Introduction

You are already familiar with using loops to solve problems, which is known as *iteration*.

This activity explores a new way to solve problems. To solve a large problem, we will use the solution of a smaller but similar problem. Eventually, we reach the most basic problem, for which we know the answer. We will explore this idea in three models.

If you have 3 people, combine Facilitator and Process Analyst.

If you have 4 people, split up into two pairs to program (Spokesperson and Process Analyst serve as drivers, Facilitator and Quality Control serve as navigators), but merge back to larger group of four to answer questions.

Team Roles	Team Member
Facilitator: reads the questions aloud, keeps track of time and makes sure everyone contributes appropriately.	
Spokesperson: talks to the instructor and other teams. Compiles and runs programs when applicable.	
Quality Control: records all answers & questions, and provides team reflection to team & instructor.	
Process Analyst: Considers how the team could work and learn more effectively.	

The facilitator should note the amount of time spent on each Model here:

Model 1:
Model 2:
Model 3:

Model I. (20 min) Factorial

When faced with a large problem to solve, we might seek to use a solution to a smaller, simpler problem. If we repeatedly decompose the original problem into smaller simpler expressions, we will eventually identify the most simple or basic component that can't be broken down any further. This is referred to as the ***base case***.

Critical Thinking Questions

1. Consider two different ways to show how to calculate 4! (the product of the numbers from 1 to 4).

   a) Write out all numbers that explicitly need to be multiplied

      4! =

   b) Write the expression using 3!

      4! =

2. Write an expression similar to CTQ 1b showing how each factorial can be calculated in terms of a "simpler" factorial. (By definition, 0! = 1.)

   a) 3! =

   b) 2! =

   c) 1! =

   d) 100! =

3. Generalize your group's answer to CTQ 2 in terms of *n* to create an equation for factorial that would be true for all factorials except the base case.

   n! =

4. What would your group propose to be the ***base case*** of a factorial method? Record your group's justification for this answer.

5. Assume you have defined a static method `factorial(int n)` that returns *n*!
   a. Copy and paste your answer to CTQ 2d, which should include how 100! can be calculated using a "simpler" factorial.

   b. Convert your expression to Java code, making the appropriate method call to the `factorial`

method and “hard coding” numbers as arguments.

c. Now convert your equation for CTQ 3 to Java code that would calculate n! This time, you should not “hard code” numbers as arguments.

6. If you used your answer for the previous question IN the definition for the `factorial` method, how does this method call differ from method calls you have used in previous programs?

7. Is a loop necessary to calculate 3! based on your group’s answers above? Describe your group’s reasoning.

8. What type of programming control statement (branching or looping) is required to differentiate the successive method calls and the base case?

Model II. (45 min) Recursion

When a method makes a call to itself this is referred to as ***recursion***. To define a recursive method in Java, you should write an if-statement that checks for the base case. When the operation is *not* the base case, you include a call to the method you are writing.

In a new file called RecursionExample.java, create a class called RecursionExample. Then add the following factorial definition:

```
/** calculates n factorial
*/
public static int factorial(int n) {
     System.out.println("n is " + n);
     if (n == 0)
         return 1;
     else {
         System.out.println("need factorial of "
                            + (n-1));
         int answer = factorial(n-1);
         System.out.println("factorial of " +(n-1)
                            + " is " + answer);
         return answer * n;
     }
}

public static void main(String[] args) {
     System.out.println(factorial(3));
}
```

Critical Thinking Questions

9. How many distinct calls are made to the factorial method to calculate the factorial of 3? Identify the value of the parameter n for each of these separate calls.

10. Examine your output when you ran the main method. How many lines were printed by the program?

11. For each printed line, identify which distinct factorial method call printed that line. In other words, which lines were printed by `factorial(3)`, which lines were printed by `factorial(2)`, and so on.

12. What happens if you try to calculate the factorial of a negative number? Fix the bug in the factorial method so this does not occur.

13. Consider two different ways to show how to calculate $\sum_{i-1}^{4} i$

    a) Write out all numbers that explicitly need to be summed

    $\sum_{i=1}^{4} i =$

    b) Write an expression showing how this sum can be calculated in terms of a "simpler" sum.

    $\sum_{i=1}^{4} i =$

14. Generalize your group's answer to CTQ 13b in terms of *n* that would be true for all sums except the base case.

15. What is the "value" of the base case of a summation expression?

Team Programming:

A. In a new file, RecursionTeam.java, create a static `recursiveSummation` method that takes a single `int` parameter *n* and returns the summation $\sum_{i=1}^{n} i$ (as an `int`) using recursion. Your method should have an if-else statement and NO loops. Your method can return any number you wish for negative numbers, but it should not crash. Your method should *not* contain all the System.out.println commands that were included as part of the original factorial method. If you use them to help you debug, you should comment them out once your method works correctly. You can test your method in Dr. Java's Interactions Pane.

B. All recursive methods can also be written with iteration (*i.e.,* loops). In the same file, add a static `iterativeSummation` method that takes a single `int` parameter *n* and returns the same summation as an `int`. This method should include either a while-loop or a for-loop. Again, you can return any number you wish for negative numbers, as long as your method does not crash.

Critical Thinking Question

16. Describe the similarities and differences in the performance of the iterative and recursive algorithms when calculating the sum of successively larger values of *n*.

17. Identify any advantages of using an iterative algorithm in contrast to a recursive algorithm.

Model III. (20 min) Order of Execution

Not all recursive methods need to have an if and else statement. Sometimes you only need the if-statement, while other times you might need multiple branches (if-else if-else). For example, type in the following method into RecursionExample.java (replacing your main method):

```
public static void countdown(int n) {
    if (n >= 1) {
        System.out.println(n);
        countdown(n-1);
    }
}

public static void main(String[] args) {
    countdown(10);
}
```

Critical Thinking Question

18. What is the ***base case*** for a recursive algorithm that does not require an `else` branching statement?

19. Modify the `countdown` method so it counts up (*i.e.*, printing 1 to 10) instead of counting down. You should be able to make this change by cutting-and-pasting entire lines of code; edits to individual lines of code (such as changing numbers) should not be necessary. Give the lines of code that were changed here:

20. Consider the modification changes required if this algorithm was written iteratively. Identify one advantage of using a recursive algorithm in contrast to an iterative algorithm.

Team Programming:

i. Open RecursionTeam.java again. Define a static *power* method that takes two ints as parameters, x and n, calculates and returns x^n. Use the following recursive definition:

$$x^n = \left(x^{n/2}\right)^2 \text{ when } n \text{ is even}$$
$$x^n = x \times \left(x^{(n-1)/2}\right)^2 \text{ when } n \text{ is odd}$$
$$x^0 = 1 \text{ when } n \text{ is } 0$$

Notice that there are multiple cases, so you should use an if-else if-else in your recursive method. You should NOT use any loops, nor should you call the Math.pow method. However, to calculate any number squared (y^2). you should multiply the number by itself ($y \times y$).

ii. Explain whether or not this recursive *power* method could be written iteratively using the same algorithm. Identify one advantage of using this recursive algorithm to an iterative approach. Include your answer as a comment in your Java file.

Group Reflection:

Quality Control:

1. What are the two parts of a recursive program?

2. Explain how the argument in a recursive program causes the recursion to repeat yet eventually stop?

Process Analyst: Has your group improved in ability to solve more difficult problems together? If yes, justify your answer. If no, brainstorm on how your group might improve.