Project Report
Sorting Algorithms

Chosen Project Number: Project 1 (Sorting Algorithms)
Programming Language used: Python
First Name: Krishna
Last Name: Sankaramani Ramamoorthie
UTA Student ID: 1001980344

Main Data Structure used: Arrays

Array Data Structure, an array organizes items sequentially, one after the another in memory.Each position in the array has an index, starting at 0. Arrays provide fast lookups and fast appends but they have fixed size,costly insertion and deletion.

Selection Sort Algorithm:

Selection sort is a simple sorting algorithm. This sorting algorithm is an in-place comparison based algorithm in which the list is divided into 2 parts: 1) the one part of the array that is already sorted and the remaining part which is unsorted.
Selection sort has a time complexity of O(n^2) for all the worst,best and average cases.

```
def Selectionsort(array): #Function Selectionsort
  for i in range(0,len(array)):
    min = i         # min holds the index of the minimum value in the array
    for j in range(i+1,len(array)):
      if (array[j]<array[min]):
        min=j                    #after looping we swap the min index value
    temp = array[i]
    array[i]=array[min]
    array[min]=temp
  return array
```

The smallest element is selected from the array and swapped with the leftmost element. Here we use variable min to find the minimum element index. After finding the minimum index we swap it to the left most index.

Bubble Sort Algorithm:

Bubble sort algorithm works by repeatedly swapping the adjacent elements if they are in the wrong order.
Time complexity for worst and average cases is O(n^2) the worst case occurs when the array is reversely ordered. The best case is O(n) best case occurs when the array is already sorted.

```
def Bubblesort(array): #Function BubbleSort
  for i in range(0,len(array)):
    for j in range(0, len(array)-1):
      if (array[j]>array[j+1]): #we compare the current index value to the
next index value and swap
        temp= array[j]
        array[j]=array[j+1]
        array[j+1]=temp
  return array
```

In this algorithm we compare the current index value to the next index value and swap if they are not in order and finally we return the sorted array

Insertion Sort Algorithm:

Insertion sort is based on the idea that one element from the input elements is consumed in each iteration to find its correct position.
Time complexity for worst and average cases is O(n^2). The best case is O(n) best case occurs when the array is already sorted then it runs in a linear time.

```
def Insertionsort(array): #Function InsertionSort
  for i in range(1,len(array)):
    value = array[i] #we assign the variable value as the element which we
need to compare
    j=i-1
    while (j>=0)and(array[j]>value): #we compare the current value with
the previous or predecessor value
      array[j+1]=array[j] #we move the larger value one position up to
make space for swapping
      j=j-1
    array[j+1]=value
  return array
```

In this function we input an unsorted array.We compare the current value with the previous or predecessor value if they are not in order we move the largest value one position up to make space for swapping.

Merge Sort Algorithm:

Merge sort is a divide and conquer algorithm based on the idea of breaking down an array into several subarrays until each subarray consists of a single element and merging those sub lists in a manner that results in a sorted list.
Selection sort has a time complexity of O(n*Log n) for all the worst,best and average cases.

```
def Mergesort(array):
  if (len(array)==1):   # if length of the array is 1 we return the array
cuz there is no need to sort the array
    return array
  mid=int((len(array)/2))  #finding the middle index of the array
  Left=array[0:mid]       #storing the first and second half of the array
in variable left and right
  Right=array[mid:len(array)]
  Left=Mergesort(Left)        #calling mergesort on the first half
  Right=Mergesort(Right)      #calling mergesort on the second half
  return Merge(Left,Right)
```

This function we find the middle index of the array and store the first and second half of the array in the left and right variable. Then we call merge sort on the first half and merge sort on the other half.

```
def Merge(L,R):
  x=0
  y=0
  z=0
  k=len(L)+len(R)
  array_t=[None]*k
  while(x<len(L))and(y<len(R)):
    if (L[x]<R[y]):    #find the smaller value in L and R store that in
the temperory array
      array_t[z]=L[x]
      x=x+1
    else:
      array_t[z]=R[y]
      y=y+1
    z=z+1
  while(x<len(L)):  #when the R array is empty we store the remaining
value of L array to the temperory array
```

```
    array_t[z]=L[x]
    z=z+1
    x=x+1
  while(y<len(R)): #when the L array is empty we store the remaining value
of R array to the temperory array
    array_t[z]=R[y]
    z=z+1
    y=y+1
  return array_t
```

In this function we create a temporary array array_t. Then we compare the L and R array and find the smaller value in L and R and store that value in the array_t. when one of the arrays becomes empty , then we just store the remaining values of the array that are not empty to array_t and finally we return array_t.

Quick Sort Algorithm:

Similar to merge sort, quick sort is a divide and conquer algorithm. It picks the element as the pivot and partitions the given array around the pivot that is picked.
Time complexity for worst and average cases is O(n^2) the worst case occurs when the partition element that is picked is the smallest or largest element in the array. The best case is O(n* Log n) best case occurs when the partition always picks the middle elements as pivot.

```
def Quicksort(array,low,high): #Function Quicksort
  if(low<high):
    piv=Partition(array,low,high) #calling partition function to partition
the array
    Quicksort(array,low,(piv-1)) #calling Quicksort for the left half of
pivot
    Quicksort(array,(piv+1),high) #calling Quicksort for the Right half of
pivot
  return array
```

In this function we simply split the array after we get the pivot index from the partition function and call quicksort recursively for the first half and the other half of the array.

```
def Partition(array,low,high):
  pivot=array[high]    #Picking the last element as pivot
  i=low-1
  for j in range(low,high+1):
    if(array[j]<pivot): # if current value is smaller than the pivot we
increment the index of the samller value
      i=i+1
      temp=array[j]
      array[j]=array[i]
      array[i]=temp
  temp1=array[i+1]     #finally we swap the last value with i+1 index
value
  array[i+1]=array[high]
  array[high]=temp1
  return (i+1)
```

In this function we chose pivot as the last element and we start from the leftmost element and keep track of index of smaller elements as i, while traversing, if we find a smaller element, we swap the current element with array[j]. Otherwise we ignore the current element.

