CSCI 132:
Basic Data Structures and Algorithms

Recursion (Part 1)

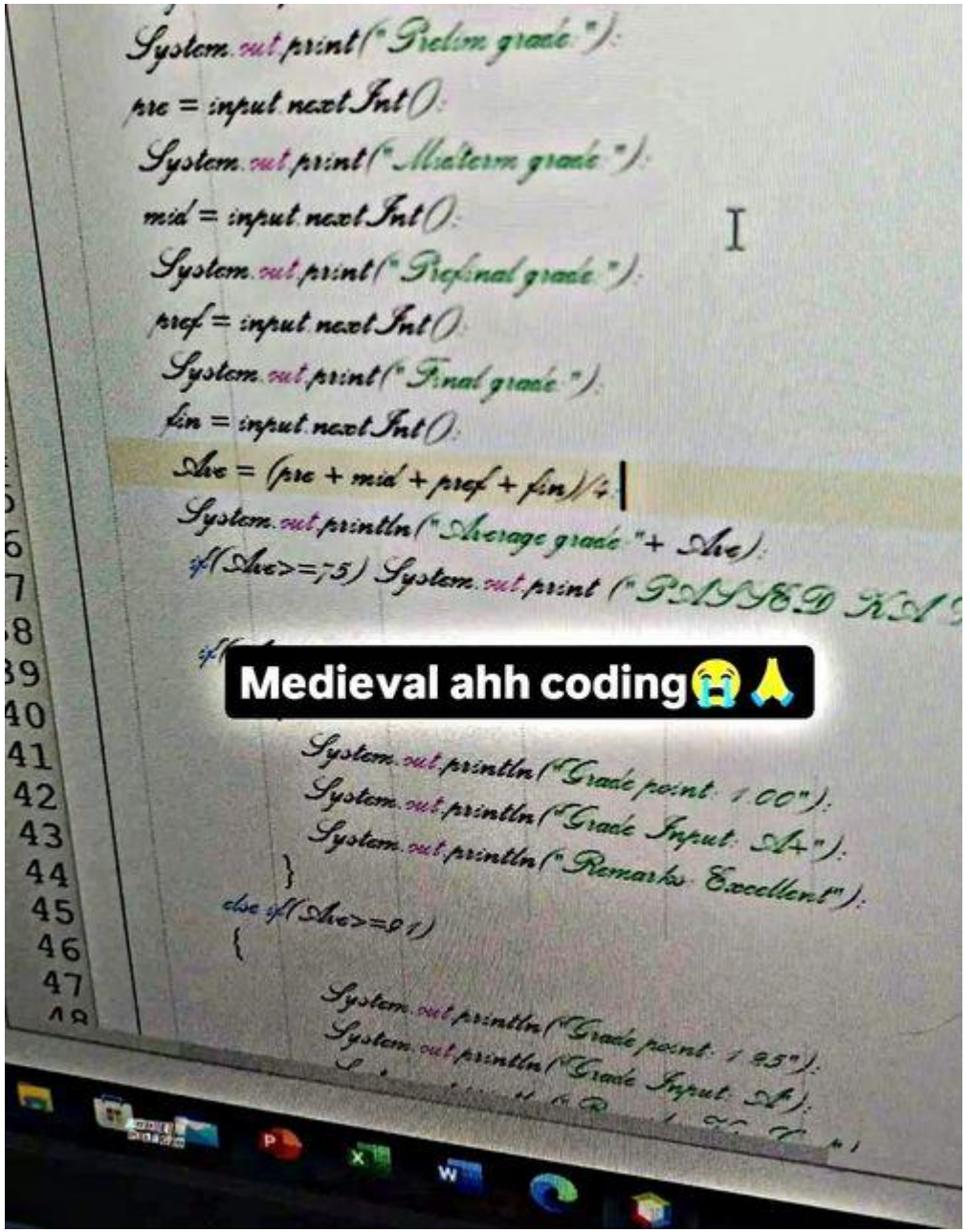
Reese Pearsall

Spring 2025

Announcements

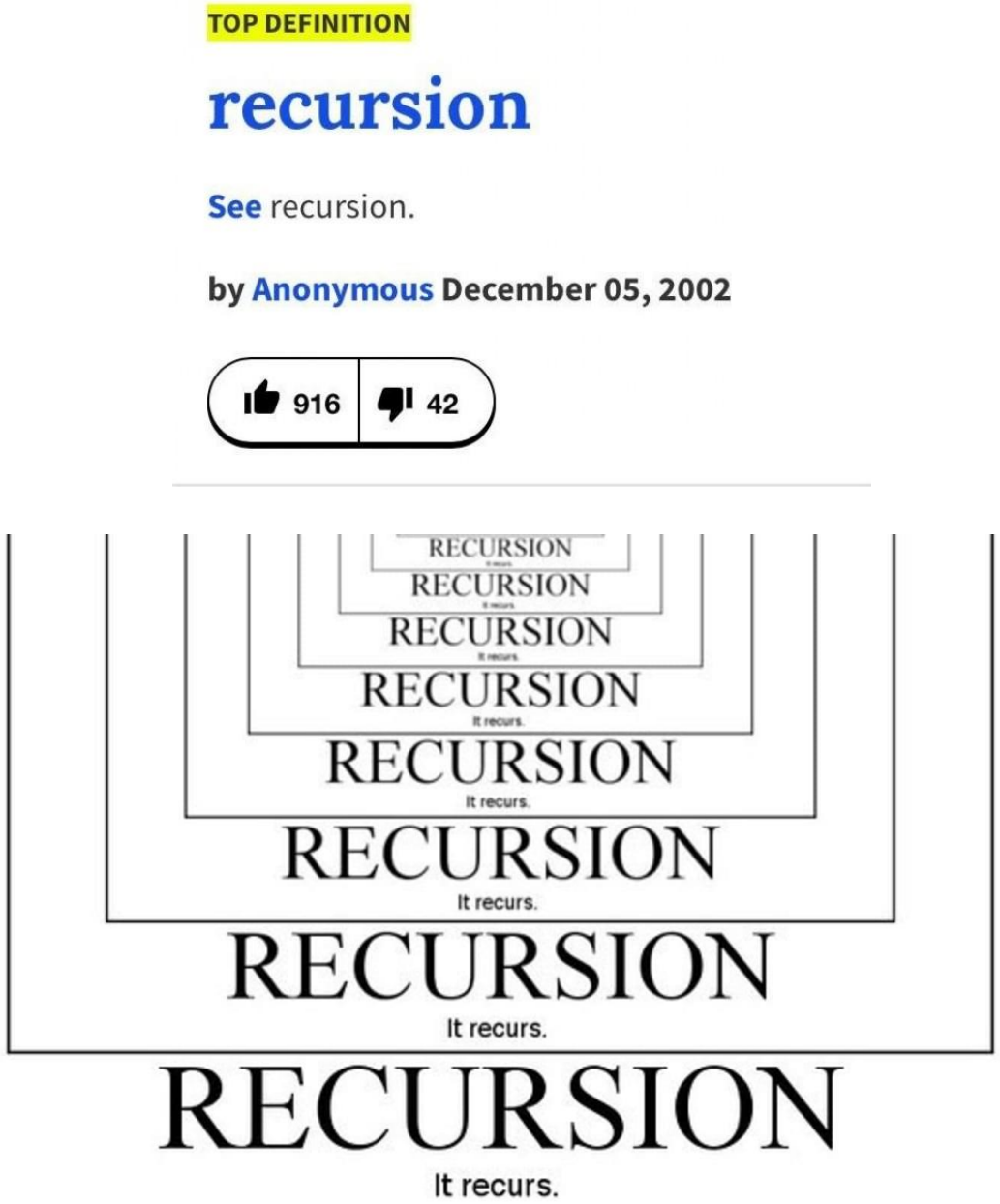
Program 3 due Friday

Friday will be a help session for program 3



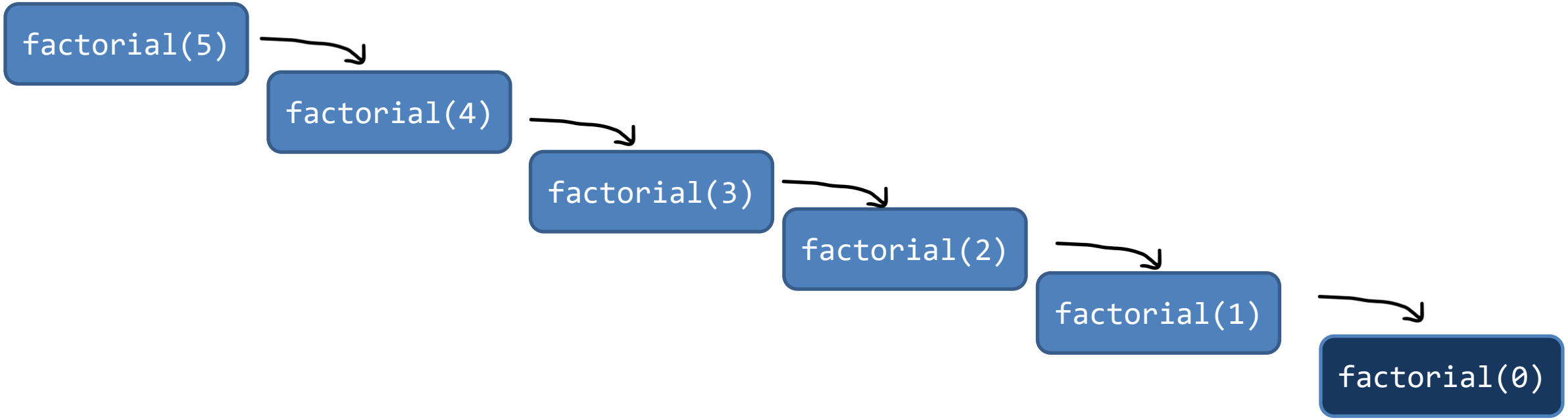
Recursion is a problem-solving technique that involves a *method calling itself* to solve some smaller problem

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

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



```
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) !

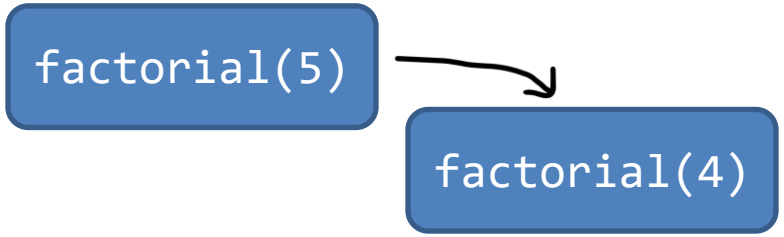
factorial(5)

```
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) !

