Politechnika Wrocławska

Projektowanie i analiza algorytmów: [W04ISA-SI0013G]

Projekt 1 – Algorytmy sortowania

Wykonał:
Michał Białek
Numer indeksu: 264285
Wydział Informatyki i Telekomunikacji
Rok akademicki: 2023/2024 - 4 semestr
Kierunek: Informatyczne Systemy Automatyki

Spis treści:

1.Cel projektu

Celem projektu jest zdobycie wiedzy na temat zasad działania poszczególnych algorytmów sortowania, oraz badanie ich złożoności czasowej oraz pamięciowej, dla najgorszego oraz średniego przypadku.

2.Wprowadzenie ogólne

Do tworzenia wydajnych i dobrze działających programów kluczową umiejętnością jest odpowiedni dobór algorytmów oraz struktur danych. Nieodpowiedni dobór struktur danych jak i również algorytmów może prowadzić do znacznego zwiększenia czasu oraz/lub zasobów potrzebnych na rozwiązanie danego problemu.

W poniższej pracy będziemy się skupiać na 3 algorytmach sortowania:

1. Sortowanie przez scalanie (merge sort)
2. Sortowanie szybkie (qucik sort)
3. Sortowanie kubełkowe (bucket sort)

Każdy z tych algorytmów ma odmienną zasadę działania i charakteryzuje się odmiennymi parametrami i zastosowaniem.

2.1 Wprowadzenie do struktury danych w języku Java

Znając specyfikę elementów, na których będziemy operować dobieramy odpowiednie struktury danych w których będziemy te elementy przechowywać. Każda struktura cechuje się specyficznymi własnościami takimi jak:

- Typ przechowywanych elementów
- Dostępne operacje na strukturze
- Efektywność czasowa oraz pamięciowa (szybkość odczytu, zapisu, modyfikacji)
- Złożoność algorytmiczna
- Zachowywana kolejność, przechowywanie duplikatów

W języku Java rozróżniamy poniższe kolekcje:

1. Set (Zbiory)
    a. HashSet
    b. LinkedHashSet
    c. TreeSet
2. Kolejki
    a. PriorityQuueue
    b. ArrayDeque
    c. LinkedList
3. Listy
    a. ArrayList
    b. LinkedList
    c. Starsze, współcześnie nie używane:Vector oraz Stack

Znając zasady działań poszczególnych metod danych struktur, oraz charakterystykę danych, należy wybrać najlepszą strukturę.

2.2 Algorytmy – wprowadzenie

Co to jest algorytm ?

Algorytm jest to zestaw instrukcji lub reguł, wykonywanych krok po roku, w celu rozwiązania problemu. W tym projekcie skupimy się na badaniu algorytmów sortowania elementów struktury, którego celem jest posortowanie elementów, w określonej kolejności według zdefiniowanej cechy (w naszym przypadku do wyboru według oceny, lub według tytułu alfabetycznie).

3.Pomiary działania algorytmów

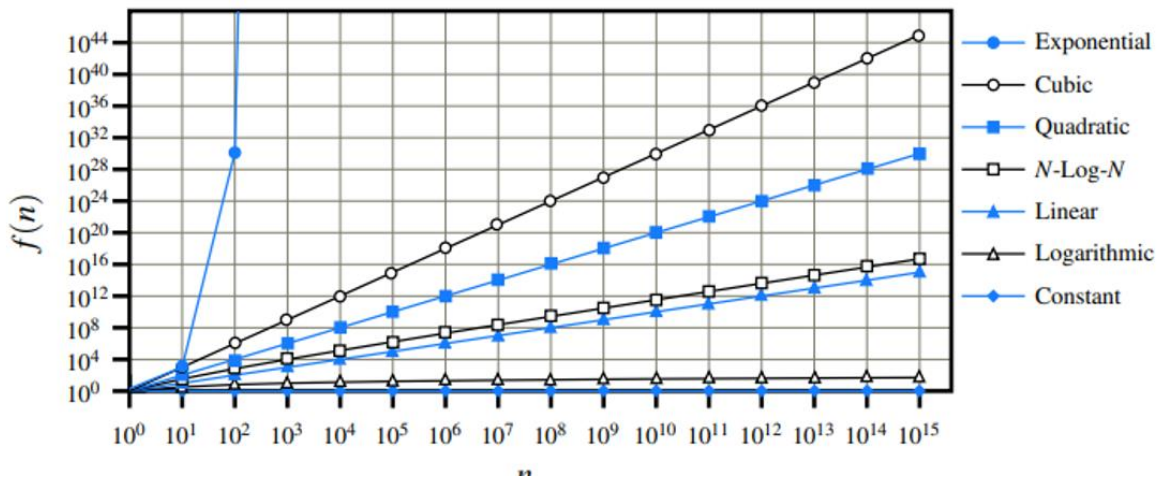
Do pomiaru efektywności algorytmu stosujemy analizę asymptotyczną, która polega na badaniu czasu wykonania w zależności od ilości elementów wejściowych (analiza czasowa) oraz badaniu zużywanych zasobów w zależności od ilości elementów wejściowych (analiza przestrzenna).

Do analizy asymptotycznej stosujemy notacje:

- Dużego O (Big O notation) – najczęściej używana notacja do określenia górnego ograniczenia czasu działania algorytmu
- Dużego Ω (Big-omega notation) – używana do określenia dolnego ograniczenia funkcji
- Theta – (Big-theta notation) – łączy oba ograniczenia, zarówno górne, jak i dolne

Na ich podstawie możemy wyróżnić kilka specyficznych funkcji asymptotycznych, których przebieg został zaprezentowany poniżej.

constant	logarithm	linear	*n*-log-*n*	quadratic	cubic	exponential
1	$\log n$	n	$n \log n$	n^2	n^3	a^n



Wykres 1 - Przebiegi specyficznych notacji asymptotycznych

Idealny algorytm powinien wykonać dane zadanie możliwie jak najszybciej, przy jednoczesnym najniższym zużyciu zasobów urządzenia.

Jednym ze sposób badania algorytmów sortowania jest badanie empiryczne, dla którego wyznaczamy najgorszy przypadek, który zwykle oznaczany jest jako T(n), oraz dla przykładu średniego, który wymaga pewnych założeń co do statystycznego rozłożenia danych. Nie bada się przypadku najlepszego z tego powodu, nie ma on wartości analitycznej.

Dodatkowo, każdy algorytm cechuje się stabilnością, lub niestabilnością. Stabilność algorytmu polega na odpowiedzi na niewielkie zmiany w danych wejściowych lub parametrach. Algorytm niestabilny może generować niedokładne wyniki obliczeń, wynikające z błędów zaokrągleń wyników pośrednich. Przykładem może być algorytm rozwiązujący równanie kwadratowe za pomocą „delty". Algorytmy stabilne eliminują błędy zaokrągleń, dzięki zwiększa się dokładność wyników końcowych.

