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
Wrocławska
Wrocław 2021

Spis treści

Wstęp

Ogólna historia algorytmów sortujących. Motywacja tworzenie algorytmów

1.1 Cel pracy

TODO: Motywem przewodnim pracy jest analiza nowoczesnych algorytmów sortujących w miejscu, takich jak koncepcja QuickMerge Sort. W tym celu przygotowano analizę porównawczą podstawowych algorytmów sortujących oraz dokonano przeglądu zmodyfikowanych wersji tych algorytmów oraz przeanalizowano nowoczesne algorytmy hybrydowe, będące połączeniem dwóch lub wielu algorytmów podstawowych.

1.2 Zakres pracy

TODO: Aby ułatwić analizę algorytmów przygotowany został silnik testujący oraz silnik graficzny, które w oparciu o plik konfiguracyjny przeprowadzają testy oraz tworzą wizualizację wyników tych testów. Wykorzystując podane narzędzia została przeprowadzona analiza podstawowych oraz hybrydowych algorytmów.

1.3 Przegląd literatury

TODO: Ogólny opis pracy Sebastiana Wilda. Pomysł na algorytm QuickMerge Sort.

1.4 Zawartości pracy

TODO: Ogólny sposób organizacji dokumentu. Rozdział pierwszy - analiza matematyczna problemu. Rozdział drugi - przegląd podstawowych algorytmów sortujących. Rozdział trzeci - przegląd hybrydowych algorytmów sortujących.

Analiza problemu

2.1 Model matematyczny przypadku średniego

2.2 Model matematyczny przypadku pesymistycznego

2.3 Założenia

2.3.1 Założenia odnośnie testowanych parametrów

TODO: Testowana będzie złożoność czasowa, w tym celu analizowane są takie parametry jak: liczba porównań, liczba swapów oraz liczba przypisań, z pominięciem operacji wykonywanych na iteratorach pętli - wyjaśnienie dlaczego.

TODO: Ponieważ rzeczywisty czas wykonywania algorytmu różni się w zależności od maszyny oraz architektury systemu na którym przeprowadzany test, zostało przyjęte następujące założenie: Złożoność czasowa została określona wzorem:

TODO silnik testujący zlicza liczbę wykonanych operacji atomowych. Do operacji atomowych zaliczamy: operację porównania, zamiany miejsc oraz przypisania. W badanym systemie pominięte zostały operacje przeprowadzane na indeksach oraz iteratorach. Powodem tej decyzji jest fakt, że zarówno indeksy, jak i iteratory w większości systemów są typami podstawowymi, koszta takich operacji jest niewspółmiernie mniejszy niż np. porównywanie typów złożonych. W tej sytuacji zdecydowano się całkowicie pominąć liczbę wykonywanych operacji na typach prymitywnych.

TODO łączna liczba wykonanych operacji jest sumą ważoną operacji atomowych. TODO wspolczynnik kosztu $\boldsymbol{\alpha}$, czyli ile razy kosztowniejsze jest porownanie od przypisania. zaklasamy ze zmiana miejsc to trzy przypisania. Dla potrzeb testow podstawowych przyjmuje sie $\boldsymbol{\alpha = 1.0}$. TODO w prostej analizie dla typow podstawowych przyjmujemy ze operacja zamiany miejsc jest 3 razy bardziej kosztowna od operacji przypisania oraz porownania.

$$\boldsymbol{T = n_c + 3 \cdot n_s + n_a}$$

TODO w celu przeprowadzenia analizy porównawczej wprowadzono pojęcie współczynnika kosztu.

Gdzie:

$\boldsymbol{T}$ - czas trwania algorytmu
$\boldsymbol{n_c}$ - liczba operacji porównania
$\boldsymbol{n_s}$ - liczba operacji zamiany miejsc
$\boldsymbol{n_a}$ - liczba operacji przypisania

Ponieważ wszystkie algorytmy zawarte w silniku testującym są rekurencyjne, podczas badania algorytmów pominięto koszt związany z rekurencyjnym przejściem drzewa wywołania. Ponieważ w analizie porównawczej uwzględniono tylko algorytmy działające w miejscu, podczas badania nie uwzględniano kosztu alokacji pamięci, która dla każdego algorytmu miejscowego jest rzędu $\boldsymbol{O(1)}$.

Rozróżnić łączny koszt operacji od łącznej liczby operacji.

2.3.2 Założenia odnośnie danych wejściowych

TODO: Wiele algorytmów sortujących bazuje na pewnych założeniach odnośnie wejściowego zbioru danych. Np. algorytm ... doskonale radzi sobie ze zbiorem danych prawie posortowanym, tzn. takim w którym ... . W tej pracy zakładamy że dane wejściowe będą losowym ciągiem liczb powtórzeniami.

Przegląd podstawowych algorytmów sortujących

3.1 Quick Sort

algorytm stabilny	**NIE**
algorytm miejscowy	**TAK**
optymistyczna złożoność czasowa	$\boldsymbol{O(n \log n)}$
pesymistyczna złożoność czasowa	$\boldsymbol{O(n^2)}$
średnia złożoność czasowa	$\boldsymbol{O(n \log n)}$
złożoność pamięciowa	$\boldsymbol{O(1)}$

Historia algorytmu Quick Sort sięga drugiej połowy XX wieku. W roku 1959 brytyjski naukowiec Tony Hoare opracował, a dwa lata później opublikował pierwszą wersję tego algorytmu. Od tamtego czasu powstało wiele udoskonaleń tego algorytmu, jednak jego koncepcja nadal jest widoczna we współczesnych językach programowania [1]. Na cześć algorytmu Quick Sort standardowa funkcja sortująca w języku C++ nosi nazwę qsort [2].

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, ponieważ problem sortowania tablicy wejściowej rozbija 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 pivot dokładnie w połowie tablicy. Z analizy eksperymentalnej wynika, że w przypadku optymistycznym algorytm działa ze złożonością czasową $\boldsymbol{O(n \log n)}$.

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

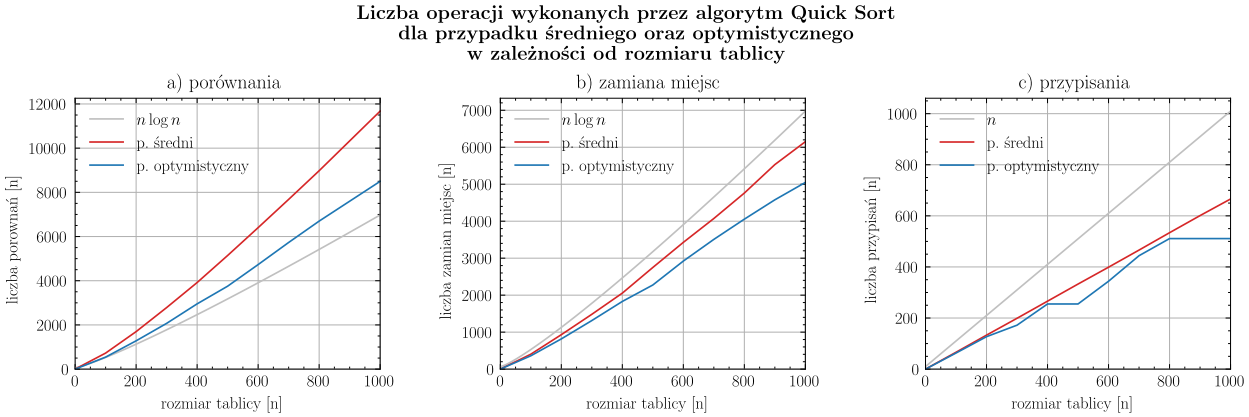
[2] Dokumentacja funkcji sortującej qsort: https://en.cppreference.com/w/cpp/algorithm/qsort

Przypadek pesymistyczny (3.2) zachodzi, gdy drzewo wowołań rekurencyjnych jest możliwie najgłębsze. Efekt ten uzyskano, wprowadzając dane już posortowane oraz stosując algorytm wybierający pivot jako ostatni element tablicy. W tej sytuacji w kolejnych iteracjach rozpatrywana jest tablica z rozmiarem o jeden mniejszy od poprzedniej, a więc drzewo wywołań rekurencyjnych ma głębokość $\boldsymbol{n}$. Z analizy wynika, że złożoność czasowa algorytmu w przypadku pesymistycznym wynosi $\boldsymbol{O(n^2)}$.

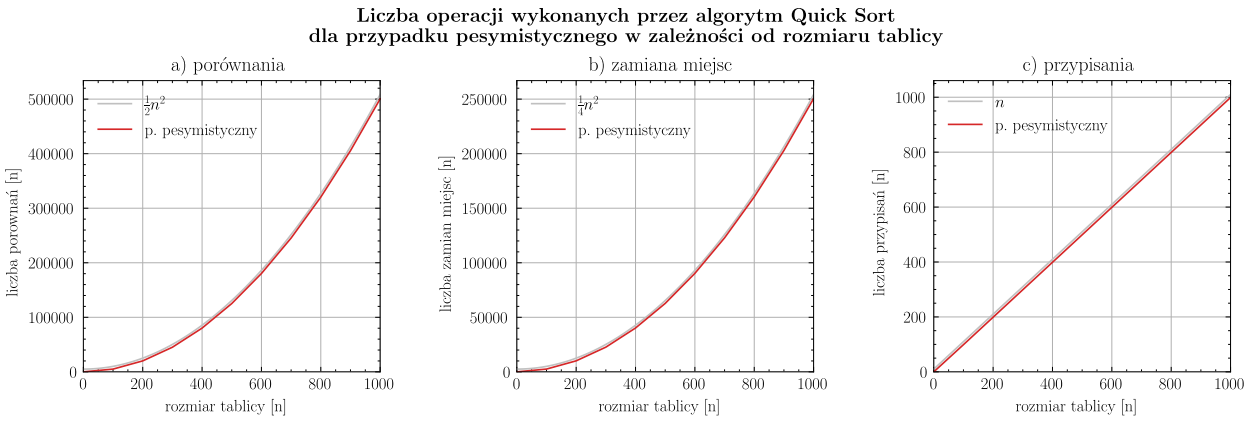
Przypadek średni (3.1) został zbadany wprowadzając losowe dane z powtórzeniami oraz stosując algorytm wybierający pivot jako ostatni element tablicy. Analiza eksperymentalna wykazała, że w przypadku średnim algorytm Quick Sort ma złożoność czasową równą $\boldsymbol{O(n \log n)}$, a więc jest tego samego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