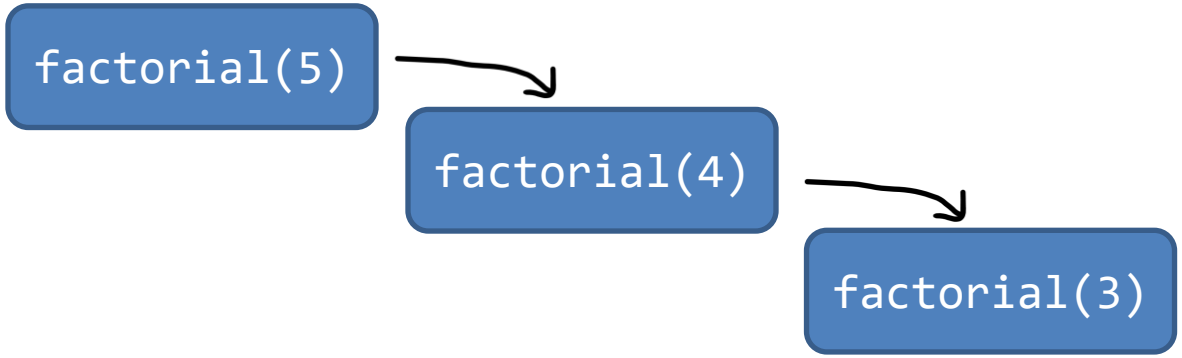


```
static int factorial(int n)
    {
        if (n == 0)
            return 1;

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

(base case)

(recursive case)

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

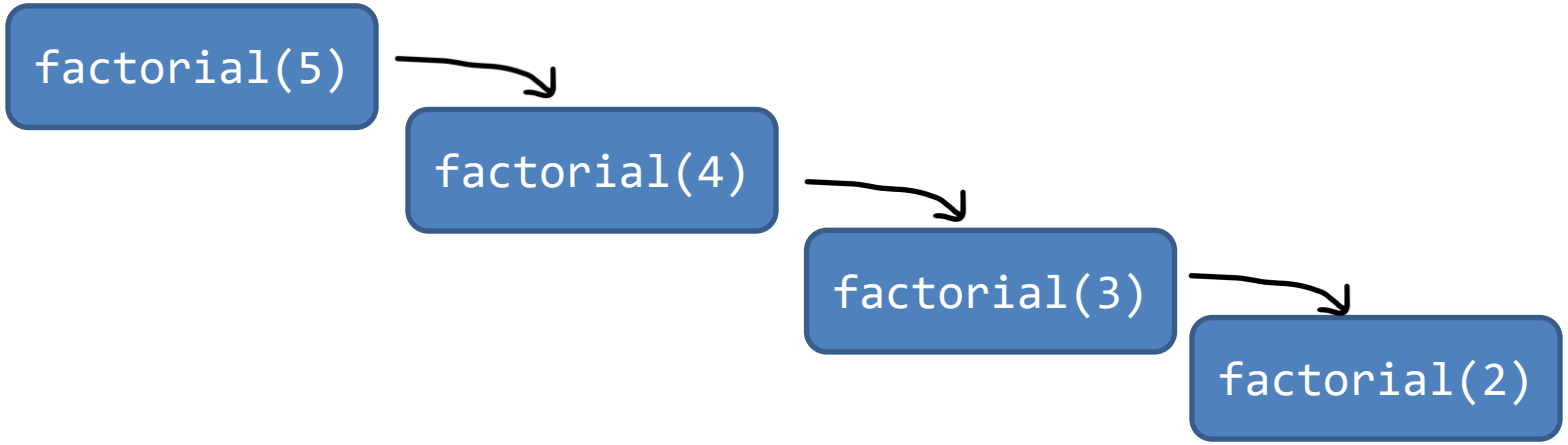


```
static int factorial(int n)
    {
        if (n == 0)
            return 1;

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

(base case)

(recursive case)

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

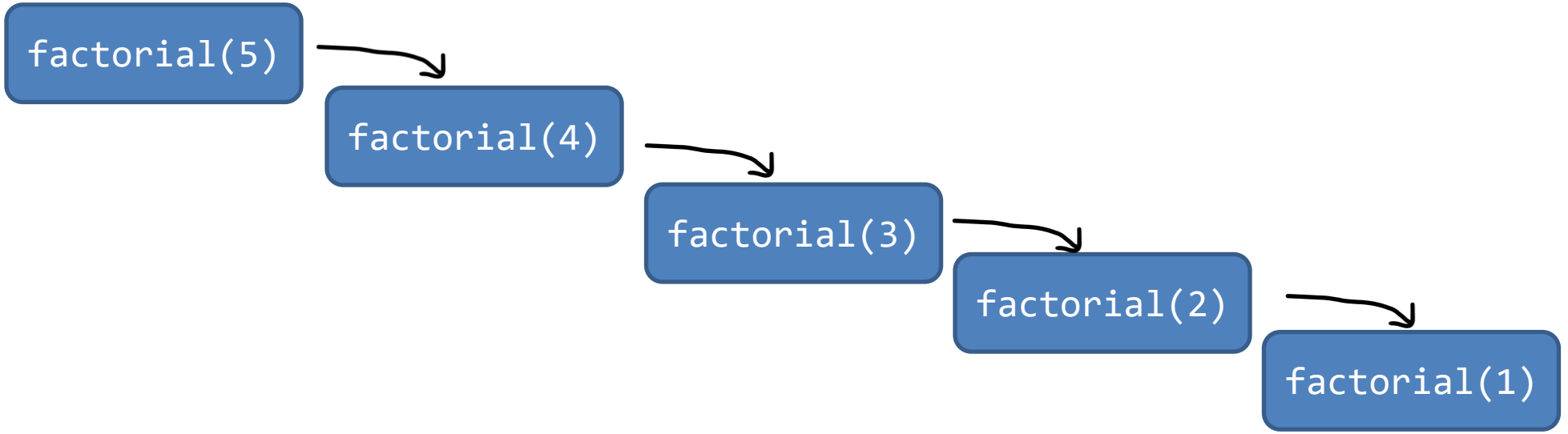


```
static int factorial(int n)
    {
        if (n == 0)
            return 1;

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

(base case)

(recursive case)

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

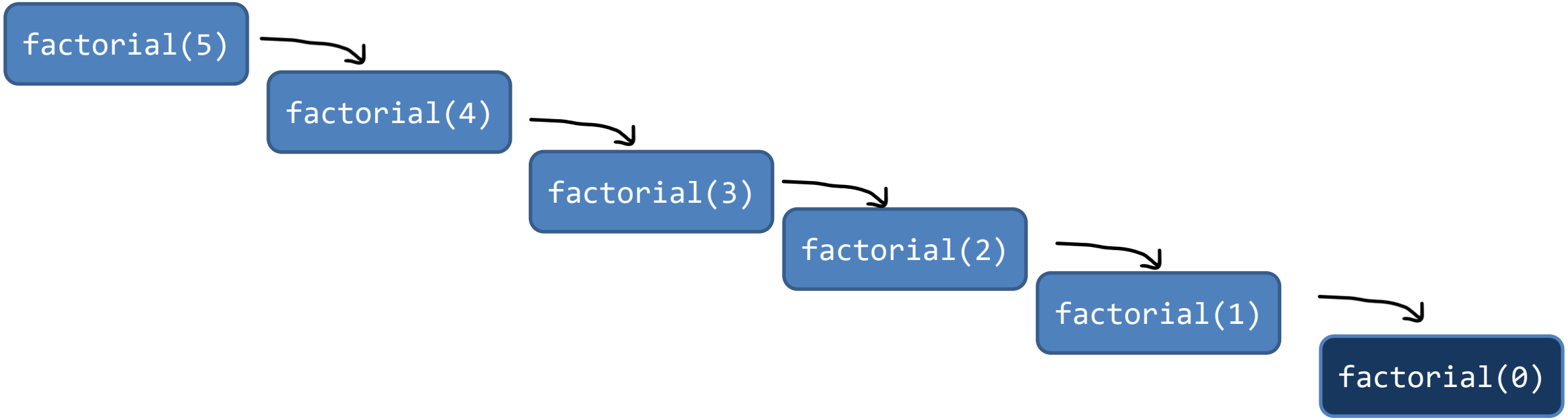


```
static int factorial(int n)
    {
        if (n == 0)
            return 1;

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

(base case)

(recursive case)