Quick Sort using 3 medians Algorithm:

The best case of quick sort is that if we could find the middle element. So we choose the first, middle and last entry of the array then we sort those 3 elements and pick the middle one that will usually give a better approximation of the actual median.
Quick sort using 3 medians has a time complexity of O(n*Log n) for all the worst,best and average cases because we pick the middle element as the pivot.

```
def quicksort3median(array,low,high):  #Function Quicksort using 3 medians
  if(low<high):
    piv=parti(array,low,high)   #calling partition function to partition
the array
    if(piv!=0):
      quicksort3median(array,low,(piv-1))  #calling Quicksort3median for
the left half of pivot
      quicksort3median(array,(piv+1),high)  #calling Quicksort3median for
the other half of pivot
  return array
```

In this function we simply split the array after we get the pivot index from the partition function and call quicksort3median recursively for the first half and the other half of the array.

```
def parti(array,low,high):
  if((high-low)<2):     #when the size of the array if less than 2 we simply insertion sort the array
    Insertionsort1(array,low,high)
    return 0
  else:
    first=low
    mid=int((low+high)/2)     #finding the middle index
    last=high
    if(array[mid]<array[first]):    #after finding the first, middle, last elements we sort them simply using if else
      temp=array[mid]
      array[mid]=array[first]
      array[first]=temp
    if(array[last]<array[mid]):
      temp=array[last]
      array[last]=array[mid]
      array[mid]=temp
      if(array[mid]<array[first]):
        temp=array[mid]
        array[mid]=array[first]
        array[first]=temp
    temp=array[mid]
    array[mid]=array[high-1]
    array[high-1]=temp
    pivot=array[high-1]   #we pick the middle element of the sorted 3 values as the pivot
    i=low
    for j in range(low+1,high): # we do the same partition as the quick sort
      if(array[j]<pivot):
        i=i+1
        temp=array[j]
        array[j]=array[i]
        array[i]=temp
    temp1=array[i+1]
    array[i+1]=array[high-1]
```

```
    array[high-1]=temp1
    return (i+1)
```

In this function when the size of the array is less than 3 we simply use insertion sort to sort the array. To find the pivot we take first, last and middle elements of the array and using the if statement we sort those three values and pick the middle of the sorted value as pivot now this will be our pivot and we do the same partition as quick sort.

```
def Insertionsort1(array,low,high): #Function Insertionsort1  we call this function when the array size is less than 2
  for i in range(low+1,high+1):
    value = array[i]
    j=i-1
    while (j>=0)and(array[j]>value):
      array[j+1]=array[j]
      j=j-1
    array[j+1]=value
  return array
```

This function is the same as the insertion sort function.

Heap Sort Algorithm:

Heap sort is a comparison-based sorting technique based on Binary Heap data structure. It is similar to selection sort where we first find the minimum element and place the minimum element at the beginning. We repeat the same process for the remaining elements.

Heap Sort has a time complexity of O(n*Log n) for all the worst,best and average cases.

```
def Heapsort(array): #Function Heapsort
  array1=[0]*(len(array)+1) #we move the array elements by one index so that it will be easy to sort
  for i in range(0,len(array)):
    array1[i+1]=array[i]
  for j in range((int(len(array)/2)),0,-1):  #the parents of the of the heapsort tree will be the first half of the array so we run the loop upto half of the array
    Heapifyarray(array1,j,len(array1)-1)   # First we heapify the given array into max heap by calling the heapifyarray function
  for k in range(len(array),1,-1): # we swap the first and last element of the array and do max heapify until all the elements in the array are sorted
```

```
    temp=array1[k]
    array1[k]=array1[1]
    array1[1]=temp
    Heapifyarray(array1,1,k)
  for p in range(0,len(array)): #Finally we move the all the elements one
index value back to original position and return the array
    array[p]=array1[p+1]
  return array
```

