

WYDZIAŁ INFORMATYKI I TELEKOMUNIKACJI
POLITECHNIKA WROCŁAWSKA

ANALIZA EKSPERYMentalna
NOWYCH ALGORYTMÓW
SORTOWANIA W MIEJSCU

DAMIAN BALIŃSKI
NR INDEKSU: 250332

Praca inżynierska napisana
pod kierunkiem
dr. inż. Zbigniewa Gołębiewskiego



Politechnika
Wrocławskiego

WROCŁAW 2021

Spis treści

1 Wstęp	1
1.1 Cel pracy	1
1.2 Zakres pracy	1
1.3 Przegląd literatury	1
1.4 Zawartość pracy	2
2 Metodologia	3
2.1 Liczba operacji	3
2.2 Koszt operacji	3
2.3 Założenia	4
2.3.1 Operacje na iteratorach	4
2.3.2 Rekurencja	4
2.3.3 Alokacja pamięci	4
2.3.4 Dane wejściowe	4
3 Przegląd podstawowych algorytmów sortujących	5
3.1 Quick Sort	5
3.1.1 Analiza algorytmu Quick Sort	5
3.1.2 Problemy związane z algorytmem Quick Sort	6
3.1.3 Możliwości optymalizacyjne	8
3.2 Merge Sort	8
3.2.1 Analiza algorytmu Merge Sort	8
3.2.2 Problemy związane z algorytmem Merge Sort	9
3.2.3 Możliwości optymalizacyjne	10
4 Przegląd hybrydowych algorytmów sortujących	11
4.1 Główne sposoby modyfikacji algorytmów	11
4.2 Rodzina deterministycznych algorytmów Quick Sort	11
4.2.1 Analiza porównawcza algorytmów	12
4.2.2 Wnioski	12
4.3 Rodzina randomizowanych algorytmów Quick Sort	15
4.3.1 Analiza porównawcza algorytmów	15
4.3.2 Wnioski	16
4.4 QuickMerge Sort	19
4.4.1 Pseudokod	19
4.4.2 Analiza deterministycznych wersji algorytmu QuickMerge Sort	20
4.4.3 Analiza randomizowanych wersji algorytmu QuickMerge Sort	21
4.4.4 Wnioski	23
4.5 Intro Sort	26
4.5.1 Eksperymentalne wyznaczanie punktu granicznego	26
4.5.2 Pseudokod	26
4.5.3 Analiza deterministycznych wersji algorytmu Intro Sort	27
4.5.4 Analiza randomizowanych wersji algorytmu Intro Sort	28
4.5.5 Wnioski	30

4.6	QuickMergeIntro Sort	32
4.6.1	Pseudokod	32
4.6.2	Analiza deterministycznych wersji algorytmu QuickMergeIntro Sort	33
4.6.3	Analiza randomizowanych wersji algorytmu QuickMergeIntro Sort	34
4.6.4	Wnioski	36
5	Podsumowanie	39
6	Implementacja systemu	41
6.1	Struktura systemu	41
6.2	Koncepcje architektury silnika testującego	41
6.2.1	Wstrzykiwanie zależności	41
6.2.2	Obiektywość	41
6.2.3	Bezstanowość	42
6.3	Model aplikacji	42
6.3.1	Diagram klas	42
6.3.2	Diagram aktywności	43
6.4	Wzorce projektowe	43
6.4.1	Fasada	43
6.4.2	Obiekt-Wartość	44
6.4.3	Strategia	44
6.4.4	Polecenie	45
Bibliografia		47
A	Słownik pojęć	49
A.1	Notacja duże- <i>O</i>	49
A.2	Algorytm działający w miejscu	49
A.3	Algorytm stabilny	49
B	Środowisko uruchomieniowe aplikacji	51
B.1	Zmienne środowiskowe	51
B.2	Biblioteki zewnętrzne	51
B.3	Cykl pracy programu	52
B.4	Przykładowy plik konfiguracyjny	52

Wstęp

Porządkowanie zbioru danych to jeden z podstawowych problemów współczesnej informatyki. Za algorytmy przełomowe w historii sortowania można uznać opracowany przez Johna von Neumanna algorytm **Merge Sort**, oraz algorytm **Quick Sort** autorstwa Tonyego Hoare. Obydwa algorytmy wykorzystują rekurencyjne podejście dziel i zwyciężaj, efektywnie rozbijając problem sortowania na mniejsze podproblemy. Od czasów ich wynalezienia ludzkość nieustannie stara się modyfikować te algorytmy, tworząc coraz efektywniejsze implementacje, które niwelują wady swoich poprzedników oraz zachowując swoją wydajność nawet dla przypadku pesymistycznego. W poniższej pracy dokonano analizy kliku nowoczesnych technik sortowania, opierających się na efektywnym łączniu kilku algorytmów podstawowych.

1.1 Cel pracy

Motywem przewodnim pracy była analiza eksperymentalna wybranych technik sortujących w miejscu, wykorzystujących nowatorskie podejście do problemu sortowania. W pracy zbadano algorytmy **QuickMerge Sort** i **Intro Sort**, oraz ich deterministyczne i randomizowane modyfikacje. Algorytmy były analizowane pod kątem czasowej złożoności obliczeniowej.

1.2 Zakres pracy

Aby ułatwić analizę algorytmów przygotowano narzędzia usprawniające proces testowania. Do tych narzędzi należą silnik graficzny oraz silnik testujący, które w oparciu o plik konfiguracyjny przeprowadzają testy oraz tworzą wizualizację wyników. Wykorzystując podane narzędzia, przygotowano analizę podstawowych algorytmów sortujących oraz dokonano przeglądu zmodyfikowanych wersji tych algorytmów. Następnie przeanalizowano nowoczesne algorytmy hybrydowe, będące połączeniem dwóch lub wielu algorytmów podstawowych.

1.3 Przegląd literatury

W pracy badano algorytm QuickMerge Sort, którego koncepcja została opisana przez Sebastiana Wilda w publikacji “**QuickXsort – A Fast Sorting Scheme in Theory and Practice**”. Informacje na temat tego algorytmu można znaleźć również w pracach “**QuickMergesort: Practically Efficient Constant-Factor Optimal Sorting**” oraz “**Worst-Case Efficient Sorting with QuickMergesort**”, których autorami są Stefan Edelkamp i Armin Weiß.

Kolejnym badanym algorytmem był Intro Sort. Informacje na ten temat znaleźć w publikacji “**Pattern-defeating Quicksort**” autorstwa Orsona R. L. Petersa.



1.4 Zawartość pracy

Praca składa się z pięciu rozdziałów. W rozdziale pierwszym przedstawiono ogólny cel oraz zakres pracy. Rozdział drugi poświęcony jest metodologii prowadzenia badań. W rozdziale tym wyjaśniono sposób, w jaki zliczano liczbę wykonywanych operacji porównania, przypisania oraz zamiany miejsc, uwzględniając koszt każdej takiej operacji. Rozdział trzeci zawiera przegląd podstawowych algorytmów sortujących, z wyszczególnieniem problemów, jakie wiążą się ze stosowania tych algorytmów. W rozdziale czwartym przedstawiono nowoczesne sposoby na tworzenie algorytmów hybrydowych. W rozdziale tym dokonano analizy porównawczej opisywanych algorytmów, z podziałem na wersje deterministyczne oraz randomizowane. Rozdział piąty zawiera podsumowanie wyników z wcześniej przeprowadzonych badań. W rozdziale tym wyłoniono kandydatów na najbardziej optymalny algorytm w danej kategorii. Ostatni rozdział to opis implementacji oraz wykorzystanych technik programowania. W tym rozdziale opisano koncepcję silnika testującego oraz wzorce projektowe użyte w systemie.

Dodatkowo praca zawiera dwa dodatki. Dodatek A to słownik pojęć wykorzystywanych w pracy. Dodatek B zawiera informacje na temat instalowania zewnętrznych zależności oraz uruchamiania aplikacji.

Metodologia

2.1 Liczba operacji

W pracy badano złożoność czasową algorytmów sortujących. Ponieważ rzeczywisty czas trwania uza-leżniony jest od maszyny oraz architektury systemu, na którym przeprowadzane są testy, podstawowym wskaźnikiem podczas analizy złożoności czasowej była łączna liczba operacji atomowych wykonywanych na danych testowych. W zbiorze operacji atomowych uwzględniono operacje porównania, zamiany miejsc oraz przypisania. Łączną liczbę operacji, będącą sumą ważoną operacji atomowych, określono wzorem:

$$N = n_c + 3n_s + n_a$$

N – łączna liczba operacji

n_c – liczba operacji porównania

n_s – liczba operacji zamiany miejsc

n_a – liczba operacji przypisania

Wzór ten wynika z faktu, że operacja zamiany może zostać rozbita na trzy operacje przypisania, z wyko-rzystaniem zmiennej tymczasowej.

2.2 Koszt operacji

Należy zauważać, że nie wszystkie operacje są równoważne pod względem czasowym. Sortując typy pod-stawowe, wszystkie operacje mają zbliżony czas trwania. Inaczej jest w przypadku złożonych struktur danych, dla których operacja porównania może trwać znacznie dłużej niż pojedyncza operacja przypisania. Różnica ta wynika ze sposobu przechowywania danych w pamięci. W przypadku złożonych struktur, efektywnym podejściem wydaje się użycie tablicy wskaźników do sortowanych struktur. W tej sytuacji, operacja przypisania ogranicza się do zmiany jednego pola tablicy. Z kolei operacja porównania może prowadzić do dereferencji wszystkich składowych struktury. Przy takim podejściu, czas trwania operacji porównania rośnie wraz ze wzrostem złożoności sortowanych danych.

Aby uwzględnić powyższy fakt, wprowadzono pojęcie współczynnika kosztu. Współczynnik kosztu okre-śla, ile razy operacja porównania jest czasowo dłuższa od operacji przypisania. Łączny koszt operacji, będący sumą ważoną operacji atomowych z uwzględnieniem współczynnika kosztu, określono wzorem:

$$C = \alpha n_c + 3n_s + n_a$$

C – łączny koszt operacji

α – wartość współczynnika kosztu



2.3 Założenia

Aby uprościć analizę algorytmów, dokonano szeregu uproszczeń oraz założeń. Uwzględnienie poniższych parametrów byłoby problematyczne z punktu widzenia systemu, zaś ich pominięcie nie wpływa znacząco na wynik analizy.

2.3.1 Operacje na iteratorach

Podczas zliczania operacji atomowych pominięte zostały operacje wykonywane na iteratorach oraz indeksach sortowanych tablic. Powodem tej decyzji jest fakt, że indeksy oraz iteratory w większości systemów są reprezentowane jako typy podstawowe, a więc koszt tych operacji jest relatywnie niski. Dodatkowo, ponieważ w większości algorytmów iterowana jest cała tablica, łączna liczba operacji na iteratorach oraz indeksach jest asymptotycznie równa dla wszystkich badanych algorytmów sortujących.

2.3.2 Rekurencja

Ponieważ wszystkie badane algorytmy są rekurencyjne, w analizie kosztu pominięto czas związany z alokacją i zwalnianiem ramek stosu. Dodatkowo pominięto czas związany z rekurencyjnym przejściem drzewa wywołań algorytmu.

2.3.3 Alokacja pamięci

W pracy badano złożoność czasową algorytmów działających w miejscu. Z tego powodu w analizie kosztu nie uwzględniano czasu związanego z alokacją oraz zwalnianiem pamięci dla zmiennych pomocniczych. Czas ten w przypadku algorytmów działających w miejscu jest rzędu $O(\log n)$, a więc nie wpływa znacząco na wynik analizy.

2.3.4 Dane wejściowe

Badając optymistyczną oraz pesymistyczną złożoność czasową algorytmów, przyjęto różne strategie generowania danych testowych. Losowe dane generowane są przy użyciu funkcji bibliotecznej, oraz mogą zawierać powtórzenia. Posortowane dane testowe to permutacja identycznościowa. Dane posortowane w odwrotnej kolejności to permutacja, dla której liczba inwersji jest maksymalna.

Przegląd podstawowych algorytmów sortujących

3.1 Quick Sort

algorytm stabilny	NIE
algorytm wykonywany w miejscu	TAK
optymistyczna złożoność obliczeniowa	$O(n \log n)$
pesymistyczna złożoność obliczeniowa	$O(n^2)$
średnia złożoność obliczeniowa	$O(n \log n)$
złożoność pamięciowa	$O(1)$

Historia algorytmu Quick Sort sięga drugiej połowy XX wieku. W roku 1959 brytyjski naukowiec Tony Hoare opracował, a dwa lata później opublikował jego pierwszą wersję. Od tamtego czasu powstało wiele udoskonaleń tego algorytmu, jednak jego koncepcja nadal jest widoczna we współczesnych językach programowania¹. Na cześć algorytmu Quick Sort standardowa funkcja sortująca w języku C++ nosi nazwę qsort².

Algorytm Quick Sort składa się z dwóch etapów. Pierwszym z nich jest partycjonowanie zbioru wejściowego. Po tym kroku tablica wejściowa jest rozbita na dwa rozłączne zbiory, w których wszystkie elementy pierwszego zbioru są skumulowane po lewej stronie tablicy oraz każdy z tych elementów jest większy od dowolnego elementu z drugiej tablicy. Drugim etapem jest rekurencyjne sortowanie lewej oraz prawej podtablicy. Algorytm Quick Sort wykorzystuje technikę dziel i zwyciężaj, rozbijając problem sortowania tablicy wejściowej na sortowanie dwóch podtablic.

3.1.1 Analiza algorytmu Quick Sort

Liczba operacji wykonywanych przez algorytm Quick Sort została przeanalizowana pod kątem trzech przypadków: optymistycznego, średniego oraz pesymistycznego.

Dla algorytmu Quick Sort przypadek optymistyczny (3.1) następuje wówczas, gdy algorytm partycjonowania przy każdym wywołaniu dzieli tablicę wejściową na dwie równe części. Efekt ten uzyskano, wprowadzając dane już posortowane oraz stosując algorytm wybierający piwot dokładnie w połowie tablicy. Analiza eksperymentalna sugeruje, że w przypadku optymistycznym algorytm działa ze złożonością obliczeniową $O(n \log n)$.

¹Dokumentacja biblioteki sortującej w języku java: <https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/util/Arrays.html>

²Dokumentacja funkcji sortującej qsort: <https://en.cppreference.com/w/cpp/algorithm/qsort>



Rysunek 3.1

Przypadek pesymistyczny (3.2) zachodzi, gdy drzewo wwołań rekurencyjnych jest możliwie najgłębsze. Efekt ten uzyskano, wprowadzając dane już posortowane oraz stosując algorytm wybierający piwot jako ostatni element tablicy. W tej sytuacji w kolejnych iteracjach rozpatrywana jest tablica z rozmiarem o jeden mniejszym od poprzedniej, a więc drzewo wywołań rekurencyjnych ma głębokość n . Analiza eksperymentalna sugeruje, że złożoność obliczeniowa algorytmu w przypadku pesymistycznym wynosi $O(n^2)$.

Przypadek średni (3.1) został zbadany wprowadzając losowe dane z powtórzeniami oraz stosując algorytm wybierający piwot jako ostatni element tablicy. Analiza eksperymentalna sugeruje, że w przypadku średnim algorytm Quick Sort ma złożoność obliczeniową równą $O(n \log n)$, a więc jest tego samego rzędu co dla przypadku optymistycznego.

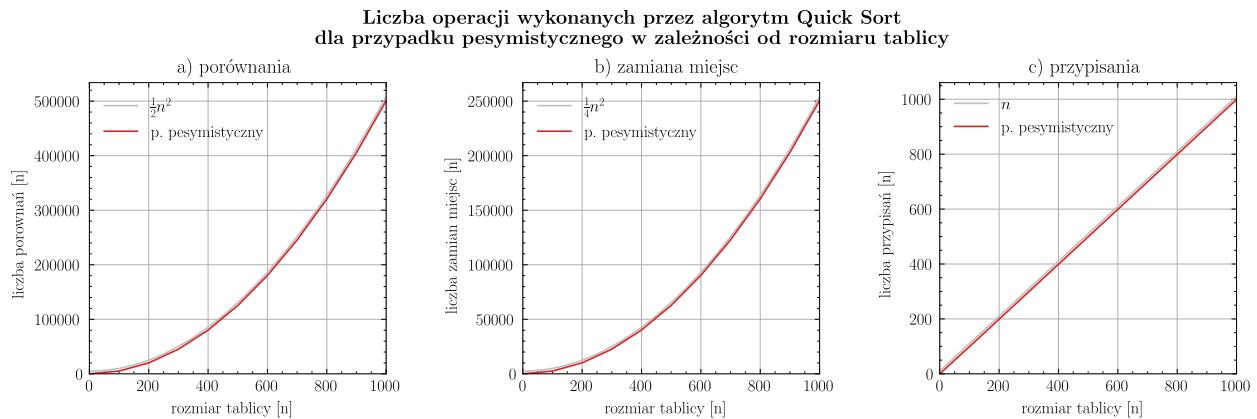
Porównując liczbę wykonywanych operacji można zauważać, że algorytm Quick Sort wykonuje prawie dwa razy więcej operacji porównania niż operacji zamiany miejsc. Liczba pojedynczych operacji przypisania rośnie liniowo, a więc jest znacznie mniejsza w porównaniu z liczbą pozostałych operacji.

Analizując histogram liczby wykonanych operacji (3.3) użyto tablicy losowych danych o stałym rozmiarze $n = 10000$. Można zauważyć, że histogramy operacji porównania oraz operacji zamiany miejsc, w porównaniu do rozkładu normalnego są asymetryczne, co może sugerować, że zmienne losowe liczby tych operacji nie są rozkładu normalnego. Z kolei histogram liczby wykonanych operacji przypisania jest symetryczny, a więc z jednakowym prawdopodobieństwem liczba operacji przypisania może być większa lub mniejsza od wartości średniej.

3.1.2 Problemy związane z algorytmem Quick Sort

Głównym problemem algorytmu Quick Sort jest jego słaba pesymistyczna złożoność obliczeniowa. Ponieważ algorytm działa rekurencyjnie, w przypadku pesymistycznym głębokość drzewa wywołań rekurencyjnych może przekroczyć maksymalną liczbę ramek stosu, powodując awaryjne zatrzymanie programu. Ponieważ pesymistyczna złożoność jest rzędu $O(n^2)$, czas trwania algorytmu Quick Sort dla tego przypadku może być nieakceptowalny.

Kolejnym problemem tego algorytmu jest stosunkowo duża liczba wykonywanych operacji porównania w stosunku do liczby pozostałych operacji. Punkt ten jest szczególnie istotny w sytuacji, gdy sortowane są złożone struktury, dla których wykonanie pojedynczej operacji porównania może być znacznie kosztowniejsze od pozostałych operacji. W tym przypadku bardziej wskazanym wydaje się użycie algorytmu Merge Sort, którego analizę przeprowadzono w kolejnym rozdziale.



Rysunek 3.2



Rysunek 3.3

3.1.3 Możliwości optymalizacyjne

Ponieważ złożoność czasowa algorytmu Quick Sort uwarunkowana jest poprzez złożoność algorytmu partycjonowania, optymalizacja algorytmu może opierać się na ulepszaniu algorytmu partycjonowania, aby jak najefektywniej wyszukiwał pivot, zapewniając podział tablicy wejściowej na dwie tablice o zbliżonej długości.

