Fibonacci

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

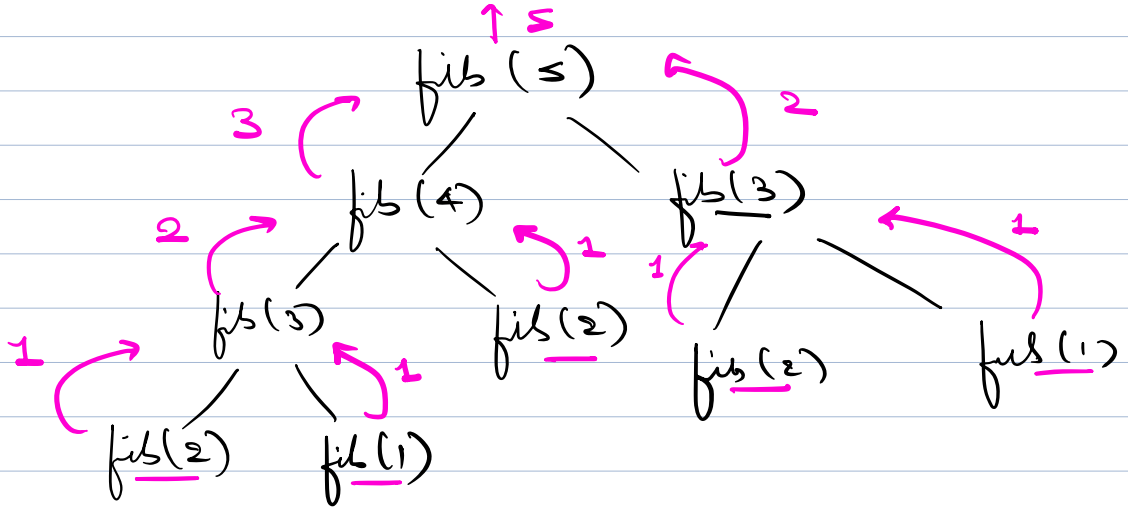
0, 1, 1, 2, 3, 5 ....

Code

```
int fib(n) {
    if (n == 1 || n == 2)
        return 1;
}

return fib(n-1) + fib(n-2)
}
```

Recursive Tree (N = 5)



T.C. = $O(2^N)$

S.C. = $O(N)$ → Recursive stack

T.C. of recursion = (# Recursive calls) × (Time / recursive Call)

2^N = (# Recursive calls) × O(1)

Recursive calls = $O(2^N)$

Nodes in Recursive tree = $O(2^N)$ ⇒ States

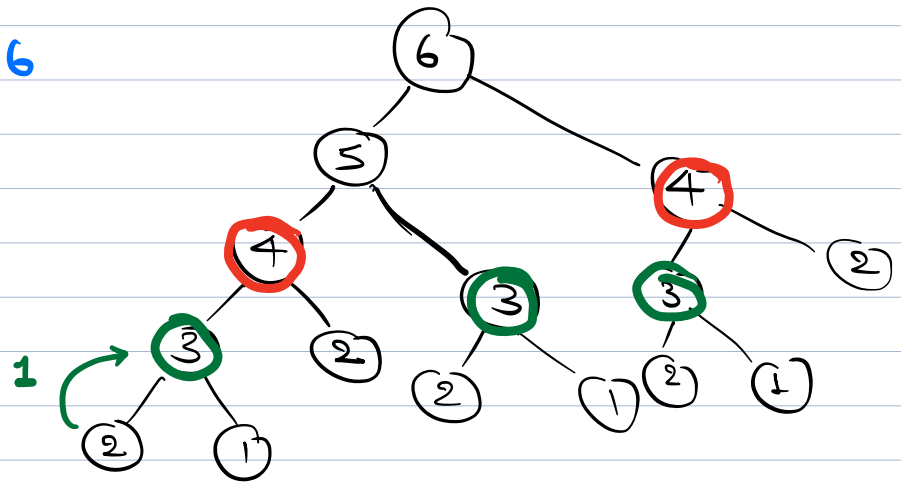
Properties

1) Optimal Substructure

⇒ Any given problem has optimal substructure if the optimal sol^n of the given problem can be obtained using the optimal solution of the subproblem.

