CSCI 132:
Basic Data Structures and Algorithms

Recursion (Part 1)

Reese Pearsall
Spring 2023

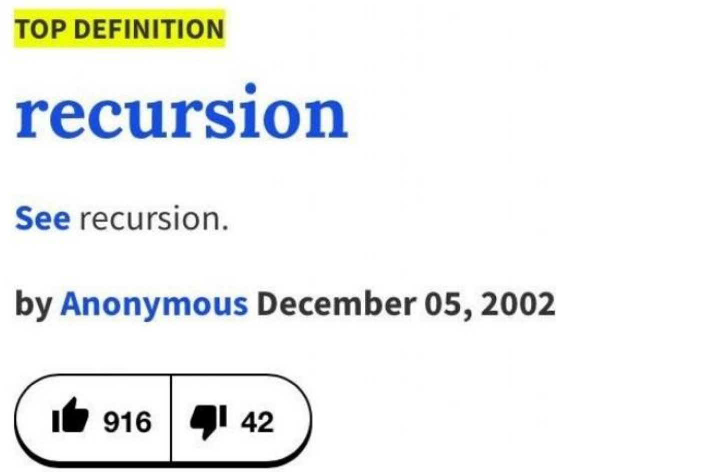
Announcements

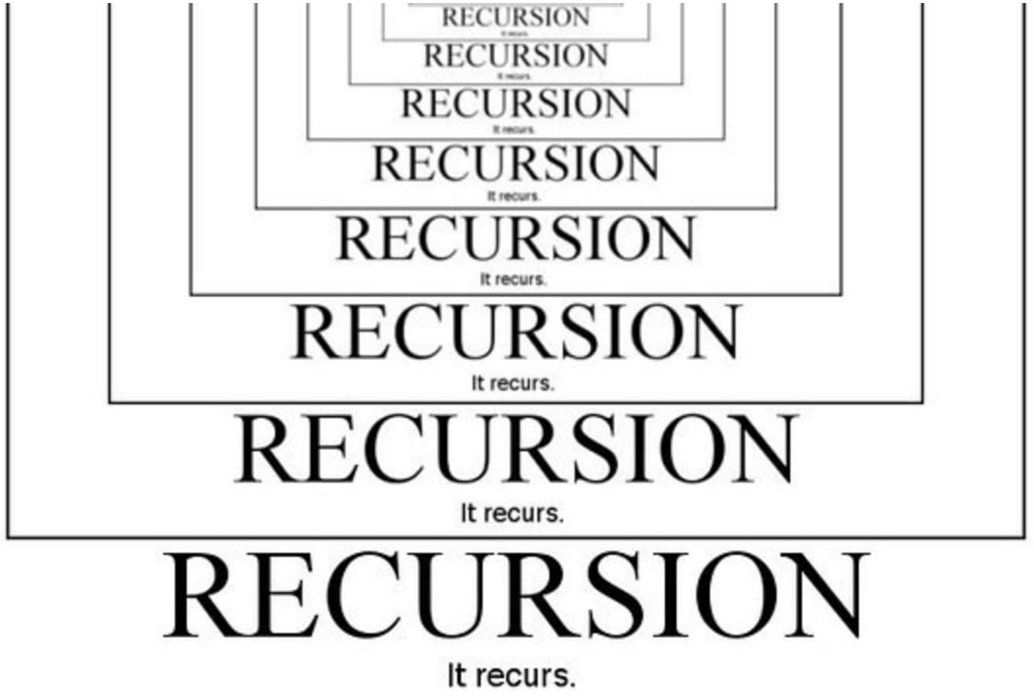
Program 4 Due April 19th

Recursion is a problem-solving technique that involves a *method calling itself* to solve some smaller problem

```java
static int factorial(int n)
    {
        if (n == 0)
            return 1;

        return n * factorial(n - 1);
    }
```

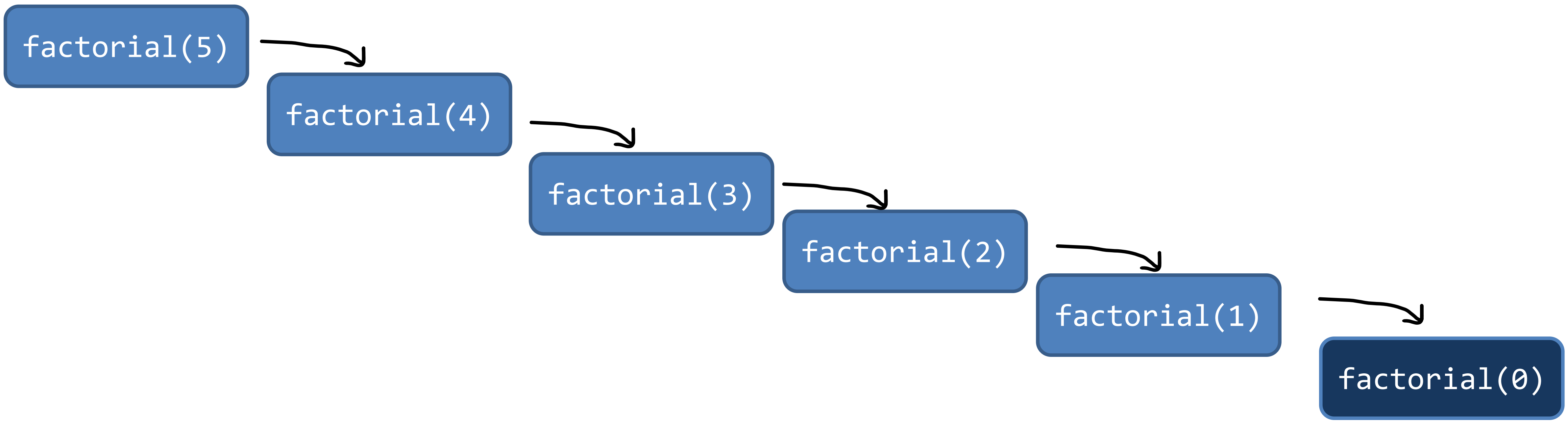




```java
static int factorial(int n)
    {
        if (n == 0)
            return 1;

        return n * factorial(n - 1);
    }
```

We can solve the factorial for n by solving smaller problems (factorial of n-1) !



```
static int factorial(int n)
    {
        if (n == 0)
            return 1;

        return n * factorial(n - 1);
    }
```

(base case)

(recursive case)

We can solve the factorial for n by solving smaller problems (factorial of n-1) !

factorial(5) → factorial(4) → factorial(3) → factorial(2) → factorial(1) → factorial(0)

Recursive solutions must have the two following conditions:

1. Base Case
2. Recursive Case

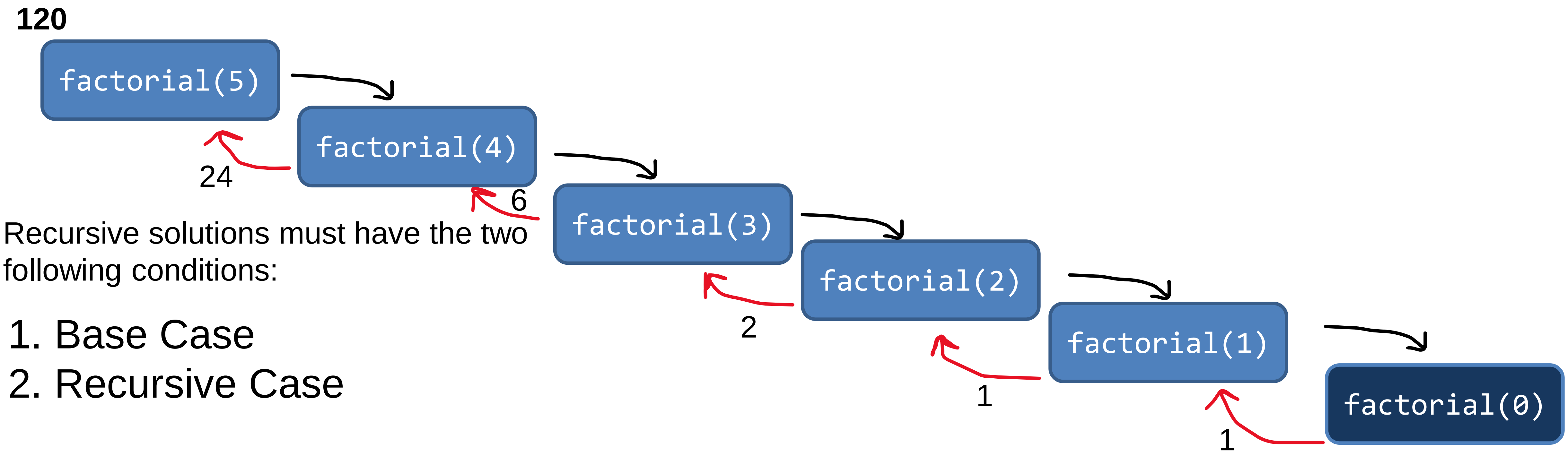
```java
static int factorial(int n)
    {
        if (n == 0)
            return 1;

        return n * factorial(n - 1);
    }
```

(base case)

(recursive case)

We can solve the factorial for n by solving smaller problems (factorial of n-1) !



Recursive solutions must have the two following conditions:

1. Base Case
2. Recursive Case

The Fibonacci sequence is a sequence in which each number is the sum of the two preceding ones

So, the N^{th} digit of the Fibonacci Sequence = f(N-1) + f(N-2)

The Fibonacci Sequence

1,1,2,3,5,8,13,21,34,55,89,144,233,377…

1+1=2	**13+21=34**
1+2=3	**21+34=55**
2+3=5	**34+55=89**
3+5=8	**55+89=144**
5+8=13	**89+144=233**
8+13=21	**144+233=377**

Because the solution to some problem can be expressed in terms of some smaller problem(s), recursion may be a good fit here

The Fibonacci sequence is a sequence in which each number is the sum of the two preceding ones

So, the N^{th} digit of the Fibonacci Sequence = f(N-1) + f(N-2)

The Fibonacci Sequence

1,1,2,3,5,8,13,21,34,55,89,144,233,377…

1+1=2	**13+21=34**
1+2=3	**21+34=55**
2+3=5	**34+55=89**
3+5=8	**55+89=144**
5+8=13	**89+144=233**
8+13=21	**144+233=377**

Base Case?

Recursive Case?

Calculate

The Fibonacci sequence is a sequence in which each number is the sum of the two preceding ones

So, the N^{th} digit of the Fibonacci Sequence = f(N-1) + f(N-2)

The Fibonacci Sequence

1,1,2,3,5,8,13,21,34,55,89,144,233,377…

1+1=2	**13+21=34**
1+2=3	**21+34=55**
2+3=5	**34+55=89**
3+5=8	**55+89=144**
5+8=13	**89+144=233**
8+13=21	**144+233=377**

Base Case?

If finding the 1st or 2nd digit, return 1

Recursive Case?

Calculate the previous two digits, f(n-1), f(n-2)

The Fibonacci sequence is a sequence in which each number is the sum of the two preceding ones

So, the N^{th} digit of the Fibonacci Sequence = f(N-1) + f(N-2)

The Fibonacci Sequence

1,1,2,3,5,8,13,21,34,55,89,144,233,377…

1+1=2	13+21=34
1+2=3	21+34=55
2+3=5	34+55=89
3+5=8	55+89=144
5+8=13	89+144=233
8+13=21	144+233=377

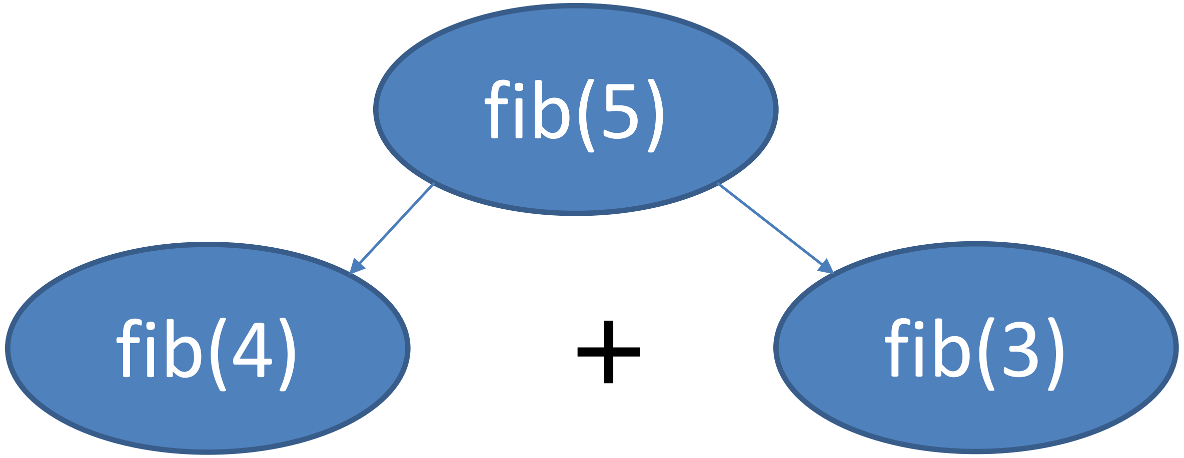
Base Case?

If finding the 1^{st} or 2^{nd} digit, return 1

Recursive Case?