3.2 Merge Sort

algorytm stabilny	TAK
algorytm wykonywany w miejscu	NIE
optymistyczna złożoność obliczeniowa	$O(n \log n)$
pesymistyczna złożoność obliczeniowa	$O(n \log n)$
średnia złożoność obliczeniowa	$O(n \log n)$
złożoność pamięciowa	$O(n)$

Algorytm Merge Sort został opracowany przez Johna von Neumanna w 1945 roku. Tak jak Quick Sort, algorytm ten wykorzystuje technikę dziel i zwycięzaj aby rekurencyjnie rozbić problem sortowania tablicy wejściowej na dwie tablice mniejszej długości. W przeciwieństwie do algorytmu Quick Sort, algorytm Merge Sort do poprawnego działania potrzebuje dodatkowej pamięci. W trakcie analizy badano zoptymalizowaną wersję tego algorytmu, dla której operacja scalania odbywa się na tablicy wejściowej. Z tego powodu program alokuje $n/2$ dodatkowej pamięci tylko dla pierwszej połowy tablicy sortowanych elementów.

Algorytm Merge Sort składa się z trzech etapów. Pierwszym krokiem jest podział tablicy wejściowej na dwie części o zbliżonej długości. Drugim etapem jest rekurencyjne sortowanie każdej z części. Ostatnim krokiem jest scalenie tablic częściowych zgodnie z porządkiem sortowania. Aby efektywnie wykonać ostatni krok, algorytm Merge Sort potrzebuje zewnętrznego bufora.

3.2.1 Analiza algorytmu Merge Sort

Liczba operacji wykonywanych przez algorytm Merge Sort została przeanalizowana pod kątem trzech przypadków: optymistycznego, średniego oraz pesymistycznego.

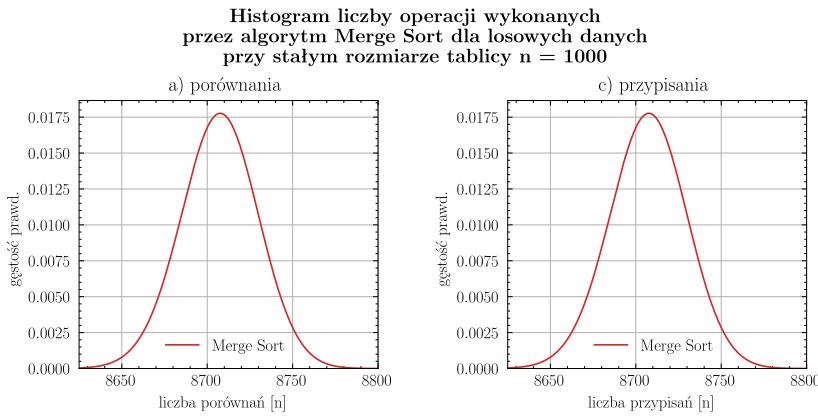
Dla algorytmu Merge Sort przypadek optymistyczny (3.4) następuje wówczas, gdy w każdym kroku scalania lewa podtablica zawiera tylko elementy mniejsze od wszystkich elementów z prawej podtablicy. Efekt ten uzyskano, wprowadzając dane już posortowane. Analiza eksperimentalna sugeruje, że w tym przypadku algorytm Merge Sort działa ze złożonością obliczeniową rzędu $O(n \log n)$.

Przypadek pesymistyczny (3.4) zachodzi, gdy liczba porównań elementów w trakcie scalania jest możliwie największa. Efekt ten uzyskano, preparując dane wejściowe w taki sposób, aby w każdym kroku scalania porównywane tablice zawierały elementy na przemian większe oraz mniejsze. Analiza eksperimentalna sugeruje, że złożoność obliczeniowa algorytmu w przypadku pesymistycznym jest równa $O(n \log n)$, a więc jest tego samego rzędu co złożoność optymistyczna.

Przypadek średni (3.4) zbadano wprowadzając na wejście losowe dane z powtórzeniami. Analiza eksperimentalna sugeruje, że w przypadku średnim złożoność algorytmu jest równa $O(n \log n)$.



Rysunek 3.4



Rysunek 3.5

W przeciwieństwie do algorytmu Quick Sort, algorytm Merge Sort wykonuje dokładnie tyle samo operacji porównania co operacji przypisania. Jest to zgodne z rzeczywistością, ponieważ w trakcie scalania każdej operacji porównania towarzyszy przypisanie wartości do bufora. Można zauważyć, że badana implementacja algorytmu Merge Sort nie wykonuje operacji zamiany miejsc.

Do analizy histogramu liczby wykonanych operacji (3.5) użyto tablicy losowych danych o stałym rozmiarze $n = 1000$. W analizie pominięto liczbę operacji przypisania, która jest zerowa dla każdego przypadku. Analiza eksperymentalna sugeruje, że w przeciwieństwie do algorytmu Quick Sort, liczba operacji porównania dla algorytmu Merge Sort jest symetryczna, a więc prawdopodobieństwa otrzymania większej oraz mniejszej liczby operacji przypisania są jednakowe. Ponieważ dla badanego algorytmu, liczba operacji przypisania jest równa liczbie operacji porównania, można założyć, że zmienne losowe liczby tych operacji są jednakowe.

3.2.2 Problemy związane z algorytmem Merge Sort

Głównym problemem algorytmu Merge Sort jest konieczność alokacji dodatkowego bufora pamięci. W rozważanej implementacji Merge Sort alokuje dodatkowy bufor długości $n/2$, a więc algorytm ten może okazać się bezużyteczny dla systemów z ograniczonym zasobem pamięci.



3.2.3 Możliwości optymalizacyjne

Jednym ze sposobów na optymalizację algorytmu Merge Sort jest próba przekształcenia go w algorytm działający w miejscu. Cel ten może zostać osiągnięty poprzez połączenie algorytmu Merge Sort z innym algorytmem sortującym w taki sposób, aby bufor dodatkowej pamięci był częścią tablicy wejściowej. Rozwiązanie to zostanie przeanalizowane w dalszych rozdziałach.

Przegląd hybrydowych algorytmów sortujących

4.1 Główne sposoby modyfikacji algorytmów

W celu poprawy wydajności algorytmów sortujących stosuje się ich modyfikacje oraz ulepszenia. W tej pracy wykorzystano dwa główne sposoby na usprawnienie algorytmów.

Pierwszym sposobem jest modyfikacja składowych danego algorytmu. Wiele spośród znanych algorytmów składa się z kilku osobnych kroków, z których każdy można wyekstrahować do oddzielnego procesu. Pomyśl ten polega na modyfikacji składowych algorytmu sortującego w taki sposób, aby np. lepiej radził on sobie w przypadku pesymistycznym.

Drugim sposobem na ulepszenie jest próba połączenia wielu algorytmów sortujących. Niektóre algorytmy zachowują się lepiej dla stosunkowo małej ilości danych, inne zaś są znaczenie wydajniejsze przy rozbudowanym zbiorze danych wejściowych. Pomyśl ten polega na opracowaniu algorytmu, którego działanie zmienia się w zależności od czynników zewnętrznych, np. długości danych do posortowania.

4.2 Rodzina deterministycznych algorytmów Quick Sort

Podczas analizy algorytmów sortujących z rodziny Quick Sort zostały przetestowane wariacje algorytmów z różnymi metodami partycjonowania oraz różnymi deterministycznymi metodami wyboru piwota. Do partycjonowania danych wejściowych wykorzystano poniższe metody:

- **metoda Lomuto** - jest domyślnym algorytmem partycjonowania w projektowanym systemie. Tablica jest iterowana od pierwszego do ostatniego elementu. Elementy mniejsze od piwota są przenoszone na lewą część tablicy, zaś elementy większe od piwota na jej prawą część. Algorytm kończy się w momencie przeniesienia ostatniego elementu.
- **metoda Hoare** - tablica jest partycjonowana za pomocą dwóch iteratorów umieszczonych po przeciwnych stronach tablicy oraz skierowanych do jej środka. Pojedyncza iteracja trwa do momentu napotkania dwóch elementów, które nie znajdują się w odpowiednich częściach tablicy, tzn. element po lewej stronie jest większy od piwota, oraz element po prawej stronie jest mniejszy od piwota. Wtedy elementy znajdujące się w miejscu iteratorów są zamieniane miejscami oraz algorytm jest kontynuowany. Program kończy się w momencie spotkania obydwu iteratorów.

Wykonując testy brano pod uwagę następujące deterministyczne metody wyboru piwota:

- **ostatni element** - piwotem jest ostatni element tablicy,
- **mediana z trzech** - piwotem jest mediana z pierwszego, środkowego oraz ostatniego elementu tablicy,
- **pseudo-mediana z dziewięciu** - piwotem jest mediana z dziewięciu równo oddalonych od siebie elementów, z których pierwszym jest pierwszy element tablicy, oraz ostatnim jest ostatni element tablicy,

- **medianą-median z pięciu** - elementy tablicy są grupowane z zbiory 5-elementowe, z każdego zbioru wybierana jest mediana, następnie algorytm jest wywoływany rekurencyjnie dla wybranych elementów,
- **medianą-median z trzech** - tablica jest dzielona na trzy równe części, w każdej z części rekurencyjnie wyszukiwana jest mediana, następnie zwracana jest mediana z otrzymanych wyników.

4.2.1 Analiza porównawcza algorytmów

Analizując wydajność algorytmów z rodziny Quick Sort, badano liczbę wykonywanych operacji atomowych z podziałem na operacje porównania, zamiany miejsc oraz przypisania. Dodatkowo badano łączny koszt wykonanych operacji jako sumę ważoną liczby operacji atomowych, z uwzględnieniem wartości współczynnika kosztu. Badając łączną liczbę wykonanych operacji założono stałą wartość współczynnika kosztu $\alpha = 1.0$.

Dla losowych danych wejściowych (4.1), czyli przypadku średniego w klasycznym algorytmie Quick Sort, przy założeniu że operacja porównania jest czasowo równoważna operacji przypisania ($\alpha = 1.0$), partycjonowanie metodą Hoare jest wydajniejsze lub tak samo wydajne jak partycjonowanie metodą Lomuto. Najlepsze wyniki otrzymano poprzez połączenie partycjonowania metodą Hoare z wyborem piwota jako ostatni element tablicy. Najgorsze wyniki otrzymano wybierając piwot jako mediana-median z pięciu, przy czym najgorszy wynik osiągnięto niezależnie od wyboru metody partycjonowania.

Dla danych wejściowych posortowanych w odwrotnej kolejności (4.2), czyli dla przypadku pesymistycznego w klasycznym algorytmie Quick Sort, przy założeniu że operacja porównania jest czasowo równoważna operacji przypisania ($\alpha = 1.0$), partycjonowanie metodą Hoare jest również wydajniejsze lub tak samo wydajne jak partycjonowanie metodą Lomuto. Najlepsze wyniki otrzymano poprzez połączenie partycjonowania metodą Hoare z wyborem piwota jako mediana z trzech. Najgorsze wyniki otrzymano dla klasycznego algorytmu Quick Sort, czyli poprzez połączenia partycjonowania metodą Lomuto z wyborem piwota jako ostatni element tablicy.

Analizując łączny koszt wykonanych operacji (4.3) w zależności od wartości współczynnika kosztu α można zauważyć, że dla typów danych o współczynniku kosztu większym bądź równym wartości $\alpha = 2.0$, partycjonowanie metodą Lomuto jest wydajniejsze niż partycjonowanie metodą Hoare. Wyniki te można interpretować jako sortowanie struktur złożonych z co najmniej dwóch typów prostych. Najsłuszniejszą z badanych strategii sortowania jest partycjonowanie metodą Lomuto z wyborem piwota jako mediana z trzech elementów. Metoda ta jest najsłuszniejsza powyżej wartości $\alpha = 2.0$. Najmniejszą skuteczność ma wybór piwota jako mediana-median z pięciu elementów, przy czym wynik najgorszy osiągnięto dla obydwu metod partycjonowania.

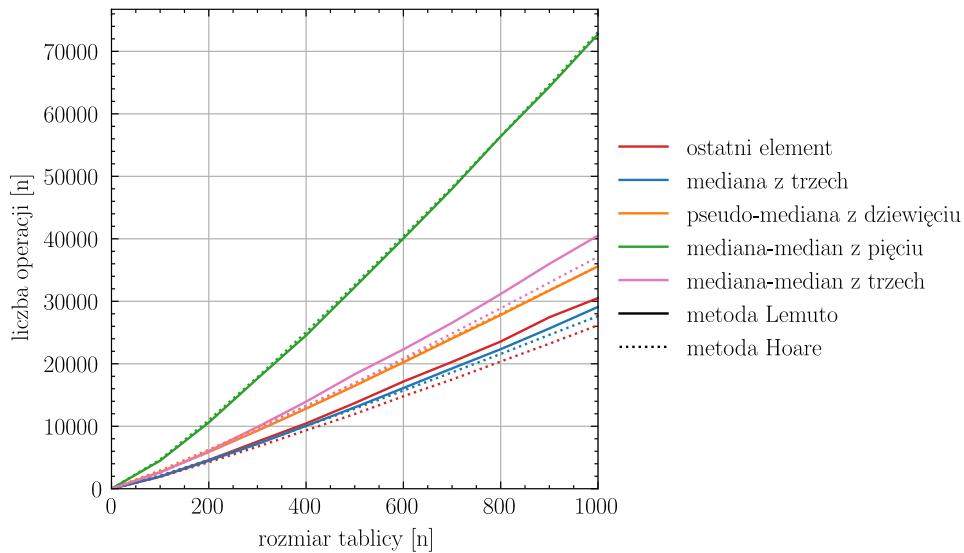
Porównując liczbę wykonanych operacji pomiędzy badanymi algorytmami partycjonowania (4.5) można stwierdzić, że partycjonowanie metodą Hoare wykonuje większą liczbę operacji porównania oraz mniejszą liczbę operacji zamiany miejsc. Czynnik ten może okazać się szczególnie istotny w przypadku sortowania złożonych struktur danych.

Do analizy histogramu liczby wykonanych operacji (4.4) użyto tablicy losowych danych o stałym rozmiarze $n = 1000$ oraz współczynnika kosztu o stałej wartości $\alpha = 1.0$. Dla badanych algorytmów najmniejsze odchylenie standardowe mają algorytmy partycjonowania metodą Hoare przy wyborze piwota jako pseudo-mediana z dziewięciu lub mediana-median z trzech. Największe odchylenie standardowe występuje dla klasycznego algorytmu Quick Sort. Najmniejsza średnia liczba wykonanych operacji jest osiągana poprzez partycjonowanie metodą Hoare z wyborem piwota jako ostatni element tablicy.

4.2.2 Wnioski

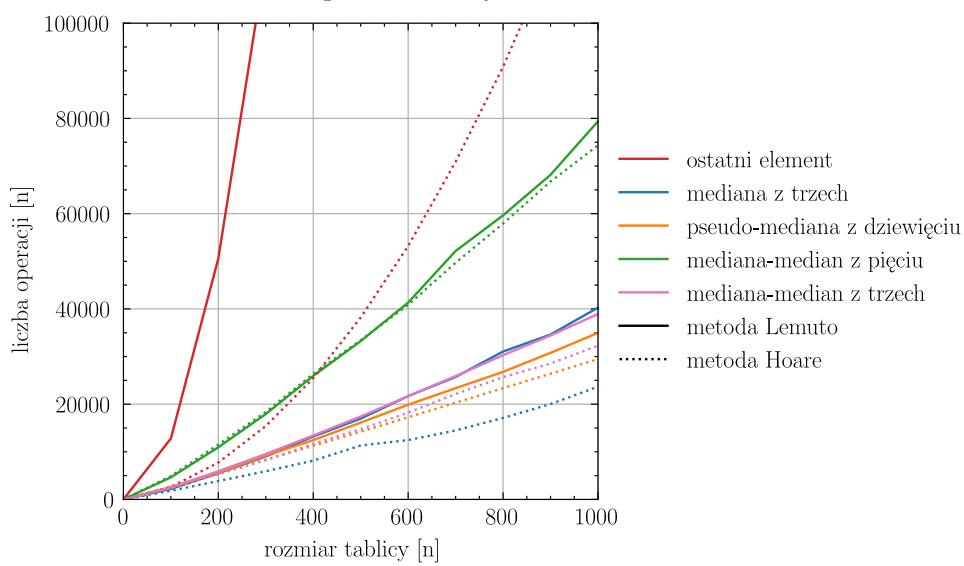
W przypadku danych wejściowych typu prostego, tzn. takich, dla których operacja porównania jest czasowo równoważna operacji przypisania, partycjonowanie metodą Hoare okazało się wydajniejsze niż partycjonowanie metodą Lomuto. W przypadku złożonych struktur danych, wydajniejsze jest partycjonowanie

Łączna liczba operacji wykonanych przez deterministyczne wersje algorytmu Quick Sort z podziałem na polityki wyboru pivota oraz metodę partycjonowania dla tablicy losowych danych



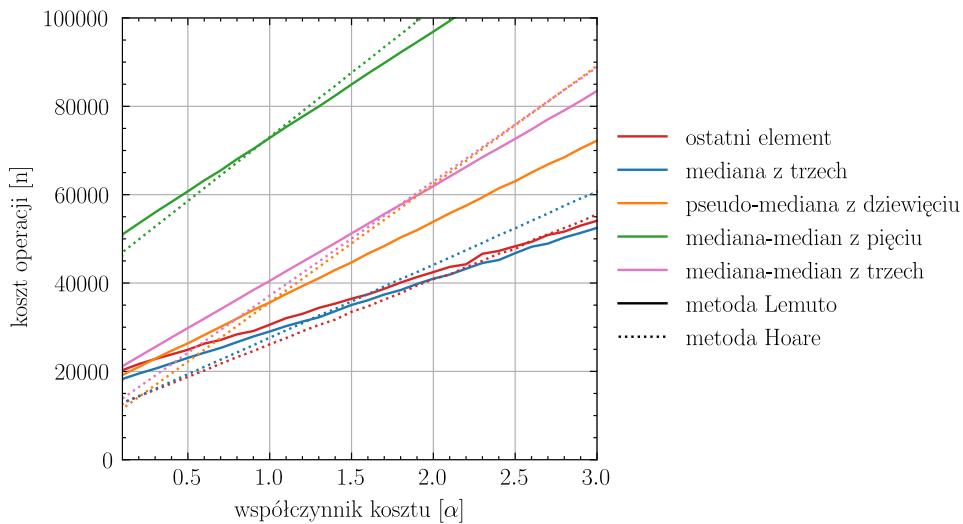
Rysunek 4.1

Łączna liczba operacji wykonanych przez deterministyczne wersje algorytmu Quick Sort z podziałem na polityki wyboru pivota oraz metodę partycjonowania dla tablicy danych posortowanych odwrotnie



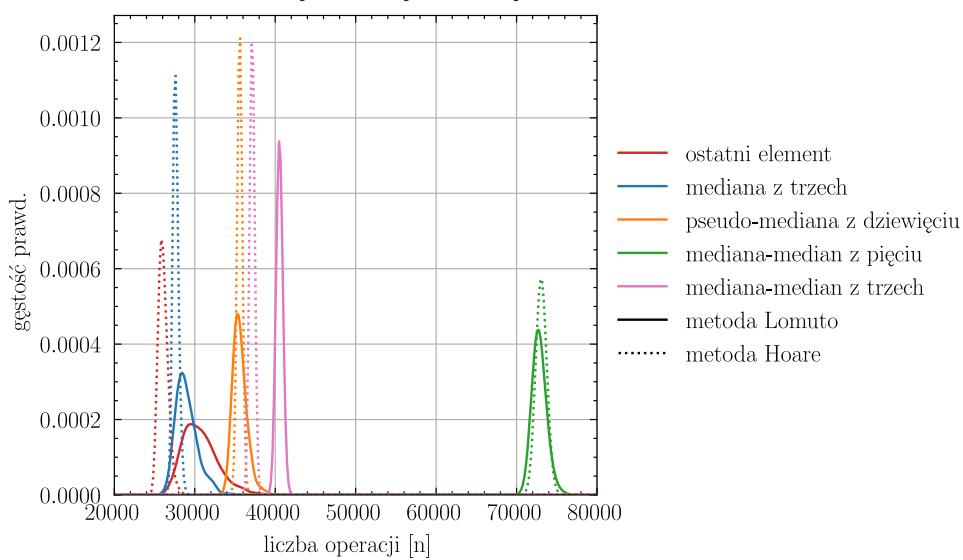
Rysunek 4.2

Łączny koszt operacji wykonanych przez deterministyczne wersje algorytmu Quick Sort z podziałem na polityki wyboru pivota oraz metody partycjonowania w zależności od wartości współczynnika kosztu dla tablicy losowych danych rozmiaru $n = 1000$



Rysunek 4.3

Histogram liczby operacji wykonanych przez deterministyczne wersje algorytmu Quick Sort dla tablicy losowych danych rozmiaru $n = 1000$



Rysunek 4.4



Rysunek 4.5

metodą Lomuto. Wartość współczynnika kosztu, od której partycjonowanie metodą Lomuto jest bardziej wydajne dla dowolnej strategii wyboru piwota wynosi $\alpha = 2.0$, co można interpretować jako sortowania struktur danych złożonych z co najmniej dwóch typów prostych.

