Politechnika Wrocławska

Sprawozdanie – Projektowanie i analiza algorytmów

Projekt

Wydział Informatyki i Telekomunikacji

Kierunek Informatyczne Systemy Automatyki

Wykonał: Paweł Zimoląg

Nr albumu:

Semestr letni 2022/2023

Kod zajęć:

Data zajęć:

Godzina zajęć:

Prowadzący:

Spis treści

Wstęp

Algorytmy sortujące są podstawowym zagadnieniem w informatyce i są powszechnie wykorzystywane do podziału danych w różnych programach. Sortowanie w programowaniu oznacza odpowiednie ustawianie elementów malejąco lub rosnąco, tak aby praca na nich była bardziej wydajna oraz łatwiejsza dla zarówno ludzi, jak i aplikacji. Dotychczas wymyślono wiele algorytmów mających za zadanie odpowiednio sortować elementy i każdy z nich ma swoje pozytywne, jak i negatywne aspekty. Wybór algorytmu zależny jest od zapotrzebowania na: rozmiar i typ danych, dostępną pamięć lub też szybkość działania. Sortowanie wykorzystywane jest we wszelakich dziedzinach życia i technologii, np.: w bazach danych do sortowania rekordów wg. określonych kryteriów; w systemach operacyjnych do sortowania plików wg. nazwy, rozmiaru czy też rozszerzenia; w wyszukiwarkach internetowych do sortowania wyników wg. trafności, popularności czy też daty publikacji.

Sprawozdanie opisuje moje wyniki dotyczące wykorzystania zaimplementowanych własnoręcznie algorytmów sortujących. Są to: sortowanie szybkie (quicksort), sortowanie przez scalanie (merge sort) oraz sortowanie kubełkowe (bucket sort).

Opis badanych algorytmów

Sortowanie szybkie (quicksort).

Quicksort to rekurencyjny algorytm wykorzystujący metodę „dziel i zwyciężaj”, która polega na dzieleniu problemu na mniejsze „pod problemy” do momentu, gdy nie ma dalszej możliwości podziału. Sortowanie szybkie w pierwszej kolejności wybiera element osiowy (tzw. *pivot)*, który jest wyznacznikiem dla zbioru danych do podziału na mniejsze zbiory. Do lewej „pod grupy” należą dane mniejsze lub równe elementowi rozdzielającemu, a do prawej pozostałe. Dzięki wykorzystaniu rekurencji zbiór danych zostaje poprawnie posortowany.

Złożoność czasowa algorytmu zależy od rozpatrywanego przypadku oraz implementacji sortowania. Wpływ na złożoność czasową mają m. in. równomiernie rozłożony zbiór danych oraz wybór elementu osiowego. W przypadku typowym wynosi $\boldsymbol{O}(\boldsymbol{n} * \mathbf{log}(\boldsymbol{n}))$, zaś w przypadku najgorszym jest to $\boldsymbol{O}(\boldsymbol{n^2})$. W typowym przypadku jest to najszybszy algorytm ze swojej klasy złożoności obliczeniowej.

Sortowanie przez scalanie (merge sort).

Sortowanie przez scalanie również wykorzystuje metodę „dziel i zwyciężaj”. Algorytm dzieli zbiór danych w połowie na dwie grupy, w których następuje rekurencyjny podział. Dzielenie trwa do momentu, aż w grupach pozostaną pojedyncze elementy. Kolejnym krokiem jest scalanie grup, zaczynając od zbiorów z jednym elementem kończąc na scaleniu całego zbioru danych.

Złożoność czasowa merge sort to $\boldsymbol{O}(\boldsymbol{n} * \mathbf{log}(\boldsymbol{n}))$, bez względu na dane wejściowe oraz dla każdego przypadku, dzięki czemu jest w praktyce jednym z najszybszych algorytmów sortujących.

Sortowanie kubełkowe (bucket sort).