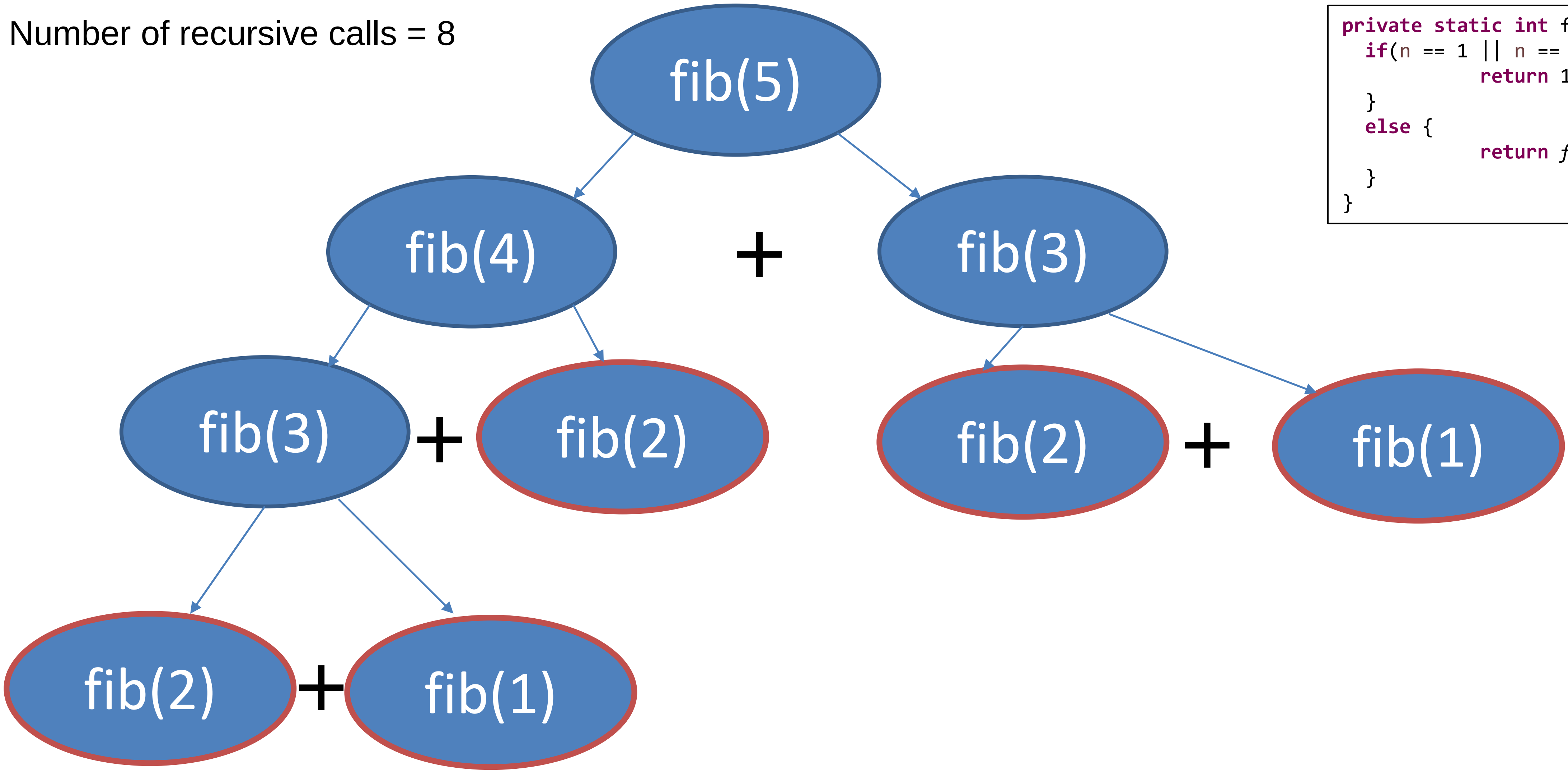
Calculate the previous two digits, f(n-1), f(n-2) fib(5)

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



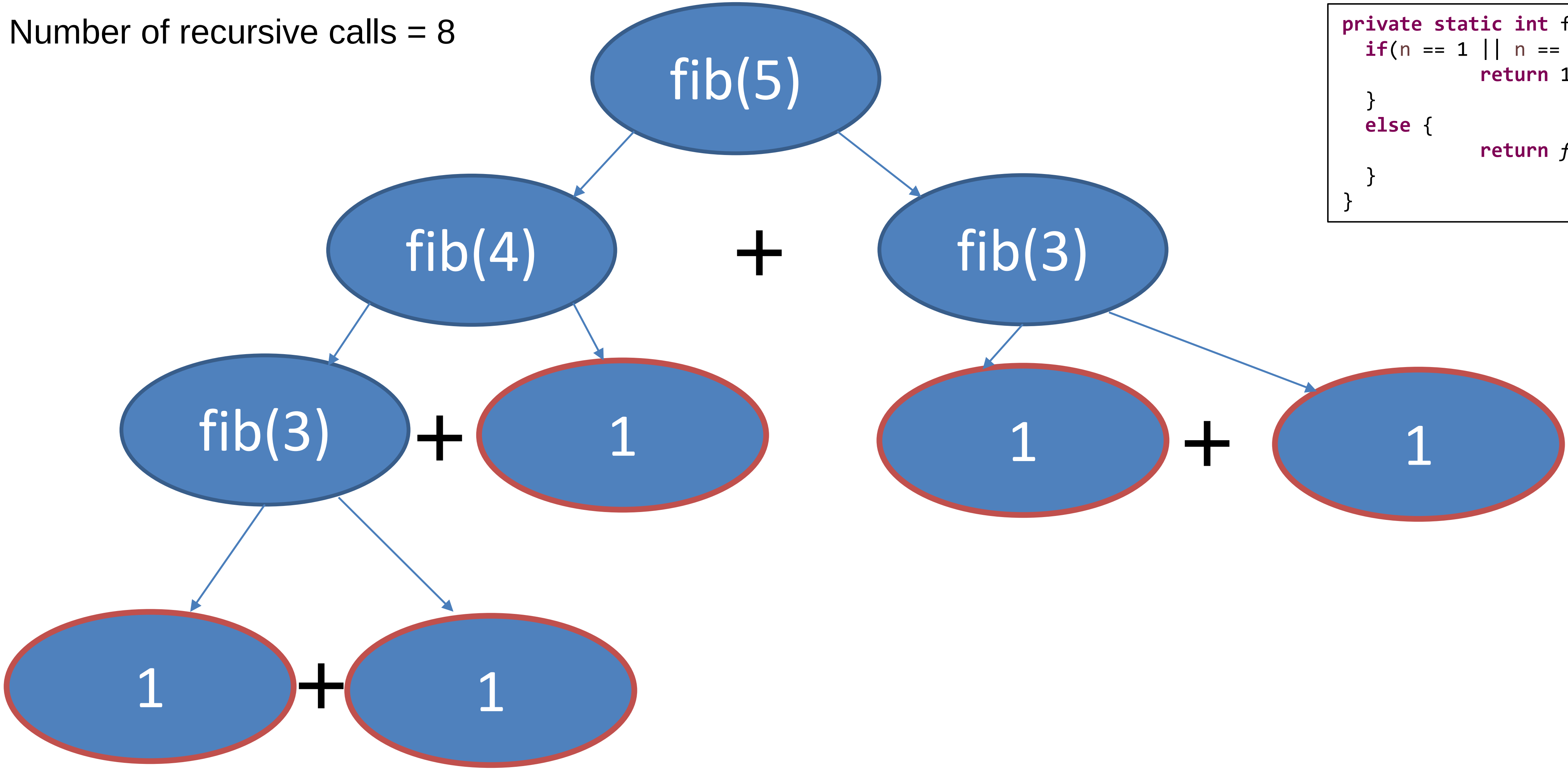

```java
private static int fib(int n) {
  if(n == 1 || n == 2) {
          return 1;
  }
  else {
          return fib(n-1) + fib(n-2);
  }
}
```

