

# Data Structure and Algorithms

## Major Task Report

Ahmed Amr Elgayar 19p8349  
Mohamed Saber Mohamed 19p9544  
Anas Salah Saadeldin 19p9033  
A"laa Mohamed Hamdy 19p6621  
Nada Amr Attya 19p1621

---

1/1/2021

CSE331

Dr. Hesham Farag

# 1. Contents

2.	Introduction.....	5
3.	Insertion Sort.....	6
	Code .....	6
	Correctness .....	6
	Running Time .....	7
4.	Bubble Sort .....	8
	Code .....	8
	Correctness .....	8
	Running time .....	9
5.	Selection sort .....	11
	Code .....	11
	Correctness .....	11
	Running time .....	12
6.	Merge sort.....	14
	Code .....	14
	Correctness .....	15
	Running time .....	16
7.	Counting Sort.....	18
	Code .....	18
	Correctness .....	18
	Running Time .....	18
8.	Heap Sort .....	20
	Heap Data Structure.....	20
	MAX-HEAPIFY.....	20
	Build Max Heap .....	21
	Heap Sort Code.....	21
	Correctness .....	22
	Running Time .....	22
9.	Radix Sort .....	24
	Code .....	24
	Correctness .....	24
	Running Time .....	25
10.	Quick Sort.....	27
	Code .....	27
	Correctness .....	27

Running time .....	28
11. GUI .....	31
Random generator .....	31
Main sort functions.....	31
GUI windows .....	34
Output Screenshots .....	38

## Table of figures

1 insertion sort with steps counter .....	7
2 insertion sort graph .....	7
3 Bubble sort code .....	8
4 bubble sort with step counter.....	9
5 bubble sort graph.....	10
6 Selection sort code.....	11
7 selection sort with step counter.....	12
8 Selection sort graph.....	13
9 merge sort function .....	14
10 merge function code .....	15
11 merge function with step counter .....	16
12 merge sort function with step counter .....	17
13 merge sort graph .....	17
14 Counting Sort Code.....	18
15 Counting Sort with Step function .....	19
16 Counting Sort Graph.....	19
17 Max Heapify Code .....	20
18 buildmaxheap code.....	21
19 heap sort code with step count 2 .....	22
20 heap sort code with step count 1 .....	22
21 Heap Sort Graph .....	23
22 Radix Sort Code .....	24
23 Counting Sort in Radix.....	24
24 Radix Code with Count steps .....	25
25 Radix Sort Graph .....	26
26 Quick Sort Code.....	27
Figure 27 Quick Sort Code .....	27
28 Quick Sort with Count Steps .....	29
Figure 29 Quick Sort with Count Steps.....	29
30 quick sort graph.....	29
31 quick sort graph .....	29
32 random generator code .....	31
33 random generator code .....	31
34 main_sort i .....	31
35 main_sort i .....	31
36 main_sort ii .....	32

37 main_sort ii .....	32
38 main_sort iii.....	32
39 main_sort iii .....	32
40 main_sort iv .....	33
41 main_sort iv.....	33
42 main_sort v .....	34
43 main_sort v .....	34
44 gui imports and window properties .....	35
45 gui imports and window properties .....	35
46 algorithms functions .....	36
47 algorithms functions.....	36
48 buttons.....	37
49 buttons.....	37
50 check boxes.....	37
51 check boxes .....	37
52 comparison graph.....	38
53 comparison graph.....	38
54 step counter array.....	38
55 main window .....	39
56 comparison window .....	39
57 n*n algorithms comparison.....	39
58 nlogn algorithms comparison .....	39
59 all algorithms comparison.....	39

## 2. Introduction

Sorting algorithms are a fundamental part of computer sciences. It simply sorts a list into a numerical order. However, efficiency is key in the sorting algorithms as heavy programs deals with arrays of thousands and millions of numbers and is required to sort these enormous amounts of elements quickly so that the program's response time is reasonable.

Each sorting algorithm has its average, worst- and best-case running times. The asymptotic notation  $\Omega(n)$  denotes the asymptotic lower bound (the best-case running time) and is usually if the array that requires sorting is already sorted. The asymptotic notation  $\Theta(n)$  denotes the growth average case. The asymptotic notation  $O(n)$  denotes the asymptotic upper bound (the worst-case running time) and is the primary notation use for general algorithm time complexity. A step counter can be used instead of the time taken for the algorithm to sort, as each step will approximately take one unit time.

The correctness of an algorithm is also crucial as it must produce a correctly sorted array, we use loop invariant or mathematical induction to proof the correctness of a sorting algorithm.

Mathematical induction is a mathematical proof technique, a form of direct proof, usually done in two steps. It is used to prove a given statement about any well-ordered set. Most commonly, the well-ordered set is the set of natural numbers. The first step, called the base case or basis, proves that the theorem is true for the number one. The second step, called the inductive step, proves that, if the theorem is true for a given number, then it is also true for the next number. These two steps establish the theorem for all natural numbers.

loop invariant helps us understand why an algorithm gives the correct answer. To use a loop invariant to prove correctness, we must show three things about it:

- Initialization: It is true prior to the first iteration of the loop.
- Maintenance: If it is true before an iteration of the loop, it remains true before the next iteration.
- Termination: When the loop terminates, the invariant gives us a useful property that helps show that the algorithm is correct.

### 3. Insertion Sort

#### Code

Insertion sort is a straightforward sorting algorithm that works like the way you sort playing cards in your hands. The array is virtually split into a sorted and an unsorted part. Values from the unsorted part are picked and placed at the correct position in the sorted part.

##### Algorithm

To sort an array of size  $n$  in ascending order:

- 1: Iterate from  $\text{arr}[1]$  to  $\text{arr}[n]$  over the array.
- 2: Compare the selected element (key) to its predecessor.
- 3: If the key element is smaller than its predecessor, compare it to the elements before. Move the greater elements one position up to make space for the swapped element.

*1 Insertion sort code*

```
""" insertionSort """  
  
def insertionSort(x):  
    for j in range(2, len(x)):  
        key = x[j]  
        i = j - 1  
        while(i >= 0 and x[i] > key):  
            x[i + 1] = x[i]  
            i = i - 1  
        x[i + 1] = key
```

#### Correctness

The loop invariant is that at the start of each iteration of the outer **for** loop, the subarray  $x[1 \dots j - 1]$  consists of the elements originally in  $x[1 \dots j - 1]$  but in sorted order.

**Initialization:** Just before the first iteration,  $j = 2$ . The subarray  $x[1 \dots j - 1]$  is the single element  $x[1]$ , which is the element originally in  $x[1]$ , and it is trivially sorted.

**Maintenance:** At iteration  $j$ , assume that the loop invariant holds

- Line3: save  $x[j]$  in key ( $x[j]$  is now empty)
  - Line4: Set  $i = j - 1$
  - Lines 5-7: The loop sets  $i$  to a value such that
    - $x[i + 1]$  is empty
    - Elements in the subarray  $x[i + 2 \dots j]$  have keys not less than key
    - Elements in the subarray  $x[i + 2 \dots j]$  are the elements originally in  $x[i + 1 \dots j - 1]$
  - Line 8: Set  $x[i + 1]$  to key
  - This makes the sub array  $x[1 \dots j]$  holds the elements originally in this places but in sorted order
- Incrementing  $j$  before the next iteration makes the loop invariant holds



**Termination:** The outer for loop ends when  $j = n + 1$ . Substituting  $n+1$  for  $j$  in the loop invariant, the subarray  $x[1 \dots n]$  consists of the elements originally in  $x[1 \dots n]$  but in sorted order. In other words, the entire array is sorted!

## Running Time

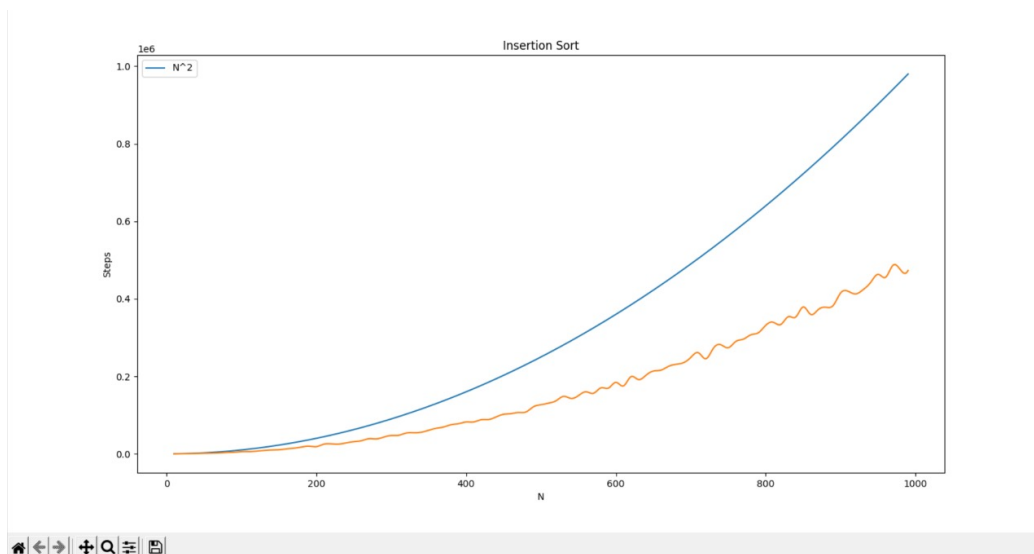
The Best-case running time of the insertion sort is  $\Omega(n) = n$ , while the worst-case is  $O(n) = n^2$

1 insertion sort with steps counter

```
""" insertionSort """
def insertionSort(x, count):
    for j in range(1, len(x)):
        count=count+3
        key=x[j]
        i=j-1
        while(i>=0 and x[i]>key):
            count=count+2
            x[i+1]=x[i]
            i=i-1
        x[i+1]=key
    return count
```

We altered the code to count the steps in each iteration, counting the significant steps only (the steps done in loops) and returning them to plot the curve against the asymptotic upper bound ( $n^2$ ). The details of plotting are explained thoroughly in section 11 GUI.

2insertion sort graph



These fluctuations happen due to the random generation of the arrays, for example an array of 150 elements can be initially sorted better than an array of 100 elements and thus take less steps to be sorted. Moreover, values of  $n^2$  have been scaled down to allow the appearance of both graphs near each other for better comparisons. What matters to us is that we validated that the Insertion sort algorithm GROWS LIKE  $n^2$  and that is what is shown on the graph.

## 4. Bubble Sort

### Code

Bubble Sort is the simplest sorting algorithm that works by repeatedly swapping the adjacent elements if they are in wrong order

*3 Bubble sort code*

```
def bubbleSort(arr):  
    n = len(arr)  
  
    for i in range(n-1):  
        for j in range(0, n-i-1):  
            if arr[j] > arr[j + 1] :  
                arr[j], arr[j + 1] = arr[j + 1], arr[j]
```

### Correctness

The loop invariant for inner loop is that after iteration  $j$  of the inner loop the maximum element of  $(arr_1, arr_2, arr_3, \dots, arr_{j+1})$  is in position  $j+1$ .

Proof by induction:

Base case: ( $j = 0$ ) Iteration  $j = 0$  is prior to the loop. In this step the maximum element of  $(a_1)$  is in position 1.

Inductive Hypothesis: Assume prior to iteration  $j + 1$  that the maximum element of  $(a_1, a_2, \dots, a_{j+1})$  is in position  $j + 1$ . Inductive Step: ( $j > 0$ ) We need to show that after iteration  $j + 1$  that the maximum element of  $(a_1, a_2, \dots, a_{j+2})$  is in position  $j + 2$ . We know by assumption that the maximum element of  $(a_1, a_2, \dots, a_{j+1})$  is in position  $j + 1$ . Either this is the maximum element of  $(a_1, a_2, \dots, a_{j+2})$  or element  $a_{j+2}$  is. In this iteration we compare  $a_{j+1}$  to  $a_{j+2}$  and swap them if  $a_{j+1} > a_{j+2}$ . So after this iteration the maximum element of  $(a_1, a_2, \dots, a_{j+2})$  is in position  $j + 2$ . So by induction after iteration  $j$  of the loop the maximum element of  $(a_1, a_2, \dots, a_n)$  is in position  $j + 1$ .

