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.

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

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

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

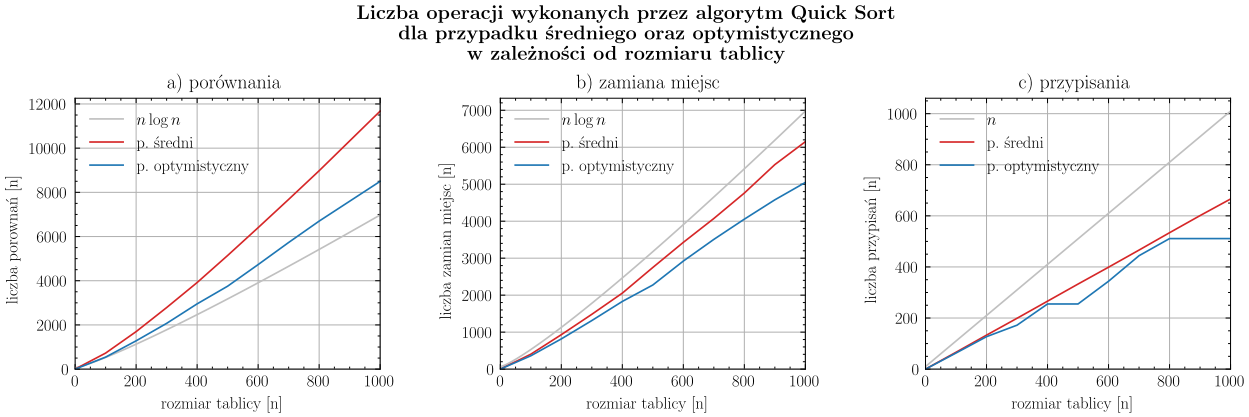
[2] Dokumentacja funkcji sortującej qsort: https://en.cppreference.com/w/cpp/algorithm/qsort

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 $\boldsymbol{O(n^2)}$.

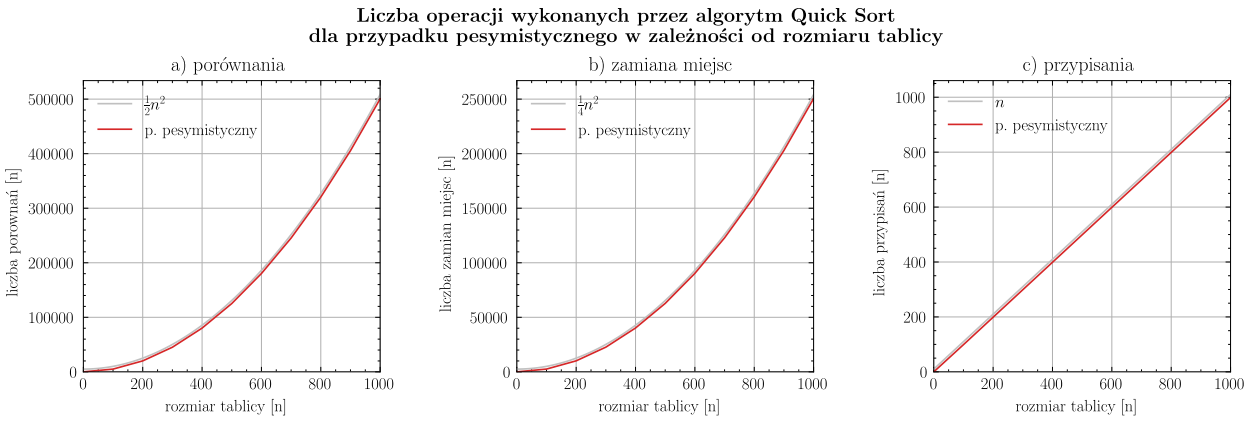
Przypadek średni (3.1) został zbadany wprowadzając losowe dane z powtórzeniami oraz stosując algorytm wybierający pivot jako ostatni element tablicy. Analiza eksperymentalna wykazała, że w przypadku średnim algorytm Quick Sort ma złożoność czasową równą $\boldsymbol{O(n \log n)}$, a więc jest tego samego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

