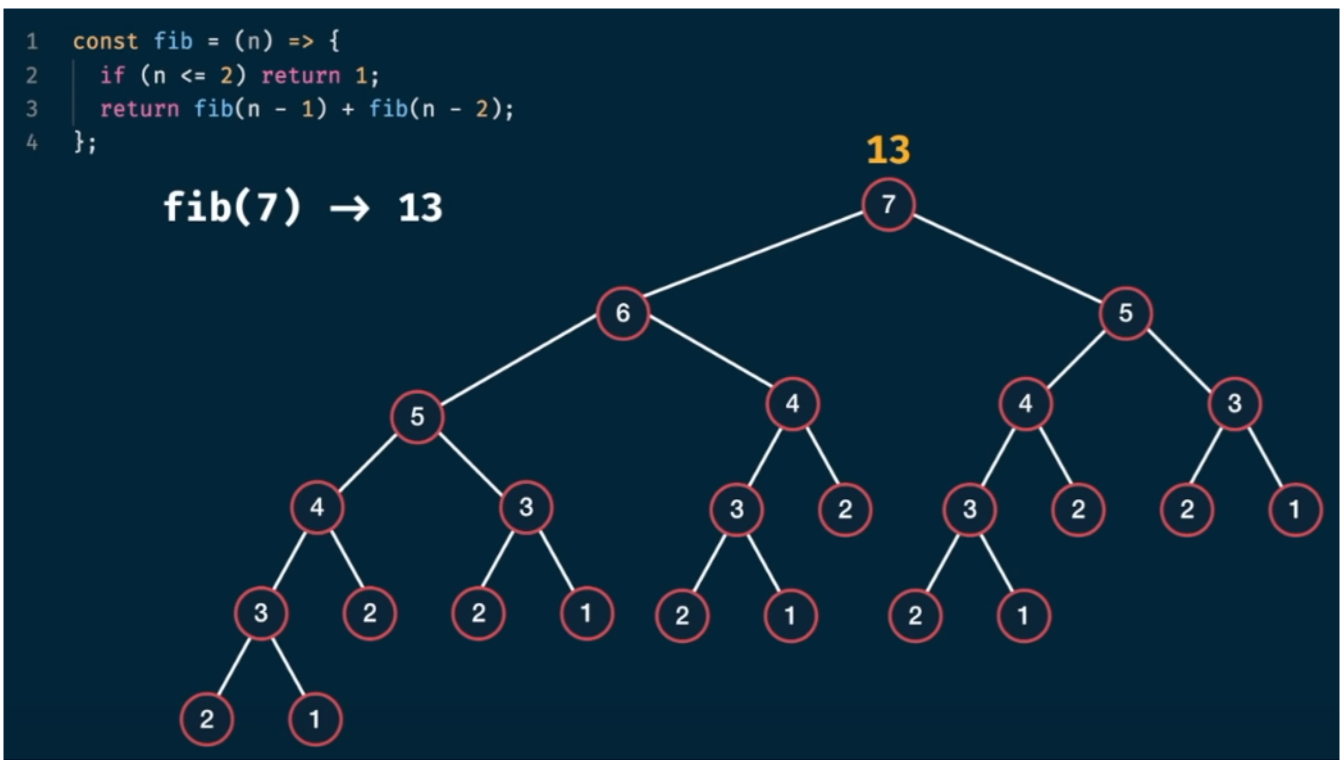
DP. Memoization

Fibonacci Sequence

```
int fib(int n){
   if(n <= 2){
       return 1;
   }
   else {
       return fib(n-1) + fib(n-2);
   }
}
```

