PW SKILLS

Lesson Plan

Maps & Sets

PW SKILLS

Introduction to HashSets

In Java, a HashSet is a part of the Java Collections Framework and is a part of the java.util package. It is a collection that does not allow duplicate elements, and it does not guarantee the order of elements. The primary purpose of a HashSet is to provide constant-time performance for basic operations like add, remove, contains, and size, assuming the hash function distributes elements properly.

Here are some key points about HashSet:

- No Duplicates: A HashSet does not allow duplicate elements. If you attempt to add an element that already exists in the set, the add method will return false, and the set remains unchanged.
- No Order Guarantee: The order of elements in a HashSet is not guaranteed to be in any particular order. If you need a specific order, you can use LinkedHashSet, which maintains the order in which elements are inserted.
- Null Values: A HashSet allows a single null element. However, if you attempt to add a duplicate null, it will still return false.
- Backed by HashMap: Internally, a HashSet is backed by a HashMap instance, where the elements are the keys, and the values are a constant dummy object (PRESENT). The use of a hash table allows for constant-time average complexity for basic operations.

STL and important methods in Hashsets

add(Object o)

- Adds the specified element to the HashSet if it is not already present.
- Returns true if the element was added successfully, and false if the element is already present.

```
HashSet<String> hashSet = new HashSet<>();
boolean isAdded = hashSet.add("Apple");
System.out.println("Is 'Apple' added successfully? " + isAdded);
```

size()

- Returns the number of elements in the HashSet.

```
HashSet<String> hashSet = new HashSet<>();
hashSet.add("Apple");
hashSet.add("Banana");
int size = hashSet.size();
System.out.println("Size of HashSet: " + size);
```

contains(Object o)

- Returns true if the HashSet contains the specified element.
- Returns false otherwise.

```
HashSet<String> hashSet = new HashSet<>();
hashSet.add("Apple");
boolean containsApple = hashSet.contains("Apple");
System.out.println("Does the HashSet contain 'Apple'? " + containsApple);
```

iterator()

- Returns an iterator over the elements in the HashSet.
- The elements are returned in no particular order

```
HashSet<String> hashSet = new HashSet<>();
hashSet.add("Apple");
hashSet.add("Banana");

Iterator<String> iterator = hashSet.iterator();
while (iterator.hasNext()) {
   String element = iterator.next();
   System.out.println("Element: " + element);
}
```

remove(Object o)

- Removes the specified element from the HashSet if it is present.
- Returns true if the element was removed successfully, and false if the element was not found.

```
HashSet<String> hashSet = new HashSet<>();
hashSet.add("Apple");
boolean isRemoved = hashSet.remove("Apple");
System.out.println("Is 'Apple' removed successfully? " + isRemoved);
```

toArray()

- Returns an array containing all the elements in the HashSet.
- The order of the elements in the array is not guaranteed to be the same as the order in the HashSet.

```
HashSet<String> hashSet = new HashSet<>();
hashSet.add("Apple");
hashSet.add("Banana");

Object[] array = hashSet.toArray();
System.out.println("Array of HashSet elements: " + Arrays.toString(array));
```

How to iterate in Hashset

iterate over a HashSet in Java using either an enhanced for loop or an iterator. Here are examples of both methods:

Using Enhanced For Loop:

Code:

```
import java.util.HashSet;

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

```
        // Creating a HashSet
        HashSet<String> hashSet = new HashSet<>();

        // Adding elements
        hashSet.add("Apple");
        hashSet.add("Banana");
        hashSet.add("Orange");

        // Iterating using enhanced for loop
        System.out.println("Iterating using enhanced for loop:");
        for (String element : hashSet) {
            System.out.println(element);
        }
    }
}
```

Using Iterator:

```
import java.util.HashSet;
import java.util.Iterator;

public class HashSetIterationExample {
    public static void main(String[] args) {
        // Creating a HashSet
        HashSet<String> hashSet = new HashSet<>();

        // Adding elements
        hashSet.add("Apple");
        hashSet.add("Banana");
        hashSet.add("Orange");

        // Iterating using Iterator
        System.out.println("Iterating using Iterator:");
        Iterator<String> iterator = hashSet.iterator();
        while (iterator.hasNext()) {
            String element = iterator.next();
            System.out.println(element);
        }
    }
}
```

Both methods will output:

Iterating using enhanced for loop:
Banana
Orange
Apple

Iterating using Iterator:
Banana
Orange
Apple

Remember that the order of elements in a HashSet is not guaranteed to be in any particular order. If you need to maintain the order of insertion, consider using a LinkedHashSet.

Q. Count Number of Distinct Integers After ReverseOperations [Leetcode - 2442]

To count the number of distinct integers after reverse operations, we need to perform a series of operations on an array of integers. The operations are as follows:

Start with an array of integers.

Perform an operation on the array:

- Reverse the subarray from index i to j (inclusive).

Repeat the operation for different subarrays.

After performing all the operations, count the number of distinct integers in the resulting array.

```
class Solution{
public static int rev(int n){
  int rev = 0;
  while (n > 0) {
   rev = rev * 10 + (n % 10);
   n /= 10;
  }
  return rev;
}
 public int countDistinctIntegers(int[] nums) {
  Set<Integer> set = new HashSet<>();
  for(int x : nums){
   set.add(x);
   set.add(rev(x));
  }
  return set.size();
 }
}
```

In this example, the countDistinctAfterReverse method takes an array of integers (nums) and an array of operations (operations). It applies the reverse operations on the array and then counts the number of distinct integers in the modified array using a HashSet.
Note: The reverseSubarray method is used to reverse the subarray specified by the given indices (start and end)

Q. Find Maximum Number of String Pairs [Leetcode - 2744]

Problem Statement: You are given two arrays words1 and words2. A string pair (i, j) is valid if words1[i] + words2[j] is a palindrome.
Return the maximum number of valid pairs you can form with the given arrays.

Example:

Input: words1 = ["abcd","dcba","lls","s","sssll"]
words2 = ["bat","tab","cat"]

Output: 5

Explanation: The valid pairs are ("abcd","dcba"), ("dcba","abcd"), ("lls","s"), ("s","lls"), ("sssll","tab").

Here is a Java solution to solve this problem:

```
class Solution
{
    public int maximumNumberOfStringPairs(String[] words)
    {
        // creating a dic for each unique pair(26 * 26)
        boolean[] dictionary = new boolean[676];
        int count = 0;

        for (String w : words)
        {
            // current word index
            int pairIndex1 = w.charAt(0) - 'a' + (w.charAt(1) -
'a') * 26;
            // reversed word index
            int pairIndex2 = w.charAt(1) - 'a' + (w.charAt(0) -
'a') * 26;

            // checking if reversed word already encountered
before or not?
            count += dictionary[pairIndex2] ? 1 : 0;
            // encountering current word
            dictionary[pairIndex1] = true;
        }

        return count;
    }
}
```

This solution uses a Map to store the count of reverse strings from words1. Then, it iterates through words2 to count the valid pairs. The isPalindrome method checks if a given string is a palindrome.
This solution has a time complexity of O(N * M), where N is the length of words1 and M is the length of words2.

Q. Find Maximum Number of String Pairs [Leetcode - 2744]

Problem Statement: You are given two arrays words1 and words2. A string pair (i, j) is valid if words1[i] + words2[j] is a palindrome.
Return the maximum number of valid pairs you can form with the given arrays.

Example:

Input: words1 = ["abcd","dcba","lls","s","sssll"]
words2 = ["bat","tab","cat"]

Output: 5

Explanation: The valid pairs are ("abcd","dcba"), ("dcba","abcd"), ("lls","s"), ("s","lls"), ("sssll","tab").

Here is a Java solution to solve this problem:

```
class Solution
{
    public int maximumNumberOfStringPairs(String[] words)
    {
        // creating a dic for each unique pair(26 * 26)
        boolean[] dictionary = new boolean[676];
        int count = 0;

        for (String w : words)
        {
            // current word index
            int pairIndex1 = w.charAt(0) - 'a' + (w.charAt(1) -
'a') * 26;
            // reversed word index
            int pairIndex2 = w.charAt(1) - 'a' + (w.charAt(0) -
'a') * 26;

            // checking if reversed word already encountered
before or not?
            count += dictionary[pairIndex2] ? 1 : 0;
            // encountering current word
            dictionary[pairIndex1] = true;
        }

        return count;
    }
}
```

This solution uses a Map to store the count of reverse strings from words1. Then, it iterates through words2 to count the valid pairs. The isPalindrome method checks if a given string is a palindrome.
This solution has a time complexity of O(N * M), where N is the length of words1 and M is the length of words2.

Q. Find Maximum Number of String Pairs [Leetcode - 2744]

Problem Statement: You are given two arrays words1 and words2. A string pair (i, j) is valid if words1[i] + words2[j] is a palindrome.
Return the maximum number of valid pairs you can form with the given arrays.

Example:

Input: words1 = ["abcd","dcba","lls","s","sssll"]
words2 = ["bat","tab","cat"]

Output: 5

Explanation: The valid pairs are ("abcd","dcba"), ("dcba","abcd"), ("lls","s"), ("s","lls"), ("sssll","tab").

Here is a Java solution to solve this problem:

```
class Solution
{
    public int maximumNumberOfStringPairs(String[] words)
    {
        // creating a dic for each unique pair(26 * 26)
        boolean[] dictionary = new boolean[676];
        int count = 0;

        for (String w : words)
        {
            // current word index
            int pairIndex1 = w.charAt(0) - 'a' + (w.charAt(1) -
'a') * 26;
            // reversed word index
            int pairIndex2 = w.charAt(1) - 'a' + (w.charAt(0) -
'a') * 26;

            // checking if reversed word already encountered
before or not?
            count += dictionary[pairIndex2] ? 1 : 0;
            // encountering current word
            dictionary[pairIndex1] = true;
        }

        return count;
    }
}
```

This solution uses a Map to store the count of reverse strings from words1. Then, it iterates through words2 to count the valid pairs. The isPalindrome method checks if a given string is a palindrome.
This solution has a time complexity of O(N * M), where N is the length of words1 and M is the length of words2.

Introduction to HashMaps

A HashMap in Java is a part of the Java Collections Framework, which is a set of interfaces and classes that provide high-performance, resizable implementations of various data structures. Specifically, a HashMap is an implementation of the Map interface, which represents a collection of key-value pairs.
Here's an introduction to HashMaps:

Key-Value Pairs:

- A HashMap stores data in key-value pairs.
- Each key is unique within the HashMap.
- The key is used to retrieve the associated value.

Null Keys and Values:

- A HashMap allows one null key and multiple null values.
- It ensures that keys are unique, so attempting to insert a duplicate key will overwrite the existing value.

Unordered Collection:

- HashMap does not guarantee any specific order of elements.
- If you need to maintain insertion order, you can use LinkedHashMap, or if you need elements sorted by their keys, you can use TreeMap.

Performance:

- HashMap provides constant-time performance for basic operations such as get, put, and remove on average, assuming a good hash function and proper sizing.
- The underlying data structure is an array of buckets, and each bucket is a linked list (or a tree in case of collisions).

Hashing:

- The keys in a HashMap are hashed using their hashCode method to determine the index of the bucket in which the key-value pair will be stored.
- Hash collisions (when two keys hash to the same index) are handled by using linked lists (or trees) at each bucket.

Iterating Over Entries:

- You can iterate over the entries of a HashMap using various methods, such as key set, entry set, or by using Java 8 streams.

STL and important methods in maps

put(K key, V value)

- Adds a key-value pair to the HashMap.
- If the key already exists, the new value will overwrite the existing one.
- Returns the previous value associated with the key, or null if the key was not present.

```
Map<String, Integer> hashMap = new HashMap<>();
hashMap.put("One", 1);
```

get(Object key)

- Returns the value associated with the specified key.
- Returns null if the key is not present.

```
int value = hashMap.get("One");
```

size()

- Returns the number of key-value mappings in the HashMap

```
int size = hashMap.size();
```

containsKey(Object key)

- Returns true if the HashMap contains the specified key, otherwise returns false.

```
boolean containsKey = hashMap.containsKey("One");
```

containsValue(Object value)

- Returns true if the HashMap contains the specified value, otherwise returns false.

```
boolean containsValue = hashMap.containsValue(1);
```

remove(Object key)

- Removes the mapping for the specified key from the HashMap.
- Returns the previous value associated with the key, or null if the key was not present.

```
int removedValue = hashMap.remove("One");
```

entrySet()

- Returns a set of the mappings contained in the HashMap.
- Each element in the set is a Map.Entry<K, V> object representing a key-value pair.

```
Set<Map.Entry<String, Integer>> entrySet = hashMap.entrySet();
```

These methods provide essential functionalities for working with HashMaps in Java, allowing you to add, retrieve, remove, and check for the presence of key-value pairs, among other operations.

How to iterate in Hashmap

There are multiple ways to iterate over a HashMap in Java. Here are three common approaches using keySet, values, and entrySet:

1. Iterating over keys using keySet():

```
import java.util.HashMap;
import java.util.Map;

public class HashMapIterationExample {
    public static void main(String[] args) {
        // Creating a HashMap
        Map<String, Integer> hashMap = new HashMap<>();
        hashMap.put("One", 1);
        hashMap.put("Two", 2);
        hashMap.put("Three", 3);

        // Iterating over keys using keySet()
        System.out.println("Iterating over keys using
keySet():");
        for (String key : hashMap.keySet()) {
            System.out.println("Key: " + key + ", Value: " +
hashMap.get(key));
        }
    }
}
```

2. Iterating over values using values():

```
import java.util.HashMap;
import java.util.Map;

public class HashMapIterationExample {
    public static void main(String[] args) {
        // Creating a HashMap
        Map<String, Integer> hashMap = new HashMap<>();
        hashMap.put("One", 1);
        hashMap.put("Two", 2);
        hashMap.put("Three", 3);

        // Iterating over values using values()
        System.out.println("Iterating over values using
values():");
        for (Integer value : hashMap.values()) {
            System.out.println("Value: " + value);
        }
    }
}
```

3. Iterating over entries using entrySet():

```
import java.util.HashMap;
import java.util.Map;

public class HashMapIterationExample {
    public static void main(String[] args) {
        // Creating a HashMap
        Map<String, Integer> hashMap = new HashMap<>();
        hashMap.put("One", 1);
        hashMap.put("Two", 2);
        hashMap.put("Three", 3);

        // Iterating over entries using entrySet()
        System.out.println("Iterating over entries using
entrySet():");
        for (Map.Entry<String, Integer> entry :
hashMap.entrySet()) {
            System.out.println("Key: " + entry.getKey() + ",
Value: " + entry.getValue());
        }
    }
}
```

Each approach has its own use case. If you need both keys and values, or if you want to perform operations on both, using entrySet() is often more efficient because it avoids unnecessary lookups. If you only need keys or values, then using keySet() or values() is a more concise choice.

Q. Valid Anagram [Leetcode - 242]

An anagram is a word or phrase formed by rearranging the letters of another.
Here's a Java solution for the problem:
import java.util.Arrays;