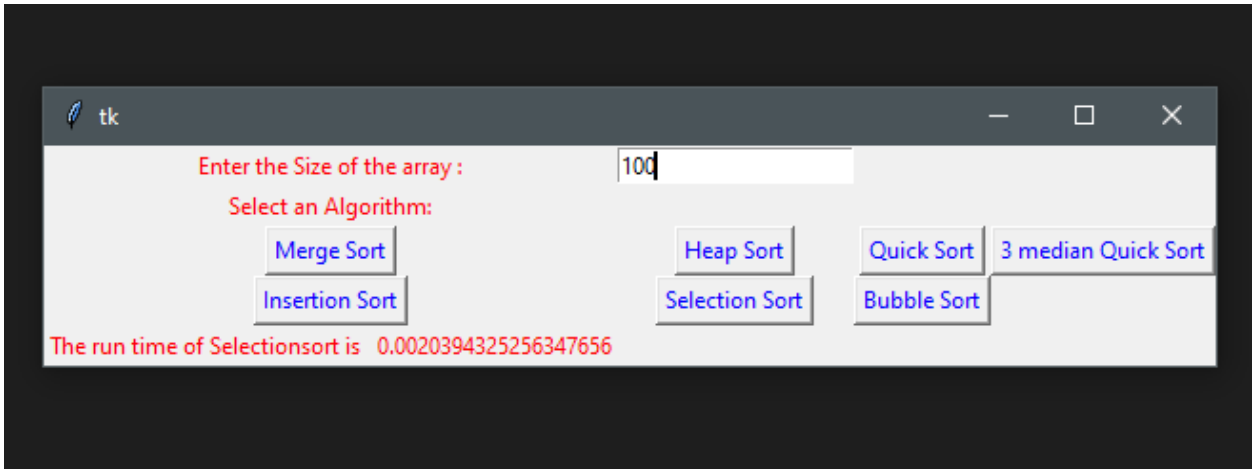
In this function we move the array elements by one index so that it will be easy to sort and then the parents of the heapsort tree will be the first half of the array so we run the loop upto half of the array. First we heapify the given array into max heap by calling the heapifyarray function after doing the max heap we swap the first and last element of the array and do max heapify until all the elements in the array are Finally we move the all the elements one index value back to original position and return the array

```
def Heapifyarray(array,parent,size):
  max=parent                          #for the heap sort array the left
child of parent is parent*2 and the right child of the parent is
parent*2+1 so we stire the index of those value in vairable leftchild and
rightchild
  leftchild=parent*2
  rightchild=(parent*2)+1
  if(leftchild<size)and(array[parent]<array[leftchild]): #now we compare
the leftchild value and the parent value and the larger ones index will be
stored in the max variable
    max=leftchild
  if(rightchild<size)and(array[max]<array[rightchild]):   #now we compare
the rightchild value and the parent value and the larger ones index will
be stored in the max variable
    max=rightchild
  if max!=parent:             #if the max index is not the parent's index
then we the swap the max index value with the parent index value
    temp=array[max]
    array[max]=array[parent]
    array[parent]=temp
    Heapifyarray(array,max,size)  #Finally we call Heapifyarray
recursively until all the elements are in the max heaptree form
```

In this function for the heap sort array the left child of parent is parent*2 and the right child of the parent is parent*2+1 so we store the index of those value in variable left child and right child ,now we compare the left child value and the parent value and the larger ones index will be stored in the max variable now we compare the rights child value and the parent value and the larger ones index will be stored in the max variable if the max index is not the parent's index then we the swap the max index value with the parent index value. If the max index is not the parent's index then we the swap the max index value with the parent index value.

Graphical User-interface Used: Tkinter

Python has a lot of GUI frameworks, but Tkinter is the only framework that's built into the Python standard library. Tkinter has several strengths. It's cross-platform, so the same code works on Windows, macOS, and Linux. Visual elements are rendered using native operating system elements, so applications built with Tkinter look like they belong on the platform where they're run



The design of the user interface is to get the size of input from the user and select any algorithms which are provided in buttons and it will print the run time of that algorithm for that input size.

```
from tkinter import *
root=Tk()  #assign the tkinter call to root variable for easy calling
inputlabel=Label(root,text="Enter the Size of the array :",fg="red")
#making label for getting the size of the input
inputlabel.grid(row=1,column=0)         #here grid is used to position all
the labels, entry, button in specific place of the GUI
enter=Entry(root,width=20)  #making an entry space to enter the size of
the array
enter.grid(row=1,column=5)
```

```
inputlabel=Label(root,text="Select an Algorithm:",fg="red") #making label
to select an algorithm
inputlabel.grid(row=3,column=0)       #making buttons for each type of
sorting algorithm
button1=Button(root,text="Merge Sort",command=Mergesort1,fg="Blue")
button1.grid(row=5,column=0)
button2=Button(root,text="Heap Sort",command=Heapsort1,fg="Blue")
button2.grid(row=5,column=5)
button3=Button(root,text="Quick Sort",command=Quicksort1,fg="Blue")
button3.grid(row=5,column=10)
button4=Button(root,text="3 median Quick
Sort",command=quicksort3median1,fg="Blue")
button4.grid(row=5,column=15)
button5=Button(root,text="Insertion
Sort",command=Insertionsort1,fg="Blue")
button5.grid(row=7,column=0)
button6=Button(root,text="Selection
Sort",command=Selectionsort1,fg="Blue")
button6.grid(row=7,column=5)
button7=Button(root,text="Bubble Sort",command=Bubblesort1,fg="Blue")
button7.grid(row=7,column=10)
root.mainloop()    #finally calling the  mainloop() to run the GUI
```

Experimental Results:

This is for the Selection sort it will be the same for all the other sorts.