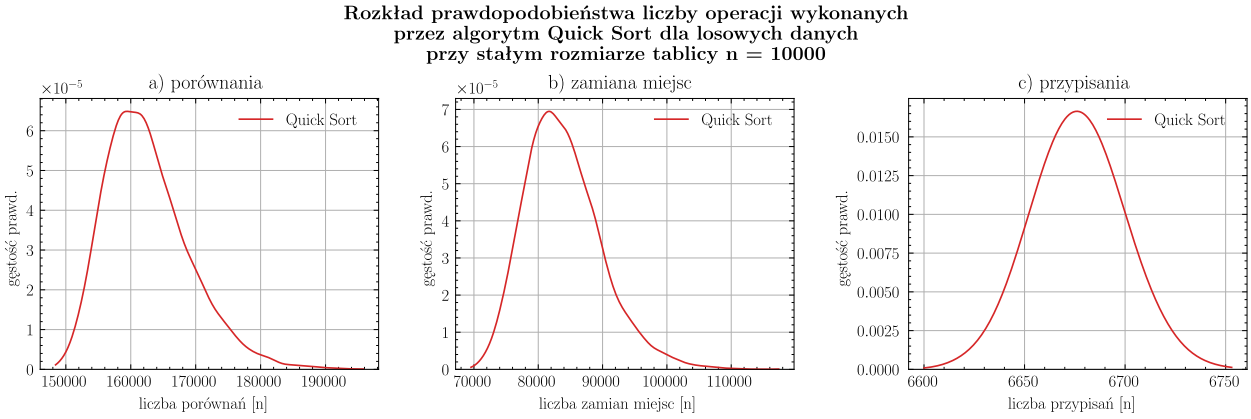
Analizując rozkład prawdopodobieństwa liczby wykonanych operacji (3.3) użyto tablicy losowych danych o stałym rozmiarze $\boldsymbol{n = 10000}$. Można zauważyć, że liczba operacji porównania oraz liczba operacji zamiany miejsc nie są przedstawiane za pomocą rozkładu normalnego. Bardziej prawdopodobne 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 $\boldsymbol{O(n \log n)}$.

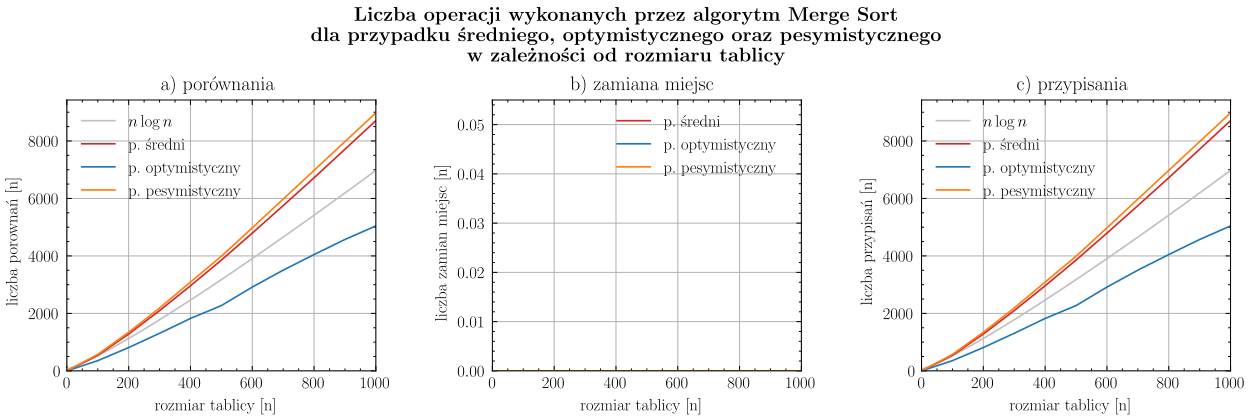
Przypadek pesymistyczny (3.4) zachodzi, gdy liczba porównać między sobą elementów w trakcie scalania jest możliwie największa. Efekt ten uzyskano, preparując dane wejściowe w taki sposób, aby w każdym kroku scalania porównywane tablice zawierały elementy na przemian większe oraz mniejsze. Z analizy wynika, 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 $\boldsymbol{O(n \log n)}$.

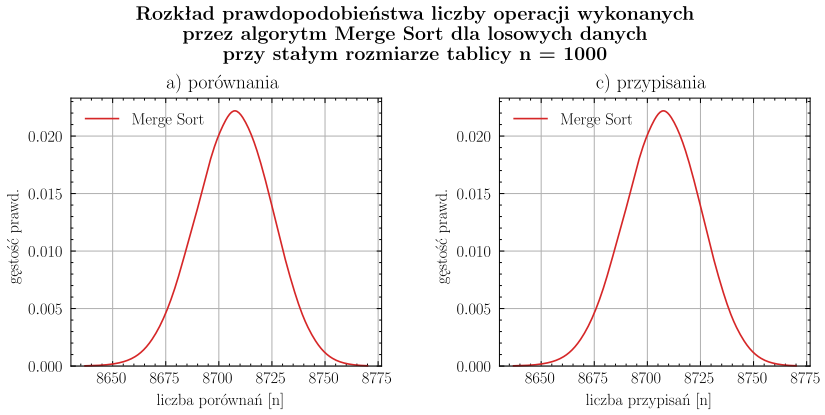
W przeciwieństwie do algorytmu Quick Sort, algorytm Merge Sort wykonuje dokładnie tyle samo operacji porównania co operacji przypisania. Jest to zgodne z rzeczywistością, ponieważ w trakcie scalania każdej operacji porównania towarzyszy przypisanie wartości do bufora. Można zauważyć, że algorytm Merge 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 łączną liczbę wykonanych operacji jako sumę ważoną liczby operacji atomowych.

