Report has been categorized into 4 sections-

1. **Introduction to LeetCode**
2. **Personal Journey and Experience**
3. **Challenges and Solutions**
4. **Impact and Future Plans**

1. Introduction to LeetCode

LeetCode Overview LeetCode is an online platform designed to help software engineers enhance their coding skills. It offers a vast collection of coding problems ranging from easy to hard, covering various topics like algorithms, data structures, databases, shell scripting, and more. Besides individual practice, LeetCode also provides features such as weekly contests, discussion forums, and company-specific problem sets, which help users prepare for coding interviews and assessments.

2. Personal Journey and Experience

Getting Started My journey with LeetCode began with a recommendation from a friend who had successfully landed a software engineering job. I signed up and was initially overwhelmed by the number of problems available. To tackle this, I started with the "Explore" section, which guides beginners through fundamental topics.

Initial Challenges At first, even the easy problems seemed daunting. Understanding the problem statements and coming up with efficient solutions was a significant challenge. The lack of immediate feedback on wrong solutions often led to frustration.

Strategies for Improvement To overcome these challenges, I adopted a systematic approach:

- **Consistency:** I committed to solving at least one problem a day.
- **Learning from Others:** I regularly read through discussion forums and solutions shared by other users to understand different approaches.
- **Focusing on Fundamentals:** I spent extra time on understanding basic algorithms and data structures, which are crucial for solving complex problems.

Progress Over Time With consistent practice, I started noticing improvement in my problem-solving skills. Problems that once seemed impossible began to feel more approachable. I also participated in LeetCode's weekly contests to test my skills under time constraints, which mirrored real-world interview scenarios.

3. Challenges and Solutions

Common Challenges

- **Time Management:** Balancing LeetCode practice with other commitments was tough.
- **Understanding Problem Statements:** Some problems had intricate statements that required careful reading and comprehension.
- **Efficiency:** Writing code that not only works but is also optimized for performance.

Solutions Implemented

- **Schedule:** I set a fixed time daily for LeetCode practice, usually in the morning when my mind was fresh.
- **Active Reading:** I started reading problem statements multiple times and breaking them down into smaller parts to ensure complete understanding.
- **Code Review:** After solving a problem, I reviewed my code and compared it with top solutions to identify areas for improvement.

4. Impact and Future Plans

Skill Development LeetCode has significantly improved my coding skills. I am now more proficient in algorithms and data structures, which are fundamental for any software engineering role. My ability to think logically and approach problems methodically has also enhanced.

Interview Preparation The platform has been instrumental in my interview preparations. I feel more confident tackling coding assessments and technical interviews. The company-specific problem sets are particularly useful for preparing for interviews with specific organizations.

Future Plans

- **Advanced Topics:** I plan to delve deeper into advanced topics like dynamic programming, graph algorithms, and system design.
- **Regular Contests:** Continuing to participate in weekly contests to keep my skills sharp and stay updated with new types of problems.
- **Contributing to Community:** Sharing my solutions and insights on the discussion forums to help other users.

Conclusion

My experience with LeetCode has been transformative. It has not only improved my coding abilities but also boosted my confidence in solving complex problems. The journey from struggling with easy problems to comfortably solving medium and hard problems has been rewarding. I look forward to continuing this journey and leveraging the skills gained through LeetCode in my professional career.

3 Problems->

1. Two Sum

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.

Example 1:

Input: nums = [2,7,11,15], target = 9

Output: [0,1]

Explanation: Because nums[0] + nums[1] == 9, we return [0, 1].

Example 2:

Input: nums = [3,2,4], target = 6

Output: [1,2]

Example 3:

Input: nums = [3,3], target = 6

Output: [0,1]

Solution-

Intuition

to improve the time complexity from 0(n^2) brute force attack to O(n) using single pass hash table

Approach

1)create a hastable using unordered map

2)create a vector to store the result

3)iterate through the given nums

4)calculate 'second number' such that current number - second number = target

->second_num=target-nums[i]

5)search for this second number in hash map which will take O(1) time

->if found,insert the indices of current number and second number into result vector and break the loop

6)if not found,insert the current element in hashmap for next iteration

7)return the result vector

Complexity

Time complexity:

O(n)

Space complexity:

O(n) for hash table

Code

```
class Solution {
public:
    vector<int> twoSum(vector<int>& nums, int target) {
    unordered_map<int, int> m; //hashmap
    vector<int> result;
```

```
    for(int i=0;i<nums.size();i++){
        int second_num=target-nums[i];
        if(m.find(second_num)!=m.end()){
            result.push_back(i); //first index
            result.push_back(m[second_num]);  //second index
            break;
         }
         m[nums[i]]=i; //insert key value pair in hashmap
       }
       return result;
    }
};
```

2. Longest Substring Without Repeating Characters

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.