The loop invariant for outer loop is that after iteration  $i$  of the outer loop the  $i$  maximum elements of  $(a_1, a_2, \dots, a_n)$  are in ascending order in positions  $n - i + 1$  to  $n$ . Proof induction:



Base case: ( $i = 0$ ) Prior to the loop the 0 maximum elements of ( $a_1, a_2, a_3, \dots$ ) are in ascending order in no positions (positions  $n + 1$  to  $n$ ).

Inductive Hypothesis: Assume prior to iteration  $i + 1$  that the  $i$  maximum elements of ( $a_1, a_2, a_3, \dots$ ) are in ascending order in positions  $n - i + 1$  to  $n$ . Inductive Step: (1 a 0) By assumption we know that the  $i$  maximum elements of ( $a_1, a_2, a_3, \dots$ ) are in ascending order in positions  $n - i + 1$  to  $n$ . We need to show that the maximum element of ( $a_1, a_2, a_3, \dots, a_{n-1}$ ) is in position  $n - i$  after iteration  $i + 1$ . By the inner loop invariant we know after iteration  $j$  of the inner loop that the maximum element of ( $a_1, a_2, a_3, \dots, a_{n-j}$ ) is in position  $n - j$ . In particular, iteration  $i + 1$  the outer loop ends after iteration  $n - i - 1$  of the inner loop. So after iteration  $n - i - 1$  of the inner loop the maximum element of ( $a_1, a_2, a_3, \dots, a_{n-i}$ ) is in position  $n - i$ . By induction we have shown that after iteration  $i$  of the outer loop the  $i$  maximum elements of ( $a_1, a_2, a_3, \dots, a_n$ ) are in ascending order in positions  $n - i + 1$  to  $n$ .

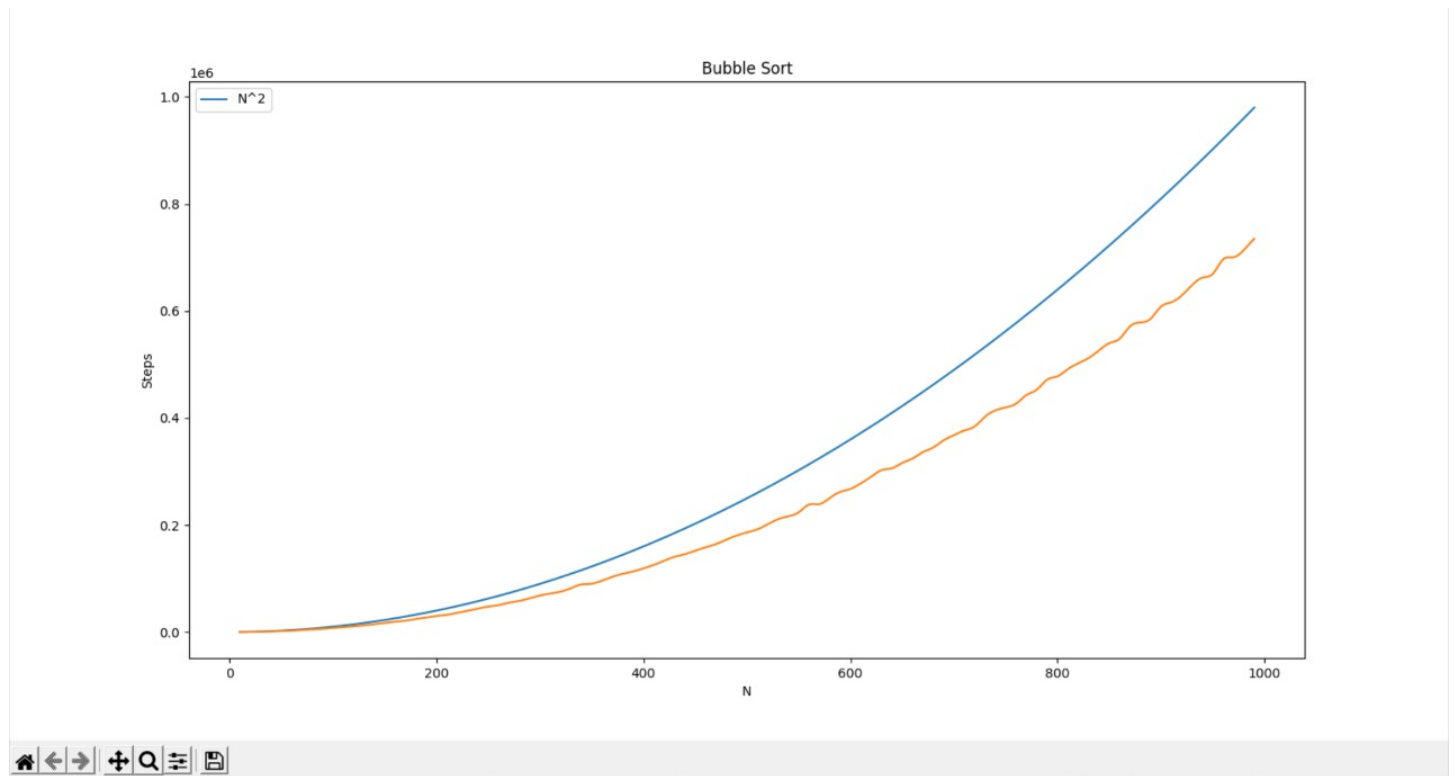
## Running time

The Best-case running time of the bubble sort is  $\Omega(n) = n$ , while the worst-case is  $O(n) = n^2$

4 bubble sort with step counter

```
def bubbleSort(arr, count):  
    n = len(arr)  
    count=count+1  
    for i in range(n-1):  
        count=count+1  
        for j in range(0, n-i-1):  
            count=count+1  
            if arr[j] > arr[j + 1] :  
                count=count+1  
                arr[j], arr[j + 1] = arr[j + 1], arr[j]  
    return count
```

We altered the code to count the steps in each iteration, counting the significant steps only (the steps done in loops) and returning them to plot the curve against the asymptotic upper bound ( $n^2$ ). The details of plotting are explained thoroughly in section 11 GUI.



These fluctuations happen due to the random generation of the arrays, for example an array of 150 elements can be initially sorted better than an array of 100 elements and thus take less steps to be sorted. Moreover, values of  $n^2$  have been scaled down to allow the appearance of both graphs near each other for better comparisons. What matters to us is that we validated that the Bubble sort algorithm GROWS LIKE  $n^2$  and that is what is shown on the graph

## 5. Selection sort

### Code

The selection sort algorithm sorts an array by repeatedly finding the minimum element (considering ascending order) from unsorted part and putting it at the beginning. The algorithm maintains two subarrays in a given array.

- 1) The subarray which is already sorted.
- 2) Remaining subarray which is unsorted.

In every iteration of selection sort, the minimum element (considering ascending order) from the unsorted subarray is picked and moved to the sorted subarray.

6 Selection sort code

```
def selectionsort(A):  
    for i in range(len(A)):  
        min_idx = i  
        for j in range(i+1, len(A)):  
            if A[min_idx] > A[j]:  
                min_idx = j  
        A[i], A[min_idx] = A[min_idx], A[i]
```

### Correctness

Loop Invariant: Before the start of each loop,  $A[\text{min\_idx}]$  is less than or equal to  $A[i..j-1]$ .

Initialization: Prior to the first iteration of the loop,  $j=i+1$ . So the array segment  $A[i..j-1]$  is really just spot  $A[i]$ . Since line 2 of the code sets  $\text{min\_idx}=i$ , we have that  $\text{min\_idx}$  indexes the smallest element (the only element) in subarray  $A[i..j-1]$  and hence the loop invariant is true. Maintenance: Before pass  $j$ , we assume that  $\text{min\_idx}$  indexes the smallest element in the subarray  $A[i..j-1]$ . During iteration  $j$  we have two cases : either  $A[j]<A[\text{min\_idx}]$  or  $A[j]\geq A[\text{min\_idx}]$ . In the second case, the if statement on line 4 is not true, so nothing is executed. But now  $\text{min\_idx}$  indexes the smallest element of  $A[i..j]$ . In the first case, line 5 switches  $\text{min}$  to index location  $j$  since it is the smallest. If  $\text{min\_idx}$  indexes an element less than or equal to subarray  $A[i..j-1]$  and now  $A[j]<A[\text{min}]$ , then it must be the case that  $A[j]$  is less than or equal to elements in subarray  $A[i..j-1]$ . Line 5 switches  $\text{min\_idx}$  to index this new location and

hence after the loop iteration finishes, `min_idx` indexes the smallest element in subarray `A[i..j]`. Termination: At termination of the inner loop, `min_idx` indexes an element less than or equal to all elements in subarray `A[i..n]` since `j=n+1` upon termination. This finds the smallest element in this subarray and is useful to us in the outer loop because we can move that next smallest item into the correct location.

## Running time

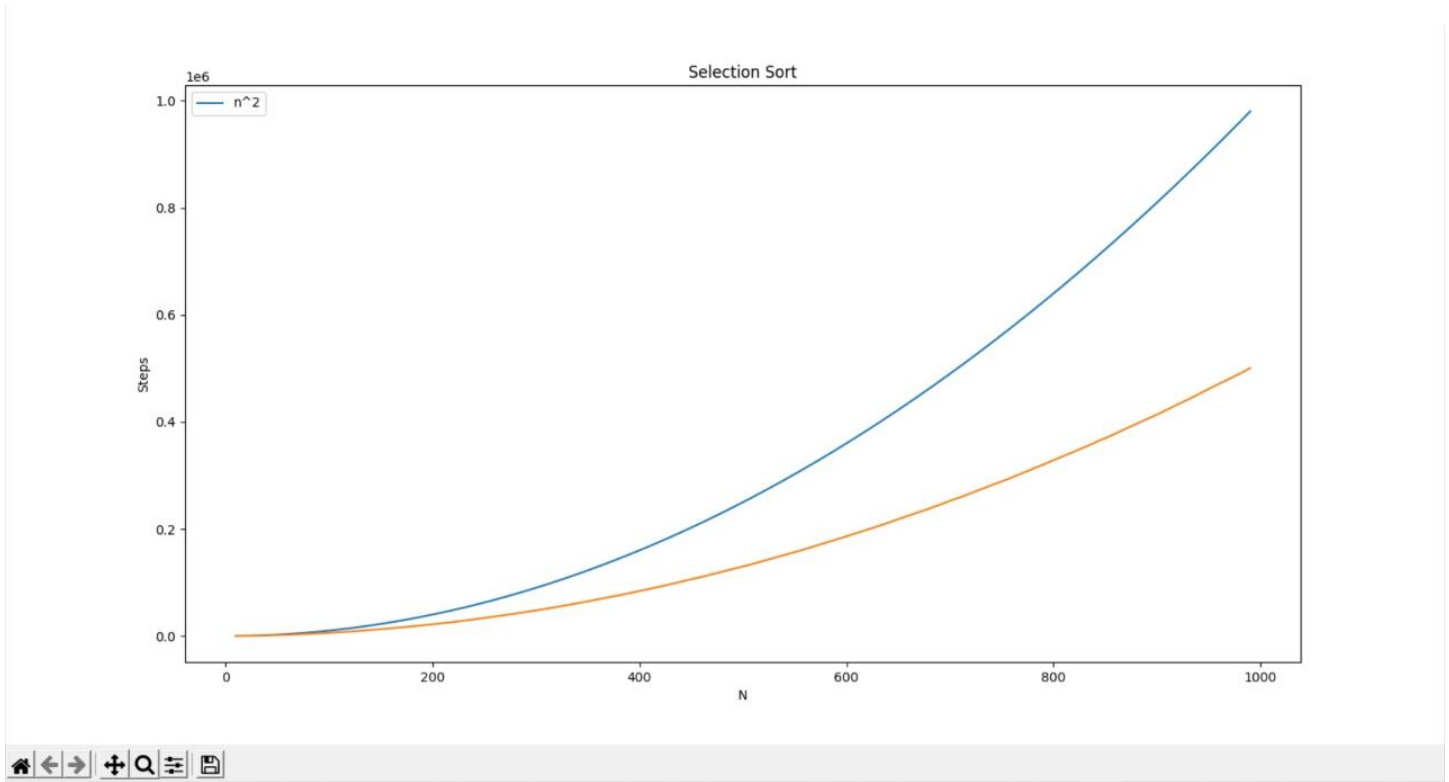
The Best-case running time of the selection sort is same as the worst-case  $\Omega(n) = O(n) = n^2$

7 selection sort with step counter

```
def selectionsort(A, count):  
    for i in range(len(A)):  
        count=count+3  
        min_idx = i  
        for j in range(i+1, len(A)):  
            if A[min_idx] > A[j]:  
                count=count+2  
                min_idx = j  
            count=count+1  
        A[i], A[min_idx] = A[min_idx], A[i]  
    return count
```

We altered the code to count the steps in each iteration, counting the significant steps only (the steps done in loops) and returning them to plot the curve against the asymptotic upper bound ( $n^2$ ). The details of plotting are explained thoroughly in section 11 GUI.

