Aashish Adhikari

CS 519 Analysis, Design, and Implementation of Algorithms Fall 2019
Homework 2

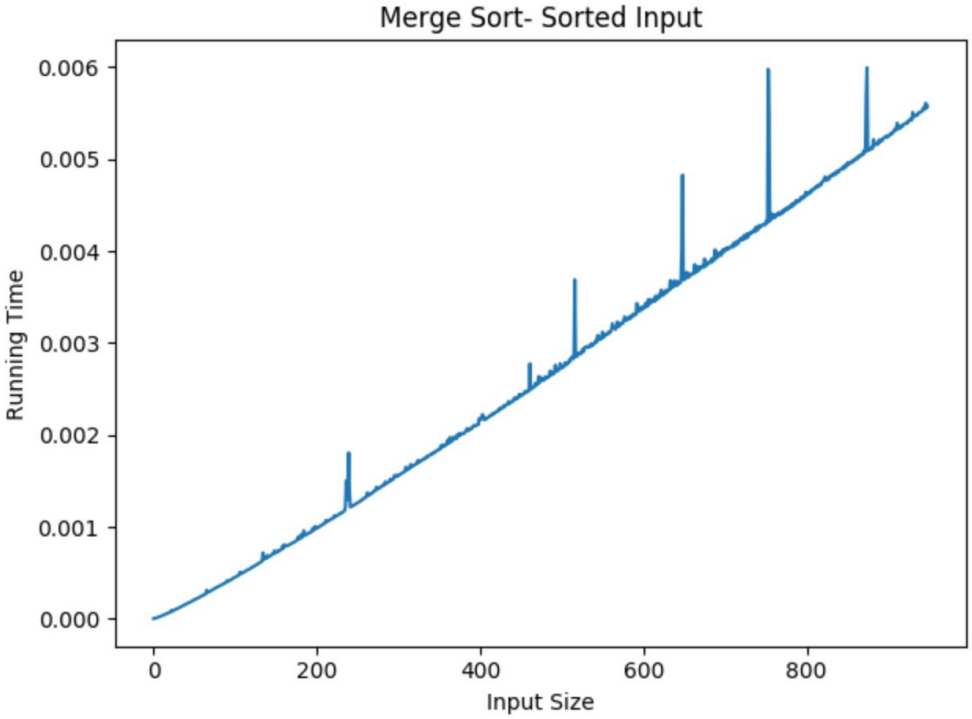
1.Implement a function mergeSort and another function quickSort to sort a set of integers in increasing order.

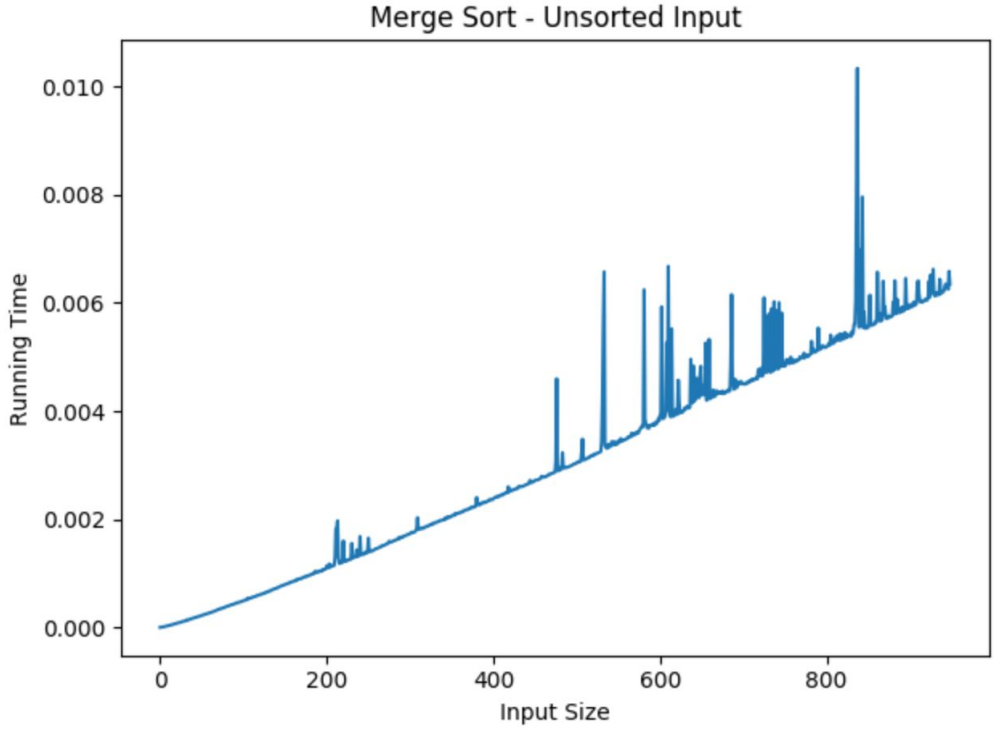
```
def mergeSort(arr):

  if len(arr) == 1:
     return arr

  else:
     left_arr = arr[:int(len(arr)/2)]
     right_arr = arr[int(len(arr)/2):]

     returned_sorted_left_arr = mergeSort(left_arr)
     returned_sorted_right_arr = mergeSort(right_arr)

     return merge(returned_sorted_left_arr, returned_sorted_right_arr)

def merge(left_arr, right_arr):
  merged_arr = []

  while( len(left_arr)> 0 and len(right_arr) > 0):
     if left_arr[0] < right_arr[0]:
        merged_arr.append(left_arr.pop(0))
     else:
        merged_arr.append(right_arr.pop(0))

  while (len(left_arr)>0):
     merged_arr.append(left_arr.pop(0))

  while (len(right_arr)>0):
     merged_arr.append(right_arr.pop(0))

  return merged_arr


def partition(arr, start, end):
  left_border = (start - 1)

  #taking the last element as the pivot
  pivot = arr[end]
```

