*Please complete the textbook sections 6.3 and 6.4 before beginning this assignment.*

# CS 278/465
# Programming Assignment 8
# Induction and Recursion

All programs you submit in this class must follow the Documentation and Style Guidelines. This document can be found in the Canvas Modules.

All programs you submit in this class must compile with the Oracle Java compiler on the Linux machines in SH 118 or SH 118B.

The programs you submit in this course must be your own work.
Any program that shows evidence of cheating will receive a grade of zero.

## Background

Recursive thinking in programming is closely related to mathematical induction. The following is an excerpt from the book “Thinking recursively with Java” by Eric Roberts where he compares induction and recursion.

Recursive thinking has a parallel in mathematics which is called mathematical induction. In both techniques, one must (1) determine a set of simple cases for which the proof or calculation is easily handled and (2) find an appropriate rule which can be repeatedly applied until the complete solution is obtained. In recursive applications, this process begins with the complex cases, and the rule successively reduces the complexity of the problem until only simple cases are left. When using induction, we tend to think of this process in the opposite direction. We start by proving the simple cases, and then use the inductive rule to derive increasingly complex results. …

There are several ways to visualize the process of induction. One which is particularly compelling is to liken the process of an inductive proof to a chain of dominos which are lined up so that when one is knocked over, each of the others will follow in sequence. In order to establish that the entire chain will fall under a given set of circumstances, two things are necessary. To start with, someone has to physically knock over the first domino. This corresponds to the base step of the inductive argument. In addition, we must also know that, whenever any domino falls over, it will knock over the next domino in the chain. If we number the dominos, this requirement can be expressed by saying that whenever domino N falls, it must successfully upset domino N+l. This corresponds to using the inductive hypothesis to establish the result for the next value of N.

More formally, we can think of induction not as a single proof, but as an arbitrarily large sequence of proofs of a similar form. For the case N = 1, the proof is given explicitly. For larger numbers, the inductive phase of the proof provides a mechanism to construct a complete proof for any larger value. For example, to prove that a particular formula is true for N = 5, we could, in principal, start with the explicit proof for N = 1 and then proceed as follows:

```
Since it is true for N = 1, I can prove it for N = 2.
Since I know it is true for N = 2, I can prove it for N = 3.
Since I know it is true for N = 3, I can prove it for N = 4.
Since I know it is true for N = 4, I can prove it for N = 5.
```

In practice, of course, we are not called upon to demonstrate a complete proof for any value, since the inductive mechanism makes it clear that such a derivation would be possible, no matter how large a value of N is chosen.

Recursive algorithms proceed in a very similar way. Suppose that we have a problem based on a numerical value for which we know the answer when N = 1. From there, all that we need is some mechanism for calculating the result for any value N in terms of the result for N- 1. Thus, to compute the solution when N = 5, we simply invert the process of the inductive derivation:

```
To compute the value when N = 5, I need the value when N = 4.
To compute the value when N = 4, I need the value when N = 3.
To compute the value when N = 3, I need the value when N = 2.
To compute the value when N = 2, I need the value when N = 1.
I know the value when N = 1 and can use it to solve the rest.
```

The following example illustrates the relation between mathematical induction and recursive implementation. Consider the following summation:

$$\sum_{i=1}^{n} 2i$$

We can prove by mathematical induction that

$$\sum_{i=1}^{n} 2i = n(n+1)$$

First, show it's true for the base case: n = 1.

$$\sum_{i=1}^{n} 2i = 2 = 1 * 2 = n(\,n+1)$$

Next, suppose that for any n = k (1 ≤ k), we have

$$\sum_{i=1}^{k} 2i = k(\,k+1)$$

This is the inductive hypothesis.

Now for the inductive step.   We need to show that:

$$\sum_{i=1}^{k+1} 2i = (k+1)(\,k+2)$$

Starting with the summation on the left hand side (LHS), we separate the last term from the summation.

$$\sum_{i=1}^{k+1} 2i = \sum_{i=1}^{k} 2i \ + 2(k+1)$$

Now, substitution for the summation from 1 to k.

$$\sum_{i=1}^{k} 2i \ + 2(k+1) = \ k(k+1) + 2(k+1)$$

Next, some algebra.

$$k(k+1) + 2(k+1) = (k+2)(k+1) = (k+1)(k+2)$$

*distributive -->* *commutative -->*

We have arrived at the right hand side (RHS) of what we were trying to show. ▮

The following Java method computes the summation shown above (the LHS), using recursion:

```
public static int summation(int k) {

   // base case:
   if (k == 1) return 2;

   // recursive (inductive) step:

   return summation(k - 1) + 2 * k;

}
```

This method has the same features as the proof by mathematical induction:
1) base case
2) recursive (inductive) step
- uses the "inductive hypothesis" to compute the value of the summation up to k - 1,
- computes the last term from the summation, and
- gets the final answer by adding the computed values.

You can use the method to verify that for every k >= 1, summation(k) = k(k + 1).

The method relies on the correctness of the recursive call to correctly compute the result.

The parameter passed to the recursive call is smaller than the incoming parameter.

Without this step we could not have established the result. Similarly, in the proof by induction, without using the inductive hypothesis we cannot establish the final result.

## The Assignment

Write a complete Java program named PA8.java that contains the following 3 methods.

1. Write a public, static, recursive method called sum1 that will accept an integer **k** as a parameter and compute the following summation:

   $$\sum_{i=0}^{k} 2i$$

   Use comments in your code to explicitly show where your base case is and where your recursive case is.

   The base case must be written before the recursive case.

2. Write a public, static, recursive method called sum2 that will accept an integer **k** as a parameter and compute the following summation:

   $$\sum_{i=1}^{k} i(i+1)$$

   Use comments in your code to explicitly show where your base case is and where your recursive case is.

   The base case must be written before the recursive case.

3. Write a main method that would prompt the user to enter the value of k and output the values of the two summations from parts 1 and 2.

   **Example:**

   A sample run of your program would look like the following (user input is in red):

   ```
   Please enter the value of k: 2

   The value of the 1st summation is 6.

   The value of the 2nd summation is 8.
   ```

Methods from parts 1 and 2 must be recursive. If they are not recursive you will not get credit for them. You need to explicitly show in your code where your base case is and where your recursive (inductive) case is in both methods (use comments for that). You will lose points for not having these comments in your code.

All print statements must be in the main method.  Output must contain meaningful messages as shown above.

**Submit PA8.java on Canvas.**