EE4371 - Assignment 3

EE19B094

Manvar Nisharg

This PDF contains the solutions to questions from assignment 3 and explanations to the source code submitted along.

Question 1

Show that the running time of the merge-sort algorithm on n-element sequence is $O(nlogn)$, even when n is not a power of 2.

Answer 1

We need to prove that $\underline{T(n) \leq k_1 \cdot nlogn + k_2 \cdot n}$

We will use the method of induction.

The recursive formula for merge sort is,

$$T(n) = T([n/2]) + T(n - [n/2]) + k \cdot n \quad where\ k\ is\ a\ computational\ constant.$$

As the base case to induction, let $n = 1 \implies T(1) = c$

Now, consider two cases

Case 1: n is even

$$\implies T(n) = 2 \cdot T(n/2) + k \cdot n$$
$$\implies T(n) \leq 2 \cdot [k_1 \cdot (n/2) \cdot log(n/2) + k_2 \cdot (n/2)] + k \cdot n$$
$$\implies T(n) \leq k_1 \cdot nlogn + (k_2 + k - k_1 log2) \cdot n$$
$$\implies T(n) \leq c_1 \cdot nlogn + c_2 \cdot n$$
$$Hence\ proved.$$

Case 2: n is odd

$$\implies T(n) = T((n-1)/2) + T((n+1)/2) + k \cdot n$$
$$\implies T(n) \leq [k_1 \cdot (n - 1/2) \cdot log(n - 1/2) + k_2 \cdot (n - 1/2)] + [k_1 \cdot (n + 1/2) \cdot log(n + 1/2) + k_2 \cdot (n + 1/2)] + k \cdot n$$
$$\implies T(n) \leq k_1 \cdot [(n-1)/2 \cdot log(n-1) + (n+1)/2 \cdot logn(n+1)] - k_1 \cdot nlog2 + k_2 \cdot n + k \cdot n$$

Thus we need a constant c_1 such that,

$$(n-1)/2 \cdot log(n-1) + (n+1)/2 \cdot log(n+1) \leq c_1 \cdot nlogn$$
$$\implies ((n-1)/n)^{n-1} \cdot ((n+1)/n)^{n+1} \leq n^{2n(c_1 - 1)}$$

Plotting LHS we see that $LHS \leq 4$. So, in our case where we have $n > 2$,

$$we\ need\ 4 \leq n^{2n(c_1-1)}$$
$$\implies c_1 = 2\ will\ satisfy\ the\ condition.$$
$$\therefore for\ c_1 = 2 \implies T(n) \leq 2 \cdot k_1 \cdot nlogn + (k_2 + k - k_1 log2) \cdot n$$
$$\implies T(n) \leq p_1 \cdot nlogn + p_2 \cdot n$$
$$Hence\ proved$$

Question 2

Consider a modification of the deterministic version of the quick-sort algorithm where we choose the element at index ⌊n/2⌋ as our pivot. Describe the kind of sequence that would cause this version of quick-sort to run in $\Omega(n^2)$ time.

Answer 3

Firstly, let us define a partition algorithm for this problem

$$\underline{Partition\ algorithm}$$

We start by moving the pivot element out of the way, by swapping it with the leftmost element in the array segment.

Image_1

Now the idea is to gradually work towards the middle, accumulating elements less than the pivot on the left and elements greater than the pivot on the right end of the segment (excluding the pivot itself). For this purpose we introduce two indices, left and right. We start them out as lower + 1 (to avoid the stashed-away pivot) and upper .

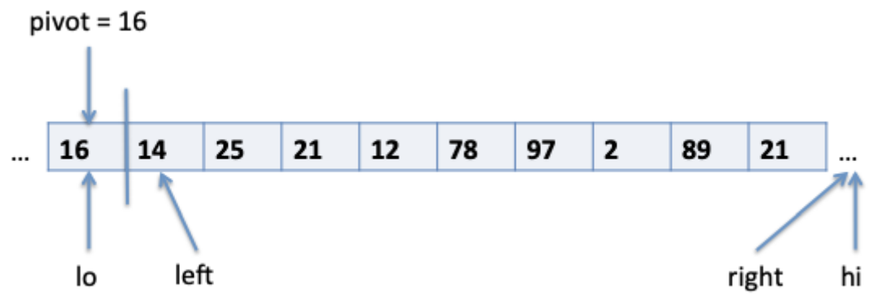


Image_2

Since 14 < pivot, we can advance the left index: this element is in the proper place.

