

DYNAMIC PROGRAMMING

A Light Introduction

DON'T BE SCARED
(I AM)

MAKE SURE YOU'RE OK WITH
RECURSION

OBJECTIVES

- Define what dynamic programming is
- Explain what overlapping subproblems are
- Understand what optimal substructure is
- Solve more challenging problems using dynamic programming

WTF IS DYNAMIC PROGRAMMING

"A method for solving a complex problem by breaking it down into a collection of simpler subproblems, solving each of those subproblems just once, and storing their solutions."

WHERE DOES
THE NAME COME FROM?

IT ONLY WORKS ON
PROBLEMS WITH...

OPTIMAL SUBSTRUCTURE &

OVERLAPPING SUBPROBLEMS

EXCUSE
ME?!

OVERLAPPING SUBPROBLEMS

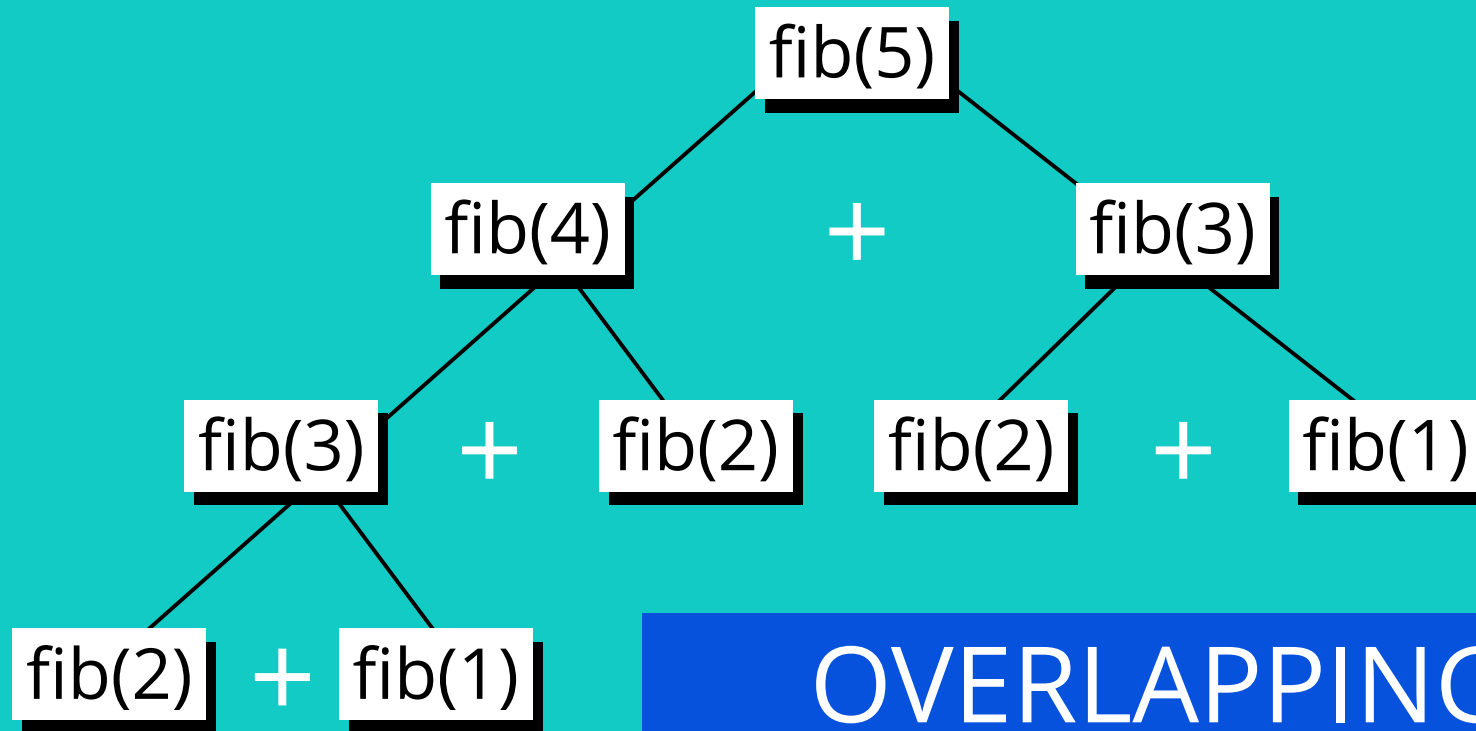
A problem is said to have **overlapping subproblems** if it can be broken down into subproblems which are reused several times

FIBONACCI SEQUENCE

"Every number after the first two is the sum of the two preceding ones"



FIBONNACI NUMBERS



OVERLAPPING
SUBPROBLEMS!

REMEMBER MERGESORT?

[10,24,73,76]

mergeSort([10,24,76,73])

[10,24]

merge

[73,76]

mergeSort([10,24])

mergeSort([76,73])

[10]

merge

[24]

[76]

merge

[73]

mergeSort([10])

mergeSort([24])

mergeSort([76])

mergeSort([73])

NO OVERLAPPING
SUBPROBLEMS!

A VERY SPECIAL CASE

mergeSort([10,24,10,24])

mergeSort([10,24])

mergeSort([10,24])

mergeSort([10])

mergeSort([24])

mergeSort([10])

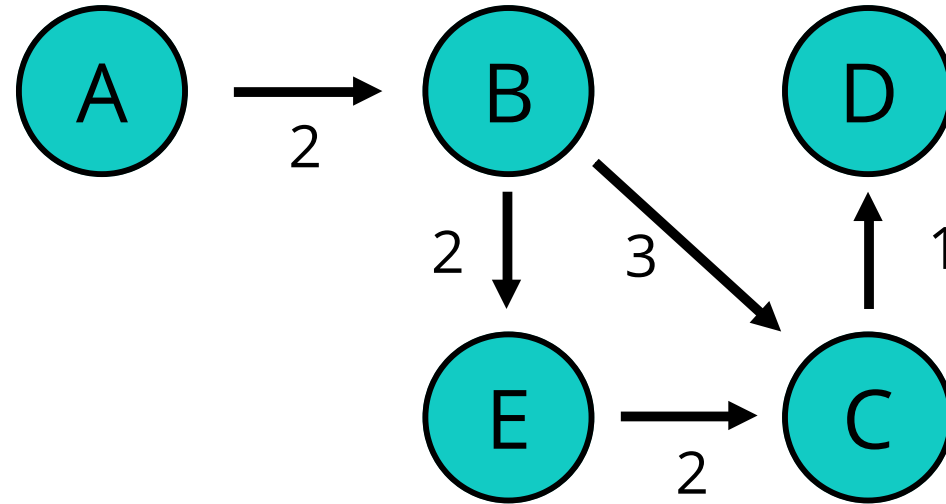
mergeSort([24])

YES OVERLAPPING
SUBPROBLEMS!

OPTIMAL SUBSTRUCTURE

A problem is said to have **optimal substructure** if an optimal solution can be constructed from optimal solutions of its subproblems

SHORTEST PATH



SHORTEST PATH FROM:

A to D

A -> B -> C -> D

A to C

A -> B -> C

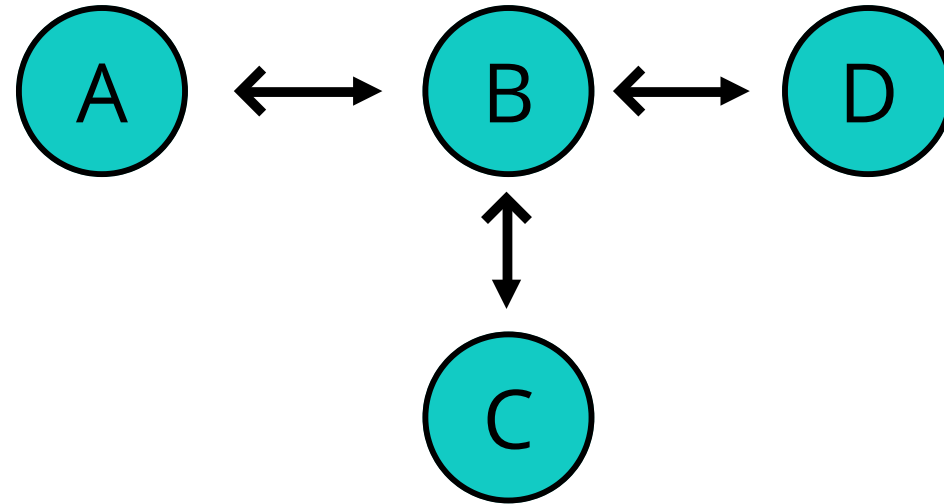
A to B

A -> B

OPTIMAL
SUBSTRUCTURE!

LONGEST SIMPLE PATH

(simple means no repeating)



LONGEST PATH FROM:

A to C

A -> B -> C

C to D

C -> B -> D

A to D

~~A -> B -> C -> B -> D~~

A -> B -> D

NO OPTIMAL
SUBSTRUCTURE!

