Day 8 / 100 :

Topic - Strings, Array

1 Problem statement: Single element in a Sorted Array (medium)

You are given a sorted array consisting of only integers where every element appears exactly twice, except for one element which appears exactly once.

Return the single element that appears only once.

Your solution must run in O(log n) time and O(1) space.

Example 1:

Input: nums = [1,1,2,3,3,4,4,8,8]
Output: 2
Example 2:

Input: nums = [3,3,7,7,10,11,11]
Output: 10

Solutions :

Approach 1- 2 pointer approach

```
class Solution {
public:
    int singleNonDuplicate(vector<int>& nums) {
        int left = 0, right = nums.size() - 1;
        while(left < right){
            int mid = (left + right)/2;
            if((mid % 2 == 0 && nums[mid] == nums[mid + 1]) || (mid %
2 == 1 && nums[mid] == nums[mid - 1]))
                left = mid + 1;
            else
                right = mid;
        }
```

```
        return nums[left];
    }
};
```

EXPLANATION:-
Suppose array is [1, 1, 2, 2, 3, 3, 4, 5, 5]
we can observe that for each pair,
first element takes even position and second element takes odd position
for example, 1 is appeared as a pair,
so it takes 0 and 1 positions. similarly for all the pairs also.

this pattern will be missed when single element is appeared in the array.

From these points, we can implement algorithm.
1. Take left and right pointers .
   left points to start of list. right points to end of the list.
2. find mid.
   if mid is even, then it's duplicate should be in next index.
      or if mid is odd, then it's duplicate should be in previous index.
      check these two conditions,
      if any of the conditions is satisfied,
      then pattern is not missed,
      so check in next half of the array. i.e, left = mid + 1
      if condition is not satisfied, then the pattern is missed.
      so, single number must be before mid.
      so, update end to mid.
3. At last return the nums[left]

Time: - O(logN)
space:- O(1)

2️ Problem statement: [Longest Palindrome substring](#) (medium)

Given a string s, return the longest palindromic substring in s.

Example 1:

Input: s = "babad"
Output: "bab"
Explanation: "aba" is also a valid answer.

Example 2:

Input: s = "cbbd"
Output: "bb"

Solutions :

Approach 1- Two Pointer

```
string longestPalindrome(string s) {
    int n = s.length();
    int start = 0, end = 0;

    for (int i = 0; i < n; i++) {
        for (int j = i; j < n; j++) {
            int left = i, right = j;
            bool isPalindrome = true;

            while (left < right) {
                if (s[left] != s[right]) {
                    isPalindrome = false;
                    break;
                }
                left++;
                right--;
            }

            if (isPalindrome && (j - i + 1) > (end - start + 1)) {
                start = i;
                end = j;
```

```
            }
        }
    }

    return s.substr(start, end - start + 1);
}
```

The brute force approach involves checking every possible substring of the given string and verifying if it is a palindrome. We can start by checking the longest substrings and gradually reduce the length until we find a palindrome. Here are the steps:

1. Initialize two variables, "start" and "end," to keep track of the longest palindromic substring found so far.
2. Iterate through every substring in the given string:
3. Nested loops to define the starting and ending indices of the substring.
4. Check if the current substring is a palindrome by comparing characters from both ends.
5. If it is a palindrome and longer than the previous longest substring, update the "start" and "end" variables.
6. Return the substring from "start" to "end" from the original string.

Solutions :

Approach 2 - Dynamic Programming

```
string longestPalindrome(string s) {
    int n = s.length();
    vector<vector<bool>> dp(n, vector<bool>(n, false));
    int start = 0, maxLen = 1;

    // Single character substrings are palindromes
    for (int i = 0; i < n; i++)
        dp[i][i] = true;

    // Check substrings of lengths 2 to n
    for (int len = 2; len <= n; len++) {
        for (int i = 0; i < n - len + 1; i++) {
            int j = i + len - 1;
```

```
            // Check if the current substring is a palindrome
            if (s[i] == s[j] && (len == 2 || dp[i + 1][j - 1])) {
                dp[i][j] = true;

                // Update the longest palindromic substring
                if (len > maxLen) {
                    start = i;
                    maxLen = len;
                }
            }
        }
    }

    return s.substr(start, maxLen);
}
```

The efficient approach to finding the longest palindromic substring involves dynamic programming. We can build a table to store whether a substring is a palindrome or not. By utilizing the properties of palindromes, we can optimize the process. Here are the steps:

1. Initialize a 2D boolean table "dp" of size [n][n], where n is the length of the string.
2. Initialize variables "start" and "maxLen" to keep track of the starting index and length of the longest palindromic substring found so far.
3. Set all entries of the table as false initially.
4. For each character in the string, set dp[i][i] as true since a single character is always a palindrome.
5. Iterate over the string, considering substrings of lengths from 2 to n:
    - Nested loops to define the starting and ending indices of the substring.
    - Check if the current substring is a palindrome using the table and the characters at the indices.
    - If it is a palindrome, update the corresponding table entry and check if it is longer than the previous longest palindrome.
6. Return the substring from "start" to "start + maxLen - 1" from the original string.