Sprawozdanie

Piotr Krawiec, Maksymilian Jucha

5 grudnia 2019

1 Sortwanie szybkie

1.1 Opis algorytmu

Jest to algorytm sortujący. Sortuje on dane poprzez wybranie pewnej liczby z tablicy, a następnie przekładanie liczb w tej tablicy tak aby po lewej stronie znalazły się liczby mniejsze od wybranej a po prawej większe. Następnie wykonuje tą samą operację dla tablicy po lewej i po prawej stronie.

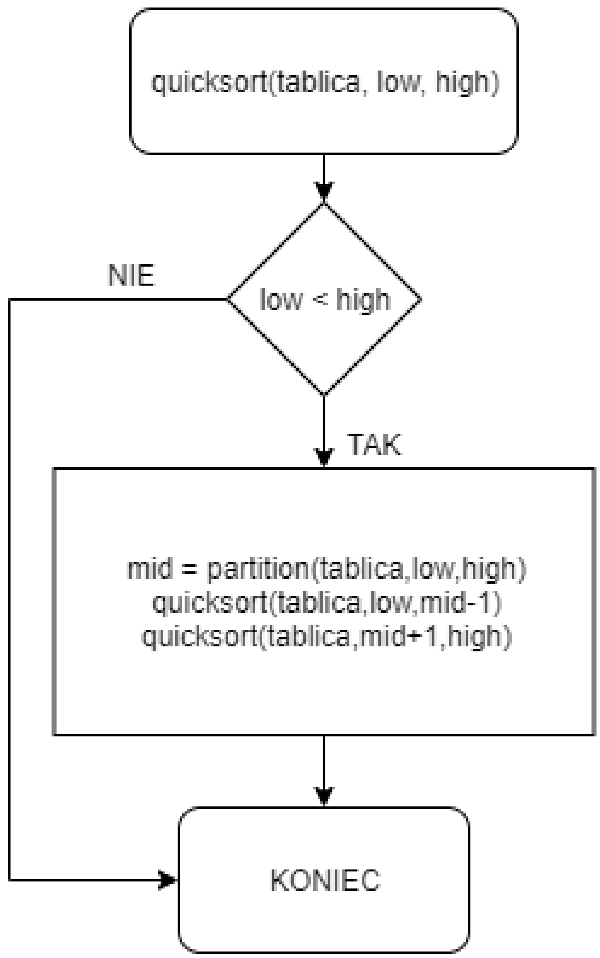
1.2 Kod

[1]:
```
# Zwraca indeks taki że, po lewej stronie są wartości mniejsze od niego, a po␣
↪prawej większe.
def partition(tab, low, high):
    # Wybieram indeks high jako wartość względem której będę porównywał elementy␣
↪i nazwę ją pivot.
    pivot = tab[high]
    # mid to indeks pivot, za którym będą znajdować się wartości większe od␣
↪pivot.,
    # na początek ustawiam go na pierwszą wartość.
    mid = low
    # Dla każdego elementu w zakresie low-high
    for i in range(low, high):
        # Jeżeli wartość w tablicy jest mniejsza od pivot
        if tab[i] < pivot:
            # Zamieniam miejscami w tablicy obecną wartość z mid
            tab[i], tab[mid] = tab[mid], tab[i]
            # zwiększam mid o 1
            mid+=1
    tab[mid], tab[high] = tab[high], tab[mid]
    return mid


def quicksort(tab, low, high):
    if low < high:
        # Poprzekładaj tablicę względem ostatniego elementu
        mid = partition(tab,low,high)
        # Przekładaj lewą stronę tablicy
```

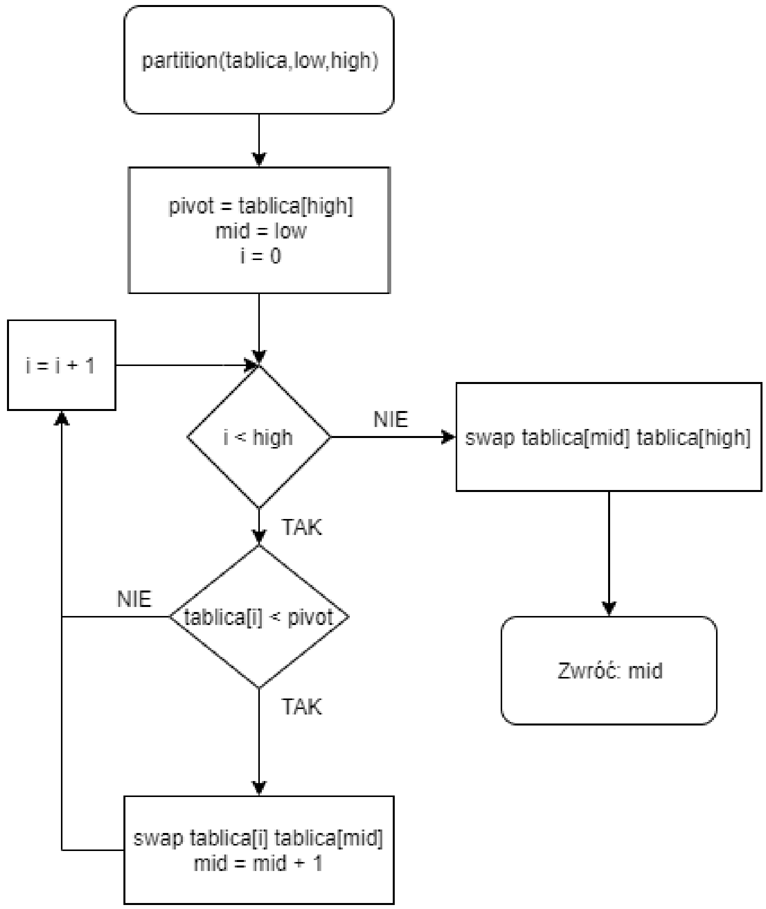
```
        quicksort(tab,low,mid-1)
        # Przekłądaj prawą stronę tablicy
        quicksort(tab,mid+1,high)
```

1.3 Schematy blokowe

1.3.1 Funkcja sortująca



1.3.2 Funkcja przestawiająca dane w tablicy



1.4 Analiza

Złożoność tego algorytmu w dużej mierze zależy od tego względem której liczby będziemy dzielić i przestawiać tablicę. Możemy wyróżnić trzy przypadki:

1.4.1 Przypadek pesymistyczny

Jako, że zawsze wybieramy ostatnią liczbę algorytm ten będzie dokonywał największej ilości porównań dla tablicy posortowanej tj. przejdzie po całej pętli i podzieli ją na dwie tablice, z czego pierwsza będzie zawierała wszystkie elementy oprócz ostatniego. Zatem wykona się ona $(n-1)+(n-2)+\ldots+2+1$ razy co daje w sumie $S=\frac{1+(n-1)}{2}*n=\frac{n^2}{2}$

1.4.2 Pozostałe przypadki

W pozostałych przypadkach zakładamy, że trafiamy na liczę która jest w przybliżeniu medianą liczb sortowanych. Wtedy algorytm dzieli tablicę na dwie części i osiąga złożoność podobną do sortowania przez scalanie $O(nlog\{n\})$.

1.5 Doświadczenia:

1.5.1 Doświadczenie Q1

- Zakres liczb: -20-20,
- Ilość liczb: 10,
- Sposób wybierania: losowy

[2]:
```
import random
tablica_przed_posortowaniem = [random.randrange(-20,21) for i in range(0,10)]

print("Przed posortowaniem: ")
print(tablica_przed_posortowaniem)

posort = list.copy(tablica_przed_posortowaniem)
quicksort(posort,0,len(posort)-1)

print("Sortowanie szybkie: ")
print(posort)
```

```
Przed posortowaniem:
[17, 9, 7, -14, 9, -17, -5, 3, -6, 8]
Sortowanie szybkie:
[-17, -14, -6, -5, 3, 7, 8, 9, 9, 17]
```

