**Dynamic Programming**

Table of Contents

[**Linear DP:** 2](#_Toc189119931)

[**Multi Dimension DP:** 3](#_Toc189119932)

[**SOLUTIONS:** 4](#_Toc189119933)

[**Linear DP:** 4](#_Toc189119934)

[**Multi Dimension DP:** 7](#_Toc189119935)

<https://leetcode.com/discuss/study-guide/5190444/Dynamic-programming-Problems-Category-wise-(link)/>

<https://leetcode.com/discuss/general-discussion/1050391/Must-do-Dynamic-programming-Problems-Catefory-wise>

https://leetcode.com/problems/partition-array-for-maximum-sum/description/?envType=problem-list-v2&envId=50vlu3z5

# **Linear DP:**

For such questions you need to find the repetitive part of solution and improve it by saving result in array or somewhere, classic example is Fibonacci series. Also use when there is multiple way to do something and we need optimal way, min-max scenarios.

1. Frog Jump

Link: <https://www.naukri.com/code360/problems/frog-jump_3621012>

1. Min cost climbing stairs

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

# **Multi Dimension DP:**

Includes 2D and 3D DP questions.

1. Ninja’s Training

Link: <https://www.naukri.com/code360/problems/ninja-s-training_3621003>

# **SOLUTIONS:**

## **Linear DP:**

1. **Frog Jump**

Express problem in terms of index

def frogJump(*n*: int, *heights*: List[int]) -> int:

    dp = [0]\**(n)*     *#frog can jump to i+1 or i+2*

    dp[0] = 0

*if* *n*==1: *return* 0

    dp[1] = abs(*heights*[0]-*heights*[1])

*for* i *in* range(2,*n*):

        dp[i] = min(dp[i-1]+abs(*heights*[i]-*heights*[i-1]), dp[i-2]+abs(*heights*[i]-*heights*[i-2]))

*return* dp[-1]

1. **Min cost climbing stairs**

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

class Solution: #Bottom-up

    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])

class Solution: #Top-down

    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))

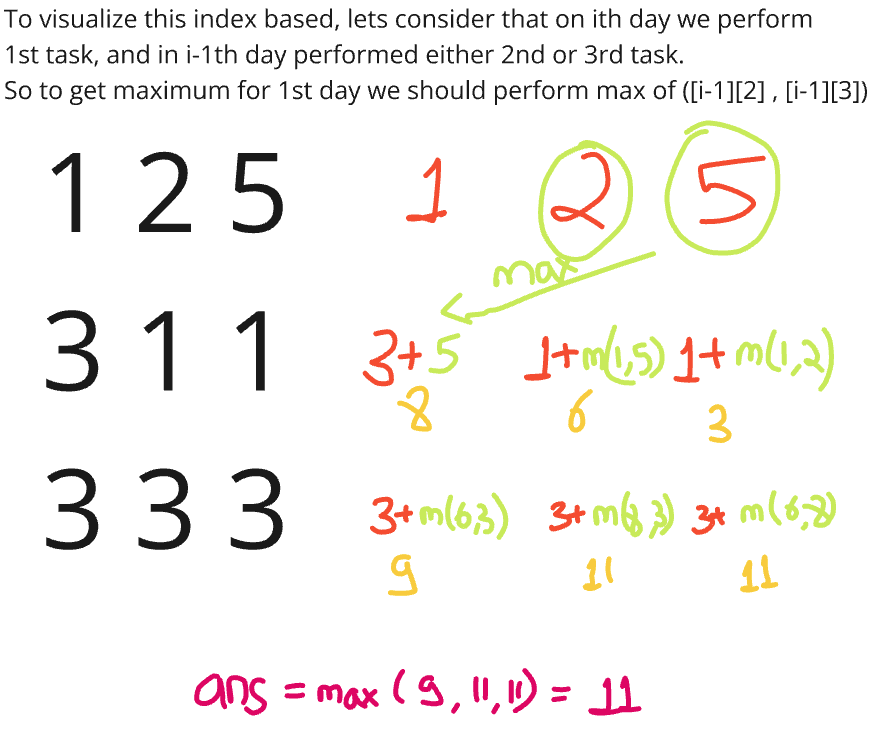
                return self.dp[n]

        helper(n-1)

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

## **Multi Dimension DP:**

1. **Ninja’s Training**



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.