

Algorytmy i struktury danych

Porównanie algorytmu sortowania bąbelkowego oraz szybkiego.

Mariusz Mańka

Informatyka III semestr, niestacjonarne

Wydział Matematyki Stosowanej

28 listopada 2020

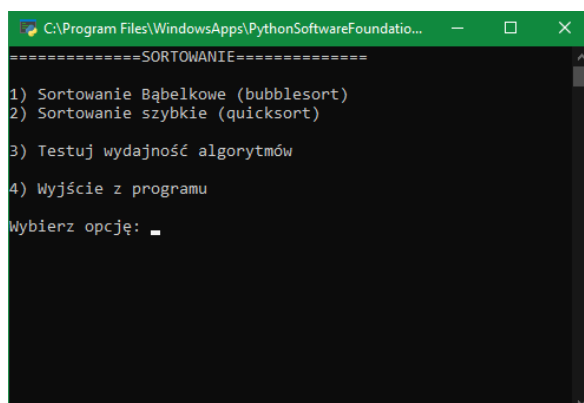
Część I

Opis programu

Program ma za zadanie posortować zbiór liczb od najmniejszej do największej używając do tego celu dwóch zaimplementowanych algorytmów - **sortowania bąbelkowego (bubblesort)** oraz **sortowania szybkiego (quicksort)**. Jako dodatkowa funkcjonalność programu została zaimplementowana opcja porównania wydajności czasowej jednego oraz drugiego algorytmu. Jedną z opcji w ramach testów pozwala posortować ten sam zbiór liczb za pomocą zarówno sortowania bąbelkowego jak i szybkiego, jako odpowiedź prezentując użytkownikowi czas w jakim poszczególne algorytmy poradziły sobie z tym zadaniem, pokazując przy tym wyższość algorytmu sortowania szybkiego nad bąbelkowym szczególnie dla dużych zbiorów liczbowych. Program również pozwala na to aby użytkownik zdefiniował ile liczb chce posortować, następnie uzupełnia plik tekstowy żadaną ilością losowo wygenerowanych liczb, po czym wczytuje te dane z pliku do tablicy którą w późniejszym czasie będą sortować oba algorytmy.

Instrukcja obsługi

Aby uruchomić program należy przejść za pomocą konsoli cmd do miejsca w którym mamy nasz skrypt z zadaniem. Następnie wpisać polecenie **python nazwa_programu.py**. Gdy to zrobimy program uruchomi się nam w konsoli. Naszym oczą ukaże się menu w którym będą widnieć cztery opcje: Pierwsza pozwoli na posortowanie zbioru liczb za pomocą **algorytmu sortowania bąbelkowego**, druga za pomocą **algorytmu quicksort**, trzecia pozwoli nam na **porównanie czasu** w jakim te dwa algorytmy będą w stanie posortować ten sam zbiór danych, a ostatnia czwarta opcja pozwoli na **wyjście z programu**. Wybierając opcje od 1 do 3 będziemy mogli zdecydować jak duży ma być zbiór liczb który chcemy posortować. Gdy już dokonamy wyboru do pliku tekstowego który znajduje się w lokalizacji **"Dane/Dane.txt"** zostanie zapisana wskazana przez nas ilość losowo wygenerowanych przez program liczb naturalnych z **przedziału od 1 do 1000**. Następnie program odczyta to liczby zapisze je w tablicy i posortuje je za pomocą wybranego przez nas algorytmu. Wynikiem działania programu będzie wyświetlenie posortowanej tablicy oraz informacji o tym ile to sortowanie zabrało naszej maszynie czasu.

A screenshot of a Windows command prompt window. The title bar shows the path 'C:\Program Files\WindowsApps\PythonSoftwareFoundatio...'. The window content displays a menu titled '=====SORTOWANIE====='. The menu options are: '1) Sortowanie Bąbelkowe (bubblesort)', '2) Sortowanie szybkie (quicksort)', '3) Testuj wydajność algorytmów', and '4) Wyjście z programu'. Below the menu, it says 'Wybierz opcję: ' followed by a cursor.

Rysunek 1: Przykładowe działanie programu

Dodatkowe informacje

Do prawidłowego działania tego programu potrzebujemy mieć zainstalowane środowisko uruchomieniowe pythona. Środowisko to możemy pobrać z [oficjalnej strony pythona](#). Zalecane jest również uruchamianie programu w środowisku Windows, gdyż bez tego niektóre funkcjonalności mogą przestać działać poprawnie.

Część II

Opis działania

Program ten wykorzystuje do działania kilka dobrze znanych programistą mechanik. Jedną z nich jest wykorzystanie generatora liczb pseudolosowych używanego do losowania liczb które oba algorytmy będą musiały posortować a zaimplementowanego z wbudowaną w pythona metodą `randomint()`. Algorytmy służące do generowania liczb pseudolosowych to w rzeczywistości seria obliczeń bazujących na liczbie, którą nazywamy **ziarnem**. Taka liczba poddana serii działań matematycznych stanowi podstawę do wytwarzania liczb przypominających losowe. Obecnie najczęstszą praktyką jest pobieranie czasu systemowego jako owego ziarna do generowania liczb pseudolosowych gdyż czas ten zawsze będzie inny. Taką praktykę stosują również pseudolosowe metody wbudowane w pythona służące do generowania liczb. Liczby te nazywane są pseudolosowymi głównie z jednego powodu - gdy podamy takie samo ziarno przy uruchomieniu takiej metody otrzymamy takie same wyniki, są one **zależne od ziarna które podamy**. Program korzysta również z obiektu `datetime` z metodą `now()` która pobiera aktualny czas systemowy w celu obliczenia czasu jaki jest potrzebny do wykonania się obu algorytmów.

Algorytm sortowania bąbelkowego (bubblesort)

Algorytm ten odznacza się dużą prostotą w koncepcji działania oraz w implementacji. Jest to jeden z podstawowych oraz najprostszych algorytmów sortowania, jednak nie jest on zbyt wydajny, ze względu na dużą liczbę powtarzających się operacji. Jego złożoność czasowa w notacji **dużego O** to $O(x^2)$ co czyni go praktycznie bezużytecznym dla większych zbiorów liczbowych. Sama idea działania algorytmu jest prosta: Porównujemy ze sobą każde dwie kolejne liczby poczynawszy od pierwszych dwóch. Następnie w przypadku gdy liczba stojąca na pierwszej pozycji jest większa niż ta stojąca na drugiej, **zamieniamy je miejscami** tak aby największa liczba w danym przebiegu znalazła się zawsze na samym końcu tablicy czyli tam gdzie znaleźć się powinna. Algorytm kończy działania gdy wszystkie liczby będą już posortowane. W najgorszym wypadku program dla *n-elementowego* zbioru wykona się **n - 1 razy**. W podstawowej i nieoptymalizowanej wersji tego algorytmu pętla sortująca będzie się wykonywać nawet w momencie gdy wszystkie liczby są już posortowane.

Data: Dane wejściowe tablica nieposortowanych liczb *collection*

Result: Posortowana tablica liczb *collection*

START:

$n := \text{length}(\text{collection});$

$i := 0;$

$j := 0;$

for $i < n$ **do**

for $j < n - i - 1$ **do**

if $\text{collection}[j] > \text{collection}[j + 1]$ **then**

 Zamień $\text{collection}[j]$ z $\text{collection}[j + 1]$

else

 Kontynuuj pętle

end

$j++$

end

$i++$

end

Zwróć *collection*;

STOP;

Algorithm 1: Algorytm sortowania bąbelkowego - Bubblesort

Algorytm sortowania szybkiego (quicksort)

Algorytm Sortowania szybkiego jest znacznie wydajniejszy od algorytmu sortowania bąbelkowego. Różne jego odmiany są bardzo często stosowane w różnego rodzaju programach oraz bibliotekach. Sprawdza się on w szczególności jeśli chodzi o sortowanie bardzo dużych zbiorów liczb. Jego złożoność czasowa w notacji **dużego O** wynosi $O(n \log n)$ co czyni go jednym z najlepszych algorytmów sortujących. Jest oparty na zasadzie **'Dziel i zwyciężaj'** w myśl której jeden większy problem dzielimy na mniejsze "podproblemy" które będą dla nas łatwiejsze do rozwiązania. I tak jeśli chodzi o ten algorytm głównym jego zadaniem jest podział zbioru liczb na wiele mniejszych podzbiorów według ustalonego wcześniej elementu zwanego **pivotem** oraz gdy to okaże się konieczne przestawienie liczb w danym podciągu miejscami tak aby mniejsza zawsze znajdowała się "przed" większą. W jednej tablicy znajdują się elementy mniejsze od *pivota* a w drugiej elementy większe. Proces ten jest powtarzany do czasu aż podzbiory będą jednoelementowe. W tym momencie otrzymamy naszą posortowaną tablicę, gdyż zbiór jednoelementowy z założenia jest już posortowany. W praktyce więc oznacza to że cała nasza tablica została podzielona na pojedyncze elementy z których każdy jest posortowany. W procesie dzielenia naszej wejściowej tablicy na mniejsze elementy bardzo dobrze objawia się **rekurencja** tego algorytmu. **Rekurencja** jest to wywoływanie funkcji przez samą siebie. Po każdym przejściu pętli po naszej tablicy, oraz oddzieleniu elementów mniejszych od *pivota* i większych od niego, funkcja wywołuje się po raz kolejny tym razem dla podzielonych elementów i robi to aż do czasu gdy wszystkie elementy zostaną podzielone i posortowane. Kolejną ważną kwestią jeśli chodzi o ten algorytm jest dobór **pivota** czyli elementu do którego porównujemy kolejne elementy tablicy sprawdzając czy są one od niego większe i tym samym decydując o dalszym ich podziale.

Najczęściej *pivotem* zostaje pierwszy element w danej tablicy, jednak jeśli będziemy mieli do czynienia ze zbiorami posortowanymi odwrotnie niż chcemy je posortować taki wybór *pivota* będzie katastrofalną decyzją prowadząc do najbardziej pesymistycznego scenariusza gdzie każdy element będzie musiał zostać podzielony i zamieniony miejscem z kolejnym. Skutkiem tego jest rosnąca do $O(n^2)$ złożoność czasowa algorytmu. Dlatego lepszym sposobem ustalenia *pivota* jest wybranie elementu który znajduje się na środkowym miejscu tablicy. Często wybiera się również ten element losowo gdyż zwiększa to szansę na to że nie będziemy musieli dokonywać tylu operacji gdy dobrze trafimy z naszym wyborem.

Data: Dane wejściowe tablica nieposortowanych liczb *collection*

Data: Index pierwszego elementu w tablicy *left* := 0

Data: Index ostatniego elementu w tablicy *right* := *length(collection)* - 1

Result: Posortowana tablica liczb *collection*

pivot := (*left* + *right*)/2;

i := *left* *j* := *right* **while** *i* <= *j* **do**

while *collection*[*i*] < *pivot* **do**

 | *i* ++

end

while *collection*[*j*] > *pivot* **do**

 | *j* --

end

if *i* <= *j* **then**

 | Zamień *collection*[*i*] oraz *collection*[*j*] miejscami;

 | *i* ++;

 | *j* --;

else

end

if *left* < *j* **then**

 | Wywołaj Quicksort z parametrami *left* := *left*, *right* := *j*

else

end

if *right* > *i* **then**

 | Wywołaj Quicksort z parametrami *left* := *i*, *right* := *right*

else

end

Zwróć *collection*;

STOP;