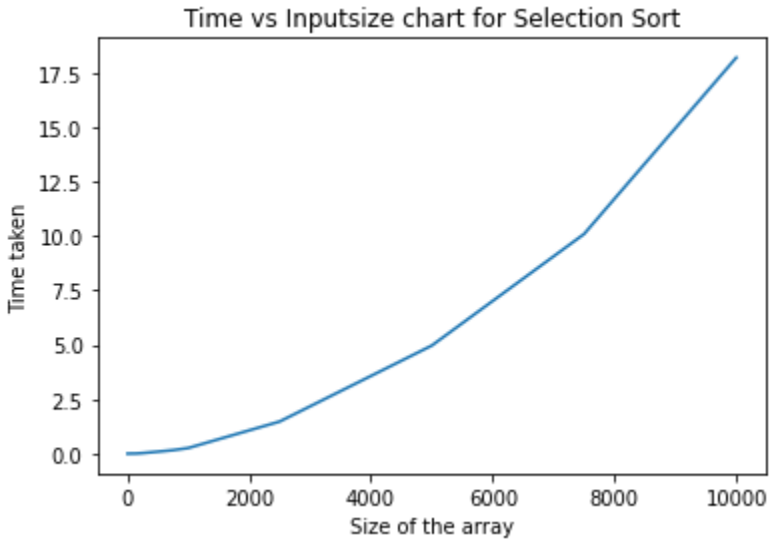
Run time vs Input size Code

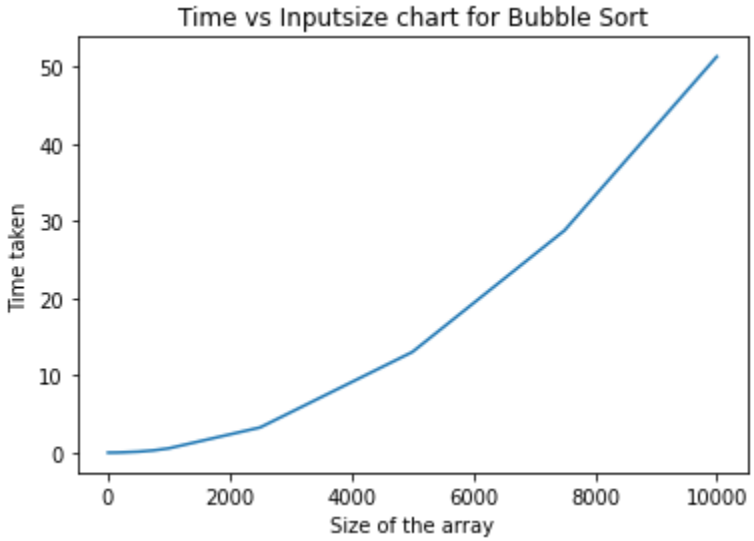
```
import time  #importing time library to calculate the run time of the
algorithm
size=[5,10,20,50,75,100,150,250,500,750,1000,2500,5000,7500,10000]
timetak=[None]*len(size)  #We initialize an array that will store the run
time  of the algorithm for each array size
i=0
for item in size:
  array_r=np.random.randint(1,100,item) #for each array size we initialize
random number between 1 to 100
  start=time.time()
  x=Selectionsort(array_r)
```

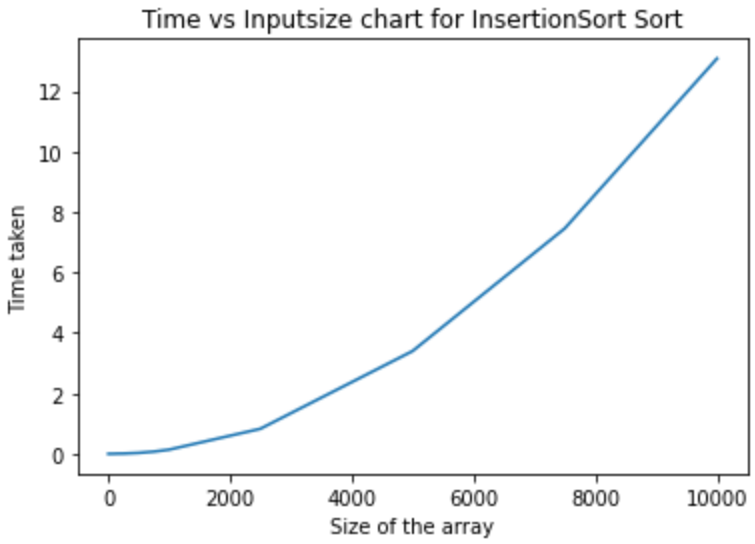
```
  end=time.time()
  timetak[i]=end-start
  i=i+1
print(timetak)
```