1.5.2 Doświadczenie Q2

- Zakres liczb: -1000-1000,
- Ilość liczb: 10 000,
- Sposób wybierania: losowy,

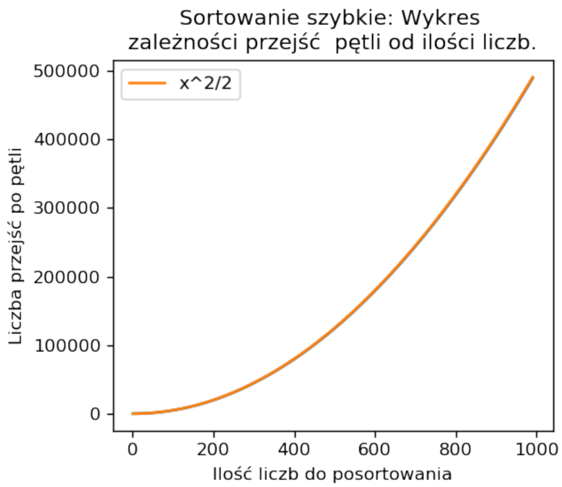
[3]:
```
import random
from timeit import default_timer as timer
tablica_przed_posortowaniem = [random.randrange(-1000,1001) for i in␣
↪range(0,10000)]
posort = list.copy(tablica_przed_posortowaniem)

qstart = timer()
quicksort(posort,0,len(posort)-1)
qend = timer()

print("Sortowanie szybkie: "+str(qend - qstart)+ " sekund")
```

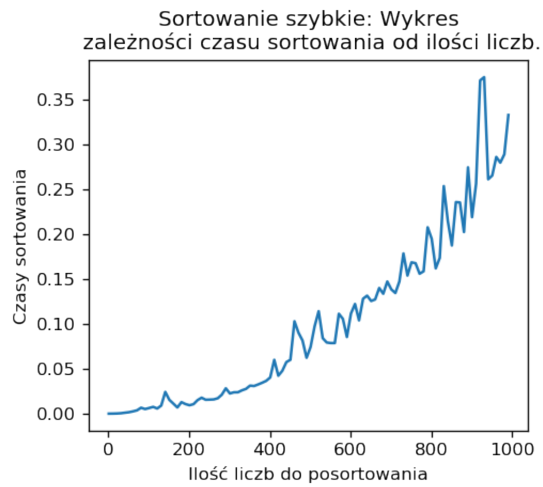
```
Sortowanie szybkie: 0.09166770000000013 sekund
```

