ASSIGNMENT-3: Sorting

Name : Kalakonda Rohini

Roll number : EE21B066

Question 1 :

We will use a type of induction to solve this problem.

As in simple induction, we have a statement P(n) about the whole number n, and we want to prove that P(n) is true for every value of n. To prove this using strong induction, we do the following:

- The base case. We prove that P(1) is true (or occasionally P(0) or some other P(n), depending on the problem).
- The induction step. We prove that if P(1), P(2), …, P(k) are all true, then P(k+1) must also be true. In this step, the assumption that P(1), …, P(k) are true is called the inductive hypothesis.

Together, these imply that P(n) is true for all whole numbers n.

The critical difference between this and simple induction is that in step 2, we do not assume that only P(k) is true, but P(k) and all earlier P(i) for i<k.

It is convenient if the proof of P(k+1) might depend on not only the previous P(k) but on any earlier P(i).

Pseudocode for Merge Sort :

```
n  - length of array
if n<=1:
      return arr
L - mergesort(arr[0:⌊n/2⌋])
R - mergesort(⌊n/2⌋:n]) return
merge(L,R)
```

We know that the time required **for merge(L,R) is always O(n)** since L and R are always obtained by splitting the original array irrespective of whether n is odd or even.

So if T(n) is time for merge sort to sort an n element array,
$T(n) = T(\lfloor n/2 \rfloor) + T(\lceil n/2 \rceil) + n$

Strong Induction Hypothesis: T(n) ≤ 2n log n for all n ≥ 1

Base Case: n = 1. T(1) = 0 since an array of size1 is already sorted and 2n log n = 0 for n = 1.

Induction Step: Assume T(k) ≤ 2k log k for all k < n and prove it for k = n. Now to

prove T(n) ≤ 2n log n,

$$
\begin{aligned}
T(n) &= T(\lfloor n/2 \rfloor) + T(\lceil n/2 \rceil) + n \\
&\leq 2\lfloor n/2(\lfloor n/2 \rfloor) + 2\lceil n/2 \rceil log \lceil n/2 \rceil + n \\
&\leq 2\lfloor n/2 \rfloor log \lceil n/2 \rceil + 2\lceil n/2 \rceil log \lceil n/2 \rceil + n \\
&\leq 2(\lfloor n/2 \rfloor + \lceil n/2 \rceil) log \lceil n/2 \rceil + n \\
&\leq 2nlog\lceil n/2 \rceil + n \\
&\leq 2nlog(2n/3) + n \qquad (since \lceil n/2 \rceil \leq 2n/3 \\
&\leq 2nlogn + n(1 - 2log3/2) \\
&\leq 2nlogn + n(log2 - log9/4) \\
&\leq 2nlogn \qquad \text{for all } n \geq 2)
\end{aligned}
$$

Thus we have shown that merge sort is Θ(n log n).

Question 2 :

Pseudocode for Quick Sort :

```
Quicksort(arr) :
      if len(arr) <=1:
            return arr
      pivot = arr( ⌊n/2⌋ )
      Partition(arr,pivot)
      Quicksort(L)
      Quicksort(R)
```

where Partition(arr,pivot) divides arr as [L,pivot,R] where L includes all elements in arr with

value less than pivot and R has all elements with values greater than pivot.
Here, the sorting basically happens at the partition step where the elements are rearranged in arr. Choosing the middle element as pivot will make the avg runtime of the quicksort algorithm as O(n log n). But in other cases the run time complexity will be $O(n^2)$. This happens when the pivot chosen is such that it is always the largest or smallest element.

Proof for the above said statement :

Consider a variation of quicksort in which partition is always around the largest element. Then after partition L will always have n-1 and R will 0 elements. So the time for quick sort will be :

T(n) = T(n-1) + T(0) + T(n)

Since the largest element is again the pivot
T(n-1) = T(n-2) + T(0) + T(n-1)

Continuing this logic, we will get

$$T(n) = \frac{n(n-1)}{2} = O(n^2)$$

So, we need to find a sequence such that the elements at $\lfloor n/2 \rfloor$ is always max/min element for time complexity to be $O(n^2)$.

Example sequence :
3 5 7 9 10 8 6 4

Step 1 : n
= 8
$\lfloor n/2 \rfloor$ = 8/2 = 4
Sequence : 3 5 7 9 8 6 4 **10**

Step 2 : n
= 7
$\lfloor n/2 \rfloor$ = 3
Sequence : 3 5 7 8 6 4 **9 10**

Step 3 :
n = 6
$\lfloor n/2 \rfloor$ = 3
Sequence : 3 5 7 6 4 **8 9 10**
Step 3 :
n = 5

$\lfloor n/2 \rfloor$ = 2

Sequence : 3 5 6 4 **7 8 9 10**

Step 4 : n
= 4

$\lfloor n/2 \rfloor$ = 2

Sequence : 3 5 4 **6 7 8 9 10**

Step 5 : n
= 3

$\lfloor n/2 \rfloor$ = 1

Sequence : 3 4 **5 6 7 8 9 10**

Step 6 : n
= 2

$\lfloor n/2 \rfloor$ = 1

Sequence : 3 **4 5 6 7 8 9 10**

Step 7 : n
= 1

$\lfloor n/2 \rfloor$ = 0

Sequence : **3 4 5 6 7 8 9 10**

Question 3 :

Describe and analyze an efficient method for removing all duplicates from a collection A of n elements.

