**Dynamic Programming**

Table of Contents

[**Theory** 2](#_Toc168635124)

[What is Dynamic Programming? 2](#_Toc168635125)

[**Sample Code** 2](#_Toc168635126)

[Fibonacci Series – Recursion: 2](#_Toc168635127)

[Fibonacci Series –Top down: 3](#_Toc168635128)

[Fibonacci Series – Bottom Up: 3](#_Toc168635129)

[Fibonacci Series – Bottom Up – Space Optimized: 3](#_Toc168635130)

[LEVEL 1: **EASY** 4](#_Toc168635131)

[1. Climbing Stairs 4](#_Toc168635132)

[2. House Robber 4](#_Toc168635133)

[3. Min cost climbing stairs 4](#_Toc168635134)

[4. Reach to one 4](#_Toc168635135)

[5. Coin change 4](#_Toc168635136)

[6. Longest Increasing Subsequence 4](#_Toc168635137)

[LEVEL 2: **Medium** 5](#_Toc168635138)

[LEVEL 3: **Difficult** 6](#_Toc168635139)

[**SOLUTIONS:** 7](#_Toc168635140)

[**LEVEL 1:** 7](#_Toc168635141)

# **Theory**

## What is Dynamic Programming?

Dynamic programming (DP) is an optimization technique used to solve complex problems by breaking them down into simpler subproblems and storing the results of these subproblems to avoid redundant computations. It is particularly useful for problems with overlapping subproblems and optimal substructure properties.

**Key Concepts of Dynamic Programming:**

1. **Overlapping Subproblems**: Solving the same subproblems multiple times.
2. **Optimal Substructure**: The optimal solution to a problem can be constructed from optimal solutions of its subproblems.
3. **Memoization**: Storing results of subproblems to reuse them, thus saving computation time. Also called top-down approach.
4. **Tabulation**: Building a table in a bottom-up manner to store solutions of subproblems.

**Steps to approach Dynamic Programming problems:**

1. Find the recurrence relationship
2. Find the base case
3. Find way to store solutions of subproblems

# **Sample Code**

## Fibonacci Series – Recursion:

def fib(n):

    if n<=1:

        return n

    else:

        return fib(n-1) + fib(n-2)

# fib : 0,1,1,2,3,5,8

print(fib(6)) #8

Here we see, our solution finding solution of same subproblems several time, which cause slowness. We can prevent this using DP, by saving solution of subproblems in an array.

## Fibonacci Series –Top down:

Here we built solution from n to i, normally via recursion.

def fib\_top\_down(n, dp):

    if n<=1:

        return n

    if dp[n]!= -1:

        return dp[n]

    else:

        dp[n]= fib\_top\_down(n-1, dp) + fib\_top\_down(n-2, dp)

        return dp[n]

n=6

dp = [-1]\*(n+1)

print(fib\_top\_down(n, dp)) #8

## Fibonacci Series – Bottom Up:

Here we built solution from i to n, normally via loop.

def fib\_bottom\_up(n):

    dp = [-1]\*(n+1)

    dp[0], dp[1] = 0, 1

    for i in range(2,n+1):

        dp[i] = dp[i-1] + dp[i-2]

    return dp[n]

print(fib\_bottom\_up(6)) #8

## Fibonacci Series – Bottom Up – Space Optimized:

In place of dp array we can use 2 variables for same purpose

def fib\_bottom\_up(n):

    prev = 1

    prev1 = 0

    curr = 0

    for i in range(2,n+1):

        curr = prev + prev1

        prev1 = prev

        prev = curr

    return prev

print(fib\_bottom\_up(6)) #8

# LEVEL 1: **EASY**

### Climbing Stairs

Link: <https://leetcode.com/problems/climbing-stairs/description/>

### House Robber

You are given money present in n adjacent houses, there is robber who wants to rob the houses. But he cannot rob from 2 adjacent houses. Find max loot of robber.

