SPRAWOZDANIE 1

Wprowadzenie:
Celem niniejszego sprawozdania jest zbadanie złożoności obliczeniowej, oraz omówienie zasad działania poszczególnych algorytmów sortowania. Wszystkie algorytmy były sprawdzane na ciągach wygenerowanych przez generatory liczb losowych odpowiednio:

-liczby naturalne poukładane losowo [RN]
```
def random_generator(zakres_poczatek, zakres_koniec, ilosc_liczb):
   random_numbers = []
   Num = random.randint(zakres_poczatek,zakres_koniec)
   random_numbers.append(Num)
   for i in range(ilosc_liczb-1):
       tmp = random.randint(zakres_poczatek,zakres_koniec)
       random_numbers.append(tmp)
   return random_numbers
```

-liczby naturalne poukładane w porządku rosnącym [IN]
```
def random_increase_generator(ilosc_liczb):
   random_numbers = []
   Num = random.randint(0,10000)
   random_numbers.append(Num)
   for i in range(ilosc_liczb-1):
       tmp = random_numbers[i] + random.randint(0,100)
       random_numbers.append(tmp)
   return random_numbers
```

-liczby naturalne poukładane w porządku malejącym [DN]
```
def random_decrease_generator(ilosc_liczb):
   random_numbers = []
   Num = random.randint(0, 100000)
   random_numbers.append(Num)
   for i in range(ilosc_liczb - 1):
       tmp = random_numbers[i] - random.randint(0, 100)
       random_numbers.append(tmp)
   return random_numbers
```

-liczby naturalne V kształtne [VN]
```
def random_Vshape_generator(ilosc_liczb):
   First_Half = ilosc_liczb//2
   Sec_Half = ilosc_liczb-First_Half
   random_numbers = []
   Num = random.randint(0,10000)
   random_numbers.append(Num)
   for i in range(First_Half-1):
```

```
        tmp = random_numbers[-1] - random.randint(1,100)
        random_numbers.append(tmp)
    for j in range(Sec_Half):
        tmp = random_numbers[-1] + random.randint(1,100)
        random_numbers.append(tmp)
    return random_numbers
```

-liczby naturalne A kształtne [AN]

```
def random_Ashape_generator(ilosc_liczb):
    First_Half = ilosc_liczb//2
    Sec_Half = ilosc_liczb-First_Half
    random_numbers = []
    Num = random.randint(0,10000)
    random_numbers.append(Num)
    for i in range(First_Half-1):
        tmp = random_numbers[-1] + random.randint(1,100)
        random_numbers.append(tmp)
    for j in range(Sec_Half):
        tmp = random_numbers[-1] - random.randint(1,100)
        random_numbers.append(tmp)
    return random_numbers
```

ALGORYTMY:
-Bubble sort [BS]
-Insertion sort [IS]
-Selection sort [SS]
-Quick sort [QS]
-Merge sort [MS]
-Heap sort [HS]

Pierwsza sekcja obejmuje prezentację kodów oraz wykresów, ukazujących zależność czasu od danych wejściowych (są one jedynie przybliżeniem). Dodatkowo zawarto wypisaną złożoność obliczeniową dla przypadków optymistycznego, średniego i pesymistycznego.

W sekcji drugiej znajdują się wykresy pokazujące zestawienie każdego algorytmu sortowania w zależności od danych wejściowych.

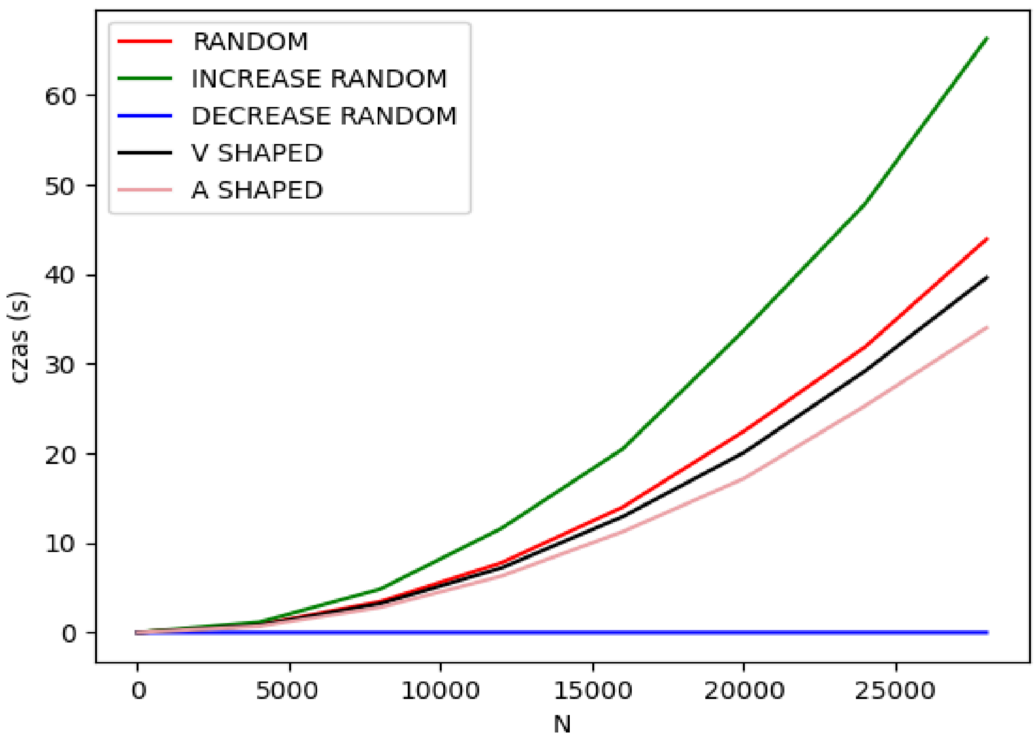
Ostatnia sekcja została poświęcona porównaniom i zamianą elementów w zależności od nadanych przez użytkownika danych wejściowych.

SEKCJA 1

I. BUBBLE SORT

przypadek pesymistyczny: $O(n^2)$
przypadek średni: $O(n^2)$
przypadek optymistyczny: $O(n)$

```
def bubble_sort(liczby=[]):
   num_comparisons = 0

   for i in range(len(liczby) - 1):
       is_swap = False
       for j in range(len(liczby) - 1 - i):
           num_comparisons += 1
           if liczby[j] < liczby[j + 1]:
               num_comparisons += 1
               tmp = liczby[j + 1]
               liczby[j + 1] = liczby[j]
               liczby[j] = tmp
               is_swap = True

       if not is_swap:
           break

   return liczby, num_comparisons
```

