CSCI 1103: Basics of Recursion

Chris Kauffman

Last Updated:
Mon Dec 11 10:56:24 CST 2017

Logistics

Date		Lecture	Outside
Mon	12/4	PrintWriter	Lab 13: cmdline args, Scanner
Wed	12/6	Recursion	P5 Tests Posted
Fri	12/8	Recursion	
Mon	12/11	Recursion	Lab 14: Review
Wed	12/13	Review	P5 Due
Wed	12/20	Final Exam	1:30pm-3:30pm KELLER HALL 3-210

Reading from Eck

Ch 9.1 on Recursion

Goals

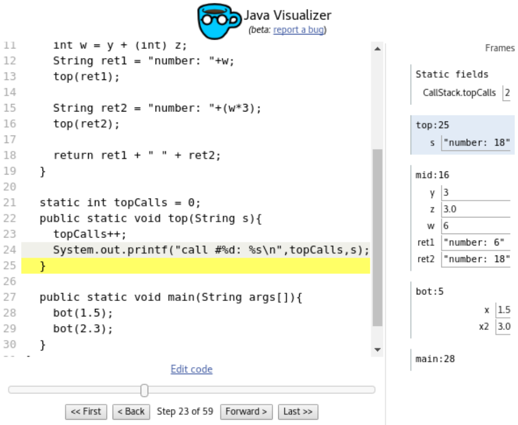
Basic Understanding of Recursion
Forward look to its use in problems

Lab13: Command Line Args and Scanner

Write a short program that counts lines, words, characters from files named on command line

The Call Stack

- Recall that methods have a stack frame (activation record)
- When one method calls another, another frame goes onto the call stack
- Frames can nest deeply, tools like Java Visualizer are useful to see how the stack moves



Exercise: Recursive Functions / Methods

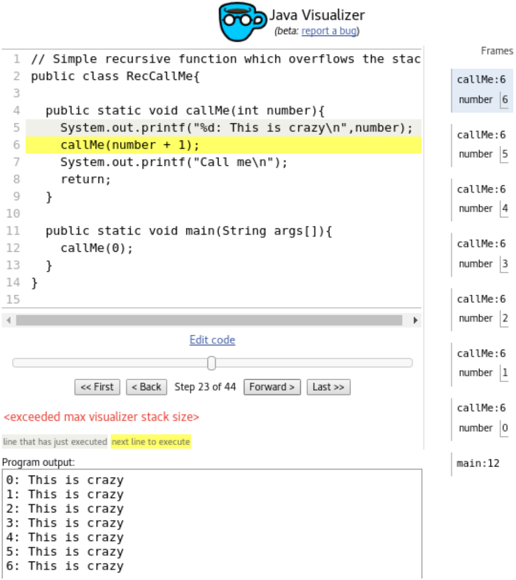
- A function that calls/invokes itself
- Looks normal, behaves normally, feels *crazy*

```
// Simple recursive function which overflows the stack
public class RecCallMe{

  public static void callMe(int number){
    System.out.printf("%d: This is crazy\n",number);
    callMe(number + 1);
    System.out.printf("Call me\n");
    return;
  }

  public static void main(String args[]){
    callMe(0);
  }
}
```

- Demonstrate the following in DrJava
- Draw some pictures to demonstrate what is happening
- Use the Java Visualizer to help with this

Answer: Recursive Functions / Methods

- Each `callMe()` invocation increases call stack depth
- Never reach `return` statement
- Calling method overflows the stack
- Never reach `printf("Call Me\n")`



Terminating Recursive Functions

To avoid a stack overflow, there must be a base case in which no recursive call is made. Usually recursive functions divide into

- Recursive cases: call the method with slightly different arguments to build call stack up another level
- Base cases: "answer found", return it, do not make another recursive call

Common code structure for single base and recursive case is to the right

```
public static X recFunc(...){
  // BASE CASE
  if(termCondition true){
    finish off answer;
    return x;
  }

  // RECURSIVE CASE
  do some stuff;
  x = recFunc(...); // recurse
  maybe do more;
  return x;
}
```