### Min cost climbing stairs

Link: <https://leetcode.com/problems/min-cost-climbing-stairs/>

### Reach to one

Given a number x, you can do 3 different operations on x: #1. Subtract 1 from it. #2 If it is divisible by 2, divide by 2. #3 If it is divisible by 3, divide by 3 .Find the minimum number of steps that it takes to get to 1 using only the above operations.

### Coin change

Link: <https://leetcode.com/problems/coin-change/description/>

### Longest Increasing Subsequence

Link: <https://leetcode.com/problems/longest-increasing-subsequence/description/>

### Stones

Link: <https://atcoder.jp/contests/dp/tasks/dp_k>

# LEVEL 2: **Medium**

### Ninja training

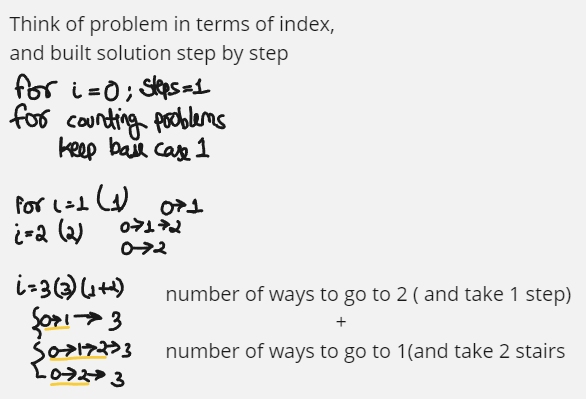
Link: [https://leetcode.com/problems/climbing-stairs/description/](https://www.naukri.com/code360/problems/ninja-s-training_3621003)

# LEVEL 3: **Difficult**

# **SOLUTIONS:**

## **LEVEL 1:**

1. **Climbing Stairs**



# for n = number of ways to go to n-1 + number of ways to go to n-2

# 0->1 for making code easy

# 1->1

# 2 -> 2  (0->2 , 0->1->2)

class Solution:

    def climbStairs(self, n: int) -> int:

        dp = [1]\*(n+1)

        for i in range(2,n+1):

            dp[i] = dp[i-1] + dp[i-2]

        return dp[n]

class Solution:

    def climbStairs(self, n: int) -> int:

        def helper(n):

            if n==0: return 1

            if n==1: return 1

            if dp[n]!=-1 : return dp[n]

            dp[n] = helper(n-1) + helper(n-2)

            return dp[n]

        dp = [-1]\*(n+1)

        return helper(n)

1. **Loot house**

F[i] = max loot done till ith house, so F[i] = max ( arr[i] + F[i-2] , F[i-1] )

*#Loot HOUSE*

def lootBU(n,arr):

    dp=[0]\*(n)

    dp[0],dp[1] = arr[0],max(arr[0],arr[1])

    for i in range(2,n):

        dp[i] = max(arr[i]+dp[i-2] ,dp[i-1])

    print(dp)

    return dp[n-1]

arr = [6,2,3,9]

print(lootBU(len(arr),arr))

1. **Min cost climbing stairs**

Here dp[i] = cost of reaching at ith step.

#Bottom-up

class Solution:

    def minCostClimbingStairs(self, cost: List[int]) -> int:

        dp = [0]\*len(cost)

        dp[0] = cost[0]

        dp[1] = cost[1]

        for i in range(2,len(cost)):

            dp[i] = cost[i]+min(dp[i-1],dp[i-2])

        return min(dp[-1],dp[-2])

#Top-down

class Solution:

    def minCostClimbingStairs(self, cost: List[int]) -> int:

        n=len(cost)

        self.dp = [-1]\*n

        self.dp[0]=cost[0]

        self.dp[1]=cost[1]

        def helper(n):

            if self.dp[n]!=-1:

                return self.dp[n]

            else:

                self.dp[n] = cost[n] + min(helper(n-1),helper(n-2))