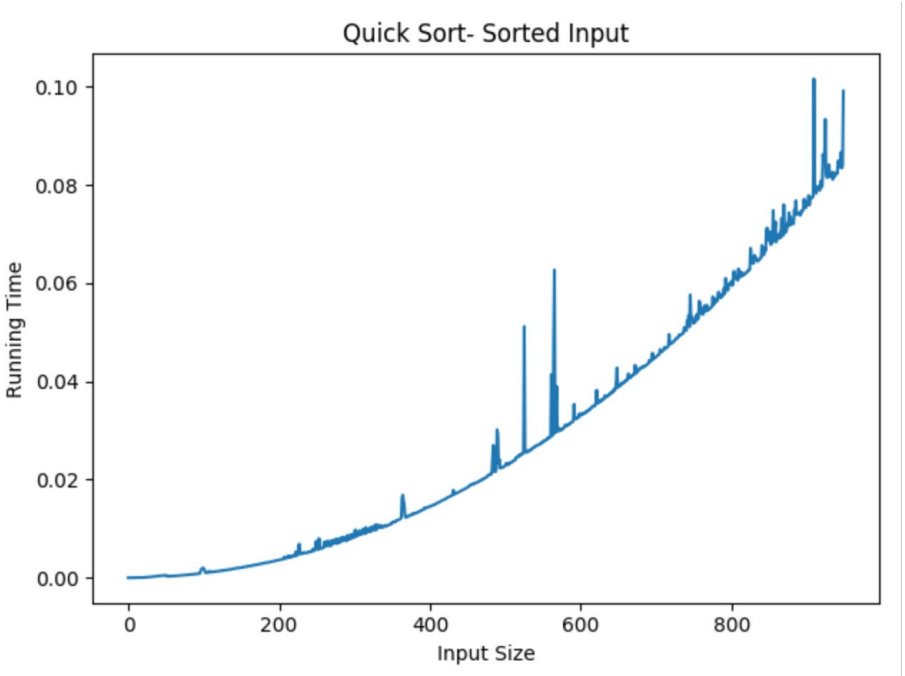
```
    for j in range(start, end):

        if arr[j] < pivot:
            left_border = left_border + 1
            arr[left_border], arr[j] = arr[j], arr[left_border]

    arr[left_border + 1], arr[end] = arr[end], arr[left_border + 1]
    return (left_border + 1)


def quickSort(arr):
    start = 0
    end = len(arr)-1
    #do in-place sorting of quick sort, we do not need any auc=xillary array
    helper_quicksort(arr,start,end)
    return arr


def helper_quicksort(arr, start, end):

    if start < end:
        partition_location = partition(arr, start, end)

        #recursively call helper_quicksort() in-place
        helper_quicksort(arr, start, partition_location - 1)
        helper_quicksort(arr, partition_location + 1, end)
```