Analizując zachowanie algorytmów można stwierdzić, że dla danych posortowanych lub prawie posortowanych, dobrym wyborem jest skorzystanie z kosztownej metody wyszukiwania piwota, która pomimo swojego dodatkowego nakładu czasowego zwiększa prawdopodobieństwo znalezienia dobrego piwota. Wyjątkiem jest wybór piwota jako mediana-median z pięciu, w przypadku którego dodatkowy nakład czasowy sprawia, że algorytm jest znacznie mniej wydajny nawet dla uporządkowanych danych wejściowych. Przez pojęcie **dobrego piwota** rozumiemy tutaj element, którego pozycja w procedurze partycjonowania znajduje się możliwie blisko środka tablicy.

4.3 Rodzina randomizowanych algorytmów Quick Sort

Klasyczny algorytm Quick Sort kiepsko sobie radzi z uporządkowanymi lub prawie uporządkowanymi danymi wejściowymi. W najmniej skutecznym wariancie, tzn. podczas wyboru piwota jako ostatni element tablicy, algorytm ten działa ze złożonością czasową $O(n^2)$. W przypadku uporządkowanych danych wejściowych skutecznym sposobem może okazać się niedeterministyczny wybór piwota. W tym rozdziale dokonano analizy randomizowanych algorytmów z rodziny Quick Sort, z podziałem na metody partycjonowania Lomuto oraz Hoare. W analizie wykorzystano następujące metody wyboru piwota:

- **losowy element** - piwotem jest losowo wybrany element tablicy,
- **mediana z trzech wyborów** - przystosowanie metody **power of two choices** do potrzeby wyznaczania mediany, piwotem jest mediana z trzech losowo wybranych elementów,
- **pseudo-mediana z dziewięciu wyborów** - piwotem jest mediana z dziewięciu losowo wybranych elementów.

4.3.1 Analiza porównawcza algorytmów

Tak jak w poprzednim rozdziale, algorytmy były analizowane pod kątem liczby wykonywanych operacji atomowych, z podziałem na operacje porównania, zamiany miejsc oraz przypisania. Dodatkowo badano łączny koszt wykonanych operacji jako sumę ważoną liczby operacji atomowych, z uwzględnieniem wartości współczynnika kosztu. Badając łączną liczbę wykonanych operacji założono stałą wartość współczynnika kosztu $\alpha = 1.0$.

Dla losowych danych wejściowych (4.6), przy założeniu że operacja porównania jest czasowo równoważna operacji przypisania ($\alpha = 1.0$), większość randomizowanych metod wyboru piwota okazuje się mniej skuteczna od klasycznego algorytmu Quick Sort. Przy partycjonowaniu metodą Lomuto jedyną skuteczniejszą metodą jest wybór piwota jako mediana z trzech losowych elementów. Przy partycjonowaniu metodą Hoare, najskuteczniejszym okazuje się wybór za piwota losowego elementu tablicy. Połączenie metody Hoare z losowym wyborem piwota jest również najskuteczniejszym podejściem podczas sortowania losowych danych. Najmniej wydajną strategią sortowania okazał się wybór piwota jako pseudo-mediana z dziewięciu losowych elementów, przy czym najgorszy wynik osiągnięto dla obydwu metod partycjonowania.

Dla danych wejściowych posortowanych w odwrotnej kolejności (4.7), czyli dla przypadku pesymistycznego w klasycznym algorytmie Quick Sort, przy założeniu że operacja porównania jest czasowo równoważna operacji przypisania ($\alpha = 1.0$), najskuteczniejszą strategią sortowania również okazało się partycjonowanie metodą Hoare przy wyborze piwota jako losowy element tablicy. Spośród randomizowanych strategii sortowania najmniej wydajną okazał się wybór piwota jako pseudo-mediana z dziewięciu losowych elementów. W przypadku danych wejściowych posortowanych w odwrotnej kolejności, każda ze strategii randomizowanych jest wydajniejsza od klasycznego podejścia w metodzie Quick Sort.

Analizując łączny koszt wykonanych operacji (4.8) w zależności od wartości współczynnika koszta α można zauważyć, że dla złożonych typów danych najskuteczniejszą z badanych strategii jest partycjonowanie metodą Lomuto z wyborem piwota jako mediana z trzech losowych elementów. Metoda ta jest najskuteczniejsza powyżej wartości $\alpha = 2.5$, co można interpretować jako sortowanie struktur złożonych z co najmniej trzech typów prostych. Najmniej skutecną strategią sortowania złożonych struktur jest wybór piwota pseudo-mediana z девицием losowych elementów.

Porównując liczbę wykonanych operacji (4.10) można stwierdzić, że dla dowolnych randomizowanych strategii wyboru piwota, partycjonowanie metodą Hoare wykonuje większą liczbę operacji porównania oraz mniejszą liczbę operacji zamiany miejsc. Tak jak w przypadku algorytmów deterministycznych, czynnik ten może okazać się istotny w przypadku sortowania złożonych struktur.

Badając histogram liczby wykonanych operacji dla algorytmów randomizowanych (4.9) można zauważyć, że algorytmy korzystające z partycjonowania metodą Hoare mają mniejsze odchylenie standardowe niż algorytmy z partycjonowaniem metodą Lomuto. Każdy z randomizowanych algorytmów wyboru piwota daje w wyniku mniejsze odchylenie standardowe od klasycznego algorytmu Quick Sort. Najmniejsze odchylenie standardowe uzyskano dla partycjonowania metodą Hoare z wyborem piwota jako pseudo-mediana z девицием wybór, jednak równocześnie dla tego algorytmu uzyskano największą średnią wartość liczby wykonanych operacji. Najmniejszą średnią liczbę operacji uzyskano dla partycjonowania metodą Hoare przy wyborze piwota jako losowy element tablicy.

4.3.2 Wnioski

W przypadku randomizowanych algorytmów z rodziny Quick Sort, podobnie jak dla algorytmów deterministycznych, partycjonowanie metodą Hoare jest wydajniejsze dla danych wejściowych typu prostego. Należy jednak zauważyć, że partycjonowanie metodą Hoare wykonuje większą liczbę operacji porównania, więc strategia ta jest mniej wydajna w przypadku sortowania złożonych struktur danych. Powyżej wartości współczynnika koszta równej $\alpha = 4.0$, partycjonowanie metodą Lomuto jest wydajniejsze niż partycjonowanie metodą Hoare, niezależnie od strategii wyboru piwota. Sytuacja ta jest równoważna z sortowaniem struktur składających się z co najmniej czterech zmiennych podstawowych. W tym przypadku należy skorzystać z partycjonowania metodą Lomuto.

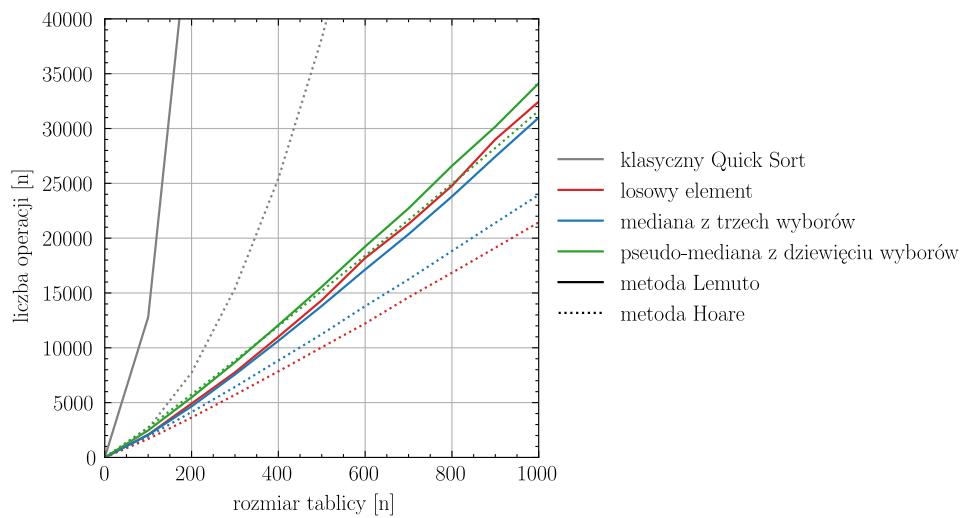
Analizując liczbę wykonanych operacji można stwierdzić, że w przypadku algorytmów randomizowanych korzystanie z kosztownych strategii wyboru piwota jest nieopłacalne. Najskuteczniejszą metodą, zarówno dla losowych jak i posortowanych danych, okazuje się wybór losowego elementu tablicy.

Łączna liczba operacji wykonanych przez randomizowane wersje algorytmu Quick Sort z podziałem na polityki wyboru pivota oraz metodę partycjonowania dla tablicy losowych danych



Rysunek 4.6

Łączna liczba operacji wykonanych przez randomizowane wersje algorytmu Quick Sort z podziałem na polityki wyboru pivota oraz metodę partycjonowania dla tablicy danych posortowanych odwrotnie



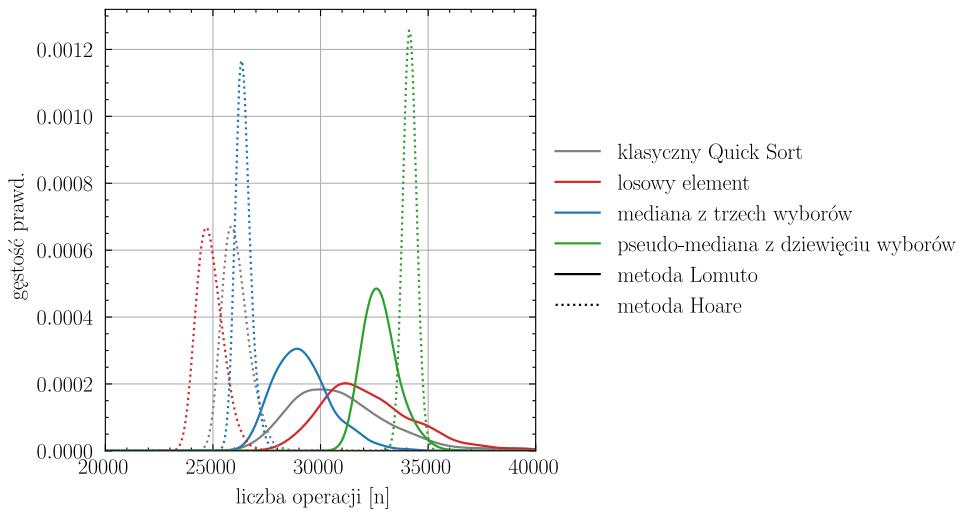
Rysunek 4.7

Łączny koszt operacji wykonanych przez randomizowane wersje algorytmu Quick Sort z podziałem na polityki wyboru pivota oraz metody partycjonowania w zależności od wartości współczynnika kosztu dla tablicy losowych danych rozmiaru $n = 1000$



Rysunek 4.8

Histogram liczby operacji wykonanych przez randomizowane wersje algorytmu Quick Sort dla tablicy losowych danych rozmiaru $n = 1000$



Rysunek 4.9



Rysunek 4.10

4.4 QuickMerge Sort

Jednym z praktycznych zastosowań algorytmów sortujących jest porządkowanie złożonych struktur danych. Dla danych tego typu koszt pojedynczej operacji porównania może być nawet kilkukrotnie większy niż koszt operacji przypisania lub zamiany miejsc. Różnica ta jest szczególnie widoczna, jeśli sortowane dane są przechowywane w formie tablicy wskaźników do struktur. W tej sytuacji, operacja przypisania ogranicza się do zmiany jednego pola tablicy, jednak operacja porównania może prowadzić do dereferencji wszystkich składowych struktury. W tym przypadku stosowanie klasycznego algorytmu Quick Sort nie jest zalecane z powodu dużej liczby kosztownych operacji porównania. Znacznie bardziej wydajnym podejściem wydaje się zastosowanie algorytmu Merge Sort, jednak w tym przypadku należy uwzględnić dodatkowe koszty związane z alokacją pamięci. Co więcej, ze względu na ograniczenia systemowe, w niektórych sytuacjach korzystanie z dodatkowego bufora pamięci jest niemożliwe. Ograniczenia te stały się motywacją do opracowania algorytmu QuickMerge Sort, którego analizę przedstawiono w bieżącym rozdziale.

Pomysł na algorytm QuickMerge Sort został przedstawiony w publikacji “**QuickXsort – A Fast Sorting Scheme in Theory and Practice**”, której autorem jest Sebastian Wild. QuickMerge Sort to algorytm hybrydowy, będący połączeniem algorytmów Quick Sort oraz Merge Sort. Celem tego algorytmu jest minimalizacja liczby porównań, przy równoczesnym zachowaniu miejscowego działania algorytmu, bez konieczności alokacji dodatkowego bufora pamięci. Zasadniczą koncepcją tego algorytmu jest sortowanie algorytmem Merge Sort w taki sposób, aby buforem potrzebnym do scalenia tablic częściowych był fragment tej samej tablicy wejściowej.

Ponieważ partycjonowanie metodą Hoare wykonuje więcej operacji porównania niż partycjonowanie metodą Lomuto, zaś stosowanie algorytmu QuickMerge Sort przynosi realne zyski podczas sortowania złożonych struktur danych, w analizie porównawczej uwzględniono tylko algorytm partycjonowania metodą Lomuto.

4.4.1 Pseudokod

QuickMerge Sort jest algorytmem rekurencyjnym składającym się z trzech etapów. Pierwszym etapem jest partycjonowanie zbioru wejściowego. W kolejnym kroku wybierana jest mniejsza z otrzymanych partycji. Część ta jest sortowana przy użyciu algorytmu Merge Sort, przy czym bufor używany do scalenia tablic częściowych stanowi druga partycja. W ostatnim kroku sortowana jest druga partycja w sposób rekurencyjny. Ponieważ buforem w trakcie sortowania jest fragment tej samej tablicy, należy zagwarantować, że algorytm scalający nie nadpisze danych zawartych w buforze. Problem ten rozwiązano, stosując operację zamiany miejsc w miejsce klasycznej operacji przypisania.

Algorytm 1 QuickMerge Sort

```

1: procedure QUICKMERGESORT(ARR)
2:
3:   if len(arr) = 1 then end
4:   end if                                ▷ warunek wyjścia
5:
6:   arr1, arr2 ← Partition(arr)          ▷ partycjonowanie
7:
8:   if len(arr1) < len(arr2) then      ▷ sortowanie
9:     buffer ← arr2
10:    MergeSortBySwaps(arr1, buffer)
11:    QuickMergeSort(arr2)
12:   else
13:     buffer ← arr1
14:     MergeSortBySwaps(arr2, buffer)
15:     QuickMergeSort(arr1)
16:   end if
17:
18: end procedure

```

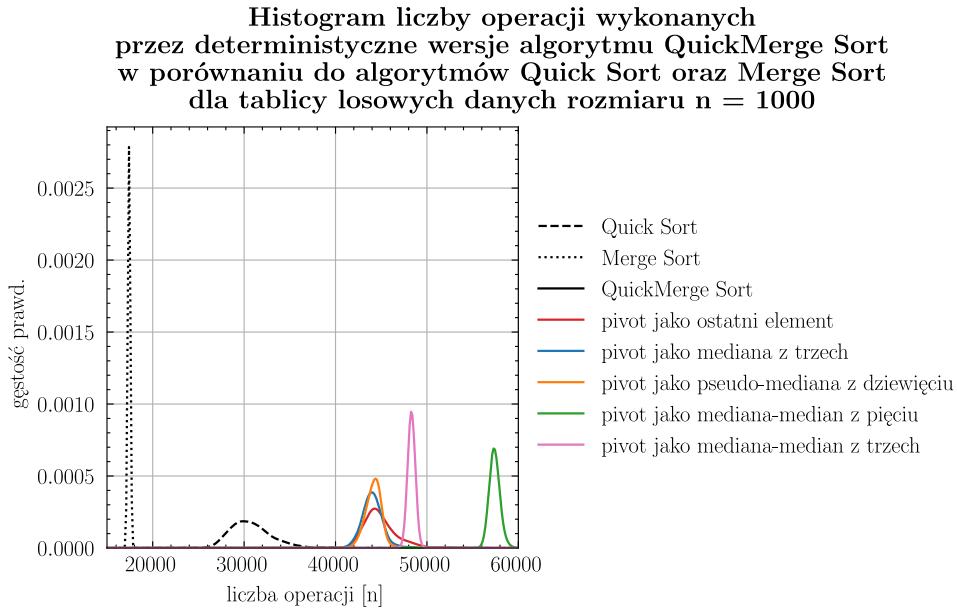
4.4.2 Analiza deterministycznych wersji algorytmu QuickMerge Sort

Analizując wydajność deterministycznych algorytmów z rodziny QuickMerge Sort, badano liczbę wykonywanych operacji atomowych z podziałem na operacje porównania, zamiany miejsc oraz przypisania. Dodatkowo badano łączny koszt wykonanych operacji jako sumę ważoną liczby operacji atomowych, z uwzględnieniem wartości współczynnika kosztu. Badając liczbę poszczególnych operacji uwzględniono algorytm Merge Sort oraz wersję algorytmu Quick Sort z partycjonowaniem metodą Lomuto oraz wyborem piwota jako ostatni element tablicy.