Sortowanie kubełkowe polega na dzieleniu zbioru elementów oraz wstawianiu ich do tzw. kubełków, czyli zbiorów liczb o określonym przedziale. Następnie algorytm tworzy zbiór wszystkich elementów, do których w odpowiedniej kolejności przydziela posortowane elementy z kubełków. Bucket sort jest bardzo efektywnym algorytmem, ale do jego działania jest konieczna znajomość rozstępu danych (różnica między największą i najmniejszą wartością ze zbioru) oraz elementy muszą być równomiernie rozłożone, aby złożoność była liniowa.

Złożoność czasowa sortowania kubełkowego jest rzędu $\boldsymbol{O}(\boldsymbol{n}+\boldsymbol{k})$, gdzie $\boldsymbol{n}$ to liczba elementów do posortowania, a $\boldsymbol{k}$ to liczba kubełków.

Omówienie przebiegu eksperymentów oraz wyniki

Uwagi.

Należało przygotować strukturę danych zawierającą odpowiednio: 10 000, 100 000, 500 000, 1 000 000 oraz maksymalną ilość danych z pliku. Z uwagi na fakt, iż w pliku znajdowało się przed filtracją 405 365 wierszy danych, dla wszystkich sortowań wykonałem pomiary dla odpowiednio: 10 000, 100 000, 250 000 oraz maksymalnej ilości danych z pliku (ilość elementów przed filtracją).

Ilość elementów znajdująca się w tabelach oraz na wykresach jest po wykonaniu funkcji filtrującej.

Aby zweryfikować poprawność algorytmów, napisałem funkcję sprawdzającą efekt sortowania. Działa ona w następujący sposób: na początku pomocnicza zmienna wynosi 1 (najmniejsza wartość oceny), po czym w pętli idącej aż do samego końca listy sprawdza, czy elementy nie są mniejsze niż pomocnicza zmienna. W przypadku,

gdy ocena zwiększa się, wartość iteratora również zostaje zwiększona i dalej sprawdza, czy kolejne węzły listy nie są mniejsze od niej.

Wyniki dla funkcji filtrującej (przypisującej).

Do wykonania projektu wykorzystałem własnoręczną implementację listy dwukierunkowej oraz stos z biblioteki „stack” do sortowania szybkiego. Metody listy:

- Dodawanie oraz usuwanie elementów z przodu listy,
- Dodawanie oraz usuwanie elementów z tyłu listy,
- Wyświetlanie wybranego przedziału elementów listy,
- Zwracanie ilości elementów w liście,
- Zwracanie konkretnego elementu listy dla podanego indeksu od przodu oraz tyłu.

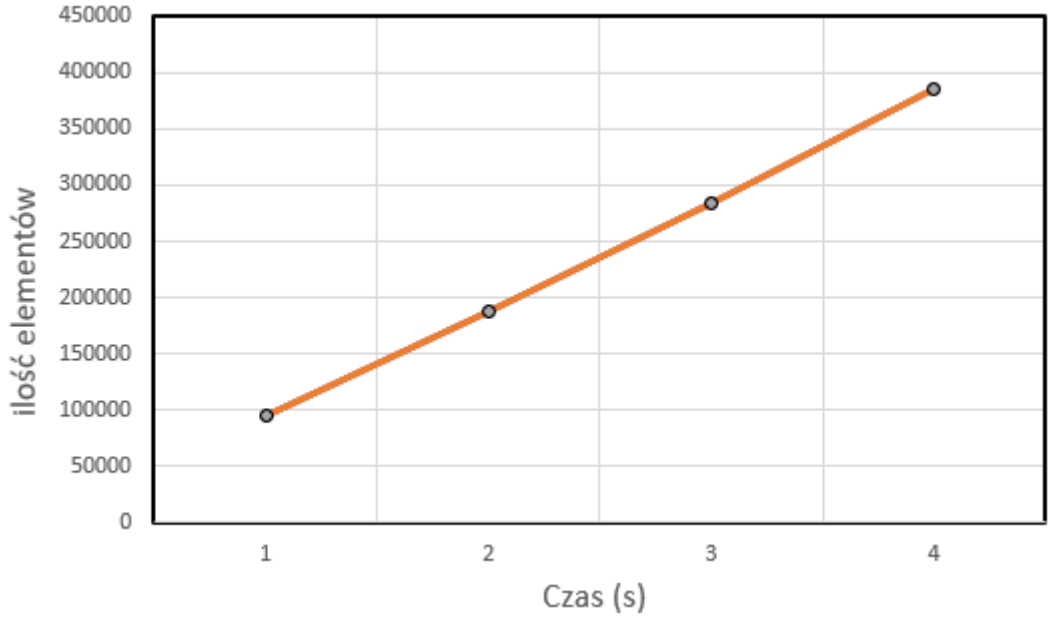
Funkcja filtrująca dodaje nowy element do listy dla każdego rekordu, chyba że brakuje w nim jednej z informacji, tj. indeksu, nazwy lub oceny. Złożoność czasowa przeszukiwania listy dwukierunkowej to $\boldsymbol{O(n)}$.

