```
Survey !!!!
Java
Python ()
C++

Mock Interview ?
1. Someone ()
2. Feedback (In general good and bad)
3. Recorded (LMS)

Ngrok
C:\Windows\System32
```

https://leetcode.com/problems/min-cost-climbing-stairs (https://leetcode.com/problems/min-cost-climbing-stairs)

Brute force

TC: O(2^n)
SC: O(n)

```
class Solution {
public:
    int minCostClimbingStairs(vector<int>& cost) {
        return min(minCostClimbingStairsUtil(cost,0), minCostClimbingStairsUtil(cost,1));
    }

    int minCostClimbingStairsUtil(vector<int>& cost, int idx) {
        if (idx >= cost.size()) {
            return 0;
        }

        int step1 = minCostClimbingStairsUtil(cost, idx + 1);
        int step2 = minCostClimbingStairsUtil(cost, idx + 2);

        return cost[idx] + min(step1, step2);
    }

};
```

DP: Memoization

TC: O(n)
SC: O(n)

```
class Solution {
public:
    int minCostClimbingStairs(vector<int>& cost) {
        std::unordered_map<int, int> cache;
        return min(minCostClimbingStairsUtil(cache, cost,0), minCostC
limbingStairsUtil(cache, cost,1));
    }

    int minCostClimbingStairsUtil(std::unordered_map<int, int>& cach
e, vector<int>& cost, int idx) {
        if (idx >= cost.size()) {
            return 0;
        }
        if (cache.count(idx)  != 0) {
            return cache[idx];
        }

        int step1 = minCostClimbingStairsUtil(cache, cost, idx + 1);
        int step2 = minCostClimbingStairsUtil(cache, cost, idx + 2);

        int ans = cost[idx] + min(step1, step2);
        cache[idx] = ans;
        return ans;
    }

};
```

DP: Tabulation

TC: O(n)
SC: O(n)

```
class Solution {
    public int minCostClimbingStairs(int[] cost) {
      int n = cost.length;
      int[] dp = new int[n + 1];

      //base case
      dp[0] = cost[0];
      dp[1] = cost[1];

      for(int i = 2; i < n; i++){
          dp[i] = cost[i] + Math.min(dp[i - 1], dp[i - 2]);
          //stores the cost of each step
```

In [ ]:

In [ ]:

Recursion Problems

- Backtracking
- DP

In [ ]:

Rethinking Climibing Stairs: Identify whether we use backtracking or DP

```
For
  N=3 [1 1 1] [1 2] [2 1]
  N=4 [1 1 1 1] [1 1 2] [1 2 1] [2 1 1] [2 2]
```

Do I need the actual permutations ?

```
Yes: Backtracking
No:
```

-> Do I need to count possible solutions ?

```
Yes:
    Backtracking: BruteForce
    DP: Optimal
```

In [ ]:

```
DP-problem
- Look for keywords: min/max, count, unique, ways etc
- Think of a recurrence relation (draw on paper / board)
- Figure out base conditions
- Figure out number of state variables (the variables on which answer depends: 1D, 2D, .....)
- Write a top-down / bottom-up code

DP problem top-down structure
- populate a (cache)/ (dp array/matrix or whatever)
- call utility function with cache

- Write a recursive utility function
    // base conditions
    // cache lookup
    // recursion
    // caching


DP problem bottom-up structure
- populate the cache with solution for the trivial cases
- run loops which start from non-trivial cases
      - use recurrence relation to determine solution at current structure
- return last answer
```

In [ ]:

In [ ]:

2D

https://leetcode.com/problems/unique-paths/description/ (https://leetcode.com/problems/unique-paths/description/)

In [ ]:

In [ ]:

In [ ]: