A First Look at Recursion

Computer Science 111
Boston University

Christine Papadakis-Kanaris

Iteration

- When we encounter a problem that requires repetition, we often use *iteration* – i.e., some type of loop.
- Sample problem: printing the series of integers from n1 to n2, where n1 <= n2.
  - example: `printSeries(5, 10)` should print the following:

    `5, 6, 7, 8, 9, 10`
- An iterative solution to this problem:

```
public static void printSeries(int n1, int n2) {
    for (int i = n1; i <= n2; i++) {
        System.out.print(i + ", ");
    }
    System.out.println(n2);
}
```

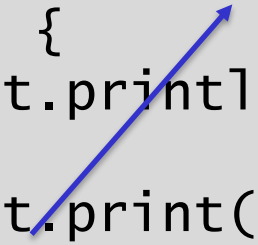
Recursion

- An alternative approach to problems that require repetition is to solve them using *recursion.*
- A recursive method is a method that calls itself.
- When we use recursion, we solve a problem by reducing it to a simpler problem of the same kind.
- We keep doing this until we reach a problem that is simple enough to be solved directly.

Recursion

- An alternative approach to problems that require repetition is to solve them using *recursion.*
- A recursive method is a method that calls itself.
- Applying this approach to the print-series problem gives:

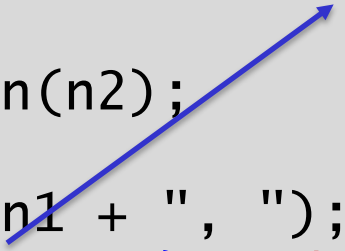
```
public static void printSeries(int n1, int n2) {
    if (n1 == n2) {
        System.out.println(n2);
    } else {
        System.out.print(n1 + ", ");
        printSeries(n1 + 1, n2);  // recursive case
    }
}
```



Recursion

- An alternative approach to problems that require repetition is to solve them using *recursion.*
- A recursive method is a method that calls itself.
- Applying this approach to the print-series problem gives:

```
public static void printSeries(int n1, int n2) {
    if (n1 == n2) {
        System.out.println(n2);
    } else {
        System.out.print(n1 + ", ");
        printSeries(n1 + 1, n2);  // recursive case
    }
}
```



Recursion

- An alternative approach to problems that require repetition is to solve them using *recursion.*
- A recursive method is a method that calls itself.
- Applying this approach to the print-series problem gives:

```
public static void printSeries(int n1, int n2) {
    if (n1 == n2) {                  // base case
        System.out.println(n2);
    } else {
        System.out.print(n1 + ", ");
        printSeries(n1 + 1, n2);  // recursive case
    }
}
```

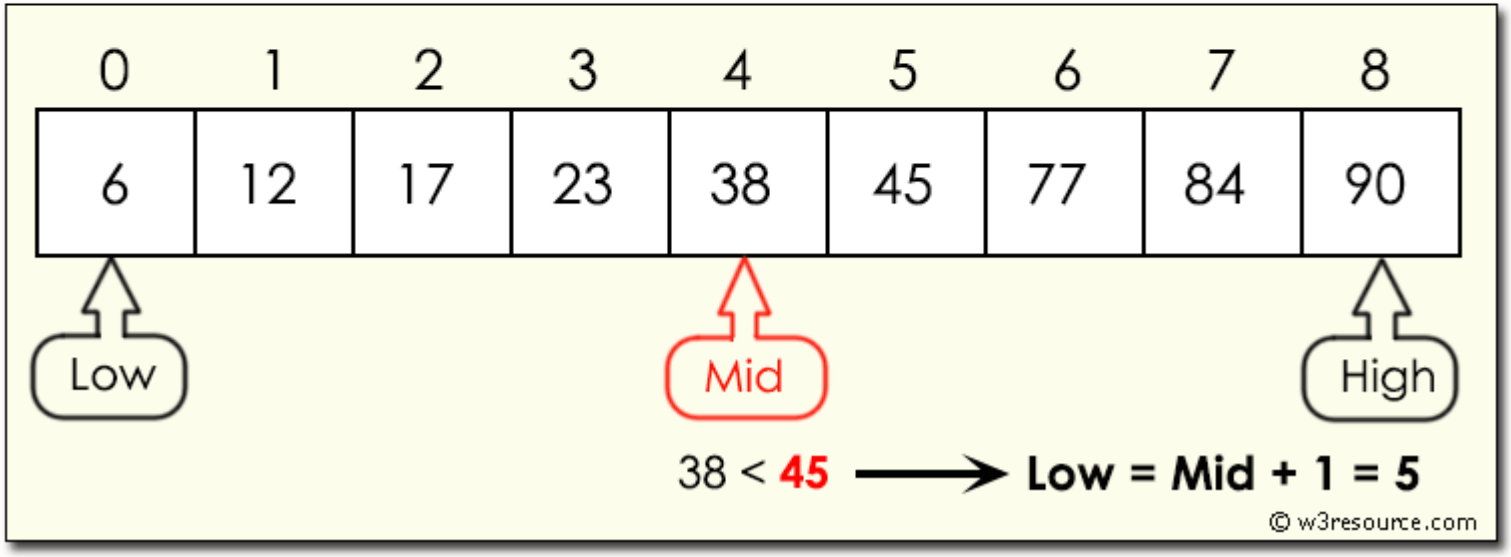
- The base case stops the recursion, because it doesn't make another call to the method.

Binary Search

	Low	High	Mid
#1	0	8	4

Search (45)

$$mid = \left[\frac{low + high}{2} \right]$$



Finding a Phone Number:

Linear Search

```
public String findNumber(String name, Book phonebook){
    String number = "unknown";

    for (int i = 1; i <= phonebook.num_pages(); i++ ){
        if (person is found on the current page) {
            number = the person's phone number
            break;
        }
    }
    return number;
}
```

Scan each page of the phone book, one page at a time, until we find the person we are looking for … or we run out of pages to search.

Finding a Phone Number:

Binary Search

```
public String findNumber(Strin
    String number = “unknown”;

    int min = 1;
    int max = phonebook.num_pages();

    while (min <= max) {
        mid = (min + max) / 2;      # the middle page
        if (person is found on page mid) {
            number = the person's number
            break;
        } else if (person comes earlier in phonebook)
            max = mid - 1;
        else:
            min = mid + 1;
    }

    return number;
}
```

min .. **max** reflect the range of pages to be searched!

Finding a Phone Number

Binary Search

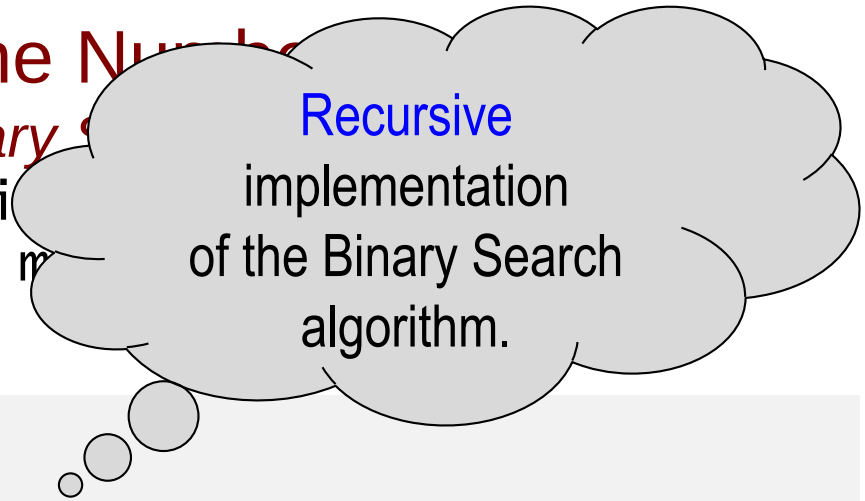
```
public String findNumber(Str
    String number = "unknown"

    int min = 1;
    int max = phonebook.num_pag  ();

    while (min <= max) {
        mid = (min + max) / 2;      # the middle page
        if (person is found on page mid) {
            number = the person's number
            break;
        } else if (person comes earlier in phonebook)
            max = mid - 1;
        else:
            min = mid + 1;
    }

    return number;
}
```

Iterative implementation of the Binary Search algorithm.

repeat the process until we find the person we are looking for **or** we can no longer cut the book in half!!

Finding a Phone Nu

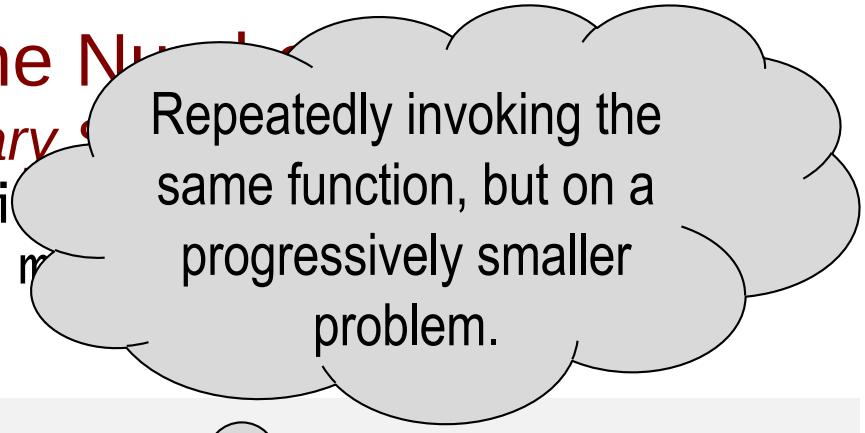
Recursive Binary S




```
public String findNumber(Stri
                         int m
  String number = "unkown";

  if ( min <= max ) {
       mid = (min + max)/2;

       if the name we are searching for is in this page
           number = assign the phone number
       else if ( name is in the first half )
           number = findNumber( name, phonebook,
                                min, mid )
       else
           number = findNumber( name, phonebook,
                                mid+1, max );
  } // if

  return( number );
}
```

Finding a Phone Nu

Recursive Binary S




```
public String findNumber(Stri
                         int m
  String number = "unkown";

  if ( min <= max ) {
      mid = (min + max)/2;

      if the name we are searching for is in this page
          number = assign the phone number
      else if ( name is in the first half )
          number = findNumber( name, phonebook,
                               min, mid )
      else
          number = findNumber( name, phonebook,
                               mid+1, max );
  } // if

  return( number );
}
```

Structure of a Recursive Method:

the general approach

```
return type recursiveMethod(parameters) {
    if (stopping condition) {    // base case
        // handle the base case
    } else {
        // recursive case:
        // possibly do something here

        recursiveMethod(modified parameters);

        // possibly do something here
    }
}
```

- There can be multiple base cases and recursive cases.
- When we make the recursive call, we typically use parameters that bring us closer to a base case.

Structure of a Recursive Method:

another approach

```
return type recursiveMethod(parameters) {

    if ( !stopping condition ) {
        // recursive case:
        // possibly do something here

        recursiveMethod(modified parameters);

        // possibly do something here
    }

}
```

- If we don't have to do anything explicitly when we reach the base case but stop the recursion, the base case is implied by an explicit check to test for the recursive case.
- When we make the recursive call, we typically use parameters that bring us closer to a base case.

Factorial Function
A Classic Recursion

Factorial Function
A Classic Recursion

5! = 5 * 4 * 3 * 2 * 1

4! = 4 * 3 * 2 * 1

3! = 3 * 2 * 1

2! = 2 * 1

1! = 1 * 0!

0! = 1

5! = 5 * 4!

4! = 4 * 3!

3! = 3 * 2!

2! = 2 * 1!

1! = 1 * 0!

0! = 1

$$n! = \begin{cases} 1 & \text{if } n = 0, \\ (n-1)! \times n & \text{if } n > 0 \end{cases}$$

Base Case

Factorial Function
A Classic Recursion

5! = 5 * 4 * 3 * 2 * 1

4! = 4 * 3 * 2 * 1

3! = 3 * 2 * 1

2! = 2 * 1

1! = 1 * 0!

0! = 1

5! = 5 * 4!

4! = 4 * 3!

3! = 3 * 2!

2! = 2 * 1!

1! = 1 * 0!

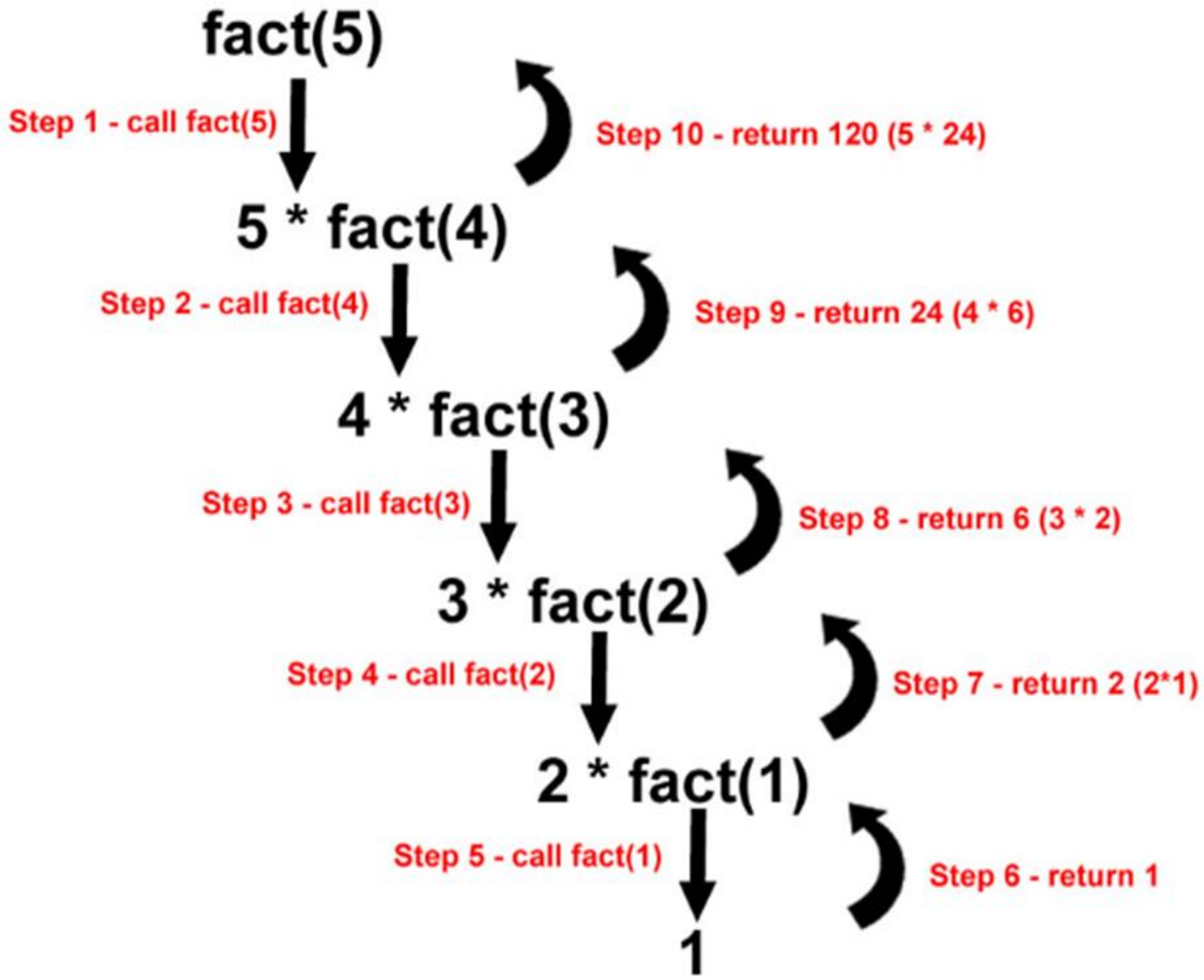
0! = 1

$$n! = \begin{cases} 1 & \text{if } n = 0, \\ (n-1)! \times n & \text{if } n > 0 \end{cases}$$

Recursive Case

Factorial Function:

Recursive Approach



Functions Calling Themselves: *Recursion!*

```
public static int fac( int n) {
    if ( n <= 1 )
        return 1
    else:
        return n * fac(n - 1)
}
```

But there has to be a special case that causes the recursion to terminate!

- Recursion solves a problem by reducing it to a *simpler* or *smaller* problem *of the same kind*.
  - the function calls itself to solve the smaller problem!

- We take advantage of *recursive substructure*.
  - the fact that we can define the problem *in terms of itself*
    - n! = n * (n-1)!

Designing a Recursive Function

1. Start by programming the base case(s).
   - *What instance(s) of this problem can I solve directly (without looking at anything smaller)?*

2. Find the recursive substructure.
   - *How could I use the solution to **any smaller version** of the problem to solve the overall problem?*
   - *Specifically, how can we guarantee that the base case is reached?*

3. Let the stack do the work!

Recursion, power of the Stack!

Computer Science 112
Boston University

Christine Papadakis-Kanaris

Tracing a (void) Recursive Method

```
public static void printSeries(int n1, int n2) {
    if (n1 == n2) {
        System.out.println(n2);
    } else {
        System.out.print(n1 + ", ");
        printSeries(n1 + 1, n2);
    }
}
```

- What happens when we execute `printSeries(5, 7)`?

Tracing a (void) Recursive Method

```
public static void printSeries(int n1, int n2) {
    if (n1 == n2) {
        System.out.println(n2);
    } else {
        System.out.print(n1 + ", ");
        printSeries(n1 + 1, n2);
    }
}
```

- What happens when we execute `printSeries(5, 7)`?

  `printSeries(5, 7):`

Tracing a (void) Recursive Method

```
public static void printSeries(int n1, int n2) {
    if (n1 == n2) {
        System.out.println(n2);
    } else {
        System.out.print(n1 + ", ");
        printSeries(n1 + 1, n2);
    }
}
```

- What happens when we execute `printSeries(5, 7)`?

