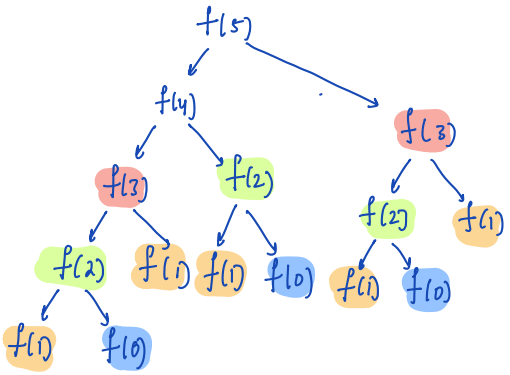
Todays Content :

- Dynamic Programming Intro : {Fib} } 1hr 15mins
- When to use Dp
- Steps for Dp
- # N Stairs } 1hr 15mins
- Dice Sum
- Party Pairs

	0	1	2	3	4	5	6	7	8
fib:	0	1	1	2	3	5	8	13	21




```
int fib(int N){ TC: O(2^N)
    if(N<=1) {return N}
    return { fib(N-1) + fib(N-2) }
}
```

1) Solving a Problem, with SubProblems → Optimal Substructure

2) Solving same Problem more than once → Overlapping Subproblems

↳ By Calling Each Unique SubProblem once → Dynamic Programming

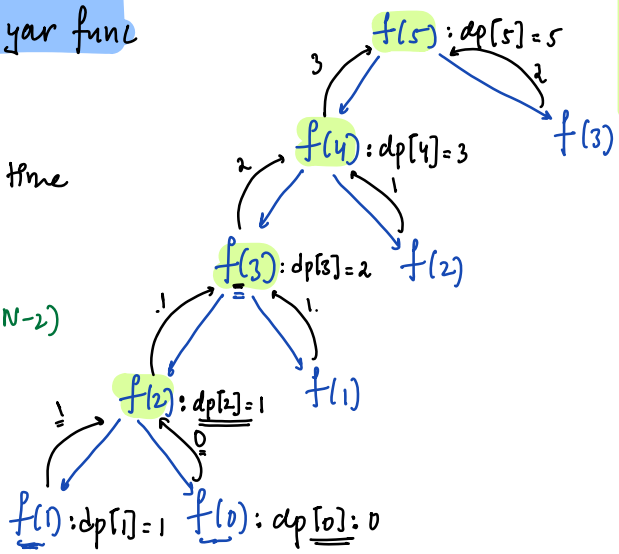
int dp[N+1] = {-1} → Will help us know, if it already calculated

```
int fib(int N){
    if(N<=1){
        dp[N]=N
        return N;
    }
    // fib(N) is called for first time
    if(dp[N]==-1){
        dp[N] = fib(N-1) + fib(N-2)
        return dp[N]
    }
    return dp[N]
}
```

pass dp[] as parameter to your func

int dp[6] =

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



Top-down Dp

Recursion + Memory = Memoization Approach

TC: O(N) * (1) ⇒ O(N), SC: O(N + N) → Call Stack Size : O(N)

```
int fibRec(int N){
    int dp[N+1]
    dp[0] = 0, dp[1] = 1
    for(int i = 2; i <= N; i++){
        dp[i] = dp[i-1] + dp[i-2]
    }
    return dp[N] // {Returning Nth fib Number}
}
```

fibRec(5):

	0	1	2	3	4	5
dp[6] =	0	1	1	2	3	5

dp(0) → dp(1) → dp(2) .. dp(5)

Bottom-up dp ; TC: O(N) * (1) ⇒ O(N), SC: O(N)

Iterative + Memory = Tabulation

// dp[i] = i^{th} fibanaci Number : dpState

// dp[i] = dp[i-1] + dp[i-2] : dp Expression

```
int fib2(int N){
    int a = 0; b = 1;
    int c;
    i = 2; i <= N; i++){
        c = a + b
        a = b
        b = c
    }
    return c;
}
```

0 1 2 3 4 5

a b c

0 1 → 2^{nd}

1 1 → 3^{rd}

1 2 → 4^{th}

2 3 → 5^{th}

TC: O(N) SC: O(1)

Steps : (Dynamic Programming)

1) Solve Problems using Sub Problems : {Optimal Substructure}
2) Overlapping SubProblems

→ Solve unique SubProblem only once, dynamic Programming

1) dp State → What should it contain
2) dp Expression → {How to calculate dp State, using SubProblems}
3) Base Conditions, {for which input values dp Expression will fail}
4) dp table → to store dp values
5) Code & Return ans
6) TC: {# No: of dp States * TC for each State} SC: {dp table size}
7) Optimize:

