REPORT

Project 1: Comparison-based Sorting Algorithms

Submitted By

1. Raj Shah (ID # 801205036)

2. Parth Patel, (ID# 801212088)

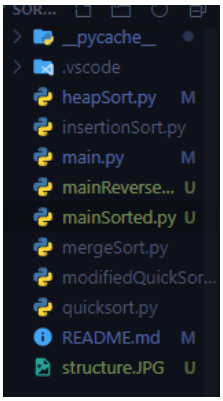
Code Repository Overview

In this project, we have **created input from random function in python.There are three main files which are starting point for Program Files names are main.py (Random Array) , mainSorted.py (Sorted Array) , mainReverse.py (reverse sorted array)** . Each of files will create different sample sized files ranging from 1,000 to 60,000.

So for example, **main.py will generate array** containing 10,000 samples in random order and **main.py will call 5 function imports** which will sort Input array and give the time taken in milliseconds.

Please check the **README** file to get source code execution instruction and i/o specification.

Project Structure :



Data Structure

In this project we use python as our programming language. To implement the sorting algorithms, in most of the cases we used a array of size same as sampledata test. We don't use any library function in implementing the algorithms of this project. We prepared several python scripts to test the execution time of our implemented algorithms.

Complexity Analysis

For **Insertion sort** the running time of different formatted input files are as follows:

1. Sorted inputs (best case): $T(n)=\theta(n)$
2. Reversely sorted inputs (worst case): $T(n)=\theta(n^2)$
3. Random inputs (average case): $T(n)=\theta(n^2)$

For **Merge sort** the running time of different formatted input files are as follows:

1. Sorted inputs: $T(n)=O(nlogn)$
2. Reversely sorted inputs: $T(n)=O(nlogn)$
3. Random inputs: $T(n)=O(nlogn)$

For **Heapsort** the running time of different formatted input files are as follows:

1. Sorted inputs: $T(n)=O(nlogn)$
2. Reversely sorted inputs: $T(n)=O(nlogn)$
3. Random inputs: $T(n)=O(nlogn)$

For **In-place quicksort** the running time of different formatted input files are as follows:

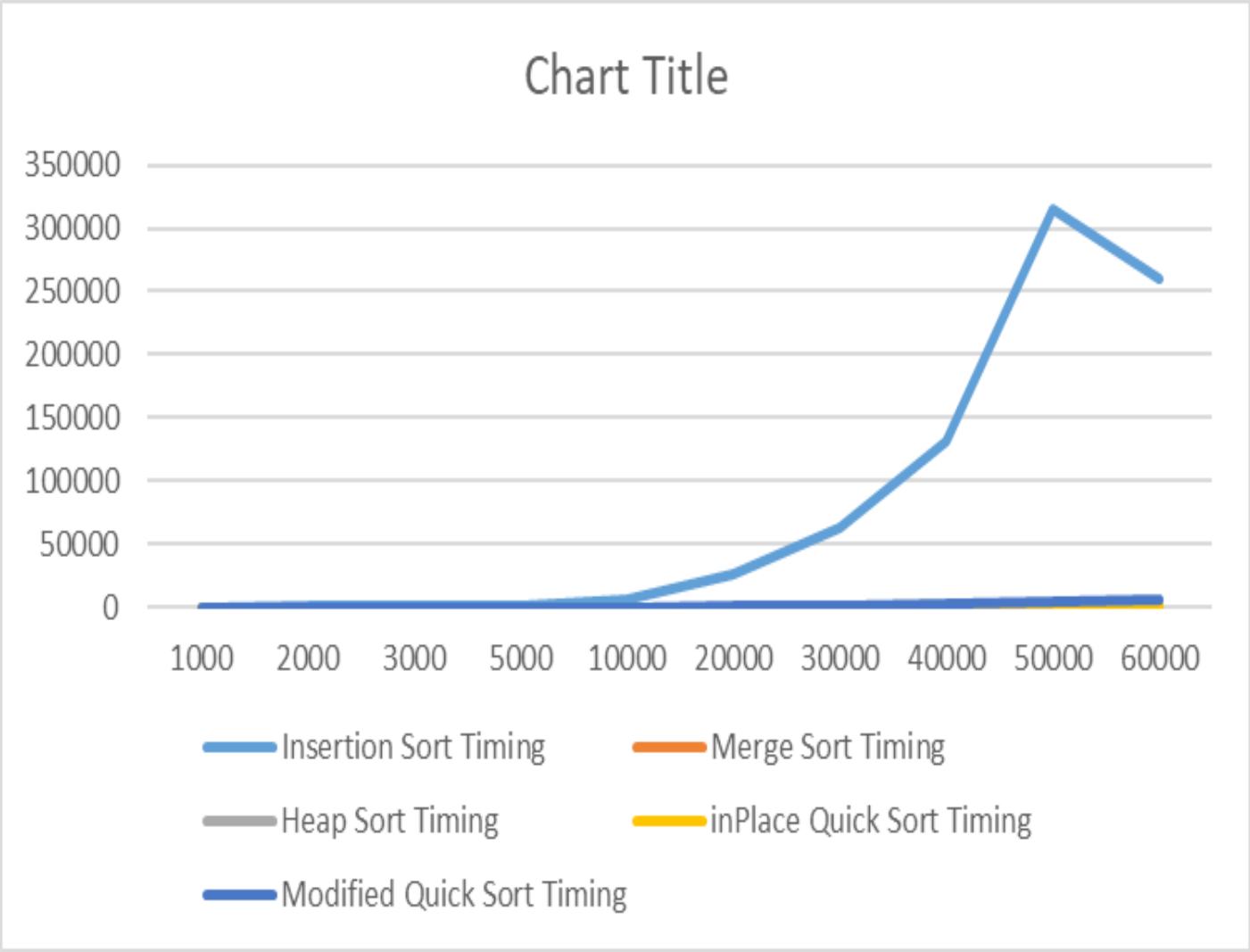
1. Sorted inputs (worst case): $T(n)=O(n^2)$
2. Reversely sorted inputs (worst case): $T(n)=O(n^2)$
3. Random inputs (average case): $T(n)=O(nlogn)$

