Technical Interview Problem Category Patterns

Binary Search

Search In Rotated Sorted Array

https://leetcode.com/problems/search-in-rotated-sorted-array/

Suppose an array sorted in ascending order is rotated at some pivot unknown to you beforehand.

(i.e., [0,1,2,4,5,6,7] might become [4,5,6,7,0,1,2]).

You are given a target value to search. If found in the array return its index, otherwise return -1.

You may assume no duplicate exists in the array.

Your algorithm's runtime complexity must be in the order of *O*(log *n*).

Example 1:

```
Input: nums = [4,5,6,7,0,1,2], target = 0

Output: 4
```

Example 2:

```
Input: nums = [4,5,6,7,0,1,2], target = 3

Output: -1
```

Find First And Last Position Of Element In Sorted Array

https://leetcode.com/problems/find-first-and-last-position-of-element-in-sorted-array/

Given an array of integers `nums` sorted in ascending order, find the starting and ending position of a given `target` value.

Your algorithm's runtime complexity must be in the order of *O*(log *n*).

If the target is not found in the array, return `[-1, -1]`.

Example 1:

```
Input: nums = [5,7,7,8,8,10], target = 8

Output: [3,4]
```

Example 2:

```
Input: nums = [5,7,7,8,8,10], target = 6

Output: [-1,-1]
```

```
React + Typescript + NextJS → Front End
```

```
Go, Python+Flask, Python+Djengo, Java+Spring, Node+Express → Backend
```

Search In A 2D Matrix

https://leetcode.com/problems/search-a-2d-matrix/

Write an efficient algorithm that searches for a value in an *m* x *n* matrix. This matrix has the following properties:

- Integers in each row are sorted from left to right.
- The first integer of each row is greater than the last integer of the previous row.

Example 1:

```
Input:

matrix = [

  [1,   3,  5,  7],

  [10, 11, 16, 20],

  [23, 30, 34, 50]

]

target = 3

Output: true
```

Example 2:

```
Input:

matrix = [

  [1,   3,  5,  7],

  [10, 11, 16, 20],

  [23, 30, 34, 50]

]

target = 13
```

```
Output: false
```

Breadth First Search

https://leetcode.com/problems/word-ladder/

```
Starting state + ending state
```

```
Breadth First Search involves some starting state and ending state. We can use both
depth first search and breadth first search most of the time
```

Given two words (*beginWord* and *endWord*), and a dictionary's word list, find the length of shortest transformation sequence from *beginWord* to *endWord*, such that:

1. Only one letter can be changed at a time.
2. Each transformed word must exist in the word list. Note that *beginWord* is *not* a transformed word.

Note:

- Return 0 if there is no such transformation sequence.
- All words have the same length.
- All words contain only lowercase alphabetic characters.
- You may assume no duplicates in the word list.
- You may assume *beginWord* and *endWord* are non-empty and are not the same.

Example 1:

```
Input:
```

```
beginWord = "hit",

endWord = "cog",

wordList = ["hot","dot","dog","lot","log","cog"]


Output: 5


Explanation: As one shortest transformation is "hit" -> "hot" -> "dot" -> "dog" ->
"cog",

return its length 5.
```

Example 2:

```
Input:

beginWord = "hit"

endWord = "cog"

wordList = ["hot","dot","dog","lot","log"]


Output: 0


Explanation: The endWord "cog" is not in wordList, therefore no possible
transformation.
```

```
}}
```

Dynamic Programming

Let's look at some patterns in dynamic programming. We might be able to pick up on some patterns just by looking at a bunch of different problem statements as well as the solutions to these problems.

Climbing Stairs Problem & Solution

https://leetcode.com/problems/climbing-stairs/

You are climbing a stair case. It takes *n* steps to reach to the top.

Each time you can either climb 1 or 2 steps. In how many distinct ways can you climb to the top?

Note: Given *n* will be a positive integer.

```
public class Solution {

public int climbStairs(int n) {

    if(n == 0 || n == 1 || n == 2){return n;}

    int[] mem = new int[n];

    mem[0] = 1;

    mem[1] = 2;

    for(int i = 2; i < n; i++){

        mem[i] = mem[i-1] + mem[i-2];

    }

    return mem[n-1];

}}
```

```java
public int rob(int[] nums) {
    if (nums.length == 0) return 0;
    int[] memo = new int[nums.length + 1];
    memo[0] = 0;
    memo[1] = nums[0];
    for (int i = 1; i < nums.length; i++) {
        int val = nums[i];
        memo[i+1] = Math.max(memo[i], memo[i-1] + val);
    }
    return memo[nums.length];
}
```

House Robber Problem & Solution

https://leetcode.com/problems/house-robber/

You are a professional robber planning to rob houses along a street. Each house has a certain amount of money stashed, the only constraint stopping you from robbing each of them is that adjacent houses have security system connected and it will automatically contact the police if two adjacent houses were broken into on the same night.

Given a list of non-negative integers representing the amount of money of each house, determine the maximum amount of money you can rob tonight without alerting the police.

```java
public int rob(int[] nums) {
    if (nums.length == 0) return 0;
    int[] memo = new int[nums.length + 1];
    memo[0] = 0;
    memo[1] = nums[0];
    for (int i = 1; i < nums.length; i++) {
        int val = nums[i];
        memo[i+1] = Math.max(memo[i], memo[i-1] + val);
    }
    return memo[nums.length];
}
```

So just looking at these two problems we can already see some similarities when it comes to problem statement. Both problems start by saying “You are doing something”. This isn’t really anything very specific to dynamic programming considering most problems set up some kind of situation for you although dynamic programming problems are usually situation based problems more so than other types of problems.

Next we'll see that we always have some kind of goal, much like other problems. In the first problem we are given are goal right away in the second sentence, while in house robber we're given our goal in the last sentence of the problem statement, but it's very important we always understand what goal we're trying to achieve.