Faculty of Engineering – Ain Shams University

Computer Engineering and Software Systems

CSE331: Data Structure and Algorithms

Fall 2021

Submitted to

Dr. Hesham Farag

Submitted by

Aya Ahmed Mohamed Gamal 19P1689

Alaa Mohamed Galal Eldin 19P4206

Shaimaa Mohamed Ahmed Hassan 19P7484

Ziad Mohamed Nasr Mohamed 19P2061

Table of Contents

Introduction

First, we have used python 3 as a programming language. Our main goal is to plot the graph of steps of many important sorting algorithms like insertion sort, merge sort, heap sort, counting sort, quick Sort, bubble and radix Sort. Moreover, we will show the notation of each algorithm, compare between steps of these algorithms to know which is quicker and efficient. By making such comparison, the uses and advantages of each algorithm will be clearly displayed.

Code Illustration

- In order to plot the graph of steps we will generate 10,000 random numbers ranging from 1 to 1000, this is successfully done throughout our code using function called Generate_RandomArray() and these random numbers is appended in large array called rand_array[] , after that we started writing this random numbers in a files to facilitate using them in our application , the function writeInfile() used to write in a file called random1.txt.
- Then we have function to use the data in the file called readFile() this function opens the file of random numbers created in read mode , so this function will be used at the start of each algorithm to use the random numbers and fulfill our target to calculate and plot the steps taken by each algorithm.
- So, we make a separate file containing the code of each algorithm, for example in insertion sort file we will have the random generator function, readFile() function to get the numbers and operate on them . Then we have the function performing the insertion sort itself, a variable counter is created to be able to increase the number of steps taken based on the number of instructions in the algorithm to make sorting.
- Inside each Algorithm, there is a function called get_steps() : this function reads data from file and start to take values from file and put in variable called temp=readFile(), then start filling the numbers taken from file in a small to be varying in size called arr_n by the numbers starting by n=10, then increasing by 50 by the end of each iteration.
- The numbers of n will get appended by this statement x.append(n) so it will be plotted in the x-axis and we supposed that our end will be limited to 3000 , another variable called count will take the return value from the sorting algorithm (counter) then start appending this returned count on the y-axis, that is the way the steps will be plotted.

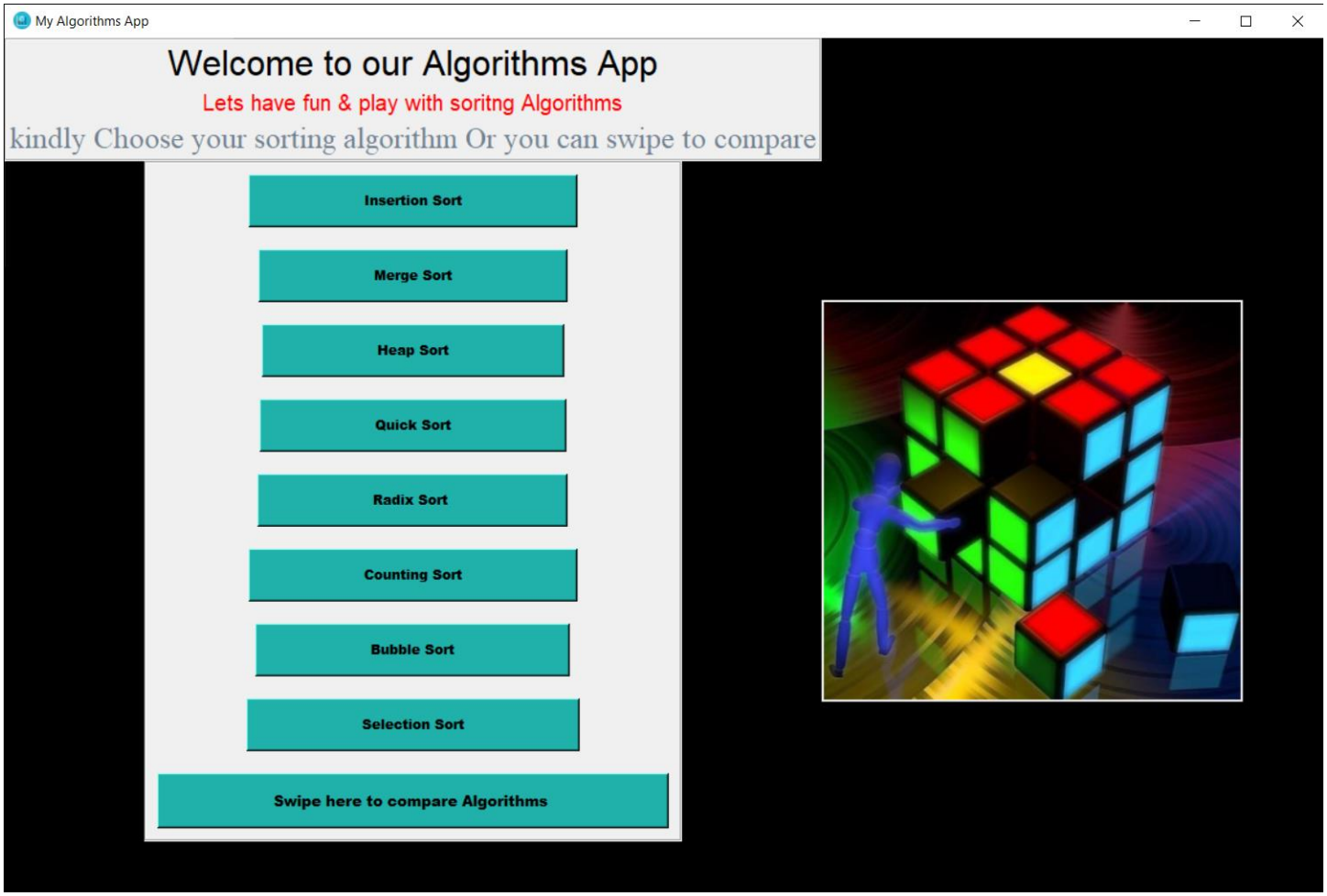
- The notation will take also values on the x and y-axis filling the n from a for loop that start increasing by 50 like we said before and y-axis will hold the best case for each algorithm like insertion sort it will be O(n^2), mergeSort O (n log n).
- In the code there will be three more functions, function responsible for getting notation and function responsible for plotting the steps called plot_steps(), another function responsible for plotting steps versus the notation for each algorithm by calling get_steps() function & get_notation() function .
- And one more function added called compare to be used when comparison is needed.

