Day 47 / 100 :

Topic - Array, Binary search

1 Problem statement: Search in rotated sorted array II (medium)

There is an integer array nums sorted in non-decreasing order (not necessarily with distinct values).

Before being passed to your function, nums is rotated at an unknown pivot index k (0 <= k < nums.length) such that the resulting array is [nums[k], nums[k+1], ..., nums[n-1], nums[0], nums[1], ..., nums[k-1]] (0-indexed). For example, [0,1,2,4,4,4,5,6,6,7] might be rotated at pivot index 5 and become [4,5,6,6,7,0,1,2,4,4].
Given the array nums after the rotation and an integer target, return true if target is in nums, or false if it is not in nums.
You must decrease the overall operation steps as much as possible.

Example 1:
Input: nums = [2,5,6,0,0,1,2], target = 0
Output: true

Example 2:
Input: nums = [2,5,6,0,0,1,2], target = 3
Output: false

Solutions :

Approach 1 - Hash table

Intuition:

The idea is to find the point of pivot, which separates the array into 2 different monotonic arrays
And apply binary search on one of them

How to find pivot point

- Since the initial array was monotonically non decreasing every element was greater than the previous one
- After rotation it divides the array into 2 halves , 1 where the array is increasing and the other where array is decreasing
- The pt of pivot would be the one where from where the array starts to decrease
- eg : Before : 0,0,1,2,2,5,6 , After 2,5,6,0,0,1,2

- clearly idx == 3 is the pt of pivot as the array starts to increase from there
- Thus the idea is to find the first number which is smaller than nums[0], it is the pt. of pivot

Modifications for duplicate

- We would use binary search to find the pivot point
- But since duplicates exist the nums[0] can exist multiple times in the front or in the back
- The idea is to keep two pointers l = 0 , r = n-1
- And increase l untill it is the last occurance of nums[0]
- decrease r untill it is no longer equals nums[0]
- eg : 1,1,1,1,0,1,1,1,1,1 reduces to 1,0
- And now we can check which part of the array we want to apply binay search and by comparing with nums[r]
- And check in either l : pivot-1 or pivot : r
- In the worst case it takes O(N) but ideally it would take num of occurance (nums[0]) + O(logn) , ideally O(logn) assuming the nums[0] occurs less times

```
class Solution {
public:
    bool search(vector<int>& nums, int target) {

        if( nums[0] == target or nums.back() == target ) return true;
        // this line is redundant it reduces only the worst case when all
elements are same to O(1)

        const int n = nums.size();
        int l = 0 , h = n-1;
        while( l+1 < n and nums[l] == nums[l+1]) l++;

        // if all elements are same
        if( l == n-1){
            if( nums[0] == target ) return true;
            else return false;
        }

        // while last element is equal to 1st element
        while( h >= 0 and nums[h] == nums[0] ) h--;
        int start = l , end = h;

        // find the point of pivot ie from where the rotation starts
```

```
        int pivot = -1;
        while( l <= h ){
            int mid = l + (h-l)/2;
            if( nums[mid] >= nums[0] ) l = mid+1;
            else {
                pivot = mid;
                h = mid-1;
            }
        }


        if( pivot == -1 ) l = start , h = end; // if no pivot exits then
search space is from start -e end
        else {
            if( target > nums[end] ) l = start , h = pivot-1; // search
space second half
            else l = pivot , h = end; // search space first half
        }

        // normal binary search
        while ( l <= h ){
            int mid = l + (h-l)/2;
            if( nums[mid] > target ) h = mid-1;
            else if( nums[mid] < target ) l = mid+1;
            else return true;
        }

        return false;

    }
};
```

2 Problem statement: Longest Common subsequence (medium)

Given two strings, find the length of longest subsequence present in both of them. Both the strings are in uppercase latin alphabets.

Example 1:
Input:
A = 6, B = 6
str1 = ABCDGH
str2 = AEDFHR
Output: 3
Explanation: LCS for input strings “ABCDGH” and “AEDFHR” is “ADH” of length 3.

