2023-03-17

10.1: Recursion

Recursion

Wikipedia: *Recursion is the process a procedure goes through when one of the steps of the procedure involves invoking the procedure itself. A procedure that goes through recursion is said to be 'recursive'.*

GeeksforGeeks: *A recursive function solves a particular problem by calling a copy of itself and solving smaller subproblems of the original problems. ... It is essential to know that we should provide a certain case in order to terminate this recursion process. So we can say that every time the function calls itself with a simpler version of the original problem.*

Chris: *Recursion is what happens when a method calls itself*

Recursion

Wikipedia: *Recursion is the process a procedure goes through when one of the steps of the procedure involves invoking the procedure itself. A procedure that goes through recursion is said to be 'recursive'.*

GeeksforGeeks: *A recursive function solves a particular problem by calling a copy of itself and solving smaller subproblems of the original problems. ... It is essential to know that we should provide a certain case in order to terminate this recursion process. So we can say that every time the function calls itself with a simpler version of the original problem.*

Chris: *Recursion is what happens when a method calls itself*

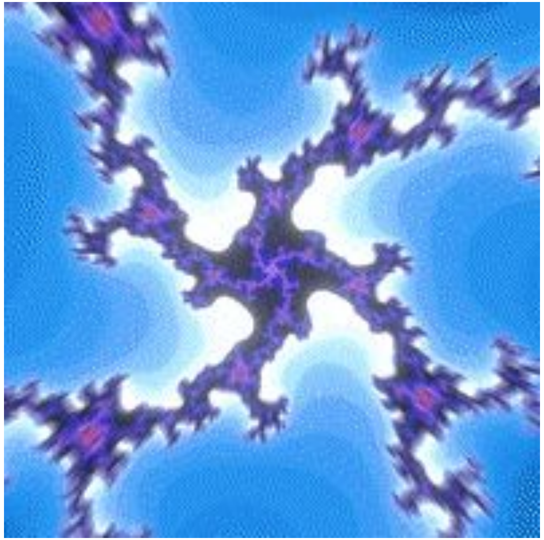
Or described another way...

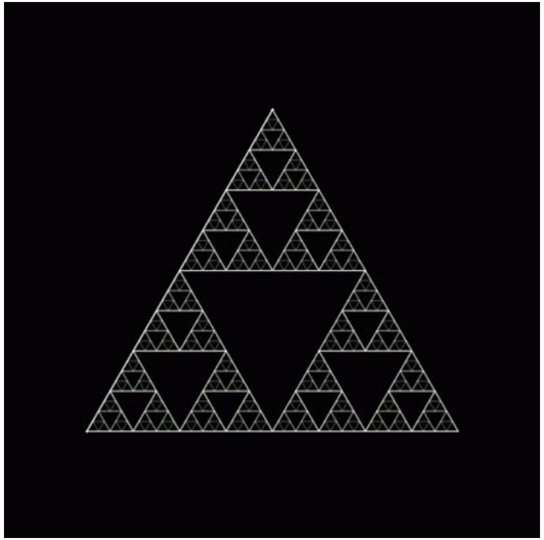
Recursion

Recursion

- Many recursive methods can be written in a non-recursive (iterative) way, however depending on the problem you are trying to solve - recursion can greatly improve the readability of your code (although it may be more challenging to conceptually understand)
- Some common examples where recursion works well
  - Calculating the Fibonacci Sequence up to a certain number
  - Traversing a list, map, tree, or filesystem
  - **Generating fractals...**

Recursion





Recursion

Common pitfalls that generate incorrect results or cause infinite recursion

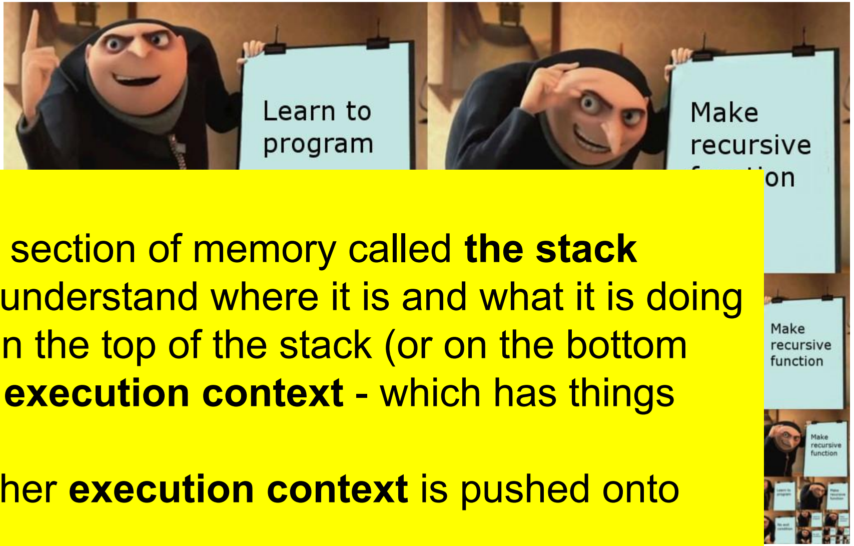
- Not properly sub-dividing the problem on each invocation
- Not properly identifying the end state (or Base Case) when the recursion should stop

