Assignment 3

Name: Akash Reddy A
Roll Number: EE17B001
Date: Nov 5, 2020

Question 1

Let $T(n)$ be the time taken by the merge sort algorithm.

In general, at each split, the subarrays are divided into size $\lfloor n/2 \rfloor$ and $\lceil n/2 \rceil$. The relationship between the times taken for the parent array and the subarrays in the next level of the recursion tree is:

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

where c is some positive constant (the term that accounts for the comparisons that occur during merging).

We can use mathematical induction to show that $T(n) = O(n \log n)$ or equivalently, $T(n) < kn \log n$.

Let us assume that $\forall i < n, T(i) < ki \log i$. We show that it is true for $i = n$ as well. Substituting this assumption in the above equation,

$$T(n) = T(\lfloor n/2 \rfloor) + T(\lceil n/2 \rceil) + cn < k\lfloor n/2 \rfloor \log(\lfloor n/2 \rfloor) + k\lceil n/2 \rceil \log(\lceil n/2 \rceil) + cn$$

Since $\lfloor n/2 \rfloor < \lceil n/2 \rceil$,

$$\begin{aligned} T(n) &< k\lfloor n/2 \rfloor \log(\lceil n/2 \rceil) + k\lceil n/2 \rceil \log(\lceil n/2 \rceil) + cn \\ &< kn \log(\lceil n/2 \rceil) + cn \end{aligned}$$

For large values of n (in fact, for any n > 1), we have $\lceil n/2 \rceil < an$ where $0.5 < a < 1$.

Therefore,

$$\begin{aligned} T(n) &< kn \log(an) + cn \\ &< kn \log(n) + [c - k \log(1/a)]n \end{aligned}$$

If k is chosen such that $[c - k \log(1/a)] < 0$, then

$$T(n) < kn \log(n)$$

holds.

Equivalently,

$$T(n) = O(n \log(n))$$

holds.

Question 2

QuickSort does best - $O(n \log n)$ - when the pivot is picked to be nearly the median of the sorted sequence each time. This splits the sequence into balanced halves in each recursion - the recursion tree remains balanced, and the height of the recursion tree will be $\log n$.

On the contrary, QuickSort does worst - $O(n^2)$ - when the recursion tree is completely imbalanced. This happens when the pivot is picked to be either the largest or the smallest value. When this happens, the next subsequence will be only 1 element smaller than the previous sequence. Therefore, the height of the recursion tree will be n, and this will lead to the worst complexity.

Therefore, to obtain $\Omega(n^2)$, we need sequences that cause QuickSort to perform the worst when the element at index $\lfloor n/2 \rfloor$ is selected as the pivot. **This means that the pivot, the element at index $\lfloor n/2 \rfloor$ (the middle element), must be the maximum or minimum value of the subsequence at every step of the recursion tree.**

Let us consider that our pivot always has to be the **maximum** of the subsequence.

This is possible if the sequence satisfies a few properties:

- In any subsequence during the QuickSort, if the number of elements is even, the maximum needs to be the right-side element out of the middle two elements, and not the left-side one.
- In any subsequence during the QuickSort, if the number of elements is odd, the maximum needs to be the exact middle element.
- At any step of the QuickSort, the number of elements in the subsequence changes from even to odd or odd to even, after the pivot is excluded. Once the subsequence changes, we have to make sure that the maximum element of the new subsequence ends up at the pivot position.

- The sequence has to be unimodal with the maximum element(s) located at the middle of the sequence.
- In addition, another rule has to be satisfied: The elements must follow a descending order beginning at the centre and slowly moving away from the centre in an alternating fashion between the two halves of the sequence (starting from a step to the right side).

  For clarity, below is a diagrammatic representation of an 8-element array, with the cells labelled in required alternating descending order of elements. Here, 1 is the largest element and 8 is the smallest. Notice that 1 is the right-side element of the middle two. Also, notice the alternating order of elements.

8 6 4 2 1 3 5 7

In order to demonstrate why this last rule is necessary, we can compare the execution on QuickSort on three unimodal sequences - first one satisfies the last rule, second one satisfies it partially, and third one does not.

- Consider three 8-element unimodal sequences with pivot highlighted (index $\lfloor n/2 \rfloor$ = index 4 in the zero-indexed sequence):

  1 4 8 13 **15** 9 7 2

  1 2 8 16 **19** 3 7 6

  1 2 3 5 **9** 8 7 6

  All three have the maximum element at the right place. However, only one sequence follows the right order.

- After one recursion, the pivot moves to the rightmost position in all 3 sequences:

  1 4 8 13 9 7 2 **15**

  1 2 8 16 3 7 6 **19**

  1 2 3 5 8 7 6 **9**

  and the "left sub-sequence" is all the elements except the pivot, while the right sub-sequence is empty.

