CSE143 Notes for Monday, 4/25/11

I began a new topic: recursion. We have seen how to write code using loops, which is a technique called iteration. Recursion is an alternative to iteration that is equally powerful. It might seem odd to study something equally powerful if it doesn't allow us to solve a new set of problems. There are several reasons for this. First, recursion versus iteration is almost a religious question for computer scientists with passionate advocates on both sides. Think of iteration as being like Christianity and recursion as being like Buddhism. It's not that one is right and one is wrong. They both are attempting to accomplish the same thing, they just do it in a slightly different way.

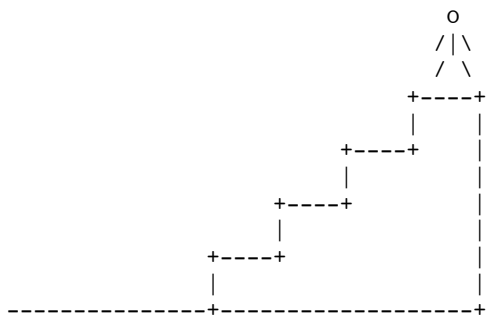
It's important to study recursion because some problems are naturally easier to express recursively or naturally easier to express iteratively. Also, you might find out that you're really a Buddhist and that you like programming recursively.

I started with a non-programming example. I picked out a student and asked him what row he was sitting in. He said, "row 3." I pointed out that he used a classic iterative solution. He started a counter at 0, moved his eyes to the front of the room and kept doing count++ while there were more rows left to count. So I asked people to imagine that the room is not full of people who have excellent vision but full of robots who can't see very far. How could we program the robots to answer this question?

The idea is to take advantage of the fact that there are many robots in the room. So I asked people to think about what the robot I talked to might ask the robot in front of him. I pointed out that it wouldn't be very helpful to ask, "Do you know what row I'm in?" Then you end up with a room full of robots all saying, "Do you know what row he's in?" No progress is being made.

Instead, the robot should ask the robot in front of him, "What row are you in?" Notice that it's the same question, but he's asking a robot that is closer to the front of the room. The fact that the question "recurs" (occurs again) is what leads us to call this recursion. The second robot asks the robot in front of him what row he is in. At that point, we have a robot who is in the first row. We assume that robots aren't so blind that they can't see when they're in the first row. So that robot can do something different. He can say, "I'm in row 1." That allows the robot just behind him to answer, "I'm in row 2." And that gets back to the robot I talked to, who can now say, "I'm in row 3."

That basic process is very much like recursion. We solve a large task by breaking it up into a series of smaller tasks that are somehow all the same. Another analogy that I said people should consider is the idea of descending a staircase. Suppose that someone is standing at the top of a staircase:



If the person wants to get down from the staircase, they need just two things: they need to be

able to get down from one step to the step below and they need a way to step onto the ground when they get to the bottom of the staircase.

All recursion problems involve different cases. The case that involves going from the current step to a lower step is referred to as the recursive case because it turns one stair-descending task into a simpler stair-descending task. The case that involves stepping onto the ground is called the base case. It's the case that stops the recursion from executing. Without a recursive case, you'd never get any closer to getting down the stairs. Without the base case, you're likely to keep trying to descend the staircase when you've run out of steps.

Then I turned our attention to a programming example. I decided to show something familiar first, so we looked at a method written with iteration that would be fairly familiar. The method produces a line of output with n stars on it (for some n greater than or equal to 0):

```
public void writeStars(int n) {
    for (int i = 0; i < n; i++)
        System.out.print("*");
    System.out.println();
}
```

This is a classic iterative solution. There is some initialization before the loop begins (int i = 0), a test for the while loop (i < n), some code that is execute each time through the loop (print and i ++) and some concluding code that is executed after the loop finishes (the println).

To write this recursively, we want to think in terms of cases. Recursion always involves case analysis. There is always at least one base case and at least one recursive case. Some people find it easier to think about the recursive case first, but I find it easier to think about the base case first.