Recursion

Common pitfalls that generate incorrect results or cause **infinite recursion**

- No
  pro
- No
  sta
  rec

- When you program runs it reserves a section of memory called **the stack**
- **The stack** is used by the program to understand where it is and what it is doing
- Usually this means that whatever is on the top of the stack (or on the bottom for some architectures) is the current **execution context -** which has things like the current local variables
- Every time you invoke a method another **execution context** is pushed onto the stack
- During **infinite recursion** - methods keep calling themselves - each time adding another **execution context** to **the stack** - until eventually **the stack** runs out of space and your program crashes with a **stack overflow** error



Recursion

Common pitfalls that generate incorrect results or cause infinite recursion

- Not properly sub-dividing the problem on each invocation
- Not properly identifying the end state (or Base Case) when the recursion should stop

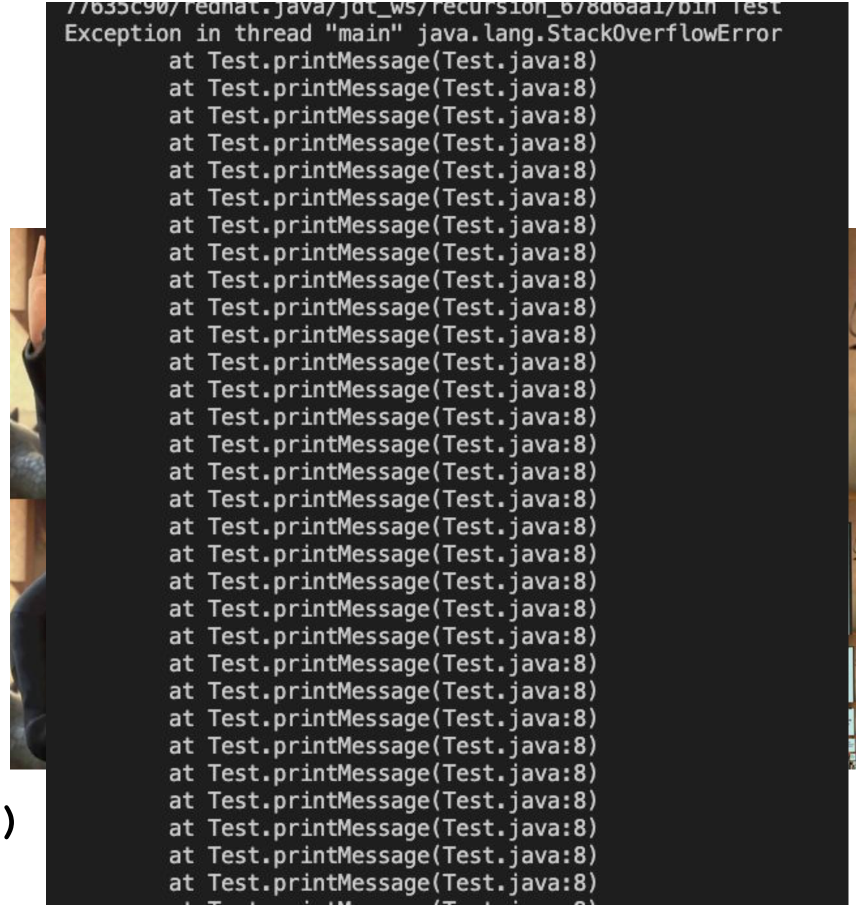
```
void printMessage(String message) {
  printMessage(message);
}
printMessage("Hello!");
```