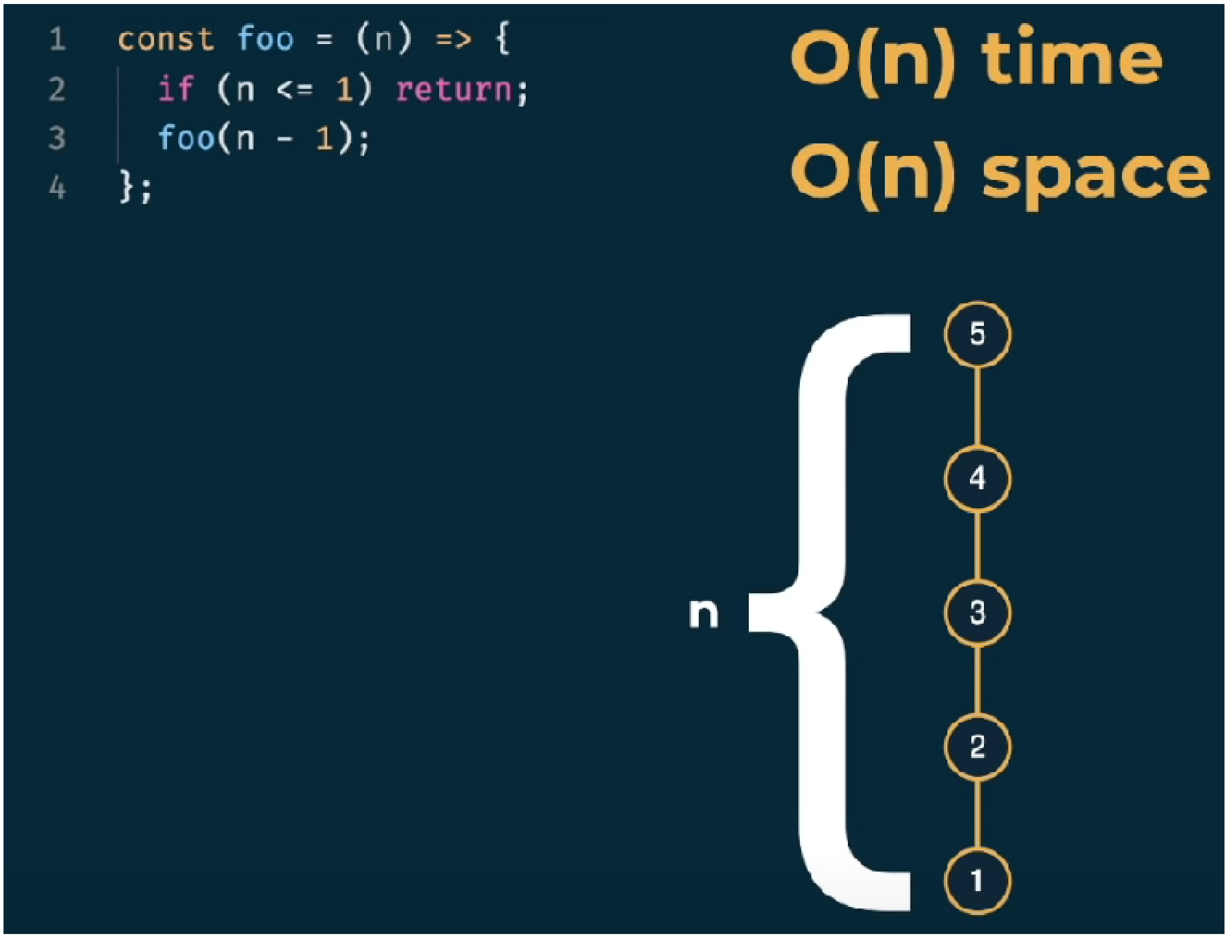
```
const fib = (n) =>{
    if(n <= 2) return 1;
    return fib(n-1) + fib(n-2);
}

console.log(fib(7));
```



- Time complexity of fib function is `O(2^n)`

Time and space complexity of recursive function

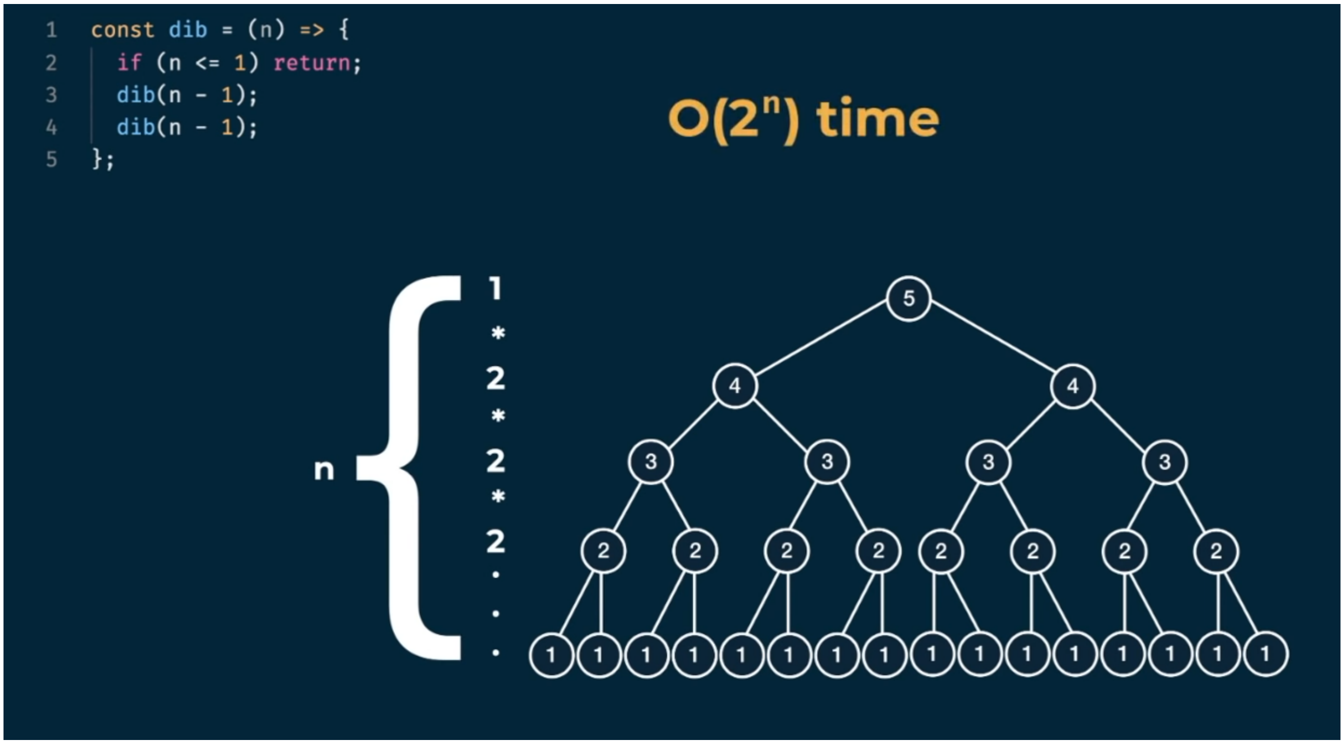
- Analysing time complexity of a recursive function
-

