CS 278
Lab: Induction and Recursion.

Recursive thinking in programming is closely related to mathematical induction. The following is a quote from "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 $\sum_{i=1}^{n} 2i = n(n+1)$.

Base case: n=1. Then, $\sum_{i=1}^{1} 2i = 2 = n(n+1)$.

Suppose that for any n= *k (0 ≤ k)*, we have $\sum_{i=1}^{k} 2i = k(k+1)$ (this is the *inductive hypothesis*).
Inductive step: We need to show that

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

LHS =$\sum_{i=1}^{k+1} 2i = \sum_{i=1}^{k} 2i + 2(k+1)$. (We separate the last term from the summation).
By the inductive hypothesis, LHS = $k(k+1) + 2(k+1) = (k+1)(k+2)$ =RHS

The following program computes the summation of the left hand side of the equation, using recursion:

```
int summation(int n) {
  // base case:
  if (n==1) return 2;

  // recursive (inductive) step:
  int inductive_hp = summation(n-1); // the inductive hypothesis corresponds to a recursive call to compute
                                      // the summation up to n-1,
  int last_term = 2*n; // this is the last term separated from the summation

  return (inductive_hp + last_term);
}
```

The above program has the same ingredients as the proof by mathematical induction:

- base case,
- recursive (inductive) step, which
    - uses the “inductive hypothesis” to compute the value of the summation up to n-1,
    - separates and computes the last term from the summation, and
    - gets the final answer by adding the computed values.

You can use this program to verify that for every input n > 0, summation(n) = n(n+1).
As in the proof by induction, where the inductive hypothesis relies on the fact that $\sum_{i=1}^{n-1} 2i = (n-1)n$ is proven, our program relies on the correctness of the recursive call, whose result is stored in the variable `inductive_hp`, to correctly compute the result. This works because the input of the recursive call, is smaller than the input of the main call (they are, n-1, and n, respectively). Without this step we could not had established the result. Similarly, in the proof by induction, without using the inductive hypothesis we cannot establish the final result.

Part 1. Write a recursive method called **sum1** that will accept n as a parameter and compute the following summation:

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

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

Part 2. Write a recursive method called **sum2** that will accept n as a parameter and compute the following summation:

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

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

Part 3. Write a program that would prompt the user to enter the value of n and output the values of the two summations from parts 1 and 2.

Example:
An output produced by your program may look like the following (user input is in **green**):

```
Please enter the value of n: 2
The value of the 1st summation is 7
The value of the 2nd summation is 8
```

Implementation details:

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 some points for not having these comments in your code).

Notice that you will need to separate off the final term in the summations to implement recursion.

What to submit:

- Submit the source code of your program using Canvas.
- If you write your program in a programming language other than Java, then submit instructions on how to compile and run your program on CS machines.