Recursion

Common pitfalls that generate incorrect results or cause infinite recursion

- Not properly sub-dividing the problem on each invocation
- Not properly identifying the end state (or Base Case) when the recursion should stop

```
void printMessage(String message)
  printMessage(message);
}
printMessage("Hello!");
```



```
77635c90/redhat.java/jdt_ws/recursion_678d6aa1/bin Test
Exception in thread "main" java.lang.StackOverflowError
        at Test.printMessage(Test.java:8)
        at Test.printMessage(Test.java:8)
        at Test.printMessage(Test.java:8)
        at Test.printMessage(Test.java:8)
        at Test.printMessage(Test.java:8)
        at Test.printMessage(Test.java:8)
        at Test.printMessage(Test.java:8)
        at Test.printMessage(Test.java:8)
        at Test.printMessage(Test.java:8)
        at Test.printMessage(Test.java:8)
        at Test.printMessage(Test.java:8)
        at Test.printMessage(Test.java:8)
        at Test.printMessage(Test.java:8)
        at Test.printMessage(Test.java:8)
        at Test.printMessage(Test.java:8)
        at Test.printMessage(Test.java:8)
        at Test.printMessage(Test.java:8)
        at Test.printMessage(Test.java:8)
        at Test.printMessage(Test.java:8)
        at Test.printMessage(Test.java:8)
        at Test.printMessage(Test.java:8)
        at Test.printMessage(Test.java:8)
        at Test.printMessage(Test.java:8)
        at Test.printMessage(Test.java:8)
        at Test.printMessage(Test.java:8)
        at Test.printMessage(Test.java:8)
        at Test.printMessage(Test.java:8)
        at Test.printMessage(Test.java:8)
        at Test.printMessage(Test.java:8)
        at Test.printMessage(Test.java:8)
        at Test.printMessage(Test.java:8)
```

Infinite Recursion due to no Base Case

Recursion - Example 1 (adjust offset)

```
void printReverse(int[] nums, int idx) {
  if (idx < nums.length) {
    printReverse(nums, idx + 1);
    System.out.println(nums[idx]);
  }
}

int[] nums = new int[] { 1, 2, 3, 4, 5 };
printReverse(nums, 0);
```

- Print the array of integers in reverse order
- For each invocation we pass in the array and an ever-increasing index into the array
- While the index is less that the length of the array - we keep calling ourselves
- Until the index has reached the end - and then we stop the process and begin to unwind

Recursion - Example 1 (adjust offset)

```
void printReverse(int[] nums, int idx) {
  if (idx < nums.length) {
    printReverse(nums, idx + 1);
    System.out.println(nums[idx]);
  }
}

int[] nums = new int[] { 1, 2, 3, 4, 5 };
printReverse(nums, 0);
```

- **Print the array of integers in reverse order**
- For each invocation we pass in the array and an ever-increasing index into the array
- While the index is less that the length of the array - we keep calling ourselves
- Until the index has reached the end - and then we stop the process and begin to unwind

Recursion - Example 1 (adjust offset)

```
void printReverse(int[] nums, int idx) {
  if (idx < nums.length) {
    printReverse(nums, idx + 1);
    System.out.println(nums[idx]);
  }
}

int[] nums = new int[] { 1, 2, 3, 4, 5 };
printReverse(nums, 0);
```

- **Print the array of integers in reverse order**
- **For each invocation we pass in the array and an ever-increasing index into the array**
- While the index is less that the length of the array - we keep calling ourselves
- Until the index has reached the end - and then we stop the process and begin to unwind

Recursion - Example 1 (adjust offset)

```
void printReverse(int[] nums, int idx) {
  if (idx < nums.length) {
    printReverse(nums, idx + 1);
    System.out.println(nums[idx]);
  }
}

int[] nums = new int[] { 1, 2, 3, 4, 5 };
printReverse(nums, 0);
```

- Print the array of integers in reverse order
- For each invocation we pass in the array and an ever-increasing index into the array
- While the index is less that the length of the array - we keep calling ourselves
- Until the index has reached the end - and then we stop the process and begin to unwind

Recursion - Example 1 (adjust offset)

```
void printReverse(int[] nums, int idx) {
  if (idx < nums.length) {
    printReverse(nums, idx + 1);
    System.out.println(nums[idx]);
  }
}

int[] nums = new int[] { 1, 2, 3, 4, 5 };
printReverse(nums, 0);
```

- Print the array of integers in reverse order
- For each invocation we pass in the array and an ever-increasing index into the array
- While the index is less that the length of the array - we keep calling ourselves
- Until the index has reached the end - and then we stop the process and begin to unwind

Recursion - Example 1 (adjust offset)

```
void printReverse(int[] nums, int idx) {
  if (idx < nums.length) {
    printReverse(nums, idx + 1);
    System.out.println(nums[idx]);
  }
}

int[] nums = new int[] { 1, 2, 3, 4, 5 };
printReverse(nums, 0);
```

- Print the array of integers in reverse order
- For each invocation we pass in the array and an ever-increasing index into the array
- While the index is less that the length of the array - we keep calling ourselves
- Until the index has reached the end - and then we stop the process and begin to unwind

```
printReverse(nums, 0)
```

Recursion - Example 1 (adjust offset)

```
void printReverse(int[] nums, int idx) {
  if (idx < nums.length) {
    printReverse(nums, idx + 1);
    System.out.println(nums[idx]);
  }
}

int[] nums = new int[] { 1, 2, 3, 4, 5 };
printReverse(nums, 0);
```

