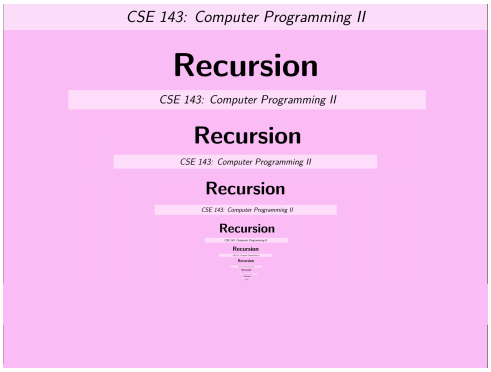
CSE 143

Computer Programming II

Recursion



Outline

Find 1337

Attempt #1

One person sifts through cards until they find the one with 1337

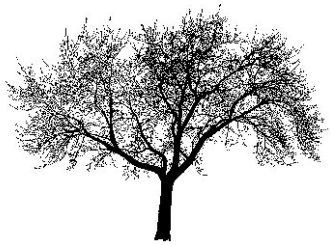
1. Loop through the cards until you find the one with 1337

Attempt #2

Lazy people offload the work onto others as follows:

1. If you are given a single card, check if it has 1337 on it. If it does, shout out "I found it!" If it doesn't, make a frowny face.
2. If you are given multiple cards...
   1. Give about half of them to the person on your left
   2. Give the other half to the person on your right

Recursion in Nature



Recursive Structures

`LinkedLists` are **recursive structures**.

A `LinkedList` is...

a piece of data and a `LinkedList`, which is
a piece of data and a `LinkedList`, which is
a piece of data and a `LinkedList`, which is
a piece of data and a `LinkedList`, which is
a piece of data and a `LinkedList`, which is
a piece of data and a `LinkedList`, which is...

A **recursive data structure** is one made up of smaller versions of the same data structure.

Recursion is... Recursion is... 4

Definition (Recursion)

Recursion is the definition of an operation **in terms of itself**.

To solve a problem with recursion, you break it down into smaller instances of the problem and solve those.

Definition (Recursive Programming)

Writing methods that call themselves to solve problems recursively

Some problems are **naturally recursive** which means they're easy to solve using recursion and much harder using loops.

Why Learn Recursion? 5

- It's a different way of thinking about problems
- Recursion leads to **much** shorter code to solve difficult problems.
- Some programming languages **do not have loops**.
- Many data structures are defined recursively, and recursion is the easiest way of dealing with those structures.

Evaluating Arithmetic Expressions 6

How do we evaluate the mathematical expression $((1*17)+(2*(3+(4*9))))$?

```
((1 * 17) + (2 * (3 + (4 * 9))))
 ------------------------------

((1 * 17) + (2 * (3 + (4 * 9))))
  ------     -----------------

((1 * 17) + (2 * (3 + (4 * 9))))
  -   --     -----------------

(   17    + (2 * (3 + (4 * 9))))
    --       -----------------

(   17    + (2 * (3 + (4 * 9))))
    --       -    ----------

(   17    + (2 * (3 + (4 * 9))))
    --       -    -   -----

(   17    + (2 * (3 + (4 * 9))))
    --       -    -   -   -

(   17    + (2 * (3 +   36   )))
    --       -    -    --

(   17    + (2 *       39      ))
    --       -         --

(   17    +          78         )
    --                --

                  95
                  --
```

Evaluating Expressions: It's Recursive! 7

Evaluation of a simple expression $(1*3)+(4+2)$ looks like:

To evaluate $(1*3)+(4+2)$, first evaluate $(1*3)$ and $(4+2)$:
$(1*3)=3$
$(4+2)=6$
So, $3+6=9$.

The big instance of the problem is

$$(1*3)+(4+2)$$

and the smaller instances are

$$(1*3) \text{ and } (4+2)$$

Psuedocode for `eval` 8

`eval` 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`$((1*3)+(4+2))$

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 outermost operation is $*$.
   2. The left is 1 and the right is 3.
   3. $a=1$
   4. $b=3$
   5. $a*b=3$

   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. $b=2$
   5. $a+b=6$

   So, $b=6$.
5. $a+b=9$

`eval` and `find1337` 9

To `eval`(e)

- If e is a number, return it.
- Otherwise, `eval` the left and the right; put them together with op

To `find1337`$(List list)$:

- If $list.size() == 1$, check if it's 1337
- Otherwise:
  1. Split the size n list into two lists `list1` and `list2`
  2. Check if 1337 is in `list1`
  3. Check if 1337 is in `list2`
  4. Give up(?!?)

Insight: The Structure of Recursive Problems

- Every recursive problem has a "trivial case" (the simplest expression is a number; the simplest number is 0; the simplest list is size 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 1337 pieces are halves of a list). This case is called the **recursive case**.

The Many Ways To Think About Recursion

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 1337 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

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

printStars

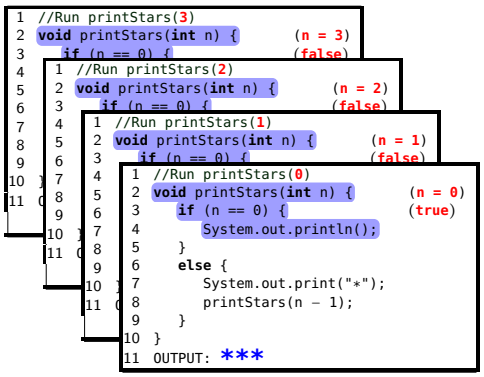
Consider the function `printStars`:

```
1 public static void printStars(int n) {
2    for (int i = 0; i < n; i++) {
3       System.out.print("*");
4    }
5    System.out.println();
6 }
```

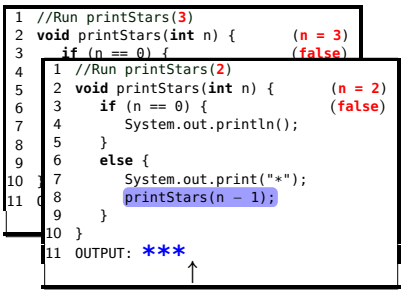
Let's write it recursively.

```
1 public static void printStars(int n) {
2    if (n == 0) {
3       System.out.println();
4    }
5    else {
6       System.out.print("*");
7       printStars(n - 1);
8    }
9 }
```

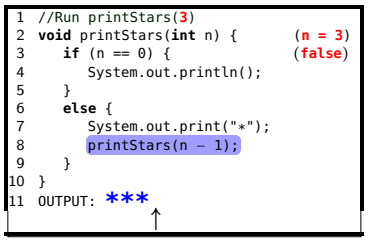
Tracing printStars



Tracing printStars



Tracing printStars



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.