Basic Recursion

Recursive methods, construction fundamentals, examples

What is recursive method?

- A **recursive method** is one that contains a call to itself.
- Idea behind recursion: Solve a "hard" problem by first solving an "easier" one and then use those results to solve the hard one.
- Example application:

  "A security system computes a check-sum on a non-negative integer by adding all of its digits. Write a method that has such an integer as a parameter and returns the sum of digits in the integer. For example, if the parameter is 1246, the method returns 13."

- One approach. Calculate the sum from right-to-left:
  - = 6 + the sum of digits in 124
  - = 6 + (4 + the sum of digits in 12)
  - = 6 + (4 + [2 + the sum of digits in 1])
  - = 6 + [4 + (2 + 1)]

Coding the sum

```
public class DigitSumExample{

 public static int sumDigits(int num){
    if (num == 0)
       return 0;
    return (num % 10) + sumDigits(num / 10);
 }

 public static void main(String[] args) {
    System.out.println("Sum for 1246: " + sumDigits(1246));
    System.out.println("Sum for 0: " + sumDigits(0));
    System.out.println("Sum for 1234567890: "
       + sumDigits(1234567890));
 }
}
```

DigitSum output

```
Sum for 1246: 13
Sum for 0: 0
Sum for 1234567890: 45
```

```
public static int sumDigits(int num){
    if (num == 0)
        return 0;
    return (num % 10) + sumDigits(num / 10);
}
```

- Execution of the first return statement does not generate any method call.
- Execution of the second return statement (for num ≠ 0) generates a recursive call, but with a smaller argument. It's an "easier" calculation.
- Eventually the calls stop when the argument for the call is 0.

sumDigits without recursion

```
public static int sumDigitsNonRecursive(int num){
    int sum = 0;
    while (num > 0) {
        sum = sum + num % 10;
        num = num / 10;
    }
    return sum;
}
```

```
// Recursive version
public static int sumDigits(int num){
    if (num == 0)
        return 0;
    return (num % 10) + sumDigits(num / 10);
}
```

Constructing recursive code

Good recursive methods have these characteristics :

1. The body of the recursive method is controlled by a selection statement (if, etc). At least one of the alternatives does not contain a recursive call. The condition associated with that alternative is the exit condition.
2. The exit condition describes the **base case** condition. The base case is the easiest problem to solve. Every recursive solution must have at least one base case.
3. At least one of the alternatives contains a recursive call. The new call is on a smaller (easier) version of the problem than the problem of the current call. This smaller/easier version is closer to the base case than the original problem.

Recursion and loops

- Recursive methods do not usually contain loops.
- To solve a problem by iteration, use recursion or use loops, but generally don't use both.
- Ultimately with recursion, the big problems have to be reduced to the very smallest problems (the base cases).

- If there is no base case or if the recursive call does not lead get closer to the base case, you can have **infinite recursion**. Infinite recursion is similar to an infinite loop except the method keeps calling itself over and over. Since each recursive call uses up some of the computer's memory, infinite recursion causes the program to run out of memory and crash. In Java, this results in a StackOverflowError.

Example – largest element

The method largest() finds the largest element in the tail of an array of ints. It has 2 parameters:

1. a - The array
2. start – The start index of the tail of the array

To find the largest element in the whole array, start at 0. For example:

```
int[] array = {5, 18, 0, 29, 9, 14}
int big = largest(array, 0); // Returns 29
```

```
public static int largest(int[] a, int start){
   if (start == a.length - 1)
      return a[start];
   return Math.max(a[start], largest(a, start + 1));
}
```