- Print the array of integers in reverse order
- For each invocation we pass in the array and an ever-increasing index into the array
- While the index is less that the length of the array - we keep calling ourselves
- Until the index has reached the end - and then we stop the process and begin to unwind

```
printReverse(nums, 0)
  printReverse(nums, 1)
```

Recursion - Example 1 (adjust offset)

```
void printReverse(int[] nums, int idx) {
  if (idx < nums.length) {
    printReverse(nums, idx + 1);
    System.out.println(nums[idx]);
  }
}

int[] nums = new int[] { 1, 2, 3, 4, 5 };
printReverse(nums, 0);
```

- Print the array of integers in reverse order
- For each invocation we pass in the array and an ever-increasing index into the array
- While the index is less that the length of the array - we keep calling ourselves
- Until the index has reached the end - and then we stop the process and begin to unwind

```
printReverse(nums, 0)
  printReverse(nums, 1)
    printReverse(nums, 2)
```

Recursion - Example 1 (adjust offset)

```
void printReverse(int[] nums, int idx) {
  if (idx < nums.length) {
    printReverse(nums, idx + 1);
    System.out.println(nums[idx]);
  }
}

int[] nums = new int[] { 1, 2, 3, 4, 5 };
printReverse(nums, 0);
```

- • Print the array of integers in reverse order
- For each invocation we pass in the array and an ever-increasing index into the array
- While the index is less that the length of the array - we keep calling ourselves
- Until the index has reached the end - and then we stop the process and begin to unwind

```
printReverse(nums, 0)
  printReverse(nums, 1)
    printReverse(nums, 2)
      printReverse(nums, 3)
```

Recursion - Example 1 (adjust offset)

```
void printReverse(int[] nums, int idx) {
  if (idx < nums.length) {
    printReverse(nums, idx + 1);
    System.out.println(nums[idx]);
  }
}

int[] nums = new int[] { 1, 2, 3, 4, 5 };
printReverse(nums, 0);
```

- Print the array of integers in reverse order
- For each invocation we pass in the array and an ever-increasing index into the array
- While the index is less that the length of the array - we keep calling ourselves
- Until the index has reached the end - and then we stop the process and begin to unwind

```
printReverse(nums, 0)
  printReverse(nums, 1)
    printReverse(nums, 2)
      printReverse(nums, 3)
        printReverse(nums, 4)
```

Recursion - Example 1 (adjust offset)

```
void printReverse(int[] nums, int idx) {
  if (idx < nums.length) {
    printReverse(nums, idx + 1);
    System.out.println(nums[idx]);
  }
}

int[] nums = new int[] { 1, 2, 3, 4, 5 };
printReverse(nums, 0);
```

- Print the array of integers in reverse order
- For each invocation we pass in the array and an ever-increasing index into the array
- While the index is less that the length of the array - we keep calling ourselves
- Until the index has reached the end - and then we stop the process and begin to unwind

```
printReverse(nums, 0)
  printReverse(nums, 1)
    printReverse(nums, 2)
      printReverse(nums, 3)
        printReverse(nums, 4)
          printReverse(nums, 5)
```

Recursion - Example 1 (adjust offset)

```
void printReverse(int[] nums, int idx) {
  if (idx < nums.length) {
    printReverse(nums, idx + 1);
    System.out.println(nums[idx]);
  }
}

int[] nums = new int[] { 1, 2, 3, 4, 5 };
printReverse(nums, 0);
```

- Print the array of integers in reverse order
- For each invocation we pass in the array and an ever-increasing index into the array
- While the index is less that the length of the array - we keep calling ourselves
- Until the index has reached the end - and then we stop the process and begin to unwind

```
printReverse(nums, 0)
  printReverse(nums, 1)
    printReverse(nums, 2)
      printReverse(nums, 3)
        printReverse(nums, 4)
          printReverse(nums, 5)
            // stop
```

Recursion - Example 1 (adjust offset)

```
void printReverse(int[] nums, int idx) {
  if (idx < nums.length) {
    printReverse(nums, idx + 1);
    System.out.println(nums[idx]);
  }
}

int[] nums = new int[] { 1, 2, 3, 4, 5 };
printReverse(nums, 0);
```

- Print the array of integers in reverse order
- For each invocation we pass in the array and an ever-increasing index into the array
- While the index is less that the length of the array - we keep calling ourselves
- Until the index has reached the end - and then we stop the process and begin to unwind

```
printReverse(nums, 0)
  printReverse(nums, 1)
    printReverse(nums, 2)
      printReverse(nums, 3)
        printReverse(nums, 4)
          printReverse(nums, 5)
            // stop
          System.out.println(nums[4])
```

```
> 5
```

Recursion - Example 1 (adjust offset)

```
void printReverse(int[] nums, int idx) {
  if (idx < nums.length) {
    printReverse(nums, idx + 1);
    System.out.println(nums[idx]);
  }
}

int[] nums = new int[] { 1, 2, 3, 4, 5 };
printReverse(nums, 0);
```

