PW SKILLS

Lesson Plan

Hashmap and Set Java

PW SKILLS

Introduction (Understanding the need)

Let's say you have been given the following balls:

Now you are asked to give unique balls. So, you pick up one ball of each color and return it back. The balls you will return are:

This case happens in real life as well. There will be many times when you will be given a lot of repeated values and you would only require unique values. And so in that case we will use SET.
Mathematically,Set is a collection of unique elements and the same thing also holds here.

TreeSet in Java

TreeSet in Java is a class that implements the Set interface and uses a self-balancing binary search tree (Red-Black tree) internally to store unique elements in sorted order (either natural order or by using a specified comparator).

Key Features:

Ordered Collection: Elements in a TreeSet are maintained in sorted order. By default, elements are sorted in their natural ordering, or a custom Comparator can be provided to define the sorting logic.

Uniqueness: Similar to a mathematical set, TreeSet doesn't allow duplicate elements. It's particularly efficient in ensuring uniqueness and maintaining the sorted order of elements.

Immutable Elements: Once added to a TreeSet, the elements themselves cannot be modified. Any modification to the elements' values that affect their ordering would require removal and reinsertion.

Operations: Supports standard set operations such as adding elements (add()), removing elements (remove()), checking for containment (contains()), getting size (size()), iteration through elements, etc.

Performance: Offers efficient log(n) time complexity for most operations like insertion, deletion, and searching.

Advantages of Set

- Set stores unique values, thus there is no scope for having duplicate values.
- Elements are stored in either ascending or descending order, which makes it efficient.
- Sets are dynamic in size, so we don't have to worry about overflowing errors.
- Sets are used to implement many algorithms because it is fast.
- Search in sets happen O(logN) time complexity.
- It is really very fast and efficient to check if the element is present in the set or not as compared to arrays.

Disadvantages of Set

- We can access the elements of arrays by their indexes, but it is not possible in sets. We can access the elements through iterator only.
- Set is not suitable for large data set because it takes O(logN) for basic operations like insertion and deletion, which makes it quite inefficient when we have to add and remove elements quite frequently.
- It is quite difficult to implement a set because of its structure and properties.
- Set uses more memory than arrays or lists because it stores each element in a separate location.

Q. Vertical Order Traversal of a Binary Tree (Leetcode 987)

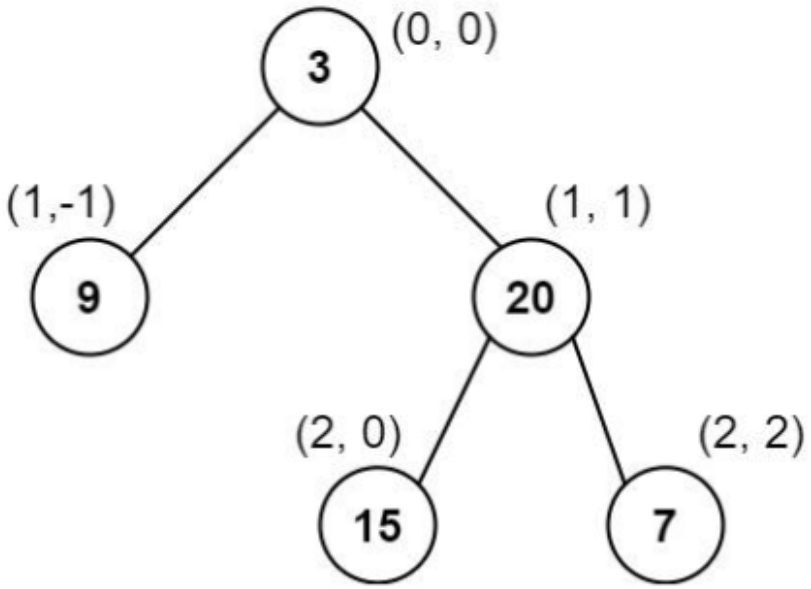
Given the root of a binary tree, calculate the vertical order traversal of the binary tree.

For each node at position (row, col), its left and right children will be at positions (row + 1, col - 1) and (row + 1, col + 1) respectively. The root of the tree is at (0, 0).

The vertical order traversal of a binary tree is a list of top-to-bottom orderings for each column index starting from the leftmost column and ending on the rightmost column. There may be multiple nodes in the same row and same column. In such a case, sort these nodes by their values.

Return the vertical order traversal of the binary tree.