```
class Solution {
    public boolean isAnagram(String s, String t) {
        if(s.length()≠t.length()){
            return false;
        }
        int map[]=new int[26];
        for(char x : s.toCharArray()){
            map[x-'a']++;
        }
        for(char x : t.toCharArray()){
            map[x-'a']--;
        }
        for(int i:map){
            if (i≠0){
                return false;
            }
        }
        return true;
    }
}
```

This solution first checks if the lengths of the two strings are equal. If they are not, the strings cannot be anagrams. If the lengths are equal, it converts both strings to character arrays, sorts them, and then compares the sorted arrays.
This solution has a time complexity of O(n log n) due to the sorting operation, where n is the length of the strings. Alternatively, you can use a frequency table or a hash map to achieve a linear time solution.
Here's an alternative approach using a frequency table:

```
import java.util.HashMap;
import java.util.Map;

public class ValidAnagram {
    public static void main(String[] args) {
        String s = "anagram";
        String t = "nagaram";

        boolean result = isAnagram(s, t);
        System.out.println("Are the strings anagrams? " +
result);
    }

    private static boolean isAnagram(String s, String t) {
        if (s.length() != t.length()) {
            return false;
        }

        // Create frequency tables for both strings
        Map<Character, Integer> frequencyS = new HashMap<>();
        Map<Character, Integer> frequencyT = new HashMap<>();

        // Update frequency table for string s
        for (char ch : s.toCharArray()) {
            frequencyS.put(ch, frequencyS.getOrDefault(ch, 0) +
1);
        }

        // Update frequency table for string t
        for (char ch : t.toCharArray()) {
            frequencyT.put(ch, frequencyT.getOrDefault(ch, 0) +
1);
        }

        // Compare frequency tables
        return frequencyS.equals(frequencyT);
    }
}
```

This alternative approach has a linear time complexity of O(n), where n is the length of the strings.

Q. Two Sum [Leetcode - 1]

Given an array of integers nums and an integer target, return indices of the two numbers such that they add up to target.
You may assume that each input would have **exactly one solution**, and you may not use the same element twice.
You can return the answer in any order.

Here's a simple Java solution using a HashMap for efficient lookup:

```
class Solution {
    public int[] twoSum(int[] nums, int target) {
        Map<Integer, Integer> numMap = new HashMap<>();
        int n = nums.length;

        for (int i = 0; i < n; i++) {
            int complement = target - nums[i];
            if (numMap.containsKey(complement)) {
                return new int[]{numMap.get(complement), i};
            }
            numMap.put(nums[i], i);
        }

        return new int[]{}; // No solution found
    }
}
```

Q. Unique Number of Occurrences [Leetcode - 1207]

Given an array of integers arr, return true if the number of occurrences of each value in the array is **unique** or false otherwise.

Example 1:
Input: arr = [1,2,2,1,1,3]
Output: true

Explanation: The value 1 has 3 occurrences, 2 has 2 and 3 has 1. No two values have the same number of occurrences.

Example 2:

Input: arr = [1,2]
Output: false

Example 3:

Input: arr = [-3,0,1,-3,1,1,1,-3,10,0]
Output: true

Constraints:

- 1 <= arr.length <= 1000
- -1000 <= arr[i] <= 1000

```java
class Solution {
    public boolean uniqueOccurrences(int[] arr) {

        HashMap<Integer, Integer> map = new HashMap<>();

        for(int val : arr) {
            map.put(val, map.getOrDefault(val,0)+1);
        }

        HashSet<Integer> set = new HashSet<>(map.values());
        return map.size() == set.size();
    }
}
```

Q. Finding 3-Digit Even Numbers [Leetcode - 2094]

You are given an integer array digits, where each element is a digit. The array may contain duplicates.

You need to find **all** the **unique** integers that follow the given requirements:

- The integer consists of the **concatenation** of **three** elements from digits in **any** arbitrary order.
- The integer does not have **leading zeros**.
- The integer is **even**.

For example, if the given digits were [1, 2, 3], integers 132 and 312 follow the requirements.

Return a **sorted** array of the unique integers.

Example 1:

Input: digits = [2,1,3,0]
Output: [102,120,130,132,210,230,302,310,312,320]

Explanation: All the possible integers that follow the requirements are in the output array.
Notice that there are no **odd** integers or integers with **leading zeros**.

Example 2:

Input: digits = [2,2,8,8,2]
Output: [222,228,282,288,822,828,882]

Explanation: The same digit can be used as many times as it appears in digits.
In this example, the digit 8 is used twice each time in 288, 828, and 882.

Example 3:

Input: digits = [3,7,5]
Output: []

Explanation: No **even** integers can be formed using the given digits.

Constraints:

- 3 <= digits.length <= 100
- 0 <= digits[i] <= 9

```
public int[] findEvenNumbers(int[] digits) {
 var unique = new ArrayList<Integer>();
 var count = new int[10];

 for (var digit : digits)
  count[digit]++;

 for (var i = 100; i < 999; i += 2) {
  var unit = i % 10;
  var ten = (i / 10) % 10;
  var hundred = (i / 100) % 10;
  count[unit]--;
  count[ten]--;
  count[hundred]--;

  if (count[unit] >= 0 && count[ten] >= 0 && count[hundred] >= 0)
   unique.add(i);

  // restore the hash for next iteration
  count[unit]++;
  count[ten]++;
  count[hundred]++;
 }

 return unique.stream()
     .mapToInt(i -> i)
     .toArray();
}
```

This Java program defines a findEvenCombinations method that generates all permutations of three digits, checks for valid even numbers without leading zeros, and returns the sorted list of unique even numbers. The generateCombinations method recursively generates all permutations. The main method demonstrates the usage with the given input array.

Q. Finding Pairs with a Certain Sum [Leetcode - 1865]

You are given two integer arrays nums1 and nums2. You are tasked to implement a data structure that supports queries of two types:

1. **Add** a positive integer to an element of a given index in the array nums2.
2. **Count** the number of pairs (i, j) such that nums1[i] + nums2[j] equals a given value (0 <= i < nums1.length and 0 <= j < nums2.length).

Implement the FindSumPairs class:

- FindSumPairs(int[] nums1, int[] nums2) Initializes the FindSumPairs object with two integer arrays nums1 and nums2.
- void add(int index, int val) Adds val to nums2[index], i.e., apply nums2[index] += val.
- int count(int tot) Returns the number of pairs (i, j) such that nums1[i] + nums2[j] == tot.

Example 1:

Input:
["FindSumPairs", "count", "add", "count", "count", "add", "add", "count"]
[[[1, 1, 2, 2, 2, 3], [1, 4, 5, 2, 5, 4]], [7], [3, 2], [8], [4], [0, 1], [1, 1], [7]]
Output:
[null, 8, null, 2, 1, null, null, 11]

Explanation:

FindSumPairs findSumPairs = new FindSumPairs([1, 1, 2, 2, 2, 3], [1, 4, 5, 2, 5, 4]);
findSumPairs.count(7); // return 8; pairs (2,2), (3,2), (4,2), (2,4), (3,4), (4,4) make 2 + 5 and pairs (5,1), (5,5) make 3 + 4
findSumPairs.add(3, 2); // now nums2 = [1,4,5,**4**,5,4]
findSumPairs.count(8); // return 2; pairs (5,2), (5,4) make 3 + 5
findSumPairs.count(4); // return 1; pair (5,0) makes 3 + 1
findSumPairs.add(0, 1); // now nums2 = [**2**,4,5,4,5,4]
findSumPairs.add(1, 1); // now nums2 = [2,**5**,5,4,5,4]
findSumPairs.count(7); // return 11; pairs (2,1), (2,2), (2,4), (3,1), (3,2), (3,4), (4,1), (4,2), (4,4) make 2 + 5 and pairs (5,3), (5,5) make 3 + 4

Constraints:

- 1 <= nums1.length <= 1000
- 1 <= nums2.length <= 105
- 1 <= nums1[i] <= 109
- 1 <= nums2[i] <= 105
- 0 <= index < nums2.length
- 1 <= val <= 105
- 1 <= tot <= 109
- At most 1000 calls are made to add and count **each**.

