More Complexities

Lecture 8, Feb 12

601.**226** Data Structures- Spring 2020

Algorithmic efficiencies

Time Complexity

- how many steps will an algorithm take based on the input size?

Space Complexity

- how much (extra) memory will the algorithm use based on the input size?

We mostly focus on time complexities

- But we need to pay attention to space as well.
- Careful: recursive methods can be deceptively poor in space (and time).

Recursion

A quick review!

Fibonacci Sequence

$$1,\ 1,\ 2,\ 3,\ 5,\ 8,\ 13,\ 21,\ 34,\ 55,\ \ldots$$

- $f(n)$ returns the n^{th} term in the Fibonacci sequence.

$$f(n) = \begin{cases} 1 & n = 1 \\ 1 & n = 2 \\ f(n-1) + f(n-2) & n \geq 3 \end{cases}$$

Fibonacci Recursive – Version 1

- Obvious solution, but worst efficiencies!
- Cost: time $\Theta(2^N)$, extra space $\Theta(N)$

Get rid of this and ask for time via clicker question (this is the first impression guess, which we will not reveal the answer to until the second guess)

```
1 public static int fib(int n) {
2     if (n>2) {
3         return fib(n-2) + fib(n-1);
4     }
5     else return 1;
6 }
```

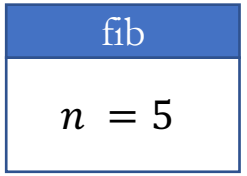
CLICKER QUESTION #1

```
1  public static int fib(int n) {
2      if (n>2) {
3          return fib(n-2) + fib(n-1);
4      }
5      else return 1;
6  }
```

- Recursion Tree

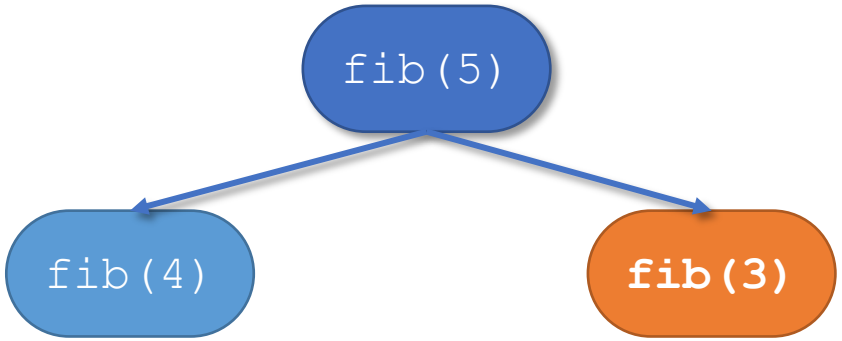
Stack

fib(5)

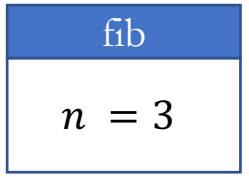


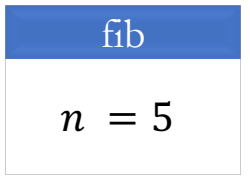
```
1  public static int fib(int n) {
2      if (n>2) {
3          return fib(n-2) + fib(n-1);
4      }
5      else return 1;
6  }
```

- Recursion Tree



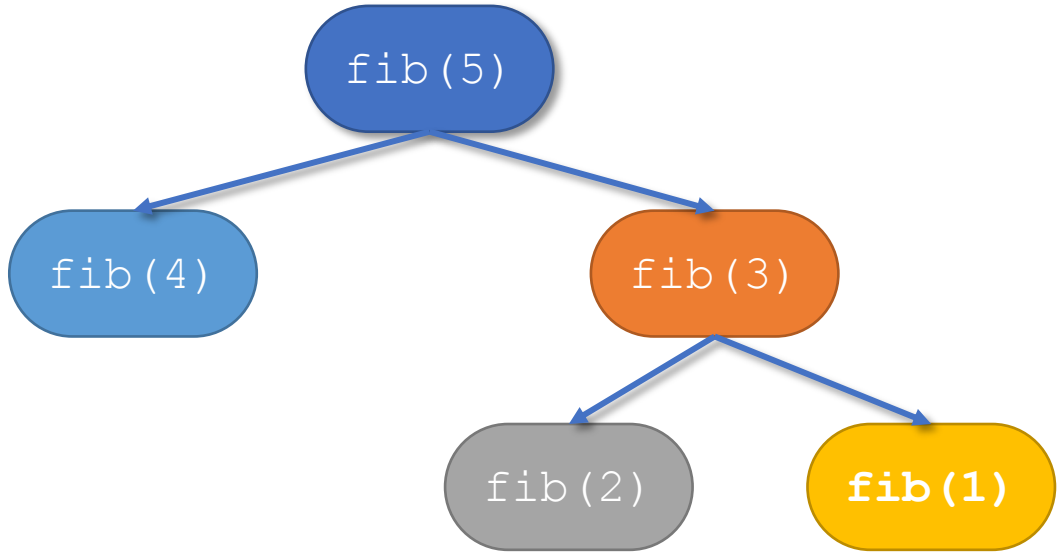
Stack grows



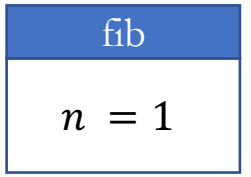


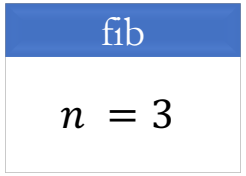
```
1  public static int fib(int n) {
2      if (n>2) {
3          return fib(n-2) + fib(n-1);
4      }
5      else return 1;
6  }
```

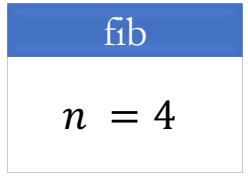
- Recursion Tree



Stack grows

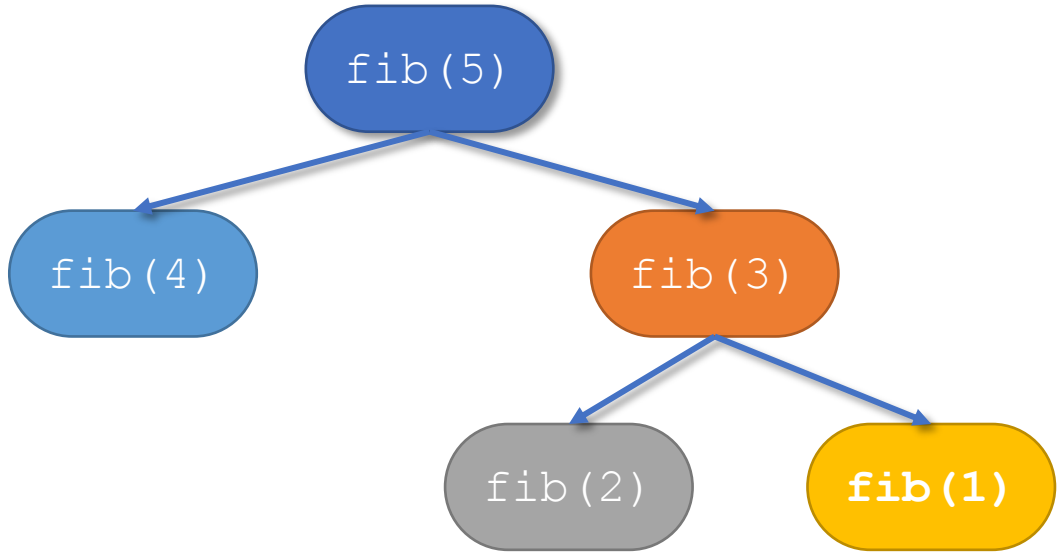






```
1  public static int fib(int n) {
2      if (n>2) {
3          return fib(n-2) + fib(n-1);
4      }
5      else return 1;
6  }
```

- Recursion Tree



Stack

fib
$n = 1$
return 1

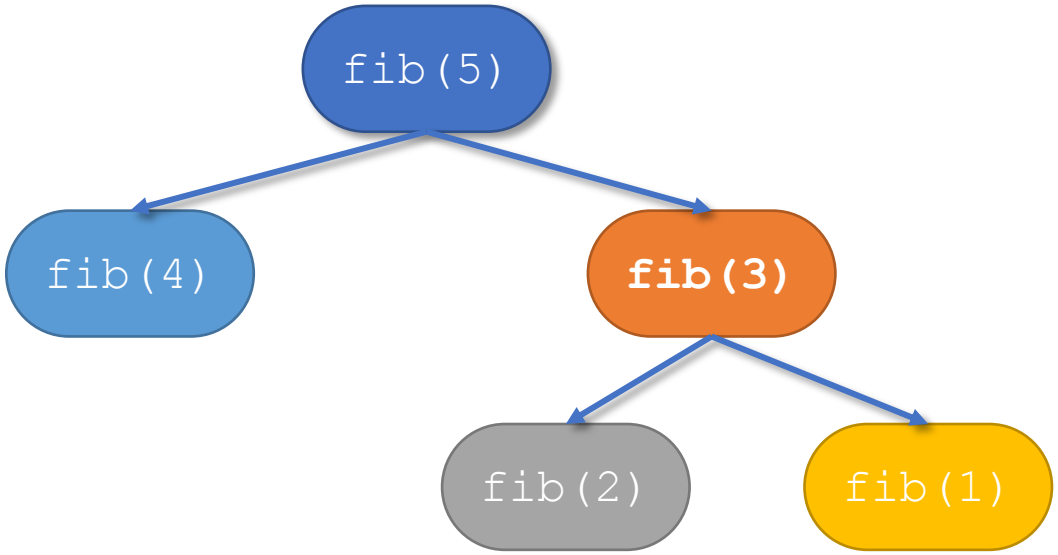
fib
$n = 3$

fib
$n = 5$

```
1  public static int fib(int n) {
2      if (n>2) {
3          return fib(n-2) + fib(n-1);
4      }
5      else return 1;
6  }
```

