

## Recursion

### I. Thinking Recursively

In this chapter we look at another method of repetition, recursion. A recursive computation solves a problem by calling itself to solve a smaller piece of the problem.

There are three basic rules for developing recursive algorithms.

- Know how to take one step.
- Break each problem down into one step plus a smaller problem.
- Know how and when to stop.

Here is one method for converting and printing the decimal number to a binary number (base 2):

ex.1) 2:

$$2/2 = 1 \text{ remainder } 0$$

$$1/2 = 0 \text{ remainder } 1$$

2 as binary number is: 10

ex.2) 100:

$$100/2 = 50 \text{ remainder } 0$$

$$50/2 = 25 \text{ remainder } 0$$

$$25/2 = 12 \text{ remainder } 1$$

$$12/2 = 6 \text{ remainder } 0$$

$$6/2 = 3 \text{ remainder } 0$$

$$3/2 = 1 \text{ remainder } 1$$

$$1/2 = 0 \text{ remainder } 1$$

100 as binary number is: 1100100

Now, let's look at this same problem recursively.

- First, know how to take one step: A step seems to be "Divide by 2 and note the quotient and the remainder."
- Second, break each problem down into one step plus a smaller problem: The quotient is smaller than the original number. The above process can be applied to the quotient. Therefore you have a smaller problem.
- Know how and when to stop: Stop when the quotient is 0. Actually, the recursion stops when the quotient  $\leq 0$ . Print out the remainders when you reach a quotient of 0, where the last remainder must be printed first.

## AP Programming - Chapter 17 Lecture

Now let's look at code:

```
public static void convertToBin(int decimalNum)
{
    int quotient = decimalNum / 2;
    int remainder = decimalNum % 2;
    if (quotient > 0)
    {
        convertToBin(quotient); // smaller problem
    }
    System.out.print(remainder); // after all recursive calls have been
                                // made last remainder printed first
}
```

Divide the decimal number by 2 and remember the remainder. Keep repeating this process with a smaller decimal number (quotient) until the quotient  $\leq 0$ . Then, print out the remainder for each call. As long as the quotient is  $> 0$ , another recursive call is made (no printing occurs yet). When the quotient is  $\leq 0$ , the if statement is not executed and the algorithm continues by executing the System.out.print statement. This statement is executed for each recursive call that was made beginning with the last. The "stack" of recursive calls is "unstacked." Thus, the remainders will be printed in the "correct" order. That is, the result on the screen will be the binary representation of the decimal number we started with.

## AP Programming - Chapter 17 Lecture

Now let's do a simulation of the code using 100 as decimalNumber:

Output:

decimalNumber = \_\_\_\_\_

quotient = \_\_\_\_\_

remainder = \_\_\_\_\_

-----

decimalNumber = \_\_\_\_\_

quotient = \_\_\_\_\_

remainder = \_\_\_\_\_

-----

decimalNumber = \_\_\_\_\_

quotient = \_\_\_\_\_

remainder = \_\_\_\_\_

-----

decimalNumber = \_\_\_\_\_

quotient = \_\_\_\_\_

remainder = \_\_\_\_\_

-----

decimalNumber = \_\_\_\_\_

quotient = \_\_\_\_\_

remainder = \_\_\_\_\_

-----

decimalNumber = \_\_\_\_\_

quotient = \_\_\_\_\_

remainder = \_\_\_\_\_

-----

decimalNumber = \_\_\_\_\_

quotient = \_\_\_\_\_

remainder = \_\_\_\_\_

## II. Tracing Through Recursive Methods

The AP Exam may include multiple-choice questions that give a recursive algorithm and then ask questions about that algorithm. You need to be able to analyze the algorithm and hand-simulate (trace through) the algorithm. Consider the recursive method below.

ex.1)

```
public class MysteryMaker
{
    //Constructor
    public MysteryMaker( ) { .. }
    // Precondition: x > 0, y > 0
    public int mystery(int x, int y)
    {
        if (y == 1)
            return x;
        else
            return x * mystery(x, y - 1);
    }
}
```

Suppose the statements below were executed:

```
MysteryMaker magic = new MysteryMaker( );
System.out.println(magic.mystery(2, 5));
```

Tracing through the program requires us to write down the work done for each recursive call. When the end condition is reached, work backwards, substituting values into the recursive calls.

mystery(2, 5) = 2 * mystery(2, 4);	2 * 16 = 32	↑
2 * mystery(2, 3);	2 * 8 = 16	
2 * mystery(2, 2);	2 * 4 = 8	
2 * mystery(2, 1);	2 * 2 = 4	
2	2	

↓

We can conclude the following:

- The statement `System.out.println(mystery(2, 5))` would print 32.
- The number of times the call to `mystery` was made, including the original call, is 5.
- `mystery(x, y)` returns  $x^y$ .

Examining the method in ex.1), we see that our three rules of recursion are satisfied.

- Taking one step is defined in the general case (the else clause).
- $y$  starts with a positive integer value and each recursive call is made with a smaller  $y$  (a smaller problem).
- The end condition (**base case**) occurs when  $y = 1$ .

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. Note: A recursive algorithm can have more than one base case.

ex.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!

Note 1: A method that has no pending statements following the recursive call is an example of **tail recursion**. Method `drawLine( )` above is such a case, but ex.1) `MysteryMaker` is not.

Note 2: The base case in the `drawLine` example is `n == 0`. Notice that each subsequent call, `drawLine(n - 1)`, makes progress toward termination of the method.

If your method has no base case, or if you never reach the base case, you will create **infinite recursion**. This is a catastrophic error that will cause your computer eventually to run out of memory and give you heart-stopping messages like `java.lang.StackOverflowError ...`

ex.3)

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

### III. Recursive Helper Methods and Mutual Recursion

Section 17.3 of your text demonstrates the use of helper methods to simplify recursive solutions. Section 17.4 discusses mutual recursion. Neither of these topics is specifically listed in the AP CS Topic Outline but both are extensions of recursive solutions. You should read through the examples presented in your text.

### IV. A Word About Efficiency

Recursion is not always the most efficient way to solve a problem. The example of finding the  $n^{\text{th}}$  Fibonacci number, which is illustrated in Section 17.5 of your text, certainly demonstrates this!

One of the best known recursive functions, Ackerman's Function, is not very useful, but it is very interesting. For lots of practice with tracing a recursive function, try tracing this algorithm to evaluate `acker(2, 3)` (or any other call with small numbers). This function grows very fast! By tracing Ackerman's function, you can see that the same method call is executed multiple times.

```
y + 1 when x = 0
acker(x - 1, 1) when x != 0, y = 0
acker(x - 1, acker(x, y - 1)) when x != 0, y != 0
```

After you try this by hand, test it out by writing a recursive method to solve the problem. If you try to evaluate this function for large values your program will report a "stack fault". Common Error 17.1 of your text discusses stack faults.