- Print the array of integers in reverse order
- For each invocation we pass in the array and an ever-increasing index into the array
- While the index is less that the length of the array - we keep calling ourselves
- Until the index has reached the end - and then we stop the process and begin to unwind

```
printReverse(nums, 0)
  printReverse(nums, 1)
    printReverse(nums, 2)
      printReverse(nums, 3)
        printReverse(nums, 4)
          printReverse(nums, 5)
            // stop
          System.out.println(nums[4])
        System.out.println(nums[3])
```

```
> 5
> 4
```

Recursion - Example 1 (adjust offset)

```
void printReverse(int[] nums, int idx) {
  if (idx < nums.length) {
    printReverse(nums, idx + 1);
    System.out.println(nums[idx]);
  }
}

int[] nums = new int[] { 1, 2, 3, 4, 5 };
printReverse(nums, 0);
```

- Print the array of integers in reverse order
- For each invocation we pass in the array and an ever-increasing index into the array
- While the index is less that the length of the array - we keep calling ourselves
- Until the index has reached the end - and then we stop the process and begin to unwind

```
printReverse(nums, 0)
  printReverse(nums, 1)
    printReverse(nums, 2)
      printReverse(nums, 3)
        printReverse(nums, 4)
          printReverse(nums, 5)
            // stop
          System.out.println(nums[4])
        System.out.println(nums[3])
      System.out.println(nums[2])
```

```
> 5
> 4
> 3
```

Recursion - Example 1 (adjust offset)

```
void printReverse(int[] nums, int idx) {
  if (idx < nums.length) {
    printReverse(nums, idx + 1);
    System.out.println(nums[idx]);
  }
}

int[] nums = new int[] { 1, 2, 3, 4, 5 };
printReverse(nums, 0);
```

- Print the array of integers in reverse order
- For each invocation we pass in the array and an ever-increasing index into the array
- While the index is less that the length of the array - we keep calling ourselves
- Until the index has reached the end - and then we stop the process and begin to unwind

```
printReverse(nums, 0)
  printReverse(nums, 1)
    printReverse(nums, 2)
      printReverse(nums, 3)
        printReverse(nums, 4)
          printReverse(nums, 5)
            // stop
          System.out.println(nums[4])
        System.out.println(nums[3])
      System.out.println(nums[2])
    System.out.println(nums[1])
```

```
> 5
> 4
> 3
> 2
```

Recursion - Example 1 (adjust offset)

```
void printReverse(int[] nums, int idx) {
  if (idx < nums.length) {
    printReverse(nums, idx + 1);
    System.out.println(nums[idx]);
  }
}

int[] nums = new int[] { 1, 2, 3, 4, 5 };
printReverse(nums, 0);
```

- Print the array of integers in reverse order
- For each invocation we pass in the array and an ever-increasing index into the array
- While the index is less that the length of the array - we keep calling ourselves
- Until the index has reached the end - and then we stop the process and begin to unwind

```
printReverse(nums, 0)
  printReverse(nums, 1)
    printReverse(nums, 2)
      printReverse(nums, 3)
        printReverse(nums, 4)
          printReverse(nums, 5)
            // stop
          System.out.println(nums[4])
        System.out.println(nums[3])
      System.out.println(nums[2])
    System.out.println(nums[1])
  System.out.println(nums[0])
```

```
> 5
> 4
> 3
> 2
> 1
```

Recursion - Example 2 (trim data)

```java
import java.util.Arrays;

void printReverse2(int[] nums) {
  if (0 != nums.length) {
    printReverse2(Arrays.copyOfRange(
      nums, 1, nums.length));
    System.out.println(nums[0]);
  }
}

int[] nums = new int[] { 1, 2, 3, 4, 5 };
printReverse2(nums);
```

- For each invocation we create a new copy of the array that removes the first element
- While the array passed is not empty - we keep calling ourselves
- When an empty array is passed - we stop the process and begin to unwind

Recursion - Example 2 (trim data)

```
import java.util.Arrays;

void printReverse2(int[] nums) {
  if (0 != nums.length) {
    printReverse2(Arrays.copyOfRange(
      nums, 1, nums.length));
    System.out.println(nums[0]);
  }
}

int[] nums = new int[] { 1, 2, 3, 4, 5 };
printReverse2(nums);
```

- **For each invocation we create a new copy of the array that removes the first element**
- While the array passed is not empty - we keep calling ourselves
- When an empty array is passed - we stop the process and begin to unwind

