**Problem-1:**

Suppose you are working as a data analyst for a digital marketing agency. Your agency manages online advertising campaigns for various clients. As part of your role, you need to analyze and optimize the performance of these campaigns to maximize the click-through rates (CTR) of ad impressions.

Input Scenario:

1. Number of ad impressions: The data analyst provides the number of ad impressions in the campaign.
2. CTR values: The data analyst collects the click-through rate (CTR) values for each ad impression. These values represent the percentage of users who clicked on the ad after viewing it.

Output Scenario:

1. Sorted Impressions: The data analyst needs to obtain a sorted list of ad impressions based on their click-through rates (CTR). The sorted list will display the ad impressions in ascending order of CTR values.

Example: Input:

* Number of ad impressions: 1000
* CTR values: [0.15, 0.07, 0.23, 0.12, 0.09, ...]

Output:

* Sorted Impressions: [0.02, 0.04, 0.07, 0.09, 0.12, 0.15, 0.18, ...]

Algorithm:

1. Input:
   * Gather the necessary data, including the number of ad impressions and the corresponding CTR values for each impression.
2. Create Buckets:
   * Determine the range of CTR values.
   * Decide on the number of buckets and calculate the interval size.
3. Distribute Impressions into Buckets:
   * Create empty buckets to hold the impressions.
   * Iterate through each ad impression and determine the bucket index based on its CTR value and the interval size.
   * Place each impression into the corresponding bucket.
4. Sort the Buckets:
   * Sort each bucket individually using a sorting algorithm like insertion sort or quicksort.
   * The time complexity of the sorting algorithm used within each bucket is not necessarily limited to O(nlog(n)), as the bucket sort algorithm's time complexity primarily depends on the sorting algorithm used for the individual buckets.
5. Concatenate Sorted Impressions:
   * Combine the sorted impressions from all the buckets into a single sorted list.
6. Output:
   * The sorted list of impressions represents the ad impressions sorted based on their CTR values.

Using the bucket sort algorithm in this context can be beneficial when dealing with a large number of impressions with a wide range of CTR values. The algorithm divides the impressions into buckets based on their CTR values, sorts each bucket individually, and then concatenates the sorted buckets to obtain the final sorted list. This approach can achieve a more efficient time complexity compared to general-purpose sorting algorithms when the number of buckets is significantly smaller than the number of impressions.

Please note that the specific implementation details may vary based on the programming language and other factors, and you may need to adapt the algorithm accordingly.

The provided code demonstrates an implementation of the bucket sort algorithm to solve a real-life scenario problem related to ad impressions and click-through rates (CTR). Let's go through the code and understand the problem and its solution step by step:

1. Reading the number of ad impressions and CTR values:
   * The code starts by prompting the user to enter the number of ad impressions.
   * Then, it reads the CTR values for each impression from the user.
2. Performing bucket sort:
   * The **bucketSort** method is called to sort the CTR values using the bucket sort algorithm.
   * The **bucketSort** method takes an array of CTR values as input and returns a list of lists (**List<List<Double>>**), where each inner list represents a bucket containing CTR values falling within a specific range.
3. Determining the range and number of buckets:
   * In the **bucketSort** method, the minimum and maximum CTR values are determined using the **min** and **max** functions from the **Arrays** class.
   * The code then decides on the number of buckets (10 in this case) and calculates the interval size by dividing the range of CTR values equally among the buckets.
4. Creating the buckets:
   * A list of lists (**List<List<Double>>**) called **buckets** is created to hold the CTR values in different buckets.
   * A loop is used to initialize each bucket as an empty ArrayList.
5. Distributing the ad impressions into buckets:
   * Another loop iterates through each CTR value in the input array.
   * Based on the CTR value and the interval size, a bucket index is calculated, indicating which bucket the CTR value belongs to.
   * The CTR value is then added to the corresponding bucket.
6. Sorting the buckets:
   * After distributing the CTR values into buckets, another loop is used to sort each bucket.
   * The **sort** method is called on each bucket, which uses the default sorting order (ascending) to sort the CTR values within each bucket.
7. Concatenating the sorted impressions:
   * In the **main** method, a new ArrayList called **sortedImpressions** is created to hold the sorted CTR values.
   * A nested loop is used to iterate through each bucket and concatenate the sorted CTR values into the **sortedImpressions** list.
8. Printing the sorted impressions:
   * Finally, the code prints the header "Sorted Impressions" and iterates through the **sortedImpressions** list to display each CTR value.

The provided code sorts the ad impressions based on their click-through rates (CTR) using the bucket sort algorithm. It groups the impressions into different buckets based on their CTR values, sorts the CTR values within each bucket, and concatenates the sorted values to obtain the final sorted list of impressions.

**Time Complexity:**

1. Reading Input: The code reads the number of ad impressions and the corresponding CTR values, which has a time complexity of O(n), where n is the number of ad impressions.
2. Determining Range and Creating Buckets: The code determines the range of CTR values and creates the buckets. This step has a time complexity of O(n), as it involves finding the minimum and maximum CTR values from the input array.
3. Distributing Impressions into Buckets: The code iterates through each CTR value and distributes it into the appropriate bucket. This step has a time complexity of O(n), as it loops through the entire array of CTR values.
4. Sorting Buckets: The code sorts each bucket individually using the sort method. The time complexity of this step depends on the sorting algorithm used within the sort method. In the given code, it uses the default sorting order (ascending), so the time complexity is typically O(m \* log(m)), where m is the average number of impressions in each bucket.
5. Concatenating Sorted Impressions: The code concatenates the sorted impressions from all the buckets into a single list. This step has a time complexity of O(n), where n is the total number of impressions.

