Revision Note - Hashing

Hashing:

- Hashing is a technique used in computer science to map data of arbitrary size to fixed-size values. It involves applying a hash function to the input data, which generates a unique hash code or hash value. Hashing is commonly used for data storage, retrieval, and quick lookup operations.
- The primary purpose of hashing is to quickly determine the location or index of stored data in a data structure called a hash table or hash map.

Data Structures that use Hashing Technique

Set

A set is a collection of unique elements where the order of elements is not significant. It ensures that no duplicate elements are stored, providing efficient membership checks and eliminating redundancy.

In various programming languages, sets are typically implemented using data structures such as hash sets or tree sets.

Map

A map is a key-value pair data structure that uses hashing for efficient key-based operations. It allows for quick insertion, deletion, and retrieval of elements based on their associated keys.

Maps are also known as associative arrays, hash tables, or dictionaries in different programming languages.

Equivalents in Different Programming Languages:

Language	Set Equivalent	Map Equivalent
Java	HashSet	HashMap
C++	Unordered_Set	Unordered_Map
Python	Dictionary	Dictionary
JS/Ruby	Set	Map
C#	HashSet	Dictionary

Understanding hashing, sets, and hash maps, along with their equivalents in different programming languages, is essential for efficient data storage, retrieval, and manipulation in various software applications.

Problem 1 - Basic

Problem Description

Q. Count the number of distinct elements in an array.

Example:

Input: **[2, 3, 2, 3, 2, 5, 7, 7]**
Output: **4**

Explanation: There are 4 different elements: **2, 3, 5 ,7**

Solution

Add all elements to a set and then print the size of the set.

Pseudo Code:

```
function countDistinctElements(array):
    // Create an empty set
    set = new empty set

    // Traverse the array
    for element in array:
        // Add each element to the set
        set.add(element)

    // Return the size of the set
    return set.size
```

Problem 2 - Intermediate

Problem Description

Q. Given an array containing both positive and negative integers, we have to find the length of the longest subarray with the sum of all elements equal to zero.

Example

Input: **[19, -4, 4, -2, 7, -5]**
Output: **5**

Explanation:
The following subarrays sum to zero:
{-4, 4} , **{-2, 7, -5}**, **{-4, 4, -2, 7, -5}**
So, the longest among the above three is the last one with length of 5.

Solution

Brute Force/Naive Solution:

One way to solve this problem is to consider all possible subarrays and calculate their sum. If the sum of a subarray is zero, we update the maximum length. We repeat this process for all subarrays and return the maximum length found.

Pseudo code:

```
function findMaxLength(arr):
    n = length of arr
    maxLength = 0

    for i = 0 to n-1:
        sum = 0
        for j = i to n-1:
            sum += arr[j]
            if sum == 0:
                maxLength = max(maxLength, j - i + 1)

    return maxLength
```

Time Complexity (TC): O(n^2)
Space Complexity (SC): O(1)

Optimal Approach using Hashmap:

We can optimize the solution using a hashmap to store the sum of elements encountered so far and their corresponding indices. We calculate the prefix sum by adding each element to the sum variable. If the current sum is zero or has been encountered before, it means the subarray from the previous occurrence of that sum to the current element has a sum of zero. We update the maximum length if necessary.

Pseudo code:

```
function findMaxLength(arr):
    n = length of arr
    maxLength = 0
    sum = 0
    hashMap = empty hashmap of mapping integer to integer

    for i = 0 to n-1:
        sum += arr[i]

        if sum == 0:
            maxLength = i + 1

        if sum in hashMap:
            maxLength = max(maxLength, i - hashMap[sum])
        else:
            hashMap[sum] = i

    return maxLength
```

Time Complexity (TC): O(n)
Space Complexity (SC): O(n) - The space used by the hashmap.

In the optimal approach, we traverse the array only once, calculating the sum and updating the hashmap. This results in a more efficient solution compared to the naive approach.

Problem 3 - Advanced

Problem Description

Q. Given an array, find the length of the largest sub-sequence which can be re-arranged to form a sequence of consecutive numbers.

Example

Input: **[100, 4, 100, 3, 1, 2]**
Output: **4**

Explanation:
The largest sub-sequence that can be rearranged to obtain a sequence of consecutive numbers is 4, 3, 1, 2 -> **1, 2, 3, 4**.

Solution

Brute Force: The brute force algorithm for finding the length of the largest subsequence that can be rearranged to form a sequence of consecutive numbers does not employ any advanced techniques. It simply examines each number in the given array and tries to count as high as possible from that number, using only the numbers present in the array. Whenever it encounters a number that is not present in the array, it records the length of the current sequence if it is greater than the previously recorded maximum length. The algorithm exhaustively explores all possibilities, making it guaranteed to find the optimal solution.