Przy losowych danych wejściowych (4.1), czyli przypadku średniego w klasycznym algorytmie Quick Sort, 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.

Przy danych wejściowych posortowanych w odwrotnej kolejności (4.2), czyli dla przypadku pesymistycznego w klasycznym algorytmie Quick Sort, 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.

Porównując liczbę wykonanych operacji pomiędzy badanymi algorytmami partycjonowania (4.4) 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, dla których czas porównania dwóch elementów jest znacznie dłuższy.

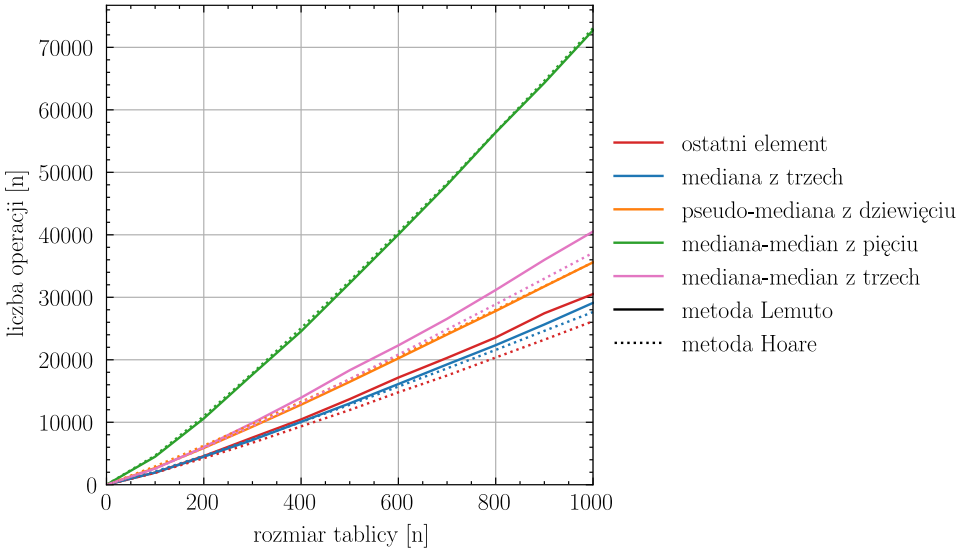
Do analizy rozkładu prawdopodobieństwa liczby wykonanych operacji (4.3) użyto tablicy losowych danych o stałym rozmiarze $\boldsymbol{n = 1000}$. Dla badanych algorytmów najmniejsze odchylenie standardowe mają algorytmy partycjonowania metodą Hoare przy wyborze pivota jako pseudo-mediana z dziewięciu 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 badanych implementacjach algorytmów z rodziny Quick Sort, partycjonowanie metodą Hoare okazało się wydajniejsze niż partycjonowanie metodą Lemuto. Ponieważ partycjonowanie metodą Hoare wykonuje większą liczbę operacji porównania niż partycjonowanie metodą Lemuto, algorytm ten nadaje się do sortowania kolekcji zawierającej dane prymitywne, jednak w przypadku sortowania bardziej złożonych struktur metoda ta może okazać się mniej wydajna.

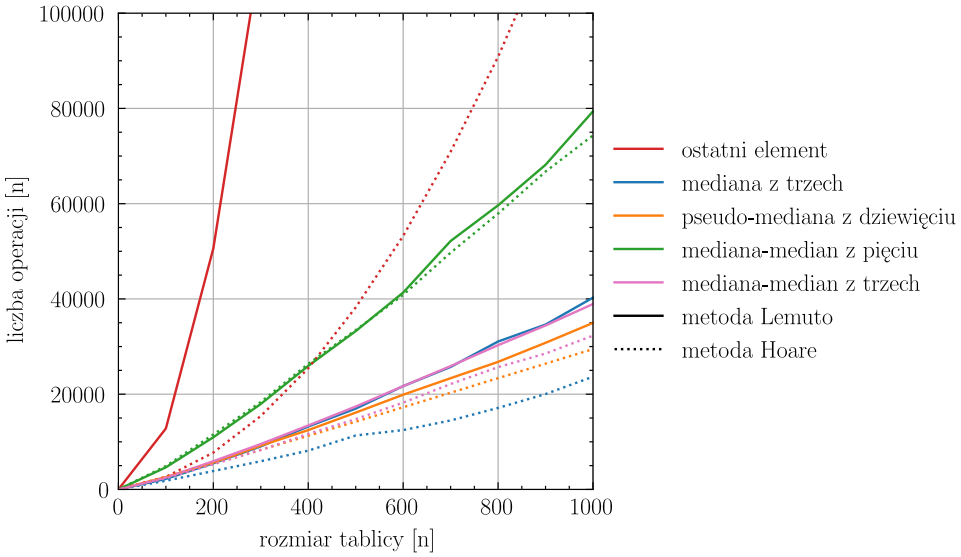
Analizując zachowanie algorytmów można stwierdzić, że dla danych posortowanych lub prawie posortowanych, dobrym wyborem jest skorzystanie z kosztownej metody wyszukiwania pivota, która pomimo swojego dodatkowego nakładu czasowego zwiększa prawdopodobieństwo na znalezienie dobrego pivota. Wyjątkiem jest 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.

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

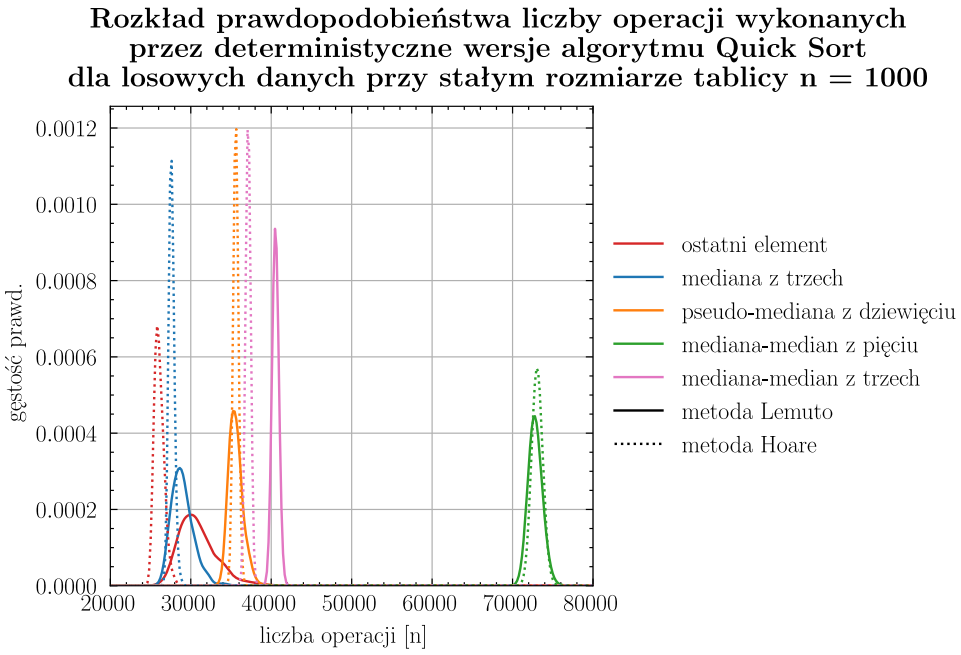


Rysunek 4.1

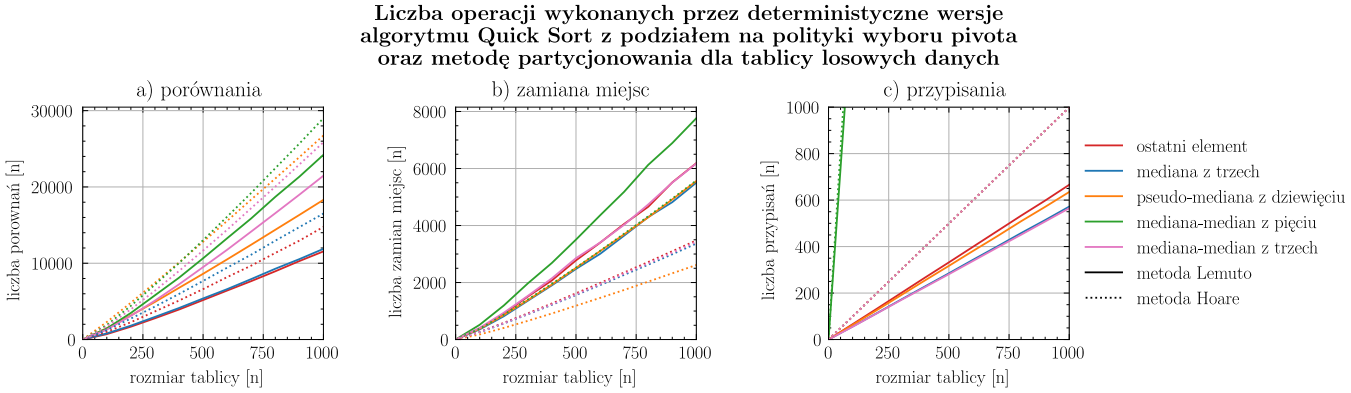
Łączna liczba operacji wykonanych przez deterministyczne wersje algorytmu Quick Sort z podziałem na polityki wyboru pivota oraz metodę partycjonowania dla tablicy danych posortowanych odwrotnie



Rysunek 4.2



Rysunek 4.3



Rysunek 4.4

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 łączną liczbę wykonanych operacji jako sumę ważoną liczby operacji atomowych.