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

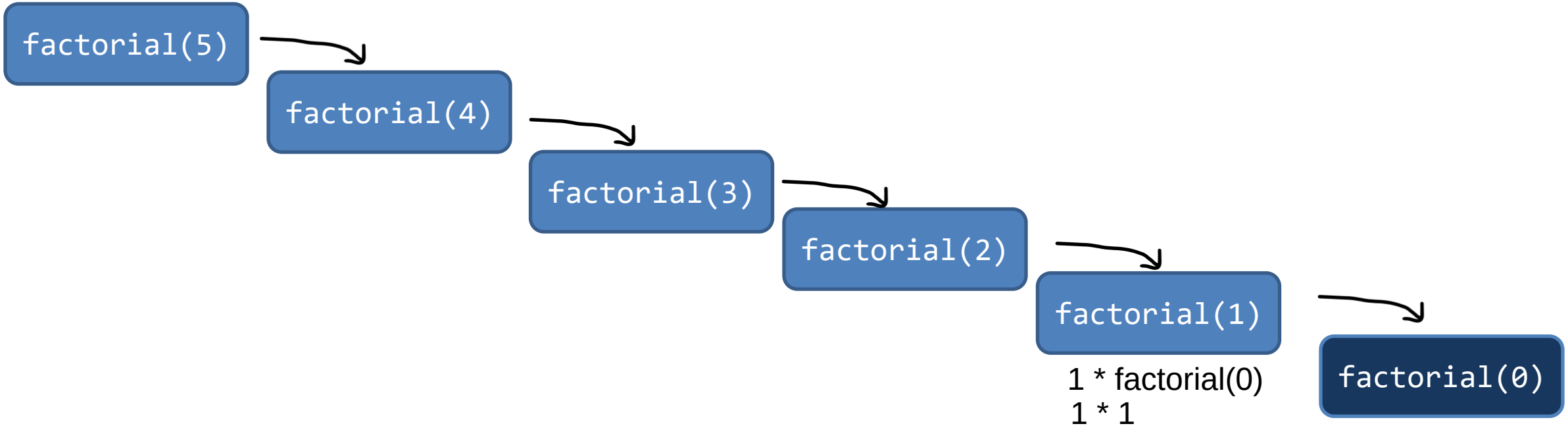


```
static int factorial(int n)
    {
        if (n == 0)
            return 1;

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

(base case)

(recursive case)

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

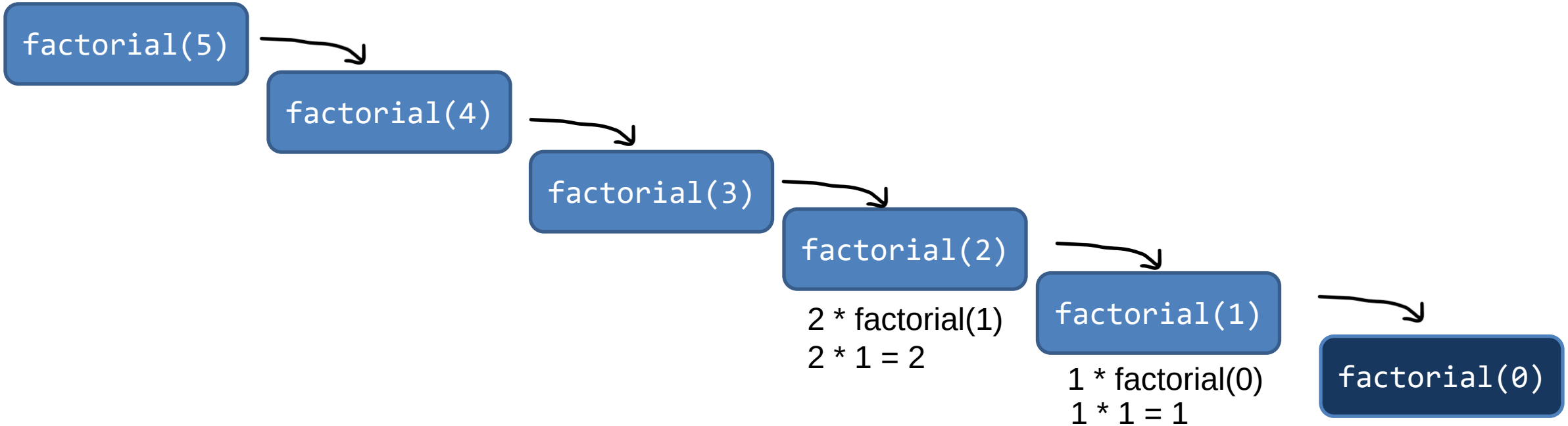


```
static int factorial(int n)
    {
        if (n == 0)
            return 1;

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

(base case)

(recursive case)

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

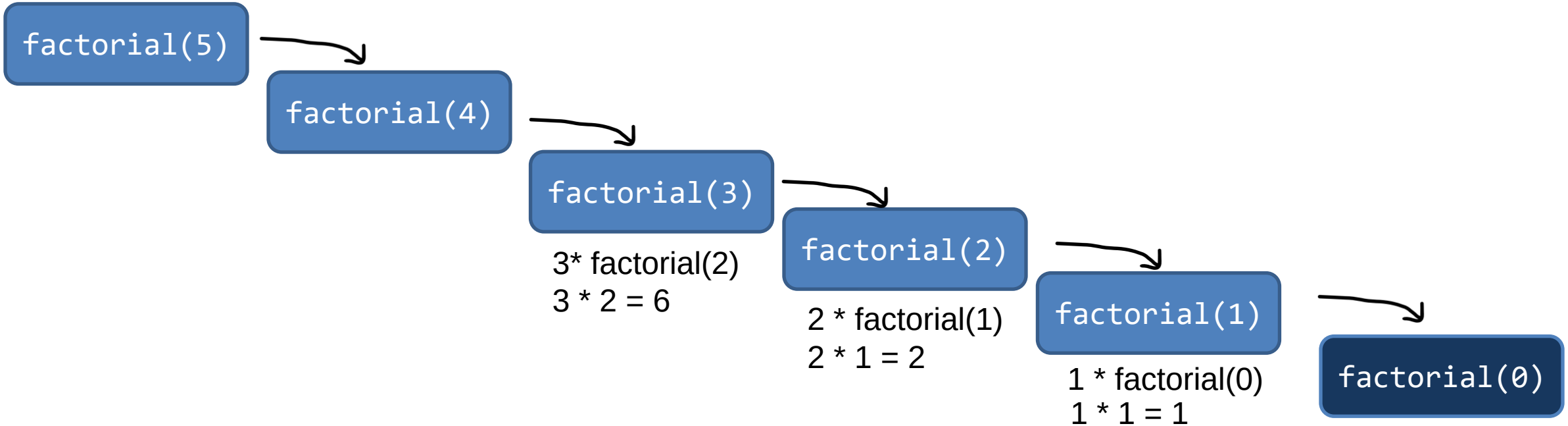


```
static int factorial(int n)
    {
        if (n == 0)
            return 1;

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

(base case)

(recursive case)

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

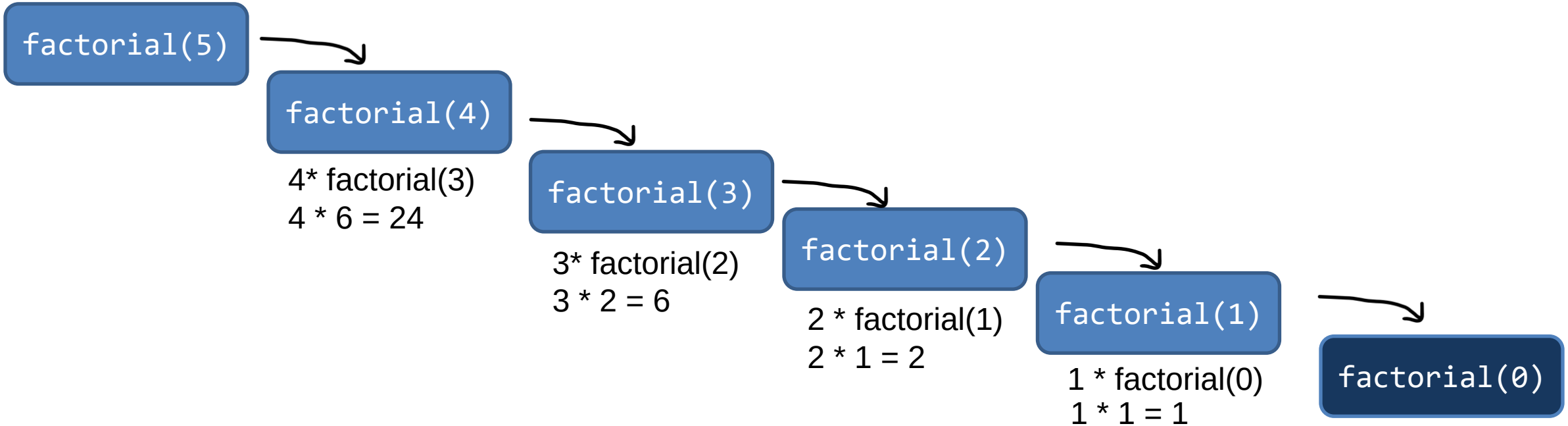


```
static int factorial(int n)
    {
        if (n == 0)
            return 1;

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

(base case)

(recursive case)

