STUDENT OUTLINE

Lesson 15 – Recursion

INTRODUCTION: Recursion is defined as the process of a subprogram calling itself as part of the solution to a problem. It is a problem solving technique that can turn a long and difficult solution into a compact and elegant answer. It can also solve problems that are very difficult to solve with a straightforward iterative solution.

The key topics for this lesson are:

A. Recursion
B. Pitfalls of Recursion
C. Recursion Practice

VOCABULARY: RECURSION BASE CASE

DISCUSSION:

A. Recursion

1. Recursion occurs when a method calls itself to solve a simpler version of the problem. With each recursive call, the problem is different from, and simpler than, the original problem.

2. Recursion involves the internal use of a stack. A stack is a data abstraction that works like this: New data is "pushed," or added to the top of the stack. When information is removed from the stack it is "popped," or removed from the top of the stack. The recursive calls of a method will be stored on a stack, and manipulated in a similar manner.

3. The problem of solving factorials is our first example of recursion. The factorial operation in mathematics is illustrated below.

```
1! = 1
2! = 2 * 1          or      2 * 1!
3! = 3 * 2 * 1      or      3 * 2!
4! = 4 * 3 * 2 *1   or      4 * 3!
```

Notice that each successive line can be solved in terms of the previous line. For example, 4! is equivalent to the problem

```
4 * 3!
```

LAB EXERCISE

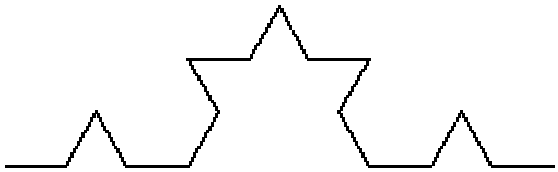
KochCurve

Background:

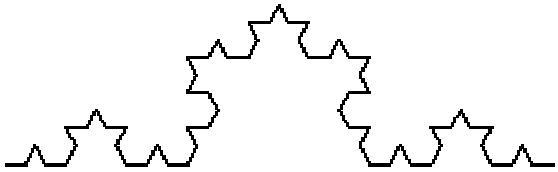
You can create a number of line drawings by starting with a simple pattern that is recursively subdivided in parts, each of which is (at least approximately) a reduced-size copy of the whole. The results are related to mathematical objects called "fractals", and so images generated in this manner are often called "fractal" images.

One example of a fractal curves is the "Koch curve" introduced by Swedish mathematician Helge von Koch in 1904. You can derive a Koch curve by beginning with the following basic four-segment piece:

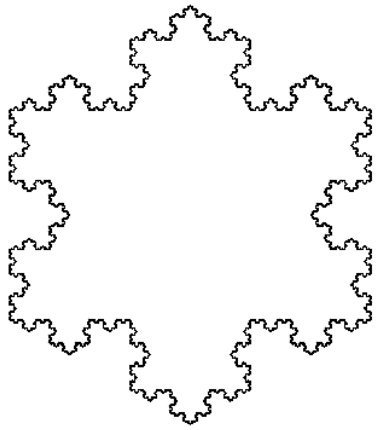
You then replace each line segment of the diagram with a smaller copy of itself.



You again replace each line segment of the diagram with a smaller copy of the basic shape.



Koch curves display an intricate beauty, as the number of levels of replacement increases. An even more remarkable figure can be created by joining three Koch curves as if they were the sides of a triangle. This figure is often referred to as a "Koch snowflake":



The procedure for creating a Koch curve is usually recursive. At each level, we observe that a Koch curve is made up of four smaller Koch curves. This process can be described in the following pseudocode:

```
if level < 1 then
  Move forward length pixels
else
  Draw a k-1 level Koch curve with segments 1/3 the current length
  Turn left 60 degrees
  Draw a k-1 level Koch curve with segments 1/3 the current length
  Turn right 120 degrees
  Draw a k-1 level Koch curve with segments 1/3 the current length
  Turn left 60 degrees
  Draw a k-1 level Koch curve with segments 1/3 the current length
```

Instructions:

1. Write a `KochCurve` program that defines a subclass of `DrawingTool` that provides a `drawKochCurve` method for drawing Koch curves. Each `drawKochCurve` method can take the number of levels and an initial size as its parameters. Sample useage of the method to draw a 6 level Koch curve of length 300 would be:

   ```
   KochCurve curve = new KochCurve();
   curve.drawKochCurve(6, 300);
   ```

2. Create a Koch Snowflake. The Koch snowflake includes three Koch curves arranged in a triangle.

3. Turn in your source code and run outputs.

LAB EXERCISE

StringReverse

Background:

Let $c_0c_1c_2...c_n$ be a string, where each c_i is a character. Then its reverse is $c_nc_{n-1}...c_2c_1c_0$.