- Recursion Tree

Stack unwinds

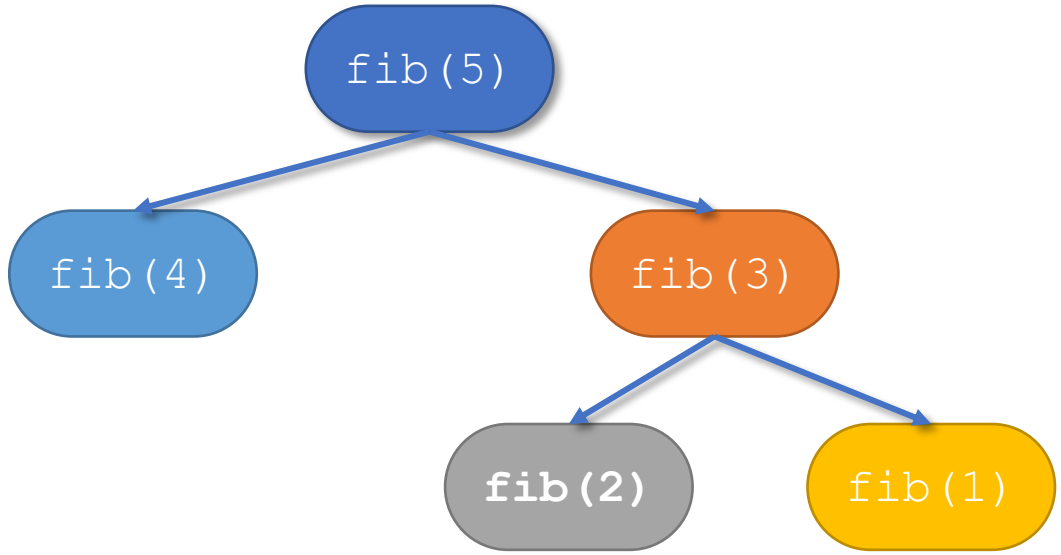


fib
$n = 3$
1 +

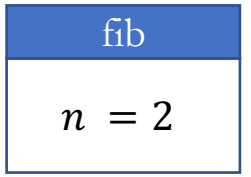
fib
$n = 5$

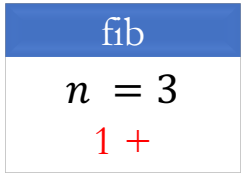
```
1  public static int fib(int n) {
2      if (n>2) {
3          return fib(n-2) + fib(n-1);
4      }
5      else return 1;
6  }
```

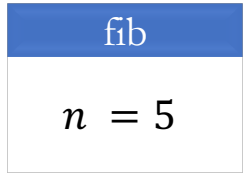
- Recursion Tree



Stack grows

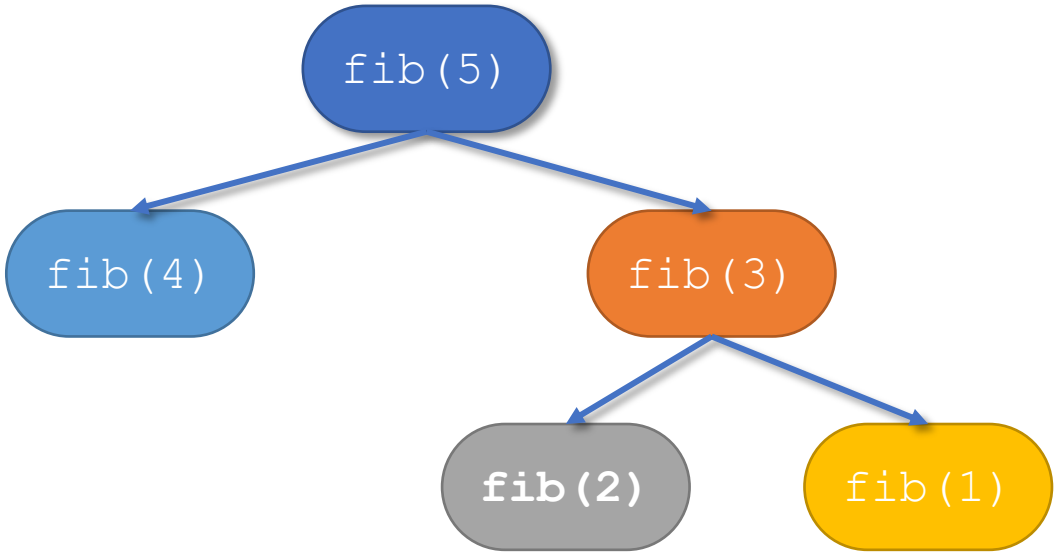






```
1  public static int fib(int n) {
2      if (n>2) {
3          return fib(n-2) + fib(n-1);
4      }
5      else return 1;
6  }
```

- Recursion Tree



Stack

fib
$n = 2$
return 1

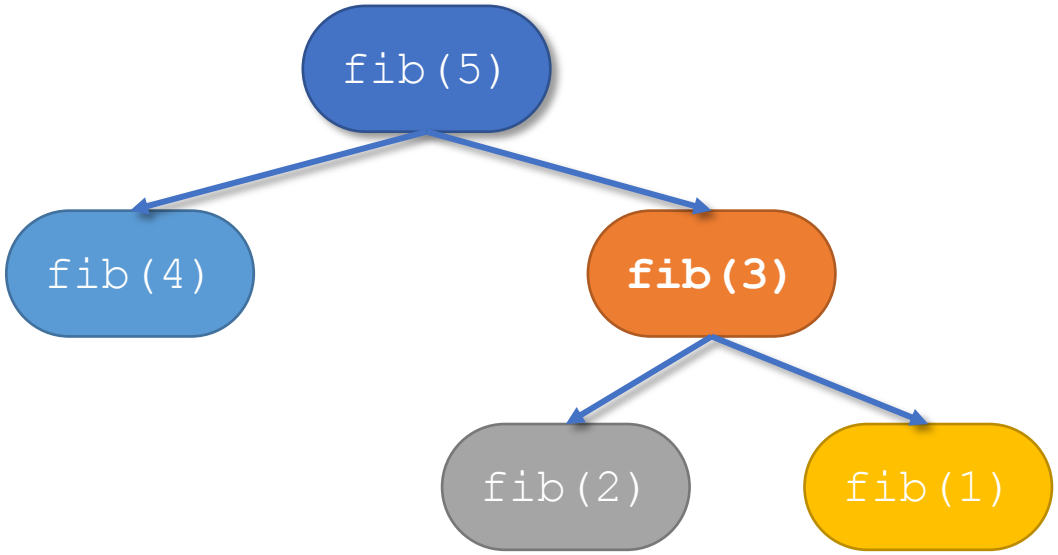
fib
$n = 3$
1 +

fib
$n = 5$

```
1  public static int fib(int n) {
2      if (n>2) {
3          return fib(n-2) + fib(n-1);
4      }
5      else return 1;
6  }
```

- Recursion Tree

Stack unwinds

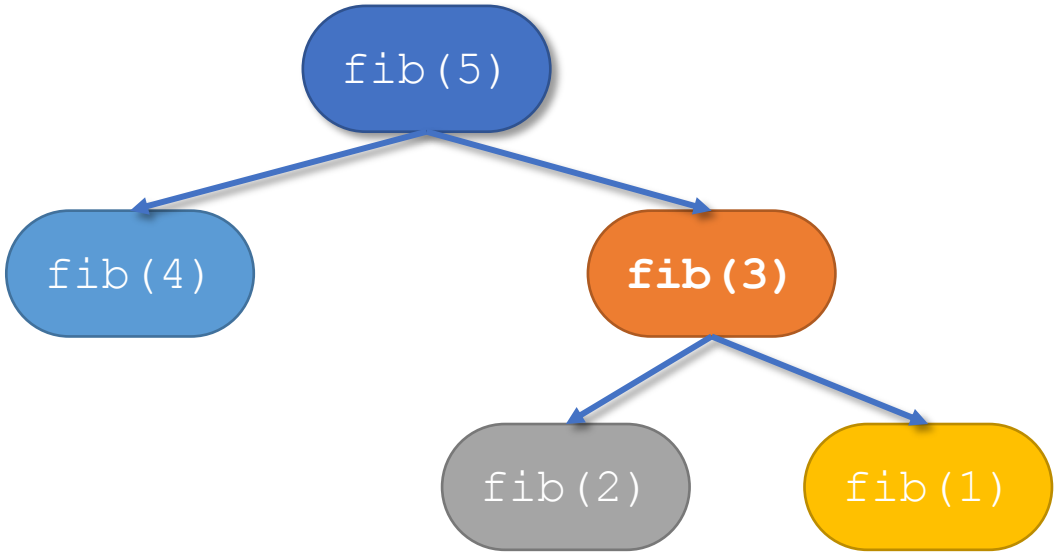


fib
$n = 3$
$1 + 1$

fib
$n = 5$

```
1  public static int fib(int n) {
2      if (n>2) {
3          return fib(n-2) + fib(n-1);
4      }
5      else return 1;
6  }
```

- Recursion Tree



Stack

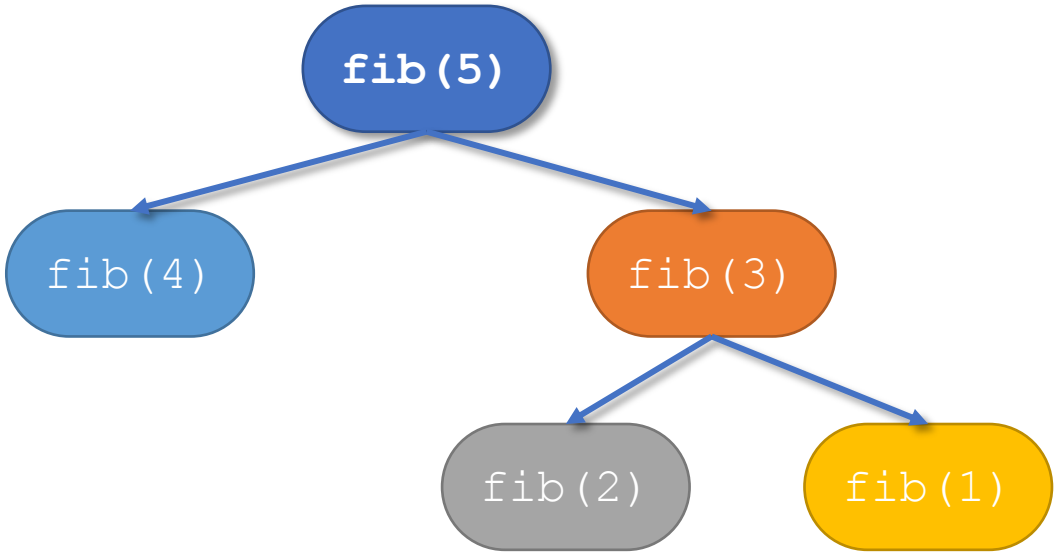
fib
$n = 3$
return 2

fib
$n = 5$

```
1  public static int fib(int n) {
2      if (n>2) {
3          return fib(n-2) + fib(n-1);
4      }
5      else return 1;
6  }
```

