Politechnika Wrocławska

Projektowanie i analiza algorytmów

Implementacja i Analiza Algorytmów Sortowania

Igor Frysiak, 272548
Grupa nr. 3, 13:15 - 15:00
Prowadzący: Piotr Ciskowski
Informatyczne Systemy Automatyki

29 kwietnia 2025

1 Streszczenie

Celem projektu było zaimplementowanie trzech algorytmów sortowania: sortowania przez scalanie, sortowania szybkiego oraz sortowania introspektywnego, a następnie przeprowadzenie analizy ich efektywności. Eksperymenty przeprowadzono na tablicach liczb całkowitych o różnych rozmiarach i stopniu uporządkowania danych, lecz w programie użyto szablonów, co umożliwia tablicom przechowywanie nie tylko liczb całkowitych, ale także liczb zmiennoprzecinkowych oraz stringów. Badania wykazały, że QuickSort, przy odpowiednim wyborze pivota, jest najszybszy dla większości danych losowych i uporządkowanych, a IntroSort zapewnia największą stabilność w trudnych przypadkach. MergeSort charakteryzował się przewidywalnym czasem działania, ale był wolniejszy przy dużych danych. Ostateczne wyniki potwierdzają znaczący wpływ charakterystyki danych wejściowych na efektywność algorytmów.

2 Wstęp

Sortowanie jest jednym z fundamentalnych problemów w informatyce, mającym kluczowe znaczenie w szerokim spektrum zastosowań i ma bezpośredni wpływ na wydajność całych systemów informatycznych.

W projekcie zdecydowano się na analizę następujących algorytmów:

- **Sortowanie przez scalanie (Merge Sort)** – stabilny algorytm sortowania o złożoności czasowej $O(n \log n)$ w każdym przypadku, wykorzystujący metodę "dziel i zwyciężaj".
- **Sortowanie szybkie (Quicksort)** – bardzo wydajny w praktyce algorytm o średniej złożoności czasowej $O(n \log n)$, lecz w najgorszym przypadku może osiągać $O(n^2)$.
- **Sortowanie introspektywne (Introsort)** – hybrydowy algorytm łączący quicksort i heapsort, gwarantujący złożoność $O(n \log n)$ nawet w najgorszym przypadku.

Celem projektu było zaimplementowanie powyższych algorytmów z wykorzystaniem szablonów oraz analiza ich efektywności na różnych typach danych wejściowych.

3 Eksperymenty i analiza wyników

3.1 Metodyka eksperymentów

Eksperymenty przeprowadzono na tablicach zawierających liczby całkowite o rozmiarach: 100, 500, 1000, 5000, 10 000, 50 000, 100 000, 500 000 oraz 1 000 000 elementów. Testy obejmowały następujące przypadki:

- Tablice z losowymi wartościami,
- Tablice, w których odpowiednio 25%, 50%, 75%, 95%, 99% i 99,7% początkowych elementów było posortowanych,
- Tablice całkowicie posortowane w odwrotnej kolejności.

W celu porównania wydajności algorytmów sortowania wykorzystano samodzielnie zaimplementowaną funkcję, która mierzy czas wykonania sortowania przy pomocy zegara wysokiej rozdzielczości *(std::chrono::high_resolution_clock)*.

Aby uzyskać bardziej miarodajne wyniki, każdy eksperyment powtarzany był **100 krotnie** dla tej samej wielkości tablicy i typu danych wejściowych, a wyniki uśredniono. Ostateczne średnie czasy zapisano w pliku CSV.

3.2 Wyniki testów

Wyniki zostały przedstawione w postaci tabel oraz wykresów. (Tabela czasów wykonywania dla różnych częściowych wstępnych posortowań byłaby za duża, więc została pominięta.)

