podziału i stabilizuje wydajność algorytmu.

2.3.3 IntroSort

- IntroSort to algorytm sortowania łączący zalety QuickSort, HeapSort (sortowanie przez kopcowanie) i InsertionSort (sortowanie przez wstawianie), co pozwala na uzyskanie optymalnej wydajności zarówno w przypadku danych posortowanych, jak i silnie nieuporządkowanych. Wykorzystuje strategię dziel i rządź, a jego kluczową cechą jest monitorowanie głębokości rekurencji, aby unikać pesymistycznej złożoności $O(n^2)$. Algorytm rozpoczyna działanie jako QuickSort, wykorzystując podział tablicy wokół pivota i rekurencyjne sortowanie podtablic. Jeśli głębokość rekurencji przekroczy ustalony próg $2 \log_2 n$, algorytm przełącza się na HeapSort, który gwarantuje $O(n \log_2 n)$ w najgorszym przypadku. Dodatkowo, dla bardzo małych tablic (zwykle poniżej 16 elementów) stosowany jest InsertionSort, który działa wydajnie dla małych zbiorów danych.

- Pesymistyczna, średnia oraz optymistyczna złożoność czasowa IntroSort wynosi $O(n \log_2 n)$. Dzięki adaptacyjnemu podejściu algorytm unika problemu degeneracji występującego w QuickSort, gdzie nierównomierne podziały mogą prowadzić do złożoności $O(n^2)$.

- IntroSort działa w miejscu i ma złożoność pamięciową $O(\log_2 n)$, wynikającą z rekurencyjnej implementacji QuickSort w początkowej fazie. Algorytm nie jest stabilny, ponieważ HeapSort może zmieniać względną kolejność elementów o tej samej wartości.

- W ramach projektu zaimplementowano wersję IntroSort, która wykorzystuje QuickSort jako podstawową metodę sortowania, monitorując głębokość rekurencji. Jeśli liczba podziałów przekroczy $2 \log_2 n$, algorytm przełącza się na HeapSort. Dodatkowo, dla podtablic mniejszych niż 16 elementów stosowany jest InsertionSort, co poprawia wydajność dla niewielkich zbiorów danych.

3 Eksperymenty i analiza wyników

3.1 Metodyka eksperymentów

Badania zostały przeprowadzone w pliku `main.cpp`, w głównej pętli programu. Na początku znajdują się deklaracje użytych funkcji:

- `fit_nlogn` — dopasowuje współczynniki A i B funkcji regresji $f(n) = A \cdot n \log_2 n + B$ do danych `data`, odpowiadającym wartościom `sizes`, przy użyciu metody najmniejszych kwadratów.
- `generate_random_data` — służy do zapełniania tablicy (wskaźnik na nią przekazywany jest jako parametr) pseudolosowymi liczbami całkowitymi z określonego zakresu. Do generowania danych użyto biblioteki `random`. Dodatkowo przekazywany jest parametr `percentage`, określający procent początkowych elementów, które mają być posortowane. Wartość ujemna oznacza, że wskazany procent elementów będzie posortowany w kolejności odwrotnej. Do wstępnego posortowania, wykorzystano własną implementację `IntroSort`.
- `measure_sorting_time` — mierzy czas wykonywania funkcji sortującej. Wszystkie algorytmy przyjmują jako parametry:
  1. wskaźnik na tablicę do posortowania,
  2. indeks elementu, od którego ma się rozpocząć sortowanie,
  3. indeks elementu, na którym ma się zakończyć sortowanie.

  Czyli jeśli chcemy posortować całą tablicę, należy przekazać 0 jako drugi parametr i `rozmiar`-1 jako trzeci. Do pomiaru czasu wykorzystano bibliotekę `chrono`, a funkcja zwraca czas wykonania w nanosekundach.

W pętli głównej programu określono rozmiary tablic do przetestowania, zakres wartości losowych oraz stopień wstępnego posortowania. Dla każdego algorytmu, rozmiaru tablicy i poziomu uporządkowania danych przeprowadzana jest seria pomiarów. Dla każdej kombinacji parametrów pętlę powtórzono `N_TIMES` razy (w eksperymencie przyjęto wartość 100).

Aby zapewnić spójność wyników, na początku tworzona jest referencyjna tablica z wygenerowanymi danymi. Następnie dla każdego pomiaru tworzona jest jej kopia, która dopiero podlega sortowaniu. Zmierzony czas jest dodawany do odpowiedniego elementu tablicy przechowującej wyniki. Po wykonaniu `N_TIMES` pomiarów, wartość każdej komórki podzielono przez `N_TIMES`, co pozwala uzyskać średni czas działania algorytmu. Ostateczne wyniki są zapisywane, a tablica wyników wyzerowana przed kolejnym pomiarem.

Dane do dalszej analizy zostały zapisane do pliku `results.csv`, który znajduje się w repozytorium.

3.2 Wyniki testów

Na kolejnych stronach przedstawiono wyniki testów w formie tabel i wykresów. Dla każdego algorytmu stworzono:

- Tabelę ze średnimi czasami wykonywania, w zależności od typu danych wejściowych.
- Dwa wykresy porównujące czasy działania — jeden w skali standardowej, drugi w skali $\log x$-$\log y$ (Wraz z dopasowaniami do $\mathrm{O}(n \log n)$ - przerywaną linią).

Dodatkowo porównano algorytmy między sobą, przedstawiając wyniki na wykresach liniowych — również w dwóch wersjach: standardowej i $\log x$-$\log y$.