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

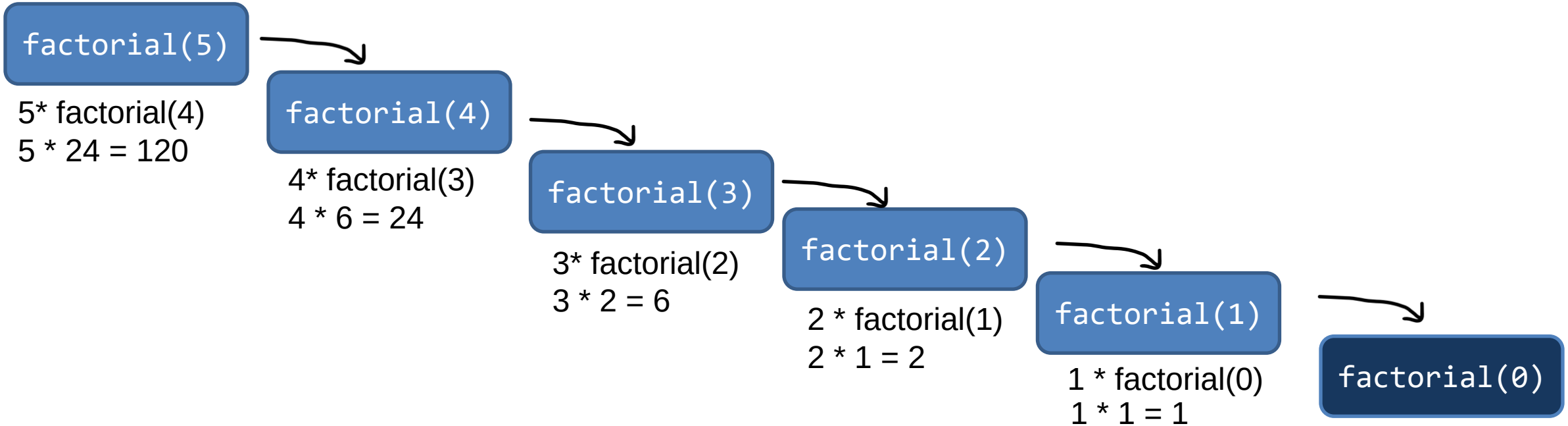


```
static int factorial(int n)
    {
        if (n == 0)
            return 1;

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

(base case)

(recursive case)

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

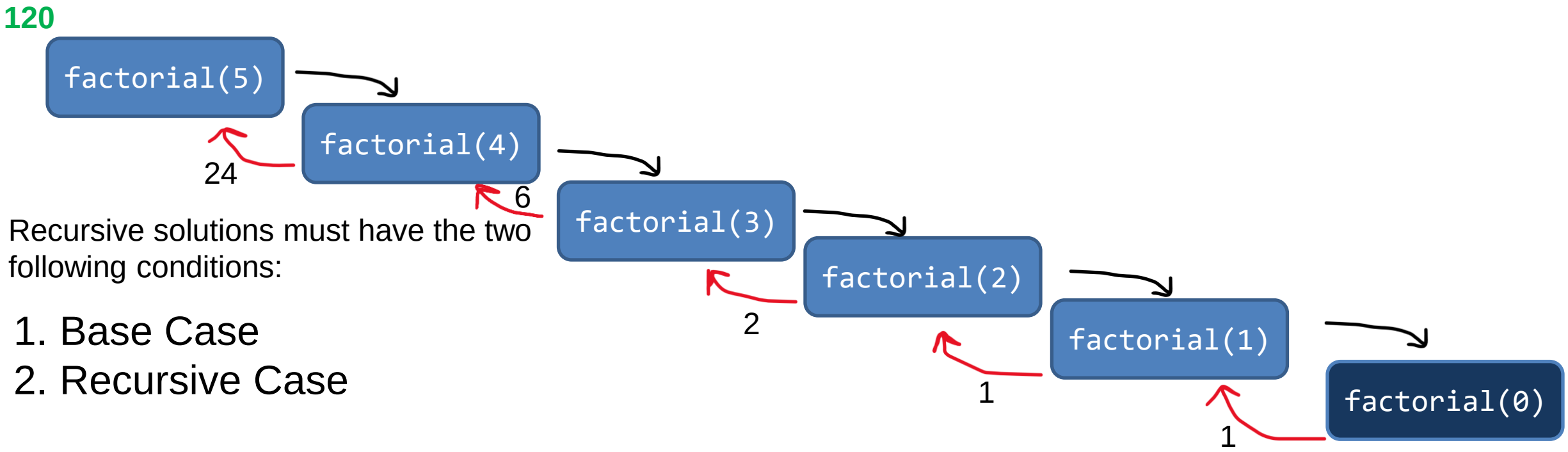


```
static int factorial(int n)
    {
        if (n == 0)
            return 1;

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

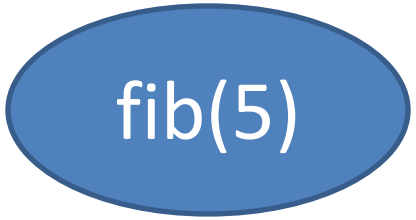
1+1=2
1+2=3
2+3=5
3+5=8
5+8=13
8+13=21
13+21=34
21+34=55
34+55=89
55+89=144
89+144=233
144+233=377

Base Case?

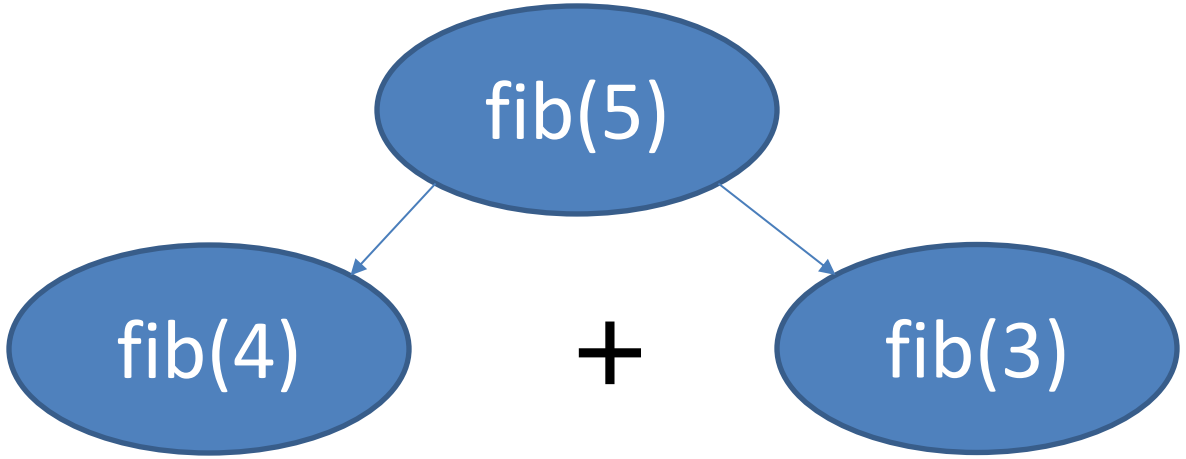
If finding the 1^{st} or 2^{nd} digit, return 1

Recursive Case?

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

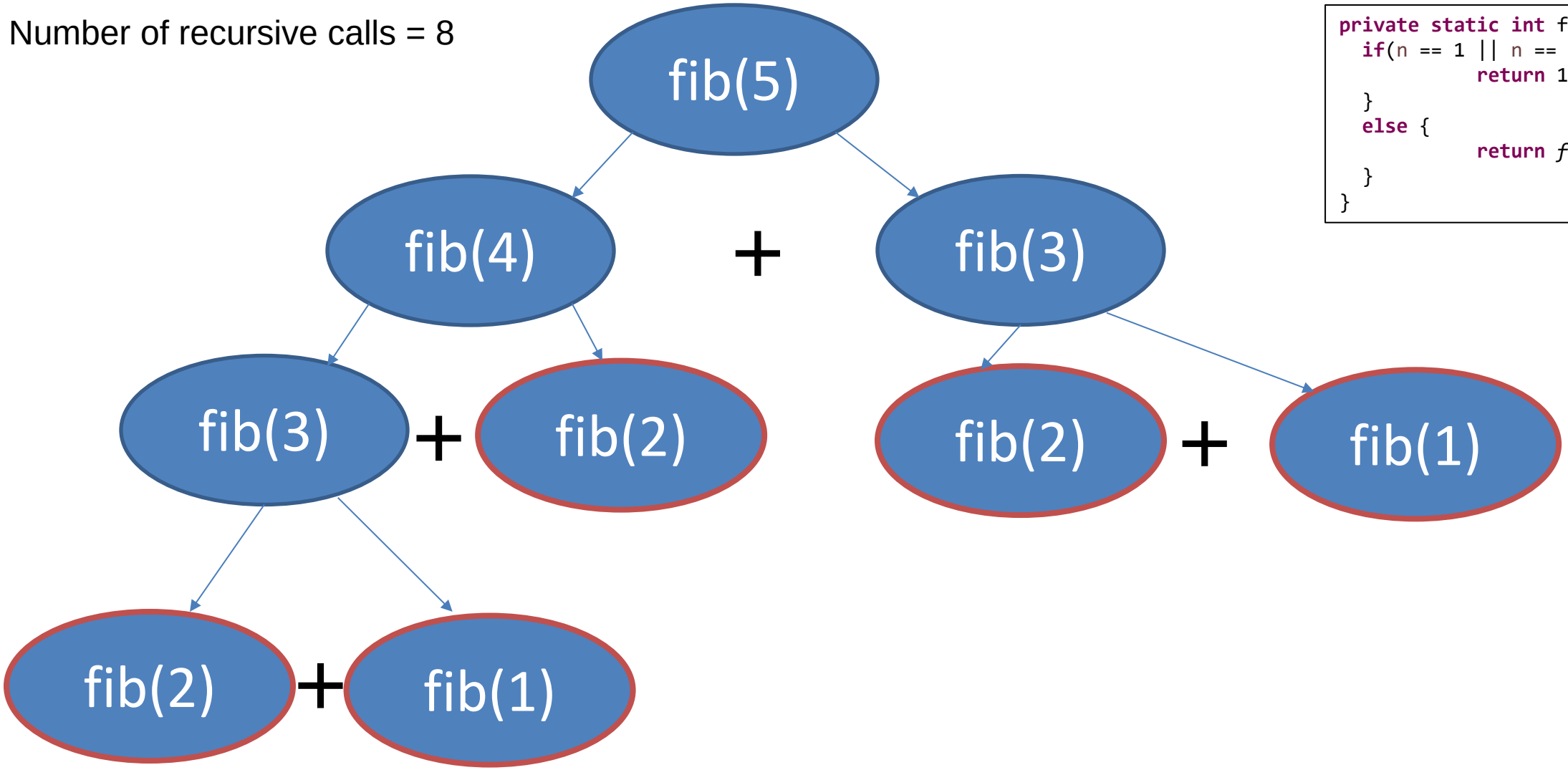



```
private static int fib(int n) {
  if(n == 1 || n == 2) {
            return 1;
  }
  else {
            return fib(n-1) + fib(n-2);
  }
}
```



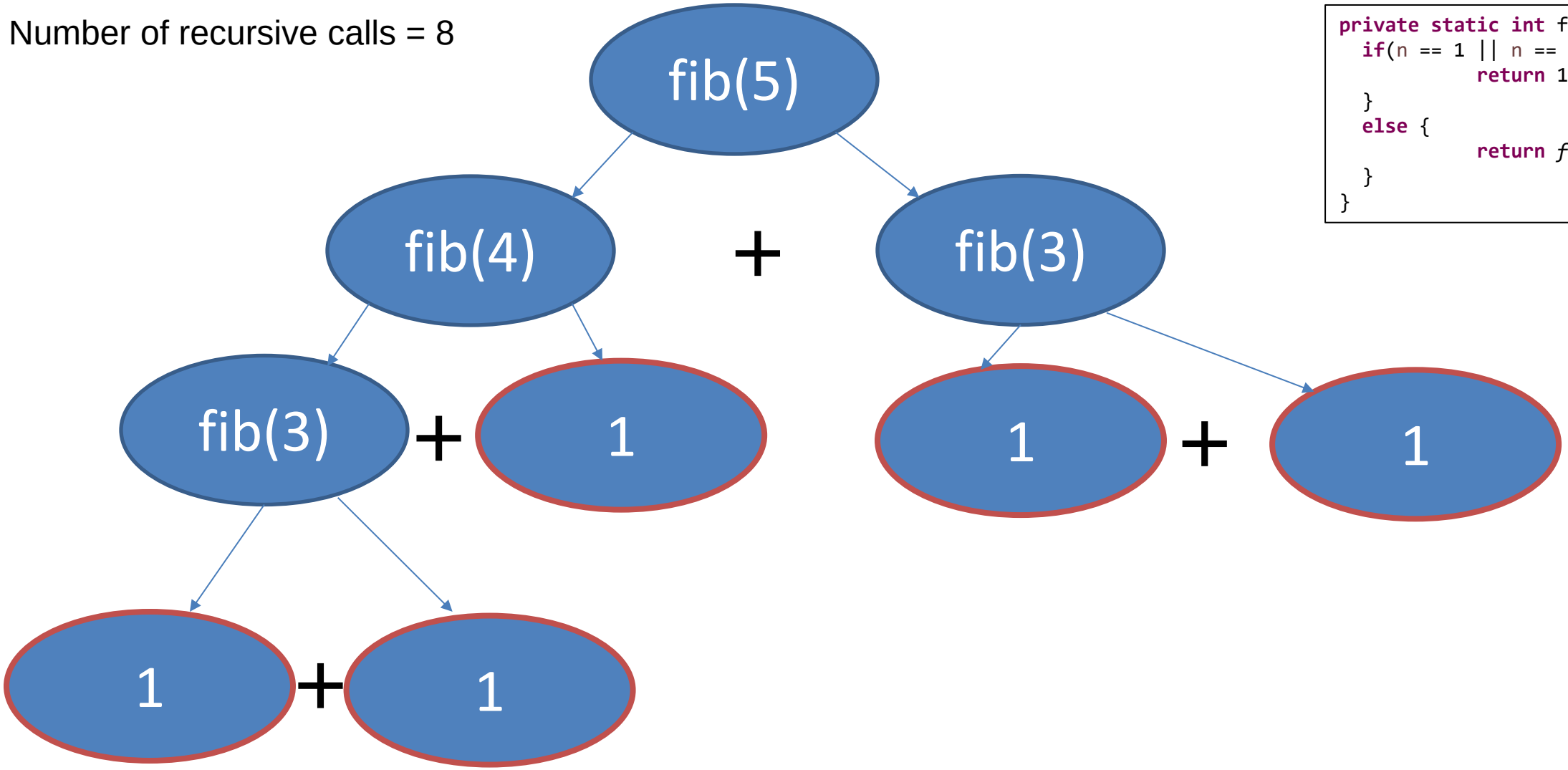

```
private static int fib(int n) {
  if(n == 1 || n == 2) {
            return 1;
  }
  else {
            return fib(n-1) + fib(n-2);
  }
}
```

Number of recursive calls = 8



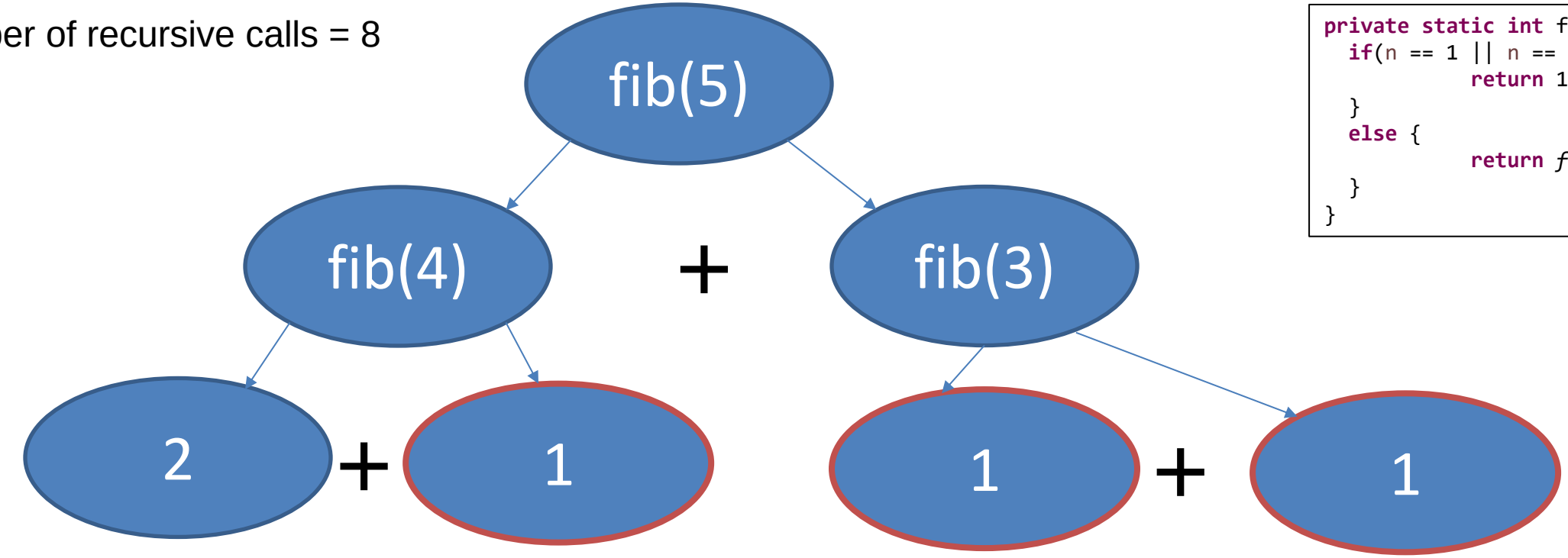

```
private static int fib(int n) {
  if(n == 1 || n == 2) {
            return 1;
  }
  else {
            return fib(n-1) + fib(n-2);
  }
}
```

Number of recursive calls = 8



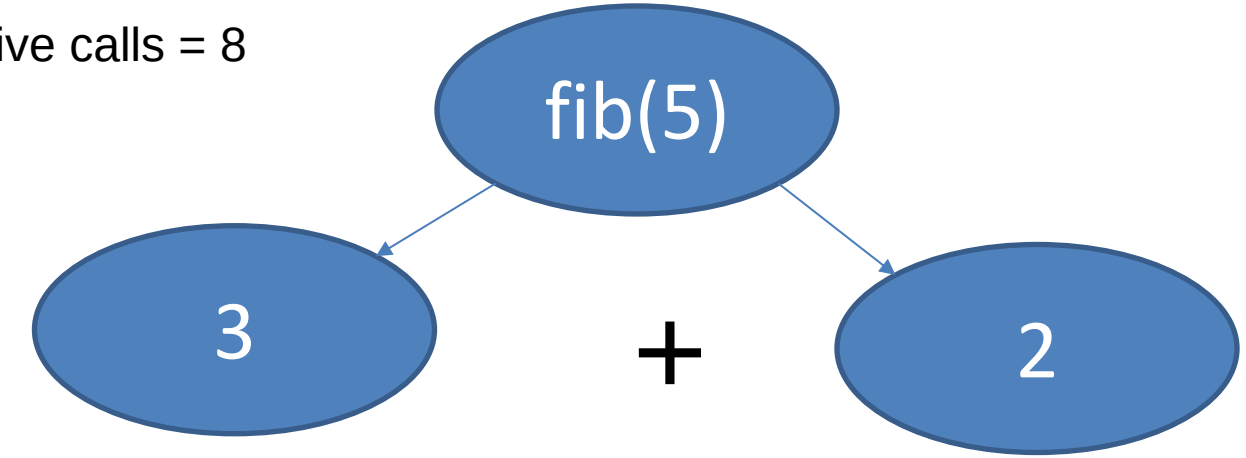

```
private static int fib(int n) {
  if(n == 1 || n == 2) {
            return 1;
  }
  else {
            return fib(n-1) + fib(n-2);
  }
}
```

