Recursion

Recursion occurs when a function is defined in terms of itself. For example, the mathematical definition of a Fibonacci number is defined as follows:

Fibonacci(0) = 0
Fibonacci(1) = 1
Fibonacci(n) = Fibonacci(n – 1) + Fibonacci(n – 2) for all $n > 1$

Recursion can be implemented in a programming language by creating methods that call themselves. For example, recall the recursive definition of the fibonacci method:

```
public static int fibonacci(int n) {
  if (n == 0) return 0;
  if (n == 1) return 1;
  return fibonacci(n-1) + fibonacci(n-2);
}
```

To write a recursive program, the programmer must specify the base case(s)—in this example, when *n* is 0 and when *n* is 1—and then write the recursive case(s). It is important that you understand this principle of writing a recursive program and not try to follow through the logic of the program as it executes. For simple cases such as the calculation of a fibonacci number, it might be possible to do so, but for this course, trying to reason through a recursive program will be a fool's errand, and it is highly recommended that you learn to write recursive methods by specifying the base case(s) and recursive case(s). The importance of this point will become apparent in the examples presented here.

Many algorithms and data structures that will be covered in this course are by nature recursive, and so recursive solutions will be the easiest, and often the best, way to solve them. However, recursion is not an easy way to reason about a problem. In fact, it can be difficult to understand, and if not properly implemented, impossible to debug. The rest of this section will be devoted to explaining how to properly implement recursion by looking at several interesting problems.

A. Binary Number with No Consecutive Ones

The first problem is to generate all possible combinations of an *n*-bit binary number that contain no consecutive ones. To see what is meant by this, consider a two-bit binary number. The possible valid numbers are 00, 01, and 10. Note that the string 11 is not valid, as there are two consecutive ones in the number. The following table gives the valid solutions for 3-, 4-, and 5-bit numbers.

3 bits	4 bits	5 bits
000	0000	00000
001	0001	00001
010	0010	00010
100	0100	00100

101	0101	00101
	1000	01000
	1001	01001
	1010	01010
		10000
		10001
		10010
		10100
		10101

At first glance, the solution to this problem might seem daunting, but on careful reflection, you'll see that the solution is to simply build each of the *n*-bit numbers recursively. The trick is to realize that if the current bit is a 1, then a 0 can be appended but a 1 cannot. If the current bit is a 0, either a 1 or a 0 can be appended. These are the recursive cases.

Next, when the number has reached the correct number of bits, it must be valid, and it can simply be written to the output. This is the base case.

The recursive and base cases are implemented in the following Java program. Note that the method to produce this result is seven lines long! Students solving this problem often turn in programs that are well into the hundreds of lines. If done correctly, recursion is normally concise; when programs start getting into large amounts of complex code, it is probably a good idea to check to see if you truly understand the problem.

To run this program, type "java PrintNumber *n*", where *n* is the number of bits in the number you wish to generate.

```
public class PrintNumber {
  /**
    nextBit - This method sees if the string is the correct
              length. If it is, the string is valid, so print it
              out. If not add a "0" and continue to build the
              string. Also add a "1" to the string if the current
              bit is not already a "1".

              Note that this method is 7 lines long!
   */
  public static void nextBit(String s, int n) {
    if (s.length() == n) {   // Base case
        System.out.println(s);
        return;
    }
    nextBit(s+"0", n);       //recursive case, always add a "0"
    if (! s.endsWith("1"))   //recursive case, do not allow two "1"
      nextBit(s+"1", n);
  }
  /* End of routine */
  public static void main(String[] args) {
    int size = 0;
    if (args.length != 1) {
      System.out.println("Run as \"java PrintNumber n\"");
      System.exit(1);
    }
    try {
      size = Integer.parseInt(args[0]);
    } catch (Exception e) {
        System.out.println("Input to program must be an integer");
        System.exit(1);
    }
    if (size <= 0) {
      System.out.println("size must be positive");
      System.exit(1);
    }
    nextBit("", size);
  }
}
```

B. Knapsack Problem

This is a slightly more difficult recursive problem than the one in the previous section and is a good illustration of how to solve a recursive problem. The problem is a variation on the classic bin-packing problem. The premise is that a thief has broken into Fort Knox, and all around him are gold bars of varying weights. But the thief has a knapsack that will hold only a limited amount of weight before it will tear. So given all the weights, is there a set of weights which will exactly fill the knapsack without tearing it?

The only way to solve this problem exactly is to try all possible combinations of the weights in the room. This can be accomplished by putting a weight into the bag and then determining whether there is some combination of all the other weights in the room that will exactly fill the bag. If no such combination exists, then the weight is removed and the process is repeated with the next weight.

To implement this solution, there are two recursive cases and three base cases.

The two recursive cases are:

1. There is a solution with this weight.
2. There is no solution with this weight.

The three base cases are:

1. The current weight has exactly filled the bag (success).
2. The current weight has made the bag too heavy (failure).
3. There are no more weights to try.