Algorithm 2: Algorytm sortowania szybkiego - Quicksort

Implementacja

Działanie programu jest oparte w pełni na funkcji sterującej **main** oraz klasie sterującej **CompareSorting**. Zadaniem funkcji **main()** jest interakcja z użytkownikiem i odpowiednie reagowanie na wprowadzane przez niego dane. Funkcja ta posiada proste menu które pozwala użytkownikowi na wybranie algorytmu którym chce posortować wygenerowane przez program dane lub na porównanie szybkości sortowania tych samych danych przez oba algorytmy. Na podstawie wprowadzonych przez użytkownika danych na samym początku zostanie zainicjowana instancja klasy **CompareSorting** na podstawie której nasz program będzie wykonywać dalsze instrukcje. W dalszej kolejności zostaną uruchomione dwie metody: **generate_random_numbers()** oraz **get_data_from_file**. Pierwsza z metod ma za zadanie otwarcie pliku tekstowego w trybie do zapisu oraz za pomocą pętli w której jednym z argumentów jest podana przez użytkownika na samym początku ilość liczb dla których algorytm ma się wykonać, zapisuje do pliku tekstowego żadaną ilość losowo wygenerowanych liczb w zakresie od 1 do 1000. Drugi z kolei algorytm ma za zadanie otworzyć plik w trybie do odczytu oraz wpisać do tablicy wszystkie przeczytane liczby - to właśnie z tej tablicy liczb będą korzystać nasze algorytmy na dalszym etapie wykonywania kodu. Na sam koniec za pomocą instrukcji **if..else** zostanie uruchomiona odpowiednia metoda z klasy **CompareSorting** w zależności od tego którą opcję wybrał użytkownik, po czym program zakończy swoje działanie, poniżej przedstawiam kod funkcji **main()**, a w dalszej części zajmę się opisem klasy sterującej **CompareSorting**:

```
1 def main():
2     system("cls")
3     print("=====SORTOWANIE=====\\n")
4     print("1) Sortowanie Babelkowe (bubblesort)")
5     print("2) Sortowanie szybkie (quicksort)")
6     print("3) Testuj wydajność algorytmów")
7     print("4) Wyjście z programu\\n")
8     try:
9         choice = int(input("Wybierz opcję: "))
10    except ValueError:
11        print("Podano złą wartość!")
12        return
13
14    if choice == 4:
15        system("cls")
16        print("Dziękuję za skorzystanie z programu!")
17        return
18
19    system("cls")
20    try:
21        amount = int(input("Jak wiele liczb chcesz posortować? "))
22    except:
23        print("Podano złą wartość!")
24        return
25
26    compareSorting = CompareSorting(amount)
27    compareSorting.generate_random_numbers()
28    compareSorting.get_data_from_file()
29
30
```

```

31     if choice == 1:
32         collection, duration = compareSorting.bubblesort()
33         print("Sortowanie babelkowe (bubblesort: ", collection)
34         print("Algorytm wykonał się w czasie: ", duration)
35         system("pause")
36         system("cls")
37     elif choice == 2:
38         collection, duration = compareSorting.quicksort(
39             0, len(compareSorting.collection)-1)
40         print("Sortowanie szybkie (quicksort): ", collection, )
41         print("\n\nAlgorytm wykonał się w czasie: ", duration)
42         system("pause")
43         system("cls")
44     elif choice == 3:
45         n = (len(compareSorting.collection) - 1)
46         collection, b_duration = compareSorting.bubblesort()
47         collection, q_duration = compareSorting.quicksort(0, n)
48         print(
49             f"Dla {amount} liczb, algorytm 'bubblesort' wykonał się w
              czasie: { b_duration} natomiast 'quicksort' w czasie: {
              q_duration}")
50     else:
51         print("Nie ma takiej opcji!")
52
53
54 if __name__ == '__main__':
55     main()

```

Klasa sterująca **CompareSorting** posiada 4 metody oraz złożona jest z konstruktora który jako jedyny parametr przyjmuje wprowadzoną przez użytkownika zmienną **amount** która informuje nas o tym dla jakiej ilości danych użytkownik chce przetestować algorytmy. Oprócz tej informacji w konstruktorze znajdują się trzy inne zmienne, zmienna **collection** jest z naszego punktu widzenia tą najważniejszą, to ona ma przechowywać wczytane z pliku liczby i być użyta przy sortowaniu do uporządkowania liczb znajdujących się w niej. Na samym początku zainicjowana jest jako pusta tablica. Dwie kolejne zmienne to zmienna **path** oraz zmienna **encoding** które kolejno przechowują informacje o ścieżce dostępowej do naszego pliku oraz o kodowaniu w którym chcemy go otworzyć (plik z danymi). Zadaniem czterech pozostałych metod które posiada nasza klasa jest obsługa zapisu do pliku losowo wygenerowanych liczb, odczyt danych z pliku i zapisanie danych do tablicy oraz obsługa dwóch rodzajów sortowania - bąbelkowego i szybkiego. Metoda **generate_random_numbers()** otwiera plik w trybie do zapisu następnie przechodzi do pętli **for** która wykonuje się od **0** do zadanej przez użytkownika ilości z atrybutu **amount**. Za pomocą metody z zaimportowanej w pythona klasy **random** - **randomint()** w ciele pętli generowana jest zadana ilość danych w postaci **losowych liczb stałoprzecinkowych** które następnie oddzielone przecinkiem wpisywane są do otwartego pliku. Za pomocą instrukcji warunkowej **if ... else** sprawdzane jest czy generowana liczba nie jest ostatniąadaną, jeśli jest nie zapisujemy przy niej przecinka gdyż to generuje błędy w odczycie i zapisie tych liczb do tablicy w kolejnej metodzie. Na sam koniec plik jest zamykany instrukcją **close()**, zapobiega to niepożądanym efektom pozostawienia pliku otwartego oraz wycieką pamięci z aplikacji. Poniżej przedstawiam kod metody **generate_random_numbers()**:

```

1     def generate_random_numbers(self):
2         try:
3             f = open(self.path, "w", encoding=self.encoding)
4         except IOError:
5             print("Bład podczas otwarcia pliku! ")
6             return
7         for i in range(1, self.amount):
8             random_num = random.randint(1, 1000)
9             if i != self.amount-1:
10                f.write(f"{random_num},")
11            else:
12                f.write(f"{random_num}")
13        f.close()