Recursion - Example 2 (trim data)

```java
import java.util.Arrays;

void printReverse2(int[] nums) {
  if (0 != nums.length) {
    printReverse2(Arrays.copyOfRange(
      nums, 1, nums.length));
    System.out.println(nums[0]);
  }
}

int[] nums = new int[] { 1, 2, 3, 4, 5 };
printReverse2(nums);
```

- **For each invocation we create a new copy of the array that removes the first element**
- **While the array passed is not empty - we keep calling ourselves**
- When an empty array is passed - we stop the process and begin to unwind

Recursion - Example 2 (trim data)

```
import java.util.Arrays;

void printReverse2(int[] nums) {
  if (0 != nums.length) {
    printReverse2(Arrays.copyOfRange(
      nums, 1, nums.length));
    System.out.println(nums[0]);
  }
}

int[] nums = new int[] { 1, 2, 3, 4, 5 };
printReverse2(nums);
```

- For each invocation we create a new copy of the array that removes the first element
- While the array passed is not empty - we keep calling ourselves
- When an empty array is passed - we stop the process and begin to unwind

Recursion - Example 2 (trim data)

```
import java.util.Arrays;

void printReverse2(int[] nums) {
  if (0 != nums.length) {
    printReverse2(Arrays.copyOfRange(
      nums, 1, nums.length));
    System.out.println(nums[0]);
  }
}

int[] nums = new int[] { 1, 2, 3, 4, 5 };
printReverse2(nums);
```

- For each invocation we create a new copy of the array that removes the first element
- While the array passed is not empty - we keep calling ourselves
- When an empty array is passed - we stop the process and begin to unwind

```
printReverse([1,2,3,4,5])
  printReverse([2,3,4,5])
    printReverse([3,4,5])
      printReverse([4,5])
        printReverse([5])
          printReverse([])
            // stop
```

Recursion - Example 2 (trim data)

```
import java.util.Arrays;

void printReverse2(int[] nums) {
  if (0 != nums.length) {
    printReverse2(Arrays.copyOfRange(
      nums, 1, nums.length));
    System.out.println(nums[0]);
  }
}

int[] nums = new int[] { 1, 2, 3, 4, 5 };
printReverse2(nums);
```

- For each invocation we create a new copy of the array that removes the first element
- While the array passed is not empty - we keep calling ourselves
- When an empty array is passed - we stop the process and begin to unwind

```
printReverse([1,2,3,4,5])
  printReverse([2,3,4,5])
    printReverse([3,4,5])
      printReverse([4,5])
        printReverse([5])
          printReverse([])
            // stop
          System.out.println(nums[0])
        System.out.println(nums[0])
      System.out.println(nums[0])
    System.out.println(nums[0])
  System.out.println(nums[0])
```

```
> 5
> 4
> 3
> 2
> 1
```

Recursion Done Correctly

Recursion - Example 1 (increase offset)

```
void printReverse(int[] nums, int idx) {
  if (idx < nums.length) {
    printReverse(nums, idx + 1);
    System.out.println(nums[idx]);
  }
}
```

Recursion - Example 2 (trim data)

```
void printReverse2(int[] nums) {
  if (0 != nums.length) {
    printReverse2(Arrays.copyOfRange(
      nums, 1, nums.length));
    System.out.println(nums[0]);
  }
}
```

Recursion Done Correctly

Recursion - Example 1 (increase offset)

```
void printReverse(int[] nums, int idx) {
  if (idx < nums.length) {
    printReverse(nums, idx + 1);
    System.out.println(nums[idx]);
  }
}
```

Recursion - Example 2 (trim data)

```
void printReverse2(int[] nums) {
  if (0 != nums.length) {
    printReverse2(Arrays.copyOfRange(
      nums, 1, nums.length));
    System.out.println(nums[0]);
  }
}
```

Includes termination criteria ("Base Case")
"Will it ever end?"

Recursion Done Correctly

Recursion - Example 1 (increase offset)

```
void printReverse(int[] nums, int idx) {
  if (idx < nums.length) {
    printReverse(nums, idx + 1);
    System.out.println(nums[idx]);
  }
}
```

Recursion - Example 2 (trim data)

```
void printReverse2(int[] nums) {
  if (0 != nums.length) {
    printReverse2(Arrays.copyOfRange(
      nums, 1, nums.length));
    System.out.println(nums[0]);
  }
}
```

Includes termination criteria ("Base Case")
"Will it ever end?"

Recursion Done Correctly

Recursion - Example 1 (increase offset)

```
void printReverse(int[] nums, int idx) {
  if (idx < nums.length) {
    printReverse(nums, idx + 1);
    System.out.println(nums[idx]);
  }
}
```

Recursion - Example 2 (trim data)

```
void printReverse2(int[] nums) {
  if (0 != nums.length) {
    printReverse2(Arrays.copyOfRange(
      nums, 1, nums.length));
    System.out.println(nums[0]);
  }
}
```

Includes termination criteria ("Base Case")
"Will it ever end?"

