CHAPTER 13

RECURSION

(edited by Simon Jones)

Slides by Rick Giles

Chapter Goals

- To learn to "think recursively"
- To understand the relationship between recursion and iteration
- To understand when the use of recursion affects the efficiency of an algorithm

Contents

- Triangle Numbers Revisited
- Problem Solving: Thinking Recursively
- The Efficiency of Recursion

General problem solving

- We often first set out a "high level outline" for the algorithm:

```
what to do at step 1;
… step 2;
… step 3;
… step 4;
```

- Then "refine" the steps to Java
  - Often basic statements: `step 2;` -> `a = b + 2;`
  - But also there may be a method that does the job:
    `step 3;` -> `int n = readInteger();`
  - The method might already exist, or might only be planned (but we have a specification for it)

13.1 Triangle Numbers Revisited

- Triangle shape of side length 4:

```
[]
[][]
[][][]
[][][][]
```

- We would like a method

```
public int getArea(int n)
```

  to compute the area of a triangle of width *n* , assuming each `[]` square has an area of 1
- Will use recursion
- Also called the n^{th} *triangle number*
- The third triangle number is 6, the fourth is 10

Handling Triangle of Width 1

- The triangle consists of a single square
- Its area is 1
- Take care of this case first:

```
public int getArea(int width)
{
   if (width == 1)
   {
     return 1;
   }
   . . .
}
```

Handling The General Case

- Break down a *large* triangle into a *smaller*, colored triangle plus an extra part:

```
[]
[][]
[][][]
[][][][]
```

- Area of larger triangle can be calculated as
  `smallerArea + width`

- To get the area of the smaller triangle
  - *We need to find the area of a triangle with side* `width – 1`
  - ***We have a method that calculates that:*** `getArea(width - 1)` !!!

Completed getArea Method

```
public int getArea(int width)
 {
    if (width == 1)
    {
       return 1;
    }
    else
    {
       return getArea(width - 1) + width;
    }
 }
```

Computing the Area of a Triangle With Width 4

- `getArea(4)` considers a smaller triangle of width 3
- It calls `getArea` for that triangle (3)
  - That method considers a smaller triangle of width 2
  - It calls `getArea` for that triangle (2)
    - That method considers a smaller triangle of width 1
    - It calls `getArea` on that triangle (1)
      - That method returns 1
    - The method returns `smallerArea+width` = 1 + 2 = 3
  - The method returns `smallerArea+width` = 3 + 3 = 6
- The method returns `smallerArea+width` = 6 + 4 = 10

Recursive Computation

- A **recursive computation** solves a problem by using the solution to the same problem with "simpler" inputs
  - "Simpler" is a very general concept
  - It might mean a *smaller* value, or a value *nearer* to some final value
  - Or *less data*
- Call pattern of a **recursive method** is not complicated
  - But can be hard to think about in general
  - Horstmann says it's complicated "*Don't think about it"*
    - I disagree!
- **A recursive method call *is an ordinary method call*:**
  - The JVM remembers where the call came from, and so where to return to when it finishes
  - At each call fresh memory is allocated for each parameter and local variable – discarded at return

Successful Recursion

- Every recursive call must simplify the computation in some way
  - The parameter is smaller, or less data, or…
  - If **not** then "infinite recursion" – very bad
- There must be special cases to handle the simplest computations directly
  - *The parameter gets "simpler" towards some final value*
  - Perhaps 0 or 1, or "no data remaining"

Other Ways to Compute Triangle Numbers

- The area of a triangle equals the sum:

```
1 + 2 + 3 + ... + width
```

- Using a simple loop:

```
double area = 0;
for (int i = 1; i <= width; i++)
   area = area + i;
```

- Using math:

$$1 + 2 + \ldots + n = n \times (n + 1)/2$$

=> `width * (width + 1) / 2`

13.4 The Efficiency of Recursion

- Fibonacci sequence:
  Sequence of numbers defined by

  $f_1 = 1$
  $f_2 = 1$
  $f_n = f_{n-1} + f_{n-2}$ "Each number is the sum of the previous two"

- First ten terms:

  1, 1, 2, 3, 5, 8, 13, 21, 34, 55

- We would like a method:

  `public long fib(int n)`

  `long` because Fibonacci numbers can be very large

RecursiveFib.java

```
// This program computes Fibonacci numbers using a recursive method.

public class RecursiveFib
{
   public static void main(String[] args)
   {
      Scanner in = new Scanner(System.in);
      System.out.print("Enter n: ");
      int n = in.nextInt();

      for (int i = 1; i <= n; i++)
      {
         long f = fib(i);
         System.out.println("fib(" + i + ") = " + f);
      }
   }
```

Continued

RecursiveFib.java (cont.)

```
    /**
       Computes a Fibonacci number.
       @param n an integer
       @return the nth Fibonacci number
    */
    public static long fib(int n)
    {
       if (n <= 2) { return 1; }
       else return fib(n - 1) + fib(n - 2);
    }
 }
```

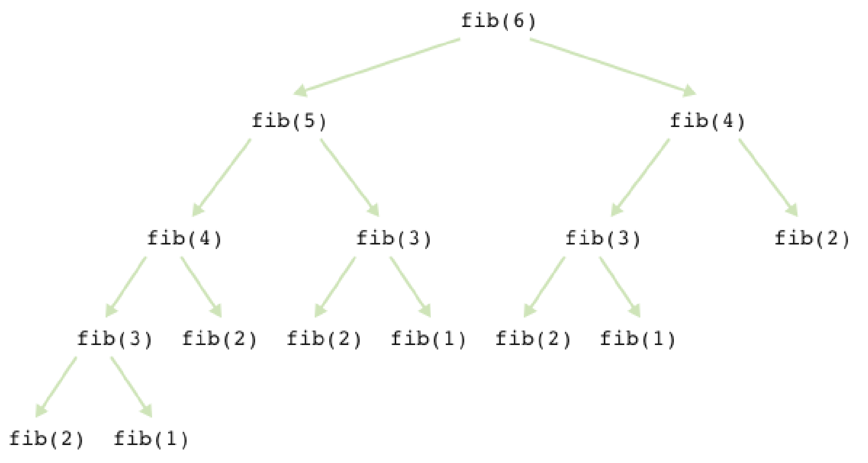
Program Run:

```
Enter n: 50
fib(1) = 1
fib(2) = 1
fib(3) = 2
fib(4) = 3
fib(5) = 5
...
fib(50) = 12586269025
```


Efficiency of Recursion

- Recursive implementation of `fib` is straightforward.
- Watch the output closely as you run the test program.
- First few calls to `fib` are quite fast.
- For larger values, the program pauses an amazingly long time between outputs.
- To find out the problem, let's map out the structure of the computation (next slide).

Call Pattern of Recursive `fib` Method



Efficiency of Recursion

- Method takes so long because it computes the *same values over and over*.
- Computation of `fib(6)` calls `fib(3)` three times.
- Better: Imitate the pencil-and-paper process to avoid computing the values more than once.

LoopFib.java

```java
   public static long fib(int n)
   {
      if (n <= 2) { return 1; }
      else
      {
         long olderValue = 1;
         long oldValue = 1;
         long newValue = 1;    // Dummy value
         for (int i = 3; i <= n; i++)
         {
            newValue = oldValue + olderValue;
            olderValue = oldValue;
            oldValue = newValue;
         }
         return newValue;
      }
   }
}
```


Efficiency of Recursion

- Occasionally, a recursive solution runs much slower than its iterative counterpart.
- In most cases, the recursive solution is only slightly slower.

Efficiency of Recursion

- Smart compilers can avoid recursive method calls if they follow simple patterns.
- Most compilers don't do that
- In many cases, a recursive solution is *easier to understand and implement correctly than an iterative solution* .
- "To iterate is human, to recurse divine."
  - L. Peter Deutsch

Summary

Control Flow in a Recursive Computation

- A recursive computation solves a problem by using the solution to the same problem with simpler inputs.
- For a recursion to terminate, there must be special cases for the simplest values.

Summary

Contrast the Efficiency of Recursive and Non-Recursive Algorithms

- Occasionally, a recursive solution runs much slower than its iterative counterpart. However, in most cases, the recursive solution is only slightly slower.
- In many cases, a recursive solution is easier to understand and implement correctly than an iterative solution.