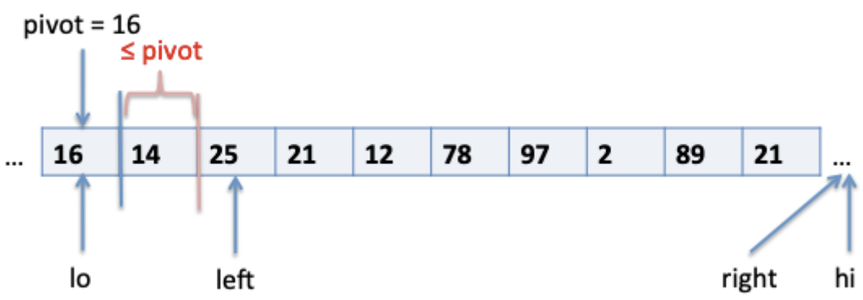


Image_3

At this point, 25 > pivot, it needs to go on the right side of the array. If we put it on the extreme right end of the array, we can then say that it is in it's proper place. We swap it into A[right – 1] and decrement the right index.

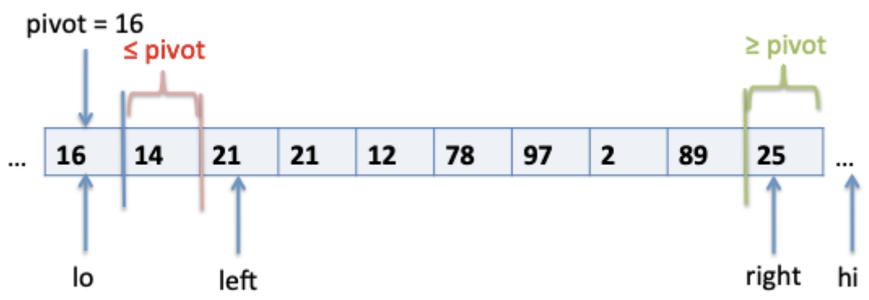
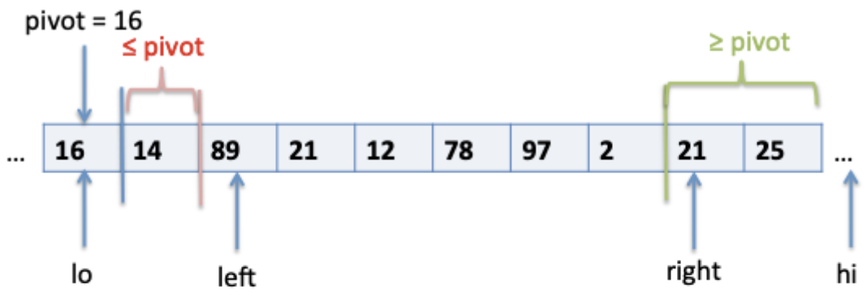


Image_4

In the next two steps, we proceed by making swaps. First, we decide that the 21 that is currently at left can be properly placed to the left of the 25, so we swap it with the element to the left of 25. Then, we have 89 at A[left], and so we can decide this is well-placed to the left of that 21.



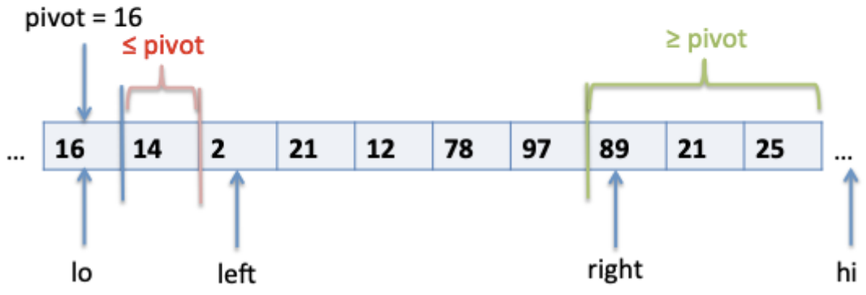


Image_5

Let's take one more step: 2 < pivot, so we again just decide that the 2 is fine where it is and increment left.

