05/01/2024
Friday

Dynamic Programming

Dynamic Programming is an optimization over plain recursion.
It is a programming technique used to solve the problems comprises -

(a) Overlapping sub-problems.
(b) Optimal sub-structure.

Overlapping sub-problems → Jb same sub-problem is getting repeat again and again.

Optimal sub-structure → Jb ek badi problem ka optimal solution depend kr rha hai same type ki chotti problem ke optimal solution pr.

* DP is like agr ek problem solve kr di ek baar, then fir se solve nhi krenge kyoki uske corresponding answer store kr rkha hai.

* **Approaching a DP problem -**

- Top-Down Approach (Recursion + memoization)
- Bottom-Up Approach (Tabulation method)
- Pattern finding for space optimisation.

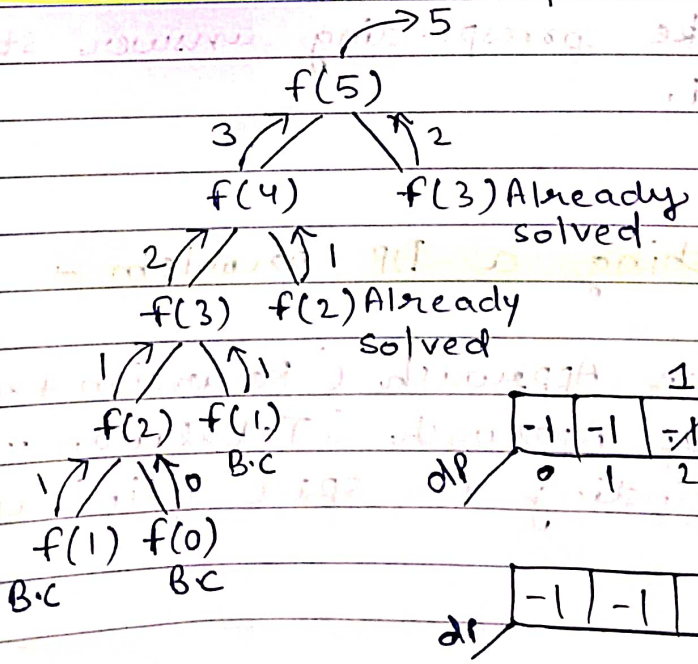
Which is better → Top-down or Bottom-up ?

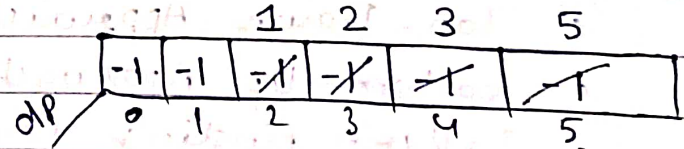
Top-down recursive approach hai.
Bottom-up iterative approach hai.
Top-down me function calls ka overhead aata hai whereas Bottom-up me nhi.
Possibility hai koi solution bottom-up approach se chale top-down se nhi kyoki bottom-up me function calls ka overhead nhi aata.

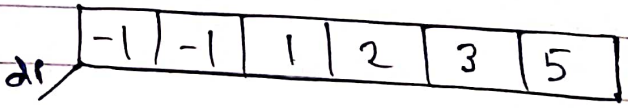
* Top-Down Approach (Recursion + memoization)

Step 1 - Create a DP array.
Step 2 - Store/return answer in DP array.
Step 3 - If answer already exists in DP array, then return it.

* Fibonacci number (Top-Down)







Code –

```
class Solution{
public:

// top-down approach
int solve(int n, vector<int>&dp){

    if(n==0 || n==1) return n;

// step 3: if ans already exists, return it
    if(dp[n] != -1) return dp[n];

//step 2: store/return using DP array
    dp[n] = solve(n-1, dp) + solve(n-2, dp);
    return dp[n];
}

int fib(int n){
  //step 1: create a DP array
    vector<int>dp(n+1, -1);
    int ans = solve(n, dp);
    return ans;
  }
};
```

T.C → O(n). The function is called each time for (0 → n) numbers.

S.C → O(n) → due to DP array
O(n) → due to recursion stack
Overall, O(n).

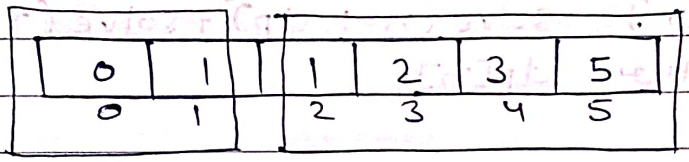
* **Bottom-up Approach (Tabulation method)**

Step 1- Create DP array.

Step 2- Analyse base case and fill DP array accordingly. This will maintain an initial state in DP array.

Step 3- Fill remaining DP array using logic/formula of recursive relation.

Fibonacci series - 0, 1, 1, 2, 3, 5, 8, 13, 21, ----



Fill initial array using base case

Fill remaining array using recursive relation

$dp[2] = dp[1] + dp[0]$

$dp[3] = dp[2] + dp[1]$

$dp[4] = dp[3] + dp[2]$

$dp[5] = dp[4] + dp[3]$

Code -

```
class Solution {
public:

// bottom-up approach

int fib(int n) {
    // create DP array
    vector<int> dp(n+1, -1);

// analyse base case to maintain initial state
// of DP array
        dp[0] = 0;  if (n == 0) return 0;
        dp[1] = 1;

// array size = n+1
// index 0 and 1 fill ho chuke hai
// remaining array to fill, index (2 -> n)

// fill remaining DP array using loop
        for (int i = 2; i <= n; i++) {
            dp[i] = dp[i-1] + dp[i-2];
        }
// return ans
    return dp[n];
    }
};
```

T.C → O(n)

S.C → O(n) → due to DP array

Note - Space optimisation krte time shifting krna bhul jate hai, vo yaad rkhna.

Fibonacci number (Space optimisation)

Fibonacci series - 0, 1, 1, 2, 3, 5, 8, 13, 21 - - - -

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

The n^{th} fibonacci number is the sum of $(n-1)^{th}$ and $(n-2)^{th}$ fibonacci number. Using this pattern, we can optimise the code space-wise.

Code -

```
class Solution {
public:

int fib(int n){
    int prev = 0;
    if (n==0) return 0;
    int curr = 1;
    if (n==1) return 1;

    int ans = 0;
    for(int i=2; i<=n; i++){
        ans = prev + curr;
        prev = curr;
        curr = ans;
    }
    return ans;
}
};
```

* 1-D, 2-D and 3-D DP →

1-D = Recursive function has only one varying argument.
2-D = Recursive function has two varying arguments.
3-D = Recursive function has three varying or non-constant arguments.

Note - (1) 99% cases mein memoization wale code mein jo argument change ho rha hota hai in 1-D dp. vhi return kr rhe hote hai.

(2) Agr memoization wala code TLE de rha h to 99% cases mein argument ko by reference pass krne se chl jata hai code.

Note - hum top-down and bottom-up approach me generally opposite direction me ja rhe hote hai.
for instance, if ~~kisi~~ top-down approach se kisi code mein 0 -> n move kr rhe hai to bottom-up mein n -> 0 move kr rhe honge.