Question 1:
Construct a O(nlogn) algorithm based on the divide and conquer approach. You have to perform the following tasks for each of the problem which is given below:

a) Write pseudo code or algorithm based on divide and conquer approach
b) Find the recurrence relation of part (a) and then solve it by using iterative substitution method discussed in class.

Problem 1. Fast University Main Campus Karachi Cafeteria Management is decided to open a bakery with the restaurant which provides services to the students, faculty and staff based on the demand. The bakery produces or manufactures (baked) many products or item based on the needs. Suppose a list given which contains the sales of items on weekly basis and a target has been set by the management to achieve in a certain deadline of weeks. Your task is to find the 2 different and distinct weeks (not necessarily to be consecutive) whose sales is equal to the target. (where list is not necessarily sorted). your algorithm must return true if there are two distinct weeks that is w1_sales+w2_sales = target_sales otherwise it returns false.

Problem 2. Upon the successful launching of Fast Main Campus bakery, the management decided to open a new branch at the Fast City Campus which provides services to the students, faculty and staff based on the demand. The bakery produces or manufactures (baked) many products or item based on the needs. Suppose a list given which contains the sales of items on weekly basis and managements again sets a target sales which they want to achieve but this time they want to know collectively contribution of both bakeries to get the sales target. Your task is to find a one unique week from each of the bakery sales list such that w1_sales+w2_sales = target_sales. Your algorithm must return true in this case otherwise it returns false.

**Solution:**
**Problem 1) :-**
Applying divide and conquer algorithm for searching in form of pseudo code

```
int find_sumSET (A[], n, P)          //A is input array of size n, and P is the target sale//
{
Sort ( A, n)                               // sorting array in increasing order//
for( i = 0 to n-1 )
{
a = BinarySearch ( A, 0, n-1, P-A[i] )    // binary search for array P-A[i] value//
if (a )
return 1                                  // if value is found ,then return true//
}
return -1                                  // if pair of values is not found in array , then returning false
//
}
```

**Problem-2**

```
def has_pair_with_sum(A, B, v):
    # Sort array B
    B.sort()

    # Iterate through array A
    for num in A:
        # Calculate the complement needed to achieve the target sum v
        complement = v - num

        # Check if the complement exists in array B using binary search
        if binary_search(B, complement):
            return True  # Found a pair with the sum v

    return False  # No such pair found

# Example usage:
A = [1, 2, 3, 4, 5]
B = [10, 9, 8, 7, 6]
v = 13
result = has_pair_with_sum(A, B, v)
print(result)  # Output: True (since 8 + 5 = 13)
```

---------------------------------------------------------

Binary search code, just for reference below:

```
def binary_search(arr, target):
    left, right = 0, len(arr) - 1
    while left <= right:
        mid = left + (right - left) // 2
        if arr[mid] == target:
            return True
        elif arr[mid] < target:
            left = mid + 1
        else:
            right = mid - 1
    return False
```

**Question 2**

Suppose you are playing a story writing game and you have given a word where all the letters in the words can be considered to appear twice consecutively (e.g. xx). It can be an error if one letter appears only once. You need to design an O(logn) algorithm to find that single letter.

For the sake of simplicity assume all the words are correct and there is no need to handle any exceptional cases., i.e. the letters either appear twice or appear as a single character, but not any other case.

Also find the recurrence relation for the above problem and solve it using iterative substitution method.

Examples : Word xxyynzzwwii returns 4
: Word xxyyzzwwii returns None
: Word kkllmmjjc returns 8

**Solution (Python):**

```
def find_single(arr, low, high):
    # Base case
    if low > high:
        return None
    if low == high:
        return low

    # Find the middle point
    mid = (low + high) // 2

    # If mid is even
    if mid % 2 == 0:
        # If the element next to mid is the same as mid,
        # then the output element lies on the right side,
        # else it lies on the left side
        if arr[mid] == arr[mid + 1]:
            return find_single(arr, mid + 2, high)
        else:
            return find_single(arr, low, mid)
    else:
        # If mid is odd
        # If the element before mid is the same as mid,
        # then the output element lies on the right side,
        # else it lies on the left side
        if arr[mid] == arr[mid - 1]:
            return find_single(arr, mid + 1, high)
        else:
            return find_single(arr, low, mid - 1)



# Input string
input_string = input()
result = find_single(input_string, 0, len(input_string) - 1)

if result is not None:
    print(result)
else:
    print("None")
```

