Worksheet - 11

Ques 1 : https://leetcode.com/problems/find-duplicate-subtrees/

Code :

```
class Solution {
   public List<TreeNode> findDuplicateSubtrees(TreeNode root) {
      List<TreeNode> res=new ArrayList<>();
      HashMap<String,Integer> hm=new HashMap<>();
      helper(res,root,hm);
      return res;
   }
   public String helper(List<TreeNode> res,TreeNode root,HashMap<String,Integer> hm){
      if(root==null)
         return "";
      String left=helper(res,root.left,hm);
      String right=helper(res,root.right,hm);
      int currroot=root.val;
      String stringformed=currroot+"$"+left+"$"+right;
      if(hm.getOrDefault(stringformed,0)==1){
         res.add(root);
      }
      hm.put(stringformed,hm.getOrDefault(stringformed,0)+1);
      return stringformed;
   }
}
```

Explanation :

This Java code defines a solution to find duplicate subtrees in a binary tree. The method findDuplicateSubtrees takes the root of a tree as input and returns a list of all subtrees that occur more than once.

1. Helper Function: The main logic is in the recursive helper function, which traverses the tree and serializes each subtree (i.e., converts each subtree into a string representation). It concatenates the current node's value and the string representations of its left and right subtrees in the format currroot$left$right. This uniquely identifies each subtree structure.
2. HashMap for Counting: A HashMap<String, Integer> (hm) is used to track how many times each subtree string has been seen. If a subtree's serialized form has been seen exactly once before, it adds the root of that subtree to the result list res.
3. Base Case: When the helper reaches a null node, it returns an empty string.

4. Detecting Duplicates: For each subtree, if its serialized form has appeared exactly once before (hm.getOrDefault(stringformed, 0) == 1), it is considered a duplicate, and its root is added to the result list.

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

Ques 2 : https://leetcode.com/problems/insert-into-a-binary-search-tree/
Code :

```
class Solution {
   public TreeNode insertIntoBST(TreeNode root, int val) {
      root =insert(root,val);
      return root;
   }
   public TreeNode insert(TreeNode  temp, int val){
      if(temp==null)
      {
         return new TreeNode(val);
      }
      else{
         if(temp.val>val)
         temp.left= insert(temp.left,val);
         else
          temp.right= insert(temp.right,val);
      }
      return temp;
   }
}
```

Explanation :
Input: root = [**4,2,7,1,3**], **val** = **5**
Output: [4,2,7,1,3,5]

Ques 3 : https://leetcode.com/problems/longest-word-in-dictionary/
Code :

```
class Solution {
   public String longestWord(String[] words) {
      Arrays.sort(words);
      Set<String> built = new HashSet<String>();
      String res = "";
```

```
        for (String w : words) {
            if (w.length() == 1 || built.contains(w.substring(0, w.length() - 1))) {
                res = w.length() > res.length() ? w : res;
                built.add(w);
            }
        }
        return res;
    }
}
```
Explanation :
Sort the Words: The array words is sorted lexicographically (alphabetically). This ensures that shorter words and potential prefixes appear before longer words, which helps in easily identifying valid longer words.
Set to Track Built Words: A HashSet<String> called built is used to store words that have been successfully "built," meaning they can be constructed by appending a character to an existing valid word.
Loop Through Words: The code iterates through the sorted array:

- If the word is either a single letter or its prefix (i.e., w.substring(0, w.length() - 1)) exists in the built set, it's considered valid.
- It compares the length of the current valid word with the current longest word (res) and updates res if the current word is longer.
- The word is then added to the built set.

Return Longest Word: Finally, it returns the longest word that satisfies the condition.

Input: words = ["a","banana","app","appl","ap","apply","apple"]
Output: "**apple**"
Explanation: Both "apply" and "apple" can be built from other words in the dictionary. However, "apple" is lexicographically smaller than "apply".