- Recursion Tree

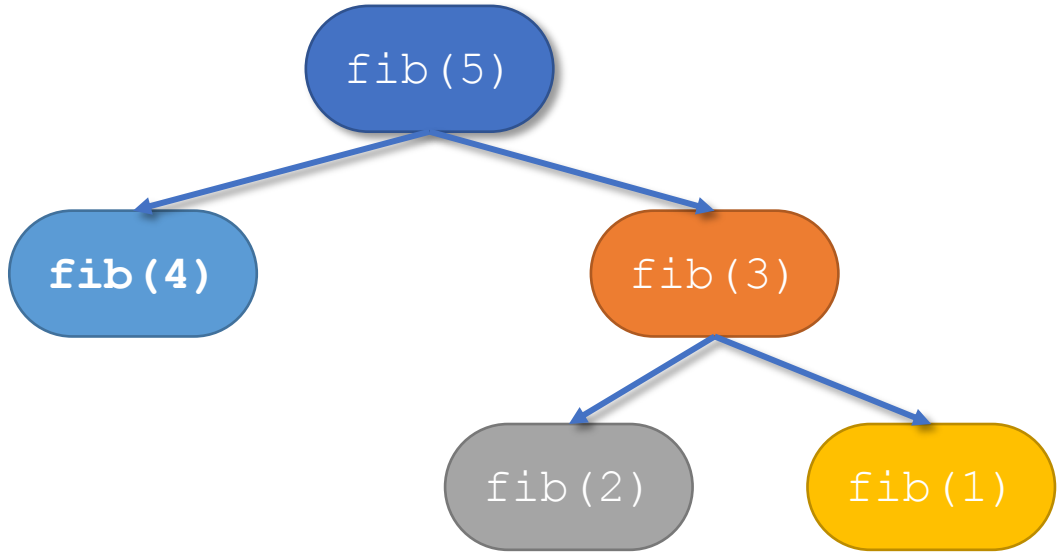
Stack unwinds



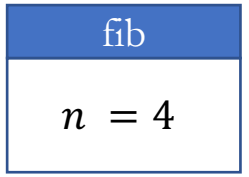
fib
$n = 5$
2 +

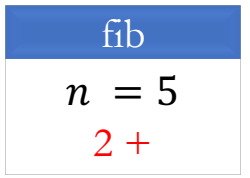
```
1  public static int fib(int n) {
2      if (n>2) {
3          return fib(n-2) + fib(n-1);
4      }
5      else return 1;
6  }
```

- Recursion Tree



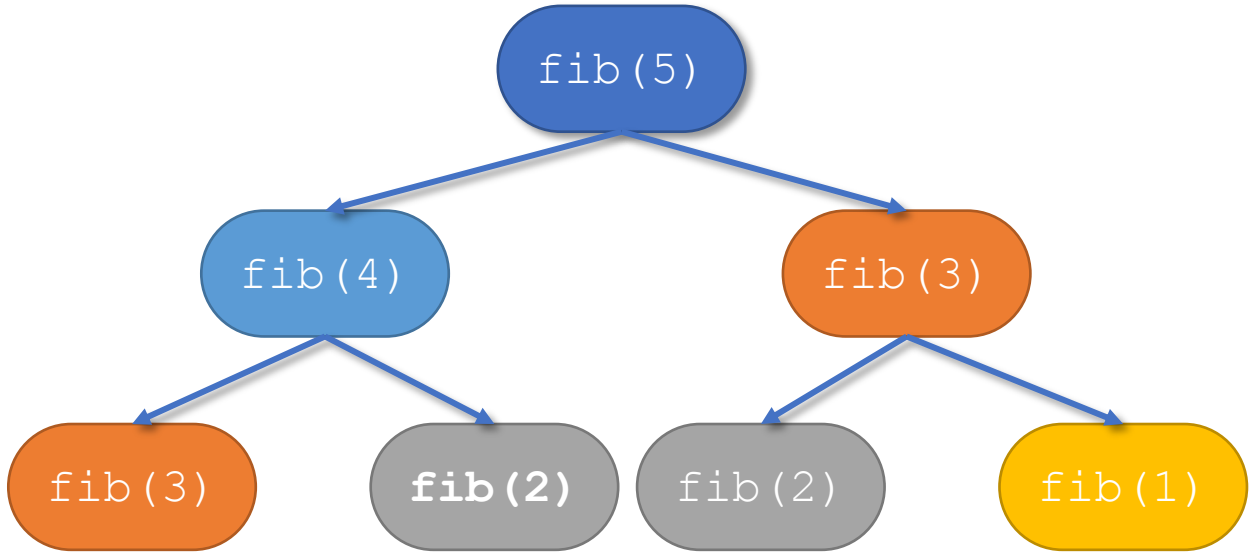
Stack grows





```
1  public static int fib(int n) {
2      if (n>2) {
3          return fib(n-2) + fib(n-1);
4      }
5      else return 1;
6  }
```

- Recursion Tree



Stack grows

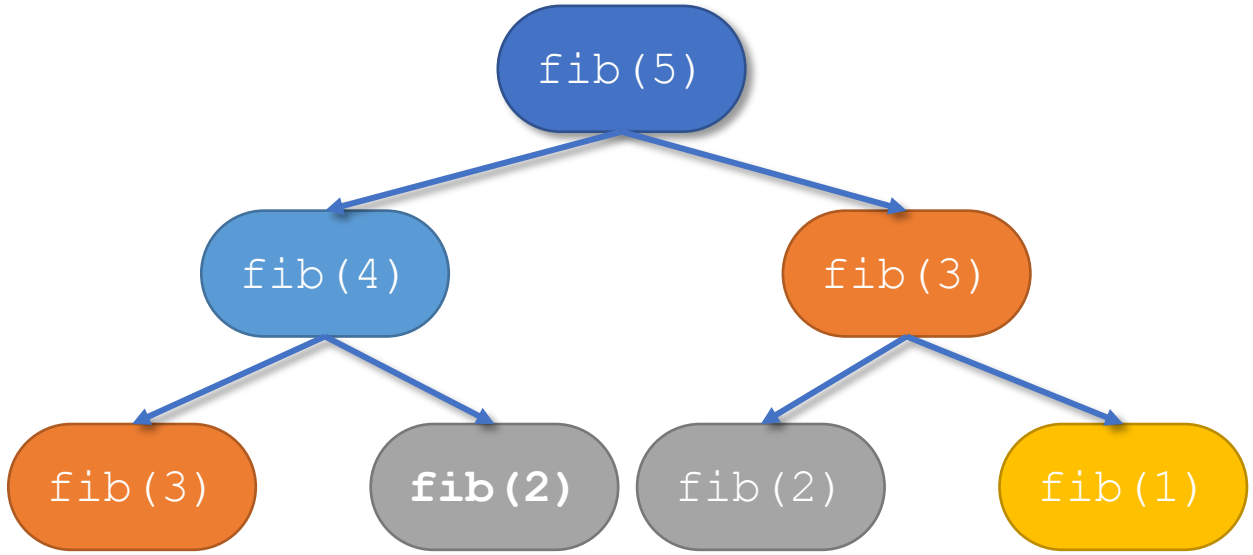
fib
$n = 2$

fib
$n = 4$

fib
$n = 5$
2 +

```
1  public static int fib(int n) {
2      if (n>2) {
3          return fib(n-2) + fib(n-1);
4      }
5      else return 1;
6  }
```

- Recursion Tree



Stack

fib
$n = 2$
return 1

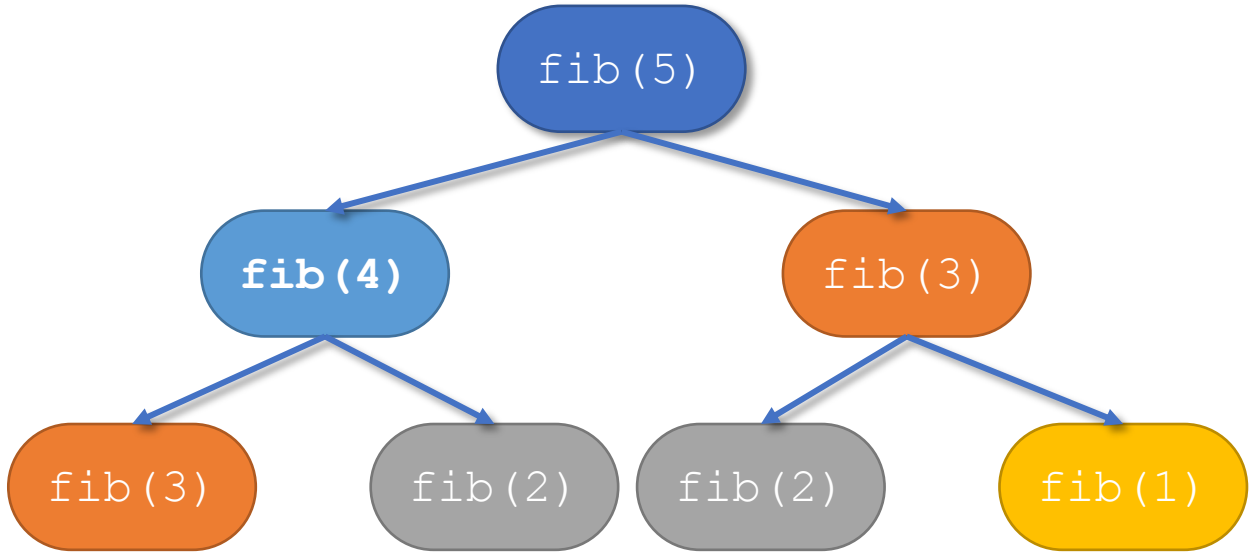
fib
$n = 4$

fib
$n = 5$
2 +

```
1  public static int fib(int n) {
2      if (n>2) {
3          return fib(n-2) + fib(n-1);
4      }
5      else return 1;
6  }
```