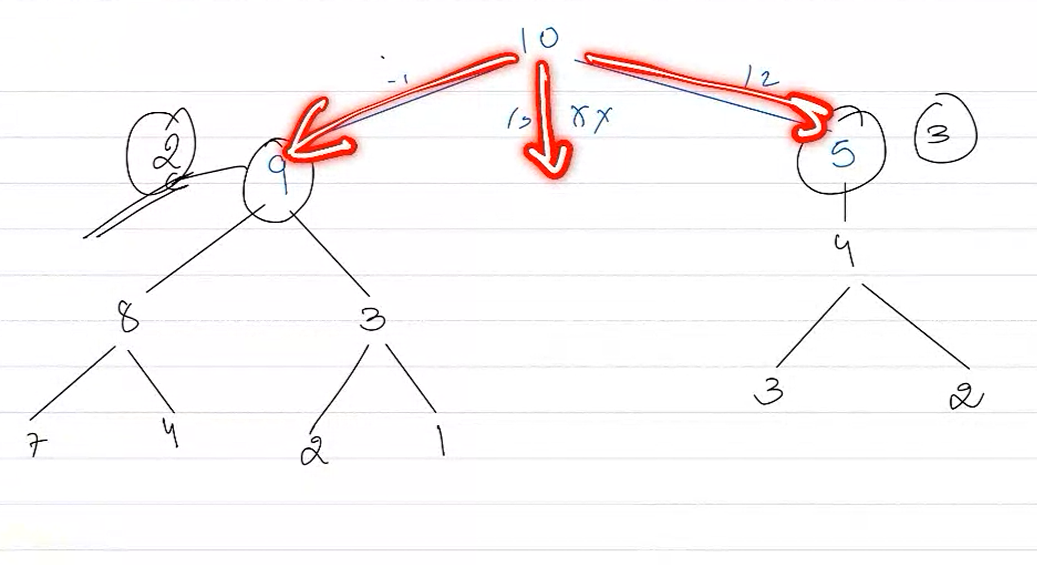
                return self.dp[n]

        helper(n-1)

        return min(self.dp[-1],self.dp[-2])

1. **Minimum steps to reach one**

If see local minimum, going from 10 to 5 is preferred than 10 to 9, however 5 takes more time to go to 1 than 9. Therefore in global way going to 9 is preferred.



*#Top down*

def minStepsToOneTD(n,dp):

    if n==1: return 0

    if n==2 or n==3: return 1

    if dp[n]!=0: return dp[n]

    div\_by\_3, div\_by\_2, less\_by\_1 = float('inf'),float('inf'),float('inf')

    if(n%3==0):

        div\_by\_3 = 1+minStepsToOneTD(n//3,dp)

    if(n%2==0):

        div\_by\_2 = 1+minStepsToOneTD(n//2,dp)

    less\_by\_1 = 1+minStepsToOneTD(n-1,dp)

    dp[n]=min(div\_by\_3, div\_by\_2, less\_by\_1)

    return dp[n]

n=7

dp=[0]\*(n+1)

print(minStepsToOneTD(n,dp))

*#Bottom Up*

def min\_steps\_to\_one(x):

    dp = [1]\*(x+1)

    dp[1]=0

    dp[2],dp[3]=1,1

    for i in range(4,x+1):

        dp[i]=1+min(dp[i-1], dp[i//2] if i%2==0 else x, dp[i//3] if i%3==0 else x)

    return dp[x]

print(min\_steps\_to\_one(10))

1. **Coin change**

#Top down

class Solution:

    def coinChange(self, coins: List[int], amount: int) -> int:

        def helper\_td(n , coins , dp):

            if n==0: return 0

            if dp[n]!=-1:return dp[n]

            temp = float('inf')

            for i in coins:

                if i <= n:

                    temp = min(temp,1+helper\_td(n-i,coins,dp))

            dp[n] = temp

            return dp[n]

        dp = [-1]\*(amount+1)

        ans=helper\_td(amount , coins, dp)