Example 2:
Input:
A = 3, B = 2
str1 = ABC
str2 = AC
Output: 2
Explanation: LCS of "ABC" and "AC" is "AC" of length 2.

Solutions :

Approach 1 - LCS

intuition:

Generate all the possible subsequences and find the longest among them that is present in both strings using recursion.

Approach:

- Create a recursive function [say lcs()].
- Check the relation between the First characters of the strings that are not yet processed.
- Depending on the relation call the next recursive function as mentioned above.
- Return the length of the LCS received as the answer.

```
// A Naive recursive implementation of LCS problem

#include <bits/stdc++.h>
using namespace std;
```

```cpp
// Returns length of LCS for X[0..m-1], Y[0..n-1]
int lcs(string X, string Y, int m, int n)
{
	if (m == 0 || n == 0)
		return 0;
	if (X[m - 1] == Y[n - 1])
		return 1 + lcs(X, Y, m - 1, n - 1);
	else
		return max(lcs(X, Y, m, n - 1),
				lcs(X, Y, m - 1, n));
}

// Driver code
int main()
{
	string S1 = "AGGTAB";
	string S2 = "GXTXAYB";
	int m = S1.size();
	int n = S2.size();

	cout << "Length of LCS is " << lcs(S1, S2, m, n);

	return 0;
}
```

Solutions :

Approach 2 - DP

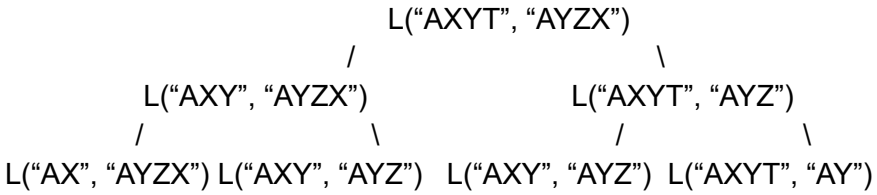
If we notice carefully, we can observe that the above recursive solution holds the following two properties:

Optimal Substructure:

See for solving the structure of L(X[0, 1, . . ., m-1], Y[0, 1, . . . , n-1]) we are taking the help of the substructures of X[0, 1, …, m-2], Y[0, 1,…, n-2], depending on the situation (i.e., using them optimally) to find the solution of the whole.

Overlapping Solution:

If we use the above recursive approach for strings “BD” and “ABCD“, we will get a partial recursion tree as shown below. Here we can see that the subproblem L(“BD”, “ABCD”) is being calculated more than once. If the total tree is considered there will be several such overlapping subproblems.



Approach:

- Because of the presence of these two properties we can use Dynamic programming or Memoization to solve the problem. Below is the approach for the solution using recursion.
- Create a recursive function. Also create a 2D array to store the result of a unique state.
- During the recursion call, if the same state is called more than once, then we can directly return the answer stored for that state instead of calculating again.

```cpp
#include <bits/stdc++.h>
using namespace std;

// Returns length of LCS for X[0..m-1],
// Y[0..n-1]
int lcs(char* X, char* Y, int m, int n,
        vector<vector<int> >& dp)
{
    if (m == 0 || n == 0)
        return 0;
    if (X[m - 1] == Y[n - 1])
        return dp[m][n] = 1 + lcs(X, Y, m - 1, n - 1, dp);

    if (dp[m][n] != -1) {
        return dp[m][n];
    }
    return dp[m][n] = max(lcs(X, Y, m, n - 1, dp),
                          lcs(X, Y, m - 1, n, dp));
}

// Driver code
int main()
{
    char X[] = "AGGTAB";
```

```
    char Y[] = "GXTXAYB";

    int m = strlen(X);
    int n = strlen(Y);
    vector<vector<int> > dp(m + 1, vector<int>(n + 1, -1));
    cout << "Length of LCS is " << lcs(X, Y, m, n, dp);

    return 0;
}
```