4. Co wpływa na wynik badań ?

Podczas badań, trzeba mieć na uwadze czynniki zewnętrzne, które mogą wpływać na wyniki badań. Są to:

1. Czynniki sprzętowe:
    a. Pamięć – rozmiar oraz szybkość pamięci RAM mogą wpłynąć na czas wykonywania operacji.
    b. Procesor – częstotliwość taktowania procesora oraz liczba rdzeni, zjawisko cache false sharingu
2. Czynniki systemowe:
    a. Zarządzanie pamięcią przez system operacyjny
    b. Równoległe działanie innych procesów w tle
    c. Aplikacje zarządzające zasilaniem np. Dell power manager, który podczas działania na baterii może spowalniać działanie procesów
3. Implementacyjne
    a. Zastosowanie różnych języków programowania

4.1 Wykonywanie pomiarów w programach opratych o JVM

W przypadku języków opartych na JVM (Java Virtual Machine) dokonywanie precyzyjnych pomiarów jest obarczone kilkoma dodatkowymi problemami:

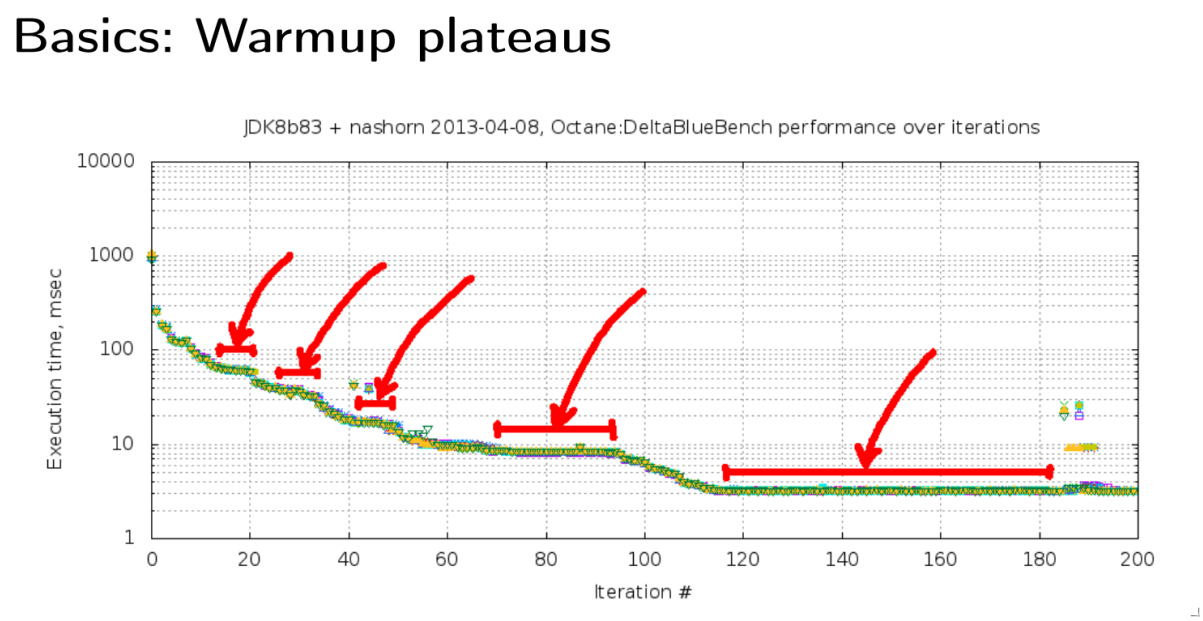
1. **Ziarnistość czasu** – metoda System.currentTimeMillis oraz System.nanoTime do pomiaru czasu cechuję się ziarnistością, która wynika z częstości aktualizacji czasu przez system operacyjny oraz przekazywania tego czasu do programu.
   W dokumentacji możemy przeczytać, że szczegółowość zwracanego czasu zależy od systemu operacyjnego, gdyż system operacyjny mierzy czas w jednostkach dziesiątek milisekund.
2. **Optymalizacje kompilatora**
   Współczesne kompilatory posiadają mechanizm optymalizujące, które potrafią bez wiedzy programisty zamienić kod, aby jego wykonanie było bardziej optymalne.

Przykładowymi zjawiskami optymalizacyjnymi, jakie stosuje kompilator to:

1. Rozwijanie pętli – loop unrolling
2. Łączenie pętli – loop fusion
3. Eliminacja martwego kodu - dead code elimination
4. Wyliczanie wartości stałych - constant folding

3.JVM warmup

Każdy nowy proces JVM powoduje ładowanie klas do pamięci poprzez ClassLoader, po czym część jest ładowana do pamięci podręcznej, co skutkuje przyśpieszeniem działania kodu. Z tego wynika, że pierwsze uruchomienie JVM trwa najdłużej, po czym następne wywołania stają się szybciej, i tak w praktyce jest.
Dlatego pomiary powinniśmy wykonywać każdorazowo na rozgrzanej wirtualnej maszynie JVM.



Wykres 2 - Pomiary czasu wykonania przykładowej operacji w zależności od rozgrzania JVM

5.JHM - Java Microbenchmark Harness

Z powyższych powodów powodowanych przez pracę JVM stworzono framework Java Microbenchmark Harness (JHM) który umożliwia wykonywanie benchmarków naszego kodu.

Framework ten umożliwia testować wydajność kodu (w tym również algorytmów) poprzez dostarczone tryby przeprowadzania benchmarku:

- Throughput – przepustowość – ilość operacji w jednostce czasu (domyślnie sekunda), czyli ile razy nasz kod da radę wykonać się w ciągu sekundy.
- Average Time – średni czas ze wszystkich prób, jaki był potrzebny na wykonanie kodu.

- Sample Time – bardziej statystyczne podejście do mierzenia czasu wykonania, wraz z podziałem na histogram oraz percentyle.
- Single Shot Time – mierzy, ile czasu zajmie wykonanie benchmarku za pierwszym razem – bez Java warm-up.
- All – wszystkie powyższe tryby razem.

Aby framework działał prawidłowo, należy rozumieć jak działają wszystkie jego adnotacje, i następnie odpowiednio skonfigurować testy, odpowiednio do naszych potrzeb.