Number of recursive calls = 8



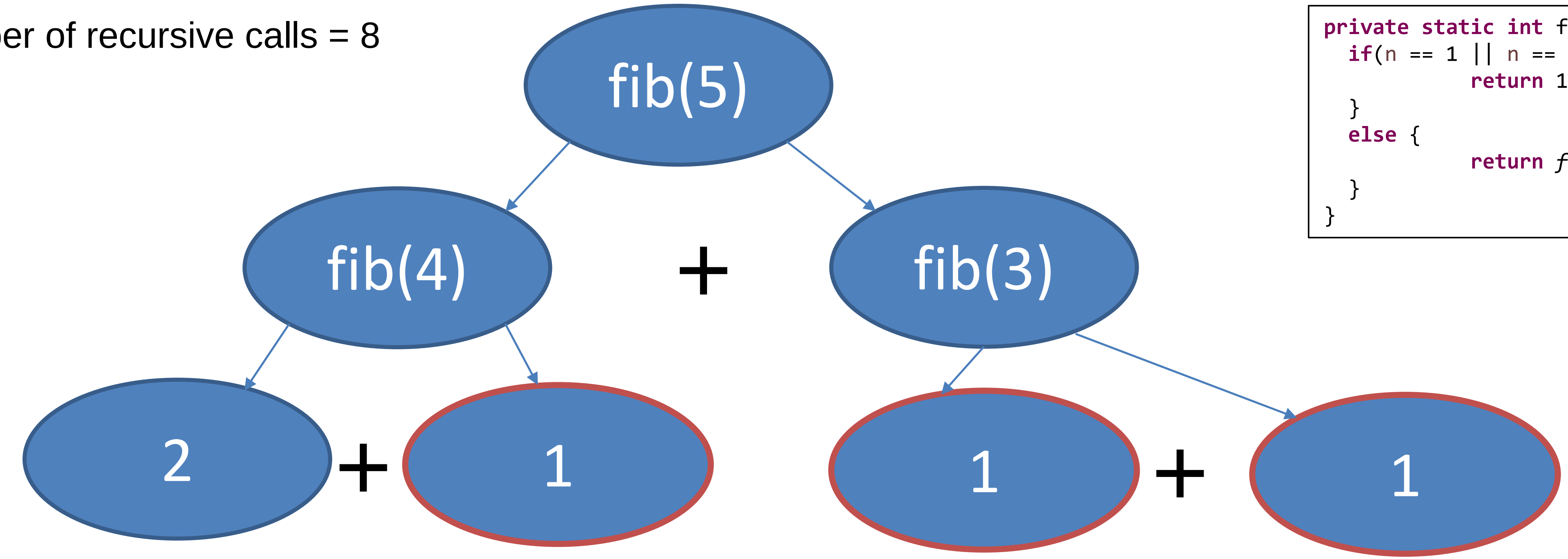

```java
private static int fib(int n) {
  if(n == 1 || n == 2) {
            return 1;
  }
  else {
            return fib(n-1) + fib(n-2);
  }
}
```

Number of recursive calls = 8



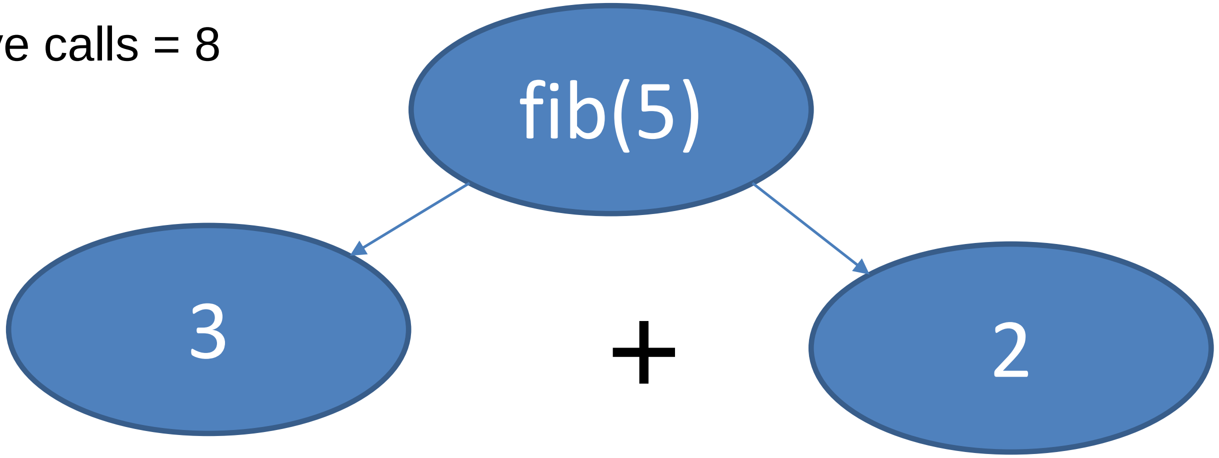

```
private static int fib(int n) {
  if(n == 1 || n == 2) {
            return 1;
  }
  else {
            return fib(n-1) + fib(n-2);
  }
}
```

Number of recursive calls = 8




```java
private static int fib(int n) {
  if(n == 1 || n == 2) {
            return 1;
  }
  else {
            return fib(n-1) + fib(n-2);
  }
}
```

Number of recursive calls = 8




```java
private static int fib(int n) {
  if(n == 1 || n == 2) {
          return 1;
  }
  else {
          return fib(n-1) + fib(n-2);
  }
}
```

Number of recursive calls = 8

5

Final answer!

```java
private static int fib(int n) {
  if(n == 1 || n == 2) {
		return 1;
  }
  else {
		return fib(n-1) + fib(n-2);
  }
}
```

```java
private static int fib(int n) {
  if(n == 1 || n == 2) {
      return 1;
  }
  else {
      return fib(n-1) + fib(n-2);
  }
}
```

Running Time?

```java
private static int fib(int n) {
  if(n == 1 || n == 2) { O(1)
      return 1; O(1)
  }
  else {         O(1)          O(1)
      return fib(n-1) + fib(n-2);
  }
}
```

Running Time?

O(1) ?

```
private static int fib(int n) {
  if(n == 1 || n == 2) { O(1)
      return 1; O(1)
  }
  else {          O(1)          O(1)
      return fib(n-1) + fib(n-2);
  }
}
```

Running Time?

~~**O(1) ?**~~

No!

When we are analyzing recursive algorithms, we have to calculate running time slightly different

```java
private static int fib(int n) {
  if(n == 1 || n == 2) {
      return 1;
  }
  else {
      return fib(n-1) + fib(n-2);
  }
}
```

Generally speaking, we can compute the running time of a recursive algorithm by using the following formula:

Running time = **# of recursive calls made** * **amount of work done in each call**

