03/06/2023

Dyanamic Programming

It says that learn from the past. It also says that if we have calculated a particular thing, then store it so that we don't have to calculate it again and again.

Phases in DP approach

1) Top-down approach ⇒ Recursion + memoization
2) Bottom-up approach ⇒ Tabulation method
3) Space optimization.

There are 4 phases. 0^{th} phase is apply the recursion.

When to apply DP?

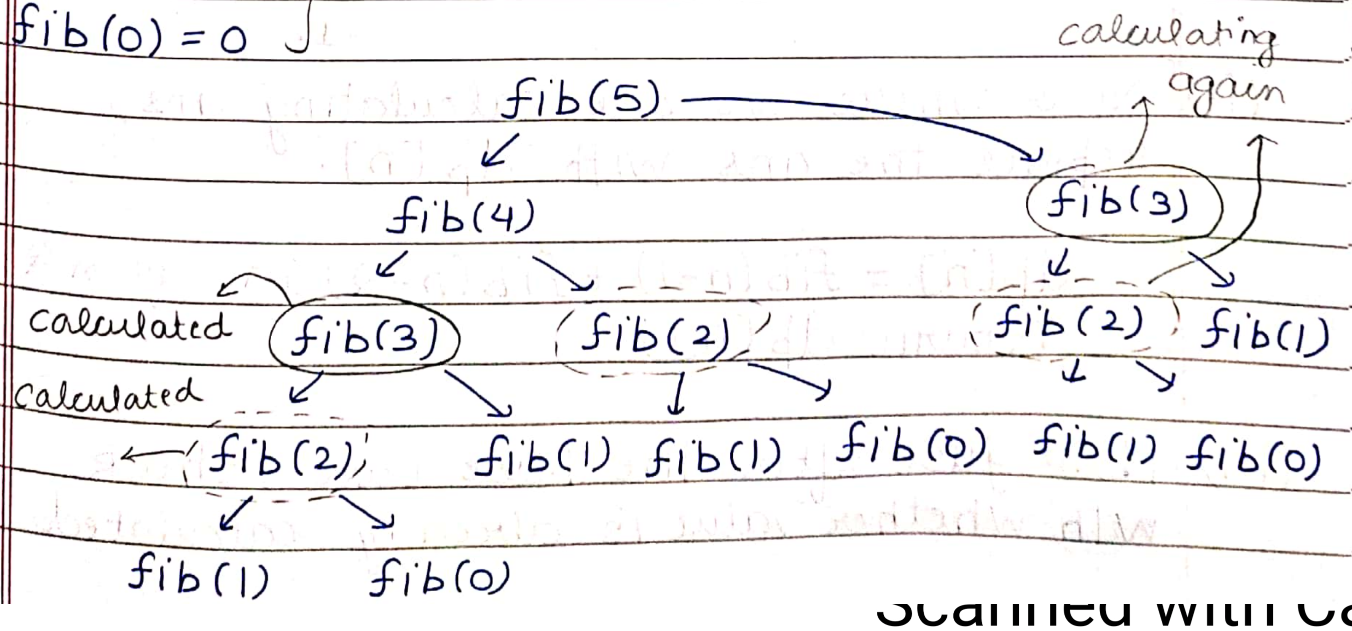
1) Overlapping subproblems → Calculating same problem again and again.
2) Optimal substructure → Optimal solution of bigger problem can be calculated using optimal solution of smaller problems.

Fibonacci number

fib(n) = fib(n-1) + fib(n-2) → Recursive Relation

fib(1) = 1
fib(0) = 0

Base case



Steps

1) Forget about DP and simply write the recursive code.

```
int fib (int n){
    //Base case
    if(n==1 || n==0)
            return n;
    int ans = fib(n-1) + fib(n-2);
    return ans;
}
```

2) Now we need to apply memoization. Memoization means storing the answer. For doing memoization, we need to follow 3 steps

(i) We need to create the dp array. Here we are going from n→0 & hence we need to create n+1 size array.

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

↳ initialize with -1.

(ii) Now where we are calculating ans, replace the ans with dp[n].