1.5.3 Doświadczenie Q3

- Zakres licznb: 1-1000,
- Ilość liczb: od 1 do 1000,

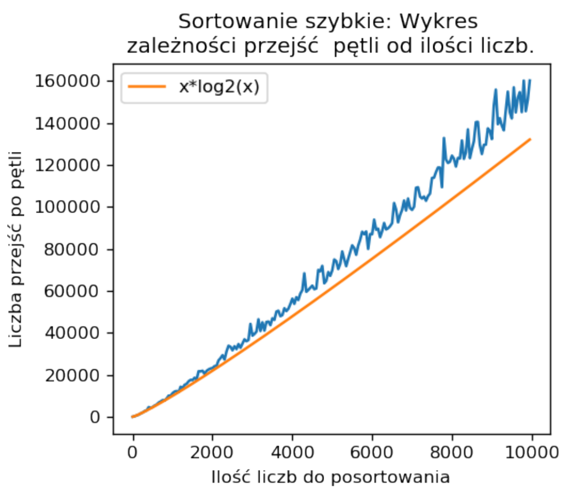
- Sposób wybierania: posortowane,

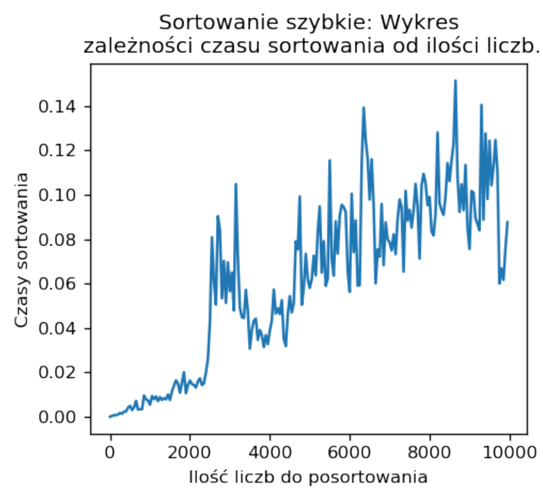




1.5.4 Doświadczenie Q4

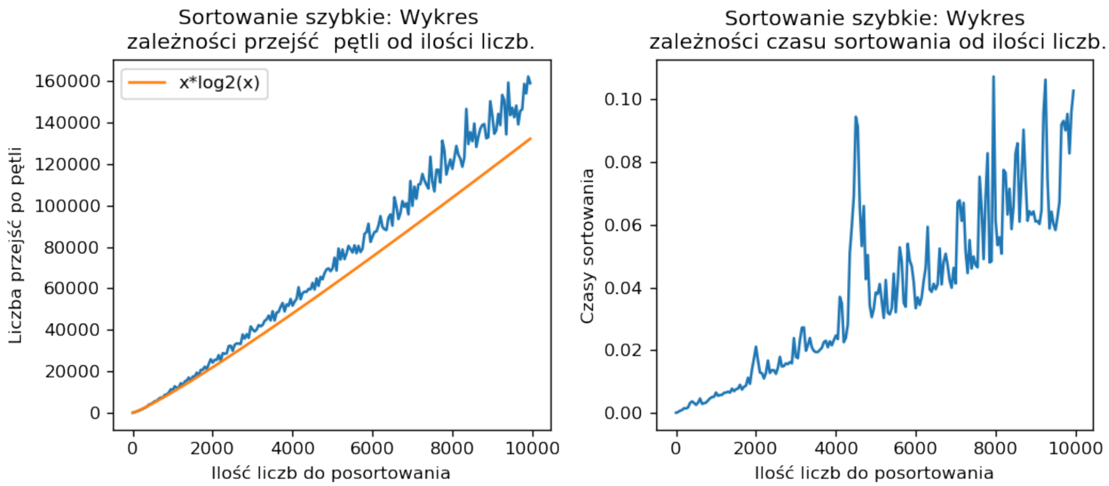
- Zakres licznb: 0-10000,
- Ilość liczb: od 1 do 10000,
- Sposób wybierania: losowe,





1.5.5 Doświadczenie Q5

- Zakres licznb: 0-10000,
- Ilość liczb: od 1 do 10000,
- Sposób wybierania: prawie posortowane,



1.6 Wnioski

- Jak widać po doświadczeniu Q3, algorytm ten w tej wersji nie radzi sobie dobrze z liczbami które są posortowane, ponieważ zawsze wybiera ostatnią liczbę a co za tym idzie, za każdym razem zmiejsza wielkość tablicy do posortowania tylko o 1. (Q3)
- Doświadczenia Q1 i Q2 pokazują że algorytm działa i sortuje liczby, zarówno ujemne jak i dodatnie. (Q1 i Q2)
- Algorytm bardzo dobrze radzi sobie z dużą liczbą losowych liczb, osiągając przy tym złożoność $O(nlogn)$ (Q4)
- Jak widać dobrze sobie radzie z liczbami które są prawie posortowane (Q5)