The recurrent relation for above problem will be:

T(n)=T(n/2)+O(1)

Solving the recurrent relation using iterative substitution method

T(n)=T(n/2)+C

So,

T(n)=T(n/4)+C x 2

T(n)=T(n/2^3)+C x 3

....

T(n)=T(1)+C x log(n)

T(n)=θ log(n))

**Question 3:**
Assume you have n jars, one of which is heavier (where n is a power of three). The weight of n/3 jars (excluding extra weight of the single jar) is given. Create an algorithm having complexity lesser than O(n), to determine the heaviest jar.

a. Clearly demonstrate how you could arrived at the solution.
b. Write your algorithm in the form of pseudo code.
c. Determine the algorithm's running time.

**Solution:**
**Demonstration:**
Assume we have 9 jars then make the 3 groups of 3 jars then take two groups of jars and place one group on each side of the balance and check. There may be 3 conditions.

Case 1: Both the groups of jars have same weight
Case 2: Group1 of jars is heavier than the Group2 of jars
Case 3: Group2 of jars is heavier than the Group1 of jars.

So if Case 1 is true then heavier jar will be in Group3, If Case 2 is true then Group1 is having heavier jar else Group2 will contain the heavier jar.

**Pseudo code:**
```
Algorithm 9_jar_puzzle(Number_of_jars, balance)
// balance is the cumulative weight of jars in a group.

Step-1: If Number_of_jars == 1
                return jar
        Else
        Step-2:   Divide the jars into 3 groups of equal jars G1,G2,G3
        Step-3:       If Weight(G1)== Weight(G2)
                          SelectedG = G3
                      Else if Weight(G1) > Weight(G2)
                          SelectedG = G1
                      Else
                          SelectedG = G2
                      Step-4: 9_Call jar_puzzle(SelectedG, balance/3)
Step-5: STOP
```

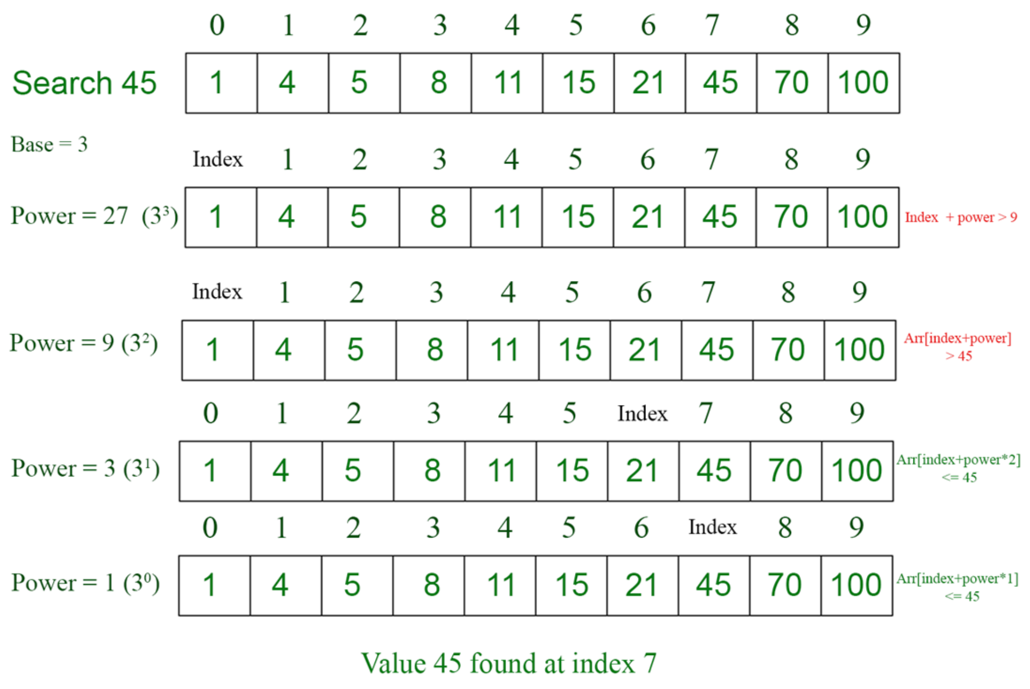
**Running time:**

As we can see we have fix number of computations at every recursive step, i.e. compare two group weights and then select one.