  `printSeries(5, 7):`

Tracing a (void) Recursive Method

```
public static void printSeries(int n1, int n2) {
    if (n1 == n2) {
        System.out.println(n2);
    } else {
        System.out.print(n1 + ", ");
        printSeries(n1 + 1, n2);
    }
}
```

- What happens when we execute `printSeries(5, 7)`?

```
printSeries(5, 7):
    System.out.print(5 + ", ");
```

Tracing a (void) Recursive Method

```
public static void printSeries(int n1, int n2) {
    if (n1 == n2) {
        System.out.println(n2);
    } else {
        System.out.print(n1 + ", ");
        printSeries(n1 + 1, n2);
    }
}
```

- What happens when we execute `printSeries(5, 7)`?

```
printSeries(5, 7):
    System.out.print(5 + ", ");
```

Tracing a (void) Recursive Method

```
public static void printSeries(int n1, int n2) {
    if (n1 == n2) {
        System.out.println(n2);
    } else {
        System.out.print(n1 + ", ");
        printSeries(n1 + 1, n2);
    }
}
```

- What happens when we execute `printSeries(5, 7)`?

```
printSeries(5, 7):
    System.out.print(5 + ", ");
    printSeries(6, 7):
```

Tracing a (void) Recursive Method

```
public static void printSeries(int n1, int n2) {
    if (n1 == n2) {
        System.out.println(n2);
    } else {
        System.out.print(n1 + ", ");
        printSeries(n1 + 1, n2);
    }
}
```

- What happens when we execute `printSeries(5, 7)`?

```
printSeries(5, 7):
    System.out.print(5 + ", ");
    printSeries(6, 7):
```

Tracing a (void) Recursive Method

```
public static void printSeries(int n1, int n2) {
    if (n1 == n2) {
        System.out.println(n2);
    } else {
        System.out.print(n1 + ", ");
        printSeries(n1 + 1, n2);
    }
}
```

- What happens when we execute `printSeries(5, 7)`?

```
printSeries(5, 7):
    System.out.print(5 + ", ");
    printSeries(6, 7):
        System.out.print(6 + ", ");
```

Tracing a (void) Recursive Method

```
public static void printSeries(int n1, int n2) {
    if (n1 == n2) {
        System.out.println(n2);
    } else {
        System.out.print(n1 + ", ");
        printSeries(n1 + 1, n2);
    }
}
```

- What happens when we execute `printSeries(5, 7)`?

```
printSeries(5, 7):
    System.out.print(5 + ", ");
    printSeries(6, 7):
        System.out.print(6 + ", ");
```

Tracing a (void) Recursive Method

```
public static void printSeries(int n1, int n2) {
    if (n1 == n2) {
        System.out.println(n2);
    } else {
        System.out.print(n1 + ", ");
        printSeries(n1 + 1, n2);
    }
}
```

- What happens when we execute `printSeries(5, 7)`?

```
printSeries(5, 7):
    System.out.print(5 + ", ");
    printSeries(6, 7):
        System.out.print(6 + ", ");
        printSeries(7, 7):
```

Tracing a (void) Recursive Method

```
public static void printSeries(int n1, int n2) {
    if (n1 == n2) {
        System.out.println(n2);
    } else {
        System.out.print(n1 + ", ");
        printSeries(n1 + 1, n2);
    }
}
```

- What happens when we execute `printSeries(5, 7)`?

```
printSeries(5, 7):
    System.out.print(5 + ", ");
    printSeries(6, 7):
        System.out.print(6 + ", ");
        printSeries(7, 7):
```

Tracing a (void) Recursive Method

```
public static void printSeries(int n1, int n2) {
    if (n1 == n2) {
        System.out.println(n2);
    } else {
        System.out.print(n1 + ", ");
        printSeries(n1 + 1, n2);
    }
}
```

- What happens when we execute `printSeries(5, 7)`?

```
printSeries(5, 7):
    System.out.print(5 + ", ");
    printSeries(6, 7):
        System.out.print(6 + ", ");
        printSeries(7, 7):
            System.out.print(7);
```

Tracing a (void) Recursive Method

```
public static void printSeries(int n1, int n2) {
    if (n1 == n2) {
        System.out.println(n2);
    } else {
        System.out.print(n1 + ", ");
        printSeries(n1 + 1, n2);
    }
}
```

- What happens when we execute `printSeries(5, 7)`?

```
printSeries(5, 7):
    System.out.print(5 + ", ");
    printSeries(6, 7):
        System.out.print(6 + ", ");
        printSeries(7, 7):
            System.out.print(7);
        return
```

Tracing a (void) Recursive Method

```
public static void printSeries(int n1, int n2) {
    if (n1 == n2) {
        System.out.println(n2);
    } else {
        System.out.print(n1 + ", ");
        printSeries(n1 + 1, n2);
    }
}
```

- What happens when we execute `printSeries(5, 7)`?

```
printSeries(5, 7):
    System.out.print(5 + ", ");
    printSeries(6, 7):
        System.out.print(6 + ", ");
        printSeries(7, 7):
            System.out.print(7);
        return
    return
```

Tracing a (void) Recursive Method

```
public static void printSeries(int n1, int n2) {
    if (n1 == n2) {
        System.out.println(n2);
    } else {
        System.out.print(n1 + ", ");
        printSeries(n1 + 1, n2);
    }
}
```

- What happens when we execute `printSeries(5, 7)`?

```
printSeries(5, 7):
    System.out.print(5 + ", ");
    printSeries(6, 7):
        System.out.print(6 + ", ");
        printSeries(7, 7):
            System.out.print(7);
        return
    return
return
```

Tracing a (void) Recursive Method:

Second Example

```
public static void mystery(int i) {
    if (i <= 0) {      // base case
        return;
    }
    // recursive case
    System.out.println(i);
    mystery(i - 1);
    System.out.println(i);
}
```

- What is the output when we execute `mystery(2)`?

A.
```
2
1
```

B.
```
2
1
0
```

C.
```
2
1
0
1
2
```

D.
```
2
1
1
2
```

Tracing a (void) Recursive Method:

Second Example

```
public static void mystery(int i) {
    if (i <= 0) {      // base case
        return;
    }
    // recursive case
    System.out.println(i);
    mystery(i - 1);
    System.out.println(i);
}
```

- What is the output when we execute `mystery(2)`?

A.
```
2
1
```

B.
```
2
1
0
```

C.
```
2
1
0
1
2
```

D.
```
2
1
1
2
```

Tracing a (void) Recursive Method:

Second Example

```
public static void mystery(int i) {
    if (i <= 0) {      // base case
        return;
    }
    // recursive case
    System.out.println(i);
    mystery(i - 1);
    System.out.println(i);
}
```

- What is the output when we execute `mystery(2)`?

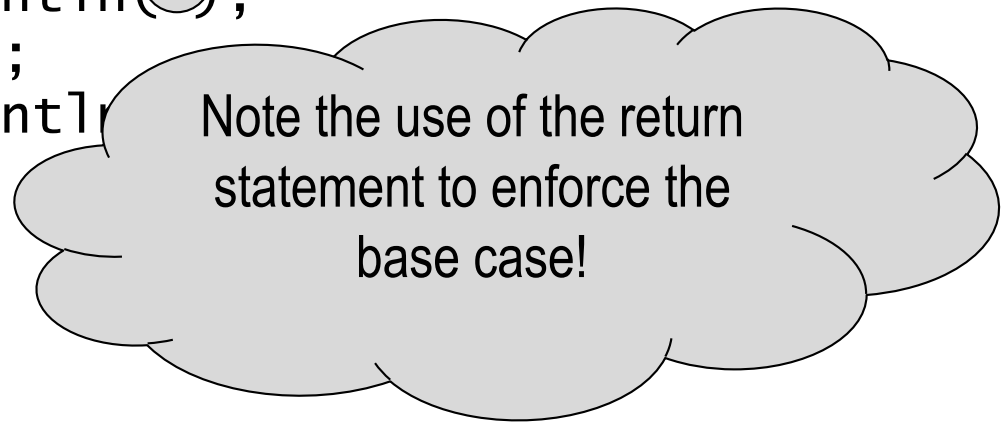
```
mystery(2) prints 2
mystery(2) calls mystery(1)
    mystery(1) prints 1
    mystery(1) calls mystery(0)
        mystery(0) just returns (base case)
    mystery(1) prints 1 and returns
mystery(2) prints 2 and returns
```

output:

2
1
1
2

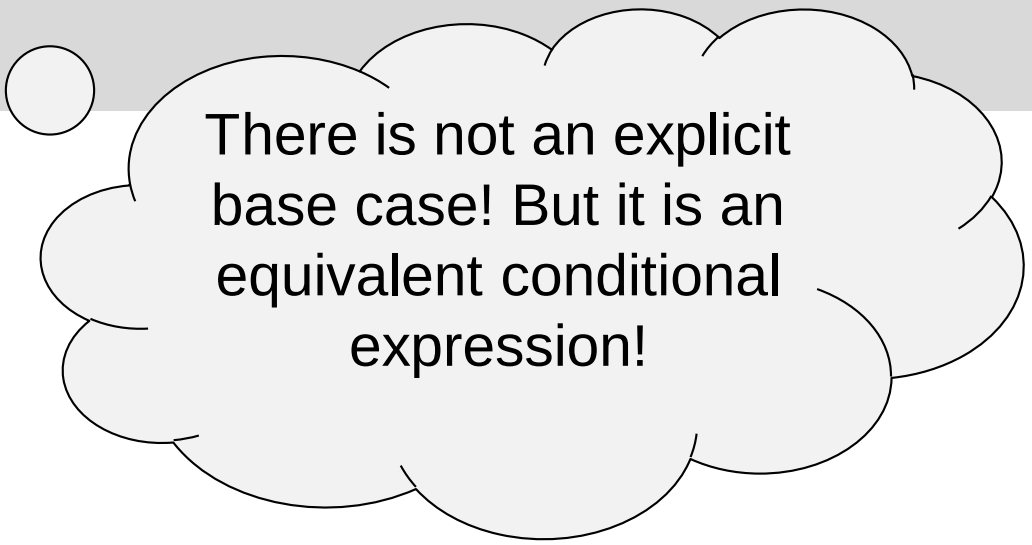
Explicit Base Case

```
public static void mystery(int i) {
    if (i <= 0) {      // base case
        return;
    }
    // recursive case
    System.out.println(i);
    mystery(i - 1);
    System.out.println(i);
}
```



Implicit Base Case

```
public static void mystery(int i) {
    if (i > 0) {
        // recursive case
        System.out.println(i);
        mystery(i - 1);
        System.out.println(i);
    }
}
```



A Recursive Method That Returns a Value

- Simple example: summing the integers from 1 to n

```
public static int sum(int n) {
    if (n <= 0) {
        return 0;
    }
    int rest = sum(n - 1);
    return n + rest;
}
```

A Recursive Method That Returns a Value

- Simple example: summing the integers from 1 to n

```
public static int sum(int n) {
    if (n <= 0) {
        return 0;
    }
    int rest = sum(n - 1);
    return n + rest;
}
```

- Example of this approach to computing the sum:

sum(6) = 6 + sum(5)
sum(5) = 5 + sum(4)

…

…

…

sum(0) = 0

Tracing a Recursive Method

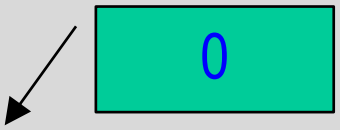
```
public static int sum(int n) {
    if (n <= 0) {
        return 0;
    }
    int rest = sum(n - 1);
    return n + rest;
}
```

- What happens when we execute `int x = sum(3);` from inside the `main()` method?

```
main() calls sum(3)
    sum(3) calls sum(2)
        sum(2) calls sum(1)
            sum(1) calls sum(0)
                sum(0) returns 0
```

Tracing a Recursive Method

```
public static int sum(int n) {
    if (n <= 0) {
        return 0;
    }
    int rest = sum(n-1);
    return n + rest;
}
```



- What happens when we execute `int x = sum(3);` from inside the `main()` method?

```
main() calls sum(3)
    sum(3) calls sum(2)
        sum(2) calls sum(1)
            sum(1) calls sum(0)
                sum(0) returns 0
            sum(1)
```

Tracing a Recursive Method

```
public static int sum(int n) {
    if (n <= 0) {
        return 0;
    }
    int rest = 0;
    return n + rest;
}
```

- What happens when we execute `int x = sum(3);` from inside the `main()` method?

```
main() calls sum(3)
    sum(3) calls sum(2)
        sum(2) calls sum(1)
            sum(1) calls sum(0)
                sum(0) returns 0
            sum(1)
```

Tracing a Recursive Method

```
public static int sum(int n) {
    if (n <= 0) {
        return 0;
    }
    int rest = 0;
    return n + rest;
}
```

- What happens when we execute `int x = sum(3);` from inside the `main()` method?

```
main() calls sum(3)
    sum(3) calls sum(2)
        sum(2) calls sum(1)
            sum(1) calls sum(0)
                sum(0) returns 0
            sum(1) returns 1 + 0 or 1
```

Tracing a Recursive Method

```
public static int sum(int n) {
    if (n <= 0) {
        return 0;
    }
    int rest = sum(n-1);
    return n + rest;
}
```

- What happens when we execute `int x = sum(3);` from inside the `main()` method?

```
main() calls sum(3)
    sum(3) calls sum(2)
        sum(2) calls sum(1)
            sum(1) calls sum(0)
                sum(0) returns 0
            sum(1) returns 1 + 0 or 1
        sum(2)
```

Tracing a Recursive Method

```
public static int sum(int n) {
    if (n <= 0) {
        return 0;
    }
    int rest = 1;
    return n + rest;
}
```

- What happens when we execute `int x = sum(3);` from inside the `main()` method?

```
main() calls sum(3)
    sum(3) calls sum(2)
        sum(2) calls sum(1)
            sum(1) calls sum(0)
                sum(0) returns 0
            sum(1) returns 1 + 0 or 1
        sum(2)
```

Tracing a Recursive Method

```
public static int sum(int n) {
    if (n <= 0) {
        return 0;
    }
    int rest = 1;
    return n + rest;
}
```

- What happens when we execute `int x = sum(3);` from inside the `main()` method?

```
main() calls sum(3)
    sum(3) calls sum(2)
        sum(2) calls sum(1)
            sum(1) calls sum(0)
                sum(0) returns 0
            sum(1) returns 1 + 0 or 1
        sum(2) returns 2 + 1 or 3
```

Tracing a Recursive Method

```
public static int sum(int n) {
    if (n <= 0) {
        return 0;
    }
    int rest = sum(n-1);
    return n + rest;
}
```

- What happens when we execute `int x = sum(3);` from inside the `main()` method?

```
main() calls sum(3)
    sum(3) calls sum(2)
        sum(2) calls sum(1)
            sum(1) calls sum(0)
                sum(0) returns 0
            sum(1) returns 1 + 0 or 1
        sum(2) returns 2 + 1 or 3
    sum(3)
```

Tracing a Recursive Method

```
public static int sum(int n) {
    if (n <= 0) {
        return 0;
    }
    int rest = 3;
    return n + rest;
}
```

- What happens when we execute `int x = sum(3);` from inside the `main()` method?

```
main() calls sum(3)
    sum(3) calls sum(2)
        sum(2) calls sum(1)
            sum(1) calls sum(0)
                sum(0) returns 0
            sum(1) returns 1 + 0 or 1
        sum(2) returns 2 + 1 or 3
    sum(3)
```

Tracing a Recursive Method

```
public static int sum(int n) {
    if (n <= 0) {
        return 0;
    }
    int rest = 3;
    return n + rest;
}
```

- What happens when we execute `int x = sum(3);` from inside the `main()` method?

```
main() calls sum(3)
    sum(3) calls sum(2)
        sum(2) calls sum(1)
            sum(1) calls sum(0)
                sum(0) returns 0
            sum(1) returns 1 + 0 or 1
        sum(2) returns 2 + 1 or 3
    sum(3) returns 3 + 3 or 6
```

Tracing a Recursive Method

```
public static int sum(int n) {
    if (n <= 0) {
        return 0;
    }
    int rest = sum(n - 1);
    return n + rest;
}
```

- What happens when we execute `int x = sum(3);` from inside the `main()` method?

```
main() calls sum(3)
    sum(3) calls sum(2)
        sum(2) calls sum(1)
            sum(1) calls sum(0)
                sum(0) returns 0
            sum(1) returns 1 + 0 or 1
        sum(2) returns 2 + 1 or 3
    sum(3) returns 3 + 3 or 6
main()
```

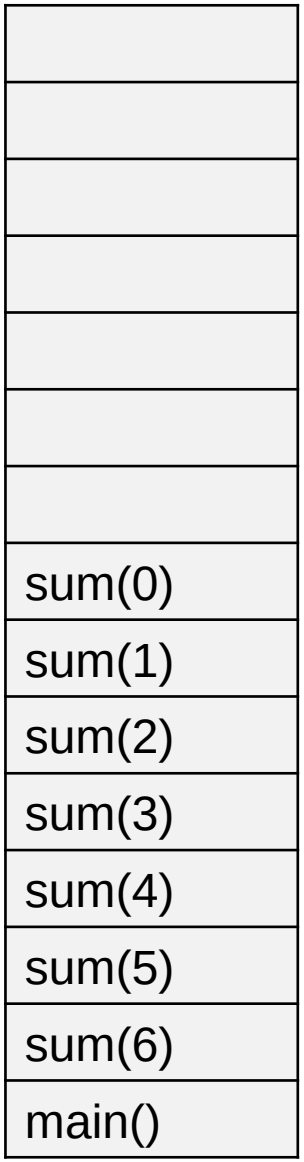
Tracing a Recursive Method

```
public static int sum(int n) {
    if (n <= 0) {
        return 0;
    }
    int rest = sum(n - 1);
    return n + rest;
}
```