Cheapest flight from SFO to FAI?



Delta flight card for SFO to FAI via SEA. The card displays the Delta logo, flight time of 12:12 pm to 6:05 pm, 1 stop at SEA, and a total duration of 6h 53m. The price is \$185 for Delta Basic Economy. A 'View Deal' button is present. Additional pricing for Main cabin (\$215) and Comfort+ (\$278) is shown.

Flight Details	Price
Delta Basic Economy	\$185
Main cabin	\$215
Comfort +	\$278

Cheapest flight from SFO to SEA?




Alaska Airlines flight card for SFO to SEA via PDX. The card displays the Alaska Airlines logo, flight time of 6:00 am to 9:45 am, 1 stop at PDX, and a total duration of 3h 45m. The price is \$109. A 'View Deal' button is present. A note at the bottom states 'Operated by Horizon Air AS Alaskahorizon'.

Flight Details	Price
Alaska Airlines	\$109

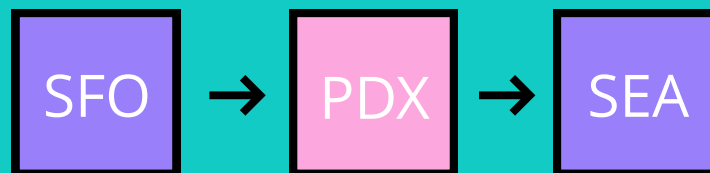
NO OPTIMAL SUBSTRUCTURE!


Cheapest flight from SFO to FAI?



	12:12 pm — 6:05 pm Delta	1 stop SEA	6h 53m SFO - FAI	<div><div>1</div><div>0</div></div> <div>\$185 Delta Basic Economy</div> <div>View Deal</div> <div>Main cabin \$215 Comfort + \$278</div>
---	------------------------------------	----------------------	----------------------------	--

Cheapest flight from SFO to SEA?



	6:00 am — 9:45 am Alaska Airlines	1 stop PDX	3h 45m SFO - SEA	<div><div>1</div><div>0</div></div> <div>\$109 Alaska Airlines</div> <div>View Deal</div> <div>Operated by Horizon Air AS Alaskahorizon</div>
---	---	----------------------	----------------------------	--

Let's return to our pal...

THE FIBONACCI SEQUENCE

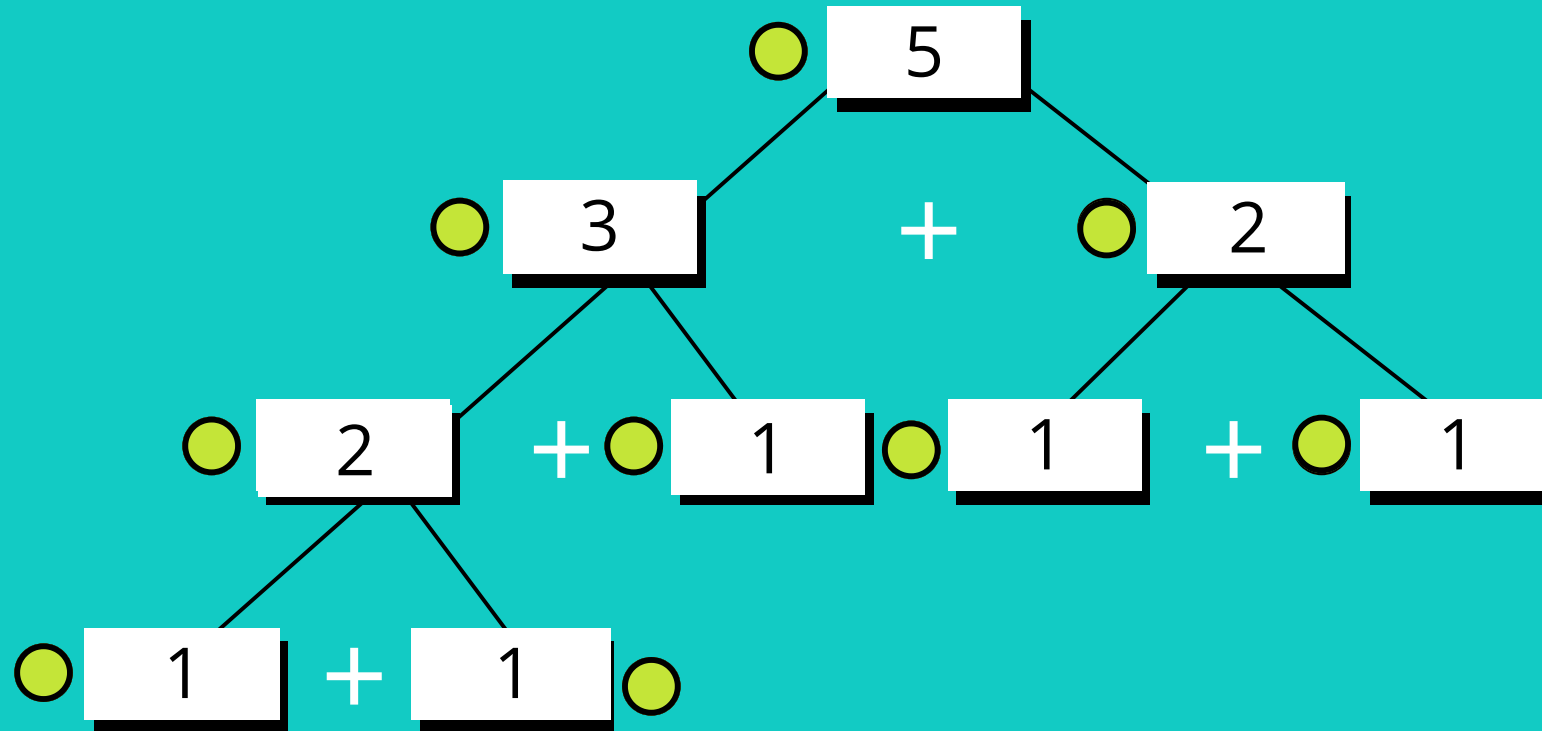
LET'S WRITE IT!

- $\text{Fib}(n) = \text{Fib}(n-1) + \text{Fib}(n-2)$
- $\text{Fib}(2)$ is 1
- $\text{Fib}(1)$ is 1

RECURSIVE SOLUTION

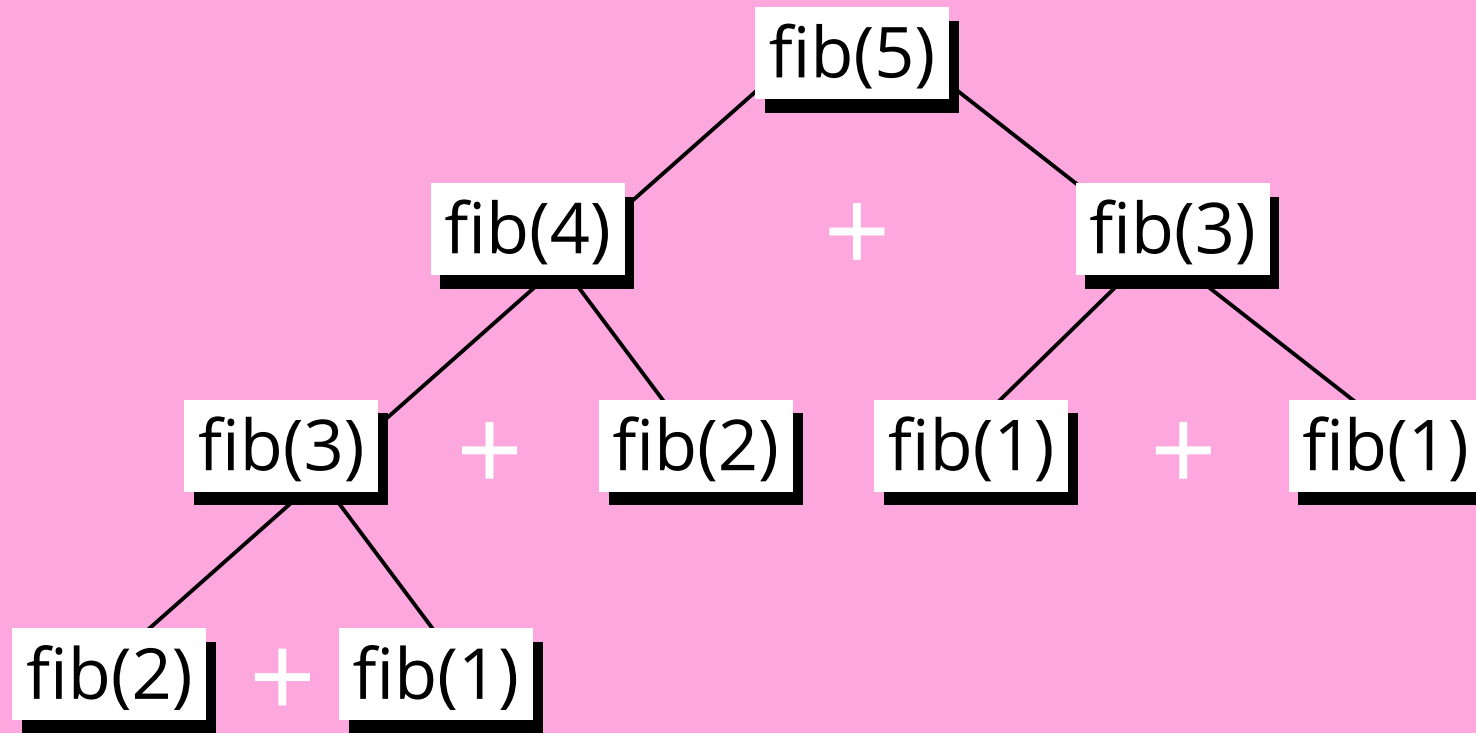
```
function fib(n) {  
    if (n <= 2) return 1;  
    return fib(n-1) + fib(n-2);  
}
```