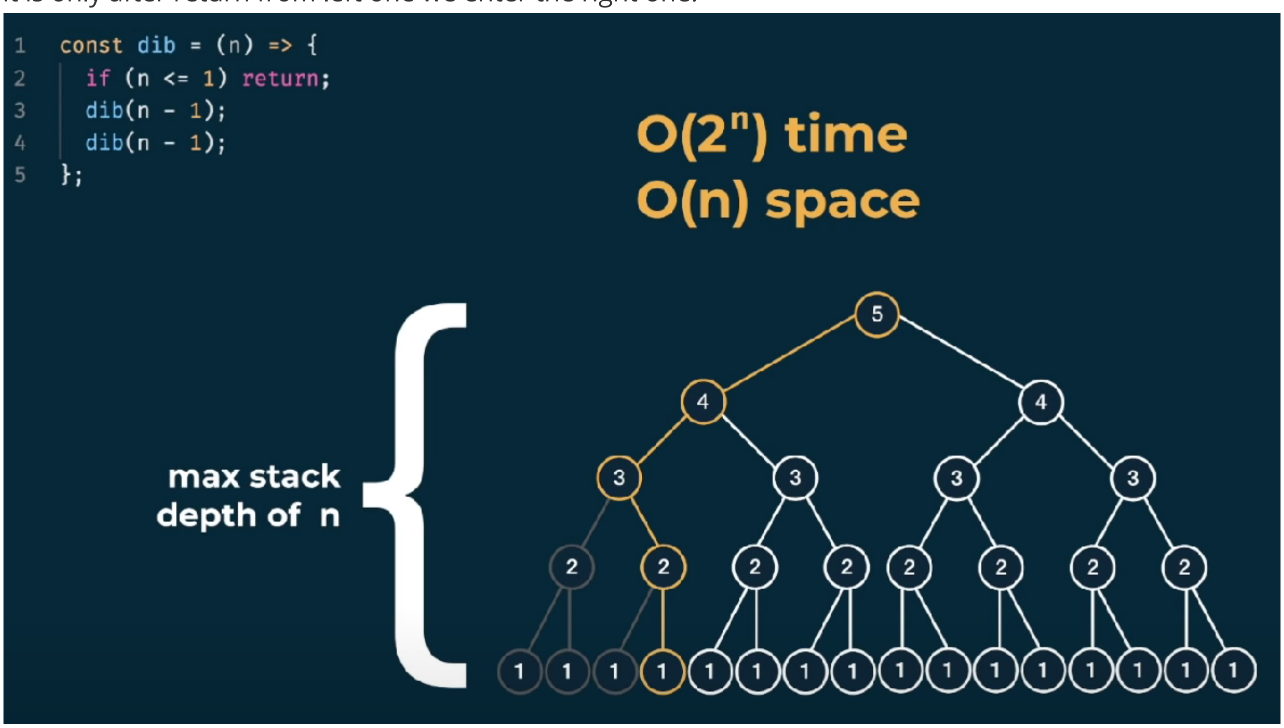


- Space complexity
  - when we analyse space complexity of recursive functions We should include any of the additional stack space
- for a function

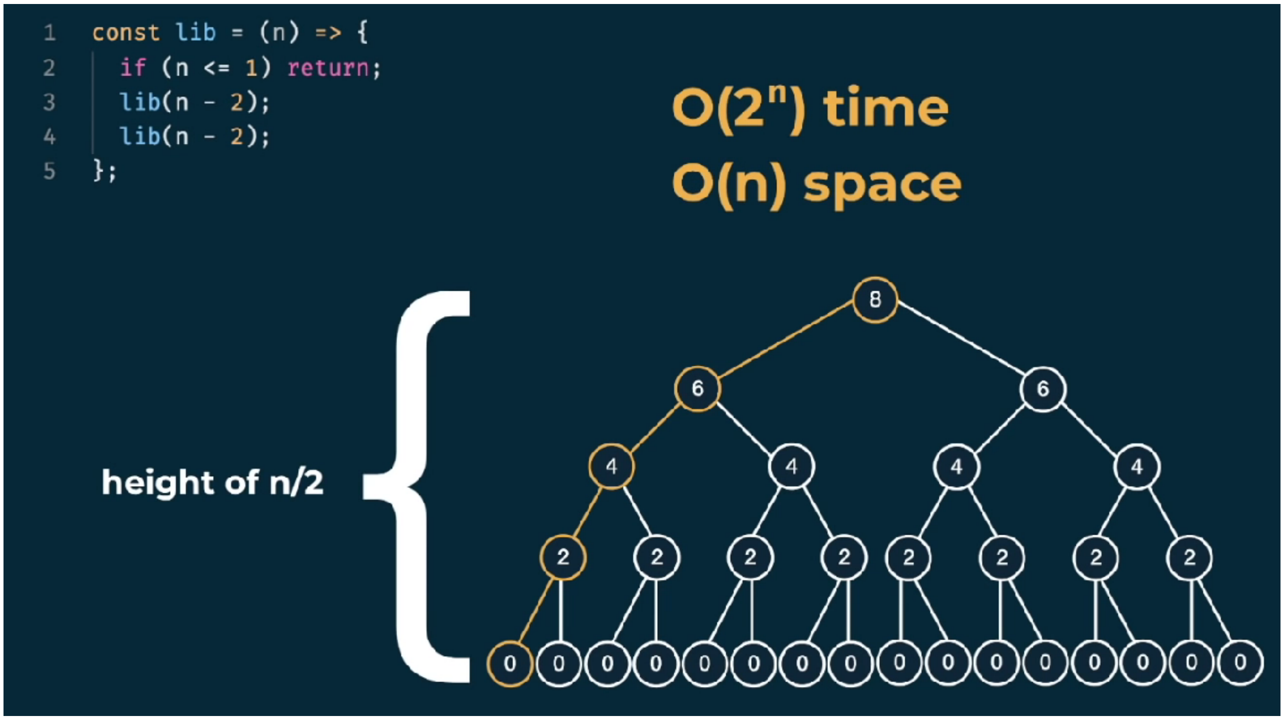
```
void dib(int n){
    if(n <= 1){
        return;
    }
    dib(n-1);
    dib(n-1);
}
```