Badając liczbę wykonanych operacji porównania (4.13) można stwierdzić, że spośród badanych algorytmów działających w miejscu, najbardziej wydajnym algorytmem jest QuickMerge Sort z partycjonowaniem metodą Lomuto oraz wyborem piwota jako pseudo-medianą z dziewięciu. Podobne, nieznacznie gorsze wyniki osiąga metoda wyboru piwota jako mediana z trzech elementów. Dla obydwu tych strategii liczba wykonywanych operacji porównania jest mniejsza niż w przypadku algorytmu Quick Sort.

Analizując liczbę operacji zamiany miejsc oraz przypisania (4.13) można zauważać, że niezależnie od strategii wyboru piwota, algorytm QuickMerge Sort wykonuje około dwa razy więcej operacji zamiany miejsc niż klasyczny algorytm Quick Sort. Fakt ten może okazać się szczególnie istotny w przypadku sortowania złożonych struktur, dla których koszt operacji przypisania jest znacznie mniejszy niż koszt operacji porównania. Ponieważ dla większości badanych algorytmów działających w miejscu liczba operacji przypisania jest znikoma w porównaniu do pozostałych operacji, ich obecność nie wpływa na wydajność algorytmów.

Badając łączny koszt wykonanych operacji (4.12) dla danych typu podstawowego ($\alpha = 1.0$) można zauważać, że najwydajniejszym spośród badanych algorytmów działających w miejscu jest klasyczny algorytm Quick Sort. Dla danych tego typu korzystanie z algorytmu QuickMerge Sort powoduje dodatkowy nakład czasowy. Sytuacja ta zmienia się w przypadku sortowania złożonych struktur wejściowych, dla których ($\alpha \geq 7.0$). Powyżej tej wartości, zastosowanie strategii wyboru piwota jako medianą z trzech lub pseudo-medianą z dziewięciu w połączeniu z algorytmem QuickMerge Sort okazuje się wydajniejsze od klasycznego podejścia Quick Sort. Dla wartości ($\alpha \geq 9.0$) również wybór piwota jako losowy element tablicy, w połączeniu z algorytmem QuickMerge Sort osiąga lepsze rezultaty. Najwydajniejszą spośród badanych metod dla złożonych struktur danych jest wybór piwota jako pseudo-medianą z dziewięciu. Najmniej wydajnymi strategiami są mediana-median z pięciu oraz mediana-median z trzech, przy czym metody te są znacznie mniej wydajne niż klasyczny algorytm Quick Sort.



Rysunek 4.11

Do analizy histogramu liczby wykonanych operacji (4.11) użyto tablicy losowych danych o stałym rozmiarze $n = 1000$ oraz współczynnika kosztu o stałej wartości $\alpha = 1.0$. Można zauważyc, że sortując dane algorytmem QuickMerge Sort, skutecznie zmniejszono odchylenie standardowe liczby wykonanych operacji, niezależnie od metody wyboru piwota. Z badanych metod najmniejszą wartość oczekiwanaą liczyby wykonanych operacji ma klasyczny algorytm Quick Sort. Największą wartość oczekiwanaą otrzymano w przypadku sortowania algorytmem QuickMerge Sort ze strategią wyboru piwota jako mediana-median. W tym przypadku otrzymano również najmniejsze odchylenie standardowe spośród badanych deterministycznych algorytmów sortujących w miejscu.

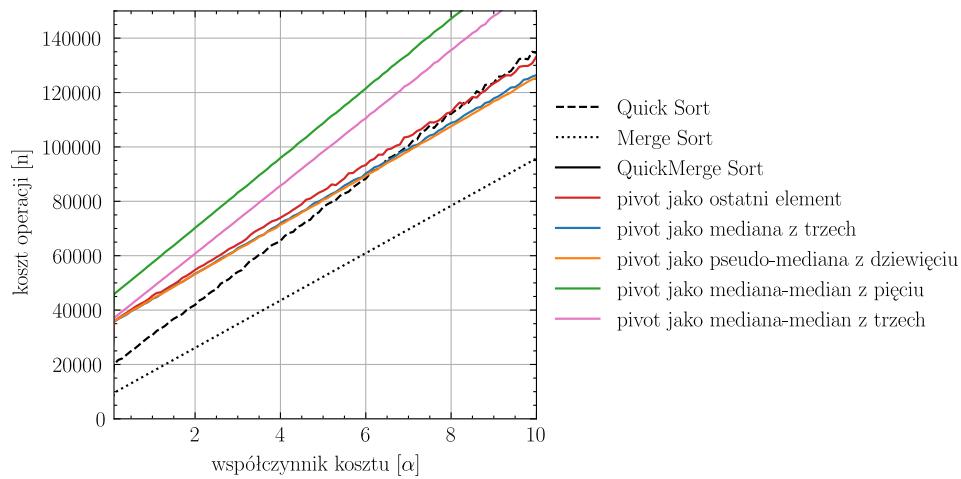
4.4.3 Analiza randomizowanych wersji algorytmu QuickMerge Sort

Tak jak w przypadku algorytmów deterministycznych, algorytmy randomizowane poddano analizie pod kątem liczby wykonywanych operacji atomowych, z podziałem na operacje porównania, zamiany miejsc oraz przypisania. Dodatkowo badano łączny koszt wykonanych operacji jako sumę ważoną liczby operacji atomowych, z uwzględnieniem wartości współczynnika kosztu. Badane strategie porównywano do algorytmu Merge Sort oraz randomizowanej wersji algorytmu Quick Sort z partycjonowaniem metodą Lomuto oraz wyborem piwota jako losowy element tablicy.

Badając liczbę wykonanych operacji porównania (4.14) można zauważyc, że każdy z randomizowanych algorytmów QuickMerge Sort wykonuje mniejszą liczbę porównań niż randomizowany algorytm Quick Sort. Najmniejszą liczbę operacji porównania spośród badanych algorytmów działających w miejscu osiąga QuickMerge Sort z partycjonowaniem metodą Lomuto oraz wyborem piwota jako pseudo-mediana z dziewięciu losowych elementów. Podobne, nieznacznie gorsze wyniki osiąga metoda wyboru piwota jako mediana z trzech losowych elementów. Najgorsze wyniki osiąga algorytm Quick Sort, z wyborem piwota jako losowy element tablicy.

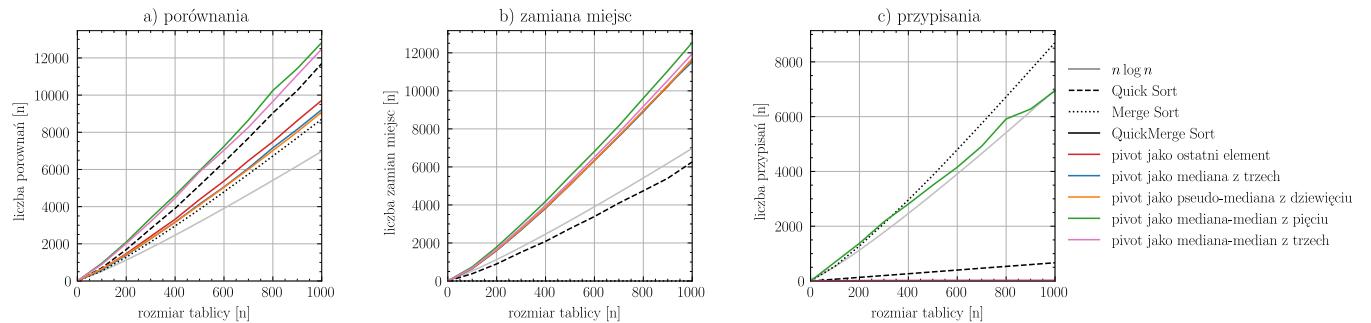
Analizując liczbę operacji zamiany miejsc oraz przypisania (4.14) można stwierdzić, że niezależnie od strategii wyboru piwota, algorytm QuickMerge Sort wykonuje około dwa razy więcej operacji zamiany miejsc niż randomizowany algorytm Quick Sort. Co więcej, żadna z randomizowanych implementacji nie wykonuje

Łączny koszt operacji wykonanych przez deterministyczne wersje algorytmu QuickMerge Sort w porównaniu do algorytmów Quick Sort oraz Merge Sort z podziałem na polityki wyboru pivota dla tablicy losowych danych rozmiaru $n = 1000$

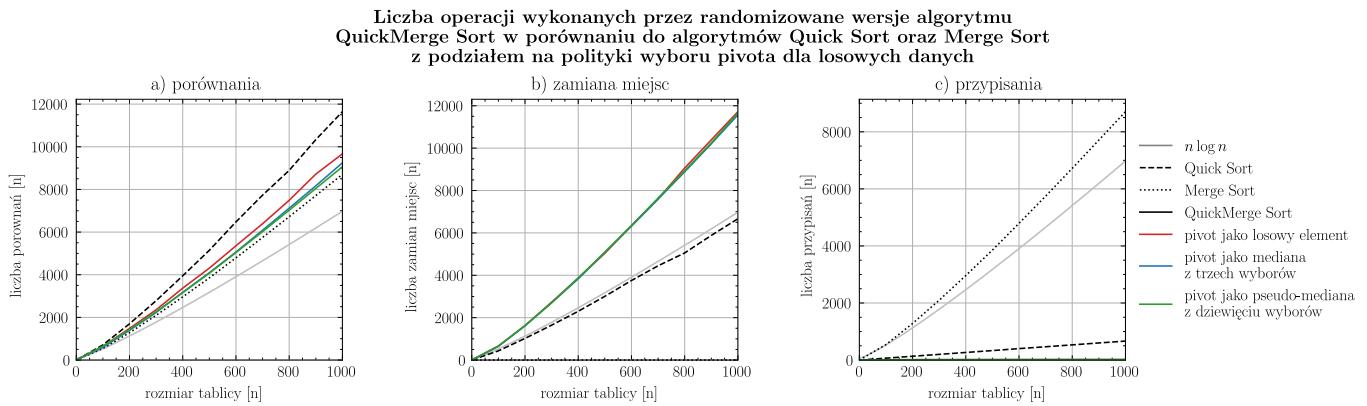


Rysunek 4.12

Liczba operacji wykonanych przez deterministyczne wersje algorytmu QuickMerge Sort w porównaniu do algorytmów Quick Sort oraz Merge Sort z podziałem na polityki wyboru pivota dla losowych danych



Rysunek 4.13



Rysunek 4.14

operacji przypisania.

Podobnie jak w przypadku algorytmów deterministycznych, badając łączny koszt wykonanych operacji (4.16) można zauważać, że dla danych typu podstawowego ($\alpha = 1.0$), najwydajniejszym algorytmem sortującym działającym w miejscu jest randomizowany algorytm Quick Sort. Sytuacja ta zmienia się w przypadku sortowania złożonych struktur wejściowych, dla których ($\alpha \geq 6.0$). Powyżej tej wartości, zastosowanie strategii wyboru piwota jako mediana z trzech losowych elementów lub pseudo-mediana z dziewięciu losowych elementów, w połączeniu z algorymem QuickMerge Sort okazuje się wydajniejsze od randomizowanego algorytmu Quick Sort. Dla wartości ($\alpha \geq 8.0$) również wybór piwota jako losowy element tablicy, w połączeniu z algorytem QuickMerge Sort osiąga lepsze rezultaty. Najwydajniejszą spośród badanych metod dla złożonych struktur danych jest wybór piwota jako pseudo-mediana z dziewięciu losowych elementów. Najmniej wydajnym jest randomizowany algorytm Quick Sort.

Do analizy histogramu liczby wykonanych operacji (4.15) użyto tablicy losowych danych o stałym rozmiarze $n = 1000$ oraz współczynnika kosztu o stałej wartości $\alpha = 1.0$. Można zauważać, że dla danych tupu podstawowego, średnia liczba wykonanych operacji jest zbliżona dla wszystkich randomizowanych algorytmów QuickMerge Sort, niezależnie od strategii wyboru piwota. W przypadku algorytmów QuickMerge Sort odchylenie standardowe jest znacznie mniejsze niż w przypadku randomizowanego algorytmu Quick Sort. Spośród badanych algorytmów działających w miejscu, najmniejszą wartość oczekiwanaą liczby wykonanych operacji ma algorytm Quick Sort.

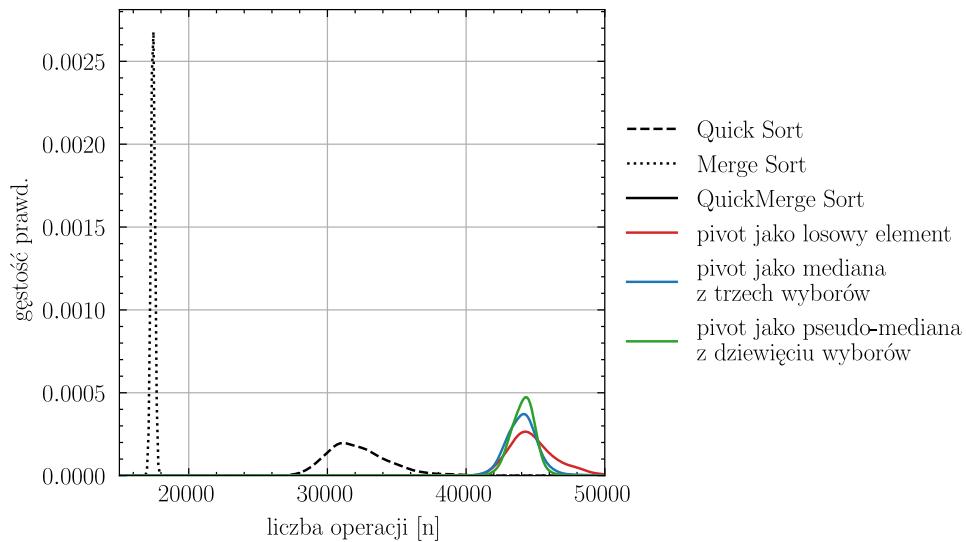
4.4.4 Wnioski

Stosowanie algorytmu QuickMerge Sort wiąże się z dodatkowym nakładem obliczeniowym, który wynika z konieczności zachowania elementów bufora w trakcie procesu scalania. Aby zapobiec nadpisywaniu elementów, zamiast klasycznego przypisania, wykorzystywana jest operacja zamiany miejsc, która jest ok. trzykrotnie kosztowniejsza od operacji przypisania. Z tego powodu, algorytm QuickMerge Sort wykonuje więcej operacji zamiany miejsc niż algorytm Quick Sort, jednak dzięki temu optymalizuje on liczbę wykonywanych operacji porównania oraz działa w miejscu, nie powodując dodatkowych kosztów związanych z alokacją oraz zwalnianiem pamięci.

Prowadzi to do wniosku, że algorytm ten przynosi realne korzyści podczas sortowania złożonych struktur danych, dla których koszt pojedynczej operacji porównania może być wielokrotnie większy od kosztu operacji przypisania. Eksperymentalnie wyznaczono, że dla struktur o współczynniku kosztu większym niż $\alpha > 6.0$, niektóre warianty algorytmu QuickMerge Sort są wydajniejsze od klasycznego podejścia Quick Sort.

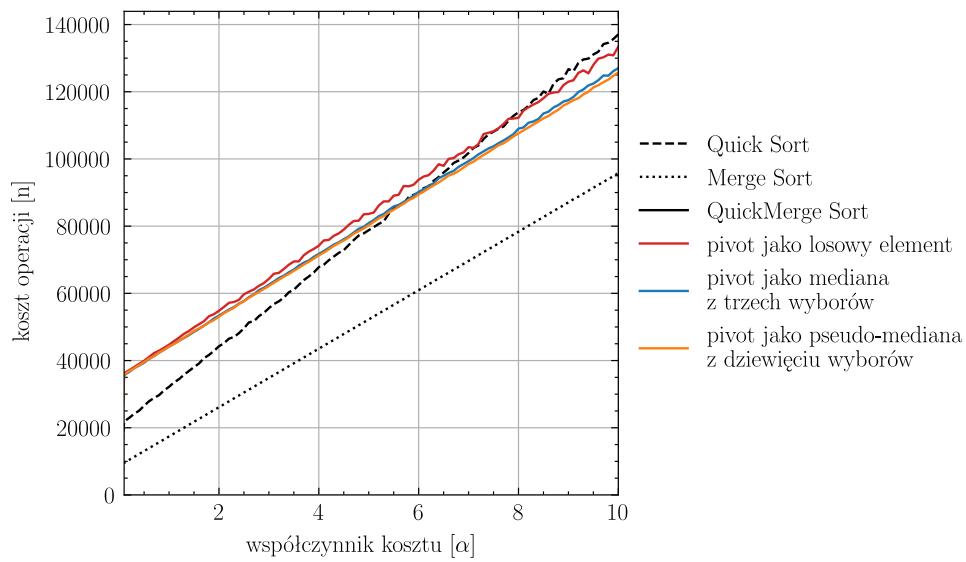
Przeprowadzona analiza sugeruje, że dla złożonych struktur danych, najwydajniejszą deterministyczną

**Histogram liczby operacji wykonanych
przez randomizowane wersje algorytmu QuickMerge Sort
w porównaniu do algorytmów Quick Sort oraz Merge Sort
dla tablicy losowych danych rozmiaru $n = 1000$**



Rysunek 4.15

**Łączny koszt operacji wykonanych przez randomizowane wersje
algorytmu QuickMerge Sort w porównaniu do algorytmów Quick Sort
oraz Merge Sort z podziałem na polityki wyboru pivota
dla tablicy losowych danych rozmiaru $n = 1000$**



Rysunek 4.16



strategią sortowania jest połączenie algorytmu QuickMerge Sort z partycjonowaniem metodą Lomuto oraz wyborem piwota jako pseudo-mediana z dziewięciu elementów. Pod względem liczby wykonywanych operacji, wariant ten jest asymptotycznie najbliższy klasycznemu podejściu Merge Sort, jednak dodatkowo działa on w miejscu. Spośród badanych algorytmów, najwydajniejszym randomizowanym wariantem okazało się połączenie algorytmu QuickMerge Sort z partycjonowaniem metodą Lomuto oraz wyborem piwota jako pseudo-mediana z девятию losowych elementów.

4.5 Intro Sort

Głównym problemem wynikającym ze stosowania algorytmu Quick Sort jest jego słaba pesymistyczna złożoność obliczeniowa. Dla uporządkowanych danych wejściowych, głębokość drzewa wywołań rekurencyjnych jest równa rozmiarowi tablicy wejściowej, zaś sam algorytm działa w czasie równym $O(n^2)$. Dodatkowo, dla tablicy małych rozmiarów, algorytm Quick Sort powoduje niepotrzebny nakład czasowy, związany ze wstępny przygotowaniem kolekcji. Obydwa te problemy rozwiązano stosując algorytm Intro Sort, którego analizę przedstawiono w bieżącym rozdziale.

Intro Sort, czyli sortowanie introspektywne, to algorytm hybrydowy będący połączeniem algorytmów Quick Sort, Insertion Sort oraz Heap Sort. Działanie algorytmu dostosowuje się w zależności od rozmiaru sortowanej tablicy oraz głębokości drzewa wywołań rekurencyjnych. Jeśli rozmiar tablicy jest mniejszy od punktu granicznego, kolekcja jest sortowana metodą Insertion Sort, wyznaczoną eksperymentalnie. Jeśli przekroczone maksymalną głębokość drzewa wywołań rekurencyjnych, łańcuch wywołań zostaje zakończony, zaś tablica zostaje posortowana metodą Heap Sort. Początkowa maksymalna głębokość drzewa wywołań rekurencyjnych jest wyznaczona wzorem $2 \log n$. Ponieważ metoda Heap Sort działa z czasem $O(n \log n)$ nawet w przypadku pesymistycznym, algorytm Intro Sort ma złożoność $O(n \log n)$, niezależnie od porządku danych wejściowych.