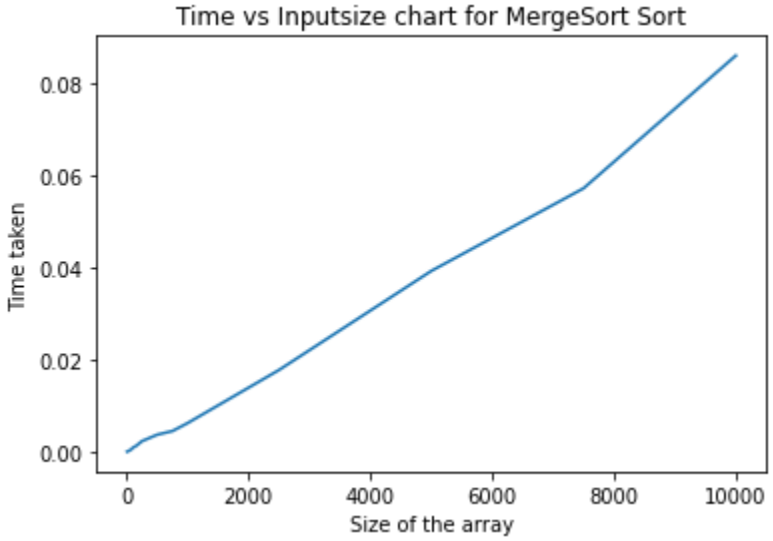
Run time vs Input

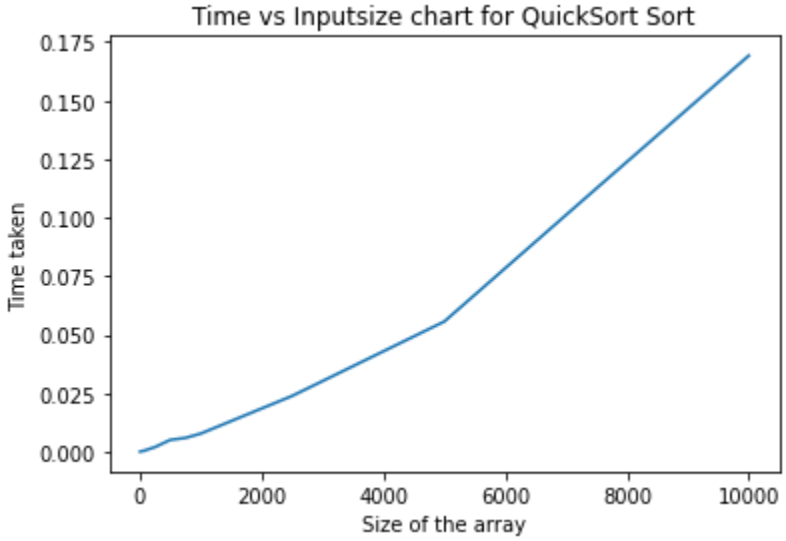
```
plt.plot(size,timetak)       #using matplotlib library we plot the time vs input chart for the given algorithm
plt.xlabel('Size of the array') #we name  the x label of the chart
plt.ylabel('Time taken') #we name the y label of the chart
plt.title('Time vs Inputsize chart for Selection Sort') #we name the title of the chart
plt.show()
```

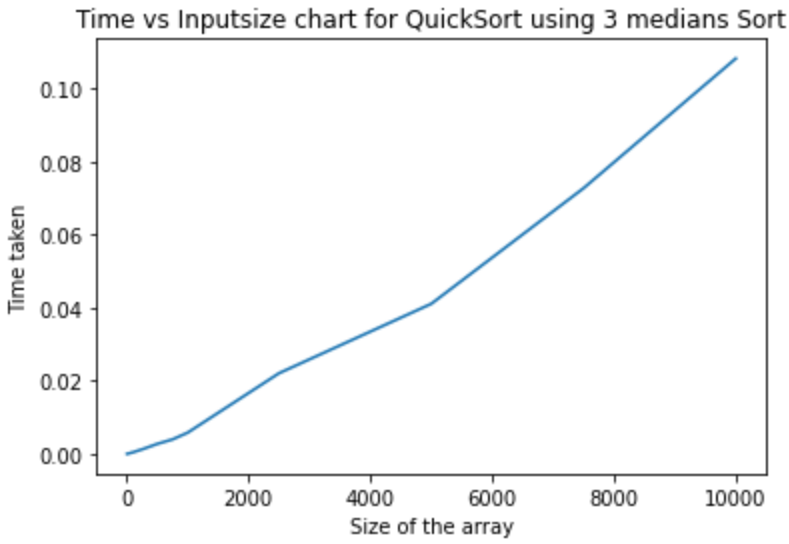


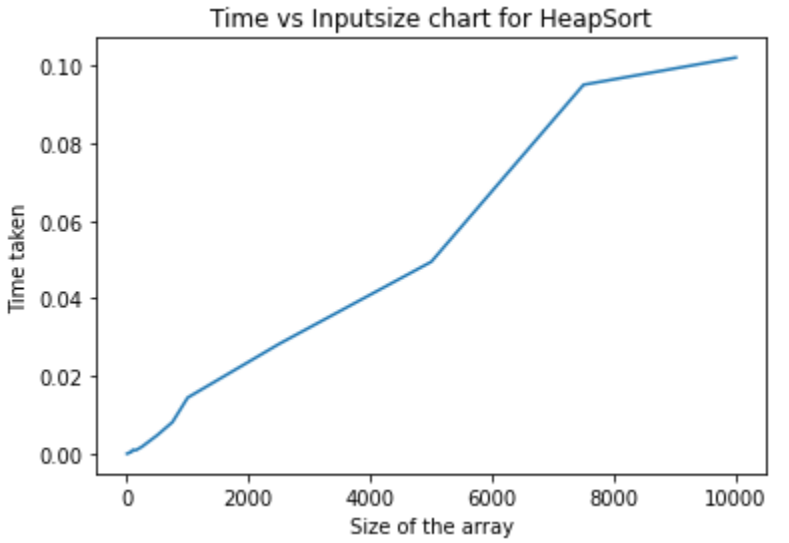



Time vs Inputsize chart for InsertionSort Sort
Time taken
Size of the array
0
2
4
6
8
10
12
0
2000
4000
6000
8000
10000


Time vs Inputsize chart for MergeSort Sort
Time taken
Size of the array
0.00
0.02
0.04
0.06
0.08
0
2000
4000
6000
8000
10000


Time vs Inputsize chart for QuickSort Sort
Time taken
Size of the array
0.000
0.025
0.050
0.075
0.100
0.125
0.150
0.175
0
2000
4000
6000
8000
10000



