Chapter 12 Notes/Handout – Recursion

recursion: The definition of an operation in terms of itself.

- Solving a problem using recursion depends on solving smaller occurrences of the same problem.

recursive programming: Writing methods that call themselves to solve problems recursively.

- An equally powerful substitute for *iteration* (loops)
- Particularly well-suited to solving certain types of problems

Every recursive method has two distinct parts:

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

Here is the framework for a simple recursive method that has no specific return type.

```
public void recursiveMeth( ... )
{
    if (base case)
        < Perform some action >
    else
    {
        < Perform some other action >
        recursiveMeth( ... );      //recursive method call
    }
}
```

The base case typically occurs for the simplest case of the problem, such as when an integer has a value of 0 or 1. Other examples of base cases are when some key is found, or an end-of-file is reached. A recursive algorithm can have more than one base case.

In the `else` or nonbase case of the framework shown, the code fragment *< Perform some other action >* and the method call `recursiveMeth` can sometimes be interchanged without altering the net effect of the algorithm. Be careful though, because what *does* change is the order of executing statements. This can sometimes be disastrous.

Example 1

```
public void drawLine(int n)
{
    if (n == 0)
        System.out.println("That's all, folks!");
    else
    {
        for (int i = 1; i <= n; i++)
            System.out.print("*");
        System.out.println();
        drawLine(n - 1);
    }
}
```

The method call `drawLine(3)` produces this output:

```
***
**
*
That's all, folks!
```

Example 2

```
//Illustrates infinite recursion.
public void catastrophe(int n)
{
    System.out.println(n);
    catastrophe(n);
}
```

Try running the case `catastrophe(1)` if you have lots of time to waste!

Note: A recursive method must have a base case!

To come up with a recursive algorithm, you have to be able to frame a process *recursively* (i.e., in terms of a simpler case of itself). This is different from framing it *iteratively*, which repeats a process until a final condition is met. A good strategy for writing recursive methods is to first state the algorithm recursively in words.

Example 3

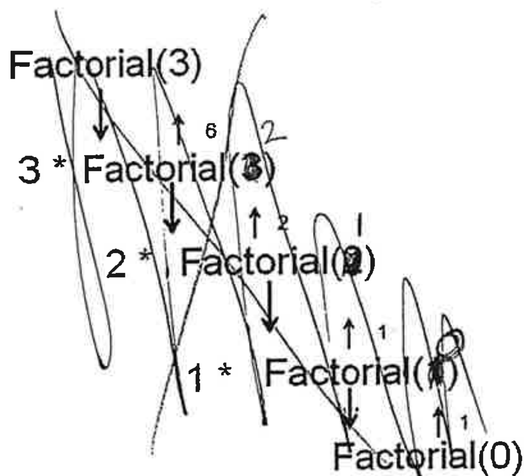
Write a method that returns $n!$ (n factorial).

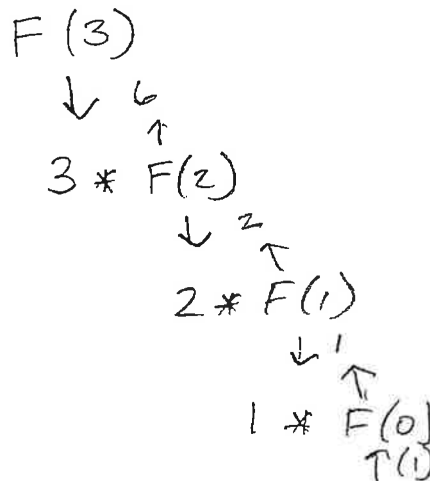
$n!$ defined iteratively	$n!$ defined recursively
$0! = 1$	$0! = 1$
$1! = 1$	$1! = (1)(0!)$
$2! = (2)(1)$	$2! = (2)(1!)$
$3! = (3)(2)(1)$	$3! = (3)(2!)$
...	...

The general recursive definition for $n!$ is

$$n! = \begin{cases} 1 & n = 0 \\ n(n-1)! & n > 0 \end{cases}$$

```
 * Precondition:  n >= 0.
 * Postcondition: returns n! */
public static int factorial(int n)
{
    if (n == 0)      //base case
        return 1;
    else
        return n * factorial(n - 1);
}
```





Example 4

Recall the Fibonacci sequence 1, 1, 2, 3, 5, 8, 13, ... The nth Fibonacci number equals the sum of the previous two numbers if $n \geq 3$. Recursively,

$$\text{Fib}(n) = \begin{cases} 1, & n = 1, 2 \\ \text{Fib}(n-1) + \text{Fib}(n-2), & n \geq 3 \end{cases}$$

```
/* Precondition:  n >= 1.
 * Postcondition: Returns the nth Fibonacci number. */
public static int fib(int n)
{
    if (n == 1 || n == 2)
        return 1;
    else
        return fib(n - 1) + fib(n - 2);
}
```

Example 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

```
public static void revDigs(int n)
{
    if (n % 10 == 0){
        System.out.println();
    }
    else {
        System.out.print(n % 10);
        revDigs(n / 10);
    }
}
```

Example 6

- Consider the following recursive method:

```
public static int mystery(int n) {
    if (n < 10) {
        return (10 * n) + n;
    } else {
        int a = mystery(n / 10);
        int b = mystery(n % 10);
        return (100 * a) + b;
    }
}
```

  - What is the result of the following call?
    `mystery(348)`

Example 7

- Write a recursive method `pow` accepts an integer base and exponent and returns the base raised to that exponent.
  - Example: `pow(3, 4)` returns 81
  - Solve the problem recursively and without using loops.

Example 8

- Write a recursive method `printBinary` that accepts an integer and prints that number's representation in binary (base 2).
  - Example: `printBinary(7)` prints 111
  - Example: `printBinary(12)` prints 1100
  - Example: `printBinary(42)` prints 101010

place	10	1
value	4	2

32	16	8	4	2	1
1	0	1	0	1	0

  - Write the method recursively and without using any loops.

```
// Prints the given integer's binary representation.
// Precondition: n >= 0
public static void printBinary(int n) {
    if (n = = 0) {
        // base case;
        System.out.println();
    } else {
        // recursive case; break number apart
        printBinary(n / 2);
        System.out.println(n % 2);
    }
}
```

Example 9

- Write a recursive method `isPalindrome` accepts a `String` and returns `true` if it reads the same forwards as backwards.

```
– isPalindrome("madam")                      → true
– isPalindrome("racecar")                    → true
– isPalindrome("step on no pets")            → true
– isPalindrome("able was I ere I saw elba")  → true
– isPalindrome("Java")                       → false
– isPalindrome("rotater")                    → false
– isPalindrome("byebye")                     → false
– isPalindrome("notion")                     → false
```

```java
// Returns true if the given string reads the same
// forwards as backwards.
// Trivially true for empty or 1-letter strings.
public static boolean isPalindrome(String s) {
    if (s.length() < 2) {
        return true;    // base case
    } else {
        return s.charAt(0) == s.charAt(s.length() - 1)
            && isPalindrome(s.substring(1, s.length() - 1));
    }
}
```

You may ask: Since every recursive algorithm can be written iteratively, when should one use recursion? Bear in mind that recursive algorithms can incur extra run time and memory. Their major plus is elegance and simplicity of code.

General Rules for Recursion

1. Avoid recursion for algorithms that involve large local arrays—too many recursive calls can cause memory overflow.
2. Use recursion when it significantly simplifies code.
3. Avoid recursion for simple iterative methods like factorial, Fibonacci, and the linear search on the next page.
4. Recursion is especially useful for
   - Branching processes like traversing trees or directories.
   - Divide-and-conquer algorithms like mergesort and binary search.

A certain type of problem crops up occasionally on the AP exam: using recursion to traverse a two-dimensional array. The problem comes in several different guises, for example,

1. A game board from which you must remove pieces.
2. A maze with walls and paths from which you must try to escape.
3. White "containers" enclosed by black "walls" into which you must "pour paint."

In each case, you will be given a starting position (row, col) and instructions on what to do. The recursive solution typically involves these steps:

Check that the starting position is not out of range:
If (starting position satisfies some requirement)
Perform some action to solve problem
RecursiveCall(row + 1, col)
RecursiveCall(row − 1, col)
RecursiveCall(row, col + 1)
RecursiveCall(row, col − 1)