CS 132 - Intro to Computer Science II - Spring 2005
Week #2 Lab Exercise and Homework #1

Week #2 Lab Exercise due: Wednesday, January 26th, END of lab
HW #1 due: Wednesday, February 2nd, beginning of lab

WEEK #2 LAB EXERCISE

TEAM exercise:

1. On a sheet of paper with your name on it, give the Big-O notation for the time complexity for an algorithm that performs the following numbers of steps for an input of size N. Write these down, and compare them with at least one other classmate's answers. If they differ, discuss why until you both agree on the answer. Write down the name of all those you conferred with on your paper, then put one of your conferring-team member'sname on the NEXT list on the board. I will meet with each conferring team.

(a) $8N^3 - 9N$ ((8 times (N cubed)) minus (9 times N)

(b) $7\log_2 N + 20$ ((7 times(log-base-2 of N) plus (20))

(c) $7\log_2 N + N$ ((7 times(log-base-2 of N) plus (N))

(d) $N^2 + 1000N + 123,456$ ((N squared) plus (1000 times N) + (123,456))

(e) 538

INDIVIDUAL exercise:

2. Do Homework Problem #1 (below), steps (a) - (g). Note that the part of the point of this problem is to walk you through the function design recipe process discussed in lecture, adding in the concepts of preconditions and postconditions as discussed in Chapter 1 of the course text. You'll also be "building" the "fullest" version (I suspect) of the course opening comment block. Not all functions/files will require this "full" version, as you also will see as you proceed through this assignment, but some will. Finally, problem #1 sets up the beginning of an experiment to help better illustrate the differences between two of the complexity categories discussed in lecture and Chapter 1 of the course text.

When you complete (g), add your name to the Next: list on the board, and wait for me to get to you. I'll then look over your opening comment block at that point, and let you know if you are on the right track. (You may safely proceed with (h) and beyond while you are waiting for your turn.)

If you complete both of the above successfully before lab is over, then you have successfully completed the lab exercise.

HOMEWORK #1:

1. (Hint: many of the steps required here are have an example shown in the code provided as part of Problem #2...)

Consider a function **seqlSearch** that accepts an array of sorted integers (of any size), the array's size, and an integer. **seqlSearch** is to see if that integer is within the passed array --- if it is, then it returns the index where that integer can be found. If it is not within the array, it returns -1. Note its name --- you should perform this search in a sequential manner, checking each element in the array in turn. However, you may stop before reaching the end of the array when it is appropriate to do so. (Consider: In what case(s) is it appropriate to do so? And in what case(s) must you search the entire array?)

Create a file **seqlSearch.cpp**, and begin an opening comment block with:

```
// File: seqlSearch.cpp
// Name: type your name here
// last modified: type date created or last modified here
```

Note: you may "border" this block in whatever style you'd like. You may also make the opening comment block a

single multi-line comment, if you prefer. But if you do so, modify the additions to the comment block specified below accordingly. (And note: there are "basic" templates for a number of opening comment block "types" available from the public course web page, under "132 opening comment block templates". These can be copied-and-pasted for your convenience.)

Now, add to this as follows:

(a) for a passed array of size N, what is the big-O complexity for the **average-case** time complexity for function **seqlSearch**? (Within the function's opening comment block, include:

```
// avg case time complexity:
```

...followed by your answer. (This will not always be part of a function's opening comment block --- it is here as part of our coverage of Big-O notation.)

Consider: if you have N items to potentially search, how many will you look at, on average?

(b) What is the **contract** for **seqlSearch**? Within the function's opening comment block, include:

```
// Contract:
```

...followed by the contract (using the format shown in lecture, and shown in the example code provided with problem #2 below).

(c) Now, you should be ready to write **seqlSearch**'s header (first line of its implementation). It should, of course, follow the opening comment block --- but you should also **copy** that header, followed by a semicolon, in a file **seqlSearch.h**. This header file should have the following header-file opening comment block:

```
// Header file for function seqlSearch
// Name: type your name here
// last modified: type date created or last modified here
```

Note that this, too, is available from "132 opening comment block templates" --- AND this template also includes the skeleton for the required **#ifndef** use you are expected to use for header files this semester.