Z powodu swojej wydajności, algorytm Intro Sort jest niezwykle popularny we współczesnych systemach komputerowych. Jego implementacja jest wykorzystywana między innymi w bibliotece standardowej libstdc jako funkcja sortująca `std::sort`¹.

4.5.1 Eksperimentalne wyznaczanie punktu granicznego

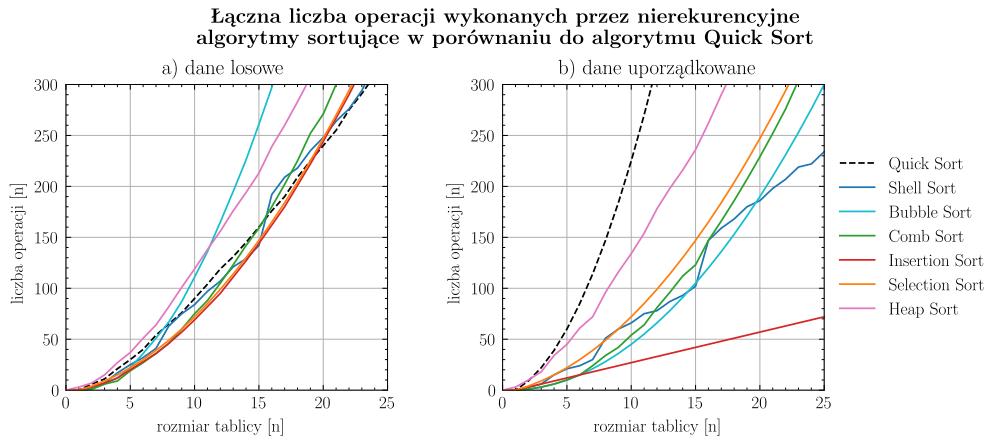
Stosowanie rekurencyjnych metod sortujących, wykorzystujących technikę dziel i zwycięzaj, wiąże się z dodatkowym nakładem czasowym spowodowanym koniecznością odpowiedniego przygotowania tablicy przed jej posortowaniem. W przypadku algorytmu Quick Sort, dodatkowy nakład powoduje metoda partycjonowania. Z tego powodu, dla tablic o ograniczonym rozmiarze, bardziej wydajne okazuje się stosowanie elementarnych algorytmów sortujących.

Analizując liczbę wykonanych operacji dla elementarnych algorytmów sortujących (4.17) można zauważać, że dla tablicy o rozmiarze nieprzekraczającym 19 elementów, najbardziej wydajnym algorytmem jest Insertion Sort. Co więcej algorytm ten działa liniowo w przypadku uporządkowanych danych wejściowych. Z tego powodu algorytm Insertion Sort został zastosowany jako algorytm sortowania kolekcji o rozmiarze poniżej punktu granicznego. Punktem granicznym jest tablica rozmiaru 19 elementów.

4.5.2 Pseudokod

Algorytm Intro Sort można podzielić na trzy rozłączne etapy. Wybór etapu uzależniony jest od wielkości tablicy wejściowej oraz głębokości drzewa wywołań rekurencyjnych. Dla tablicy o rozmiarze nieprzekraczającym punktu granicznego, dane są sortowane algorytmem Insertion Sort (linia 3). Dla tablicy rozmiaru powyżej punktu granicznego, w przypadku gdy przekroczone maksymalną głębokość drzewa wywołań rekurencyjnych, stosowany jest algorytm Heap Sort (linia 6). W pozostałych sytuacjach stosowane jest klasyczne podejście z algorytmu Quick Sort (linia 9), przy czym każdemu rekurencyjnemu wywołaniu algorytmu towarzyszy dekrementacja głębokości drzewa.

¹Dokumentacja funkcji sortującej std::sort: <https://gcc.gnu.org/onlinedocs/libstdc++/libstdc++-html-USERS-4.4/a01027.html>



Rysunek 4.17

Algorytm 2 Intro Sort

```

1: procedure INTROSORT(ARR, DEPTH)
2:
3:   if len(arr) ≤ maxLength then                                ▷ sortowanie przez wstawianie
4:     InsertionSort(arr)
5:
6:   else if depth = 0 then                                     ▷ sortowanie przez kopcowanie
7:     HeapSort(arr)
8:
9:   else                                                       ▷ szybkie sortowanie
10:    arr1, arr2 ← Partition(arr)
11:    IntroSort(arr1, depth-1)
12:    IntroSort(arr2, depth-1)
13:  end if
14:
15: end procedure

```

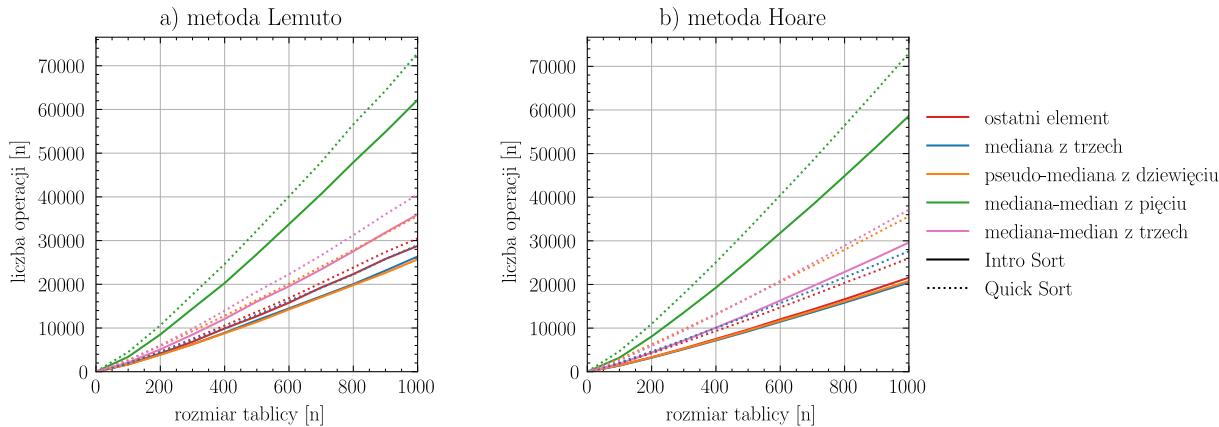
4.5.3 Analiza deterministycznych wersji algorytmu Intro Sort

Analizując wydajność deterministycznych algorytmów z rodziny Intro Sort, badano liczbę wykonywanych operacji atomowych z podziałem na operacje porównania, zamiany miejsc oraz przypisania. Badając łączną liczbę wykonanych operacji założono stałą wartość współczynnika kosztu $\alpha = 1.0$.

Dla losowych danych wejściowych (4.18), sortowanie metodą Intro Sort jest wydajniejsze od algorytmu Quick Sort, niezależnie od metody partycjonowania oraz strategii wyboru piwota. Najlepsze wyniki otrzymano, łącząc algorytm Intro Sort z partycjonowaniem metodą Hoare oraz wyborem piwota jako mediana z trzech elementów. W przypadku partycjonowania metodą Lomuto, najlepsze wyniki otrzymano wybierając piwot jako pseudo-mediana z dziewięciu elementów oraz sortując algorymem Intro Sort. Najmniej wydajną spośród badanych strategii jest wybór piwota jako mediana-median z pięciu elementów, przy czym najgorsze wyniki otrzymano niezależnie od wyboru metody partycjonowania.

Dla uporządkowanych danych wejściowych (4.19), sortowanie metodą Intro Sort również jest wydajniejsze od algorytmu Quick Sort, niezależnie od metody partycjonowania oraz strategii wyboru piwota. Najwydajniejszym algorytmem sortującym dla tego typu danych wejściowych jest Intro Sort z metodą partycjonowania Hoare oraz wyborem piwota jako mediana z trzech elementów. Bardzo podobne wyniki otrzymano dla strategii wyboru piwota jako pseudo-mediana z dziewięciu elementów. W przypadku partycjonowania

Łączna liczba operacji wykonanych przez deterministyczne wersje algorytmu Intro Sort w porównaniu do algorytmu Quick Sort z podziałem na polityki wyboru pivota dla tablicy losowych danych



Rysunek 4.18

metodą Lomuto, najwydajniejszą strategią wyboru piwota jest pseudo-mediana z dziewięciu. Dla badanych algorytmów z rodziny Intro Sort, maksymalna złożoność obliczeniowa nie przekracza wartości $O(n \log n)$, niezależnie od metody partycjonowania oraz strategii wyboru piwota. Jest to znacząca optymalizacja w stosunku do algorytmu Quick Sort, dla którego pesymistyczna złożoność obliczeniowa wynosi $O(n^2)$.

Analizując liczbę wykonanych operacji porównania (4.21) dla losowych danych wejściowych można zauważać, że dla takich samych strategii wyboru piwota, metoda Lomuto osiąga nieznacznie lepsze rezultaty niż partycjonowanie metodą Hoare. Najmniejszą liczbę operacji porównania otrzymano, partycjonując dane metodą Lomuto przy wyborze piwota jako pseudo-mediana z dziewięciu lub mediana z trzech elementów. Badając liczbę operacji zamiany miejsc można stwierdzić, że partycjonowanie metodą Hoare jest znacznie wydajniejsze niż partycjonowanie metodą Lomuto. Najmniejszą liczbę operacji zamiany miejsc otrzymano poprzez partycjonowanie metodą Hoare, przy wyborze piwota jako mediana-median z trzech.

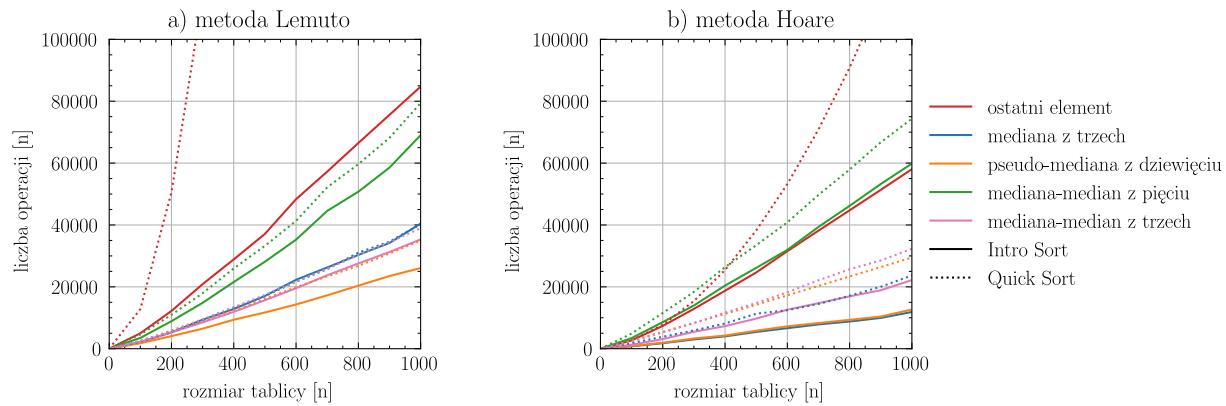
Do analizy histogramu liczby wykonanych operacji (4.20) użyto tablicy losowych danych o stałym rozmiarze $n = 1000$. Dla badanych algorytmów z rodziny Intro Sort, najmniejszą średnią liczbę wykonanych operacji otrzymano dla metody partycjonowania Hoare, przy wyborze piwota jako mediana z trzech oraz pseudo-mediana z dziewięciu elementów, jednak w przypadku pseudo-mediany z dziewięciu otrzymano znacznie mniejsze odchylenie standardowe. Dodatkowo, dla metody tej otrzymano najmniejsze odchylenie standardowe spośród wszystkich badanych strategii, więc jest to zalecana metoda wyboru piwota podczas sortowania algorytmem Intro Sort.

4.5.4 Analiza randomizowanych wersji algorytmu Intro Sort

Tak jak w poprzednim rozdziale, algorytmy były analizowane pod kątem liczby wykonywanych operacji atomowych, z podziałem na operacje porównania, zamiany miejsc oraz przypisania. Badając łączną liczbę wykonanych operacji założono stałą wartość współczynnika kosztu $\alpha = 1.0$.

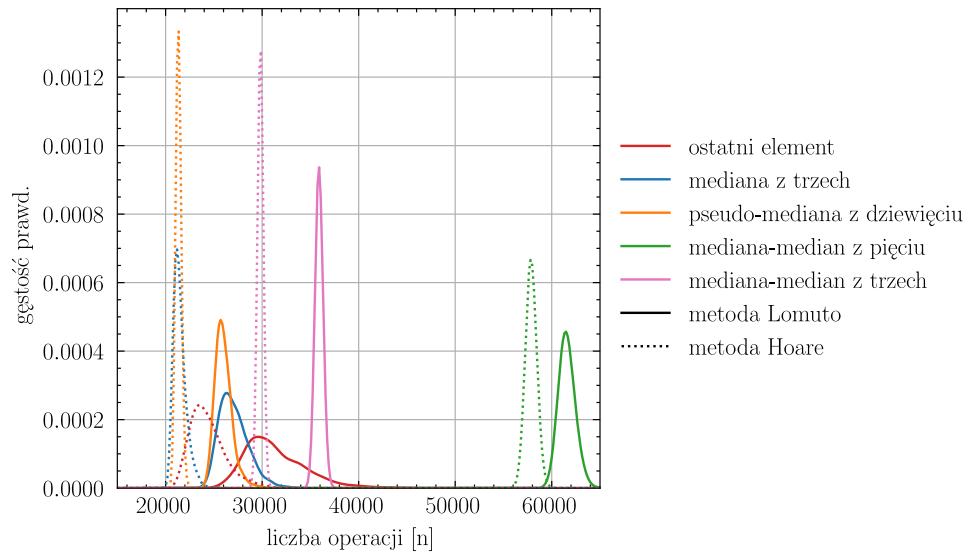
Dla losowych danych wejściowych (4.23), sortowanie metodą Intro Sort jest wydajniejsze od randomizowanego odpowiednika Quick Sort dla każdej badanej metody partycjonowania oraz strategii wyboru piwota. Najlepsze wyniki otrzymano, łącząc algorytm Intro Sort z partycjonowaniem metodą Hoare oraz wyborem piwota jako mediana z trzech losowych elementów lub pseudo-mediana z dziewięciu losowych elementów. Stosując algorytm Intro Sort, strategia wyboru piwota jako pseudo-mediana z dziewięciu losowych elementów jest również najlepsza przy partycjonowaniu metodą Lomuto. Co ciekawe, dla randomizowanych algorytmów

Łączna liczba operacji wykonanych przez deterministyczne wersje algorytmu Intro Sort w porównaniu do algorytmu Quick Sort z podziałem na polityki wyboru pivota dla tablicy danych posortowanych odwrotnie

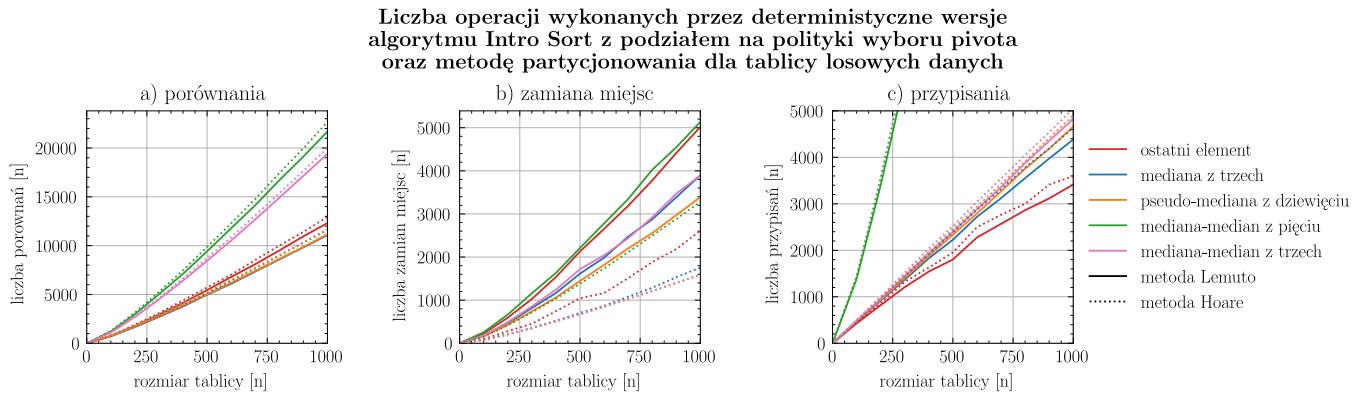


Rysunek 4.19

Histogram liczby operacji wykonanych przez deterministyczne wersje algorytmu Intro Sort dla tablicy losowych danych rozmiaru $n = 1000$



Rysunek 4.20



Rysunek 4.21

Quick Sort, strategia ta osiąga najgorsze wyniki spośród badanych algorytmów.

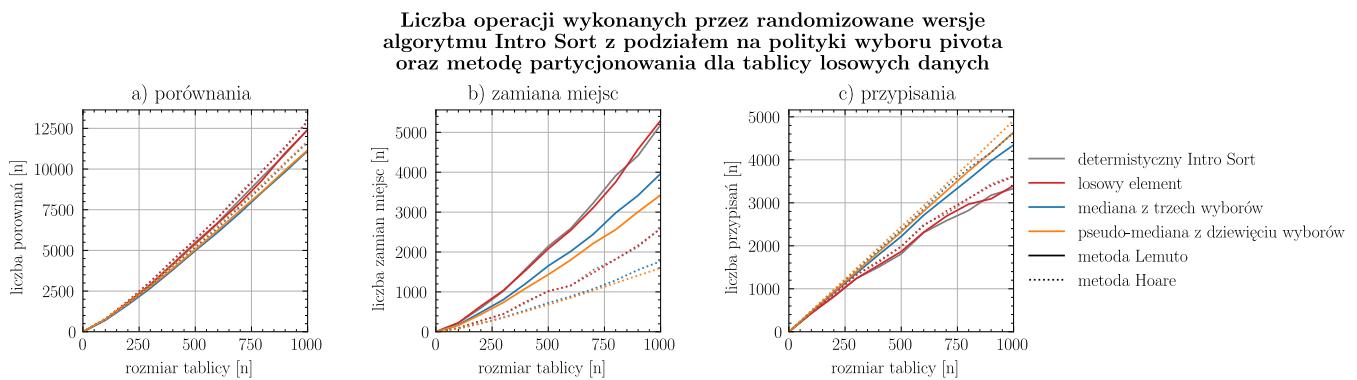
Dla uporządkowanych danych wejściowych (4.24), podobnie jak w przypadku danych losowych, sortowanie metodą Intro Sort jest wydajniejsze od randomizowanego odpowiednika Quick Sort, niezależnie od metody partycjonowania oraz strategii wyboru piwota. Dla obydwu metod partycjonowania, najlepsze wyniki otrzymano wybierając piwot jako pseudo-mediana z dziewięciu losowych elementów, przy czym stosując partycjonowanie metodą Hoare łączna liczba wykonanych operacji jest mniejsza niż w przypadku partycjonowania metodą Lomuto. Co ciekawe, strategia ta nie sprawdza się w przypadku randomizowanych algorytmów Quick Sort, dla których otrzymano najgorsze wyniki spośród badanych algorytmów.