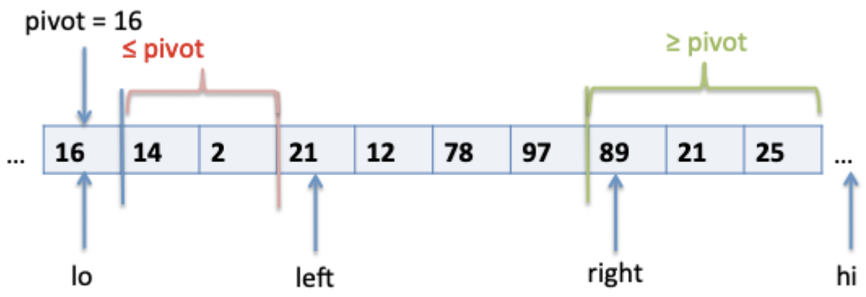


Image_6

Thus using this algorithm we get the partitoned array.

$$Condition\ for\ worst\ case$$

We know that the recursive equation for quicksort is,

$$T(n) = T(L) + T(R) + \Theta(n)$$

Now, consider the case where $L = 0\ and\ R = n - 1$ i.e. pivot is either the maximum or the minimum element,

$$T(n) = T(0) + T(n-1) + \Theta(n) \quad (where\ T(0) = 0)$$

$$On\ expanding,\ we\ get\ T(n) = (\sum_{1}^{n} \Theta(n)) + k$$

$$\therefore T(n) = \Theta(n^2)$$

$$\implies T(n) = \Omega(n^2)$$

Thus, for every partition the pivot element chosen should be the maximum or minimum element in that partition to get $\Omega(n^2)$ complexity.

Construct exapmles

So, we know that for the given complexity to occur the pivot must always be the greatest/least value of the array.

So, in the beginning let the largest element be at the position [n/2], now after applying partition algorithm on this array using the partition algorithm we defined above, we will find that the pivot would be shifted to the last element and relative ordering of other elements would not change at the end of the algorithm. Thus, now the next largest element should be at the middle of that partition i.e. at position [(n-1)/2] as that would be the pivot for that partition and so on for next partitions.

Consider the following example (1 indexed), $array = 3,\ 5,\ 7,\ 9,\ 10,\ 8,\ 6,\ 4,\ 2,\ 1$. If we go through this example then,

$$
\begin{aligned}
3,\ 5,\ 7,\ 9,\ 10,\ 8,\ 6,\ 4,\ 2,\ 1 \quad & pivot\ =\ 10 \\
3,\ 5,\ 7,\ 9,\ 8,\ 6,\ 4,\ 2,\ 1,\ |\, 10 \quad & pivot\ =\ 9 \\
3,\ 5,\ 7,\ 8,\ 6,\ 4,\ 2,\ 1,\ |\, 9,\ 10 \quad & pivot\ =\ 8 \\
3,\ 5,\ 7,\ 6,\ 4,\ 2,\ 1,\ |\, 8,\ 9,\ 10 \quad & pivot\ =\ 7 \\
3,\ 5,\ 6,\ 4,\ 2,\ 1,\ |\, 7,\ 8,\ 9,\ 10 \quad & pivot\ =\ 6 \\
3,\ 5,\ 4,\ 2,\ 1,\ |\, 6,\ 7,\ 8,\ 9,\ 10 \quad & pivot\ =\ 5 \\
3,\ 4,\ 2,\ 1,\ |\, 5,\ 6,\ 7,\ 8,\ 9,\ 10 \quad & pivot\ =\ 4 \\
3,\ 2,\ 1,\ |\, 4,\ 5,\ 6,\ 7,\ 8,\ 9,\ 10 \quad & pivot\ =\ 3 \\
2,\ 1,\ |\, 3,\ 4,\ 5,\ 6,\ 7,\ 8,\ 9,\ 10 \quad & pivot\ =\ 2 \\
1,\ |\, 2,\ 3,\ 4,\ 5,\ 6,\ 7,\ 8,\ 9,\ 10 \quad & pivot\ =\ 1 \\
|\, 1,\ 2,\ 3,\ 4,\ 5,\ 6,\ 7,\ 8,\ 9,\ 10 \quad & Array\ is\ sorted
\end{aligned}
$$

Thus, here we see that the pivot for any partition is always the greatest element.

If we want, we can generalise this type of sequence, although there would be many other sequences too.
Let the size of array be n and let it be even,

For i = 1 to (n/2)-1, let the value at index = i be be $(2 \cdot i\ +\ 1)$
For i = (n/2), let the value at index = i be n
For i = (n/2)+1 to n-1, let the value at index = i be $(2 \cdot (n - i))$
For i = n, let the value be 1

Also, note that a sequence which runs in $\Omega(n^2)$ time for one partition algorithm might not run for another. Thus this example is limited to this partition algorithm only.

