AP Computer Science Name _____ Per ____
Chapter 8 Notes – Recursion

re·cur·sion [ri-**kur**-zh*uh* n]

–noun *See* recursion.

8.1 Recursive Thinking

Recursion is a programming technique in which a method can call itself to solve a problem. Every recursive method has two distinct parts:

- A base case or termination condition that causes the method to end.
- A non-base case whose actions move the algorithm towards the base case and termination.

```
public void drawLine (int n)
{
   if (n == 0)
      System.out.println("All done!");
   else
   {
      for (int i = 1; i <= n; i++)
         System.out.print ("*");
      System.out.println();
      drawLine(n-1);
   }
}
```

- What is the output for the call `drawLine(3)` ?

- What is the base case in the `drawLine` method?

Infinite Recursion

All recursive definitions must have a non-recursive part or base case. If they don't, there is no way to terminate the recursive path. A definition without a non-recursive part causes infinte recursion. This problem is similar to an infinite loop with the definition itself causing the infinite "loop".

```
public void catastrophic (int n)
{
   System.out.println(n);
   catastrophic(n);
}
```

Recursive Definition

Mathematical formulas often are expressed recursively. A good strategy for writing recursive methods is to first state the algorithm recursively in words.

Write a method that returns *n*! (*n* factorial). *n*!, for any positive integer *n*, is defined to be the product of all integers between 1 and *n* inclusive. This definition can be expressed recursively as:

$$1! = 1 \qquad n = 1$$
$$n * (n - 1)! = n! \qquad n > 1$$

The concept of the factorial is defined in terms of another factorial until the base case of 1! is reached

```
public static int factorial(int n)                              5!
{









}
```

Write a recursive method `revDigs` that outputs its integer parameter with the digits reversed. For example,

`revDigs(147)`	outputs	741
`revDigs(4)`	outputs	4

First, describe the process recursively:

```
public static void revDigs( int n )
{









}
```

8.2 Recursive Programming

A method in Java can invoke itself; if set up that way, it is called a *recursive* method. The code of a recursive method must be structured to handle both the base case and the recursive case. Each call to the method sets up a new execution environment, with new *parameters* and new *local variables.* As always, when the method execution completes, control returns to the method that invoked it (which may be an earlier invocation of itself).

Example: For the method below, what does `result(5)` return? Draw the recursive call tree. If $n > 0$, how many times will `result` be called to evaluate `result(n)` (including the initial call)?

```
public int result(int n)
{
   if (n == 1)
      return 2;
   else
      return 2 * result(n – 1);
}
```

Example: What would be returned by `t(5)` using the following method:

```
//Precondition: n >= 1
public int t(int n)
{
   if (n == 1 || n == 2)
      return 2 * n;
   else
      return t(n - 1) – t(n - 2);
}
```

Example: Consider the problem of computing the sum of all the numbers between 1 and any positive integer *n*, inclusive. What will be the result of `sum(4)`?

```
public int sum (int num)
{
      int result = 0;
      if (num == 1)
         result = 1;
      else
         result = num + sum (num - 1);
      return result;
}
```

Recursion vs Iteration

Just because we can use recursion to solve a problem, doesn't mean we should. For instance, we usually would not use recursion to solve the sum of 1 to *n* problem, because the *iterative* version is easier to understand. Write the iterative version of the method below:

```
public int sum (int num)
{









}
```

You must be able to determine when recursion is the correct technique to use. Every recursive solution has a corresponding *iterative* solution. Recursion has the overhead of multiple method invocations. Nevertheless, *recursive* solutions often are more simple and elegant than iterative solutions.

Rewrite the following iterative method as a recursive method that computes the same thing. NOTE: your recursive method will require an extra parameter.

```
public int nums(int x)
{
   int count = 0, factor = 2;
   while(factor < x)
   {
        if (x % factor = = 0) count++;
        factor++;
   }
   return count;
}
```

- Write a *recursive* method to compute the power of x^n for non-negative n.

- Write an *iterative* method to compute the power of x^n for non-negative n.

Indirect Recursion

A method invoking itself is considered to be *direct* recursion. A method could invoke another method, which invokes another, etc., until eventually the original method is invoked again.

For example, method `m1` could invoke `m2`, which invokes `m3`, which in turn invokes `m1` again until a base case is reached. This is called *indirect* recursion, and requires all the same care as direct recursion. It is often more difficult to trace and debug.

8.3 Using Recursion

Strings

Example: Write a recursive method, `len`, which accepts a string and returns the number of characters in the string .

The length of a string is:

- 0 if the string is the empty string ("").
- 1 more than the length of the rest of the string beyond the first character .

Example: Write a recursive method named `makeStarBucks` which receives a non-negative integer `n` and returns a `String` consisting of `n` asterisks followed by `n` dollars signs.

```
makeStarBucks(5) -> *****$$$$$

makeStarBucks(3) ->  ***$$$
```

Arrays

Example: Write a `void` method named `clear` which accepts an `integer` array, and the number of elements in the array and sets the elements of the array to 0. The items can be cleared recursively as follows:

- An array of size 0 is already cleared;
- Otherwise, set the first element of the array to 0, and clear the rest of the array

Example: Write a `void` method named `init` which accepts an integer array, and the number of elements in the array and recursively initializes the array so that `a[i] == i`. The elements can be initialized recursively as follows:

- An array of size 0 is already initialized;
- Otherwise
    - set the last element of the array to n-1 (where n is the number of elements in the array, for example, an array of size 3 will have its last element (index 2) set to 2; and
    - initialize the portion of the array consisting of the first n-1 elements (i.e., the other elements of the array)

8.4 Recursion and Sorting

Mergesort divides a list in half; recursively sorts each half, and then combines the two lists. At the deepest level of recursion, one-element lists are reached. A one-element list is already sorted. The work of the sort comes in when the sorted sublists are merged together.

Here is how the mergesort works:

- Break the array into two halves.
- Mergesort the left half.
- Mergesort the right half.
- Merge the two subarrays into a sorted array.

Example a)

5	-3	2	4	0	6

Example b)

5	9	2	1	2	4	3	7