Tabela 1: Wyniki - MergeSort

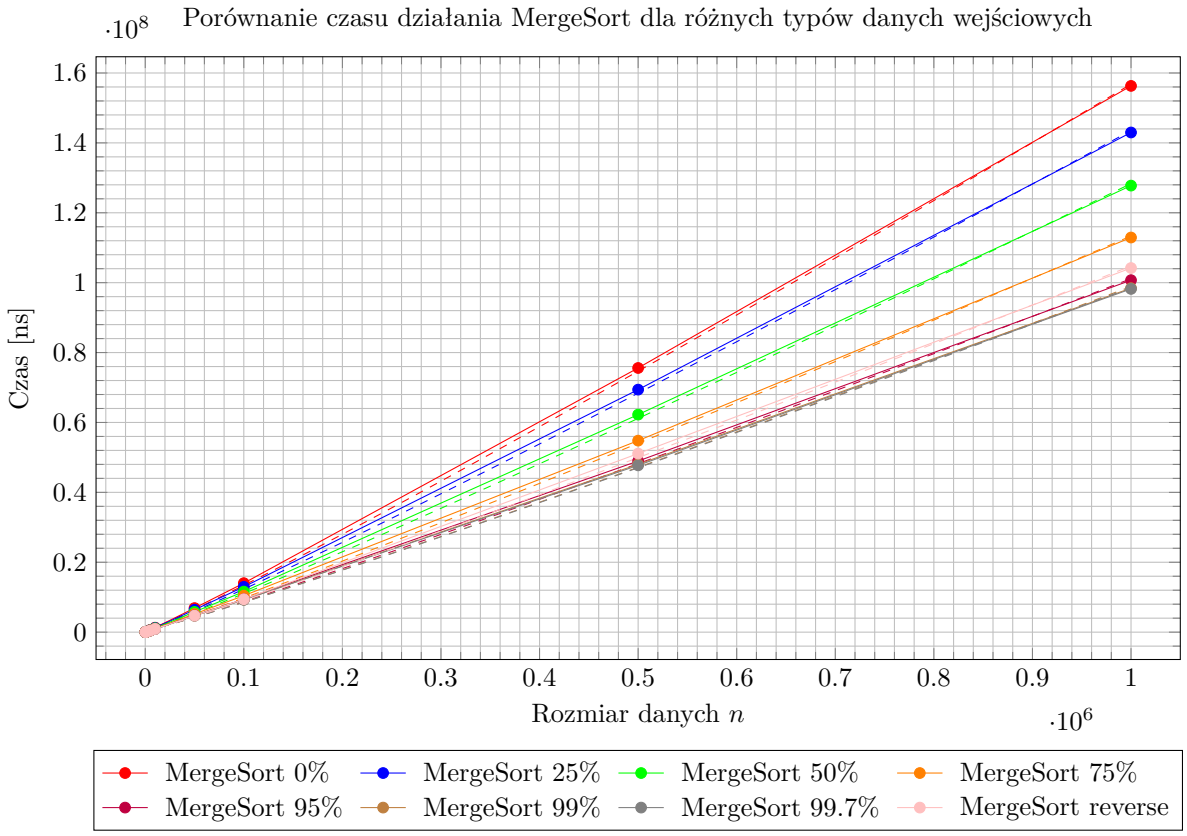
Rozmiar	**Liczba posortowanych elementów tablicy**							**Odwrotna kolejność**
	0%	**25%**	**50%**	**75%**	**95%**	**99%**	**99.7%**	
100	13555	13137	12354	11871	11475	11183	11036	11199
500	57803	55545	52154	48290	45770	46210	44967	45006
1000	118410	112750	102745	95659	85064	88843	86000	88313
5000	612420	574560	529193	488446	449699	445578	446804	441721
10000	1278355	1208814	1104473	992131	911512	914029	895552	910384
50000	6882214	6405957	5796715	5209414	4786295	4679167	4691047	4717525
100000	14017424	12961719	11597704	10436457	9426712	9267727	9253230	9422576
500000	75589987	69396532	62248190	54800777	48958996	48020130	47722694	51095125
1000000	156305716	142976116	127775782	112921487	100714852	98424189	98212175	104190329

Tabela 2: Wyniki - QuickSort

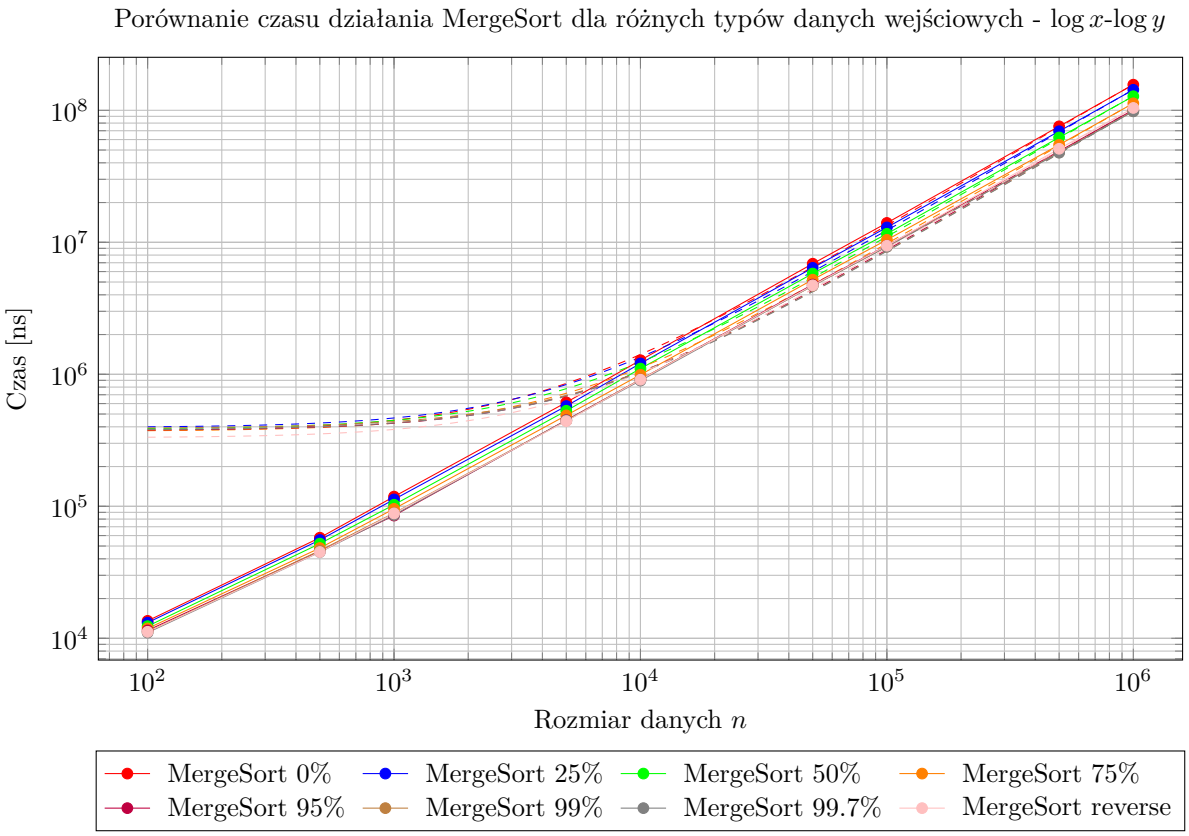
Rozmiar	**Liczba posortowanych elementów tablicy**							**Odwrotna kolejność**
	0%	**25%**	**50%**	**75%**	**95%**	**99%**	**99.7%**	
100	6088	6001	5746	5209	3879	2039	2007	2171
500	32867	31817	30932	27560	20159	27730	33560	9598
1000	69358	64959	62388	56623	40217	65404	100285	19396
5000	391295	369438	346359	309600	237769	323913	712714	101237
10000	811042	794946	755762	646305	506684	719476	1575020	212284
50000	4510968	4353827	4096905	3562289	2865942	4060813	9100817	1242321
100000	9301253	9109573	8495236	7377083	5953662	8770250	17295849	2706966
500000	50455001	49264995	46121118	39922028	32974323	46581273	94227390	16979709
1000000	103054387	100127953	93209573	80471789	66312083	101542895	198032896	36434506