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

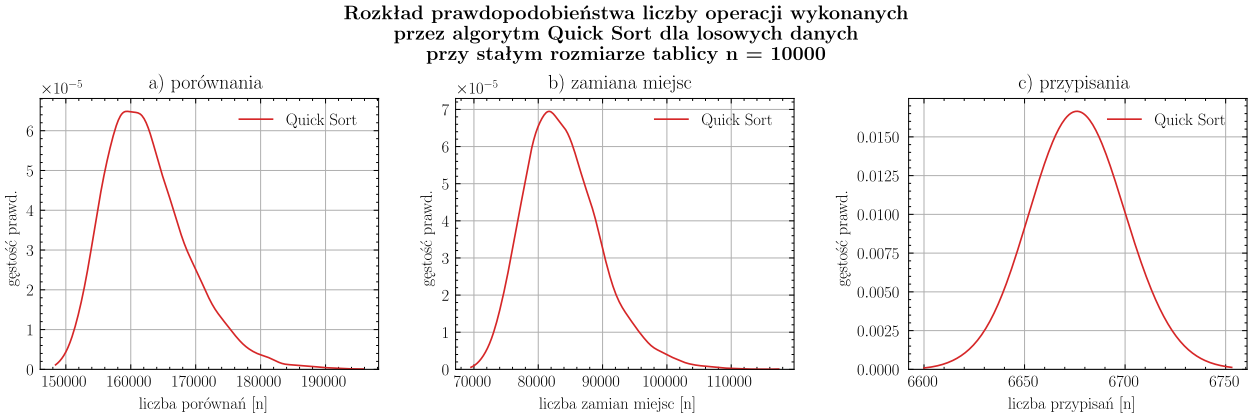
Analizując rozkład prawdopodobieństwa liczby wykonanych operacji (3.3) użyto tablicy losowych danych o stałym rozmiarze $\boldsymbol{n = 10000}$. Można zauważyć, że liczba operacji porównania oraz liczba operacji zamiany miejsc nie są przedstawiane za pomocą rozkładu normalnego. Bardziej prawdopodobne jest wykonanie większej liczby tych operacji w stosunku do wartości średniej. Z kolei rozkład liczby wykonanych operacji przypisania przedstawia się za pomocą rozkładu normalnego, z jednakowym prawdopodobieństwem liczba ta może być większa lub mniejsza od wartości średniej.



Rysunek 3.1



Rysunek 3.2



Rysunek 3.3

3.1.2 Problemy związane z algorytmem Quick Sort

Głównym problemem algorytmu Quick Sort jest jego słaba pesymistyczna złożoność czasowa. Ponieważ algorytm działa rekurencyjnie, w przypadku pesymistycznym głębokość drzewa wywołań rekurencyjnych może przekroczyć maksymalną liczbę ramek stosu, powodując awaryjne zatrzymanie programu.

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 jest znacznie kosztowniejsze od pozostałych operacji. W tym przypadku bardziej wskazanym wydaje się użycie algorytmu Merge Sort, którego analizę przeprowadzono w kolejnym rozdziale.

3.1.3 Możliwości optymalizacyjne

Ponieważ złożoność czasowa algorytmu Quick Sort uwarunkowana jest poprzez złożoność algorytmu partycjonowania, optymalizacja algorytmu może opierać się ulepszanie 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 miejscowy	**NIE**
optymistyczna złożoność czasowa	$\boldsymbol{O(n \log n)}$
pesymistyczna złożoność czasowa	$\boldsymbol{O(n \log n)}$
średnia złożoność czasowa	$\boldsymbol{O(n \log n)}$
złożoność pamięciowa	$\boldsymbol{O(n)}$

Algorytm Merge Sort został opracowany przez Johna von Neumanna w 1945 roku. Tak jak Quick Sort, algorytm ten wykorzystuje technikę dziel i zwyciężaj aby rekurencyjnie rozbić problem sortowania danych wejściowych na dwie listy mniejszej długości. W przeciwieństwie do algorytmu Quick Sort, algorytm Merge Sort do poprawnego działania potrzebuje dodatkowej pamięci. W rozważanej implementacji algorytm alokuje dodatkowy bufor długości $\boldsymbol{n/2}$.

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 cząstkowych 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. Z analizy eksperymentalnej wynika, że w tym przypadku algorytm Merge Sort działa ze złożonością czasową rzędu $\boldsymbol{O(n \log n)}$.

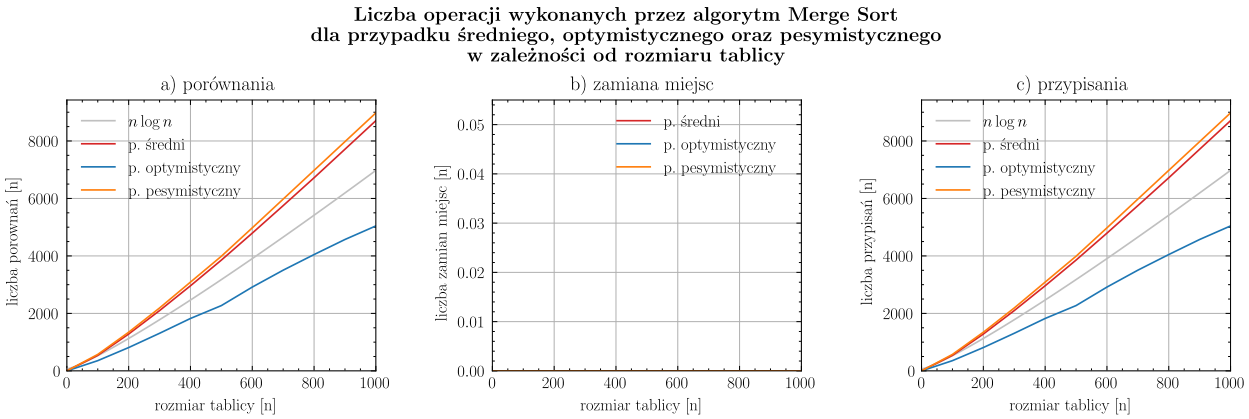
Przypadek pesymistyczny (3.4) zachodzi, gdy liczba porównać między sobą elementów w trakcie scalania jest możliwie największa. Efekt ten uzyskano, preparując dane wejściowe w taki sposób, aby w każdym kroku scalania porównywane tablice zawierały elementy na przemian większe oraz mniejsze. Z analizy wynika, że złożoność czasowa algorytmu w przypadku pesymistycznym jest równa $\boldsymbol{O(n \log n)}$, a więc jest tego samego rzędu co złożoność optymistyczna.

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

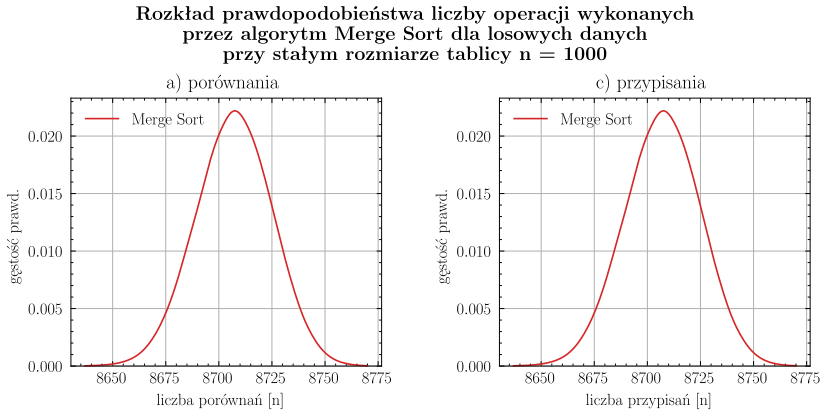
W przeciwieństwie do algorytmu Quick Sort, algorytm Merge Sort wykonuje dokładnie tyle samo operacji porównania co operacji przypisania. Jest to zgodne z rzeczywistością, ponieważ w trakcie scalania każdej operacji porównania towarzyszy przypisanie wartości do bufora. Można zauważyć, że algorytm Merge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

Do analizy rozkładu prawdopodobieństwa liczby wykonanych operacji (3.5) użyto tablicy losowych danych o stałym rozmiarze $\boldsymbol{n = 1000}$. W analizie pominięto liczbę operacji przypisania, która jest zerowa dla każdego przypadku. Analiza eksperymentalna dowodzi, że w przeciwieństwie do algorytmu Quick Sort, liczba operacji

porównania dla algorytmu Merge Sort tworzy rozkład normalny, a więc prawdopodobieństwa otrzymania większej oraz mniejszej liczby operacji przypisania są jednakowe. Ponieważ dla badanego algorytmu, liczba operacji przypisania jest równa liczbie operacji porównania, rozkłady tych wartości są jednakowe.



Rysunek 3.4



Rysunek 3.5

3.2.2 Problemy związane z algorytmem Merge Sort

Głównym problemem algorytmu Merge Sort jest konieczność alokacji dodatkowego bufora pamięci. W rozważanej implementacji algorytm alokuje dodatkowy bufor długości $\boldsymbol{n/2}$, a więc 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 skrzyżowanie 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. Pomysł 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. Pomysł 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 pivota. Do partycjonowania danych wejściowych wykorzystano poniższe metody:

- **metoda Lemuto** - jest domyślnym algorytmem partycjonowania w projektowanym systemie. Tablica jest iterowana od pierwszego do ostatniego elementu. Elementy mniejsze od pivota są przenoszone na lewą część tablicy, zaś elementy większe od pivota na jej prawą część. Algorytm kończy się w momencie przeniesienia ostatniego elementu.
- **metoda Hoarego** - 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 pivota, oraz element po prawej stronie jest mniejszy od pivota. 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 metody wyboru pivota:

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

- **mediana-median z pięciu** - pivot jest medianą pięciu median obliczanych rekurencyjnie,
- **mediana-median z trzech** - pivot jest medianą trzech median obliczanych rekurencyjnie.

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 $\boldsymbol{\alpha = 1.0}$.

Dla losowych danych wejściowych (4.1), czyli przypadku średniego w klasycznym algorytmie Quick Sort, przy założeniu że operacja porównania jest czasowo równoważna operacji przypisania ($\boldsymbol{\alpha = 1.0}$), partycjonowanie metodą Hoarego jest wydajniejsze lub tak samo wydajne jak partycjonowanie metodą Lemuto. Najlepsze wyniki otrzymano poprzez połączenie partycjonowania metodą Hoarego z wyborem pivota jako ostatni element tablicy. Najgorsze wyniki otrzymano wybierając pivot 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 ($\boldsymbol{\alpha = 1.0}$), partycjonowanie metodą Hoarego jest również wydajniejsze lub tak samo wydajne jak partycjonowanie metodą Lemuto. Najlepsze wyniki otrzymano poprzez połączenie partycjonowania metodą Hoarego z wyborem pivota jako mediana z trzech. Najgorsze wyniki otrzymano dla klasycznego algorytmu Quick Sort, czyli poprzez połączenia partycjonowania metodą Lemuto z wyborem pivota jako ostatni element tablicy.