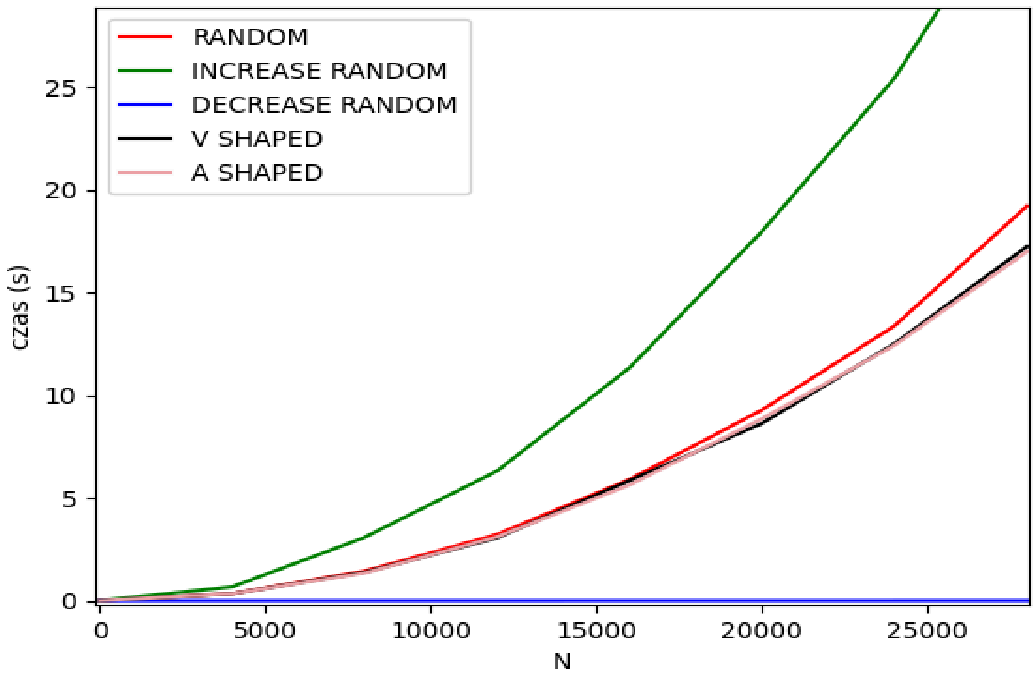


W przypadku nadania ciągu malejącego czas jest równy zero z względu na linię kodu: is_swap = False które sprawdza czy zaszły zmiany w ciągu liczbowym. Jeżeli ta wartość nie zostanie zmieniona na “True” kod jest przerywany i zwracana jest posortowana lista.

II. Insertion sort:

przypadek pesymistyczny: O(n^2)
przypadek średni: O(n^2)
przypadek optymistyczny: O(n)

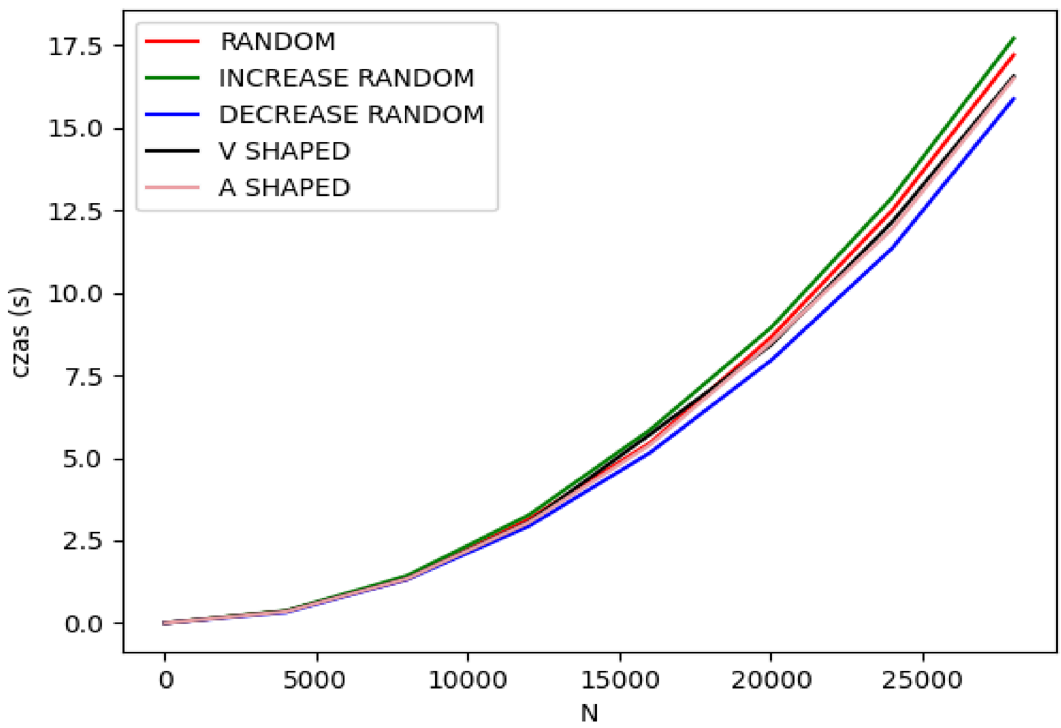
```
def insertion_sort(liczby=[]):
    num_comparisons = 0

    for i in range(1, len(liczby)):
        key = liczby[i]
        j = i - 1

        while j >= 0 and liczby[j] < key:
            liczby[j + 1] = liczby[j]
            j -= 1
            num_comparisons += 1

        liczby[j + 1] = key
        num_comparisons += 1

    return liczby, num_comparisons
```



III. Selection sort

przypadek pesymistyczny: O(n^2)
przypadek średni: O(n^2)
przypadek optymistyczny: O(n^2)

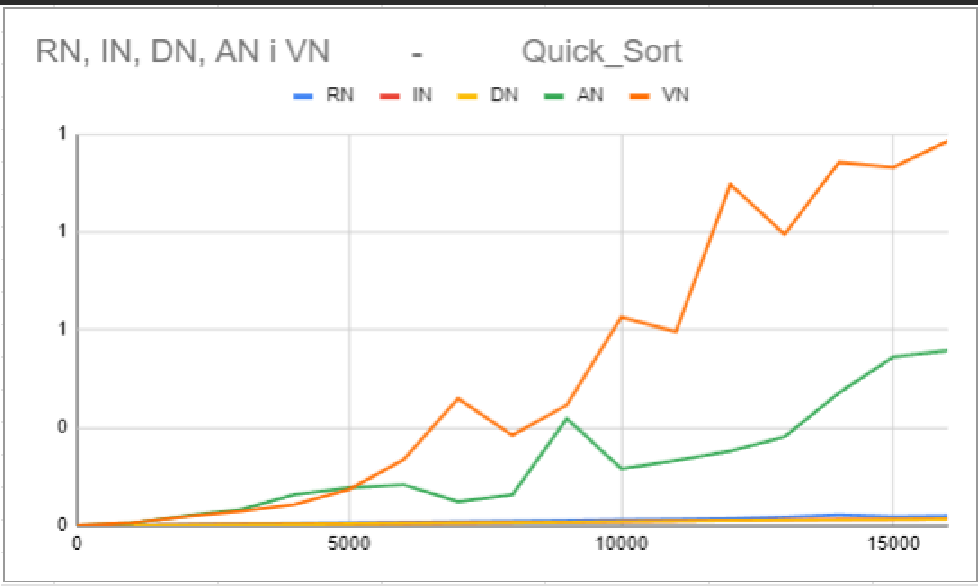
```
def selection_sort(liczby=[]):
   num_comparisons = 0


   for i in range(len(liczby) - 1):
       minindex = i

       for j in range(i + 1, len(liczby)):
           if liczby[j] > liczby[minindex]:
               minindex = j
               num_comparisons += 1

       liczby[i], liczby[minindex] = liczby[minindex], liczby[i]
       num_comparisons += 1

   return liczby, num_comparisons
```