Question 3

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

Answer 3

There are two ways to go about this problem with different time complexities. In one case, we use dictionaries which is an abstract data type and is complex to be implemented from scratch and therefore it's inbuilt implementation is used, whereas in the other method we use sorting and therefore all it's aspects are implemented in the code from scratch.

Method 1 - Using sorting

In [ ]:

```
"""*****************************************************************************************************
PURPOSE : Remove duplicates from a sequence of numbers using sorting (Assign-3 Q3)
AUTHOR : Manvar Nisharg (EE19B094)
INPUT : Sequence of numbers
OUTPUT : Sequence of numbers with duplicates removed in sorted and original form
NOTE : Please use python 3.x as several functions used as only compatible with python 3 or later versions
*****************************************************************************************************"""

from array import array #Importing necessary files

"""
Function to merge two sorted subarrays into one

:param arr: The array whose two subarrays are to be merged
:param type: list
:param start: The starting index in arr from where elements are to be merged(i.e. starting index of first subarray)
:param type: int
:param mid: The midpoint index in arr (i.e. index where two subarrays are to be divided)
:param type: int
:param end: The endpoint index in arr till where elements are to be merged(i.e. ending index of second subarray)
:param type: int
"""
def merge(arr, start, mid, end):
    #size of arrays to be merged
    size_left = mid - start + 1
    size_right = end - mid


    ##Initalize temporary arrays and copy data from original arrays to temporary arrays
    temp_left = arr[start : start + size_left]
    temp_right = arr[start + size_left : end+1]


    #Pointers to current elements in all three arays
    i = 0     # Initial index of first temporary array
    j = 0     # Initial index of second temporaary array
    k = start     # Initial index of merged array

    #Comparing elements and adding them into the original array until end of any one array is reached
    while i < size_left and j < size_right :
        if temp_left[i] <= temp_right[j]:
            arr[k] = temp_left[i]
            i += 1
        else:
            arr[k] = temp_right[j]
            j += 1
        k += 1

    # Copy the remaining elements of temp_left array if any, into the original array
    while i < size_left:
        arr[k] = temp_left[i]
        i += 1
        k += 1

    # Copy the remaining elements of temp_right array if any, into the original array
    while j < size_right:
        arr[k] = temp_right[j]
```

```
        j += 1
        k += 1


"""
Function to sort a array using merger sort algorithm

:param arr: The array which is to be merged
:param type: list
:param start: The index of first element to be sorted
:param type: int
:param end: The index of last element to be sorted
:param type: int
"""
def mergeSort(arr,start,end):
    if start >= end:
        return

    mid = (start+(end-1))//2

    # Dividing array into two halves and sorting both the halves first
    mergeSort(arr, start, mid)
    mergeSort(arr, mid+1, end)
    # Merging the already sorted arrays into one
    merge(arr, start, mid, end)



"""
Function to return index of a number in a sorted 2D array using binary search algorithm (w.r.t. first coloumn)

:param arr: The arr in which number is to be searched
:param type: list
:param number: The number to be searched
:param type: float

:return: index of number in array
"""
def binary_search(arr,number):
    #Base Cases
    if(len(arr)==0):
        return
    elif(len(arr)==1):
        return 0

    #Initalize pointers to beginning and end of array
    start = 0
    end = int(len(arr)-1)

    while start <= end:

        mid = int((start + end)/2)  #Middle index of active window of search

        if(arr[mid][0] == number):     #If number at mid is equal to 'number' then return mid
            return mid
        elif(arr[mid][0] > number):  #Else if number at mid is greater than 'number' then change end pointer to mid-1
            end = mid - 1
        else:    #Else if number at mid is less than 'number' then change start pointer to mid+1
            start = mid + 1

    return -1   #If number is not found return -1


"""
Call function to remove duplicates from a sequence of numbers

:input: Sequence of numbers
:output: Sequence of numbers with duplicates removed in sorted and original form
"""
def remove_duplicates():
    print("\nTo remove duplicates from a sequence of numbers :")

    sequence = input("Enter sequence of numbers seperated by space : ")     #Storing sequence of numbers as string (sequence)
    list_of_numbers_original = sequence.split()  #Splitting sequence string into individual numbers with space as delimitter in a list (list_of_numbers_original)

    if len(list_of_numbers_original)==0:
        print()
        print("No sequence was given as input.")
        return

    list_of_numbers_original = list(map(float, list_of_numbers_original))   #Converting numbers stored as string to float
    list_of_numbers = list_of_numbers_original.copy()   # Making a copy of original list for processing

    mergeSort(list_of_numbers,0,len(list_of_numbers)-1)     # Sorting the list of numbers using merge-sort

    #Initalize an empty 2D array whose first coloumn will contain all the sorted numbers once and
    #second coloumn contains if that number has already been printed or not (0 for NO and 1 for YES)
    #so initially 0 for all
    #This array will be used later to print sequence in original order
    binary_array_for_numbers = []



    #Prining sorted list so that one number is printed only once

    print("The sequence with duplicates removed in sorted order is :")
    current = None   #Initalize variable which stores value of active number


    #Looping through list from index 1 to end
    for i in range(0,len(list_of_numbers)):
        if list_of_numbers[i]==current:    #If number is already encountered, skip
            continue
        #else the number is encountered for the first time print it and change the value of current to that value
        print(list_of_numbers[i],end = " ")
        current = list_of_numbers[i]
        #Adding number into binary_array_for_numbers array as it is encountered for first time
        temp = []
        temp.append(list_of_numbers[i])
        temp.append(0)
        binary_array_for_numbers.append(temp)

    print()
    print()


    #Printing sequence with duplicates removed in original order

    print("The sequence with duplicates removed in original order is :")

    #Looping through the numbers in the original list given by user
    for i in range(len(list_of_numbers_original)):
        index = binary_search(binary_array_for_numbers,list_of_numbers_original[i])  #Finding index of that number in binary_array_of_numbers using binary search
        if(binary_array_for_numbers[index][1] == 1):    #If the value of second coloumn is 1 for that index that means number has alrady been printed
            continue
        else:
            print(binary_array_for_numbers[index][0],end = " ")     #Else print the number and change value of second coloumn to 1
            binary_array_for_numbers[index][1] = 1


remove_duplicates() #Calling function to remove duplicates
```