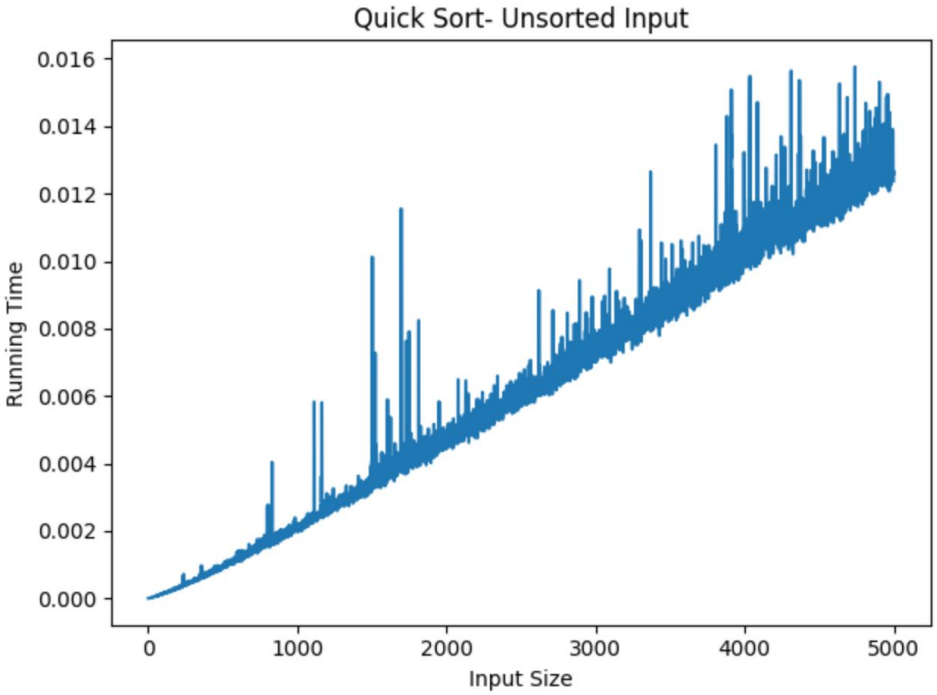
2.Plot the running times of the two algorithms against the size of the input when the input arrays are randomly sorted. What functions best characterize the running times? Do the same when the inputs are already sorted. What functions best characterize the running times now? Discuss the results and their implication.

Note: For each of the cases, I used 950 arrays (unless mentioned otherwise), input varying from array size == 1 element to array size == 950 elements. Each element in the array was randomly sampled from the range [0,100000].


Merge Sort- Sorted Input
Running Time
0.000
0.001
0.002
0.003
0.004
0.005
0.006
0
200
400
600
800
Input Size


Merge Sort - Unsorted Input
Running Time
0.000
0.002
0.004
0.006
0.008
0.010
0
200
400
600
800
Input Size


Quick Sort- Sorted Input
Running Time
Input Size
0.00
0.02
0.04
0.06
0.08
0.10
0
200
400
600
800


Quick Sort- Unsorted Input
Running Time
Input Size
0.000
0.002
0.004
0.006
0.008
0.010
0.012
0.014
0.016
0
1000
2000
3000
4000
5000

RESULTS and IMPLICATIONS:

Merge_sort_sorted_input:
The running time is best represented by O(nlogn). I ran the algorithm for this setting multiple times and the graphs obtained were similar.

Merge_sort_unsorted_input:
The running time is best represented by O(nlogn). I ran the algorithm for this setting multiple times and the graphs obtained were similar.

Regardless of being sorted or not, merge sort still divides the task into half at each level and does n amount of task. Thus, since there are log n levels of tasks to be done, the complexity becomes O(nlogn). **Hence merge sort is an efficient algorithm since it has a time complexity of O(nlogn) in both the cases** which is better than bubble sort or any other algorithm with a time complexity of O(n^2).