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

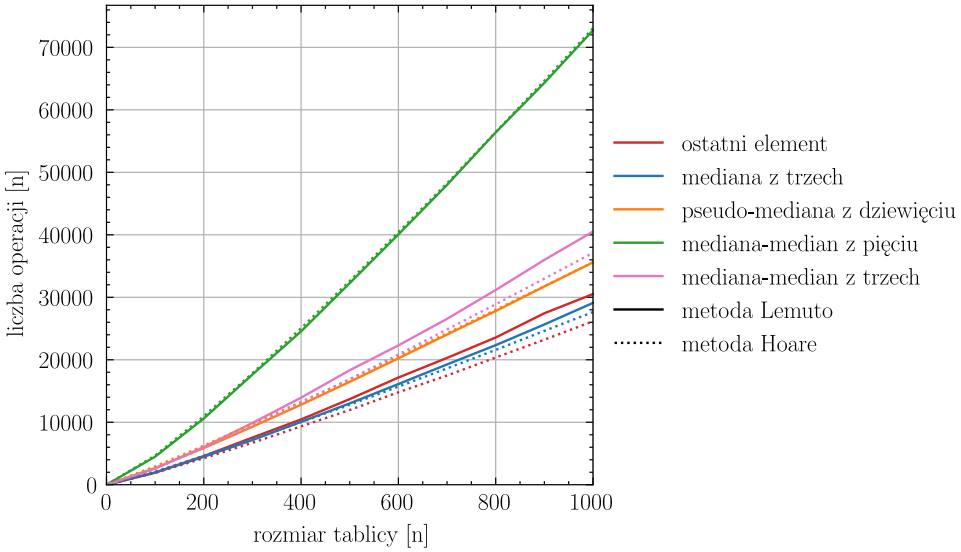
Porównując liczbę wykonanych operacji pomiędzy badanymi algorytmami partycjonowania (4.5) można stwierdzić, że partycjonowanie metodą Hoare wykonuje większą liczbę operacji porównania oraz mniejszą liczbę operacji zamiany miejsc. Czynnik ten może okazać się szczególnie istotny w przypadku sortowania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

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

4.2.2 Wnioski

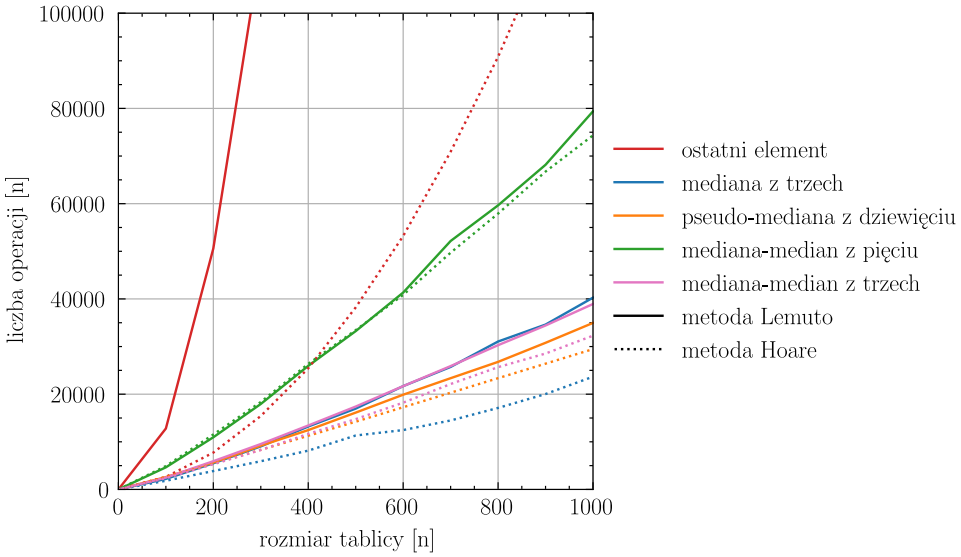
W przypadku danych wejściowych typu prostego, tzn. takich, dla których operacja porównania jest czasowo równoważna operacji przypisania, partycjonowanie metodą Hoare okazało się wydajniejsze niż partycjonowanie metodą Lemuto. W przypadku złożonych struktur danych, wydajniejsze jest partycjonowanie metodą Lemuto. Wartość współczynnika kosztu, od której partycjonowanie metodą Lemuto jest bardziej wydajne dla dowolnej strategii wyboru pivota wynosi $\boldsymbol{\alpha = 2.0}$, co można interpretować jako sortowania struktur danych

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



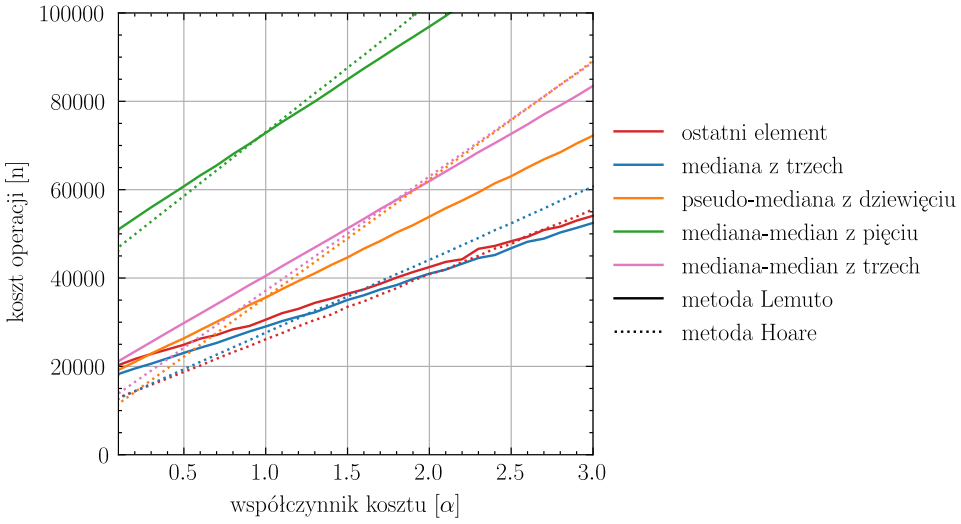
Rysunek 4.1

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



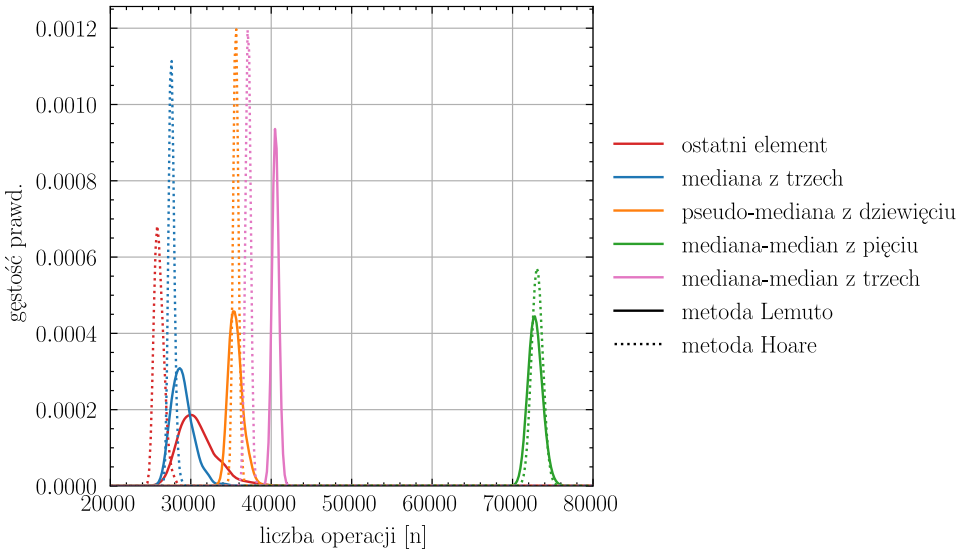
Rysunek 4.2

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

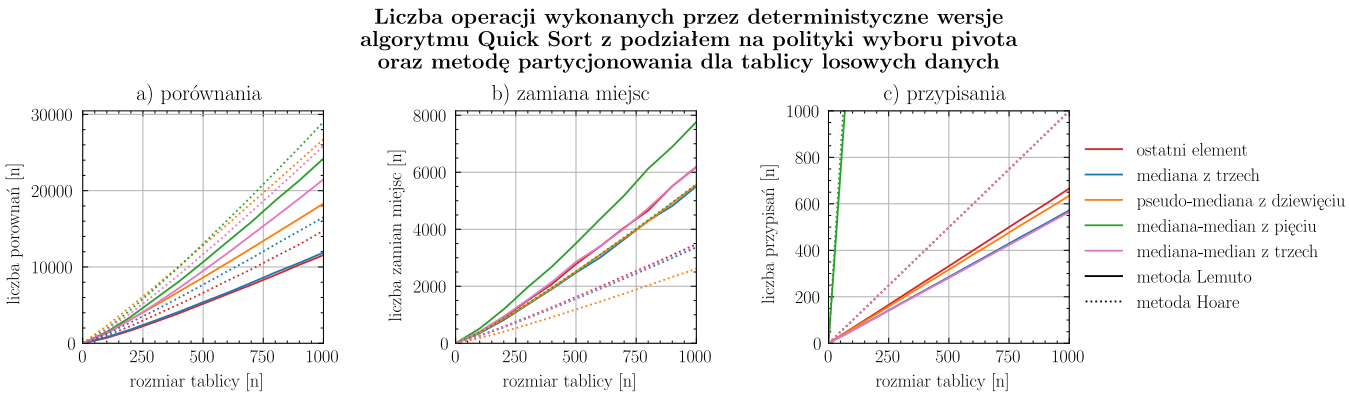


Rysunek 4.3

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



Rysunek 4.4



Rysunek 4.5

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 pivota, która pomimo swojego dodatkowego nakładu czasowego zwiększa prawdopodobieństwo na znalezienie dobrego pivota. Wyjątkiem jest wybór pivota 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 pivota** rozumiemy tutaj pozycję możliwie blisko środka partycjonowanej 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 pivota jako ostatni element tablicy, algorytm ten działa ze złożonością czasową $\boldsymbol{O(n^2)}$. W przypadku uporządkowanych danych wejściowych skutecznym sposobem może okazać się niedeterministyczny wybór pivota. W tym rozdziale dokonano analizy randomizowanych algorytmów z rodziny Quick Sort, z podziałem na metody partycjonowania Lemuto oraz Hoarego. W analizie wykorzystano następujące metody wyboru pivota:

- **losowy element** - pivotem jest losowo wybrany element tablicy,
- **mediana z trzech wyborów** - przystosowanie meotdy **power of two choices** do potrzeby wyznaczania mediany, pivotem jest mediana z trzech losowo wybranych elementów,
- **pseudo-mediana z dziewięciu wyborów** - pivotem 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 $\boldsymbol{\alpha = 1.0}$.