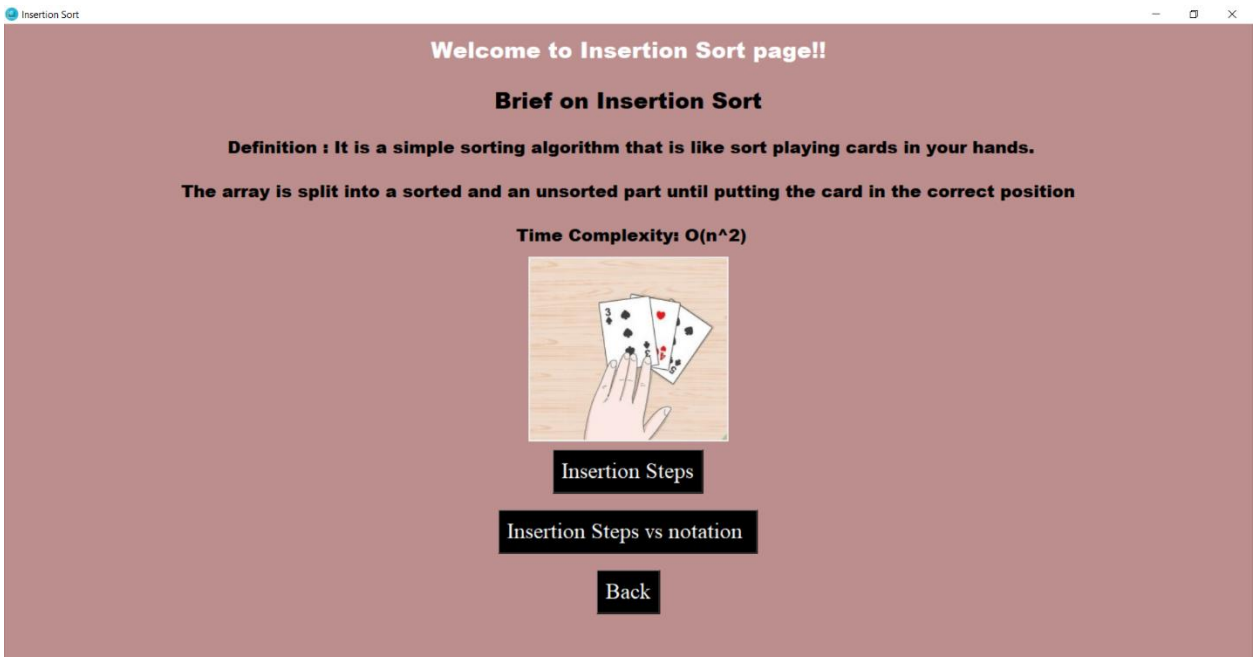
Functions readFile () & get_steps () code in insertion Sort

```
def readFile():
    fileObj = open("random1.txt", "r")  # opens the file in read mode
    words = fileObj.read().splitlines()  # puts the file into an array
fileObj.close()
    return words
def get_steps():
    temp=readFile()
    n=10
    for z in range (0,10000):
        if n >=3000:
            break
        for i in range (0,n):
            s = temp[i]
            arr_n.append(s)
        count=insertionSort(arr_n)
        name='InsertionSort'
        x.append(n)
        y.append(count)
        n+=50
        arr_n.clear()
    return x,y
```

```
def get_insertion_notation():
    n=10
    for z in range (0,10000):
        if n >=3000:
            break
        x1_time=n
        y1_time=pow(n, 2)
        name="InsertionSort"
        x1.append(x1_time)
        y1.append(y1_time)
        n+=50
        arr_n.clear()
    return x1,y1
def plot_insertion_steps_notation() :
    x,y = get_steps()
    x_time,y_time =get_insertion_notation()
    plt.plot(x,y,label='Insertion steps ')
    plt.plot(x_time, y_time , label='Insertion Notation O(n^2)')
    plt.title('InsertionSort')
    plt.xlabel("n")
    plt.ylabel("f(n)")
    plt.legend()
    plt.show()
```

Main Application GUI





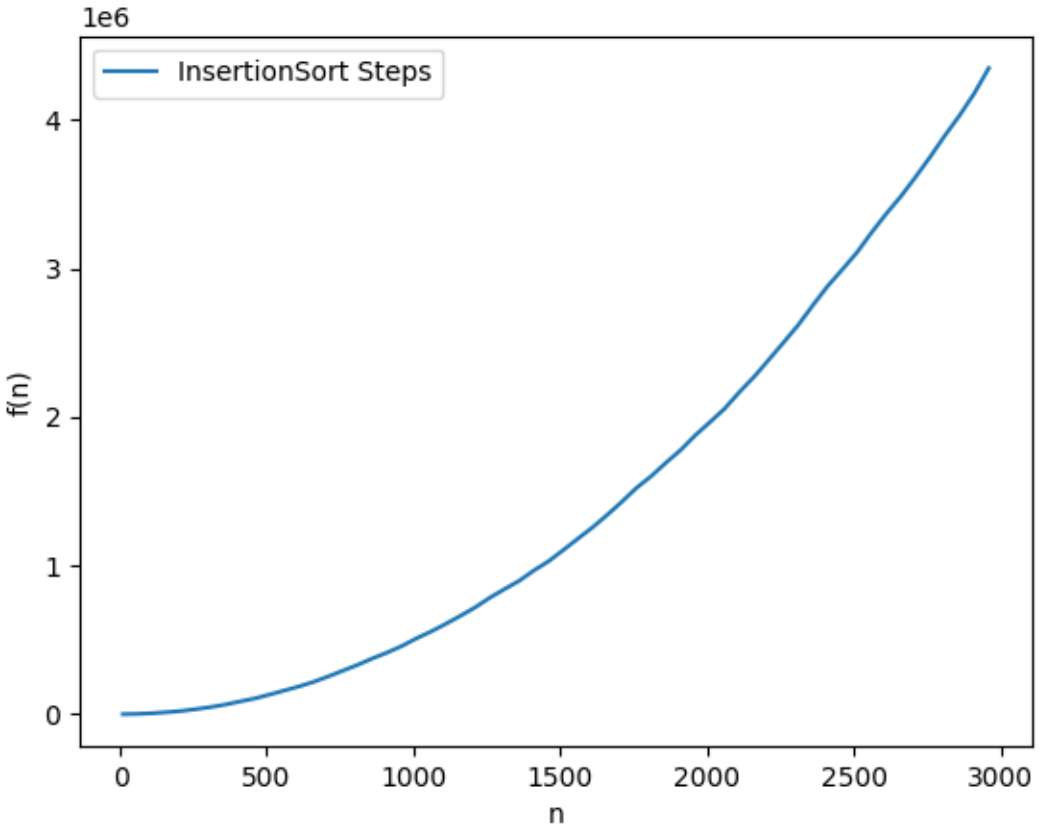
1. Insertion Sort

Definition: It is a simple sorting algorithm that works like the way you sort playing cards in your hands. The array is split into a sorted and an unsorted part. Values from the unsorted part are picked and places at the correct position of the sorted part.

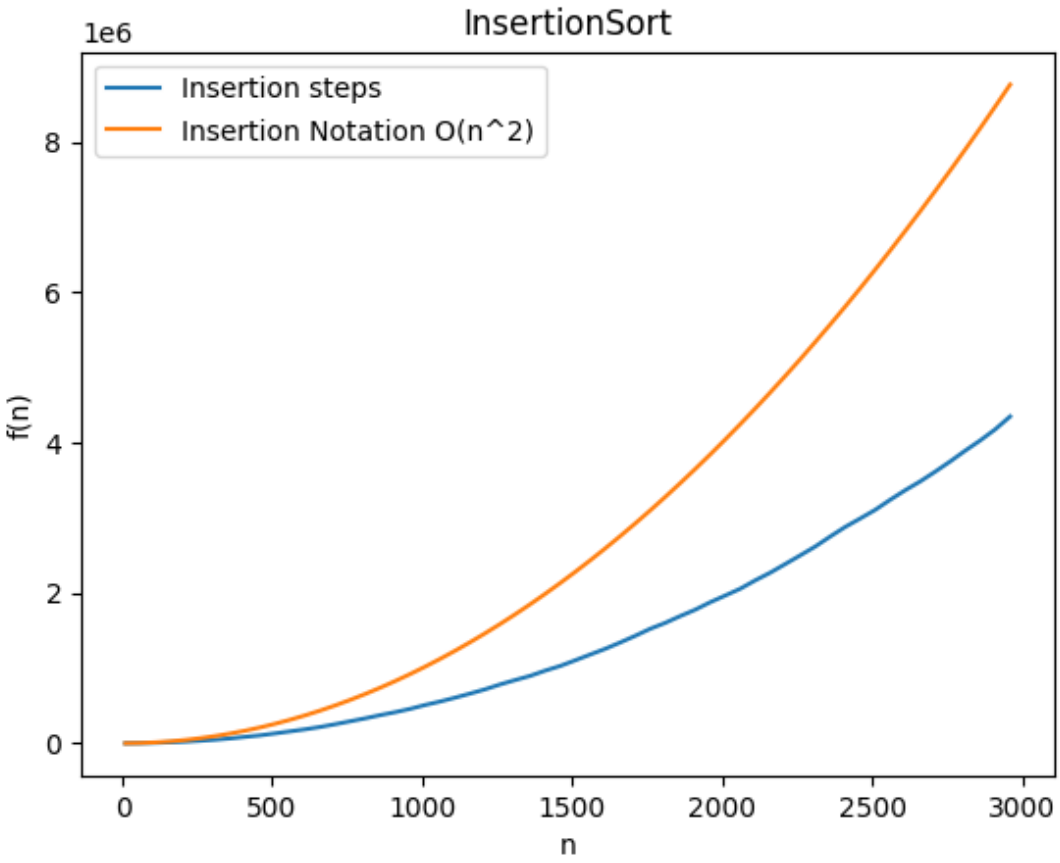
Algorithm: To sort an array of size n in ascending order:

1) Iterate from the second element in array till last one.
2) Compare the key element to its predecessor.
3) If the key element is smaller than its predecessor, compare it to the elements before, then move the greater elements one position up to make space for the swapped element.