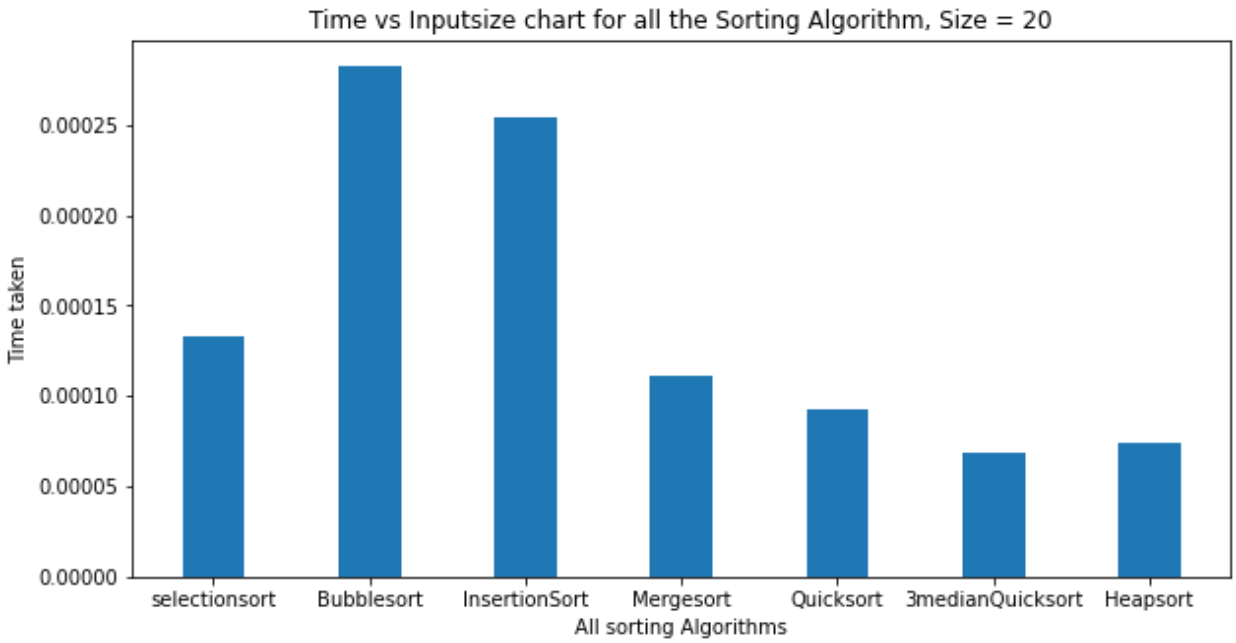


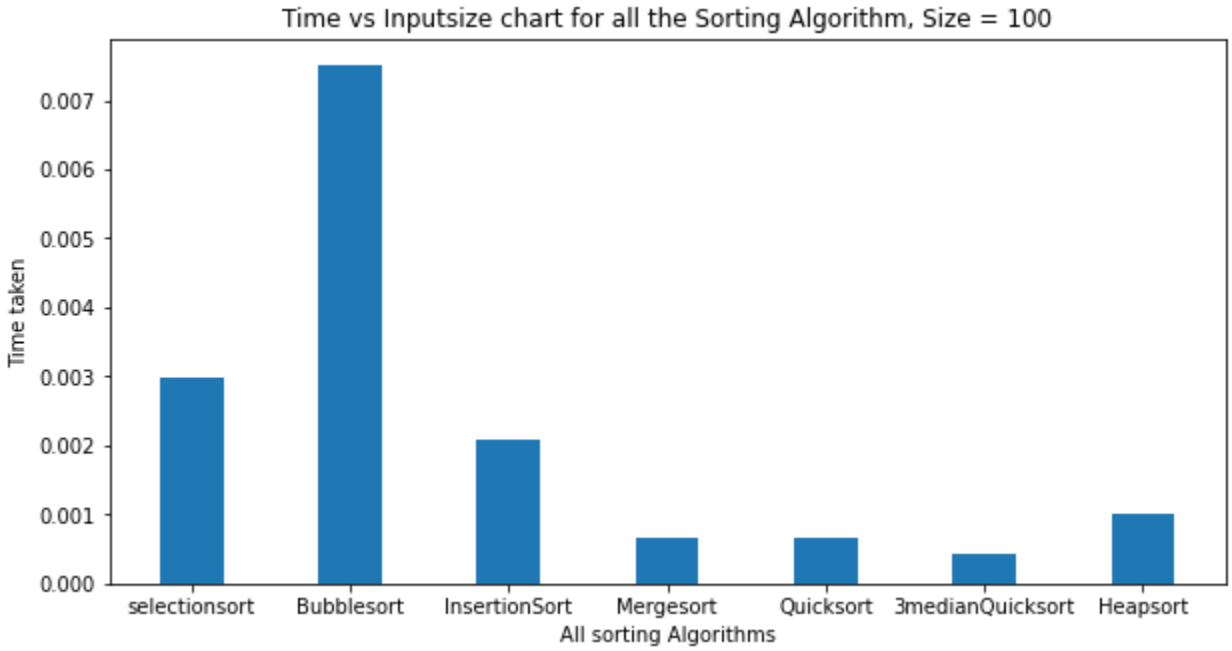
Run time vs all Sorting Algorithms for a specific input size