Dla losowych danych wejściowych (4.6), przy założeniu że operacja porównania jest czasowo równoważna operacji przypisania ($\boldsymbol{\alpha = 1.0}$), większość randomizowanych metod wyboru pivota okazuje się mniej skuteczna od klasycznego algorytmu Quick Sort. Przy partycjonowaniu metodą Lemuto jedyną skuteczniejszą

metodą jest wybór pivota jako mediana z trzech losowych elementów. Przy partycjonowaniu metodą Hoarego, najskuteczniejszym okazuje się wybór za pivota losowego elementu tablicy. Połączenie metody Hoarego z losowym wyborem pivota jest również najskuteczniejszym podejściem podczas sortowania losowych danych. Najmniej wydajną strategią sortowania okazał się wybór pivota 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 ($\boldsymbol{\alpha = 1.0}$), najskuteczniejszą strategią sortowania również okazało się partycjonowanie metodą Hoarego przy wyborze pivota jako losowy element tablicy. Spośród randomizowanych strategii sortowania najmniej wydajną okazał się wybór pivota 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 kosztu $\boldsymbol{\alpha}$ można zauważyć, że dla złożonych typów danych najskuteczniejszą z badanych strategii jest partycjonowanie metodą Lemuto z wyborem pivota jako mediana z trzech losowych elementów. Metoda ta jest najskuteczniejsza powyżej wartości $\boldsymbol{\alpha = 2.5}$, co można interpretować jako sortowanie struktur złożonych z co najmniej trzech typów prostych. Najmniej skuteczną strategią sortowania złożonych struktur jest wybór pivota pseudo-mediana z dziewięciu losowych elementów.

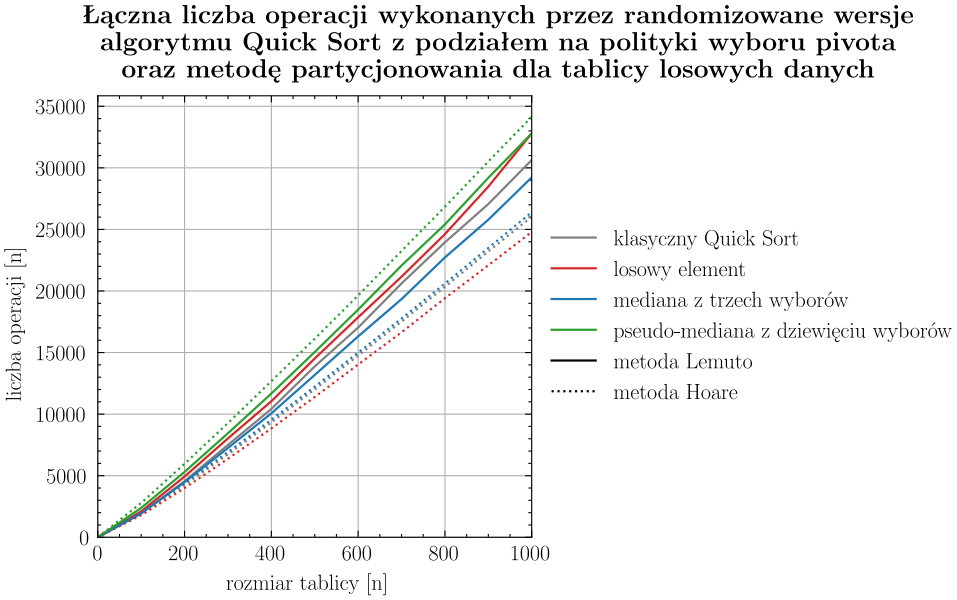
Porównując liczbę wykonanych operacji (4.10) można stwierdzić, że dla dowolnych randomizowanych strategii wyboru pivota, partycjonowanie metodą Hoare wykonuje większą liczbę operacji porównania oraz mniejszą liczbę operacji zamiany miejsc. Tak jak w przypadku algorytmów deterministycznych, czynnik ten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

Badając rozkład prawdopodobieństwa 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 Lemuto. Każdy z randomizowanych algorytmów wyboru pivota daje w wyniku mniejsze odchylenie standardowe od klasycznego algorytmu Quick Sort. Najmniejsze odchylenie standardowe uzyskano dla partycjonowania metodą Hoare z wyborem pivota jako pseudo-mediana z dziewięciu wyborów, jednak równocześnie dla tego algorytmu uzyskano największą wartość oczekiwaną liczby wykonanych operacji. Najmniejszą wartość oczekiwaną uzyskano dla partycjonowania metodą Hoare przy wyborze pivota jako losowy element tablicy.

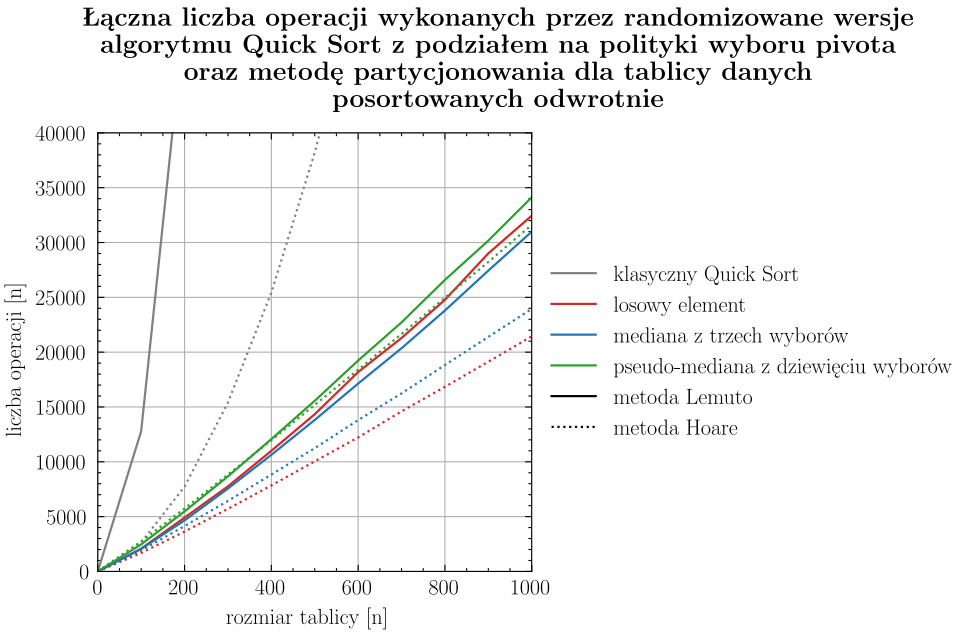
4.3.2 Wnioski

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 kosztu równej $\boldsymbol{\alpha = 4.0}$, partycjonowanie metodą Lemuto jest wydajniejsze niż partycjonowanie metodą Hoare, niezależnie od strategii wyboru pivota. Sytuacja ta jest równoważna z sortowaniem struktur składających się z co najmniej czterech zmiennych podstawowych. W tym przypadku należy skorzystać z partycjonowania metodą Lemuto.