2) Overlapping Subproblem

N = 6



⇒ Calculating the same subproblem multiple times.

⇒ If you are able to observe these 2 properties, you can apply DP

Memoization

⇒ Storing the value of a subproblem that you have computed to use it again in the future.

states that you need to compute > # Unique states.

$O(2^N)$ $O(N)$

Calculate fibonacci Using Memoization

N unique states ⇒ [1, N]

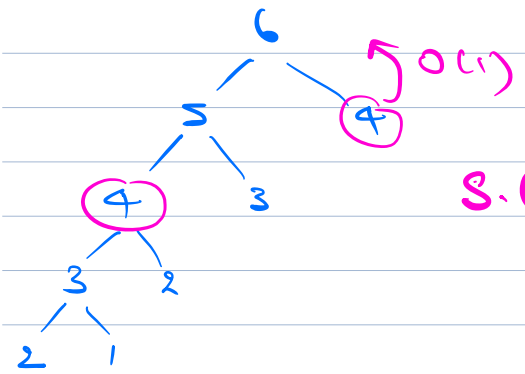
↓

Array of size N+1

Code

```
fib[N+1] = {-1};
int fibonacci (N) {
    if (N==1 || N==2) { return 1;}
    if (fib[N] != (-1)) {
            return fib[N];
    }
    fib[N] = fib(n-1) + fib(n-2);
    return fib[N];
}
```

T.C. = O(N)



S.C. = O(N) + O(N)

↓ Stack ↓ Memoization

⇒ Recursive + Memoization.

2 ways to solve DP problems.

1) Top Down DP

(Recursion + Memoization)

⇒ We start with a bigger problem.

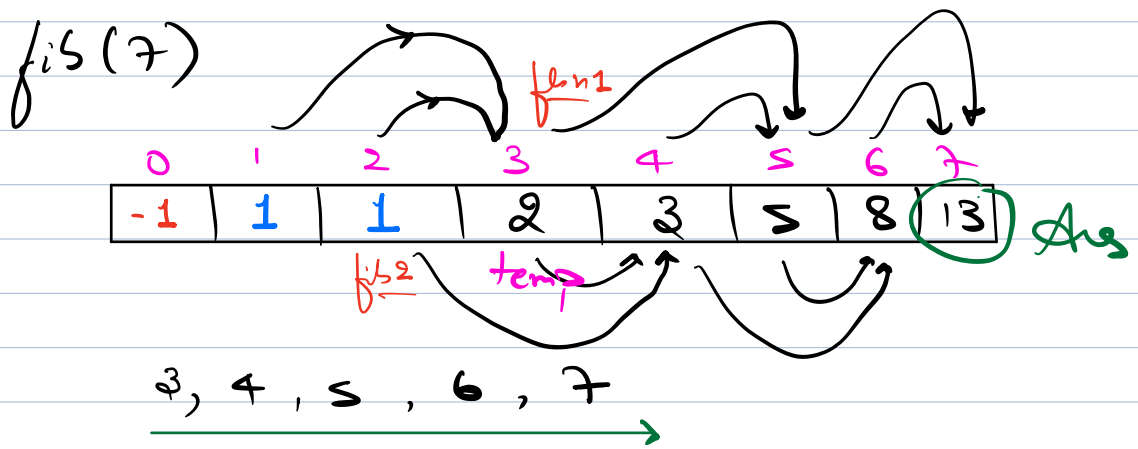
⇒ Go down recursive to smaller subproblems for which we already know the answer. (Base Cases)

2) Bottom Up DP.

(Iterative)

⇒ Start from the subproblems for which you know the solution & use that to iteratively solve the bigger problems.

$$fib(n) = fib(n-1) + fib(n-2);$$



Code

```
fib[N+1];

fib[1] = 1;
fib[2] = 1;

for (i=3; i<=N; i++) {
        fib[i] = fib[i-1] + fib[i-2];
}
```

T.C. = O(N)

S.C. = O(N) (No stack space)

↓

Memoization.