```
class FindSumPairs {
    int[] nums1, nums2;
    HashMap<Integer, Integer> freq = new HashMap<>();

    public FindSumPairs(int[] nums1, int[] nums2) {
        this.nums1 = nums1;
        this.nums2 = nums2;
        for (int x : nums2) increaseFreq(x, 1);
    }
```

```
    private void increaseFreq(int key, int inc) {
        freq.put(key, freq.getOrDefault(key, 0) + inc);
    }
    public void add(int index, int val) {
        increaseFreq(nums2[index], -1);  // Remove old one
        nums2[index] += val;
        increaseFreq(nums2[index], 1);  // Count new one
    }
    public int count(int tot) {
        int ans = 0;
        for (int a : nums1)
            ans += freq.getOrDefault(tot - a, 0); // a + b = tot
→ b = tot - a
        return ans;
    }
}
```

Q. Longest Substring without Repeating Characters. [Leetcode - 3]

Given a string s, find the length of the **longest**

substring: without repeating characters.

Example 1:

Input: s = "abcabcbb"
Output: 3

Explanation: The answer is "abc", with the length of 3.

Example 2:

Input: s = "bbbbb"
Output: 1

Explanation: The answer is "b", with the length of 1.

Example 3:

Input: s = "pwwkew"
Output: 3

Explanation: The answer is "wke", with the length of 3.
Notice that the answer must be a substring, "pwke" is a subsequence and not a substring.

Constraints:

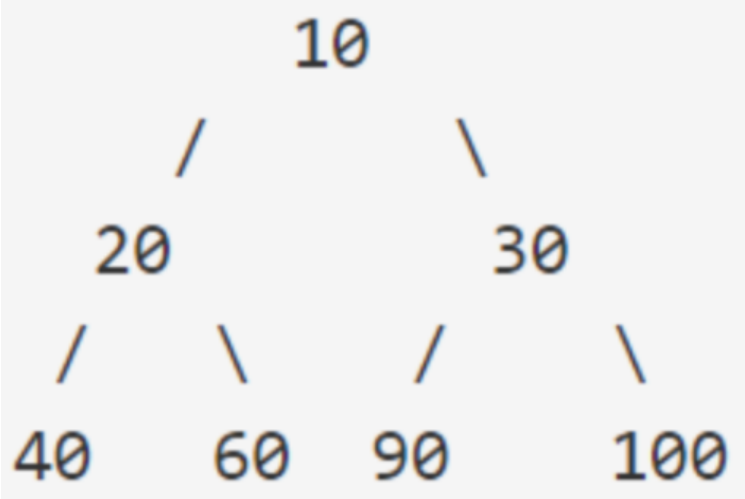
- 0 <= s.length <= 5 * 104
- s consists of English letters, digits, symbols and spaces.

Code:

```
class Solution {
    public int lengthOfLongestSubstring(String s) {
        int n = s.length();
        int maxLength = 0;
        Set<Character> charSet = new HashSet<>();
        int left = 0;

        for (int right = 0; right < n; right++) {
            if (!charSet.contains(s.charAt(right))) {
                charSet.add(s.charAt(right));
                maxLength = Math.max(maxLength, right - left +
1);
            } else {
                while (charSet.contains(s.charAt(right))) {
                    charSet.remove(s.charAt(left));
                    left++;
                }
                charSet.add(s.charAt(right));
            }
        }

        return maxLength;
    }
}
```

Top view of binary tree

The top view of a binary tree refers to the set of nodes visible when the tree is viewed from the top. It is essentially the set of nodes that are visible when looking down from the root of the tree. Nodes at the same horizontal distance from the root are considered to be at the same level, and only the topmost node at each level is included in the top view.
To print the top view of a binary tree, you need to traverse the tree in a way that allows you to identify the topmost node at each level. One way to achieve this is to perform a level-order traversal (BFS) of the tree and maintain a map that stores the topmost node at each horizontal distance.



Output: 40 20 10 30 100

eg.
Here is a Java implementation to print the top view of a binary tree:

```
import java.util.*;

class TreeNode {
    int val;
    TreeNode left, right;

    public TreeNode(int val) {
        this.val = val;
        this.left = this.right = null;
    }
}

public class TopViewBinaryTree {
    public static void topView(TreeNode root) {
        if (root == null)
            return;

        class QueueNode {
            TreeNode node;
            int horizontalDistance;

            public QueueNode(TreeNode node, int
horizontalDistance) {
                this.node = node;
                this.horizontalDistance = horizontalDistance;
            }
        }

        Map<Integer, Integer> topViewMap = new TreeMap<>();
        Queue<QueueNode> queue = new LinkedList<>();

        // Perform a level-order traversal
        queue.add(new QueueNode(root, 0));

        while (!queue.isEmpty()) {
            QueueNode qNode = queue.poll();
            TreeNode node = qNode.node;
            int hd = qNode.horizontalDistance;

            // Store the topmost node at each horizontal distance
            if (!topViewMap.containsKey(hd)) {
                topViewMap.put(hd, node.val);
            }

            // Enqueue left and right children with updated
horizontal distances
            if (node.left != null) {
```

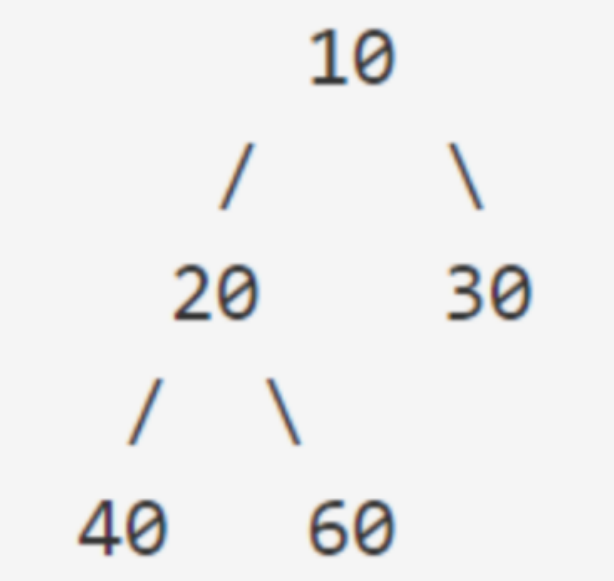
```
                queue.add(new QueueNode(node.left, hd - 1));
            }
            if (node.right != null) {
                queue.add(new QueueNode(node.right, hd + 1));
            }
        }

        // Print the top view nodes
        for (int value : topViewMap.values()) {
            System.out.print(value + " ");
        }
    }

    public static void main(String[] args) {
        /*
               1
             /   \
            2     3
             \
              4
               \
                5
        */
        TreeNode root = new TreeNode(1);
        root.left = new TreeNode(2);
        root.right = new TreeNode(3);
        root.left.right = new TreeNode(4);
        root.left.right.right = new TreeNode(5);