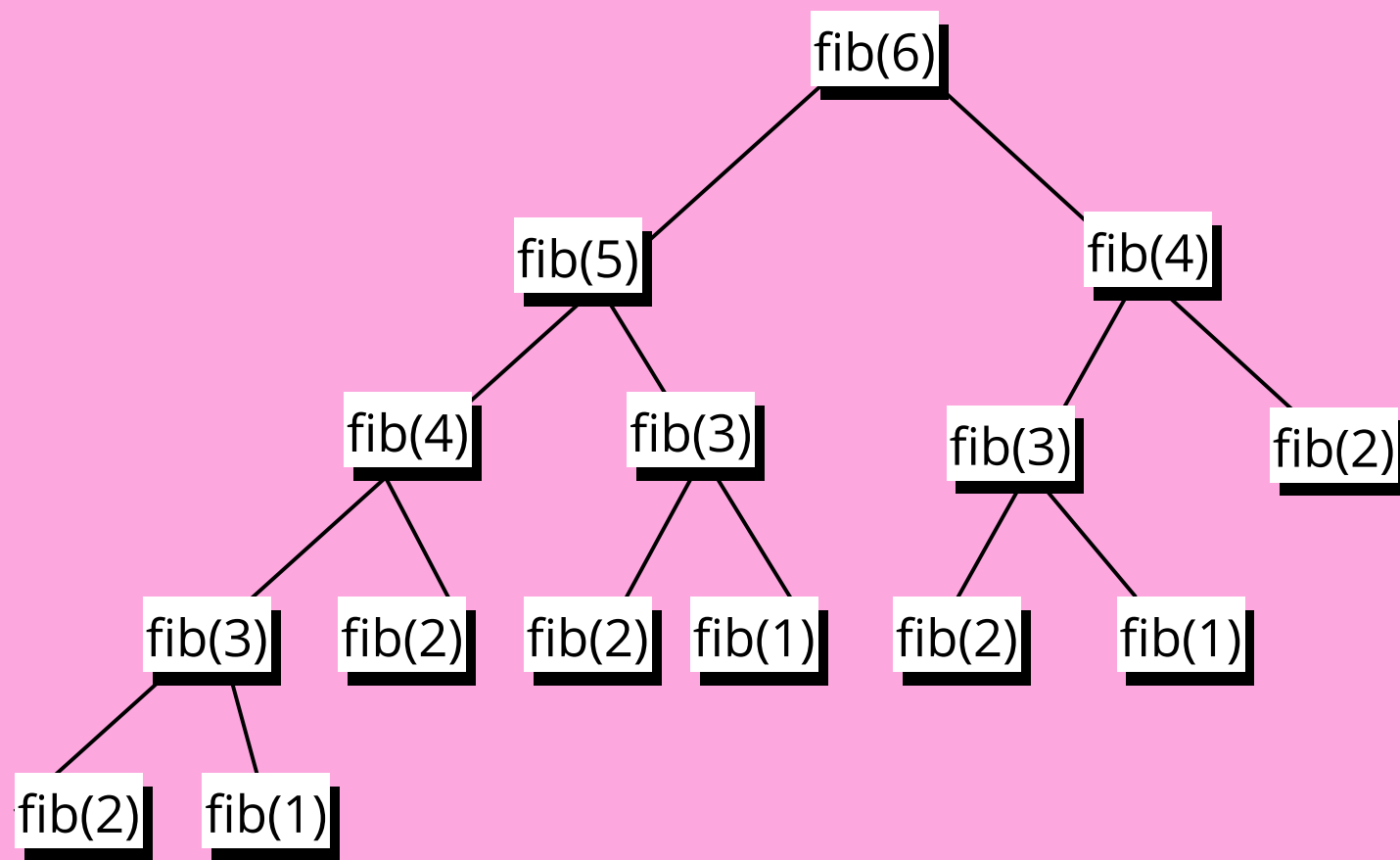
PLAIN OLD RECURSION



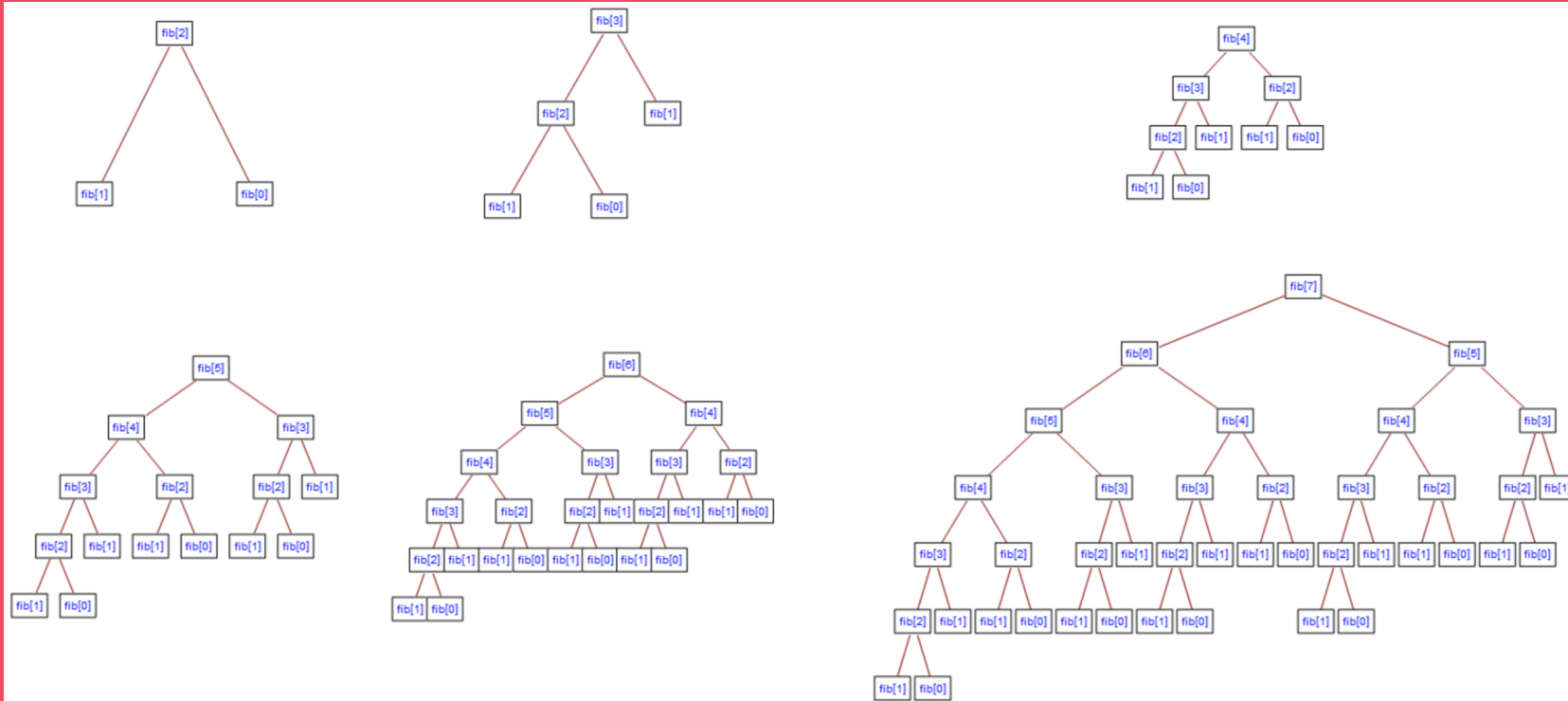
LET'S CHAT ABOUT

Big O





Look at that growth!