## 8 Selection sort graph



values of  $n^2$  have been scaled down to allow the appearance of both graphs near each other for better comparisons .

## 6. Merge sort

### Code

Merge Sort is a Divide and Conquer algorithm. It divides the input array into two halves, conquers by recursively sorting the two subarrays and then merges the two sorted halves. The recursion bottoms out when the subarray has just 1 element, so that it's trivially sorted.

*9 merge sort function*

```
""" merge sort function """  
  
def merge_sort(array, left_index, right_index):  
    if left_index >= right_index:  
        return  
  
    middle = (left_index + right_index)//2  
    merge_sort(array, left_index, middle)  
    merge_sort(array, middle + 1, right_index)  
    merge(array, left_index, right_index, middle)
```

```

""" Merge code function """
""" In merge code i dont use sentinels """

def merge(array, left_index, right_index, middle):

    left_copy = array[left_index:middle + 1]
    right_copy = array[middle+1:right_index+1]

    left_copy_index = 0
    right_copy_index = 0
    sorted_index = left_index

    while left_copy_index < len(left_copy) and right_copy_index < len(right_copy):

        if left_copy[left_copy_index] <= right_copy[right_copy_index]:
            array[sorted_index] = left_copy[left_copy_index]
            left_copy_index = left_copy_index + 1

        else:
            array[sorted_index] = right_copy[right_copy_index]
            right_copy_index = right_copy_index + 1

        sorted_index = sorted_index + 1

    while left_copy_index < len(left_copy):
        array[sorted_index] = left_copy[left_copy_index]
        left_copy_index = left_copy_index + 1
        sorted_index = sorted_index + 1

    while right_copy_index < len(right_copy):
        array[sorted_index] = right_copy[right_copy_index]
        right_copy_index = right_copy_index + 1
        sorted_index = sorted_index + 1

```

## Correctness

the loop invariant:

- At the start of each iteration of the for loop, the subarray `array[left_index...k-1]` contains the `k-left_index` smallest elements of `L_copy[1..n1+1]` and `Right_copy[1..n2+1]`, in sorted order.
- Moreover, `Left_copy[i]` and `Right_copy[j]` are the smallest elements of their arrays that have not been copied back into A.

Initialization: prior to the first iteration of the loop, we have `k=left_index`, so that the subarray `array[left_index...k-1]` is empty. This empty subarray contains the `k-left_index=0` smallest elements of `Left_copy` and `Right_copy`,

and since `i=j=1`, both `Left_copy[i]` and `Right_copy[j]` are the smallest elements of their arrays that have not been copied back into array.



Maintenance: Let us suppose that  $\text{Left\_copy}[i] \leq \text{Right\_copy}[j]$ . Then  $\text{Left\_copy}[i]$  is the smallest element not yet copied back into array. Because  $\text{array}[\text{left\_index}..k-1]$  contains the  $k - \text{left\_index}$  smallest elements, after the loop copies  $\text{Left\_copy}[i]$  into  $\text{array}[k]$ , the subarray  $\text{array}[\text{left\_index}..k]$  contains the  $k - \text{left\_index} + 1$  smallest elements. Incrementing  $k$  and  $i$  re-establishes the loop invariant for the next iteration.

Termination : At termination,  $k = \text{middle} + 1$ . The subarray  $\text{array}[\text{left\_index}..k-1]$  which is  $\text{array}[\text{left\_index}..\text{middle}]$ , contains  $k - \text{left\_index} = \text{middle} - \text{left\_index} + 1$  smallest elements of  $\text{Left\_copy}[1..n_1+1]$  and  $\text{Right\_copy}[1..n_2+1]$  in sorted order.

## Running time

The Best-case running time of the merge sort is same as the worst-case  $\Omega(n) = O(n) = n \log n$

*11 merge function with step counter*

```

""" Merge code function """
""" In merge code i dont use sentinels """

def merge(array, left_index, right_index, middle):

    left_copy = array[left_index:middle + 1]
    right_copy = array[middle+1:right_index+1]

    countM=len(left_copy)+len(right_copy)
    left_copy_index = 0
    right_copy_index = 0
    sorted_index = left_index

    while left_copy_index < len(left_copy) and right_copy_index < len(right_copy):

        if left_copy[left_copy_index] <= right_copy[right_copy_index]:
            array[sorted_index] = left_copy[left_copy_index]
            left_copy_index = left_copy_index + 1

        else:
            array[sorted_index] = right_copy[right_copy_index]
            right_copy_index = right_copy_index + 1

        countM=countM+2
        sorted_index = sorted_index + 1

    while left_copy_index < len(left_copy):
        array[sorted_index] = left_copy[left_copy_index]
        left_copy_index = left_copy_index + 1
        sorted_index = sorted_index + 1
        countM=countM+3

    while right_copy_index < len(right_copy):
        array[sorted_index] = right_copy[right_copy_index]
        right_copy_index = right_copy_index + 1
        sorted_index = sorted_index + 1
        countM=countM+3

    return countM

```

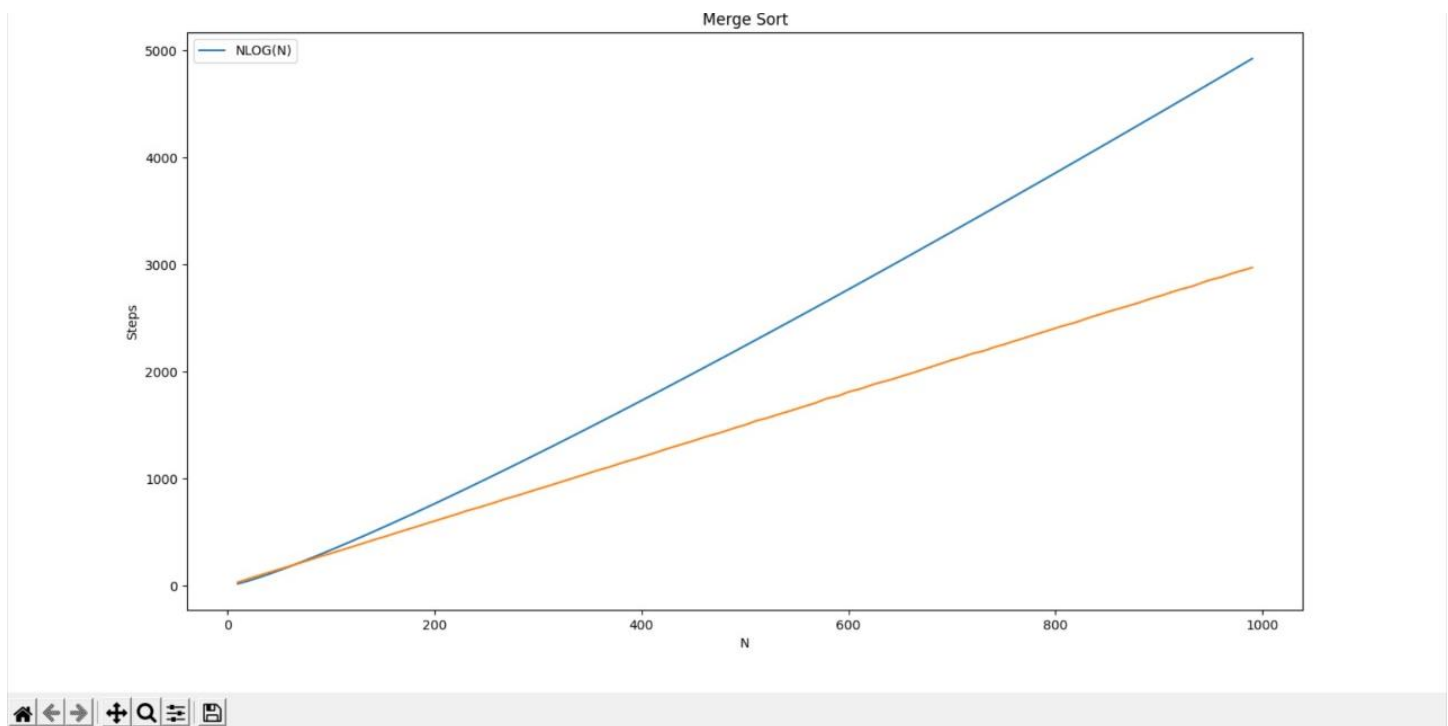
```

""" merge sort function """
def merge_sort(array, left_index, right_index, count):
    if left_index >= right_index:
        return count
    middle = (left_index + right_index)//2
    merge_sort(array, left_index, middle, count)
    merge_sort(array, middle + 1, right_index, count)
    c=merge(array, left_index, right_index, middle)
    count=count+c

    return count

```

We altered the code to count the steps in each iteration, counting the significant steps only (the steps done in loops) and returning them to plot the curve against the asymptotic upper bound ( $n \log n$ ). The details of plotting are explained thoroughly in section 11 GUI.



values of  $n \log n$  have been scaled down to allow the appearance of both graphs near each other for better comparisons.

## 7. Counting Sort

### Code

Counting sort is a sorting technique based on keys between a specific range. It works by counting the number of objects having distinct key values (kind of hashing). Then doing some arithmetic to calculate the position of each object in the output sequence

```
def countingSort(inputArray, count):  
    maxElement= max(inputArray)  
  
    countArrayLength = maxElement+1  
  
    countArray = [0] * countArrayLength  
    count=count+(2*len(inputArray))+countArrayLength-1  
  
    for el in inputArray:  
        countArray[el] += 1  
  
    for i in range(1, countArrayLength):  
        countArray[i] += countArray[i-1]  
  
    outputArray = [0] * len(inputArray)  
    i = len(inputArray) - 1  
    while i >= 0:  
        currentEl = inputArray[i]  
        countArray[currentEl] -= 1  
        newPosition = countArray[currentEl]  
        outputArray[newPosition] = currentEl  
        i -= 1  
  
    return count
```

14 Counting Sort Code

### Correctness

Because the algorithm uses only simple for loops, without recursion or subroutine calls, it is straightforward to analyze. The initialization of the count array, and the second for loop which performs a prefix sum on the count array, each iterate at most  $k + 1$  times and therefore take  $O(k)$  time. The other two for loops, and the initialization of the output array, each take  $O(n)$  time. Therefore, the time for the whole algorithm is the sum of the times for these steps,  $O(n + k)$ .

Because it uses arrays of length  $k + 1$  and  $n$ , the total space usage of the algorithm is also  $O(n + k)$ . For problem instances in which the maximum key value is significantly smaller than the number of items, counting sort can be highly space-efficient, as the only storage it uses other than its input and output arrays is the Count array which uses space  $O(k)$ .