Tabela 3: Wyniki - IntroSort

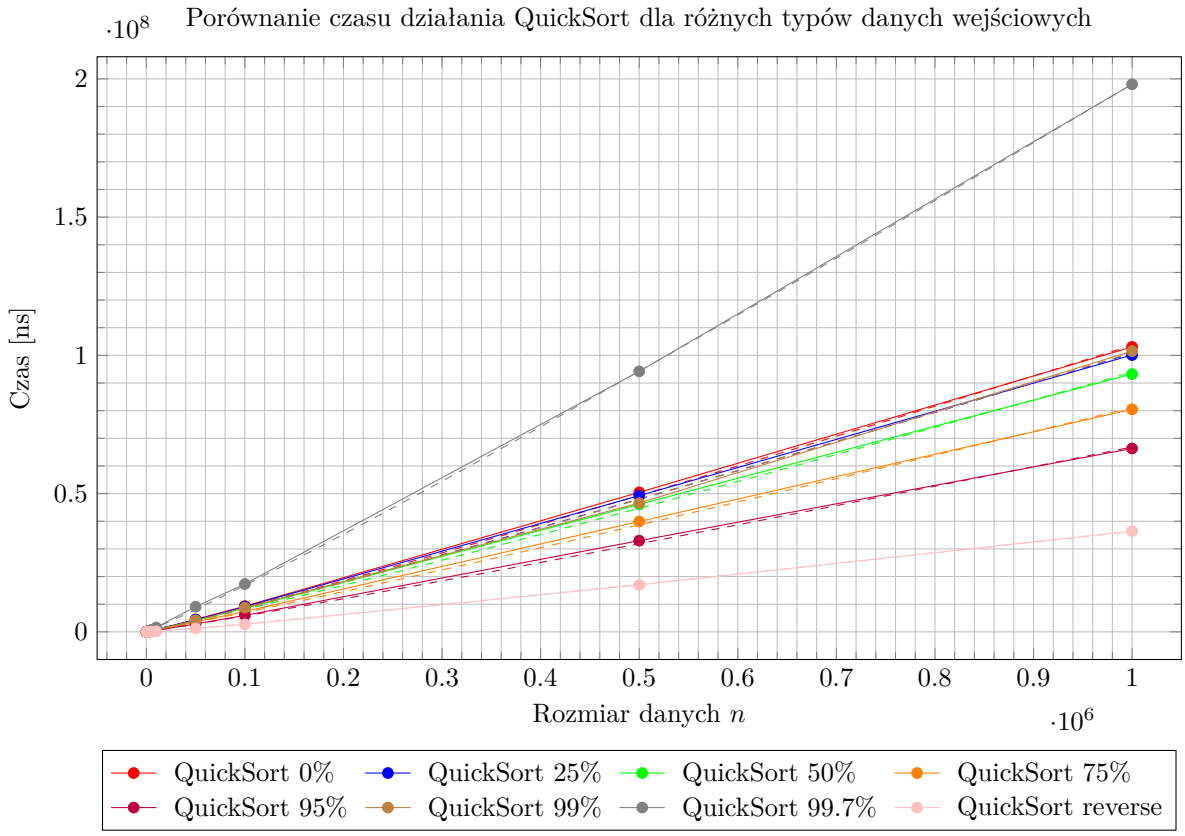
Rozmiar	**Liczba posortowanych elementów tablicy**							**Odwrotna kolejność**
	0%	**25%**	**50%**	**75%**	**95%**	**99%**	**99.7%**	
100	3339	3254	3235	2899	3192	936	896	1132
500	19468	18671	17650	16347	18436	22259	18071	6264
1000	44048	41481	39989	34528	35461	53578	43998	9975
5000	274567	269416	243666	210908	223583	290414	337257	56126
10000	603311	582110	535381	446945	462781	628213	726991	116378
50000	3451034	3366455	3119560	2564458	2658446	3615918	3976967	631374
100000	7250424	7084533	6497195	5367159	5506300	7356761	8228708	1361381
500000	40527549	39390826	36167260	30074500	29984518	39912655	45105135	8484843
1000000	84567442	81820359	74699346	61301587	58048657	84182975	95980156	18607565