Quick_sort_sorted_input:
The running time is best represented by O(n^2) as it can be seen in the figure. I ran the algorithm for this setting multiple times and the graphs obtained were similar. This is because when the elements are sorted in ascending order and the pivot is chosen to be the rightmost element every time rather than a randomized element, quick sort has to do comparatively more comparisons. **This can be relaxed by using randomization.**

Quick_sort_unsorted_input (5000 inputs):
Even though the theoretical running time is best represented by O(n^2), the graph inclines towards the complexity of O(nlogn) which I believe is **because of the random nature of the elements in each array.** When the pivot is chosen at random, it relaxes the time complexity and thus helps us achieve a better complexity than the worst case of O(n^2). I ran the algorithm for this setting multiple times and the graphs obtained were similar. **Also, it should be noted that even if both had performed with O(nlogn), for memory constraints, quick-sort is more effective because of its in-place operations.**

3. The following code implements Bubble Sort which sorts the array a in ascending order. Fill in the assertions which must hold at each of the points indicated below.

Solution

```
def bubbleSort(a):
    swap,j = 1, len(a)
   while swap == 1:
#  Assertion 1:
a) swap == 1, b) j <= len[a]


        swap,j = 0,j-1
        for i in range(j):
#  Assertion 2:
a)i <= j, b) a[i] >= a[i-1] >= a[i-2] >= ….. >= a[1] >= a[0]


            if a[i] > a[i+1]:
                a[i], a[i+1], swap = a[i+1], a[i], 1
# Assertion 3: [after the while-loop finally ends]
a[len(a) -1] >=  a[len(a) -2] >=  ……..>=  a[0]
```

Argue informally that Assertions 1 and 2 hold during the execution and Assertion 3 holds at the end.

Solution:

Assertion 1 :a) swap == 1, b) j <= len[a]

- Proof:
    - Base Case: In the 1st iteration, swap was initialized to 1 and j to len[a]
    - Inductive Hypothesis: Assume these assertions hold for n^{th} iteration.
    - Inductive Step: At $n+1^{th}$ iteration,
        - swap == 1 since it would not have entered this while loop without this condition satisfying

        - j<= len[a] because j has only been decreasing at each iteration.

Assertion 2: a)i <= j, b) a[i] >= a[i-1] >= a[i-2] >= ….. >= a[1] >= a[0]

- Proof:
    - Base Case:
        - i starts always at 0 because of its initialization to the range of value of j which is <= j
        - In the first iteration of the for loop, a[i] is the only element and thus the maximum.
    - Inductive Hypothesis: Let these hold true for i^{th} iteration
    - Inductive Step:
        - At each new iteration, i is incremented by 1 but is still less than the max value possible i.e., j because of how range() works.
        - At each new iteration, we swap a[i+1] with a[i] if a[i] is greater than a[i+1]. Thus, in the beginning of the next for loop, this a[i+1] becomes the new a[i] which is the maximum of all the elements to the left of itself.

Assertion 3: a[len(a) -1] >= a[len(a) -2] >= ……..>= a[0]

- Proof: Since at k^{th} iteration of while loop, the algorithm is pushing the k^{th} largest element to the k^{th} index of the array a, at the end of the while loop, the array will have been sorted when this while loop ends. **Note that we are not trying to show loop invariance here and thus do not require an inductive proof of loop invariance.**

4.Express the running time of the above algorithm in *O*-notation. Justify your answer.

Solution:

The time complexity of this algorithm is O(n^2). Assume the worst case that the elements are sorted in the descending order. In that case, element i beginning at 0 has to compare with every other element . This is O(n). However, since every element is

doing so in its turn (ignoring minor time reduction at each iteration), the upper bound becomes O(n^2) for n such elements.

5.How long did you take for the following tasks?

- Coding
  - Merge sorting took about 30 minutes.
  - Quick sorting took about 3 hours because I could not debug a certain portion of the code in the beginning. I had to rewrite from scratch because I could not trace the bug.
- Get the code to work correctly.
  - The same merge sort algorithm failed to pass the last test case in the beginning but re-running it a couple hours later worked fine. So, may be the server was busy earlier. Quick sort was difficult to debug at first because the in-place sorting worked for partitions but at the end gave jumbled output. As mentioned above, I spent quite some time re-doing it.
- Answer the questions 2-4 above.
  - Generating codes to plot the graphs, then making sure I do not miss assertions and proving them took some time. I ran the codes for plotting multiple times to make sure that I am getting a consistent output. Also, assertion generation is tricky and took some while. Question 4 is very easy.