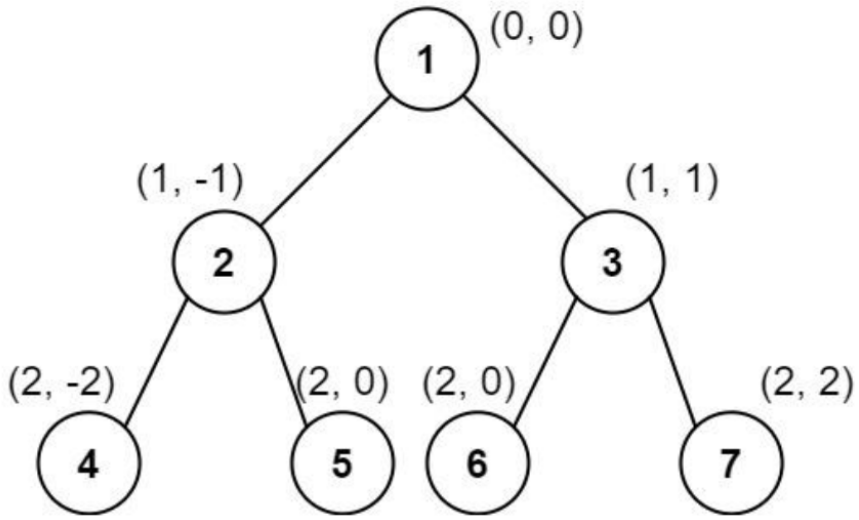
Example 1:



Input: root = [3,9,20,null,null,15,7]
Output: [[9],[3,15],[20],[7]]

Explanation:
Column -1: Only node 9 is in this column.
Column 0: Nodes 3 and 15 are in this column in that order from top to bottom.
Column 1: Only node 20 is in this column.
Column 2: Only node 7 is in this column.

Example 2:



Input: root = [1,2,3,4,5,6,7]
Output: [[4],[2],[1,5,6],[3],[7]]

Explanation:
Column -2: Only node 4 is in this column.
Column -1: Only node 2 is in this column.
Column 0: Nodes 1, 5, and 6 are in this column.
1 is at the top, so it comes first.
5 and 6 are at the same position (2, 0), so we order them by their value, 5 before 6.
Column 1: Only node 3 is in this column.
Column 2: Only node 7 is in this column.

Code:

```
public class Solution {
    public List<List<Integer>> verticalTraversal(TreeNode root) {
        HashMap<Integer, HashMap<Integer, TreeSet<Integer>>>
nodes = new HashMap<>();
        Queue<Pair<TreeNode, Pair<Integer, Integer>>> todo = new
LinkedList<>();
        todo.add(new Pair<>(root, new Pair<>(0, 0)));

        while (!todo.isEmpty()) {
            Pair<TreeNode, Pair<Integer, Integer>> p =
todo.poll();
            TreeNode node = p.getKey();
            int x = p.getValue().getKey(), y =
p.getValue().getValue();

            nodes.computeIfAbsent(x, k -> new HashMap<>())
                 .computeIfAbsent(y, k -> new TreeSet<>())
                 .add(node.val);
```

```
            if (node.left != null) {
                todo.add(new Pair<>(node.left, new Pair<>(x - 1,
y + 1)));
            }
            if (node.right != null) {
                todo.add(new Pair<>(node.right, new Pair<>(x + 1,
y + 1)));
            }
        }

        List<List<Integer>> ans = new ArrayList<>();
        nodes.entrySet().stream()
                .map(Map.Entry::getValue)
                .map(HashMap::entrySet)
                .map(Collection::stream)
                .map(stream ->
stream.sorted(Map.Entry.comparingByKey()))
                .forEachOrdered(stream -> {
                    List<Integer> col = new ArrayList<>();
                    stream.forEach(entry ->
col.addAll(entry.getValue()));
                    ans.add(col);
                });
        return ans;
    }
}
```

Explanation:

Algorithm: Conducts a vertical traversal of a binary tree, storing nodes' values in columns based on their horizontal positions using a map of maps and sets.

Approach: Utilizes a queue for level-order traversal, with a map of maps to categorize nodes by their x-coordinate (horizontal distance) and y-coordinate (level).

Time Complexity: O(N log N) - N: Number of nodes. log N for map operations during traversal.
Space Complexity: O(N) - Space utilized by the map structure and the queue during traversal.

Q. Range frequency queries (Leetcode 2080)

Design a data structure to find the frequency of a given value in a given subarray.

The frequency of a value in a subarray is the number of occurrences of that value in the subarray.