Time Complexity: O (n) for best and average cases but O (n^2) for worst case.



Uses: When the array elements are small or when the elements are almost sorted, so only few elements will not be at their correct place in complete big array.

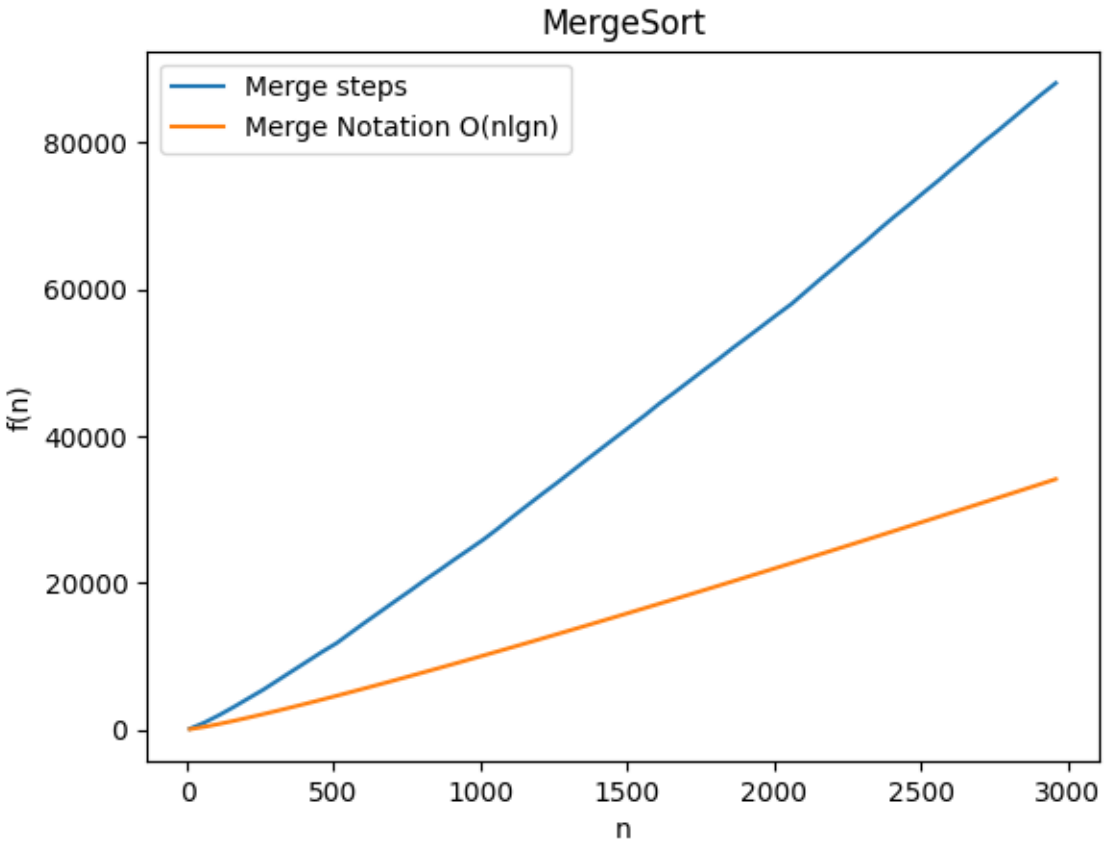


2. Merge Sort:

Definition: It is a Divide and Conquer algorithm that divides the input array into two halves, then it calls itself for the two halves to recursively sort them, and then merges the two sorted halves in the combine step using the merge function.

Time Complexity: O (n log n) for all cases.

Uses: It is an efficient, stable sorting algorithm in dealing with large data, when we want to sort in running time O (n log n) which is much better than and faster than many average and worst case running time of sorting algorithms.


MergeSort
Merge steps
Merge Notation O(nlgn)
80000
60000
40000
20000
0
f(n)
0
500
1000
1500
2000
2500
3000
n

3. Heap Sort

Definition: It is a comparison-based sorting technique based on Binary Heap. We first find the minimum element and place the minimum element at the beginning. We repeat the same process for the remaining elements.

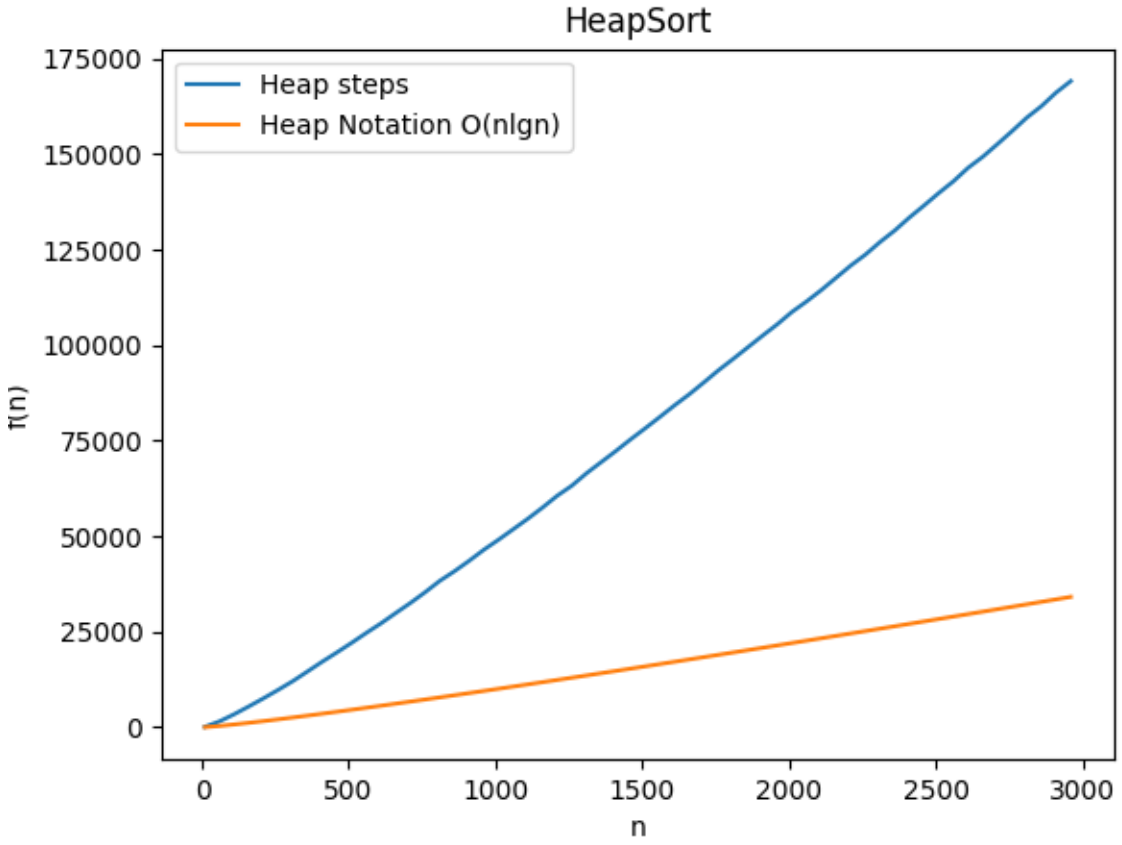
Binary Heap: It is a complete binary tree in which things are stored in such a way that the value in a parent node is more (or less) than the values in its two child nodes. The former is known as max heap, whereas the latter is known as min-heap.

Algorithm:

1) Use the input data to build max heap.
2) The largest item is stored at the root of the heap. Then, replace it with the last item of the heap followed by reducing the size of heap by 1. Finally, heapify the root of the tree.
3) Repeat step 2 while the size of the heap is greater than 1.

Time Complexity: O (n*log(n)) for all cases.

Uses: Heap sort is needed when the highest or the smallest element is needed instantly, also in finding the order in statistics and is efficient in dealing with priority queues in Prim's algorithm and useful in data compression.