```

Kolejną funkcją wartą uwagi jest funkcja `get_data_from_file`. Otwiera ona plik w trybie do odczytu następnie za pomocą stosownej instrukcji `for` iteruje się po każdej linii pliku. Następnie za pomocą kolejnej pętli `for` z każdej linii odczytywany każdy kolejny oddzielony od siebie przecinkiem łańcuch znakowy - to są nasze liczby, które następnie zostają parsowane na typ liczbowy `int` co umożliwi algorytmom sortującym pracę z nimi. Oddzielenie dokonywane jest za pomocą instrukcji `split(",")` gdzie jako argument podany jest separaor, w tym przypadku przecinek. Następnie każda liczba jest dodawana do naszej tablicy. Program na sam koniec zamyka plik a następnie do atrybutu `collection` naszej klasy przypisuje odczytaną z pliku tablicę nieposortowanych liczb. Kod metody poniżej:

```

1     def get_data_from_file(self):
2         tab = []
3         with open(self.path, 'r', encoding=self.encoding) as file:
4             for line in file:
5                 if line.split():
6                     line = [int(x) for x in line.split(",")]
7                     tab.append(line)
8         file.close()
9         self.collection = tab[0]

```

Pierwszą z naszego punktu widzenia najważniejszych metod w tym programie jest metoda obsługująca sortowanie bąbelowe - **bubblesort**. Na samym jej początku jeszcze przed przystąpieniem do realizacji właściwego kodu wywoływana jest metoda z wbudowanej do pythona klasy `datetime`. Funkcja `now()` tej metody pozwala na pobranie aktualnego czasu. Jest to używane do obliczenia w jakim czasie algorytm się wykona gdyż po wykonaniu wszystkich pętli funkcja ta jest wywoływana ponownie. Odejmując czas przed wykonaniem wszystkich instrukcji od czasu po ich wykonaniu otrzymujemy liczbę sekund która reprezentuje czas wykonywania się naszego algorytmu. Ta liczba jest również zwracana z naszej funkcji wraz z posortowaną tablicą. Po rozpoczęciu mierzenia czasu, nasz program przechodzi do konkretnego bloku kodu pozwalającego na zrealizowanie sortowania bąbelkowego. Kod jak i sam algorytm jest prosty. Na początku ustawiane są **dwie pętle for**. Pierwsza z nich iteruje od początku do końca naszej tablicy po każdym z elementów, natomiast druga pętla pozwala na iterację po każdym kolejnym elemencie tablicy aż do końca. W środku wewnętrznej (czyli drugiej) pętli znajduje się warunek odpowiedzialny za sortowanie liczb. **Jeśli liczba w miejscu n będzie większa niż liczba w miejscu $n+1$ algorytm dokonuje ich zamiany miejscami** w wyniku czego w każdym obiegu pętli na samym jej końcu znajdzie się liczba

największa. Gdy wszystkie pętle zakończą swoje działanie cała tablica zostanie posortowana. Po zakończeniu sortowania metoda kończy działanie zwracając posortowaną tablicę oraz czas jaki zajęło jej to sortowanie. Kod metody poniżej:

```
1
2     def bubblesort(self):
3         start = datetime.datetime.now()
4         n = len(self.collection)
5         for i in range(n):
6             for j in range(0, n-i-1):
7                 if self.collection[j] > self.collection[j+1]:
8                     self.collection[j], self.collection[j +
9                                                         1] = self.
                                                         collection[j
                                                         +1], self.
                                                         collection[j]
10
11         duration = datetime.datetime.now() - start
12         return self.collection, duration
```