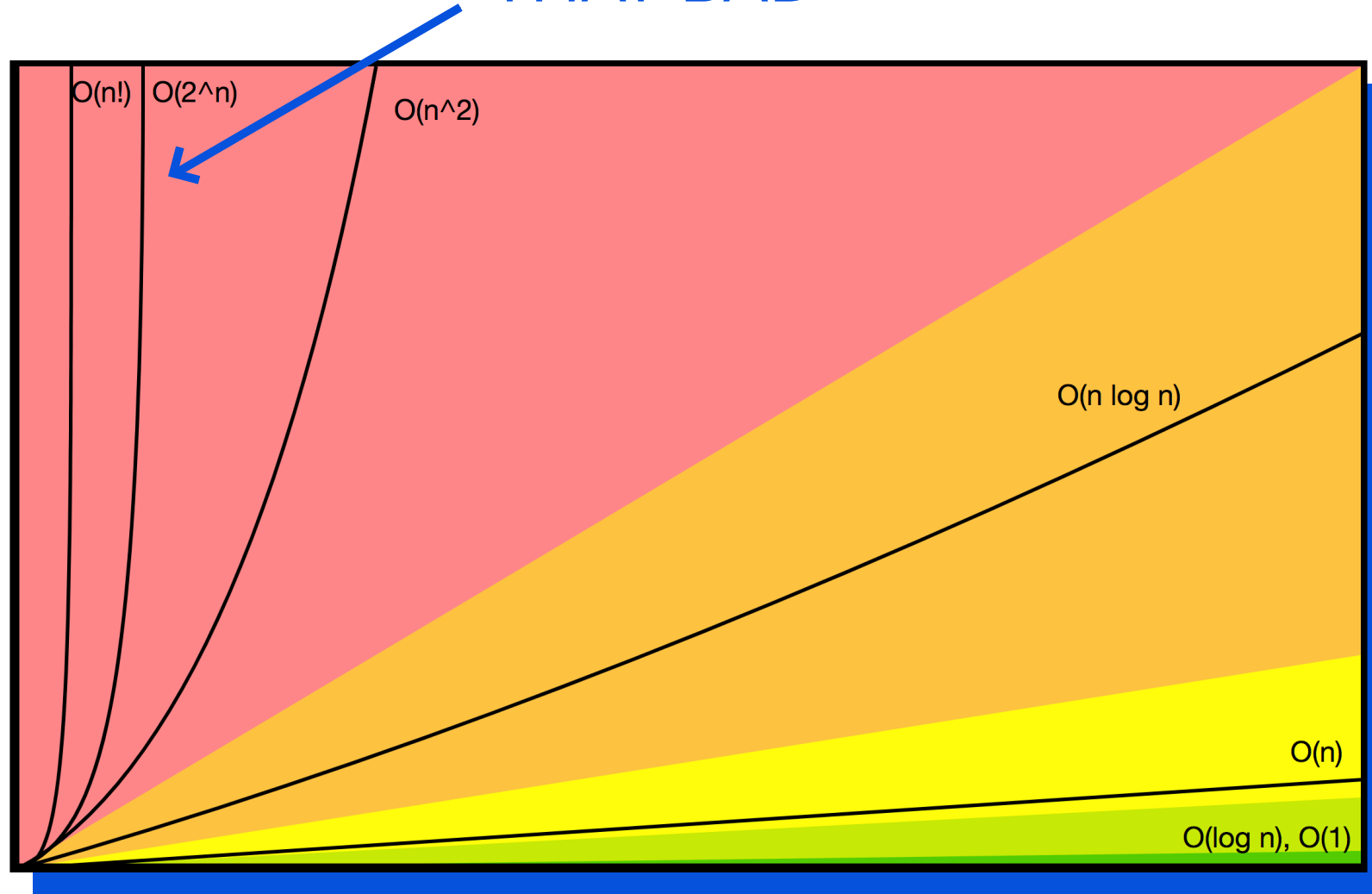
<https://i.stack.imgur.com/kgXDS.png>

HOW BAD?