HeapSort
Heap steps
Heap Notation O(nlgn)
175000
150000
125000
100000
75000
50000
25000
0
f(n)
0
500
1000
1500
2000
2500
3000
n

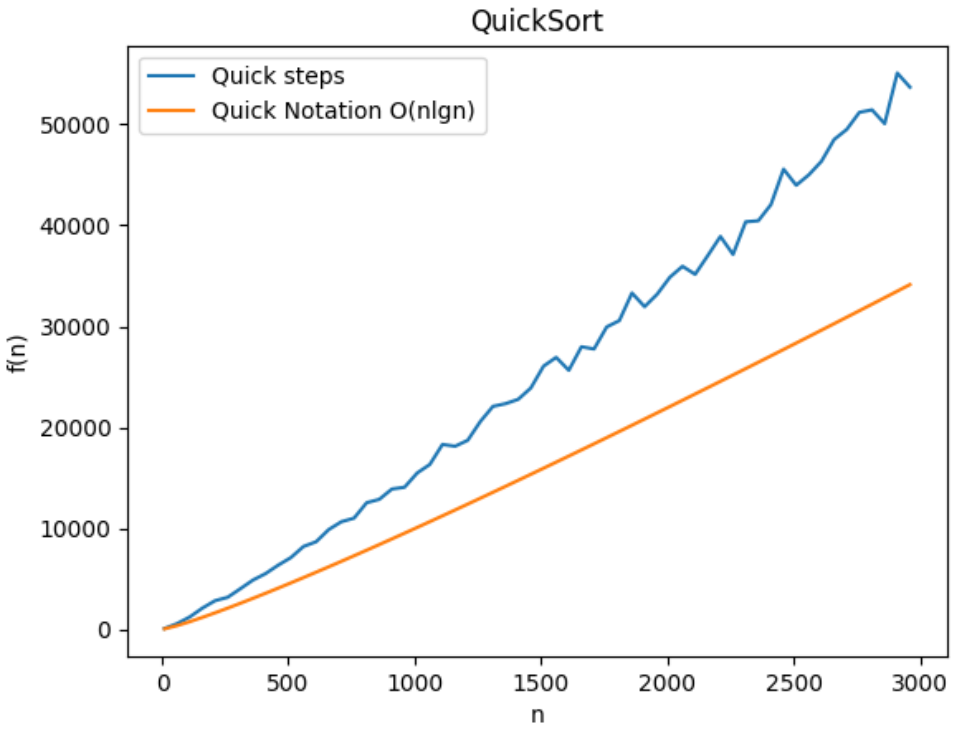
4. Quick Sort

Definition: It is a Divide and Conquer algorithm. It picks an element as pivot and partition the given array around the picked pivot. Then, we put the elements around the pivot correctly to make left of pivot less than it and right side greater than it, until pivot becomes in correct place, then we start choosing another pivot until all elements become sorted.

Partition Algorithm: Partitioning can be done in a variety of ways. The concept is straightforward: we begin with the leftmost element and keep track of the index of smaller (or equal to) items as the index of the smallest element. If we come across a smaller element while traversing, we exchange the current element with arr[i]. Otherwise, we ignore the current element.

Time Complexity: O(n*log(n)) for best and average cases but O(n^2) for worst case.

Uses: It is mainly used in searching for information, and it can be considered from fastest algorithms so, it is the better one in searching, it is a cache-friendly algorithm with good locality as it used for arrays, it doesn't require extra space as sorting is done in-place, moreover it is used in commercial computing that is badly needed in governmental organizations to sort various data.



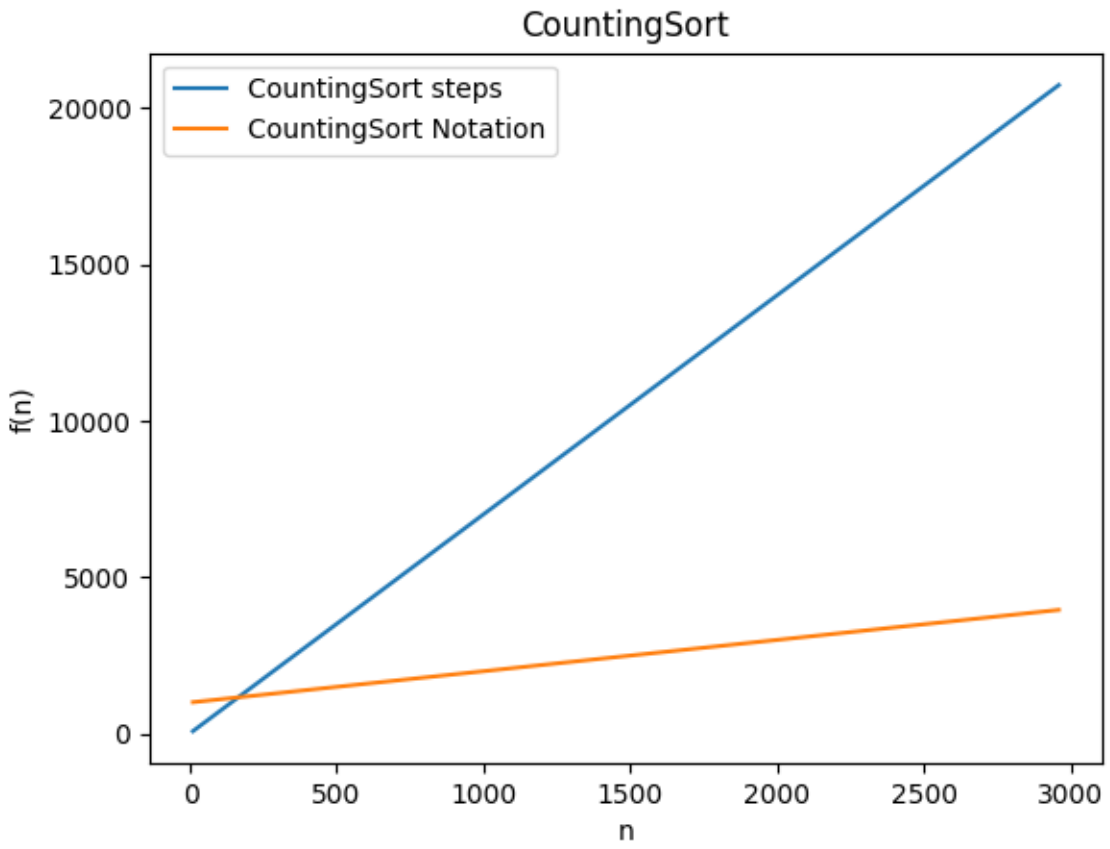
5. Counting Sort

Definition: It is based on keys that fall inside a certain range. It operates by determining the number of items with unique key values (kind of hashing). The position of each object in the output sequence is then calculated using arithmetic.

Time Complexity: O (n+k) for all cases, where k is the base for representing numbers (d digits in input integers).

Uses:

- It is used when k is not larger than n that means the range of input elements isn't bigger than the numbers going to be sorted, so it needs small amount of numbers with multiple counts and needed if we want to sort in linear time.
- It is often used as a sub-routine to another sorting algorithm like radix sort.
- Counting sort uses partial hashing to count the occurrences of the data item in O (1).



