Dynamic Programming

Those who forget the past are condemned to repeat it,

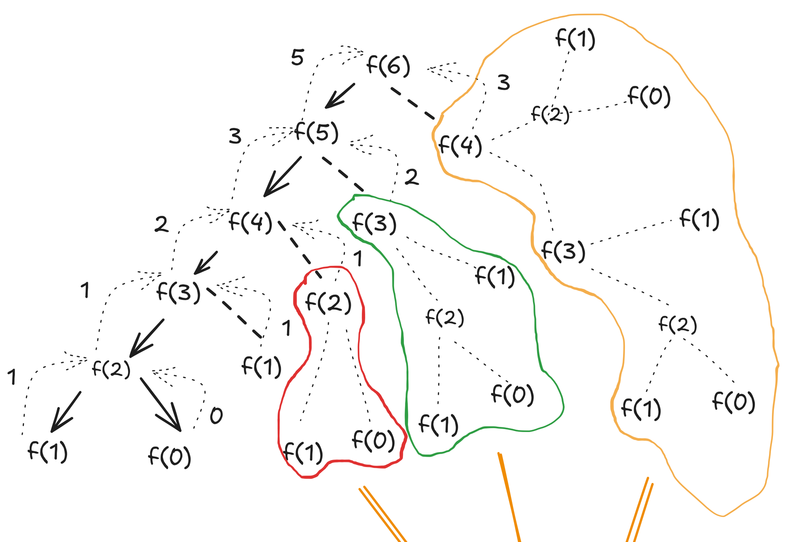
Top Down : Recursion + Memoisation
Bottom Up : Tabulation

When the DP is applied ?

- Problem can be solved by finding the optimum solution of the subproblems.
- Overlapping subproblems should be there.

Fibonacci function : 0 , 1, 1, 2, 3, 5, 8, 13, 21 .... f[n] = f[n-1] + f[n-2]

Recursion Table :



Explaining the flow :

f(6) calls f(5), f(5) calls f(4), f(4) calls f(3),
f(3) calls f(2) , f(2) calls f(1), f(1) returns 1, f(2)
calls f(0), f(0) returns 0, f(2) = 0 + 1 = 1.

f(3) had called f(2), f(2) returns 1, f(3) calls f(1) , it returns 1
f(3) = 2.

f(3) was called by f(4) actually, so f(3) returns 2 and now f(4)
calls f(2) , f(2) will call f(1) which returns 1 and then call f(0) which
return 0, hence f(2) returns 1 and f(4) becomes 3.

f(4) was called by f(5), f(4) returns 3, and then f(5) calls f(3),
for f(3), f(2) is called , f(2) calls f(1) which returns 1 and then it
calls f(0) which returns 0, f(2) does f(1) + f(0) which becomes 1,
f(2) returns 1, then f(3) calls f(1) which returns 1, f(3) returns 2,
f(5) does f(4) + f(3) which becomes 3 + 2 = 5.

f(5) was called by f(6), which returns 5, f(6) also called f(4), f(4)
will call f(3) and f(2), f(3) will call f(2) and f(1) , f(2) will call f(1)
f(0) which becomes 1, f(3) becomes 2, f(4) also calls f(2), f(2) calls
f(1) and f(0) which return 1, f(2) becomes 1, so f(4) becomes 3,
f(6) becomes 5 + 3 as 8.

This is the issue, Why am I calculating
same thing again and again ?
f(4) I already found out in one call, that
its value is 3, why to solve a separeate tree,
It is too much computationally expensive.

Memoization : Storing the values of the subproblems,
like suppose I store f(2),f(4),f(5) etc.

TOP DOWN APPROACH : RECURSION + MEMOIZATION

```
int fib( n -1 , vector<int>& dp){

    // base case
    if( n == 1) {
    return 1;
    }
    else if( n == 0) {
    return 0;
    }

    // Accessing the storage
    if(dp[n] != -1){
        return dp[n];
    }

    // the recursive expression

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

}
```

-1	-1	-1	-1	-1	-1	-1

- Initially all are initialised as - 1.
- From the main function call was : f(6,dp)

  // base case - not executes
  // accessing - dp[6] = -1 , not executes
  // recursive : dp[6] = fib(5,dp) + fib(4,dp).