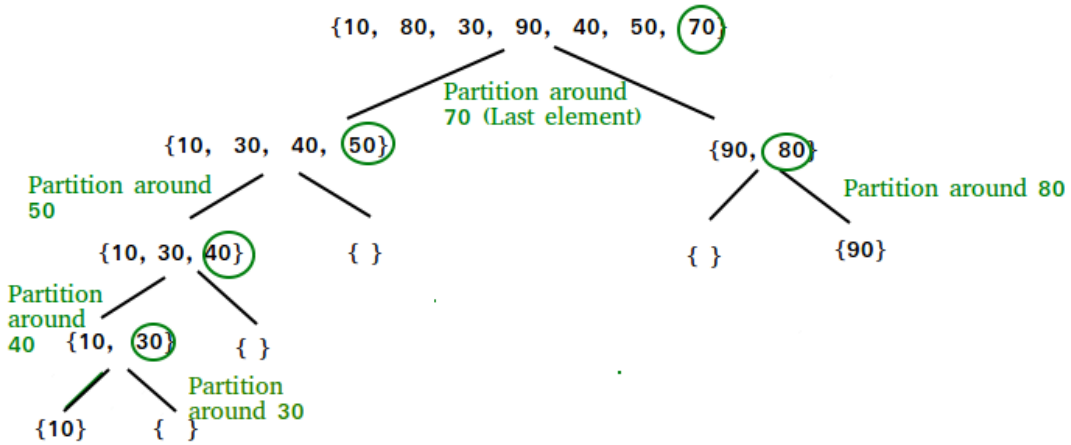
6. Algorytmy sortowania

1. QucicSort

1.Opis działania

Jest to algorytm, który działa na zasadzie divide and conquer, czyli dziel i zwyciężaj .Jego działanie jest następujące:

1. Wybór elementu podziałowego (pivot) względem którego następuje porównywanie innych elementów. Wybór pivota może być losowy, ale może być też zoptymalizowany np. poprzez wybór mediany
2. Następnie algorytm partycjonuje tablicę na dwie części: elementy mniejsze lub równe pivotowi, elementy większe od pivota. Elementy mniejsze od pivota dajemy na jego lewej stronie, a większe po prawej. Po zakończeniu tej fazy pivot jest na swojej ostatecznej pozycji w posortowanej tablicy.
3. Algorytm stosuje tę samą procedurę rekurencyjnie do sortowania dwóch powstałych podtablic: Jedna podtablica zawiera elementy przed pivotem, druga podtablica zawiera elementy po pivocie. Rekursja nie obejmuje elementu, który już jest pivotem (jest już na właściwym miejscu).
4. Zakończenie - Rekursja kończy się, gdy podtablica do posortowania ma rozmiar 1 lub 0, co oznacza, że jest już posortowana. Dzięki temu, że każde rekurencyjne wywołanie obejmuje coraz mniejsze fragmenty tablicy, cały proces jest bardzo szybki.



Rysunek 1 - Poglądowe działanie algorytmu quickSort

2.Analiza średniego i najgorszego przypadku – złożoność czasowa i pamięciowa

1.Złożoność czasowa:

- Najgorszy przypadek: $O(n^2)$ – przypadek gdy pivot nie dzieli tablicy na równomiernie wielkości podtablice (np. jest zawsze najmniejszym lub największym elementem).
- Średni przypadek - O(nlogn) - gdy pivot nie dzieli tablicy na równomiernie wielkości podtablice (np. jest zawsze najmniejszym lub największym elementem).

2.Złożoność pamięciowa:

- O(logn) w przypadku QuickSorta rekurencyjnego ze względu na stos wywołań rekurencyjnych. Możliwość optymalizacji poprzez wywołania iteracyjne.

3.Stabilność

Algorytm QuickSort w swojej podstawowej wersji nie jest stabilnym algorytmem sortowania, co oznacza, że równoważne elementy mogą zmienić swoje wzajemne położenie po sortowaniu. Aby zmodyfikować QuickSort tak, aby stał się stabilnym algorytmem, można zastosować różne podejścia. Poniżej opiszę dwie popularne metody:

- Użycie dodatkowej pamięci
- Zmodyfikowane algorytmu partycjonowania

4.Implementacje

- Standardowy Quicksort - Używa losowego pivota wybranego z tablicy, aby zminimalizować ryzyko wystąpienia najgorszego przypadku.
- Quicksort z medianą - Ulepszona wersja, która wybiera pivot jako medianę kilku losowo wybranych elementów, zapewniając bardziej zrównoważony podział, co jest korzystne w najgorszych scenariuszach.
- Introsort - Hybrydowy algorytm rozpoczynający się od quicksorta, który przełącza się na heapsort, gdy głębokość rekurencji przekracza logarytmiczną wartość liczby elementów, co zapobiega degradacji do złożonosc $O(n^2)$.

5. Możliwe optymalizacje

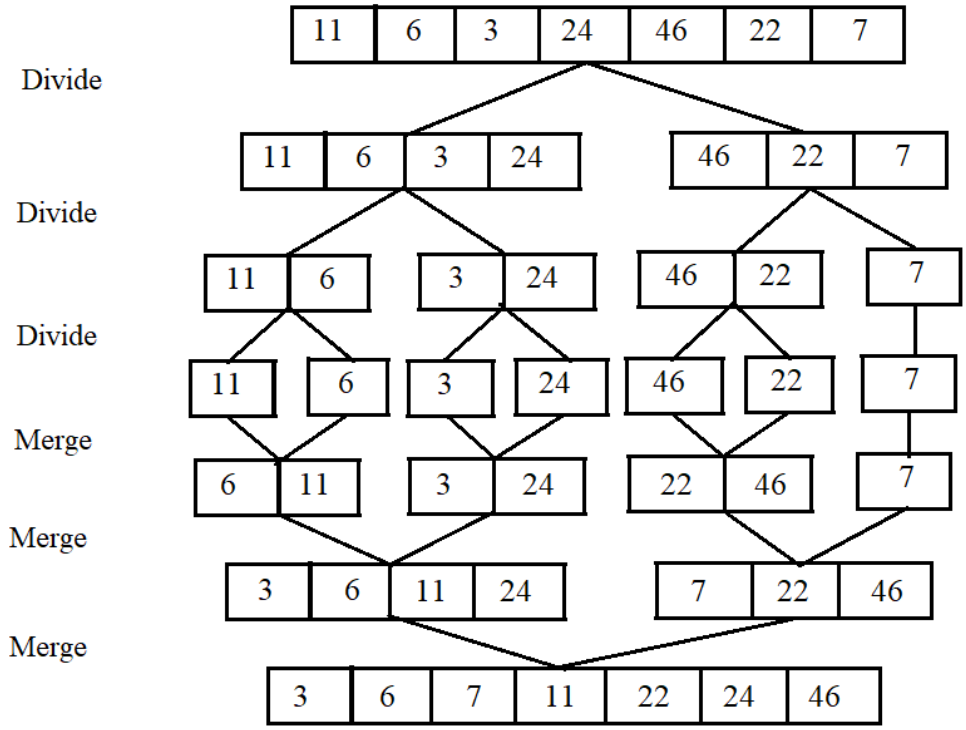
- Wybór Pivotu - Zastosowanie bardziej zaawansowanych metod wyboru pivotu, jak median of medians, może znacząco zwiększyć wydajność, szczególnie w niekorzystnych scenariuszach danych.
- Optimized Tail Recursion - W niektórych implementacjach quicksort, po podziale, rekurencyjne wywołanie jest potrzebne tylko dla jednej z podtablic, podczas gdy druga może być kontynuowana iteracyjnie, co zmniejsza zużycie stosu.