$O(2^N)$



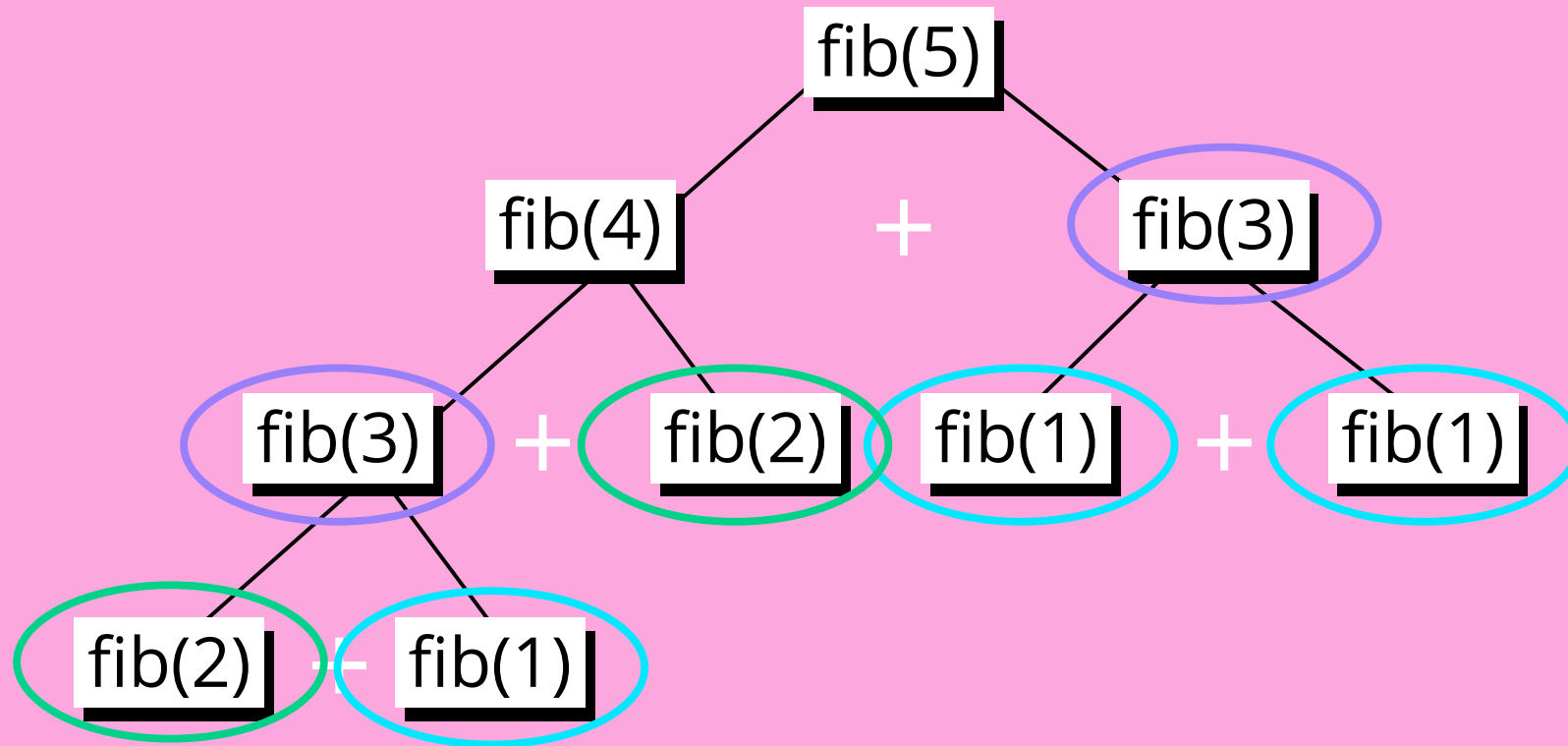
THAT BAD



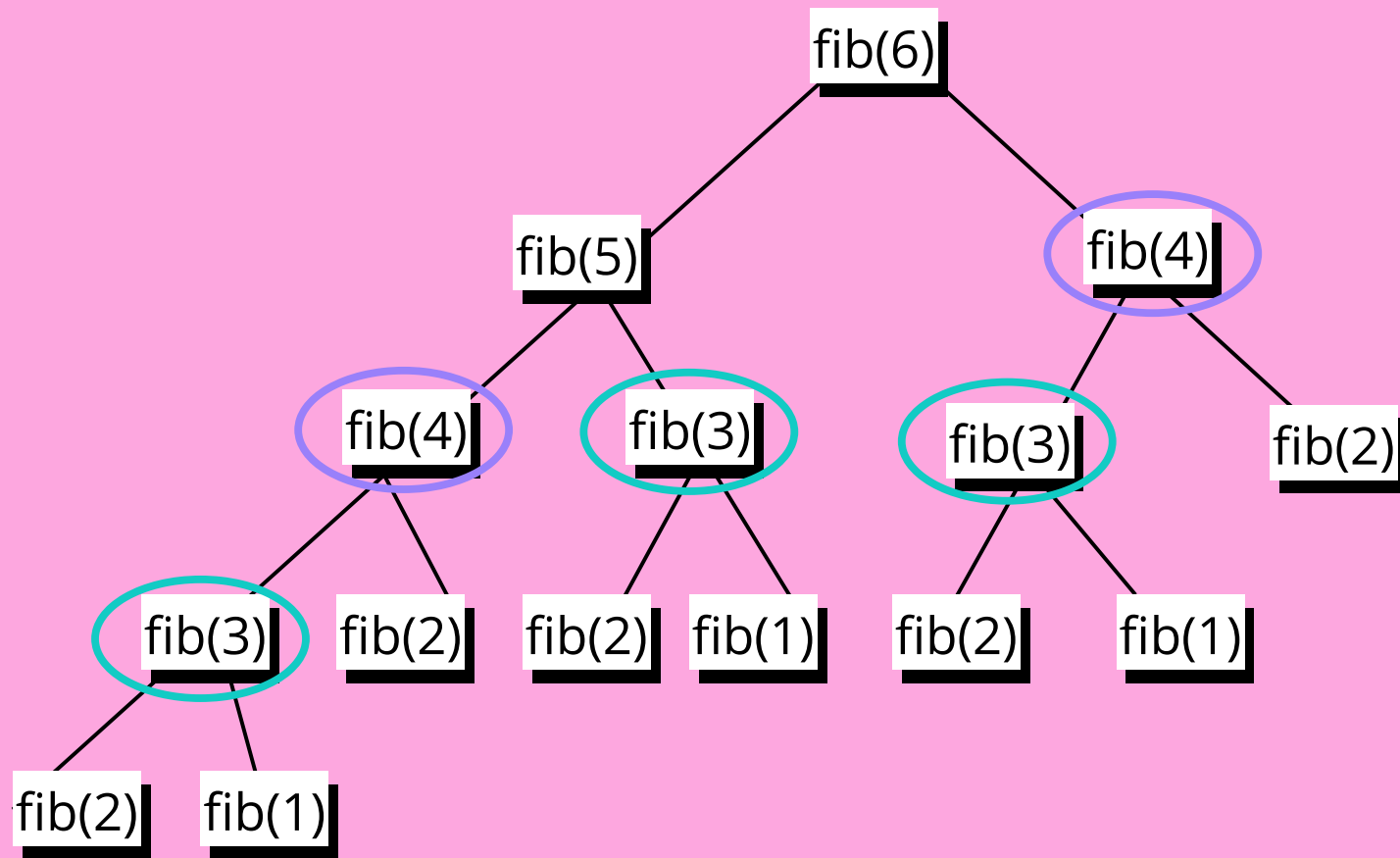
<https://i.stack.imgur.com/kgXDS.png>

WHAT CAN WE
IMPROVE??

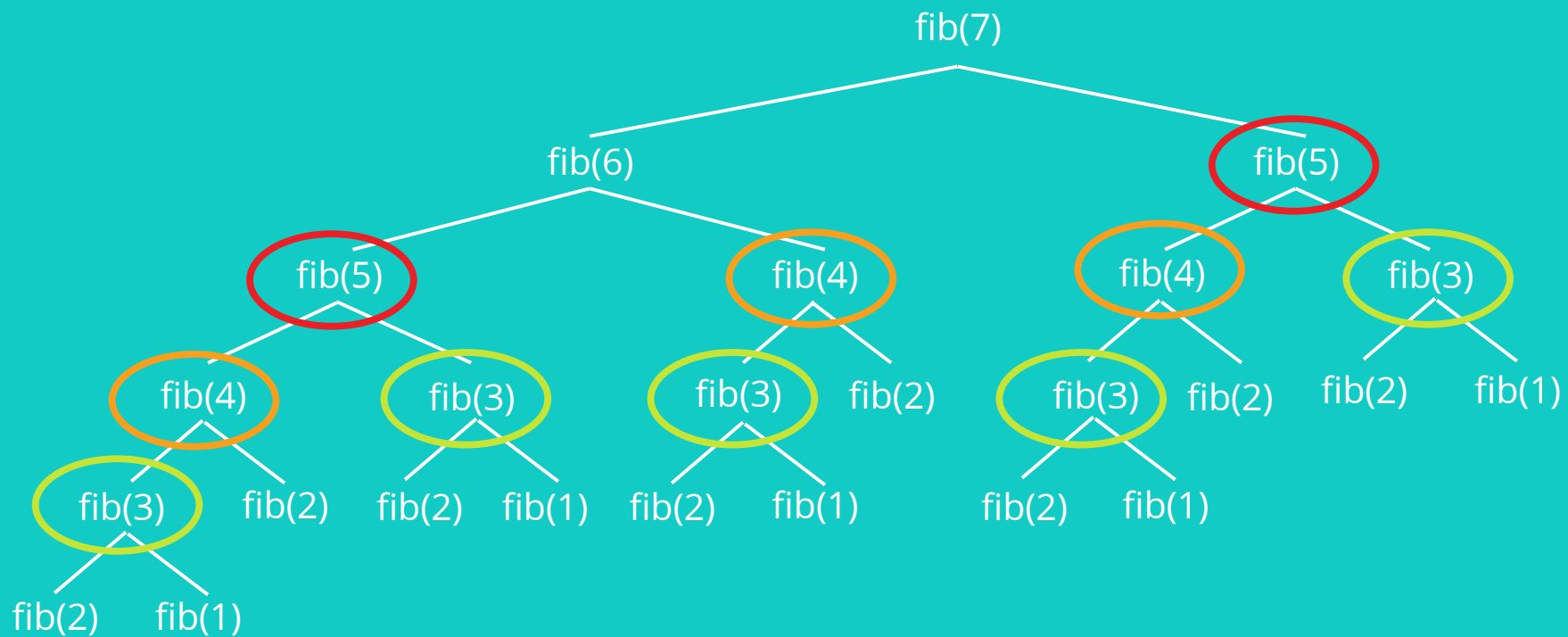
WE'RE REPEATING THINGS!



THIS IS NOT GOOD.



REALLY NOT GOOD



WHAT IF WE COULD
"REMEMBER" OLD VALUES?

ENTER DYNAMIC PROGRAMMING

"Using past knowledge to make
solving a future problem easier"

ENTER (AGAIN) DYNAMIC PROGRAMMING

"A method for solving a complex problem by breaking it down into a collection of simpler subproblems, solving each of those subproblems just once, and storing their solutions."