### Running Time

The Best-case and the worst case running time of the Counting sort is  $n + k$

```

def main_counting(a):
    times=[0]*10
    arr_n=[0]*10
    time_notation=[0]*10
    i=0

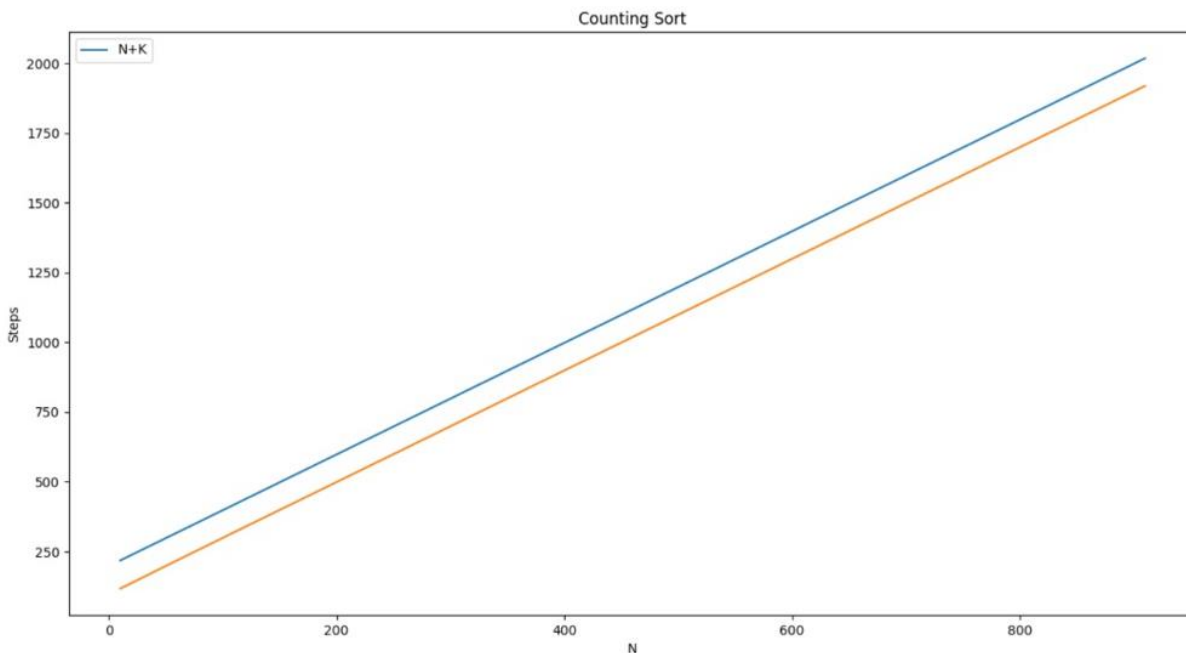
    for n in range(10,1000,100):
        rand_array=[0]*n
        randomgenerator(rand_array,n)
        count=0
        count=countingSort(rand_array,count)
        times[i]=count
        i+=1

    """ filling arr_n """
    z=0
    for i in range(10,1000,100):
        arr_n[z]=i
        z+=1
    '''filling time_notation'''
    if(a==0):
        w=0
        for w in range(len(arr_n)):
            time_notation[w]=(arr_n[w]+99)*2

```

We altered the code to count the steps in each iteration, counting the significant steps only (the steps done in loops) and returning them to plot the curve against the asymptotic upper bound ( $n+k$ ). The details of plotting are explained thoroughly in section 11 GUI.

16 Counting Sort Graph



## 8. Heap Sort

### Heap Data Structure

The (binary) heap data structure is an array object. Each node of the tree corresponds to an element of the array. The tree is completely filled on all levels except possibly the lowest, which is filled from the left up to a point. The root of the tree is  $A[1]$ . A node at  $A[i]$  (as a parent) has at most two children. The left child with index  $2[i]$  and the right child with index  $2[i] + 1$ .

There are two kinds of binary heaps: max-heaps and min-heaps. In both kinds, the values in each node must satisfy a heap property. Max-heap property:  $A_{Parent\ i} \geq A(i)$  Min-heap property:  $A_{Parent\ i} \leq A(i)$ . The value at the root is the largest for max-heap, and the smallest for minheap.

### MAX-HEAPIFY

The height of a node is defined as the number of edges on the longest simple downward path from the node to a leaf. The height of an  $n$  nodes heap  $h$  is the height of its root =  $\lg [n]$ . The heap-size property represents how many elements in the heap are stored within the array  $A$ .

The MAX-HEAPIFY procedure, which runs in  $O(\lg n)$  time, is the key to maintaining the max-heap property.

The largest(i) procedure, returns the index of the parent node.

The l(i) procedure, returns the index of the left child node.

The r(i) procedure, returns the index of the right child node.

MAX-HEAPIFY procedure is used to maintaining the max-heap property. It takes as parameters the array  $A$ , and an index  $i$  and  $n$  for random generated numbers. It assumes that the binary trees rooted at  $l(i)$  and  $r(i)$  are maxheaps, but that  $A(i)$  might be smaller than its children.

```
def MaxHeapify(arr, n, i, count):
    largest = i
    l = 2 * i + 1
    r = 2 * i + 2
    count=count+6

    if l < n and arr[i] < arr[l]:
        largest = l

    if r < n and arr[largest] < arr[r]:
        largest = r

    if largest != i:
        arr[i],arr[largest] = arr[largest],arr[i]
        MaxHeapify(arr, n, largest, count)
    return count
```

17 Max Heapify Code

## Build Max Heap

BUILD-MAX-HEAP procedure, which runs in linear time, produces a maxheap from an unordered input array.

The BUILD-MAX-HEAP procedure is used to convert an array to a maxheap. The elements of the subarray  $A[\{(n/2) + 1, \dots, n\}]$  are all leaves of the heap tree, and so each is a 1-element heap to start with.

The procedure MAX-HEAPIFY is used in a bottom-up manner to convert the array into a max-heap. It goes through the remaining nodes of the tree and runs MAXHEAPIFY on each one

*18 buildmaxheap code*

```
def buildmaxheap(arr,n,count):  
    for i in range((n // 2) - 1, -1, -1):  
        count=MaxHeapify(arr, n, i,count)  
    return count
```

## Heap Sort Code

The heapsort algorithm starts by using BUILD-MAX-HEAP to build a max-heap on the input array  $A[1 \dots n]$  where  $n = A.length$ . Since the maximum element of the array is stored at the root  $A[1]$  we can put it into its correct final position by exchanging it with  $A[n]$ . •Decrement the heap size to eliminate the last element from the heap. Restore the heap property by calling MAX-HEAPIFY ( $A, 1$ ). Repeat this operation down to a heap size of 2.

*heap sort code*

```
def heapSort(arr,count):  
    n = len(arr)  
  
    count=buildmaxheap(arr,n,count)  
  
    for i in range(n-1, 0, -1):  
        arr[i], arr[0] = arr[0], arr[i]  
        count=MaxHeapify(arr, i, 0,count)  
    return count
```

## Correctness

**Initialization:** Prior to the first iteration. Everything is a leaf so it is already a heap.

**Maintenance:** Let us assume that we have a working solution till now. The children of node (i) are numbered higher than (i). MAX-HEAPIFY preserves the loop invariant as well. We maintain the invariance at each step.

**Termination:** Terminates when the (i) drops down to 0 and by the loop invariant, each node is the root of a max-heap.

## Running Time

Heapsort has a running time of  $O(n \log n)$ .

Building the max-heap from the unsorted list requires  $O(n)$  calls to the MAX-HEAPIFY function, each of which takes  $O(\log n)$  time. Thus, the running time of heap sort is  $O(n \log n)$ .

```
def MaxHeapify(arr, n, i, count):
    largest = i
    l = 2 * i + 1
    r = 2 * i + 2
    count=count+6

    if l < n and arr[i] < arr[l]:
        largest = l

    if r < n and arr[largest] < arr[r]:
        largest = r

    if largest != i:
        arr[i], arr[largest] = arr[largest], arr[i]
        MaxHeapify(arr, n, largest, count)
    return count

def buildmaxheap(arr, n, count):
    for i in range((n // 2) - 1, -1, -1):
        count=MaxHeapify(arr, n, i, count)
    return count
```

19 heap sort code with step count 2

```
def heapSort(arr, count):
    n = len(arr)

    count=buildmaxheap(arr, n, count)

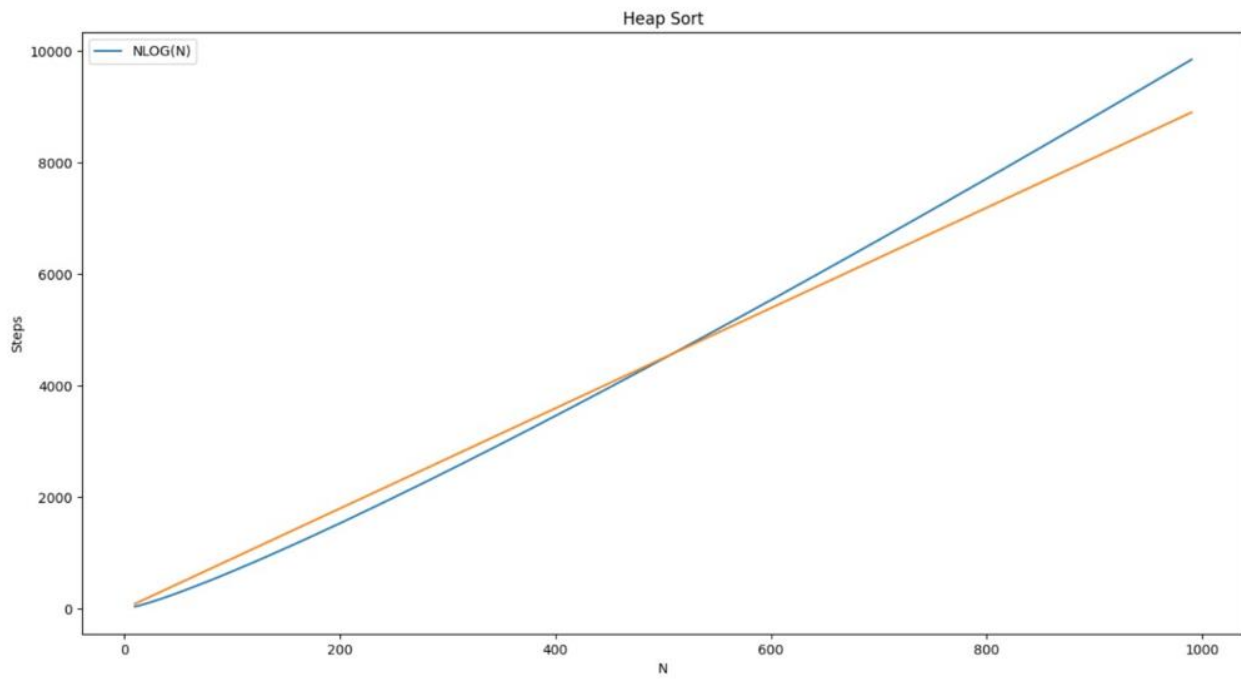
    for i in range(n-1, 0, -1):
        arr[i], arr[0] = arr[0], arr[i]
        count=MaxHeapify(arr, i, 0, count)
    return count
```

20 heap sort code with step count 1

We altered the code to count the steps in each iteration, counting the significant steps only (the steps done in loops) and returning them to plot the curve against the asymptotic upper bound ( $n \log n$ ). The details of plotting are explained thoroughly in section 11 GUI.



### 21 Heap Sort Graph



values of  $n \log n$  have been scaled down to allow the appearance of both graphs near each other for better comparisons .

