13-11-24
Wednesday

Dynamic Programming (Andrey Grehov)

What is Dynamic Programming (DP):

- It is a method to solve a certain set of problems
- Eg: Game theory, computer science, etc.,

$$\begin{tikzcd} \text{problem} \arrow[r] & \boxed{\text{DP}} \arrow[r] & \text{solution.} \end{tikzcd}$$

- The main use of DP is that it can be used to solve the problems in Polynomial Time, where the Naive approach would take Exponential Time.

Polynomial $\rightarrow$ $O(n^c)$ Eg: $O(n^2)$, $O(n^3)$, etc.,
Non-polynomial $\rightarrow$ $O(c^n)$ Eg: $O(2^n)$

"DP is an OPTIMIZATION TECHNIQUE" ✓

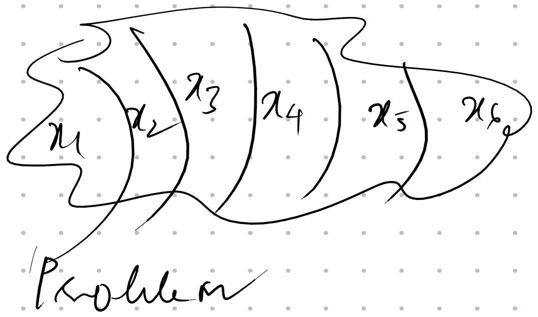
"DP is breaking the problem into smaller sub-problems"

In order to solve the problem using DP
It must have, two props.

i) → optimal substructure

best   subproblem.

We go solving the problem step-by-step.

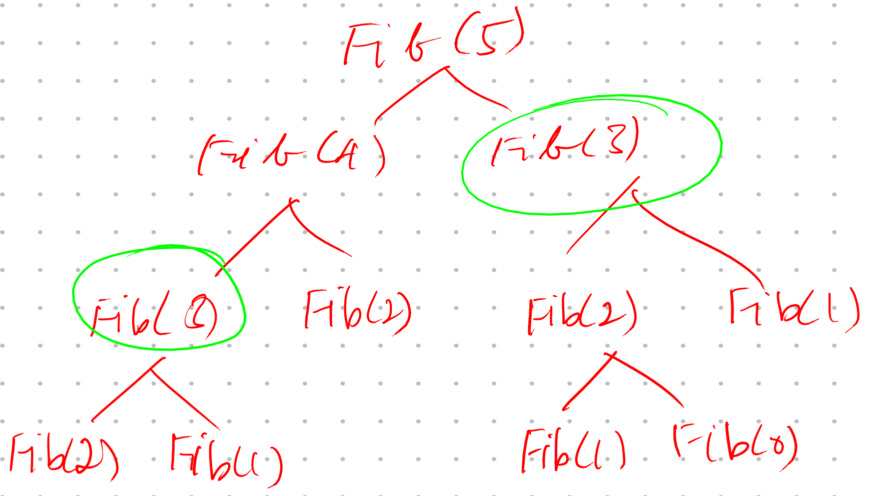


In this we reuse the result the previous solved subproblems.

ii) → overlapping subproblems.

Many times, when we breakdown the problems we may need to recalculate the results multiple times.

Eg: Fibonacii sequence Fib(5)



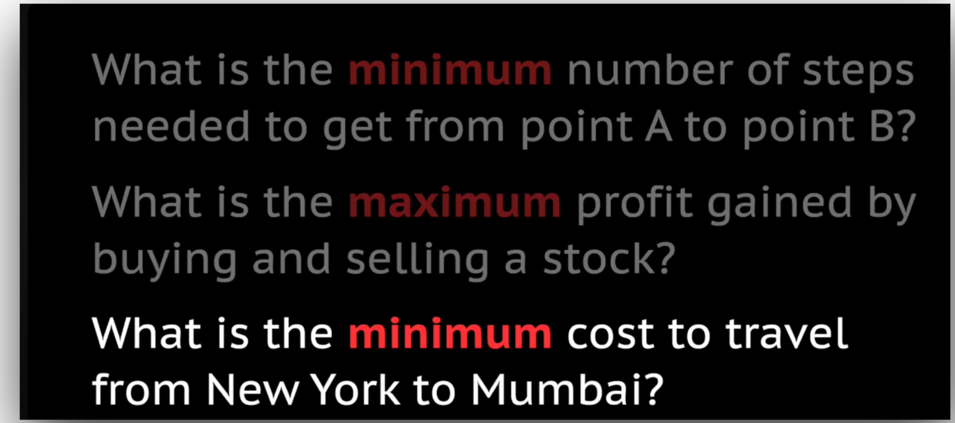
How to find out the DP problems:

2 types { - Combinatorics ← {how many} eg: i) how many ways to make a change
ii) how many ways to transverse a graph
iii) how many steps needed to get from pt A to pt B.