Exercise: Call Me Maybe

- Identify Recursive and Base Cases in the following code
- What condition terminates the recursion?
- What do you expect for output?

```java
public class RecCallMeMaybe{

  public static void callMe(int number){
    if(number == 0){
      System.out.printf("Here's my number: %d\n",
                        number);
      return;
    }


    System.out.printf("%d: This is crazy\n",
                      number);
    callMe(number - 1);
    System.out.printf("Call me maybe\n");
    return;
  }

  public static void main(String args[]){
    callMe(7);
  }
}
```

Answer: Call Me Maybe

Code Analysis

```java
// Simple recursive function which terminates
public class RecCallMeMaybe{

  public static void callMe(int number){
    if(number == 0){                // BASE CASE
      System.out.printf("Here's my number: %d\n",
                        number);
      return;                       // finished!
    }

    // Recursive Case
    System.out.printf("%d: This is crazy\n",
                      number);
    callMe(number - 1);             // RECURSE
    System.out.printf("Call me maybe\n");
    return;
  }

  public static void main(String args[]){
    callMe(7);
  }
}
```

Output

```
> javac RecCallMeMaybe.java
> java RecCallMeMaybe
7: This is crazy
6: This is crazy
5: This is crazy
4: This is crazy
3: This is crazy
2: This is crazy
1: This is crazy
Here's my number: 0
Call me maybe
Call me maybe
Call me maybe
Call me maybe
Call me maybe
Call me maybe
Call me maybe
```

Why would I use recursion?

- Looks a bit novel but hard to see a use until...
- Some problems are recursive, either explicitly or implicitly
- We will examine a few of these:
  - Factorial
  - Fibonacci numbers
  - Finding a specific combination
  - Maybe 2D maze search...

Factorial of an Integer

The *factorial* of a number is written with an exclamation mark and means to do the following:

$$5! = 5 \times 4 \times 3 \times 2 \times 1$$

$$7! = 7 \times 6 \times 5 \times 4 \times 3 \times 2 \times 1$$

$$N! = N \times (N - 1) \times (N - 2) \times ... \times 2 \times 1$$

Notice that factorial has a natural recursive definition

$$5! = 5 \times 4!$$

$$7! = 7 \times 6!$$

$$N! = N \times (N - 1)!$$

One oddity: $0! = 1$ by definition, not defined for negatives

Exercise: Recursive Factorial Execution

Show the output of executing the code below

Show/Explain how the recursive function works

```
# Run the program:
> java RecFact 4
???
```

```java
public class RecFact{
  public static void main(String args[]){
    int n = Integer.parseInt(args[0]);
    int factN = factRec(n);
    System.out.printf("%d! = %d\n",
                      n,factN);
  }

  public static int factRec(int n){
    if(n == 0 || n == 1){
      return 1;
    }
    int smaller =  factRec(n-1);
    int fact = n * smaller;
    return fact;
  }
}
```

Exercise: Factorial Methods

Iterative

Easy to write a loop to compute factorial.

```
public static int factLoop(int n){
  int fact = 1;
  for(int i=1; i<=n; i++){
    fact = fact*i;
  }
  return fact;
}
```

- What are metrics by which to compare programs?
- Which of these implementations is better?

Recursive

Also easy to use recursion for to do the same thing.

```
public static int factRec(int n){
  if(n == 0 || n == 1){
    return 1;
  }
  int smaller = factRec(n-1);
  int fact = n * smaller;
  return fact;
}

public static int factRecShort(int n)
  if(n == 0 || n == 1){
    return 1;
  }
  return n * factRecShort(n-1);
}
```

Answer: Factorial Methods

```
public static int factLoop(int n){
  int fact = 1;
  for(int i=1; i<=n; i++){
    fact = fact*i;
  }
  return fact;
}

public static int factRec(int n){
  if(n == 0 || n == 1){
    return 1;
  }
  int smaller = factRec(n-1);
  int fact = n * smaller;
  return fact;
}

public static int factRecShort(int n){
  if(n == 0 || n == 1){
    return 1;
  }
  return n * factRecShort(n-1);
}
```

- Loop version will likely be a little faster because pushing stack frames on in the recursive version takes some time
- Loop version will take less memory than recursive version as it uses a single stack frame while recursive version uses `n` frames
- Both are fairly easy to read and understand
- This makes it relatively easy verify that they are correct: the MOST IMPORTANT CODE METRIC

Based on this, one would likely prefer the Loop version, but this will not always be the case. . .