Solution-

Intuition

The intuition behind the 3 solutions is to iteratively find the longest substring without repeating characters by maintaining a sliding window approach. We use two pointers (left and right) to represent the boundaries of the current substring. As we iterate through the string, we update the pointers and adjust the window to accommodate new unique characters and eliminate repeating characters.

Approach - Unordered Map

1. We improve upon the first solution by using an unordered map (charMap) instead of a set.
2. The map stores characters as keys and their indices as values.
3. We still maintain the left and right pointers and the maxLength variable.

4. We iterate through the string using the right pointer.
5. If the current character is not in the map or its index is less than left, it means it is a new unique character.
6. We update the charMap with the character's index and update the maxLength if necessary.
7. If the character is repeating within the current substring, we move the left pointer to the next position after the last occurrence of the character.
8. We update the index of the current character in the charMap and continue the iteration.
9. At the end, we return the maxLength as the length of the longest substring without repeating characters.

Code-

```
class Solution {
public:
   int lengthOfLongestSubstring(string s) {
      int n = s.length();
      int maxLength = 0;
      unordered_map<char, int> charMap;
      int left = 0;

      for (int right = 0; right < n; right++) {
         if (charMap.count(s[right]) == 0 || charMap[s[right]] < left)
       {
            charMap[s[right]] = right;
            maxLength = max(maxLength, right - left + 1);
       }
       else {
```

```
                left = charMap[s[right]] + 1;
                charMap[s[right]] = right;
            }
        }

        return maxLength;
    }
};
```

3. Word Break II

Given a string s and a dictionary of strings wordDict, add spaces in s to construct a sentence where each word is a valid dictionary word. Return all such possible sentences in any order.

Note that the same word in the dictionary may be reused multiple times in the segmentation.

Example 1:

Input: s = "catsanddog", wordDict = ["cat","cats","and","sand","dog"]

Output: ["cats and dog","cat sand dog"]

Example 2:

Input: s = "pineapplepenapple", wordDict = ["apple","pen","applepen","pine","pineapple"]

Output: ["pine apple pen apple","pineapple pen apple","pine applepen apple"]

Explanation: Note that you are allowed to reuse a dictionary word.

Example 3:

Input: s = "catsandog", wordDict = ["cats","dog","sand","and","cat"]

Output: []

Solution-

Intuition

- We can use DFS to solve this problem. We can iterate over our input string and as we find a word that is in our dictionary we can create new branch which starts from last index and do same process.
- If we reach our intital string end it means that string is valid and we can push it to our result set

Approach

- We are using function rec to do dfs here it will have
    - string s - which is the original input string
    - int beg - which represents starting pointer in s string on this branch of tree
    - string curr - which represents space separated string which is result string till index beg if beg reaches s.size() we can push it to our result set
    - vector result - It represents all possible strings which we can make
- We start iterating over string s from beg index and accumulate words in a temporary string temp.

- If dict has word temp we can create a new branch with starting index next to current index and change our curr string to include space separated temp string as well
- If our beg index has reached end of s we can say that our curr word is a valid word and we can push it in our result set

Complexity

- Time complexity: O(2^n) we have possibility of creating two branches at each step
- Space complexity: O(2^n) stack space is used for calling out at each branch

Code-

```
class Solution {
public:
    set<string> dict;
    vector<string> wordBreak(string s, vector<string>& wordDict) {
        for(auto x : wordDict)dict.insert(x);
        vector<string> res;
        rec(s, 0, "", res);
        return res;
    }

    void rec(string s, int beg, string curr, vector<string>& res) {
        if(beg==s.size()) {
```

```
            if(curr[0]==' ')curr = curr.substr(1);
            res.push_back(curr);
        }
        string temp = "";
        for(int i=beg;i<s.size();i++) {
            temp.push_back(s[i]);
            if(dict.count(temp)) {
                rec(s, i+1, curr + " " + temp, res);
            }
        }
    }
};
```