        System.out.println("Top view of the binary tree:");
        topView(root);
    }
}
```

In this example, the topView method prints the top view of the binary tree. The QueueNode class is used to store the node and its horizontal distance during the BFS traversal. The TreeMap is used to maintain the topmost node at each horizontal distance, ensuring that the nodes are sorted by their horizontal distance. The output will be the values of the top view nodes in the order they appear from left to right.

Bottom view of binary tree

The bottom view of a binary tree refers to the set of nodes visible when the tree is viewed from the bottom. It is essentially the set of nodes that are visible when looking up from the bottom of the tree. Nodes at the same horizontal distance from the root are considered to be at the same level, and only the bottommost node at each level is included in the bottom view.

To print the bottom view of a binary tree, you need to traverse the tree in a way that allows you to identify the bottommost node at each level. One way to achieve this is to perform a level-order traversal (BFS) of the tree and maintain a map that stores the bottommost node at each horizontal distance.



Output: 40 20 60 30

```
import java.util.*;

class TreeNode {
    int val;
    TreeNode left, right;

    public TreeNode(int val) {
        this.val = val;
        this.left = this.right = null;
    }
}

public class BottomViewBinaryTree {
    public static void bottomView(TreeNode root) {
        if (root == null)
            return;

        class QueueNode {
            TreeNode node;
            int horizontalDistance;

            public QueueNode(TreeNode node, int
horizontalDistance) {
                this.node = node;
                this.horizontalDistance = horizontalDistance;
            }
        }

        Map<Integer, Integer> bottomViewMap = new TreeMap<>();
        Queue<QueueNode> queue = new LinkedList<>();

        // Perform a level-order traversal
        queue.add(new QueueNode(root, 0));
```

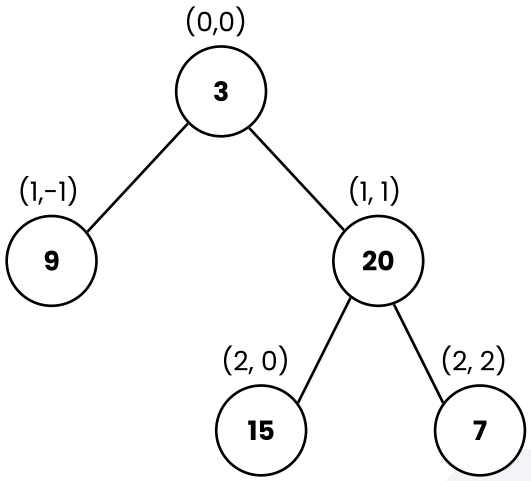
```
        while (!queue.isEmpty()) {
            QueueNode qNode = queue.poll();
            TreeNode node = qNode.node;
            int hd = qNode.horizontalDistance;

            // Update the bottommost node at each horizontal
distance
            bottomViewMap.put(hd, node.val);

            // Enqueue left and right children with updated
horizontal distances
            if (node.left != null) {
                queue.add(new QueueNode(node.left, hd - 1));
            }
            if (node.right != null) {
                queue.add(new QueueNode(node.right, hd + 1));
            }
        }

        // Print the bottom view nodes
        for (int value : bottomViewMap.values()) {
            System.out.print(value + " ");
        }
    }

    public static void main(String[] args) {
        /*
               1
             /   \
            2     3
             \
              4
               \
                5
        */
        TreeNode root = new TreeNode(1);
        root.left = new TreeNode(2);
        root.right = new TreeNode(3);
        root.left.right = new TreeNode(4);
        root.left.right.right = new TreeNode(5);

        System.out.println("Bottom view of the binary tree:");
        bottomView(root);
    }
}
```

In this example, the bottomView method prints the bottom view of the binary tree. The QueueNode class is used to store the node and its horizontal distance during the BFS traversal. The TreeMap is used to maintain the bottommost node at each horizontal distance, ensuring that the nodes are sorted by their horizontal distance. The output will be the values of the bottom view nodes in the order they appear from left to right.

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.

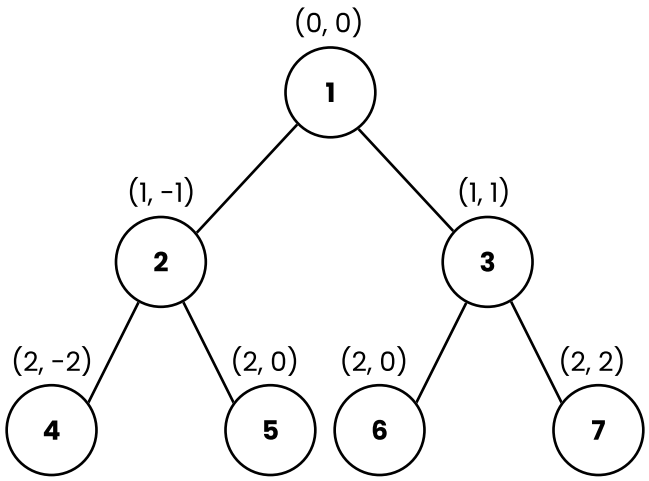
Example 1:



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:

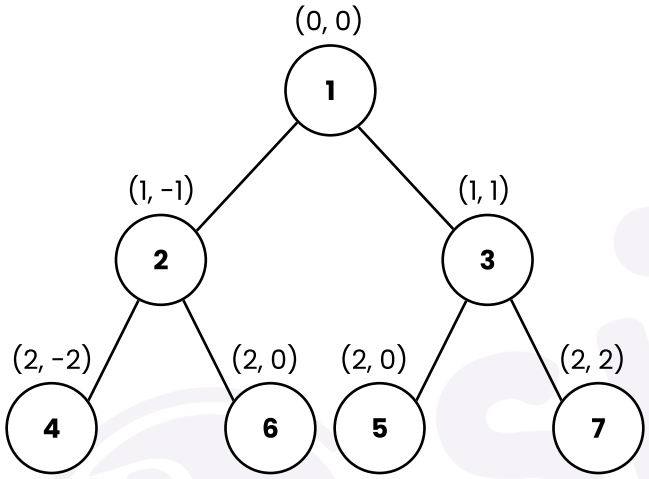


Input: root = [1,2,3,4,5,6,7]
Output: [[4],[2],[1,5,6],[3],[7]]

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.

Example 3:



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

Explanation: This case is the exact same as example 2, but with nodes 5 and 6 swapped.
Note that the solution remains the same since 5 and 6 are in the same location and should be ordered by their values.

Constraints:

- The number of nodes in the tree is in the range [1, 1000].
- 0 <= Node.val <= 1000

Code:

```
class Solution {
    Map<Integer, TreeMap<Integer, PriorityQueue<Integer>>>
levelItems;
    int minIndex, maxIndex;
    public List<List<Integer>> verticalTraversal(TreeNode root) {
        if (root == null) return new ArrayList<>();
```

```
        this.levelItems = new HashMap<>();
        this.minIndex = Integer.MAX_VALUE;
        this.maxIndex = Integer.MIN_VALUE;

        dfs(root, 0, 0);

        List<List<Integer>> res = new ArrayList<>();
        for (int i = minIndex; i <= maxIndex; i++) {
            List<Integer> innerList = new ArrayList<>();
            TreeMap<Integer, PriorityQueue<Integer>> treeMap =
levelItems.get(i);
            for (int row : treeMap.keySet()) {
                PriorityQueue<Integer> pq = treeMap.get(row);
                while (!pq.isEmpty()) {
                    innerList.add(pq.poll());
                }
            }
            res.add(innerList);
        }
        return res;
    }

    private void dfs(TreeNode node, int row, int col) {
        if (node == null) return;
        minIndex = Math.min(minIndex, col);
        maxIndex = Math.max(maxIndex, col);
        levelItems.computeIfAbsent(col, k -> new TreeMap<>())
                .computeIfAbsent(row, k -> new
PriorityQueue<>()).offer(node.val);
        dfs(node.left, row + 1, col - 1);
        dfs(node.right, row + 1, col + 1);
    }
}
```

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.