2. MergeSort (Sortowanie przez scalanie)

1.Opis działania:

MergeSort również jest oparty na strategii „dziel i zwyciężaj”, lecz w przeciwieństwie do QuickSort jest algorytmem zewnętrznym i wymaga dodatkowej pamięci. Algorytm dzieli zbiór na dwie równe części, które są rekurencyjnie sortowane, a następnie scalane. Scalenie

dwóch posortowanych list odbywa się poprzez porównywanie ich elementów i budowanie nowej posortowanej listy, która zawiera wszystkie elementy z obu podlist.



Rysunek 2- Poglądowe działanie algorytmu mergeSort

2.Analiza średniego i najgorszego przypadku – złożoność czasowa i pamięciowa

Złożoność czasowa:

- Ten algorytm ma zawsze złożoność czasową O (n log n), zarówno w przypadku średnim, jak i najgorszym, dzięki równomiernemu dzieleniu danych.

Złożoność pamięciowa:

- Wymagana jest dodatkowa pamięci, proporcjonalna do rozmiaru sortowanych danych, co jest jego główną wadą.

3.Stabilność

- Jest stabilny, co sprawia, że jest preferowany przy sortowaniu danych, gdzie stabilność jest wymagana.

4.Implementacje

- Top-down Merge Sort - To klasyczna implementacja rekurencyjna, która dzieli dane na coraz mniejsze fragmenty, aż do osiągnięcia pojedynczych elementów, które są następnie łączone w posortowane listy.
- Bottom-up Merge Sort - Ta iteracyjna wersja algorytmu zaczyna od sortowania małych, naturalnie występujących ciągów w danych, stopniowo łącząc je w coraz większe posortowane sekwencje. Jest to szczególnie efektywne, gdy dane wejściowe zawierają wiele już posortowanych sekwencji (Wikipedia).

- Parallel Merge Sort - Dzięki równoczesnej naturze merge sortu, istnieje wiele im‘plementacji, które wykorzystują równoległe przetwarzanie do przyspieszenia sortowania. Algorytmy takie jak parallel merge sort dzielą dane na podlisty, które są sortowane równolegle, wykorzystując moc obliczeniową wielu procesorów czy rdzeni.

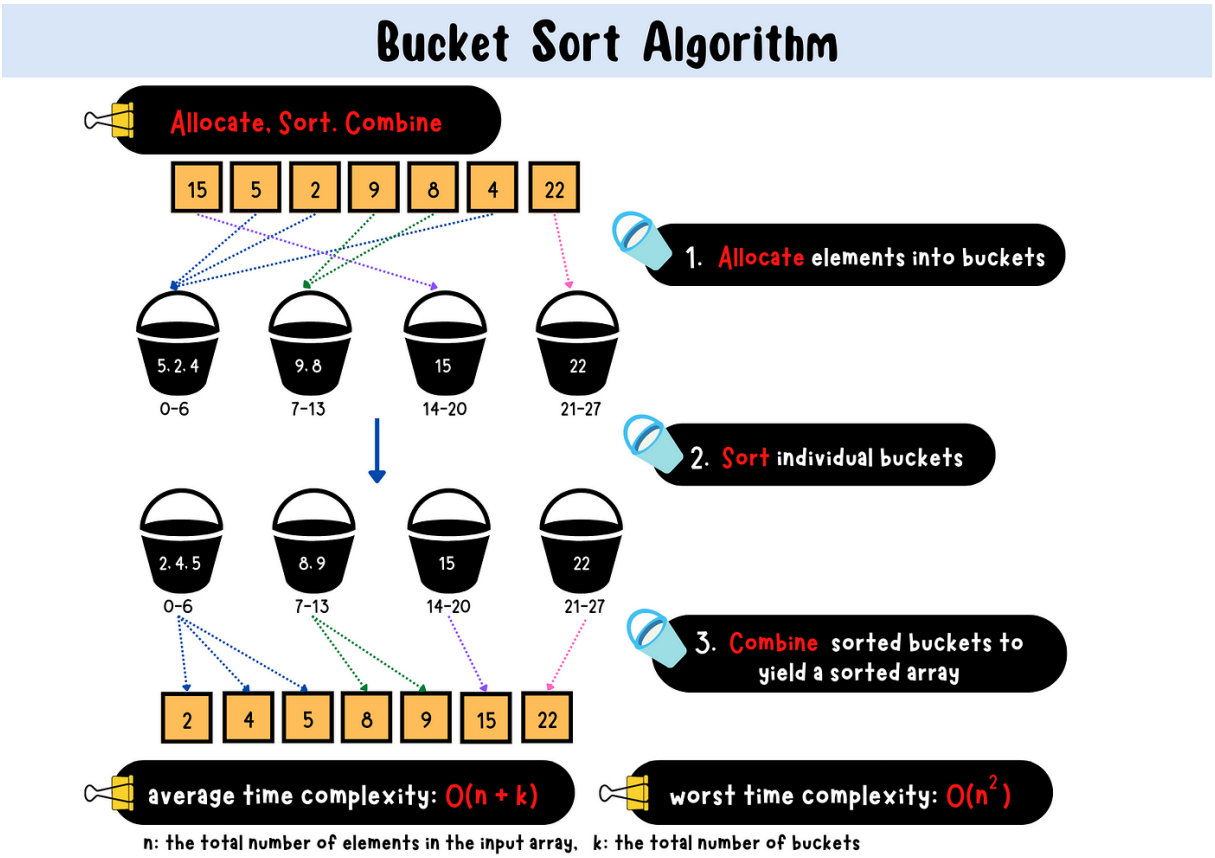
5. Możliwe optymalizacje:

- Implementacja sortowania in-place - Aby zmniejszyć dodatkowe wykorzystanie pamięci, można zaimplementować wariant sortowania merge sort działający na miejscu (in-place), co minimalizuje potrzebę alokacji dodatkowych struktur danych podczas scalania list. Chociaż czysty merge sort in-place jest trudny do realizacji, istnieją algorytmy, które mogą osiągnąć zbliżoną wydajność przy niewielkim dodatkowym zużyciu pamięci.
- Optymalne dobieranie podziału - Implementacja naturalnego sortowania merge polega na równomiernym podziale danych na dwie części w każdym kroku rekurencji, co zapewnia zbalansowaną głębokość rekurencji i efektywność procesu scalania, prowadząc do minimalizacji czasu wykonania algorytmu.