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




```
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

Final answer!

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

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

Running Time?

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

O(1) O(1) O(1)

Running Time?

O(1) ?

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

O(1)

O(1)

O(1)

Running Time?

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

No!

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

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

O(1)

O(1)

O(1)

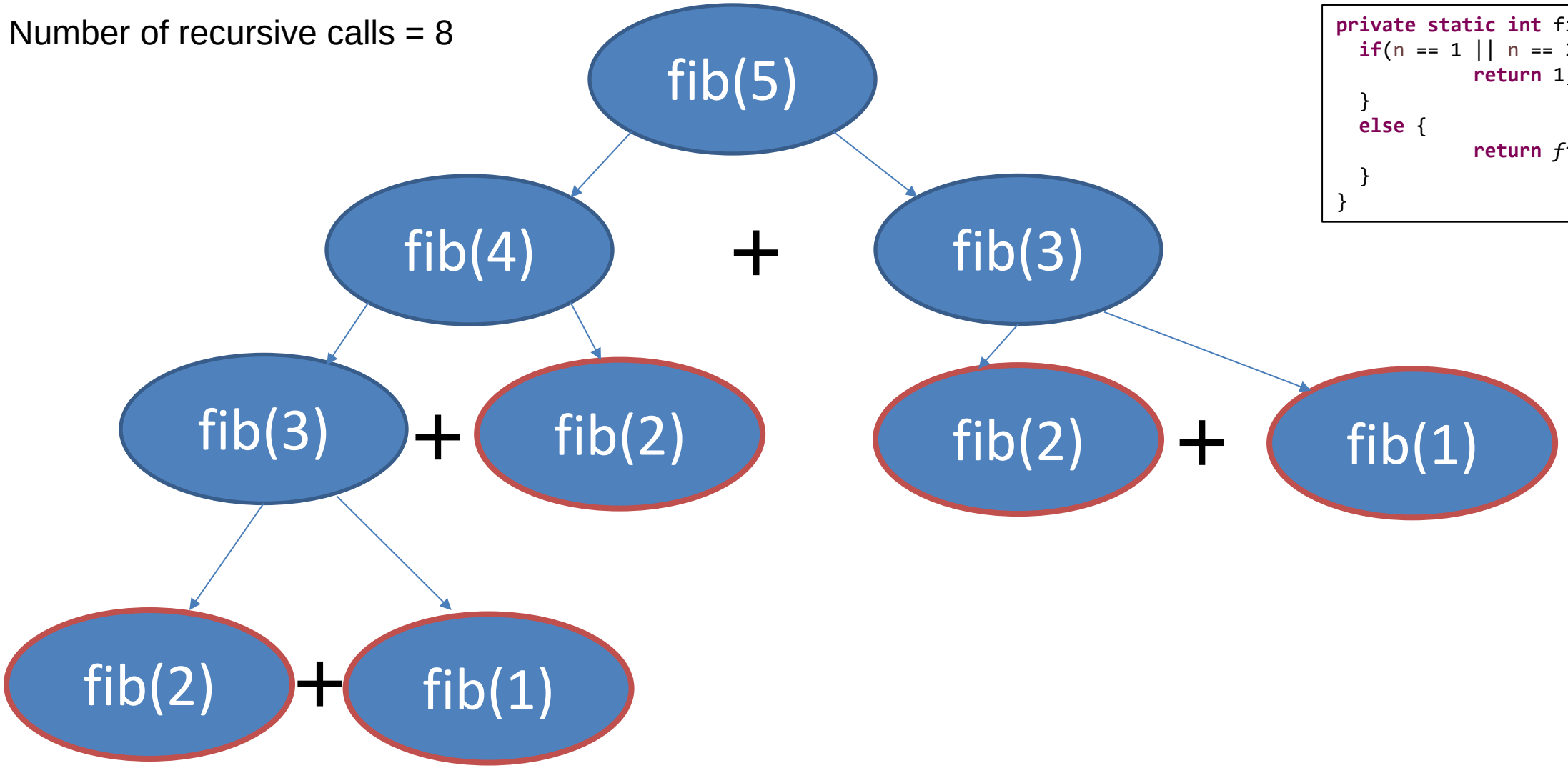
O(1)

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**

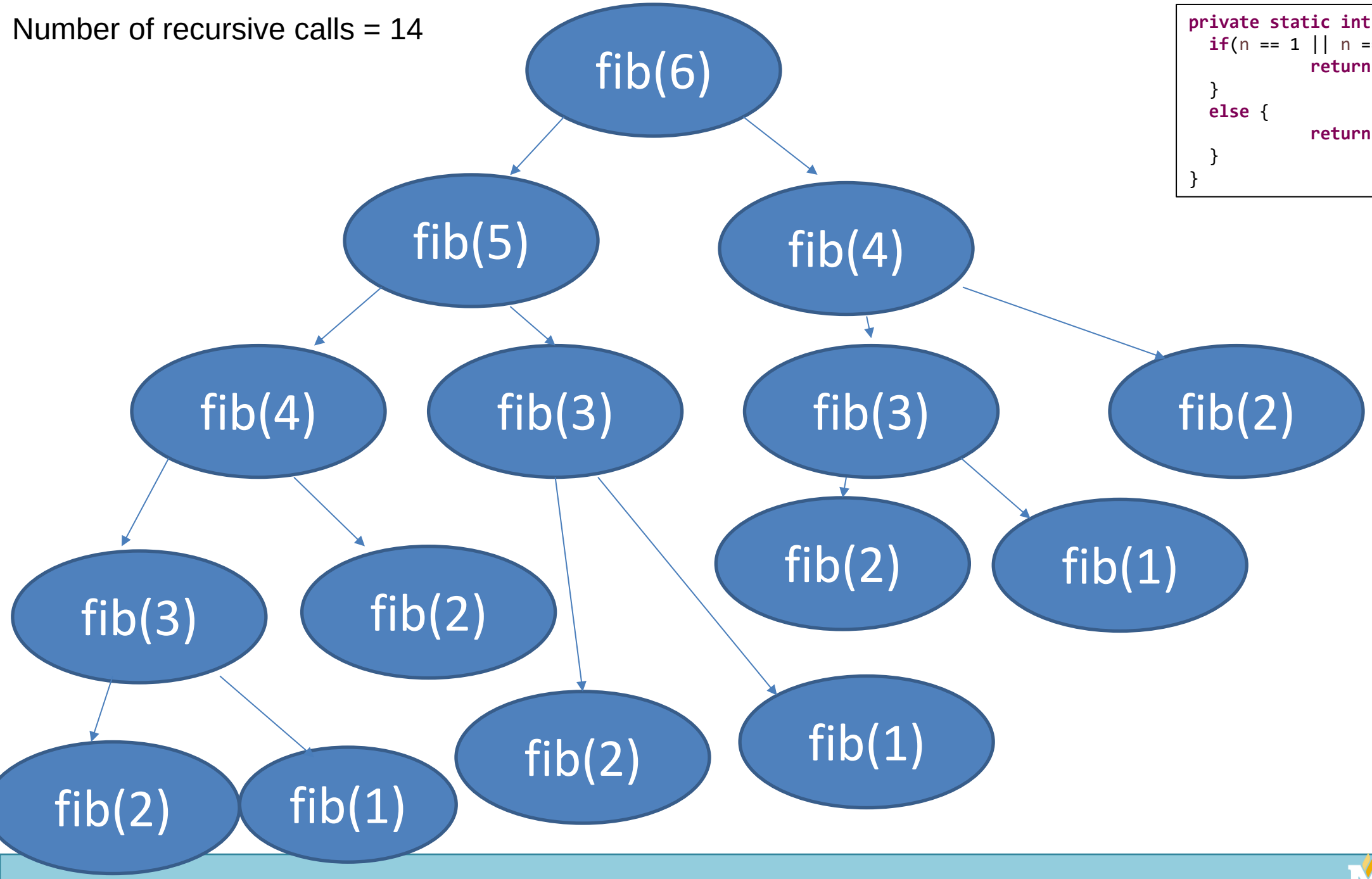
Running time = ??? * **O(1)**

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);
  }
}
```