Output for some test case

```
To remove duplicates from a sequence of numbers :
Enter sequence of numbers seperated by space : 2.1 -4.5 1.0 2.1 8.9 8.9 7.7

The sequence with duplicates removed in sorted order is :
-4.5 1.0 2.1 7.7 8.9

The sequence with duplicates removed in original order is :
2.1 -4.5 1.0 8.9 7.7
```

Algorithm

The algorithm can be divided into two parts. One is implemeted using sort by using merge sort to print the sequence in sorted order. And another is binary search to print the sequence in original order. In this we take advantage of the fact that we sorted the sequence to implement the first part.
Assumption : All the elements are floating point numbers.

$\underline{Sorted\ Order}$:

In this we use the conventional merge sort algorithm to sort the numbers first. Functions in the code to implement this are :

- **mergeSort()**: Recursive function to implement merge sort. This function divides the input array into two parts, sorts them using recursion and then finally join them using the merge() function.
- **merge()**: Function that merges two sorted arrays into one. It uses two pointer method where one pointer points to one element in each array and compare them and add the smaller number to the result array and increment that pointer untill all elements are added.

So, we first sort the list of numbers using merge sort. Now, we traverse the sorted array and print the first appearance of each number and, as the array is sorted we need to traverse only once. Now, we already know that merge sort takes $O(nlogn)$ complexity. And then traversing the sorted array once takes $O(n)$ time. Thus, overall complexity of this part is $O(nlogn)\ +\ O(n)\ =\ \underline{O(nlogn)}$.

Also, we know merge sort takes up $\underline{O(n)}$ space complexity.

$\underline{Original\ Order}$:

In this we use binary search on the already sorted array in the first part to implement the algorithm. Only function for this part is :

- **binary_search()**: This function takes a 2D array and the number to be searched (x). It then uses conventional binary search algorithm to find the number in the first coloumn of the 2D array.
  Overview of the binary search algorithm is :
  1. Compare x with the middle element.
  2. If x matches with the middle element, we return the mid index.
  3. Else if x is greater than the mid element, then x can only lie in the right (greater) half subarray after the mid element. Then we apply the algorithm again for the right half.
  4. Else if x is smaller, the target x must lie in the left (lower) half. So we apply the algorithm for the left half.

  As we can see we reduce the size of array by half each time so binary search takes O(nlogn) time to execute.