So we have to think about the range of possible tasks we might be asked to perform. Of those, which is the simplest? Which writeStars task is the easiest of all? One person suggested that writing one star is easy and it is, but there's something even easier. Writing 0 stars is even easier. So we can begin by handling that case:

```
if (n == 0)
    System.out.println();
else {
    ...
}
```

It is common to see if/else statements in recursive definitions because we are so often distinguishing cases. So what do we do in the else part? This is where your programming instincts are going to lead you astray. You're going to want to solve the whole problem (a for loop starting at 0...). With recursive solutions, we want to figure out how to solve just a little bit of the problem. How can we get a little closer? Someone suggested writing out one star:

```
if (n == 0)
    System.out.println();
else {
    System.out.print("*");
    ...
}
```

This is where the second problem comes in. You have to believe that the method you are writing actually works. I call this the "leap of faith" that you have to make to be able to write recursive code (I said it was almost like a religion). So after we've written one star, what we have left to do is to write a line of (n-1) stars. How can we do that? If only we had a method that would write

out a line of stars, then we could call it. But we do have such a method! It's the one we're writing:

```
public void writeStars2(int n) {
    if (n == 0)
        System.out.println();
    else {
        System.out.print("*");
        writeStars2(n - 1);
    }
}
```

This is the entire definition. It doesn't seem right to us because we're used to solving the entire problem ourselves. And it doesn't seem like we should be able to call writeStars2 inside of writeStars2. But that's what recursion is all about.

To understand this, I told people that you have to realize that a method like writeStars2 is not just a single method. Think of it as an entire army of robots all programmed with the writeStars2 code. We can have a series of robots solve one of these tasks. This is like the "what row are you in?" question where we had a group of robots solve the problem by working together.

Consider what happens when I call writeStars2(2). We bring out the first robot of the writeStars2 army and we tell it to write a line of 2 stars. It sees that 2 isn't 0, so it writes one star and calls up a second robot to write a line of 1 star. That robot sees that 1 isn't 0, so it writes out one star and calls up a third robot to write a line of 0 stars. That robot is the one that ends up with the easy task, the base case. It sees that it has been asked to write a line of 0 stars and 0 equals 0, so it does the println and finishes executing. So the first two robots each printed one star and the third did a println to produce our line of 2 stars. It is useful to draw diagrams of the series of calls, using indentation to keep track of the fact that one version called another:

```
writeStars2(2);
    writeStars2(1);
        writeStars2(0);
```

I spent a few more minutes reminding people to look at how these two methods differ. The iterative version has a familiar loop structure. The recursive version has an if/else that distinguishes between a base case and a recursive case. The recursive case includes a call on the method we're writing, but with a different value than the original (a smaller "n" than in the original call).

The writeStars example is a good one for seeing how recursion works, but it doesn't help you to see the power of recursion. So for a second example I said to consider the task of reversing the lines of text in a file. Suppose that we have a Scanner variable tied to a file that contains these four lines of text:

```
this
is
fun
no?
```

We want to write a method called reverse that takes the Scanner as a parameter and that reads these lines and writes them in reverse order:

```
no?
fun
is
this
```

This isn't hard to do with a loop as long as you have some kind of data structure like an ArrayList<String> to store the lines. But with recursion, we can solve it without defining a data structure.

I again began with the base case. What is the easiest file to reverse? A one-line file is pretty easy to reverse, but even easier would be a 0-line or empty file. Assume that the Scanner variable is called "input". With a Scanner, we know that there are no more lines to read if a call on hasNextLine() returns false. In other words, we could write code like the following for the base case:

```
if (!input.hasNextLine())
    // handle base case
```

But in this case, there isn't anything to do. An empty file has no lines to reverse. So in this case, it makes more sense to turn around the if/else so that we test for the recursive case. That way we can make it a simple if that has an implied "else there is nothing to do":

```
if (input.hasNextLine()) {
    ...
}
```

Again the challenge is to do a little bit. How do we get a bit closer to being done? Someone suggested reading one line of text:

```
if (input.hasNextLine()) {
    String line = input.nextLine();
    ...
}
```

So in our example, this would read the line "this" into the variable line and would leave us with these three lines of text in the Scanner:

```
is
fun
no?
```

We're trying to produce this overall output:

```
no?
fun
is
this
```

You might be asking yourself questions like, "Is there another line of input to process?" That's not how to think recursively. If you're thinking recursively, you'd think about what a call on the method would get you. Since the Scanner is positioned in front of the three lines: is/fun/no?, a call on reverse should read in those lines and produce the first three lines of ouput that we're looking for. If that works, then we'd just have to write out the line "this" afterwards to complete the output.