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

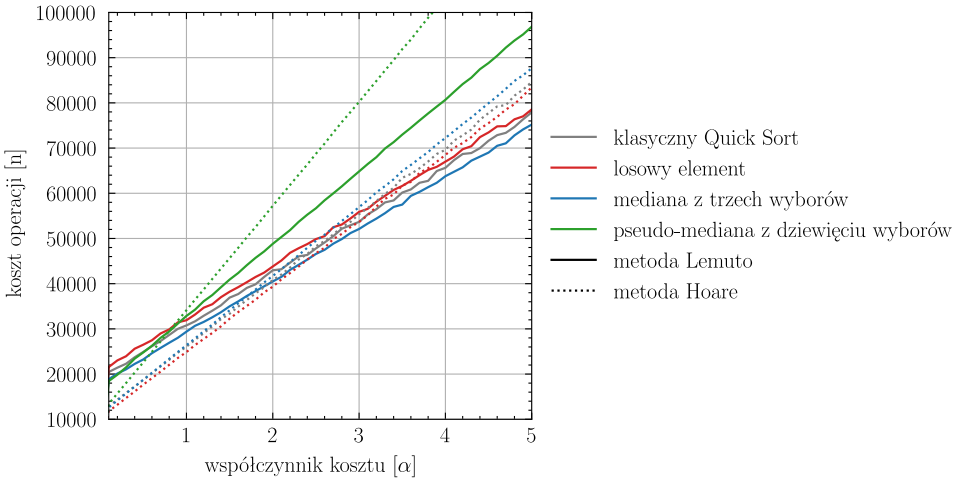


Rysunek 4.6



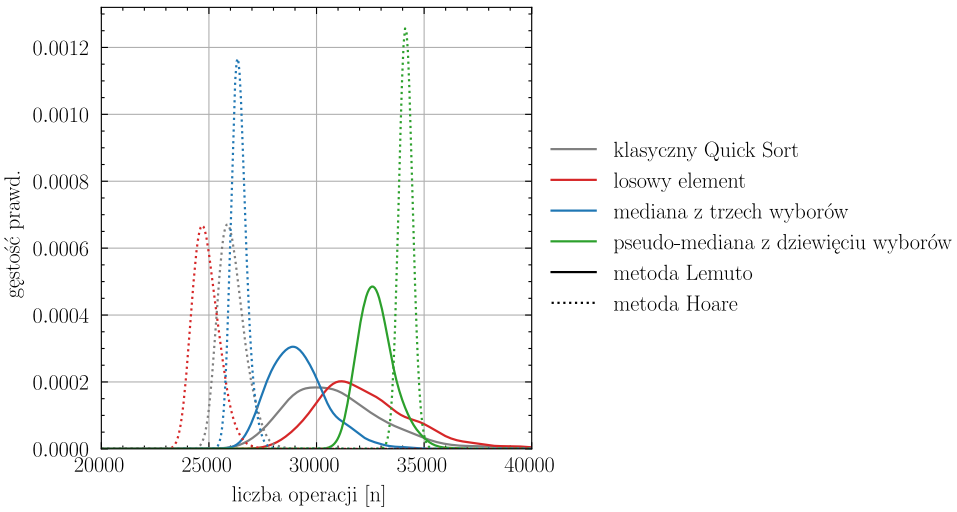
Rysunek 4.7

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

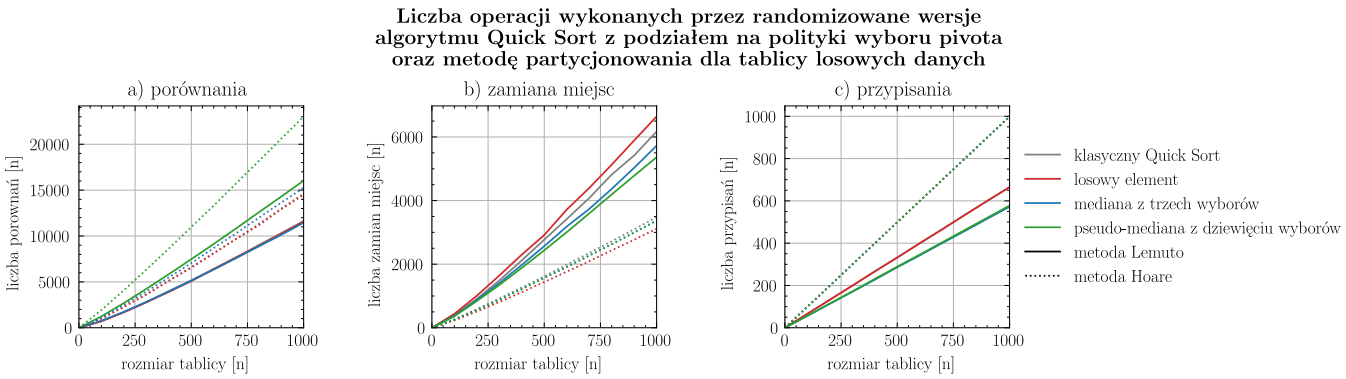


Rysunek 4.8

Rozkład prawdopodobieństwa 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. 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.

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 cząstkowych był fragment tej samej tablicy wejściowej.