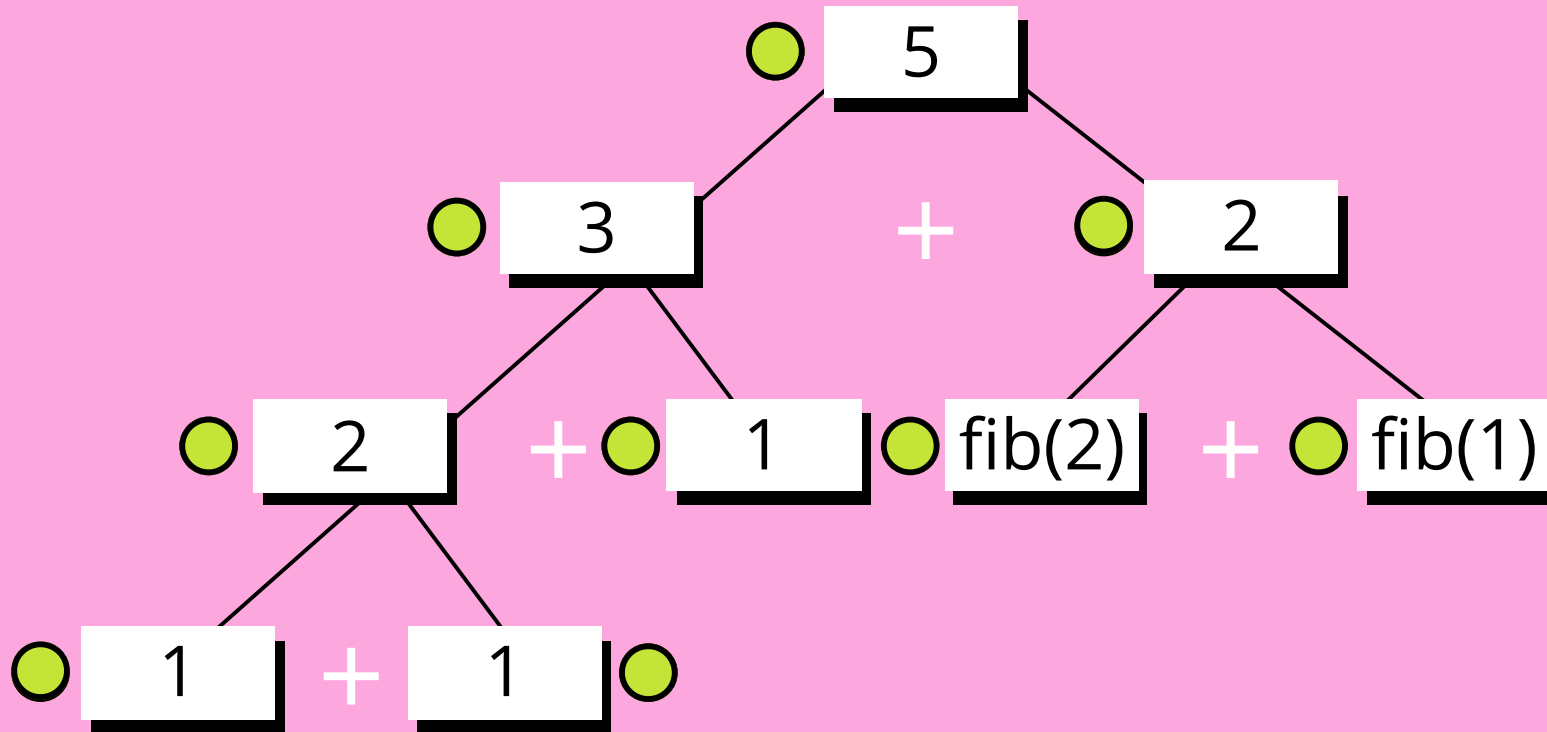
MEMOIZATION

Storing the results of expensive function calls and returning the cached result when the same inputs occur again

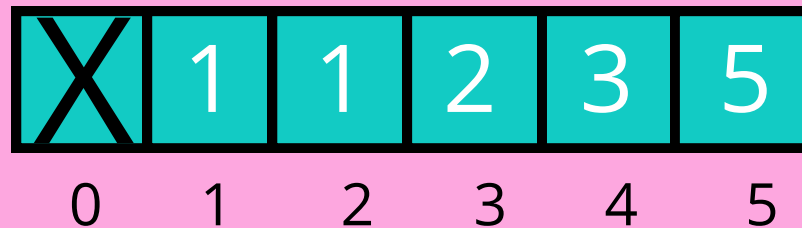
A MEMO-IZED SOLUTION

```
function fib(n, memo=[]){  
  if(memo[n] !== undefined) return memo[n]  
  if(n <= 2) return 1;  
  var res = fib(n-1, memo) + fib(n-2, memo);  
  memo[n] = res;  
  return res;  
}
```

RECURSION + MEMOIZATION



What we've already calculated

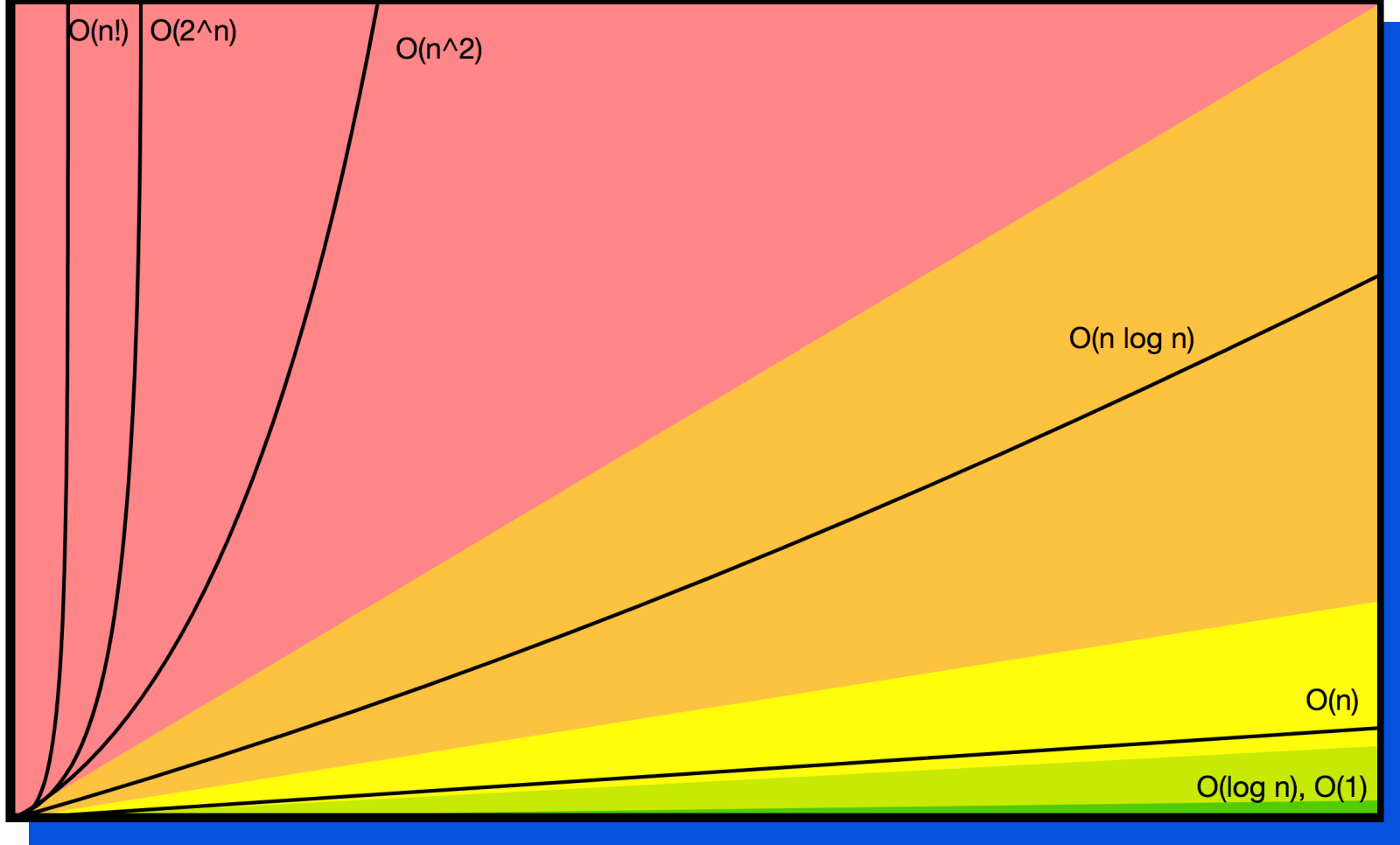


LET'S CHAT ABOUT

Big O

MUCH BETTER

$O(N)$



ONCE AGAIN DYNAMIC PROGRAMMING

"A method for solving a complex problem by breaking it down into a collection of simpler subproblems, solving each of those subproblems just once, and storing their solutions."

WE'VE BEEN WORKING

TOP-DOWN

BUT THERE IS ANOTHER WAY!

BOTTOM-UP

TABULATION

Storing the result of a previous result in a "table" (usually an array)

Usually done using **iteration**

Better **space complexity** can be achieved using tabulation

TABULATED VERSION

```
function fib(n) {  
    if (n <= 2) return 1;  
    var fibNums = [0,1,1];  
    for (var i = 3; i <= n; i++) {  
        fibNums[i] = fibNums[i-1] + fibNums[i-2];  
    }  
    return fibNums[n];  
}
```

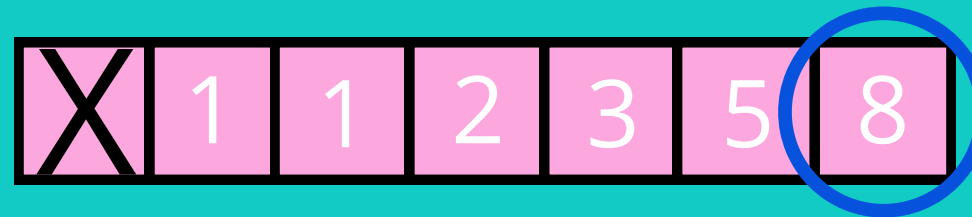

TABULATED FIB(6)