This is where the leap of faith comes in. We have to believe that the reverse method actually works. Because if it does, then this code can be completed as follows:

```
public void reverse(Scanner input) {
    if (input.hasNextLine()) {
        String line = input.nextLine();
        reverse(input);
        System.out.println(line);
```

```
    }
}
```

Then I went through a demonstration of how this would execute using overheads that I would stack one on top of the other. Each overhead had the definition for the reverse method and a place for storing the value of the variable "line". In other words, they each looked like this:

```
+-----------------------------------------------+
|  public void reverse(Scanner input) {         |
|      if (input.hasNextLine()) {               |
|          String line = input.nextLine();      |
|          reverse(input);                      |
|          System.out.println(line);            |
|      }                                        |
|  }                                            |
|      +------------------+                     |
| line |                  |                     |
|      +------------------+                     |
+-----------------------------------------------+
```

By stacking them one on top of the other, I was giving a visual representation of what is referred to as the "call stack". As methods are called, Java uses a Stack to keep track of the next method invocation. The first time we call the method, it reads the first line of text into its variable line and then it reaches the recursive call on reverse:

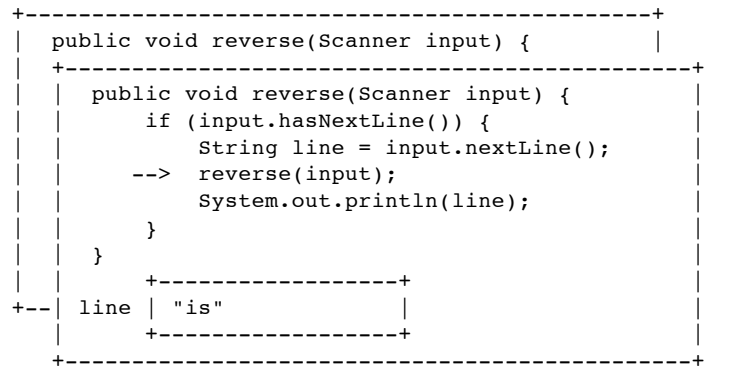
```
+-----------------------------------------------+
|  public void reverse(Scanner input) {         |
|      if (input.hasNextLine()) {               |
|          String line = input.nextLine();      |
|     -->  reverse(input);                      |
|          System.out.println(line);            |
|      }                                        |
|  }                                            |
|      +------------------+                     |
| line | "this"           |                     |
|      +------------------+                     |
+-----------------------------------------------+
```

Then what happens? What happens whenever a method is called? Java sets aside this method and turns its attention to the new method. So I took a second overhead from my stack and put it on top of this one.

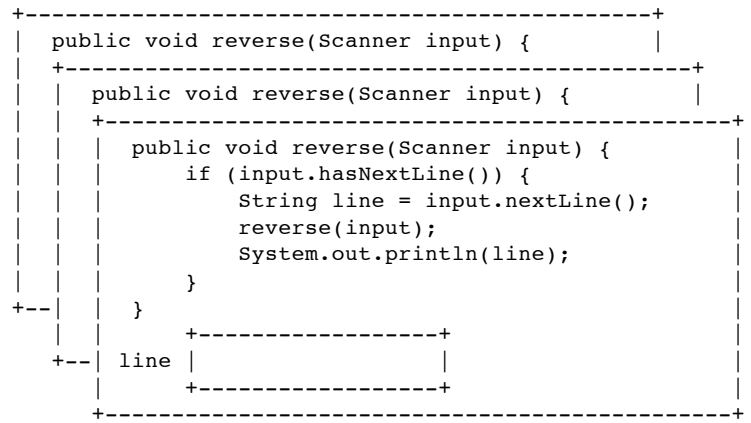
```
+-----------------------------------------------+
|  public void reverse(Scanner input) {         |
|  +--------------------------------------------+
|  |  public void reverse(Scanner input) {      |
|  |      if (input.hasNextLine()) {            |
|  |          String line = input.nextLine();   |
|  |          reverse(input);                   |
|  |          System.out.println(line);         |
|  |      }                                     |
|  |  }                                         |
|  |      +------------------+                  |
+--| line |                  |                  |
   |      +------------------+                  |
   +--------------------------------------------+
```