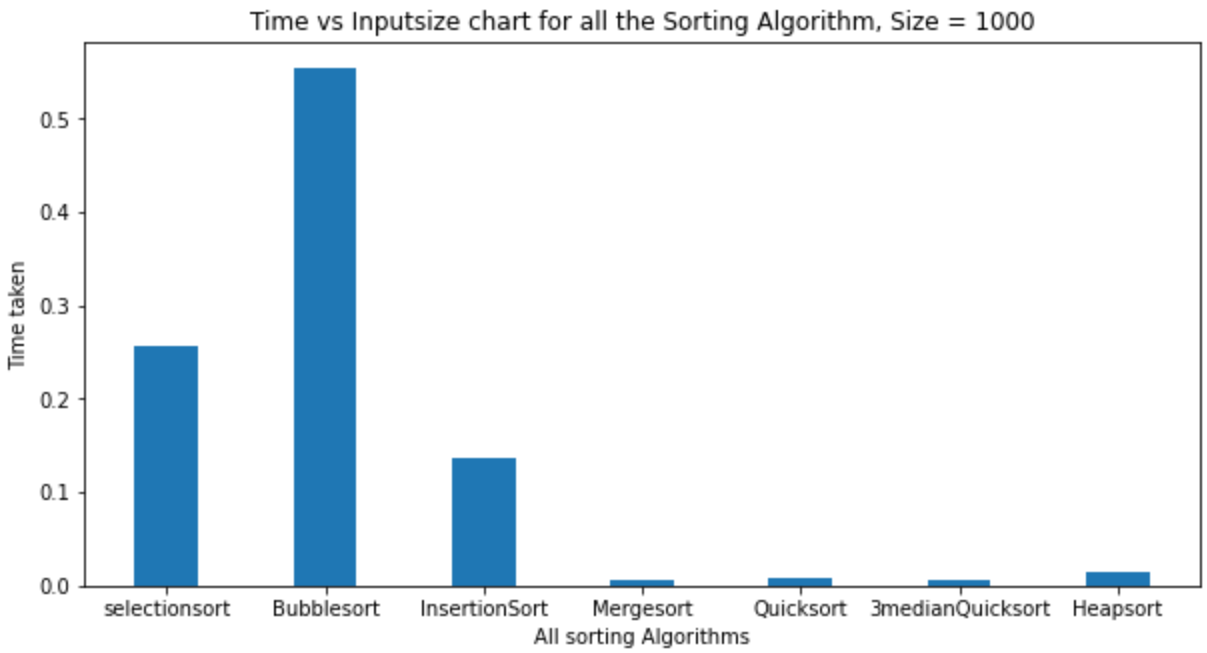
```
alltimetaken=[timetak,timetak1,timetak2,timetak3,timetak4,timetak5,timetak
6] #we store  the time taken in each and every algorithm in a single array
alltimetaken
for i in range (0,len(size)):         #now using Bar char we represent all
the comparison of the algorithms for different inputsize
  tik=[0]*7          #in tik array we store all the run time of algorithms
for each input size
  k=0
  for time in alltimetaken:
    tik[k]=time[i]
```

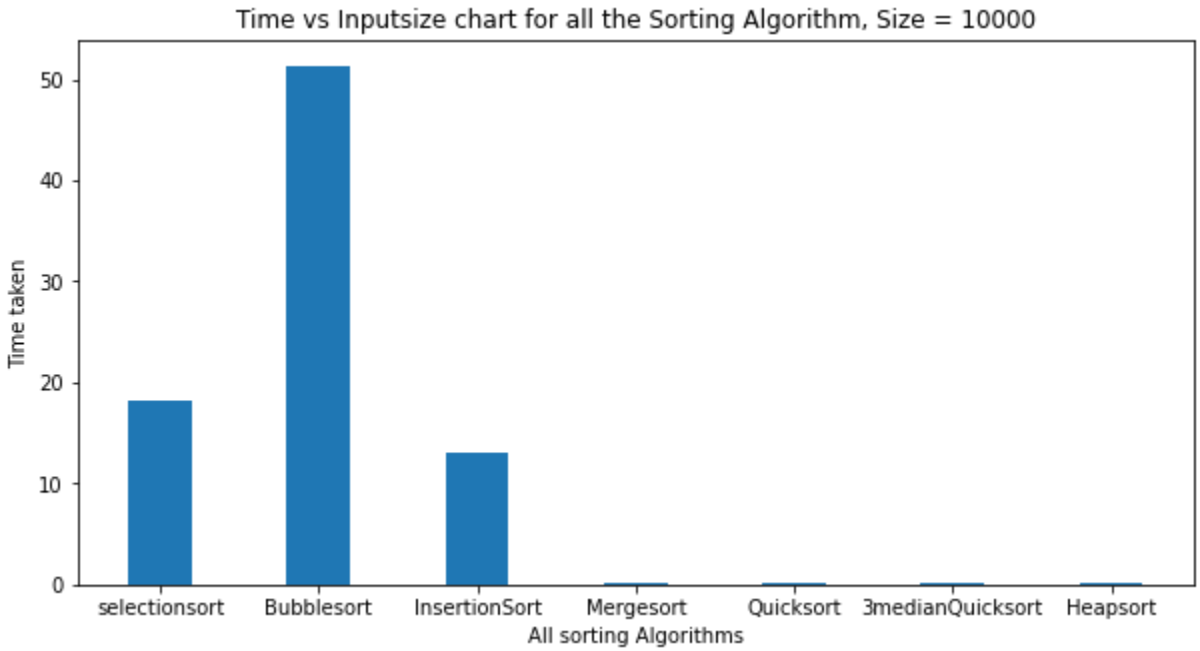
```
    k=k+1
  print(tik)
  plt.figure(figsize=(10,5)) #size of the figure

plt.bar(["selectionsort","Bubblesort","InsertionSort","Mergesort","Quicksort","3medianQuicksort","Heapsort"],tik,width=0.4) #calling barplot
  plt.xlabel('All sorting Algorithms')  #we name  the x label of the chart
  plt.ylabel('Time taken')  #we name the y label of the chart
  plt.title('Time vs Inputsize chart for all the Sorting Algorithm, Size = '+str(size[i])) #we name the title of the chart
  plt.show()
```