```
class RangeFreqQuery {
    Map<Integer, TreeMap<Integer, Integer>> map = new
HashMap<>();
    public RangeFreqQuery(int[] arr) {
        for(int i = 0; i < arr.length;i++){
            map.putIfAbsent(arr[i], new TreeMap<>());
            map.get(arr[i]).put(i, map.get(arr[i]).size());
        }

    }

    public int query(int left, int right, int value) {
        if(!map.containsKey(value)) return 0;
        TreeMap<Integer, Integer> nums = map.get(value);
        Integer a = nums.ceilingKey(left), b =
nums.floorKey(right);
        if(a == null || b == null) return 0;
        return nums.get(b) - nums.get(a) +1;
    }
}
```

Count Nice Pairs in an Array Leetcode 1814

You are given an array nums that consists of non-negative integers. Let us define rev(x) as the reverse of the non-negative integer x. For example, rev(123) = 321, and rev(120) = 21. A pair of indices (i, j) is nice if it satisfies all of the following conditions:

- 0 <= i < j < nums.length
- nums[i] + rev(nums[j]) == nums[j] + rev(nums[i])

Return the number of nice pairs of indices. Since that number can be too large, return it modulo 109 + 7.

Example 1:

Input: nums = [42,11,1,97]
Output: 2

Explanation:
The two pairs are:
- (0,3) : 42 + rev(97) = 42 + 79 = 121, 97 + rev(42) = 97 + 24 = 121.
- (1,2) : 11 + rev(1) = 11 + 1 = 12, 1 + rev(11) = 1 + 11 = 12.

Example 2:

Input: nums = [13,10,35,24,76]
Output: 4

Constraints:

- 1 <= nums.length <= 1e5
- 0 <= nums[i] <= 1e9

```
import java.util.HashMap;
import java.util.Collection;
import java.util.Map;

class Solution {
    public int countNicePairs(int[] nums) {
        /// Create a HashMap to store occurrences of temporary
numbers
        Map<Integer, Integer> numbers = new HashMap<>();

        /// Iterate through the array
        for (int num : nums) {
            /// Calculate the temporary number
            int temporary_number = num - reverse(num);

            /// Update the count in the HashMap
            if (numbers.containsKey(temporary_number)) {
                numbers.put(temporary_number,
numbers.get(temporary_number) + 1);
            } else {
                numbers.put(temporary_number, 1);
            }
        }

        /// Calculate the total number of nice pairs
        long result = 0;
        Collection<Integer> values = numbers.values();
        int mod = 1000000007;
        for (int value : values) {
            result = (result % mod + ((long) value * ((long)
value - 1) / 2)) % mod;
        }

        // Return the result as an integer
        return (int) result;
    }

    /// Helper function to reverse a number
    private int reverse(int number) {
        int reversed_number = 0;
```

```
        while (number > 0) {
            reversed_number = reversed_number * 10 + number % 10;
            number /= 10;
        }
        return reversed_number;
    }
}
```

Unique Length-3 Palindromic Subsequences [Leetcode 1930]

Given a string s, return the number of **unique palindromes of length three** that are a **subsequence** of s.

Note that even if there are multiple ways to obtain the same subsequence, it is still only counted **once**.

A **palindrome** is a string that reads the same forwards and backwards.
A **subsequence** of a string is a new string generated from the original string with some characters (can be none) deleted without changing the relative order of the remaining characters.

- For example, "ace" is a subsequence of "abcde".

Example 1:

Input: s = "aabca"
Output: 3

Explanation: The 3 palindromic subsequences of length 3 are:
- "aba" (subsequence of "aabca")
- "aaa" (subsequence of "aabca")
- "aca" (subsequence of "aabca")

Example 2:

Input: s = "adc"
Output: 0

Explanation: There are no palindromic subsequences of length 3 in "adc".

Example 3:

Input: s = "bbcbaba"
Output: 4

Explanation: The 4 palindromic subsequences of length 3 are:
- "bbb" (subsequence of "bbcbaba")
- "bcb" (subsequence of "bbcbaba")
- "bab" (subsequence of "bbcbaba")
- "aba" (subsequence of "bbcbaba")

Constraints:

- 3 <= s.length <= 105
- s consists of only lowercase English letters.

```
class Solution {
    public int countPalindromicSubsequence(String inputString) {
        int firstIndex[] = new int[26], lastIndex[] = new
int[26], result = 0;
        Arrays.fill(firstIndex, Integer.MAX_VALUE);
        for (int i = 0; i < inputString.length(); ++i) {
            firstIndex[inputString.charAt(i) - 'a'] =
Math.min(firstIndex[inputString.charAt(i) - 'a'], i);
            lastIndex[inputString.charAt(i) - 'a'] = i;
        }
        for (int i = 0; i < 26; ++i)
            if (firstIndex[i] < lastIndex[i])
                result += inputString.substring(firstIndex[i] +
1, lastIndex[i]).chars().distinct().count();
        return result;
    }
}
```

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

Constraints:

- 1 <= s.length <= 500
- s consists of only lowercase English letters.

```
class Solution {
    // Function to calculate the beauty of a substring based on
character frequencies
    public static int calculateBeauty(int[] charFreq) {
        int minFreq = Integer.MAX_VALUE;
        int maxFreq = 0;
```

```
        // Iterate through the character frequency array
        for (int freq : charFreq) {
            if (freq > maxFreq) {
                maxFreq = freq;
            }
            if (freq > 0 && freq < minFreq) {
                minFreq = freq;
            }
        }

        // Calculate and return the beauty of the substring
        return maxFreq - minFreq;
    }

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

        // Iterate through all possible substrings
        for (int i = 0; i < length; i++) {
            // Initialize an array to store character frequencies
for the current substring
            int[] charFreq = new int[26]; // Assuming lowercase
English letters

            for (int j = i; j < length; j++) {
                // Update the character frequency array based on
characters in the substring
                charFreq[s.charAt(j) - 'a']++;

                // Calculate the beauty of the current substring
and add it to the total beauty
                totalBeauty += calculateBeauty(charFreq);
            }
        }

        // Return the total sum of beauty for all substrings
        return totalBeauty;
    }
}
```

Count Substrings That Differ by One Character [Leetcode 1638]

Given two strings s and t, find the number of ways you can choose a non-empty substring of s and replace a **single character** by a different character such that the resulting substring is a substring of t. In other words, find the number of substrings in s that differ from some substring in t by **exactly** one character.
For example, the underlined substrings in "computer" and "computation" only differ by the 'e'/'a', so this is a valid way.
Return the number of substrings that satisfy the condition above.
A **substring** is a contiguous sequence of characters within a string.

Example 1:

Input: s = "aba", t = "baba"
Output: 6

Explanation: The following are the pairs of substrings from s and t that differ by exactly 1 character:
("aba", "baba")
("aba", "baba")
("aba", "baba")
("aba", "baba")
("aba", "baba")
("aba", "baba")
The underlined portions are the substrings that are chosen from s and t.

Example 2:

Input: s = "ab", t = "bb"
Output: 3

Explanation: The following are the pairs of substrings from s and t that differ by 1 character:
("ab", "bb")
("ab", "bb")
("ab", "bb")
The underlined portions are the substrings that are chosen from s and t.

Constraints:

- 1 <= s.length, t.length <= 100
- s and t consist of lowercase English letters only.