Fibonacci Sequence

- The classic example of a recursively defined mathematical entity
- The *Fibonacci Number Sequence* are a sequence of numbers which are defined as follows
  - The 0th Fibonacci number is 0, called f_0
  - The 1th Fibonacci number is 1, called f_1
  - All other Fibonacci numbers are the sum of the previous two Fibonacci numbers
    - Example: $f_2 = f_1 + f_0 = 1 + 0 = 1$; so $f_2 = 1$
    - Example: $f_3 = f_2 + f_1 = 1 + 1 = 2$; so $f_3 = 2$
- The general description is

$$f_i = f_{i-1} + f_{i-2}, \text{ with } f_0 = 0, f_1 = 1$$

- Fibonacci numbers show nicely in a table

i	0	1	2	3	4	5	6	7	8	9	10	..
f_i	0	1	1	2	3	5	8	13	?	?	?	..

A good Origin Story From WikiP "Fibonnacci Numbers"

Fibonacci (in AD 1707) considers the growth of an idealized (biologically unrealistic) rabbit population, assuming that: a newly born pair of rabbits, one male, one female, are put in a field; rabbits are able to mate at the age of one month so that at the end of its second month a female can produce another pair of rabbits; rabbits never die and a mating pair always produces one new pair (one male, one female) every month from the second month on. The puzzle that Fibonacci posed was: how many pairs will there be in one year?

- At the end of the first month, they mate, but there is still only 1 pair.
- At the end of the second month the female produces a new pair, so now there are 2 pairs of rabbits in the field.
- At the end of the third month, the original female produces a second pair, making 3 pairs in all in the field.
- At the end of the fourth month, the original female has produced yet another new pair, and the female born two months ago also produces her first pair, making 5 pairs.

i	0	1	2	3	4	5	6	7	8	9	10	11	12
f_i	0	1	1	2	3	5	8	13	21	34	55	89	144

CK: However, Indian Mathematicians had already developed the "Fibonacci" sequence some 1000 to 1900 years prior and had Twitter been around at that time we'd be studying the "Pingala" Sequence.

Exercise: Recursive Fibonacci

- Fib Definitions:
  - $f_0 = 0$
  - $f_1 = 1$
  - $f_i = f_{i-1} + f_{i-2}$
- Fibonacci numbers lend themselves well to a recursive solution because the sequence is defined recursively
- Fill in the template to the right to complete the definition of the numbers

```
public class FibRec{
  public static void main(String args[])
  {
    int i = Integer.parseInt(args[0]);
    int fibI = fibRec(i);
    System.out.printf("fib_%d = %d\n",
                      i,fibI);
  }
  // FILL IN THE TEMPLATE BELOW
  public static int fibRec(int i){
    if( ?? ){     // BASE CASE 1
      ??
    }
    if( ?? ){     // BASE CASE 2
      ??
    }
    // RECURSIVE CASE
    int fibPrev1 = ???  // 1 back
    int fibPrev2 = ???  // 2 back
    int fibI =     ???  // add last 2
    return fibI;
  }
}
```

Answer: Recursive Fibonacci

- Fib Definitions:
  - $f_0 = 0$
  - $f_1 = 1$
  - $f_i = f_{i-1} + f_{i-2}$
- Base case for f_0
- Base case for f_1
- Recursive case makes two recursive calls to look back two places

```java
public class FibRec{
  public static void main(String args[])
  {
    int i = Integer.parseInt(args[0]);
    int fibI = fibRec(i);
    System.out.printf("fib_%d = %d\n",
                      i,fibI);
  }
  // Recursive Fibonacci function
  public static int fibRec(int i){
    if(i == 0){         // base case 1
      return 0;
    }
    if(i == 1){         // base case 2
      return 1;
    }
    // recursive case
    int fibPrev1 = fibRec(i-1);
    int fibPrev2 = fibRec(i-2);
    int fibI = fibPrev1 + fibPrev2;
    return fibI;
  }
}
```