Ques 4 : https://leetcode.com/problems/increasing-order-search-tree/
Code :
```
/**
 * Definition for a binary tree node.
 * public class TreeNode {
 *     int val;
 *     TreeNode left;
 *     TreeNode right;
 *     TreeNode() {}
 *     TreeNode(int val) { this.val = val; }
 *     TreeNode(int val, TreeNode left, TreeNode right) {
 *         this.val = val;
 *         this.left = left;
 *         this.right = right;
 *     }
 * }
 */
class Solution {

    TreeNode ans = new TreeNode();
    TreeNode ansRoot = ans;

    public TreeNode increasingBST(TreeNode root) {

        inOrder(root);
        return ansRoot.right;
    }

    void inOrder(TreeNode node) {

        if(node == null) {
            return;
        }
        inOrder(node.left);
        ans.right = new TreeNode(node.val);
        ans = ans.right;
        inOrder(node.right);
    }
}
```

Explanation :

This Java code defines a solution to rearrange a binary search tree (BST) into an **increasing order search tree**. The goal is to create a new tree where each node only has a right child, and the nodes appear in increasing order.

The TreeNode class defines a standard structure for a binary tree node with a value val and references to left and right children.

The class has two member variables:

1. ans: A pointer to build the new tree.
2. ansRoot: Holds the reference to the root of the new tree. This is initialized as an empty node, and the actual tree starts at ansRoot.right.

increasingBST Method is the entry point.

1. It calls the helper method inOrder to traverse the original tree in an in-order fashion (left-root-right).
2. After the traversal, it returns ansRoot.right, which is the root of the newly formed tree.

inOrder Traversal:

1. This is a recursive method that traverses the tree in in-order (visiting the left subtree, then the root, then the right subtree).
2. For each node visited, it creates a new node with the current value (node.val) and assigns it as the right child of the current node (ans). The ans pointer is then updated to this newly created node.
3. It ensures that the nodes are added in increasing order because of the in-order traversal.

The code effectively transforms the given BST into a new tree that consists only of right children, with nodes arranged in increasing order. This is achieved by recursively visiting nodes in order and appending them to the right of the newly constructed tree.

Input: root = [5,3,6,2,4,null,8,1,null,null,null,7,9]
Output: [1,null,2,null,3,null,4,null,5,null,6,null,7,null,8,null,9]
Ques 5 : https://leetcode.com/problems/univalued-binary-tree/
Code :

```
class Solution {
   Set<Integer> s = new HashSet<>();
   public void traversal(TreeNode root) {
      if (root != null) {
         traversal(root.left);
         s.add(root.val);
         traversal(root.right);
      }
   }
   public boolean isUnivalTree(TreeNode root) {
      if (root == null) return true;
      traversal(root);

      return ((s.size() == 1)? true : false);
```

```
    }
}
```
Explanation :

This Java code defines a solution to check if a binary tree is a univalued tree, meaning all the nodes in the tree have the same value.

1. Traversal Method: The traversal method is a recursive in-order traversal that visits every node of the tree. It adds each node's value to a HashSet called s. Since a HashSet only stores unique values, it ensures that all values in the tree are tracked without duplicates.
2. Check Univalued Condition: After the traversal, the isUnivalTree method checks the size of the HashSet. If the size is 1, it means the tree has only one unique value, making it a univalued tree. Otherwise, it returns false.

Input: root = [1,1,1,1,1,null,1]

Output: true

Ques 6 : https://leetcode.com/problems/day-of-the-year/

Code :

```
class Solution {
    public int dayOfYear(String date) {
        int days[] = {31,28,31,30,31,30,31,31,30,31,30,31};
                int year = Integer.parseInt(date.substring(0, 4));
                int month = Integer.parseInt(date.substring(5, 7));
                int day = Integer.parseInt(date.substring(8));


                if(month > 2 && year % 4 == 0 && (year % 100 != 0 || year % 400 == 0))
                {
                        day=day+1;
                }
                while(--month>0)
                {
                        day = day+ days[month-1];
                }
                return day;
    }
}
```
Explanation :

Parse the Date: The input date is a string in the format "YYYY-MM-DD". The code extracts the year, month, and day using substring and converts them to integers.
Days in Each Month: The array days[] holds the number of days in each month, assuming it's a non-leap year.
Leap Year Check: If the year is a leap year and the month is after February, one extra day is added to the day (since leap years have 29 days in February). A year is considered a leap year if:

1. It is divisible by 4, and
2. It is not divisible by 100 unless it is also divisible by 400.

Add Days from Previous Months: The loop sums the days from the previous months by adding the number of days in each month (from the days[] array) to the day variable.
Return the Result: The method returns the total number of days from the start of the year to the given date.
Input: date = "2019-01-09"
Output: 9
Explanation: Given date is the 9th day of the year in 2019.