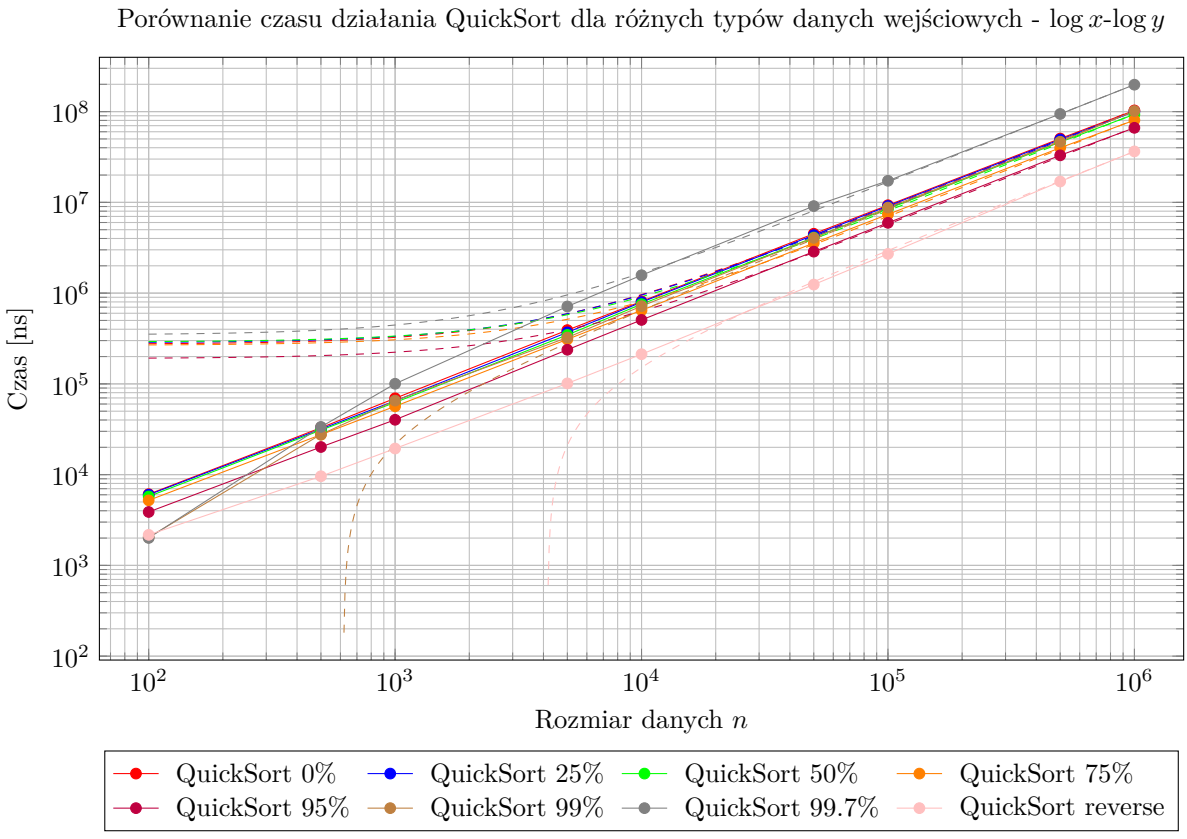
Wykres 1: MergeSort



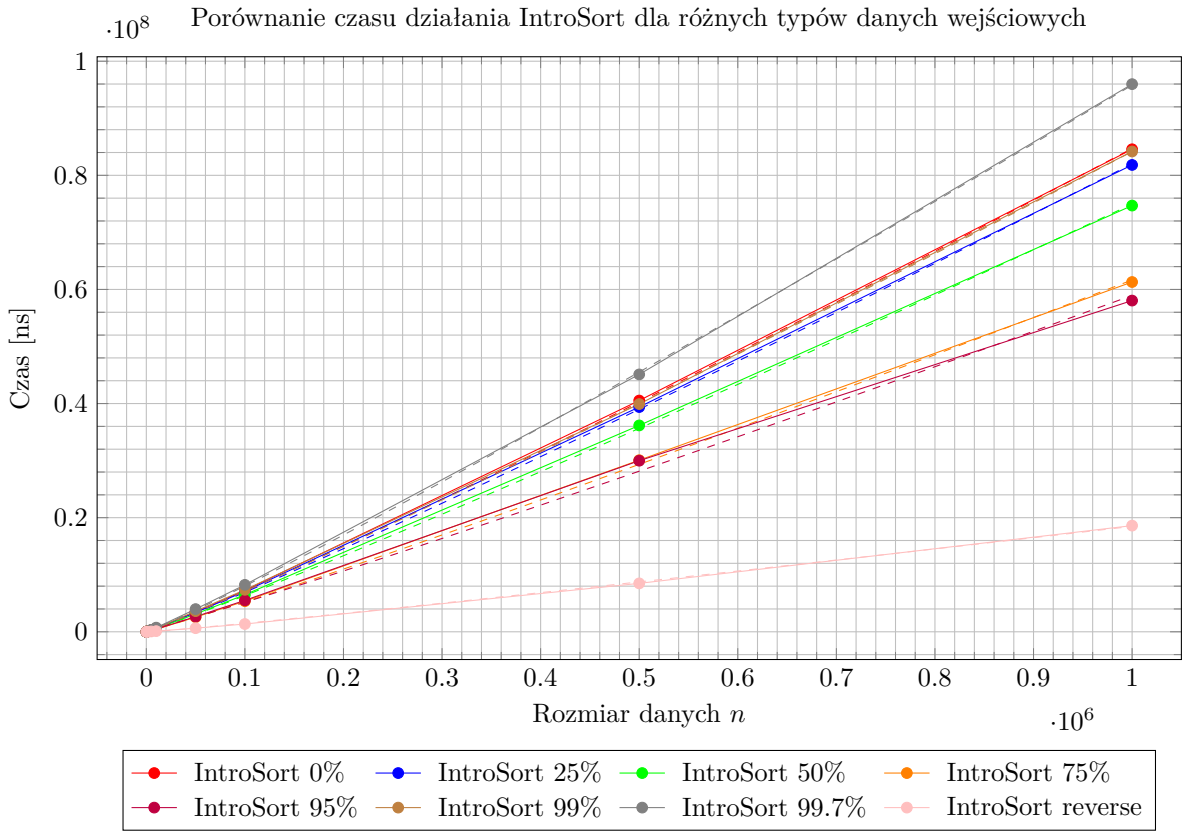
Wykres 2: MergeSort (w skali $\log x$-$\log y$)