Analizując liczbę wykonanych operacji porównania (4.22) dla losowych danych wejściowych można zauważać, że dla takich samych strategii wyboru piwota, metoda Lomuto osiąga nieznacznie lepsze rezultaty niż partycjonowanie metodą Hoare. Najmniejszą liczbę operacji porównania otrzymano, partycjonując dane metodą Lomuto przy wyborze piwota jako pseudo-mediana z dziewięciu lub mediana z trzech losowych elementów. Najlepsze strategie wyboru piwota podczas minimalizacji liczby porównań okazują się również najlepszymi podczas minimalizacji liczby wykonywanych operacji zamiany miejsc. Badając liczbę operacji zamiany miejsc można stwierdzić, że partycjonowanie metodą Hoare jest znacznie wydajniejsze niż partycjonowanie metodą Lomuto.

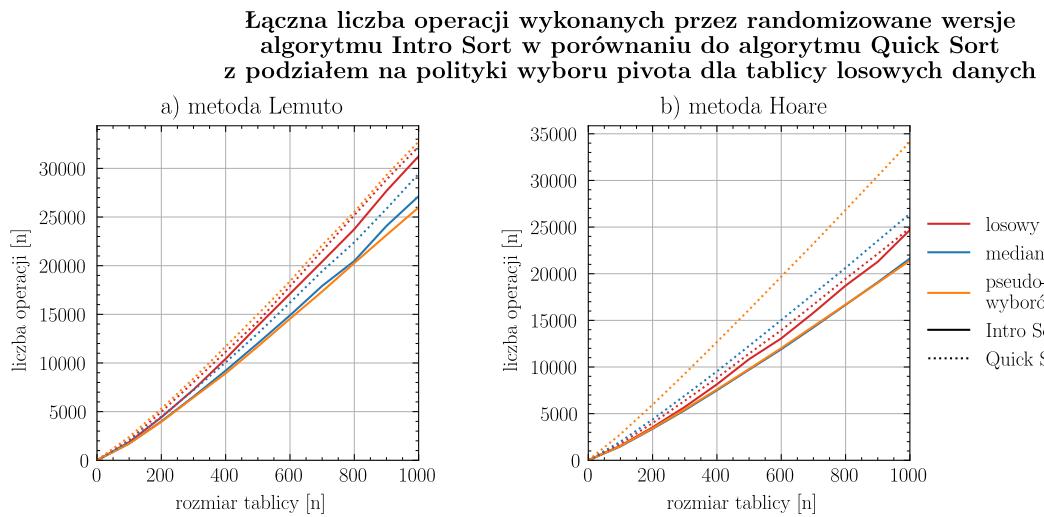
Do analizy histogramu liczby wykonanych operacji (4.25) użyto tablicy losowych danych o stałym rozmiarze $n = 1000$. Dla badanych algorytmów z rodziny Intro Sort, najmniejszą średnią liczbę wykonanych operacji otrzymano dla metody partycjonowania Hoare, przy wyborze piwota jako pseudo-mediana z dziewięciu oraz mediana z trzech losowych elementów, jednak w przypadku pseudo-mediany z dziewięciu otrzymano znacznie mniejsze odchylenie standardowe. Fakt ten jest analogiczny w przypadku deterministycznych odpowiedników algorytmu Intro Sort. Dodatkowo, wybierając piwot jako pseudo-mediana z девицием losowych elementów, otrzymano najmniejsze odchylenie standardowe spośród wszystkich randomizowanych strategii wyboru piwota.

4.5.5 Wnioski

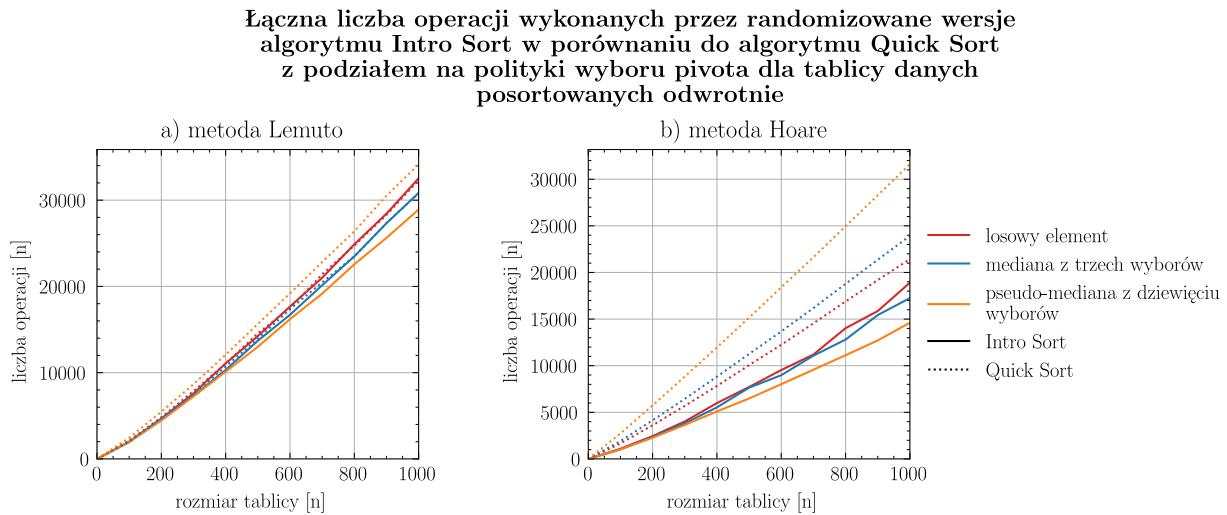
TODO wnioski



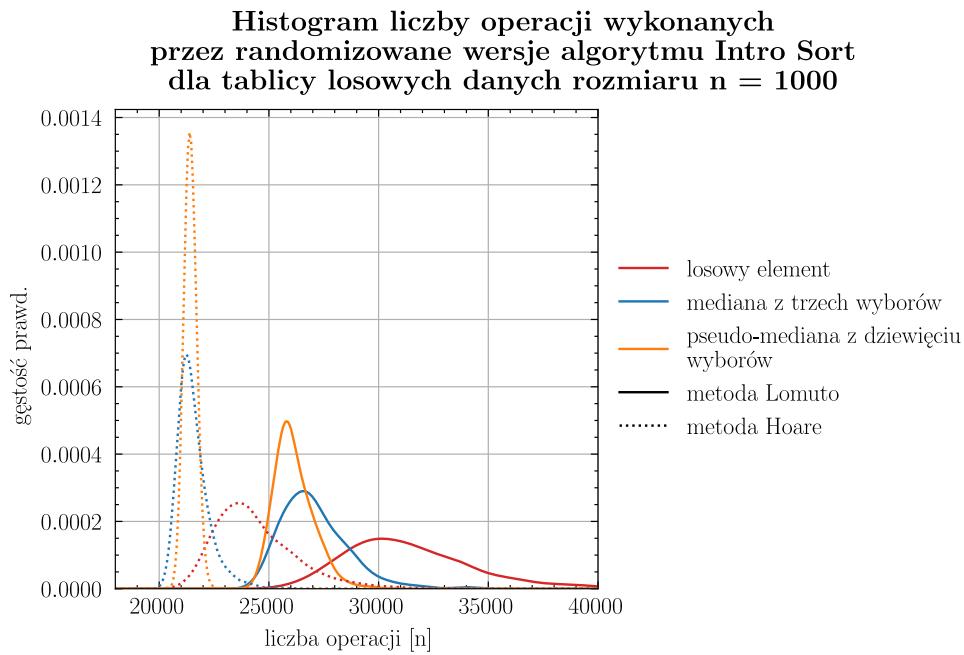
Rysunek 4.22



Rysunek 4.23



Rysunek 4.24



Rysunek 4.25

4.6 QuickMergeIntro Sort

W poprzednich rozdziałach omówione zostały algorytmy Intro Sort oraz QuickMerge Sort. Algorytm Intro Sort rozwiązuje problem pesymistycznej złożoności obliczeniowej obecny w klasycznym algorytmie Quick Sort. Z kolei algorytm QuickMerge Sort optymalizuje liczbę wykonywanych operacji porównania, przez co jest niezwykle wydajny podczas sortowania złożonych typów danych. W tym rozdziale podjęto próbę połączenia tych dwóch technik, tworząc algorytm QuickMergeIntro Sort.

Algorytm QuickMergeIntro Sort, tak jak algorytm Intro Sort, zmienia swoje działanie w zależności od czynników zewnętrznych. Jeśli rozmiar tablicy jest mniejszy od punktu granicznego, tablica jest sortowana metodą Insertion Sort. Jeśli przekroczono maksymalną głębokość drzewa wywołań rekurencyjnych, tablica jest sortowana algorytmem Heap Sort. W przeciwnym wypadku, jeśli żaden w powyższych warunków nie zachodzi, stosowane jest podejście znane z algorytmu QuickMerge Sort. Algorytm partycjonujący dzieli tablicę wejściową na dwie części, z których mniejsza jest sortowana metodą Merge Sort, zaś pozostała jest sortowana rekurencyjnie algorytmem QuickMergeIntro Sort.

Ponieważ stosowanie algorytmu QuickMerge Sort przynosi realne korzyści podczas sortowania złożonych struktur danych, w testach założono, że współczynnik kosztu ma wartość $\alpha = 10.0$. Przeprowadzane wcześniej testy sugerują, że dla złożonych struktur danych partycjonowanie metodą Lomuto osiąga lepsze rezultaty niż partycjonowanie metodą Hoare. Z tego powodu w analizie uwzględniono tylko pierwszą z przedstawionych metod. Dodatkowo w analizie zrezygnowano ze strategii wyboru piwota jako mediana-median, które dla poprzednich testów okazały się niewydajne, powodując dodatkowy nakład obliczeniowy.

4.6.1 Pseudokod

Podobnie jak w przypadku algorytmu Intro Sort, algorytm QuickMergeIntro Sort można podzielić na trzy rozłączne etapy. Wybór etapu uzależniony jest od wielkości tablicy wejściowej oraz głębokości drzewa wywołań rekurencyjnych. Dla tablicy o rozmiarze nieprzekraczającym punktu granicznego, dane są sortowane algorytmem Insertion Sort (linia 3). Dla tablicy rozmiaru powyżej punktu granicznego, w przypadku gdy

przekroczone maksymalną głębokość drzewa wywołań rekurencyjnych, stosowany jest algorytm Heap Sort (linia 6). Jeśli żaden z powyższych przypadków nie zachodzi, stosowane jest podejście znane z algorytmu QuickMerge Sort (linia 9). Tablica wejściowa jest partycjonowana, zaś krótsza z partycji jest sortowana metodą Merge Sort. Tak jak w przypadku algorytmu QuickMerge Sort, bufor służący do scalania tablicy wynikowej stanowi druga partycja. Na koniec druga partycja zostaje posortowana algorytmem QuickMergeIntro Sort, przy czym każdym rekurencyjnemu wywołaniu towarzyszy dekrementacja głębokości drzewa.

Algorytm 3 QuickMergeIntro Sort

```
1: procedure QUICKMERGEINTROSORT(ARR, DEPTH)
2:
3:   if len(arr) ≤ maxLength then                                ▷ sortowanie przez wstawianie
4:     InsertionSort(arr)
5:
6:   else if depth = 0 then                                     ▷ sortowanie przez kopcowanie
7:     HeapSort(arr)
8:
9:   else
10:    arr1, arr2 ← Partition(arr)                                ▷ partycjonowanie
11:
12:    if len(arr1) < len(arr2) then                            ▷ sortowanie
13:      buffer ← arr2
14:      MergeSortBySwaps(arr1, buffer)
15:      QuickMergeIntroSort(arr2, depth-1)
16:    else
17:      buffer ← arr1
18:      MergeSortBySwaps(arr2, buffer)
19:      QuickMergeIntroSort(arr1, depth-1)
20:    end if
21:  end if
22:
23: end procedure
```

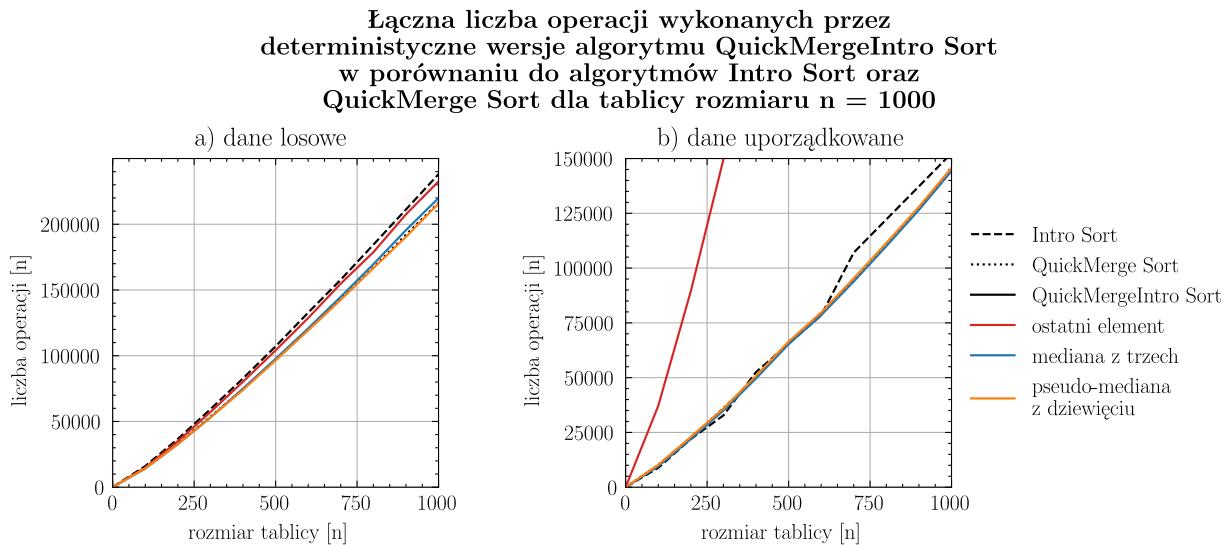
4.6.2 Analiza deterministycznych wersji algorytmu QuickMergeIntro Sort

Analizując wydajność deterministycznych algorytmów z rodziny QuickMergeIntro Sort, badano liczbę wykonywanych operacji atomowych z podziałem na operacje porównania, zamiany miejsc oraz przypisania. W testach przyjęto stałą wartość współczynnika kosztu równą $\alpha = 10.0$. Dodatkowo we wszystkich testach zastosowano partycjonowanie metodą Lomuto.

Badając liczbę wykonanych operacji dla losowych danych wejściowych (4.26) można zauważać, że połączenie algorytmu QuickMergeIntro Sort z wyborem piwota jako mediana z trzech lub pseudo-mediana z dziewięciu elementów, osiąga podobne rezultaty co zastosowanie algorytmu QuickMerge Sort. Dodatkowo, algorytm QuickMergeIntro Sort dla złożonych struktur danych jest wydajniejszy od algorytmu Intro Sort.

Analizując liczbę wykonanych operacji dla uporządkowanych danych wejściowych (4.26), podobnie jak w przypadku danych losowych, można zauważać, że połączenie algorytmu QuickMergeIntro Sort z wyborem piwota jako mediana z trzech lub pseudo-mediana z девятиu elementów, prowadzi do zbliżonych rezultatów co zastosowanie algorytmu QuickMerge Sort. W tym przypadku zastosowanie strategii wyboru piwota jako ostatni element tablicy powoduje drastyczny wzrost złożoności obliczeniowej.

Badając średnią liczbę wykonanych operacji dla losowych danych wejściowych (4.27), porównywano różne wersje algorytmu QuickMergeIntro Sort z algorytmami QuickMerge Sort oraz Intro Sort. W przypadku dwóch ostatnich, zastosowano wybór piwota jako pseudo-mediana z девятиu elementów. Analiza eksperymentalna



Rysunek 4.26

sugeruje, że dla złożonych danych wejściowych, połączenie algorytmu QuickMergeIntro Sort z partycjonowaniem metodą Lomuto oraz wyborem piwota jako pseudo-mediana z dziewięciu elementów jest nieznacznie wydajniejsze od algorytmu QuickMerge Sort oraz sporo wydajniejsze od swojego odpowiednika Intro Sort.

Analizując średnią liczbę wykonanych operacji dla uporządkowanych danych wejściowych (4.27), również porównywano wariancję algorytmu QuickMergeIntro Sort z algorytmami QuickMerge Sort oraz Intro Sort. Dla algorytmów QuickMerge Sort oraz Intro Sort zastosowano wybór piwota jako mediana z trzech elementów. Analiza eksperymentalna sugeruje, że podczas sortowania uporządkowanych danych wejściowych, połączenie algorytmu QuickMergeIntro Sort z partycjonowaniem metodą Lomuto oraz wyborem piwota jako mediana z trzech elementów jest wydajniejsze od algorytmów QuickMerge Sort oraz Intro Sort.

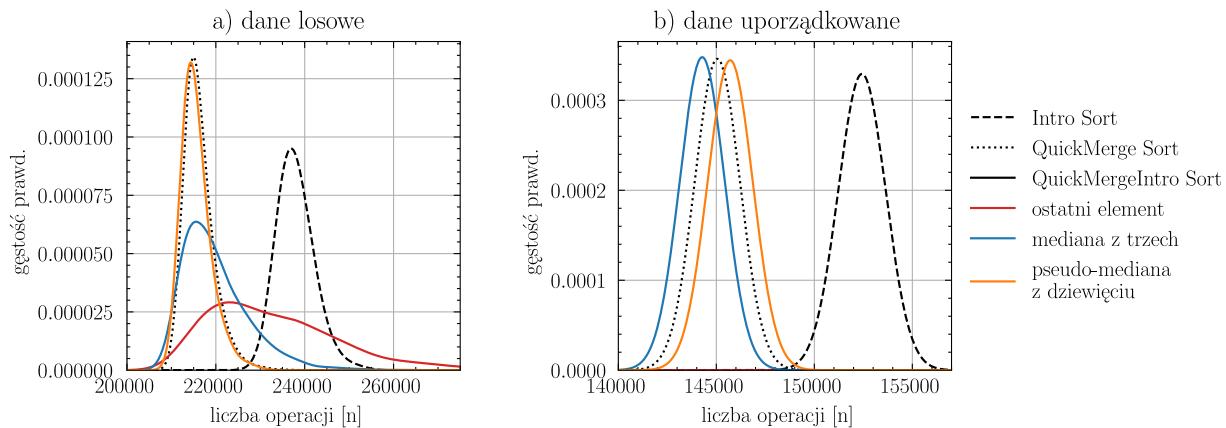
Badając średnią liczbę wykonanych operacji z rozbiciem na poszczególne operacje (4.28), pominięto operacje przypisania, które w większości badanych przypadków były bliskie zeru. Można zauważać, że połączenie algorytmu QuickMergeIntro Sort z partycjonowaniem metodą Lomuto oraz wyborem piwota jako pseudo-mediana z dziewięciu elementów, wykonuje nieznacznie mniejszą liczbę operacji porównania od swojego odpowiednika QuickMerge Sort. Wszystkie badane wersje algorytmu QuickMergeIntro Sort wykonywały zbliżoną liczbę operacji zamiany miejsc. Najmniejsze odchylenie standardowe zaobserwowano dla algorytmu QuickMerge Sort.