Notice that the first character of the reverse is the last character of the original and that what follows it is the reverse of the string obtained by chopping off the last character of the original string. Combining this with the observation that the reverse of the empty string is itself, we surmise that

```
reverse(s) = { s                        if s has length 0
             { a + reverse(s')          otherwise
```

where + denotes concatenation, `s'` denotes s with its last character chopped off, and `a` denotes the last character of `s`.

Indeed, this recursive definition of `reverse` corresponds to our intended meaning. For example, if we apply the definition to the string "`abcde`" we get:

```
reverse("abcde") = 'e' + reverse("abcd")
                 = 'e' + ('d' + reverse("abc"))
                 = 'e' + ('d' + ('c' + reverse("ab")))
                 = 'e' + ('d' + ('c' + ('b' + reverse("a"))))
                 = 'e' + ('d' + ('c' + ('b' + ('a' + reverse("")))))
                 = 'e' + ('d' + ('c' + ('b' + ('a' + ""))))
                 = 'e' + ('d' + ('c' + ('b' + "a")))
                 = 'e' + ('d' + ('c' + "ba"))
                 = 'e' + ('d' + "cba")
                 = 'e' + "dcba"
                 = "edcba"
```

Each of the first five lines follows from the recursive case of the definition of `reverse`; the sixth line follows from the base case; the remaining lines follow from the meaning of concatenation.

Instructions:

1. Write a `StringReverse` program that repeatedly accepts a line of input and produces the reverse form of that line. The program should contain a reverse method that, given a String `s`, returns the reverse of `s`.

2. Your program should keep prompting the user for strings and printing out the reverse version until the user enters "`Q`" or "`q`", which will signal termination of the program.

3. Turn in your source code and run outputs.

LAB EXERCISE

Fibonacci

Background:

The Fibonacci number series is defined as follows:

Position	0	1	2	3	4	5	6	7	8	etc.
Fib number	0	1	1	2	3	5	8	13	21	etc.

Positions 0 & 1 are definition values. For positions greater than 1, the corresponding Fibonacci value of position `N = Fib (N-1) + Fib (N-2)`.

Assignment:

1. Write a recursive method that takes in a single integer (`x >= 0`) and returns the appropriate Fibonacci number of the Fibonacci number series.

2. Write a non-recursive Fibonacci method that solves the same problem as the recursive version.

3. Write a method that solves a multiplication problem recursively. Use this method header:

```
int mult(int a,  int b)
//  solves for (a * b) by recursively adding a, b times.
//  precondition:  0 <= a <= 10;  0 <= b <= 10.
```

Instructions:

Use these sample run output values:

Recursive fibonacci: `fib(0)`, `fib(3)`, `fib(11)`

Non-recursive Fibonacci: `nonRecFib(1)`, `nonRecFib(5)`, `nonRecFib(14)`

Recursive multiplication: `mult(0,4)`, `mult(3,1)`, `mult(7,8)`, `mult(5,0)`

A recursive method to solve the factorial problem is given below. Notice in the last line of the method the recursive call. The method calls another implementation of itself to solve a smaller version of the problem.

```
int fact(int n)
//  returns the value of n!
//  precondition:  n >= 1
{
  if (n == 1)
    return 1;
  else
    return  n * fact(n - 1);
}
```

4. The base case is a fundamental situation where no further problem solving is necessary. In the case of finding factorials, the answer of `1!` is by definition `== 1`. No further work is needed.

5. Suppose we call the method to solve `fact(4)`. This will result in four calls of method `fact`.

   `fact(4)`: This is not the base case (`1 == n`), so we return the result of `4 * fact(3)`. This multiplication will not be carried out until an answer is found for `fact(3)`. This leads to the second call of `fact` to solve `fact(3)`.

