

**CS 1160-01 Introduction to Computer Science
and Programming Methods
Programming Assignment 5
Due, Monday, February 28, 2005
Triple Treat (Threat?)**

A Guessing Game (Intro to Binary Search)

Your first program is a simple game program which will ask the user to choose *and keep secret* a positive integer in the range 0 to 1000, inclusive, and then to answer (honestly) a series of no more than *ten* “yes”/“no” questions which the program will pose. At that point the program should be able to tell the user/player *exactly* what the original secret number was. The questions the program will ask will be of the form: “Is the number between XXX and YYY (inclusive)?”

The simple idea is to start out with two variables, say `max` and `min` (or `top` and `bottom`) which start out at the values 1000 and 0, respectively. You then divide the interval into two (almost) equal subintervals, one from `min` to `mid` and the other from `mid + 1` to `max`, where `mid = (min + max)/2` (NOTE: integer division). Your question can then use `min` for “XXX” and `mid` for “YYY”. If the answer is “yes”, move the value of `max` down to `mid`; if it is “no”, move the value of `min` up to `mid + 1`. Then repeat the process on the shorter interval. Each time the interval will be half as big as the previous one. When `min` and `max` “meet” (i.e., have the same value), *that* is the “secret” number.

Ordered Storage of Integers—with a shift

This program should have an *array* of `ints` of size 1000. It should accept integers from the user—in *any* order—and store them in the array so that they are in *increasing* order. This means, for example, that if the name of the array is `data`, then the numbers stored in `data` should satisfy `data[0] <= data[1] <= data[2] <= ...` until you reach the “empty” region of the array. This means that when you print out the array using a simple, standard for-loop, the original integers should be returned to the user in *sorted* order.

In order to accomplish this, you should have a variable, say, `endlist` whose value “tells” you where the “full” part of the array ends. (There are two *styles* for this: 1) Initialize `endlist` to 0 and think of `data[endlist]` as the first *available* location in the array, OR 2) Initialize `endlist` to -1 and think of `data[endlist]` as the last *used* location in the array. Use whichever suits you.)

Now, when your program *reads* a new integer from input, it must determine *where* in the array it should be inserted: If it is larger than all the integers currently stored in the array, then it goes into the free location at the end of the currently used portion of the array. However, the new integer, say `nn` is such that `data[i] <= nn < data[i+1]` for some index `i` in the “used” portion of the array, `nn` properly *belongs* in `data[i+1]`. *However*, that means that you must *shift* all the integers currently stored at index `i+1` “up” (or “to the right”) one location, in order to make room for `nn`. This *shift* is a simple loop—and could be encapsulated into a function—but notice that the shifting must start at the *end* of the array and move “backwards” to avoid overwriting data.

In order to save wear and tear on your fingers—and patience—you may want to have your program accept test data from a *file*. In that way you will only need to type in the scrambled list of integers for test data once.

Your program should print out the sorted list of integers *after* all input has been completed.

Ordered Storage of Integers—shiftlessly

This program has the same task as the last one—to read in a collection of integers and to print them out in sorted order. In this one we can avoid the “shift” operation. However, the price we need to pay is to keep a *second* array of `ints`, say, `next`. In this version, the integers *read* in will be stored in the `data` array, say, `data` in the order in which they are read: `data[0]` gets the first

one, `data[1]` gets the second one, etc. However, the array `next` will contain the information as to where the next integer in *sorted order* can be found in the array: That is, the integer following (in order) the one in `data[0]` will be found in `data[next[0]]`, etc.

This means that you will need a variable, say, `first` to hold the index of the *smallest* integer in the data. Whenever a new integer is read in, the values of `next` might need to be adjusted. That is, when a new integer `nn` is stored, one needs to find an index `i` such that `data[i] <= nn < data[next[i]]`. In this case, `next[i]` should get the value of the index, say `j`, where `nn` has been stored, and `next[j]` should get the (former) value of `next[i]`.

Now, after all the integers have been read in, tracing through the data starting at index `first` and following the `next` indices, should allow one to print out a sorted list of integers.

What to turn in

As usual, you should turn in your source code, which should be in good style and well commented. You should also turn in the results of enough runs of your program to convince one that it works correctly.