Now to implement the algorithm we traverse the original sequence and print a number we encounter that number for the first time else skip.
To implement that we keep a 2D array (binary_array_for_numbers) whose first numbers contain all the numbers once in sorted order and second coloumn contains 0 if we haven't encountered the number yet and 1 if we have. We create this array while we traverse the sorted array in the previous part, every time we print a number in that part we also add the number in binary_array_for_numbers with initial encounter value as 0. Now, as we traverse the original sequence we use binary search to find that number in binary_array_for_numbers and check if we have encounter it before or not. If no print it, change it's encounter value to 1 else skip the number.
As we use binary search for every number in the original sequence the algorithm takes up $\underline{O(nlogn)}$ time as well.

Also, as we use a 2D array, space complexity of this part is $\underline{O(n)}$.

So, both the part of the codes take up $O(nlogn)$ time therefore,

$$the\ overall\ time\ complexity\ of\ the\ code\ is\ \underline{O(nlogn)}$$

Also, both the part of code takes up $O(n)$ space complexity therefore,

$$the\ overall\ space\ complexity\ of the\ code\ is\ \underline{O(n)}$$

Method 2 - Using python dictionary

In [ ]:
```
"""*********************************************************************************************************
PURPOSE : Remove duplicates from a sequence of numbers using python dictionaries (Assign-3 Q3)
AUTHOR : Manvar Nisharg (EE19B094)
INPUT : Sequence of numbers
OUTPUT : Sequence of numbers with duplicates removed in sorted and original form
NOTE : Please use python 3.x as several functions used as only compatible with python 3 or later versions
*********************************************************************************************************"""

"""
Function to remove duplicates from a given sequence of numbers

:Param: NULL
:Return: NULL

:Input: Sequence of numbers seperated by space
:Output: Print sequence with duplicates removed
"""
def remove_duplicates():
        print()
        print("To remove duplicates from a sequence of numbers : ")

        sequence = input("Enter the sequence of numbers seperated by space : ")
        list_of_numbers = sequence.split()              #Spliting sequence string into individual numbers with space as delimitter

        dict_of_numbers = {}    #Dictionary to store already encountered numbers

        print()
        print("The sequence with duplicates removed in sorted order is :",end = " ")
        for number in list_of_numbers:  #Loop through the numbers in sequence
                if number in dict_of_numbers:   #If number already encountered then pass
                        pass
                else:           #Else print the number and add the number in the dictionary
                        print(number, end=" ")
                        dict_of_numbers[number] = 1
        print()


remove_duplicates()             #Call the function
```

Output for some test case

```
To remove duplicates from a sequence of numbers :
Enter the sequence of numbers seperated by space : 2.1 -4.5 1.0 2.1 8.9 8.9 7.7

The sequence with duplicates removed in sorted order is : 2.1 -4.5 1.0 8.9 7.7
```

Basic overview of python dictionaries

- Python dictionaries are implemented as hash tables.
- When adding entries to the table, we start with some slot, that is based on the hash of the key.
- If that slot is empty, the entry is added to the slot.
- If the slot is occupied, Python compares the hash AND the key of the entry in the slot against the hash and key of the current entry to be inserted respectively. If both match, then it thinks the entry already exists, gives up and moves on to the next entry to be inserted. If either hash or the key don't match, it starts probing.
- Probing just means it searches the slots by slot to find an empty slot. Python uses random probing. In random probing, the next slot is picked in a pseudo random order. The entry is added to the first empty slot.What is important is that the slots are probed until first empty slot is found.
- The same thing happens for lookups, just starts with the initial slot i (where i depends on the hash of the key). If the hash and the key both don't match the entry in the slot, it starts probing, until it finds a slot with a match. If all slots are exhausted, it reports a fail.
- When a new dict is initialized it starts with 8 slots and the dict will be resized if it is two-thirds full. This avoids slowing down lookups.

This implementation causes adding and lookup in a dictionary (using in) takes $O(1)$ in general.

Now, analysing the time complexity of the code.
The main idea is that we add the element into the dictionary when we encounter it for the first time. We loop through the input sequence, and check if the element is already present in the dictionary else add it into the dictionary and print it. As, adding and lookup is O(1), the above algorithm takes O(n) as we loop the sequence once.

$$Therefore,\ the\ overall\ complexity\ of\ the\ code\ is\ \underline{O(n)}.$$

Question 4

Given an array A of n integers in the range $[0, n^2 - 1]$, describe a simple method for sorting A in $O(n)$ time.