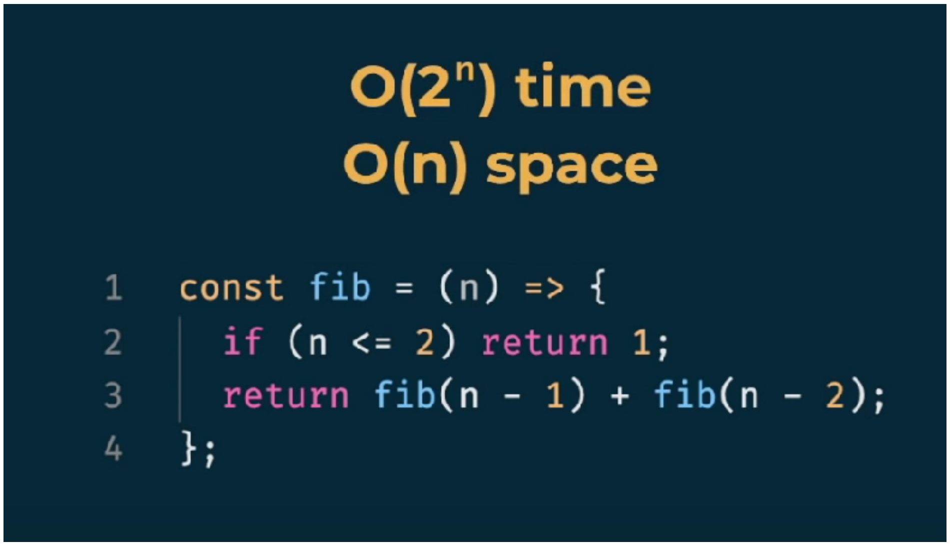
- Height is the distance between the root node and the furthest leaf node
- At every level we are calling the functions doubling the number of function calls
- for `n` levels our time complexity becomes `O(2^n)`



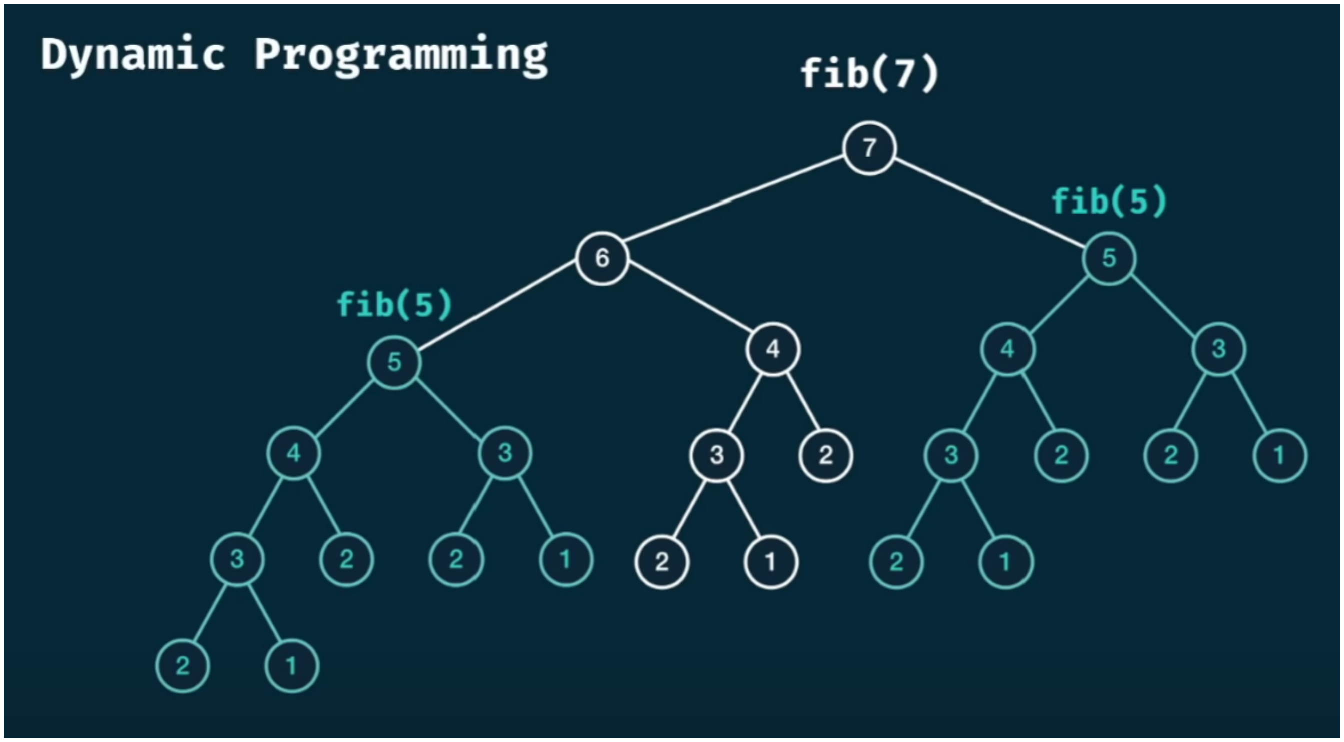
- At any point we use only 5 stacks for `dib(5)`
- Because after reaching five stack calls the functions returns
- it is only after return from left one we enter the right one.
- 