Ostatnią metodą naszej klasy jest metoda odpowiedzialna za sortowanie szybkie - **quicksort()**. Na samym początku jej działania analogicznie jak w metodzie **bubblesort()** rozpoczynamy mierzenie czasu. W jakim wykona się nasz algorytm za pomocą odpowiedniej funkcji klasy **datetime**. Nasza metoda wywoływana jest zawsze z dwoma parametrami - **left** oraz **right** które będą oznaczały końce przedziałów sortowania. W pierwszym obiegu parametr *left* będzie wskazywał na początek naszej tablicy, natomiast *right* na jej koniec. Przed rozpoczęciem bloku kodu zawierającego pętlę przypisujemy zmiennym **i** oraz **j** parametry **left** i **right** oraz ustalamy co będzie naszym **pivotem**. W celach optymalizacyjnych znajdujemy środkowy element naszej tablicy **collection** dzieląc krańce przedziału *left* oraz *right* przez 2. Następnie ustalamy że nasz *pivot* jest elementem środkowym tablicy *collection* oraz po tym kroku rozpoczynamy pierwszą pętlę **while** która będzie wykonywać się dopóki dopóty końce przedziałów nie będą sobie równe, czyli w naszym przypadku dopóki zmienna *j* będzie większa lub równa zmiennej *i*. W kolejnym kroku za pomocą dwóch osobnych następujących po sobie pętli **while**, szukamy pierwszej liczby z lewej strony tablicy poruszając się licznikiem *i* ku wartości **pivot**, która jest **nie mniejsza niż pivot** oraz pierwszej liczby z prawej strony, która jest **nie większa niż pivot**. Po tej operacji sprawdzamy instrukcją warunkową **if** czy nasze liczniki się minęły, jeżeli nie to zamieniamy ze sobą miejscami elementy leżące po niewłaściwej stronie elementu **pivot** a następnie zwiększamy *i* o jeden natomiast *j* zmniejszamy o tą samą wartość. Następnie dochodzimy do miejsca w którym objawia się rekurencyjny charakter naszego algorytmu. Jeśli parametr **left** jest mniejszy od naszego parametru **j** wywołujemy jeszcze raz naszą funkcję **quicksort**, tym razem ze zmienionym przedziałem od **left** aż do **j**. W praktyce oznacza to że wywołujemy naszą funkcję dla podzbioru naszej głównej tablicy który zawiera elementy mniejsze od **pivota** (od początku do wartości na lewo od pivota). Funkcja ta powtarzana jest do czasu aż nasza tablica liczb mniejszych od **pivota** będzie już posortowana, analogicznie dzieje się z liczbami na prawo od **pivota**, z tą różnicą że warunkiem do ponownego wywołania się funkcji jest to żeby zmienna *i* była większa od parametru **right**. W takim przypadku funkcja zostanie wywołana ponownie na przedziale od *i* do *right* czyli liczb większych od **pivota**. Metoda kończy działanie gdy przedział **left** będzie mniejszy lub równy **right**. Oznacza to że wszystkie elementy tablicy zostały posortowane a funkcja może ją zwrócić wraz z czasem w jakim ją posortowała. Kod funkcji poniżej:

```

1      def quicksort(self, left, right):
2          start = datetime.datetime.now()
3          i, j = left, right
4          middle = int((left + right) / 2)
5          pivot = self.collection[middle]
6          while i <= j:
7              while self.collection[i] < pivot:
8                  i += 1
9              while self.collection[j] > pivot:
10                 j -= 1
11             if i <= j:
12                 self.collection[i], self.collection[j] = self.collection
13                     [j], self.collection[i]
14                 i += 1
15                 j -= 1
16         if left < j:
17             self.quicksort(left, j)
18         if right > i:
19             self.quicksort(i, right)
20         duration = datetime.datetime.now() - start
21         retu

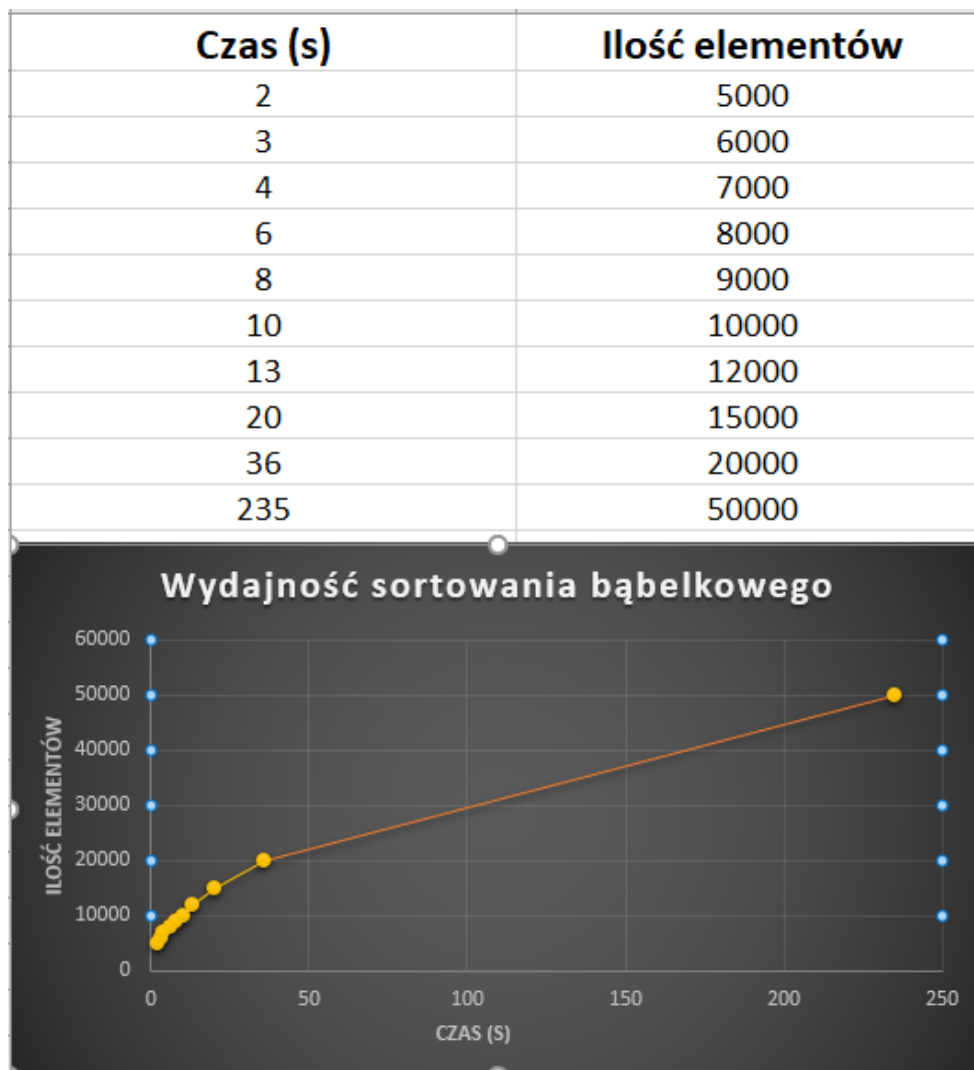
```

Testy

Dwa zaimplementowane przeze mnie algorytmy mają różną złożoność czasową wyrażoną w **notacji dużego O**. Dla algorytmu sortowania bąbelkowego złożoność czasowa wynosi: $O(n^2)$ natomiast dla sortowania szybkiego wynosi $O(n \log n)$. W przedstawionym programie podczas sortowania naprawdę dużych zbiorów liczbowych naprawdę dobrze widać miażdżącą przewagę sortowania szybkiego nad bąbelkowym. Dzięki zaimplementowanemu w obu funkcjach mechanizmowi pozwalającemu zmierzyć czas wykonania się poszczególnych algorytmów dla losowo wygenerowanego zbioru liczb możemy z całym przekonaniem stwierdzić że algorytm sortowania bąbelkowego nie nadaje się do sortowania dużych zbiorów liczbowych, gdyż posortowanie zbioru złożonego z **50000** elementów zajęło mu **prawie 4 minuty** podczas gdy algorytm sortowania bąbelkowego poradził sobie z tym zadaniem w nieco **ponad 0,1 sekundy**. Na tym przykładzie możemy zaobserwować kolosalną różnicę w wydajności tych dwóch algorytmów. Warto również zaznaczyć że mechanizm mierzący czas w programie jest skonstruowany tak aby mierzył bezpośrednio wykonanie się samych algorytmów, więc w naszej analizie możemy całkowicie pominąć odrębne kwestie związane z zapisem wylosowanych danych do pliku, odczytaniem ich z pliku do tablicy czy również wypisaniem posortowanych wartości na ekran, czas który przedstawiony jest na poniższych wykresach obejmuje samo wykonanie się algorytmów. Podsumowując, algorytm sortowania bąbelkowego przestaje być wydajny dla zbiorów powyżej **50000** elementów, nawet z mniejszymi o połowę zbiorami zaczyna on już mieć problem. Inaczej rzecz ma się w przypadku algorytmu sortowania szybkiego, tutaj doskonale widać wydajność zawdzięczaną **rekurencji oraz podejściu dziel i zwyciężaj**. Algorytm ten nie przestaje być wydajny nawet dla ogromnych zbiorów danych z którymi algorytm sortowania bąbelkowego nie dałby sobie rady w zadowalającym czasie. Pierwsze spadki wydajności możemy zauważyć sortując zbiory większe niż **1000000** elementów, chociaż i przy nich radzi sobie naprawdę dobrze. Dopiero sortując zbiór o wielkości **5000000** nasz algorytm zanotował