- Optimization ← We are Interested in finding a strategy which maximizes or minimizes a function

Eg: what is the minimum number of steps needed to get from pt A to pt B.

other Examples:


What is the **minimum** number of steps needed to get from point A to point B?

What is the **maximum** profit gained by buying and selling a stock?

What is the **minimum** cost to travel from New York to Mumbai?

- In optimization problem, our goal is to minimize or maximize some function.

DP DEFINITION:

Dp is a algo technique to solve the combinatorial and optimization problem utilizing the fact the optimal sol. to overall problem depends upon the optimal sol. to its overlapping subproblem.

Problem 1:

Sum of first 'N' numbers.

$1+2+3+\ldots\ldots+n = \sum_{i=1}^{n} i$

if $n=1$

1 (only)

$F(1) = 1$

if $n=2$

$1+2$

$F(2) = F(1)+2 = 1+2 = 3$

if $n=3$

$1+2+3.$

$F(3) = F(2)+3$

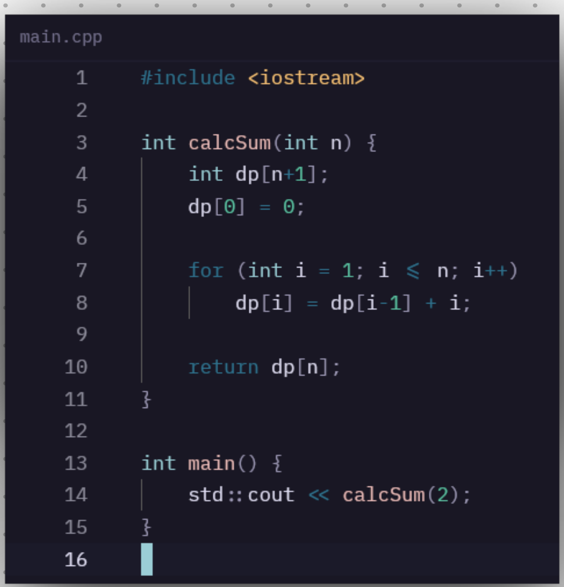
$= 3+3$

$= 6$

In general,

Recurrence Relation.

$F(n) = F(n-1)+n$



main.cpp

```cpp
#include <iostream>

int calcSum(int n) {
    int dp[n+1];
    dp[0] = 0;

    for (int i = 1; i <= n; i++)
        dp[i] = dp[i-1] + i;

    return dp[n];
}

int main() {
    std::cout << calcSum(2);
}

```

Just an example for understanding.

14-11-2024
Thursday

Memoization → is the process of caching the already computed results.

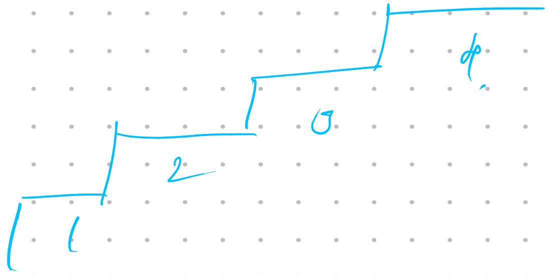
memoize / cache results.

In this, we reduce the time complexity by giving up on the $\boxed{\text{space}}$.

Problem: Climbing stairs.

ways

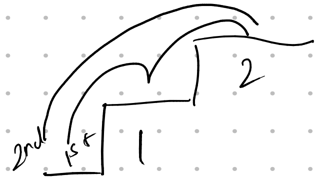
1) 1 - 2 - 3 - 4
2) 1 - 3 - 4
3) 2 - 4
4) 2 - 3 - 4
5) 1 - 2 - 4.



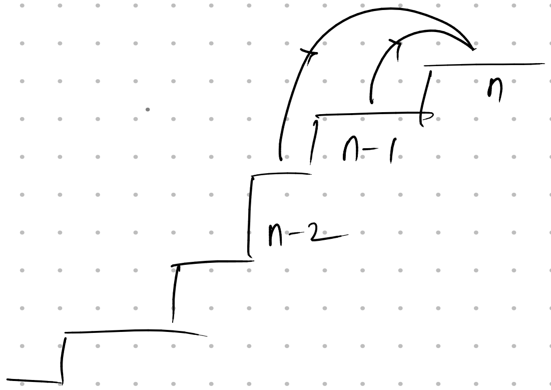
i) In this problem, F(i) is the no. of steps to the 'i' th stair.