- What happens when we execute `int x = 6;` from inside the `main()` method?

```
main() calls sum(3)
    sum(3) calls sum(2)
        sum(2) calls sum(1)
            sum(1) calls sum(0)
                sum(0) returns 0
            sum(1) returns 1 + 0 or 1
        sum(2) returns 2 + 1 or 3
    sum(3) returns 3 + 3 or 6
main() assigns 6 to x
```

Importance of the Stack

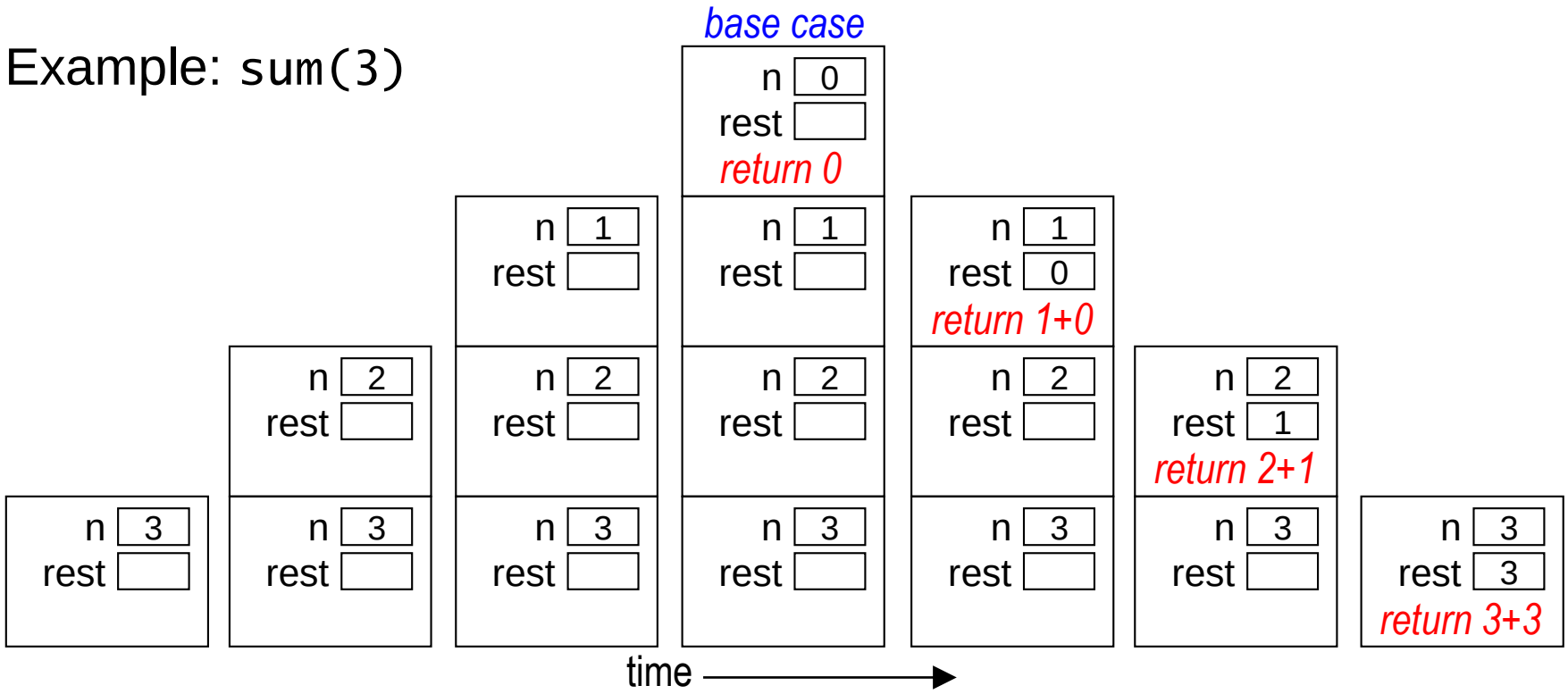


Maintains the sequence of function calls that allows us to get back to where we started from. *But more importantly, it gives us a way to preserve the **scope** of local variables.*

Tracing a Recursive Method on the Stack

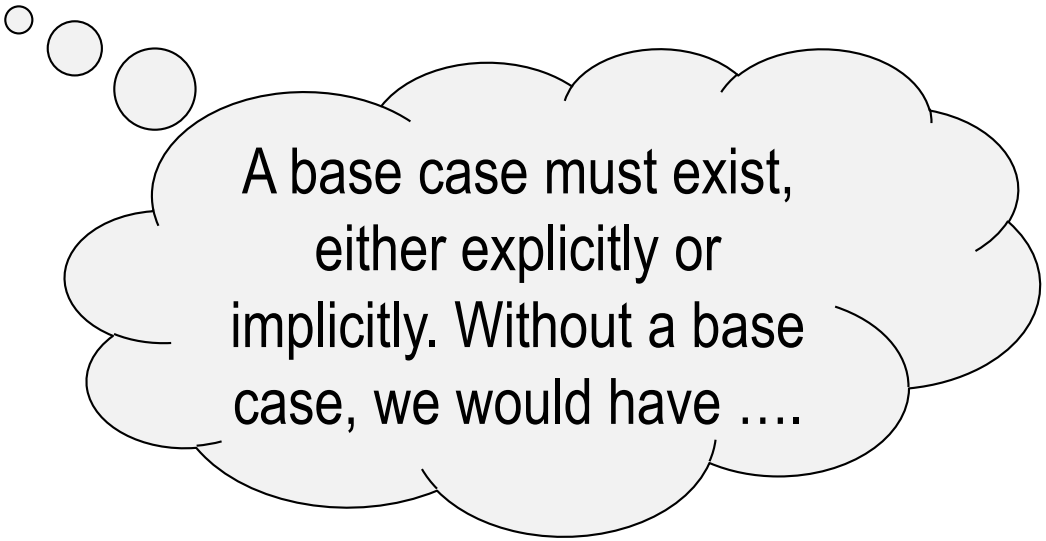
```
public static int sum(int n) {
    if (n <= 0) {
        return 0;
    }
    int rest = sum(n - 1);
    return n + rest;
}
```

Example: `sum(3)`



An alternative method

```
public static int sum(int n) {
        rest = 0;
        if (n > 0)
           rest = n + sum(n – 1);

        return(rest);
   }
```



Infinite Recursion

- We have to ensure that a recursive method will eventually reach a base case, regardless of the initial input.
- Otherwise, we can get *infinite recursion*.
  - produces *stack overflow* – there's no room for more frames on the stack!

Infinite Recursion

- We have to ensure that a recursive method will eventually reach a base case, regardless of the initial input.
- Otherwise, we can get *infinite recursion*.
  - produces *stack overflow* – there's no room for more frames on the stack!
- **Example: here's a version of our `sum()` method that uses a different test for the base case:**

```
public static int sum(int n) {
    if (n == 0) {
        return 0;
    }
    int rest = sum(n - 1);
    return n + rest;
}
```

  - what values of n would cause infinite recursion?

Processing a String Recursively

- A string is a recursive data structure. It is either:
  - empty ("")
  - a single character, followed by a string
- Thus, we can easily use recursion to process a string.
  - process one or two of the characters
  - make a recursive call to process the rest of the string
- Example: print a string vertically, one character per line:

```
public static void printVertical(String s) {
    if (s == null || s.equals("")) {
        return;
    } else {
        System.out.println(s.charAt(0)); // first char
        printVertical(s.substring(1));   // rest of s
    }
}
```

What happens if we swap the order?

```
public static void printVertical(String s) {
     if (s == null || s.equals("")) {
         return;
     } else {
         System.out.println(s.charAt(0)); // first char
         printVertical(s.substring(1));   // rest of s
     }
 }
```

What happens if we swap the order?

```
public static void printVertical(String s) {
     if (s == null || s.equals("")) {
          return;
     } else {
          printVertical(s.substring(1));   // rest of
          System.out.println(s.charAt(0)); // first char
     }
}
```

A. Nothing

B. Something, but I don't know what?

C. The string is output horizontally.

D. The string is output vertically in reverse order.

E. The string is output vertically.

What happens if we call this function

within a Java print statement?

```
public static void printVertical(String s) {
     if (s == null || s.equals("")) {
          return;
     } else {
          printVertical(s.substring(1));   // rest of
          System.out.println(s.charAt(0)); // first char
     }
 }
```

System.out.println(printVertical("I hate recursion!"));

A. Nothing

B. Something, but I don't know what?

C. Compilation error.

D. Java outputs the address of the function printVertical.

E. Java outputs the address of the string.

What happens if we call this function

within a Java print statement?

```
public static void printVertical(String s) {
     if (s == null || s.equals("")) {
          return;
     } else {
          printVertical(s.substring(1));   // rest of
          System.out.println(s.charAt(0)); // first char
     }
 }
```

System.out.println(printVertical("I hate recursion!"));

A. Nothing

B. Something, but

C. Compilation err

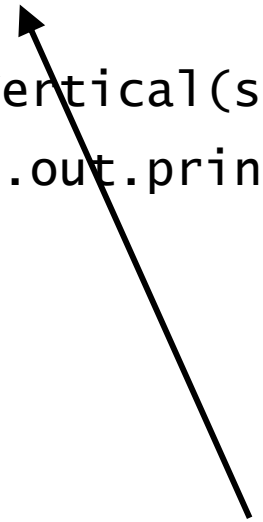
D. Java outputs th
   printVertical.

E. Java outputs th

The method println is attempting to print the return of the function printVertical! In this case printVertical is a void function and so nothing is being returned. Java does **not** return a None object!

Structuring our Recursive Method

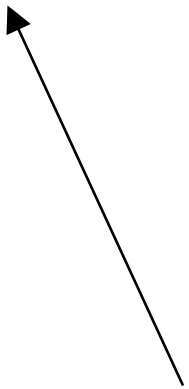
```
public static void printVertical(String s) {
     if (s == null || s.equals("")) {
          return;
     } else {
          printVertical(s.substring(1));   // rest of
          System.out.println(s.charAt(0)); // first char
     }
}
```



Note again the explicit base case with a *forced* return statement.

Structuring our Recursive Method

```
public static void printVertical(String s) {

    if (s != null && !s.equals("")) {
        printVertical(s.substring(1));   // rest of
        System.out.println(s.charAt(0)); // first char
    }
}
```



The base case is implicit in the conditional statement!

Counting Occurrences of a Character in a String

- Let's design a recursive method called `numOccur()`.
- `numOccur(ch, s)` should return the number of times that the character `ch` appears in the string `s`
- Thinking recursively:

Counting Occurrences of a Character in a String

```
public static int numOccur(char ch, String s) {
    if (s == null || s.equals("")) {
        return 0;
    } else {  // recursive class
        int rest = numOccur(ch, s.substring(1));
        if (s.charAt(0) == ch) {
            return 1 + rest; // add to count and return
        } else {
            return rest;
        }
    }
}
   numOccur('a', "aha")
```

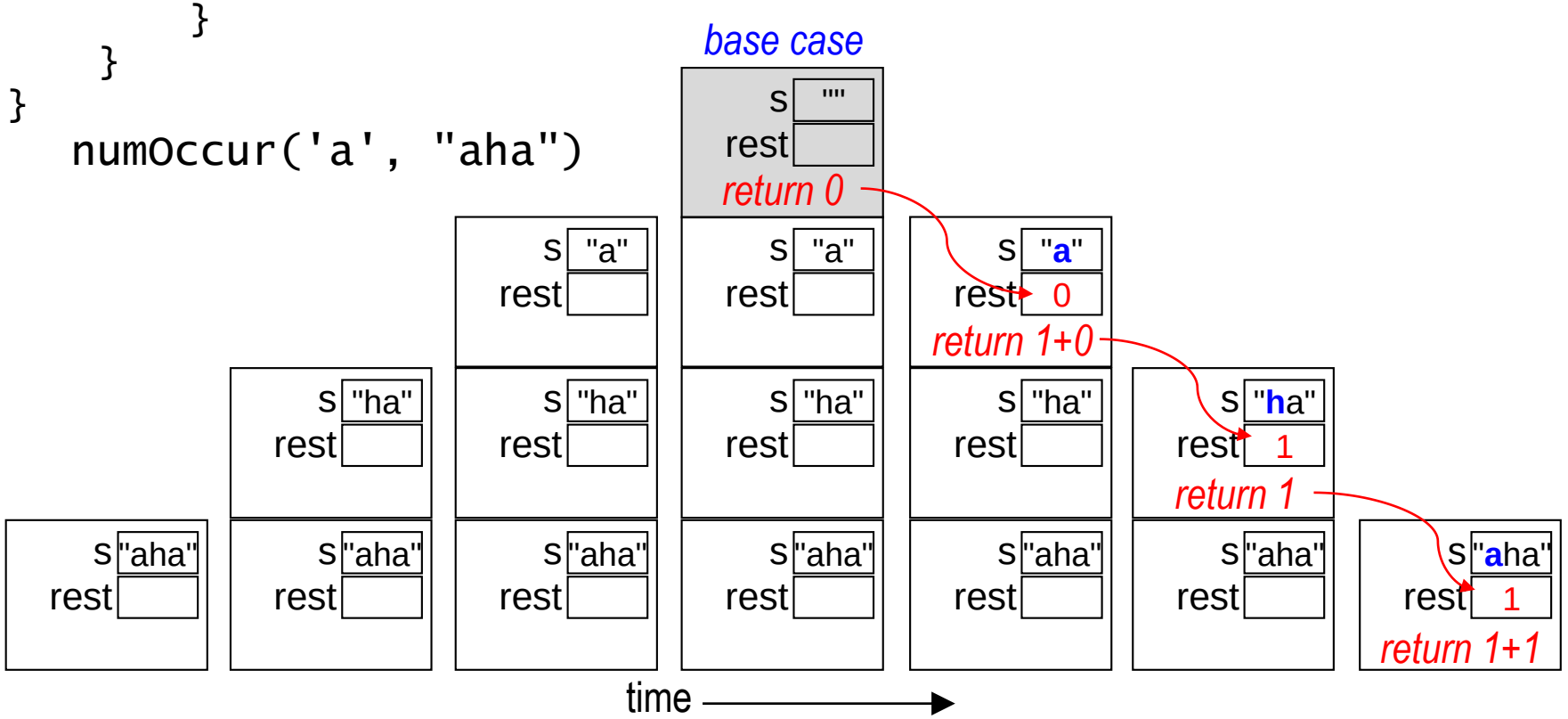
Counting Occurrences of a Character in a String

```
public static int numOccur(char ch, String s) {
    if (s == null || s.equals("")) {
        return 0;
    } else {  // recursive class
        int rest = numOccur(ch, s.substring(1));
        if (s.charAt(0) == ch) {
            return 1 + rest;
        } else {
            return rest; // ignore and return
        }
    }
}
   numOccur('a', "aha")
```

Tracing a Recursive Method on the Stack

```
public static int numOccur(char ch, String s) {
    if (s == null || s.equals("")) {
        return 0;
    } else {
        int rest = numOccur(ch, s.substring(1));
        if (s.charAt(0) == ch) {
            return 1 + rest;
        } else {
            return rest;
        }
    }
}
```

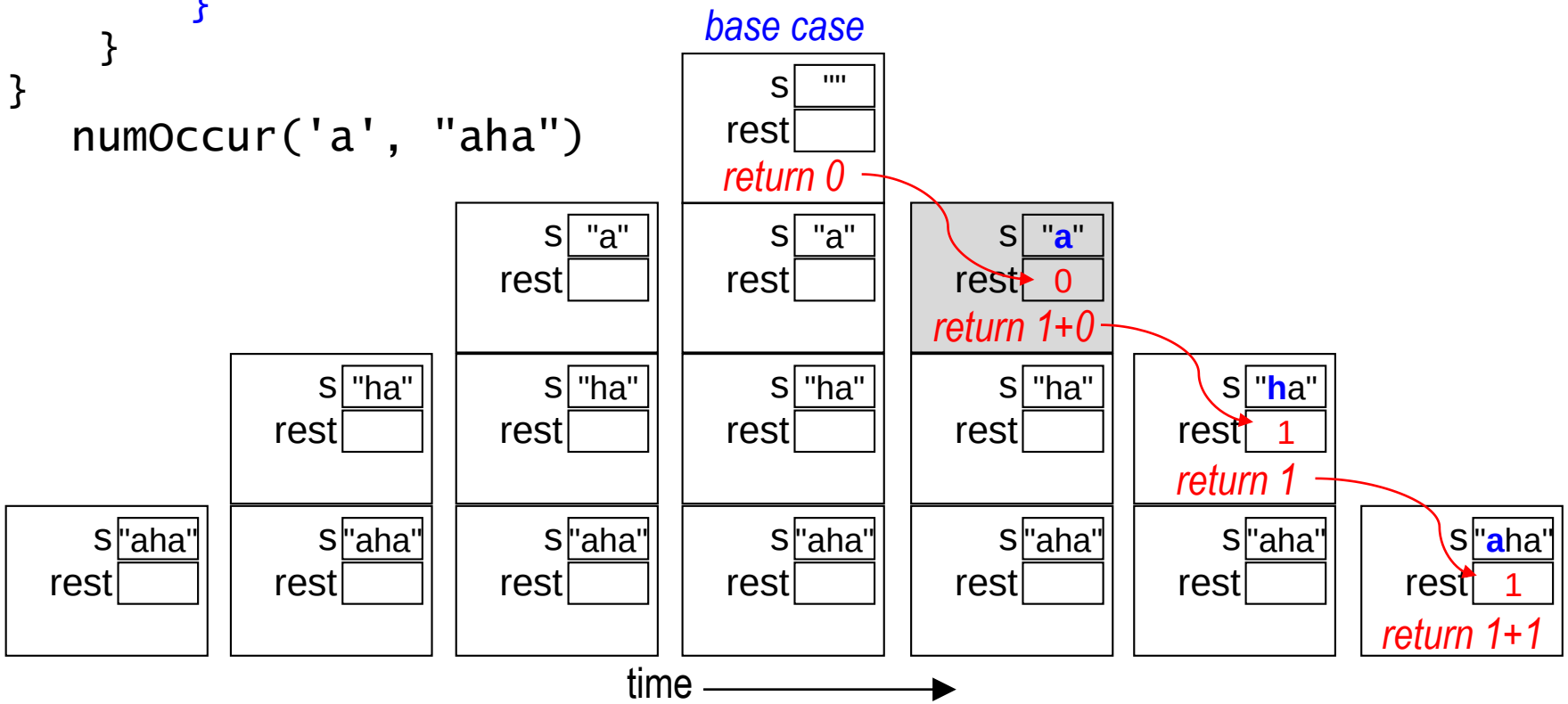
numOccur('a', "aha")