Przy losowych danych wejściowych (4.5) 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.

Przy danych wejściowych posortowanych w odwrotnej kolejności (4.6), czyli dla przypadku pesymistycznego w klasycznym algorytmie Quick Sort, 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.

Porównując liczbę wykonanych operacji (4.8) 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.

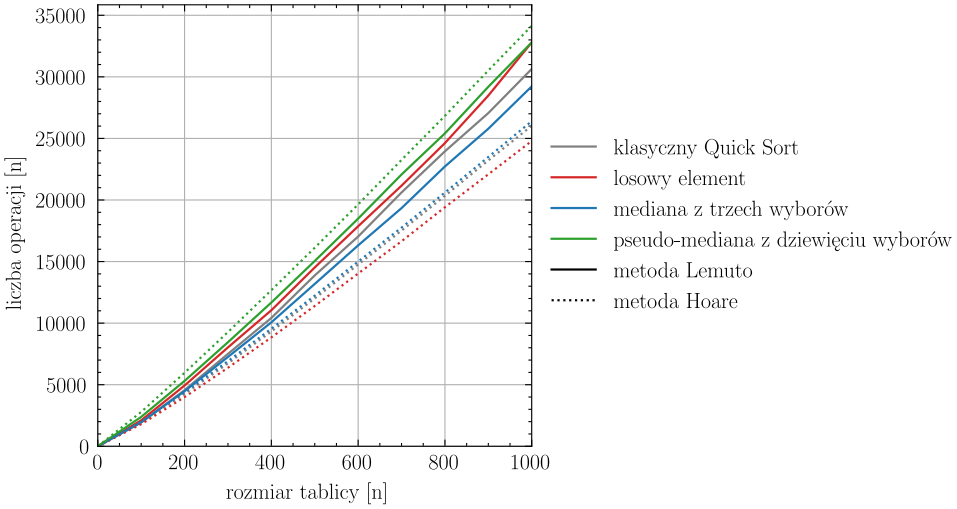
Badając rozkład prawdopodobieństwa liczby wykonanych operacji dla algorytmów randomizowanych (4.7),

4.3.2 Wnioski

4.4 QuickMerge Sort

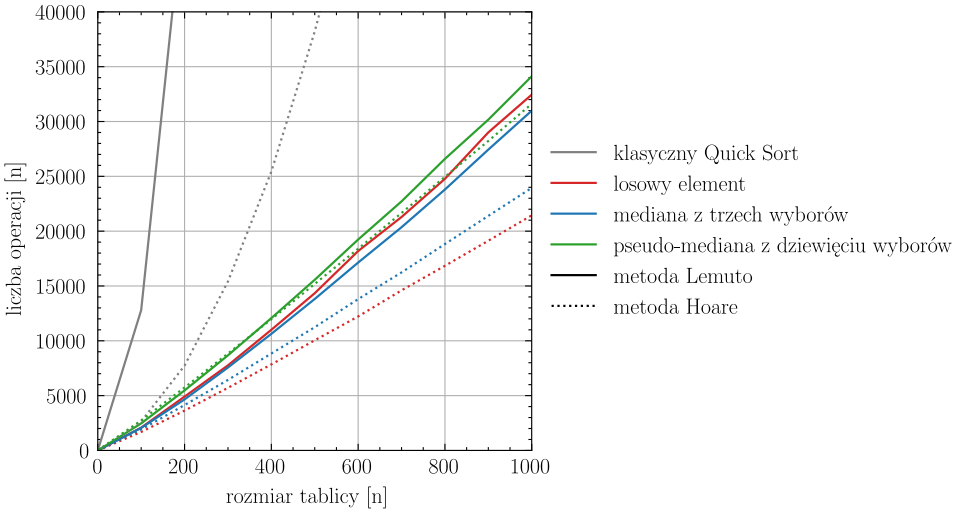
Koncepcja algorytmu, mocne strony (Merge Sort bez konieczności alokacji pamięci)