✓

Recursion Done Correctly

Recursion - Example 1 (increase offset)

```
void printReverse(int[] nums, int idx) {
  if (idx < nums.length) {
    printReverse(nums, idx + 1);
    System.out.println(nums[idx]);
  }
}
```

Recursion - Example 2 (trim data)

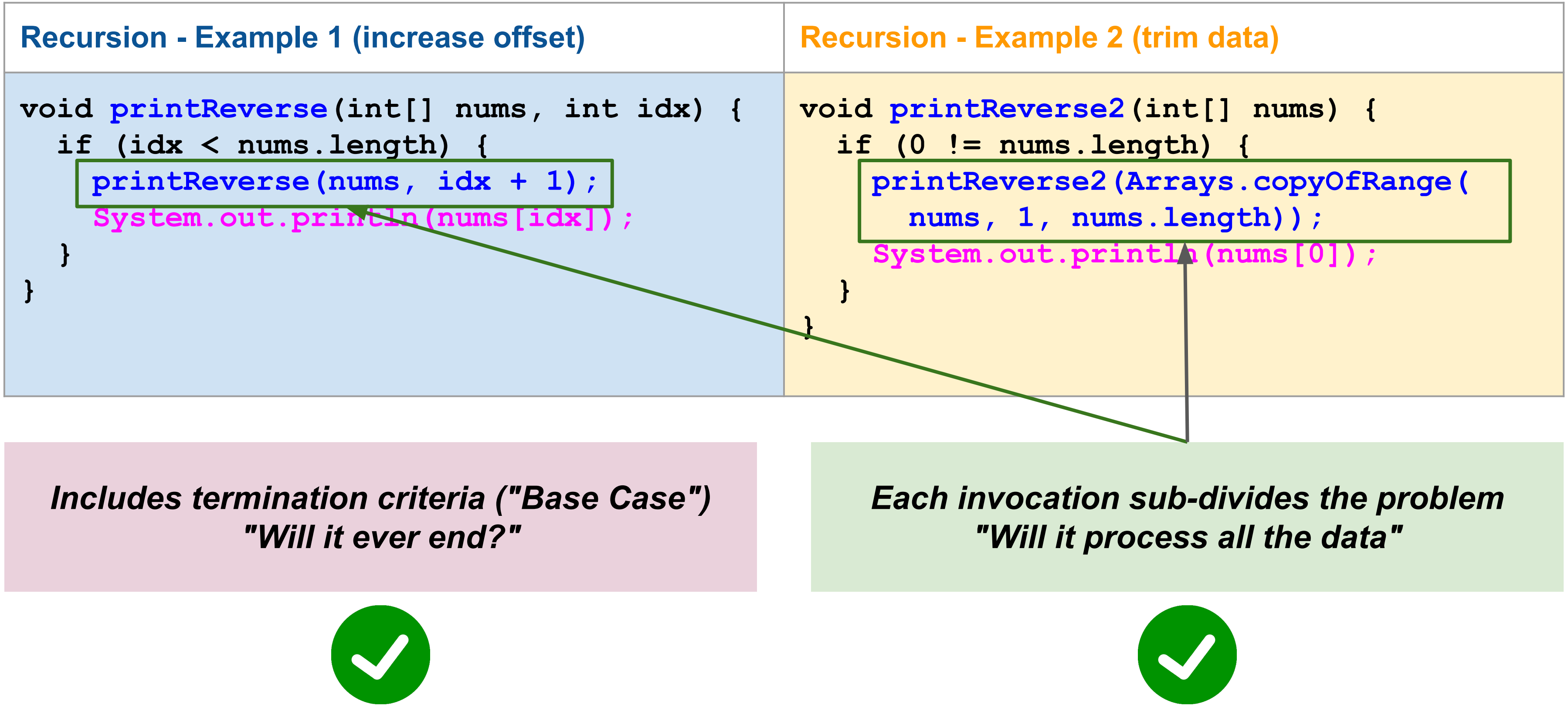
```
void printReverse2(int[] nums) {
  if (0 != nums.length) {
    printReverse2(Arrays.copyOfRange(
      nums, 1, nums.length));
    System.out.println(nums[0]);
  }
}
```

Includes termination criteria ("Base Case")
"Will it ever end?"

✓

Each invocation sub-divides the problem
"Will it process all the data"

Recursion Done Correctly



Recursion - Example 1 (increase offset)

```java
void printReverse(int[] nums, int idx) {
  if (idx < nums.length) {
    printReverse(nums, idx + 1);
    System.out.println(nums[idx]);
  }
}
```

Recursion - Example 2 (trim data)

```java
void printReverse2(int[] nums) {
  if (0 != nums.length) {
    printReverse2(Arrays.copyOfRange(
      nums, 1, nums.length));
    System.out.println(nums[0]);
  }
}
```

Includes termination criteria ("Base Case")
"Will it ever end?"