## 9. Radix Sort

### Code

The idea of Radix Sort is to do digit by digit sort starting from least significant digit to most significant digit. Radix sort uses counting sort as a subroutine to sort

Radix sort sorts the input array first based on the value of the least significant digit first using a *stable* algorithm. The sorted array is then sorted based on the next least significant digit, and so on. For a  $d$  digit number, it makes  $d$ -sorts. Radix sort is often used to sort records of information that are keyed by multiple fields.

23 Counting Sort in Radix

```
def countingSort(arr, exp1, count1):  
    n = len(arr)  
    output = [0] * (n)  
    count = [0] * (10)  
  
    for i in range(0, n):  
        index = (arr[i]/exp1)  
        count[int((index)%10)] += 1  
  
    for i in range(1,10):  
        count[i] += count[i-1]  
  
    i = n-1  
    while i>=0:  
        index = (arr[i]/exp1)  
        output[ count[ int((index)%10) ] - 1] = arr[i]  
        count[int((index)%10)] -= 1  
        i -= 1  
  
    i = 0  
    for i in range(0,len(arr)):  
        arr[i] = output[i]  
    return count1
```

22 Radix Sort Code

```
def radixSort(arr, count1):  
    max1 = max(arr)  
    exp = 1  
  
    while (max1/exp) > 0:  
        countingSort(arr, exp, count1)  
        exp *= 10  
  
    count1=count1+(2*len(arr))+9  
    return count1*2
```

### Correctness

**Loop Invariant:** After the  $i$ th iteration of the loop, the elements are sorted by their last  $i$  digits.

**Initialization:** The array is trivially sorted on the last 0 digits.

**Maintenance:** Let's assume that the array is sorted on the last  $i-1$  digits. After we sort on the  $i$ th digit, the array will be sorted on the last  $i$  digits. It is obvious that elements with different digit in the  $i$ th position are ordered accordingly; in the case of the same  $i$ th digit, we still get a correct order, because we're using a stable sort and the elements were already sorted on the last  $i-1$  digits.

**Termination:** The loop terminates when  $i=d+1$ . Since the invariant holds, we have the numbers sorted on  $d$  digits.

## Running Time

Given  $n$   $d$ -digit numbers in which each digit can take on up to  $k$  possible values, RADIX-SORT correctly sorts these numbers in  $\Theta(d(k+n))$  time if the stable sort it uses takes  $\Theta(k+n)$  time.

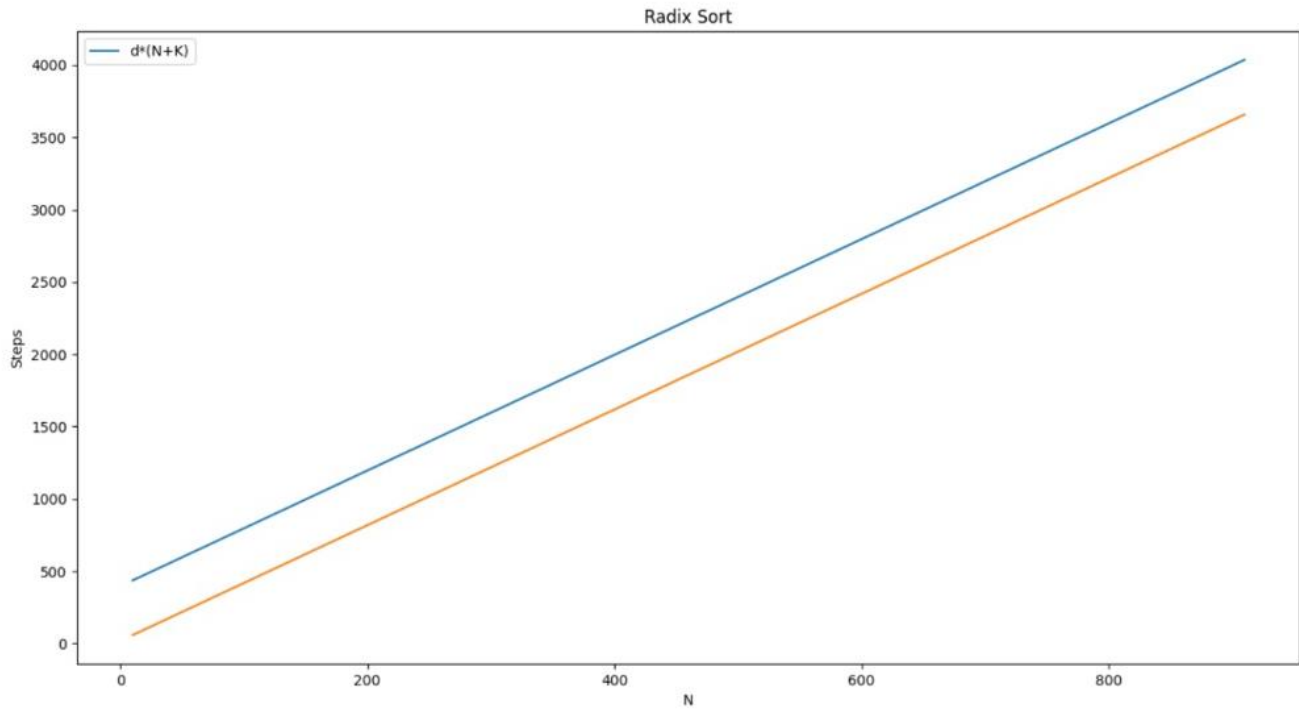
When  $d$  is constant, and  $k = O(n)$  the running time is  $\Theta(n)$ .

The running time is  $O(d*(N+K))$

24 Radix Code with Count steps

```
1 def countingSort(arr, exp1, count1):
2     n = len(arr)
3     output = [0] * (n)
4     count = [0] * (10)
5
6     for i in range(0, n):
7         index = (arr[i]/exp1)
8         count[int((index)%10)] += 1
9
10    for i in range(1,10):
11        count[i] += count[i-1]
12
13    i = n-1
14    while i>=0:
15        index = (arr[i]/exp1)
16        output[ count[ int((index)%10) ] - 1] = arr[i]
17        count[int((index)%10)] -= 1
18        i -= 1
19
20    i = 0
21    for i in range(0,len(arr)):
22        arr[i] = output[i]
23    return count1
24
25 def radixSort(arr,count1):
26     max1 = max(arr)
27     exp = 1
28
29     while (max1/exp) > 0:
30         countingSort(arr,exp,count1)
31         exp *= 10
32
33     count1=count1+(2*len(arr))+9
34     return count1*2
```

We altered the code to count the steps in each iteration, counting the significant steps only (the steps done in loops) and returning them to plot the curve against the asymptotic upper bound ( $d*(N+K)$ ). The details of plotting are explained thoroughly in section 11 GUI.



## 10. Quick Sort

### Code

Quicksort, like merge sort, applies the divide-and-conquer paradigm. The three-step divide-and-conquer process for sorting a typical subarray  $A[p \dots r]$  are:

- Divide: Partition (and rearrange) the subarray  $A[p \dots r]$  into two (possibly empty) subarrays  $A[p \dots q - 1]$  and  $A[q + 1 \dots r]$  such that:  $A[p \dots q - 1] \leq A[q] < A[q + 1 \dots r]$
- Conquer: Sort the two subarrays  $A[p \dots q - 1]$  and  $A[q + 1 \dots r]$  by recursive call to quicksort
- Combine: Not needed

26 Quick Sort Code

```
def partition(arr, low, high, count):
    i = (low-1)
    pivot = arr[high]

    for j in range(low, high):
        if arr[j] <= pivot:
            i = i+1
            arr[i], arr[j] = arr[j], arr[i]
            count=count+3
        count=count+1

    arr[i+1], arr[high] = arr[high], arr[i+1]
    return (i+1), count;

def quickSort(arr, low, high, count):
    if len(arr) == 1:
        return count

    if low < high:
        pi, count = partition(arr, low, high, count)
        quickSort(arr, low, pi-1, count)
        quickSort(arr, pi+1, high, count)
    return count
```

### Correctness

**Initialization:** At the start of the first iteration,  $i = p - 1$ , and  $j = p$ . The three parts of the loop invariant will be trivially satisfied

- 1- No values lie between  $p$  and  $p - 1$
- 2- No values lie between  $p$  and  $p - 1$
- 3- Line 1 sets  $x = A[r]$

**Maintenance:** At the start of an iteration  $j$ , and assuming that the loop invariant holds. The third part is always satisfied since the loop does not change  $A[r]$ , or  $x$ . There are two cases depending on the comparison in line 4

Case 1:  $A[j] > x$ , Case 2:  $A[j] \leq x$ .

- Case 1:  $A[j] > x$ . At the start of the next iteration  $j_n = j + 1$ , and  $i_n = i$ .

The three parts are satisfied at  $j = j_n$  since:

Since  $A[k] \leq x$  for values of  $k$  between  $p$  and  $i$  this implies that  $A[k] \leq x$  for values of  $k$  between  $p$  and  $i_n$

2- Since  $A[k] > x$  for values of  $k$  between  $i + 1$  and  $j - 1$ , and  $A[j] > x$  this implies that  $A[k] > x$  for values of  $k$  between  $i + 1$  and  $j$  in other words  $A[k] > x$  for values of  $k$  between  $i_n + 1$  and  $j_n - 1$ . This part is still satisfied

- Case 2:  $A[j] \leq x$ . At the start of the next iteration  $j_n = j + 1$ , and  $i_n = i + 1$ . Line 6 swaps  $A[i + 1]$  and  $A[j]$  making  $A[i + 1] = A[i_n] \leq x$ , and  $A[j] = A[j_n - 1] > x$ . The three parts are satisfied at  $j = j_n$  since :
  - 1- Since  $A[k] \leq x$  for values of  $k$  between  $p$  and  $i$ , and  $A[i + 1] \leq x$  this implies that  $A[k] \leq x$  for values of  $k$  between  $p$  and  $i_n$
  - 2- Since  $A[k] > x$  for values of  $k$  between  $[i + 1]$  and  $[j - 1]$ , and  $A[j] > x$  this implies  $A[k] > x$  for values of  $k$  between  $[i_n + 1]$  and  $j_n - 1$ . These two parts are still satisfied

**Termination:** As the loop terminates  $j = r$ . The loop invariant becomes

- 1-  $A[k] \leq x$  for values of  $k$  between  $p$  and  $i$
- 2-  $A[k] > x$  for values of  $k$  between  $[i + 1]$  and  $[r - 1]$
- 3-  $A[r] = x$

Line 7 swaps the first element in the partition with values greater than  $x$ . Line 8 returns the position of the pivot element

## Running time

**Worst case:** The worst case occurs when the partition process always picks greatest or smallest element as pivot. If we consider above partition strategy where last element is always picked as pivot, the worst case would occur when the array is already sorted in increasing or decreasing order. Following is recurrence for worst case.