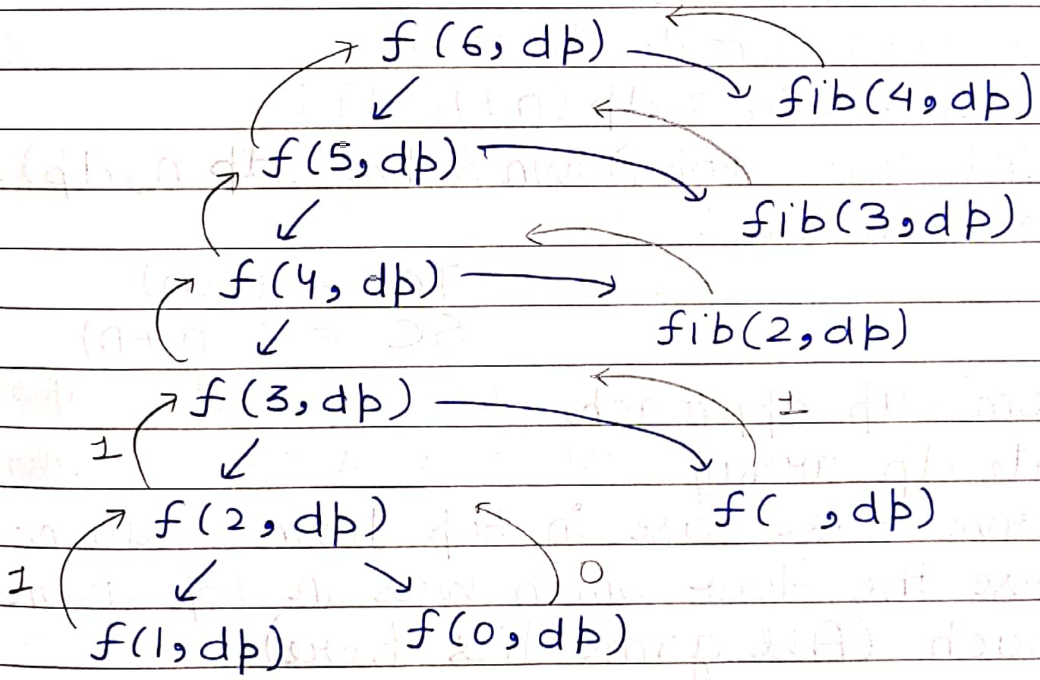
```
dp[n] = fib(n-1) + fib(n-2);
return dp[n];
```

(iii) Now just after the base case, check whether value is already calculated or not.

```
if (dp[n] != -1)
        return dp[n];
```

Dry run

	0	1	2	3	4	5	6
dp →	-1	-1	~~-1~~ 1	~~-1~~ 2	~~-1~~ 3	~~-1~~ 5	~~-1~~ 8



Note → It is called top-down as we are going from n to 0.

0   1 (Base case)

Code

```
int topDownSolve (int n, vector<int>&dp){
    // Base case
    if(n==1 || n==0)
        return n;
    //Step 3: Check if already calculated
```

```
    if (dp[n] != -1)
            return dp[n];
    //Step 2: Replace ans with dp[n]
    dp[n] = topDownSolve(n-1, dp) + topDownSolve
                                        (n-2, dp);
    return dp[n];
}

int fib(int n){
    //Step 1: Create dp array
    vector<int> dp(n+1, -1);
    int ans = topDownSolve(n, dp);
    return ans;
}
```

TC = O(n)
SC = O(n+n)
recursive stack   dp array

Bottom-up approach

1) Create dp array (Not same as top-down always)
2) Observe base case in top down solution.
3) Reverse the flow which was in top-down approach. (All game lies here)

Code

```
int BottomUpSolve(int n) {
    //Step 1: Create dp array
    vector<int> dp(n+1, -1);
    //Step2: Observe base case
    dp[0] = 0;
    if(n == 0)
            return dp[0];
    dp[1] = 1;
    if(n == 1)
            return dp[1];
```

```
//Step 3 : Reverse flow
  for (int i=2 ; i<=n ; i++) {
        dp[i] = dp[i-1] + dp[i-2] ;
  }
  return dp [n] ;
}
int fib (int n){
      int ans = BottomUpSolve (n);
      return ans ;
}
```

TC = O(n)
SC = O(n)
↳ dp array

Dry run

0	1	1	2	3
~~-1~~	~~-1~~	~~-1~~	~~-1~~	~~-1~~
0	1	2	3	4

dp[0] = 0 ;
dp[1] = 1 ;

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

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

i = 4
dp[4] = dp[3] + dp[2] = 2 + 1 = 3

Note → Sometimes like in above question bottom-up approach has better space complexity & in questions where constraints are restricted, top down approach can give error like TLE or memory limit exceeded.

Space optimization

We see that dp[i] is dependent on dp[i-1] and dp[i-2] only and not on the entire array. Hence instead of dp array we can take variables and then move these variables accordingly.

Code

```
int spaceOptSolve (int n){
    // Take variables instead of dp array
    int prev2 = 0;
    int prev1 =  1;
    if (n = =0)
            return prev2;
    if (n = = 1)
            return prev1;
    int curr;
    for (int i=2; i<=n; i++){
        curr = prev1 + prev2;
        prev2 = prev1;
        prev1 = curr;
    }
    return curr;
}
```

Dry run

0 1

↑ ↑

prev2 prev1

* i=2

curr = 0 + 1 = 1
prev2 = 1
prev1 = 1

0 1 1

prev2 prev1

* i = 3
curr = 1 + 1 = 2
prev2 = 1
prev1 = 2

0 1 1 2

prev prev1

* i = 4
curr = 2 + 1 = 3
prev2 = 2
prev1 = 3

0 1 1 2 3

Ans of fib(4).

We have to return curr as answer.

1D array

Note → 1 parameter in function call changing → 1D DP
2 parameter in function call changing → 2D DP

2D array