Tracing a Recursive Method on the Stack

```
public static int numOccur(char ch, String s) {
    if (s == null || s.equals("")) {
        return 0;
    } else {
        int rest = numOccur(ch, s.substring(1));
        if (s.charAt(0) == ch) {
            return 1 + rest;
        } else {
            return rest;
        }
    }
}
```

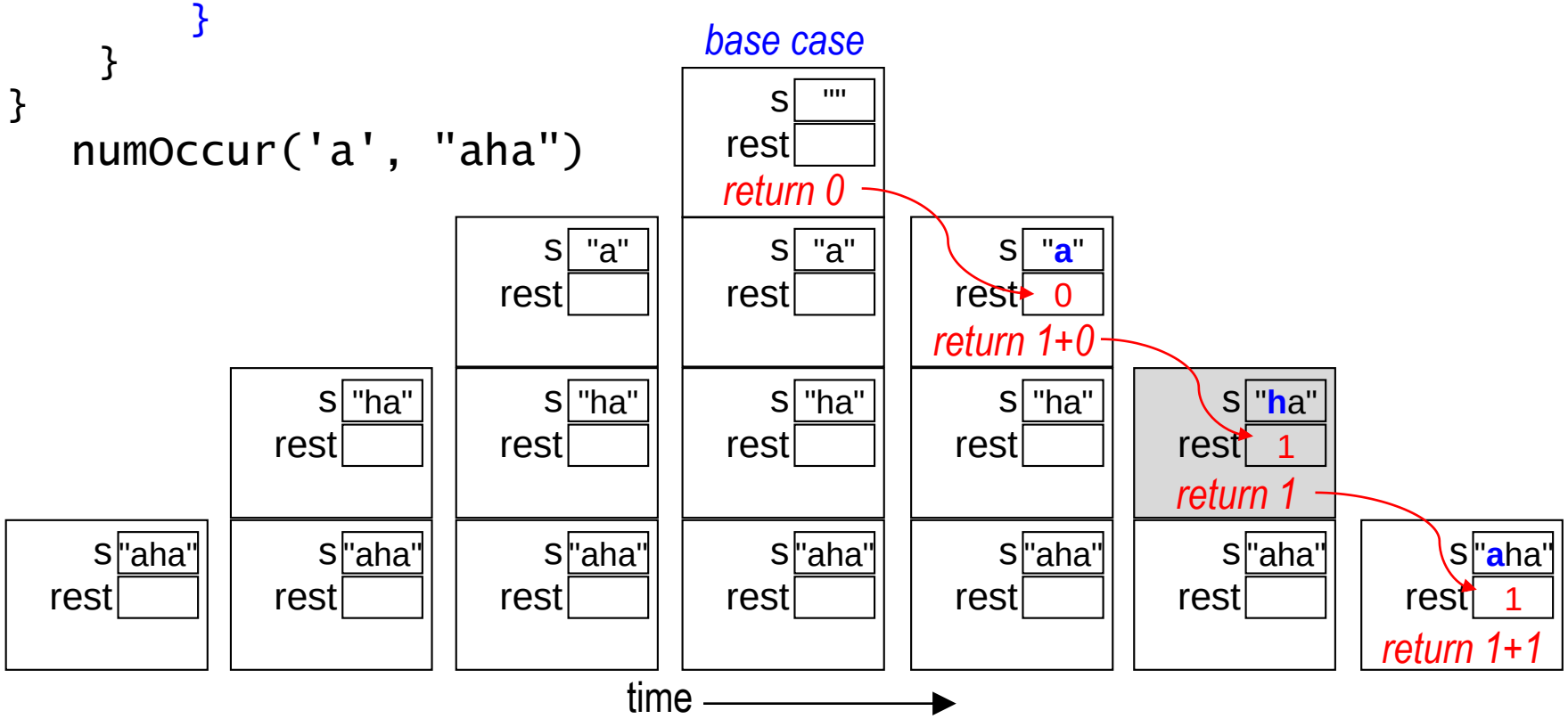
numOccur('a', "aha")



Tracing a Recursive Method on the Stack

```
public static int numOccur(char ch, String s) {
    if (s == null || s.equals("")) {
        return 0;
    } else {
        int rest = numOccur(ch, s.substring(1));
        if (s.charAt(0) == ch) {
            return 1 + rest;
        } else {
            return rest;
        }
    }
}
```

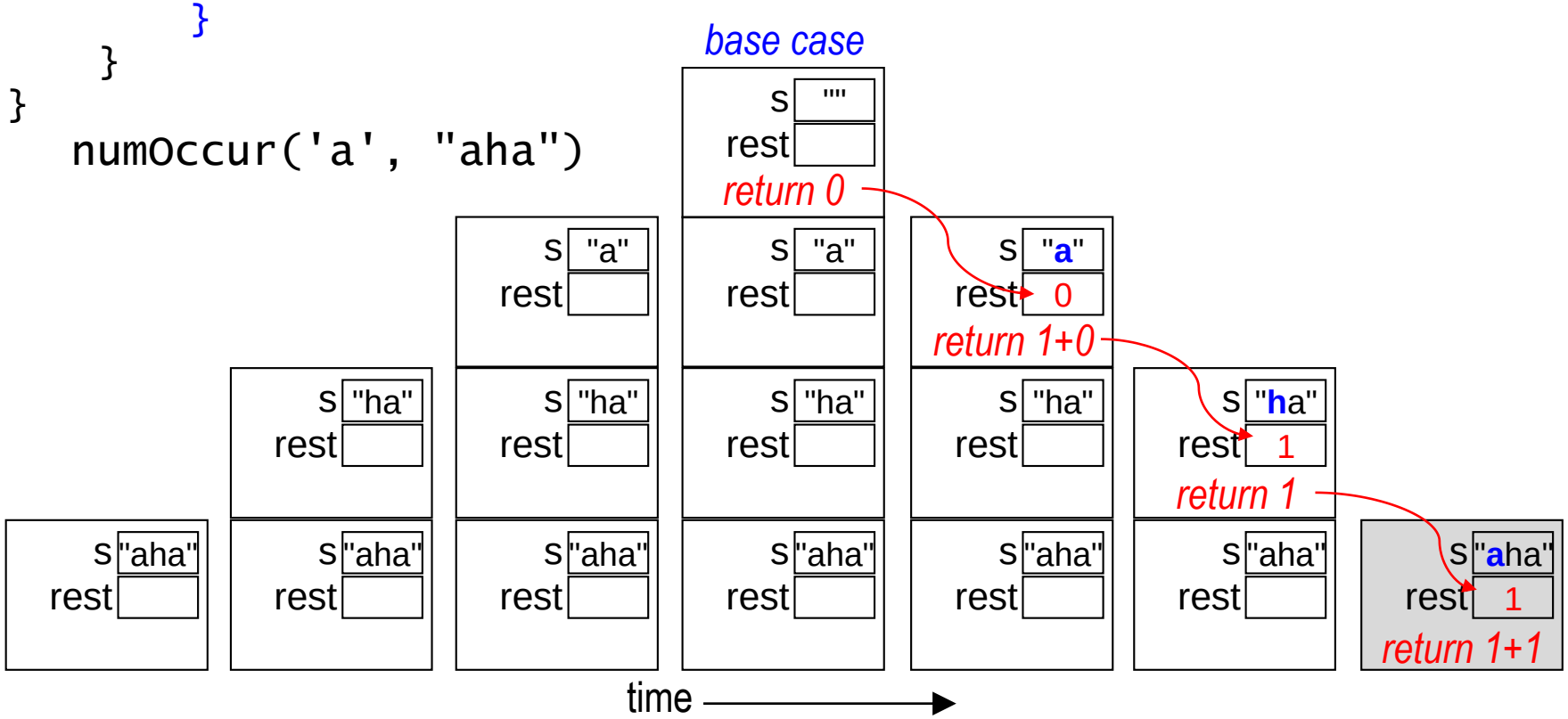
numOccur('a', "aha")



Tracing a Recursive Method on the Stack

```
public static int numOccur(char ch, String s) {
    if (s == null || s.equals("")) {
        return 0;
    } else {
        int rest = numOccur(ch, s.substring(1));
        if (s.charAt(0) == ch) {
            return 1 + rest;
        } else {
            return rest;
        }
    }
}
```

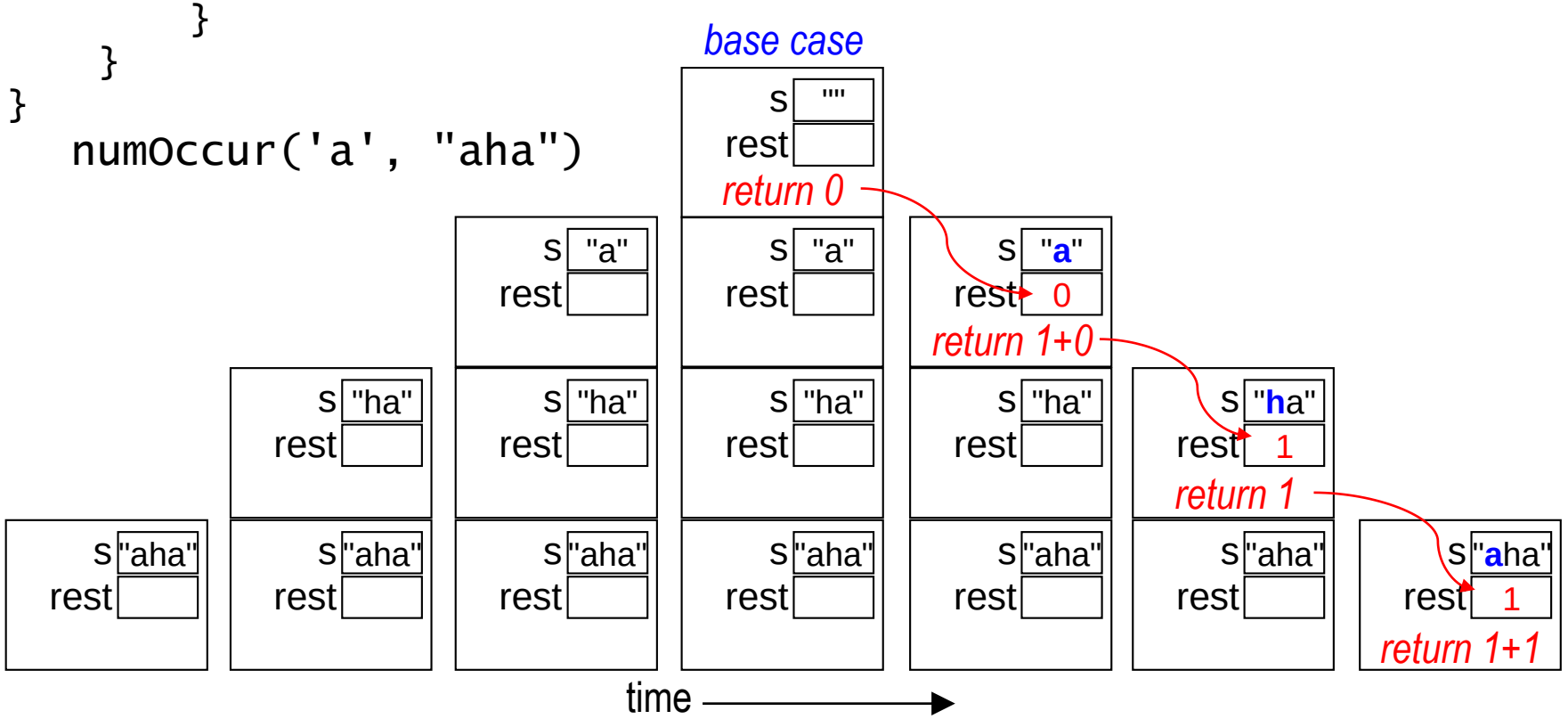
```
numOccur('a', "aha")
```



Tracing a Recursive Method on the Stack

```
public static int numOccur(char ch, String s) {
    if (s == null || s.equals("")) {
        return 0;
    } else {
        int rest = numOccur(ch, s.substring(1));
        if (s.charAt(0) == ch) {
            return 1 + rest;
        } else {
            return rest;
        }
    }
}
```

```
numOccur('a', "aha")
```



Common Mistake(s)

- This version of the method does ***not*** work:

```
public static int numOccur(char ch, String s) {
    if (s == null || s.equals("")) {
        return 0;
    } else {
        int count = 0;
        if (s.charAt(0) == ch) {
            count++;
        }
        numOccur(ch, s.substring(1));
        return count;
    }
}
```

Common Mistake(s)

- This version of the method does *not* work:

```
public static int numOccur(char ch, String s) {
    if (s == null || s.equals("")) {
        return 0;
    } else {
        int count = 0;
        if (s.charAt(0) == ch) {
            count++;
        }
        numOccur(ch, s.substring(1));
        return count;
    }
}
```

Creating a local variable count within each function call. The same value will be assigned and returned by each function!

True or False

We can fix the prior problem by passing in the variable count!

- This version of the method **does** work?

```
public static
int numOccur(char ch, String s, int count ) {
    if (s == null || s.equals("")) {
        return 0;
    } else {

        if (s.charAt(0) == ch) {
            count++;
        }
        numOccur(ch, s.substring(1), count);
        return count;
    }
}
```

A. True

B. False, parameters to a function are local to the function!

Another Faulty Approach

- Some people make `count` "*global*" to fix the prior version:

```
public static int count = 0;

public static int numOccur(char ch, String s) {
    if (s == null || s.equals("")) {
        return 0;
    } else {
        if (s.charAt(0) == ch) {
            count++;
        }
        numOccur(ch, s.substring(1));
        return count;
    }
}
```

- Not recommended, and not allowed on the problem sets!

- Problems with this approach?

An alternative version…

```
public static int numOccur(char ch, String s) {
      int rest = 0;      // assign base case return
      if (s !=  null && !s.equals("")) {
          // recursive case
          rest = numOccur(ch, s.substring(1));
           // solution forms as the recursion unwinds
          if (s.charAt(0) == ch)
              rest += 1;
      }
      // return the solution to each sub-problem
      return( rest );
   }
```

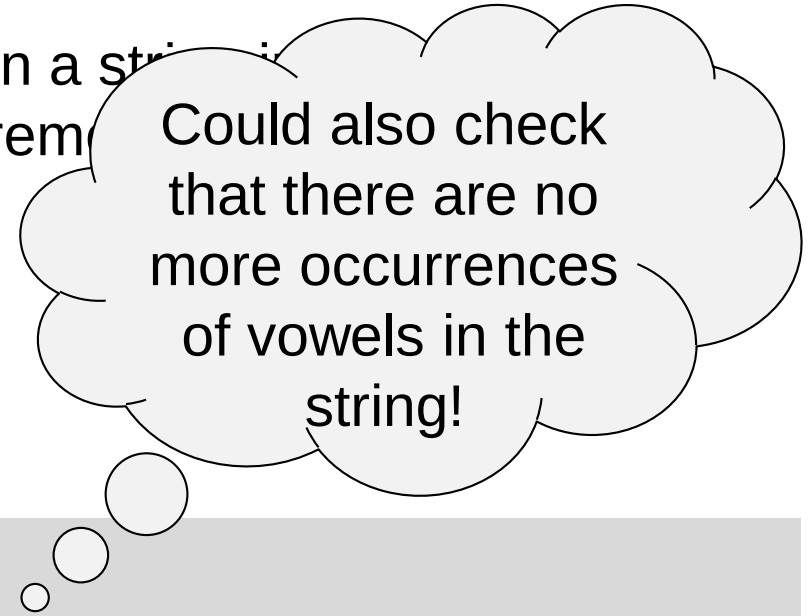
Removing Vowels from a String

- Let's design a recursive method called `removeVowels()`.
- `removeVowels(str)` should return a str
  vowels in the string `s` have been rem
  - example:

    `removeVowels("recurse")`

    should return

    `"rcrs"`



- Thinking recursively:
  - What is the base case? The empty string!
  - What is the recursive case? A recursive call with a *substring* of the input string.
  - **What should be returned in each case?**

removeVowels()

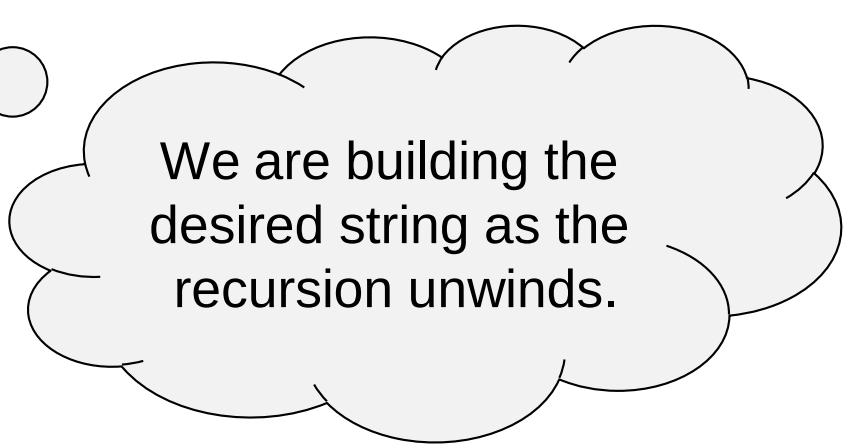
```
public static String removeVowels(String s) {

    if (s.equals("")) {         // base case
        return("");
    } else {                    // recursive case
        String rem_rest = removeVowels(s.substring(1));

        if ("aeiou".indexOf(s.charAt(0)) == -1) {
            return( s.charAt(0) + rem_rest );
        } else
            return rem_rest;
    }
}
```

removeVowels()

```
public static String removeVowels(String s) {

    if (s.equals("")) {         // base case
        return("");
    } else {                    // recursive case
        String rem_rest = removeVowels(s.substring(1));

        if ("aeiou".indexOf(s.charAt(0)) == -1) {
            return( s.charAt(0) + rem_rest );
        } else
            return rem_rest;
    }
}
```