Implement the RangeFreqQuery class:

RangeFreqQuery(int[] arr) Constructs an instance of the class with the given 0-indexed integer array arr.
int query(int left, int right, int value) Returns the frequency of value in the subarray arr[left...right].
A subarray is a contiguous sequence of elements within an array. arr[left...right] denotes the subarray that contains the elements of nums between indices left and right (inclusive).

Example 1:

Input:
["RangeFreqQuery", "query", "query"]
[[[12, 33, 4, 56, 22, 2, 34, 33, 22, 12, 34, 56]], [1, 2, 4], [0, 11, 33]]
Output:
[null, 1, 2]

Explanation:

RangeFreqQuery rangeFreqQuery = new RangeFreqQuery([12, 33, 4, 56, 22, 2, 34, 33, 22, 12, 34, 56]);
rangeFreqQuery.query(1, 2, 4); // return 1. The value 4 occurs 1 time in the subarray [33, 4]
rangeFreqQuery.query(0, 11, 33); // return 2. The value 33 occurs 2 times in the whole array.

Code:

```
public class RangeFreqQuery {
    private Map<Integer, List<Integer>> mp;

    public RangeFreqQuery(int[] arr) {
        mp = new HashMap<>();
        for (int i = 0; i < arr.length; i++) {
            mp.computeIfAbsent(arr[i], k -> new
ArrayList<>()).add(i);
        }
    }

    public int query(int L, int R, int V) {
        int count = 0;
        if (mp.containsKey(V)) {
            List<Integer> indices = mp.get(V);
            count = upperBound(indices, R) - lowerBound(indices,
L);
        }
        return count;
    }

    private int upperBound(List<Integer> list, int target) {
        int low = 0;
        int high = list.size();
        while (low < high) {
            int mid = low + (high - low) / 2;
            if (list.get(mid) <= target) {
                low = mid + 1;
            } else {
                high = mid;
            }
        }
        return low;
    }
```

```
    private int lowerBound(List<Integer> list, int target) {
        int low = 0;
        int high = list.size();
        while (low < high) {
            int mid = low + (high - low) / 2;
            if (list.get(mid) < target) {
                low = mid + 1;
            } else {
                high = mid;
            }
        }
        return low;
    }
}
```

Explanation: We just need to maintain a vector of indices for each element in the order they occur. Then querying for frequency between a range [L, R] is simply finding highest index and lowest index of that element within the given range and returning number of elements between them in the vector of indices. The highest and lowest index can be found using binary search (upper_bound and lower_bound) in O(logN) time.

Time Complexity :
RangeFreqQuery: O(N), to traverse the array
query: O(logN), for upper bound
Space Complexity : O(N), required for maintaining hashmap

Q. Sum of Beauty of All Substrings (Leetcode 1781)
The beauty of a string is the difference in frequencies between the most frequent and least frequent characters.

For example, the beauty of "abaacc" is 3 - 1 = 2.
Given a string s, return the sum of beauty of all of its substrings.

Example 1:

Input: s = "aabcb"
Output: 5

Explanation: The substrings with non-zero beauty are ["aab","aabc","aabcb","abcb","bcb"], each with beauty equal to 1.

Example 2:

Input: s = "aabcbaa"
Output: 17

Code:

```
public class Solution {
    private int beauty(int[] hash) {
        int maxFreq = -1, minFreq = (int)1e9;
        for (int i = 0; i < 26; i++) {
            maxFreq = Math.max(maxFreq, hash[i]);
            if (hash[i] >= 1) {
                minFreq = Math.min(minFreq, hash[i]);
            }
        }
        return maxFreq - minFreq;
    }

    public int beautySum(String s) {
        int res = 0;
        int n = s.length();

        for (int i = 0; i < n; i++) {
            int[] hash = new int[26];
            for (int j = i; j < n; j++) {
                hash[s.charAt(j) - 'a']++;
                res += beauty(hash);
            }
        }
        return res;
    }
}
```

Explanation:

The approach involves iterating through the string, generating substrings at each index, and calculating the minimum and maximum frequency of characters within those substrings. For each substring, the minimum and maximum frequencies are tracked. The final answer represents the difference between the maximum and minimum frequencies encountered among all generated substrings. This process is done by iterating through the string and updating the result based on the frequencies calculated for each substring. In the end, the result signifies the maximum difference between character frequencies within any substring of the given string.