Rozmiar	MergeSort [ms]	QuickSort [ms]	IntroSort [ms]
100	0.008768	0.003684	0.003519
500	0.045630	0.022487	0.021131
1000	0.095808	0.046925	0.047069
5000	0.515320	0.270084	0.272491
10000	1.086410	0.585419	0.597011
50000	5.995940	3.282650	3.385160
100000	12.127200	6.842120	7.153870
500000	66.218400	38.134000	40.320900
1000000	137.235000	79.505300	84.553300

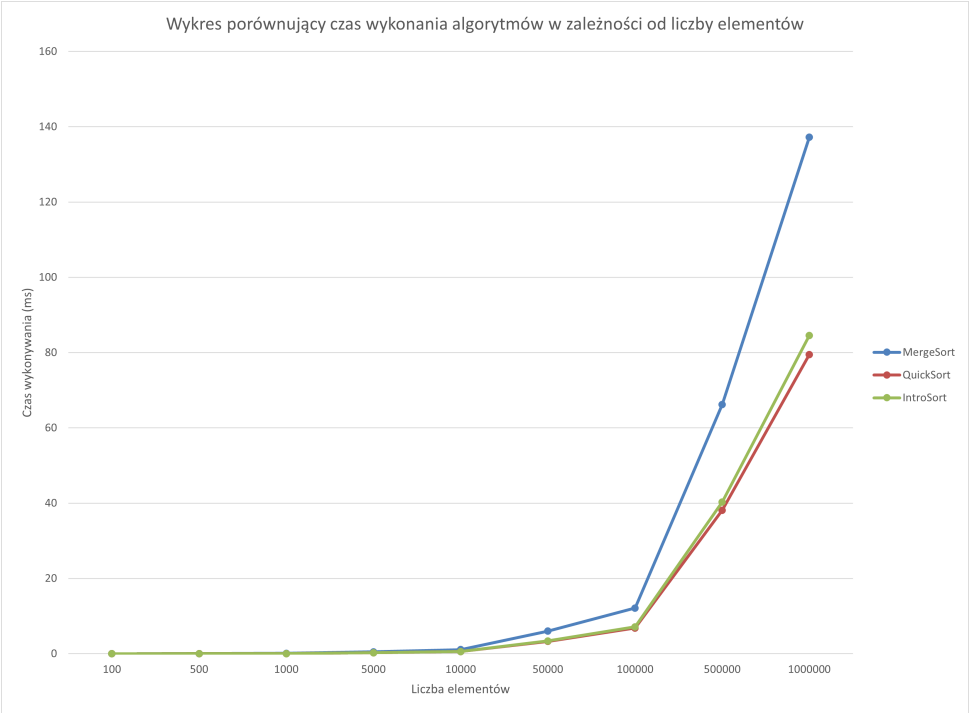
Tabela 1: Czasy wykonania różnych algorytmów sortowania dla tablic o różnych rozmiarach (tablice losowe).

Uporządkowanie	MergeSort [ms]	QuickSort [ms]	IntroSort [ms]
25%	0.498484	0.264211	0.269307
50%	0.443464	0.288037	0.251372
75%	0.406070	0.203681	0.259899
95%	0.366771	0.138837	0.355464
99%	0.363273	0.116281	0.447884
99.7%	0.360155	0.103331	0.500532