As $n=3^k$
the n required computations only k times.
So the complexity will be $T(n) = O(\log_3 n)$
Every time problem size reduced to 1/3 so it will make a recursion tree which is having height only $\log_3 n$, at each level one need to use balance only once. So $T(n) = O(\log_3 n)$

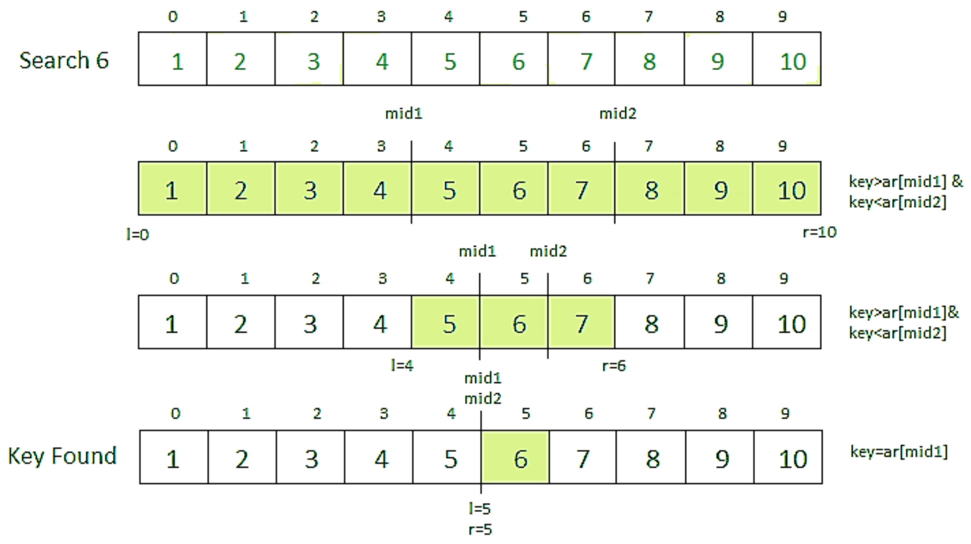
**Question 4**
You have studied Binary Search algorithm to search a number in an array. There are slight variants of it known as **Ternary Search & Meta Binary Search.** Learn about theses variants from below links;
https://www.geeksforgeeks.org/ternary-search/
https://www.geeksforgeeks.org/meta-binary-search-one-sided-binary-search/
Take an arbitrary input sorted array of size 10 (other than presented in the above links) and search a number present at the end of array using both Ternary Search & Meta Binary Search**.**
Show each iteration of both algorithms, Also Discuss pros and cons of both techniques as compared to original Binary Search algorithm in different scenarios.

**<u>Solution:</u>**



The advantage of **Meta Binary Search** over binary search is that it can perform fewer comparisons in some cases, particularly when the target element is close to the beginning of the list. The disadvantage is that the algorithm may perform more comparisons than binary search in other cases, particularly when the target element is close to the end of the list. Therefore, Meta Binary Search is most effective when the list is ordered in a way that is consistent with the distribution of the target elements.

- **Ternary Search** is a divide-and-conquer algorithm that is used to find the position of a specific value in a given array or list.
- It works by dividing the array into three parts and recursively performing the search operation on the appropriate part until the desired element is found.
- The algorithm has a time complexity of O(log3n) and is more efficient than a linear search, but less commonly used than other search algorithms like binary search.
- It’s important to note that the array to be searched must be sorted for Ternary Search to work correctly.