TC:
- → depends on problems
- We will see in later Problems

SC: In general only possible in iterative codes, based on no: of dp states we use any given instant

N Stairs:

Q) Given N stairs, how many ways we can go from 0^{th} → N^{th} step

Note: from i^{th} step we can directly go to $(i+1)^{th}$ or $(i+2)^{nd}$ step:

Ex: N = 1

: ways: 1

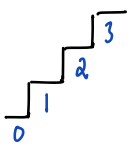
N = 2

: ways: 2

1 1

2

N = 3



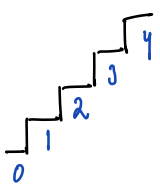
: ways: 3

1 1 1

1 2

2 1

N = 4



: ways: 5

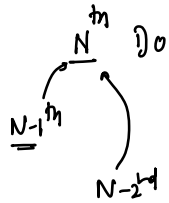
1 1 1 1

1 1 2

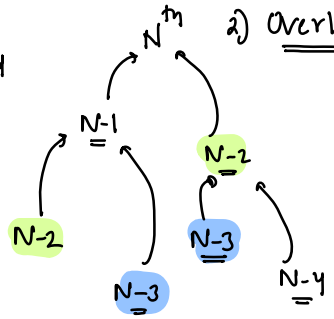
1 2 1

2 1 1

2 2



1) Optimal substructure



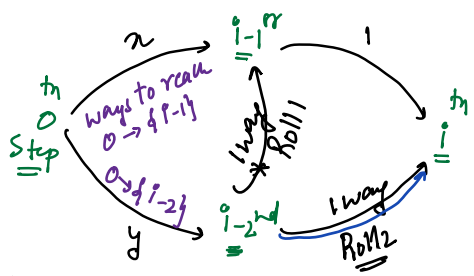
2) Overlapping SubProblems

Dp steps:

Dp State: dp[i] → significance

dp[i] ⇒ # ways to reach i^{th} step

Dp Impression: Calculating dp State Using SubProblems



#ways = x + 2y

$$dp[i] = dp[i-1] + 2\,dp[i-2]$$ * wrong?

dp[1] = 1, dp[2] = 2

dp[3] = dp[2] + 2*dp[1] ⇒ 4 *

dp[4], dp[5] .. dp[N] *

N=4:

Ways:

1111 → 0 → 1 → 2 → 3 → 4

112 → 0 → 1 → 2 → 4

121 → 0 → 1 → 3 → 4

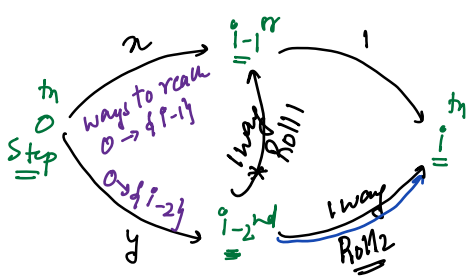
211 → 0 → 2 → 3 → 4

22 → 0 → 2 → 4

0 → 1 → 2 → 3 → 4 * don't consider repeation

0 → 1 → 2 → 4

Correct dp Expression:



Consider ways we can directly reach from (i-1) → i & (i-2) → i

Total ways = n + y

dp[i] = dp[i-1] + dp[i-2] ⇒ Correct dp Expression

Edge Cases:

dp[1] = 1

dp[2] = 2

5 stairs: dp[6]

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

By expression: dp[2] = dp[1] + dp[0]

2 = 1 + dp[0]

dp[0] = 1, if dp[0] = 1, then all value will hold

N=0: 0 steps

Q : No: of ways to reach ground floor = 0 → you cannot reach ground floor *

↳ No: of ways to reach ground floor = 1 → standing there itself

3Q) 2-face : 1, 2 , We can roll dice as many times as we want

No: of ways to get sum N?