4.6.3 Analiza randomizowanych wersji algorytmu QuickMergeIntro Sort

Analizując wydajność randomizowanych algorytmów z rodziny QuickMergeIntro Sort, badano liczbę wykonywanych operacji atomowych z podziałem na operacje porównania, zamiany miejsc oraz przypisania. W testach przyjęto stałą wartość współczynnika kosztu równą $\alpha = 10.0$. Dodatkowo we wszystkich testach zastosowano partycjonowanie metodą Lomuto.

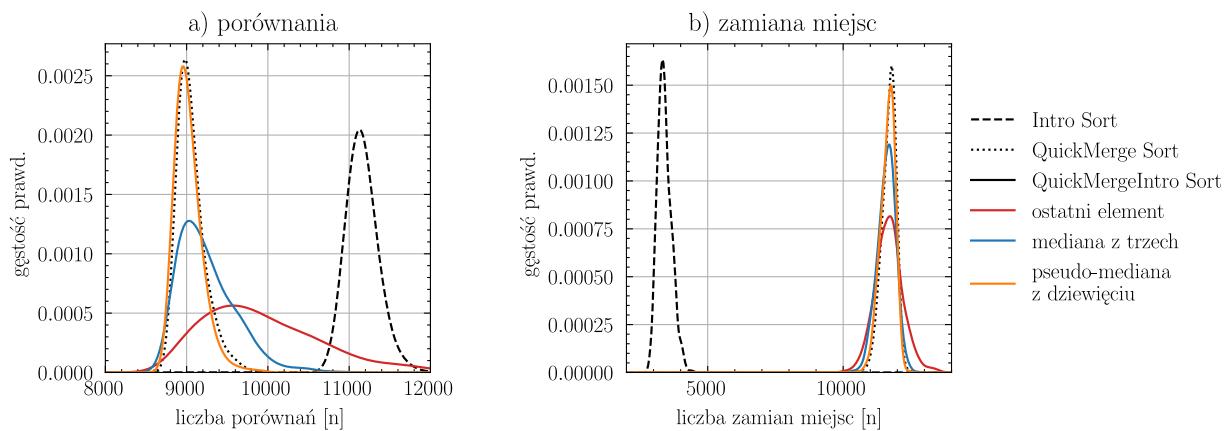
Badając liczbę wykonanych operacji dla losowych danych wejściowych (4.29) można zauważać, podobnie jak w przypadku algorytmów deterministycznych, że połączenie algorytmu QuickMergeIntro Sort z wyborem piwota jako mediana z trzech lub pseudo-mediana z девятиu losowych elementów, osiąga podobne rezultaty co odpowiednik QuickMerge Sort. W tym przypadku, wybór piwota jako losowy element tablicy okazał się mniej wydajną strategią. Dodatkowo, dla złożonych typów danych, każda z badanych wariacji algorytmu QuickMergeIntro Sort osiąga lepsze rezultaty od algorytmu Intro Sort.

Histogram liczby operacji wykonanych przez deterministyczne wersje algorytmu QuickMergeIntro Sort w porównaniu do algorytmów Intro Sort oraz QuickMerge Sort dla tablicy rozmiaru $n = 1000$



Rysunek 4.27

Histogram liczby operacji wykonanych przez deterministyczne wersje algorytmu QuickMergeIntro Sort w porównaniu do algorytmów Intro Sort oraz QuickMerge Sort dla tablicy uporządkowanych danych rozmiaru $n = 1000$



Rysunek 4.28

Analizując liczbę wykonanych operacji dla uporządkowanych danych wejściowych (4.29), można zauważyc, że połączenie algorytmu QuickMergeIntro Sort z wyborem piwota jako pseudo-mediana z dziewięciu losowych elementów, osiąga zbliżonych rezultaty co algorytm QuickMerge Sort. Co ciekawe, w przypadku algorytmów randomizowanych, zastosowanie strategii wyboru piwota jako mediana z trzech losowych elementów okazało się mniej wydajne.

Badając średnią liczbę wykonanych operacji dla losowych danych wejściowych (4.30), porównywano różne wersje algorytmu QuickMergeIntro Sort z algorytmami QuickMerge Sort oraz Intro Sort. W przypadku dwóch ostatnich, zastosowano wybór piwota jako pseudo-mediana z dziewięciu losowych elementów. Analiza eksperymentalna sugeruje, że dla złożonych danych wejściowych, połączenie algorytmu QuickMergeIntro Sort z partycjonowaniem metodą Lomuto oraz wyborem piwota jako pseudo-mediana z dziewięciu losowych elementów jest nieznacznie wydajniejsze od randomizowanego algorytmu QuickMerge Sort oraz sporo wydajniejsze od swojego odpowiednika Intro Sort. Jednocześnie, przedstawione podejście z wykorzystaniem algorytmu QuickMergeIntro Sort jest najwydajniejszym badanym sposobem na sortowanie złożonych struktur losowych danych.

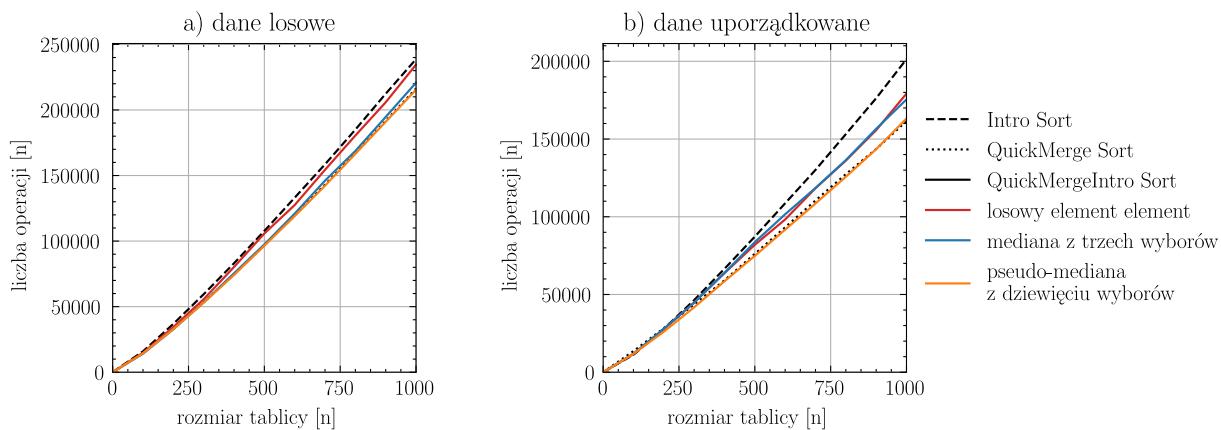
Analizując średnią liczbę wykonanych operacji dla uporządkowanych danych wejściowych (4.30), również porównywano wariancie algorytmu QuickMergeIntro Sort z randomizowanymi algorytmami QuickMerge Sort oraz Intro Sort. Dla algorytmów QuickMerge Sort oraz Intro Sort zastosowano wybór piwota jako pseudo-mediana z dziewięciu losowych elementów. Analiza eksperymentalna sugeruje, że podczas sortowania uporządkowanych danych wejściowych, połączenie algorytmu QuickMergeIntro Sort z partycjonowaniem metodą Lomuto oraz wyborem piwota jako pseudo-mediana z dziewięciu losowych elementów jest wydajniejsze od algorytmów QuickMerge Sort oraz Intro Sort. Korzystając z algorytmu QuickMerge Sort otrzymano nieznacznie mniejsze odchylenie standardowe.

Badając średnią liczbę wykonanych operacji z rozbiciem na poszczególne operacje (4.31), pominięto operacje przypisania, które w większości badanych przypadków były bliskie zeru. Można zauważyc, że połączenie algorytmu QuickMergeIntro Sort z partycjonowaniem metodą Lomuto oraz wyborem piwota jako pseudo-mediana z dziewięciu elementów, wykonuje nieznacznie mniejszą liczbę operacji porównania od swojego odpowiednika QuickMerge Sort. Co ciekawe, stosując tę strategię otrzymano najmniejsze odchylenie standardowe liczby wykonywanych operacji spośród badanych algorytmów z rodziny QuickMergeIntro Sort. Analizując liczbę operacji zamiany miejsc, najwydajniejsza badana wersja algorytmu QuickMergeIntro Sort osiągnęła zbliżoną wydajność co odpowiednik QuickMerge Sort.

4.6.4 Wnioski

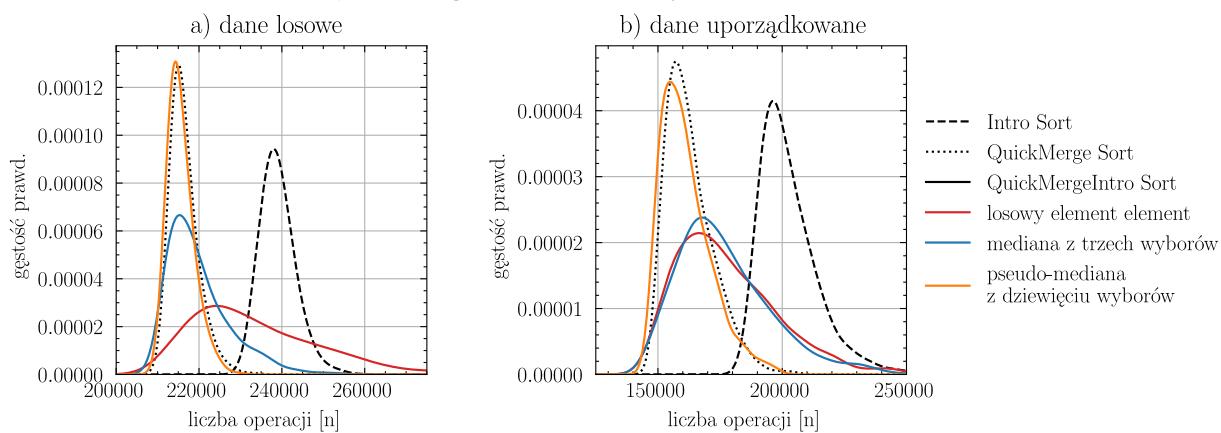
- TODO QuickMergeIntro Sort najwydajniejsze dla deterministycznych (Strategia ta jest najwydajniejsza spośród badanych algorytmów.) - TODO bardzo podobny do QuickMerge Sort, gorszy dla losowych danych, lepszy dla uporzadkowanych danych, dlatego bedzie preferowanym algorytmem dzialajacym w miejscu

**Łączna liczba operacji wykonanych przez
randomizowane wersje algorytmu QuickMergeIntro Sort
w porównaniu do algorytmów Intro Sort oraz
QuickMerge Sort dla tablicy rozmiaru n = 1000**



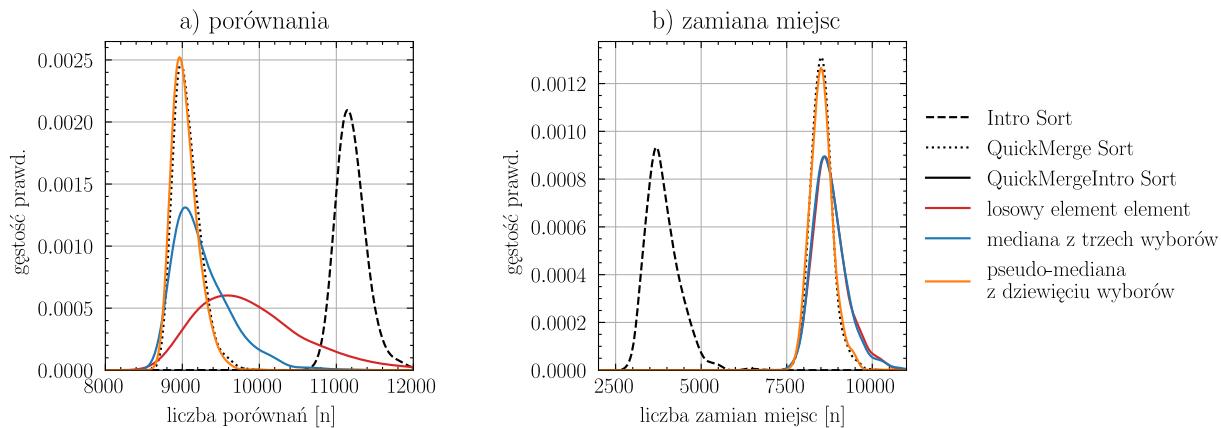
Rysunek 4.29

**Histogram liczby operacji wykonanych przez
randomizowane wersje algorytmu QuickMergeIntro Sort
w porównaniu do algorytmów Intro Sort oraz
QuickMerge Sort dla tablicy rozmiaru n = 1000**



Rysunek 4.30

**Histogram liczby operacji wykonanych przez
randomizowane wersje algorytmu QuickMergeIntro Sort
w porównaniu do algorytmów Intro Sort oraz QuickMerge Sort
dla tablicy uporządkowanych danych rozmiaru $n = 1000$**



Rysunek 4.31

Podsumowanie

Badania były przeprowadzane z podziałem na 4 kategorie. W każdej kategorii został wyłoniony najbardziej optymalny z badanych algorytmów, czyli algorytm wykonujący najmniejszą liczbę operacji, z uwzględnieniem współczynnika kosztu. Podsumowanie najlepszych algorytmów działających w miejscu Podsumowanie wyników testowania algorytmów. Wnioski z analizy algorytmów hybrydowych. Wybrać najlepsze z badanych metod sortujących z podziałem na 4 kategorie:

- dane losowe, typ podstawowy Intro Sort, Hoare partition, pseudo-medianą z dziewięciu - i podobna wartość oczekiwana liczby wykonanych operacji do Intro Sort, Hoare partition, mediana z trzech, jednak dla psuedo-mediany z dziewięciu jest mniejsze odchylenie standardowe

- dane losowe, typ złożony

- dane uporządkowane, typ podstawowy - dane uporządkowane, typ złożony

Dodać wykres przedstawiający najlepsze algorytmy w każdej z kategorii, w porównaniu do klasycznych Quick Sort oraz merge sort.



Implementacja systemu

6.1 Struktura systemu

Aplikacja składa się z dwóch modułów: silnika testującego oraz silnika graficznego. Działanie systemu jest określone na podstawie współdzielonego pliku konfiguracyjnego. W pliku konfiguracyjnym określone są rodzaje testów jakie należy przeprowadzić oraz metadane potrzebne do wygenerowania wizualizacji.

Silnik testujący to generyczna biblioteka algorytmów sortujących oraz narzędzie przetwarzające te algorytmy. Aplikacja w oparciu o plik konfiguracyjny generuje zestaw testowy oraz utrwała wyniki przeprowadzonych testów na dysku. W zależności od konfiguracji, silnik testujący może sumować, zliczać lub uśredniać liczbę wykonywanych operacji takich jak: liczba porównań, liczba operacji zamiany miejsc, liczba operacji przypisania oraz czas trwania algorytmu. Ta część aplikacji została napisana w języku C++¹ z wykorzystaniem technik programowania obiektowego.

Silnik graficzny to zbiór skryptów przetwarzających wyniki z silnika testującego. Na podstawie pliku konfiguracyjnego oraz danych testowych generowane są wizualizacje graficzne w postaci wykresów, dzięki czemu użytkownik końcowy może w łatwy sposób analizować oraz porównywać badane algorytmy. Ta część systemu została napisana w języku Python² przy użyciu biblioteki matplotlib³.

6.2 Koncepcje architektury silnika testującego

6.2.1 Wstrzykiwanie zależności

Większość algorytmów sortujących składa się z kilku odrębnych kroków. Niektóre z tych kroków są na tyle złożone, że stanowią osobne algorytmy. Dla przykładu jednym etapów sortowania metodą Quick Sort jest partyjonowanie danych wejściowych na rozłączne zbiory. Aby w łatwy sposób umożliwić modyfikację testowanych algorytmów, bez konieczności ponownej implementacji całego procesu, zastosowano technikę wstrzykiwania zależności. Jeżeli algorytm testujący korzysta z innego algorytmu, to algorytm składowy jest wstrzykiwany w trakcie działania programu. Dzięki temu lekka modyfikacja testowanego algorytmu ogranicza się do podmiany jego algorytmów składowych, bez konieczności ingerowania w strukturę bazową.

6.2.2 Obiektowość

Aby uprościć organizację kodu zastosowano model obiektowy. Każdy z algorytmów wykorzystywanych w systemie został zamodelowany za pomocą odrębnej klasy. Dla każdej rodziny algorytmów tego samego typu istnieje nadrzędna klasa bazowa określająca interfejs dla tej rodziny. Korzyści wynikające z zastosowanego modelu, takie jak statyczny polimorfizm oraz dziedziczenie, gwarantują bardziej wiarygodne działanie programu oraz umożliwiają wykrywanie błędów strukturalnych już na etapie kompilacji projektu.

¹Dokumentacja języka C++: <https://en.cppreference.com>

²Dokumentacja języka Python: <https://docs.python.org/3/>

³Dokumentacja biblioteki matplotlib: <https://matplotlib.org/>

6.2.3 Bezstanowość

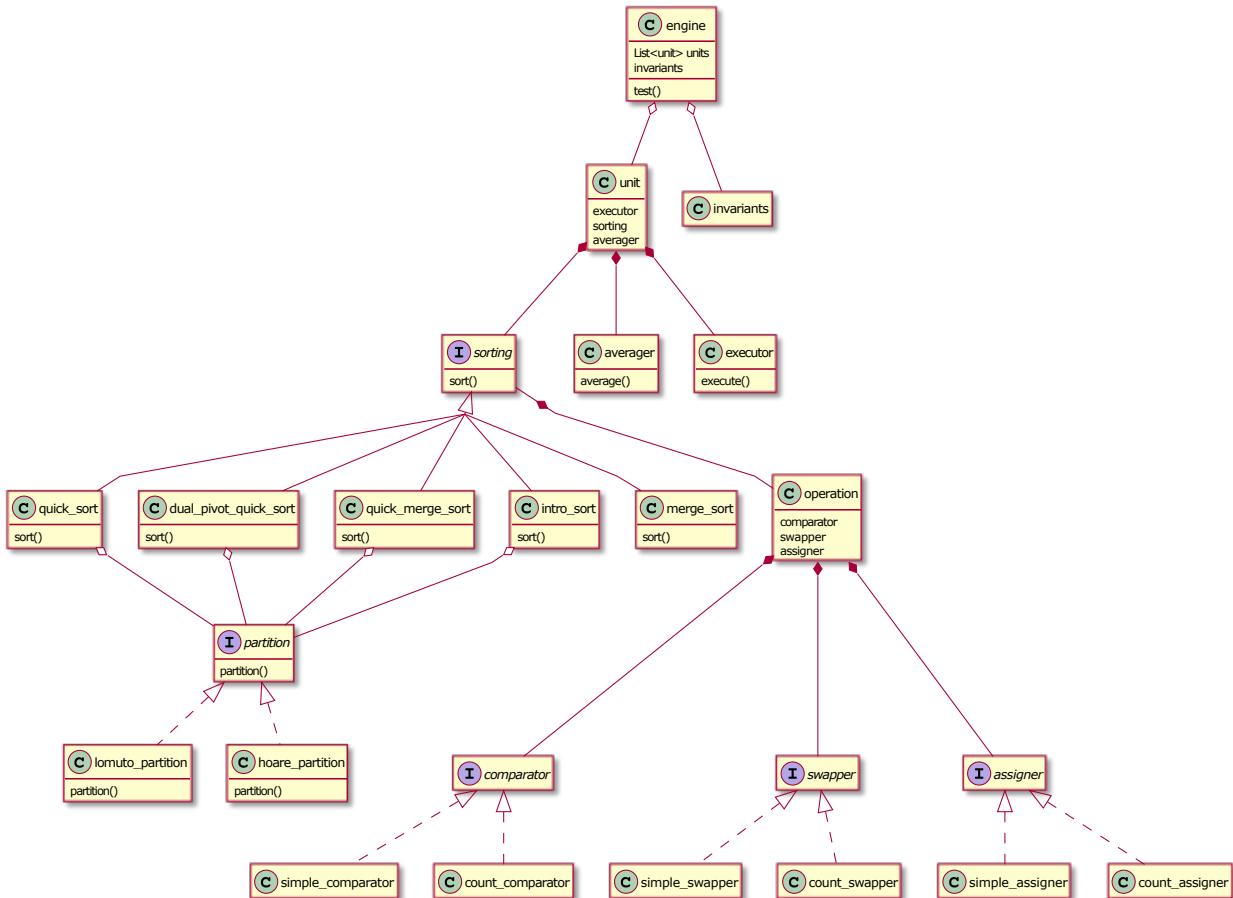
Powszechnym problemem w programowaniu obiektowym jest przechowywanie stanu. Problem ten wynika po części z praktyki hermetyzacji danych wewnętrz obiektowej abstrakcji. Użytkownik zewnętrzny korzystając z interfejsu danego komponentu nie ma dostępu do procesów zachodzących w jego wnętrzu. Może to prowadzić do tzw. efektów ubocznych (ang. side effects), przez co wyniki zwarcane przez program stają się niewiarygodne.