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

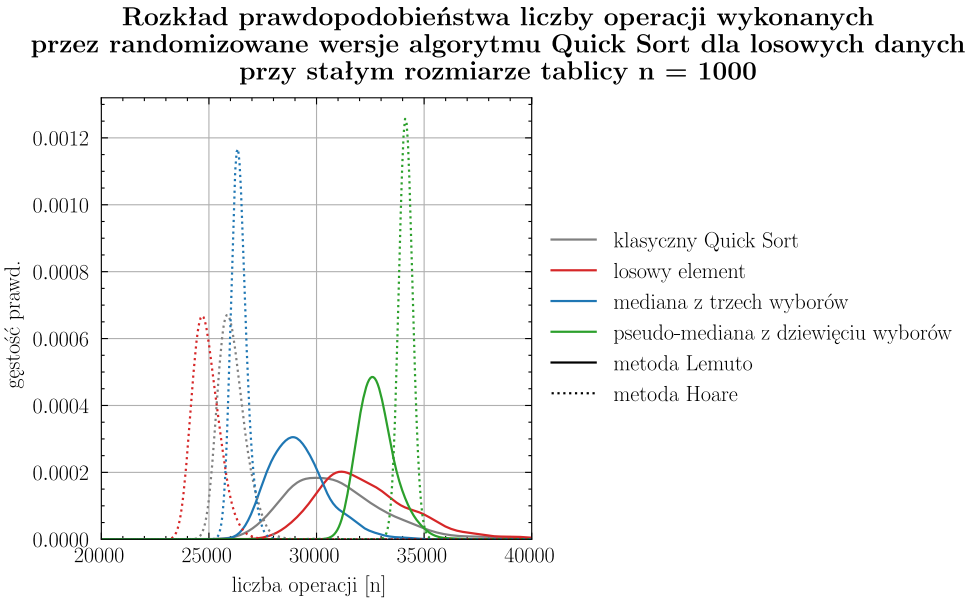


Rysunek 4.5

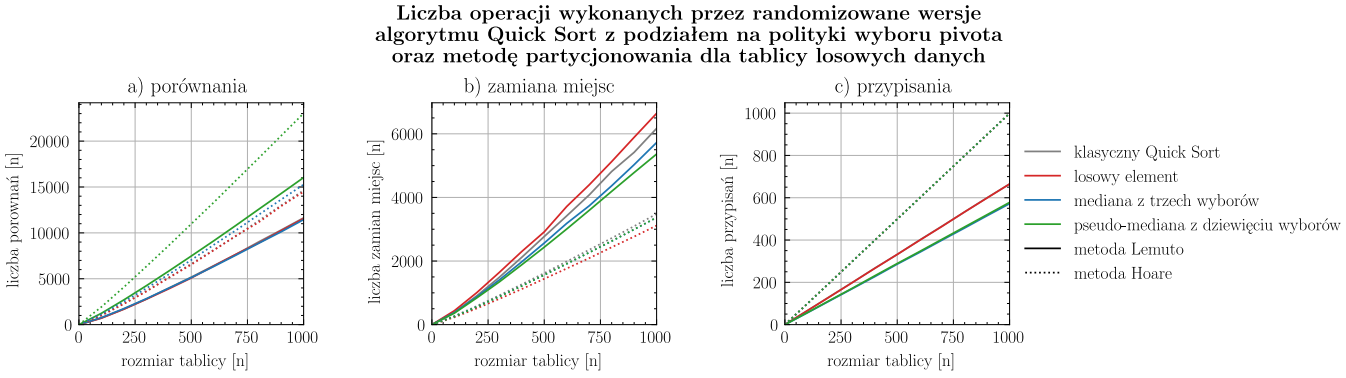
Łączna liczba operacji wykonanych przez randomizowane wersje algorytmu Quick Sort z podziałem na polityki wyboru pivota oraz metodę partycjonowania dla tablicy danych posortowanych odwrotnie



Rysunek 4.6



Rysunek 4.7



Rysunek 4.8

4.4.1 Pseudokod

4.4.2 Analiza eksperymentalna algorytmu

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

4.4.3 Wnioski

Wyniki analizy porównawczej

4.5 Intro Sort

Ogólny opis algorytmu, gdzie jest wykorzystywany (std::sort w g++), zalety.

4.5.1 Pseudokod

4.5.2 Analiza eksperymentalna algorytmu

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.