Since there is no constraint for order preservation, we use a modified version of Merge sort which runs in O(n logn). In the usual implementation, if there are duplicates while merging, the duplicates are also added to the merged array. However, here we modify this condition such that the duplicate elements are added once only. Thus the merged array will be a sorted one with no duplicates.

Code:

```
def merge(l, r):

    arr = []

    n = len(l) + len(r)
```

```
    x = 0
    y = 0
    for i in range(n):
        while x < len(l) and y < len(r):
            if l[x] == r[y]:
                arr.append(l[x])
                x += 1
                y += 1

            elif l[x] < r[y]:
                arr.append(l[x])
                x += 1
            else:
                arr.append(r[y])
                y += 1
        while x < len(l):
            arr.append(l[x])
            x += 1
        while y < len(r):
            arr.append(r[y])
            y += 1

    return arr
def merge_sort(arr):
    if len(arr) <= 1:
        return arr
    l = merge_sort(arr[0:len(arr)//2])
    r = merge_sort(arr[len(arr)//2:len(arr)])
    return merge(l, r)
```

```
# some trial cases

arr1 = [2, 2, 2, 2]

arr2 = [-2.3, 9.8, 6.7, -2.3, 6.7, 10]

arr3 = [2, 6, 8, 3, 2, 6, 2, 6, 2, 6]

print("arr1 with duplicates removed: ", merge_sort(arr1))

print("arr2 with duplicates removed: ", merge_sort(arr2))

print("arr3 with duplicates removed: ", merge_sort(arr3))

# for any desired input

arr = [float(x) for x in input("Enter the elements of the array as space separated values: ").split()]

print("The sorted array with duplicates removed: ", merge_sort(arr))
```

Output:

```
arr1 with duplicates removed:  [2]
arr2 with duplicates removed:  [-2.3, 6.7, 9.8, 10]
arr3 with duplicates removed:  [2, 3, 6, 8]
Enter the elements of the array as space separated values: 2 3 45 98 0.001
The sorted array with duplicates removed:  [0.001, 2.0, 3.0, 45.0, 98.0]
```

Question 4 :

It is known that the best case running time for quicksort is obtained when the pivot in each iteration is the median element.

Pseudocode :

```
QuickSort(arr)      :      if
        len(arr)       <=1:
        return arr
        pivot --- median of arr
        Partition(arr,pivot)
        Quicksort(l)
        Quicksort(r)
```

where Partition(arr,pivot) divides arr as [L,pivot,R] where L includes all elements in arr with value less than pivot and R has all elements with values greater than pivot.

The partition always runs in O(n) where n is the size of the array when the pivot is the median. Array L will contain $\lfloor n/2 \rfloor$ and Array R will contain $\lceil n/2 \rceil$ or vice versa.

Therefore, running time of the algorithm is

T(n) = T($\lfloor n/2 \rfloor$) + T($\lceil n/2 \rceil$) + O(n)

It is identical to the relation of Mergesort. Therefore, similar to Question 1, we can prove that quicksort with pivot as median runs in O(n logn).
Thus, we can say the best case algorithm will be :

$\Omega(nlogn)$.

Question 5 :

Implement in python, a bottom-up merge-sort for a collection of items by placing each item in its own queue, and then repeatedly merging pairs of queues until all items are sorted within a single queue.
In this question we are asked to implement a python function that does a bottom-up merge-sort using queues. The function will take in a list and print out the sorted list.

We start by putting each item in the list in its own unique queue and store all these queues in the queue **mainqueue**. Now, we will repeat the following until only one queue remains in **mainqueue** :

- get the first two queues from the front of **mainqueue**
- merge them, and push the merged queue back into **mainqueue**.

When the algorithm terminates, the single queue remaining in **mainqueue** will contain all of the items in sorted order.

To sort two queues :

- compare the front of the two queues and pop the smaller of these two elements from its corresponding queue.
- Conclude by taking, in order, all of the elements remaining in the non-empty queue.
- If the sizes of q1, and q2, are respectively n1 and n2 then merging the two queues will require, in the worst case, n1 + n2 - 1 comparisons. **merge(q1,q2)** will be a function that takes pointers to two sorted queues and returns a pointer to the sorted merged list.

Code:

```
from queue import Queue


def merge(q1,q2):
```

```
    t = Queue(maxsize=1000)
    while (not q1.empty() and not q2.empty()):
        if (q1.queue[0]<q2.queue[0]):
            t.put(q1.get())
        else:
            t.put(q2.get())
    while (not q1.empty()):
        t.put(q1.get())
    while (not q2.empty()):
        t.put(q2.get())
    return t
def perm(list):
    mainqueue = Queue(maxsize=1000)
    for i in list:
        temp = Queue(maxsize=1000)
        temp.put(i)
```

```
        mainqueue.put(temp)
    while (mainqueue.qsize()!=1):
        q1 = mainqueue.get()
        q2 = mainqueue.get()
        mainqueue.put(merge(q1, q2))
    sortedq = Queue(maxsize=1000)
    sortedq = mainqueue.get()
    while (not sortedq.empty()):
        print(sortedq.get(), end=" ")
list = [5,98,45,67,23,14]
perm(list)
```

```
Output: 5 14 23 45 67 98
```