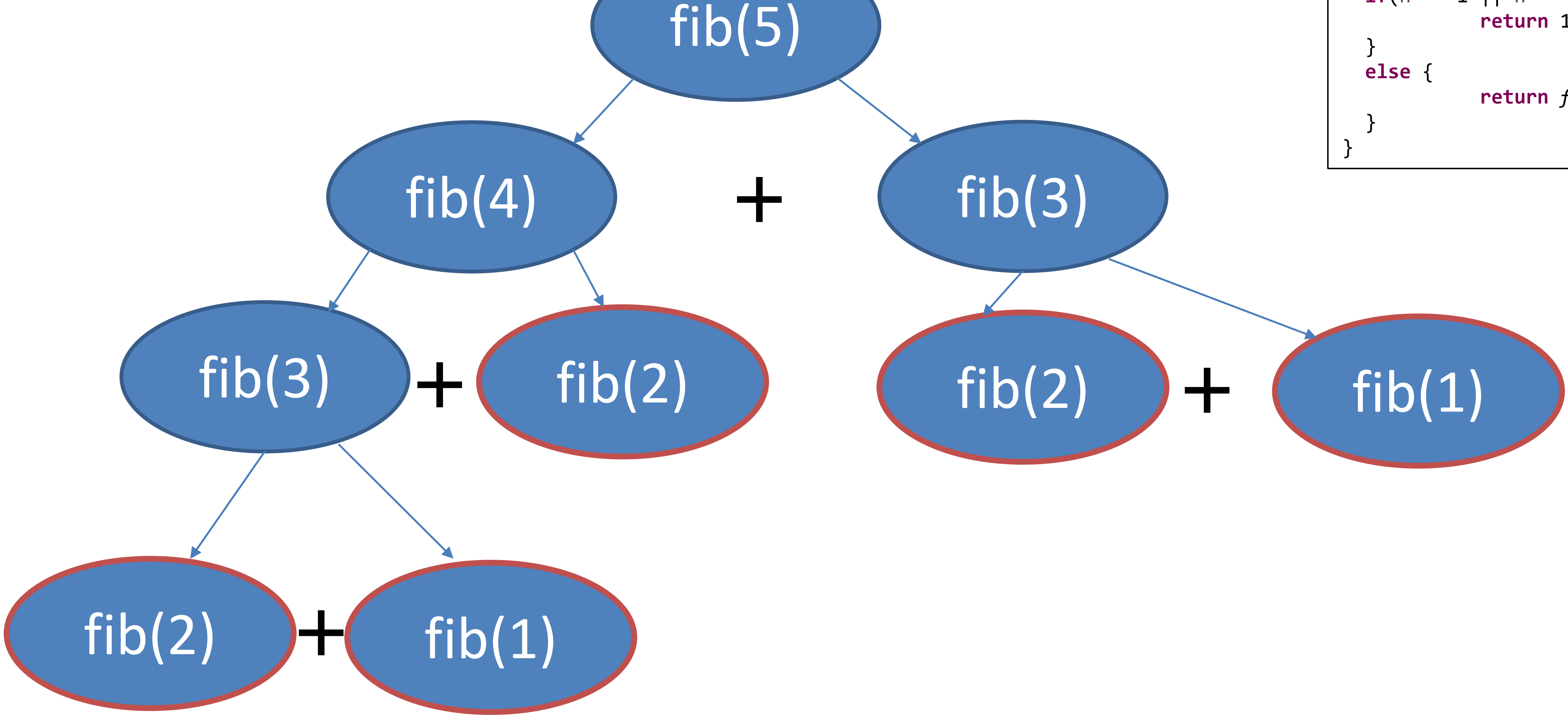
```
private static int fib(int n) {
  if(n == 1 || n == 2) { O(1)
      return 1;  O(1)
  }
  else {           O(1)            O(1)
      return fib(n-1) + fib(n-2);
  }
}
```

Generally speaking, we can compute the running time of a recursive algorithm by using the following formula:

Running time = **# of recursive calls made** * **amount of work done in each call**