Time Complexity:O(N^2 * 26)
Nested loops iterating through the string and considering the 26 characters of the alphabet.
Space Complexity:O(26) → O(1)
Space used for the hash array with 26 characters, which simplifies to constant space a

Q. Find the Longest Substring Containing Vowels in Even Counts (Leetcode 1371)

Given the string s, return the size of the longest substring containing each vowel an even number of times. That is, 'a', 'e', 'i', 'o', and 'u' must appear an even number of times.

Example 1:

Input: s = "eleetminicoworoep"
Output: 13

Explanation: The longest substring is "leetminicowor" which contains two each of the vowels: e, i and o and zero of the vowels: a and u.

Example 2:

Input: s = "leetcodeisgreat"
Output: 5

Explanation: The longest substring is "leetc" which contains two e's.

Example 3:

Input: s = "bcbcbc"
Output: 6

Explanation: In this case, the given string "bcbcbc" is the longest because all vowels: a, e, i, o and u appear zero times.

Code:

```
public class Solution {
    public int findTheLongestSubstring(String s) {
        Map<Integer, Integer> m = new HashMap<>();
        m.put(0, -1);
        int res = 0, n = s.length(), cur = 0;
        for (int i = 0; i < n; i++) {
            cur ^= 1 << "aeiou".indexOf(s.charAt(i)) + 1 >> 1;
            if (!m.containsKey(cur)) {
                m.put(cur, i);
            }
            res = Math.max(res, i - m.get(cur));
        }
        return res;
    }
}
```

Explanation:
cur records the count of "aeiou"
cur & 1 = the records of a % 2
cur & 2 = the records of e % 2
cur & 4 = the records of i % 2
cur & 8 = the records of o % 2
cur & 16 = the records of u % 2
seen note the index of first occurrence of cur

Note that we don't really need the exact count number,
we only need to know if it's odd or even.

If it's one of aeiou,
'aeiou'.find(c) can find the index of vowel,
cur ^= 1 << 'aeiou'.find(c) will toggle the count of vowel.

But for no vowel characters,
'aeiou'.find(c) will return -1,
that's reason that we do 1 << ('aeiou'.find(c) + 1) >> 1.

Time Complexity: O(N) ,The algorithm iterates through the string once and performs constant time operations for each character.

Space Complexity: O(1), Utilizes a fixed-size unordered map regardless of the input size, resulting in constant space complexity. The space usage doesn't increase with the input length, only keeping track of specific occurrences using bit manipulation.

Q. Delete Nodes And Return Forest (Leetcode 1110)

Given the root of a binary tree, each node in the tree has a distinct value.

After deleting all nodes with a value in to_delete, we are left with a forest (a disjoint union of trees).

Return the roots of the trees in the remaining forest. You may return the result in any order.

Example 1:

Input: root = [1,2,3,4,5,6,7], to_delete = [3,5]
Output: [[1,2,null,4],[6],[7]]

Example 2:

Input: root = [1,2,4,null,3], to_delete = [3]
Output: [[1,2,4]]

Code:

```
public class Solution {
    List<TreeNode> result = new ArrayList<>();
    Set<Integer> toDeleteSet = new HashSet<>();

    public List<TreeNode> delNodes(TreeNode root, int[] toDelete)
{
        for (int i : toDelete)
            toDeleteSet.add(i);
        helper(root, true);
        return result;
    }
```

```
    private TreeNode helper(TreeNode node, boolean isRoot) {
        if (node == null) return null;
        boolean deleted = toDeleteSet.contains(node.val);
        if (isRoot && !deleted) result.add(node);
        node.left = helper(node.left, deleted);
        node.right = helper(node.right, deleted);
        return deleted ? null : node;
    }
}
```

Explanation:
Intuition
As I keep saying in my video "courses",
solve tree problem with recursion first.

If a node is root (has no parent) and isn't deleted,
when will we add it to the result.
Absolutely! That's a concise and accurate summary:

Time Complexity:
delNodes: O(N)
helper: O(N)
Iterates through each node in the tree once.

Space Complexity:
delNodes: O(N + H)
helper: O(H)
H: Height of the tree.
Involves space for the result vector, to_delete_set, and recursive stack space, with the primary factor being the height of the tree.

Multimap in Java

A Multimap in Java is a data structure that associates keys with multiple values. Unlike a Map, it allows duplicate keys and stores multiple values for each key. This structure is useful when one key can map to multiple values and requires organized storage.