- Recursion Tree

Stack unwinds

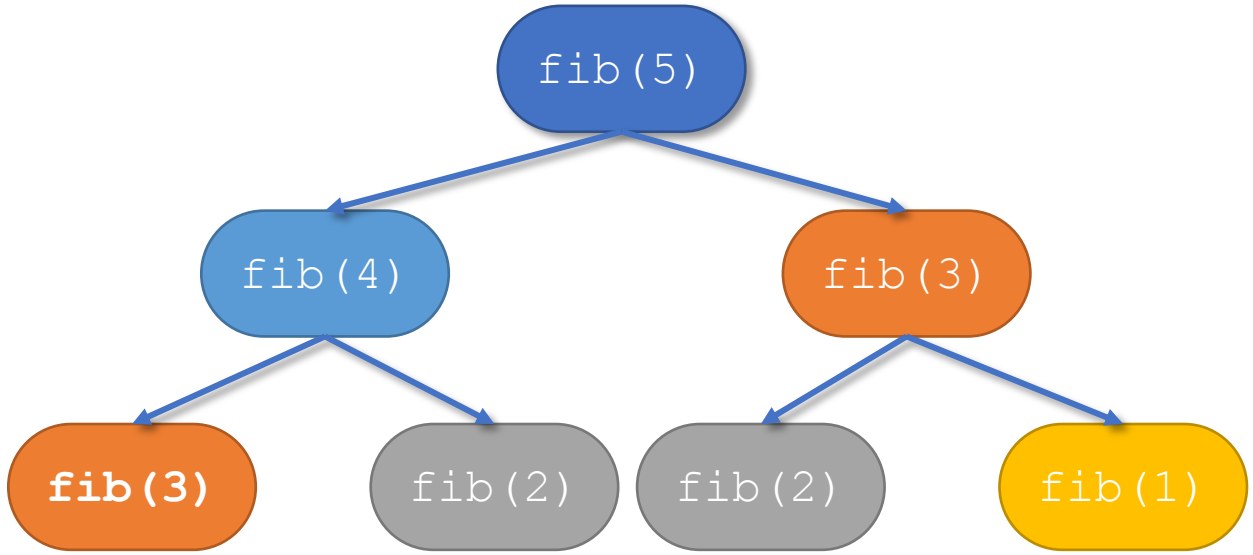


fib
$n = 4$
1 +

fib
$n = 5$
2 +

```
1  public static int fib(int n) {
2      if (n>2) {
3          return fib(n-2) + fib(n-1);
4      }
5      else return 1;
6  }
```

- Recursion Tree



Stack grows

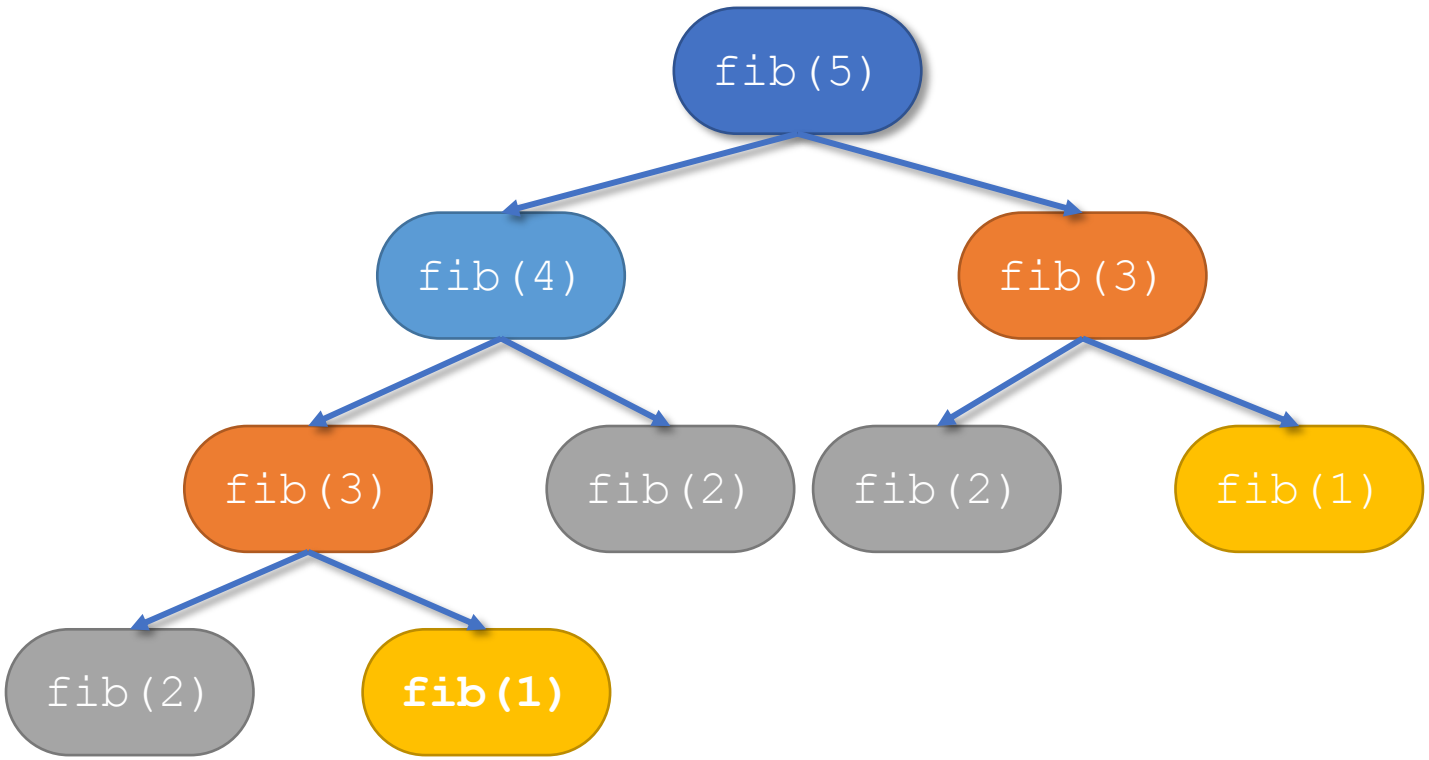
fib
$n = 3$

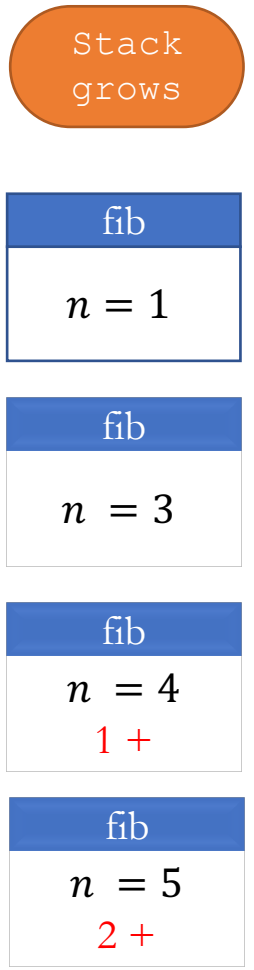
fib
$n = 4$
1 +

fib
$n = 5$
2 +

```
1  public static int fib(int n) {
2      if (n>2) {
3          return fib(n-2) + fib(n-1);
4      }
5      else return 1;
6  }
```

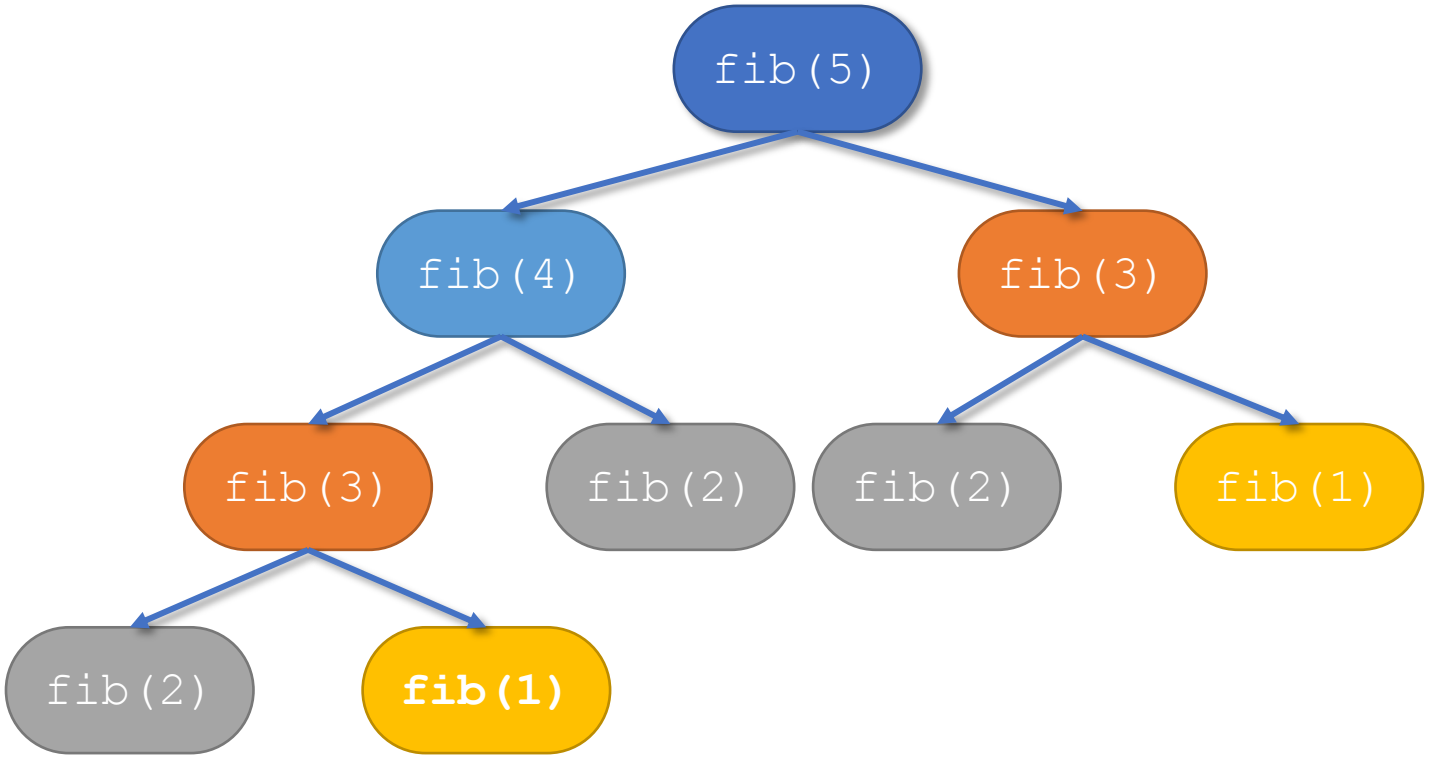
- Recursion Tree

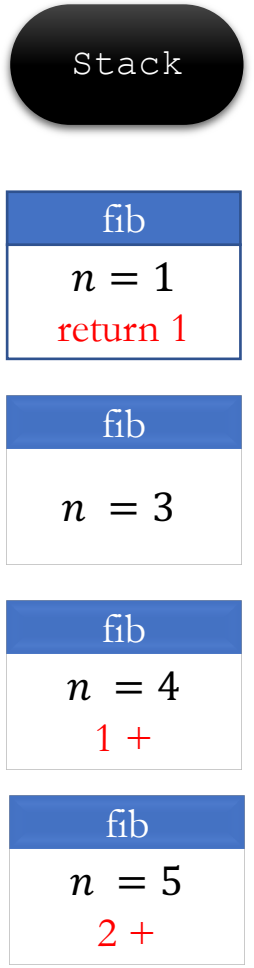




```
1  public static int fib(int n) {
2      if (n>2) {
3          return fib(n-2) + fib(n-1);
4      }
5      else return 1;
6  }
```