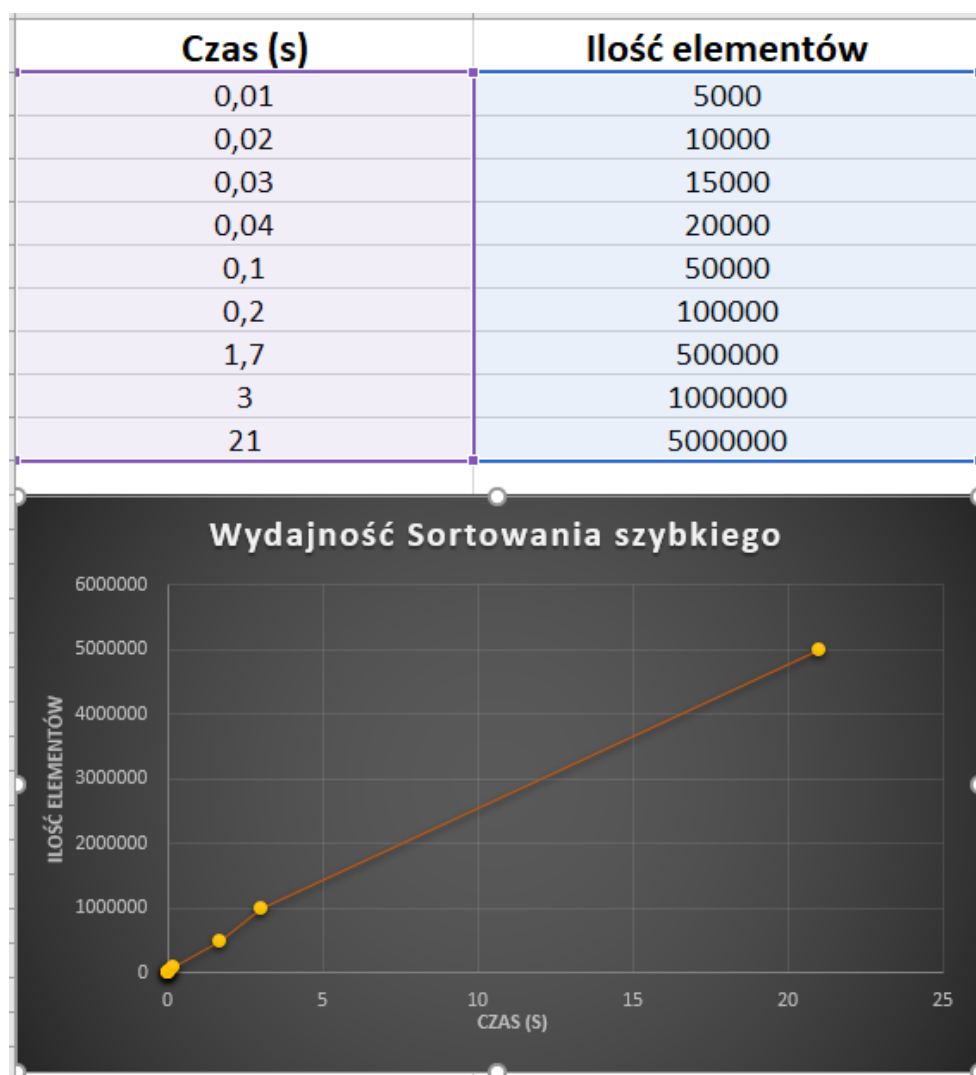
nieznaczny spadek wydajności, ale biorąc pod uwagę wielkość zbioru, dalej jest to naprawdę imponujący wynik. Poniżej przedstawiam tabelę z wykresem zależności czasu potrzebnego do wykonania algorytmu z ilością sortowanych elementów:

Wydajność sortowania bąbelkowego:



Rysunek 2: Wykres przedstawiający wydajność sortowania bąbelkowego

Wydajność sortowania szybkiego:

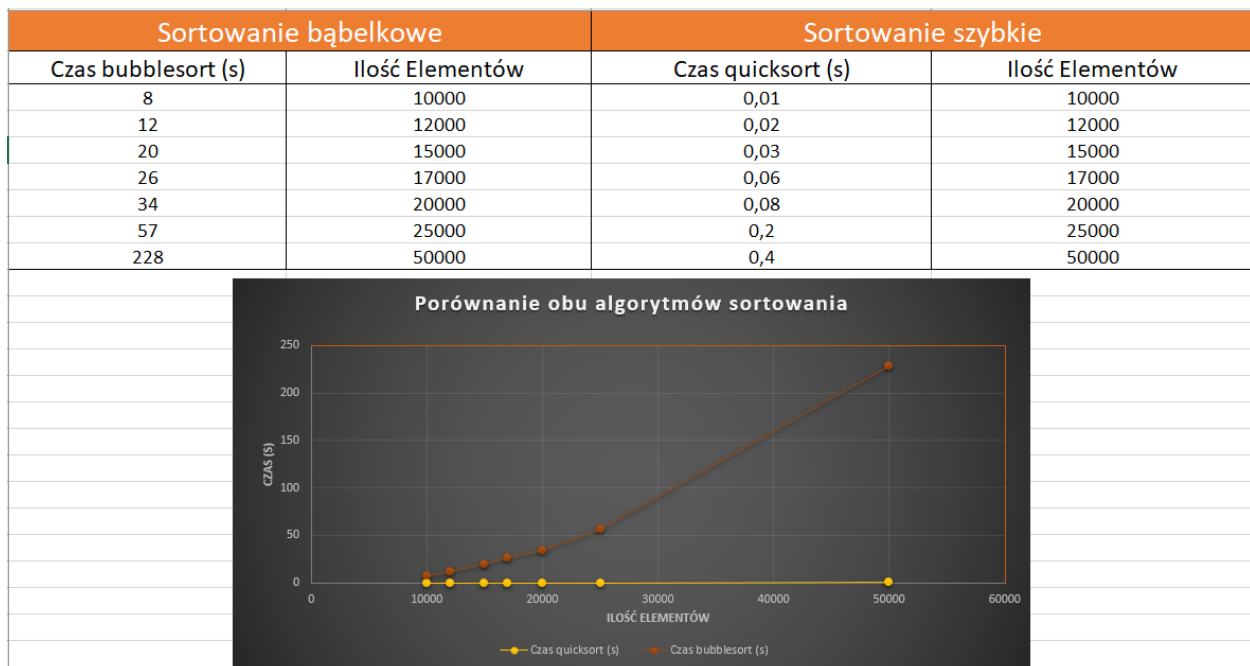


Rysunek 3: Wykres przedstawiający wydajność sortowania szybkiego

Eksperymenty

Nic tak dobrze nie obrazuje wyższości sortowania szybkiego nad bąbelkowym jak zestawienie razem na jednym wykresie czasu w jakim oba algorytmy są w stanie posortować ten sam zbiór danych. Moja aplikacja ma zaimplementowaną funkcję porównania tych dwóch algorytmów sortowania pod względem czasowym. Podczas uruchamiania tej opcji algorytmy działają na tej samej tablicy, która swe źródło czerpie z liczb losowo wygenerowanych przez algorytm do pliku, więc eliminujemy w tym porównaniu przypadek w którym jeden algorytm dostanie łatwiejsze do posortowania dane co prowadziło by do przekłamania w wynikach. Korzystając więc z tej opcji zapisałem czasy które zajęły naszym algorytmom posortowanie podanych przeze mnie wielkości tablic. Jak możemy zauważyć na poniższym wykresie podczas gdy używanie algorytmu sortowania bąbelkowego przestaje mieć sens, gdyż czas w jakim się on

wykonał dąży do 4 minut, drugi algorytm sortowania szybkiego wydaje się nie mieć najmniejszych problemów z posortowaniem naszego zbioru. Przewagę jednego algorytmu nad drugim doskonale obrazuje wykres poniżej:



Rysunek 4: Wykres zestawiający czasy potrzebne do posortowania danego zbioru danych z podziałem na poszczególne algorytmy

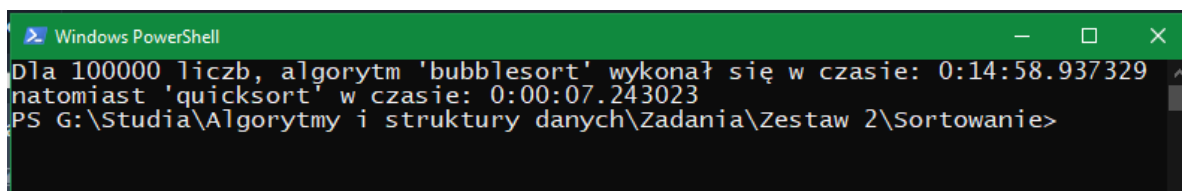
```

Windows PowerShell
Jak wiele liczb chcesz posortować? 50000
Dla 50000 liczb, algorytm 'bubblesort' wykonał się w czasie: 0:03:48.097340
natomiast 'quicksort' w czasie: 0:00:04.111702
PS G:\Studia\Algorytmy i struktury danych\Zadania\Zestaw 2\Sortowanie>

```

Rysunek 5: Wynik działania programu dla opcji porównującej dwa algorytmy

Dla zbioru zawierającego **50000** elementów czas wykonania się algorytmu bąbelkowego wynosi aż **3 min i 48 sekund** podczas gdy sortowanie szybkie poradziło sobie z nim w **nieco ponad 4 sekundy**. Pójdźmy o krok dalej i zwiększymy zbiór danych dwukrotnie:

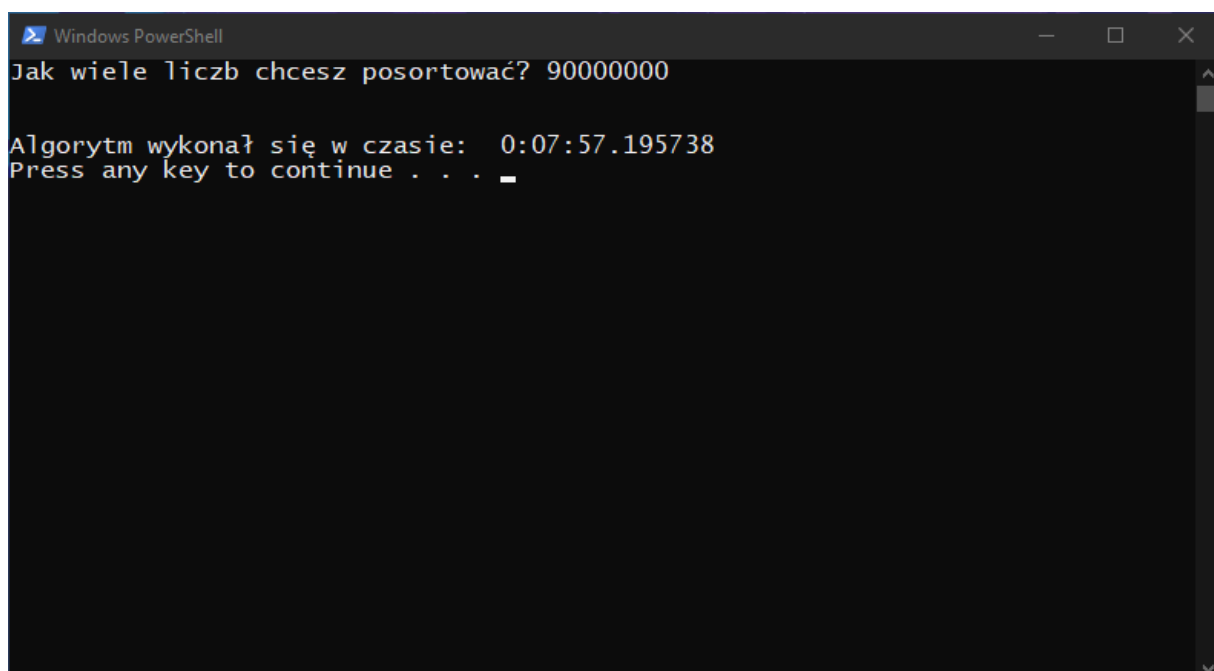


```
Windows PowerShell
Dla 100000 liczb, algorytm 'bubblesort' wykonał się w czasie: 0:14:58.937329
natomiast 'quicksort' w czasie: 0:00:07.243023
PS G:\Studia\Algorytmy i struktury danych\Zadania\Zestaw 2\Sortowanie>
```

Rysunek 6: Wynik porównania obu algorytmów dla zbioru 100000 elementów

Biorąc pod uwagę czas jaki był potrzebny do wykonania operacji przez algorytm sortowania bąbelkowego (**prawie 15 min.**) dalsze porównywanie ich na coraz to większych zbiorach nie ma sensu.

Jednak jeszcze ciekawszym zagadnieniem może być wydajność algorytmu sortowania szybkiego - dla jak dużych zbiorów straci na swojej wydajności? I tak sprawdziłem wydajność tego algorytmu dla zbioru danych złożonych z aż **10000000 losowych liczb**. Nawet to nie stanowiło dla niego problemu gdyż wykonał się w czasie **49,3 sekundy**. Widać tu jednak pierwsze spadki wydajności, dlatego przetestowałem ten algorytm dla znacznie większego zbioru - **30000000**, z którym również sobie poradził w czasie **2 min 32 sekundy**. Ostatnim zbiorem dla którego testowałem swój algorytm jest zbiór - **90000000 liczb**. Tutaj widać już spadek wydajności, jednak biorąc pod uwagę wielkość zbioru i ograniczenia obliczeniowe maszyny na której algorytm był testowany wciąż jest to zaskakująco dobry wynik- **7 minut i 57 sekund**.



```
Windows PowerShell
Jak wiele liczb chcesz posortować? 90000000

Algorytm wykonał się w czasie: 0:07:57.195738
Press any key to continue . . .
```

Rysunek 7: Wynik sortowania za pomocą algorytmu quicksort dla zbioru 90000000 elementów

Pełen kod aplikacji

```
1 from os import system
2 import random
3 import datetime
4
5
6 class CompareSorting:
7
8     def __init__(self, amount):
9         self.collection = []
10        self.amount = amount
11        self.path = "Dane/Dane.txt"
12        self.encoding = "utf-8"
13
14    def quicksort(self, left, right):
15        start = datetime.datetime.now()
16        i, j = left, right
17        middle = int((left + right) / 2)
18        pivot = self.collection[middle]
19        while i <= j:
20            while self.collection[i] < pivot:
21                i += 1
22            while self.collection[j] > pivot:
23                j -= 1
24            if i <= j:
25                self.collection[i], self.collection[j] = self.collection
26                    [j], self.collection[i]
27                i += 1
28                j -= 1
29            if left < j:
30                self.quicksort(left, j)
31            if right > i:
32                self.quicksort(i, right)
33            duration = datetime.datetime.now() - start
34            return self.collection, duration
35
36    def bubblesort(self):
37        start = datetime.datetime.now()
38        n = len(self.collection)
39        for i in range(n):
40            for j in range(0, n-i-1):
41                if self.collection[j] > self.collection[j+1]:
42                    self.collection[j], self.collection[j +
43                        1] = self.
44                            collection[j
45                                +1], self.
46                                    collection[j]
47
48            duration = datetime.datetime.now() - start
49            return self.collection, duration
```

```

49     def get_data_from_file(self):
50         tab = []
51         with open(self.path, 'r', encoding=self.encoding) as file:
52             for line in file:
53                 if line.split():
54                     line = [int(x) for x in line.split(",")]
55                     tab.append(line)
56         file.close()
57         self.collection = tab[0]
58
59     def generate_random_numbers(self):
60         try:
61             f = open(self.path, "w", encoding=self.encoding)
62         except IOError:
63             print("Bład podczas otwarcia pliku! ")
64             return
65         for i in range(1, self.amount):
66             random_num = random.randint(1, 1000)
67             if i != self.amount-1:
68                 f.write(f"{random_num},")
69             else:
70                 f.write(f"{random_num}")
71         f.close()
72
73
74     def main():
75         system("cls")
76         print("=====SORTOWANIE=====\\n")
77         print("1) Sortowanie Babelkowe (bubblesort)")
78         print("2) Sortowanie szybkie (quicksort)")
79         print("3) Testuj wydajnosć algorytmow")
80         print("4) Wyjście z programu\\n")
81         try:
82             choice = int(input("Wybierz opcje: "))
83         except ValueError:
84             print("Podano zła wartość!")
85             return
86
87         if choice == 4:
88             system("cls")
89             print("Dziękuję za skorzystanie z programu!")
90             return
91
92         system("cls")
93         try:
94             amount = int(input("Jak wiele liczb chcesz posortować? "))
95         except:
96             print("Podano zła wartość!")
97             return
98
99         compareSorting = CompareSorting(amount)
100         compareSorting.generate_random_numbers()
101         compareSorting.get_data_from_file()
102
103

```



```

104     if choice == 1:
105         collection, duration = compareSorting.bubblesort()
106         print("Sortowanie babelkowe (bubblesort: ", collection)
107         print("Algorytm wykonał się w czasie: ", duration)
108         system("pause")
109         system("cls")
110     elif choice == 2:
111         collection, duration = compareSorting.quicksort(
112             0, len(compareSorting.collection)-1)
113         print("Sortowanie szybkie (quicksort): ", collection, )
114         print("\n\nAlgorytm wykonał się w czasie: ", duration)
115         system("pause")
116         system("cls")
117     elif choice == 3:
118         n = (len(compareSorting.collection) - 1)
119         collection, b_duration = compareSorting.bubblesort()
120         collection, q_duration = compareSorting.quicksort(0, n)
121         print(
122             f"Dla {amount} liczb, algorytm 'bubblesort' wykonał się w
              czasie: { b_duration} natomiast 'quicksort' w czasie: {
              q_duration}")
123     else:
124         print("Nie ma takiej opcji!")
125
126
127 if __name__ == '__main__':
128     main()

```