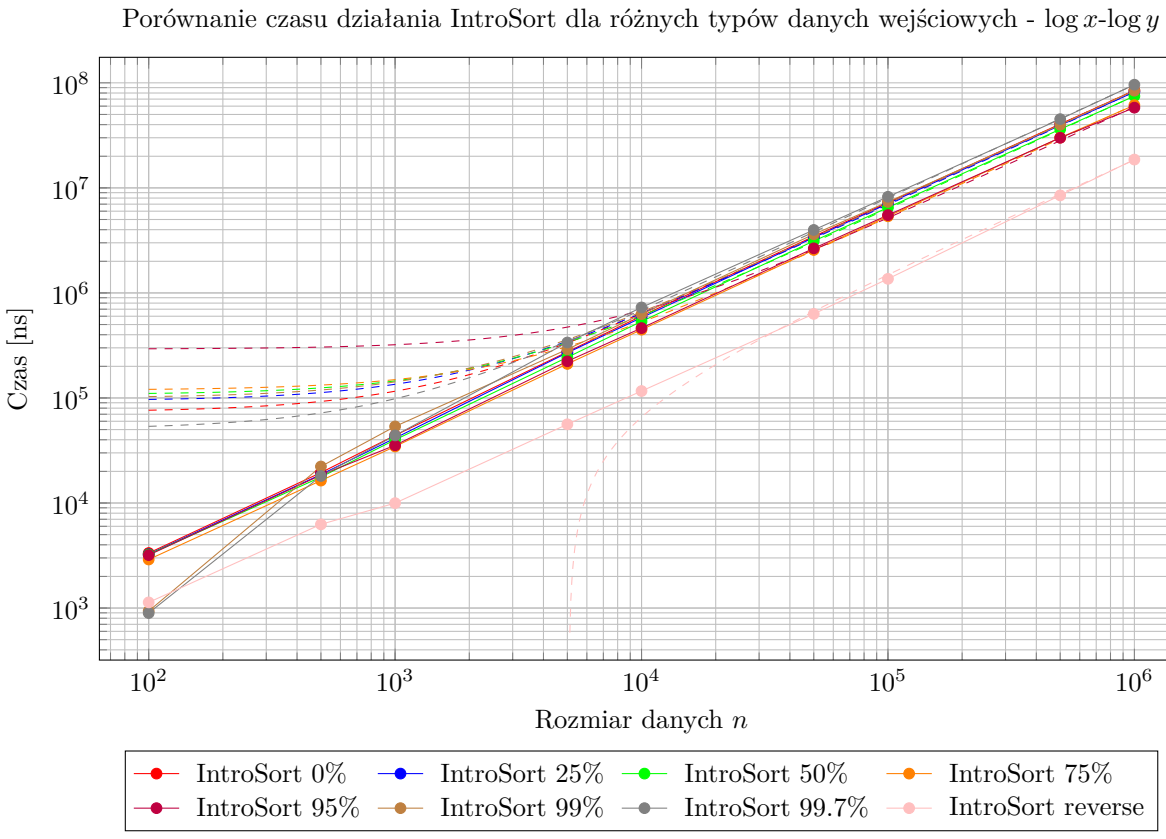
Wykres 3: QuickSort



Wykres 4: QuickSort (w skali log x-log y)

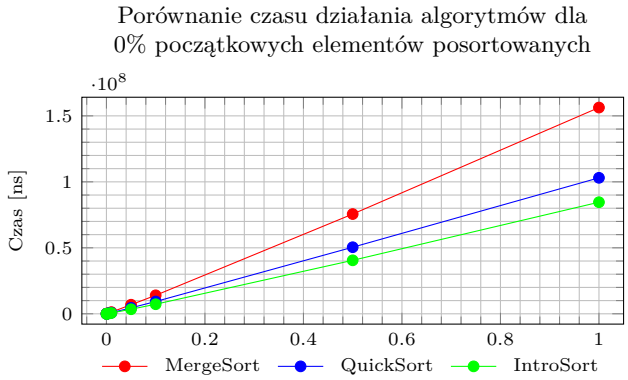


Wykres 5: IntroSort

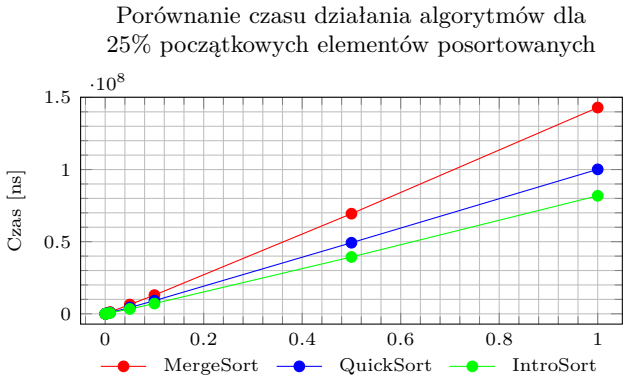


Wykres 6: IntroSort (w skali log x-log y)