- Recursion Tree

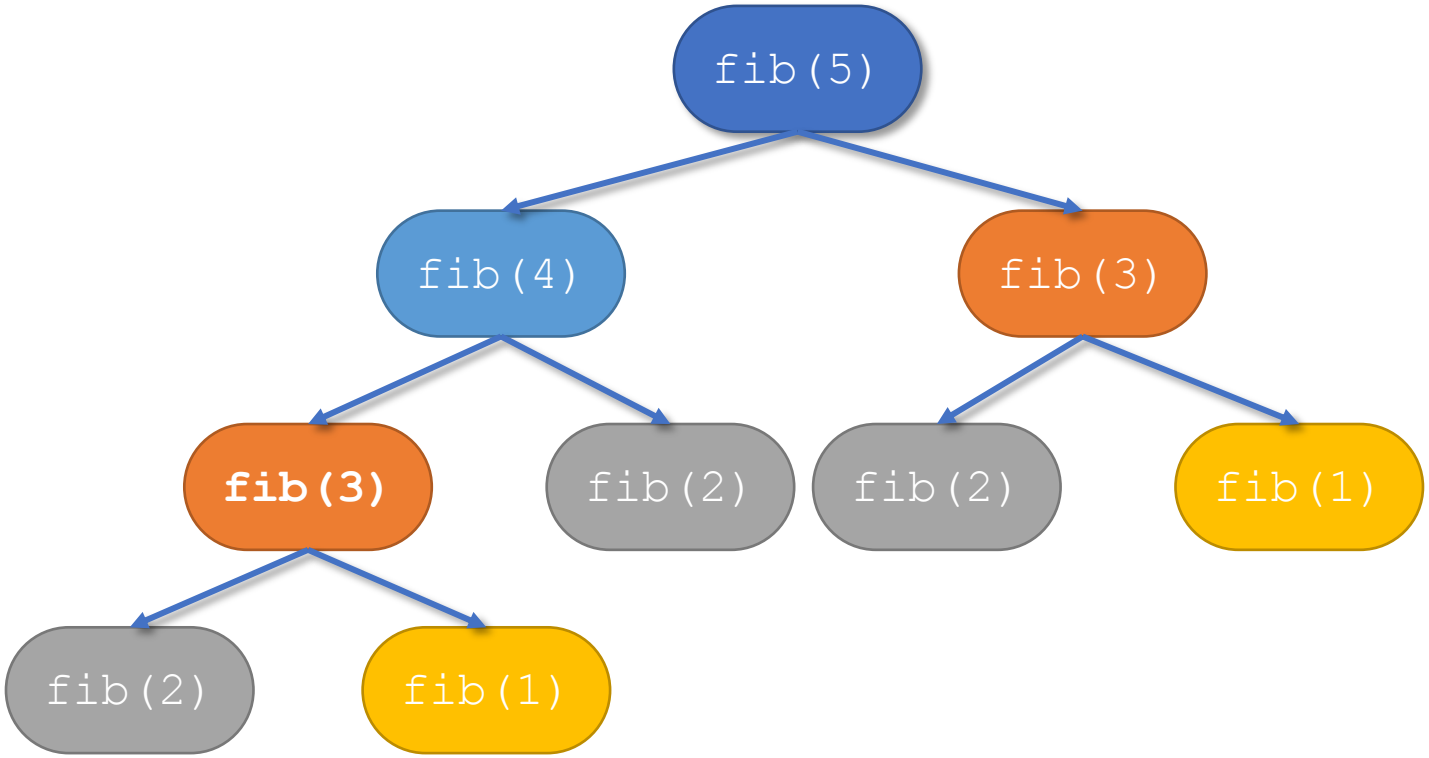




```
1  public static int fib(int n) {
2      if (n>2) {
3          return fib(n-2) + fib(n-1);
4      }
5      else return 1;
6  }
```

- Recursion Tree

Stack unwinds



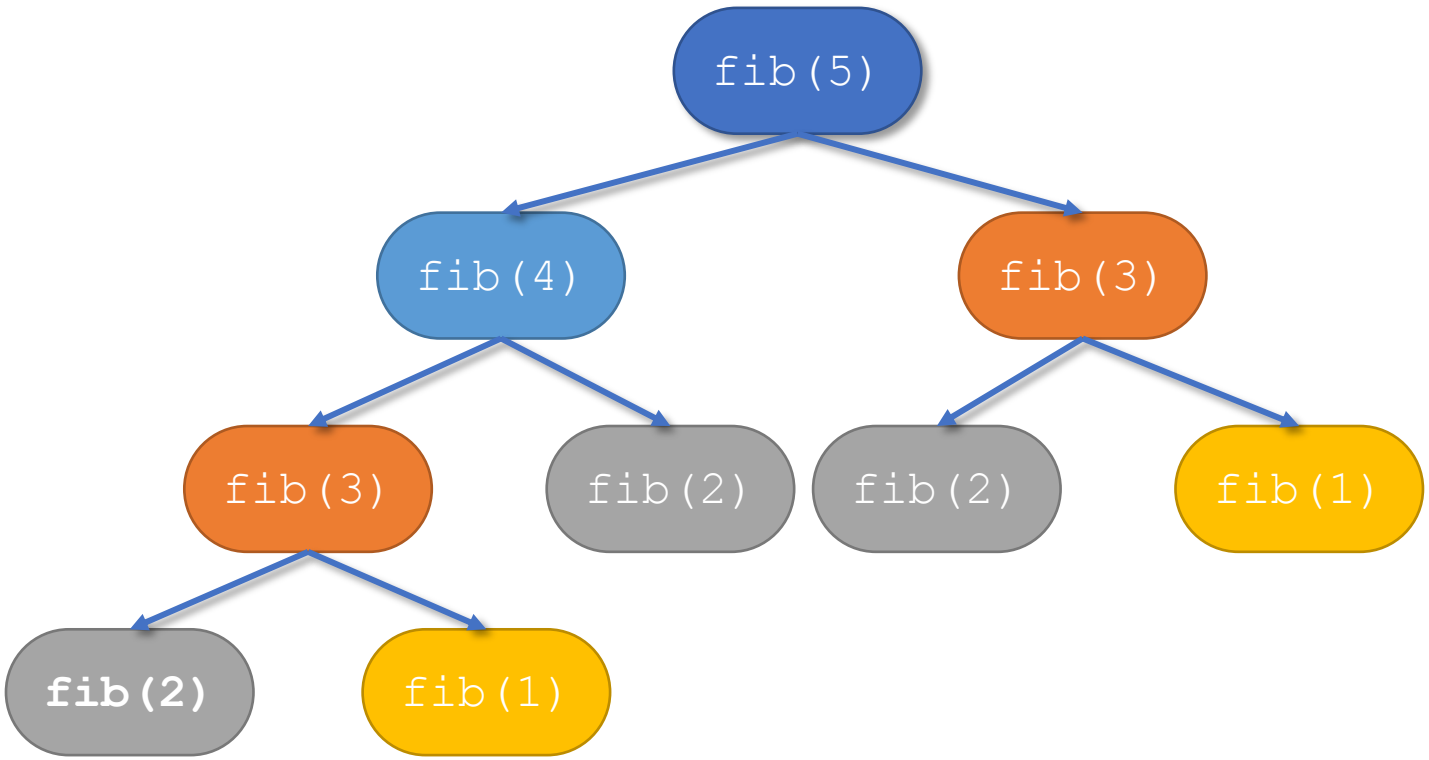
fib
$n = 3$
1 +

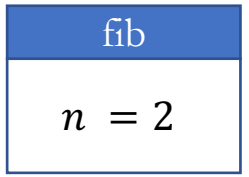
fib
$n = 4$
1 +

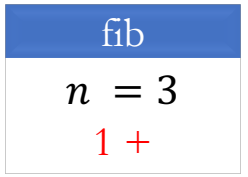
fib
$n = 5$
2 +

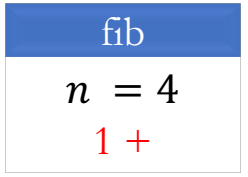
```
1  public static int fib(int n) {
2      if (n>2) {
3          return fib(n-2) + fib(n-1);
4      }
5      else return 1;
6  }
```