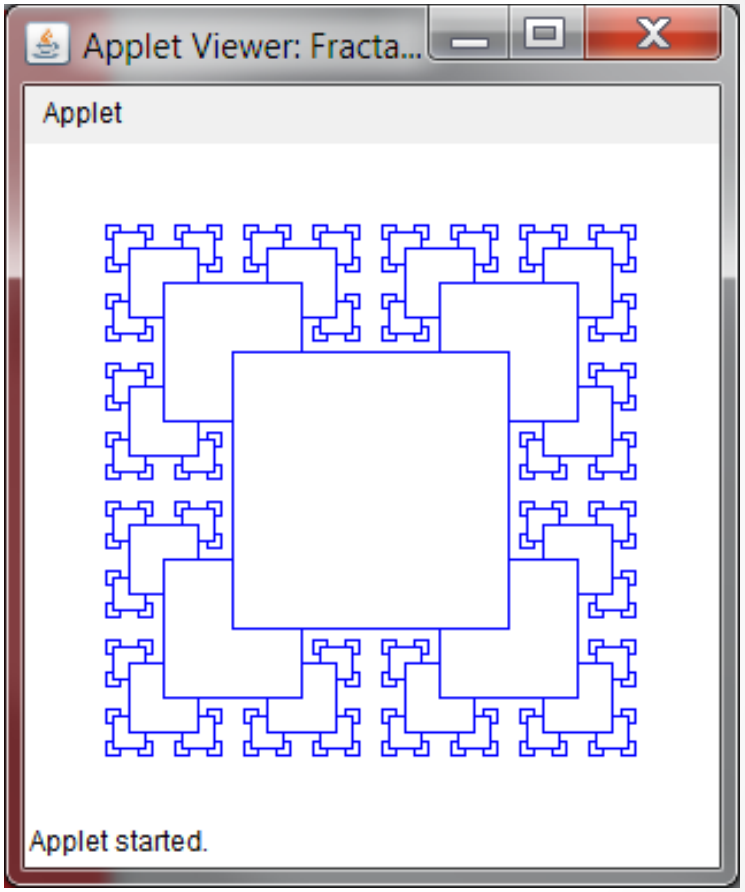
Largest element discussion

- The exit condition is (start == a.length - 1). The alternatives are a simple return statement and a recursive call.
- The base problem occurs when the condition is true. In that case, there is nothing to compare, since there is only one element.
- The recursive call moves the start index further down the array. In the subsequent version of the problem, you need to consider the tail of the array with 1 fewer elements.

```
public static int largest(int[] a, int start){
   if (start == a.length - 1)
      return a[start];
   return Math.max(a[start], largest(a, start + 1));
}
```

Fractal example

- A **fractal** is a kind of image that is defined recursively. Each part of the image is a smaller version of the whole.
- The image here is created by a recursive method of an applet.
- In the recursion, the smaller squares are drawn first.



Fractal code

```
import java.awt.*;
import java.applet.*;

public class Fractal extends Applet {

   private void drawSquare(int x, int y, int r, Graphics2D g2) {
      // Draw the rectangle boundary, clear the interior
      g2.draw(new Rectangle(x - r, y - r, 2*r, 2*r));
      g2.clearRect(x - r + 1, y - r + 1, 2*r - 1, 2*r - 1);
   }

// Recursive drawFractal() goes here

   public void paint (Graphics g) {
      Graphics2D g2 = (Graphics2D) g;
      g2.setColor(Color.blue);
      drawFractal(150, 150, 60, g2);
   }
}
```

Fractal code (cont)

```
// Recursive drawFractal method

public void drawFractal(int x, int y, int r, Graphics2D g2){
   if (r > 1) {
      // Draw the half-sized squares at the 4 corners
      drawFractal(x - r, y + r, r/2, g2);
      drawFractal(x + r, y + r, r/2, g2);
      drawFractal(x - r, y - r, r/2, g2);
      drawFractal(x + r, y - r, r/2, g2);

      // Now, draw the square in the middle
      drawSquare(x, y, r, g2);
   }
}
```

Fractal code trace

Initial call:

```
drawFractal(150, 150, 60, g2);
```

Method declaration:

```
public void drawFractal(int x, int y, int r, Graphics2D g2)
```

The initial value for r is 60. Each time drawFractal is called, it's with r/2.

```
drawFractal(x - r, y + r, r/2, g2);
```

r: 60 –> 30 –> 15 –> 7 –> 3 –> 1

Base condition:

```
if (r > 1) {
```

So 5 drawing iterations occur (none for 1).

Tail recursion

- Fractal was an example of a method where the recursive call occurred *before* other statements that execute after the call. (in this case, the drawSquare statement).
- Recursion that occurs like this, with other statements to execute after the recursive call is completed, can be translated to loops but not easily.
- Recursion in which there are no statements to execute after the return from a recursive call is **tail recursion**.
- Tail recursion is easy to convert to simple loops.