Answer 4

In [ ]:

```
"""*****************************************************************************************************
PURPOSE : Sort an array of size n with numbers between 0 to n^2 - 1
AUTHOR : Manvar Nisharg (EE19B094)
INPUT : Sequence of numbers following the condition
OUTPUT : Sorted sequence
NOTE : Please use python 3.x as several functions used as only compatible with python 3 or later versions
*****************************************************************************************************"""
import math

"""
Function to implement counting sort on a list of numbers with given radix

:param list_of_numbers: list of numbers to be sorted
:param type: list

:param radix: The radix or the base of the number system
;param type: int

:param place_value: The digit the list is to be sorted
:param type: int

:return: sorted list
:return type: list
"""
def counting_sort(list_of_numbers, radix, place_value) :

        #Lists to store count of elements and the final sorted list
        count_of_elements = [0]*radix
        sorted_array = [0]*len(list_of_numbers)

        #Value to be divided to the numbers to throw the bits before 'place_value'
        divide_by = radix**place_value

        #List of digits at 'place_value'
        radix_array = [(number//divide_by)%radix for number in list_of_numbers]

        #Calculating count of numbers
        for number in radix_array:
                count_of_elements[ number ] += 1

        #Modifying count list by adding counts of elements before it
        for index in range(len(count_of_elements)-1):
                count_of_elements[index+1] += count_of_elements[index]

        #Reversing the arrays
        list_of_numbers.reverse()
        radix_array.reverse()

        #Stroing the numbers in sorted_array at appropriate index and decreasing index in count list
        for index in range(len(list_of_numbers)):
                sorted_array[ count_of_elements[ radix_array[index] ] -1 ] = list_of_numbers[index]
                count_of_elements[ radix_array[index] ] -= 1

        #Return sorted list
        return sorted_array



"""
Function to implement radix sort using counting sort subroutine

:param list_of_numbers: List of numbers to be sorted
:param type: List
"""
def radix_sort(list_of_numbers) :

        #Number of bits in max number in the list
        max_bits = math.floor( math.log( max(list_of_numbers), len(list_of_numbers) ) ) + 1

        #Set radix
        radix = len(list_of_numbers)

        #Sorting numbers with respect to LSB moving towards MSB
        for iteration in range(max_bits):
                list_of_numbers = counting_sort(list_of_numbers,radix,iteration)

        #Return sorted list
        return list_of_numbers

"""
Call function to implement algorithm

:input: Sequence of numbers seperated by space with given conditions
:output: Soreted sequence
"""
def sort_in_range_n2():
        print()
```

```
        print("To sort an array of size n which contains numbers only between 0 to (n^2)-1 :")
        sequence = input("Enter sequence of numbers seperated by space : ")
        print()

        #Seperating individual numbers from sequence using space as delimitter
        list_of_numbers = sequence.split()

        #Max element allowed for given size of input sequence
        max_element = (len(list_of_numbers)**2)-1

        #Converting the numbers from string to int and checking if all numbers fall into given range
        for index in range(len(list_of_numbers)):
                list_of_numbers[index] = int(list_of_numbers[index])
                number = list_of_numbers[index]
                if number > max_element:
                        print("%d out of range"%number)
                        return

        #Special case
        if len(list_of_numbers) == 1:
                print("The list of numbers in sorted order is : [0]")
                return

        #Calling radix sort and printing reesult
        print("The list of numbers in sorted order is : ",end="")
        print(radix_sort(list_of_numbers))

sort_in_range_n2()
```

Output for some test case

- ```
  To sort an array of size n which contains numbers only between 0 to (n^2)-1 :
  Enter sequence of numbers seperated by space : 10 43 20 63 4 15 20 57

  The list of numbers in sorted order is : [4, 10, 15, 20, 20, 43, 57, 63]
  ```

Explanation of Radix sort

Radix sort is a non comparative integer sorting technique that sorts the elements by first grouping the individual digits of the same place value. Then, sort the elements according to their increasing/decreasing order. It uses counting sort as it's subroutine.

$\underline{counting\_sort()}$:

Counting sort finds final index of elements in sorted array by using the count of numbers smaller than it.
For simplicity, consider the data in the range 0 to 9.
Input sequence: 1, 4, 1, 2, 7, 5, 2

1)Take a count array to store the count of each unique object.

$$\begin{array}{ll} Index: & 0\ 1\ 2\ 3\ 4\ 5\ 6\ 7\ 8\ 9 \\ Count: & 0\ 2\ 2\ 0\ 1\ 1\ 0\ 1\ 0\ 0 \end{array}$$

2) Modify the count array such that each element at each index stores the sum of previous counts.

$$\begin{array}{ll} Index: & 0\ 1\ 2\ 3\ 4\ 5\ 6\ 7\ 8\ 9 \\ Count: & 0\ 2\ 4\ 4\ 5\ 6\ 6\ 7\ 7\ 7 \end{array}$$

The modified count array indicates the position of each object in the output sequence.