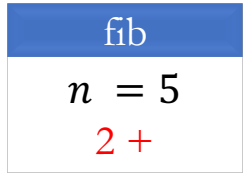
- Recursion Tree





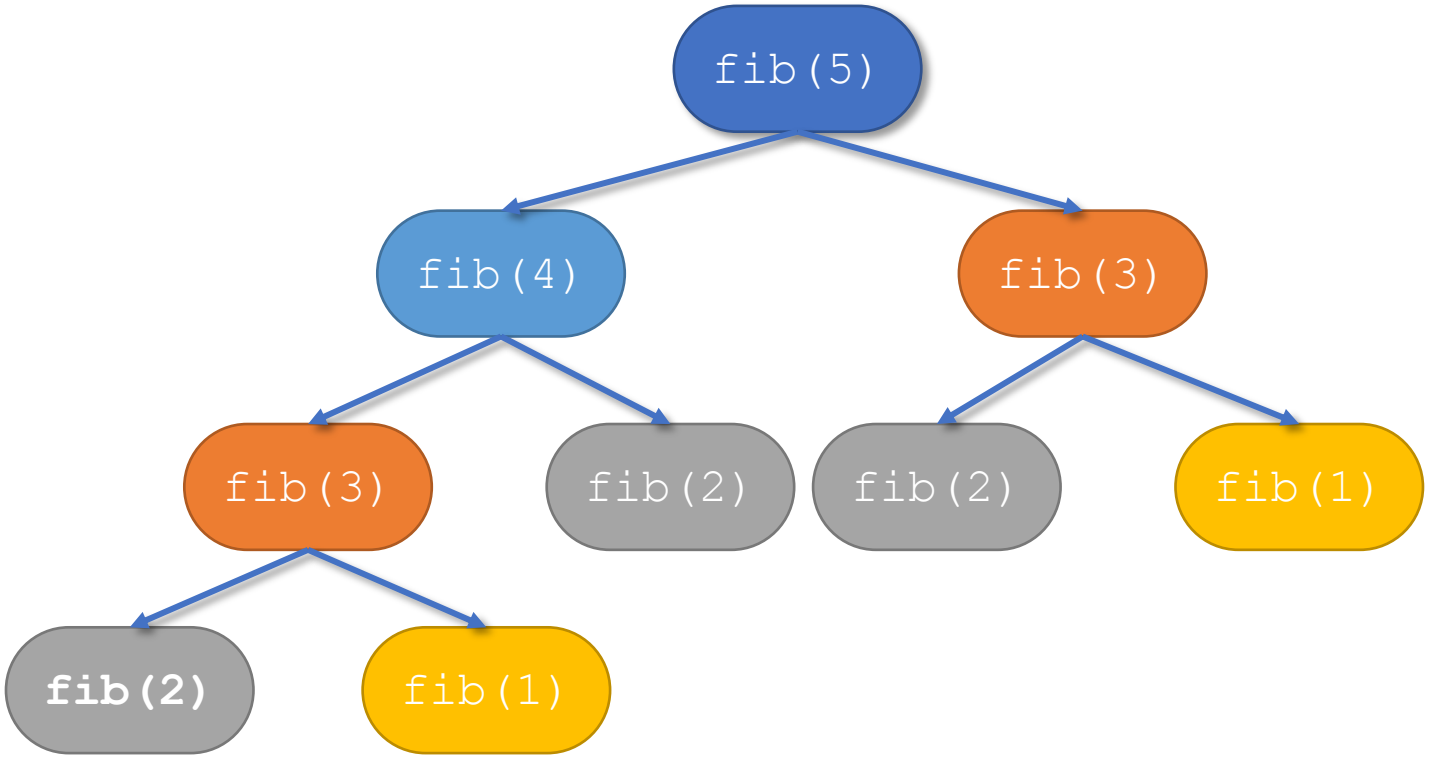


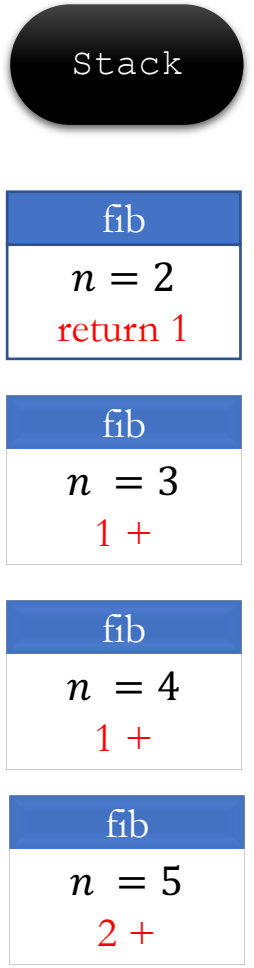




```
1  public static int fib(int n) {
2      if (n>2) {
3          return fib(n-2) + fib(n-1);
4      }
5      else return 1;
6  }
```

- Recursion Tree

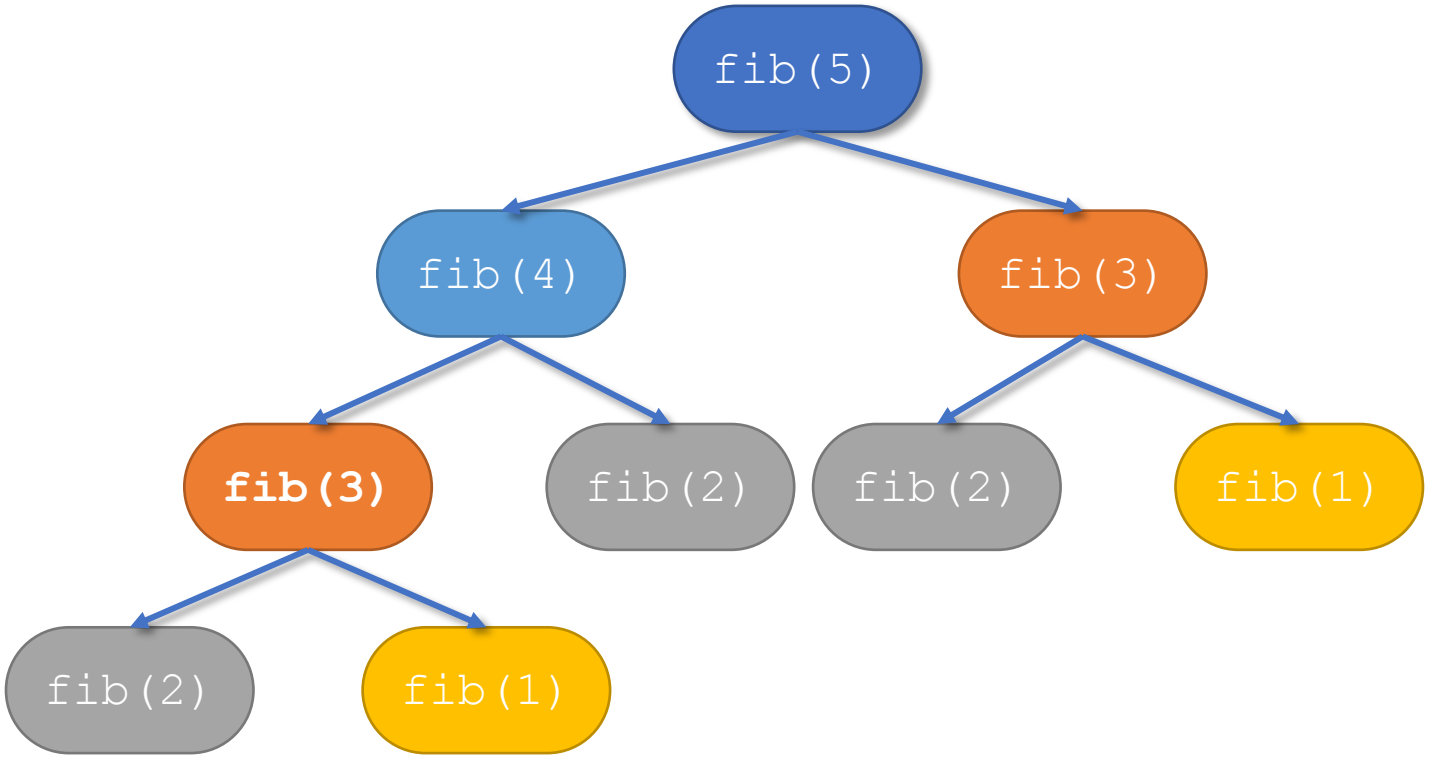




```
1  public static int fib(int n) {
2      if (n>2) {
3          return fib(n-2) + fib(n-1);
4      }
5      else return 1;
6  }
```

- Recursion Tree

Stack unwinds



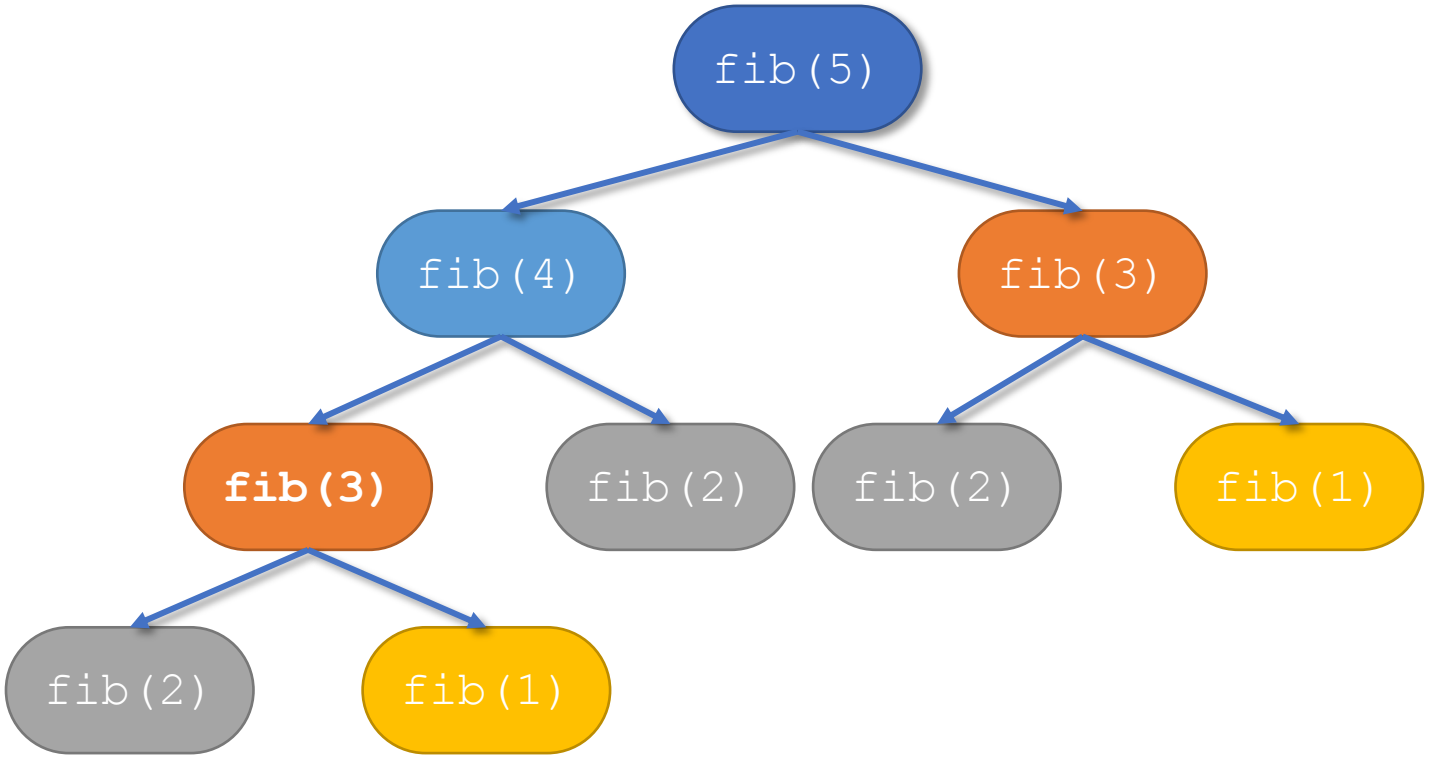
fib
$n = 3$
1 + 1

fib
$n = 4$
1 +

fib
$n = 5$
2 +

```
1  public static int fib(int n) {
2      if (n>2) {
3          return fib(n-2) + fib(n-1);
4      }
5      else return 1;
6  }
```