NOTE: In Bottom up DP, we need to be aware about the order in which the states needs to be computed.

```
fibn1 = 1;
fibn2 = 1;

for (i=3; i<=N; i++) {
    int temp = fibn1 + fibn2;
    fibn2 = fibn1;
    fibn1 = temp;
}

return fibn2;
```

T.C. = O(N)

S.C. = O(1)

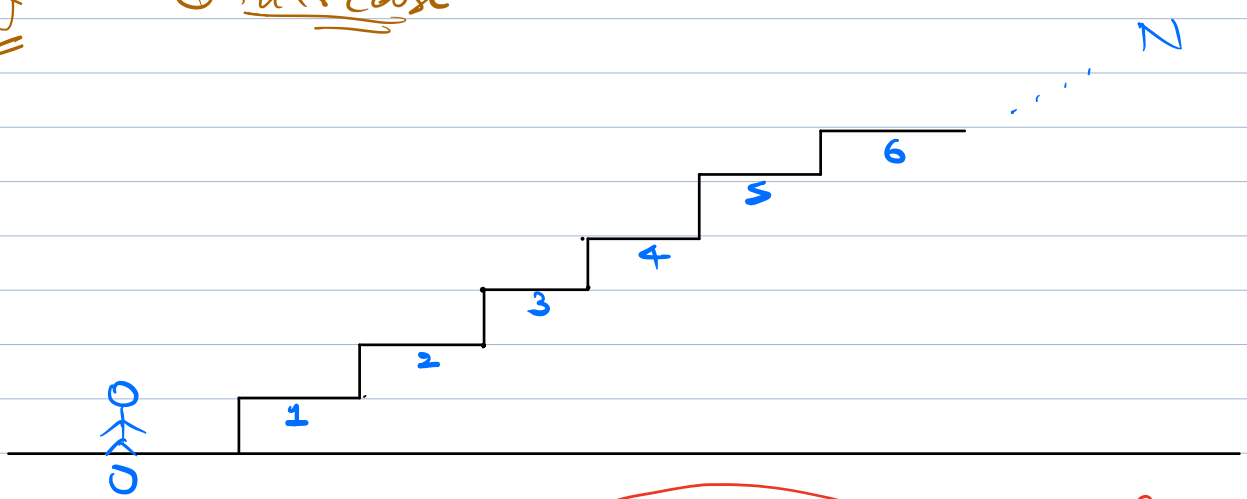
DP State

DP[i] → What are you computing in the ith state.

DP[i] ⇒ ith fibonacci no. (fib(i))

DP[i] = DP[i-1] + DP[i-2];

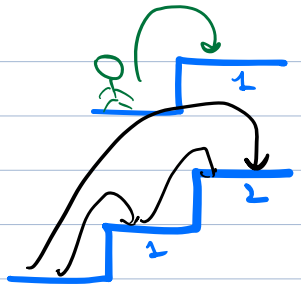
Staircase



One step → Jump 1 stair or Jump 2 stairs. ⇒ choice

Find the no. of ways to reach Nth staircase.

N = 1



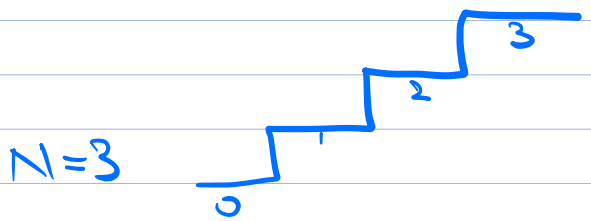
(1)

N = 2

0 → 1 → 2

0 ⟶ 2

(2)