IV. Quick sort

przypadek pesymistyczny: $O(n^2)$
przypadek średni: $O(n \log_2 n)$
przypadek optymistyczny: $O(n \log_2 n)$

```
def quick_sort(array, x):
   global comparisons
   less = []
   equal = []
   greater = []

   if len(array) > 1:
       if x == 'right':
           pivot = array[len(array)-1]
       else:
           pivot = array[len(array) // 2]
       for x in array:
           comparisons += 1
           if x < pivot:
               greater.append(x)
           elif x == pivot:
               equal.append(x)
           elif x > pivot:
               less.append(x)
       return quick_sort(less, x) + equal + quick_sort(greater, x)
   else:
       return array
```



pivot jest ustawiony na ostatnim elemencie tablicy, następnie w pętli for następują porównania elementów do pivota, następnie te elementy trafiają do odpowiedniej tablicy (less, equal, greater). Po wszystkich porównaniach następuje scalenie wszystkich tablic i ich zwrócenie co automatycznie ustawia pivota na odpowiednim miejscu.

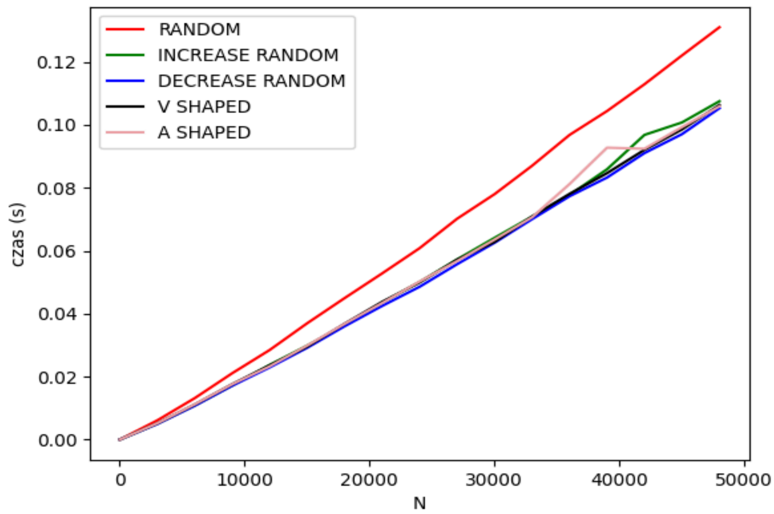
```
       return quick_sort(less, x) + equal + quick_sort(greater, x)
```

Algorytm jest szybszy od innych gdyż zaczyna sortować już podczas dzielenia tablicy na mniejsze, a nie dopiero po podziale (tak jak np. w merge sort).

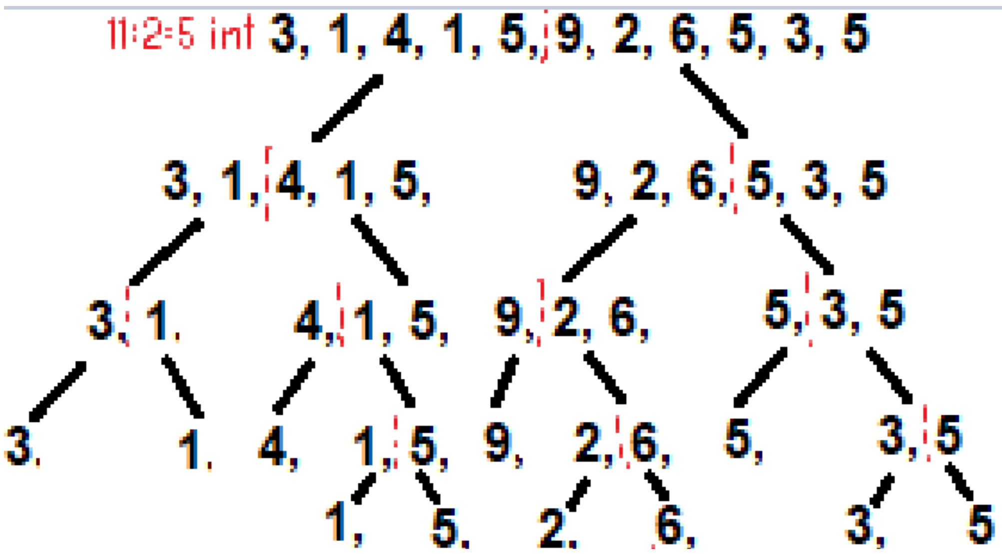
V. Merge sort

przypadek pesymistyczny: O(n $\log_2$n)
przypadek średni: O(n $\log_2$n)
przypadek optymistyczny: O(n $\log_2$n)

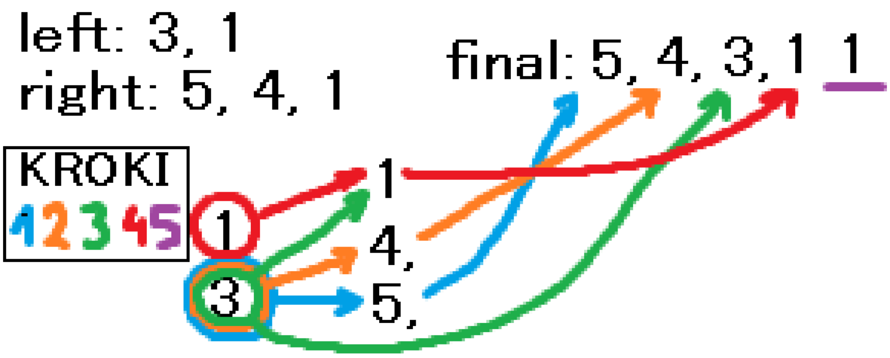
```
def merge_sort(list_to_sort):
   global comparisons
   if len(list_to_sort) > 1:
       division_point = len(list_to_sort) // 2
       left_side = list_to_sort[:division_point]
       right_side = list_to_sort[division_point:]

       merge_sort(left_side)
       merge_sort(right_side)

       i = j = k = 0

       while i < len(left_side) and j < len(right_side):
           comparisons += 1

           if left_side[i] >= right_side[j]:
               list_to_sort[k] = left_side[i]
               i += 1
           else:
               list_to_sort[k] = right_side[j]
               j += 1
           k += 1

       while i < len(left_side):
           comparisons +=1
           list_to_sort[k] = left_side[i]
           i += 1
           k += 1
       while j < len(right_side):
           comparisons += 1
           list_to_sort[k] = right_side[j]
           j += 1
           k += 1
   return list_to_sort
```




```
#rozkład podziału dla liczb random_arr = [3, 1, 4, 1, 5, 9, 2, 6, 5, 3, 5]
```



Pierwsza częśc algorytmu dzieli rekurencyjnie daną tablice na dwie mniejsze (lewa, prawa), aż do momentu gdzie zostanią tylko po jednym elemencia na końcu, następnie te liczby są zwracane i porównywane do siebie w pierwszej pętli while. Na poniższym schemacie zostało pokazane jak przebiega porównanie i scalanie strony lewej i prawej.