Dla dokładniejszych wyników, pomiary zostały wykonane trzykrotnie, a następnie do przedstawienia wykresu wykorzystana została średnia.

Wyniki pomiarów:

Ilość elementów\Pomiar (s)	I	II	III	Średnia (s)
95 814	0,967	0,926	0,919	0,937
188 206	1,747	1,746	1,793	1,762
284 246	2,974	2,724	2,745	2,814
385 636	3,652	3,686	3,649	3,662

Tabela 1 – ilość elementów użytych do pomiarów oraz średnie czasów



Wykres 1 – wykres złożoności czasowej dla dodawania elementów do listy

Wykres pokazuje liniowy wzrost czasu dla dodawania elementów, który zgadza się z oczekiwaną złożonością czasową.

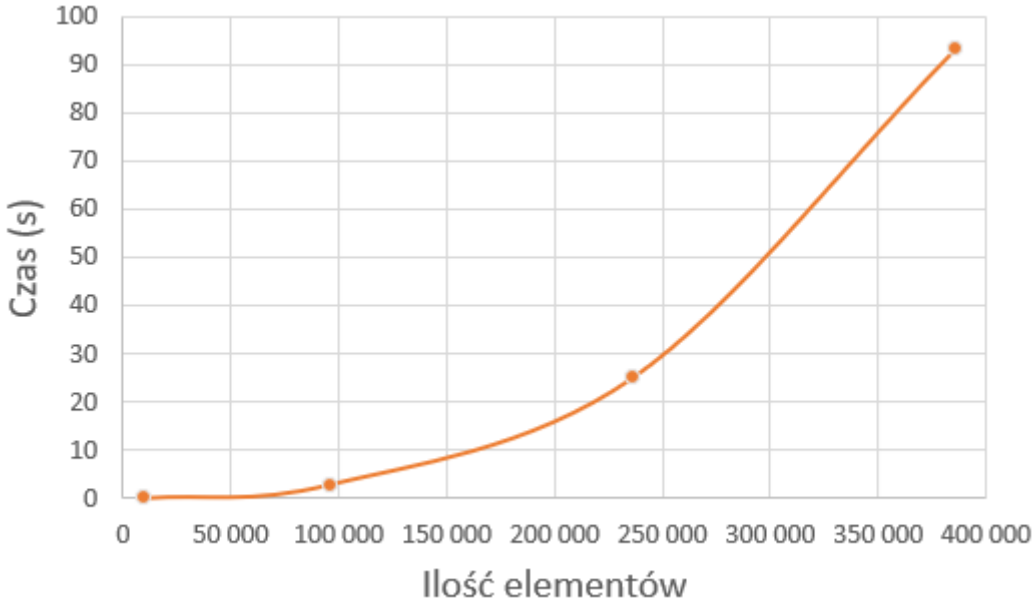
Wyniki dla sortowania szybkiego (quicksort).