**Binary search Vs Ternary Search**
The time complexity of the binary search is more than the ternary search but it does not mean that ternary search is better. In reality, the number of comparisons in ternary search much more which makes it slower than binary search.

**Uses:** In finding the maximum or minimum of a unimodal function.

**When to use:**

- When you have a large ordered array or list and need to find the position of a specific value.
- When you need to find the maximum or minimum value of a function.
- When you need an alternative algorithm for binary search with an efficient time complexity.
- When you are interested in reducing the number of comparisons.

**Advantages:**

- Ternary Search has a time complexity of O(log3n), which is more efficient than linear search and comparable to binary search.
- Number of comparisons get reduced.
- Works well for large datasets.
- Fits well with optimization problems.
- Ternary Search is a non-recursive algorithm, so it does not require additional memory to store function call stack, thus it’s space efficient.

**Disadvantages:**

- Ternary Search is only applicable to ordered lists or arrays, and cannot be used on unordered or non-linear data sets.
- Requires an in depth understanding of recursion.
- Implementation is not easy.
- Ternary search is not suitable for non-continuous function as it is based on dividing the search space into 3 parts.

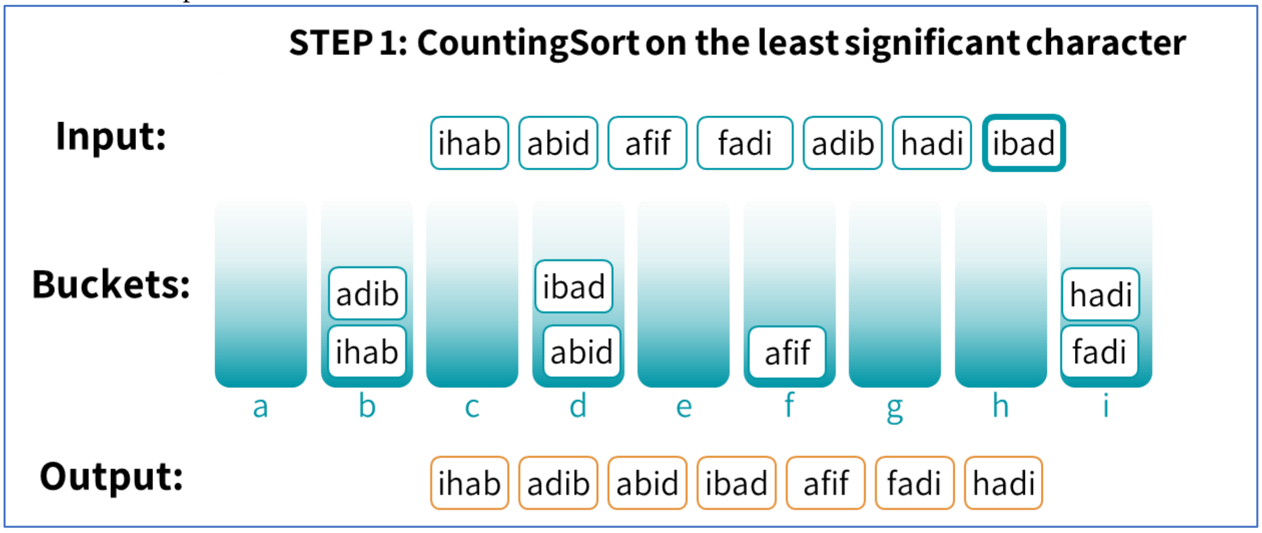
**Question 5:**
Describe an algorithm that, given n integers in the range 0 to k, preprocesses its input and then answers any query about how many of the n integers fall into a range [a..b] in O(1) time. Your algorithm should use Θ(n + k) preprocessing time.