Using these recursive and base cases, the knapsack problem is solved in the following program. To make it easier for you to concentrate on the recursive knapsack algorithm, this program is implemented with the total weight the bag can hold and the array of weights in the room defined as constants. Other possible solutions can be generated by simply changing the constant for the weight of the bag or the array of weights in the room.

```
public class KnapSack
{

  /**
   *  This method recursively attempts to find a
   *  solution which exactly fills the sack.
   */
  public static boolean fillSack(int weightLeftToFill,
                                 int[] weightsAvailable,
                                 int currentWeight)
  {
    //base case 1 - The bag is full
    if (weightLeftToFill == 0) {
      System.out.println("Solution Found!! Weights are: ");
      return true;
    }

    // base case 2 - There are no weights left to consider
    else if (currentWeight >= weightsAvailable.length)
      return false;

    // base case 3 - The bag is over full
    else if (weightLeftToFill < 0)
      return false;

    //recursive cases.  First assume there is an answer
    //using the currentWeight
    else if (fillSack(
         (weightLeftToFill – weightsAvailable[currentWeight]),
          weightsAvailable, currentWeight+1)) {
      System.out.println(weightsAvailable[currentWeight]);
      return true;
    }
    // Second case.  Since no solution was found in the
    // previous step, there is no solution with the current
    // weight, so look for a solution without it.
    else
      return fillSack(weightLeftToFill,
                      weightsAvailable,
                      currentWeight+1);
  }

  /**
   * The main method
   */
  static public void main(String args[]) {
    // You can change the problem by changing the
    // following two variables.
    int weightBagHolds = 12;
    int[] weightsAvailable = {13, 5, 11, 2, 4, 6, 3, 1};
    // Another solution for testing
    // int[] weightsAvailable = {5, 11, 2, 4, 6, 3};
    // int[] weightsAvailable = {13, 11, 5, 8, 6, 3};

    if (! fillSack(weightBagHolds, weightsAvailable, 0))
      System.out.println("No Solution was found");

  }
}
```

C. Prime Number Sieve

The last recursive problem we'll discuss in this module is called Eratosthenes' Sieve, also known as a prime number sieve. The idea is that a "sieve" of prime numbers is created in which each node is a prime number. Each number in turn is passed through the first element of the sieve. If the number in question is divisible by the prime number in the first element, it will be rejected as not prime. If the number in question is not divisible by the current number, it is passed on to the next element in the sieve. If the number reaches the end of the sieve and is not divisible by the current final element, then it is a prime number and a new node is added to the sieve, making it the new final element in the sieve. The process is illustrated in the following diagram:

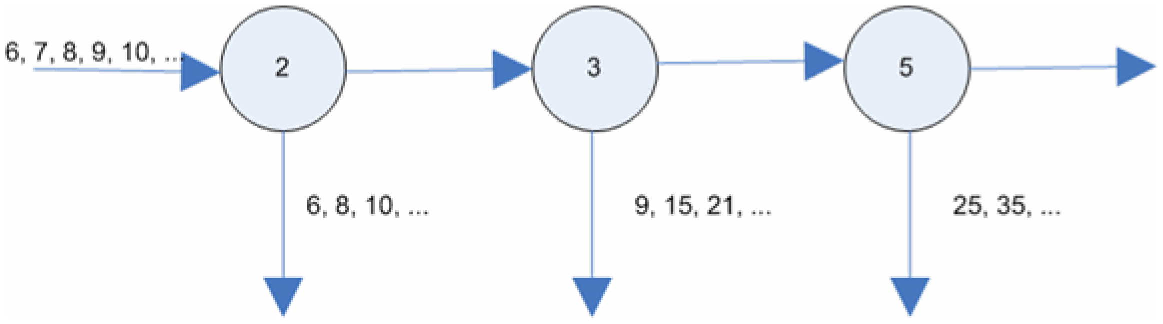

Figure 1.IV.1 – Prime number sieve

This problem is interesting because it behaves somewhat differently than the recursive programs we have studied up to this point, and it is presented to help you expand the way you reason about recursion. In this course, you'll encounter a great many recursive algorithms, and while all will have base and recursive cases, they will not all follow the simple case example of having the base case(s) and recursive case(s) contained in a single method.

In the prime number sieve, there are two base cases and one recursive case. The base cases are:

1. The input number is divisible by the prime number contained in this node, and thus is not a prime number and can be rejected.
2. The input number is not divisible by the prime number contained in this node, but there is no further node in the sieve to pass it to, so a new node must be created for this input number.

The recursive case is:

The input number is not divisible by the prime number contained in this node, so it is passed on to the next node in the sieve.

The code to implement this prime number sieve is presented below. Note that the program is hard coded to calculate and print out the prime numbers between 0 and 100.

```
public class PrimeSieve {

  private static class SieveNode {
    private int myPrime;
    private SieveNode nextSieveNode;

    /**
     *  When a node is created, print out the value
     *  as it is a new prime number
     */
    public SieveNode(int prime) {
      myPrime = prime;
      System.out.println(myPrime);
    }

    /**
     *  check to see if the number is divisible
     *  by this node in the sieve. If not,
     *  pass it to the next node in the sieve.
     *
     *  If this is the last node in the sieve,
     *  the number is prime, and create a new
     *  sieve node.
     */
    public void checkPrime(int number) {
      if ((number % myPrime) == 0)    //base case, not a prime
               return;
      else if (nextSieveNode == null) // base case, no next node
          nextSieveNode = new SieveNode(number);
      else                            // recursive case
          nextSieveNode.checkPrime(number);
    }
  }

  public static void main(String[] args) {
    // Print out 1 (do not put it in sieve)
    System.out.println(1);
    // Initialize the sieve with the prime 2
    SieveNode sieve = new SieveNode(2);
    for (int i = 3; i < 100; i = i + 2) {
      sieve.checkPrime(i);
    }
  }
}
```