Sprawozdanie z algorytmów sortowania

Adam Piaseczny, Igor Szczepaniak

Marzec 2022

Spis treści

1 Wprowadzenie

W tym sprawozdaniu porównaliśmy ze sobą osiem różnych algorytmów sortowania w zróżnicowanych warunkach i przy określonych ilościach danych w nich zawartych. Analizując wyniki otrzymane podczas doświadczenia byliśmy w stanie sprawdzić w kontrolowanym środowisku, jakie są właściwości poszczególnych algorytmów i jak może to pomóc zastosowaniu ich w praktyce. Cały program został napisany w języku `python` z powodu prostoty implementacji funkcji testujących.

1.1 Generator tablic

Z racji powolnej natury list w języku `python`, postanowiliśmy użyć biblioteki `numpy` jako głównego narzędzia do generowania tablic. Pomogło to znacznie w procesie tworzenia pseudolosowych rozkładów liczb oraz ich analizie.

2 Analiza zadań

Do wykonania było zadanych łącznie 7 testów. Trzy z zadnia drugiego, dwa z trzeciego oraz dwa z czwartego.

2.1 Pierwszy test - wszystkie algorytmy

W tym teście zawarliśmy 20 prób, z wartościami n od 1000 do 20000 w rozkładzie losowym dla następujących prostych algorytmów sortowania:

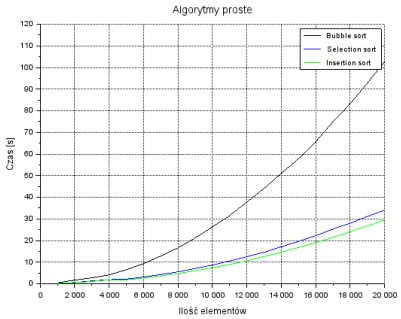
1. Selection Sort
2. Insertion Sort
3. Bubble Sort

Oraz dla czterech złożonych:

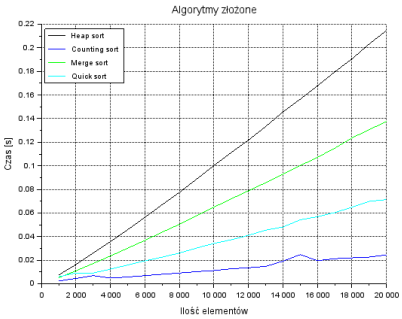
1. Heap Sort
2. Merge Sort
3. Counting Sort
4. Quicksort z podziałem według środkowego elementu

2.1.1 Rozkład losowy

Z racji dużych różnic w czasie wykonywania zrobiliśmy dwa osobne wykresy dla poszczególnych kategorii. Dane dla algorytmów prostych są zawarte w wykresie nr 1, a dla złożonych w wykresie nr 2.



Rysunek 1: $t = f(n)$ - proste

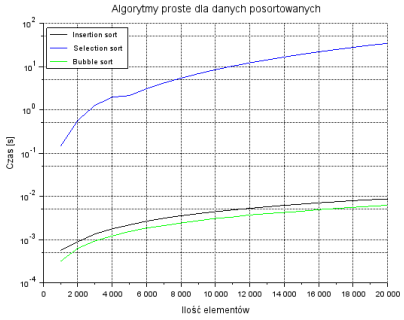


Rysunek 2: $t = f(n)$ - złożone

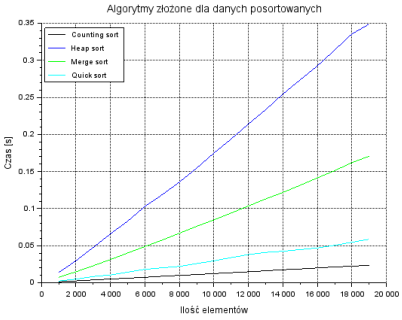
Analizując wykresy zauważyliśmy, że czas pracy algorytmów prostych znacznie odbiega od złożonych. Nie jest to zaskakujące, znając złożoność obliczeniową tych algorytmów w przypadku średnim. Wykres pokazuje w sposób graficzny, jak bardzo różnią się te algorytmy, widać to po samej skali w jakiej rozpatrywane są wyniki.

2.1.2 Rozkład rosnący

Przy rozkładzie rosnącym (posortowanej tablicy) od razu zauważamy, że Bubble Sort, który w poprzedniej próbie wypadł najgorzej, zakończył swoje wykonywanie w rekordowym czasie, ponieważ implementacja naszego algorytmu przewidywuje tzw. "flagę", która pozwala szybko wykryć częściowe posortowanie tablicy wejściowej.