- Time complexity is `O(n^2)` and space complexity is `O(n)`

- 

- Our Fibonacci function has `O(2^n)` time complexity and `O(n)` space complexity
- 

Dynamic programming definition



- When we have some larger problem, we can decompose into smaller instances of the same problem. This concept of break down is called **Dynamic Programming**

memoization

- We store duplicate subproblems as we can get those results later on.
- To implement a memoization we pick a fast access data structure
- Usually a HashMap or a JavaScript object
- JavaScript passes data to function by reference

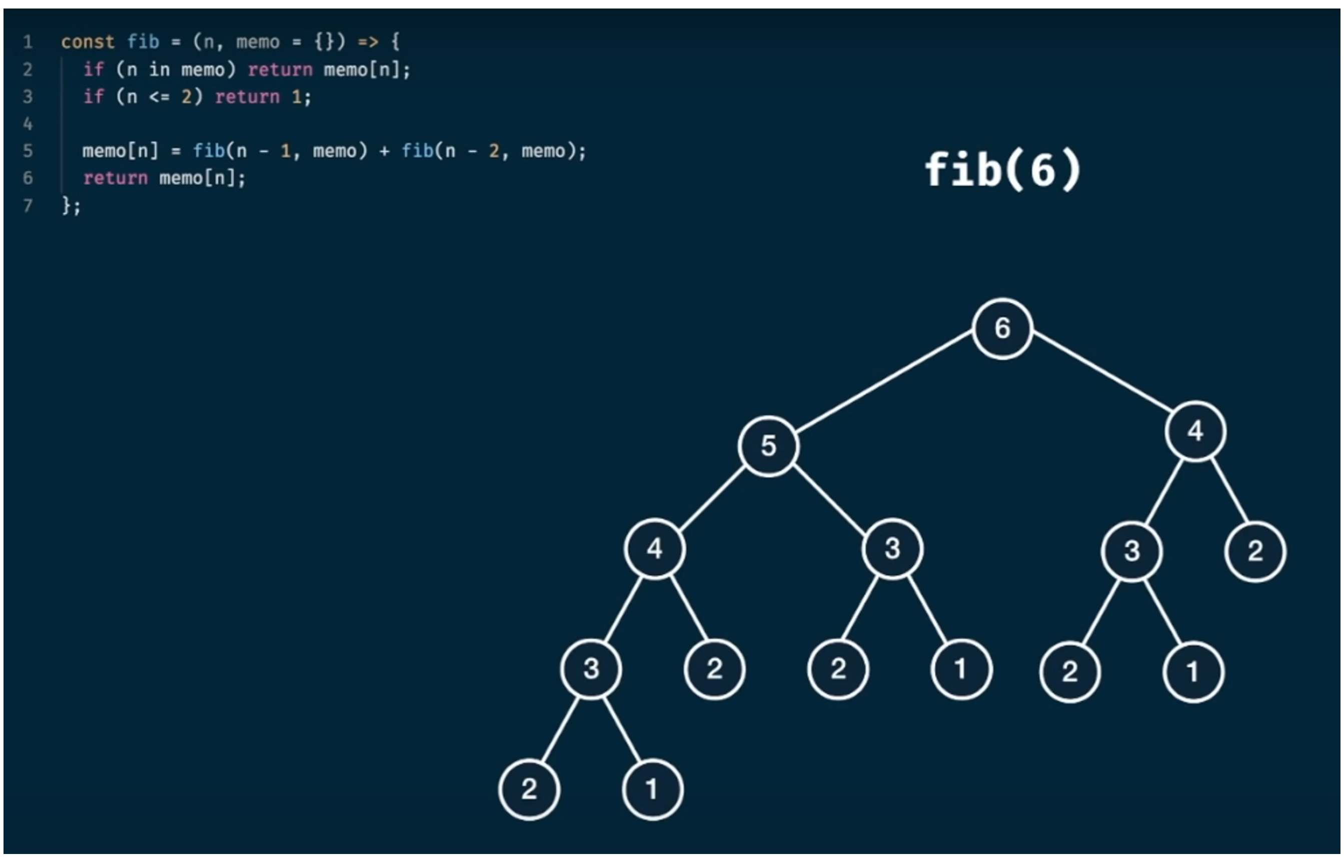
Memoization for Fibonacci

```
int fib(int n){
    // create a memo
    static unordered_map<int, int> memo;
    // find the number
    auto it = memo.find(n);
    if(it != memo.end()){
        return memo[n];
    }

    if(n <= 2){
        return 1;
    }

    // add to the memo
    memo[n] = fib(n-1) + fib(n-2);

    // return from memo
    return memo[n];
}
```