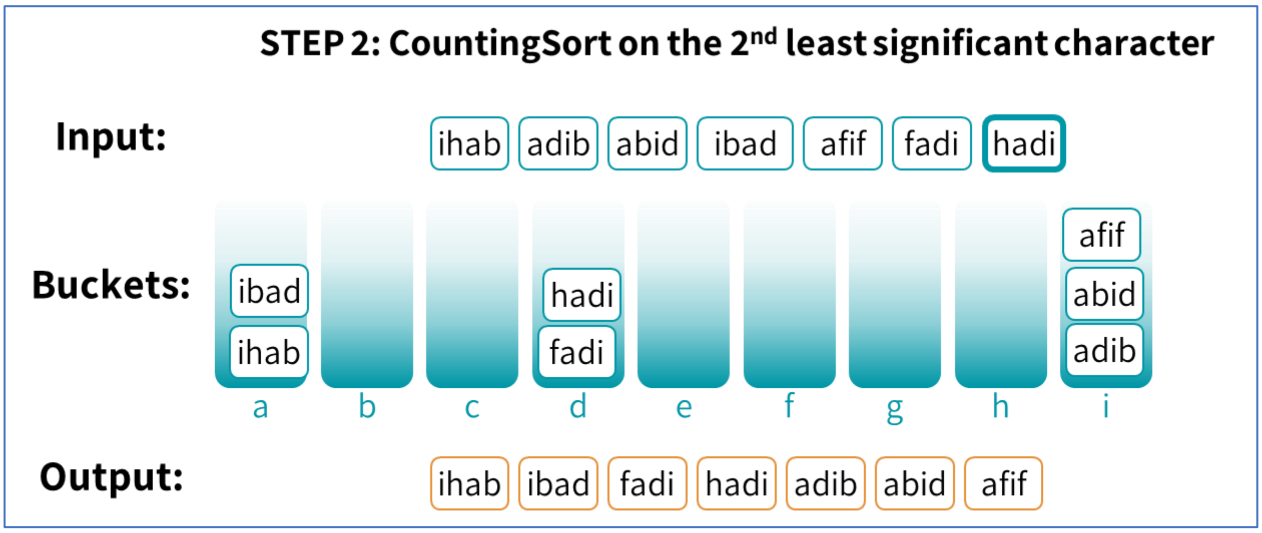
Solution:
Compute the C array as is done in counting sort. The number of integers in the range [a:b] is C[b] - C[a-1], where we interpret C[-1] as 0.

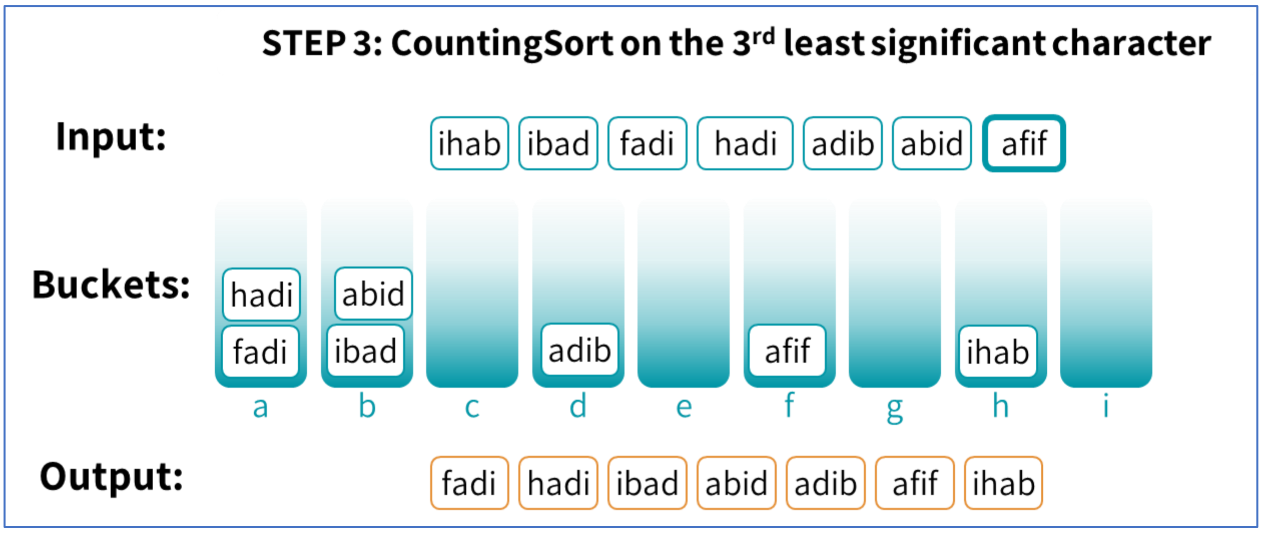
**Question 6**
Show the steps by using an O(n) algorithm, to sort the following list of names in ascending order. Show all the steps in each iteration.