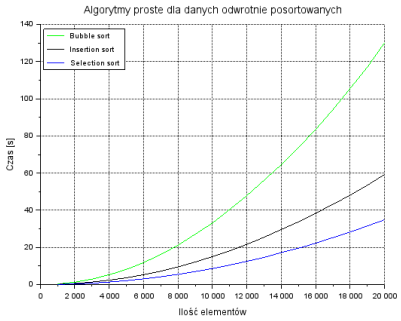
Rysunek 3: $t = f(n)$ - proste



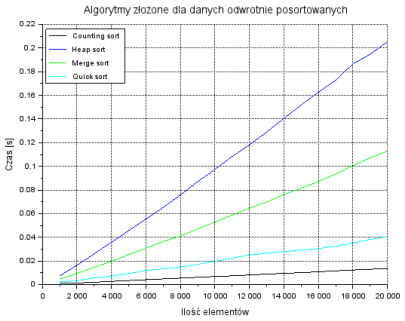
Rysunek 4: $t = f(n)$ - złożone

2.1.3 Rozkład malejący

Przy rozkładzie malejącym (tablicy odwrotnie posortowanej) zauważamy najgorszy przypadek dla algorytmów selection oraz bubble sort.



Rysunek 5: $t = f(n)$ - proste



Rysunek 6: $t = f(n)$ - złożone

2.1.4 Wnioski

Z wykonanych testów byliśmy w stanie potwierdzić, jakie są najgorsze oraz najlepsze przypadki dla poszczególnych algorytmów sortowania i dowiedzieć się, kiedy najlepiej ich używać w praktyce. Metody proste zwykle nie są efektywne dla dużej ilości danych lecz dla wszystkich istnieje zastosowanie. Dla każdego przypadku wypisaliśmy najlepszy według nas użytek dla danego algorytmu.

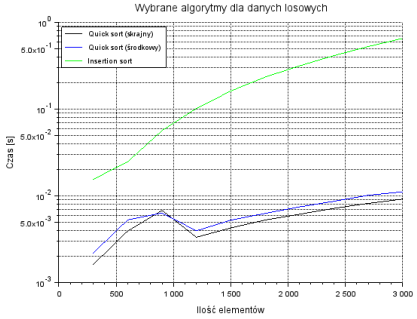
1. Insertion Sort - kiedy n jest małe
2. Selection Sort - kiedy potrzebna jest szybka implementacja
3. Bubble Sort - w przypadku prawie posortowanych tablic, dodatkowym atutem jest łatwość implementacji
4. Heap Sort - kiedy potrzebny jest stabilny algorytm i mamy ograniczony z góry czas
5. Merge Sort - kiedy potrzebne jest stabilne sortowanie $O(n \log n)$
6. Counting Sort - kiedy sortujemy liczby całkowite o małej rozpiętości
7. Quicksort (`pivot` na środku) - kiedy potrzeba szybkiego sortowania i średni przypadek ma duże znaczenie

2.2 Drugi test - szybkie oraz wstawanie

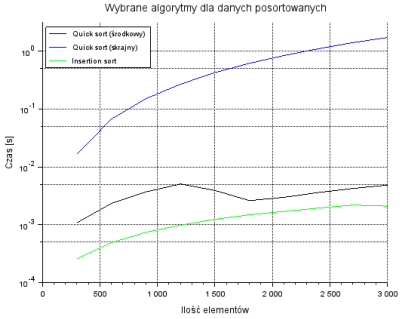
Tutaj porównane będą dwie wersje algorytmu Quick Sort (z podziałem według skrajnego i środkowego elementu tablicy) wraz z algorytmem sortowania przez proste wstawianie. Nasze testy są w zakresie wartości n od 300 do 3000 włącznie.

2.2.1 Rozkład losowy i rosnący

Przy rozkładzie losowym obie wersje Quick Sort mają porównywalne czasy wykonania, o wiele szybsze od sortowania przez proste wstawianie. Natomiast przy rozkładzie rosnącym zauważamy znaczne pogorszenie czasu obu sortowań szybkich, które w tym teście wypadają dużo gorzej niż prosty algorytm przez wstawianie. Warto zaznaczyć, że Quick Sort z podziałem według elementu skrajnego ma znacznie większy czas pracy niż wersja z elementem środkowym.