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

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 cząstkowych 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                                  ▷ warunek wyjścia
4:      end if
5:
6:      arr_1, arr_2 ← Partition(arr)                             ▷ partycjonowanie
7:
8:      if len(arr_1) < len(arr_2) then                           ▷ sortowanie
9:          buffer ← arr_2
10:         MergeSortBySwaps(arr_1, buffer)
11:         QuickMergeSort(arr_2)
12:     else
13:         buffer ← arr_1
14:         MergeSortBySwaps(arr_2, buffer)
15:         QuickMergeSort(arr_1)
16:     end if
17:
18: end procedure
```

4.4.2 Analiza deterministycznych wersji algorytmu QuickMerge Sort

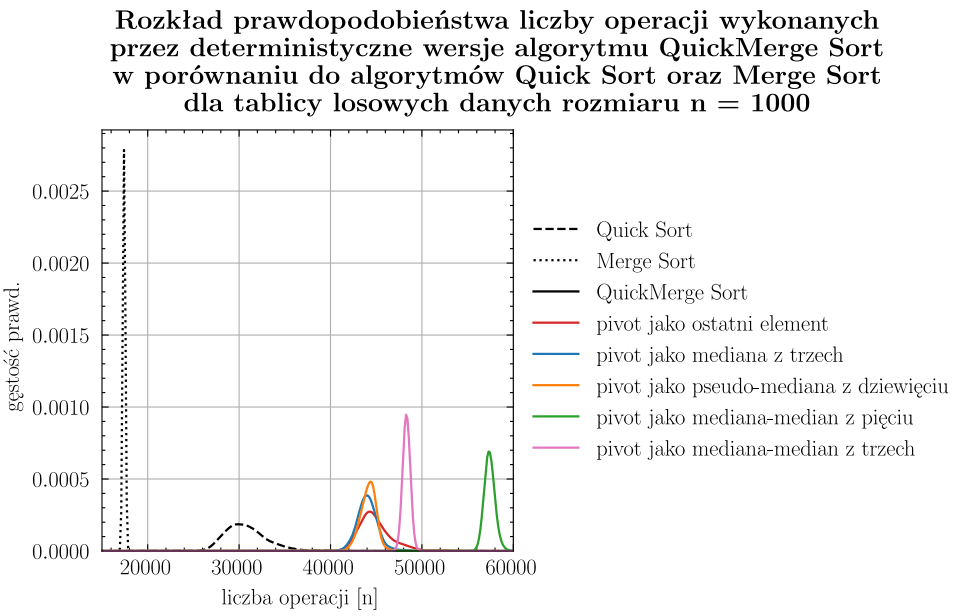
Analizując wydajność deterministycznych algorytmów z rodziny QuickMerge Sort, badano liczbę wykonywanych operacji atomowych z podziałem na operacje porównania, zamiany miejsc oraz przypisania. Dodatkowo badano łączny koszt wykonanych operacji jako sumę ważoną liczby operacji atomowych, z uwzględnieniem wartości współczynnika kosztu. Badając liczbę poszczególnych operacji uwzględniono klasyczny algorytm Merge Sort oraz najbardziej klasyczną wersję algorytmu Quick Sort z partycjonowaniem metodą Lemuto 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 Lemuto oraz wyborem piwota jako pseudo-mediana 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 zauważyć, 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 ($\boldsymbol{\alpha = 1.0}$) można zauważyć, że najwydajniejszym algorytmem działającym w miejscu jest klasyczny algorytm Quick Sort. Dla danych tego typu korzystanie z algorytmu QuickMerge Sort powoduje dodatkowy narzut czasowy. Sytuacja ta zmienia się w przypadku sortowania złożonych struktur wejściowych ($\boldsymbol{\alpha \geqslant 7.0}$). Od tego momentu łączny koszt wykonywanych operacji w przypadku algorytmu QuickMerge Sort jest mniejszy niż klasyczne podejście Quick Sort, jednak tylko dla strategii wyboru piwota jako ostatni element, mediana z trzech lub pseudo-mediana z dziewięciu. Najwydajniejszą spośród badanych metod dla złożonych struktur danych jest wybór piwota jako pseudo-mediana 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.

Do analizy rozkładu prawdopodobieństwa liczby wykonanych operacji (4.11) użyto tablicy losowych da-



Rysunek 4.11

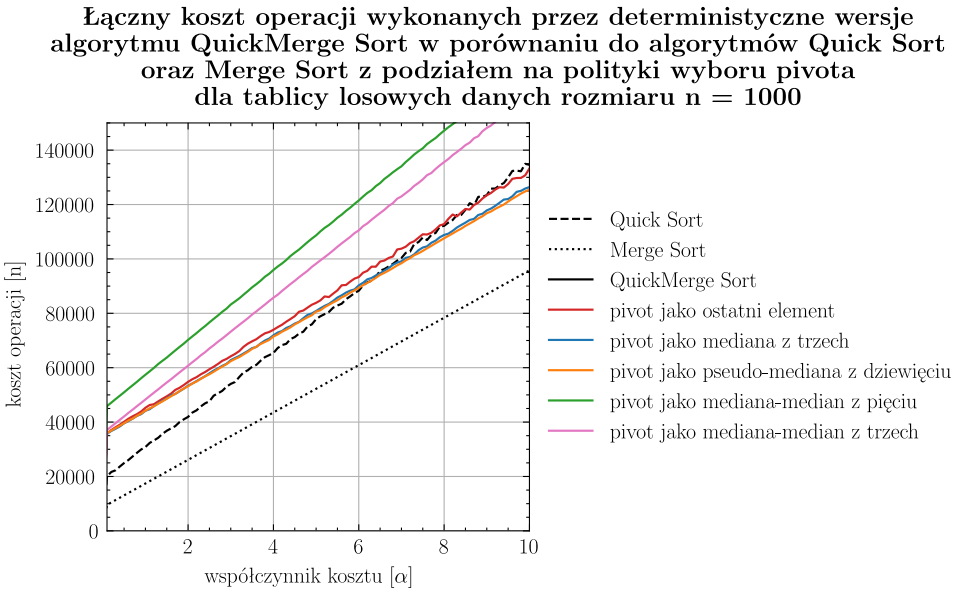
nych o stałym rozmiarze $\boldsymbol{n = 1000}$ oraz współczynnika kosztu o stałej wartości $\boldsymbol{\alpha = 1.0}$. Można zauważyć, że sortując dane algorytmem QuickMerge Sort, skutecznie zmniejszono wartość oczekiwaną liczby wykonanych operacji, niezależnie od metody wyboru piwota. Z badanych metod najmniejszą wartość oczekiwaną liczby wykonanych operacji ma klasyczny algorytm Quick Sort. Największą wartość oczekiwaną 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

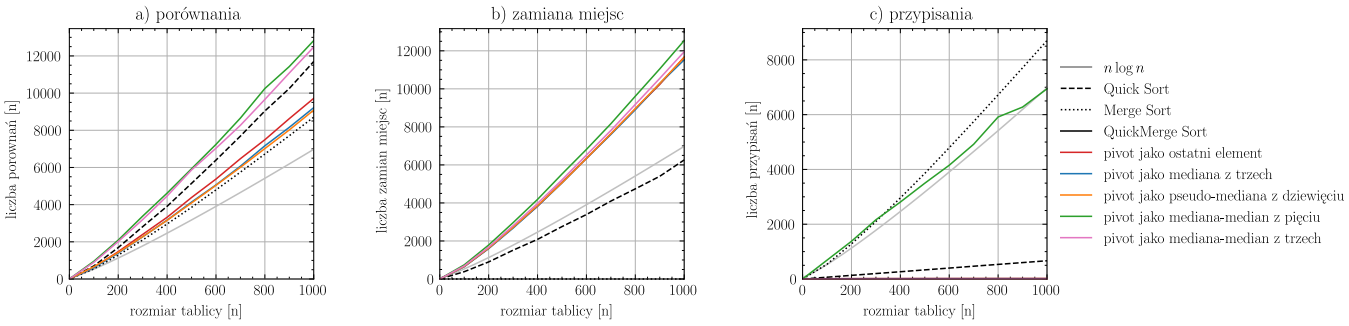
Tak jak w przypadku algorytmów deterministycznych, algorytmy randomizowane poddano analizie pod kątem liczby wykonywanych operacji atomowych, z podziałem na operacje porównania, zamiany miejsc oraz przypisania. Dodatkowo badano łączny koszt wykonanych operacji jako sumę ważoną liczby operacji atomowych, z uwzględnieniem wartości współczynnika kosztu. Badając liczbę poszczególnych operacji uwzględniono klasyczny algorytm Merge Sort oraz wersję algorytmu Quick Sort z partycjonowaniem metodą Lemuto oraz wyborem piwota jako losowy element tablicy.

Badając liczbę wykonanych operacji porównania (4.14) można zauważyć, 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 Lemuto 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.

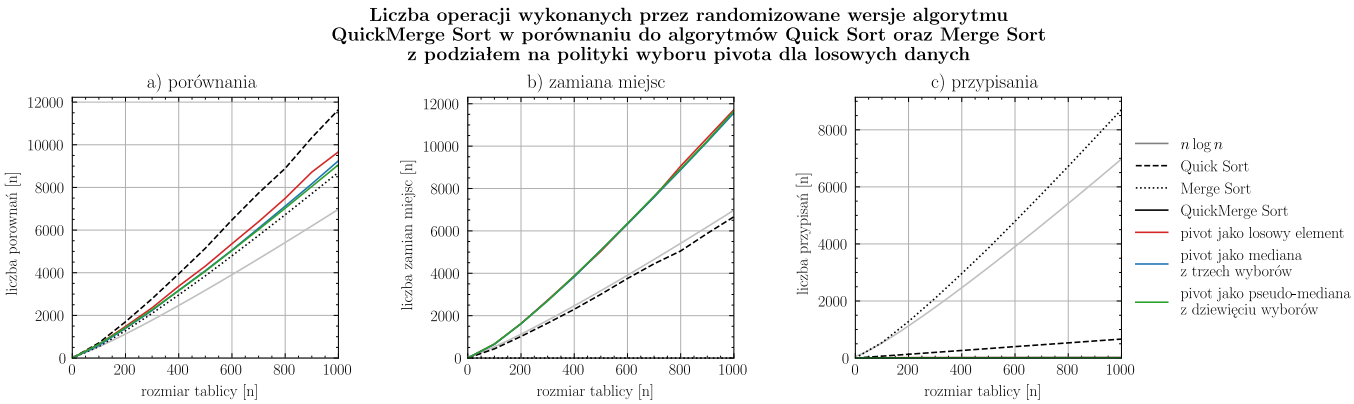
Analizując liczbę operacji zamiany miejsc oraz przypisania (4.14) można stwierdzić, że niezależnie od strategii wyboru piwota, algorytm QuickMerge Sort wykonuje około dwa razy więcej operacji zamiany miejsc niż randomizowany algorytm Quick Sort. Co więcej, żadna z randomizowanych implementacji nie wykonuje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



Rysunek 4.12



Rysunek 4.13



Rysunek 4.14

Podobnie jak w przypadku algorytmów deterministycznych, badając łączny koszt wykonanych operacji (4.16) można zauważyć, że dla danych typu podstawowego ($\boldsymbol{\alpha = 1.0}$), najwydajniejszym algorytmem sortującym działającym w miejscu jest randomizowany algorytm Quick Sort. W przypadku złożonych struktur danych ($\boldsymbol{\alpha \geqslant 6.0}$), wydajniejszym podejściem staje się randomizowany algorytm QuickMerge Sort, jednak tylko dla strategii wyboru piwota jako mediana z trzech losowych elementów oraz pseudo-mediana z dziewięciu losowych elementów. Dla bardzo złożonych danych ($\boldsymbol{\alpha \geqslant 8.0}$) algorytm QuickMerge Sort z wyborem piwota jako losowy element tablicy również jest wydajniejszy od randomizowanego algorytmu Quick Sort. 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 rozkładu prawdopodobieństwa liczby wykonanych operacji (4.15) użyto tablicy losowych danych o stałym rozmiarze $\boldsymbol{n = 1000}$ oraz współczynnika kosztu o stałej wartości $\boldsymbol{\alpha = 1.0}$. Można zauważyć, że dla danych tupu podstawowego, wartość oczekiwana liczby 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 oczekiwaną liczby wykonanych operacji ma algorytm Quick Sort.

4.4.4 Wnioski

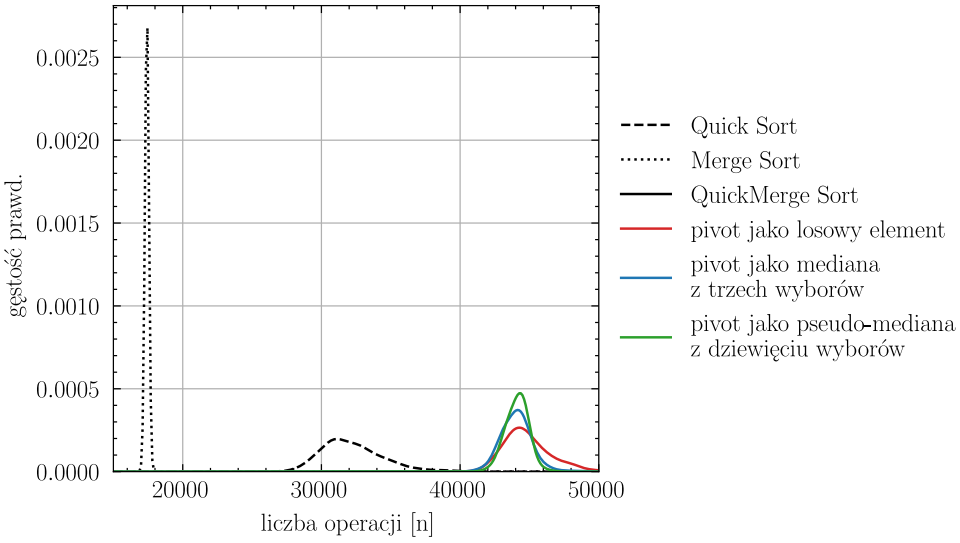
TODO wnioski Z analizy porównawczej wynika, że dla struktór alfa ¿ 7.0 QuickMerge Sort jest wydajniejsze niż Quick Sort.

Z przeprowadzonej analizy wynika, że dla złożonych struktor wejściowych, stosowanie algorytmu QuickMerge Sort jest wydajniejsze od klasycznego podejścia Quick Sort. Najwydajniejszym z badanych algorytmów jest ...

Analiza algorytmow deterministycznych - najblizej asymptoteycznie do merge sort jest lemuto z pseudomediana z dziewieciu

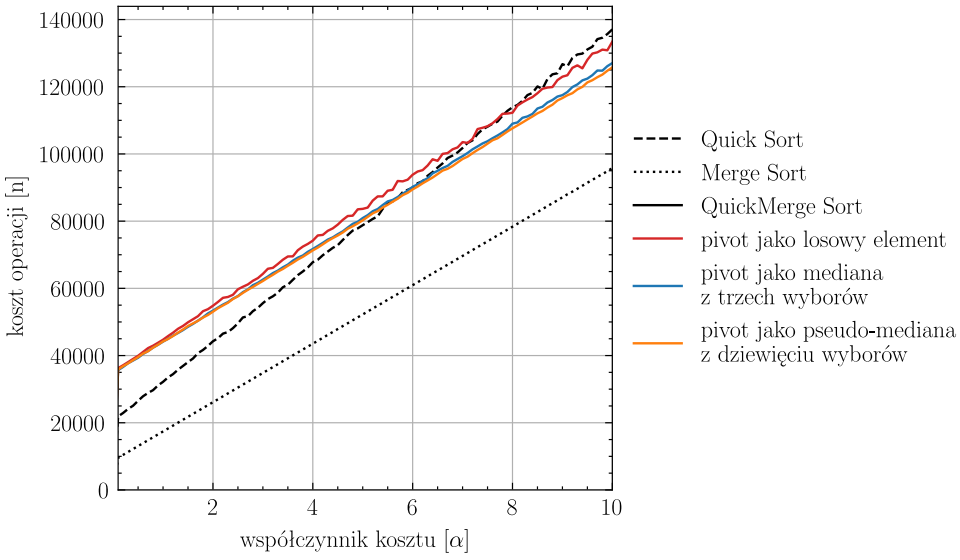
że dla dowolnych randomizowanych strategii wyboru pivota, partycjonowanie metodą Hoare wykonuje większą liczbę operacji porównania oraz mniejszą liczbę operacji zamiany miejsc. Tak jak w przypadku algorytmów deterministycznych, czynnik ten może okazać się istotny w przypadku sortowania złożonych struktur.

Rozkład prawdopodobieństwa liczby operacji wykonanych przez niedeterministyczne 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

4.5 Intro Sort

Głównym problemem wynikającym ze stosowania algorytmu Quick Sort jest jego słaba pesymistyczna złożoność czasowa. Dla uporządkowanych danych wejściowych, głębokość drzewa wywołań rekurencyjnych jest równa rozmiarowi tablicy wejściowej, zaś sam algorytm działa w czasie równym $\boldsymbol{O(n^2)}$. Dodatkowo, dla tablicy małych rozmiarów, algorytm Quick Sort powoduje niepotrzebny nakład czasowy, związany ze wstępnym 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 przekroczono maksymalną głębokość drzewa wywołań rekurencyjnych, łańcuch wywołań zostaje zakończony, zaś kolekcja zostaje posortowana metodą Heap Sort. Początkowa maksymalna głębokość drzewa wywołań rekurencyjnych jest wyznaczona wzorem $\boldsymbol{2 \log n}$. Ponieważ metoda Heap Sort działa z czasem $\boldsymbol{O(n \log n)}$ nawet w przypadku pesymistycznym, algorytm Intro Sort ma złożoność $\boldsymbol{O(n \log n)}$, 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 standardowej funkcji sortującej **std::sort** wewnątrz kompilatora gcc [1].

4.5.1 Eksperymentalne wyznaczanie punktu granicznego

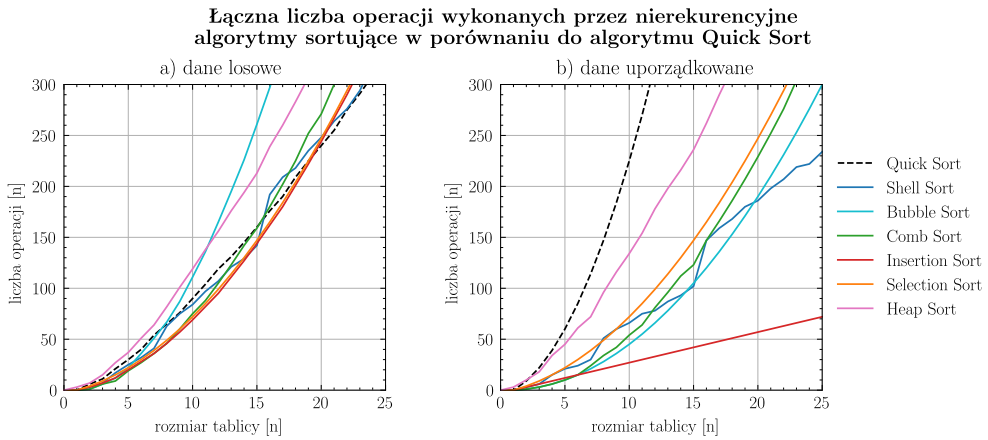
Stosowanie rekurencyjnych metod sortujących, wykorzystujących technikę dziel i zwyciężaj, wiąże się z dodatkowym nakładem czasowym spowodowanym koniecznością odpowiedniego przygotowania kolekcji przed jej posortowaniem. W przypadku algorytmu Quick Sort, dodatkowy nakład powoduje metoda partycjonowania. 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 zauważyć, 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 przekroczono 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.

[1] Dokumentacja 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:         arr_1, arr_2 ← Partition(arr)
11:         IntroSort(arr_1, depth-1)
12:         IntroSort(arr_2, depth-1)
13:     end if
14:
15: end procedure
```