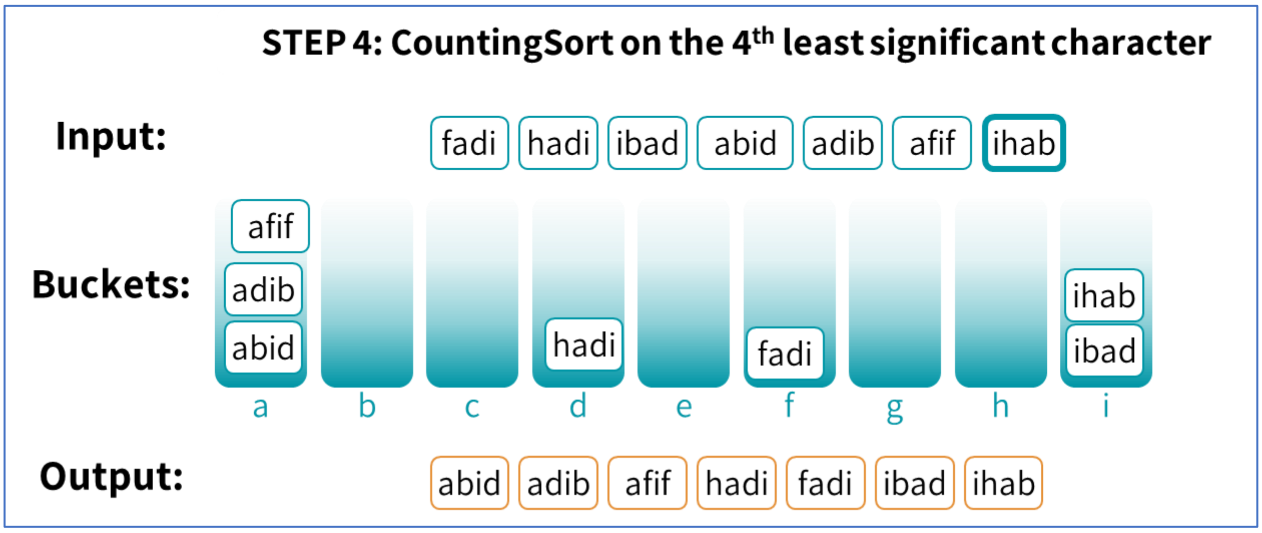
ihab, abid, afif, fadi, adib, hadi, ibad
Hint: You do not need to show all the letters from a to z, instead you can take the last letter to be "i" for simplicity (i.e. from a to i).

Solution:
It is Radix sort problem.


STEP 1: CountingSort on the least significant character
Input:
ihab abid afif fadi adib hadi ibad
Buckets:
adib ihab
ibad abid
afif
hadi fadi
a b c d e f g h i
Output:
ihab adib abid ibad afif fadi hadi


STEP 2: CountingSort on the 2nd least significant character
Input:
ihab adib abid ibad afif fadi hadi
Buckets:
ibad ihab
hadi fadi
afif abid adib
a b c d e f g h i
Output:
ihab ibad fadi hadi adib abid afif


STEP 3: CountingSort on the 3rd least significant character
Input:
ihab
ibad
fadi
hadi
adib
abid
afif
Buckets:
hadi
fadi
abid
ibad
adib
afif
ihab
a
b
c
d
e
f
g
h
i
Output:
fadi
hadi
ibad
abid
adib
afif
ihab


STEP 4: CountingSort on the 4th least significant character
Input:
fadi
hadi
ibad
abid
adib
afif
ihab
Buckets:
afif
adib
abid
hadi
fadi
ihab
ibad
a
b
c
d
e
f
g
h
i
Output:
abid
adib
afif
hadi
fadi
ibad
ihab