4.5.3 Wnioski

Wyniki analizy porównawczej

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

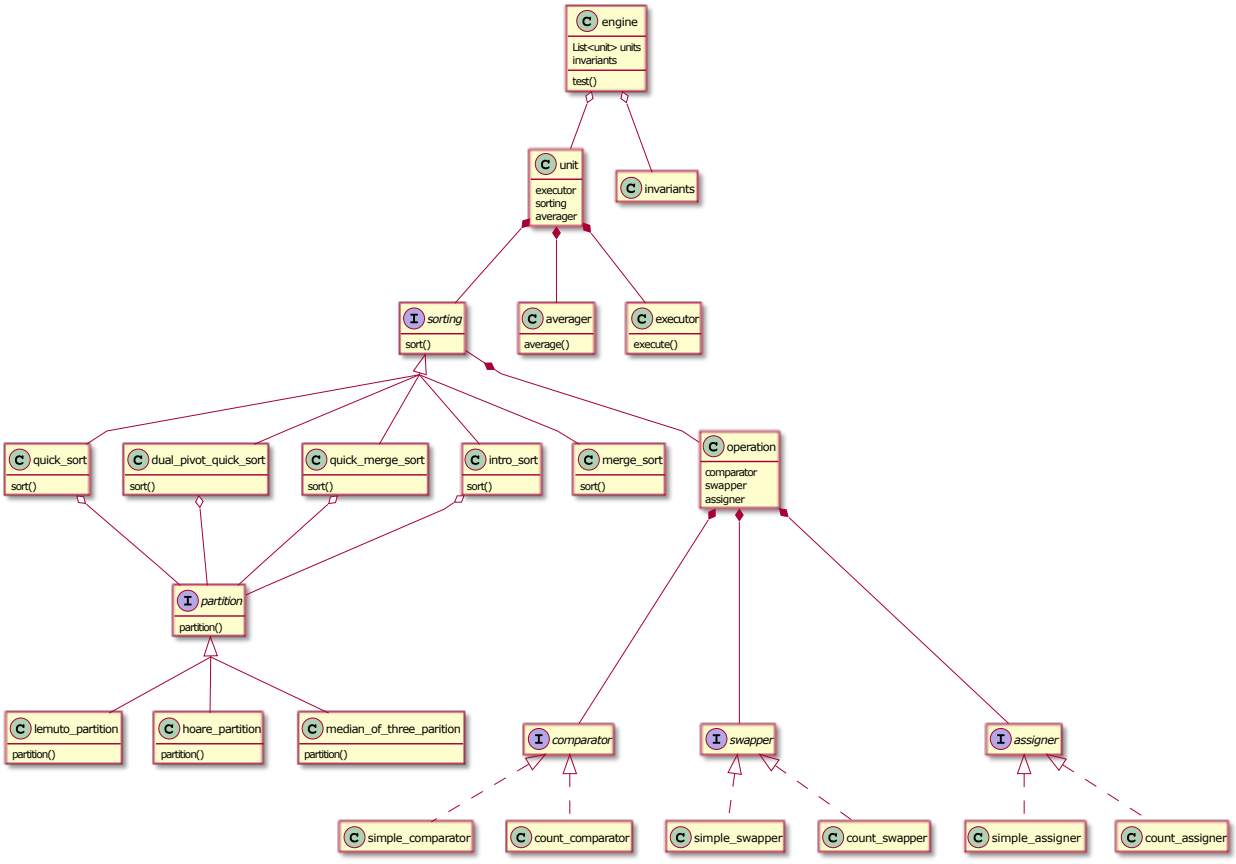
Powszechnym problemem w programowaniu obiektowym jest przechowywanie stanu. Problem ten wynika po części z praktyki hermetyzacji danych wewnątrz obiektowej abstrakcji. Użytkownik zewnętrzny korzystając z interfejsu danego komponentu nie ma dostępu do procesów zachodzących w jego wnętrzu. Może to prowadzić 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

