Lecture 2: Recursion

Reading materials
Dale, Joyce, Weems: 4.1 - 4.3
OpenDSA: 2.6
Liang: 20

Topics Covered

1 Recursion 2
2 Computing Factorials 2
3 Fibonacci Numbers 3
4 Towers of Hanoi 5
5 Eight Queens Problem 6
6 Fractals 8
   6.1 Simple Circle Fractal 8
   6.2 Sierpinski Triangle Fractal 9
1 Recursion

Recursion is a powerful tool for solving certain kinds of problems. Recursion breaks a problem into smaller problems that are, in some sense, identical to the original, in such a way that solving the smaller problems provides a solution to the larger one.

Every recursive solution to a problem can be rewritten as an iterative solution and every iterative solution can be written as a recursive algorithm.

Every recursive function consists of two parts:

- **Base case** the case for which the solution can be stated non-recursively (this is the trivial case, but it is necessary for the recursion to terminate)
- **Recursive case** the case for which the solution is expressed in terms of a smaller version of itself.

In direct recursion the recursive function makes calls to itself. In indirect recursion, there is a chain of two or more function calls that eventually returns to the function that originated the chain.

Recursion comes with a price tag. In most case when recursion is implemented a function/method is called over and over again (by itself in direct recursion, or by another function/method in indirect recursion). As a programmer you need to worry about the function call overhead that this brings to your programs. Recursion is a powerful tool, but there are many problems (computing factorials, computing Fibonacci numbers) for which the iterative solution is as simple as the recursive solution. In all such cases, you should opt for iterative implementation. On the other hand, there are problems for which recursive solution is extremely simple (towers of Hanoi), and iterative solution is prohibitively complicated - these are the good candidates for recursive implementations.

2 Computing Factorials

Computation of factorials is a very intuitive example of recursive function:

\[ n! = n \times (n - 1)! \]

(The factorial of number \( n \) is a product of all the integers between 1 and \( n \).) The special case is that factorial of zero is equal to 1, i.e., \( 0! = 1 \).
The recursive algorithm that implements factorials is just Java implementation of the mathematical formula.

```java
public static long factorial ( long number ) {
    //base case
    if (number == 0 )
        return 1;
    //recursive case
    else
        return number * factorial ( number - 1 );
}
```

The iterative solution is equally simple and avoids the overhead of a function/method call:

```java
public static long factorial ( long number) {
    long tmpResult = 1;
    for ( ; number > 0; number--)
        tmpResult = tmpResult * number;
    return tmpResult;
}
```

**Source code**  Factorial01.java, Factorial02.java and Factorial03.java all provide recursive implementations of the factorial problem (they differ by the type of data that is displayed). Factorial04_Iterative.java implements the factorial function iteratively. Factorial_Competition.java compares the running time of recursive and iterative implementations.

### 3 Fibonacci Numbers

Computation of Fibonacci numbers is another mathematical concept that is defined recursively and tends to be among first recursive problems that are introduced. The Fibonacci numbers are defined as follows:

\[
\begin{align*}
    fib(0) &= 0, \\
    fib(1) &= 1, \\
    fib(n) &= fib(n - 1) + fib(n - 2). 
\end{align*}
\]

The interesting thing about this definition is that we have two base cases, not just one.
This definition, once again, translates trivially to Java implementation that uses recursion

```java
public static long fibonacci ( long number ) {
    //base cases
    if (number == 0 )
        return 0;
    else if (number == 1 )
        return 1;
    //recursive case
    else
        return fibonacci( number - 1 ) + fibonacci(number - 2);
}
```

The iterative solution requires a bit more thinking and making sure that two previous values are stored, but it is only a bit longer than the above recursive solution

```java
public static long fibonacci ( long number ) {
    //base cases
    if (number == 0 )
        return 0;
    else if (number == 1 )
        return 1;
    //recursive case
    else {
        long tmp1 = 0;
        long tmp2 = 1;
        long result = 0;
        int counter = 2;
        while ( counter <= number ) {
            result = tmp1 + tmp2;
            tmp1 = tmp2;
            tmp2 = result;
            counter++;
        }
        return result;
    }
}
```

The computation time gained when running the iterative solution is very large. Particularly because it lets us avoid repeated computations of the same values. See the source code example for the details.
Source code  Fibonacci01.java and Fibonacci02.java provide recursive implementations of the Fibonacci numbers (they differ by the type of data that is displayed). Fibonacci03_Iterative.java implements the iterative algorithm. Fibonacci_Competition.java compares the running time of recursive and iterative implementations (run it with increasing argument to the Fibonacci function and see what happens with the running times).

WARNING: Recursive solution to the Fibonacci numbers problem is very inefficient and hence, you should always use the iterative solution (unless, of course you are teaching someone about inefficient uses of recursive solutions).

4     Towers of Hanoi

Towers of Hanoi can be a game for a kid or a challenging computer science problem. It is an example of a problem for which the recursive solution is very short and clear, but the iterative solutions can be stated relatively simply but the implementations of them is not trivial (see the Wikipedia page for description of an iterative solution to the Towers of Hanoi problem: http://en.wikipedia.org/wiki/Tower_of_Hanoi#Iterative_solution).