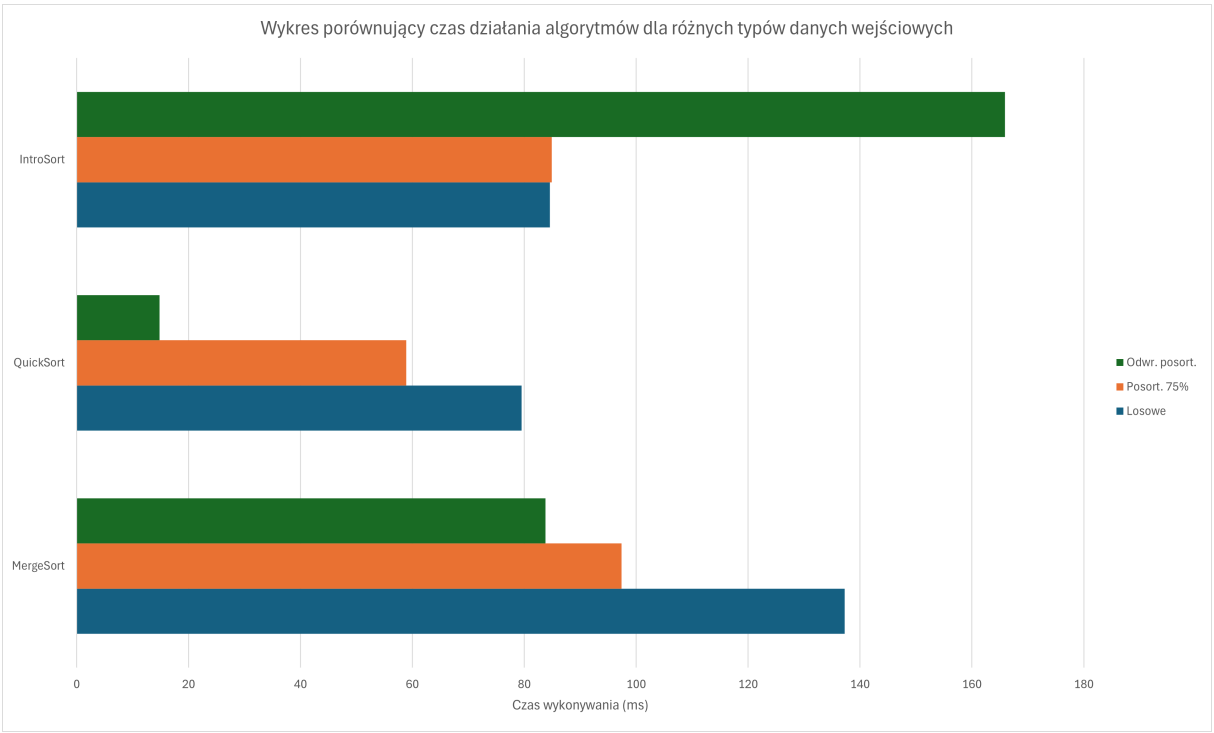
Tabela 2: Czasy wykonania algorytmów sortowania dla różnych rozmiarów tablic i poziomów uporządkowania (procent posortowania danych wejściowych). Pokazane są przykładowe dane **dla tablicy o wielkości 5000**.

Rozmiar	MergeSort [ms]	QuickSort [ms]	IntroSort [ms]
100	0.006804	0.000959	0.003802
500	0.033580	0.004786	0.028728
1000	0.066860	0.010182	0.071769
5000	0.348344	0.058767	0.519924
10000	0.736439	0.123369	1.128000
50000	3.964450	0.688533	6.461710
100000	7.803250	1.412990	13.712200
500000	40.899200	7.156760	78.286000
1000000	83.752200	14.805700	165.909000

Tabela 3: Czasy wykonania różnych algorytmów sortowania dla tablic o różnych rozmiarach (dane wejściowe posortowane odwrotnie).