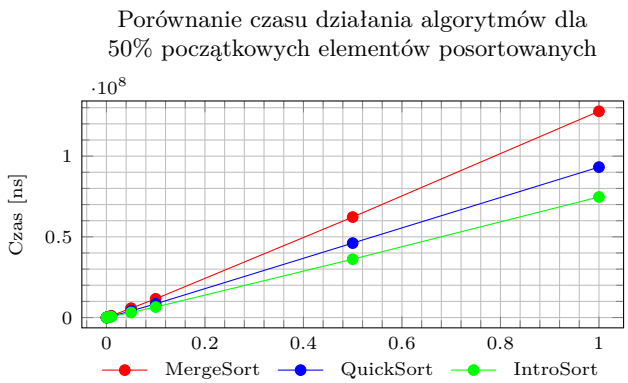
Wykres 7: Porównanie czasu działania algorytmów dla różnych typów danych wejściowych



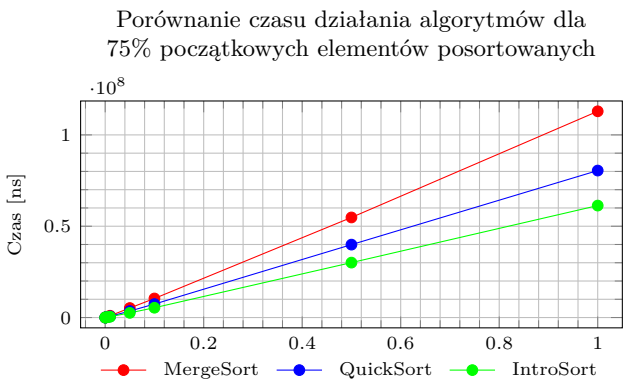
(a) 0% początkowych elementów posortowanych



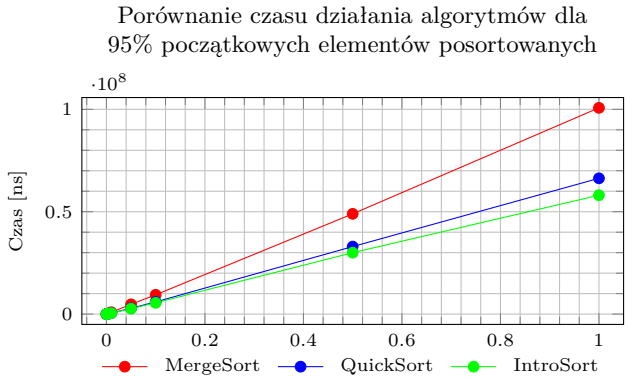
(b) 25% początkowych elementów posortowanych



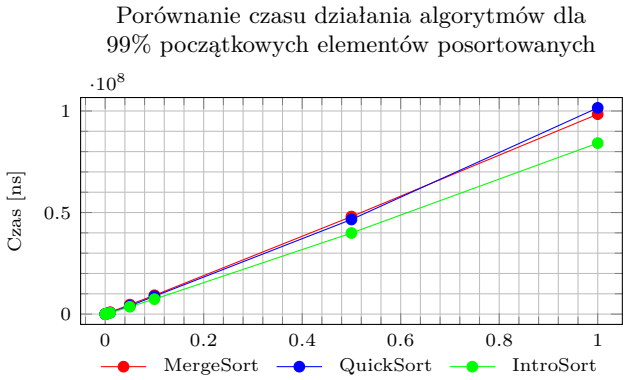
(c) 50% początkowych elementów posortowanych



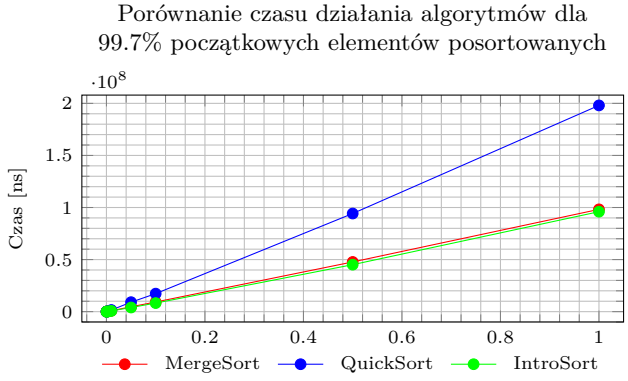
(d) 75% początkowych elementów posortowanych



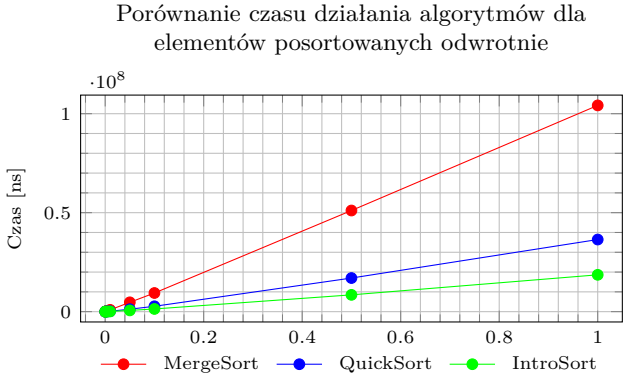
(e) 95% początkowych elementów posortowanych