Time vs Inputsize chart for all the Sorting Algorithm, Size = 20
Time taken
0.00025
0.00020
0.00015
0.00010
0.00005
0.00000
selectionsort
Bubblesort
InsertionSort
Mergesort
Quicksort
3medianQuicksort
Heapsort
All sorting Algorithms


Time vs Inputsize chart for all the Sorting Algorithm, Size = 100
Time taken
0.007
0.006
0.005
0.004
0.003
0.002
0.001
0.000
selectionsort
Bubblesort
InsertionSort
Mergesort
Quicksort
3medianQuicksort
Heapsort
All sorting Algorithms

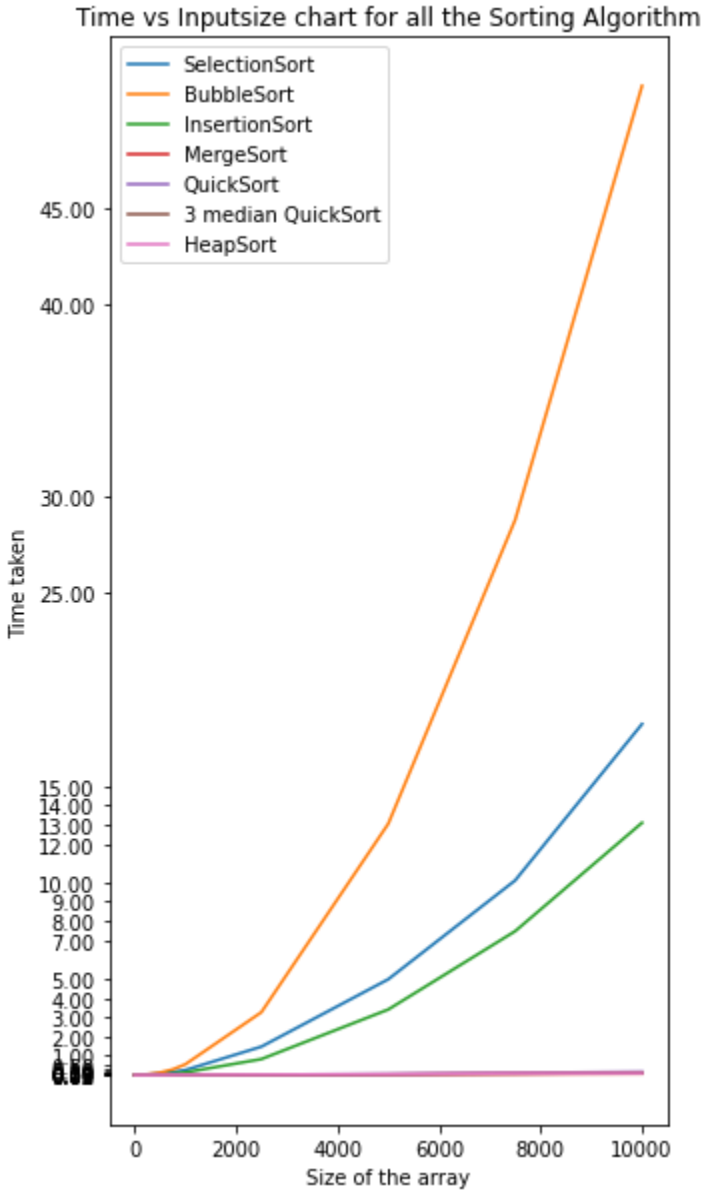




Run time vs all Sorting Algorithms

```
plt.figure(figsize=(10,20))
plt.plot(size,timetak,label="SelectionSort")  #using matplotlib library we
plot the time vs input chart for all the given algorithm
plt.plot(size,timetak1,label="BubbleSort")
plt.plot(size,timetak2,label="InsertionSort")
plt.plot(size,timetak3,label="MergeSort")
```

```
plt.plot(size,timetak4,label="QuickSort")
plt.plot(size,timetak5,label="3 median QuickSort")
plt.plot(size,timetak6,label="HeapSort")
plt.xlabel('Size of the array')  #we name  the x label of the chart
plt.ylabel('Time taken')  #we name the y label of the chart
plt.title('Time vs Inputsize chart for all the Sorting Algorithm') #we
name the title of the chart
plt.yticks([0.01,0.02,0.03,0.05,0.08,0.09,0.1,0.2,0.3,0.5,0.10,0.11,0.12,0
.13,1,2,3,4,5,7,8,9,10,12,13,14,15,25,30,40,45])
plt.legend()
plt.show()
```



Conclusion:

All the Sorting algorithms, GUI, and the experimental results are mentioned.
The algorithm that is better from all the algorithms is the Merge sort, Quick sort, 3 median Quicksort, and Heap sort as we can see at the input size 10000 their runtime is very low compared to all the other algorithms. The improvement of the running time of the algorithms depends on various cases but mainly it depends on the input size and functions. To improve the performance we can replace a nested loop by first building a hash and then looping, removing unnecessary accumulations.