Step 1: define the problem in terms of objective fn
(objective fn is the problem minimize the cost
or maximize the profit).

$F(2) = 2$
$F(1) = 1$
$F(0) = 0$



for 2 steps.



The only way to get to the Top is from the prev step or from the prevprevious step.

So, Recurrence Relation, $F(n) = F(n-1) + F(n-2)$
(The Rule of sum).

STEPS TO SOLVE "DP" PROBLEM. (FRAMEWORK)

1) Define the objective function.
2) Identifying the base cases.
3) Recurrence Relation. ← Also called as the Transition function.
4) Order of Computation. ← The order in which the subproblems are solved.
5) Location of the answer. (i.e) $F(n)$
   ↑ sometimes in the $F(0)$ depends on the implementation.

Eg: using the "FRAMEWORK" for the climbing stairs.

i) Defining the objective function.

→ $f(i)$ is the no. of distinct ways to reach the i^{th} stair.

ii) Identify the base case

→ $f(0) = 1$

→ $f(1) = 1$

iii) Recurrence Relation.

$$f(n) = f(n-1) + f(n-2)$$

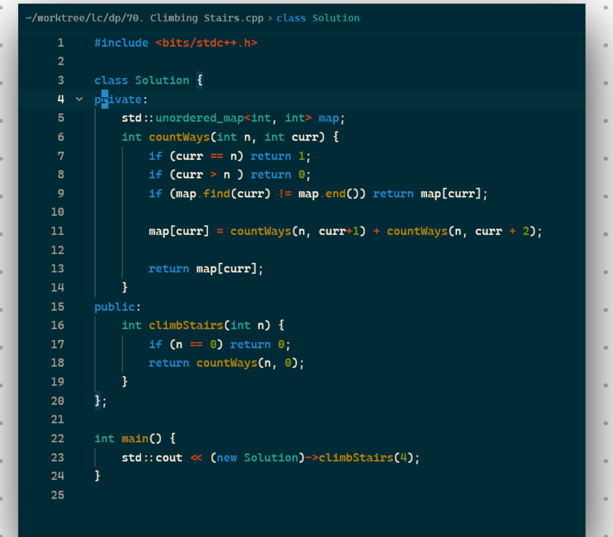
iv) Order of Execution

Bottom-up. (As we rely on the prev. two computation)

v) where to look for the answer?

$f(n)$

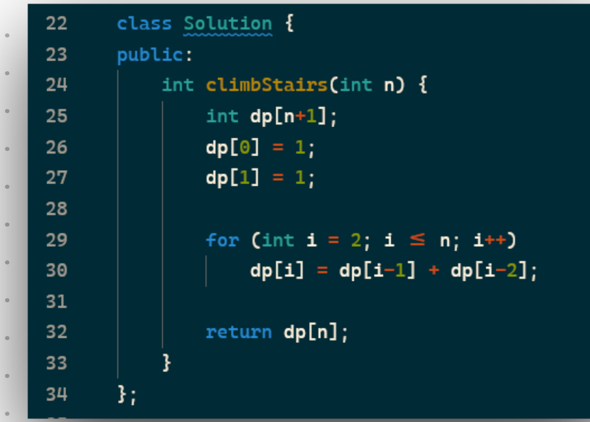
my-logic. (Top-Down) I guess!



~/worktree/lc/dp/70. Climbing Stairs.cpp › class Solution

```cpp
#include <bits/stdc++.h>

class Solution {
private:
    std::unordered_map<int, int> map;
    int countWays(int n, int curr) {
        if (curr == n) return 1;
        if (curr > n ) return 0;
        if (map.find(curr) != map.end()) return map[curr];

        map[curr] = countWays(n, curr+1) + countWays(n, curr + 2);

        return map[curr];
    }
public:
    int climbStairs(int n) {
        if (n == 0) return 0;
        return countWays(n, 0);
    }
};

int main() {
    std::cout << (new Solution)->climbStairs(4);
}
```

Aadsey's way (Bottom-up Approach).