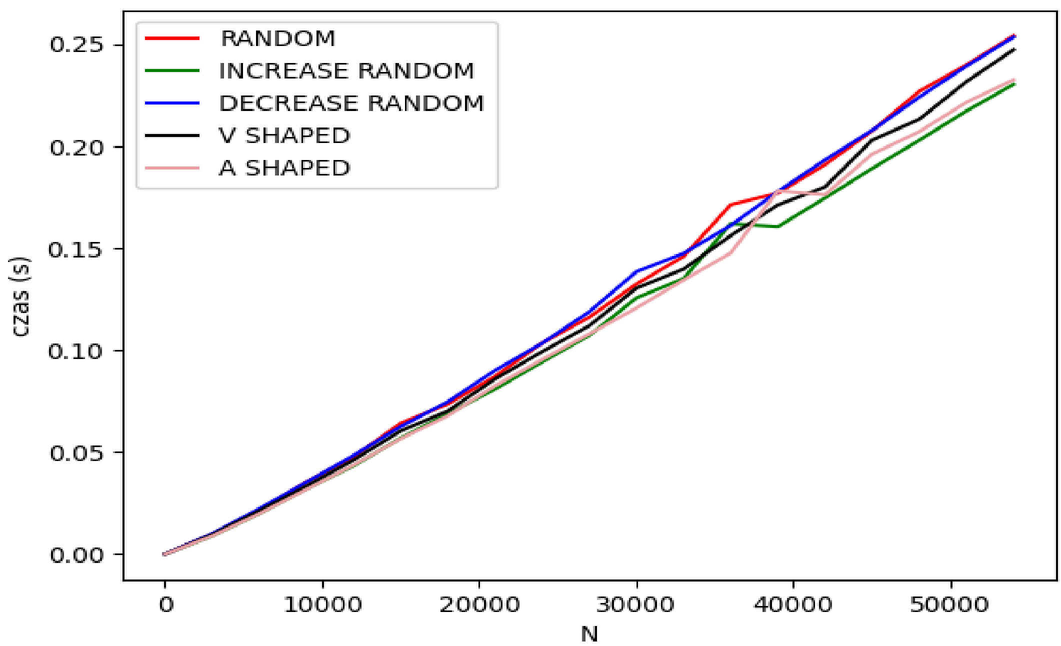


Ostatnia pętla odpowiada za dodanie do tablicy pozostałych elementów z lewej i prawej połówki, które zostały po procesie porównywania (zaznaczone kolorem fioletowym).

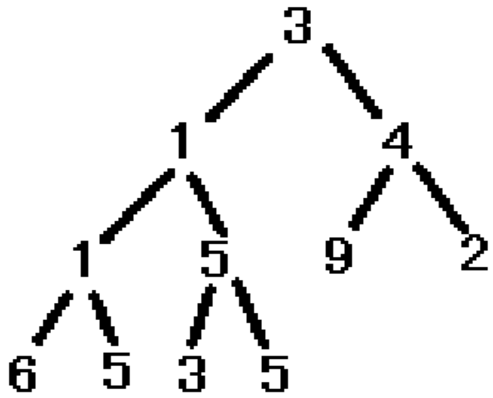
VI. Heap sort

przypadek pesymistyczny: O(n $\log_2 n$)
przypadek średni: O(n $\log_2 n$)
przypadek optymistyczny: O(n $\log_2 n$)

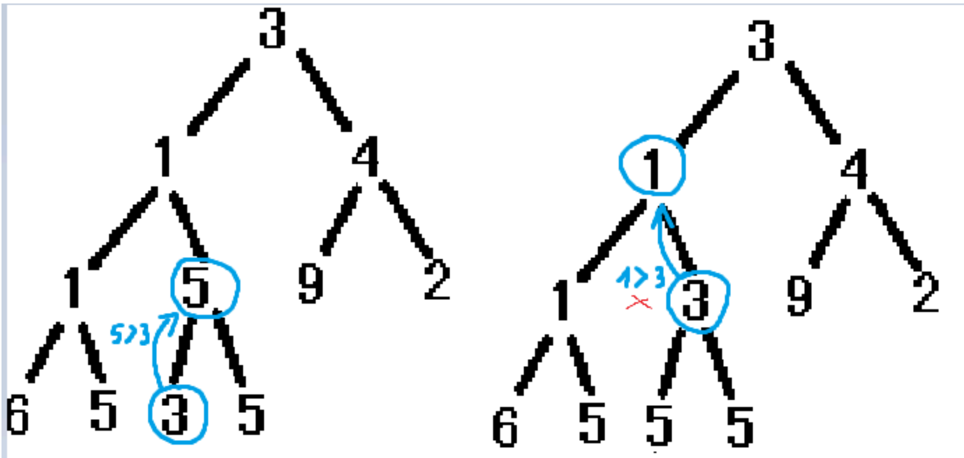
```
def heapify(arr, n, i):
    global comparisons
    largest = i
    l = 2 * i + 1
    r = 2 * i + 2
    if l < n and arr[l] < arr[largest]:
        comparisons += 1
        largest = l
    if r < n and arr[r] < arr[largest]:
        comparisons += 1
        largest = r
    if largest != i:
        arr[i], arr[largest] = arr[largest], arr[i]
        comparisons += 1
        heapify(arr, n, largest)
def heap_sort(arr):
    global comparisons
    n = len(arr)
    for i in range(n // 2 - 1, -1, -1):
        heapify(arr, n, i)
    for i in range(n - 1, 0, -1):
        arr[i], arr[0] = arr[0], arr[i]
        comparisons += 1
        heapify(arr, i, 0)
    return arr
```



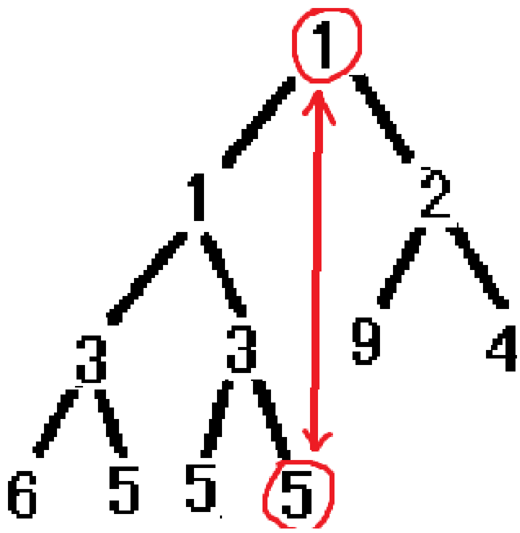

```
#kopiec dla liczb random_arr = [3, 1, 4, 1, 5, 9, 2, 6, 5, 3, 5]
```



Po wywołaniu funkcji heap_sort następuje pętla iterująca po rodzicach drzewa. Powoduje to rekurencyjne wywołanie funkcji heapify, która sprawdza i zamienia dzieci z rodzicem w przypadku, gdy rodzic jest większy od dziecka.



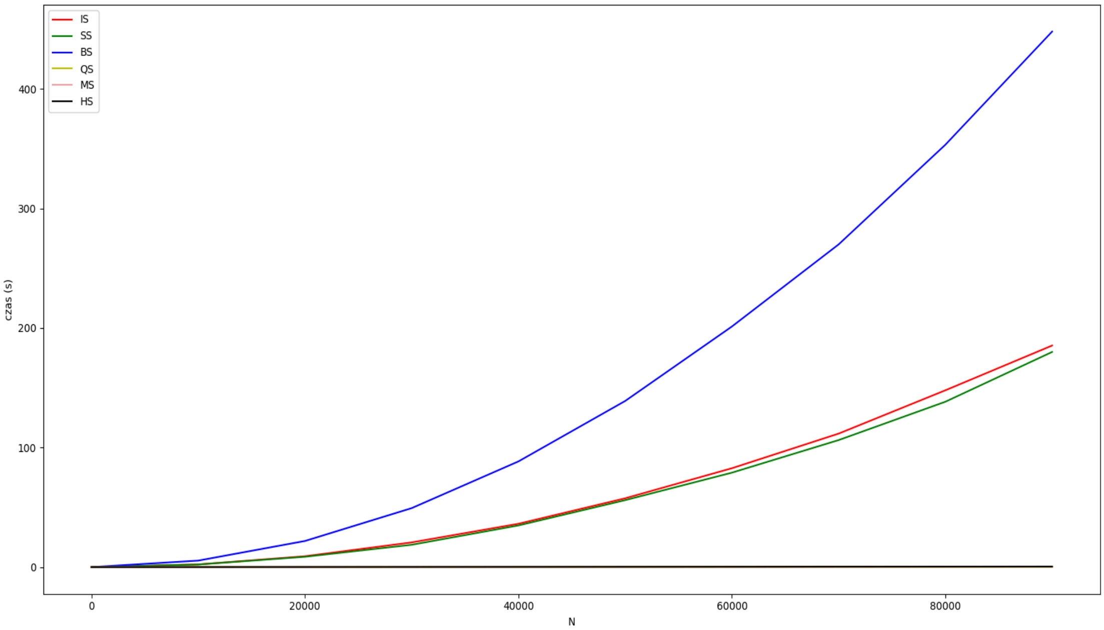
Na samym końcu następuje kolejne rekurencyjne wywołanie funkcji tak, aby najmniejszy element znalazł się na samym szczycie drzewa. Wtedy dochodzi do zakończenia wywoływanej funkcji heapify. W quick_sort kolejna pętla for zamienia ostatni element z pierwszym oraz wywołuje rekurencyjnie funkcję heapify.



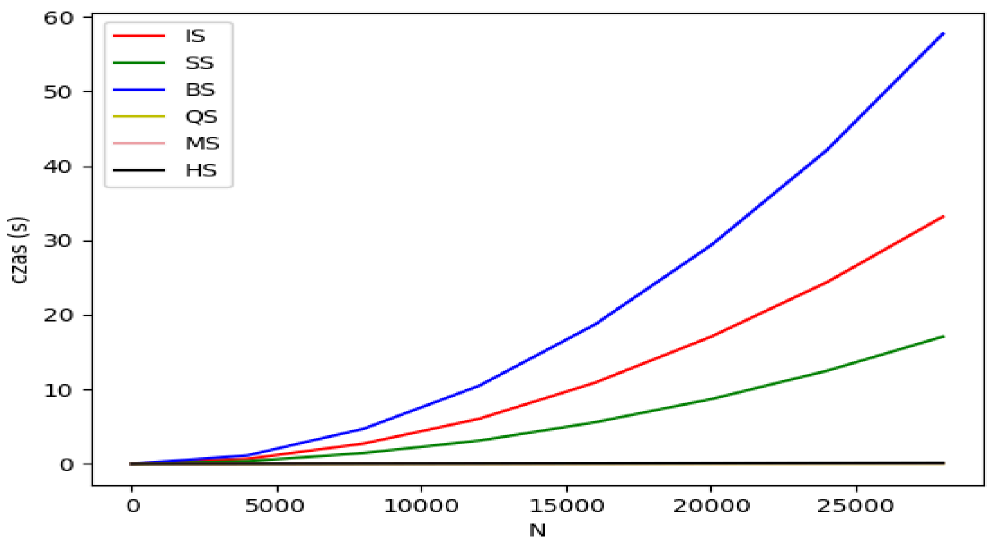
SEKCJA 2

Na wykresach widnieją skróty oznaczające odpowiednio:
- -Bubble sort [BS]
- -Insertion sort [IS]
- -Selection sort [SS]
- -Quick sort [QS]
- -Merge sort [MS]
- -Heap sort [HS]

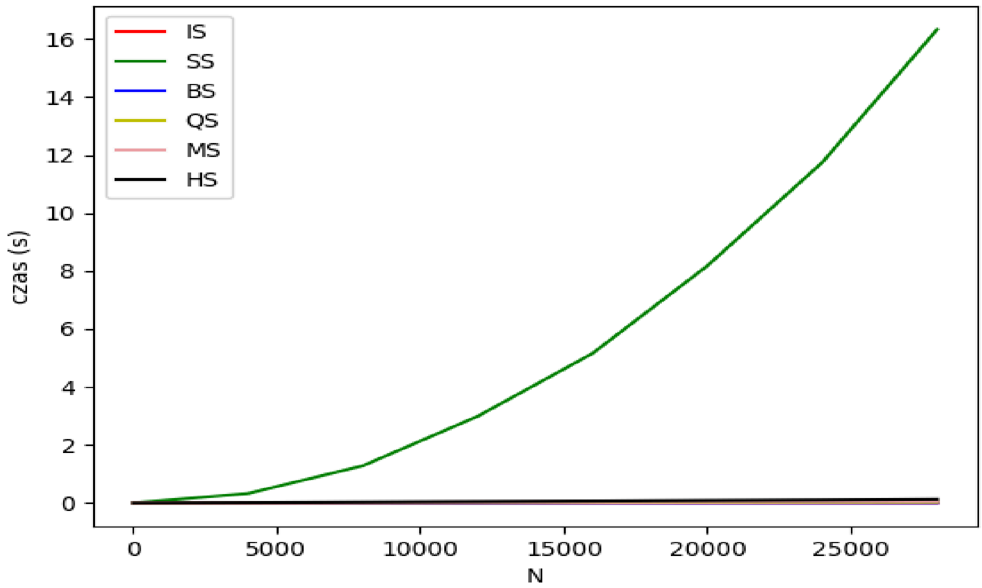
I. Zależność czasu od danych wejściowych ułożonych w losowym porządku:



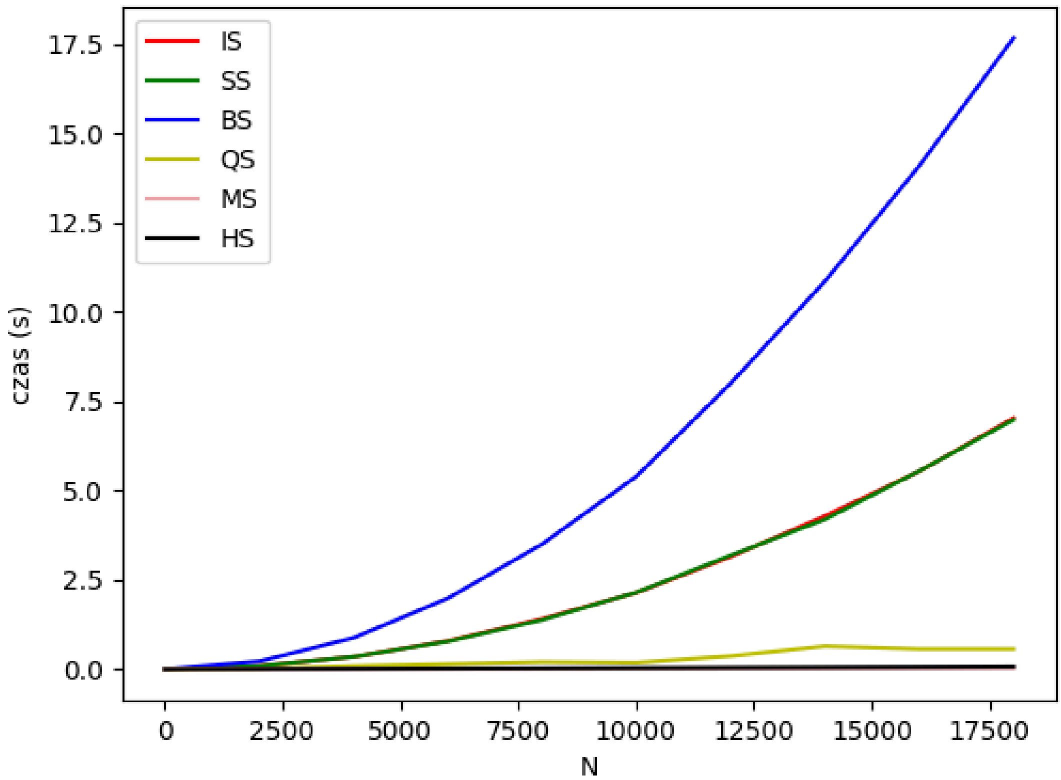
II. Zależność czasu od danych wejściowych ułożonych rosnąco:



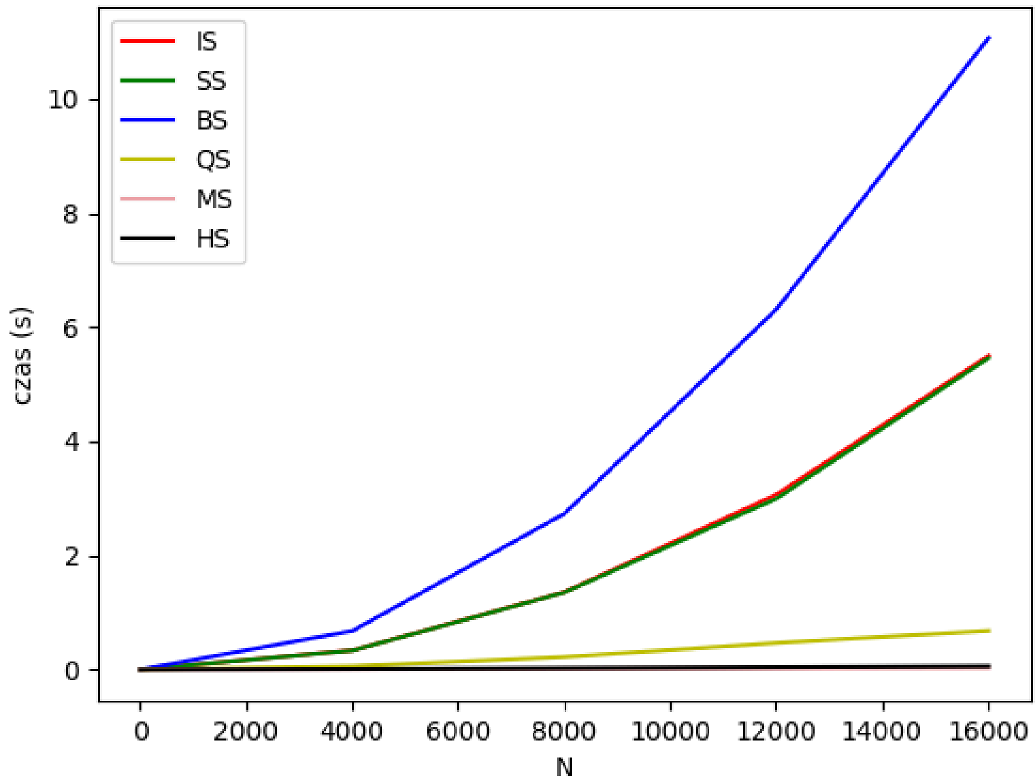
III. Zależność czasu od danych wejściowych ułożonych w porządku malejącym:



IV. Zależność czasu od danych wejściowych V kształtnych:



V. Zależność czasu od danych wejściowych A kształtnych:



Na wykresach VI oraz V możemy zaobserwować, że quick sort ma większa złożoność obliczeniową od innych algorytmów bazujących na rekurencji dla danych A i V kształtnych.

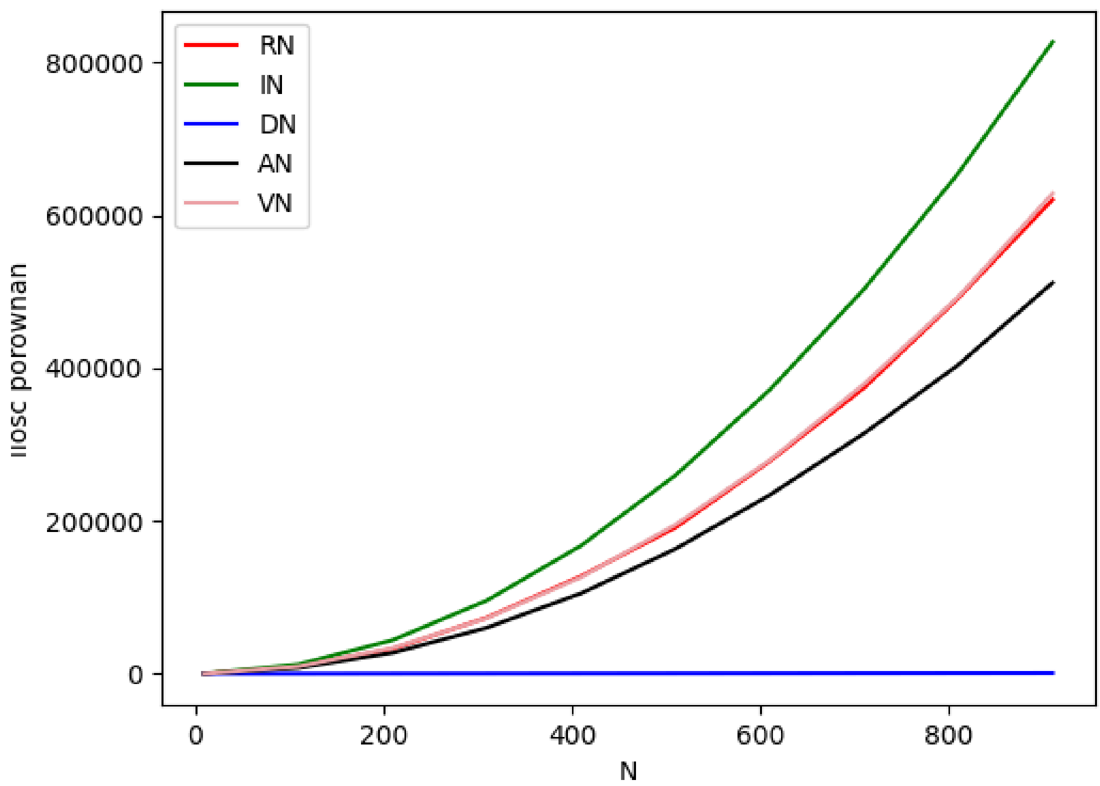
Quick sort mimo, że jest uznawany za najszybszy z algorytmów sortowania (spośród przedstawionych w tym sprawozdaniu) nie jest najlepszy dla każdych danych wejściowych.