This new version of the method has its own variable called "line" in which it can store a line of text. And even though the previous version (the one underneath this one) is in the middle of its execution, this new one is at the beginning of its execution. Think back to the analogy of an entire army of robots all programmed with the same code, but they're executing slightly

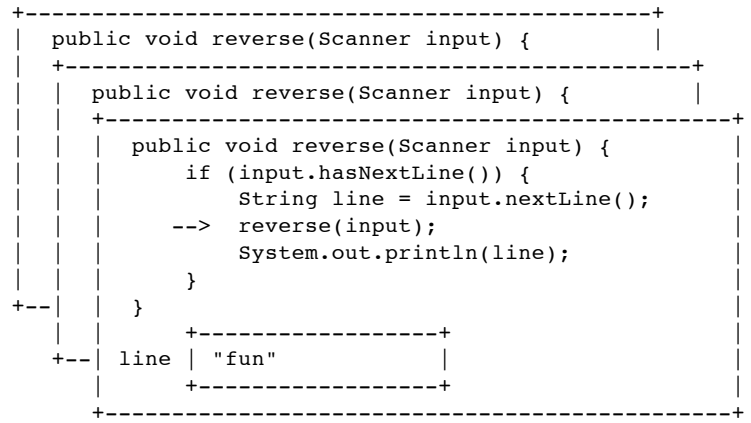
different tasks. This one finds that there is a line of text to be read and it reads it in (the second line that contains "is"). Then it finds itself executing a recursive call on reverse:



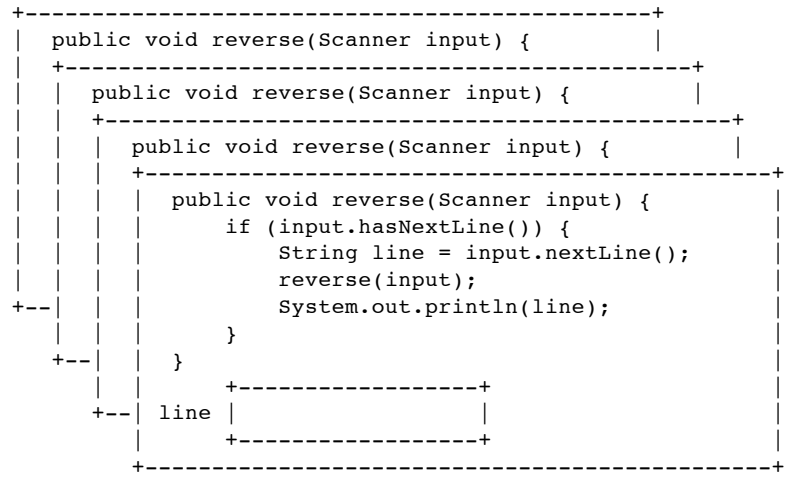
So Java sets aside this version of the method as well and brings up a third version. I brought out a third overhead:



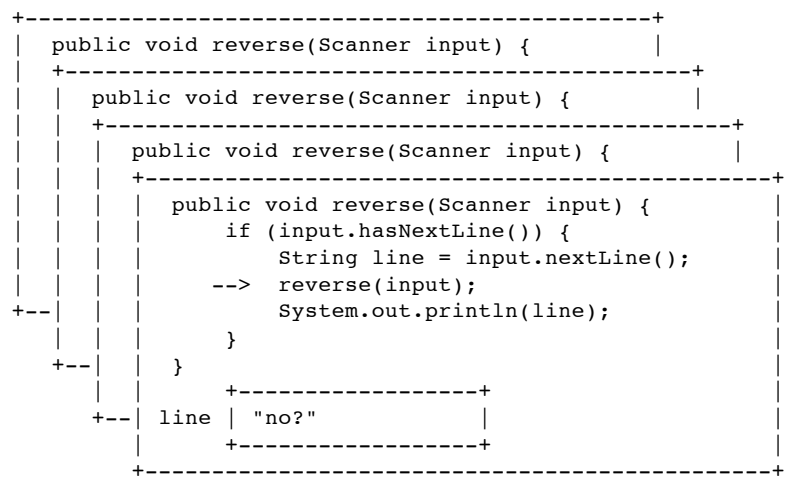
This is like the third robot of the "reverse" army that we've called up to work on this problem. It finds that there is a line to reverse (the third line, "fun"), so it reads it in and reaches a recursive call on reverse:



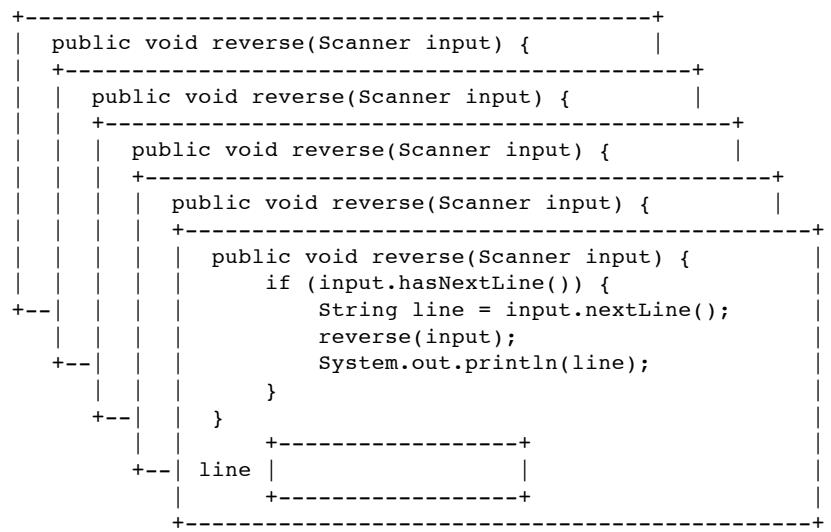
This brings up a fourth version of the method:



This one finds a fourth line of input ("no?"), so it reads that in and reaches the recursive call:

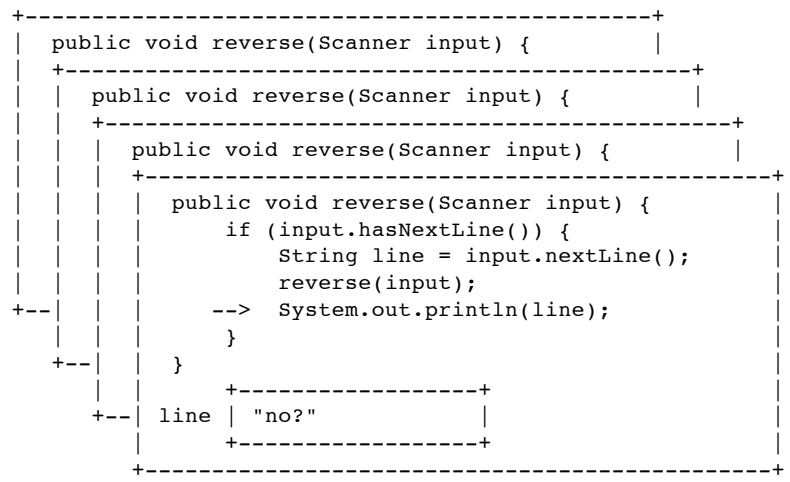


This brings up a fifth version of the method:

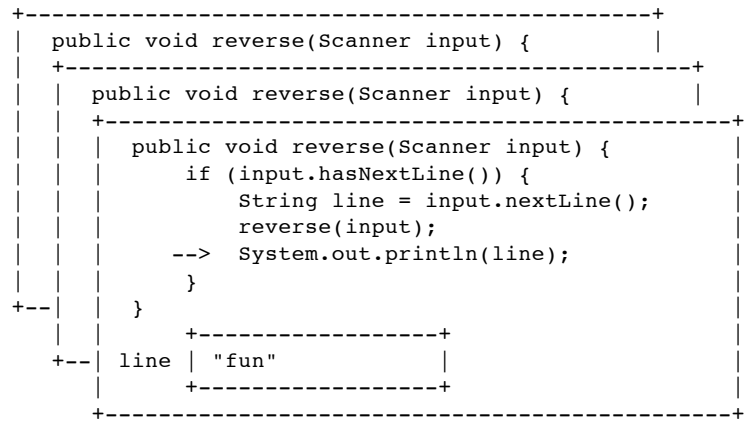


This one turns out to have the easy task, like our robot sitting in the front row or the call on writeStars2 asking for a line of 0 stars. This time around the Scanner is empty (input.hasNextLine() returns false). This is our very important base case that stops this process from going on indefinitely. This version of the method recognizes that there are no lines to reverse, so it simply terminates.