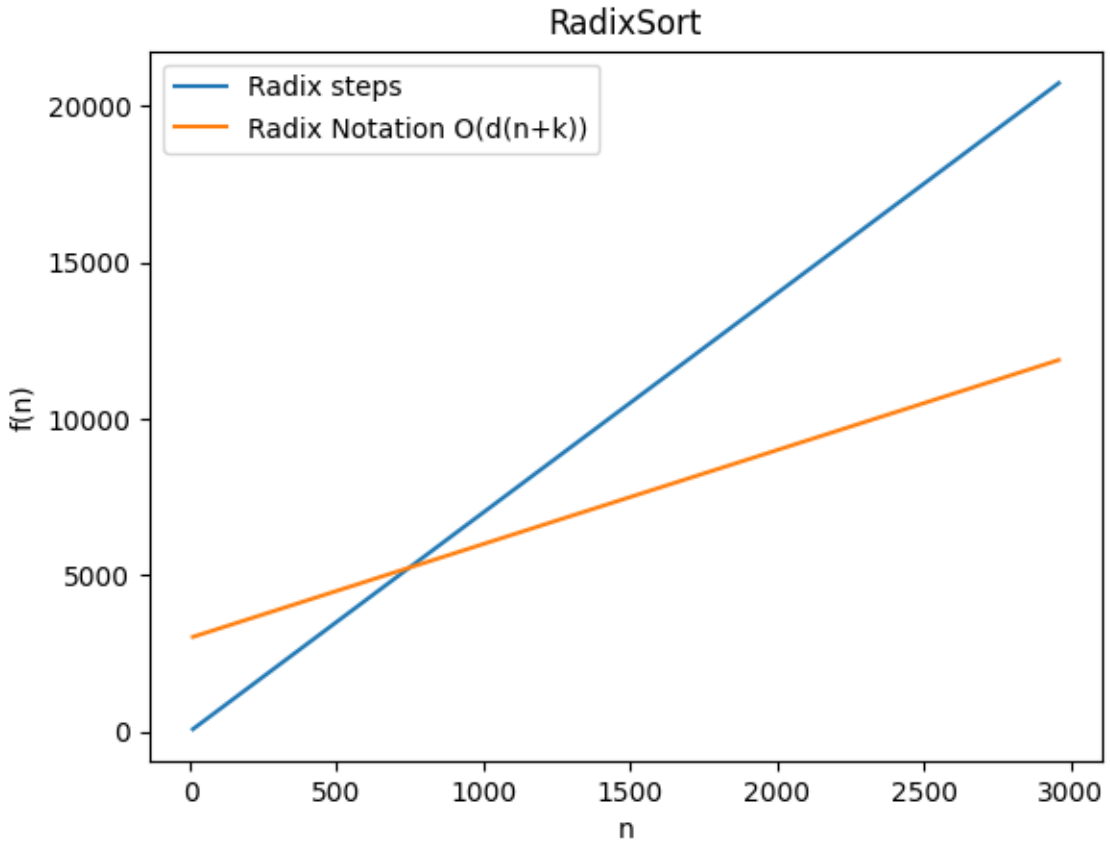
6. Radix Sort

Definition: It is based on doing digit by digit sort starting from least significant digit to most significant digit. Radix sort uses counting sort as a subroutine to sort.

Time Complexity: O (d(n+k)) for all cases, where k is the base for representing numbers (d digits in input integers).

Uses:

- It can lexicographically sort a variety of data types, including integers, words, and emails, but it is most commonly used to sort collections of integers and strings (that are mapped to appropriate integer keys).
- It is employed in a normal computer, which is a sequential random-access machine with records keyed by many fields. For example, suppose you wish to sort by month, day, and year. You might compare two records based on year, then month, and finally date. Alternatively, the data might be sorted three times using Radix sort, first on the date, then on the month, and finally on the year
- It was also employed in card sorting machines with 80 columns, and the machine could only punch a hole in 12 locations in each column. The sorter was then configured to sort the cards in the order in which they were punched. The operator then utilized this to collect the cards with the first row punched, then the second row, and so on.


RadixSort
Radix steps
Radix Notation O(d(n+k))
f(n)
n
0
5000
10000
15000
20000
0
500
1000
1500
2000
2500
3000

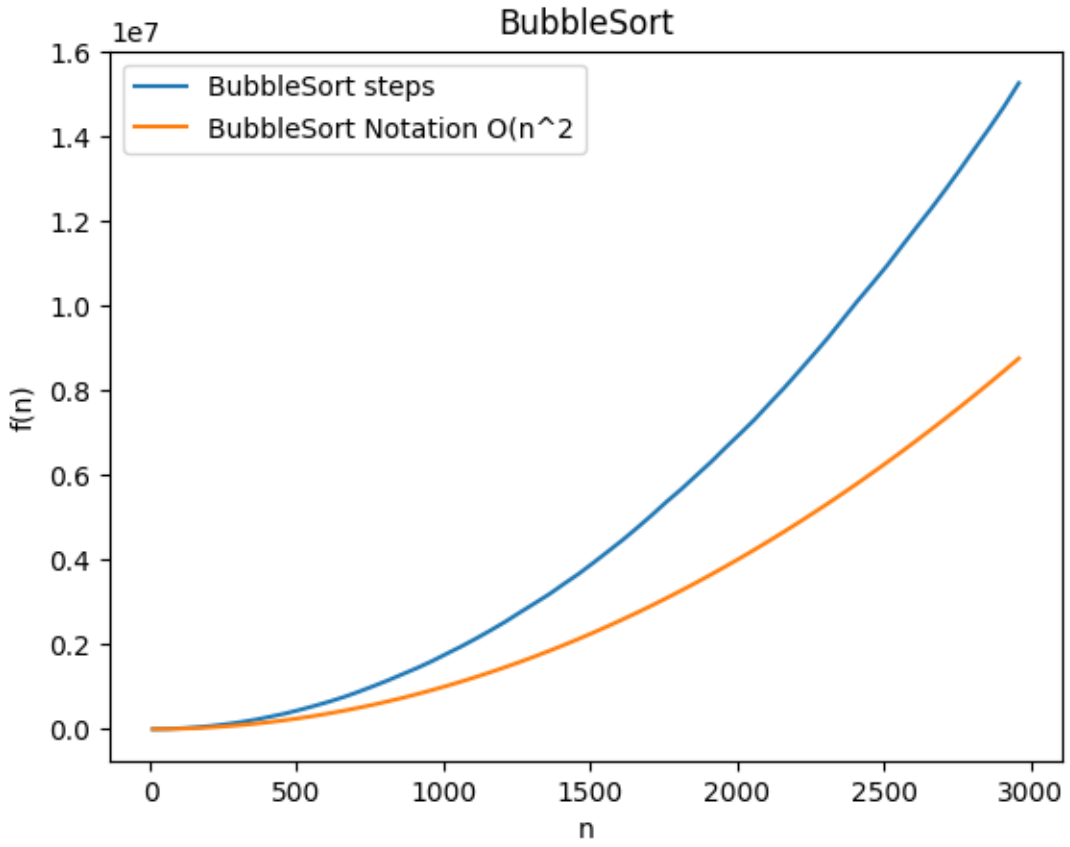
7. Bubble Sort

Definition: It is the simplest sorting algorithm that works by repeatedly swapping the adjacent elements if they are in wrong order so, it depends if we want to sort them ascending so, we will make a minimum value and continue updating it making swaps to reach that all numbers are sorted.

Time Complexity: O (n) for best case and O (n^2) for average and worst cases.

Uses:

- Bubble sort is mostly used in educational settings to assist students in grasping the fundamentals of sorting. This is used to see if the list has previously been sorted. If the list has already been sorted (which is the best-case scenario), its best case complexity is O(n).
- It is popular in computer graphics for its capability to detect very small errors in almost-sorted arrays and fix it.