2 Sortowanie bąbelkowe

2.1 Opis algorytmu

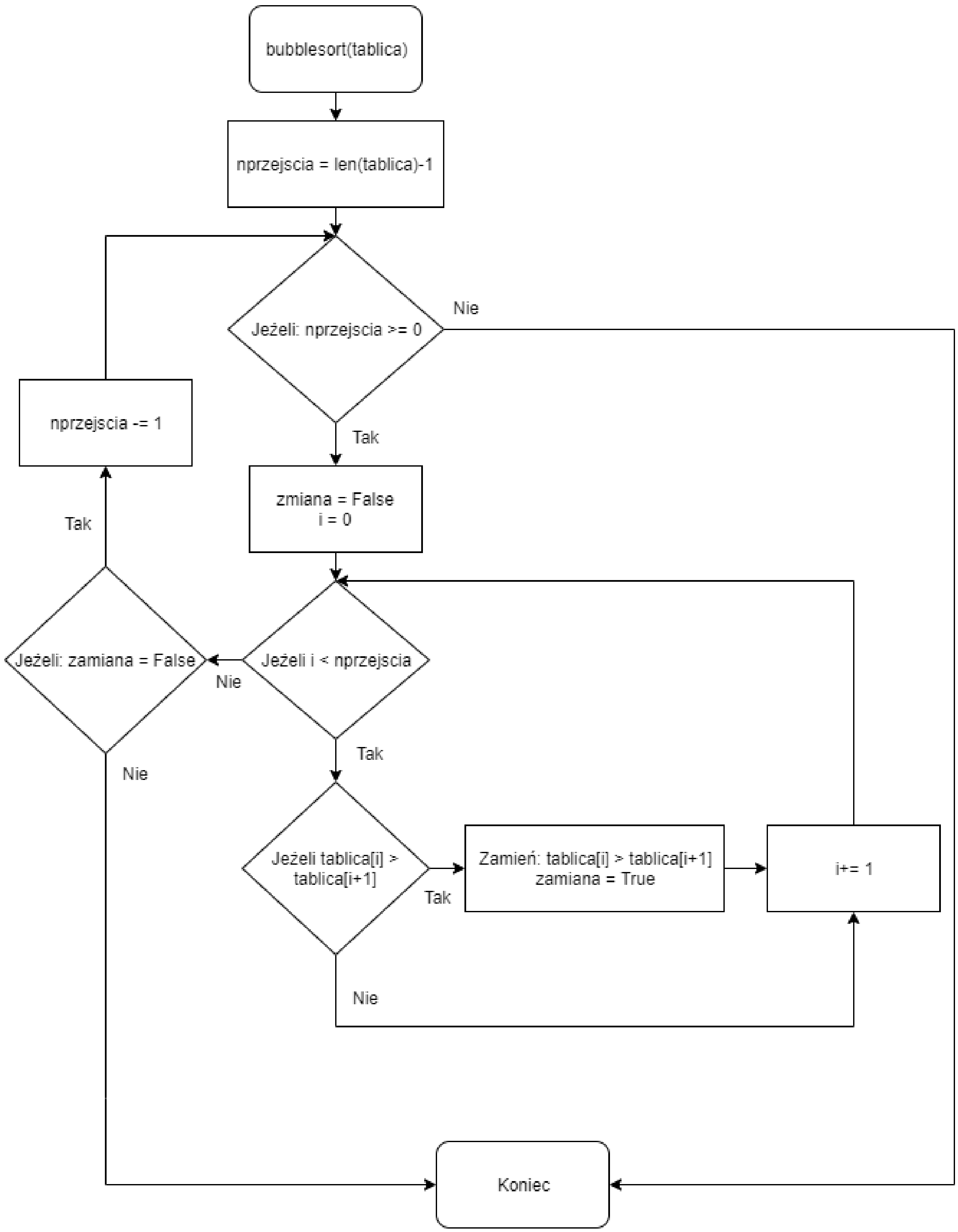
Sortowanie bąbelkowe to algorytm, który przechodzi od początku tablicy do jej końca porównując kolejne elementy i zamieniając je ze sobą jeżeli są w złej kolejności. W ten sposób na koniec tablicy trafia zawsze największa wartość z tej tablicy, dzięku temu kolejne wywołanie algorytmu przechodzi już tylko do przedostatniego elementu. Cykl ten się powtarza aż do posortowaniazostanie tylko jeden element.

2.2 Kod

[7]:
```
def bubbleSort(sorted_list):
    for nprzejscia in range(len(sorted_list)-1,0,-1):
        zamiana = False
        for i in range(nprzejscia):
            if sorted_list[i]>sorted_list[i+1]:
                sorted_list[i], sorted_list[i+1] = sorted_list[i+1],␣
↪sorted_list[i]
```

```
                zamiana = True
        if not zamiana:
            break
```

2.3 Schematy blokowe



2.4 Analiza

2.4.1 Przypadek optymistyczny

W tym przypadku algorytm sortuje posortowaną tablicę. Zatem już po pierwszym przejściu po tablicy z powodu braku zmian zakończy on działanie osiągają złożoność $O(n)$.