#Bottom up  (performs better both space and time wise)

class Solution:

    def coinChange(self, coins: List[int], amount: int) -> int:

        dp = [float('inf')]\*(amount+1)

        dp[0]=0

        for i in range(1,amount+1):

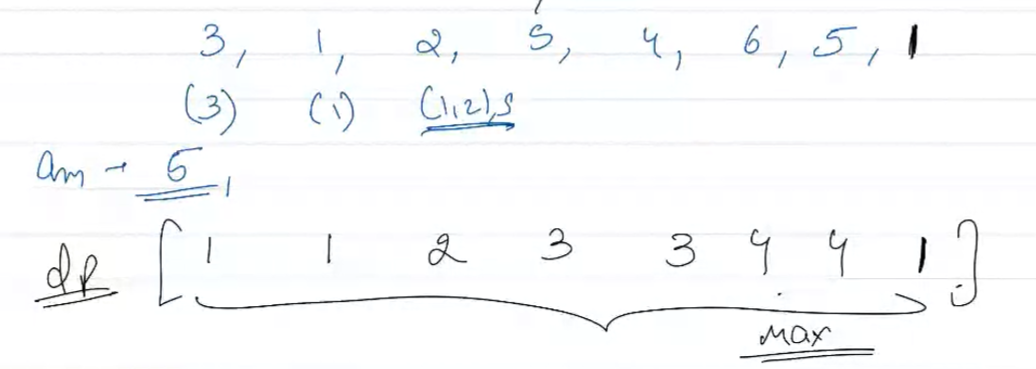
            for j in coins:

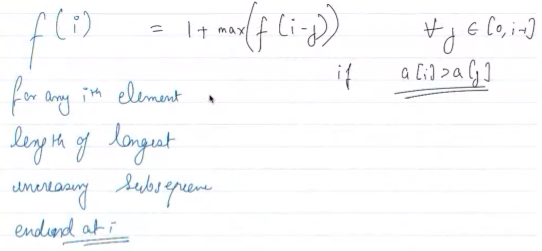
                if j<=i:

                    dp[i] = min(dp[i], 1+dp[i-j])

        return dp[amount] if dp[amount]!=float('inf') else -1

1. **Longest Increasing subsequence**





class Solution:

    def lengthOfLIS(self, nums: List[int]) -> int:

        n = len(nums)

        dp =[1]\*n

        for i in range(n):

            for j in range(i):

                if nums[i]>nums[j]:

                    dp[i] = max(dp[i] , 1+dp[j])

        return max(dp)

1. **Stone**

If k==0 : any player who reaches this state will lose

If k<min(a): here also if any player reaches this state will lose

So winning or losing depends on state and independent of who plays. So if state=k is winning state, player one wins else player 2.

State K is winning state if any state ( k-a[i] ) is losing state for all a[i] in array a. Meaning if first player can push second player to any losing state then first player can win. But if all ( k-a[i] ) states are winning states, then kth state is losing state.

n , k = map(int,input().split())

a = list(map(int,input().split()))

dp = [-1]\*(k+1)

#at state 0 all will lose

dp[0]=0

#if a=[2,3] so at 2 and 3, player takes all stone and next player won't have any stone to pick.

#so that will be winning state.

for i in range(1,k+1):

    flag=0

    for j in a:

        if j<=i and dp[i-j]==0: #can send next player to any losing state

            flag=1

    dp[i]=flag

print("First" if dp[k]==1 else "Second")

## **LEVEL 2:**

def ninjaTraining(n: int, points: List[List[int]]) -> int:

    # Write your code here.

    dp = [[0]\*3 for i in range(n)]

    dp[0][0] = points[0][0]

    dp[0][1] = points[0][1]

    dp[0][2] = points[0][2]