Aby tego uniknąć zastosowano model bezstanowy. Żaden z algorytmów sortujących w zaimplementowanym systemie nie posiada zmiennych składowych, które mogły by zostać zmodyfikowane w trakcie działania programu. Podczas testowania dane są przekazywane poprzez sygnatury metod, wzorując się na technice programowania funkcyjnego. Dzięki temu wszystkie algorytmy sortujące wykorzystane w implementacji są oznaczone jako niemodyfikowalne, nie mogą zmienić stanu aktualnie testowanego algorytmu. Gwarantuje to całkowitą separację poszczególnych testów.

6.3 Model aplikacji

6.3.1 Diagram klas

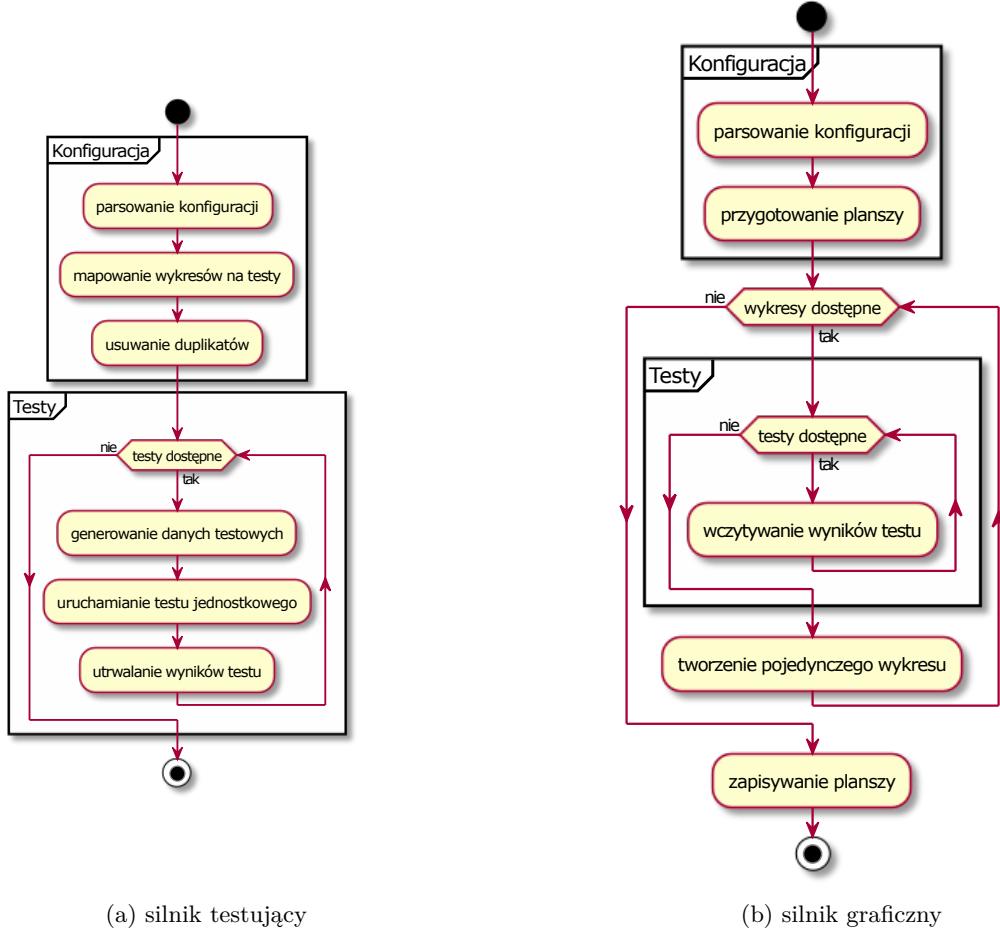
TODO: opis diagramu



Rysunek 6.1: Diagram klas silnika testującego

6.3.2 Diagram aktywności

TODO: opis diagramu



(a) silnik testujący

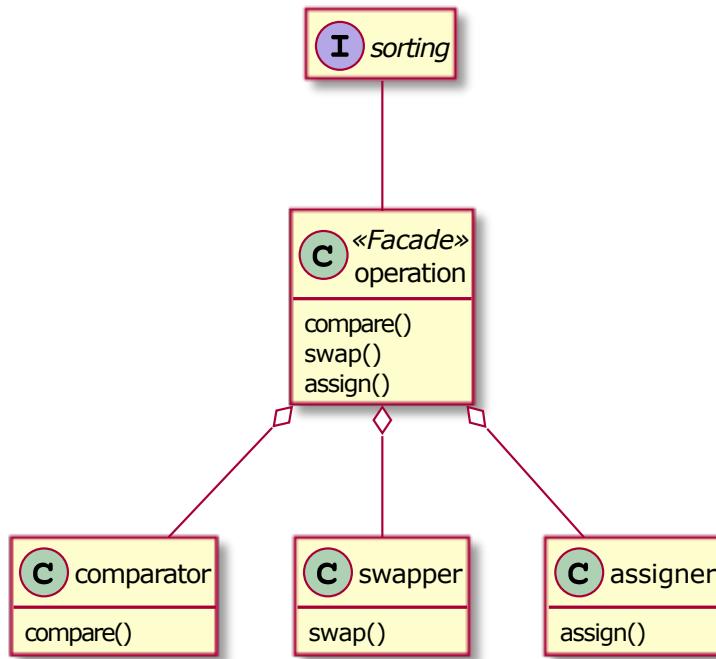
(b) silnik graficzny

Rysunek 6.2: Diagram aktywności projektowanego systemu

6.4 Wzorce projektowe

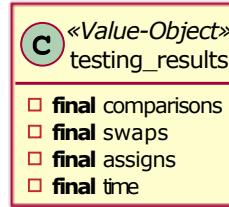
6.4.1 Fasada

W trakcie działania algorytm sortujący wykonuje wiele operacji atomowych, takich jak porównywanie elementów, zamiana elementów miejscami oraz operacje przypisania. Aby uniknąć nadmiaru odpowiedzialności dla klas sortujących zastosowano obiekt pośredniczący **operation** będący równocześnie **fasadą**. Fasada zapewnia jednolity interfejs dla wszystkich operacji atomowych oraz przekierowuje ich działanie do obiektów bezpośrednio odpowiedzialnych za ich wykonanie.



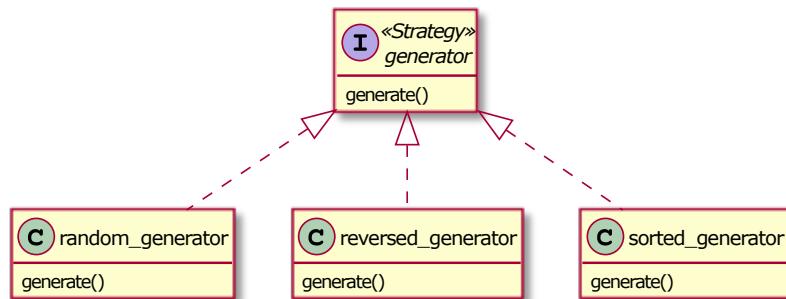
6.4.2 Obiekt-Wartość

Proces testowania algorytmu składa się z wielu iteracji. Każdy z atomowych testów wchodzących w skład iteracji powinien być całkowicie niezależny i odseparowany od innych testów. Aby to zapewnić, dane pochodzące z osobnych testów są przekazywane za pomocą **obiektów-wartości**. Pola w takim obiekcie po inicjalizacji stają się niemodyfikowalne. Użytkownik może jedynie odczytać ich wartość, bez możliwości ich modyfikacji. **Obiekt-wartość** jest gwarancją, że wyniki pochodzące z testu są rzetelne oraz nie zostały zmodyfikowane w trakcie przepływu danych pomiędzy procesami.



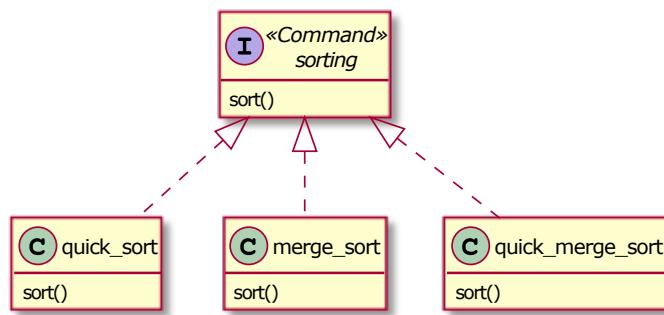
6.4.3 Strategia

Aby wymusić działanie algorytmu w przypadku optymistycznym, pesymistycznym oraz średnim konieczne jest przygotowanie sporej ilości danych, które powodują wystąpienie takiego przypadku. Aby to osiągnąć, dane testowe są tworzone za pomocą **generatora**, który jest równocześnie **strategią**. W zależności od zestawu testowego, używana jest inna strategia generowania danych, co przekłada się na późniejsze wyniki testowania.



6.4.4 Polecenie

Aby możliwe było całościowe sparsowanie pliku konfiguracyjnego jeszcze przed wykonaniem testów, algorytmy sortujące są utrwalane w pamięci w formie wzorca projektowego **polecenia**. W trakcie parsowania, polecenia są kolejkowane w formie algorytmów sortujących, a następnie przekazywane do silnika testującego. W fazie testowania wykonywane są kolejne testy z przekazanej listy poleceń.





Bibliografia

- [1] C++ documentation. <https://en.cppreference.com/w/>.
- [2] Java documentation. <https://docs.oracle.com/en/java/javase/11/docs/>.
- [3] S. Edelkamp, A. Weiß. Quickmergesort: Practically efficient constant-factor optimal sorting. 2018.
- [4] S. Edelkamp, A. Weiß. Worst-case efficient sorting with quickmergesort. 2018.
- [5] O. R. L. Peters. Pattern-defeating quicksort. 2021.
- [6] S. Wild, S. Edelkamp, A. Weiß. Quickxsort – a fast sorting scheme in theory and practice. 2019.



Słownik pojęć

A.1 Notacja duże- O

Określa asymptotyczne tempo wzrostu danej funkcji. Wykorzystywana jest do opisu złożoności obliczeniowej. Mówimy, że funkcja f ma złożoność rzędu g , tzn. $f \in O(g)$, jeśli spełniony jest warunek:

$$(\exists n_0 > 0)(\exists c > 0)(\forall n \geq n_0) f(n) \leq c \cdot g(n)$$

A.2 Algorytm działający w miejscu

Aby efektywnie posortować tablicę wejściową algorytm potrzebuje co najwyżej $O(\log n)$ dodatkowej pamięci. Algorymem działającym w miejscu nie jest klasyczny Merge Sort.

A.3 Algorytm stabilny

Algorytm stabilny w trakcie sortowania nie zamienia kolejnością elementów równoważnych względem porządku sortowania. Relatywne ułożenie elementów o tej samej wartości pozostaje takie samo.



Środowisko uruchomieniowe aplikacji

Dedykowanym środowiskiem dla implementowanego systemu jest platforma Windows 10. Do prawidłowego funkcjonowania aplikacji, konieczna jest wcześniejsza instalacja kompilatora języka C++¹ oraz interpretera języka Python².

intro_sort	1000	OK
intro_sort_median_of_three_pivot_selector	1000	OK
intro_sort_pseudo_median_of_nine_pivot_selector	1000	OK
intro_sort_median_of_medians_of_five_pivot_selector	1000	OK
intro_sort_median_of_medians_of_three_pivot_selector	1000	OK
intro_sort_hoare_partition	1000	OK
intro_sort_hoare_partition_median_of_three_pivot_selector	1000	OK
intro_sort_hoare_partition_pseudo_median_of_nine_pivot_selector	1000	OK
intro_sort_hoare_partition_median_of_medians_of_five_pivot_selector	1000	OK
intro_sort_hoare_partition_median_of_medians_of_three_pivot_selector	1000	OK

Rysunek B.1: Fragment ekranu w trakcie działania silnika testującego

B.1 Zmienne środowiskowe

Ponieważ parametry wejściowe powinny być współdzielone przez wszystkie składowe systemu, dane te są przekazywane za pomocą zmiennych środowiskowych. Każda ze zmiennych określa konkretną lokalizację na dysku systemowym:

- **CONFIG_DIRECTORY** - lokalizacja pliku konfiguracyjnego,
- **TEST_DIRECTORY** - lokalizacja pośrednich plików testowych,
- **PLOT_DIRECTORY** - lokalizacja wynikowego pliku wizualizacji.

B.2 Biblioteki zewnętrzne

W systemie wykorzystywano następujące biblioteki języka C++ w podanych wersjach:

- **fmt:7.1.3** - formatowane wyjście z interpolacją napisów,
- **nlohmann-json:3.9.1** - deserializacja pliku w formacie json,
- **scnlib:0.4** - parsowanie pliku w formacie csv.

Silnik graficzny wykorzystuje poniższe biblioteki języka Python:

- **scipy:1.7.1** - implementacje algorytmów numerycznych,
- **numpy:1.21.3** - agregacja danych liczbowych,
- **matplotlib:3.4.3** - tworzenie wizualizacji,

¹Kompilator języka C++: <https://www.mingw-w64.org/downloads/>

²Interpreter języka Python: <https://www.python.org/downloads/>



- **pandas:1.3.4** - parsowanie pliku w formacie csv,
- **SciencePlots:1.0.9** - naukowy styl wykresów.

B.3 Cykl pracy programu

Pracę ze środowiskiem można podzielić na trzy części. Na początku tworzony jest plik konfiguracyjny w formacie json. Następnie, w oparciu o przygotowaną konfigurację, uruchamiany jest silnik testujący, który generuje pośrednie pliki wyjściowe. W ostatnim kroku uruchamiany jest silnik graficzny, który w oparciu o pliki wyjściowe pochodzące z silnika testującego, przygotowuje ostateczną wizualizację wyników. Ponieważ przy starcie silnika graficznego weryfikowana jest poprawność wszystkich plików pośrednich, każdy z kroków wykonywany jest sekwencyjnie, zaś cały program działa synchronicznie.

B.4 Przykładowy plik konfiguracyjny

Plik konfiguracyjny to dokument tekstowy w formacie json, zawierający wszystkie parametry potrzebne do przeprowadzanie testów. W pliku ten można wyodrębnić dwie główne części. Część pierwsza zawiera parametry konfiguracyjne wspólne dla wszystkich testów, jak np. tytuł zestawu testowego, nazwa pliku wyjściowego czy rozmiar generowanej wizualizacji. W części drugiej zawarte są parametry unikatowe dla poszczególnych testów. Do parametrów tych należą m. in. opisy osi rzędnych i odciętych, generatory danych wejściowych dla poszczególnych testów oraz tytuły dla każdego wykresów. Dokładny spis wszystkich parametrów zawarty jest na końcu tego rozdziału. Poniżej znajduje się przykładowy plik konfiguracyjny.

```
{  
    "title": "Tytul zestawu testowego",  
    "output": "nazwa-pliku-wizualizacji",  
    "prefix": "test1",  
    "grid": "(1, 2)",  
    "size": "(8, 3.5)",  
    "invariants": {  
        "range": {  
            "begin": 1000,  
            "end": 1000000,  
            "step": 1000  
        },  
        "repeats": 1000  
    },  
    "plots": [  
        {  
            "type": "average",  
            "metadata": {  
                "title": "a) dane losowe",  
                "xlabel": "rozmiar tablicy [n]",  
                "ylabel": "liczba operacji [n]",  
                "xcolumn": "n",  
                "ycolumn": "cost",  
                "legend": "none"  
            },  
            "functions": [  
                {  
                    "label": "$n$ ",  
                    "expression": "n"  
                }  
            ],  
            "sortings": [  
                {  
                    "algorithm": "quick_sort",  
                    "generator": "random"  
                },  
                {  
                    "algorithm": "quick_sort_hoare_partition",  
                    "generator": "reversed"  
                }  
            ]  
        }  
    ]  
}
```

Kod źródłowy B.1: Przykładowy plik konfiguracyjny



title	tytuł wizualizacji
output	nazwa pliku wyjściowego
prefix	prefiks dodawany do nazw plików pośrednich
grid	układ wykresów na wizualizacji (<i>x,y</i>)
size	rozmiar wizualizacji (<i>x,y</i>)
invariants	rozmiar tablicy oraz liczba powtórzeń
range	rozmiar tablicy
begin	początkowy rozmiar tablicy
end	końcowy rozmiar tablicy
step	inkrement podczas zmiany rozmiaru tablicy
repeats	liczba powtórzeń dla każdego z testów
plots	osobne konfiguracje dla poszczególnych wykresów
type	rodzaj testu
metadata	parametry graficzne dla pojedynczego wykresu
title	tytuł wykresu
ticklabel.format	tryb drukowania etykiet na osiach
xlabel	podpis osi odciętych
ylabel	podpis osi rzędnych
xcolumn	dane do osi odciętych
ycolumn	dane do osi rzędnych
xmin	minimalna wartość na osi odciętych
xmax	maksymalna wartość na osi odciętych
legend	pozycja legendy
functions	funkcje pomocnicze drukowane na wykresie
sortings	wyniki testów drukowane na wykresie
label	podpis linii
color	kolor linii
linestyle	styl linii
expression	generator dla funkcji pomocniczej
algorithm	algorytm dla przeprowadzanego testu
generator	generator danych wejściowych przeprowadzanego testu

Tablica B.1: Opis parametrów zawartych w pliku konfiguracyjnym.