```
    public int countSubstrings(String s, String t) {
        int res = 0 ;
        for (int i = 0; i < s.length(); ++i)
            res += helper(s, t, i, 0);
        for (int j = 1; j < t.length(); ++j)
            res += helper(s, t, 0, j);
        return res;
    }


    public int helper(String s, String t, int i, int j) {
        int res = 0, pre = 0, cur = 0;
        for (int n = s.length(), m = t.length(); i < n && j < m;
++i, ++j) {
            cur++;
            if (s.charAt(i) != t.charAt(j)) {
                pre = cur;
                cur = 0;
            }
            res += pre;
        }
        return res;
    }
```

Determine if Two Strings Are Close [Leetcode 1657]

Two strings are considered close if you can attain one from the other using the following operations:

- **Operation 1:** Swap any two **existing** characters.
  - For example, abcde -> aecdb
- Operation 2: Transform **every** occurrence of one **existing** character into another **existing** character, and do the same with the other character.
  - For example, aacabb -> bbcbaa (all a's turn into b's, and all b's turn into a's)

You can use the operations on either string as many times as necessary.
Given two strings, word1 and word2, return true if word1 and word2 are **close**, and false otherwise.

Example 1:

Input: word1 = "abc", word2 = "bca"
Output: true

Explanation: You can attain word2 from word1 in 2 operations.
Apply Operation 1: "abc" -> "acb"
Apply Operation 1: "acb" -> "bca"

Example 2:

Input: word1 = "a", word2 = "aa"
Output: false

Explanation: It is impossible to attain word2 from word1, or vice versa, in any number of operations.

Example 3:

Input: word1 = "cabbba", word2 = "abbccc"
Output: true

Explanation: You can attain word2 from word1 in 3 operations.
Apply Operation 1: "cabbba" -> "caabbb"
Apply Operation 2: "caabbb" -> "baaccc"
Apply Operation 2: "baaccc" -> "abbccc"

Constraints:

- 1 <= word1.length, word2.length <= 105
- word1 and word2 contain only lowercase English letters.

```
class Solution {
    public int co(char arr[],int key){
        int c=0;
        for(int i=0;i<arr.length;i++){
            if(arr[i]==key){
                c++;
                arr[i]='-';
```

```
            }
        }
        return c;
    }
    public int[] count(char arr[]){
        int n=arr.length;
        int[] arrN=new int[n];
        for(int i=0;i<n;i++){
            if(arr[i]=='-'){
                continue;
            }
            arrN[i]=co(arr,arr[i]);
        }

        Arrays.sort(arrN);

        return arrN;
    }
    public boolean closeStrings(String w1, String w2) {
        char[] c1 = w1.toCharArray();
        char[] c2 = w2.toCharArray();
        for(int i=0;i<c1.length&&i<c2.length;i++){
            if(w1.indexOf(c2[i])==-1){
                return false;
            }
        }
        Arrays.sort(c1);
        Arrays.sort(c2);
        if(Arrays.equals(c1, c2)){
            return true;
        }

        int arr1[]=count(c1);
        int arr2[]=count(c2);
        if(Arrays.equals(arr1, arr2)){
            return true;
        }
        return false;
    }
}
```

Max Number of K-Sum Pairs [Leetcode 1679]

You are given an integer array nums and an integer k.

In one operation, you can pick two numbers from the array whose sum equals k and remove them from the array.

Return the maximum number of operations you can perform on the array.

Example 1:

Input: nums = [1,2,3,4], k = 5
Output: 2

Explanation: Starting with nums = [1,2,3,4]:
- Remove numbers 1 and 4, then nums = [2,3]
- Remove numbers 2 and 3, then nums = []
There are no more pairs that sum up to 5, hence a total of 2 operations.

Example 2:

Input: nums = [3,1,3,4,3], k = 6
Output: 1

Explanation: Starting with nums = [3,1,3,4,3]:
- Remove the first two 3's, then nums = [1,4,3]
There are no more pairs that sum up to 6, hence a total of 1 operation.

Constraints:

- 1 <= nums.length <= 105
- 1 <= nums[i] <= 109
- 1 <= k <= 109

```
class Solution {
    public int maxOperations(int[] nums, int k) {
        Arrays.sort(nums);
        int i=0;
        int j=nums.length-1;
        int count=0;

        while(i<j && i<nums.length && j>=0)
        {
            int sum=nums[i]+nums[j];
            if(sum==k)
            {
                count++;
                i++;
                j--;
            }

            else if(sum<k)
            {
                i++;
            }
            else if(sum>k)
            {
```

```
                j--;
            }
        }
        return count;
    }
}
```

Check If Array Pairs Are Divisible by k [Leetcode 1497]

Given an array of integers arr of even length n and an integer k.
We want to divide the array into exactly n / 2 pairs such that the sum of each pair is divisible by k.
Return true If you can find a way to do that or false otherwise.

Example 1:

Input: arr = [1,2,3,4,5,10,6,7,8,9], k = 5
Output: true

Explanation: Pairs are (1,9),(2,8),(3,7),(4,6) and (5,10).

Example 2:

Input: arr = [1,2,3,4,5,6], k = 7
Output: true

Explanation: Pairs are (1,6),(2,5) and(3,4).

Example 3:

Input: arr = [1,2,3,4,5,6], k = 10
Output: false

Explanation: You can try all possible pairs to see that there is no way to divide arr into 3 pairs each with sum divisible by 10.

Constraints:

- arr.length == n
- 1 <= n <= 1e5
- n is even.
- -1e9 <= arr[i] <= 1e9
- 1 <= k <= 1e5

```
class Solution {
    public boolean canArrange(int[] arr, int k) {
        int[] frequency = new int[k];
        for(int num : arr){
            num %= k;
            if(num < 0) num += k;
            frequency[num]++;
        }
        if(frequency[0]%2 != 0) return false;

        for(int i = 1; i <= k/2; i++)
            if(frequency[i] != frequency[k-i]) return false;
        return true;
    }
}
```

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.

Constraints:

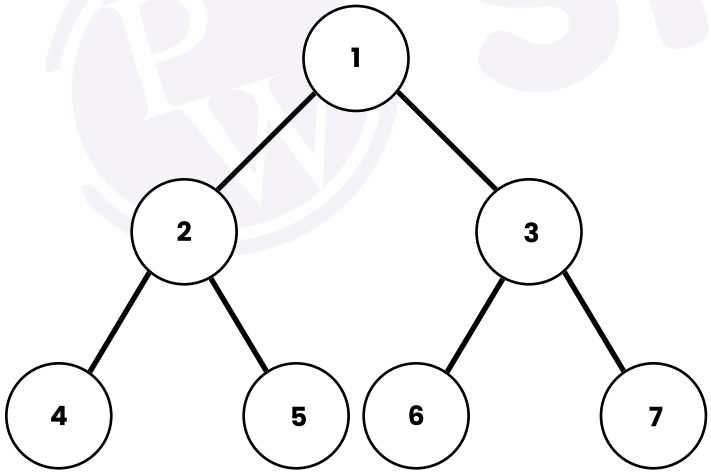
- 1 <= s.length <= 5 x 10^5
- s contains only lowercase English letters.