- Call goes to fib(5,dp) and control goes to
dp[5] = fib(4,dp) + fib (3,dp)

- Call goes to fib(4,dp) and control goes to the line,
dp[4] = fib(3,dp) + fib(2,dp)

- Call goes to fib(3,dp) and control goes to the line,
dp[3] = fib(2,dp) + fib(1,dp)

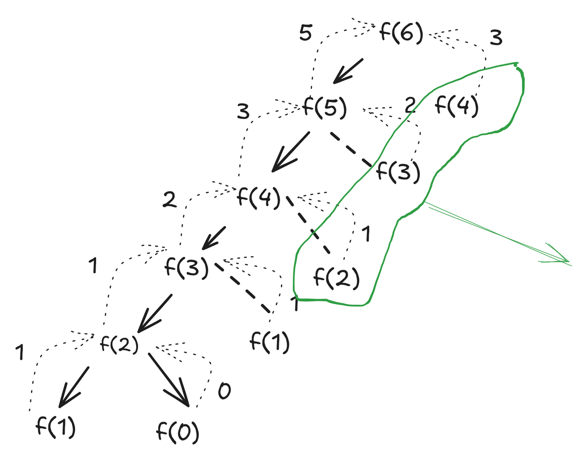
-Call goes to fib(2,dp) and control goes to the line,
dp[2] = fib(1,dp) + fib(0,dp)

- Call goes to fib(1,dp) which returns 1.

- Call goes to fib(0,dp) which returns 0.

-fib(2,dp) : dp[2] = 1 + 0 = 1, it returns dp[2], that is 1.

0	1	2	3	4	5	6
-1	-1	1	-1	-1	-1	-1



Now you can see that
we no longer need those
recursion trees again and the
computation is saved

- fib(3,dp) : dp[3] = return 1 + returned 1, dp [3] = 2

0	1	2	3	4	5	6
-1	-1	1	2	-1	-1	-1

TIME COMPLEXITY : O(n)
SPACE COMPLEXITY : O(n + n)

- fib(4,dp) : dp[4] = return 2 + stored 1 , dp[4] = 3

- fib(5,dp) : dp[5] = return 3 + stored 2, dp[5] = 5

0	1	2	3	4	5	6
-1	-1	1	2	3	5	-1

- fib(6,dp) : dp[6] = return 5 + stored 3, so dp[6] = 8

0	1	2	3	4	5	6
-1	-1	1	2	3	5	8

- return dp[n], ie dp[6], dp[6] = 8

BOTTOM UP APPROACH : TABULARIZATION

```
int main(){

    int n = 6;

    // step 1: define the dp array
    vector<int> dp(n+1);

    // step 2: define the base case
    dp[0] = 0;
    dp[1] = 1;

    // step 3: fill the dp array

    for( int i = 2; i <= n ; i++) {
        dp[i] = dp[i-1] + dp[i-2];
    }

    answer = dp[n];

    return 0;

}
```

0	1	2	3	4	5	6
0						

0	1	2	3	4	5	6
0	1					

0	1	2	3	4	5	6
0	1	1	2			

0	1	2	3	4	5	6
0	1	1	2	3		

0	1	2	3	4	5	6
0	1	1	2	3	5	

0	1	2	3	4	5	6
0	1	1	2	3	5	8

Return this

SPACE OPTIMISATION

```
int main(){

    int n = 6;
    int prev = 1;
    int prevprev = 0;

    for( int i =2; i <=n; i++){
        curr = prev + prevprev;
        prev = curr;
        prevprev = prev;
    }

return curr;
}
```

- for i = 2 :
curr = 1 + 0;

- for i = 3;
curr = 1 + 1;

-for i = 4;
curr = 2 + 1;

- for i = 5;
curr = 3 + 2;

- for i = 6;
curr = 5 + 3

return curr = 8

Return this