- Recursion Tree



Stack

fib
$n = 3$
return 2

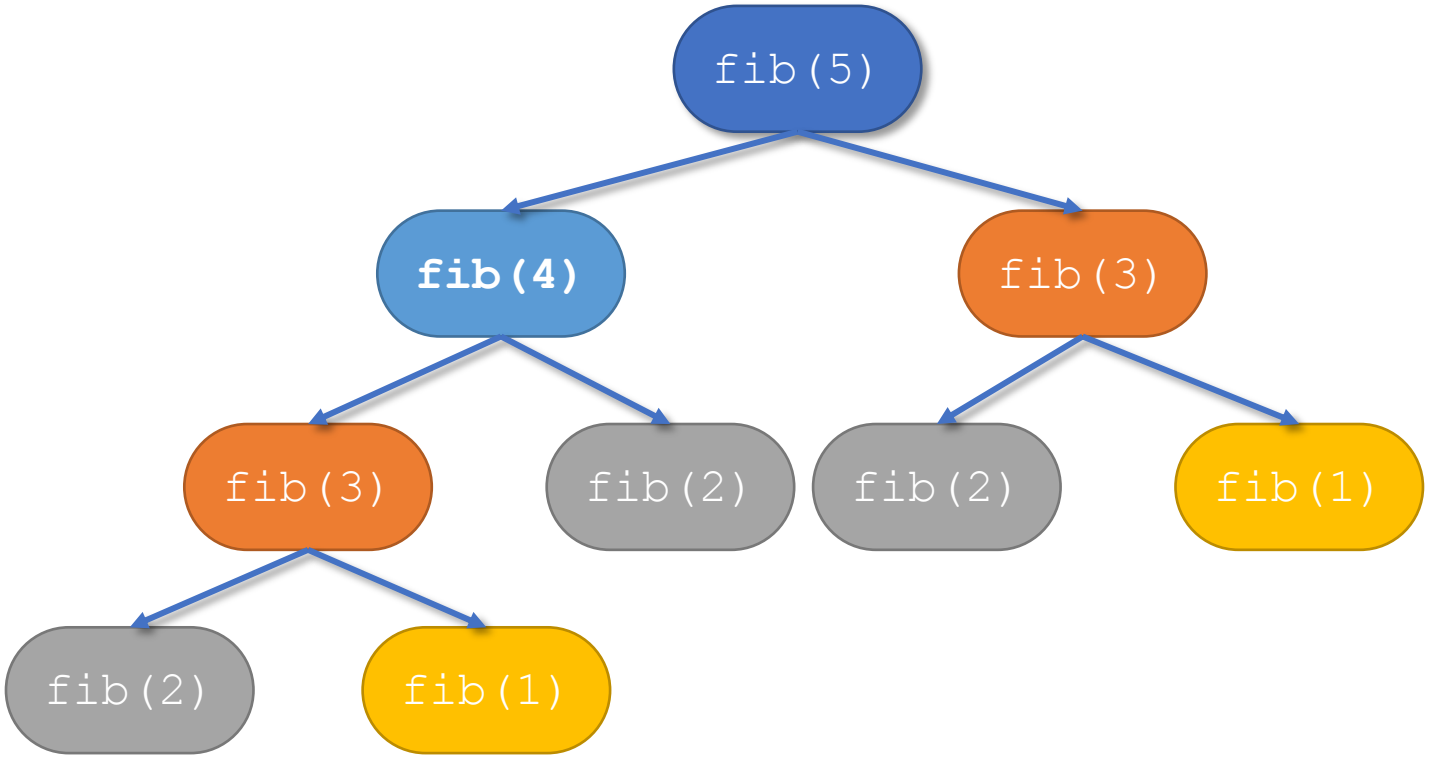
fib
$n = 4$
1 +

fib
$n = 5$
2 +

```
1  public static int fib(int n) {
2      if (n>2) {
3          return fib(n-2) + fib(n-1);
4      }
5      else return 1;
6  }
```

- Recursion Tree

Stack unwinds

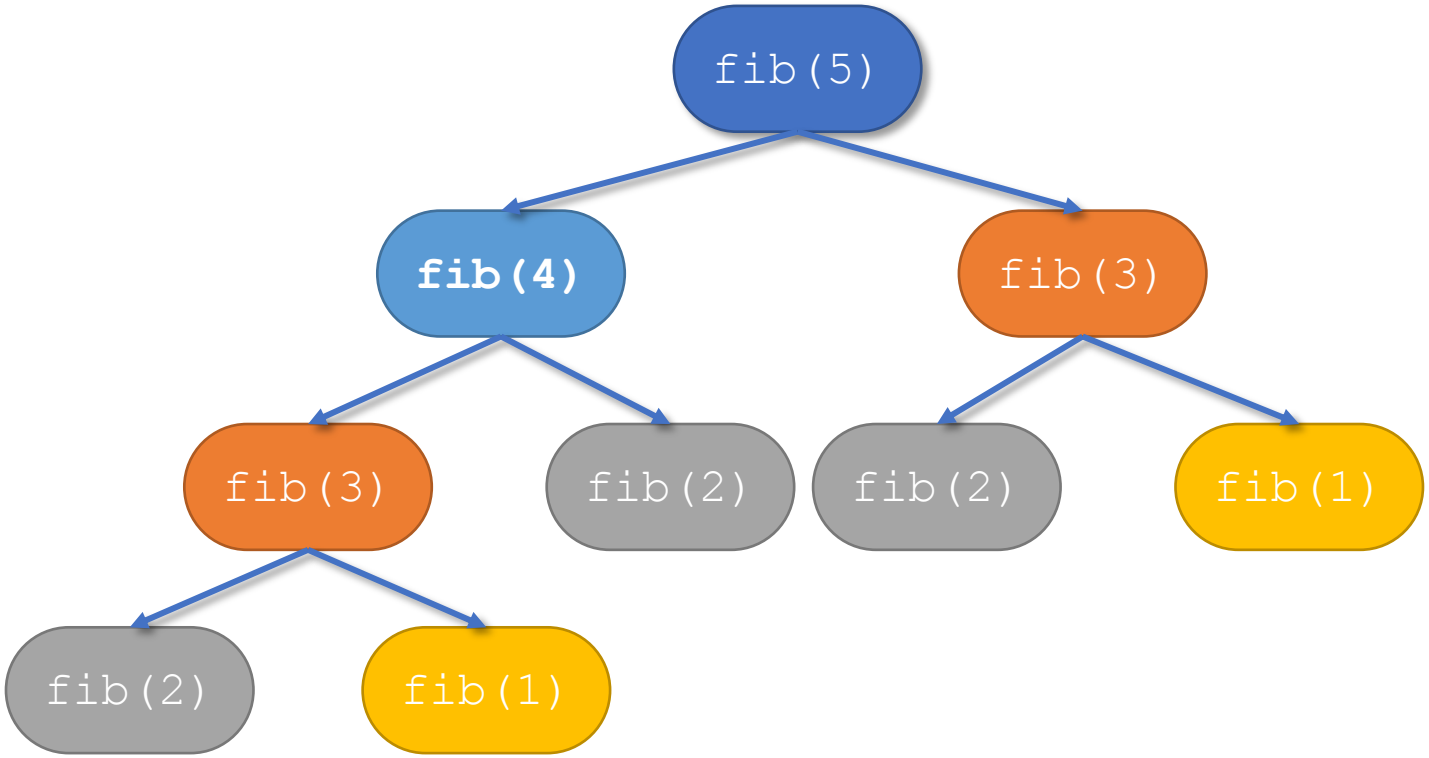


fib
$n = 4$
1 + 2

fib
$n = 5$
2 +

```
1  public static int fib(int n) {
2      if (n>2) {
3          return fib(n-2) + fib(n-1);
4      }
5      else return 1;
6  }
```