TC: O(n^3)
SC: O(1)

Optimized Solution: This involves using the concept of Hashing.

- Create a Set to store all the numbers from the input array.
- Iterate through the array and add each element to the Set.
- Iterate through the elements in the HashSet. For each element, check if its previous number (i-1) is not present in the HashSet. If the previous number is not present, it means the current number is the starting point of a consecutive subsequence.
- Start counting the length of the subsequence from the current number by incrementing a length and the current number by 1.
- Continue incrementing the currentNumber and counting the length until the consecutive subsequence ends (i.e., the next number is not present in the HashSet).
- Update the maxLen if the current length (len) is greater than the previous maximum.
- Repeat steps the steps for all elements in the HashSet.
- Return the maxLen as the length of the longest consecutive subsequence.

```
function longestConsecutive(nums):
    set = new set of integers
    for i in nums:
        set.add(i)

    maxLen = 0
    for i in set:
        if set does not contain (i-1):
            len = 0
            currentNumber = i
            while set contains currentNumber:
                len++
                currentNumber++

            maxLen = max(maxLen, len)

    return maxLen
```

- Time Complexity: **O(n)**. Although the time complexity appears to be quadratic due to the while loop nested within the for loop, closer inspection reveals it to be linear. Because the while loop is reached only when currentNum marks the beginning of a sequence (i.e. currentNum-1 is not present in nums), the while loop can only run for nnn iterations throughout the entire runtime of the algorithm. This means that despite looking like O(n^2) complexity, the nested loops actually run in **O(n+n)=O(n)** time. All other computations occur in constant time, so the overall runtime is linear.
- Space Complexity: **O(n)** for the Set.