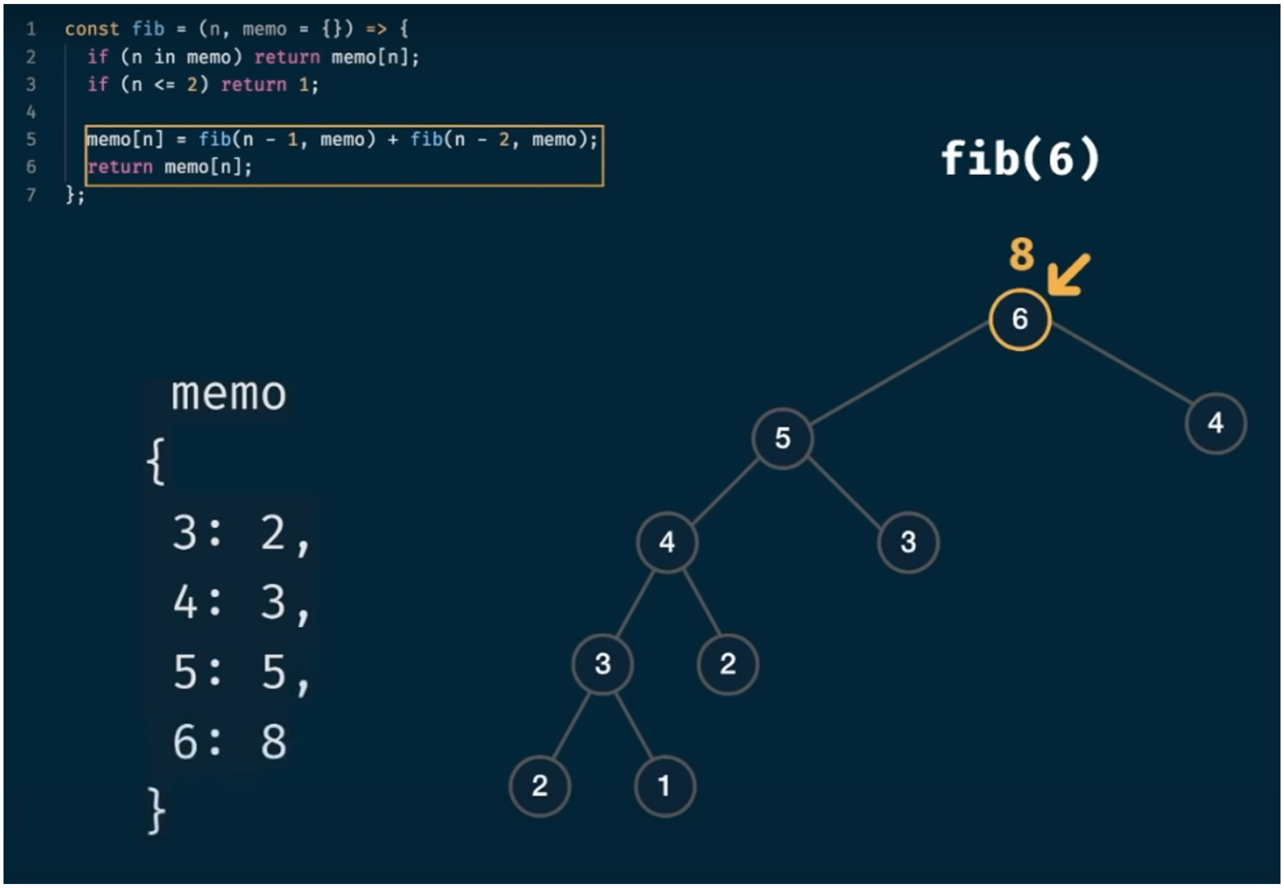
```
const fib = (n, memo = {}) =>{
    // check in memo
    if(n in memo) {
        return memo[n];
    }

    if(n <= 2) {
        return 1;
    }

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

console.log(fib(30));
```