Key Features:

Ordered Structure: A Multimap maintains a sorted order based on its keys. In Java, this is often achieved using implementations like TreeMap for key-value associations, ensuring keys are sorted in their natural order or using a custom comparator.

Supports Duplicates: Unlike traditional Map implementations, a Multimap allows duplicate keys and can store multiple values corresponding to each key.

Container Properties: The Java Multimap isn't directly available in the standard library (java.util). However, it can be simulated by using Map<K, List<V>> or third-party libraries like Google Guava's Multimap to efficiently manage collections of values for each key.

Usage Scenarios: It finds application in scenarios where multiple values can be associated with a single key while preserving their ordering. For instance, handling groupings of data where duplicates are permissible and ordered retrieval is required.

Example in Java:

```
public class MultimapExample {
    public static void main(String[] args) {
        TreeMap<Character, ArrayList<Integer>> multimap = new
TreeMap<>();

        // Insert elements into the multimap
        insertIntoMultimap(multimap, 'a', 1);
        insertIntoMultimap(multimap, 'b', 2);
        insertIntoMultimap(multimap, 'a', 3); // Duplicate key
'a'

        // Print the elements
        printMultimap(multimap);
    }

    private static void insertIntoMultimap(TreeMap<Character,
ArrayList<Integer>> multimap, char key, int value) {
        multimap.computeIfAbsent(key, k -> new
ArrayList<>()).add(value);
    }

    private static void printMultimap(TreeMap<Character,
ArrayList<Integer>> multimap) {
        for (Map.Entry<Character, ArrayList<Integer>> entry :
multimap.entrySet()) {
            char key = entry.getKey();
            ArrayList<Integer> values = entry.getValue();
            for (int value : values) {
                System.out.println(key + " : " + value);
            }
        }
    }
}
```

Q. Largest Values From Labels (Leetcode 1090)

There is a set of n items. You are given two integer arrays values and labels where the value and the label of the ith element are values[i] and labels[i] respectively. You are also given two integers numWanted and useLimit.

Choose a subset s of the n elements such that:

The size of the subset s is less than or equal to numWanted.
There are at most useLimit items with the same label in s.
The score of a subset is the sum of the values in the subset.

Return the maximum score of a subset s.

Example 1:

Input: values = [5,4,3,2,1], labels = [1,1,2,2,3], numWanted = 3, useLimit = 1
Output: 9

Explanation: The subset chosen is the first, third, and fifth items.

Example 2:

Input: values = [5,4,3,2,1], labels = [1,3,3,3,2], numWanted = 3, useLimit = 2
Output: 12

Explanation: The subset chosen is the first, second, and third items.

Example 3:

Input: values = [9,8,8,7,6], labels = [0,0,0,1,1], numWanted = 3, useLimit = 1
Output: 16

Explanation: The subset chosen is the first and fourth items.

Code:

```
public class Solution {
    public int largestValsFromLabels(int[] vs, int[] ls, int
wanted, int limit) {
        TreeMap<Integer, Integer> s = new
TreeMap<>(Collections.reverseOrder());
        HashMap<Integer, Integer> m = new HashMap<>();

        for (int i = 0; i < vs.length; ++i) {
            s.put(vs[i], ls[i]);
        }

        int res = 0;
        for (Map.Entry<Integer, Integer> entry : s.entrySet()) {
            if (wanted > 0) {
                int label = entry.getValue();
                m.put(label, m.getOrDefault(label, 0) + 1);
                if (m.get(label) <= limit) {
                    res += entry.getKey();
                    --wanted;
                }
```

```
            } else {
                break;
            }
        }

        return res;
    }
}
```

Explanation: Sort all labels by value. Then, start with the largest value, greedily pick labels. Track how many labels we have used in m, and do not pick that label if we reached the limit.

Time Complexity: O(N * log N)
Constructing the multimap involves sorting by value, which takes O(N * log N) time.

Space Complexity: O(N) Utilizes space for the multimap s and the unordered map m, proportional to the input sizes.

Q. Number of Substrings Containing All Three Characters (Leetcode 1358)

Given a string s consisting only of characters a, b and c.

Return the number of substrings containing at least one occurrence of all these characters a, b and c.

Example 1:

Input: s = "abcabc"
Output: 10

Explanation: The substrings containing at least one occurrence of the characters a, b and c are "abc", "abca", "abcab", "abcabc", "bca", "bcab", "bcabc", "cab", "cabc" and "abc" (again).

Example 2:

Input: s = "aaacb"
Output: 3

Explanation: The substrings containing at least one occurrence of the characters a, b and c are "aaacb", "aacb" and "acb".