✓

Each invocation sub-divides the problem
"Will it process all the data"

✓

Recursion - Example 3 (data aggregator)

```
int getWordCount(String text, String word) {
  int count = 0;
  if (text.length() >= word.length()) {
    String s = text.substring(0, word.length());
    if (s.equals(word)) {
      count += 1;
    }
    count += getWordCount(text.substring(1), word);
  }
  return count;
}
```

- For each invocation we determine if the word appears at the beginning of text
- While the text is at least as long as word - we keep calling ourselves
- Each recursive call returns a count that is added to the count of the caller as we unwind

Recursion - Example 3 (data aggregator)

```
int getWordCount(String text, String word) {
  int count = 0;
  if (text.length() >= word.length()) {
    String s = text.substring(0, word.length());
    if (s.equals(word)) {
      count += 1;
    }
    count += getWordCount(text.substring(1), word);
  }
  return count;
}
```

- **For each invocation we determine if the word appears at the beginning of text**
- While the text is at least as long as word - we keep calling ourselves
- Each recursive call returns a count that is added to the count of the caller as we unwind

Recursion - Example 3 (data aggregator)

```
int getWordCount(String text, String word) {
  int count = 0;
  if (text.length() >= word.length()) {
    String s = text.substring(0, word.length());
    if (s.equals(word)) {
      count += 1;
    }
    count += getWordCount(text.substring(1), word);
  }
  return count;
}
```

- **For each invocation we determine if the word appears at the beginning of text**
- **While the text is at least as long as word - we keep calling ourselves**
- Each recursive call returns a count that is added to the count of the caller as we unwind

Recursion - Example 3 (data aggregator)

```
int getWordCount(String text, String word) {
  int count = 0;
  if (text.length() >= word.length()) {
    String s = text.substring(0, word.length());
    if (s.equals(word)) {
      count += 1;
    }
    count += getWordCount(text.substring(1), word);
  }
  return count;
}
```

- For each invocation we determine if the word appears at the beginning of text
- While the text is at least as long as word - we keep calling ourselves
- Each recursive call returns a count that is added to the count of the caller as we unwind

Recursion - Example 3 (data aggregator)

```
int getWordCount(String text, String word) {
  int count = 0;
  if (text.length() >= word.length()) {
    String s = text.substring(0, word.length());
    if (s.equals(word)) {
      count += 1;
    }
    count += getWordCount(text.substring(1), word);
  }
  return count;
}
```

- For each invocation we determine if the word appears at the beginning of text
- While the text is at least as long as word - we keep calling ourselves
- Each recursive call returns a count that is added to the count of the caller as we unwind

```
getWordCount("hello","l")
  getWordCount("ello","l")
    getWordCount("llo","l")
      getWordCount("lo","l")
        getWordCount("o","l")
          getWordCount("","l")
            // stop
```

Recursion - Example 3 (data aggregator)

```
int getWordCount(String text, String word) {
  int count = 0;
  if (text.length() >= word.length()) {
    String s = text.substring(0, word.length());
    if (s.equals(word)) {
      count += 1;
    }
    count += getWordCount(text.substring(1), word);
  }
  return count;
}
```

- For each invocation we determine if the word appears at the beginning of text
- While the text is at least as long as word - we keep calling ourselves
- Each recursive call returns a count that is added to the count of the caller as we unwind

```
getWordCount("hello","l")
  getWordCount("ello","l")
    getWordCount("llo","l")
      getWordCount("lo","l")
        getWordCount("o","l")
          getWordCount("","l")
            // stop
          return count (=0)
        return count (=0)
      return count (=1)
    return count (=1)
  return count (=2)
return count (=2)
```

Recursion - Example 3 (data a

Includes termination criteria ("Base Case")
"Will it ever end?"

```
int getWordCount(String text, String word) {
  int count = 0;
  if (text.length() >= word.length()) {
    String s = text.substring(0, word.length());
    if (s.equals(word)) {
      count += 1;
    }
    count += getWordCount(text.substring(1), word);
  }
  return count;
}
```

- For each invocation we determine if the word appears at the beginning of text
- While the text is at least as long as word - keep calling ourselves
- Each recursive call returns a count that is added to the count of the caller as we unwind

```
getWordCount("hello","l")
  getWordCount("ello","l")
    getWordCount("llo","l")
      getWordCount("lo","l")
        getWordCount("o","l")
          getWordCount("","l")
            // stop
          return count (=0)
        return count (=0)
      return count (=1)
    return count (=1)
```

Each invocation sub-divides the problem
"Will it process all the data"

Practice on your own

- CSAwesome 10.1 - Recursion
- Replit - Recursion