Rysunek 7: Rozkład losowy



Rysunek 8: Rozkład rosnący

2.2.2 Wnioski

Duży czas pracy sortowania z elementem skrajnym w przypadku dla danych rosnących jest wynikiem, rekurencyjnego wykonywania funkcji aż $n-1$ razy. Napotykając ten problem, byliśmy zmuszeni obniżyć wartości n w poszczególnych testach, jak również zwiększyć limit stosu rekurencyjnego. W tym właśnie przypadku sortowanie szybkie zajmowało skrajnie dużo pamięci, co było znacznym problemem w procesie sprawdzania jego zachowania.

W implementacji algorytmu Quick Sort niezbędne jest zastanowienie się nad sposobem wyboru elementu `pivot`, widząc jak bardzo wpływa on na efektywność algorytmu i jego najgorszy przypadek. Wybieranie tego elementu jako losowego w tablicy minimalizuje możliwość najgorszego przypadku, popularnym rozwiązaniem jest również wybranie trzech losowych indeksów i wybranie tego, który jest środkowym.

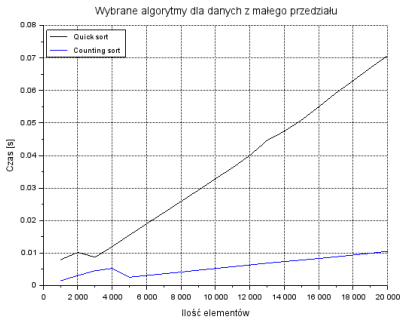
Metoda Insertion Sort jest zwykle wolniejsza od sortowania szybkiego, lecz ważnym faktem jest to, że jej najgorszy oraz średni najlepszy przypadek mają taką samą złożoność czasową, a najlepszy ma złożoność $O(n)$ co przydaje się mając prawie posortowane dane. Sortowanie przez wstawianie również nie wymaga dodatkowego nakładu pamięci - wykonuje się "w miejscu" podczas gdy sortowanie szybkie potrafi pochłonąć prawie cały stos rekurencyjny w najgorszym przypadku i jest bardzo zachłanny w kontekście pamięci. Quick Sort dla średniego przypadku jest za to o wiele bardziej efektywny i ma bardziej ogólne zastosowanie.

2.3 Trzeci test - rozpiętość przedziałowa danych

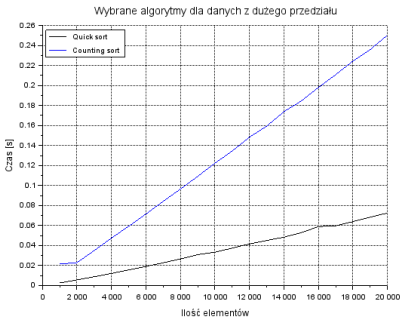
W tym teście sprawdzamy jak zachowują się dwa algorytmy - Counting Sort oraz Quick Sort (z podziałem środkowym) dla wartości n od 1000 do 20000 mając 20 puktów pomiarowych. Za to zakres wartości znacząco różni się miedzy testami.

2.3.1 Przentacja danych

Na pierwszym wykresie zakres danych mieści się w zakresie $[1, 100*n]$, na drugim za to widzimy zależność przy zakresie $[1, 0.01 * n]$.



Rysunek 9: Mała rozpiętość danych



Rysunek 10: Duża rozpiętość danych

2.3.2 Wnioski

Na pierwszym wykresie w konteście czasu wykonania przoduje Quick Sort, lepiej radząc sobie z dużą rozpiętością analizowanych danych, za to Counting Sort z powodu zasady swojego działania ma możliwość pokazania swojej "szybkiej strony" przy małej rozpiętości danych. Dzieje się tak z racji tworzenia mniejszej tablicy pomocniczej i operowania na niej. Warto również zauważyć, że rozpiętość danych prawie w ogóle nie wpływa na szybkość działania sortowania szybkiego, podczas gdy w sortowaniu przez zliczanie zauważamy ogromną różnicę. Przy doborze metod sortowania zależy dokładnie przyjrzeć się zakresowi rozpatrywanych danych, aby na pewno nie przeoczyć lepszego rozwiązania.

3 Podsumowanie

W sprawozdaniu porównaliśmy i przeanalizowaliśmy różne algorytmy sortujące, pokazaliśmy ich dobre i złe strony oraz jak zachowują się w poszczególnych warunkach w kontrolowanym środowisku. Kod do wykorzystanego programu oraz pliki z danymi wyjściowymi dostępne są pod następującą witryną:

https://github.com/TypicalAM/Sorting-Algorithms