3) Output each object from the input sequence followed by decreasing its count by 1.

$$Sorted\ array: \ 1,\ 1,\ 2,\ 2,\ 4,\ 5,\ 7$$

Moreover in our function we first change the base of numbers from 10 to 'radix' in linear time by traversing the array once, therefore the time compleixty of this step is $O(n)$.

$$Therefore\ overall\ time\ complexity\ of\ counting\ sort\ is\ O(n+b)\ where\ b\ is\ the\ base.$$

$\underline{radix\_sort()}$:

As said earlier, radix sort sorts the elements by first grouping the individual digits of the same place value. For this it calls the counting sort sub routine for all bits starting from leftmost to rightmost (Number of bits in the maximum number).
Now, let d be the maximum number of digits in the maximum number.Therefore, counting sort will be called d times Also, we take base/radix to be equal to size of array for maximum efficiency.
Therefore, the complexity of radix sort is $O(d(n+n))$ as base = n.
Also, in our case the maximum number is of the order n^2, therefore in base n it will have at most $log_n(n^2) = 2$ digits. Therefore, d = 2.

$$Therefore,\ overall\ complexity\ of\ radix\ sort\ in\ our\ case\ is\ O(n)$$

Question 5

Show that quicksort's best-case running time is $\Omega(nlogn)$.

Answer 5

Firstly, let us see the pseudo code for quicksort.

As we know, quick sort consists of first partitioning the array around a pivot and then recursively calling itself on both the partition arrays.

Therefore, the pseudo code for partition function is

In [ ]:
```
PARTITION(A, start, end)
  pivot = A[end]
  i = start-1

  for j in start to end-1
    if A[j] <= pivot
      i = i+1
      swap(A[i], A[j])
```
```

```
  swap(A[i+1], A[end]) //swapping pivot
  return i+1
```

Clearly, the PARTITION function is linear (i.e., $\Theta(n)$), because there is just one loop which is iterating over the entire array (of size n) and all the other statements are constant time.

Now for the quicksort function.

In [ ]:
```
QUICKSORT(A, start, end)
  if start < end
    q = PARTITION(A, start, end)
    QUICKSORT(A, start, q-1)
    QUICKSORT(A, q+1, end)
```

So, the overall time taken by the QUICKSORT function would be the sum of time taken by PARTITION function and the two recursive calls.

So, the time taken would be

$$T(n) = T(L) + T(R) + \Theta(n)$$

Now, let $L = i\,and\,R = n - i - 1$, then the equation becomes,

$$T(n) = T(i) + T(n - i - 1) + \Theta(n)$$

We see that at level 0 of the recursion tree the sum of non recursive terms is $\Theta(n)$.
Breaking it down one step further,

$$T(n) = (T(A) + T(i - A - 1) + \Theta(i)) + (T(B) + T(n - i - 2 - B) + \Theta(n - i - 1)) + \Theta(n)$$

We can see that at level 1 of the recursion tree the sum of non recursive terms is $\Theta(i) + \Theta(n - i - 1) = \Theta(n - 1)$. We see that at any level k in the recursion tree the upper bound of summation of non recursive terms will be $O(n)$. The number of elements reduced at a particular level will be equal to the numbers of pivots in the previous level. Thus, the summation of non recursive terms through levels, till level k will be,

$$\sum_{1}^{k}(O(n))$$

Thus, we see that to minimise the time complexity we need to minimize the levels in the recursion tree. At any level the partition with the maximum number of elements will have the maximum depth from that level. Thus, to minimise the overall height we should try and keep all partitions of equal size at a level.

Thus, for minimum height we need to keep dividing the array into equally sized partitions, which will result in the recursive equation,

$$T(n) = 2T(n/2) + \Theta(n)$$

The solution to this equation using case 1 of Master Theorem, is

$$T(n) = \Theta(nlogn)$$

So, the in the best case scenario, quick-sort runs in $\Theta(nlogn)$ time.
$\therefore$ We can say that quick-sort best case running time is,

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