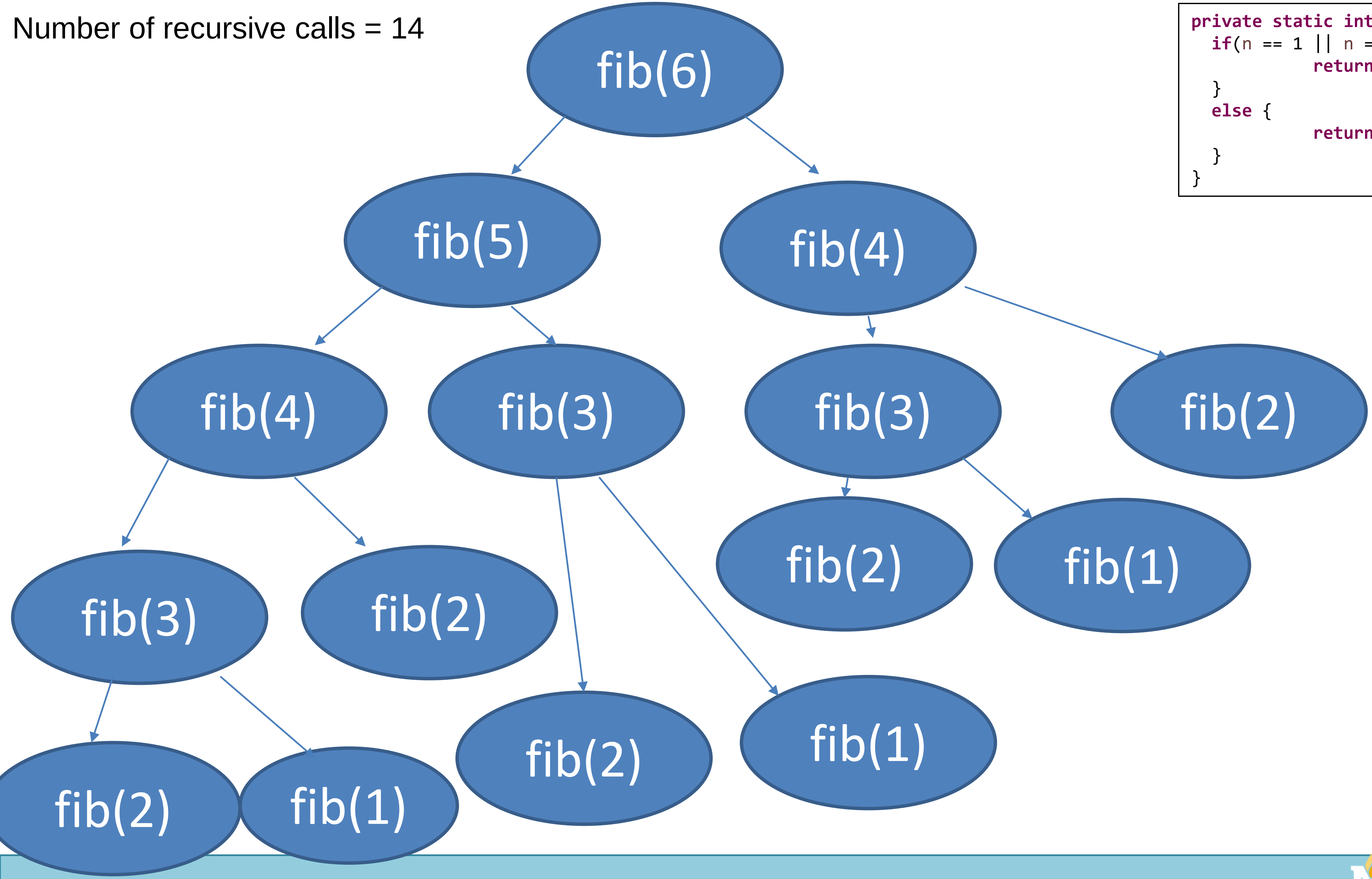
Running time = ??? * **O(1)**

Number of recursive calls = 8



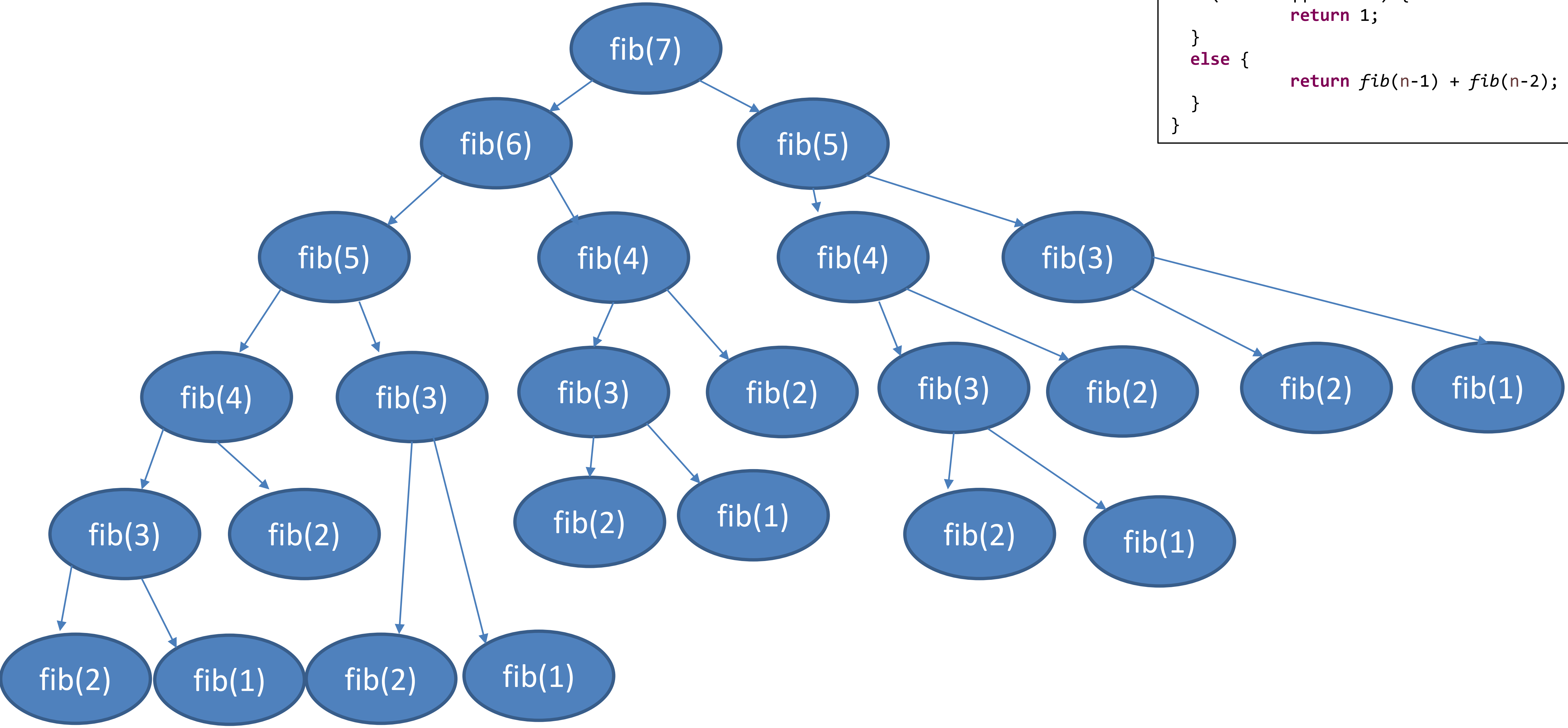

```
private static int fib(int n) {
  if(n == 1 || n == 2) {
            return 1;
  }
  else {
            return fib(n-1) + fib(n-2);
  }
}
```

Number of recursive calls = 14




```java
private static int fib(int n) {
  if(n == 1 || n == 2) {
            return 1;
  }
  else {
            return fib(n-1) + fib(n-2);
  }
}
```

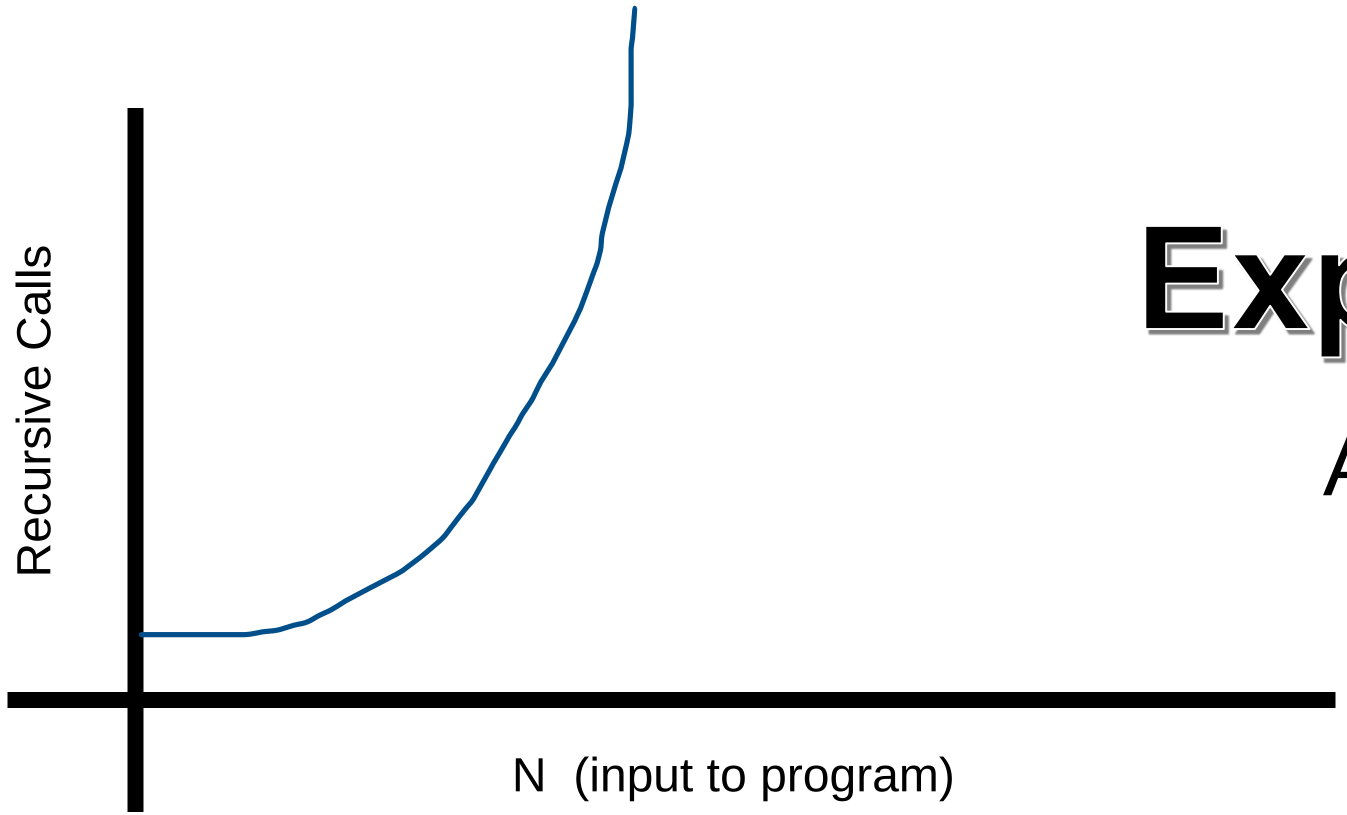
Number of recursive calls = 24




```java
private static int fib(int n) {
  if(n == 1 || n == 2) {
            return 1;
  }
  else {
            return fib(n-1) + fib(n-2);
  }
}
```

```java
private static int fib(int n) {
  if(n == 1 || n == 2) {
            return 1;
  }
  else {
            return fib(n-1) + fib(n-2);
  }
}
```