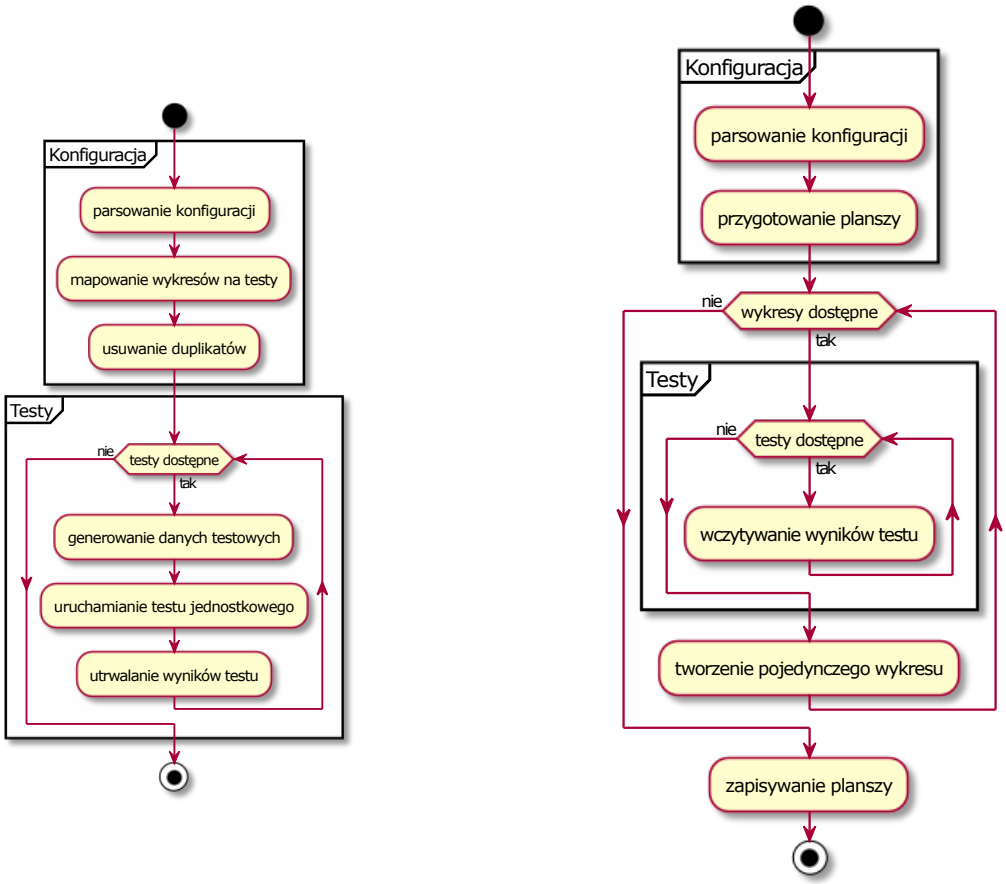
TODO: opis diagramu



Rysunek 5.1: Diagram klas silnika testującego

5.3.2 Diagram aktywności

TODO: opis diagramu



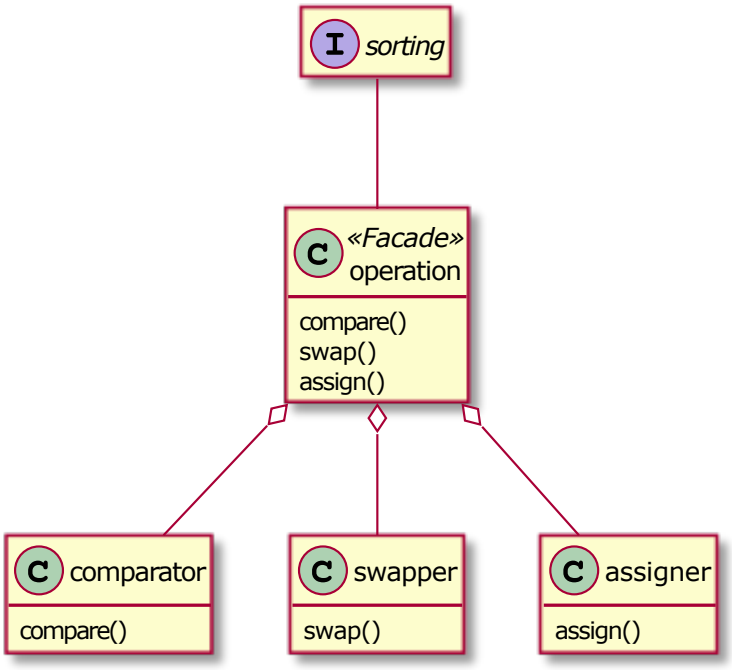
(a) silnik testujący (b) silnik graficzny

Rysunek 5.2: Diagram aktywności projektowanego systemu

5.4 Wzorce projektowe

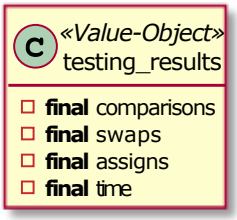
5.4.1 Fasada

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.



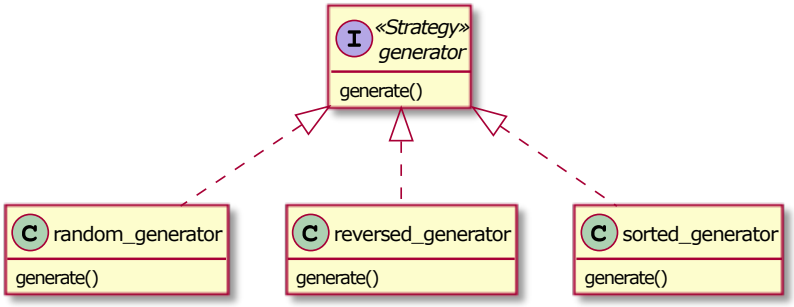
5.4.2 Obiekt-Wartość

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.



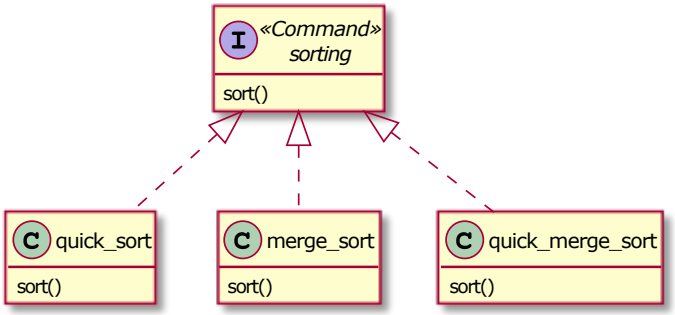
5.4.3 Strategia

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 wyników testowania algorytmów. Wnioski z analizy algorytmów hybrydowych.

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.