Recursion

Questions From Last Time

1. Why does the order matter for null checking tests?

```java
while (current.next.data < value && current.next != null) {
    ... 
}
```

Suppose our linked list is current.next.data = 2. When we move current over, we get current.next = null. When we try to say current.next.data, we get a NullPointerException. If we had done the current.next != null check first, we would have broken out of the loop instead of getting a NullPointerException.

2. Do you have to reset this.front if you assign it to something else?

Unless you’re inserting at the front of the list, you should never edit this.front
Evaluating Arithmetic Expressions 7
How do we evaluate the mathematical expression \(((1 \times 17) + (2 \times (3 + (4 \times 9))))\)?

\[
((1 \times 17) + (2 \times (3 + (4 \times 9)))) \\
( 17 + (2 \times (3 + (4 \times 9)))) \\
( 17 + (2 \times 39 )) \\
( 17 + 78 ) \\
95
\]

Psuedocode for eval 9

```plaintext
// Algorithm
1 Find the outermost operation.
2 Figure out the left and right operands.
3 If left is not a number, eval it. Call the result a.
4 If right is not a number, eval it. Call the result b.
5 Return a op b.
```

Running eval\(((1+3)+(4+2))\)

```plaintext
1 The outermost operation is +.
2 The left is \((1+3)\) and the right is \((4+2)\).
3 \((1+3)\) is not a number. So, evaluate it:
   1 The left is 1 and the right is 3.
   2 \(a = 1\)
   3 \(a + b = 3\)
   So, \(a = 3\).
4 \((4+2)\) is not a number. So, evaluate it:
   1 The outermost operation is +.
   2 The left is 4 and the right is 2.
   3 \(a = 4\)
   4 \(a + b = 6\)
   So, \(a + b = 6\).
5 \(a + b = 9\)
```
Making Change 10

Someone will ask you “Can you make change for \(N\) spirals?”

**Instructions**
- If you were asked “Can you make change for 0 spirals?”, answer “yes”.
- If the answer is “yes”:
  - Take the bills the person gives you
  - Add the bill you used to the pile
  - Tell the person who asked you, “yes”, and hand them the pile of bills.
- If the answer is “no”:
  - If you have any bills left, go back to step 2 and follow the same procedure attempting to use one.
  - If you are out of bills to try, tell the person who asked you, “no”.

The Many Ways To Think About Recursion 12

**The Code Already Works!**
This is the most important strategy for recursion!

When you are writing a recursive function, **pretend that it already works** and use it whenever possible.

**Let Someone Else Do The Rest**
Recursion is an army of people who can answer instances of your question. You solve a tiny piece and pass it on to someone else.

**This is like the change example!**

**Where Can I Use My Function?**
Before writing your recursive function, write down what it is supposed to do. Then, when writing it, try to find places that you can apply that idea to.

Writing the Evaluator 13

Now, let’s go ahead and write the eval function we talked about. The goals of writing this function are to see the following about recursive code:

- The code is short
- The version with loops is horrid
- You can do really cool things with recursion

**eval and makeChange**

**To eval(e)**
- If e is a number, return it.
- Otherwise, eval the left and the right; put them together with e.

**To makeChange(n):**
- If \(n = 0\), return true
- Otherwise:
  - Check if we can make change for \(n\) by using a 2 bill; if so, return true
  - Check if we can make change for \(n\) by using a 5 bill; if so, return true
  - Give up and return false

**Insight: The Structure of Recursive Problems**
- Every recursive problem has a “trivial case” (the simplest expression is a number; the simplest number is 0). This case is called the **base case**.
- Every recursive problem breaks the problem up into smaller pieces (the expression pieces are left and right; the change pieces are use each type of bill). This case is called the **recursive case**.

printStars 14

Consider the function `printStars`:
```java
public static void printStars(int n) {
    for (int i = 0; i < n; i++) {
        System.out.print("\*");
    }
}
```

Let’s write it recursively:
```java
public static void printStars(int n) {
    if (n == 0) {
        System.out.println();
    } else {
        System.out.print("\*");
        printStars(n - 1);
    }
}
```

**Tracing printStars**

```
//Run printStars(3)
public static void printStars(int n) {
    // ...
}
```

```
//Run printStars(2)
public static void printStars(int n) {
    // ...
}
```

```
//Run printStars(1)
public static void printStars(int n) {
    // ...
}
```

```
//Run printStars(0)
public static void printStars(int n) {
    // ...
}
```
Some Recursion Tips!

- Once you have a solution, it might feel obvious. This is a tricky feeling. Solving recursion problems is much harder than understanding a solution to a recursion problem.

- Understand the metaphors/ideas/ways to think about recursion. Choose one that makes the most sense to you, and run with it.

- Recursion will always have at least one base case and at least one recursive call.

- Be able to write down the steps in a recursive trace when given a recursive function.