(f) 99% początkowych elementów posortowanych

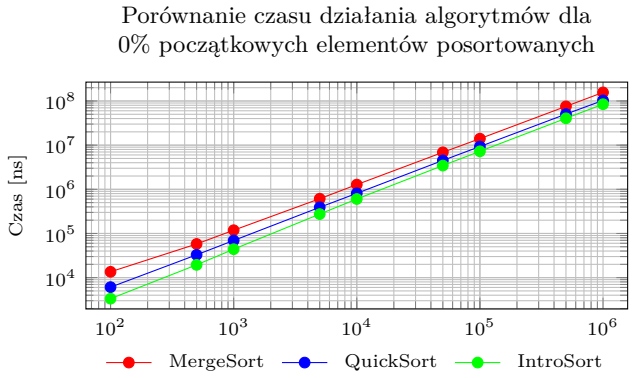


(g) 99.7% początkowych elementów posortowanych

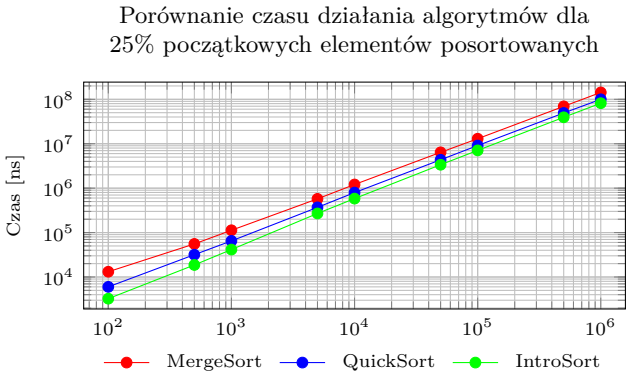


(h) Elementy posortowane odwrotnie

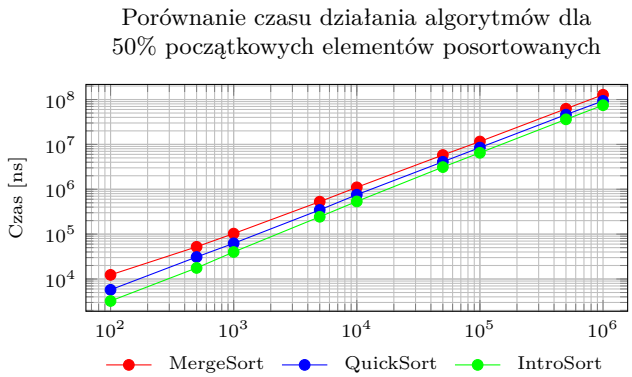
Wykres 8: Porównanie czasu działania algorytmów dla różnych typów danych wejściowych - $\log x$-$\log y$



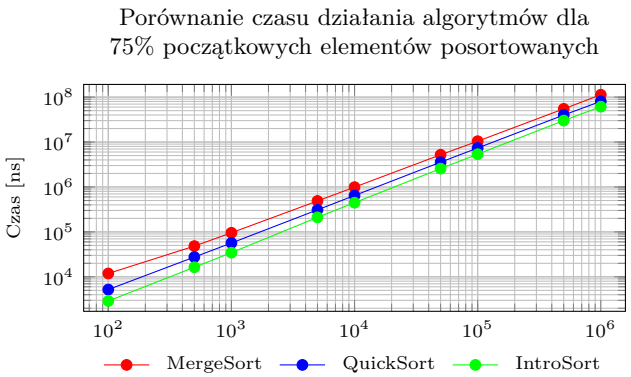
(a) 0% początkowych elementów posortowanych



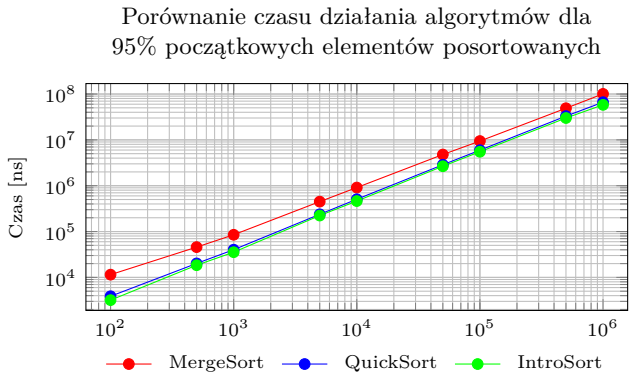
(b) 25% początkowych elementów posortowanych



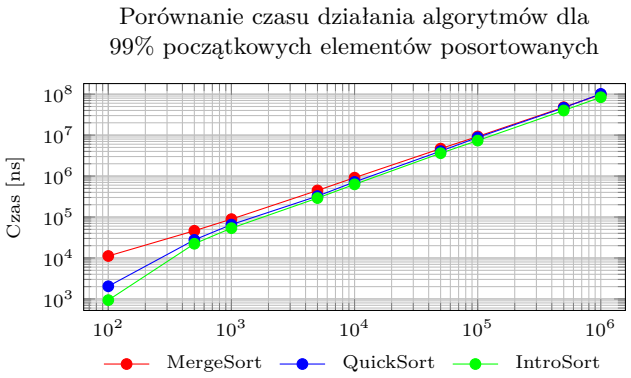
(c) 50% początkowych elementów posortowanych



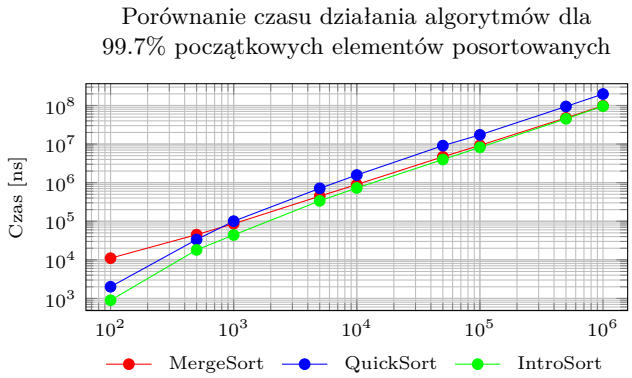
(d) 75% początkowych elementów posortowanych