4.5.3 Analiza deterministycznych wersji algorytmu Intro Sort

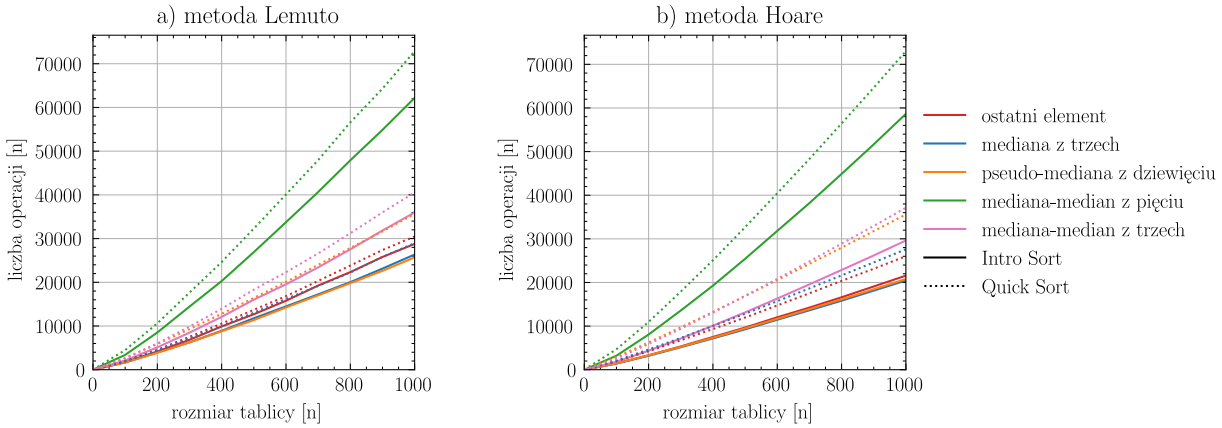
Wykresy liczby wykonywanych operacji w porównaniu do algorytmów bazowych. Wykresy gęstości liczby wykonywanych operacji dla stałej liczby n, np n = 10000. Wykresy dla różnych algorytmów partycjonowania.

4.5.4 Analiza randomizowanych wersji algorytmu Intro Sort

4.5.5 Wnioski

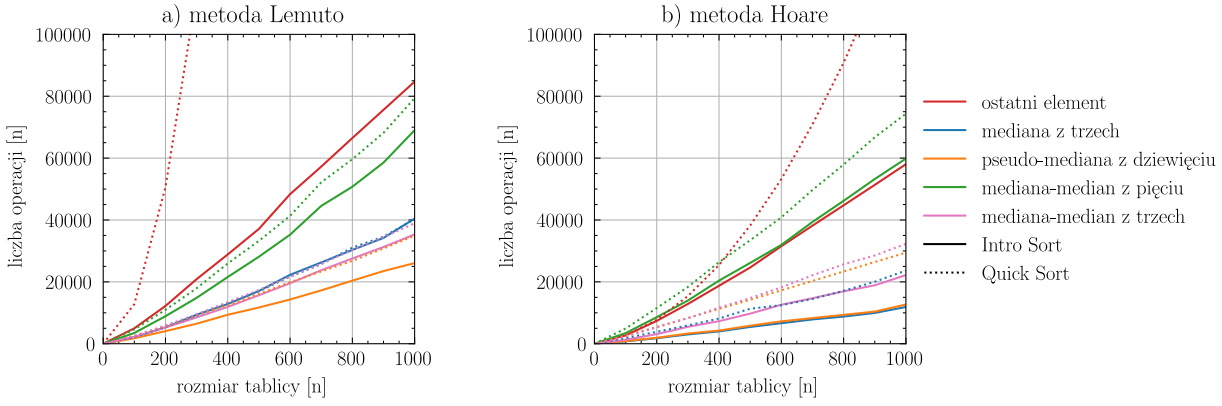
Wyniki analizy porównawczej

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

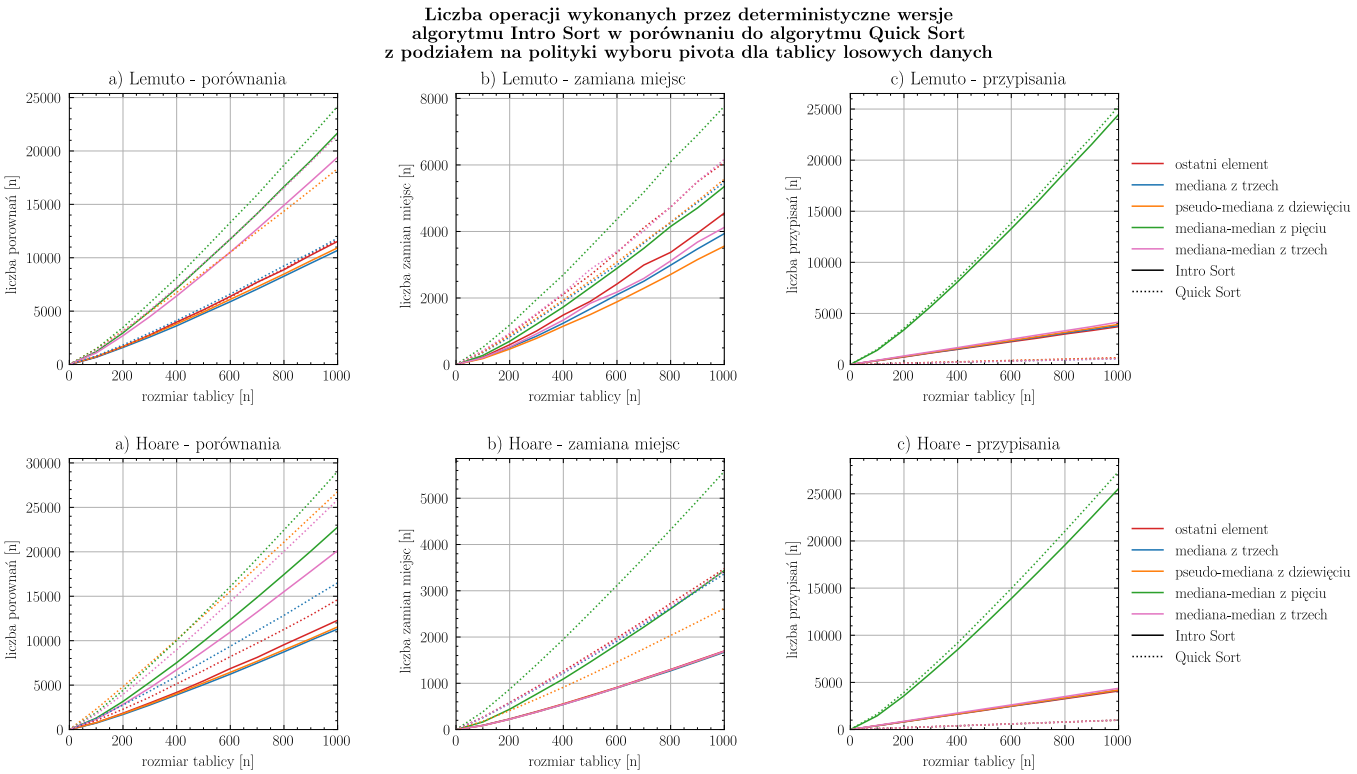


Rysunek 4.18

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


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
a) Lemuto - porównania
b) Lemuto - zamiana miejsc
c) Lemuto - przypisania
a) Hoare - porównania
b) Hoare - zamiana miejsc
c) Hoare - przypisania
liczba porównań [n]
liczba zamian miejsc [n]
liczba przypisań [n]
rozmiar tablicy [n]
ostatni element
mediana z trzech
pseudo-mediana z dziewięciu
mediana-median z pięciu
mediana-median z trzech
Intro Sort
Quick Sort

Rysunek 4.20

Implementacja systemu

5.1 Struktura systemu

Aplikacja składa się z dwóch modułów: silnika testującego oraz silnika graficznego. Działanie systemu jest określane 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 utrwala 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 Python [2] przy użyciu biblioteki matplotlib [3].

5.2 Koncepcje architektury silnika testującego

5.2.1 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 partycjonowanie 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.

5.2.2 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.

[1] Dokumentacja języka C++: https://en.cppreference.com
[2] Dokumentacja języka Python: https://docs.python.org/3/
[3] Dokumentacja biblioteki matplotlib: https://matplotlib.org/