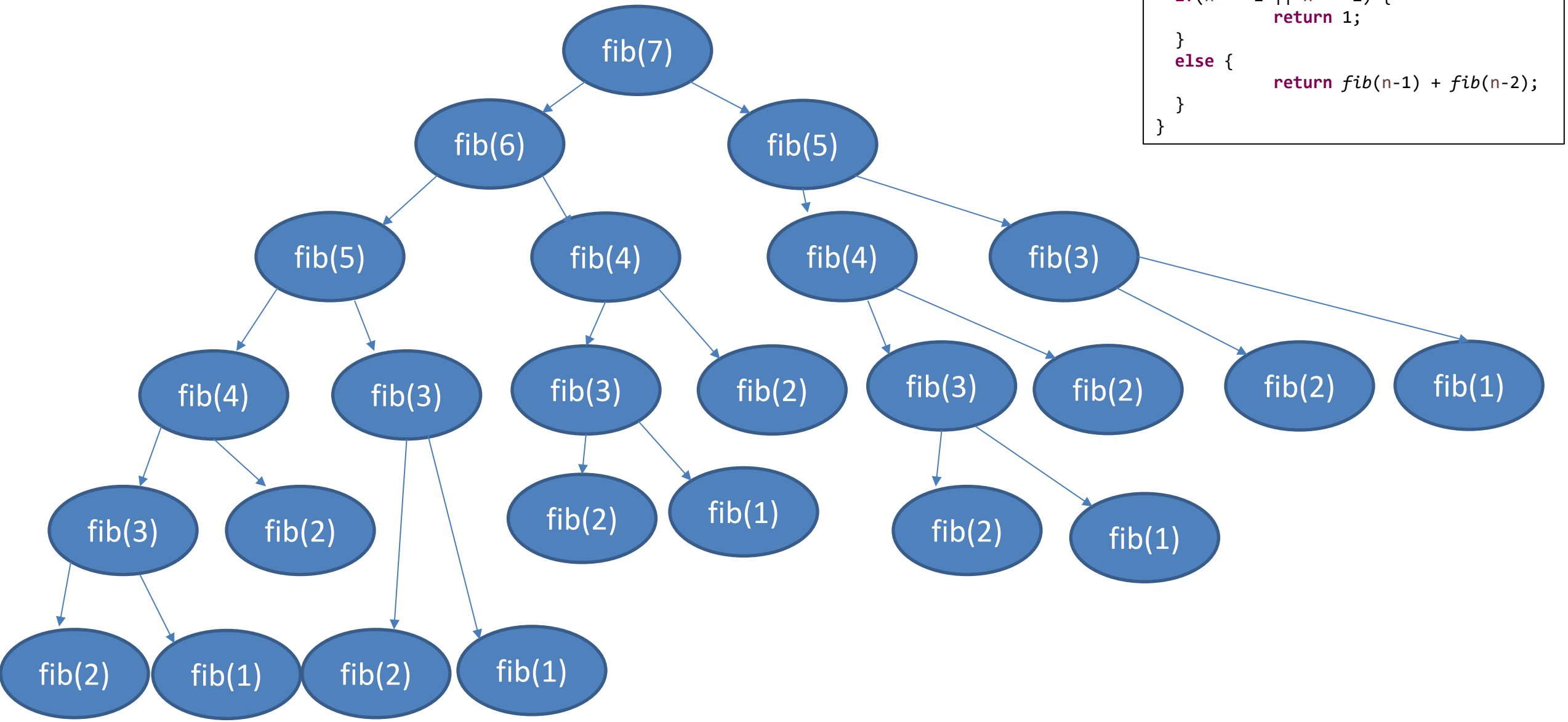
Number of recursive calls = 14




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

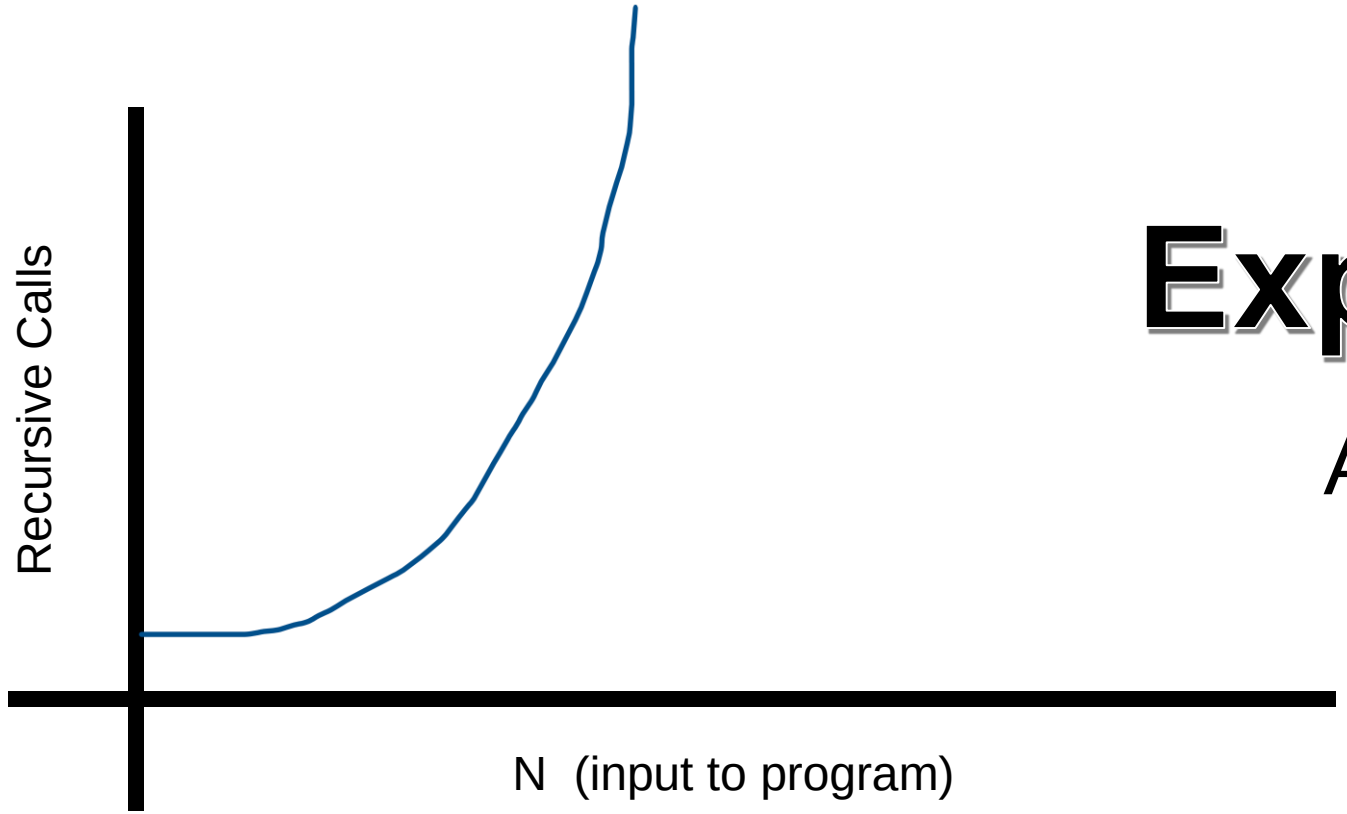
Number of recursive calls = 24

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



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

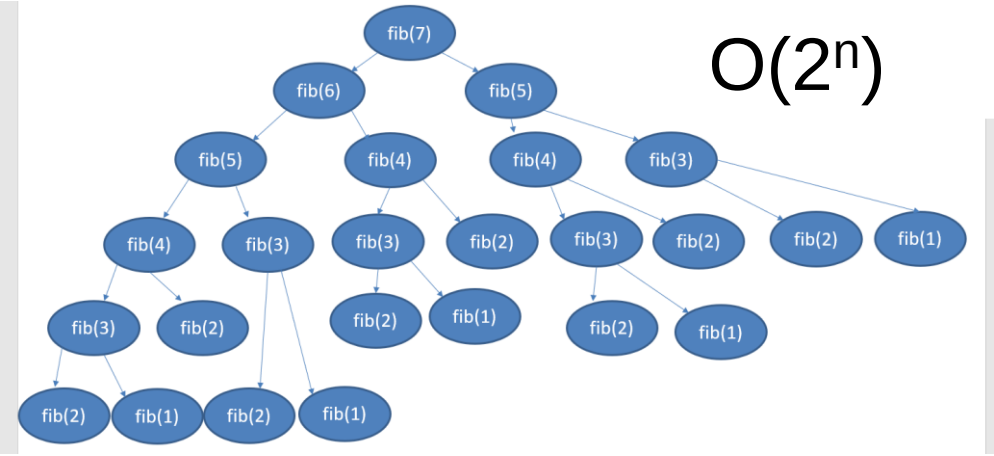
If we were to plot the number of recursive calls made as n increases, it would look something like his:



Exponential

Aka. $O(2^n)$

```
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 = **$O(2^n)$** * **$O(1)$**

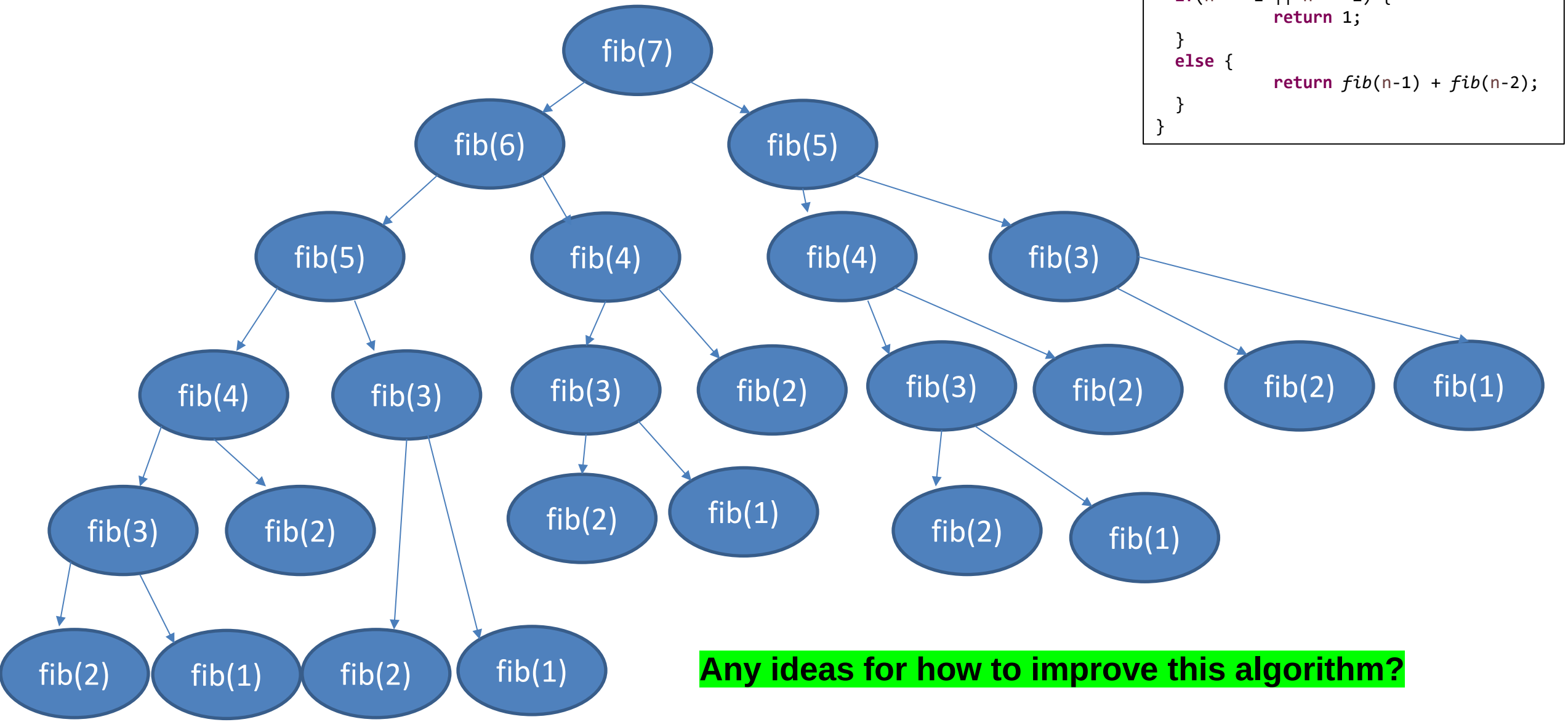
Total running time = **$O(2^n)$**

n = requested Fibonacci digit

$O(2^n)$ is very bad…

Number of recursive calls = 24

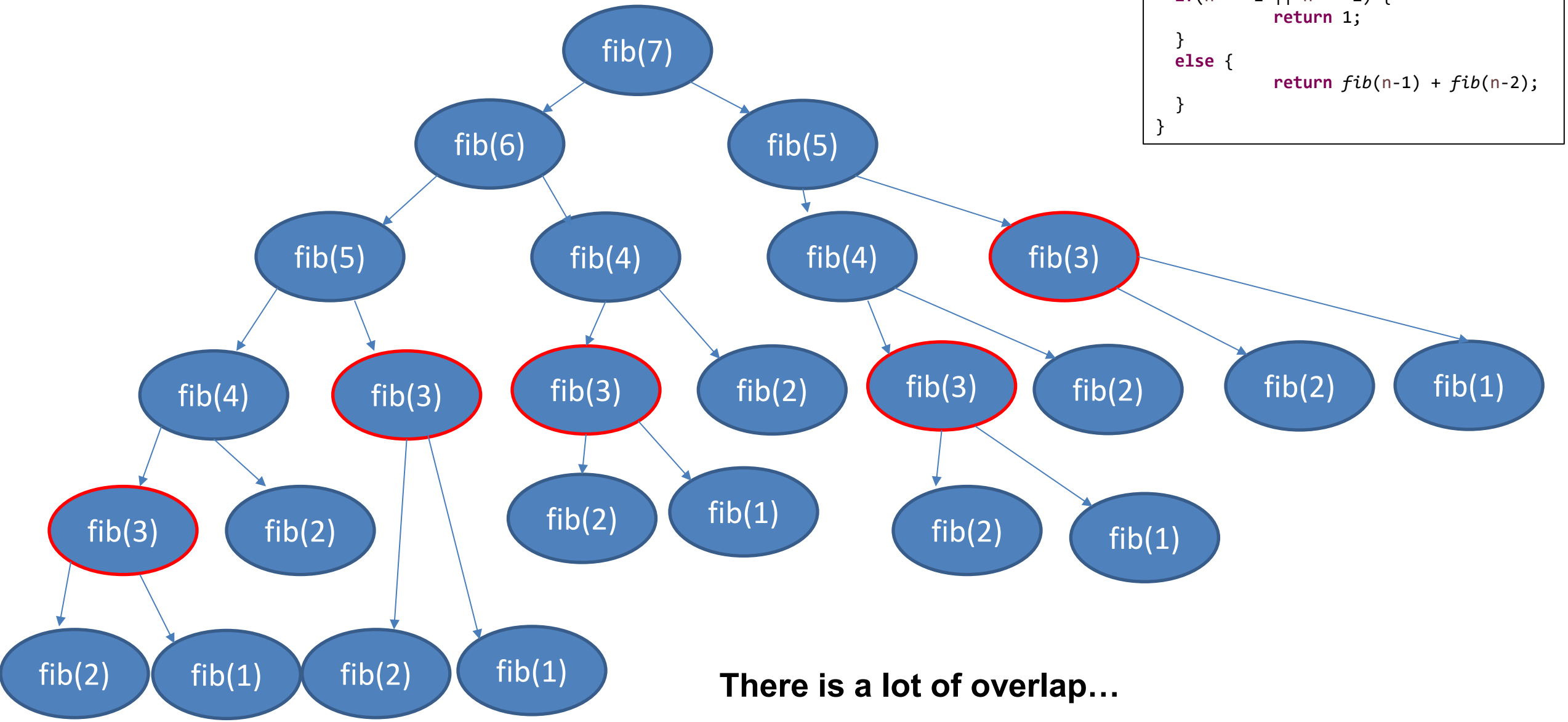
```
private static int fib(int n) {
  if(n == 1 || n == 2) {
            return 1;
  }
  else {
            return fib(n-1) + fib(n-2);
  }
}
```



Any ideas for how to improve this algorithm?

Number of recursive calls = 24

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

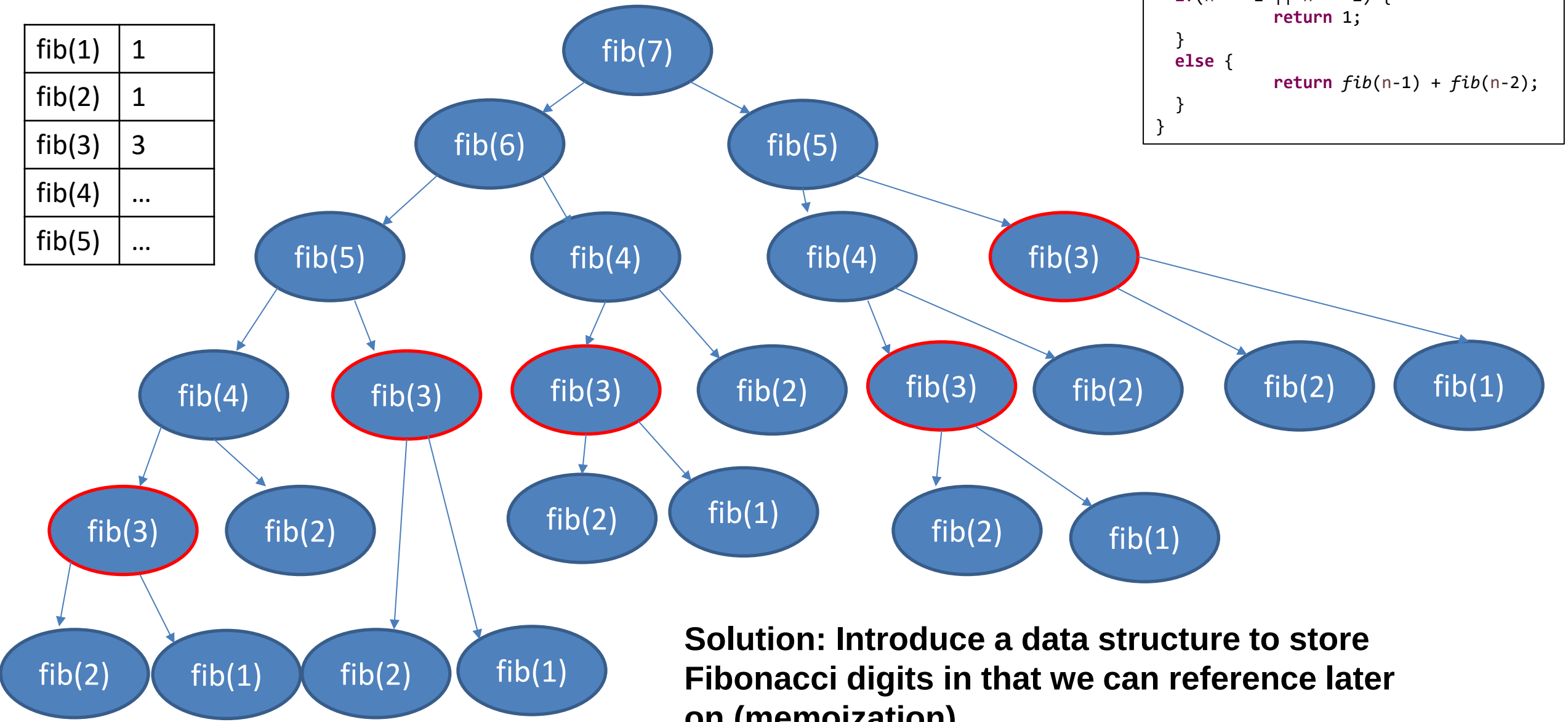


There is a lot of overlap…

Number of recursive calls = 24

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

```
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!)

countX(“oxxo”)

```
public static int countX(String str) {
    if(str.length() == 0){
        return 0;
    }
    if(str.charAt(0) == 'x'){
        return 1 + countX(str.substring(1));
    }
    else{
        return 0 + countX(str.substring(1));
    }
}
```

countX(“**o**xxo”)

0 + countX(“xxo”)

```
public static int countX(String str) {
    if(str.length() == 0){
        return 0;
    }
    if(str.charAt(0) == 'x'){
        return 1 + countX(str.substring(1));
    }
    else{
        return 0 + countX(str.substring(1));
    }
}
```

countX(“**o**xxo”)

0 + countX(“**x**xo”)

1 + countX(“xo”)

```
public static int countX(String str) {
    if(str.length() == 0){
        return 0;
    }
    if(str.charAt(0) == 'x'){
        return 1 + countX(str.substring(1));
    }
    else{
        return 0 + countX(str.substring(1));
    }
}
```

countX(“oxxo”)

0 + countX(“xxo”)

1 + countX(“xo”)

1 + countX(“o”)

```
public static int countX(String str) {
    if(str.length() == 0){
        return 0;
    }
    if(str.charAt(0) == 'x'){
        return 1 + countX(str.substring(1));
    }
    else{
        return 0 + countX(str.substring(1));
    }
}
```

countX(“oxxo”)

0 + countX(“xxo”)

1 + countX(“xo”)

1 + countX(“o”)

0 + countX(“”)

```
public static int countX(String str) {
    if(str.length() == 0){
        return 0;
    }
    if(str.charAt(0) == 'x'){
        return 1 + countX(str.substring(1));
    }
    else{
        return 0 + countX(str.substring(1));
    }
}
```

countX(“**o**xxo”)

0 + countX(“**x**xo”)

1 + countX(“**x**o”)

1 + countX(“o”)

0 + countX(“”)

0

```
public static int countX(String str) {
    if(str.length() == 0){
        return 0;
    }
    if(str.charAt(0) == 'x'){
        return 1 + countX(str.substring(1));
    }
    else{
        return 0 + countX(str.substring(1));
    }
}
```

countX(“**o**xxo”)

0 + countX(“**x**xo”)

1 + countX(“**x**o”)

1 + countX(“o”)

0 + 0

```
public static int countX(String str) {
    if(str.length() == 0){
        return 0;
    }
    if(str.charAt(0) == 'x'){
        return 1 + countX(str.substring(1));
    }
    else{
        return 0 + countX(str.substring(1));
    }
}
```

countX(“oxxo”)

0 + countX(“xxo”)

1 + countX(“xo”)

1 + 0

```
public static int countX(String str) {
    if(str.length() == 0){
        return 0;
    }
    if(str.charAt(0) == 'x'){
        return 1 + countX(str.substring(1));
    }
    else{
        return 0 + countX(str.substring(1));
    }
}
```

countX(“oxxo”)

0 + countX(“xxo”)

1 + countX(“xo”)

1 + 0

```
public static int countX(String str) {
    if(str.length() == 0){
        return 0;
    }
    if(str.charAt(0) == 'x'){
        return 1 + countX(str.substring(1));
    }
    else{
        return 0 + countX(str.substring(1));
    }
}
```

countX(“**o**xxo”)

0 + countX(“**x**xo”)

1 + 1

```
public static int countX(String str) {
    if(str.length() == 0){
        return 0;
    }
    if(str.charAt(0) == 'x'){
        return 1 + countX(str.substring(1));
    }
    else{
        return 0 + countX(str.substring(1));
    }
}
```

countX(“oxxo”)

0 + 2

```
public static int countX(String str) {
    if(str.length() == 0){
        return 0;
    }
    if(str.charAt(0) == 'x’){
        return 1 + countX(str.substring(1));
    }
    else{
        return 0 + countX(str.substring(1));
    }
}
```

Final answer = 2

```
public static int countX(String str) {
    if(str.length() == 0){
        return 0;
    }
    if(str.charAt(0) == 'x'){
        return 1 + countX(str.substring(1));
    }
    else{
        return 0 + countX(str.substring(1));
    }
}
```

Limitations of recursion?