For **Modified quicksort** the running time of different formatted input files are as follows:

4. Sorted inputs: $T(n)=O(nlogn)$
5. Reversely sorted inputs: $T(n)=O(nlogn)$
6. Random inputs: $T(n)=O(nlogn)$

Results

Random data(unsorted):



Insertion Sort Timing :
[121.9930648803711, 297.00684547424316, 880.000114440918, 1638.995885848999, 6577.994585037231, 26594.9809551239, 63078.937292099, 130860.46147346497, 314660.1912975311, 259416.87440872192]
Merge sort Timing :
[17.99774169921875, 19.99640464782715, 28.003931045532227, 36.00502014160156, 84.00487899780273,

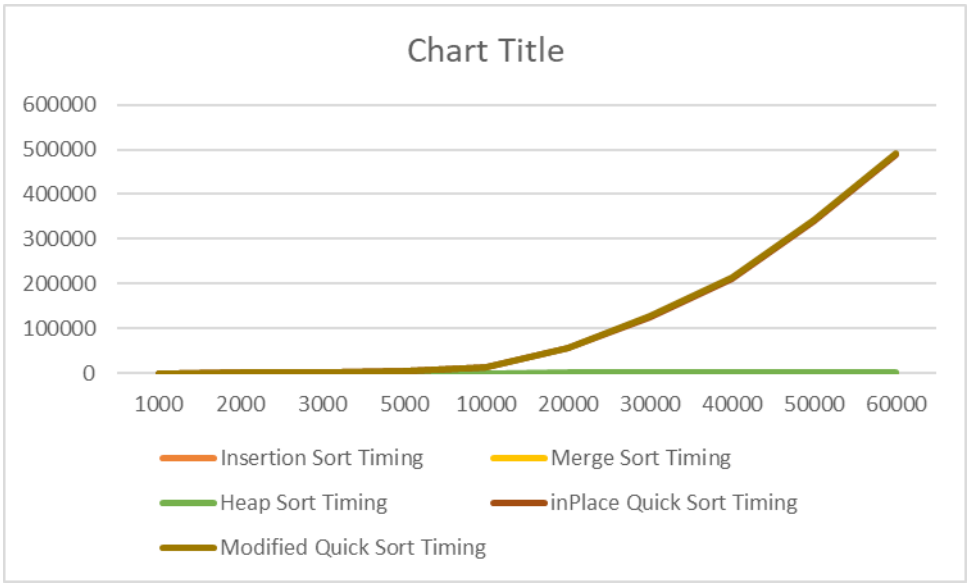
162.0025634765625, 261.9976997375488, 353.99913787841797, 450.000524520874, 561.030387878418]

Heapsort Timing :
[7.055759429931641, 17.04859733581543, 21.006345748901367, 41.004180908203125, 91.00103378295898, 201.93743705749512, 595.9982872009277, 1205.014944076538, 854.0024757385254, 1000.0002384185791]

In-Place Quicksort Timing :
[4.996061325073242,11.004924774169922,12.999296188354492, 18.00227165222168, 42.99783706665039, 94.00296211242676, 134.0012550354004, 231.0347557067871, 239.00079727172852, 591.008186340332]

Modified quicksort Timing :
[5.003690719604492,16.000032424926758,22.000551223754883,52.99878120422363,176.00417137145996,722.996711730957,1600.996732711792,2887.9692554473877,4360.9983921051025,6260.988712310791]