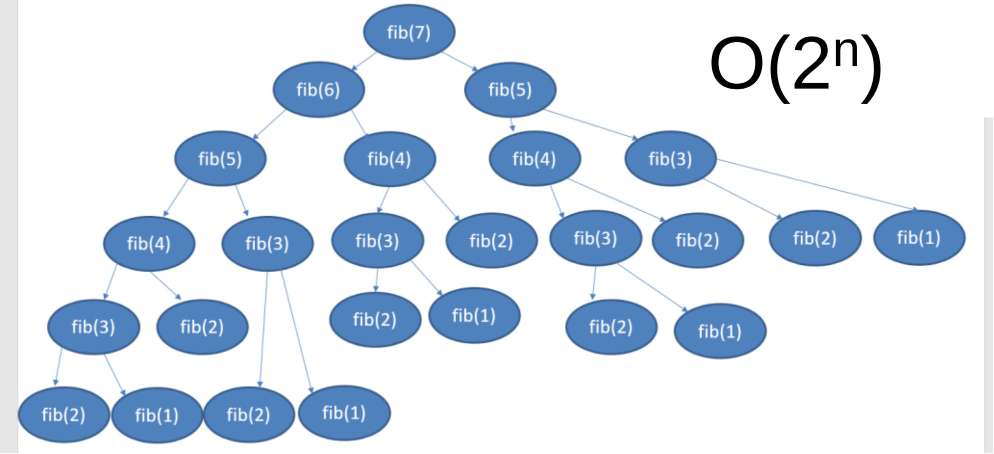
If we were to plot the number of recursive calls made as n increases, it would look something like his:



Exponential

Aka. $O(2^n)$

```java
private static int fib(int n) {
  if(n == 1 || n == 2) { O(1)
      return 1;  O(1)
  }
  else {          O(1)          O(1)
      return fib(n-1) + fib(n-2);
  }
}
```



$O(2^n)$

Generally speaking, we can compute the running time of a recursive algorithm by using the following formula:

Running time = **# of recursive calls made** * **amount of work done in each call**