W ogólnym zestawieniu wszystkich algorytmów widać, że najgorzej wypada Bubble sort. A złożoność obliczeniowa Selection sort i Insertion sort jest do siebie bardzo zbliżona niezależnie od danych podanych na wejście.

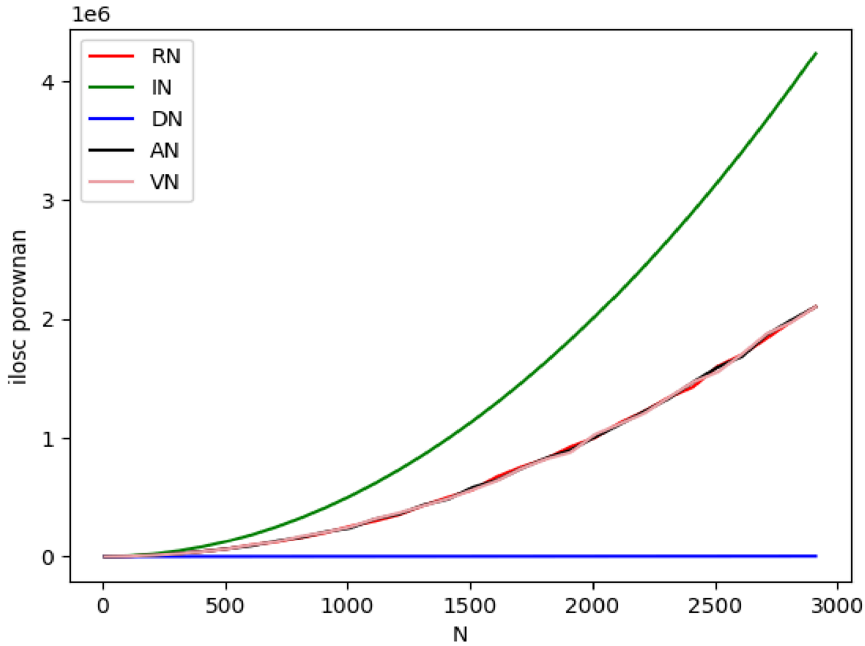
SEKCJA 3

Na wykresach widnieją skróty oznaczające odpowiednio:
-liczby losowe [RN]
-liczby poukładane rosnąco [IN]
-liczby poukładane malejąco [DS]
-liczby A kształtne [AN]
-liczby V kształtne [VN]

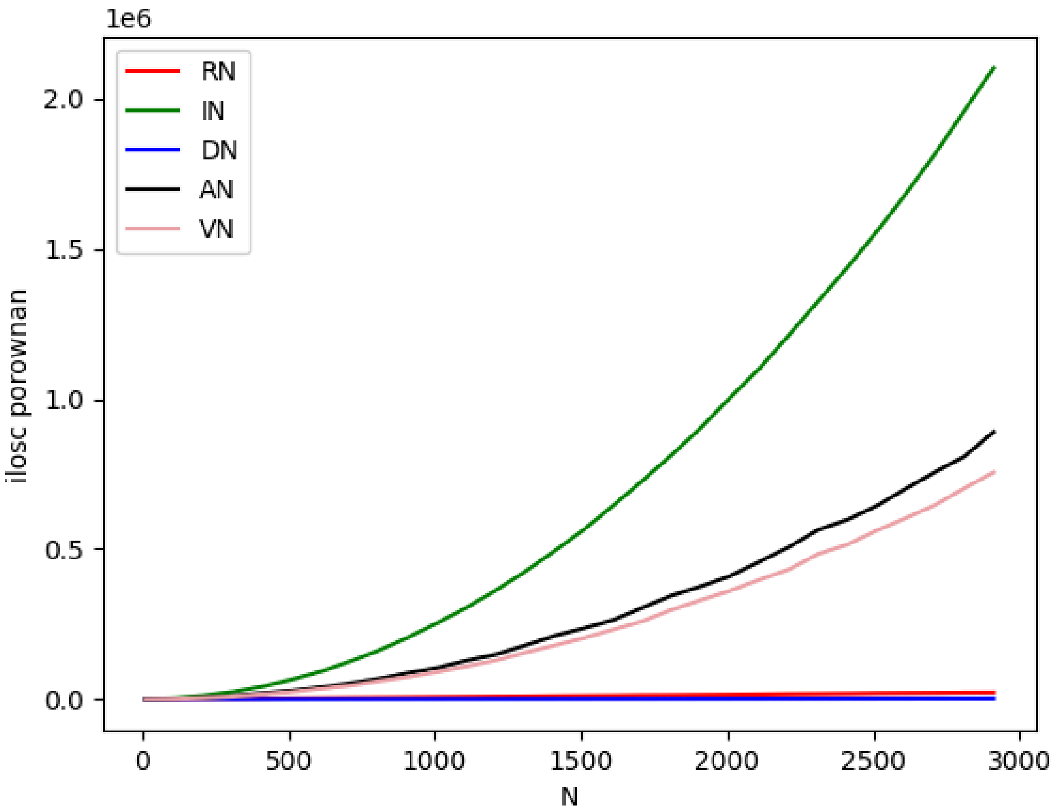
I. Zależność ilości porównań do ilości elementów danych wejściowych dla algorytmu Bubble sort:



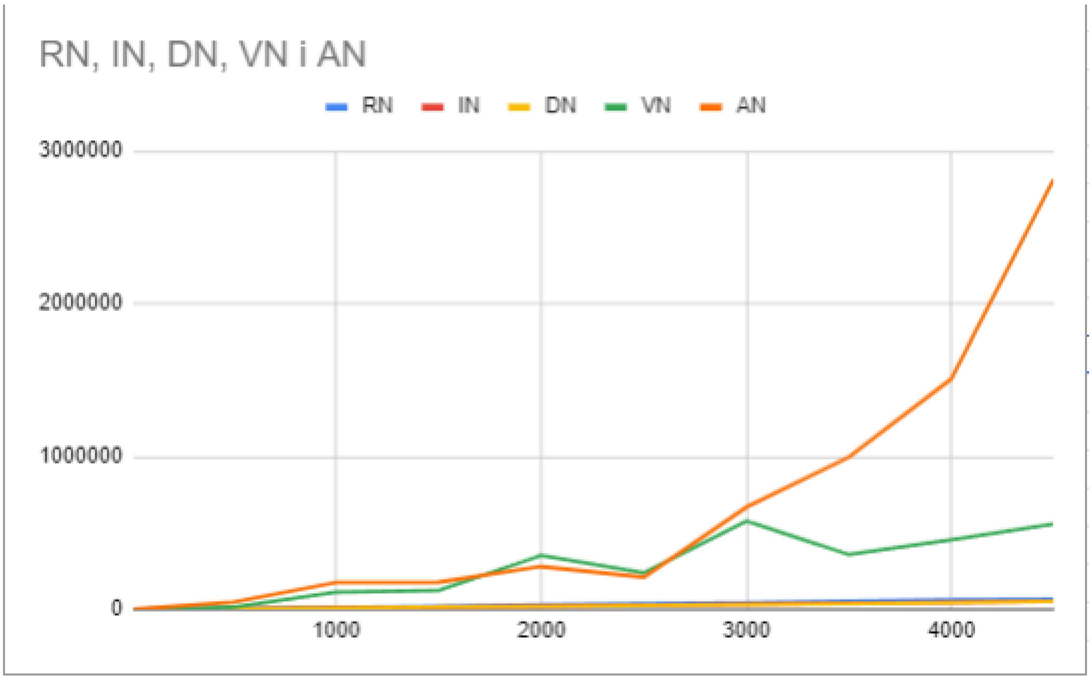
II. **Zależność ilości porównań do ilości elementów danych wejściowych dla algorytmu Insertion sort:**



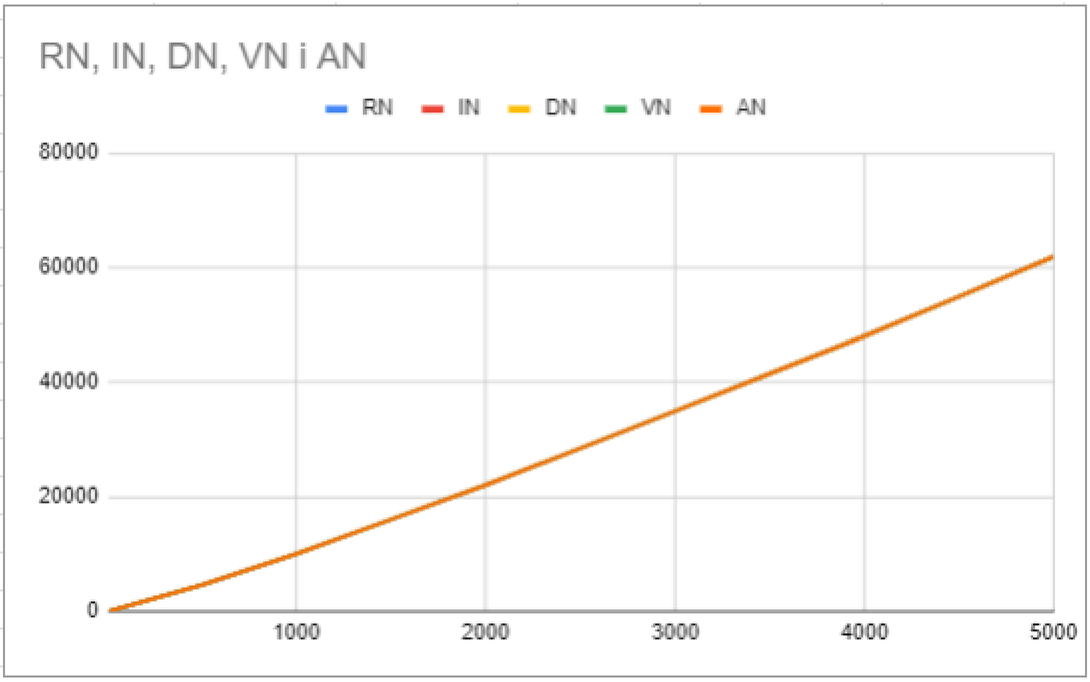
III. **Zależność ilości porównań do ilości elementów danych wejściowych dla algorytmu Selection sort:**



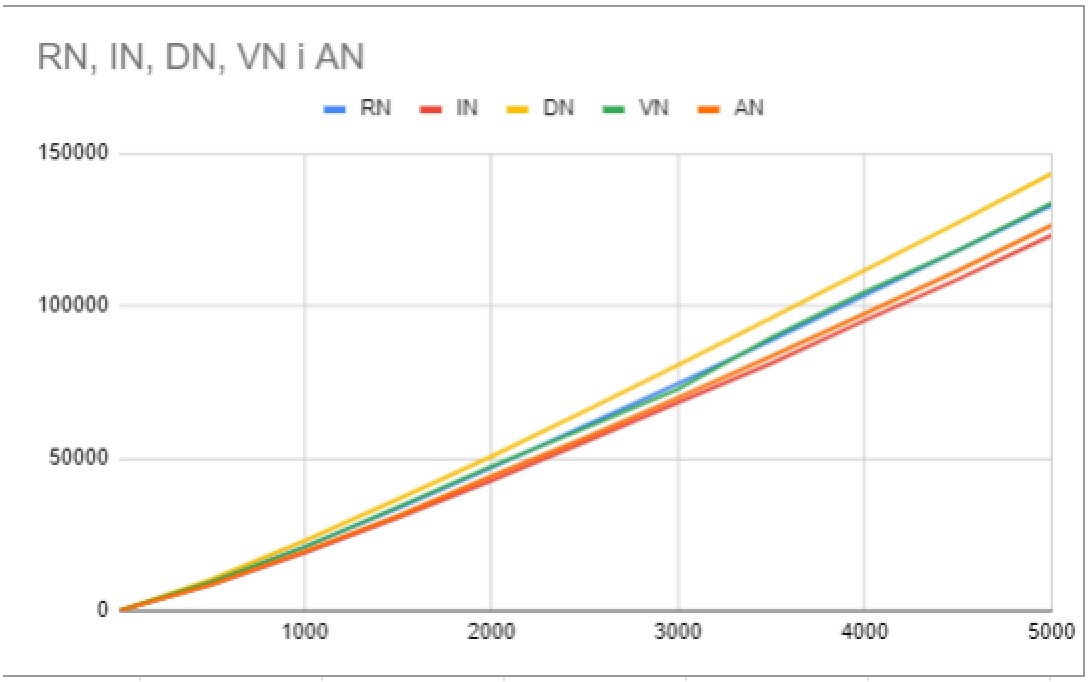
IV. Zależność ilości porównań do ilości elementów danych wejściowych dla algorytmu Quick sort:



V. Zależność ilości porównań do ilości elementów danych wejściowych dla algorytmu Merge sort:



IV. Zależność ilości porównań do ilości elementów danych wejściowych dla algorytmu Heap sort:



MATERIAŁY UZUPEŁNIAJĄCE:

arkusz excel z danymi:
https://docs.google.com/spreadsheets/d/1LtT5JMPimuwgjOlWz7C3-s33baWzhHN4ClAVNHOi3ik/edit?usp=sharing