Dynamic Programming Notes

1. Memoization

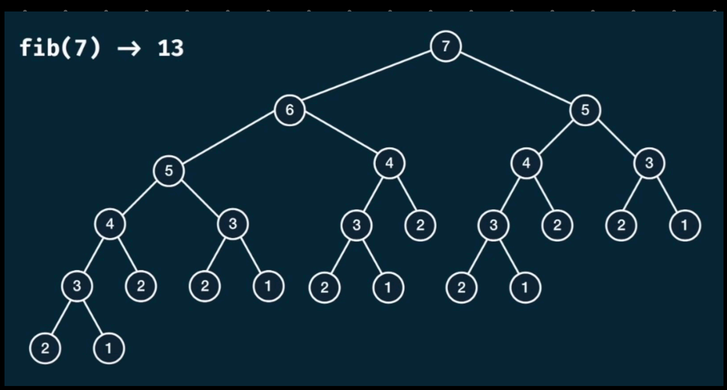
2. Tabulation

Fib(n) Sequence

Recursive approach:

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

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

- Solution we're familiar with:
  - slow
  - takes long to compute large fib( ) values

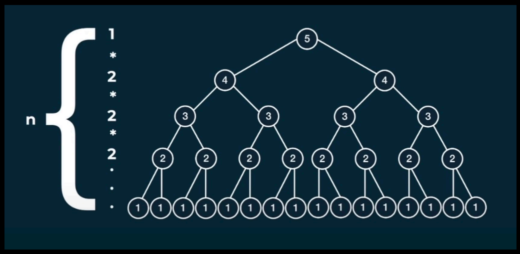


* Visualization of fib(7) recursion calls

Time Complexity?

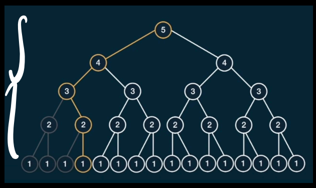
↳ # of total nodes (# of recursive calls) = $O(2^n)$ time complexity

↳ "multiply 2, n times over"



Space Complexity → # of recursive calls on stack

max stack depth n



Will at most have 5 calls on stack, old, returned calls are removed from stack

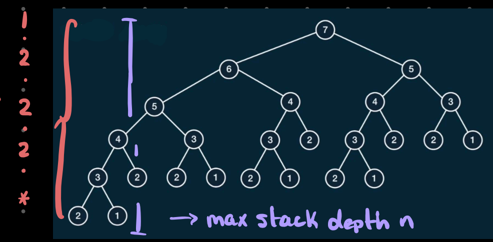
$O(n)$ space complexity

Back to fib( ) example:

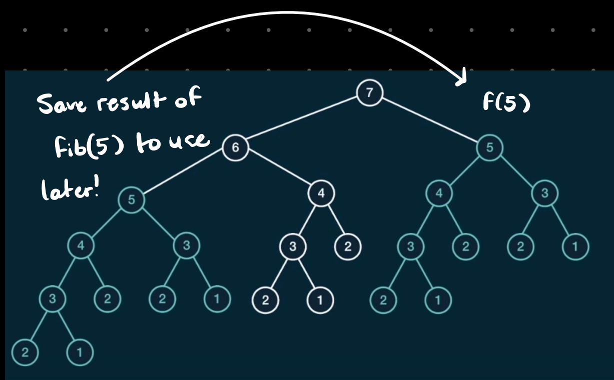
→ $O(2^n)$ time complexity

→ $O(n)$ space complexity

n times



Can we improve?



Here we notice some repetitions on some subtrees

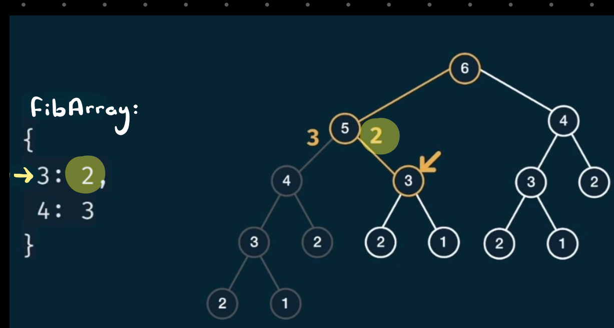
Overlapping Supproblems

↳ Dynamic Programming!