2.4.2 Pozostałe przypadki

W każdym pozostałym przypadku algorytm ten będzie osiągał złożoność $O(n^2)$

2.5 Doświadczenia:

2.5.1 Doświadczenie B1

- Zakres liczb: -20-20,
- Ilość liczb: 10,
- Sposób wybierania: losowy

[8]:
```
import random
tablica_przed_posortowaniem = [random.randrange(-20,21) for i in range(0,10)]

print("Przed posortowaniem: ")
print(tablica_przed_posortowaniem)

posort = list.copy(tablica_przed_posortowaniem)
bubbleSort(posort)

print("Sortowanie bąbelkowe: ")
print(posort)
```

```
Przed posortowaniem:
[6, 6, 20, 18, 2, -10, -15, -2, 10, -19]
Sortowanie bąbelkowe:
[-19, -15, -10, -2, 2, 6, 6, 10, 18, 20]
```

2.5.2 Doświadczenie B2

- Zakres liczb: -1000-1000,
- Ilość liczb: 10 000,
- Sposób wybierania: losowy,

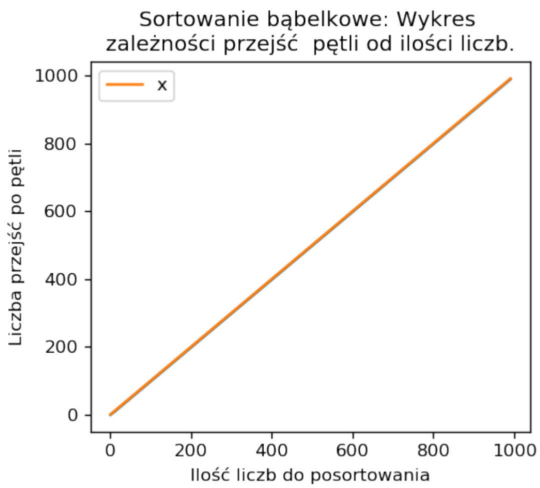
[9]:
```
import random
from timeit import default_timer as timer
tablica_przed_posortowaniem = [random.randrange(-1000,1001) for i in␣
 ↪range(0,10000)]
posort = list.copy(tablica_przed_posortowaniem)

qstart = timer()
bubbleSort(posort)
```

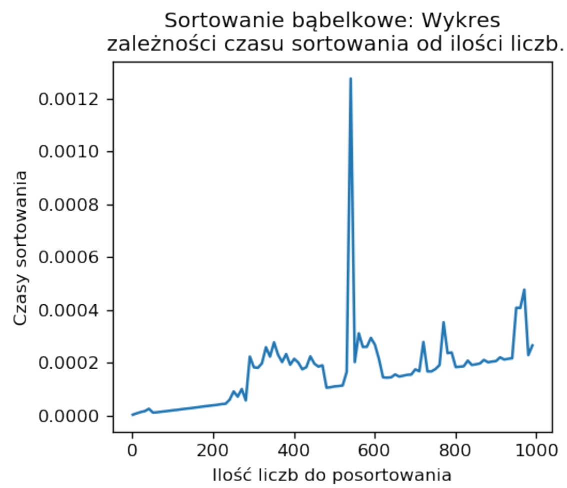
```
qend = timer()

print("Sortowanie bąbelkowe: "+str(qend - qstart)+ " sekund")
```

```
Sortowanie bąbelkowe: 22.224884600000003 sekund
```

2.5.3 Doświadczenie B3

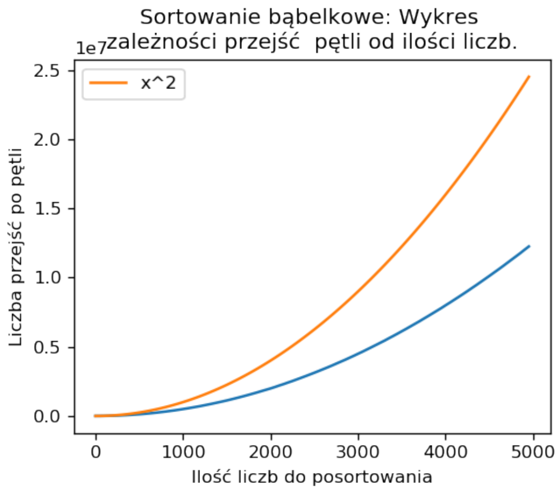
- Zakres licznb: 1-1000,
- Ilość liczb: od 1 do 1000,
- Sposób wybierania: posortowane,