(d) Write a purpose statement for **seqlSearch**; within the function's opening comment block, include:

```
// Purpose:
```

...followed by this purpose statement. You are required to use your parameter names appropriately within this purpose statement.

(e) What are the **preconditions** for **seqlSearch**? There are certain assumptions about the arguments that must be true for this function to work; what are they? Within the function's opening comment block, include:

```
// preconditions:
```

...followed by the preconditions. (Note that you do not need to include aspects that are guaranteed, in a sense, by C++ syntax --- you know it must be called using 3 arguments of compatible type, for example. But, given that, what *must* be true of these arguments for this function to be able to perform its task?)

Note that, for preconditions where it is reasonably easy to do so, you are expected to use **assert** in your implementation, near the beginning of the function (or method), to ensure that that precondition holds.

(f) What are the **postconditions** for **seqlSearch**? That is, if the preconditions are true when seqlSearch is called, what can you say **will** be true when seqlSearch is finished executing? Within the function's opening comment block, include:

```
// postconditions:
```

...followed by the postconditions.

(g) Write appropriate examples for **seqlSearch**. Part of what I am looking for here is example coverage --- what are the major "cases" that you should include? Within the function's opening comment block, include:

```
// Examples:
```

...followed by your examples, written in the form of conditional equalities:

`seqlSearch(`*explicit example arguments sep'd by commas*`) == `*exp'd return val*

(That is, it looks like a "real" call to this function, except you have a choice in how you express the example array:

you may either say something like "for arr1 = { *specific values* }", and then use arr1 in the examples following, or you can type the example argument array within the example calls in the form you'd use in initializing an array in its declaration: { *specific values* }*)*

Please ASK ME during lab if this is not clear! Sometimes, for Examples sections, we'll have to just describe what should happen for the examples, but when we can, we will express them as logical expressions such as this. Again, there's an example of this in Problem #2's example code, if you'd like to refer to it.

(h) Now, write the body for **seqlSearch**. Be sure to indent any statements within any { } by at least four spaces, use descriptive identifiers, use blank lines for readability, add appropriate comments --- all of the basic style standards you should have learned in your first programming course.

(i) Finally, write **test_seqlSearch.cpp**, a main function whose purpose is to test **seqlSearch.cpp**. For this simple testing program, here is the basic testing-program opening comment block (a template for which is also available at "132 opening comment block templates"):

```
// File: test_seqlSearch.cpp
// Name: type your name here
// last modified: type date created or last modified here
//
// Purpose: tester for function seqlSearch
```

This tester program should set up and run the examples you included in **seqlSearch**'s opening comment block, with one slight twist: set up the arrays and make the calls to **seqlSearch**, but do not print the returned results of those calls to the screen. Instead, print a message to the screen indicating that 1's mean passed tests and 0's mean failed tests, and then print the results of **comparing** each returned result with its expected result. That is, you hope to see results such as the following as a result of running program **test_seqlSearch**:

```
Testing function seqlSearch...
1's mean test passed, 0's mean test failed:
-----------------------------------------
1
1
1
1
```

(HINT: you can PRINT the result of a logical comparison!! That's the most straightforward way to produce the above. Avoid the if-statement-for-every-test-to-print-test-result approach...)

Ask me now if you do not know what I mean here! And, be sure that this main (and all main functions that you write for this course) returns **EXIT_SUCCESS** when it is successfully complete, instead of simply returning 0.

(j) Compile, test, run, debug, repeat as necessary --- when you are satisfied with your code, run:
`test_seqlSearch > prob1_out`

...to create an example output to turn in. Don't submit your code to me yet, however.