3. BucketSort (Sortowanie kubełkowe)

1.Opis działania

1. Utworzenie kubełków - algorytm rozpoczyna od utworzenia określonej liczby „kubełków” (które mogą być reprezentowane jako listy, tablice itp.). Liczba kubełków zależy od zakresu danych oraz od rozmiaru danych wejściowych. Każdy kubełek odpowiada pewnemu zakresowi wartości. Na przykład, jeśli dane zawierają wartości od 0 do 99, a tworzymy 10 kubełków, każdy kubełek może odpowiadać przedziałowi 10 wartości (0-9, 10-19, ..., 90-99).
2. Rozdzielanie Elementów do Kubełków - elementy danych są następnie rozdzielane do odpowiednich kubełków na podstawie ich wartości. W najprostszym przypadku rozdzielanie może polegać na obliczeniu odpowiedniego indeksu kubełka, np. przez podzielenie wartości elementu przez rozmiar kubełka.
3. Sortowanie Kubełków - każdy z kubełków jest osobno sortowany, co można zrealizować za pomocą dowolnego algorytmu sortowania – najczęściej quickSort lub insertionSort
4. Konkatenacja Kubełków - po posortowaniu każdego z kubełków, ich zawartość jest łączona w jedną posortowaną listę. Kubełki są przeglądane po kolei i ich zawartość jest dodawana do końcowego ciągu wynikowego.



Rysunek 3 - Poglądowy schemat działania algorytmu bucketSort

2.Analiza średniego i najgorszego przypadku – złożoność czasowa i pamięciowa

- Przypadek średni zakłada równomierne rozłożenie elementów. Sortowanie składa się z kilku etapów:
  1. Inicjalizacja kubełków i znajdowanie maksymalnej wartości klucza w tablicy – O(n) operacja – czyli przejście przez wszystkie elementy w celu znalezienia maksymalnego klucza oraz ustawienie struktury kubełków
  2. Rozmieszczanie elementów w kubełkach - Każdy element jest przypisywany do odpowiedniego kubełka na podstawie obliczonej wartości klucza – O(n)
  3. Sortowanie elementów w kubełkach - Jeśli w każdym kubełku używane jest sortowanie przez wstawianie, to całkowity czas potrzebny na posortowanie wszystkich kubełków wynosi $O(\sum_{i=1}^{k} n_i^2)$ - dla n elementów w i-tym kubku.
- Najgorszy przypadek - W przypadku algorytmu bucketSort, kluczowe jest uzyskanie dużej ziarnistości elementów wejściowych, czyli kryterium przydzielania elementów do kubełków powinien w równomierny sposób rozdzielać elementy do kubełków. Najgorszym przypadkiem jest sortowanie elementów podobnych, dla których kryterium przydzielania elementów, przydzieli jeden kubełek. W tym przypadku sortowanie będzie miało efektywność równą efektywności sortowania zastosowanego w obrębie pojedynczego kubka. W celu uniknięcia wystąpienia najgorszego przypadku należy mieć pewność, że dane wejściowe nie będą równomierne, lub posiadać zoptymalizowany algorytm przydzielania elementów do kubełków. Brak tych elementów skutkuje złożonością czasową $O(n^2)$.

Złożoność pamięciowa:

- Złożoność pamięciowa algorytmu kubełkowego jest równa O(n+k), gdzie n to ilość elementów, natomiast k to ilość kubełków.

*3.*Stabilność

- Stabilność algorytmu jest zależna od zastosowanego algorytmu sortowania dla pojedynczego kubełka. Jeżeli zastosujemy algorytm niestabilny np. standardowy qucikSort, bucketSort nie będzie stabilny.

4.Implementacje

Rozróżniamy 4 implementacje BucketSort:

- Generic bucket sort - Podstawowa forma, gdzie dane są dzielone równomiernie na kubełki, a następnie sortowane wewnątrz każdego z nich.
- ProxmapSort - Wariant, który wykorzystuje specjalną funkcję (map key function) do określenia, jak dane powinny być przypisane do kubełków, zachowując częściowe uporządkowanie.
- Histogram sort - Metoda ta najpierw zlicza, ile elementów znajdzie się w każdym kubełku, co pozwala na bardziej efektywne zarządzanie pamięcią i przyspieszenie sortowania.
- Postman's sort - Wykorzystywana przede wszystkim w sortowaniu listów, gdzie dane są grupowane i sortowane według hierarchii atrybutów, jak region czy rodzaj przesyłki.
- Shuffle sort - Odmiana, która najpierw sortuje część danych, a potem rozdziela resztę między utworzone wcześniej grupy, by finalnie połączyć je w posortowaną całość.

5. Możliwe optymalizacje

- Kluczową optymalizacją organizacji kubełków w algorytmie bucketSort jest stosowanie na grupowanych w kubełki obiektach metod hasuhjących (hashCode). Dostępne są różnorodne implementację metod hashujących, które służą do obliczania kodu, na podstawie którego następnie są przydzielane kubełki, które są równomiernie wypełniane, korzystając z operacji modulo. Dobra funkcja haszująca powinna minimalizować konflikty, czyli sytuacje, gdy dwa różne klucze mają tę samą wartość haszującą.
- Inną powszechnie stosowaną optymalizacją jest umieszczenie nieposortowanych elementów z kubłów z powrotem w oryginalnej tablicy, używamy sortowania przez wstawianie na całej tablicy. Dzieje się tak, ponieważ sortowanie przez wstawianie działa szybko, jeśli elementy są blisko swoich ostatecznych pozycji. Liczba porównań pozostaje niewielka, co sprawia, że wykorzystanie pamięci jest efektywniejsze, gdy lista jest przechowywana ciągło w pamięci.

7. Implementacja kodu:

Kod składa się z klasy uruchominiowej Main, która tworzy instancję klasy Menu, i następnie wywołuje run.

W tej metodzie następuje utworzenie instancji klas: Comunicator, DataService, oraz Measurement. Następuje zapytanie się użytkownika o wybór struktury: ArrayLisy, lub LinkedList. W zależności od wyboru tworzy się odpowiednia struktura, po czym następnie następuje odczyt danych z pliku, którego ścieżka jest zdefiniowana w klasie DataService. Odczyt danych następuje poprzez odczyt danych wejściowych jako String, i następnie podział przy użyciu regex'a. Po odczycie następuje sprawdzenie czy odczytana linijka zawiera ocenę, lub nie zawiera. W przypadku zawierania oceny, tworzony jest obiekt z 3 polami int id, String filmTitle, oraz Optional<Integer> rating. Jeżeli odczytana linijka nie zawiera oceny, jest zapisywana do struktury z wartością rating jako Optiona.empty().Dodatkowo jest dostępna również zakomentowana opcja z detekcją pierwszego przecinka od lewej strony, jak i również od prawej strony, i następne podzielenie wpisów na obiekty.