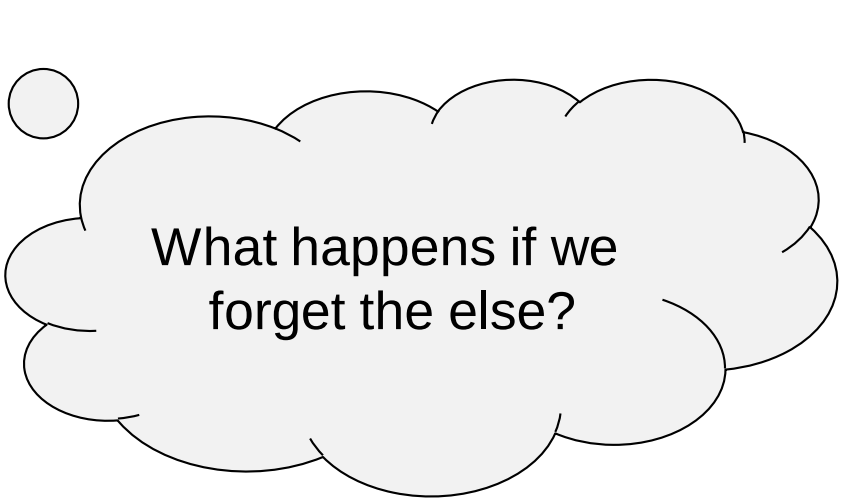
removeVowels()

```
public static String removeVowels(String s) {

    if (s.equals("")) {          // base case
        return("");
    } else {                     // recursive case
        String rem_rest = removeVowels(s.substring(1));

        if ("aeiou".indexOf(s.charAt(0)) == -1) {
            return( s.charAt(0) + rem_rest );
        } else
            return rem_rest;
    }
}
```



removeVowels()

```
public static String removeVowels(String s) {

    if (s.equals("")) {          // base case
        return("");
    } else {                     // recursive case
        String rem_rest = removeVowels(s.substring(1));

        if ("aeiou".indexOf(s.charAt(0)) == -1) {
            return( s.charAt(0) + rem_rest );
        }

    }
}
```



removeVowels()

an alternative

```
public static String removeVowels(String s) {
    String rem_rest;              // return variable

    if (s.equals("")) {           // base case
        rem_rest = "";
    } else {                      // recursive case
        rem_rest = removeVowels(s.substring(1));

        if ("aeiou".indexOf(s.charAt(0)) == -1) {
            rem_rest = s.charAt(0) + rem_rest;
        }
    }

    return(rem_rest);
}
```

removeVowels()

an alternative

```
public static String removeVowels(String s) {
    String rem_rest;            // return variable

    if (s.equals("")) {         // base case
        rem_rest = "";
    } else {                    // recursive case
        rem_rest = remov

        if ("aeiou".inde
            rem_rest = s
        }
    }

    return(rem_rest);
}
```

Note that we are **not** guarding against the method being called with a null string!

removeVowels()

an alternative

```
public static String removeVowels(String s) {
    String rem_rest;            // return variable

    if (s == null || s.equals("")) { // base case
        rem_rest = "";
    } else {                    // recursive case
        rem_rest = remov

        if ("aeiou".inde
            rem_rest = s
        }
    }

    return(rem_rest);
}
```

If the ***first comparison*** is true (i.e. the string is null), the second relation is not tested!

removeVowels()
an alternative

```
public static String removeVowels(String s) {
    String rem_rest;            // return variable

    if (s == null || s.equals("")) { // base case
        rem_rest = "";
    } else {                    // recursive case
        rem_rest = remov

        if ("aeiou".inde
            rem_rest = s
        }
    }

    return(rem_rest);
}
```

If the first comparison is true (i.e. the string is null), ***the second relation*** is not tested!

removeVowels()

an alternative

```
public static String removeVowels(String s) {
    String rem_rest;            // return variable

    if (s.equals("") || s == null) { // base case
        rem_rest = "";
    } else {                    // recursive case
        rem_rest = remov

        if ("aeiou".inde
            rem_rest = s
        }
    }

    return(rem_rest);
}
```

What would happen if we reversed the comparison and the string was null?

removeVowels()

an alternative

```
public static String removeVowels(String s) {
    String rem_rest;            // return variable

    if (s.equals("") || s == null) { // base case
        rem_rest = "";
    } else {                    // recursive case
        rem_rest = remov

        if ("aeiou".inde
            rem_rest = s
        }
    }

    return(rem_rest);
}
```

Java null exception error if a null string s passed!

removeVowels()

another alternative

```
public static String removeVowels(String s) {
    String rem_rest = "";     // assign base return

    if (s != null && !s.equals("")) {
        // recursive case
        rem_rest = removeVowels(s.substring(1));

        if ("aeiou".indexOf(s.charAt(0)) == -1) {
            rem_rest = s.charAt(0) + rem_rest;
        }
    }

    // return the result of each sub-problem
    return(rem_rest);
}
```

Raising a Number to a Power

- We want to write a recursive method to compute

$$x^n = \underbrace{x*x*x*\ldots*x}_{n \text{ of them}}$$

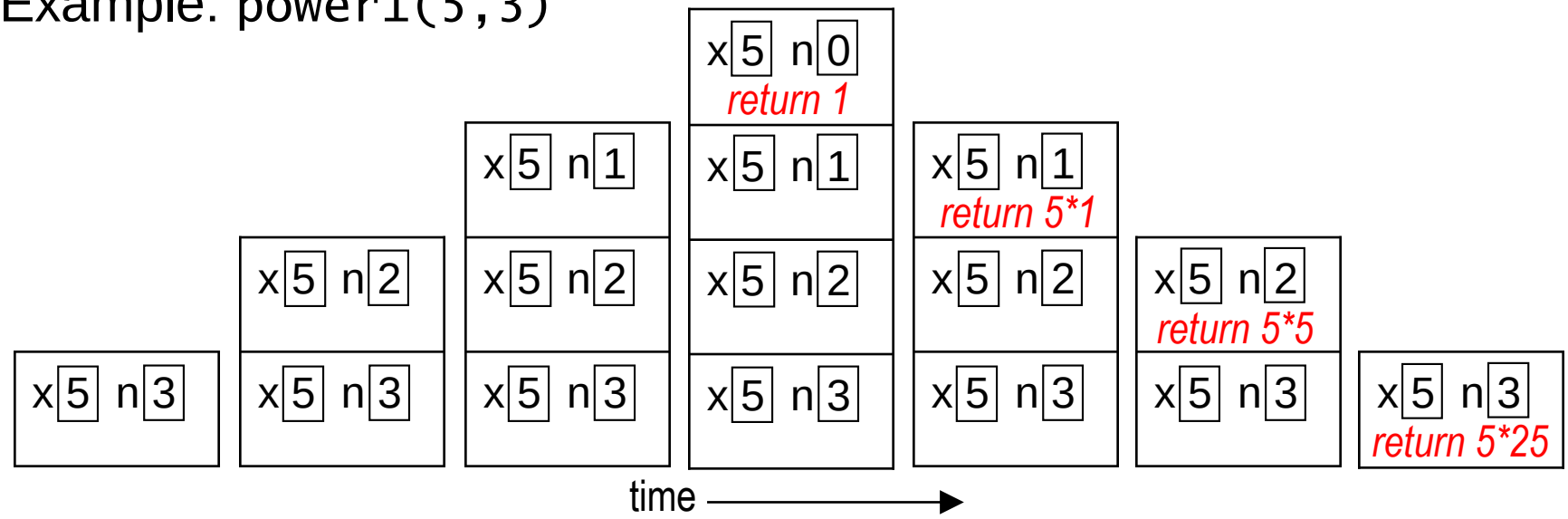
  where x and n are both integers and $n >= 0$.

- Examples:
  - $2^{10} = 2*2*2*2*2*2*2*2*2*2 = 1024$
  - $10^5 = 10*10*10*10*10 = 100000$
- Computing a power recursively: $2^{10} = 2*2^9$
  $= 2*(2 * 2^8)$
  $= \ldots$
- Recursive definition: $x^n = x * x^{n-1}$ when $n > 0$
  $x^0 = 1$

Power Method: First Try

```
public static int power1(int x, int n) {
    if (n < 0) {
        throw new IllegalArgumentException();
    } else if (n == 0) {
        return 1;
    } else {
        int pow_rest = power1(x, n-1);
        return x * pow_rest;
    }
}
```

Example: power1(5,3)



Power Method: Second Try

- There's a better way to break these problems into subproblems.
  For example: $2^{10} = (2*2*2*2*2)*(2*2*2*2*2)$
  $= (2^5) * (2^5) = (2^5)^2$

- A more efficient recursive definition of x^n (when $n > 0$):
  $x^n = (x^{n/2})^2$ when n is even
  $x^n = x * (x^{n/2})^2$ when n is odd (using integer division for n/2)

```
public static int power2(int x, int n) {
    // code to handle n < 0 goes here...
    if (n == 0) {
        return 1;
    } else {
        int pow_rest = power2(x, n/2);
        if (n % 2 == 0) {
            return pow_rest * pow_rest;
        } else {
            return x * pow_rest * pow_rest;
        }
    }
}
```

Analyzing power2

- How many method calls would it take to compute 2^{1000} ?

```
power2(2, 1000)
  power2(2, 500)
    power2(2, 250)
      power2(2, 125)
        power2(2, 62)
          power2(2, 31)
            power2(2, 15)
              power2(2, 7)
                power2(2, 3)
                  power2(2, 1)
                    power2(2, 0)
```

- Much more efficient than power1() for large n.
- It can be shown that it takes approx. $\log_2 n$ method calls.

An Inefficient Version of power2

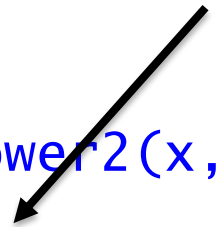
- What's wrong with the following version of power2()?

```
public static int power2(int x, int n) {
    // code to handle n < 0 goes here...
    if (n == 0) {
        return 1;
    } else {
        // int pow_rest = power2(x, n/2);
        if (n % 2 == 0) {
            return power2(x, n/2) * power2(x, n/2);
        } else {
            return x * power2(x, n/2) * power2(x, n/2);
        }
    }
}
```

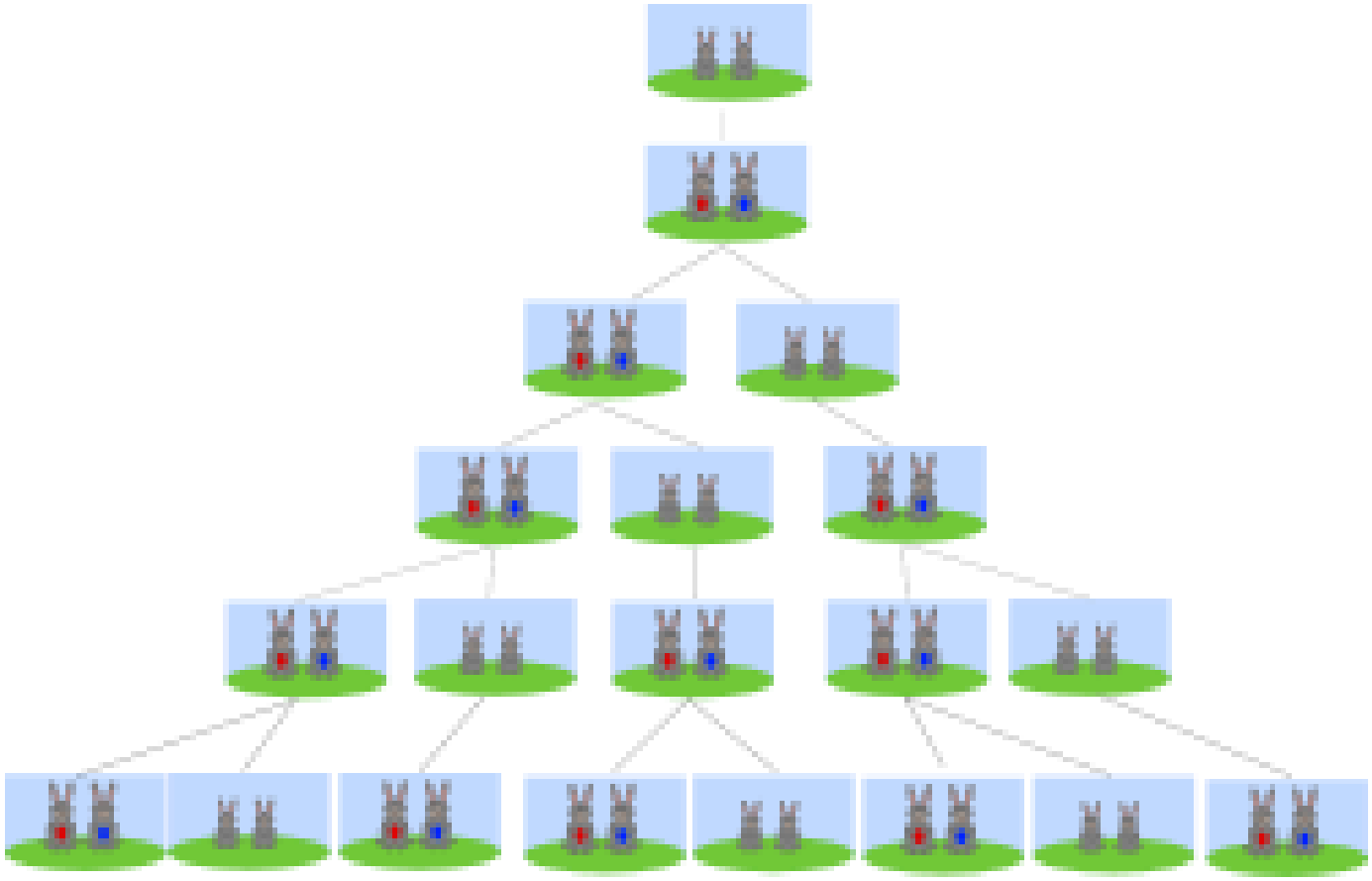
An Inefficient Version of power2

- What's wrong with the following version of power2()?

```
public static int power2(int x, int n) {
    // code to handle n < 0 goes here...
    if (n == 0) {
        return 1;
    } else {
        // int pow_rest = power2(x, n/2);
        if (n % 2 == 0) {
            return power2(x, n/2) * power2(x, n/2);
        } else {
            return x * power2(x, n/2) * power2(x, n/2);
        }
    }
}
```

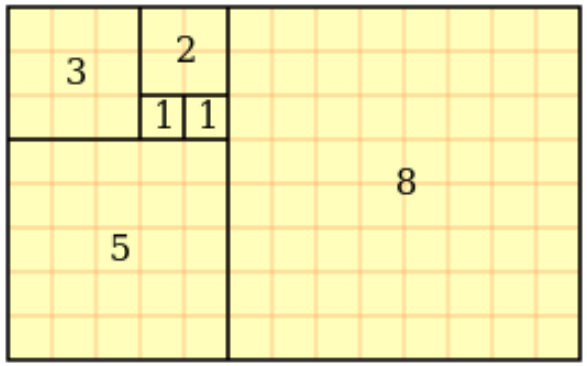


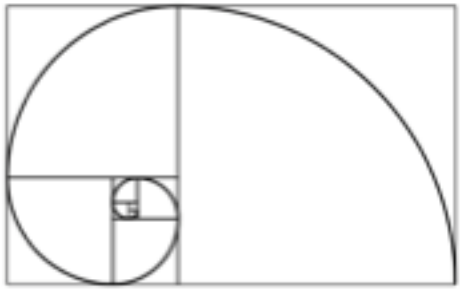
Fibonnacci Number Series

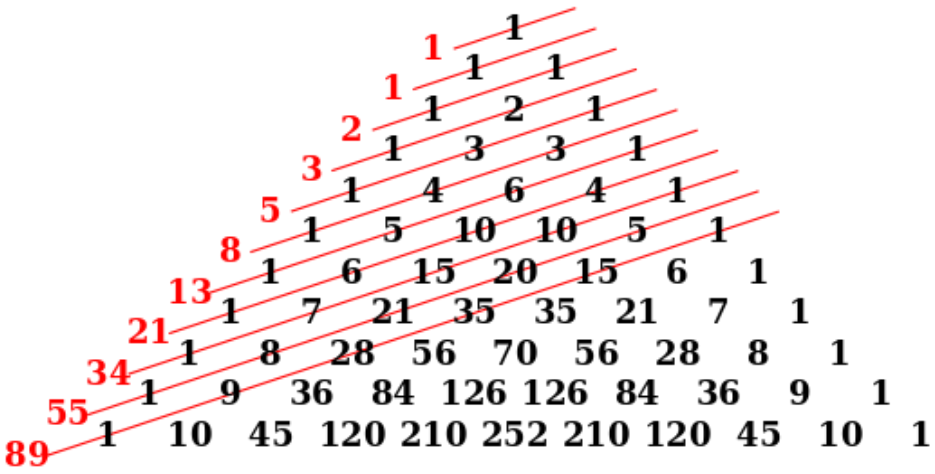


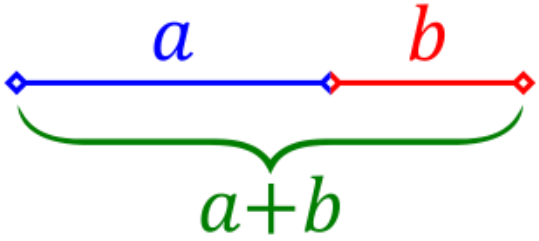
Recursion: Fibonacci

Digression: The Fibonacci Numbers have a long history; the earliest mention is in an analysis of Sanskrit poetry, c. 200 AD, but the name comes from **Leonardo of Pisa**, better known as Fibonacci, who invented them to explain the geometric growth of a family of rabbits. It has many interesting mathematical properties.....