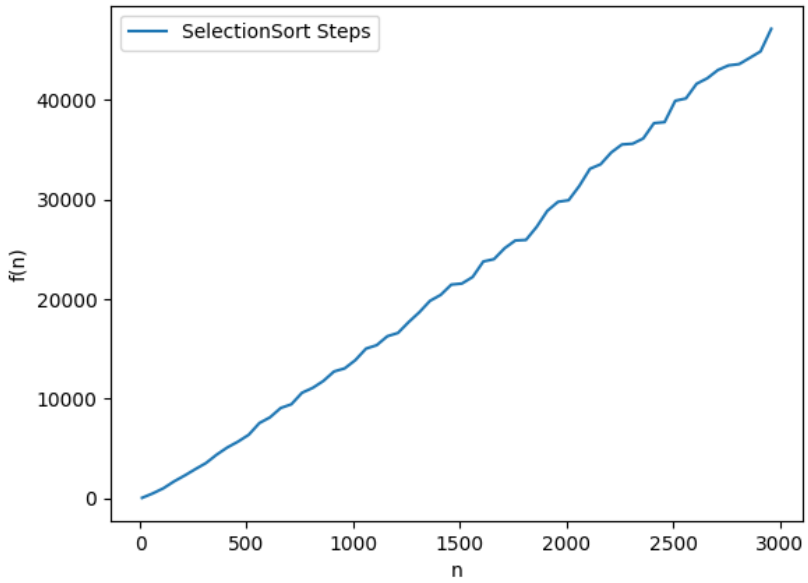


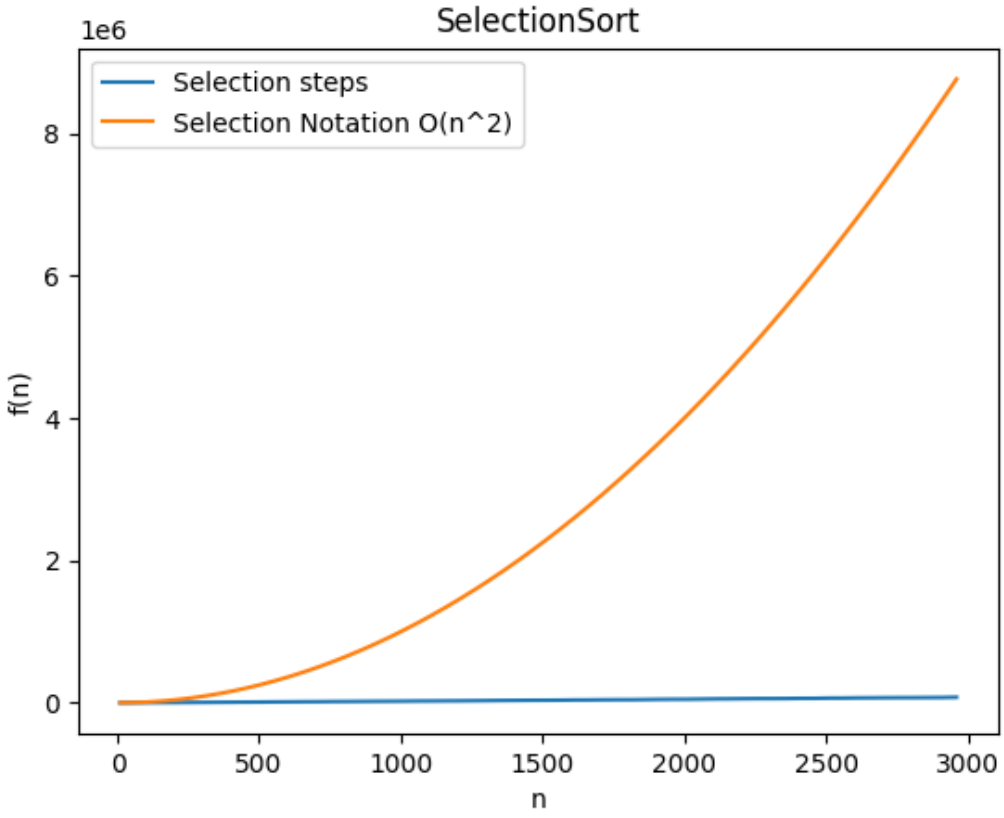
8. Selection Sort

Definition: It is a simple algorithm, as you set the first element as minimum, then it compares the first and second elements. Assign the second element to the position of minimum if it is smaller than the first. Then Compare the third element to the minimum. If the third element is smaller, assign the minimum value to it; otherwise, do nothing. The process continues until the final ingredient is added. Minimum is moved to the front of the unsorted list after each iteration until all elements become sorted.

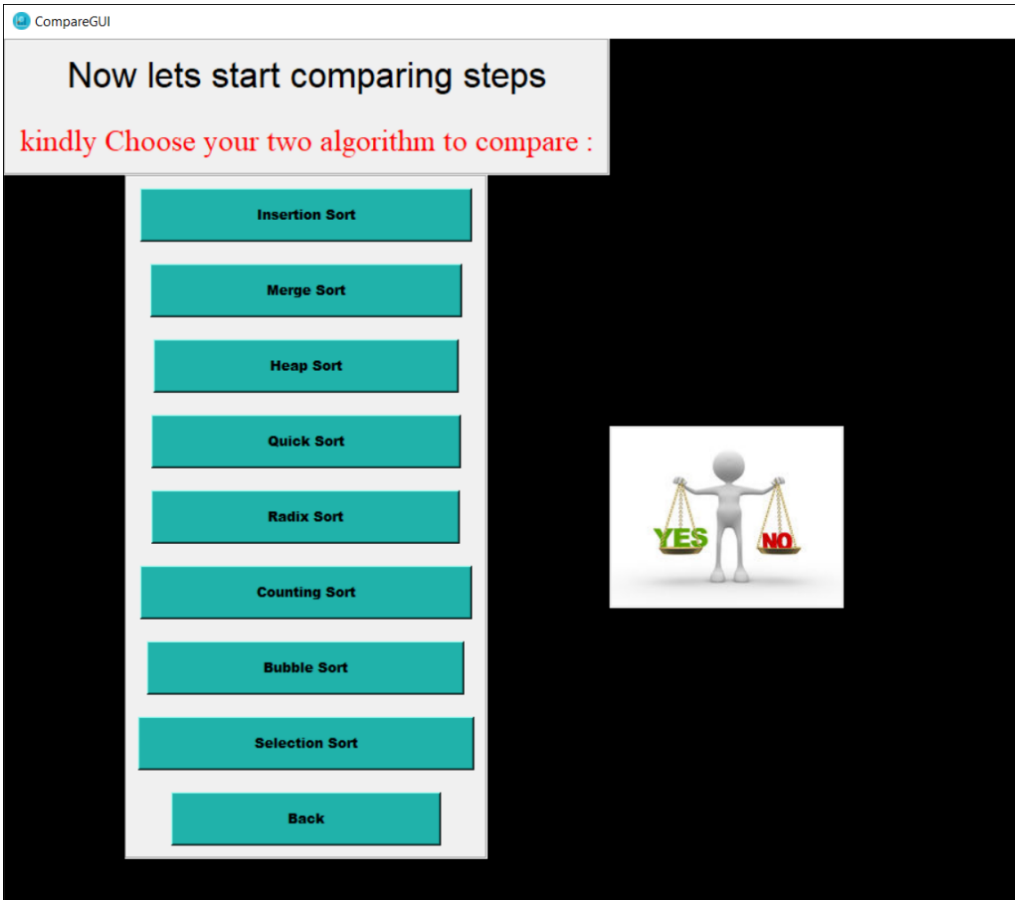
Time Complexity: O (n^2) for all cases.

Uses: When sorting a small list, the selection sort is employed, when the expense of exchanging is irrelevant. It also used when Checking all components is required, and the cost of writing to memory matters, as it does in flash memory (number of writes/swaps is O (n) versus O(n2) for bubble sort).




SelectionSort
1e6
Selection steps
Selection Notation O(n^2)
f(n)
0
2
4
6
8
0
500
1000
1500
2000
2500
3000
n

Compare page



Comparing Technique

- The user will have a page to choose two algorithms from some buttons to compare between them based on the number of steps taken by each algorithm.
- In our interface , you will start clicking first button you have chosen then it will take some time , then the button realized giving you the opportunity to choose another button to make comparison , after that one graph will be displayed emphasizing the steps of each algorithm. Code for compare is included in compare page which uses a counter to detect clicks then get increased by one to switch and choose another button to click, when counter reaches 2 so, now he have the 4 parameters needed for function compare to plot the graph including the selected algorithms, also function compare_two_algorithms ( ) placed in Compare page is called in each algorithm to bind the needed x and y axes then this function. And in GUI each button is set command to use for each algorithm to get x and y but wait until another one selected and give them to compare_two_algorithms ( ).

- Illustrating in more depth : for example In insertion_sort there is function called compare_insertion , It combines x, y from function that returns n and steps like this x,y=get_steps() then start calling compare_two_algorithms(x,y, name) , so, when the button of insertion is clicked it calls compare_insertion() , and continue binding , so one click is made waiting for another click for example : merge sot so, same steps & binding will be done and calls compare_two_algorithms () then ,when two buttons are selected after each other , it plots the graph of comparison between insertion and merge steps.