- Recursion Tree



Stack

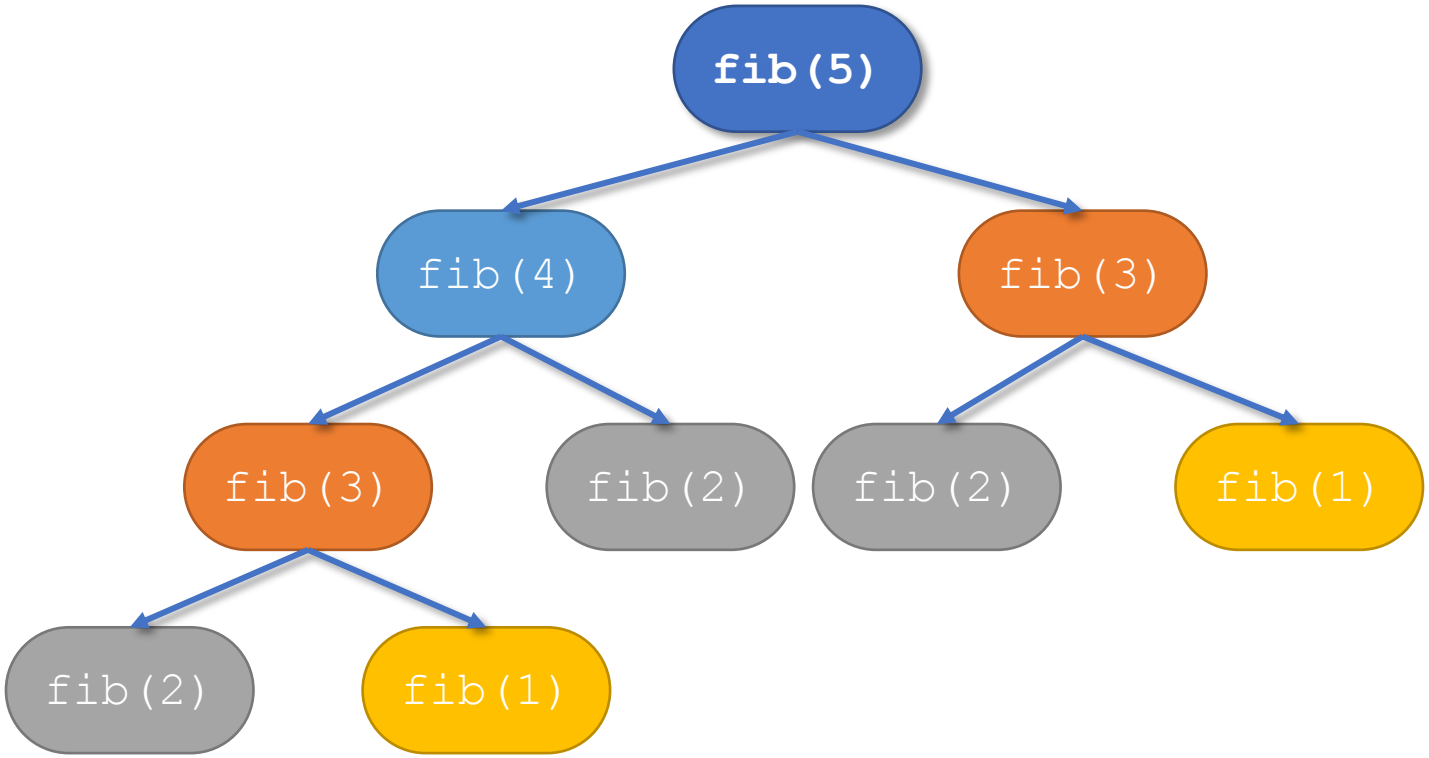
fib
$n = 4$
return 3

fib
$n = 5$
2 +

```
1  public static int fib(int n) {
2      if (n>2) {
3          return fib(n-2) + fib(n-1);
4      }
5      else return 1;
6  }
```

- Recursion Tree

Stack unwinds

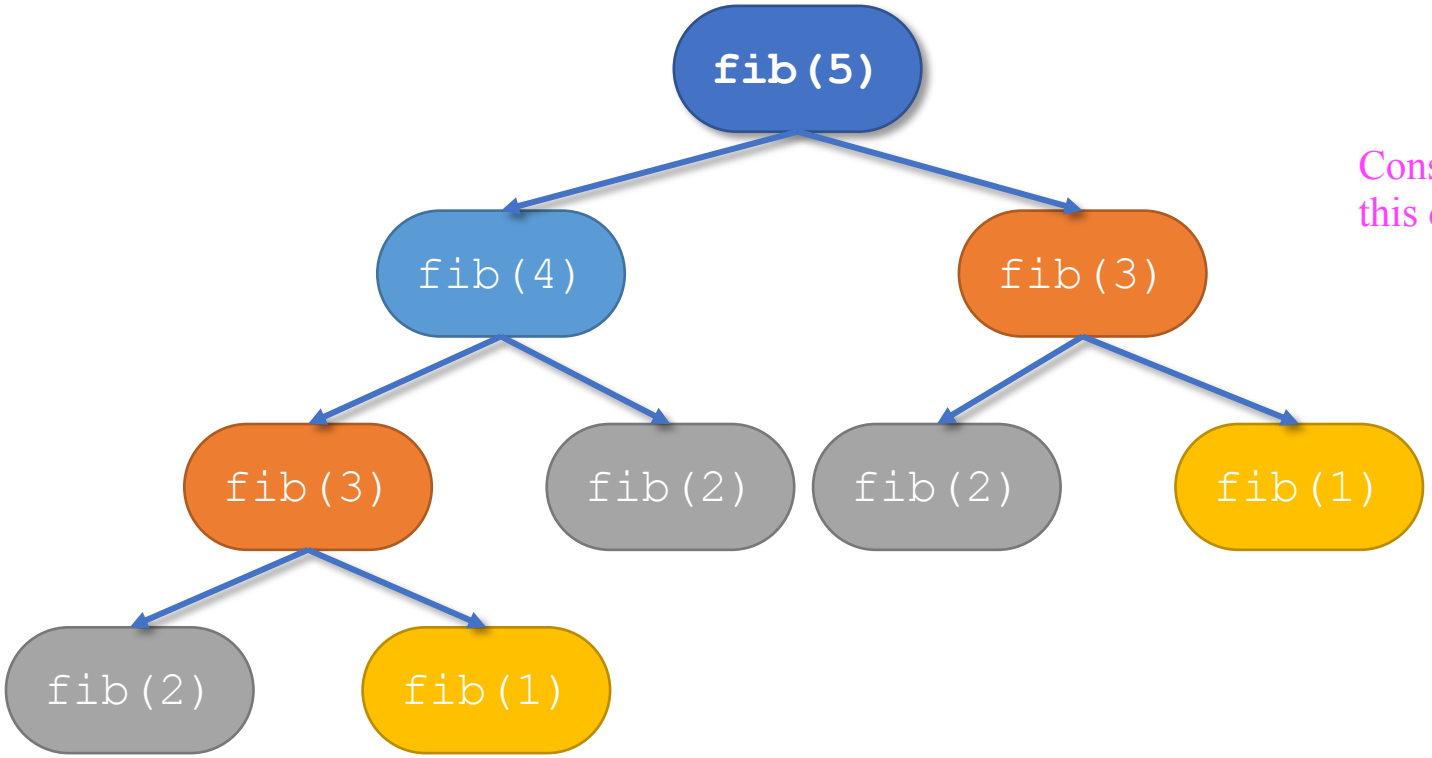


fib
$n = 5$
$2 + 3$

```
1  public static int fib(int n) {
2      if (n>2) {
3          return fib(n-2) + fib(n-1);
4      }
5      else return 1;
6  }
```

- Recursion Tree

Stack



Consider giving students a website to go through this demo themselves using different numbers

fib
$n = 5$
return 5

CLICKER QUESTION #2

Fibonacci Iterative

- Cost: time is linear, extra space is constant

Get rid of this and use clicker question to ask for extra space

```
1  public static double fib(int n) {
2      double f1 = 0, f2 = 1, temp;
3      for (int i = 2; i <= n; i++) {
4          temp = f2;
5          f2 += f1;
6          f1 = temp;
7      }
8      return f2;
9  }
```

CLICKER QUESTION #3

Fibonacci Recursive – Version 2

- Exercise: Cost: time ?, extra space ?

This is confusing. An actual example of small value would be helpful tp trace and understand underlying idea.

```
1  public static int fib(int n, int a, int b) {
2      if (n>1) {
3          return fib(n-1, b, a + b);
4      } else {
5          return b;
6      }
7  }
8
9  // call with fib(n, 0, 1);
```

Fibonacci Recursive – Version 2

- Cost: time is linear, extra space is linear

```
1 public static int fib(int n, int a, int b) {
2     if (n>1) {
3         return fib(n-1, b, a + b);
4     } else {
5         return b;
6     }
7 }
8
9 // call with fib(n, 0, 1);
```

Fibonacci Arithmetic

- Cost: time is constant (if we assume sqrt and pow are constant, but are they?!), extra space is constant, but roundoff errors to consider

```
1 public static double fib(int n) {
2     final double root5 = Math.sqrt(5);
3     double n1 = (1 + root5) / 2;
4     double n2 = (1 - root5) / 2;
5     return  (Math.pow(n1, n) -
6             Math.pow(n2, n)) / root5;
7 }
```

Worksheet

Growth rates!

Worksheet

Fun Example

- Set A has $n-1$ unique integers from the inclusive range $[1, n]$
- Find the missing number.
  - ❑ Really bad ideas?!
  - ❑ Reasonable ideas?
  - ❑ Better/clever ideas?

Worksheet

- Set A has $n-1$ unique integers from the inclusive range $[1, n]$
- Find the missing number.

Bad Solutions

- Generate all possible permutations of each group of $n-1$ values and compare to set A, this is really bad:

$$O(N! \sim sqrt(2\ pi\ N)\left(\frac{N}{e}\right)^N)$$

Worksheet

- Set A has $n-1$ unique integers from the inclusive range $[1, n]$
- Find the missing number.

Bad Solutions

- Generate all possible permutations of each group of $n-1$ values and compare to set A, this is really bad: $O(N! \sim sqrt(2\ pi\ N)\left(\frac{N}{e}\right)^N)$
- Generate a random number between $[1, n]$, see if it's in the set, repeat until found, this is bad too: $O(N^2)$ expected, but not deterministic.

Worksheet

- Set A has $n-1$ unique integers from the inclusive range $[1, n]$
- Find the missing number.

OK Solutions

- For each number 1 to n, see if set contains it, somewhat obvious: $O(N^2)$

Worksheet

- Set A has $n-1$ unique integers from the inclusive range $[1, n]$
- Find the missing number.

OK Solutions

- For each number 1 to n, see if set contains it, somewhat obvious: $O(N^2)$
- Sort the set, scan through to find missing value,

$$O\big(O(\text{sort}) + O(\text{scan})\big) = O(N + O(\text{sort})) = O(\text{sort}).$$

Worksheet

- Set A has $n-1$ unique integers from the inclusive range $[1, n]$
- Find the missing number.

Good Solutions

- Boolean array `b` size N, put true into `b[a-1]` for each value `a` in `A`, linear check on `b[i]` for false value – $O(n)$ time, extra $O(n)$ space.

Worksheet

- Set A has $n-1$ unique integers from the inclusive range $[1, n]$
- Find the missing number.

Good Solutions

- Boolean array `b` size N, put true into `b[a-1]` for each value `a` in `A`, linear check on `b[i]` for false value – $O(n)$ time, extra $O(n)$ space.

- Add values in set A, subtract from sum of 1 to n, which is $\frac{n(n+1)}{2}$; $O(N)$ time to add values, constant space.

HW3 Overview

We build some operation counts into our data structure in order to compare algorithms

Generics with bounds

```
1 public interface SortingAlgorithm<T extends Comparable<T>> {
2
3 }
```

- This is an interface with `T` as the generic type
- `T` has to be a type that extends or implements `Comparable<T>` meaning that it has an implementation for `int compareTo(T);`

We use the generic type with bounds to guarantee there will be a way to compare generic elements when sorting.