```
class Solution {
    public int findTheLongestSubstring(String s) {
        int[] map = new int[32];
        Arrays.fill(map, -2);
        map[0] = -1;
        int n = s.length(), mask = 0, len = 0;
        for (int i = 0; i < n; i++) {
            char ch = s.charAt(i);
            if (ch == 'a' || ch == 'e' || ch == 'i' || ch == 'o'
|| ch == 'u') mask ^= 1 << (4 -
                    (ch == 'a' ? 0 : ch == 'e' ? 1 : ch == 'i' ?
2 : ch == 'o' ? 3 : 4));
            if (map[mask] == -2) map[mask] = i;
            else len = Math.max(len, i - map[mask]);
        }
        return len;
    }
}
```

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]]

Constraints:

- The number of nodes in the given tree is at most 1000.
- Each node has a distinct value between 1 and 1000.
- to_delete.length <= 1000
- to_delete contains distinct values between 1 and 1000.

```
   Set<Integer> to_delete_set;
    List<TreeNode> res;
    public List<TreeNode> delNodes(TreeNode root, int[]
to_delete) {
        to_delete_set = new HashSet<>();
        res = new ArrayList<>();
        for (int i : to_delete)
            to_delete_set.add(i);
        helper(root, true);
        return res;
    }

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

What is a multimap?

A multimap, short for "multiple map," is a data structure that allows multiple values to be associated with a single key. In other words, it is a variation of a map or dictionary where each key can have multiple associated values. This contrasts with a regular map or dictionary, where each key is associated with a single value.
In a multimap, the relationship between keys and values is often expressed as a one-to-many mapping. This can be useful in scenarios where you need to store and manage a collection of values for each key. Multimaps are particularly helpful in situations where you want to group related data together under the same key.
In programming, multimap-like structures are available in various languages and libraries:

```
import com.google.common.collect.ArrayListMultimap;
import com.google.common.collect.Multimap;

public class MultiMapExample {
    public static void main(String[] args) {
        // Create a Multimap where keys are integers and values
are strings
        Multimap<Integer, String> myMultimap =
ArrayListMultimap.create();
```

```
        // Add key-value pairs to the Multimap
        myMultimap.put(1, "apple");
        myMultimap.put(2, "banana");
        myMultimap.put(1, "apricot");
        myMultimap.put(3, "cherry");
        myMultimap.put(2, "blueberry");
        myMultimap.put(3, "cranberry");

        // Print the contents of the Multimap
        System.out.println("Multimap contents:");
        myMultimap.entries().forEach(entry →
                System.out.println("Key: " + entry.getKey() + ",
Value: " + entry.getValue())
        );

        // Retrieve values for a specific key
        int keyToRetrieve = 2;
        System.out.println("\nValues for key " + keyToRetrieve +
": " + myMultimap.get(keyToRetrieve));
    }
}
```

In this example:

We create a Multimap using ArrayListMultimap.create(). This specific implementation allows multiple values for the same key and preserves the insertion order.
We add key-value pairs to the multimap using put().
We print the contents of the multimap using entries().
Finally, we retrieve all values associated with a specific key using get().

Run this program, and you'll see how the Multimap allows you to associate multiple values with the same key:

Multimap contents:
Key: 1, Value: apple
Key: 1, Value: apricot
Key: 2, Value: banana
Key: 2, Value: blueberry
Key: 3, Value: cherry
Key: 3, Value: cranberry

Values for key 2: [banana, blueberry]

As you can see, the Multimap provides a convenient way to work with scenarios where you need to associate multiple values with a single key.

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 2:

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.

Constraints:

- n == values.length == labels.length
- 1 <= n <= 2 * 104
- 0 <= values[i], labels[i] <= 2 * 104
- 1 <= numWanted, useLimit <= n

```
class Solution {

    class Pair{
        int num;
        int label;
        Pair(int num, int label){
            this.num = num;
            this.label = label;
```

```
        }
    }

    public int largestValsFromLabels(int[] val, int[] lab, int
numWanted, int useLimit) {
        HashMap<Integer,Integer> hp = new HashMap<>();
        ArrayList<Pair> arr = new ArrayList<>();

        for(int i=0; i<lab.length; i++){
            arr.add(new Pair(val[i],lab[i]));
        }

        Collections.sort(arr,(a,b)->(b.num-a.num));
        int ans = 0;
        for(int i=0; i<arr.size(); i++){
            Pair p = arr.get(i);
            int num = p.num;
            int label = p.label;

            if(numWanted>0){
                if(!hp.containsKey(label)){
                    ans = ans+num;
                    hp.put(label,1);
                    numWanted--;
                }else{
                    if(hp.get(label)<useLimit){
                        ans = ans + num;
                        hp.put(label,hp.get(label)+1);
                        numWanted--;
                    }
                }
            }
        }
        return ans;
    }
}
```

Explanation:

We use a priority queue (pq) to store values-labels pairs in descending order of values. This allows us to always pick the pair with the largest value.

We use a map (labelCount) to keep track of the count of each label used so far.

We iterate through the priority queue until num_wanted becomes zero or the queue is empty. In each iteration, we check if the label count for the current pair is less than the use_limit. If it is, we add the value to the result, increment the label count, and decrement num_wanted.

Finally, we return the total result.

This algorithm ensures that we pick the largest values while respecting the constraints on the number of labels used.

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

Constraints:

- 3 <= s.length <= 5 x 10^4
- s only consists of a, b or c characters.

```
public int numberOfSubstrings(String s) {
     int count[] = {0, 0, 0}, res = 0 , i = 0, n = s.length();
     for (int j = 0; j < n; ++j) {
         ++count[s.charAt(j) - 'a'];
         while (count[0] > 0 && count[1] > 0 && count[2] > 0)
             --count[s.charAt(i++) - 'a'];
         res += i;
     }
     return res;
 }
```

Explanation:
We use an array lastIndices to store the last index of each character ('a', 'b', 'c') encountered so far.
We maintain a sliding window [left, right] and iterate through the string from left to right.
At each position right, we update the lastIndices array based on the current character.
We check if the current window is valid, i.e., it contains at least one occurrence of each character ('a', 'b', 'c'). If not, we move the left pointer to the right until the window becomes valid.
The number of valid substrings ending at the current position is calculated based on the length of the current window (left + 1) and added to the result.

Finally, we return the total result.
This algorithm has a time complexity of O(n), where n is the length of the input string, as each character is processed once.

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

Constraints:

- 1 <= matrix.length <= 100
- 1 <= matrix[0].length <= 100
- -1000 <= matrix[i] <= 1000
- -10^8 <= target <= 10^8

```
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];
        Map<Integer, Integer> counter = new HashMap<>();
        for (int i = 0; i < n; i++) {
            for (int j = i; j < n; j++) {
                counter.clear();
                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:
We calculate the prefix sum for each row in the matrix array.
We iterate over all possible pairs of columns (col1, col2).
For each pair of columns, we use a hashmap (prefixSumFreq) to store the frequency of prefix sums for the submatrix formed by the columns col1 and col2.
We iterate over all rows and calculate the sum of the submatrix. We keep track of the current sum and check if there is a prefix sum with the difference (currentSum - target). If yes, we update the result.
We update the prefix sum frequency map.
The final result contains the total number of submatrices with the sum equal to the target.
This solution has a time complexity of O(cols^2 * rows) and a space complexity of O(rows).

PW SKILLS

THANK YOU !

PW SKILLS