- Now, in the new sub-sequences, the new pivots, the index $\lfloor n/2 \rfloor$ elements are highlighted:

  1 4 8 **13** 9 7 2

  1 2 8 **16** 3 7 6

  1 2 3 **5** 8 7 6

  The new pivot is the maximum element for sequences 1 and 2, but NOT for 3. Therefore, 3 is bound to have a faster QuickSort from here on.

- Let us continue with sequences 1 and 2. The pivot moves to the right extreme, and the new sub-sequences 1 and 2 are (element at index $\lfloor n/2 \rfloor$ highlighted) :

  1 4 8 **9** 7 2

  1 2 8 **3** 7 6

In sequence 1, the new pivot is once again the maximum element. But the new pivot in sequence 2 is not the maximum. In the previous iteration, the new pivot was the maximum index because it has some PARTIAL alternating descending order.

Therefore, sequence 2 is bound to have a faster QuickSort than sequence 1 from here on.

- We can continue the analysis to see that the new pivots for sequence 1 (which follows all the described rules) are always the maximum value in the subsequence, leading to the slowest - $\Omega(n^2)$ QuickSort.

Conclusion: For $\Omega(n^2)$ time, the array needs to be unimodal with the maximum (or minimum) at the centre, and the elements must follow a descending (or ascending) order starting from the centre, in alternating fashion moving outward (starting with a step to the right in the case of zero-indexed sequences)

Question 3

An efficient method is to:

- Traverse through the sequence in $O(n)$
- Look for the element in an initially empty hash map (dictionary or set in Python) in $O(1)$ for each element (because hash functions make accessing easy, in $O(1)$)
- If the element is not already present, add the element to the hash map in $O(1)$.
- Eventually, return the hash map/dictionary keys - this will be the collection with no duplicates.

  This method adds up to a time complexity of $O(n) + nO(1) + nO(1) = O(3n) = O(n)$. It has been coded below.

```
# Function to implement the above algorithm

def remove_duplicates(arr):
    # Initialising an empty hash map/dictionary
    elm_dict = {}

    # Looping through the elements in the input collection
    for i in arr:
        # Checking if the element is present in the hash map/dictionary
        if i not in elm_dict:
            # Adding the element to the hash map/dictionary
            elm_dict[i]=1

    return list(elm_dict.keys())    # Returning the de-duplicated collection
```

```
# Arbitrary input sequence - can be modified
input_array = [1, 3, 3, 2, 1, 2, 5, 5 ,7, 4, 8, 5, 3]
```

```
3
4 # Printing de-duplicated array
5 print("The array with no duplicates = ", remove_duplicates(input_array))
```

```
The array with no duplicates =  [1, 3, 2, 5, 7, 4, 8]
```

Question 4

With an array of n integers belonging to range $[0, n]$, counting sort is a great algorithm that sorts in $O(n)$ time. This is a non-comparison based algorithm in which the following steps happen:

- Consider the following array:

  4 2 2 8 3 3 1 0

- The maximum integer max is found and a new array of length $max + 1$ is initialised. Here, $max = 8$, so the length of the array is 9.

  0 0 0 0 0 0 0 0 0

- Then, the count of each element in our original sequence is stored at the respective index in the new array.

  1 1 2 2 1 0 0 0 1

  and this count array is converted into a cumulative count array:

  1 2 4 6 7 7 7 7 8

- After initialising an output array with the same size as the input sequence, each element in the input sequence is taken, and the (value of the element's cumulative count - 1) is the index of the element in the output array. After an element has been placed in the output array, its entry in the cumulative count array is reduced by 1.
- For example, the first element in the original sequence is 4. The corresponding cumulative count is 7, so 4 is placed at the 6th index in the output array, and the new cumulative count of 4 is $7 - 1 = 6$.
- Once this is repeated for all elements in the input sequence, the output sequence contains the sorted array:

  0 1 2 2 3 3 4 8

- Since this algorithm involves $O(n)$ to find the maximum element, $O(n)$ to generate the cumulative count array, and $O(n)$ to .

However, the counting sort algorithm, when applied to the integer range $[0, n^2 - 1]$ will run in $O(n^2)$ time because the cumulative count array will take $O(n^2)$ for computation and it will

become the rate-determining step. This is worse than the $O(n \log n)$ time of comparison-based sorting methods like QuickSort and Merge Sort.

Radix Sort is an algorithm that can be used to run the sorting of the given range of integers in $O(n)$.

Radix sort arranges the elements by first sorting them according to the units place, next according to the tens place, then the hundreds place, and so on. It uses counting sort for each of these sorts. Therefore, it does m counting sorts, where m is the number of places/digits in the largest number.

In the integer range $[0, n^2 - 1]$, the highest number $n^2 - 1$ has a maximum of $2d$ digits where d is the number of digits in n.

Since the radix sort will have to do $2d$ counting sorts in the worst case, the complexity of radix sort is $O(2d(3n + b))$, where b = base of the place value system (or the number of possible digits in each place). For example, in the decimal system, there are $b = 10$ possible integers for each place.