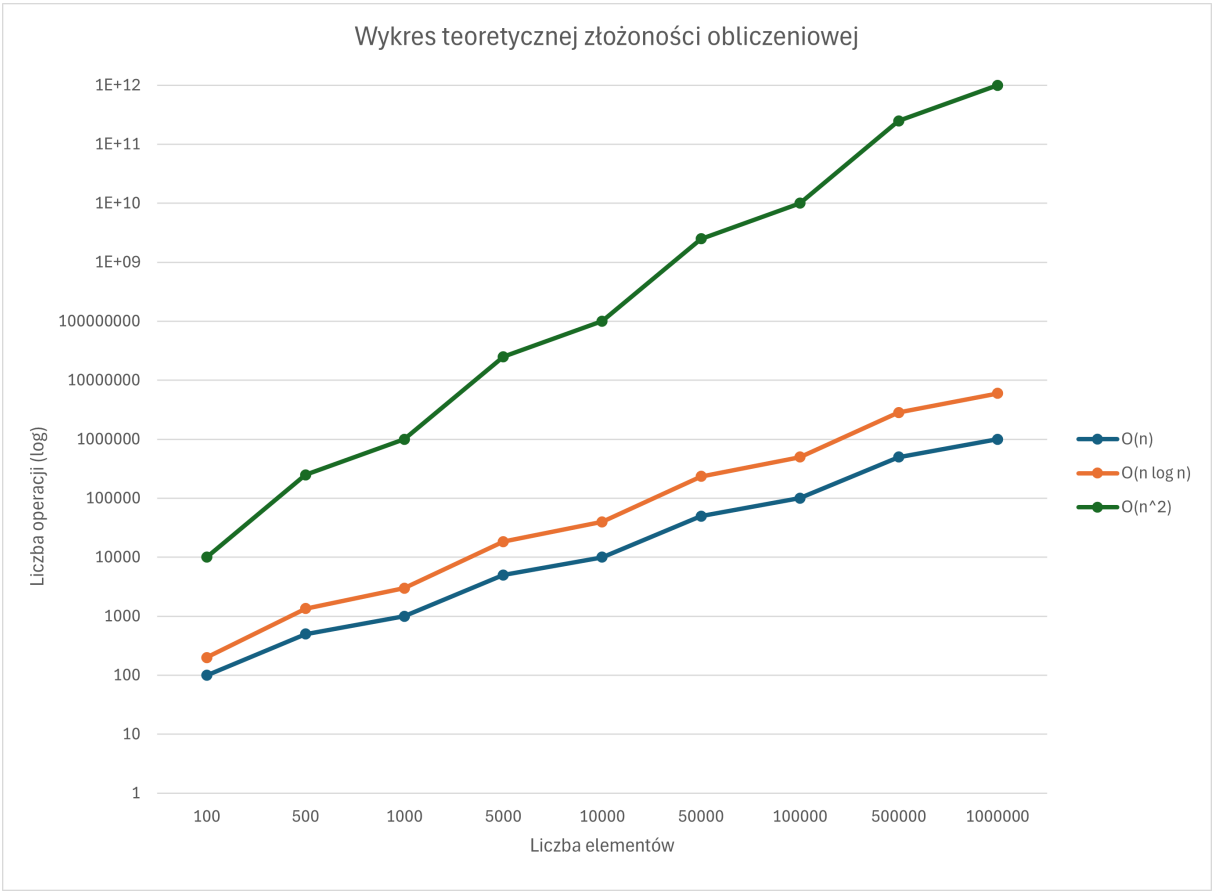


Rysunek 1: Czas działania algorytmów w zależności od rozmiaru danych (losowe dane).



Rysunek 2: Porównanie czasów działania algorytmów dla różnych typów danych wejściowych.

Za dane wejściowe dla częściowo posortowanych danych przyjęto te w 75% posortowane. Przyjęto dane dla tablicy o wielkości 1 000 000.



Rysunek 3: Teoretyczna złożoność obliczeniowa: O(n), O(n log n), O(n^2).

3.3 Analiza wyników

Na podstawie uzyskanych danych można zaobserwować kilka istotnych trendów:

- Merge Sort:
  - Czas działania jest dość stabilny niezależnie od uporządkowania danych.
  - Jego złożoność teoretyczna O(n log n) przekłada się na przewidywalne wyniki.
  - Dla dużych rozmiarów danych Merge Sort działał jednak wolniej niż pozostałe algorytmy.
- Quick Sort:
  - Najszybszy dla tablic losowych i częściowo posortowanych.
  - W przypadku odwrotnie posortowanych danych wydajność QuickSortu nawet wzrosła dzięki odpowiedniemu wyborowi pivota.
- Intro Sort:
  - Łączy zalety QuickSort (dla przypadków prostych) i HeapSort (w sytuacji głębokiej rekurencji).
  - W trudnych przypadkach (głęboka rekurencja) zapewnia stabilność wykonania, choć w praktyce przy odwrotnie posortowanych danych był wolniejszy od QuickSort.

4 Wnioski

Wyniki pokazały, że:

- Quick Sort dominował pod względem szybkości zarówno dla danych losowych, częściowo posortowanych, jak i odwrotnie posortowanych.
- Intro Sort osiągał podobne wyniki do Quick Sorta dla danych losowych i częściowo uporządkowanych, zapewniając dodatkowe bezpieczeństwo przed najgorszym przypadkiem poprzez przejście na HeapSort.
- Merge Sort, mimo nieco wolniejszego działania, oferował stabilność wyników niezależnie od stopnia uporządkowania danych i jest szczególnie wartościowy, gdy wymagana jest stabilność sortowania.

5 Bibliografia

- Michael T. Goodrich. *Data structures and algorithms in C++.*
- Wikipedia: `https://pl.wikipedia.org/wiki/Sortowanie_przez_scalanie`
- Wikipedia: `https://pl.wikipedia.org/wiki/Sortowanie_szybkie`
- Wikipedia: `https://pl.wikipedia.org/wiki/Sortowanie_introspektywne`