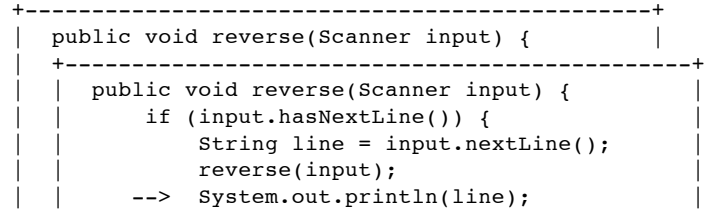
Then what? What happens in general when a method terminates? We're done with it, so we throw it away and go back to where we were just before this call. In other words, we pop the call stack and return to the line of code right after the method call:



We've finished the call on reverse, so now we're positioned at the println right after it. So we print the text in our variable "line" (we print "no?") and terminate. And where does that leave us? This method goes away and we return to where we were just before (we pop the call stack and go to the line of code right after the recursive call):



So we print our line of text, which is "fun" and this version goes away (we pop the stack again):



```
|  |      }                                          |
|  |  }                                              |
|  |      +------------------+                       |
+--| line | "is"             |                       |
   |      +------------------+                       |
   +-------------------------------------------------+
```

So we execute this println for the text "is" and pop once more:

```
+-------------------------------------------------+
|  public void reverse(Scanner input) {           |
|      if (input.hasNextLine()) {                 |
|          String line = input.nextLine();        |
|          reverse(input);                        |
|     -->  System.out.println(line);              |
|      }                                          |
|  }                                              |
|      +------------------+                       |
| line | "this"           |                       |
|      +------------------+                       |
+-------------------------------------------------+
```

Notice that we've written out three lines of text so far:

```
no?
fun
is
```

So our leap of faith was justified. The recursive call on reverse read in the three lines of text that came after the first and printed them in reverse order. We complete the task by printing our line of text, which leads to this overall output:

```
no?
fun
is
this
```

Then this version of the method terminates and we're done.

Then I turned to a different problem. We want to write a method called stutter that will take an integer value like 348 and return the integer 334488 (the value you get by replacing each digit with two of that digit). How do we write this? We again started with simple cases. What are easy numbers to stutter? Someone suggested 0, although that's actually a hard number to stutter. We can't really return 00, although we'll consider it acceptable to return 0. So what are some easy numbers to stutter? 1, 2, any one digit number.

So how do we stutter one-digit numbers? To turn a digit n into nn, we'd need to do something like this:

```
n0  10 of n
 n  plus 1 of n
--
nn
```

So we need a total of 11 times n. Some people saw this without thinking about individual digits and it's pretty obvious when you think about it that 1 times 11 is 11, 2 times 11 is 22, 3 times 11 is 33, and so on. I said to assume for now a precondition that n is greater than or equal to 0. So we have our base case:

```
//pre : n >= 0
public int stutter(int n) {
    if (n < 10)
        return 11 * n;
    else
        return ...;
}
```

So what do we do with numbers like 348? We need some way to break it up into smaller chunks, a way to split off one or more digits. Someone suggested that division by 10 would help:

```
n = 348

     34 | 8
 -------+-------
 n / 10 | n % 10
```

What next? Here you have to think recursively. The original number was 348. We've split it up into 34 and 8. What if we stuttered those values? You have to make the leap of faith and believe that the method actually works. If it does, then it turns 34 into 3344 and 8 into 88. How do we combine them to get 334488? We can't do simple addition:

```
3344
  88
----
wrong
```

We need to shift the 88 over to the right relative to the 3344. We do this by multiplying 3344 by 100:

```
334400
    88
------
334488
```

But that means we've solved the problem:

```
//pre : n >= 0
public int stutter(int n) {
    if (n < 10)
        return 11 * n;
    else
        return 100 * stutter(n / 10) + stutter(n % 10);
}
```

I mentioned that the final version of the method in the handout includes an extra case for negative numbers. So we remove the precondition and add a case for negative n. For a number like -348, we return -334488 by stuttering the positive number (stutter 348) and then negating the result.

I didn't have time to discuss the final example from the handout, so I said I'd discuss that in Wednesday's lecture.

Stuart Reges
Last modified: Mon Apr 25 17:05:26 PDT 2011