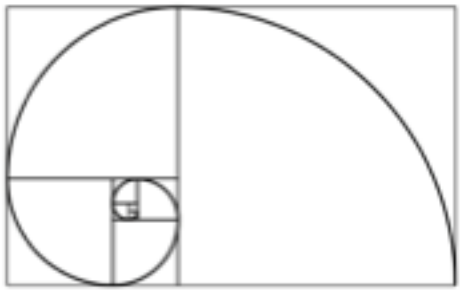
$a+b$ is to a as a is to b

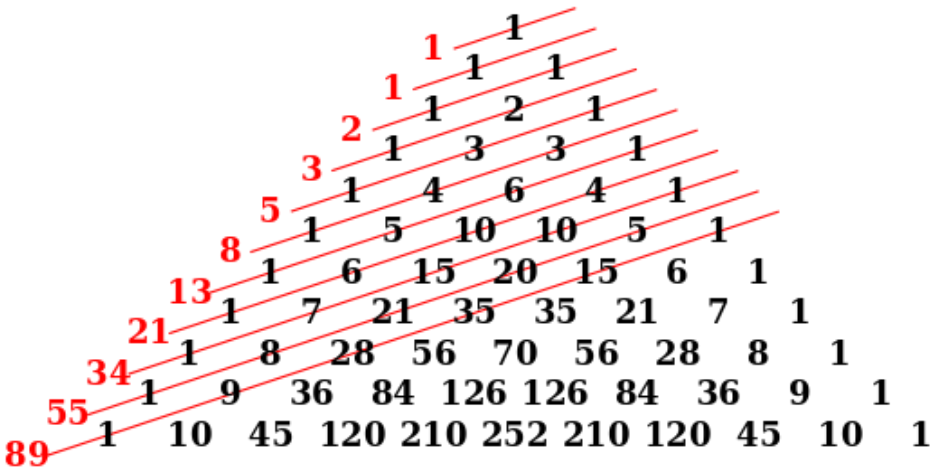
Golden Ratio: 1.6180339....

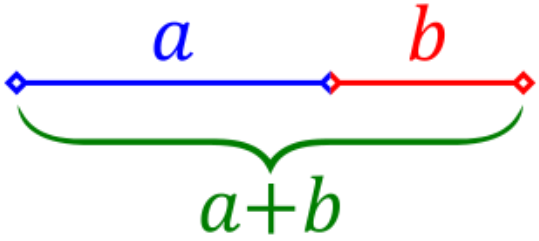
Recursion
F

In 1202 wrote
Liber Abaci or
Book of Calculations

Digression: The Fibonacci Numbers ha
a long history; the earliest mention is in
an analysis of Sanskrit poetry, c. 200 AD
but the name comes from **Leonardo of Pisa**,
better known as **Fibonacci**, who invented
them to explain the **geometric growth**
of a family of rabbits. It has many interesting
mathematical properties.....







$a+b$ is to a as a is to b

Golden Ratio: 1.6180339....

Recursion: Fibonacci

Let's look at another example, the Fibonacci numbers, defined (two ways) as follows:

F = { 1, 1, 2, 3, 5, 8, 13, 21, ..., **(sum of previous 2 terms)**, ... }

0 1 2 3 4 5 6 7

```
// returns the ith Fibonacci number

    int fib(int i) {
       int lim = 2;
       if( i < lim )
          return 1;
       else
          return fib(i-1) + fib(i-2);
    }
```

Recursion:
Fibonacci

Let's look at another example, the Fibonacci numbers, defined (two ways) as follows:

F = { 1, 1, 2, 3, 5, **8**, 13, 21, …., **(sum of previous 2 terms)**, … }

0 1 2 3 4 **5** 6 7

```
// returns the ith Fibonacci number

    int fib(int i) {
       int lim = 2;
       if( i < lim )
          return 1;
       else
          return fib(i-1) + fib(i-2);
    }
```

Recursion:
Fibonacci

Let's look at another example, the Fibonacci numbers, defined (two ways) as follows:

F = { 1, 1, 2, 3, **5**, 8, 13, 21, ..., **(sum of previous 2 terms)**, ... }

0 1 2 3 **4** 5 6 7

```
// returns the ith Fibonacci number

    int fib(int i) {
       int lim = 2;
       if( i < lim )
          return 1;
       else
          return fib(i-1) + fib(i-2);
    }
```

Recursion:
Fibonacci

Let's look at another example, the Fibonacci numbers, defined (two ways) as follows:

F = { 1, 1, 2, **3**, 5, 8, 13, 21, …., **(sum of previous 2 terms)**, … }

0 1 2 **3** 4 5 6 7

```
// returns the ith Fibonacci number

    int fib(int i) {
       int lim = 2;
       if( i < lim )
          return 1;
       else
          return fib(i-1) + fib(i-2);
    }
```

Recursion: Fibonacci

Let's look at another example, the Fibonacci numbers, defined (two ways) as follows:

F = { 1, 1, **2**, 3, 5, 8, 13, 21, ..., (sum of previous 2 terms), ... }

0 1 **2** 3 4 5 6 7

```
// returns the ith Fibonacci number

    int fib(int i) {
       int lim = 2;
       if( i < lim )
          return 1;
       else
          return fib(i-1) + fib(i-2);
    }
```

Recursion:
Fibonacci

Let's look at another example, the Fibonacci numbers, defined (two ways) as follows:

F = { **1**, **1**, 2, 3, 5, 8, 13, 21, ..., (sum of previous 2 terms), ... }

0 **1** 2 3 4 5 6 7

```
// returns the ith Fibonacci number

    int fib(int i) {
       int lim = 2;
       if( i < lim )
          return 1;
       else
          return fib(i-1) + fib(i-2);
    }
```

Recursion:
Fibonacci

Let's look at another example, the Fibonacci numbers, defined (two ways) as follows:

F = { 1, 1, 2, 3, 5, 8, 13, 21, ..., **(sum of previous 2 terms)**, ... }

0 1 2 3 4 5 6 7

```
// returns the ith Fibonacci number

    int fib(int i) {
       int lim = 2;
       if( i < lim )
          return 1;
       else
          return fib(i-1) + fib(i-2);
    }
```

Recursion:
Fibonacci

Let's look at another example, the Fibonacci numbers, defined (two ways) as follows:

F = { 1, 1, 2, 3, 5, 8, 13, 21, ..., **(sum of previous 2 terms)**, ... }

0 1 2 3 4 5 6 7

```
// returns the ith Fibonacci number

    int fib(int i) {
       int lim = 2;
       if( i < lim )
          return 1;
       else
          return fib(i-1) + fib(i-2);
    }
```

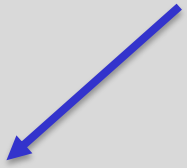
Recursion:
Fibonacci

Let's look at another example, the Fibonacci numbers, defined (two ways) as follows:

F = { 1, 1, 2, 3, 5, 8, 13, 21, ..., **(sum of previous 2 terms)**, ... }

0 1 2 3 4 5 6 7

```
// returns the ith Fibonacci number

    int fib(int i) {
       int lim = 2;
       if( i < lim )
          return 1;
       else
          return fib(i-1) + fib(i-2);
    }
```



Recursion:
Fibonacci

Let's look at another example, the Fibonacci numbers, defined (two ways) as follows:

F = { 1, 1, 2, 3, 5, 8, 13, 21, ..., **(sum of previous 2 terms)**, ... }

0 1 2 3 4 5 6 7

```
// returns the ith Fibonacci number

    int fib(int i) {
       int lim = 2;
       if( i < lim )
          return 1;
       else
          return fib(i-1) + fib(i-2);
    }
```

Recursion:
Fibonacci

Let's look at another example, the Fibonacci numbers, defined (two ways) as follows:

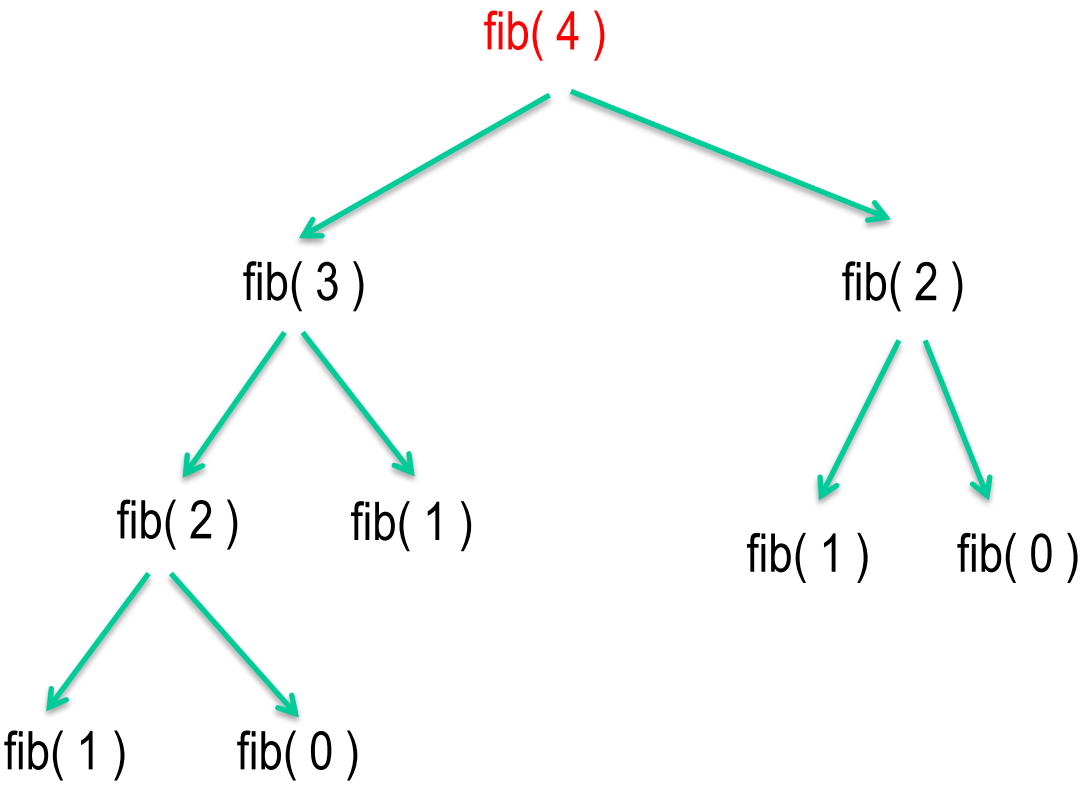
F = { 1, 1, 2, 3, 5, 8, 13, 21, ..., **(sum of previous 2 terms)**, ... }

0 1 2 3 4 5 6 7

```
// returns the ith Fibonacci number

    int fib(int i) {
       int lim = 2;
       if( i < lim )
          return 1;
       else
          return fib(i-1) + fib(i-2);
    }
```

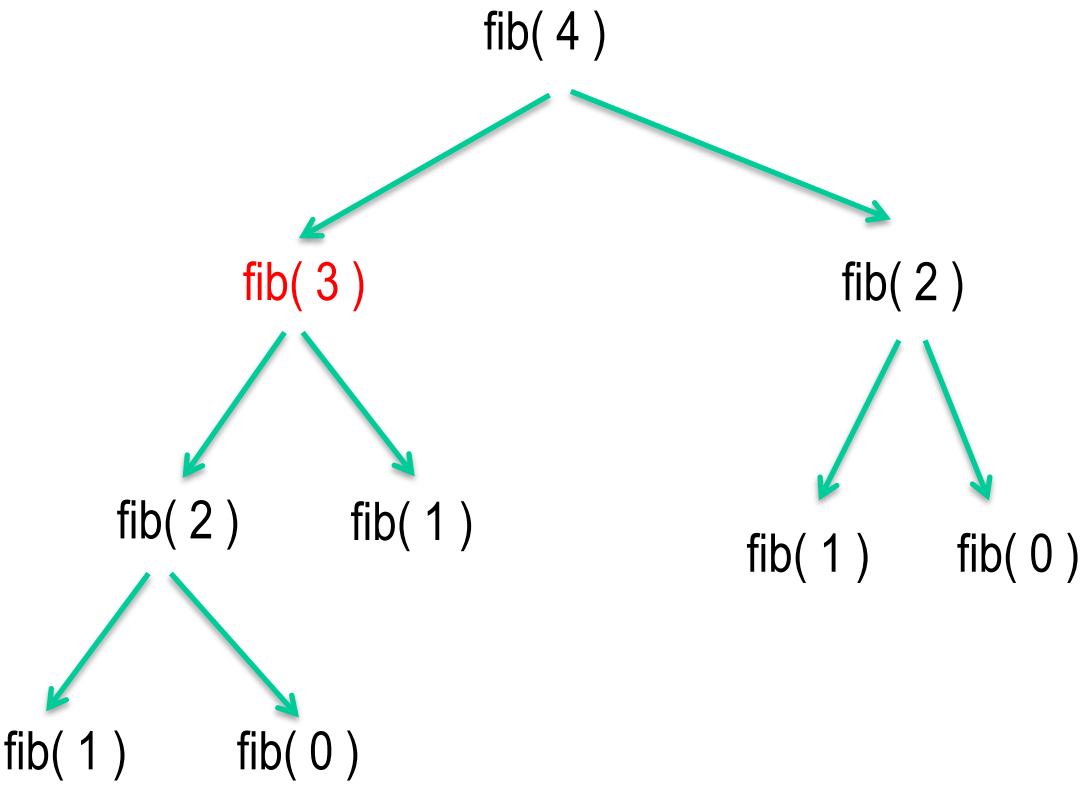
Recursion: Fibonacci

Tree of Calls to fib(4)



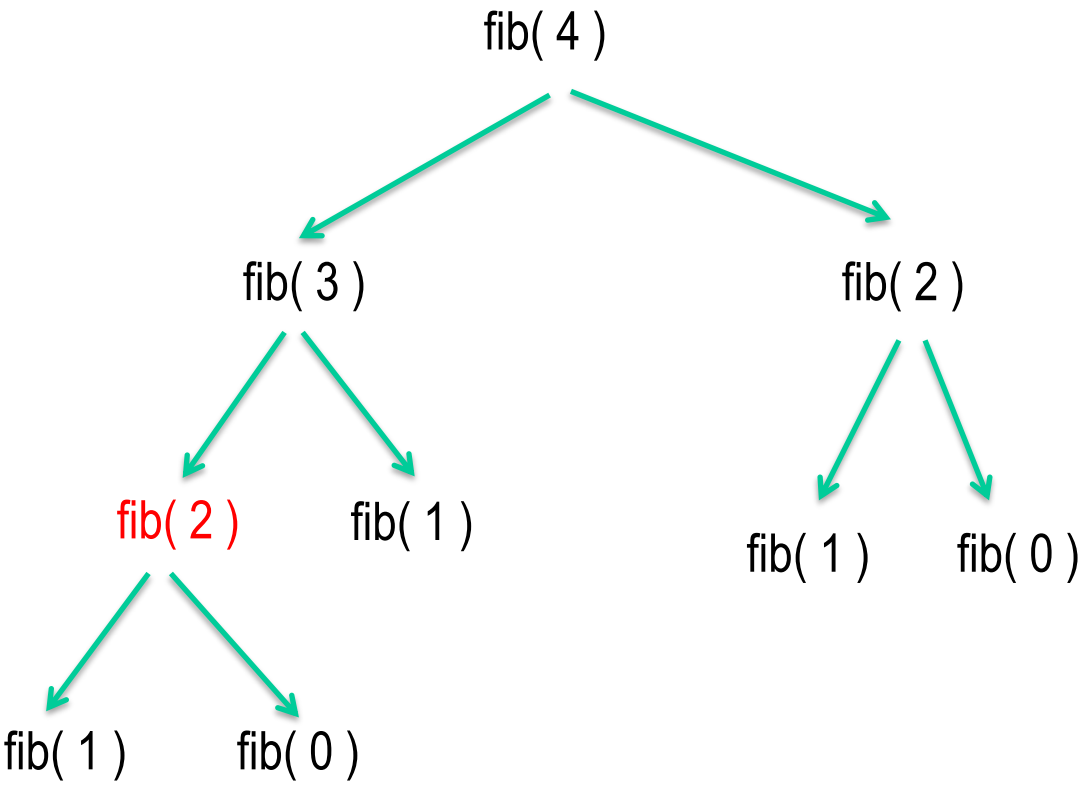
Recursion: Fibonacci

Tree of Calls to fib(4)