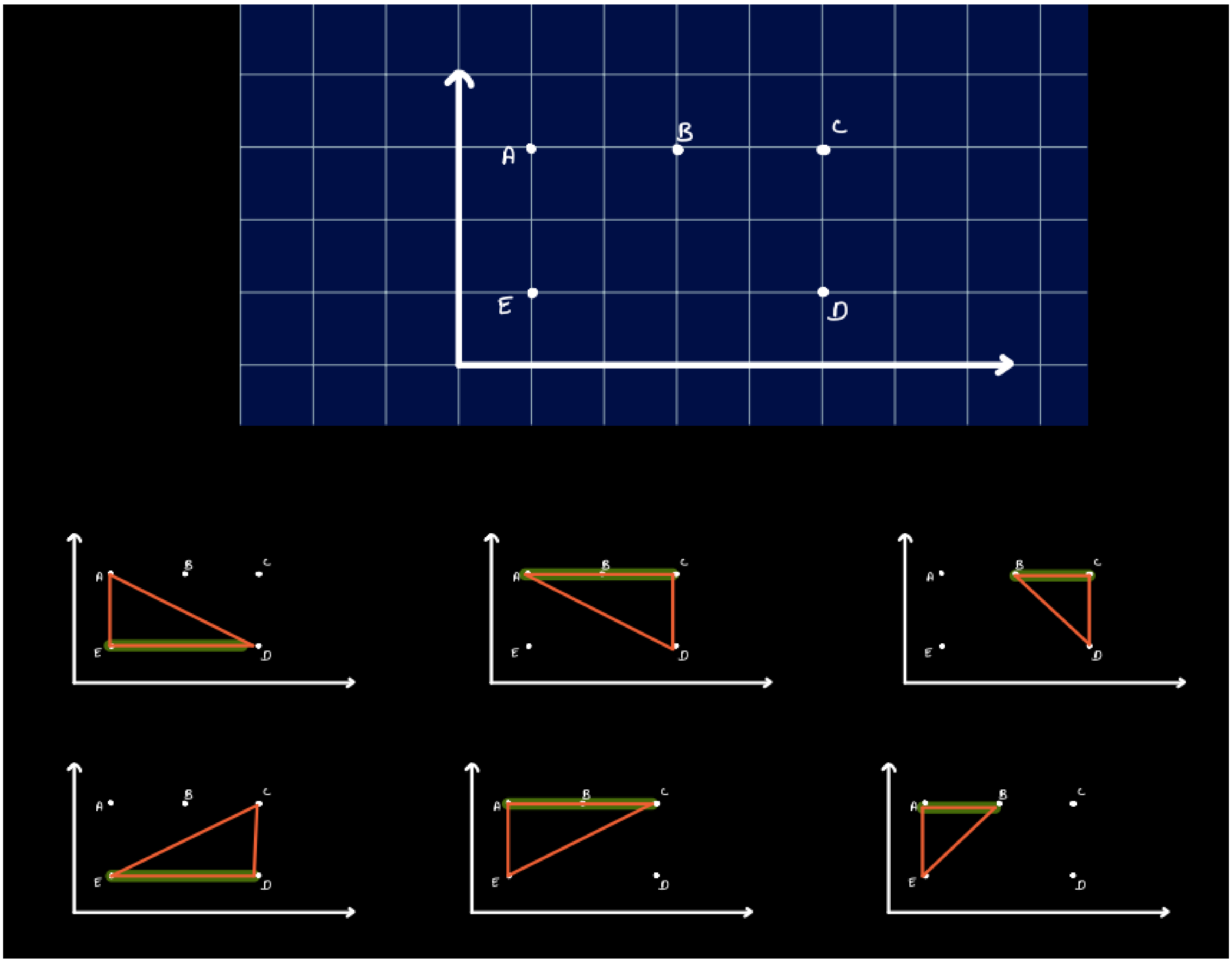
Problem 3 - Advanced

Problem Description

Q. Given N points in a 2D plane, count the number of right angled triangles which has atleast one non-hypoteneus side parallel to the x-axis.

Example:
Input: **(1, 2), (3, 3), (5, 3), (5, 1), (1, 1)**
Output: **6**

Explanation:



Hence, the answer is **6**.

Solution

Brute Force: The solution approach for the given pseudo code is to iterate over each point on the x-axis and for each point, iterate over all points on the y-axis. Within these nested loops, check if the points satisfy the conditions for a right-angled triangle with one non-hypotenuse side parallel to the x-axis. If the conditions are met, increment the count. Finally, return the count as the result.

```
count_right_angled_triangles(PointsOnXAxis[], PointsOnYAxis[]):
    count = 0
    n = number of points
    for i = 0 to ln - 1:
        for j = 0 to n - 1:

            if(i==j) continue
            for k = 0 to n - 1:

                if(i==k || k==j) continue

                if (y[i] == y[j]) or (x[i] == x[k]):
                    count += 1

    return count
```

- The time complexity of this solution is O(n^3) as it involves three nested loops, where n is the total number of points.
- The space complexity is O(1) as it only requires a constant amount of space for variables.

Optimal Approach: For every point (x,y), if there are n points with the same x co-ordinate and there are m points with the same y co-ordinate, then the `ans` will be `ans = ans + (n-1) * (m-1)`.

Why this solution?
The solution works because it leverages the fact that for each point (x, y), the number of right-angled triangles that can be formed with one non-hypotenuse side parallel to the x-axis is determined by the count of points with the same x-coordinate (n) and the count of points with the same y-coordinate (m).

Consider a point (x, y). If there are n points with the same x-coordinate as (x, y), it means that there are n - 1 other points that can potentially form a right-angled triangle with (x, y) as one of its vertices and one non-hypotenuse side parallel to the x-axis. Similarly, if there are m points with the same y-coordinate as (x, y), it means that there are m - 1 other points that can potentially form a right-angled triangle with (x, y) as one of its vertices and one non-hypotenuse side parallel to the x-axis.

By multiplying the counts (n-1) and (m-1) for each point, we obtain the total number of right-angled triangles that can be formed with one non-hypotenuse side parallel to the x-axis. Summing up these counts for all points gives us the final result.

This approach ensures that each right-angled triangle is counted only once, as we consider each point individually and exclude the point itself from contributing to the count. Therefore, the solution accurately calculates the number of right-angled triangles satisfying the given conditions.

Pseudo Code:

```
count_right_angled_triangles(points):
    countOfTriangles = 0
    xCounts = Map of integer to integer to store frequencies of x coordina
    yCounts = Map of integer to integer to store frequencies of y coordina

    for point in points:
        x, y = point
        if x exists in xCounts:
            increment the count of x in xCounts
        else:
            add x to xCounts with a count of 1
        if y exists in yCounts:
            increment the count of y in yCounts
        else:
            add y to yCounts with a count of 1

    for point in points:
        x, y = point
        n = get the count of x from xCounts
        m = get the count of y from yCounts
        contribution = (n - 1) * (m - 1)
        increment countOfTriangles by contribution

    return countOfTriangles
```

Time Complexity: O(n), we do not have any nested loops. In one iteration we form the Map, mapping the frequency of x and y coordinates. In the next iteration we perform the counting operation them.

TreeSet and TreeMap

- **TreeSet:** A sorted set implementation that stores elements in a self-balancing binary search tree, allowing efficient operations like insertion, deletion, and retrieval with a time complexity of O(log n).

- **TreeMap:** A sorted map implementation that stores key-value pairs in a self-balancing binary search tree, providing efficient operations like insertion, deletion, and retrieval based on keys with a time complexity of O(log n).