Ques 7 : https://leetcode.com/problems/day-of-the-week/
Code :

```
class Solution {
    public String dayOfTheWeek(int day, int month, int year) {
        String[] week = { "Thursday", "Friday", "Saturday", "Sunday", "Monday", "Tuesday",
"Wednesday" };
        HashMap<Integer, Integer> map = new HashMap<>();
        for (int i = 1; i <= 12; i++) {
            if (i == 2)
                map.put(i, 28);
            else if (i == 4 || i == 6 || i == 9 || i == 11)
                map.put(i, 30);
            else
                map.put(i, 31);
        }

        int count = 0;
        for (int i = 1971; i < year; i++) {
            if (i % 4 == 0 && ((i % 100 != 0) || (i % 400 == 0)))
                count += 366;
            else
            count += 365;
        }

        if (year % 4 == 0 && ((year % 100 != 0) || (year % 400 == 0)))
            map.put(2, 29);

        int total = 0;
        for (int i = 1; i < month; i++)
            total += map.get(i);

        int totalTillTheDate = count + total + day;

        int rem = (totalTillTheDate) % 7;

        return week[rem];
    }
}
```

Explanation :

Weekday Mapping: The week array contains the days of the week, with the index corresponding to days starting from **Thursday**. This is based on the fact that January 1st, 1971, was a Friday.

Days in Each Month: A HashMap<Integer, Integer> is used to map each month to its number of days. February is initially set to 28 days, and the months with 30 days (April, June, September, November) are adjusted accordingly.

Count Days From 1971: The code iterates from the year **1971** up to the given year, summing up the total number of days. It adds **366 days** for leap years and **365 days** for non-leap years. Leap years are determined using the standard rules:

1. A year is a leap year if divisible by 4, except for years divisible by 100 unless also divisible by 400.

Adjust for Leap Year in the Current Year: If the current year is a leap year, the code updates February's days to 29.

Sum Days in Current Year: The loop sums the days from January to the given month and adds the day to get the total days from January 1st, 1971, to the given date.

Calculate Day of the Week: The total number of days is divided by 7, and the remainder (rem) gives the index in the week array. This index corresponds to the day of the week. The method returns the corresponding day of the week for the given date.

Input: day = 15, month = 8, year = 1993

Output: "**Sunday**"

Ques 8 :
https://leetcode.com/problems/check-if-a-word-occurs-as-a-prefix-of-any-word-in-a-sentence/
Code :
```
class Solution {
    public int isPrefixOfWord(String sentence, String searchWord) {
        String s = sentence.strip();
        String[] s_arr = s.split(" ");
        for (int i = 0; i < s_arr.length; i++) {
            if (s_arr[i].startsWith(searchWord)) {
                return i + 1; // Return the 1-based index
            }
        }

        return -1; // Return -1 if no word starts with searchWord
    }
}
```
Explanation :
String s is used to remove leading and trailing whitespace and String s_arr is used to split the sentence into words. Iterate through each word by using a for loop. Check if the word starts with searchWord then return the 1-based index otherwise return -1.
Input: sentence = "i love eating burger", searchWord = "burg"
Output: 4
Explanation: "burg" is the prefix of "burger" which is the 4th word in the sentence.

Ques 9 :
https://leetcode.com/problems/minimum-insertions-to-balance-a-parentheses-string/
Code :
```
class Solution {
    public int minInsertions(String s) {
        int count=0;
        int open=0;
        for(int i=0;i<s.length();i++){
            char ch=s.charAt(i);
            if(ch==')'){
                if(i+1<s.length() && s.charAt(i+1)==')'){
```

```
                if(open!=0)open--;
                else count++;
                i++;
            }else{
                if(open!=0){
                    count++;
                    open--;
                }else count+=2;
            }
        }else open++;

    }
    return open*2+count;
  }
}
```

Explanation :

A parentheses string is balanced if:

- Any left parenthesis '(' must have a corresponding two consecutive right parenthesis '))'.
- Left parenthesis '(' must go before the corresponding two consecutive right parenthesis '))'.

count: Tracks the number of insertions required.

open: Tracks the number of unpaired opening parentheses (.