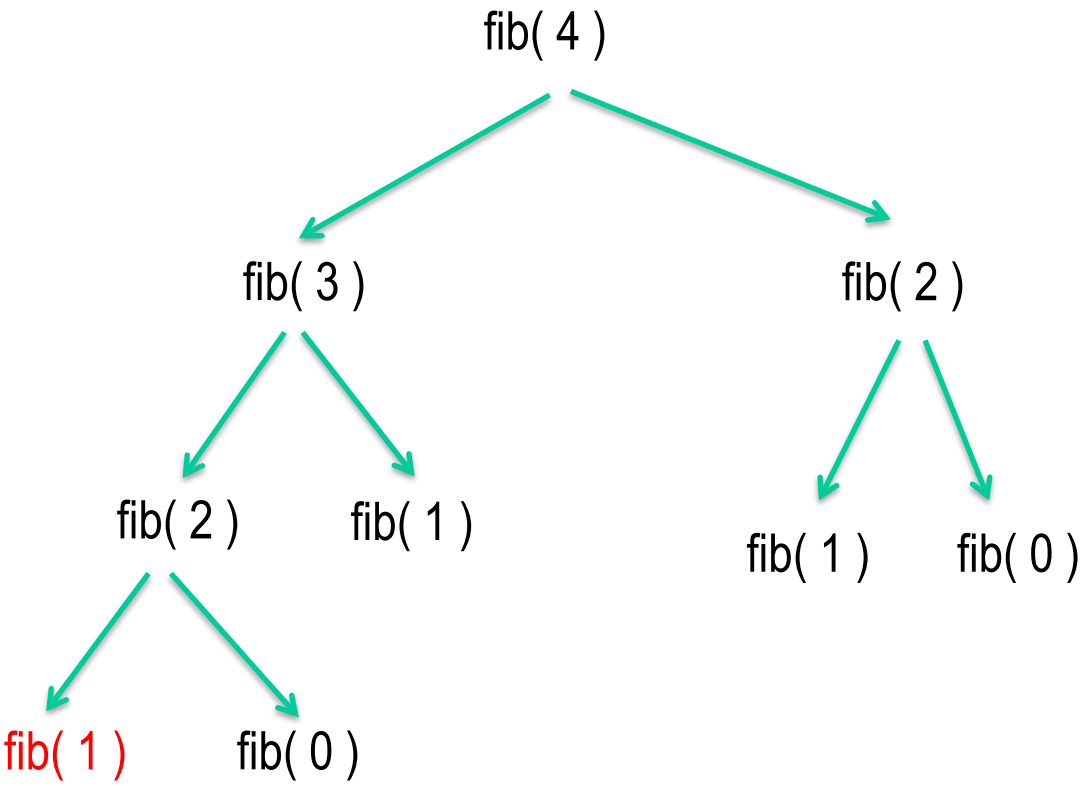
Recursion: Fibonacci

Tree of Calls to fib(4)



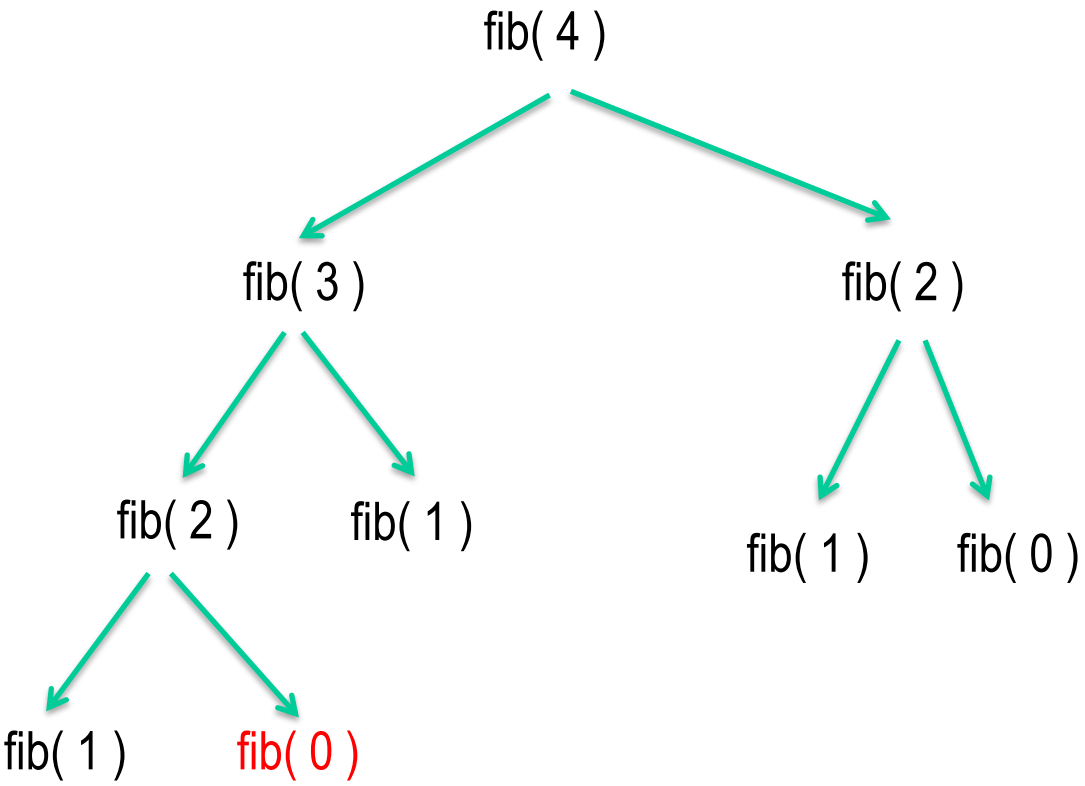
Recursion: Fibonacci

Tree of Calls to fib(4)



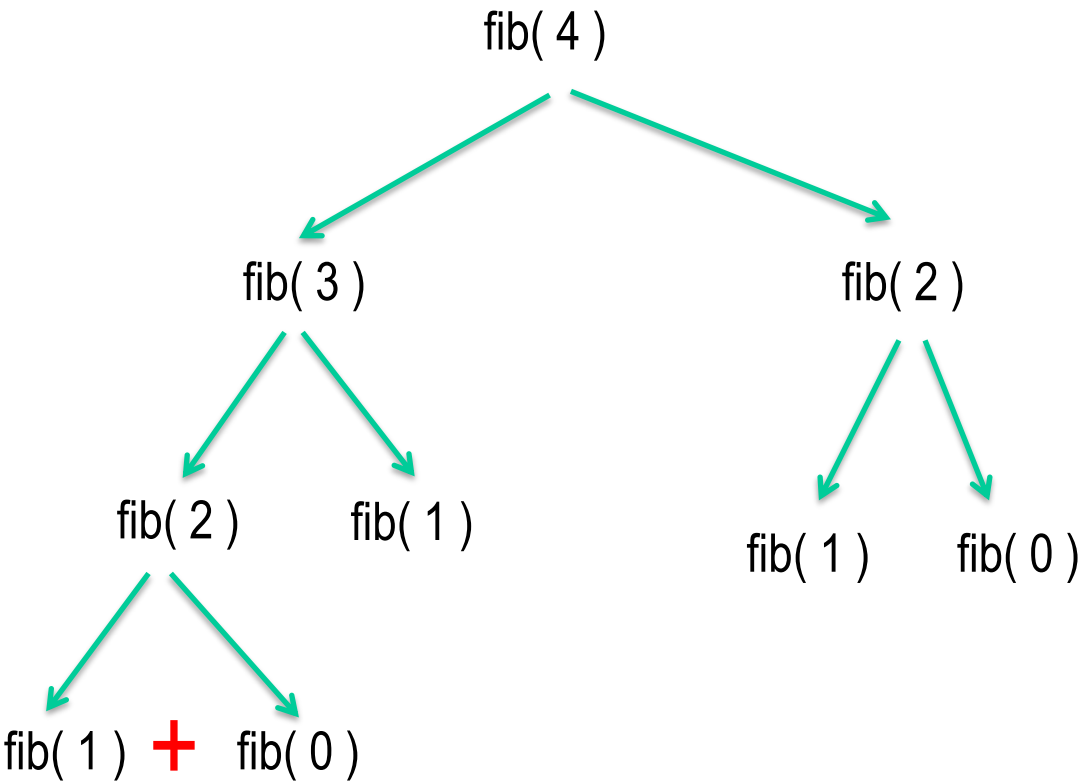
Recursion: Fibonacci

Tree of Calls to fib(4)



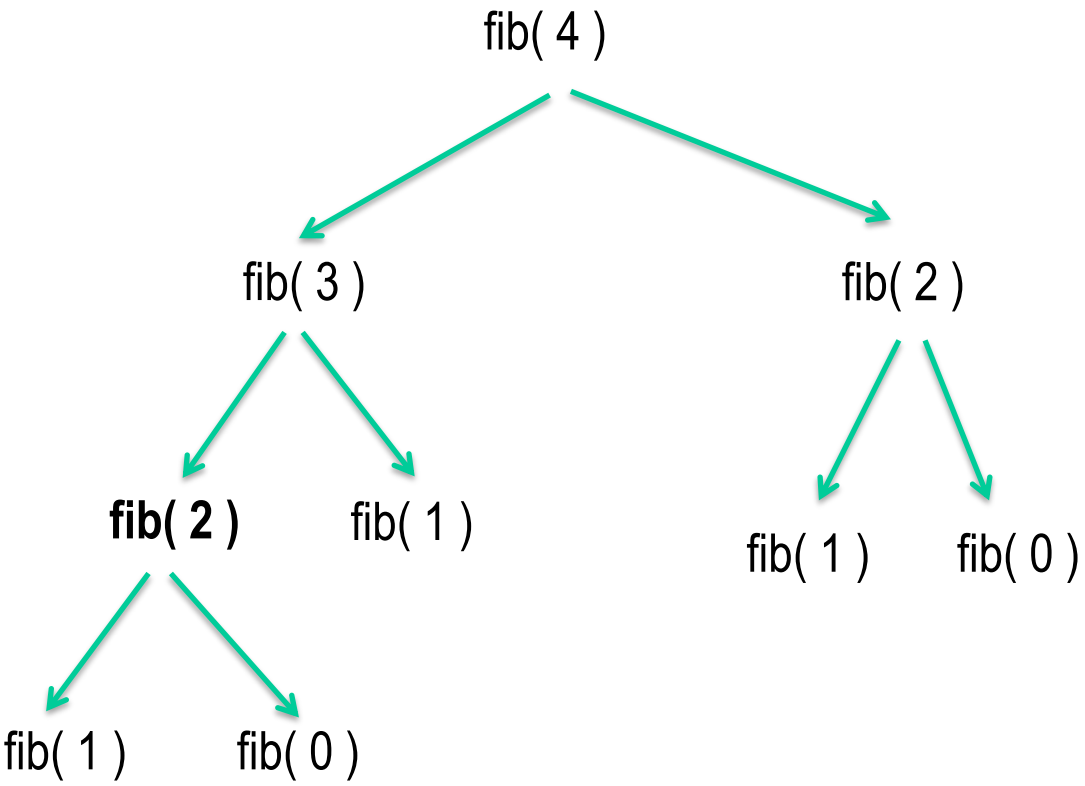
Recursion: Fibonacci

Tree of Calls to fib(4)



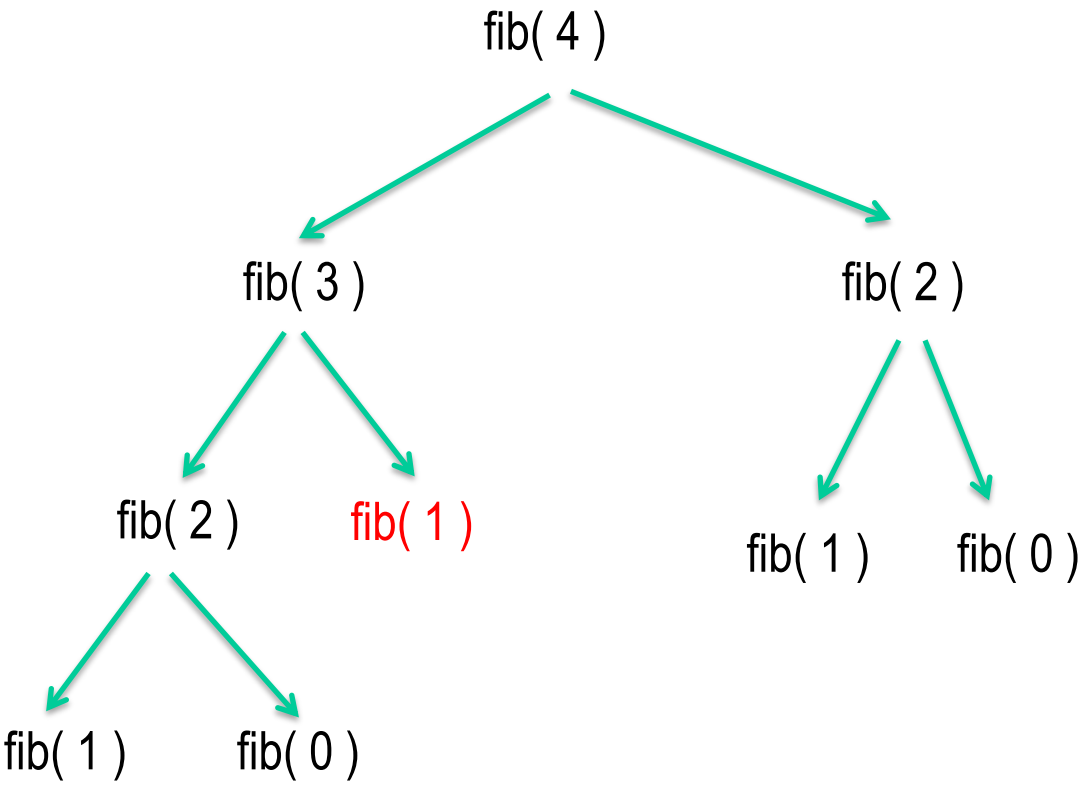
Recursion: Fibonacci

Tree of Calls to fib(4)



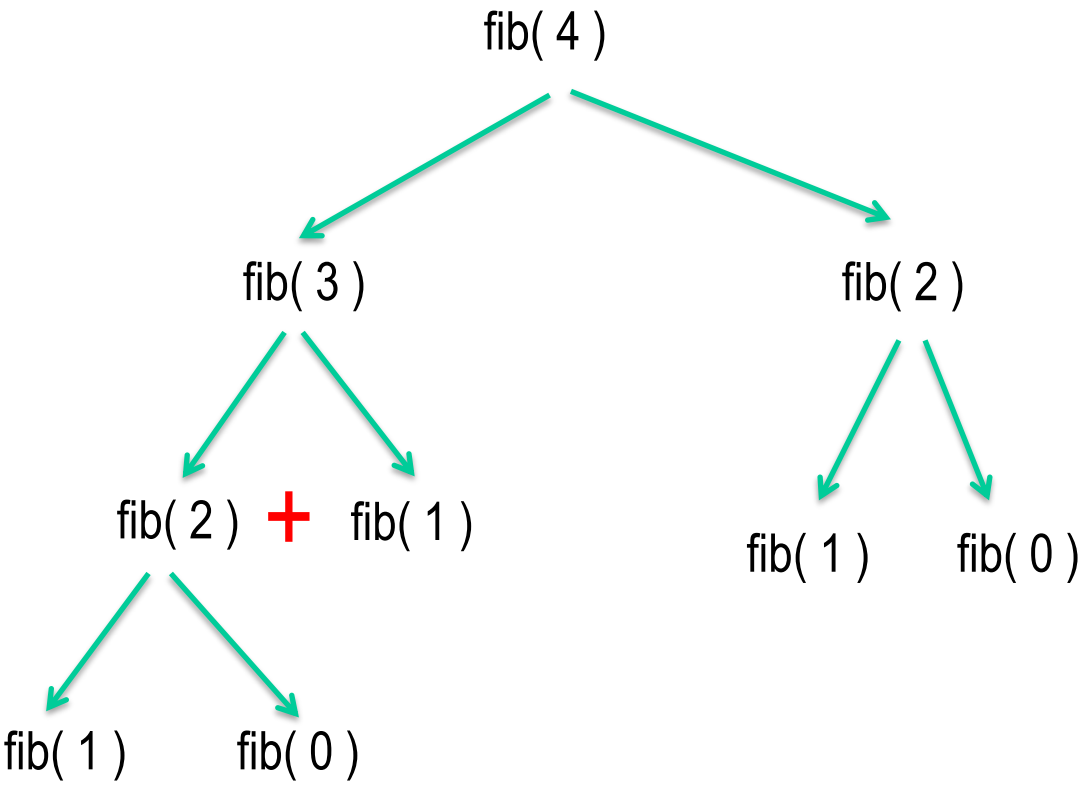
Recursion: Fibonacci

Tree of Calls to fib(4)



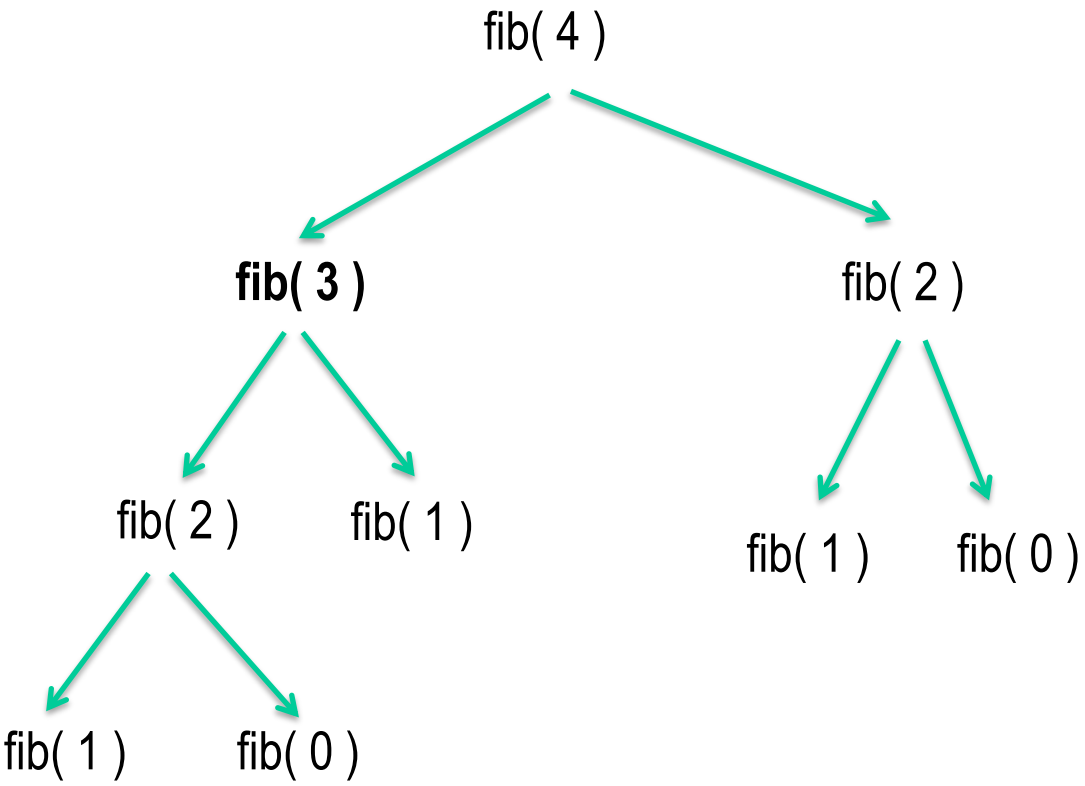
Recursion: Fibonacci

Tree of Calls to fib(4)