Running time = $O(2^n)$ * $O(1)$

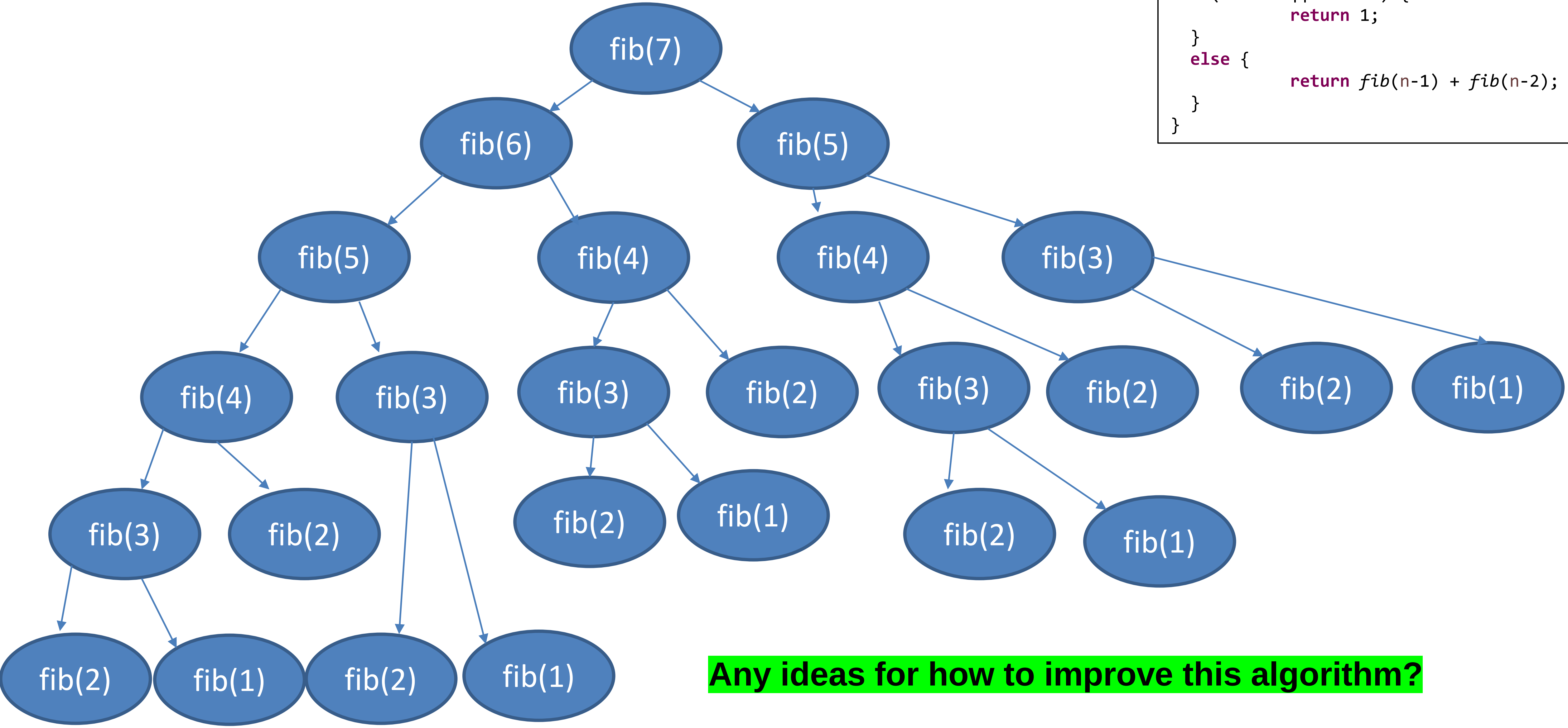
Total running time = $O(2^n)$

n = requested Fibonacci digit

$O(2^n)$ is very bad…

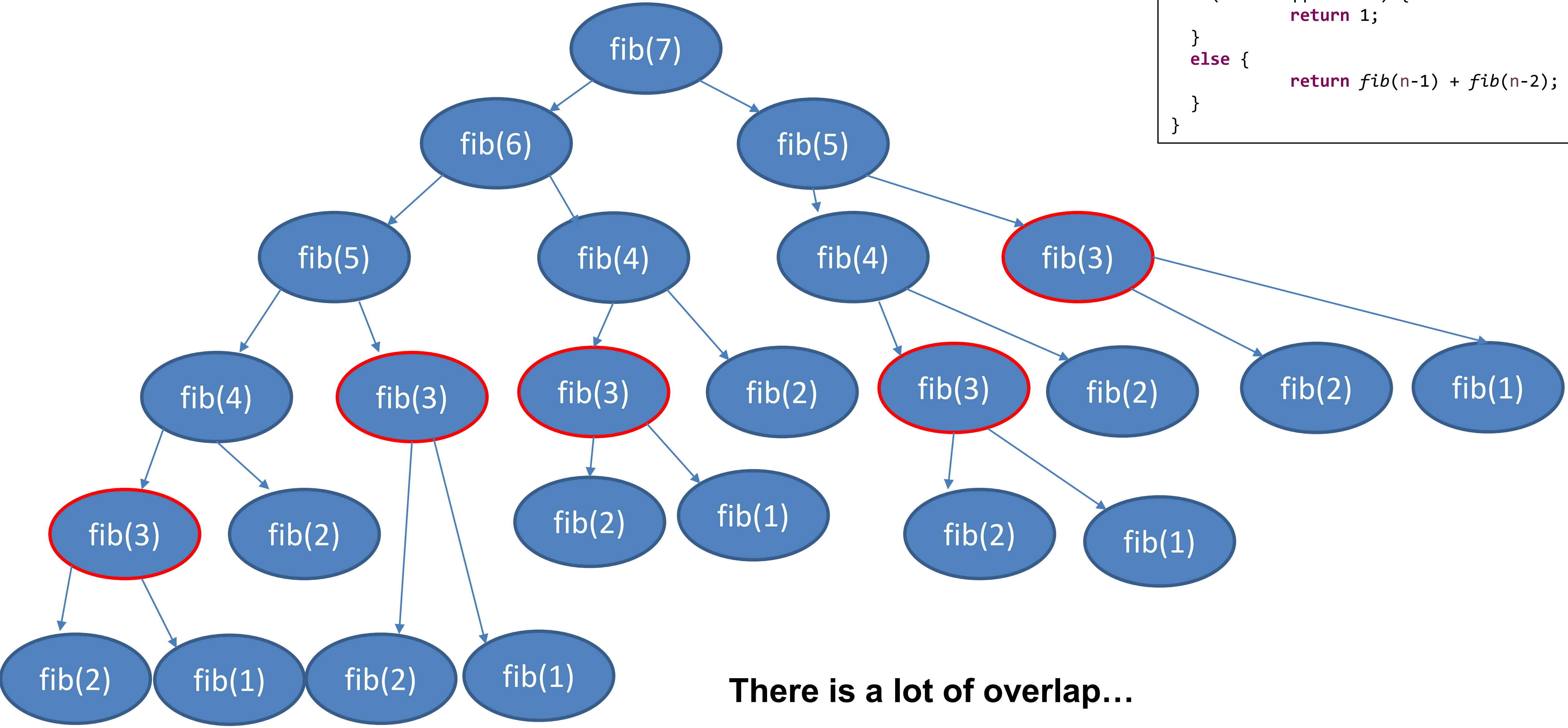
Number of recursive calls = 24

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



Any ideas for how to improve this algorithm?

Number of recursive calls = 24



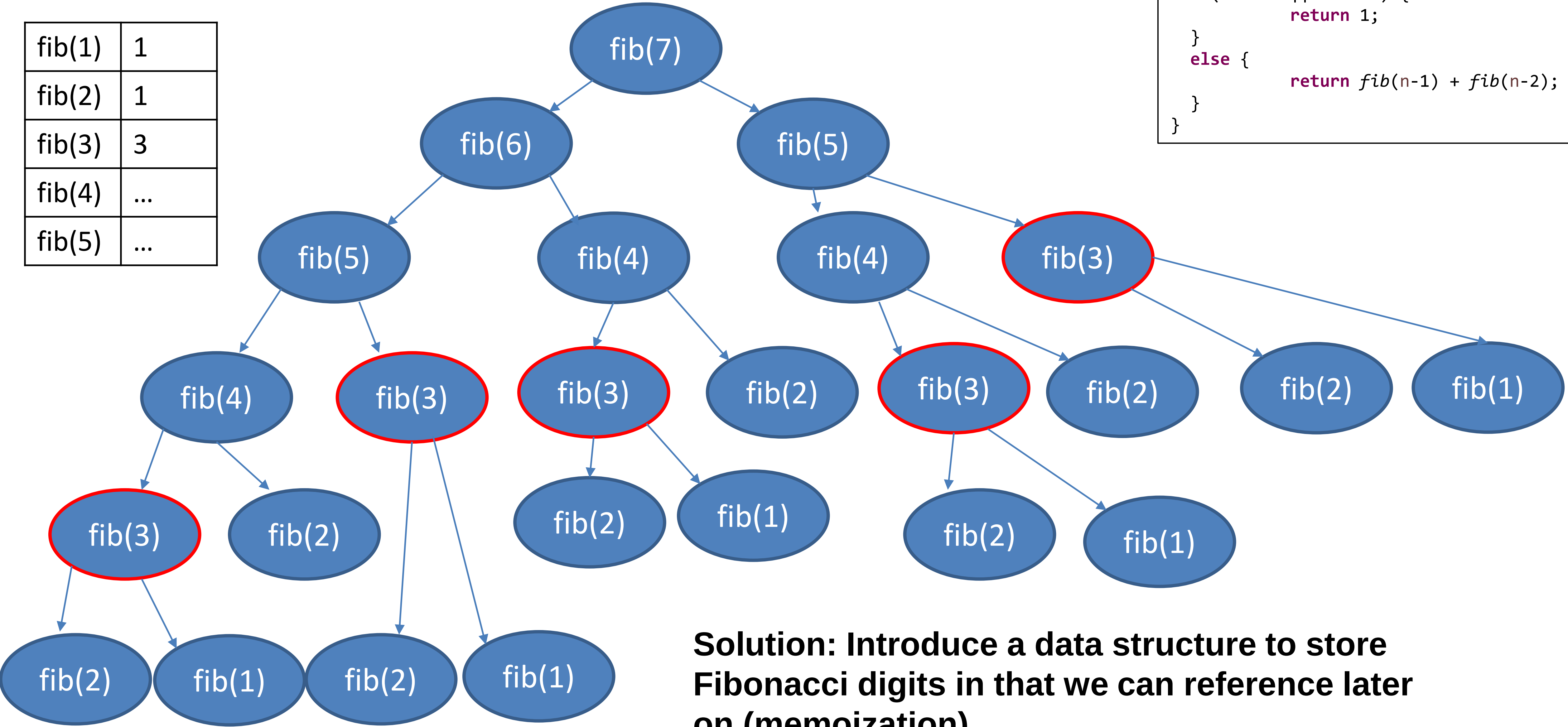

```java
private static int fib(int n) {
  if(n == 1 || n == 2) {
            return 1;
  }
  else {
            return fib(n-1) + fib(n-2);
  }
}
```

There is a lot of overlap…

Number of recursive calls = 24

fib(1)	1
fib(2)	1
fib(3)	3
fib(4)	…
fib(5)	…

```java
private static int fib(int n) {
  if(n == 1 || n == 2) {
            return 1;
  }
  else {
            return fib(n-1) + fib(n-2);
  }
}
```



Solution: Introduce a data structure to store Fibonacci digits in that we can reference later on (memoization)

(These lookups happen in constant time!)

Limitations of recursion?

Bubble Sorting Recursively

2	5	1	3	4

2	1	3	4	5

1	2	3	4	5

1	2	3	4	5

1	2	3	4	5

Bubble sort can be solved be solving smaller instances of bubble sort!

Base Case:

If the unsorted portion of the array is of size 1, return current array

Recursive Case:

Do one iteration of bubble sort, recursively call method and pass smaller array

```java
static void bubbleSort(int arr[], int n)
   {
       // Base case
       if (n == 1)
           return;

        int count = 0;
        for (int i=0; i<n-1; i++)
           if (arr[i] > arr[i+1])
           {
               // swap
               int temp = arr[i];
               arr[i] = arr[i+1];
               arr[i+1] = temp;
               count = count+1;
           }


        if (count == 0)
           return;


        bubbleSort(arr, n-1);
   }
```

Recursive case