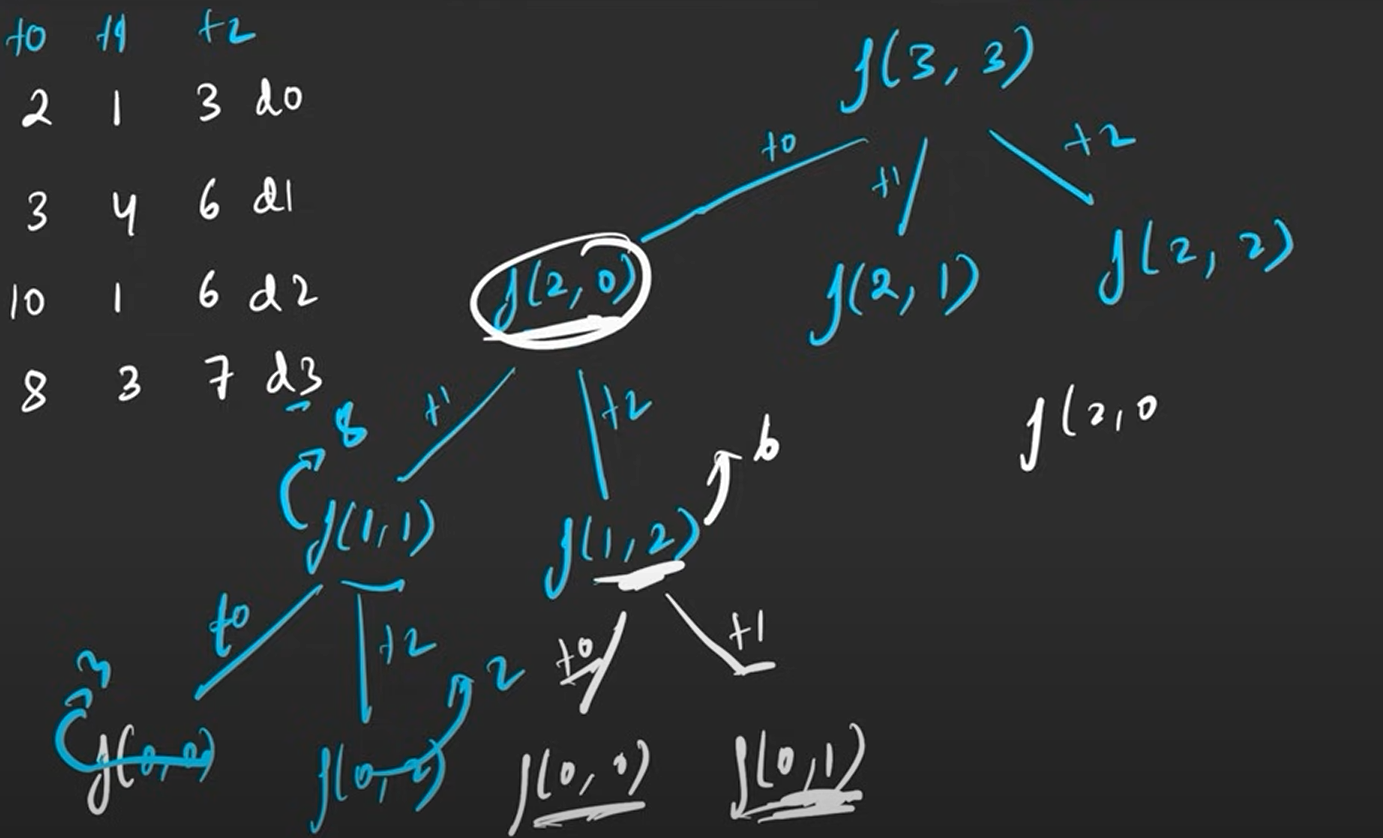
    for i in range(1,n):

        dp[i][0] = points[i][0] + max(dp[i-1][1] , dp[i-1][2])

        dp[i][1] = points[i][1] + max(dp[i-1][0] , dp[i-1][2])

        dp[i][2] = points[i][2] + max(dp[i-1][0] , dp[i-1][1])

    return max(dp[n-1])



Can further do space optimization as in each step we just need to get the previous state. So in place of saving whole dp, just save the previous state.