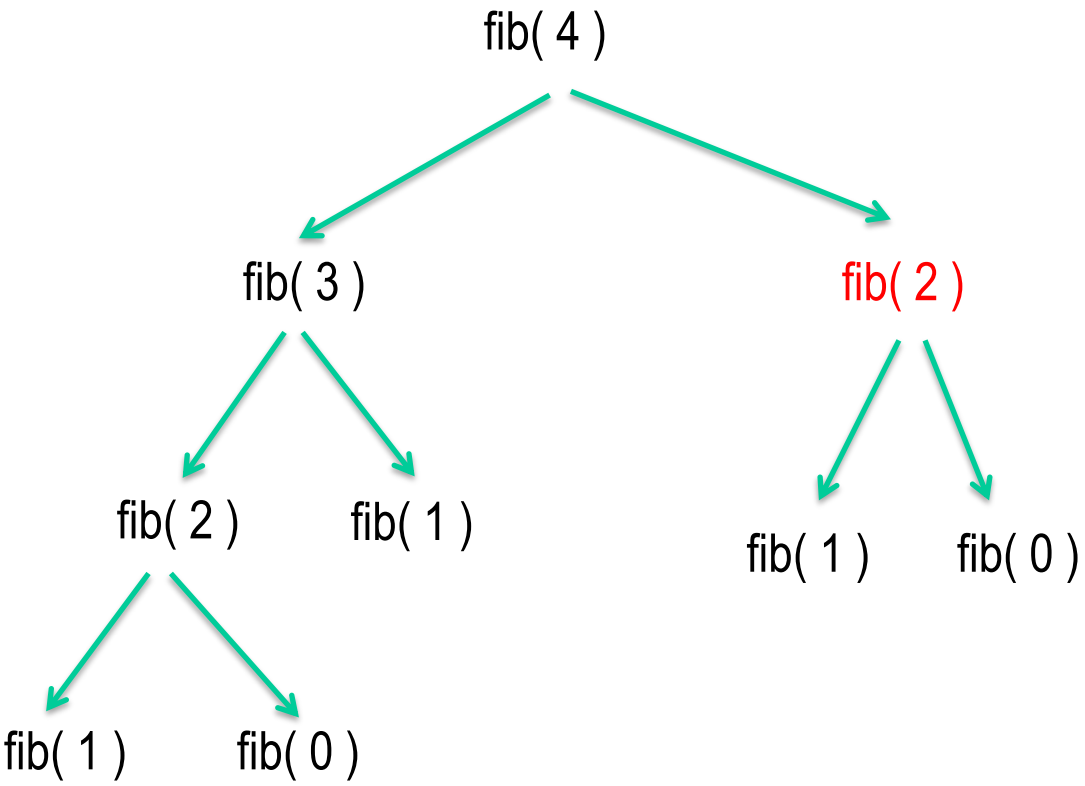
Recursion: Fibonacci

Tree of Calls to fib(4)



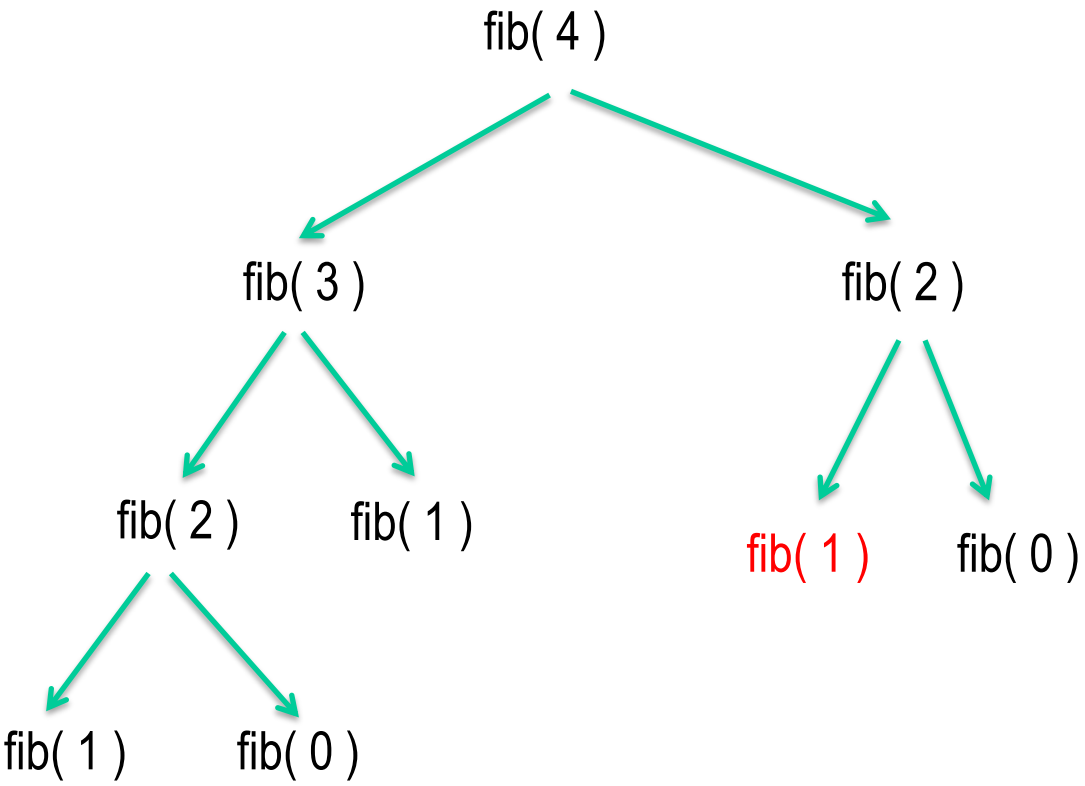
Recursion: Fibonacci

Tree of Calls to fib(4)



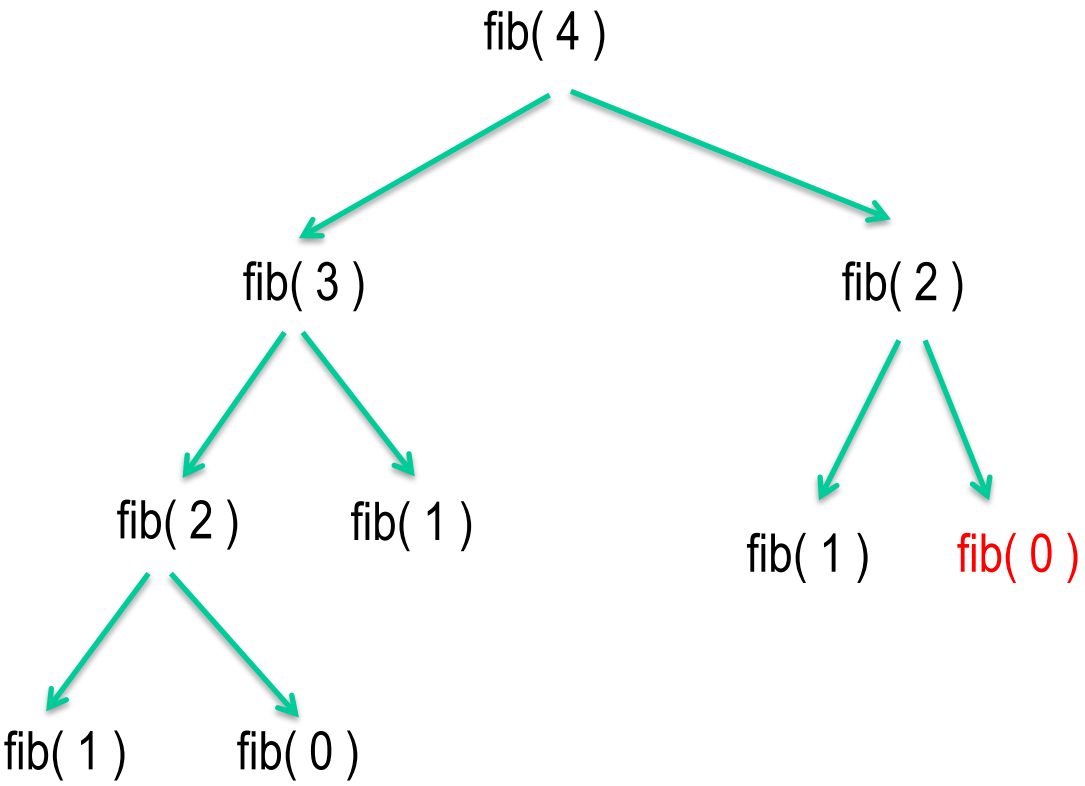
Recursion: Fibonacci

Tree of Calls to fib(4)



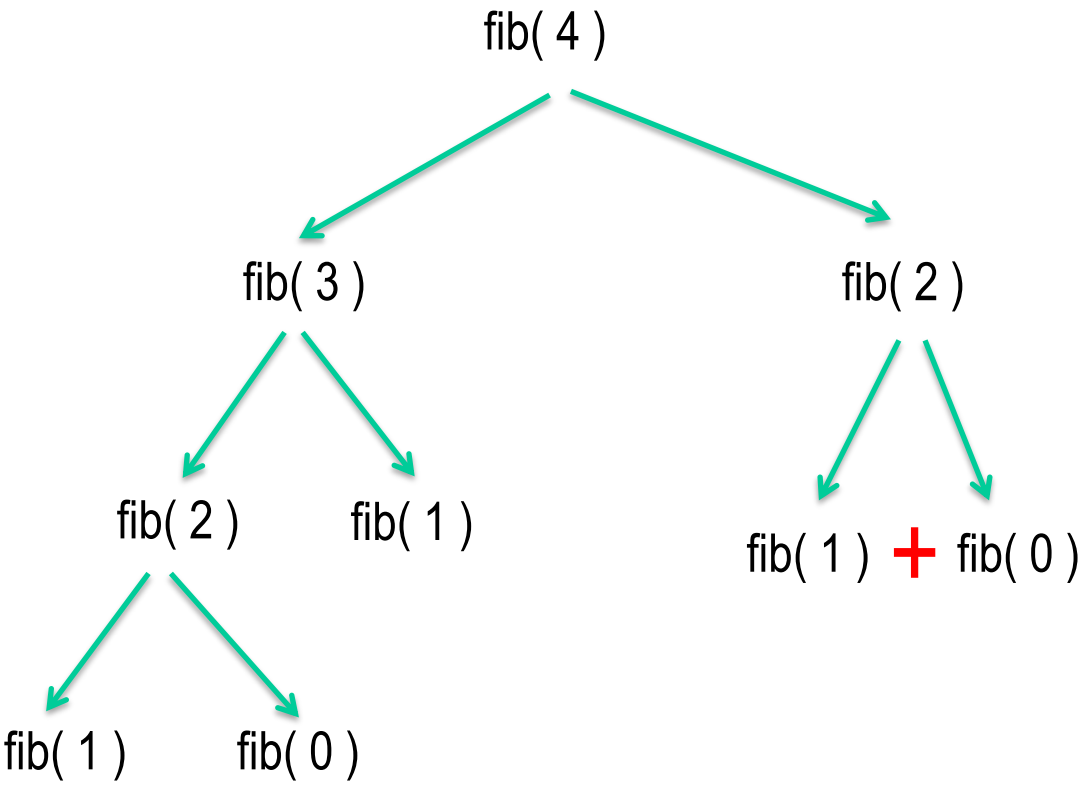
Recursion: Fibonacci

Tree of Calls to fib(4)



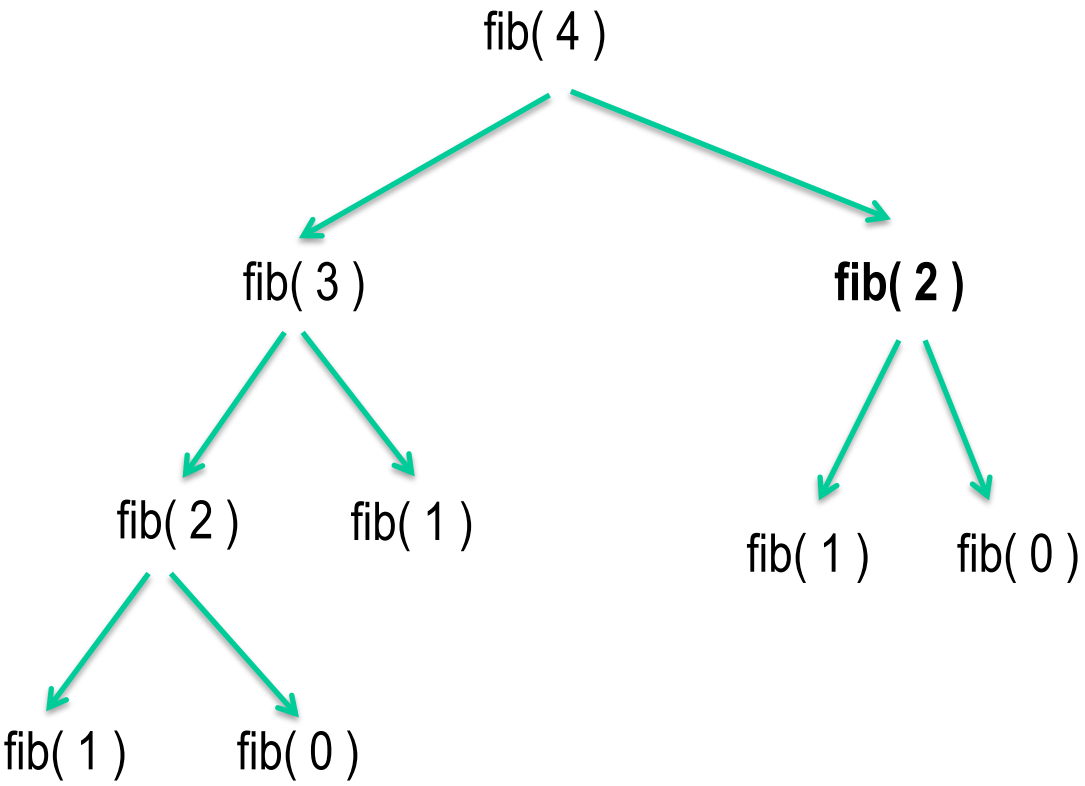
Recursion: Fibonacci

Tree of Calls to fib(4)



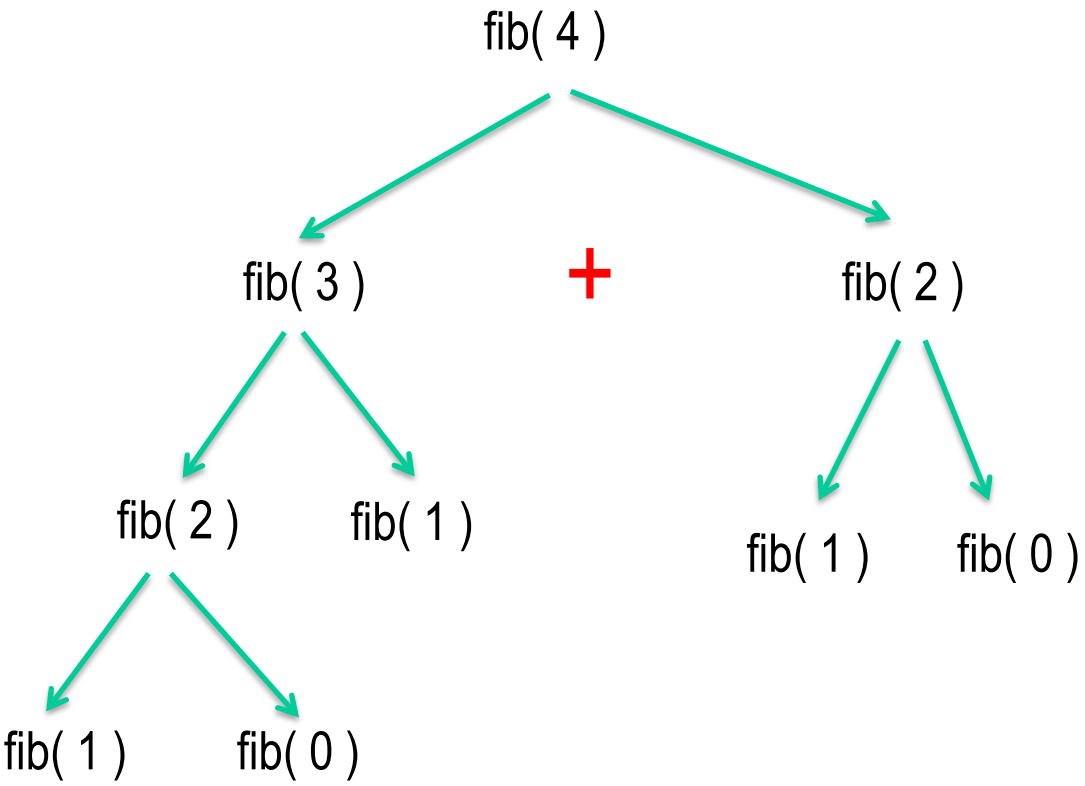
Recursion: Fibonacci

Tree of Calls to fib(4)



Recursion: Fibonacci

Tree of Calls to fib(4)



Recursion: Fibonacci

Tree of Calls to fib(4)