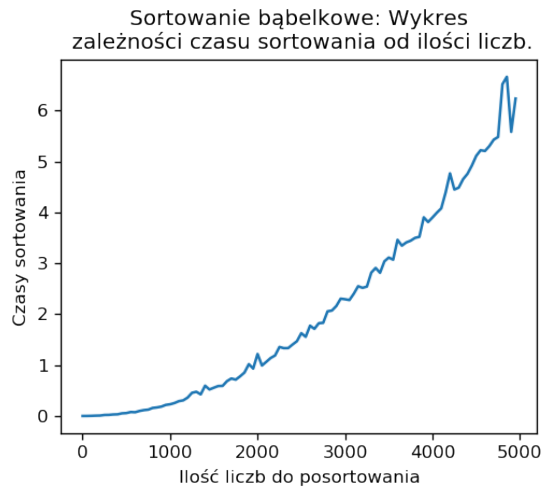




2.5.4 Doświadczenie B4

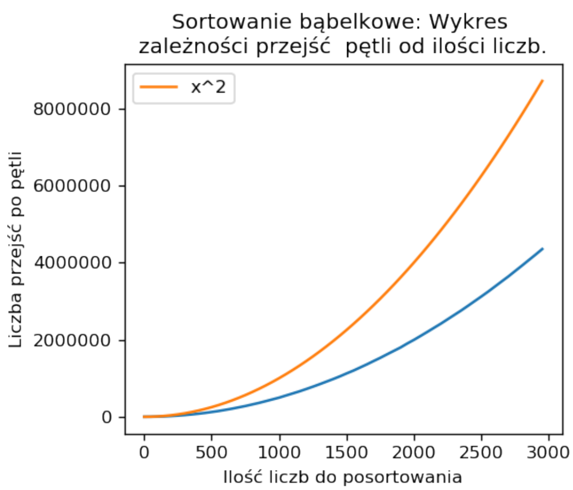
- Zakres licznb: 0-10000,
- Ilość liczb: od 1 do 5000,
- Sposób wybierania: losowe,

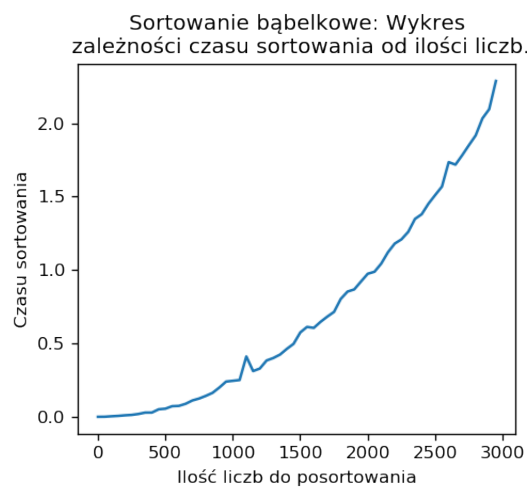




2.5.5 Doświadczenie B5

- Zakres licznb: 0-10000,
- Ilość liczb: od 1 do 5000,
- Sposób wybierania: prawie posortowane,





2.6 Wnioski

- Algorytm dobrze sobie radzi ze sprawdzeniem czy liczby są posortowane (B3),
- Algorytm prawidłowo sortuje tablce, zarówno liczby dodatnie jak i ujemne (B1 B2),
- Jak widać na doświadczeniu B2 czas sortowania rośnie gwałtownie wraz ze wzrostem długości tablicy, wynika to z jego złożoności $O(n^2)$

- Doświadczenia B4 i B5 pokazują, że niezależnie czy liczby są losowe czy prawie posortowane algorytm nadal wykona około $O(n^2)$ iteracji.

3 Podsumowanie

Jak wynika z powyższych doświadczeń, sortowanie szybkie jest znacznie bardziej wydajne. Jedyną jego wadą jest to, że zachowuje się jak bąbelkowe przy posortowanej tablicy, co wynika wyłącznie z tej implementacji, gdyż wybieram zawsze ostatnią liczbę jako tą względem której porównuję. W pozostałych przypadkach wyprzedza bąbelkowe, co szczególnie widać na wykresach Q4 i B4.