5.2.3 Bezstanowość

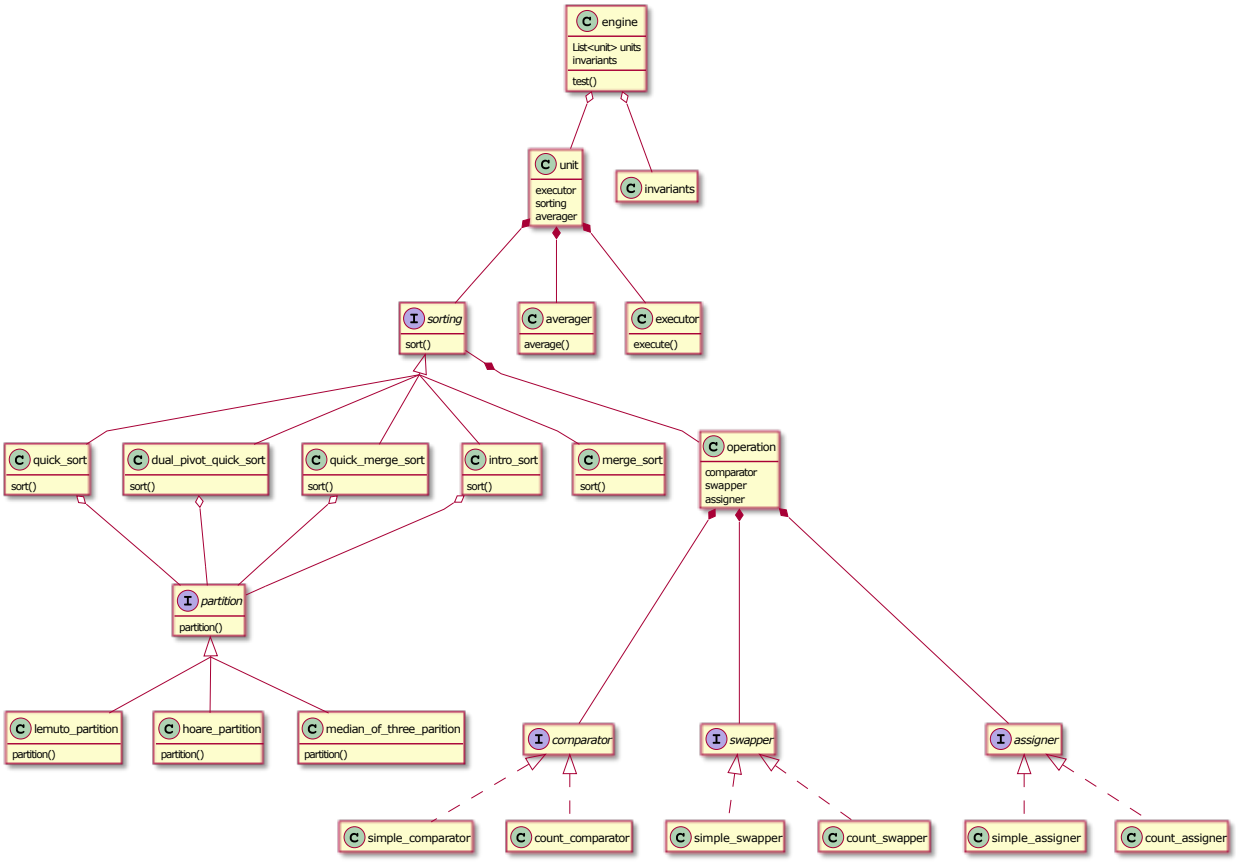
Powszechnym problemem w programowaniu obiektowym jest przechowywanie stanu. Problem ten wynika po części z praktyki hermetyzacji danych wewnątrz obiektowej abstrakcji. Użytkownik zewnętrzny korzystając z interfejsu danego komponentu nie ma dostępu do procesów zachodzących w jego wnętrzu. Może to prowadzić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

5.3 Model aplikacji

5.3.1 Diagram klas

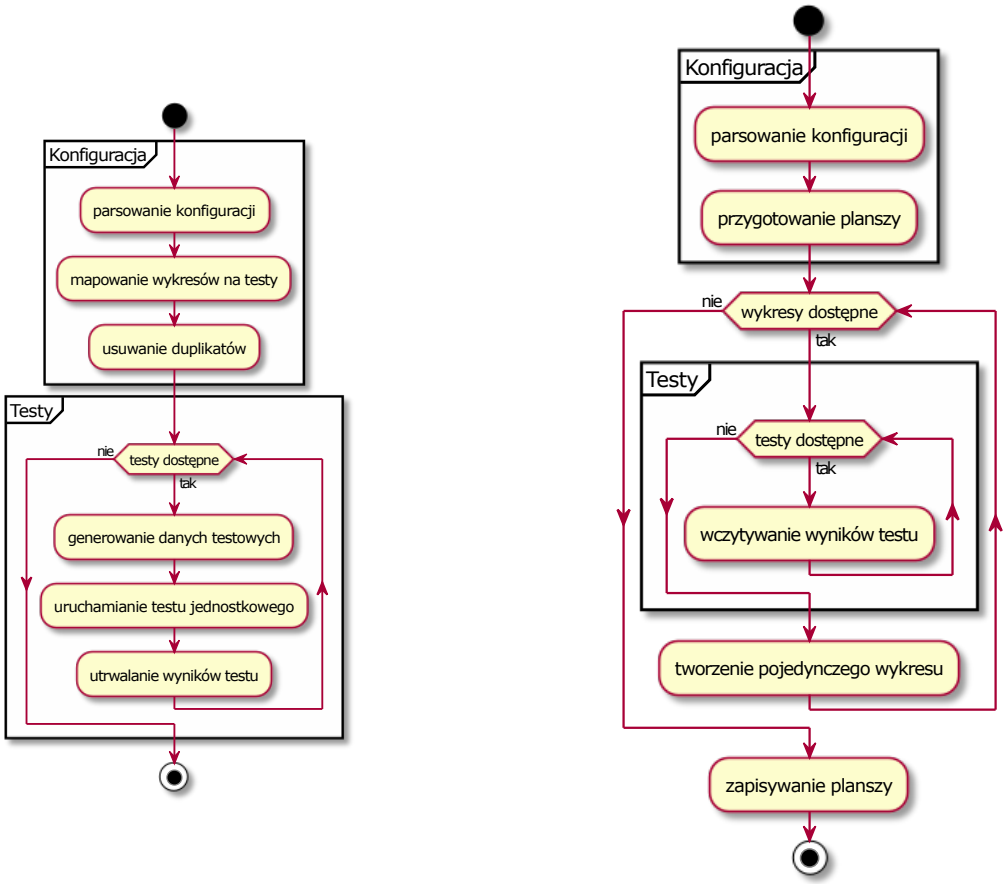
TODO: opis diagramu



Rysunek 5.1: Diagram klas silnika testującego

5.3.2 Diagram aktywności

TODO: opis diagramu



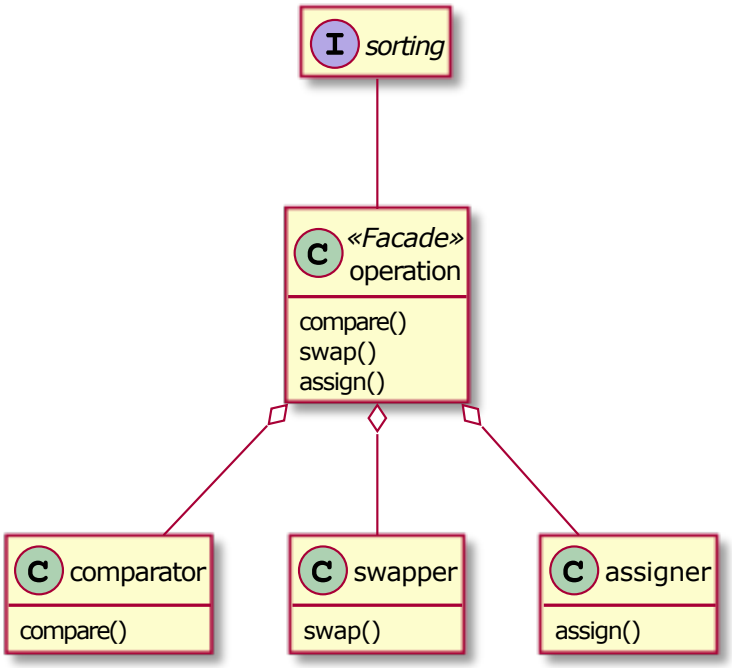
(a) silnik testujący (b) silnik graficzny

Rysunek 5.2: Diagram aktywności projektowanego systemu

5.4 Wzorce projektowe

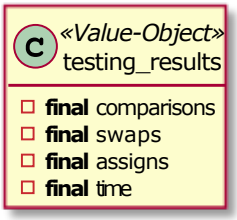
5.4.1 Fasada

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.



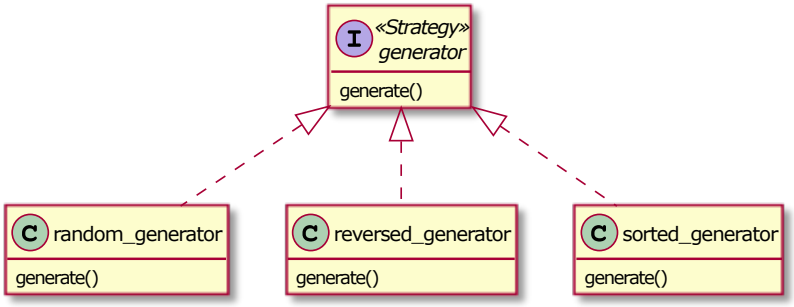
5.4.2 Obiekt-Wartość

Proces testowania algorytmu składa się z wielu iteracji. Każdy z atomowych testów wchodzących w skład iteracji powinien być całkowicie niezależny i odseparowana 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.



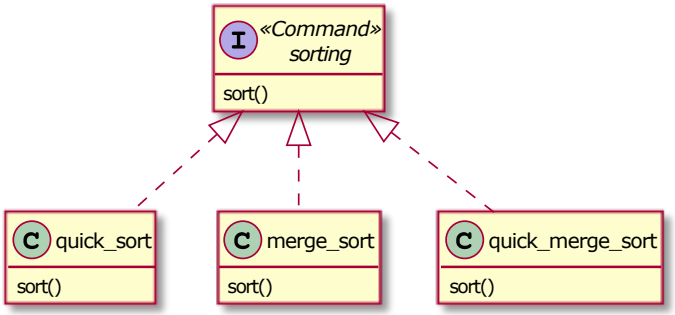
5.4.3 Strategia

Aby wymusić działanie algorytmu w przypadku optymistycznym, pesymistycznym oraz średnim konieczne jest przygotowanie spreparowanych 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.



5.4.4 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.



Podsumowanie

Podsumowanie najlepszych algorytmów działających w miejscu Podsumowanie wyników testowania algorytmów. Wnioski z analizy algorytmów hybrydowych. Wybrac najlepsze z badanych metod sortujacych z podziałem na 4 kategorie:

- dane losowe, typ podstawowy - 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.

Bibliografia

Słownik pojęć

A.1 Notacja O())

A.2 Algorytm działający w miejscu

A.3 Algorytm stabilny

Algorytm nie zamienia kolejnością elementów o tej samej wartości.

Środowisko uruchomieniowe aplikacji

Platforma uruchomieniowa aplikacji - Windows. Wykorzystane języki programowania - C++, Python.

B.1 Zmienne środowiskowe

Opis zmiennych środowiskowych TEST-DIRECTORY, CONFIG-DIRECTORY, PLOT-DIRECTORY.

B.2 Biblioteki zewnętrzne

Biblioteki w C++ oraz Pythonie potrzebne do uruchomienia aplikacji wraz z numerami wersji.

B.3 Instalowanie aplikacji

Uruchamianie skryptu zaciągającego potrzebne zależności. Uruchamianie pliku makefile kompilującego i instalującego aplikację. Cykl pracy programu - tworzenie konfiguracji, testowanie silnikiem testującym, wizualizacja wyników przy użycia silnika graficznego.

B.4 Przykładowy plik konfiguracyjny

Plik konfiguracyjny w jsonie. Omówienie pliku, na początku są dane współdzielone przez wszystkie testy. Potem plik zawiera listę elementów typu plot, czyli listę osobnych testów.