Try to solve the towers of Hanoi problem by hand with 4 or more disks at http://www.mathsisfun.com/games/towerofhanoi.html.

Doing it might be confusing and it will most likely take you many more steps that are necessary for solving the problem.

Imagine that you could

- magically move the top \( n - 1 \) disks from Tower 1 to Tower 2,
- move the one disk from Tower 1 to Tower 3,
- magically move all the disks from Tower 2 to Tower 3.

This is, believe it or not, the recursive algorithm. The ”magic” is in recursive calls.
5 Eight Queens Problem

The eight queen problem is a chess related problem of how to place eight queens on a chess board so that all of them are “safe” - not under attack by another queen on the board. This requires you to know something about the queen and her powers in chess:

- queen attacks every position in its own column
- queen attacks every position in its own row
- queen attacks every position on its two diagonals

Given the above rules, there are many ways of arranging eight queens on the board without any attacks. But coming up with such arrangements may not be easy. The figure shows one of possible solutions, but there are many.

Solving the problem   First approach of solving the problem is as follows

- Place the first queen anywhere in the top row.
- Then find a position in the second row that is not under attack and put your second queen there.
- Then find a position in the third row that is not under attack and put your third queen there.
- ...
- Then find a position in the eighth row that is not under attack and put your eighth queen there.

PROBLEM: There is a major problem with this approach: you may not be able to find any space in a given row that is not under attack (and unless we can place one queen in each row, we cannot place all the queens since the chess board is eight by eight).

So how can the above solution be changed to handle situation in which we get to a row where all spaces are under attack?
• Whenever you get to a row in which all squares are under attack, go back to a previous row and move the queen in that row to an alternative unattacked position.

PROBLEM: This may solve some of the problem situations of the original approach, but it ignores two things:

1. There might not be another unattacked square in a previous row.
2. Even if there are alternative squares and we move the queen there, this does not guarantee that we can find an unattacked square in the current row.

Fortunately, the solution mentioned above can be applied multiple times, so if we find ourselves in the situation of 1 or 2 above, we go back to the row before the previous row and try to move the queen there. If that still does not work, we go back again, and so on. Eventually we might be moving the queen on the very first row.

This approach of solving problems is called backtracking: we keep going as long as we can, and if we run into problems, we just go back to a previous place where we had a choice to make and pick an alternative way. It seems that it might be hard to keep track of all the steps forwards and step backwards (especially if we go back more than one step). Recursion, can keep track of all such steps.

Pseudo code:

```python
placeEightQueens( chessboard )
    placeQueen ( chessboard, row = 0 )

placeQueen( chessboard, row )
    if row is greater than 8,
        return true (problem is solved)
    for each column from 0 to 8
        try to add queen to that column,
        if the row, column position is valid for the new queen
            (i.e., it is not under attack)
            then move on to the next row of the chessboard
            if placeQueen ( chessboard, row + 1) is successfull
            then return true to stop the for loop from checking
                remaining columns
    return false, no position in the current row is valid
```

If we implement the code according to this specification, you’ll see that it finds always the same solution. That’s because we always start checking from column 0 and advance one column at a time.

An alternative solution, randomizes which columns are used in the for loop.
6 Fractals

A fractal is a geometric figure that can be divided into parts, each of which is a smaller copy of the whole (if you could zoom in on a fractal, it still would look like the original shape).

Displaying a fractal is an ideal task for recursion, because fractals are inherently recursive.

6.1 Simple Circle Fractal

(You might have seen this example if you took the class with Craig Kapp.)

This figure may look complicated, but in fact it is very simple to produce if you know about recursion. There are very few steps required to produce this fractal. Assume that the point (0,0) in the image above is in the upper left corner.

1. for a given center (X, Y) and radius R, draw a circle
2. pick the center and a radius for the inner two circles as follows
   (a) circle 1: center = (X - R/2, Y), radius = R/2
   (b) circle 1: center = (X + R/2, Y), radius = R/2
   for each of (a) and (b) go back to step 1

In practice the program cannot recurse indefinitely, so we need to end somewhere. In the following code, we end when the radius becomes smaller than 10 (whatever units we operate in).
public void drawCircle(int x, int y, int radius) {
    // draw a circle of the desired size
    ellipse(x, y, 2 * radius, 2 * radius);
    // base case is to stop when radius is < 10
    if (radius >= 10) {
        int newRadius = radius / 2;
        int offset = radius / 2;
        drawCircle(x - offset, y, newRadius);
        drawCircle(x + offset, y, newRadius);
    }
}

Source Code: Circles_GUI.java shows the Processing code that produces the above circle fractal.

6.2 Sierpinski Triangle Fractal

This fractal is probably one of the most famous ones. You may not know its name (or not know how to pronounce it), but you most likely have seen it.

In this fractal, each triangle has three triangles inside of it: one attached to each corner (that’s why the middle looks empty). For each triangle we need to know three points in order to draw it. Then, we need to figure out the three points for each of the three triangles inside of it.

Assuming a "big" triangle has its vertices at points (x1, y1), (x2, y2), (x3, y3), what are the vertices of the three triangles inside?

Source Code: SierpinskiTriangle_GUI.java and SierpinskiTriangleAnimated_GUI.java show the Processing code that produces the above fractals.