[2]Sorted data:



Insertion Sort Timing :

[0.0, 1.0020732879638672, 0.0, 2.0172595977783203, 3.0159950256347656 6.999969482421875, 10.001420974731445, 10.027408599853516,15.043973922729492, 21.99554443359375]

Merge sort Timing :

[4.996776580810547, 15.00248908996582, 29.000043869018555, 30.982017517089844 60.96625328063965,123.86918067932129, 215.99674224853516,285.9675884246826,341.95566177368164, 456.024169921875]

Heapsort Timing :

[11.001825332641602,27.001142501831055,36.99827194213867, 73.00281524658203, 169.9998378753662, 402.9829502105713, 508.9995861053467, 717.0071601867676, 853.9958000183105, 1159.428358078003]

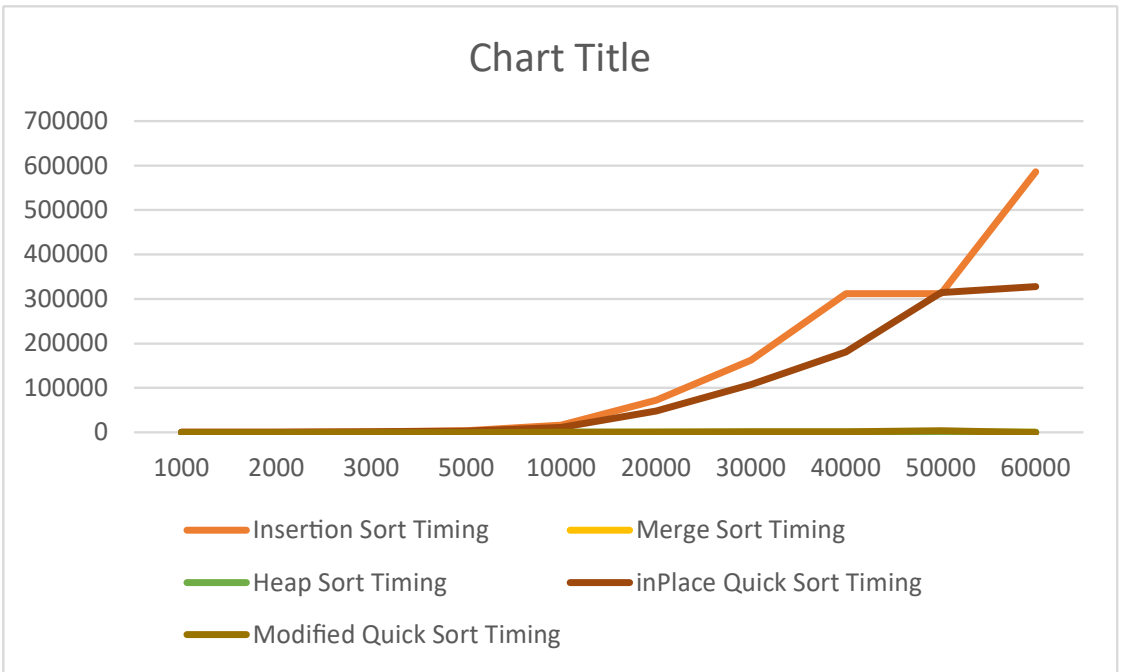
In-Place Quicksort Timing :

[188.00020217895508, 978.996992111206, 1344.0313339233398, 4027.99916267395, 13016.411304473877, 54444.00119781494, 125225.1501083374, 210117.05875396729, 337838.9916419983, 486028.78427505493]

Modified quicksort Timing :

[6.006717681884766, 11.003732681274414, 20.969152450561523, 33.03718566894531,120.03397941589355, 406.0351848602295, 1170.0026988983154, 1484.0011596679688, 3423.007011413574, 4113.03448677063]

[3]Reverse Sorted data:



Insertion Sort Timing :
[133.99958610534668, 656.9981575012207, 1437.002420425415, 4026.9994735717773,16232.070446014404,72323.18019866943, 162794.16847229004,312062.44349479675,471556.4212799072, 585642.1818733215]
Merge sort Timing :
[3.998279571533203,17.002105712890625,17.996549606323242, 37.99772262573242, 68.9995288848877, 195.00041007995605, 249.96018409729004,364.00294303894043,437.02006340026855, 467.95129776000977]
Heapsort Timing :
[10.000228881835938, 14.01209831237793, 28.00464630126953, 52.00052261352539, 132.06839561462402, 269.0014839172363, 386.99960708618164, 548.9969253540039, 722.9986190795898, 722.0017910003662]
In-Place Quicksort Timing :
[113.9991283416748,450.99854469299316,1006.9968700408936,28 26.9999027252197,11238.833904266357,48383.84437561035,1076 99.40948486328,180644.06895637512,314159.6176624298, 327561.9509220123]]
Modified quicksort Timing :