N = 3

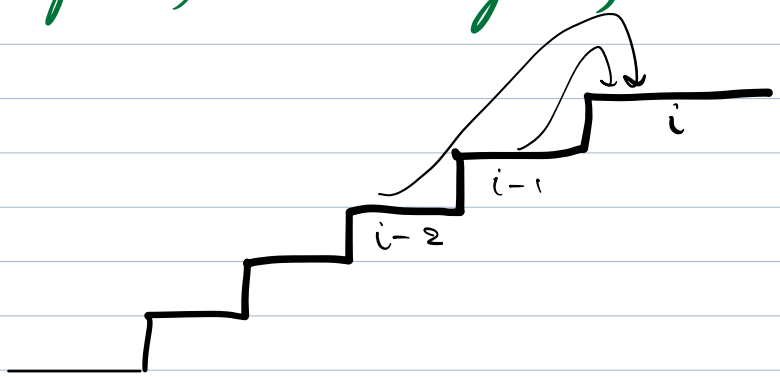
✓ ✓ 0 → 1 → 2 → 3

✓ 0 → 1 ⟶ 3

✓ 0 ⟶ 2 → 3

(3)

Ways (3) = ways (2) + ways (1)



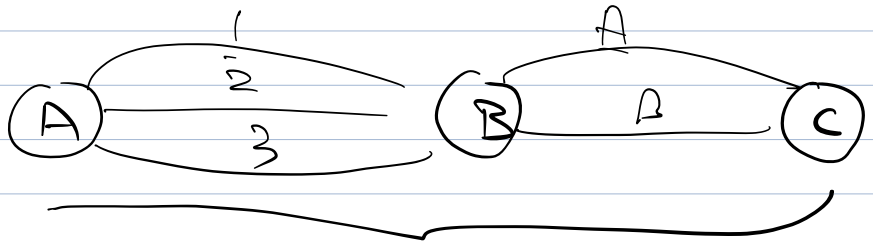
ways (i) = ways (i-1) × (no. of ways to reach from i-1 to i)

1

+

ways (i-2) × (no. of ways to reach from i-2 to i)

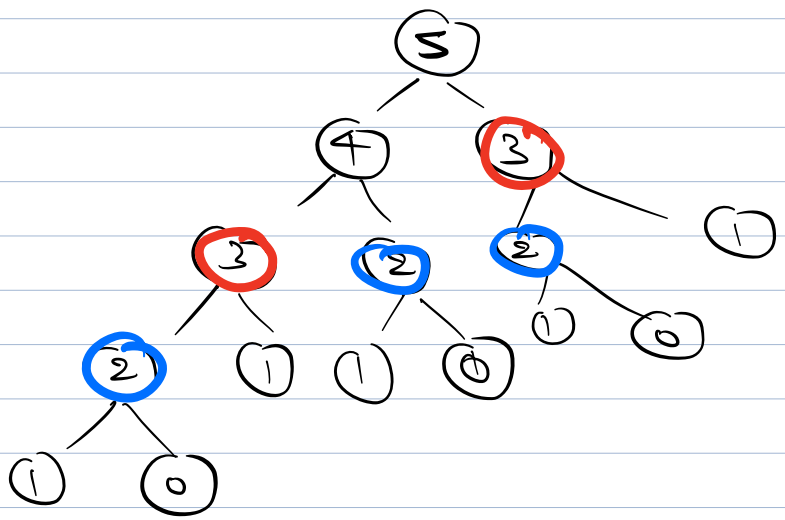
1



$2 \times 3 = 6$

1) Optimal Substructure is present

2) Overlapping Subproblems ??



Base Cases

$n = 0$, 0

$n = 1$, 1

$n = 2$, 2

✱ Steps to Solve DP Problem

1) Elements Of Choice (Step 1 size 1 / or / Step 1 size 2)

2) How to represent a state (What should my state represent)

→ No. of ways to reach i^{th} stair ⇒ Ways(i)

3) Use Step 1 to write recursive solution.

↳ (3.1) Overlapping subproblems.

4) Which state is the final answer of the problem

[1 → (N)] → N^{th} state (Ways(N))

(3) ways(i)

(i-1) → ways(i-1)

+

(i-2) → ways(i-2)

Mutually Exclusive & Mutually Exhaustive

$$ways(i) = ways(i-1) + ways(i-2)$$

1) Mutually Exclusive

⇒ Choices should not overlap.

2) Mutually Exhaustive

⇒ Our choices should cover all the cases