$$T(n) = T(n-1) + O(n)$$

Where worst case  $O(n^2)$

**Best Case:** The best case occurs when the partition process always picks the middle element as pivot. Following is recurrence for best case

$$T(n) = 2T(n/2) + O(n)$$

Where best case is  $O(n \log n)$

```

def partition(arr, low, high, count):
    i = (low-1)
    pivot = arr[high]

    for j in range(low, high):
        if arr[j] <= pivot:
            i = i+1
            arr[i], arr[j] = arr[j], arr[i]
            count=count+3
        count=count+1

    arr[i+1], arr[high] = arr[high], arr[i+1]
    return (i+1),count;

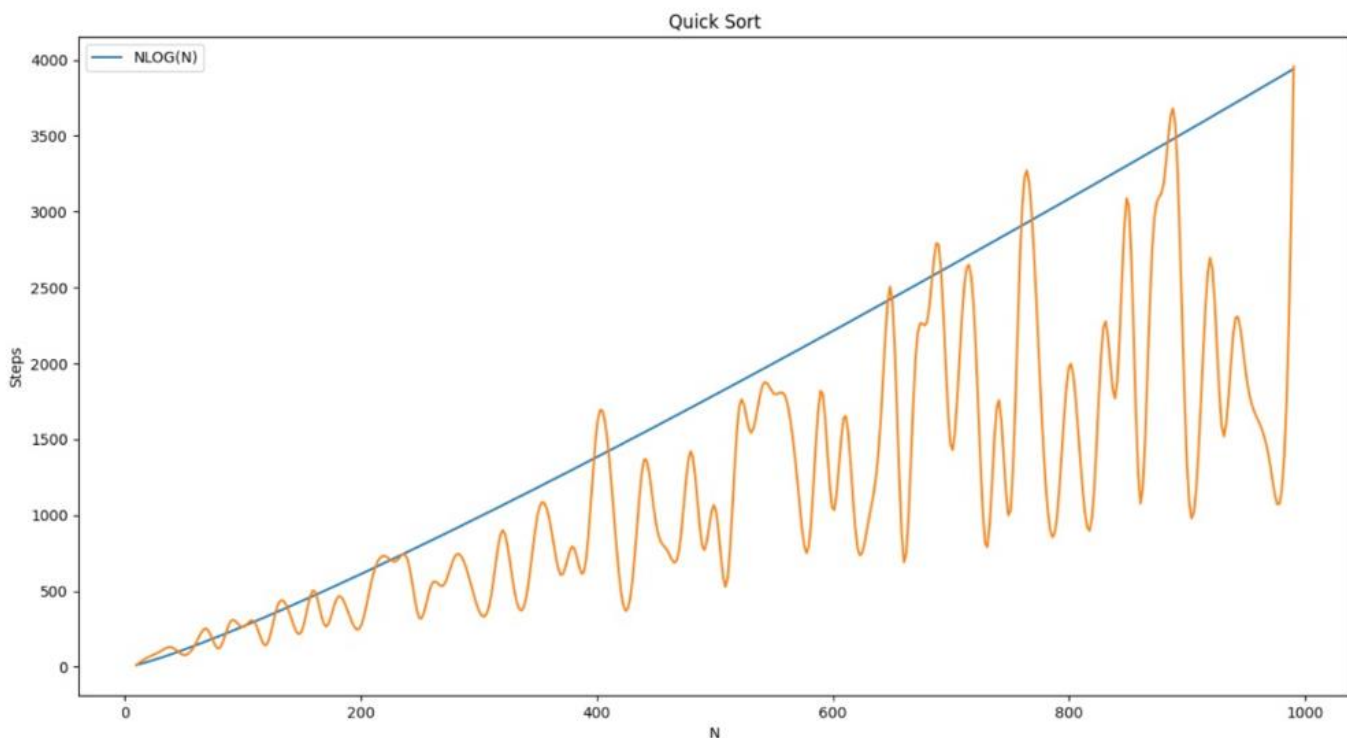
def quickSort(arr, low, high, count):
    if len(arr) == 1:
        return count

    if low < high:
        pi, count = partition(arr, low, high, count)
        quickSort(arr, low, pi-1, count)
        quickSort(arr, pi+1, high, count)
    return count

```

We altered the code to count the steps in each iteration, counting the significant steps only (the steps done in loops) and returning them to plot the curve against the asymptotic upper bound ( $n \log n$ ). The details of plotting are explained thoroughly in section 11 GUI.

30 quick sort graph





These fluctuations happen due to the random generation of the arrays, for example an array of 150 elements can be initially sorted better than an array of 100 elements and thus take less steps to be sorted. Moreover, values of  $n \log n$  have been scaled down to allow the appearance of both graphs near each other for better comparisons. What matters to us is that we validated that the Insertion sort algorithm GROWS LIKE  $n \log n$  and that is what is shown on the graph

## 11. GUI

### Random generator

This function generates random numbers between 10 and 99 and fills the array with them. This function only helps speed up the array creation and filling with random elements during run time.

32 random generator code

```
import random

def randomgenerator(arr,n):
    for i in range(n):
        x = random.randint(10,99)
        arr[i]= x
    return arr
```

### Main sort functions

Inside each sorting algorithm file, alongside the sorting algorithm function explained previously for each sorting algorithms, there is a function called main\_sortname (main\_insertion for insertion sort, main\_quick for quick sort, ... etc.).

34 main\_sort i

```
def main_insertion(a):
```

These functions generates arrays filled with random numbers(using random genaretor) of sizes 10 to 1000 with step 10(array of size 10, then array of size 20, array of size 30,.....,array of size 980, array of size 990, array of size 1000); we found this range optimal for the response time of our app and for the accuracy of the graphs plotted using these data.

```
for n in range(10,1000,10):
    rand_array=[0]*n
    randomgenerator(rand_array,n)
```

Next the functions calls the sorting algorithm functions previously explained and stores the step counter of each array sorted into a new array that holds the steps (taken to sort the arrays) in ascending order(first element is the steps taken to sort array of 10, next element steps taken to sort array of 20,...etc.), this array will be used as the y-axis data when drawing the curve of the sorting algorithm against its asymptotic notation or against another algorithm.

```
def main_insertion(a):
    times=[0]*99
    arr_n=[0]*99
    time_notation=[0]*99
    i=0

    for n in range(10,1000,10):
        rand_array=[0]*n
        randomgenerator(rand_array,n)

        count=0

        count=insertionSort(rand_array,count)

        times[i]=count
        i+=1
```

Moving on, we fill 2 extra arrays, the first one being the asymptotic upper bound ( $O(n)$ ) of the sorting algorithm in which will be used as second set of y-axis data case it is required to be shown against the running time steps(the asymptotic upper bound graph is scaled down to allow the appearance of both graphs near eachother for better comparisons), and the second array is the set of data to be used for the x-axis (n), which is automatically filled with the values form 10 to 1000 with step 10(the n's we used in the arrays we sorted). Furthermore, for easier debugging we print in cli the steps used in the sorting.

```

""" filling arr_n """
z=0
for i in range(10,1000,10):
    arr_n[z]=i
    z+=1
''' filling time_notation '''
if(a==0):
    w=0
    for w in range(len(arr_n)):
        time_notation[w]=pow(arr_n[w],2)

print("For Each Array Generated with different N sorted using Insertion Sort, Steps were:")
print(times)

```

After that, using the aid of python libraries numpy in mathematical operations and matplotlib.pyplot and scipy.interpolate in plotting the graphs we either plot the real time steps alongside the asymptotic upper bound ( $O(n)$ ) or plotted solely to be added to the final graph of the comparison alongside another sorting algorithms. This choice is made using a variable *a* that is sent to this function by the GUI buttons depends on whether it was clicked to be shown comparatively or against its own asymptotic upper bound. The data set of the plotting graph is filled with the arrays just mentioned above.

```

# Dataset
x = np.array(arr_n)
y = np.array(times)
if(a==0):
    x1 = np.array(arr_n)
    y1 = np.array(time_notation)
    X_Y_Spline1 = make_interp_spline(x1, y1)
    x_1 = np.linspace(x1.min(), x1.max(), 500)
    y_1 = X_Y_Spline1(x_1)
    plt.plot(x_1,y_1,label='N^2')
    plt.title("Insertion Sort")
    plt.legend()

X_Y_Spline = make_interp_spline(x, y)

# Returns evenly spaced numbers
# over a specified interval.
X_ = np.linspace(x.min(), x.max(), 500)
Y_ = X_Y_Spline(X_)
# Plotting the Graph
plt.plot(X_, Y_,label = 'Insertion Sort')
plt.xlabel("N")
plt.ylabel("Steps")

```

\* Please note that the screenshots used above are from the main\_insertion function form the insertion sort algorithm, however all other sorting algorithms have almost identical functions with the difference in the algorithm name and its asymptotic upper bound.

## GUI windows

After importing previous algorithm files, we create a window with appropriate size title and fonts

```

from tkinter import ttk
from tkinter import *
from tkinter import font
import tkinter as tk
from typing import Sized
from InsertionSort import main_insertion
from merge_code import main_merge, merge
from HeapSort import main_heap
from SelectionSort import main_selection
from Counting_sort import main_counting
from radix_sort import main_radix
from quickSort import main_quick
from Bubble_sort import main_bubble
import matplotlib.pyplot as plt
import tkinter.font as tkFont

window=Tk()
window.title("Sorting Application")
window.iconbitmap("university.ico")
window.geometry("700x700")
window.config(bg='#2E2E2E')
fontStyle = tkFont.Font(family="Poor Richard", size=30)
fontStyle1 = tkFont.Font(family="Poor Richard", size=25)
fontbutton=tkFont.Font(family="Poor Richard", size=15)
header_frame=Frame(window).pack(padx=200)
header=Label(header_frame,text="SORTING ALGORITHM APP",bg='#2E2E2E',pady=10,fg="#1A5BF3",font=fontStyle).pack(fill=X)

```

Then we define 8 functions each with the name of a sorting algorithm, these functions will call the main\_sort algorithm corresponding to its sorting algorithm but will not show the plot unless the variable a (previously mentioned in main\_sort section to determine whether it was clicked to be shown comparatively or against its own asymptotic upper bound) is equal to zero which is as we recall means to plot the graph against its own asymptotic upper bound. otherwise the plot will be on hold to be used in comparison

```

""" functions """
def insertion(a):
    main_insertion(a)
    if(a==0):
        plt.show()
def Merge(a):
    main_merge(a)
    if(a==0):
        plt.show()
def Heap(a):
    main_heap(a)
    if(a==0):
        plt.show()
def selection(a):
    main_selection(a)
    if(a==0):
        plt.show()
def counting(a):
    main_counting(a)
    if(a==0):
        plt.show()
def radixSort(a):
    main_radix(a)
    if(a==0):
        plt.show()
def quick(a):
    main_quick(a)
    if(a==0):
        plt.show()
def bubble(a):
    main_bubble(a)
    if(a==0):
        plt.show()

```

After that we add buttons to the window each with a title of a sorting algorithm of if pressed will call the function of the sort that was defined right before add send variable a with 0. an extra button with the title compare is added to open a new window to let the user chose which algorithms to compare



```

""" buttons """
body=Frame(window).pack(padx=10,pady=10)
choose=Label(body,text="PLEASE CHOOSE AN ALGORITHM",font=fontStyle1,fg="#1C49B4",bg="#2E2E2E").pack()
insertion_button=Button(body,text="Insertion Sort",bg="#622D7E",font=fontbutton,pady=10,command=lambda : insertion(),relief=RAISED).pack(fill=X)
Mergesort_button=Button(body,text="Merge Sort",bg="#79379C",font=fontbutton,pady=10,command=lambda : Merge(),relief=RAISED).pack(fill=X)
Heapsort_button=Button(body,text="Heap Sort",bg="#8A4480",font=fontbutton,pady=10,command=lambda : Heap(),relief=RAISED).pack(fill=X)
selection_button=Button(body,text="Selection Sort",bg="#9B4FC3",font=fontbutton,pady=10,command=lambda : selection(),relief=RAISED).pack(fill=X)
Counting_button=Button(body,text="Counting Sort",bg="#A75DCE",font=fontbutton,pady=10,relief=RAISED,command=lambda : counting()).pack(fill=X)
Radix_button=Button(body,text="Radix Sort",bg="#AE68D3",font=fontbutton,pady=10,relief=RAISED,command=lambda : radixSort()).pack(fill=X)
Quick_button=Button(body,text="Quick Sort",bg="#BA7ADB",font=fontbutton,pady=10,relief=RAISED,command=lambda : quick()).pack(fill=X)
Bubble_button=Button(body,text="Bubble Sort",bg="#C388E1",font=fontbutton,pady=10,relief=RAISED,command=lambda : bubble()).pack(fill=X)

next_window =Button(
    body,text="CLICK TO CHOOSE MULTIPLE ALGORITHMS TO COMPARE ",
    bg="#DBAEF2",
    fg="black",
    font=fontbutton,
    pady=10,command=openNewWindow).pack(pady=20)

```

Then we create variables var1 through var8 to store whether the check box of each sort algorithm is checked. Then we create said check boxes with the algorithms names as titles and variables var1...8 as saving value to determine which algorithms to be compared to each other and then a button to show the plotted graph

#### 50 check boxes

```

var1 = tk.IntVar()
var2 = tk.IntVar()
var3 = tk.IntVar()
var4 = tk.IntVar()
var5 = tk.IntVar()
var6 = tk.IntVar()
var7 = tk.IntVar()
var8 = tk.IntVar()
c1 = tk.Checkbutton(newWindow, text='Insertion Sort',variable=var1, onvalue=1, offvalue=0 ,font=fontbutton,bg="#C388E1").pack(pady=10)
c2 = tk.Checkbutton(newWindow, text='Merge Sort',variable=var2, onvalue=1, offvalue=0,font=fontbutton,bg="#C388E1").pack(pady=10)

c3 = tk.Checkbutton(newWindow, text='Heap Sort',variable=var3, onvalue=1, offvalue=0,font=fontbutton,bg="#C388E1").pack(pady=10)
c4 = tk.Checkbutton(newWindow, text='Selection Sort',variable=var4, onvalue=1, offvalue=0,font=fontbutton,bg="#C388E1").pack(pady=10)

c5 = tk.Checkbutton(newWindow, text='Counting Sort',variable=var5, onvalue=1, offvalue=0,font=fontbutton,bg="#C388E1").pack(pady=10)
c6 = tk.Checkbutton(newWindow, text='Radix sort',variable=var6, onvalue=1, offvalue=0,font=fontbutton,bg="#C388E1").pack(pady=10)

c7 = tk.Checkbutton(newWindow, text='Quick Sort',variable=var7, onvalue=1, offvalue=0,font=fontbutton,bg="#C388E1").pack(pady=10)
c8 = tk.Checkbutton(newWindow, text='Bubble Sort',variable=var8, onvalue=1, offvalue=0,font=fontbutton,bg="#C388E1").pack(pady=10)

Show_comparison=Button(newWindow,text="CLICK TO SHOW COMPARISON !!",bg="#1C49B4",font=("arial",12),pady=30,relief=RAISED,command=compare).pack(fill=X,pady=50)

```

Finally the new window opened will inherit the properties of the previous one, then we define a function called compare that calls the functions of the algorithms if they were selected by the user (var=1) but sends the variable a =1 to plot the graphs comparatively. We only plot(plt.show) after all if conditions have been checked to show all required graphs

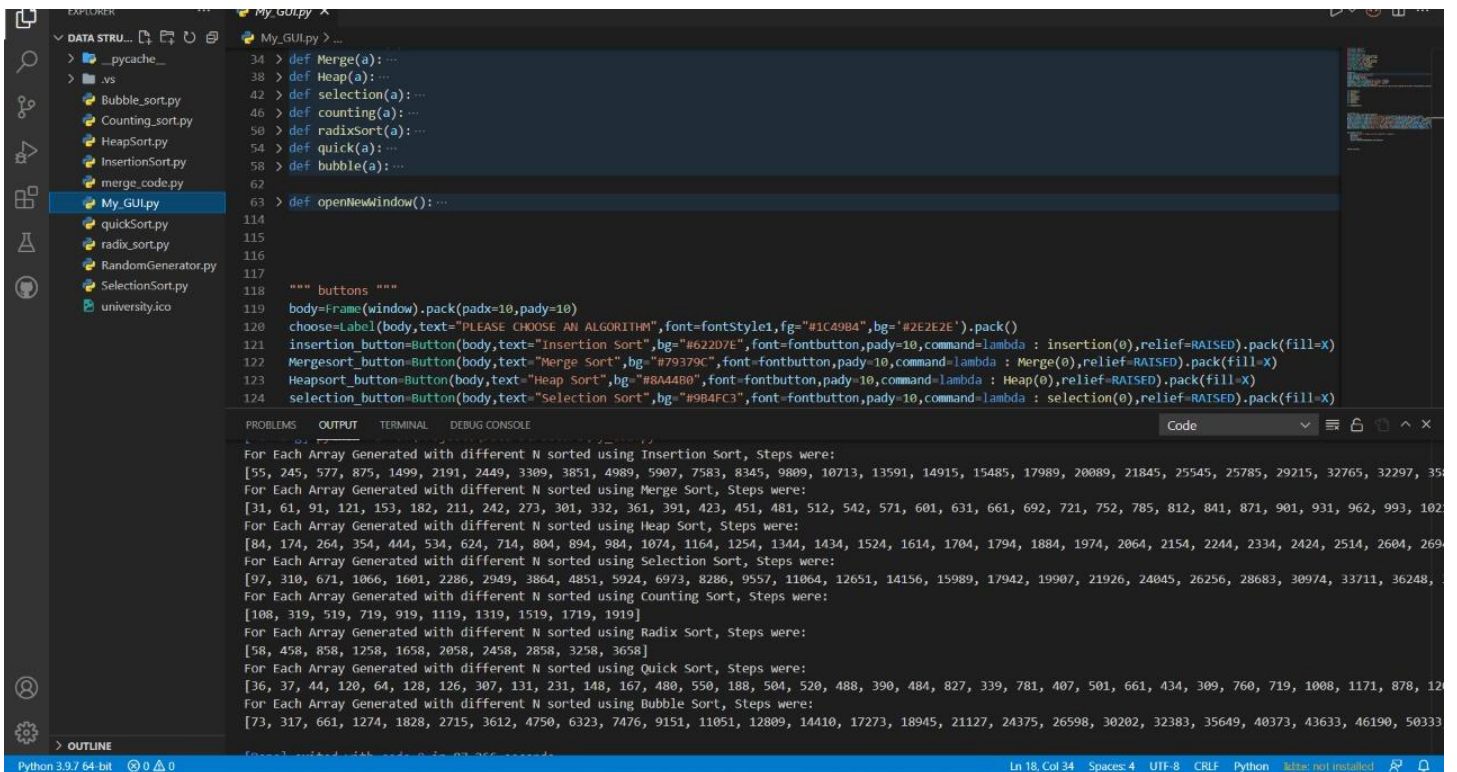
## 52 comparison graph

```
def openNewWindow():
    newWindow = Toplevel(window)
    newWindow.title("Project")
    newWindow.iconbitmap("university.ico")
    newWindow.geometry("750x750")
    newWindow.config(bg="#2E2E2E")
    fontStyle2 = tkFont.Font(family="Poor Richard", size=20)
    Label(newWindow, text = "CHOOSE ANY NUMBER OF ALGORITHMS TO COMPARE", fg="#1C49B4", bg='#2E2E2E', font=fontStyle2).pack()

def compare():
    fig = plt.figure()
    if(var1.get()==1):
        insertion(1)
    if(var2.get()==1):
        Merge(1)
    if(var3.get()==1):
        Heap(1)
    if(var4.get()==1):
        selection(1)
    if(var5.get()==1):
        counting(1)
    if(var6.get()==1):
        radixSort(1)
    if(var7.get()==1):
        quick(1)
    if(var8.get()==1):
        bubble(1)
    plt.legend()
    plt.show()
```