Now $d = log_b(n)$, therefore the complexity reduces to $O(n \log_b(n))$. **This complexity can be made $O(n)$ when the base value $b = n$.**

Therefore, radix sort with an $n-$ary place value system to represent the numbers has complexity $O(n)$. Any number in integer range $[0, n^2 - 1]$ can be represented using 2 digits in this place value system, since $d = 1$ in this system. Therefore, a consistent number of 2 counting sorts of $O(n)$ are required, which makes the radix sort $O(2n) = O(n)$.

```
# Importing required packages

import math
import numpy as np

# Function to find a certain place value digit of a number
# in a certain place value system
def digit_generator(num, base, pwr):

    # num - Input number
    # base - Base of the place value system
    # pwr - The index / place of the "digit" required in the new place system

    # Calculating the required digit
    digit = ((num%(base**(pwr+1))) - (num%(base**pwr)))//(base**pwr)
    return digit

# Function to perform counting sort on a sequence
def counting_sort(seq, base, pwr):
```

```
    # seq - The sequence of numbers to be sorted
    # base - The base of the place value system, defines the size of count array
    # pwr - The current index/place of the "digit" required in the new place
    #     system, used as a parameter for the digit_generator function as needed

    count = [0]*base        # count array initialised

    n = len(seq)     # value of n taken from the length of n-integer sequence
    output = [0]*n   # output array initialised with same size as input sequence

    # Filling the count array with counts of each "digit"
    for i in range(n):
        count[digit_generator(seq[i], base, pwr)] += 1

    # Generating the cumulative count function
    cum_count = np.cumsum(count)

    # Looping through the input sequence, the element is filled into the output
    #  array at the index of cum_count - 1 of the corresponding digit.

    # Subsequently, the value of cum_count itself is decreased by 1,
    #  as described in the textual explanation before the code.

    i = n - 1
    while i>=0:
        output[cum_count[digit_generator(seq[i], base, pwr)] - 1] = seq[i]
        cum_count[digit_generator(seq[i], base, pwr)] -= 1
        i -= 1
    return output

# Function to perform Radix sort
def radix_sort(seq):

    # seq - The input sequence of length n, with integers in range [0, n**2-1]

    base = n = len(seq)

    # The max power/place value of place value system with the required base,
    #  calculated using the fact that n**2-1 is the largest integer possible
    max_pow = math.ceil(math.log(n**2-1)/math.log(base))

    # A copy of the original sequence is generated,
    #  to store the sorted array after each counting sort
    sorted_seq = seq.copy()

    # Iterating from units place (k=0) to the max place (k=max_pow)
    for k in range(max_pow):
        # Counting sort done repeatedly on the array starting from units place
        sorted_seq = countingSort(sorted_seq, base, k)

    # Radix sorted array returned
    return sorted_seq


# Input sequence given to the Radix sorting algorithm- arbitrary, can be changed
```

```
input_seq = [18, 32, 4, 3, 14, 13, 9, 8]

# Printing the Radix-sorted array
print("The radix sorted sequence is:", radixSort(input_seq))
```

```
The radix sorted sequence is: [3, 4, 8, 9, 13, 14, 18, 32]
```

Question 5

Let $T(n)$ be the time taken by the merge sort algorithm.

For the best case in QuickSort, at each split, the pivot is placed nearly at the middle, so the subarrays obtained are of size $\lfloor n/2 \rfloor$ and $\lceil n/2 \rceil$. The relationship between the times taken for the parent array and the subarrays in the next level of the recursion tree is:

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

where c is some positive constant (the term that accounts for the comparisons that occur during merging).

We can use mathematical induction to show that $T(n) = \Omega(n \log n)$ or equivalently, $T(n) > kn \log n$.

Let us assume that $\forall i < n, T(i) > ki \log i$. We show that it is true for $i = n$ as well. Substituting this assumption in the above equation,

$$T(n) = T(\lfloor n/2 \rfloor) + T(\lceil n/2 \rceil) + cn > k\lfloor n/2 \rfloor \log(\lfloor n/2 \rfloor) + k\lceil n/2 \rceil \log(\lceil n/2 \rceil) + cn$$

Since $\lceil n/2 \rceil > \lfloor n/2 \rfloor$,

$$\begin{aligned} T(n) &> k\lfloor n/2 \rfloor \log(\lfloor n/2 \rfloor) + k\lceil n/2 \rceil \log(\lfloor n/2 \rfloor) + cn \\ &> kn \log(\lfloor n/2 \rfloor) + cn \end{aligned}$$

For large values of n (in fact, for any n > 1), we have $\lfloor n/2 \rfloor > an$ where $0 < a < 0.5$.

Therefore,

$$\begin{aligned} T(n) &> kn \log(an) + cn \\ &> kn \log(n) + [c - k \log(1/a)]n \end{aligned}$$

If k is chosen such that $[c - k \log(1/a)] > 0$, then

$$T(n) > kn \log(n)$$

holds.

Equivalently,

$$T(n) = \Omega(n \log(n))$$

holds for the best-case QuickSort time complexity.