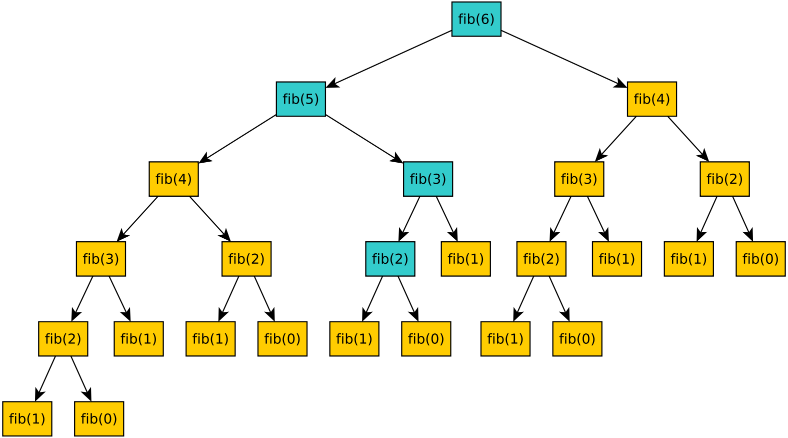
Exercise: How does `fibRec()` work?

- Take some time to examine the Function Call Stack as `fibRec()` executes
- Use the `FibRec6` code to the right in the Java Visualizer to see what happens
- Does any redundant computation get done?
- How deep does the stack get?
- What's a good way to describe/draw the overall computation?

```java
public class FibRec6{
  public static void main(String args[])
  {
    int i = 6;
    int fibI = fibRec(i);
    System.out.printf("fib_%d = %d\n",
                      i,fibI);
  }
  // Recursive Fibonacci function
  public static int fibRec(int i){
    if(i == 0){        // base case 1
      return 0;
    }
    if(i == 1){        // base case 2
      return 1;
    }
    // recursive case
    int fibPrev1 = fibRec(i-1);
    int fibPrev2 = fibRec(i-2);
    int fibI = fibPrev1 + fibPrev2;
    return fibI;
  }
}
```

Answers: How does `fibRec()` work?

- Best visualized by a tree of calls that occur at some point
- Actual active function calls in stack occupy one path in the tree, example is cyan
- Tons of redundant computation done in the recursive version, entire `fib(4)` tree is done twice unnecessarily



Exercise: Loopy Fibonacci

- Recursive version of Fibonacci is easy to specify but is inefficient due to the redudancy
- How about a non-recursive version of Fibonacci?
- Would need to use iteration (loops) in some way as repeated work is done
- Pitch me some ideas

Answers: Loopy Fibonacci

```
// Iterative version with an array
// Easy
public static int fibArray(int n){
  int fibs[] = new int[n+1];
  fibs[0] = 0;
  fibs[1] = 1;
  for(int i=2; i < n; i++){
    fibs[i] = fibs[i-1] + fibs[i-2];
  }
  return fibs[n];
}
// Iterative version w/o an array
// Tricky
public static int fibI(int n){
  int f1 = 1, f2 = 0, fn = 0;
  for(int i=0; i < n; i++){
    fn = f1 + f2;
    f1 = f2;
    f2 = fn;
  }
  return fn;
}
```

```
// Recursive
public static int fibR(int n){
  if(n==1){ return 1; }
  if(n==0){ return 0; }
  return fibR(n-1) + fibR(n-2);
}
```

Comparisons

Each of these codes exhibits a trade-off between

- Readability/correctness
- Use of more/less memory
- Speed of execution

If recursion still seems elegant but flawed, wait for the next set of examples.