Do implementacji algorytmu szybkiego wykorzystałem stos, aby przekazywane wskaźniki na elementy listy dwukierunkowej podczas rekurencji nie zajmowały dodatkowo pamięci. Kod sortowania składa się z głównej funkcji „quicksort", która zamienia miejscami elementy mniejsze i większe niż osiowy oraz funkcji „partition", która ma na celu wyznaczenie pivota.

Wyniki pomiarów:

Ilość elementów/Pomiar (s)	I	II	III	Średnia	Mediana
10 000	0,042	0,037	0,037	0,039	0,037
95 814	2,858	2,866	2,809	2,844	2,858
236 302	24,975	24,846	25,42	25,080	24,975
385 636	92,952	97,665	89,094	93,237	92,952

Tabela 2 – ilość elementów użytych do pomiarów, średnie oraz mediany czasów



Wykres 2 – wykres złożoności czasowej dla quicksort

Powyższy wykres zgadza się z zakładaną złożonością czasową dla sortowania szybkiego. W tym przypadku, większość elementów była przypadkiem typowym, dzięki czemu algorytm działał szybko.

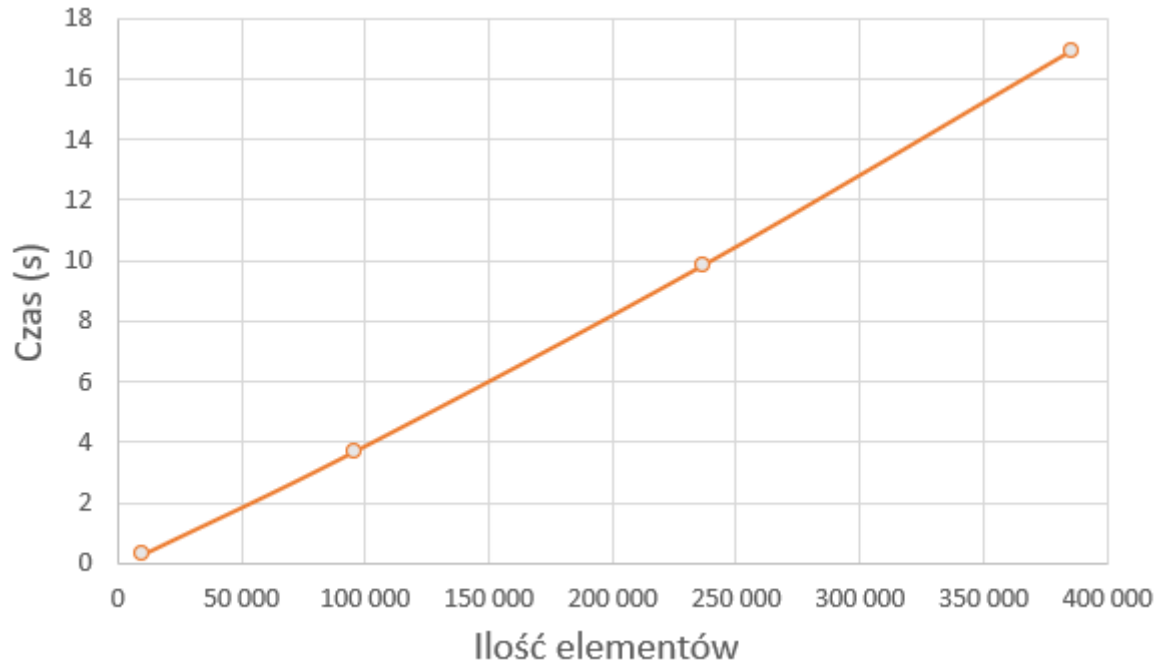
Wyniki dla sortowania przez scalanie (merge sort).