N: 1 : ways: 1 : {1}

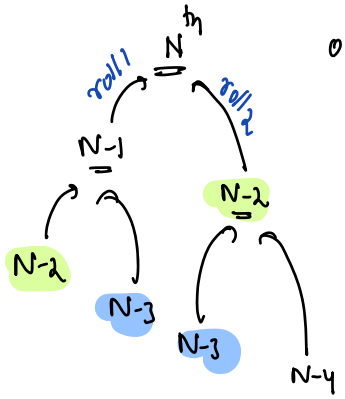
N: 2 : ways: 2 : {1, 1}
{2}

N: 3 ways: 3 : {1, 1, 1}
{1, 2}
{2, 1}

N: 4 ways: 5 : {1, 1, 1, 1}
{1, 1, 2}
{1, 2, 1}
{2, 1, 1}
{2, 2}

Rolling → 1 : Taking 1 step

Rolling → 2 : Taking 2 step



optimal Substructure
&
Overlapping

dp steps:

dp[i] = #ways to get sum i

dp Expression

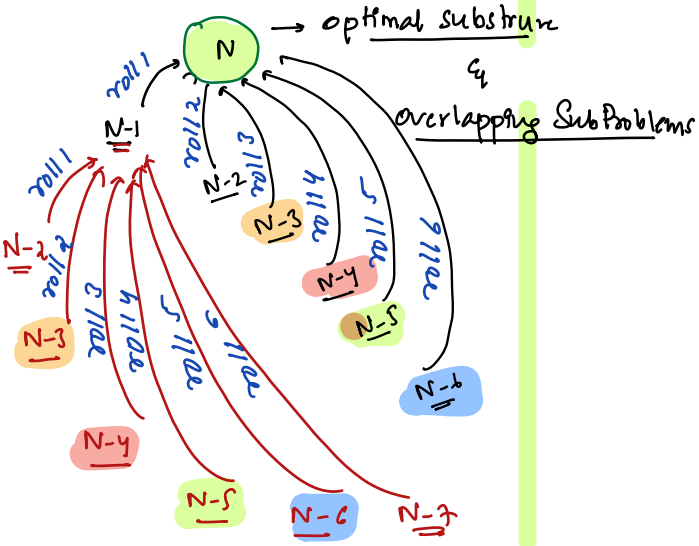
$dp[i] = dp[i-1] + dp[i-2]$

$$dp[i] = \sum_{j=1 \text{ & } i >= j}^{2} dp[i-j]$$

modified Expression

Q) 6-faced dice, we can throw {1 2 3 4 5 6}

No: of ways to get sum = N?



Ex:

N=1 : ways : {1}

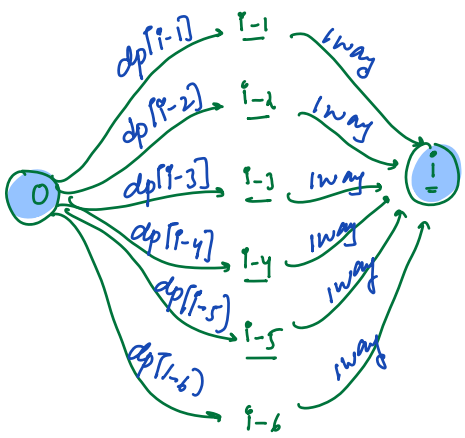
N=2 : ways : 2 {1 1}
{2}

N=3 : ways : 4 {1 1 1}
{1 2}
{2 1}
{3}

N=4 : ways : 8 {1 1 1 1}
{4} {2 1 1} {1 3} {1 1 2}
{3 1} {2 2} {1 2 1}

dp steps:

1) dp[i] = # ways to get sum i



// dp Expression

2) $dp[i] = dp[i-1] + dp[i-2] + dp[i-3] + dp[i-4] + dp[i-5] + dp[i-6]$

$$dp[i] = \sum_{j=1}^{6} dp[i-j]$$

3) Base: dp[0] = 1, dp[1] = 1, dp[2] = 2, dp[3] = 4, dp[4] = 8, dp[5] = 16

But it's highly time consuming

Modified dp Expression:

$$dp[i] = \sum_{j=1 \,\&\&\, i>=j}^{6} dp[i-j]$$

Edge Case:

$dp[0] = 1$

$$dp[1] = \{ j: 1, \; dp[0] = 1 \}$$

$$dp[2] = \{ j: 1, 2, \; dp[1] + dp[0] = 1+1 = 2 \}$$

$$dp[3] = \{ j: 1, 2, 3, \; dp[2]+dp[1]+dp[0] = 1+1+2 = 4 \}$$

$$dp[4] = \{ j: 1, 2, 3, 4, \; dp[3]+dp[2]+dp[1]+dp[0] = 4+2+1+1 = 8 \}$$

Code:

```
int DiceSum(int N)
{
    int dp[N+1];
    dp[0] = 1;
    for(int i=1; i<=N; i++){
        int s=0;
        for(int j=1; j<=6 && j<=i; j++){
            s = s + dp[i-j]
        }
        dp[i] = s;
    }
    return dp[N]
}
```

TC: $\to (\#N)*(6)$ → TC $\to O(N)$

SC: $O(N)$

optimization: TODO