Code of compare Page:

```
counter=0
x_axis1 = []
y_axis1 = []
x_axis2 = []
y_axis2 = []
name1=''
name2=''




def compare_algorithms(x,y,name):
    if(counter==0):
        globals()['x_axis1'] = x
        globals()['y_axis1'] = y
        globals()['name1'] = name
        globals()['counter'] += 1
    elif(counter==1):
        globals()['x_axis2'] = x
        globals()['y_axis2'] = y
        globals()['name2'] = name
        globals()['counter'] += 1
    if(counter==2):
        plt.plot(x_axis1,y_axis1,label=name1)
        plt.plot(x_axis2,y_axis2,label=name2)
        plt.xlabel('n numbers')
        plt.ylabel ('steps taken')
        plt.legend()
        plt.show()
```

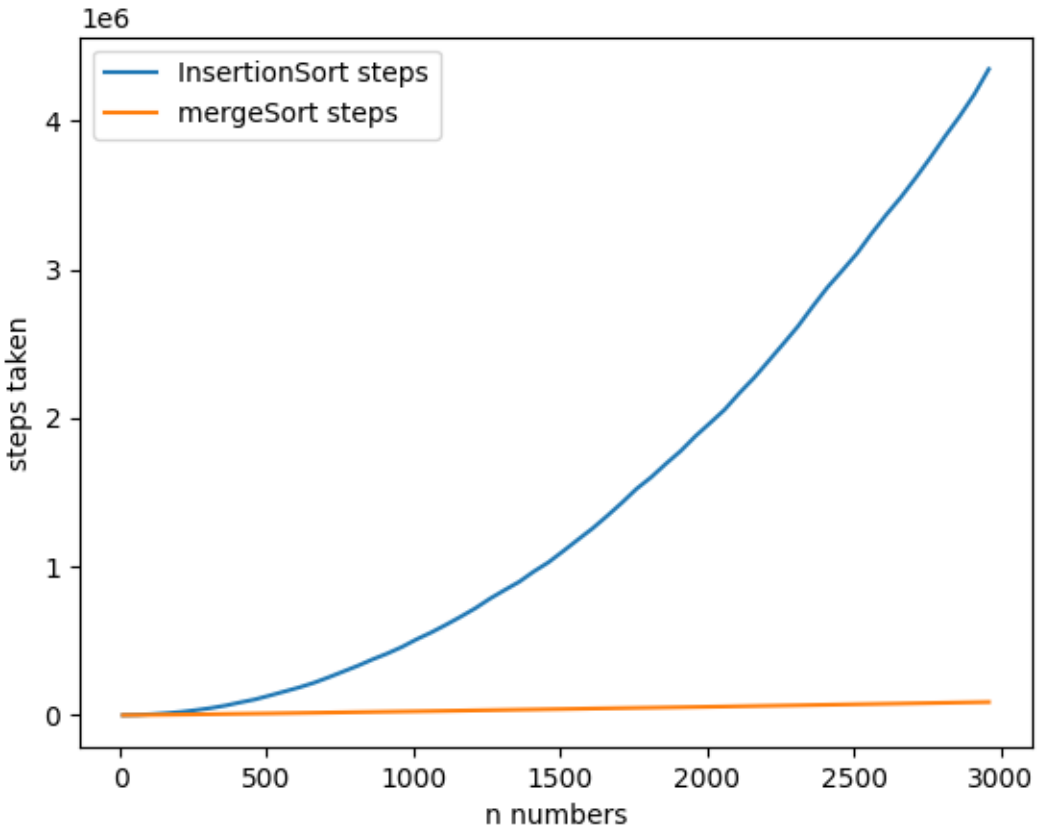
In Insertion Page

```
def compare_insertion():
    x,y = get_steps()
    name = 'InsertionSort steps'
    compare_algorithms(x,y,name)
```

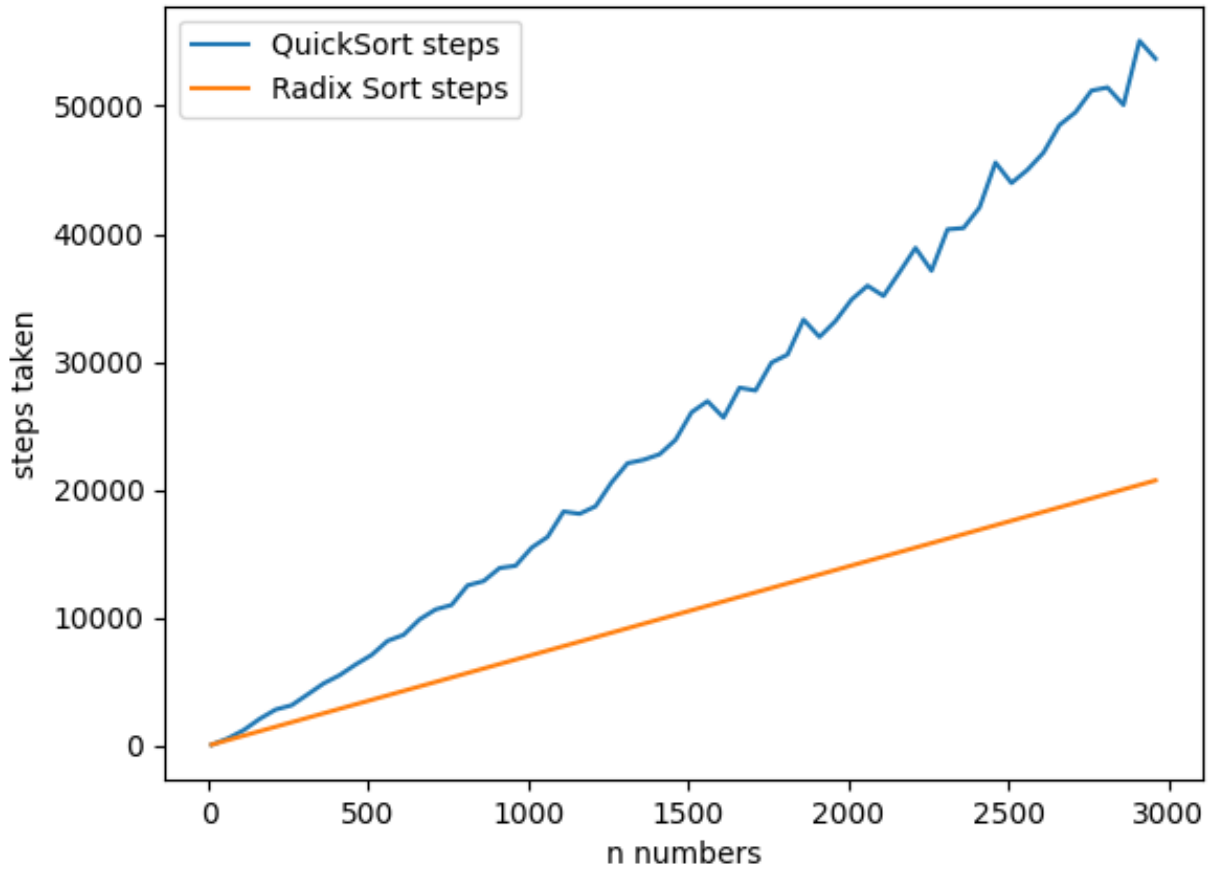
And the button in GUI calls compare_insertion and these steps are repeated with every selected algorithm.

Comparing Algorithms Examples

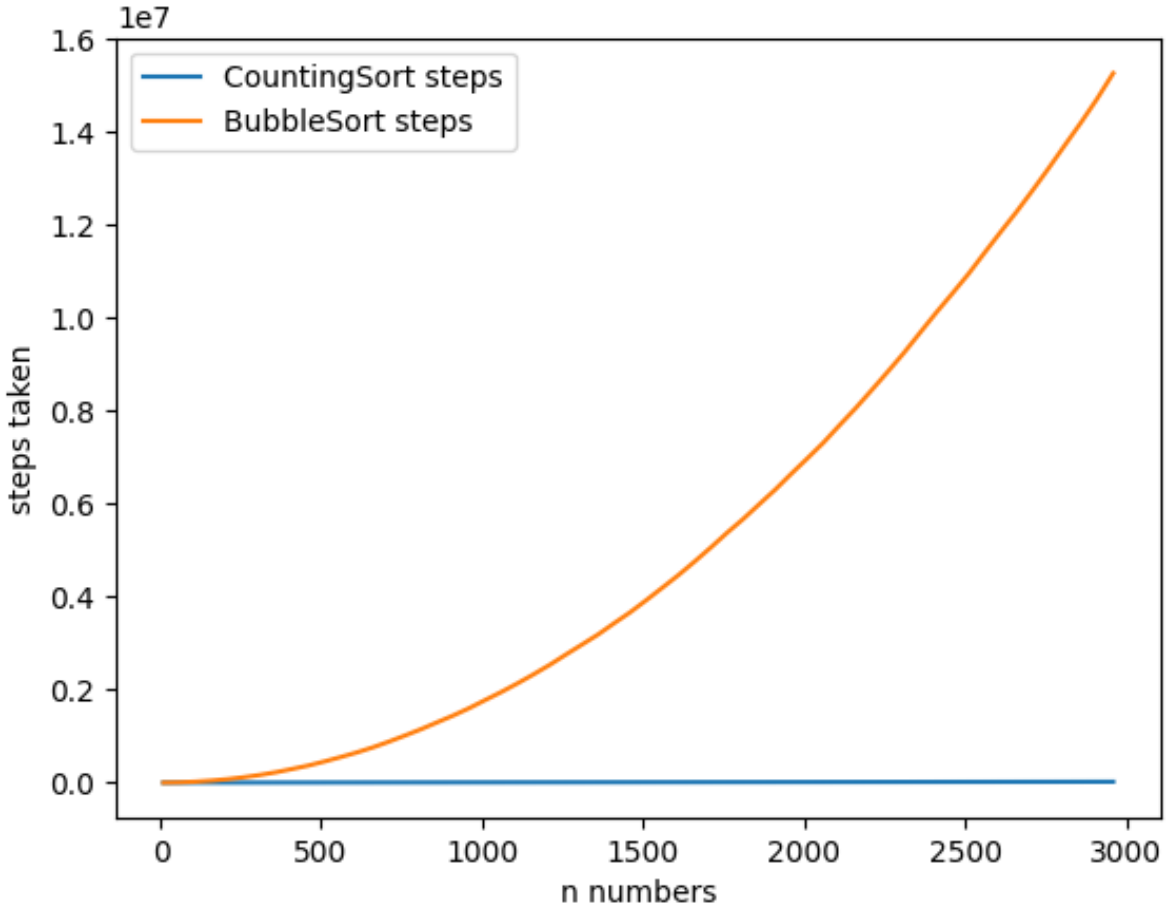
1) Insertion Sort Versus Merge Sort Steps