2. Now, since we haven't quite gotten to discussing recursion further yet, I'll go easy on you and give you the following recursive version of **binary search**, which I'll call **binSearch**. (This is one implementation of the "phone book search" that we discussed in lecture.)

```
//-----------------------------------------------------
// File: binSearch.cpp
// Name: Sharon Tuttle
// last modified: 1-27-04
//
// avg case time complexity: you'll need to fill this in...
//
```

```
// Contract: binSearch : int[] int int int -> int
//
// Purpose: See if val is contained within ordered array
//          valArr within the range of indices
//          [leftIndex, rightIndex]; return the index
//          where found if so, and -1 if not.
//
// preconditions:
//    *   the elements within valArr are in ascending order;
//    *   there are no duplicate values in valArr;
//    *   leftIndex is in the range [0, size of valArr];
//    *   rightIndex is in the range [0, size of valArr];
//
// postconditions:
//    *   returns -1 if val is not in the range [leftIndex, rightIndex]
//    *   returns i if valArr[i] == val
//
// Examples: if arr1 == {1, 3, 8, 27, 56, 77, 78, 79, 100}:
//         binSearch(arr1, 0, 8, 50) == -1
//         binSearch(arr1, 0, 8, 150) == -1
//         binSearch(arr1, 0, 8, 0) == -1
//         binSearch(arr1, 0, 8, 1) == 0
//         binSearch(arr1, 0, 8, 100) == 8
//         binSearch(arr1, 0, 8, 56) == 4
//-----------------------------------------------------------------------

#include <iostream>
#include <cmath>
#include <cassert>
using namespace std;

int binSearch(int valArr[], int leftIndex, int rightIndex, int val)
{
    int midIndex;   // index of the middle element in this range

    // which preconditions can we reasonably test with assert
    //    statements before beginning?
    assert(leftIndex >= 0);
    assert(rightIndex >= 0);

    // base case #1 --- is there anything to search?
    if (rightIndex < leftIndex)
    {
        // element was NOT found --- return -1, failure
        return -1;
    }

    // base case #2 --- there is only ONE element to search
    else if (rightIndex == leftIndex)
    {
        if (valArr[leftIndex] == val)
        {
            // element FOUND! return its index
            return leftIndex;
        }
        else
        {
            // element NOT in array --- return -1, failure
```

```
            return -1;
        }
    }

    // if get here --- there's still more to search;
    else
    {
        // look at element in MIDDLE of this range;
        midIndex = static_cast<int>(ceil(
             leftIndex +
             (static_cast<double>
                 (rightIndex - leftIndex)/2)
                      ));

        // base case #3 --- element IS in the middle;
        if (valArr[midIndex] == val)
        {
            return midIndex;
        }

        // element is NOT in the middle...

        else
        {
            // recursive case #1: val is LESS THAN
            //    middle value;
            //    continue looking in LEFT of range;
            if (val < valArr[midIndex])
            {
                return binSearch(valArr,
                                 leftIndex, midIndex-1,
                                 val);
            }

            // recursive case #2: val is MORE THAN
            //    middle value;
            //    continue looking in RIGHT of range;
            else
            {
                return binSearch(valArr,
                                 midIndex+1, rightIndex,
                                 val);
            }
        }
    }
}
```

(a) Copy the above into a file **binSearch.cpp**. You can paste it from the posted version on the public course web page or from the electronic version of this handout, if you do not want to type it in --- BUT note that you need to **replace** the italicized text by hand **yourself**, giving the average case time complexity for binSearch on an array of N elements using big-O notation.

(How many of the elements are "thrown out" during each pass? Consider your reading so far on big-O and complexity...)

(b) Create the appropriate header file **binSearch.h**, using the same style of header-file opening comment block used for **seqlSearch.h** in problem #1.

(c) Create the appropriate tester program **test_binSearch.cpp**using the same style of testing-program opening comment block and same approach as used for **test_seqlSearch.cpp** in problem #1. Be sure to set up and run all of the Examples given in **binSearch**'s opening comment block, and of course the desired output should look like this when you run this and print the results of comparing the actual execution results with the expected run results:

```
Testing function binSearch...
1's mean test passed, 0's mean test failed:
-----------------------------------------
1
1
1
1
1
1
```

(d) Compile, test, run, debug, repeat as necessary --- when you are satisfied with your code, run:

```
test_binSearch > prob2_out
```

...to create an example output to turn in. Don't submit this code to me yet, however, either.

3. You now have tested versions of sequential search and binary search for an ordered array of integers. Let's see if we can see, first-hand, some of the practical differences between algorithms with the different time complexities exhibited by these two examples.

We need two things to really see this. First, consider the examples we ran above: only 8 elements! That's not a big enough N to really get the point across. We need a bigger array. Second, how will we "see" the difference in performance?