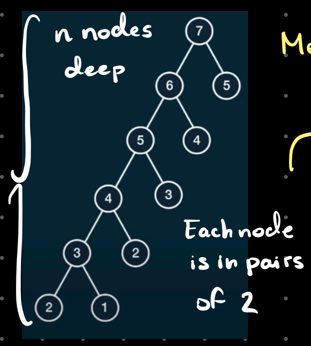
Memoization - storing duplicate subproblems

```
//memoized fibonacci
int fibMemo(int n, unordered_map<int, int> &memo){
    //check if n is in memo
    if(memo.find(n) != memo.end())
        return memo[n];

    if(n <= 2)
        return 1;

    //store result in memo
    memo[n] = fibMemo(n-1, memo) + fibMemo(n-2, memo);

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

Memo stores fib(n) computations so when it's later needed again, no need to re-calculate



* Less traversing through subtrees.



Memoized fib()

→ Time and Space Complexity $O(n)$

$O(2n) \rightarrow O(n)$

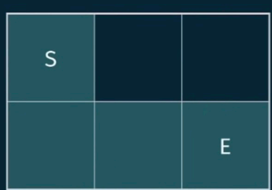
---

gridTraveler

Say you're traveling on a 2D grid. You begin in the top-left corner and your goal is to travel to the bottom right corner. May only move down or right.

In how many ways can you travel to the goal on a grid with dimensions $m * n$?

1. right, right, down
2. right, down, right
3. down, right, right



gridTraveler(2,3) = 3 ways to get to end of grid

Think about how we can possibly break down problem

gridTraveler(3,3)

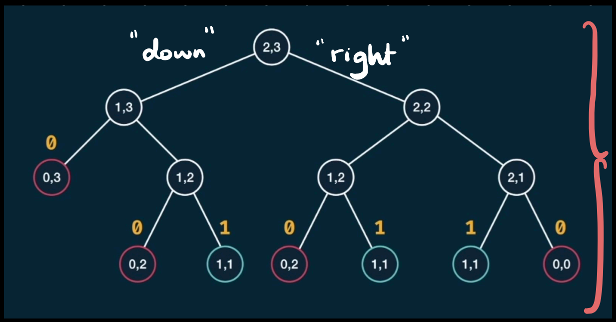
Notice that when we move, you're basically shrinking problem size

Reached a "base" case now

(1,1) or if 0 shows up in either (x,y)

gridTraveler(2,3)

Programmatic approach, tree-visualization



n+m height

Each node splits into 2 calls

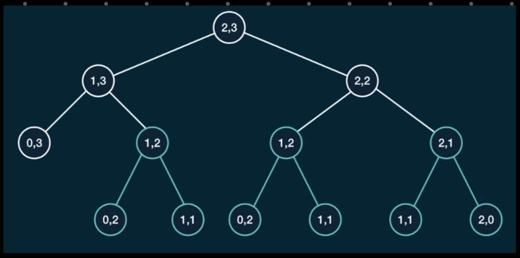
$O(2^{n+m})$ time complexity

$O(n+m)$ space complexity

Recursive Approach

```cpp
int gridTraveler(int x, int y)
{
    //base case
    if(x == 1 && y == 1)
        return 1;

    //invalid grid
    if(x == 0 || y == 0)
        return 0;

    return gridTraveler(x - 1, y) + gridTraveler(x, y - 1);
}
```

Memoization



Find patterns!

If you think about it, # of ways to travel (1,2) grid is the same as (2,1) grid

↳ Order of arguments doesn't matter

```cpp
int memoGridTraveler(int m, int n, unordered_map<int, int> &memo)
{
    //check if m,n is in memo
    int key = m * 1000 + n;
    if(memo.find(key) != memo.end())
        return memo[key];

    //base case
    if(m == 1 && n == 1)
        return 1;

    //invalid grid
    if(m == 0 || n == 0)
        return 0;

    //store result in memo
    memo[key] = memoGridTraveler(m - 1, n, memo) + memoGridTraveler(m, n - 1, memo);

    //return result
    return memo[key];
}
```

Time Complexity: $m*n$ possible combinations

$O(m*n)$ Improvement!

Space Complexity: $O(n+m)$

Tips so Far! → visualize problem as a tree to think of a solution

Memoization Recipe

1. Make it work (even if it's recursive and slow)

   - visualize problem as a tree
   - implement tree using recursion
   - test it

2. Make it efficient (dynamic programming solution)

   - add memo object (Key-value relationship)
   - add a base case to return memo values (if value in memo, return...)
   - Store return values into memo