$\text{arr}[3] = \text{arr}[2] + \text{arr}[1]$

$\text{arr}[4] = \text{arr}[3] + \text{arr}[2]$

$\text{arr}[5] = \text{arr}[4] + \text{arr}[3]$

$\text{arr}[6] = \text{arr}[5] + \text{arr}[4]$

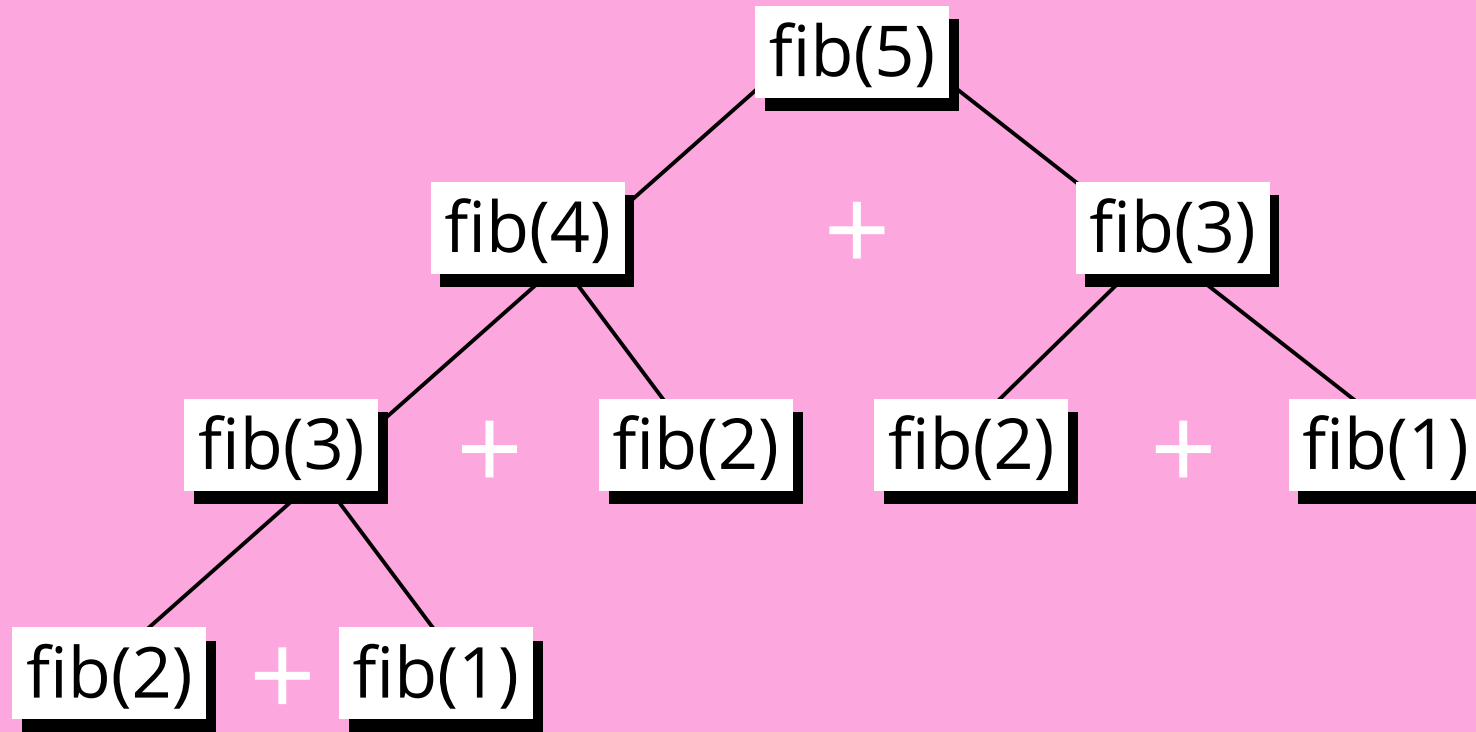


0 1 2 3 4 5 6

ALL DONE!

THE REST IS
UNDER
CONSTRUCTION!

RECURSION + MEMOIZATION



What we've already calculated

X	1	1	2	3	5
0	1	2	3	4	5

TOP DOWN VS. BOTTOM UP

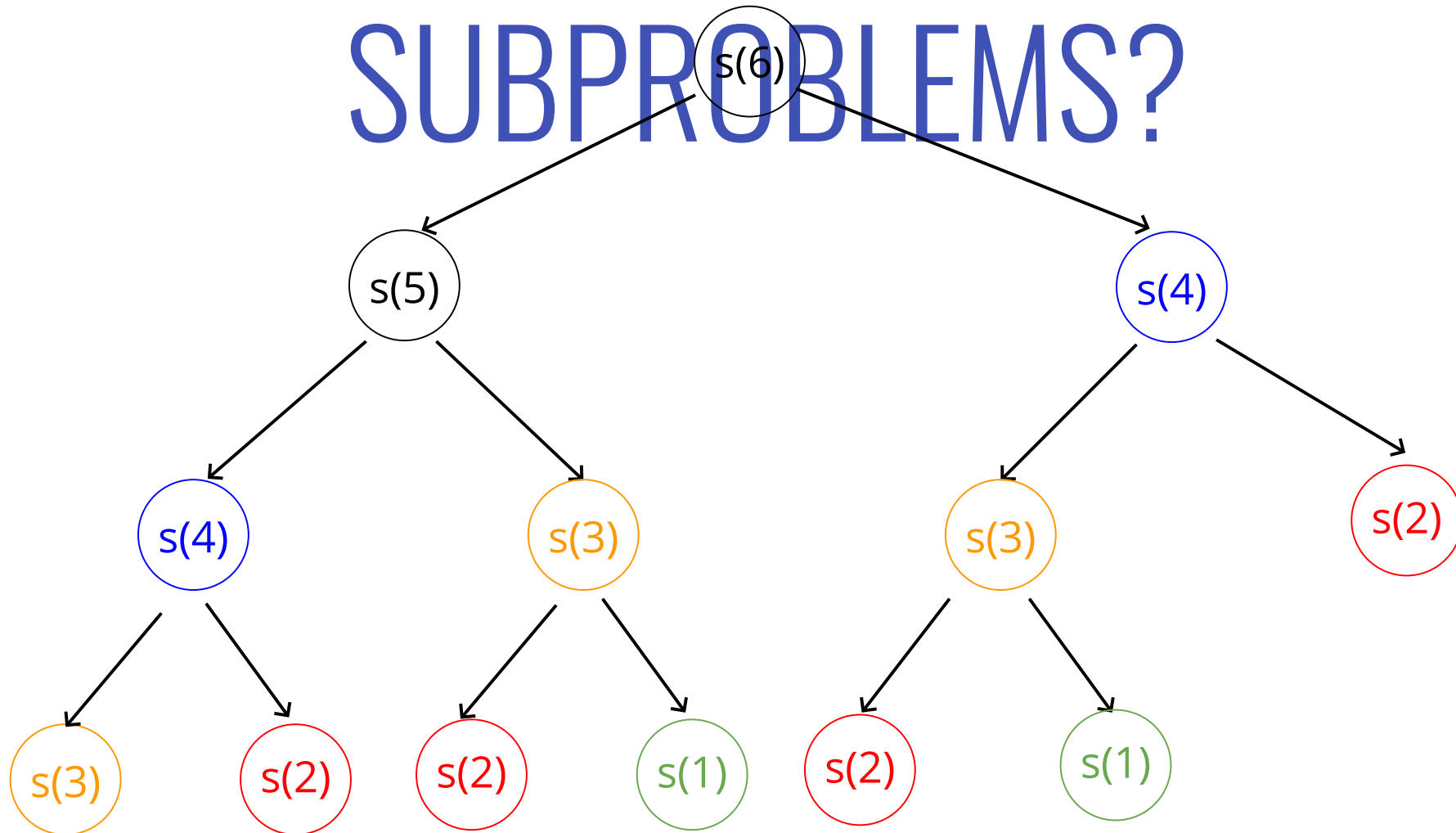
AN EXAMPLE:

Write a function called **stairs** which accepts **n** number of stairs. Imagine that a person is standing at the bottom of the stairs and wants to reach the top and the person can climb either 1 stair or 2 stairs at a time. Your function should return the number of ways the person can reach the top by only climbing 1 or 2 stairs at a time.

START WITH SOMETHING SMALL

Stairs(1)	Stairs(2)	Stairs(3)	Stairs(4)	Stairs(5)
<ul style="list-style-type: none">• 1	<ul style="list-style-type: none">• 1,1• 2	<ul style="list-style-type: none">• 1,1,1• 1,2• 2,1	<ul style="list-style-type: none">• 1,1,1,1• 2,1,1• 1,2,1• 1,1,2• 2,2	<ul style="list-style-type: none">• 1,1,1,1,1• 2,1,1,1• 1,2,1,1• 1,1,2,1• 2,2,1• 1,1,1,2• 1,2,2• 2,1,2
1	2	3	5	8

WHAT ARE THE SUBPROBLEMS?



WHAT IS THE SUBSTRUCTURE?

```
stairs(n) = stairs(n - 1) + stairs(n - 2);
```

This is usually the hardest part of dynamic programming and takes a lot of practice!