[5.0067175621884766,13.06839561462402,21.002420425415,34.9967 4224853516,130.44349479675,429.0351848602295, 1182.1026988983121,1484.0011596679688,3521.00052261352539, 4195. 00119781494]

Test platform:

- Processor: Intel(R) Xeon(R) CPU E5-2620 2.00GHz (12 Core)
- Windows 10
- Python 3.6.2

Code

1. Code for insertion_sort.py

```
def insertionSort(array):

    for x in range(1,len(array)):
        key=array[x]
        y=x-1
        while y>=0 and array[y]>key:
            array[y+1]=array[y]
            y=y-1
        array[y+1]=key
    # Use This Print to verify  while invoking from main file
    #print(array)


# Use for Indiviual Analysis and Code Correctness

#array=[2,1,6,9,10,11,60,14,15,22,8,88,40,24,36,75,888,14]
#InsertSort(array)
#print(array)
```

2.Code for merge_sort.py

```
def mergeSort(array):
    if len(array) > 1:

        mid = len(array)//2

        L = array[:mid]

        R = array[mid:]

        mergeSort(L)

        mergeSort(R)

        i = j = k = 0


        while i < len(L) and j < len(R):
            if L[i] < R[j]:
                array[k] = L[i]
                i += 1
            else:
                array[k] = R[j]
                j += 1
            k += 1
```

```
        while i < len(L):
            array[k] = L[i]
            i += 1
            k += 1

        while j < len(R):
            array[k] = R[j]
            j += 1
            k += 1
    return array


#array = [2,1,6,9,10,11,60,14,15,22,8,88,40,24,36,75,888,14]
#mergeSort(array)
#printList(array)
```

3.Code for heapsort.py

```
def heapify(arr, n, i):
    largest = i  # Initialize largest as root
    l = 2 * i + 1     # left = 2*i + 1
    r = 2 * i + 2     # right = 2*i + 2

    # See if left child of root exists and is
    # greater than root
    if l < n and arr[largest] < arr[l]:
        largest = l

    # See if right child of root exists and is
    # greater than root
    if r < n and arr[largest] < arr[r]:
        largest = r

    # Change root, if needed
    if largest != i:
        arr[i], arr[largest] = arr[largest], arr[i]  # swap

        # Heapify the root.
        heapify(arr, n, largest)

# The main function to sort an array of given size


def heapSort(arr):
    n = len(arr)

    # Build a maxheap.
    for i in range(n//2 - 1, -1, -1):
        heapify(arr, n, i)
```

```
    # One by one extract elements
    for i in range(n-1, 0, -1):
        arr[i], arr[0] = arr[0], arr[i]  # swap
        heapify(arr, i, 0)
    return arr


#arr = [12, 11, 13, 5, 6, 7]
#heapSort(arr)
```

4.Code for inplace_quicksort.py

```
def partition(arr, low, high):
   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]

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



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

      # pi is partitioning index, arr[p] is now
      # at right place
      pi = partition(arr, low, high)
        quickSort(arr, low, pi-1)
      quickSort(arr, pi+1, high)
```

5.Code for modified_quicksort.py

```
medianC = 0
def median(a, b, c):
    if ( a - b) * (c - a) >= 0:
        return a
    elif (b - a) * (c - b) >= 0:
        return b
    else:
        return c

def partition_median(array, smallValArr, highValArr):
    small = array[smallValArr]
    high = array[highValArr - 1]
    length = highValArr - smallValArr
    middle = array[smallValArr + length // 2]
    pivot = median(small, high, middle)
    pivotindex = array.index(pivot)
    array[pivotindex] = array[smallValArr]
    array[smallValArr] = pivot
    i = smallValArr + 1
    for j in range(smallValArr + 1, highValArr):
        if array[j] < pivot:
            temp = array[j]
            array[j] = array[i]
            array[i] = temp
            i += 1

    highEndVal = array[smallValArr]
    array[smallValArr] = array[i - 1]
    array[i - 1] = highEndVal
    return i - 1

def quicksort_median(array, smallIndex, highIndex):
    global medianC
    if smallIndex+ 10  <= highIndex:
        newpivotindex = partition_median(array, smallIndex, highIndex)

        medianC += (highIndex - smallIndex - 1)
        quicksort_median(array, smallIndex, newpivotindex)
        quicksort_median(array, newpivotindex + 1, highIndex)

    else:
        insertion_sortt(array,smallIndex,highIndex)

def insertion_sortt(array,a,b):
    for i in range(a, b):
        j = i
        while j > 0 and array[j] < array[j-1]:
            array[j],array[j-1]=array[j-1],array[j]
            j = j - 1
```

```
def mquick_sort(inputArr):
    quicksort_median(inputArr, 0, len(inputArr))
    return inputArr
```