Implementacja merge sort wykorzystuje dwie funkcje: „mergesort” oraz „merge”. Pierwsza z nich ma na celu rekurencyjny podział listy na mniejsze pod listy, a druga sortuje listy oraz łączy je w większe listy, rozdzielone wcześniej.

Wyniki pomiarów:

Ilość elementów/Pomiar (s)	I	II	III	Średnia	Mediana
10 000	0,32	0,302	0,298	0,307	0,302
95 814	3,806	3,652	3,701	3,720	3,701
236 302	9,699	9,938	9,912	9,850	9,912
385 636	16,649	17,122	17,081	16,951	17,081

Tabela 3 – ilość elementów użytych do pomiarów, średnie oraz mediany czasów



Wykres 3 – wykres złożoności czasowej dla merge sort

Powyższy wykres różni się od zakładanej złożoności czasowej dla sortowania przez scalanie. Powodów może być kilka, np.: błędy w implementacji algorytmu lub błąd w analizie złożoności czasowej.

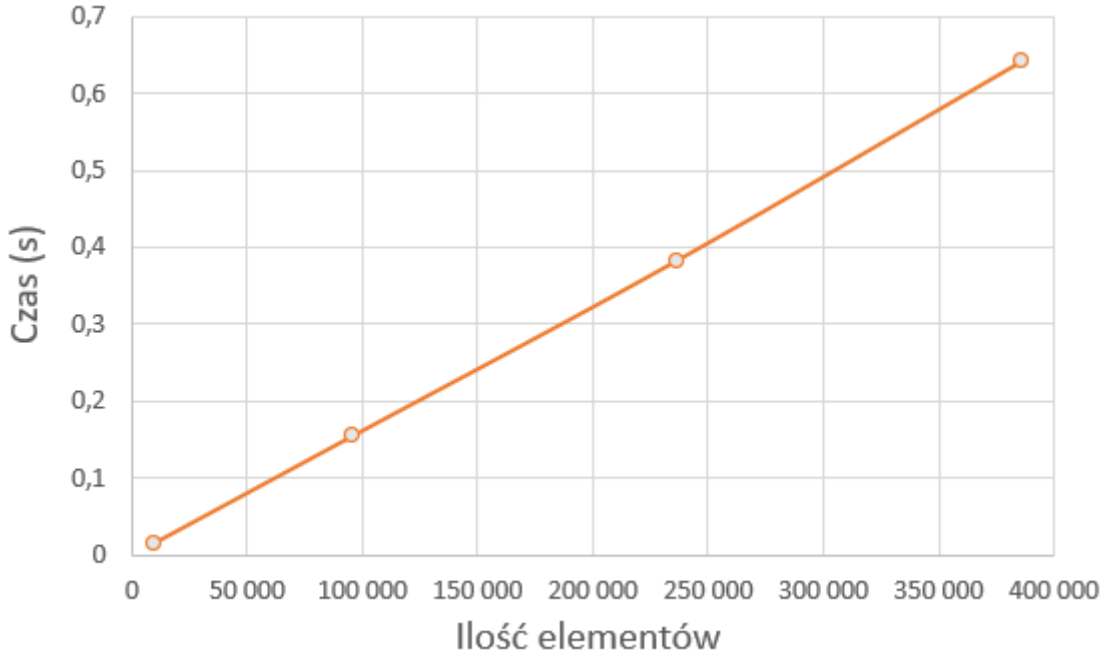
Wyniki dla sortowania kubełkowego (bucket sort).

Implementacja sortowania kubełkowego jest prosta, ale wymaga spełnienia dwóch warunków: znajomości rozstępu danych oraz w miarę równomiernie rozłożone elementy. W przypadku danych, na których wykonywany był projekt, najmniejsza wartość to 1, a największa to 10. Jako kubełki na dane wykorzystałem tablicę list dwukierunkowych, które przetrzymywały tylko rekordy o konkretnej ocenie od 1 do 10.