## Output Screenshots

### 54 step counter array

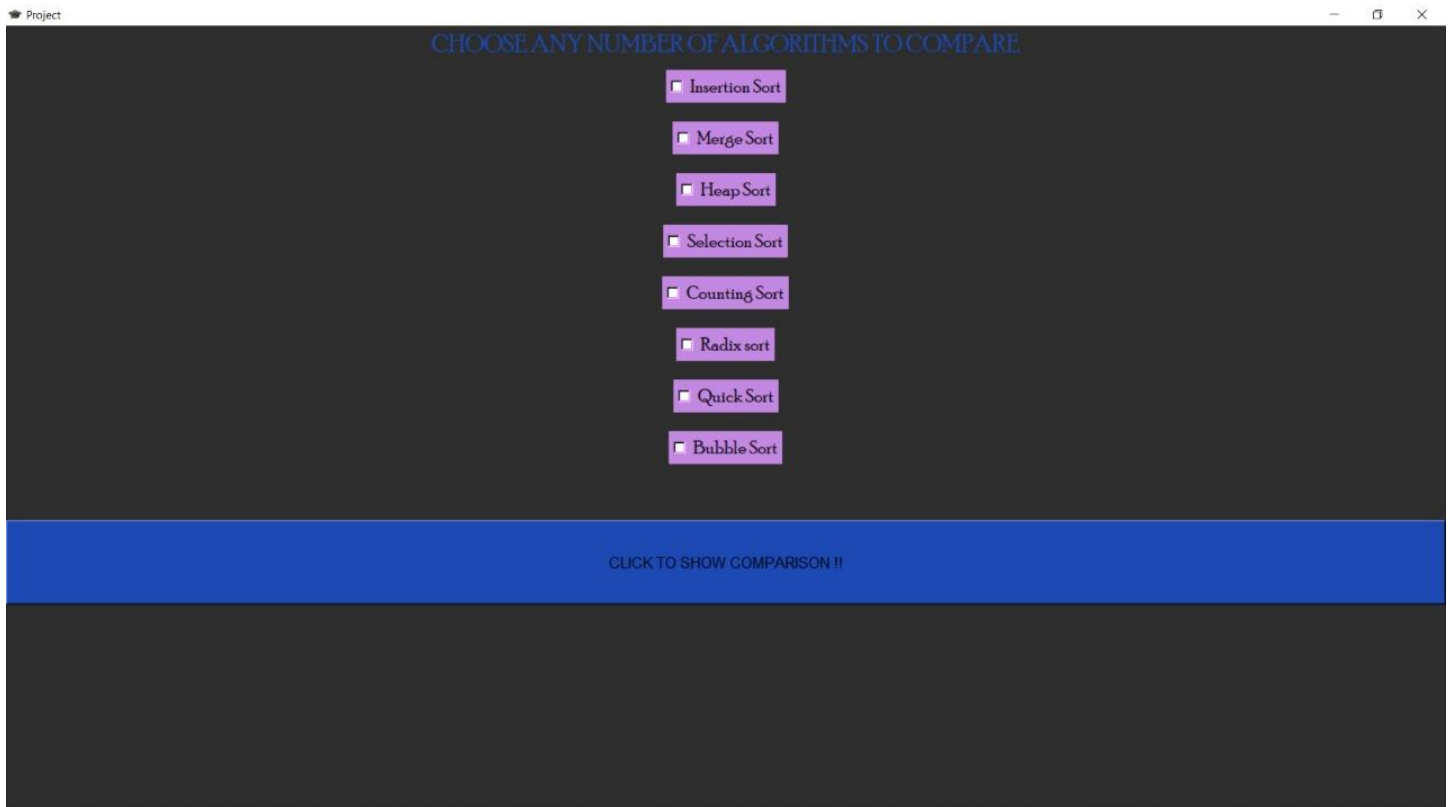
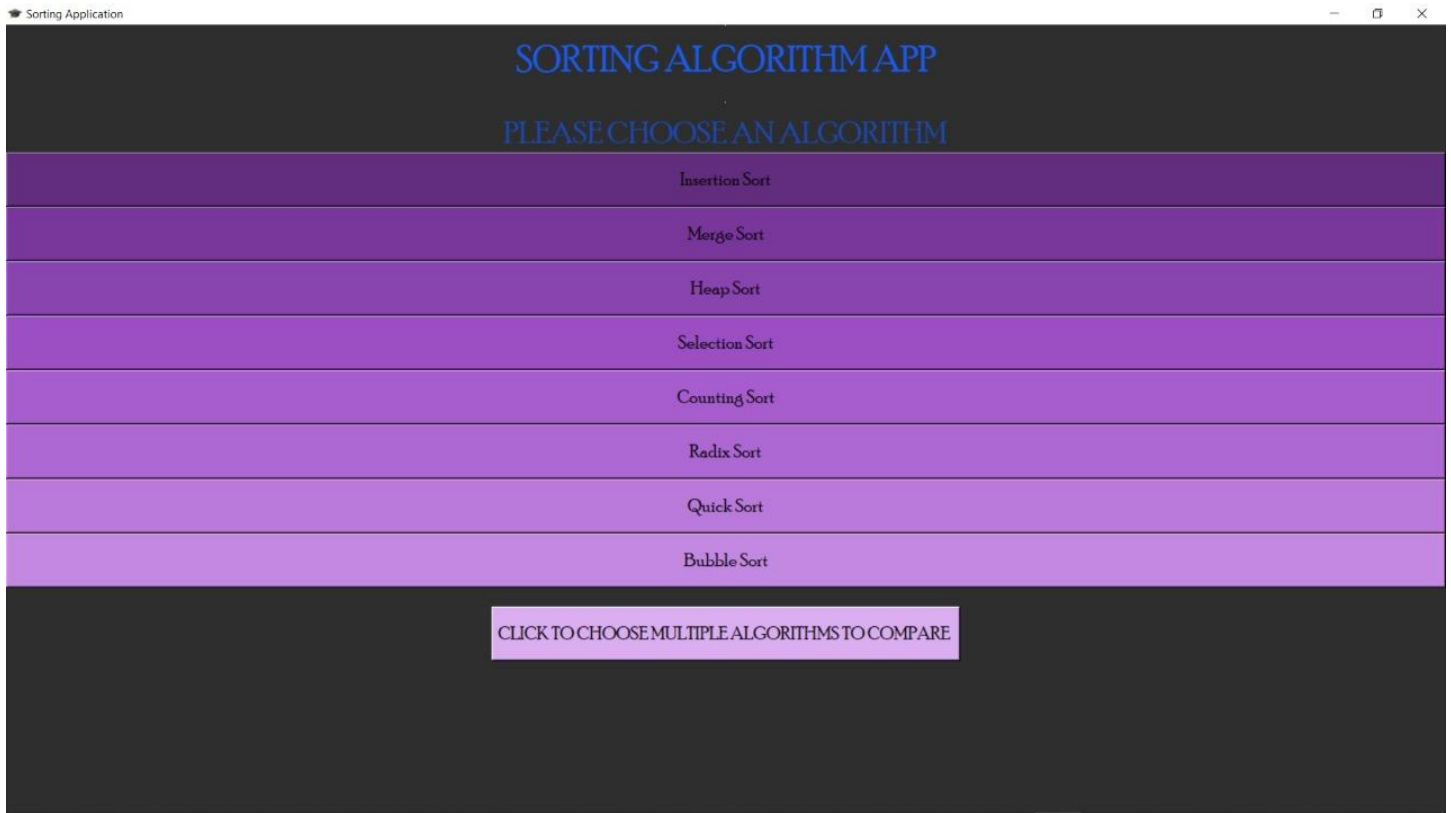


```
def Merge(a): ...
def Heap(a): ...
def selection(a): ...
def counting(a): ...
def radixSort(a): ...
def quick(a): ...
def bubble(a): ...

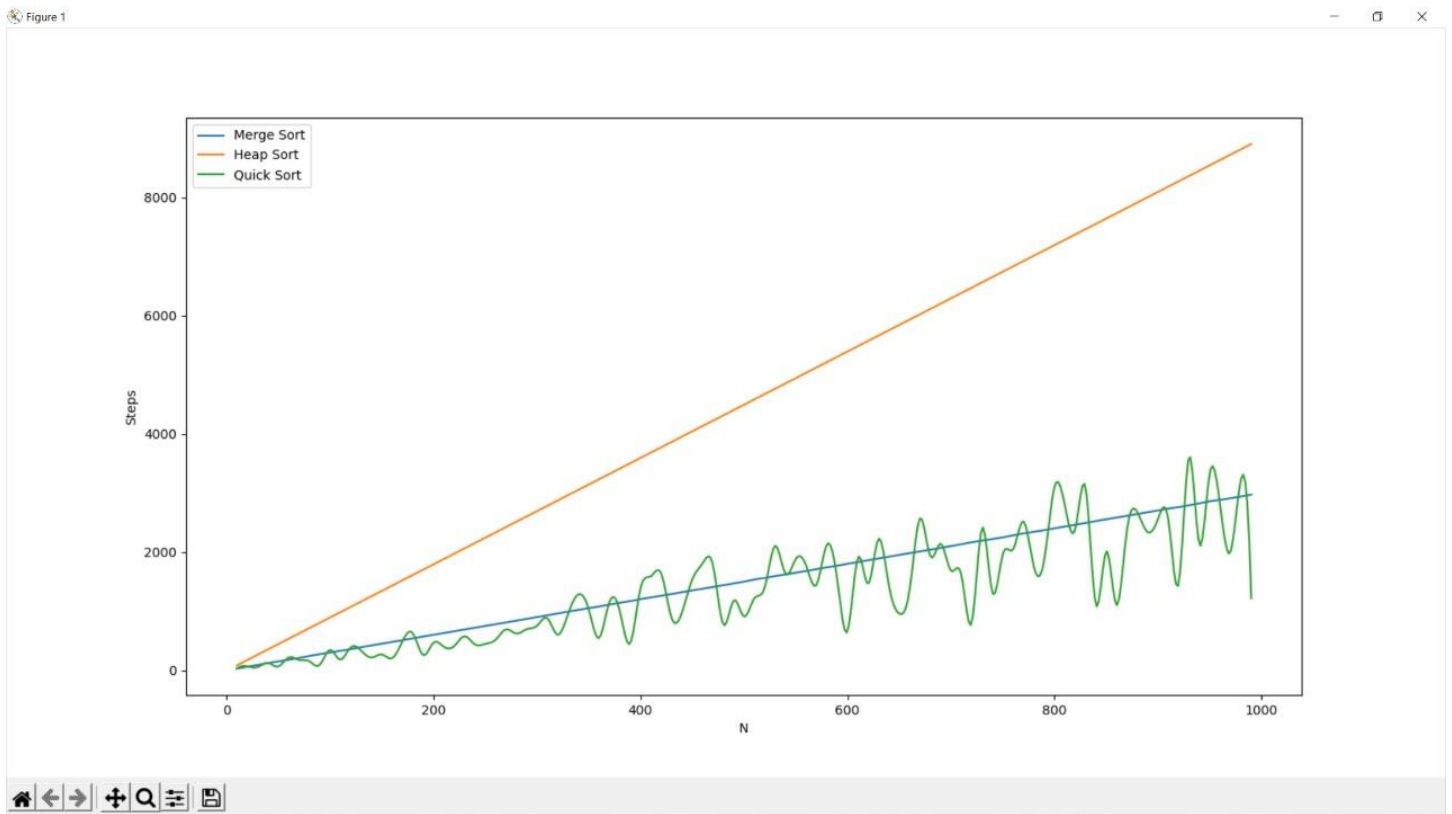
def openNewWindow(): ...
    choose=Label(body, text="PLEASE CHOOSE AN ALGORITHM", font=fontStyle1, fg="#1C49B4", bg='#2E2E2E').pack()
    insertion_button=Button(body, text="Insertion Sort", bg="#62207E", font=fontbutton, pady=10, command=lambda : insertion(0), relief=RAISED).pack(fill=X)
    Mergesort_button=Button(body, text="Merge Sort", bg="#79379C", font=fontbutton, pady=10, command=lambda : Merge(0), relief=RAISED).pack(fill=X)
    Heapsort_button=Button(body, text="Heap Sort", bg="#8A4480", font=fontbutton, pady=10, command=lambda : Heap(0), relief=RAISED).pack(fill=X)
    selection_button=Button(body, text="Selection Sort", bg="#9B4FC3", font=fontbutton, pady=10, command=lambda : selection(0), relief=RAISED).pack(fill=X)

    """ buttons """
    body=Frame(window).pack(padx=10, pady=10)

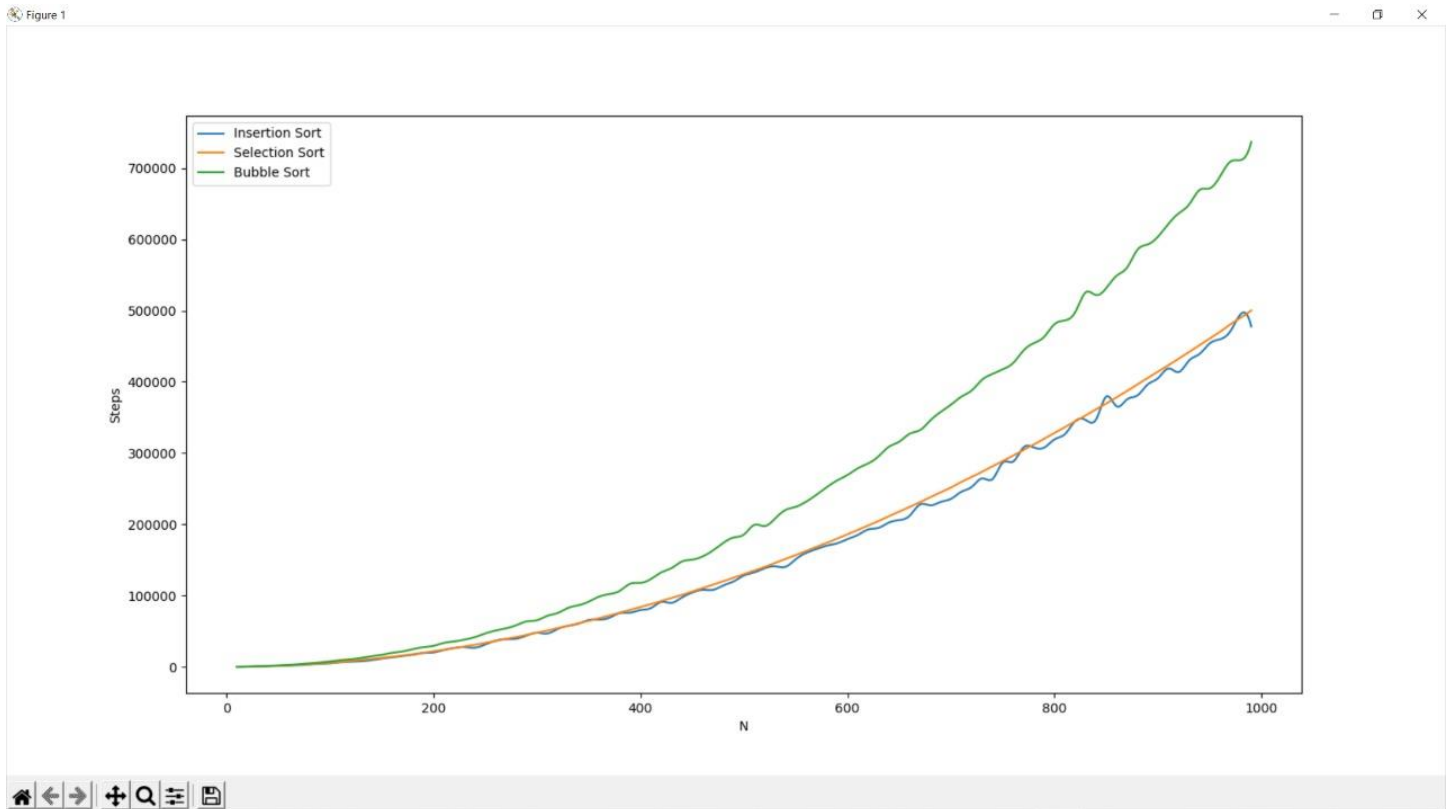
    For Each Array Generated with different N sorted using Insertion Sort, Steps were:
    [55, 245, 577, 875, 1499, 2191, 2449, 3309, 3851, 4989, 5907, 7583, 8345, 9809, 10713, 13591, 14915, 15485, 17989, 20089, 21845, 25545, 25785, 29215, 32765, 32297, 35...
```



### 57 $n^2$ algorithms comparison



### 58 $n \log n$ algorithms comparison



59 all algorithms comparison