Example 3:

Input: s = "abc"
Output: 1

Code:

```
public class Solution {
    public int numberOfSubstrings(String s) {
        Map<Character, Integer> f = new HashMap<>();
        int l = 0, r = 0, cnt = 0;
        int n = s.length();
```

```
        while (r < n) {
            char charR = s.charAt(r);
            f.put(charR, f.getOrDefault(charR, 0) + 1);

            while (f.getOrDefault('a', 0) > 0 &&
f.getOrDefault('b', 0) > 0 && f.getOrDefault('c', 0) > 0) {
                cnt += n - r;
                char charL = s.charAt(l++);
                f.put(charL, f.get(charL) - 1);
            }
            r++;
        }
        return cnt;
    }
}
```

Explanation: using a map(f) for storing the frequency of the characters.
l and r are the left and right pointers each.
Once we reach a point where the frequency of all the characters >0 then the number of substrings possible are size(s) - r. Now shifting the left pointer and calculating the answer for each case while the condition holds that f[a], f[b], f[c] > 0

Time Complexity: O(N)
The algorithm iterates through the string once using two pointers, l and r, to count the number of substrings containing 'a', 'b', and 'c'.

Space Complexity: O(1)
Utilizes only a fixed-size map (f) storing the counts of 'a', 'b', and 'c', regardless of the input size.

Q. Number of Submatrices That Sum to Target (Leetcode 1074)

Given a matrix and a target, return the number of non-empty submatrices that sum to target.

A submatrix x1, y1, x2, y2 is the set of all cells matrix[x][y] with x1 <= x <= x2 and y1 <= y <= y2.

Two submatrices (x1, y1, x2, y2) and (x1', y1', x2', y2') are different if they have some coordinate that is different: for example, if x1 != x1'.

Example 1:

0	1	0
1	1	1
0	1	0

Input: matrix = [[0,1,0],[1,1,1],[0,1,0]], target = 0
Output: 4

Explanation: The four 1x1 submatrices that only contain 0.

Example 2:

Input: matrix = [[1,-1],[-1,1]], target = 0
Output: 5

Explanation: The two 1x2 submatrices, plus the two 2x1 submatrices, plus the 2x2 submatrix.

Example 3:

Input: matrix = [[904]], target = 0
Output: 0

Explanation: The two 1x2 submatrices, plus the two 2x1 submatrices, plus the 2x2 submatrix.

Code:

```
public class Solution {
    public int numSubmatrixSumTarget(int[][] A, int target) {
        int res = 0, m = A.length, n = A[0].length;
        for (int i = 0; i < m; i++) {
            for (int j = 1; j < n; j++) {
                A[i][j] += A[i][j - 1];
            }
        }

        for (int i = 0; i < n; i++) {
            for (int j = i; j < n; j++) {
                Map<Integer, Integer> counter = new HashMap<>();
                counter.put(0, 1);
                int cur = 0;
                for (int k = 0; k < m; k++) {
                    cur += A[k][j] - (i > 0 ? A[k][i - 1] : 0);
                    res += counter.getOrDefault(cur - target, 0);
                    counter.put(cur, counter.getOrDefault(cur, 0)
+ 1);
                }
            }
        }
        return res;
    }
}
```

Explanation: For each row, calculate the prefix sum.
For each pair of columns, calculate the accumulated sum of rows.
Now this problem is same to, "Find the Subarray with Target Sum".

Time Complexity:
O(M * N^2)
M: Number of rows in the matrix.
N: Number of columns in the matrix.
The nested loops iterate through the matrix elements (O(M * N)) and perform additional operations within each iteration, leading to a complexity of O(M * N^2).

Space Complexity: O(M), Utilizes space for the unordered map counter, where the maximum size could grow proportional to the number of rows (M) in the matrix.

Q. LRU CACHE (Leetcode 146)

Design a data structure that follows the constraints of a Least Recently Used (LRU) cache.

Implement the LRUCache class:

LRUCache(int capacity) Initialize the LRU cache with positive size capacity.
int get(int key) Return the value of the key if the key exists, otherwise return -1.
void put(int key, int value) Update the value of the key if the key exists. Otherwise, add the key-value pair to the cache. If the number of keys exceeds the capacity from this operation, evict the least recently used key.
The functions get and put must each run in O(1) average time complexity.