Therefore, the overall time complexity of the given code can be considered as O(n + m \* log(m)), where n is the number of ad impressions and m is the average number of impressions in each bucket.

In terms of space complexity, the code uses additional space to store the buckets and the sorted impressions. The space complexity is O(n + m), where n is the number of ad impressions and m is the average number of impressions in each bucket.

**Problem-2:**

You have been given an array A of size N. The array contains integers. You need to divide the elements of this array into buckets on the basis of the number of set bits in its binary representation. You need to then print the content of each bucket in a new line. The buckets should appear in the output in ascending order, i.e the bucket that stands for lesser number of set bits should appear before any other bucket which stands for higher number of set bits.The elements of each bucket should appear in ascending order too. That is if 2 integers appear in the same bucket, the one with the lower value should appear in the bucket list before the one with higher value.

**Input Format**:

The first line contains a single integer *N* denoting the size of the array. The next line contains *N* space separated integers denoting the elements of the array.

**Output Format**:

The output should contain the number of lines equal to the number of distnict bucket. If a bucket remains empty, it should not appear in the output. Print the contents of each bucket on a new line.

To solve this problem, you can follow these steps:

1. Read the input values, including the size of the array N and the array elements A.
2. Create a dictionary or a hash map to represent the buckets. The keys of the dictionary will be the count of set bits in the binary representation of the numbers, and the values will be lists to store the numbers belonging to each bucket.
3. Iterate through each element in the array A.
4. For each element, calculate the count of set bits in its binary representation. You can use the bitwise AND operator (&) and right shift (>>) operations to count the set bits.
5. Check if the count exists as a key in the dictionary. If it doesn't, create a new key-value pair with the count as the key and an empty list as the value.
6. Append the current element to the list corresponding to the count in the dictionary.
7. After iterating through all the elements, sort the elements within each bucket in ascending order.
8. Sort the keys of the dictionary in ascending order.
9. Print the contents of each bucket in a new line, according to the sorted keys of the dictionary.

**Algorithm;**

1. Define the **countSetBits** function that takes an integer **num** as input and returns the count of set bits in its binary representation.
   * Initialize **count** to 0.
   * While **num** is not zero, do the following:
     + Increment **count** by performing a bitwise AND of **num** with 1.
     + Right-shift **num** by 1.
   * Return **count**.
2. Define the **bucketizeElements** function that takes an integer array **A** as input and performs the bucketization of elements based on the count of set bits in their binary representation.
   * Create an empty **HashMap** called **buckets** to store the buckets.
   * For each element **num** in **A**, do the following:
     + Calculate the count of set bits in **num** by calling the **countSetBits** function.
     + If **setBitCount** is not a key in **buckets**, create a new key-value pair in **buckets** with **setBitCount** as the key and an empty **ArrayList** as the value.
     + Append **num** to the list corresponding to **setBitCount** in **buckets**.
   * For each bucket in the values of **buckets**, do the following:
     + Sort the elements in the bucket in ascending order.
     + For each element **num** in the bucket, print **num** followed by a space.
     + Print a new line.
3. In the **main** function:
   * Read the value of **N** from input.
   * Create an empty integer array **A** of size **N**.
   * For each index **i** from 0 to **N-1**, do the following:
     + Read the element **A[i]** from input.
   * Call the **bucketizeElements** function with **A** as the argument.

**Java Implementation:**

package Presentation;

import java.util.\*;

public class Bucket {

static int countSetBits(int num) {

int count = 0;

while (num != 0) {

count += num & 1;

num >>= 1;

}

return count;

}

static void bucketizeElements(int[] A) {

Map<Integer, List<Integer>> buckets = new HashMap<>();

for (int num : A) {

int setBitCount = countSetBits(num);

if (!buckets.containsKey(setBitCount)) {

buckets.put(setBitCount, new ArrayList<>());

}

buckets.get(setBitCount).add(num);

}

for (List<Integer> bucket : buckets.values()) {

Collections.sort(bucket);

for (int num : bucket) {

System.out.print(num + " ");

}

System.out.println();

}

}

public static void main(String[] args) {

Scanner scanner = new Scanner(System.in);

int N = scanner.nextInt();

int[] A = new int[N];

for (int i = 0; i < N; i++) {

A[i] = scanner.nextInt();

}

scanner.close();

bucketizeElements(A);

}

}

**Time Complexity:**

1. Counting Set Bits: The **countSetBits** function calculates the count of set bits in a binary number representation. It iterates over the bits of the number, which takes O(log M) time, where M is the maximum value in the array.
2. Bucketization: The **bucketizeElements** function performs the bucketization of elements based on the count of set bits. It iterates over each element in the array, which takes O(N) time, where N is the size of the array. For each element, it performs constant time operations like checking and inserting elements into the map. Therefore, the overall time complexity for bucketization is O(N).
3. Sorting: Sorting the elements within each bucket takes O(K log K) time, where K is the maximum number of elements in a single bucket. Since each element is assigned to a unique bucket, the total number of elements across all buckets is N. Therefore, the overall time complexity for sorting is O(N log N).
4. Printing: Printing the contents of each bucket takes O(N) time, as each element is printed once.

Considering the above steps, the dominant factor affecting the time complexity is the sorting step, which is O(N log N). Therefore, the overall time complexity of the given code is O(N log N).