Wyniki pomiarów:

Ilość elementów/Pomiar (s)	I	II	III	Średnia	Mediana
10 000	0,017	0,016	0,016	0,016	0,016
95 814	0,159	0,153	0,154	0,155	0,154
236 302	0,382	0,384	0,383	0,383	0,383
385 636	0,638	0,647	0,64	0,642	0,64

Tabela 4 – ilość elementów użytych do pomiarów, średnie oraz mediany czasów



Wykres 4 – wykres złożoności czasowej dla bucket sort

Powyższy wykres zgadza się z zakładaną złożonością czasową dla sortowania kubełkowego. Algorytm dzięki swojej złożoności sortuje elementy w bardzo szybkim tempie.

Wnioski

Na podstawie wykonanego projektu oraz analizy można wywnioskować, że skuteczność algorytmów zależy od wielu czynników, takich jak rozmiar sortowanych danych czy też poprawność implementacji kodu. Na początkowym etapie wykonywania zadań, algorytmy sortujące potrzebowały znacznie więcej czasu na wykonanie sortowania, niż w finalnej wersji. Zdarzały się również przypadki, w których algorytm zakończył swoje działanie pozostawiając błędnie posortowaną listę elementów. Dzięki optymalizacji algorytmów i przystosowaniu ich pod odpowiednią strukturę danych, efekt końcowy jest znacznie lepszy.

Algorytm quicksort przy mniejszej ilości danych sortował w podobnym czasie co pozostałe algorytmy. Przy większym zbiorze elementów wartość ta rosła, co pokazuje, iż algorytm ten ma mniejszą wydajność niż pozostałe.

Merge sort ma stabilną wydajność przy dużej bazie danych, co pokazuje wykres stale rosnący. Według mnie nie jest do końca poprawny, ale pokazuje, że działa stabilnie. Wadą jest spora ilość wymaganej pamięci podczas wykonywania sortowania.

Sortowanie kubełkowe w tym przypadku jest najszybciej działającym algorytmem. Czas, w którym posortował elementy jest znacznie mniejszy od pozostałych. Wynika to z liniowej charakterystyki złożoności czasowej. Jednak przy nieznanej rozbieżności danych oraz nierównomiernego rozłożenia elementów nie ma możliwości implementacji sprawnie działającego bucket sorta.

Projekt miał na celu przedstawienie różnych algorytmów sortujących oraz zrozumienie, że każdy z nich ma swoje zalety i wady oraz może być wykorzystywany w różnych sytuacjach. Przy programowaniu aplikacji opartej na bazie danych warto jest zrobić testy wydajnościowe oraz porównać, który z algorytmów jest najbardziej optymalny.

Bibliografia

1. Encyklopedia Algorytmów, [online], Polska, 05.01.2018, aktualizacja 13.02.2019, dostępny w Internecie: https://www.algorytm.edu.pl/algorytmy-maturalne/sortowanie-przez-scalanie.html
2. Serwis Edukacyjny Nauczycieli I LO, [online], Polska, dostępny w Internecie: https://eduinf.waw.pl/inf/alg/003_sort/0018.php
3. Serwis Edukacyjny Nauczycieli I LO, [online], Polska, dostępny w Internecie: https://eduinf.waw.pl/inf/alg/003_sort/0010.php
4. Serwis Edukacyjny Nauczycieli I LO, [online], Polska, dostępny w Internecie: https://eduinf.waw.pl/inf/alg/003_sort/0020.php
5. GeeksForGeeks, [online], dostęp 09.12.2022, dostępny w Internecie: https://www.geeksforgeeks.org/quick-sort/
6. Simplilearn, [online], dostęp 23.02.2023, dostępny w Internecie: https://www.simplilearn.com/tutorials/data-structure-tutorial/bucket-sort-algorithm
7. GormAnalysis, [online], dostęp 16.01.2019, dostępny w Internecie: https://www.gormanalysis.com/blog/reading-and-writing-csv-files-with-cpp/