Example 1:

Input
["LRUCache", "put", "put", "get", "put", "get", "put", "get", "get", "get"]
[[2], [1, 1], [2, 2], [1], [3, 3], [2], [4, 4], [1], [3], [4]]
Output
[null, null, null, 1, null, -1, null, -1, 3, 4]

Explanation:
LRUCache lRUCache = new LRUCache(2);
lRUCache.put(1, 1); // cache is {1=1}
lRUCache.put(2, 2); // cache is {1=1, 2=2}
lRUCache.get(1);   // return 1
lRUCache.put(3, 3); // LRU key was 2, evicts key 2, cache is {1=1, 3=3}
lRUCache.get(2);   // returns -1 (not found)
lRUCache.put(4, 4); // LRU key was 1, evicts key 1, cache is {4=4, 3=3}
lRUCache.get(1);   // return -1 (not found)
lRUCache.get(3);   // return 3
lRUCache.get(4);   // return 4

Code:

```
public class LRUCache {
   int capacity;
   LinkedHashMap<Integer, Integer> cache;

   public LRUCache(int capacity) {
```

```
        this.capacity = capacity;
        this.cache = new LinkedHashMap<Integer,
Integer>(capacity, 0.75f, true) {
            protected boolean
removeEldestEntry(Map.Entry<Integer, Integer> eldest) {
                return size() > capacity;
            }
        };
    }

    public int get(int key) {
        return cache.getOrDefault(key, -1);
    }

    public void put(int key, int value) {
        cache.put(key, value);
    }
}
```

Explanation: An LRU cache is built by combining two data structures: a doubly linked list and a hash map.

We'll set up our linked list with the most-recently used item at the head of the list and the least-recently used item at the tail:

This lets us access the LRU element in O(1)O(1) time by looking at the tail of the list.

What about accessing a specific item in the cache ?

In general, finding an item in a linked list is O(n) time, since we need to walk the whole list. But the whole point of a cache is to get quick lookups. How could we speed that up?

We'll add in a hash map that maps items to linked list nodes i.e we;ll map each key with its corresponding iterator of the linked list

Time Complexity:

- LRUCache Constructor: O(1)
- get: O(1)
- put: O(1)

Space Complexity:

- LRUCache: O(N)
- N: Capacity of the LRUCache (Capacity variable)
- Utilizes space for the doubly linked list dq and the unordered map mp. The space grows proportionally with the capacity of the cache.

Q. Given an array containing only 0s and 1s, find the largest subarray which contains equal no of 0s and 1s. The expected time complexity is O(n).

Code:

```
public class MaxEqualSubarray {

    public static int maxEqualSubarray(List<Integer> nums) {
        Map<Integer, Integer> count = new HashMap<>();
        int maxLength = 0;
        int sum = 0;
        count.put(0, -1);

        for (int i = 0; i < nums.size(); ++i) {
            sum += (nums.get(i) == 0) ? -1 : 1;

            if (count.containsKey(sum)) {
                maxLength = Math.max(maxLength, i -
count.get(sum));
            } else {
                count.put(sum, i);
            }
        }

        return maxLength;
    }

    public static void main(String[] args) {
        List<Integer> nums = Arrays.asList(0, 1, 0, 1, 1, 0, 1,
0);
        System.out.println("Largest subarray with equal 0s and 1s
length: " + maxEqualSubarray(nums));
    }
}
```

Explanation:

- The code employs a hashmap (unordered_map) to keep track of the cumulative sum and its corresponding index.
- It traverses the input array, treating 0 as -1 and 1 as 1, summing these values.
- If the sum is encountered again, it calculates the length of the subarray by subtracting the previous index where the sum occurred.
- It updates the maximum length of the subarray whenever a sum is found again or records the first occurrence of the sum in the hashmap.

Time Complexity:

- maxEqualSubarray: O(N)
- N: Number of elements in the input array.
- The algorithm iterates through the array once, performing constant time operations for each element.

Space Complexity:

- maxEqualSubarray: O(N)
- N: Number of elements in the input array.
- Utilizes space for the unordered map count, which grows with the number of elements in the array to store cumulative sums and their indices.