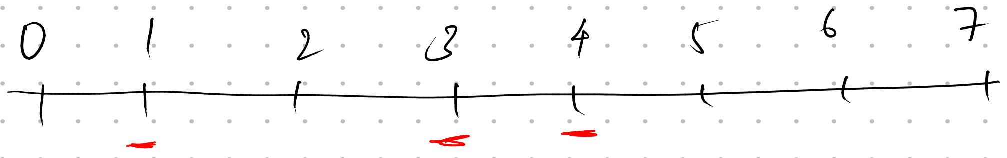
```cpp
class Solution {
public:
    int climbStairs(int n) {
        int dp[n+1];
        dp[0] = 1;
        dp[1] = 1;

        for (int i = 2; i <= n; i++)
            dp[i] = dp[i-1] + dp[i-2];

        return dp[n];
    }
};
```

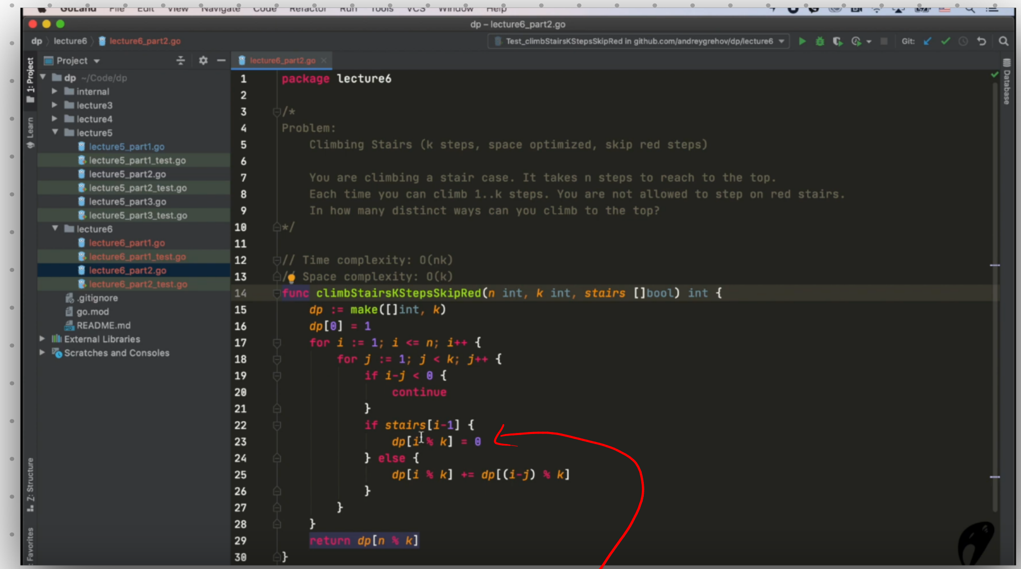
Climbing stairs with some Red stairs.

$n = 7$ total no. of stairs

$k = 3$ no. of steps

Red_stairs = [1, 3, 4]





```go
package lecture6

/*
Problem:
    Climbing Stairs (k steps, space optimized, skip red steps)

    You are climbing a stair case. It takes n steps to reach to the top.
    Each time you can climb 1..k steps. You are not allowed to step on red stairs.
    In how many distinct ways can you climb to the top?
*/

// Time complexity: O(nk)
// Space complexity: O(k)
func climbStairsKStepsSkipRed(n int, k int, stairs []bool) int {
    dp := make([]int, k)
    dp[0] = 1
    for i := 1; i <= n; i++ {
        for j := 1; j < k; j++ {
            if i-j < 0 {
                continue
            }
            if stairs[i-1] {
                dp[i % k] = 0
            } else {
                dp[i % k] += dp[(i-j) % k]
            }
        }
    }
    return dp[n % k]
}
```

See this code (on stepping the Red stairs we make it to '0'). Otherwise we set to the $\boxed{\sum_{i=1}^{k} (n-i)}$

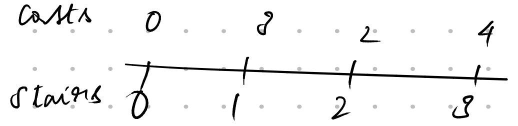
Optimization Problem.

In this we wanna minimize or maximize some function.

Keywords to identify optimization DP: shortest, cheapest, expensive,....

Problem: what is the cheapest route to get to top.

$n = 3$ $\quad K = 2$ $\quad$ cost = [0, 3, 2, 4]



defining objective fn.

i) F(i) is the min cost path to the top.

ii) Identifying the base case.

$F(0) = 0$ $\quad F(2) = 2$

$F(1) = 3$ $\quad F(3) = 6$

iii) Recurrence Relation.

$F(n) = \text{cost}(n) + \min[f[n-1] + f[n-2]]$.

iv) Order of Execution

Bottom-up

v) Location of answer.

$F(n)$