Następnie użytkownik wybiera rozmiar testowanej struktury, po czym następuje utworzenie nowej struktury, do której iteracyjnie dodawane są obiekty aż do zadanej przez użytkownika ilości elementów.

Następnie obiekt o odpowiednim rozmiarze i ilości wpisów jest przekazywany do metody sortingMeasurements z klasy Measurement, w której następuje rozpoczęcie pomiarów, oraz wykonanie metody getNumberOfFalseRecords, w której definiowana jest zmienna counter, po czym następuje przejście przez każdy element struktury, i wywołanie sprawdzenia, czy pole obiektu klasy Film zawiera opinię. Jeżeli nie, zwiększany jest counter, i po przeiterowaniu przez wszystkie elementy, zwracana jest ilość obiektów bez opisu.

W celu usunięcia wpisów z brakiem oceny wywoływana jest metoda deleteRecords, która iteruje po strukturze i następnie w przypadku, kiedy element nie posiada oceny jest usuwany ze struktury.

Użytkownik w następnej kolejności wybiera jaki algorytm sortowania ma zostać zastosowany na strukturze zawierającej filmy wyłącznie z opiniami.

Do przeprowadzenia sortowania służą odpowiednio klasy: BucketSort, MergeSort oraz QucikSort.

8. Wyniki pomiarów:

Pomiary wyszukiwania błędnych elementów:

Możliwe nieprawidłowości pomiaru:

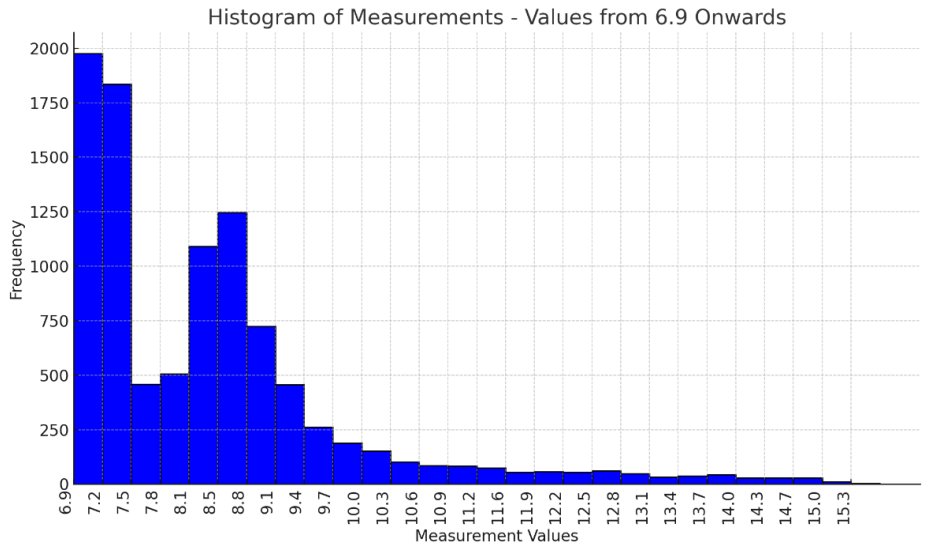
- Klasa film posiada pole rating typu Optional<Integer>, czyli zastosowano klasę opakowującą (wrapper class). Zastosowanie tej klasy powoduje, że sprawdzanie, czy obiekt zawiera ocenę, wymaga odpakowania klasy opakowującej, co generuje dodatkowe opóźnienie

- Stworzona metoda zawiera counter, którego zwiększanie również generuje dodatkowe straty czasowe. Jest możliwość napisania metody void, która wykonywałaby get() na strukturze, i nic by nie zwracała.

Podane wyniki wyszukiwania elementów została zawężona stosując regułę 3 sigm, w celu odrzucenia wartości skrajnych.

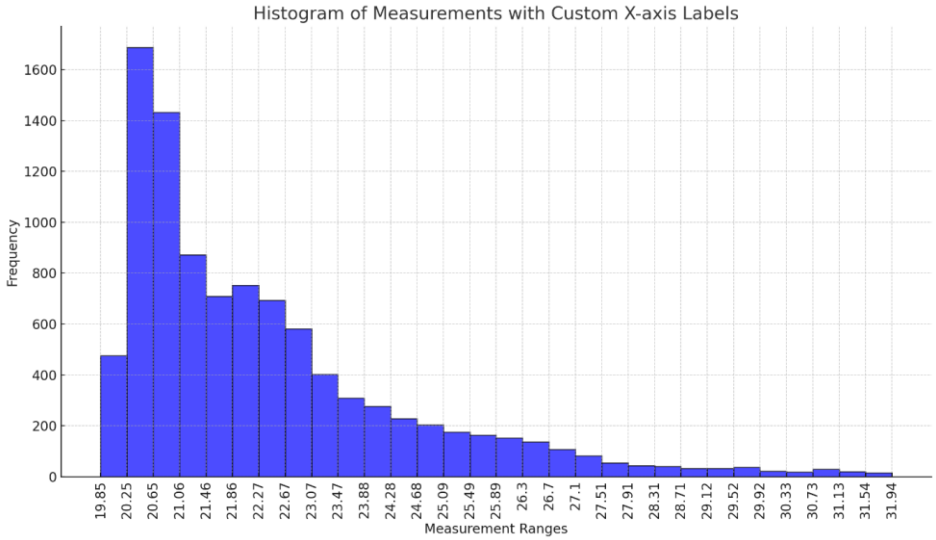
1. Dla struktury: ArrayList:

- Średni czas: 8.65ms
- Minimalny: 6,91ms
- Maksymalny 29,94ms



2. Dla struktury: LinkedList:

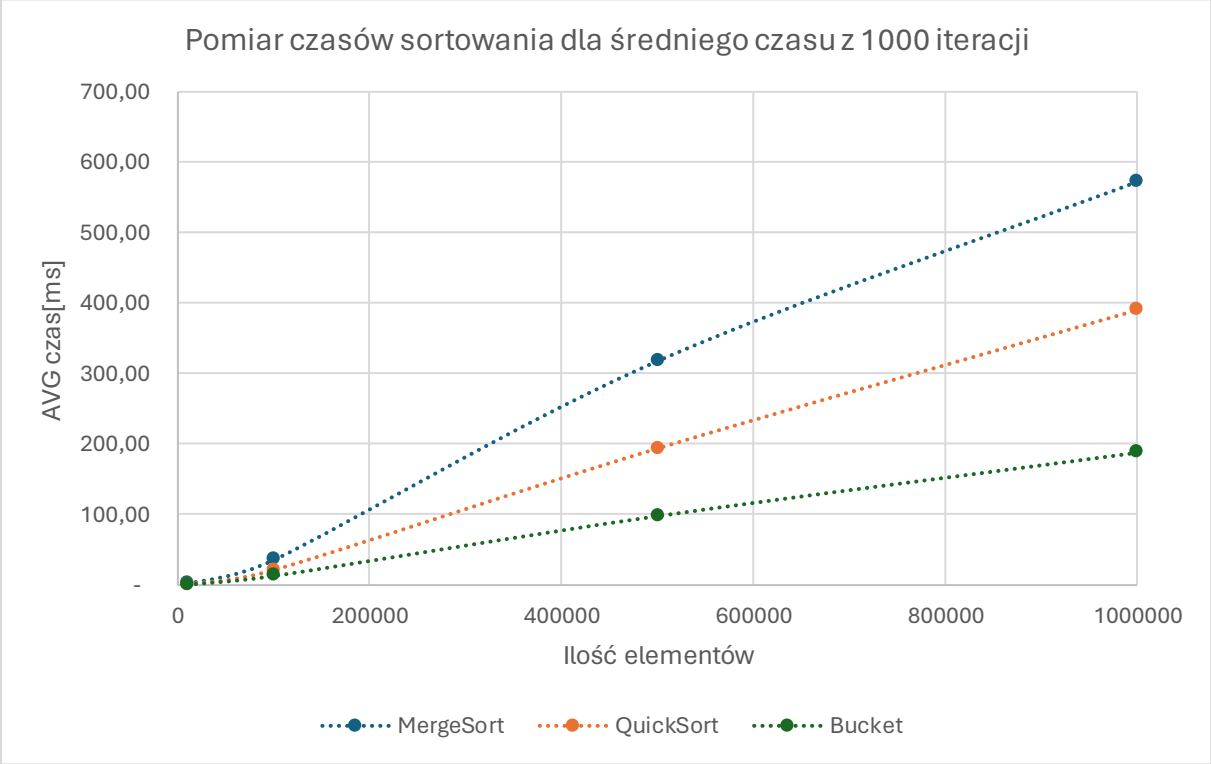
- Średni czas: 22,65ms
- Minimalny: 19,84ms
- Maksymalny 89,88ms



Pomiary sortowania elementów według rankingu:

Ilość danych:	MergeSort	QuickSort	Bucket
10K	1.84ms	1.11ms	0.53ms
100K	35.60ms	21.31ms	12.94ms
500K	317.74 ms	193.80ms	97.78ms
1M	572.16 ms	390.19ms	187.74ms

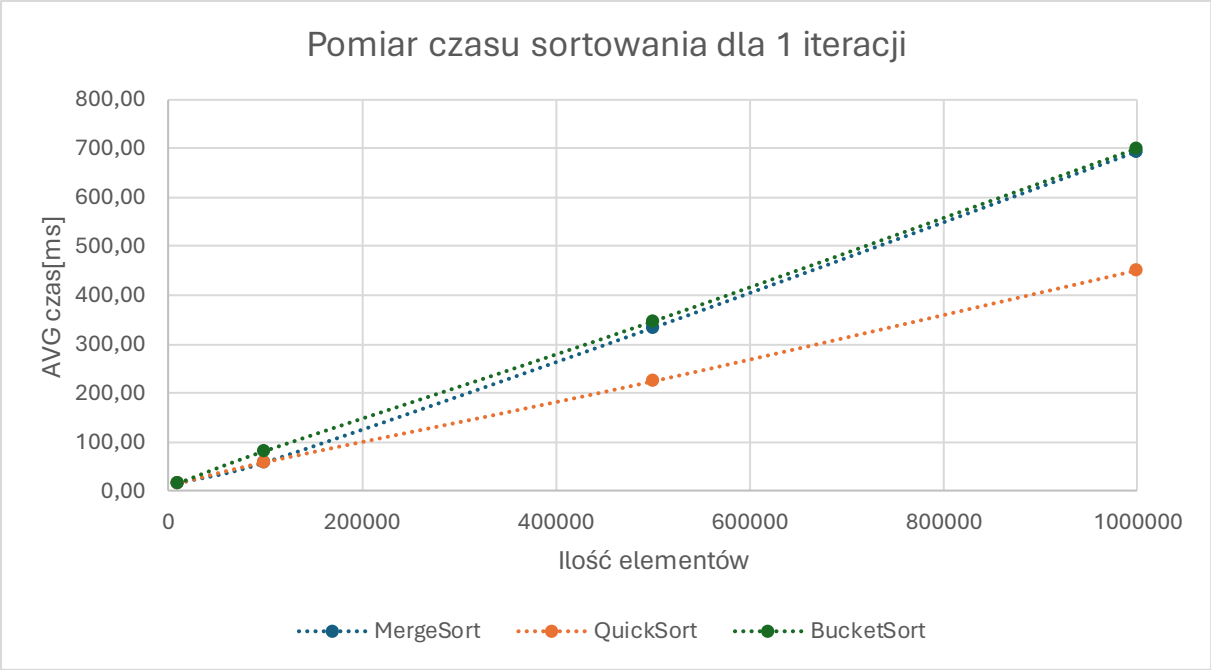
Tabela 1 - Pomiary czasów sortowania dla średniego czasu z 1000 pomiarów



Wykres 3 - Pomiary czasów sortowania dla średniego czasu z 1000 pomiarów

Ilość danych:	MergeSort	QuickSort	BucketSort
10K	15.45 ms	14.40 ms	14.70 ms
100K	57.58 ms	58.18 ms	81.14 ms
500K	332.85 ms	223.51 ms	346.01 ms
1M	694.03 ms	450.89 ms	700.52 ms

Tabela 2 - Pomiar czasu sortowania dla 1 iteracji



Wykres 4 - Pomiar czasu sortowania dla 1 iteracji

MergeSort	QuickSort	Bucket
5434.78	9023.57	18839.96
2809.28	4691.69	7729.79
1573.63	2580.04	5113.60
1747.76	2562.84	5326.54

Tabela 3 - Przepustowość – ilość operacji w ciągu 1 sekundy

Pomiary zostały dokonane dla pojedynczego pomiaru (Single Shot Time) oraz dla iteracji 1000 pomiarów, na podstawie których została wyliczona średnia (Average Time). W bardziej rozbudowanej wersji, jest możliwość zastosowania bardziej statystycznego podejścia uwzględniając podział na histogram oraz percentyle.

Ilość elementów	Średnia ocena	Mediana ocen
10000	5.4603	5
100000	6.0900	7
500000	6.6657	7
1000000	6.6366	7

Tabela 4 - Średnia ocen oraz mediana dla struktury o określonym rozmiarze

10. Analiza asymptotyczności na podstawie zmierzonych danych :

1.Porównanie skalowalności

Dane empiryczne					Krotność wzrostu czasu wykonania		
n	MS	QS	BS	Krotność wzrostu ilości danych	MS	QS	BS

10000	1.84	1.11	0.53	1	1	1	1
100000	35.60	21.31	12.94	10	19.35	19.23	24.37
500000	317.74	193.80	97.78	5	8.93	9.09	7.56
1000000	572.16	390.19	187.74	2	1.80	2.01	1.92

Tabela 5 - Porównanie skalowalności czasu wykonania algorytm, gdzie: n – ilość elementów struktury, MS – mergeSort, QS – quickSort, BS – bucketSort

Z powyższej tabeli idzie wywnioskować jak wygląda skalowalność czasu wykonania poszczególnego algorytmu sortowania, w zależności od wzrostu liczby danych:

Krotność wzrostu danych	Dla 10 krotnego wzrostu danych:	Dla 5 krotnego wzrostu danych:	Dla 2 krotnego wzrostu danych:
mergeSort	Wzrósł 19.35 krotnie	Wzrósł 8,93	Wzrósł 1.80
qucikSort	wzrósł 19.23 krotnie	Wzrósł 9,09	Wzrósł 2,01
bucketSort	wzrósł 24,37 krotnie	Wzrósł 7.56	Wzrósł1,92

Z tej tabeli widać, że mergeSort oraz quickSort skalują się lepiej niż liniowo, co sugeruje, że empiryczna złożoność może pasować do oczekiwanej teoretycznej złożoności O(nlogn).

11. Podsumowanie

Znajomość algorytmów sortowania, ich złożoności czasowej, jak i również pamięciowej jest kluczowe do tworzenia optymalnie działających programów. Mimo, że w powyższym sprawozdaniu skupiłem się na złożoności obliczeniowej, warto jest również brać pod uwagę złożoność pamięciową, gdyż niektóre algorytmy mimo szybkiego sortowania, potrafią alokować bardzo duże ilości pamięci. Przydaną umiejętnością jest również znajomość możliwych optymalizacji danych algorytmów, oraz ich wdrożenia, gdyż często małą zmianą da się znacząco podnieść wydajność algorytmu.

Kiedy zastosować jaki algorytm:

quickSort:

- Kiedy potrzebujemy wydajnego sortowania in-place (bez dodatkowego znacznego zapotrzebowania na pamięć).
- Dane są losowe lub nie możemy przewidzieć rozkładu danych (dobrze radzi sobie z różnorodnością danych).
- Nie wymagamy stabilności sortowania (gdzie stabilność oznacza, że elementy o równych kluczach zachowują swoją wzajemną kolejność).
- Dobrze sprawdza się dla dużych zbiorów danych, nawet jeśli wymaga więcej pamięci, co jest akceptowalne w systemach, gdzie pamięć nie jest głównym ograniczeniem.

- Idealny do sortowania list, gdzie dodatkowe koszty pamięciowe są zminimalizowane przez możliwość używania wskaźników zamiast fizycznego kopiowania elementów.

mergeSort:

- Używamy kiedy chcemy uzyskać stałą złożoność czasową niezależnie od danych wejściowych
- Dobrze sprawdza się dla dużych zbiorów danych, nawet jeśli wymaga więcej pamięci, co jest akceptowalne w systemach, gdzie pamięć nie jest głównym ograniczeniem.
- Idealny do sortowania list, gdzie dodatkowe koszty pamięciowe są zminimalizowane przez możliwość używania wskaźników zamiast fizycznego kopiowania elementów.

bucketSort:

- Najlepiej sprawdza się, gdy dane są jednorodnie rozłożone i należą do dobrze znanego zakresu.
- Efektywny, gdy klucze sortowania są nie tylko liczbami, ale mogą być także innymi danymi, które łatwo przypisać do wiader, np. słowa czy znaki.
- Może być bardzo szybki w przypadku danych o specyficznych charakterystykach, które pozwalają na efektywne podział na wiadra.
- Najlepiej sprawdza się dla danych, które można łatwo i równomiernie podzielić na kubełki.
- Efektywny dla sortowania dużych ilości danych, gdzie klucze są dobrze znane i mogą być łatwo podzielone na przedziały.
- Odpowiedni w środowiskach, gdzie dostępna jest duża ilość pamięci, co pozwala na efektywne zarządzanie dużymi kubełkami

12. Biografia:

1. Ashok Kumar Karunanithi, Department of Computing Science, Umea University, June 2014, A Survey, Discussion and Comparison of Sorting Algorithms, dostępny online: https://citeseerx.ist.psu.edu/document?repid=rep1&type=pdf&doi=d010950f6b3c9521eb437334fa69c0b2b9353010
2. Baeldung, Serwis informatyczny, Microbenchmarking with Java, January 10, 2024, dostępny online: https://www.baeldung.com/java-microbenchmark-harness
3. Docs Oracle, Dokumentacja techniczna firmy Oracle, Class System, dostępny online: https://docs.oracle.com/javase/9/docs/api/java/lang/System.html#nanoTime
4. GeeksforGeeks, Serwis informatyczny, Artykuł: Bucket Sort, dostępny online: https://www.geeksforgeeks.org/bucket-sort-2/
5. GeeksforGeeks, Serwis informatyczny, Artykuł: Merge Sort – Data Structure and Algorithms Tutorials, 08 Apr, 2024
6. GeeksforGeeks, Serwis informatyczny, Artykuł: Quick Sort, dostępny online: https://www.geeksforgeeks.org/quick-sort/
7. Michael T. Goodrich, Roberto Tamassia, Michael H. Goldwasser, Data Structures and Algorithms in Java. Sixth Edition, Wiley, March 2016, ISBN: 978-1-118-77133-4
8. OpenJdk, Dokumentacja techniczna firmy Oracle, Code Tools: jmh, dostępny online: https://openjdk.org/projects/code-tools/jmh/
9. StormIT, Serwis informatyczny [online], Benchmark sposobem na wydajniejsze aplikacje – JMH, dostępny online: https://stormit.pl/benchmark-jmh/#pulapka-1-ziarnistosc-pomiaru-czasu-systemcurrenttimemillis
10. Wikipedia, the free encyclopedia, Artykuł: Analysis of algorithms, dostępny online: https://en.wikipedia.org/wiki/Analysis_of_algorithms
11. Wikipedia, the free encyclopedia, Artykuł: Bucket sort, dostępny online: https://en.wikipedia.org/wiki/Bucket_sort
12. Wikipedia, the free encyclopedia, Artykuł: Merge sort, dostępny online: https://en.wikipedia.org/wiki/Merge_sort
13. Wikipedia, the free encyclopedia, Artykuł: Quicksort, dostępny online: https://en.wikipedia.org/wiki/Quicksort
14. Dr. inż. Łukasz Jeleń, Prezentacje: Projektowanie i analiza algorytmów
15. Algorytm Edu PL, Serwis edukacyjny, Quick Sort, dostępny online: https://www.algorytm.edu.pl/algorytmy-maturalne/quick-sort.html