COSC–111 Introduction to Computer Science I
Lab 08: Searching
Due: Friday, April 19, 8.59am

1 Introduction and Setup

Find a partner, and play the following game: Player 1 thinks of a number between 0 and 1000. Player 2 tries to guess this number. Each time Player 2 makes a guess, Player 1 responds by saying that the guess was either too low, too high, or correct. This continues until Player 2 has guessed the correct number. Player 2's goal is to accomplish this using as few guesses as possible.

Play this game a few times, switching off roles between Player 1 and Player 2. Try to figure out the best possible strategy for Player 2: what sequence of guesses should Player 2 make to figure out Player 1's number in the fewest possible guesses? Read on once you've come up with a good strategy.

1.1 Binary Search

Suppose Player 1 is thinking of the number `436`. One possible exchange between Player 1 and Player 2 might look like:

```
Player 1: I'm thinking of a number between 0 and 1000.
Player 2: 0?
Player 1: Too low.
Player 2: 1?
Player 1: Too low.
Player 2: 2?
Player 1: Too low.
...many guesses later...
Player 2: 435?
Player 1: Too low.
Player 2: 436?
Player 1: You got it!
```

At every step along the way, Player 2 is able to narrow the range of values to consider: at first, Player 2 only knows that the number is in the range $[0, 1000]$. After the first guess, Player 2 knows that the number is in the range $[1, 1000]$. After the second guess, Player 2 knows that the number is in the range $[2, 1000]$. Each time, the range under consideration shrinks, but only by one number each time. It would be better if Player 2 could eliminate a larger set of numbers with every guess.

Of course, the largest possible set of numbers that can be eliminated with a single guess depends on what number Player 1 has in mind. If Player 1's number is 997, then 995 is a very good first guess for Player 2, because after Player 1 responds "too low", Player 2 knows that the correct number is in the range $[996, 1000]$—only 5 possibilities. But if Player 1's number is 3, then 995 is a very bad first guess for Player 2, because now Player 2 only knows that the answer is in the range $[0, 994]$—which is 995 possibilities. So Player 2's best strategy involves coming up with a guess that eliminates a large number of possibilities no matter what number Player 1 has chosen.

It turns out that the best first guess for Player 2 is 500—right in the middle of the range. If Player 1 responds that 500 is too low, then Player 2 is left with the range $[501, 1000]$—500 possibilities. And if Player 1 responds that 500 is too high, then Player 2 is left with the range $[0, 499]$—again, 500 possibilities. In general, the trick is for Player 2 to always make the guess that will eliminate half of the current possibilities: that is, Player 2 should guess the number that falls exactly in the middle of the current range of possibilities.

This idea is called *binary search*, and your goal in this lab will be to write a recursive binary search method. This method will work a little differently than the low/high number guessing game, because instead of trying to guess a number in a particular range, our goal will be to search for a specific value in a given array that we know is already sorted. Our method will answer the question "Does this array contain value `x`?" where both the (sorted) array and `x` will be input parameters to the method. We've written a method that solves this problem using a `for` loop (without assuming the array is sorted)—this time, you will do it recursively.

1.2 Setup

Begin by making a new directory for this lab, changing into that directory, and copying a java file:

```
$ mkdir lab08
$ cd lab08
$ wget -nv -O Search.java http://bit.ly/111s19L08F
```

You should now have a file called `Search.java`. This file contains several complete methods:

- `print(int[] list, int lo, int hi)`: Prints out the contents of the array `list` in the range `lo` to `hi` (inclusive).
- `initialize(int[] list)`: Fills in the array `list` with random numbers, sorted in increasing order.
- `fillRandom(int[] list)`: Fills in the array `list` with random numbers in no particular order.
- `insertionSort(int[] list)`: Sorts the numbers in `list` in increasing order.

You do not need to change the code in any of these methods.

2 Your Tasks

Work on your own for this part of the lab.

Your job in this lab is to write the `binarySearch(int[] list, int toFind, int lo, int hi)` method. The method should return `true` if the number `toFind` is stored somewhere in `list` in the range `lo` to `hi` (inclusive), and `false` if not. **Please do not change the input parameters to this method.** Your method should be recursive.

3 Submit your work

Submit your modified `Search.java` using either the submission web site or the `cssubmit` command.

4 Totally Optional Challenge

If you finish this week's lab early and want some extra practice with recursion, here are some other problems that you can solve using recursion (they are roughly in order of difficulty). Put your solutions in a new Java file (i.e., not `Search.java`).

- Multiplication. Admittedly recursion is not the most straightforward approach to multiply two numbers, but it can be done. Write a recursive method to compute `x*y`. You should not use the `*` operator anywhere in your solution.
- Euclid's GCD algorithm. The greatest common divisor (GCD) of two numbers x and y is the largest factor that they have in common. For example, the GCD of 24 and 16 is 8, and the GCD of 100 and 72 is 4. There are many ways to compute the GCD of a pair of numbers. Euclid came up with the following algorithm to compute the GCD of x and y (assuming $x > y$):

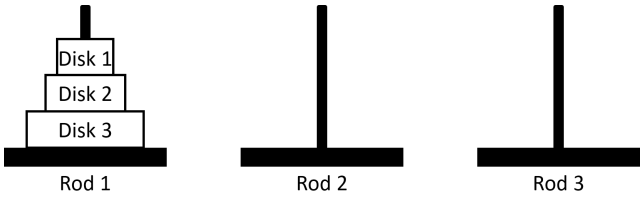
  1. If x is evenly divisible by y, then the answer is y.
  2. Otherwise, let r be the remainder when x is divided by y. The GCD of x and y is the same as the GCD of y and r.

  For example, suppose we want to compute the GCD of $x = 48$ and $y = 18$. Since 48 is not divisible by 18, we let $r = 12$, which is the remainder when 48 is divided by 18. We then compute the GCD of 18 and 12. Since 18 is not divisible by 12, we compute the remainder

when 18 is divided by 12. This is 6. We then compute the GCD of 12 and 6. Since 12 is divisible by 6, our answer is 6. This turns out to be the GCD of our original pair of numbers, 48 and 18.

Write a recursive method to compute the GCD of two numbers `x` and `y` (you can assume that `x` is greater than `y`) using Euclid's algorithm.

- Towers of Hanoi. The Towers of Hanoi puzzle is as follows: we have three vertical rods in a row. Some number of disks (with holes in their centers) are sitting on the first rod, with the smallest disk on top and the largest disk on the bottom and all other disks sorted in between. The other two rods are empty. So things look like this:



The goal is to move all of the disks from the first rod to the last rod, again with the smallest disk on top and the largest disk on the bottom and all other disks sorted in between, using as few moves as possible. However, disks only can be moved according to the following rules:

1. Only one disk can be moved at a time.
2. A move involves taking a single disk from the top of one rod and moving it onto the top of another rod.
3. A larger disk never can be placed on top of a smaller disk.

For a stack of three disks, the solution with the fewest moves is:

```
Move disk 1 from rod 1 to rod 3
Move disk 2 from rod 1 to rod 2
Move disk 1 from rod 3 to rod 2
Move disk 3 from rod 1 to rod 3
Move disk 1 from rod 2 to rod 1
Move disk 2 from rod 2 to rod 3
Move disk 1 from rod 1 to rod 3
```

Write a recursive method to print out the best sequence of moves to solve the Towers of Hanoi puzzle with `n` disks.

You can submit this new code using the submission web site or the `cssubmit` command; choose the `Lab 08 (OPTIONAL PART)` assignment.