For the first point, we need to build a really big array --- say, 1000 integers. But, you should be able to use a loop to do this. I'll leave it to you how to fill it, as long as the resulting array is guaranteed to contain at least 1000 increasing, distinct integers that are **not** simply consecutive. (That is, the elements cannot be 0-999 --- there need to be some "gaps", some integers between the integers within the array.)

That addresses the first point. What about the second? We need to modify/"instrument" **seqlSearch** and **binSearch** so that they'll count how many comparisons they make, as a rough measure of the amount of work they are doing.

That is, **modify** your **seqlSearch** from problem #1, adding a counter that starts out at 0, and is incremented when the desired value is compared to an element within the passed array. Right before it returns its result, it should print the number of comparisons to the screen:

`seqlSearch # comparisons:` *value*

(Not elegant, but it will do for our purposes.) Rerun **test_seqlSearch**; you should then see the above before each **1** resulting from one of its tests.

Now, this isn't going to work so well for **binSearch** --- because of its recursive calls, we'd need to actually add a parameter to count the comparisons, and it would get more complicated than I want for this assignment. Fortunately, this behaves well enough (even with 1000 elements) that we can get away with the following kluge: note that each call essentially compares one element (the one in the middle) with the desired value; so, the number of recursive calls made is roughly the number of comparisons. Simply add a statement to the beginning of the function that prints "called binSearch" along with the leftIndex and rightIndex of that call:

`binSearch called with leftIndex:` *val* `and rightIndex:` *val*

This is a simple change, but it does modify the code, so add your name after mine in binSearch's opening comment block, change the last-modified date appropriately, and make this addition. And, rerun your **test_binSearch** and see the result (you'll now see how often binSearch is called before each of the 1's indicating a successful execution --- you'll see a line printed for each recursive call, and how leftIndex and rightIndex converge on the desired element if it is there.)

Now, we are ready for our test. Write a program in file **bigOPlay.cpp** that builds an array of 1000 increasing-but-distinct elements, and then calls BOTH of the modified versions of **seqlSearch** and **binSearch** for EACH of the following scenarios, preceding EACH scenario with a message printed to the screen describing which scenario it is before making the TWO calls to **seqlSearch** and **binSearch** for that scenario:

* an element smaller than any in the array
* an element larger than any in the array
* an element between some elements within the array but **not** actually in the array
* an element "early" in the array
* an element "late" in the array.

(that is, you'll print a scenario description, THEN call seqlSearch and binSearch for that scenario;
THEN print the NEXT scenario description, THEN call seqlSearch and binSearch for THAT scenario,
ETC.)

What kind of opening comment block should **bigOPlay** have? It needs a bit more than our simple testers, but less than our functions, and less than a main that actually accepts user input. This should suffice:

```
// File: bigOPlay.cpp
// Name: type your name here
// last modified: type date created or last modified here
//
// Purpose: write an appropriate purpose statement here. Obviously there are
//          no parameter names to include, though, nor example input to describe...
//
```

Compile, test, run, debug, repeat as necessary --- when you are satisfied with your code, run:

```
bigOPlay > prob3_out
```

...to create an example output to turn in.

You should now be ready to submit the required files:

prob1_out, **prob2_out**, **prob3_out**,
seqlSearch.h, seqlSearch.cpp, test_seqlSearch.cpp (as they are after problem #3)
binSearch.h, binSearch.cpp, **test_binSearch.cpp** (as they are after problems #3)
bigOPlay.cpp

In the cs-server directory where you have "final" versions of all of these files, run the program **~st10/132submit** . It will walk you through the homework submission process, and ask you to type in the name of each file being submitted. Only files submitted using this tool by the deadline will be accepted for credit.

[IF YOU ENCOUNTER ANY PROBLEMS ACCESSING cs-server, LET ME KNOW A.S.A.P. This is even worthy of a brief phone message to 825-7727 (my home phone), so I can contact the lab administrator as promptly as possible. It is another example of the importance of starting assignments early. One more comment in this regard: what if cs-server has a burp early in the week? I would strongly advise beginning your development on paper, or on redwood or sorrel (whose g++ versions are slightly older, so be careful!) or Dev-C++, and then using **sftp** to transfer your work-in-progress to cs-server when you are able.]