   `fact(3)`: Again, this is not the base case and we return `3 * fact (2)`. This leads to another recursive call to solve `fact(2)`.

   `fact(2)`: Still, this is not the base case, we solve `2 * fact(1)`.

   `fact(1)`: Finally we reach the base case, which returns the value 1.

6. When a recursive call is made, the current computation is temporarily suspended and placed on the stack with all its current information available for later use.

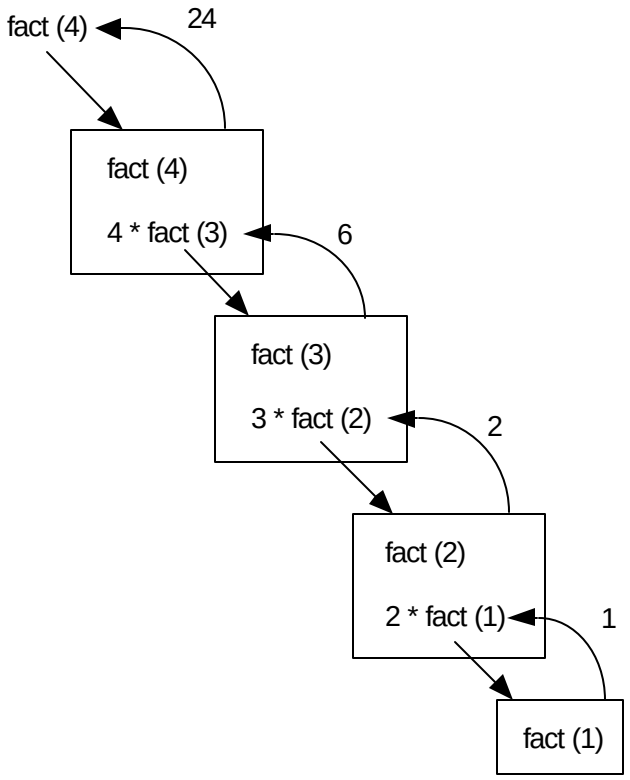
7. A completely new copy of the method is used to evaluate the recursive call. When that is completed, the value returned by the recursive call is used to complete the suspended computation. The suspended computation is removed from the stack and its work now proceeds.

8. When the base case is encountered the recursion will now unwind and result in a final answer. The expressions below should be read from right to left.

```
fact(4) = 4 * fact(3) = 3 * fact(2) = 2 * fact(1) = 1

24 ← 4 * 6 ← 3 * 2 ← 2 * 1
```

Here is a picture. Look at what happens:



Each box represents a call of method `fact`. To solve `fact(4)` requires four calls of method `fact`.

9. Notice that when the recursive calls were made inside the **`else`** statement, the value fed to the recursive call was (`n-1`). This is where the problem is getting smaller and simpler with the eventual goal of solving `1!`.

B. Pitfalls of Recursion

1. If the recursion never reaches the base case, the recursive calls will continue until the computer runs out of memory and the program crashes. Experienced programmers try to examine the remains of a crash. The

message “stack overflow error” or “heap storage exhaustion” indicates a possible runaway recursion.

2. When programming recursively, you need to make sure that the algorithm is moving toward the base case. Each successive call of the algorithm must be solving a simpler version of the problem.

3. Any recursive algorithm can be implemented iteratively, but sometimes only with great difficulty. However, a recursive solution will always run more slowly than an iterative one because of the overhead of opening and closing the recursive calls.

C. Recursion Practice

1. Write a recursive power method that raises a base to some exponent, n. We will use integers to keep things simple.

```
double power(int base, int n)
//  Recursively determines base raised to
//  the nth power. Assumes 0 <= n <= 10.
```

SUMMARY/ REVIEW: Recursion takes some time and practice to get used to. Eventually you want to be able to think recursively without the aid of props and handouts. Study the examples provided in these notes and work it through for yourself. Recursion is a very powerful programming tool for solving difficult problems.

ASSIGNMENT: Lab Exercise L.A.15.1, *Fibonacci*
Lab Exercise L.A.15.2, *StringReverse*
Lab Exercise L.A.15.3, *KochCurves*