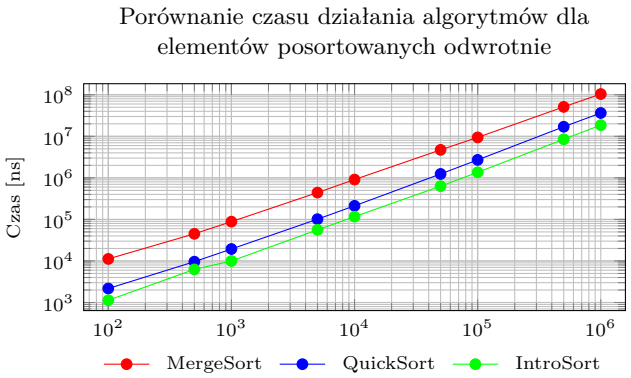
(e) 95% początkowych elementów posortowanych



(f) 99% początkowych elementów posortowanych



(g) 99.7% początkowych elementów posortowanych



(h) Elementy posortowane odwrotnie

3.3 Wnioski

3.3.1 Podsumowanie wyników

Przeprowadzona analiza obejmuje porównanie czasów wykonywania algorytmów MergeSort, QuickSort oraz IntroSort w różnych przypadkach początkowego uporządkowania danych. Wyniki przedstawiają czas sortowania (w nanosekundach) dla różnych rozmiarów tablicy i stopni uporządkowania elementów.

- MergeSort zapewnia stabilne czasy wykonywania, lecz są one wyraźnie wyższe niż w przypadku pozostałych algorytmów.
- QuickSort wykazuje dobrą wydajność na losowych danych, ale jego czas wykonania znacząco rośnie przy częściowo posortowanych danych, zwłaszcza przy 50%, 75% i 99.7% uporządkowanych elementach. W skrajnych przypadkach (np. 99.7% posortowanych) może być nawet kilkukrotnie wolniejszy niż IntroSort.
- IntroSort łączy zalety QuickSort i HeapSort, co zapewnia mu najlepszą wydajność w większości przypadków, szczególnie na dużych zbiorach danych.

3.3.2 Porównanie efektywności algorytmów w różnych przypadkach

1. Losowa kolejność elementów (0% posortowanych)
   - QuickSort działa szybko, ale IntroSort jest jeszcze szybszy (np. dla $n = 10^5$: QuickSort ≈ 9.30 ms, IntroSort ≈ 7.25 ms).
   - MergeSort jest wyraźnie wolniejszy (np. dla $n = 10^5$: ≈ 14.02 ms).
2. Częściowo posortowane dane (25-75%)
   - QuickSort wykazuje stopniowy wzrost czasu wykonania wraz ze wzrostem uporządkowania danych. Dla 75% posortowanych elementów czas może być nawet 2–3 razy dłuższy niż dla danych losowych.
   - IntroSort pozostaje szybki i odporny na częściowe uporządkowanie danych.
   - MergeSort utrzymuje względnie stały czas, choć nieco krótszy niż dla danych losowych.
3. Silnie uporządkowane dane (95-99.7%)
   - QuickSort staje się wyjątkowo nieefektywny (np. dla $n = 10^5$ i 99.7% posortowanych: ≈ 198.03 ms, podczas gdy IntroSort ≈ 95.98 ms).
   - IntroSort nadal zachowuje wysoką wydajność.
   - MergeSort zachowuje stabilność, jednak ustępuje wydajnością IntroSortowi.
4. Odwrócona kolejność elementów
   - QuickSort i IntroSort radzą sobie bardzo dobrze (np. dla $n = 10^5$: QuickSort ≈ 2.70 ms, IntroSort ≈ 1.36 ms).
   - MergeSort jest znacznie wolniejszy (np. dla $n = 10^5$: ≈ 9.42 ms).

3.3.3 Najbardziej efektywne algorytmy

- IntroSort okazuje się najlepszym wyborem w większości przypadków, łącząc szybkość QuickSorta z odpornością na zdegenerowane dane.
- QuickSort działa bardzo wydajnie dla danych losowych, lecz jego efektywność znacząco maleje w przypadku częściowego uporządkowania.
- MergeSort, mimo stabilności czasu wykonywania, jest najwolniejszy i powinien być wybierany tylko tam, gdzie wymagana jest stabilność sortowania.

3.3.4 Możliwe ulepszenia

- Natural MergeSort – może przyspieszyć sortowanie, jeśli dane zawierają naturalne uporządkowane podciągi.
- Lepszy wybór pivota w QuickSort i IntroSort – wybrana strategia mediany z trzech jest już optymalizacją, jednak dla innych rodzajów danych inne strategie wyboru pivota mogą okazać się skuteczniejsze.
- Sortowanie równoległe – wykorzystanie wielowątkowości mogłoby przyspieszyć działanie wszystkich algorytmów, szczególnie dla dużych zbiorów danych.

Źródła

[1] M.T. Goodrich, R. Tamassia, D. Mount, *Data Structures and Algorithms in C++*, 2nd ed., John Wiley & Sons, Inc., Hoboken, NJ, USA, 2011

[2] T.H. Cormen, C.E. Leiserson, R.L. Rivest, C. Stein, *Introduction to Algorithms*, 3rd ed., MIT Press, 2009

[3] Wikipedia, *Merge sort*, dostęp: 13 kwietnia 2025, `https://en.wikipedia.org/wiki/Merge_sort`

[4] Wikipedia, *Quicksort*, dostęp: 13 kwietnia 2025, `https://en.wikipedia.org/wiki/Quicksort`

[5] Wikipedia, *Introsort*, dostęp: 13 kwietnia 2025, `https://en.wikipedia.org/wiki/Introsort`

[6] GeeksForGeeks, *Introsort – C++'s Sorting Weapon*, dostęp: 13 kwietnia 2025, `https://www.geeksforgeeks.org/introsort-cs-sorting-weapon/`

[7] Wikipedia, *Heapsort*, dostęp: 13 kwietnia 2025, `https://en.wikipedia.org/wiki/Heapsort`

[8] Wikipedia, *Insertion sort*, dostęp: 13 kwietnia 2025, `https://en.wikipedia.org/wiki/Insertion_sort`