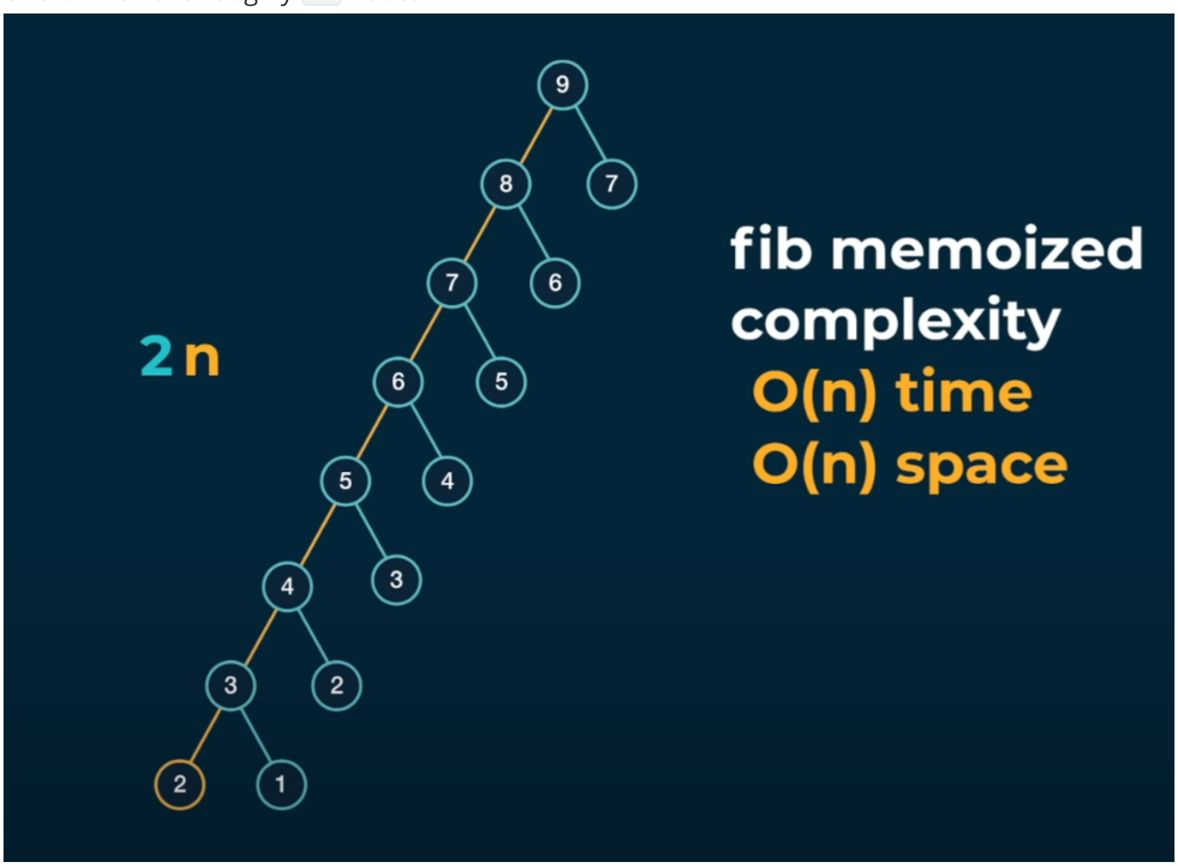
Memoized solution recursion tree

1. Without Memorization



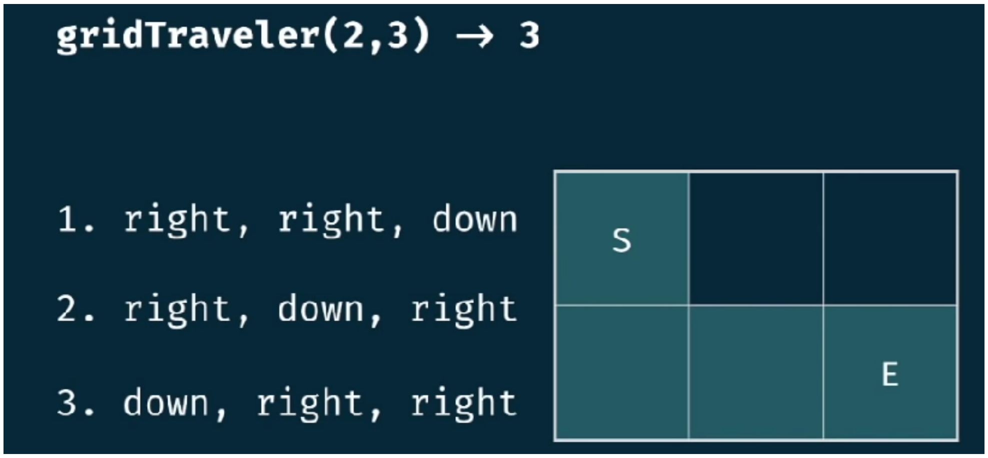
2. With memoization



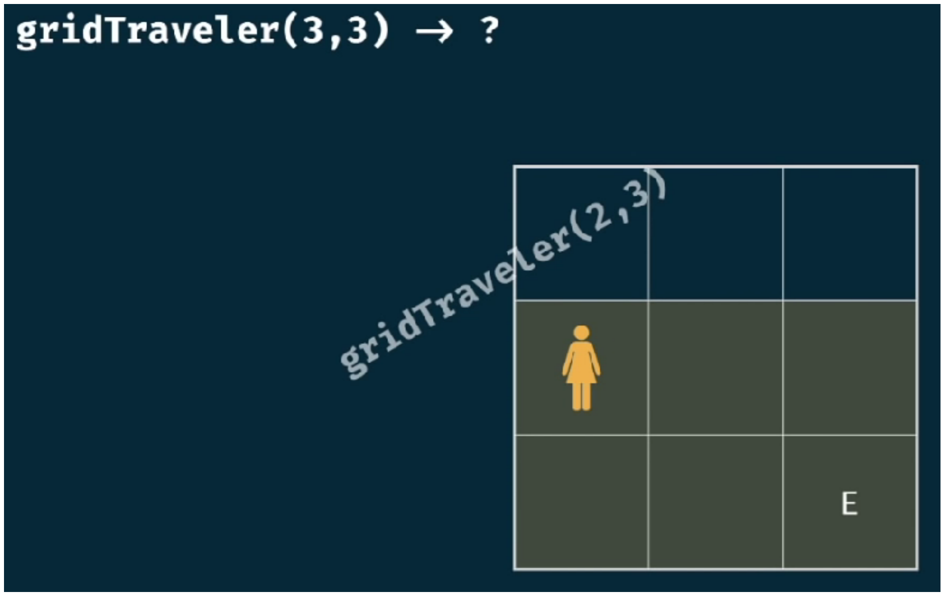
- Overall we have roughly `2n` nodes
- 

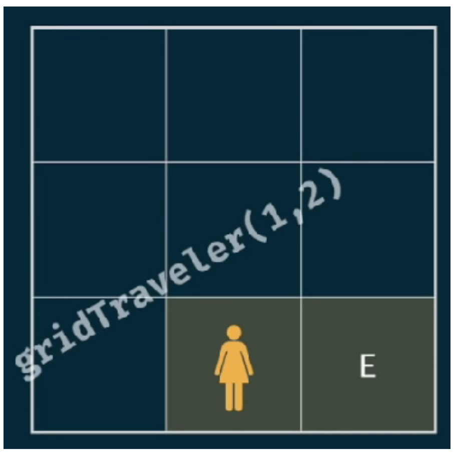
- `Fib(9)` has this structure
- New time complexity is `O(n)` and space complexity is `O(n)`

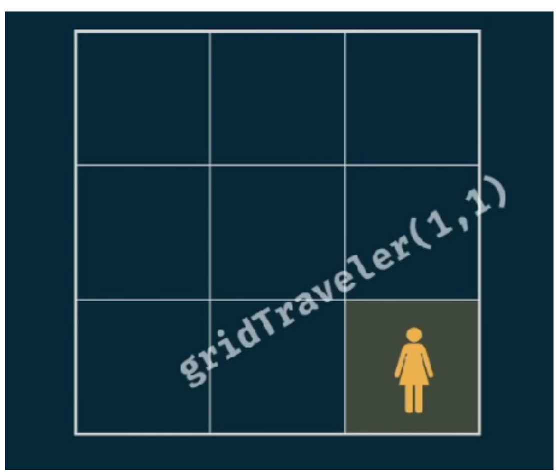
Grid Traveller

- You are a traveller on a 2D Grid. You begin in the top-left corner and your goal is to travel to the bottom right corner. You may only move down or right.
- In how many ways can you travel to the goal on a grid with dimensions m*n?
- 

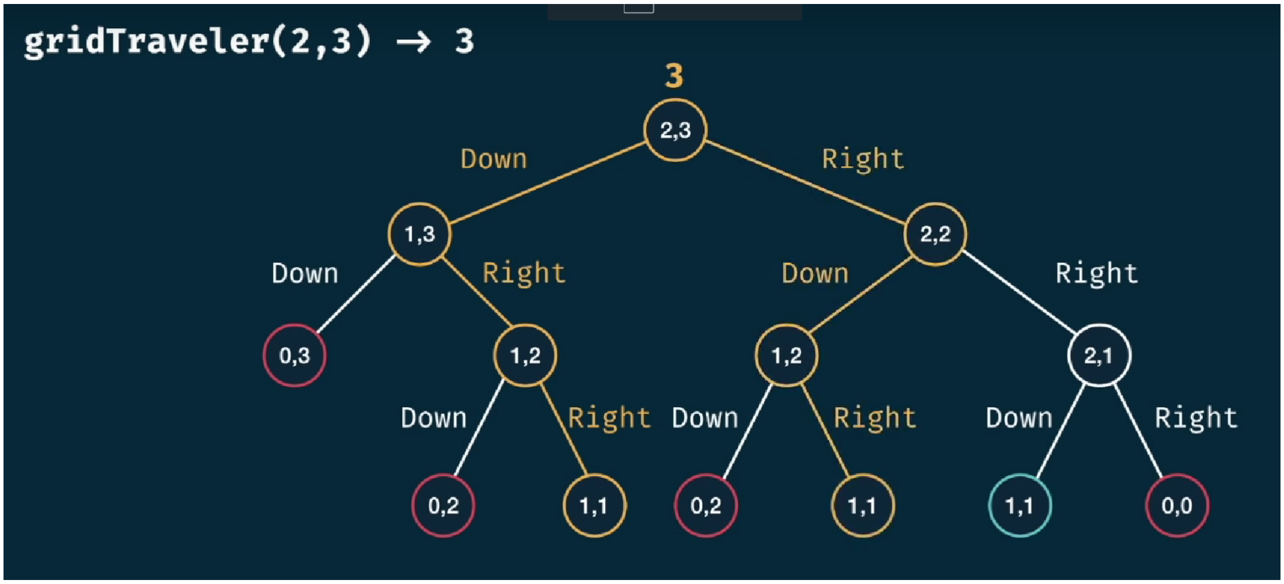
- Problems like this.
  - Always start with the smallest possible scenario
    - `gridTraverler(1,1) -> 1` since start is already at end
      - do nothing
    - `gridTraveler(0,1)` -> 0
      - invalid
    - `gridTraveler(3,3)`
      -
  - These can be thought of as base cases and can be used to construct the larger solution
- for a `gridTraveler(3,3)`
  - As me move down the problem reduces to `gridTraveler(2,3)`
- 

- 

- 

- 

- Any problem that looks like recursion always visualize a tree
- At every recursive call we are either reducing by 1 row or by 1 column
- From the tree structure we can already tell which tree nodes were formed by which combination of moves.
- which combination gives a positive answer and which moves lead us to dead end

- 

- The tree looks like a binary tree because at every point there are two possible ways to traverse the the grid either down or right
- This tree is now composed of two different factors
- `m` and `n`
- We must take in account the other parameter
-