**Dynamic Programming**

‘’Those who cannot remember the past are condemned to repeat it’’ ~ dp

**DP - 2 techniques**

1) **Memoization(Top- down approach)**:

2) **Tabulation (bottom- up approach):**

**For every problem we will be doing 4 things**

* **Recursion**
* **Memoization (Top- down approach)**
* **Tabulation (Bottom- up approach)**
* **Space Optimization**

**How to identify recursive problem?**

If questions is asking these things then definitely you need to get all possible ways using recursion to reach to the answer:

* All possible paths/ ways of doing something
* Max / min cost of all paths/ ways or the best path

**=>How to write recursive solution?**

**Step-1** Express problem in terms of index f(i)/ f(i, j)

**Step-2** Explore all possible operations that should be done on index acc to problem

**Step-3** In the last get the ans acc to ques

- all possible ways/ sum of all ways : sum of all stuffs

- max/ min of all: take min/ max of all stuffs

**=>Recursion -> Memoization?**

Step1- make dp[] acc to the number of indexes

Step2- return the already calculated value: if (dp[i] != -1) return dp[i];

Step3- calculate and store the new value at dp[i] position

**=>Memoization -> Tabulation?**

How we are gonna build the dp[] table:

**Step-1** Use base cases to build the base of the dp[]

**Step-2** whatever left except base cases in memoization, copy that recursion to build dp[] from bottom to up (next of the base to end).

**Step-3** Store every calculated value in dp and return the required value (dp[n-1])

**=>Tabulation -> Space Optimization?**

Remove dimension that is common in recurrence relation. dp[i][j] = prev[j] / next [j]

* 1d problem - 2 variables
* 2d problem- 1d vector
* 3d problem- 2d vector

**Problem-** [**Fibonacci Number**](https://leetcode.com/problems/fibonacci-number/)

**1. \*Recursive solution**

**class Solution {**

**public:**

**int fib(int n) {**

**if(n==0)**

**return 0;**

**if(n==1)**

**return 1;**

**return fib(n-1)+fib(n-2);**

**}**

**};**

//TC = O(N)

//SC = O(N) {recursive stack space}

**2. \*recursion + DP memoization(top -down approach)**

**class Solution {**

**public:**

**//memoization**

**int fib(int n) {**

**vector<int> dp(n+1, -1);**

**return f(dp, n);**

**}**

**int f(vector<int> &dp, int n){**

**if(n<=1)**

**return n;**

**if(dp[n] != -1) return dp[n]; //subproblem solved previously**

**return dp[n] = f(dp, n-1) + f(dp, n-2); //store ans in dp[]**

**}**

**};**

//TC = O(N)

//SC = O(N){recursive stack space} + O(N) {dp[]}

**3. \*DP Tabulation technique (bottom -up approach)**

**class Solution {**

**public:**

**//tabulation(bottom-up approach)**

**int fib(int n) {**

**if(n<=1)**

**return n;**

**vector<int> dp(n+1, -1);**

**dp[0] = 0; dp[1] = 1; //initialize base cases**

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

**dp[i] = dp[i-1] + dp[i-2]; //build from base to answer**

**}**

**return dp[n];**

**}**

**};**

//TC = O(N)

//SC = O(N)

**4. \*DP Tabulization with optimized space complexity(If problem is solved using n-1 and n-2 (i.e, previous values) it can definitely be optimised)**

**class Solution {**

**public:**

**//tabulation(bottom-up approach) with optimized space complexity**

**int fib(int n) {**

**if(n<=1)**

**return n;**

**int prev2 = 0;**

**int prev1 = 1;**

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

**int cur = prev1 + prev2;**

**prev2 = prev1;**

**prev1 = cur;**

**}**

**return prev1;**

**}**

**};**

//TC = O(N)

//SC = O(1)