Q. Given an array of size N and an integer K, return the count of distinct numbers in all windows of size K.

Input:
Array: [1, 2, 1, 3, 4, 2, 3]
K (Window Size): 4

Output:
Count of distinct numbers in windows of size 4: [3, 4, 4, 3]

Code:

```
 import java.util.*;

public class CountDistinctNumbers {

    public static List<Integer>
countDistinctNumbers(List<Integer> nums, int k) {
        Map<Integer, Integer> count = new HashMap<>();
        List<Integer> result = new ArrayList<>();
        int distinctCount = 0;

        for (int i = 0; i < k; ++i) {
            int num = nums.get(i);
            count.put(num, count.getOrDefault(num, 0) + 1);
            if (count.get(num) == 1) {
                distinctCount++;
            }
        }
        result.add(distinctCount);

        for (int i = k; i < nums.size(); ++i) {
            int num = nums.get(i - k);
            count.put(num, count.get(num) - 1);
            if (count.get(num) == 0) {
                distinctCount--;
            }

            num = nums.get(i);
            count.put(num, count.getOrDefault(num, 0) + 1);
            if (count.get(num) == 1) {
                distinctCount++;
            }

            result.add(distinctCount);
        }

        return result;
    }

    public static void main(String[] args) {
```

```
        List<Integer> nums = Arrays.asList(1, 2, 1, 3, 4, 2, 3);
        int k = 4;
        List<Integer> result = countDistinctNumbers(nums, k);

        System.out.print("Count of distinct numbers in windows of
size " + k + ": [");
        for (int i = 0; i < result.size(); ++i) {
            System.out.print(result.get(i));
            if (i != result.size() - 1) {
                System.out.print(", ");
            }
        }
        System.out.println("]");
    }
}
```

Explanation:

The code utilizes a sliding window approach to count distinct numbers in each window of size K.
It employs an unordered map to keep track of counts of each number within the current window.
Initially, it counts distinct numbers in the first window of size K and stores the count in the result vector.
Then, it slides the window by one element at a time, updating the distinct count accordingly and storing the counts in the result vector.

Time Complexity: O(N), The algorithm iterates through the array once with linear complexity.

Space Complexity: O(N), Utilises space for the unordered map count and the result vector, both proportional to the input size.

Q. Given an array A[] of N positive integers and 3 integers X, Y, and Z, the task is to check if there exist 4 indices (say p, q, r, s) such that the following conditions are satisfied:

0 < p < q < r < s < N
Sum of the subarray from A[p] to A[q – 1] is X
Sum of the subarray from A[q] to A[r – 1] is Y
Sum of the subarray from A[r] to A[s – 1] is Z

Input: N = 10, A[] = {1, 3, 2, 2, 2, 3, 1, 4, 3, 2}, X = 5, Y = 7, Z = 5
Output: YES

Explanation: The 4 integers p, q, r, s are {1, 3, 6, 8}.

A[1] + A[2] = 5
A[3] + A [4] + A[5] = 7
A[6] + A[7] = 5

Input: N = 9, A[] = {31, 41, 59, 26, 53, 58, 97, 93, 23}, X = 100, Y = 101, Z = 100
Output: NO

Code:

```
 import java.util.*;

public class Main {
    static boolean isPossible(int N, int[] A, int X, int Y, int
Z) {
        Set<Integer> S = new HashSet<>();
        S.add(0);
        int curr = 0;
        for (int i = 0; i < N; i++) {
            curr += A[i];
            S.add(curr);
        }
        for (int it : S) {
            if (S.contains(it + X) && S.contains(it + X + Y) &&
S.contains(it + X + Y + Z)) {
                return true;
            }
        }
        return false;
    }

    public static void main(String[] args) {
        int N = 10, X = 5, Y = 7, Z = 5;
        int[] A = { 1, 3, 2, 2, 2, 3, 1, 4, 3, 2 };

        boolean answer = isPossible(N, A, X, Y, Z);
        if (answer) {
            System.out.println("YES");
        } else {
            System.out.println("NO");
        }
    }
}
```

Explanation:

The solution approach utilizes cumulative sums and a set. It first calculates cumulative sums by iterating through the array, storing these sums in a set. The key step involves iterating through this set, searching for elements that satisfy the conditions required for the four indices (p, q, r, s). To achieve this, it checks if a particular element in the set, along with three other elements (also present in the set), meets the given conditions without explicitly using binary searches each time. If a valid sequence of indices is found, the function returns true; otherwise, it returns false. This approach streamlines the search process by utilizing the precalculated cumulative sums stored in the set.

Time Complexity (O(N * log(N))): Due to the nested iteration through the set with three find() operations, each with a logarithmic time complexity in relation to the size of the set (N).
Auxiliary Space (O(N)): The set S used to store cumulative sums grows in proportion to the input size N, occupying O(N) space.

PW SKILLS

THANK YOU !

PW SKILLS