Iterate Over the String: The code loops through the string, checking each character:

If the character is a closing parenthesis), it checks if it's part of a double closing)).

If yes, it reduces open (if there are unpaired () or increments count if no unpaired (is available.

If it's a single), it increments count by 1 or 2 to ensure a valid pair.

If the character is an opening parenthesis (, it increments open.

Final Adjustment: After the loop, any remaining unpaired ((tracked by open) needs two closing parentheses for each. Thus, open * 2 is added to count.

Input: s = "(()))"

Output: 1

.Ques 10 :

https://leetcode.com/problems/convert-1d-array-into-2d-array/description/

Code :

```
class Solution {
    public int[][] construct2DArray(int[] original, int m, int n) {
        if (m * n != original.length) {
            return new int[0][0];
        }
        int resultArray[][] = new int[m][n];
            int index = 0;
        for (int i = 0; i < m; i++) {
            for (int j = 0; j < n; j++) {
                        resultArray[i][j] = original[index];
                index++;
            }
        }
        return resultArray;
    }
}
```

Explanation :
Check if it is possible to form an m x n 2D array. If not, return an empty 2D array. Initialize the result 2D array with m rows and n columns. Initialize a counter to track the current index in the original array. Assign the current element from the original array to the 2D array. Move to the next element in the original array.
Input: original = [1,2,3,4], m = 2, n = 2
Output: [[1,2],[3,4]]
Explanation: The constructed 2D array should contain 2 rows and 2 columns.
The first group of n=2 elements in the original, [1,2], becomes the first row in the constructed 2D array.
The second group of n=2 elements in the original, [3,4], becomes the second row in the constructed 2D array.

Ques 11 : https://leetcode.com/problems/vowels-of-all-substrings/description/
Code :

```
class Solution {
    public long countVowels(String word) {
        long count=0;
        int len=word.length();
        for(int i=len-1;i>=0;i--){
            if(vowelschck(word.charAt(i))){
                count+=(long)(i+1)*(len-i);
            }
        }
        return count;
    }
     boolean vowelschck(char c){
        if(c=='a' || c=='e' || c=='i' || c=='o' || c=='u'){
            return true;
        }
        return false;
    }
}
```
Explanation :

In this question we have to find a count of all the vowel present in all possible substring of the given string, the easiest way to solve this question by : first of all write the function to check the vowel is present in string or not if no vowel present in string return count 0 otherwise count the no of vowels present in string. After calculating the count, count should be multiplied to the length of the string.

All possible substrings are: "a", "ab", "aba", "b", "ba", and "a".

- "b" has 0 vowels in it
- "a", "ab", "ba", and "a" have 1 vowel each
- "aba" has 2 vowels in it

Hence, the total sum of vowels = 0 + 1 + 1 + 1 + 1 + 2 = 6.

Input: word = "aba"

Output: 6

Ques 12 : https://leetcode.com/problems/number-of-common-factors/

Code :

```
class Solution {
    public int commonFactors(int a, int b) {
        int cnt = 0;
        int n = Math.max(a, b);
        for (int i = 1; i <= n; i++) {
            if (a % i == 0 && b % i == 0)
                cnt++;
        }
        return cnt;
    }
}
```
Explanation :

The common factors of 12 and 6 are 1, 2, 3, 6.
Input: a = 12, b = 6
Output: 4

Ques 13 : https://leetcode.com/problems/find-closest-number-to-zero/
Code :
```
class Solution
{
    public int findClosestNumber(int[] nums)
    {
        int res = Integer.MAX_VALUE;
        for(int i: nums)
            if(Math.abs(i) < Math.abs(res) || i == Math.abs(res))
                res = i;
        return res;
    }
}
```
Explanation :
First of all calculate the differences between given distances of array and 0.
The distance from -4 to 0 is |-4| = 4.
The distance from -2 to 0 is |-2| = 2.
The distance from 1 to 0 is |1| = 1.
The distance from 4 to 0 is |4| = 4.
The distance from 8 to 0 is |8| = 8.
Thus, the closest number to 0 in the array is 1.
Input: nums = [-4,-2,1,4,8]
Output: 1