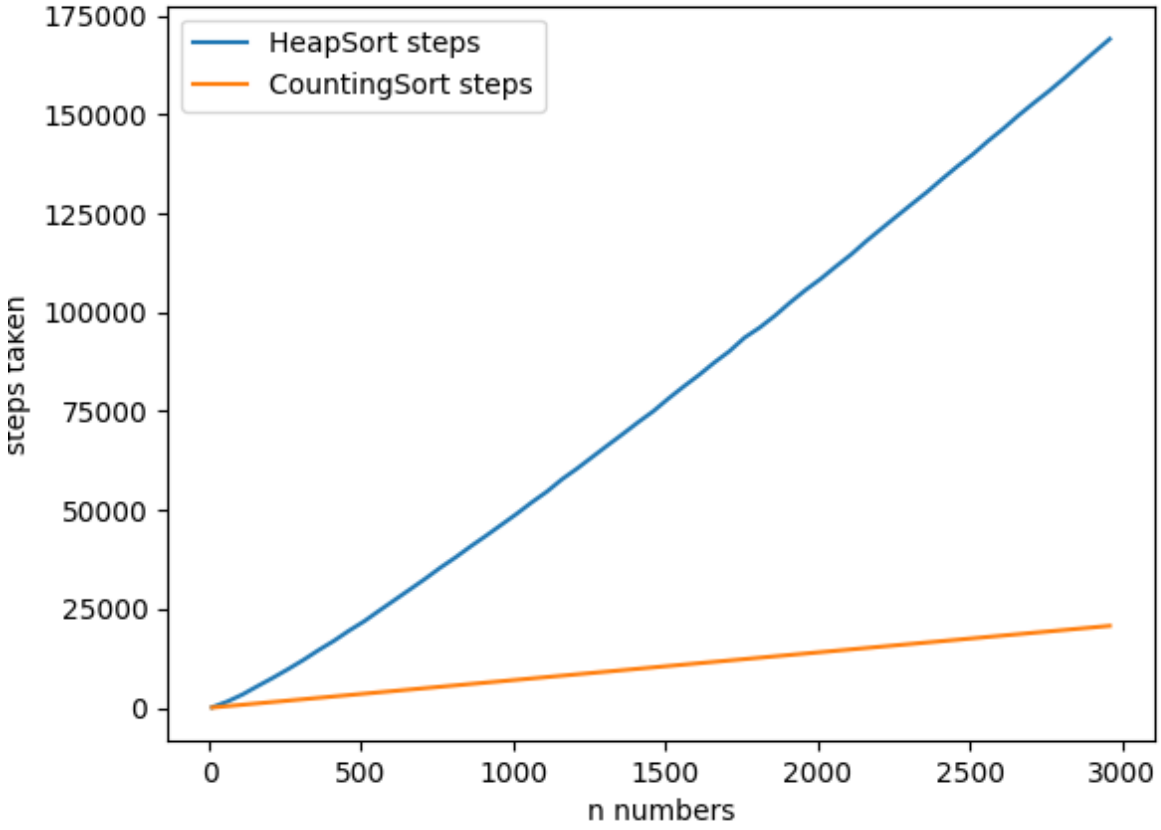
2) Quick Sort Versus Radix Sort Steps



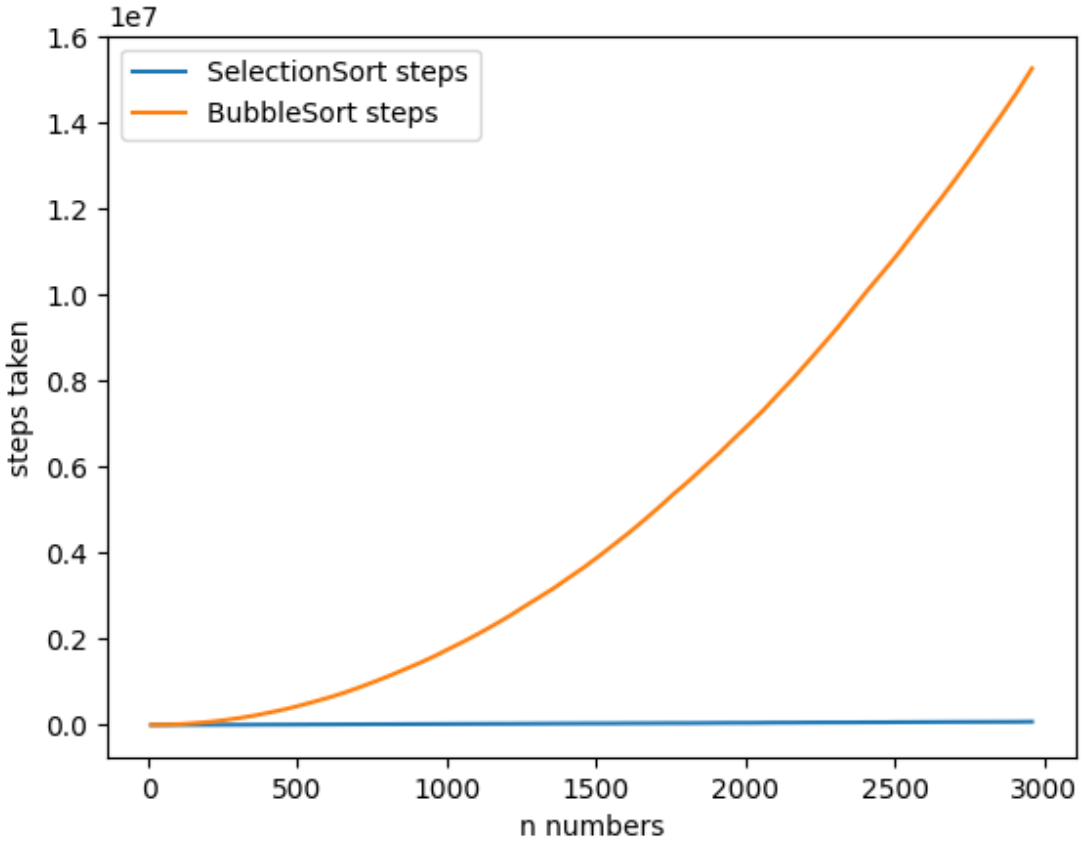
3) Counting Sort Versus Bubble Sort Steps



4) Heap Sort Versus Counting Sort Steps



5) Selection Sort Versus Bubble Sort Steps



Student	Contribution
Aya Ahmed Gamal 19P1689	Algorithms Insertion Sort, Heap Sort in coding & document.
Alaa Mohamed Galal 19P4206	Algorithms Selection Sort , Merge Sort in Coding and document.
Shaimaa Mohamed 19P7484	Algorithms Quick Sort, Bubble Sort in coding and document.
Ziad Mohamed Naser 19P2061	Algorithms Counting Sort, Radix Sort in coding and document.
	All contributed in GUI and making comparison code