HOW DO WE SOLVE IT?

```
function stairs(n) {  
  if (n <= 0) return 0;  
  if (n <= 2) return n;  
  return stairs(n - 1) + stairs(n - 2);  
}
```

Brute force

Time Complexity **$O(2^N)$**

MEMOIZATION

Storing the result of an
expensive function

Usually done using **recursion**

MEMOIZATION SOLUTION

```
function stairs(n, memo=[]) {  
  if (n <= 0) return 0;  
  if (n <= 2) return n;  
  if (memo[n] > 0) return memo[n];  
  memo[n] = stairs(n - 1, memo) + stairs(n - 2, memo);  
  return memo[n];  
}
```

Time Complexity - $O(N)$

TABULATION

Storing the result of a previous result in a "table" (usually an array)

Usually done using **iteration**

Better **space complexity** can be achieved using tabulation

TABULATION SOLUTION

```
function stairs(n) {  
  if(n < 3) return n;  
  let store = [1,1];  
  for(let i = 2; i <= n; i++) {  
    let total = store[1] + store[0]  
    store[0] = store[1]  
    store[1] = total  
  }  
  return store[1];  
}
```

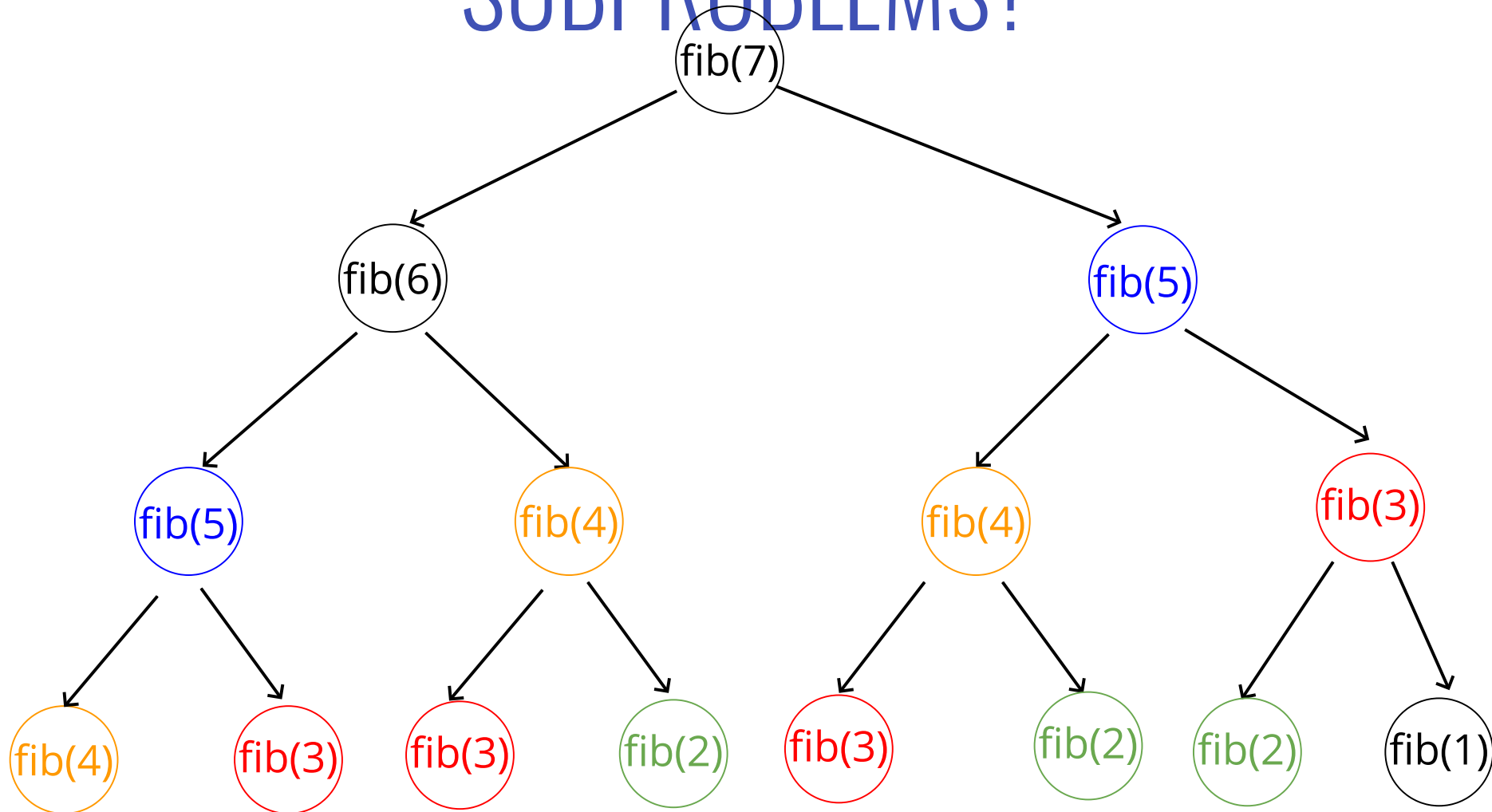
Time Complexity - $O(N)$

Space Complexity - $O(1)$

ANOTHER EXAMPLE

```
function fib(n) {  
    if (n <= 0) return 0;  
    if (n <= 2) return 1;  
  
    return fib(n - 1) + fib(n - 2);  
}
```

WHAT ARE THE OVERLAPPING SUBPROBLEMS?



WHAT IS THE SUBSTRUCTURE?

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


MEMOIZATION

```
function fib(n, savedFib={}) {  
  // base case  
  if (n <= 0) { return 0; }  
  if (n <= 2) { return 1; }  
  
  // memoize  
  if (savedFib[n - 1] === undefined) {  
    savedFib[n - 1] = fib(n - 1, savedFib);  
  }  
  
  // memoize  
  if (savedFib[n - 2] === undefined) {  
    savedFib[n - 2] = fib(n - 2, savedFib);  
  }  
  
  return savedFib[n - 1] + savedFib[n - 2];  
}
```

TABULATION

```
function fib(n){  
  let arr = [0,1]  
  // calculating the fibonacci and storing the values  
  for(let i = 2; i <= n; i++){  
    arr[i] = arr[i-1] + arr[i-2]  
  }  
  return arr[n]  
}
```

USING LISTS AND MATRICES TO BREAK DOWN PROBLEMS

AN EXAMPLE:

Write a function called **coinChange** which accepts two parameters: an array of denominations and a value. The function should return the number of ways you can obtain the value from the given collection of denominations. You can think of this as figuring out the number of ways to make change for a given value from a supply of coins.

BUILDING A LIST

Amount - 10 / Denominations - [1,2,5]

Start with 1

If amount > coin:

```
combinations[amount] += combinations[amount-coin]
```

[illegible]

MOVE TO 2

Amount - 10 / Denominations - [1,2,5]

Current Coin - **2**

If amount > coin:

combinations[amount] += combinations[amount-coin]

0	1	2	3	4	5	6	7	8	9	10
1	1	2	2	3	3	4	4	5	5	6

MOVE TO 5

Amount - 10 / Denominations - [1,2,5]

Current Coin - **5**

If amount > coin:

combinations[amount] += combinations[amount-coin]

0	1	2	3	4	5	6	7	8	9	10
1	1	2	2	3	4	5	6	7	8	10

WHERE IS THIS ACTUALLY USED?

- Artificial Intelligence
- Speech Recognition
- Caching
- Image Processing
- Shortest Path Algorithms
- Much, much more!

YOUR

TURN

GREEDY ALGORITHMS

A **greedy algorithm** is an algorithmic paradigm that follows the problem solving heuristic of making the locally optimal choice at each stage with the hope of finding a global optimum.

WHAT?

An algorithm that makes the best guess about what the right answer is and tries to solve it that way as quickly as possible!

WHERE ARE THEY USED?

You've seen one already!

Remember how Dijkstra's Algorithm works!

AN EXAMPLE:

The coin change problem - again!

A greedy algorithm takes the highest denomination and works it's way down

PSEUDOCODE FOR COIN CHANGE

- Start with the largest denomination
- Once the total can not use the largest
- Move to the 2nd largest
- Work your way down until there is no more change

DO THEY WORK?

Sometimes! Not always!

If we wanted the **least** amount of coins, a dynamic programming solution would be more efficient

YOUR

TURN

BACKTRACKING

"Backtracking is a general algorithm for finding all (or some) solutions to notably **constraint satisfaction problems**

It incrementally builds candidates to the solutions, and abandons a candidate ("backtracks") as soon as it determines that the candidate cannot possibly be completed to a valid solution"

WHAT IS IT?

Going through a solution and retracing steps backward if the solution is not valid.

HOW DOES IT WORK?

Visualizer for N queens

WHERE IS IT USED?

Puzzle Solving - Sudoku

N Queens / Rooks

RECAP

- Dynamic Programming is the idea of breaking down a problem into smaller subproblems - it's **hard**
- Optimal substructure is required to use dynamic program and involves figuring out the correct expression to consistently solve subproblems
- Overlapping subproblems is the second essential part of dynamic programming
- Greedy Algorithms are a more aggressive and not always efficient way of solving algorithms
- Backtracking is quite useful when solving for restrictive conditions with unknown possibilities