CSE 373 -- Spring 1999 -- Assignment #1

(*due Friday, April 9*)

1. **Write a recursive implementation of binary search on an array**

2. **Prove:** $fib_n > (3/2)^n$ for all $n > 4$

3. **Exercise 2.1**

4. **Exercise 2.6 (a)** (1)-(4)

5. **Exercise 2.16**

6. **Consider a function call `fib(n)` to the recursive implementation of `fib()`:**
   (*a*) What is the total number of function calls made? (give a precise expression in terms of n)
   (*b*) Asymptotically speaking, how many calls are made?
   (*c*) How much space is used by the call? (asymptotically)

7. **Write an implementation of `fib()` that requires only O(1) space**

8. **After reading about the binary search algorithm, Hank Hacker writes a version of it called *ternary search*, which works as follows:**

   - *Given an n-element sorted input array and a target element to search for...*
   - *Compare the target value against the array values at positions n/3 and 2n/3.*
   - *If the target value matches either value, return the position at which it was found.*
   - *If the target value is less than both, narrow the search to the first third of the array*
   - *If the target value is greater than both elements, narrow the search to the last third of the array*
   - *Otherwise, narrow the search to the middle third of the array*
   - *Repeat this process until the target is found or we've reduced our search to a subarray of length 0*

   Hank predicts that his search will run faster than a normal binary search because even though he's doing more comparisons at each step, he's closing in on the correct value faster by narrowing his area of search more quickly.

   (*a*) Do you agree that Hank's ternary search will run faster? Why or why not?
   (*b*) What is the asymptotic running time of ternary search?
   (*c*) Do you think Hank's approach is a good one? Why or why not?
   (*d*) Carl Copycat decides to write an "octary search" which follows the same principle as the binary and ternary searches, but chooses 7 values from the array and refines the search to one of the resulting eight subarrays. What do you think of Carl's approach?

9. **Programming Assignment**

Goals: This programming assignment has two goals: (1) to make sure that everyone's programming skills and environment are working before we reach more difficult assignments; and (2) to experimentally measure the impact of primary and secondary performance effects.

Program Description: Write a program that performs integer searches as follows:

- Take as input a problem size *n*, a filename, a target integer value to search for, and an indication of which search algorithm to use. These input values can be supplied on the command line, read from the console, or read from files, whichever you prefer.
- Dynamically allocate an array of *n* integers and initialize it by reading the first *n* integers from the specified file. It will be assumed that the file contains at least *n* integers in sorted order (*i.e.*, you should not write a sorting routine, simply create input files that are pre-sorted).
- Once the array is initialized, use the timing routines that we supply to measure the execution time of the specified search algorithm. You should implement **four** of the following searches (feel free to do more if you like):

  - *iterative binary search* (as given in the book)
  - *recursive binary search* (your implementation from above)
  - *iterative linear search* (walk across the array from low to high until you find the target, or reach an element that's bigger than the target)
  - *recursive linear search* (same as above but recursively)
  - *ternary search* as described by Hank Hacker above
  - *stupid search* -- a completely dumb search in which an outer loop iterates n times; in the *i*th iteration of the loop, we perform a linear search over the first *i* elements of the array. Thus, elements will be searched in the following order:
    0; 0, 1; 0, 1, 2; 0, 1, 2, 3; 0, 1, 2, 3, 4; etc.
    (Obviously, nobody would ever use this, it's just to provide a painfully slow search algorithm for comparison purposes).

- Your program should output the position of the target value in the array if it was found, or an appropriate message if it was not found.

Experimentation: Once you've verified that your search algorithms are working, construct cases that exhibit the worst-case behavior for each of your algorithms (note that the worst case differs for different algorithms). Do this for a variety of increasing problem sizes until you reach one that takes a painfully long time to complete (for me, this would be just a couple of minutes -- by all means don't waste hours on this step...).

Write-up: Write this up as though it was a mini-lab: (1) Describe your methodology: the algorithms you implemented, their expected asymptotic running times, the problem sizes you ran, the worst case input you used, etc. (2) Plot your results on a graph showing search time vs. problem size. This can either be done using graphing software, or simple pencil and graph paper. (3) Make observations about your results. For example: Did the algorithms demonstrate the expected asymptotic behaviors? Did secondary effects result in differences between algorithms whose asymptotic behaviors were the same? If so, were these differences significant?

What we're looking for Again, one of the main goals for this assignment is to get everyone programming and comfortable with things we'll be using throughout the quarter: dynamic memory allocation, input parameters, file I/O, etc. Your grade will primarily be based on the correctness and clarity of your program and the content of your writeup, not the beauty of your graphs or the measured asymptotic behavior of your implementation. If you have any questions about what I expect or if you need help with this process, don't hesitate to contact me.

Timing Routines: We're providing you with a set of timing routines for the cross-product of C/C++ and PC/UNIX. Check the web pages to get copies of these timers. All timers behave like a stopwatch and support *Reset* and *Check* operations: *Reset* sets the timer to 0. *Check* returns the number of seconds since the last call to *Reset* (as a double floating point value) . The C++ versions are implemented as a timer class. The UNIX versions simply support the calls directly. You should be able to figure out how to use the calls by looking through the header files.

General notes on performing timings on computers: Our timing routines report wall clock time: the actual time that passed between the point that you reset the timer and the point that you checked it. Wall clock time can be tricky on most computers, because the CPU may be shared between multiple programs or users running simultaneously. Therefore, when running on a PC, your timings will be most accurate when no other programs are actively using the CPU (to be safe you can shut down as many of them as possible). When running on hilbert, your timings may be influenced by other people who are sharing your CPU via telnet. Here are some tips on getting timings that show the trends we're hoping to get:

- Treat all of the algorithms fairly, by gathering all of your results in the same time period so that they're subject to similar conditions and similar interference with other users and programs.
- Run each experiment a few times in hopes of filtering out any stray numbers. In CS experiments, you can generally use the fastest time from a set of trial runs, since it's indicative of what's achievable with the least amount of interference.
- By all means, do not time any console or file I/O. I/O is painfully slow on all computers, and will tend to dominate the time spent actually doing the search (*i.e.*, make sure to time the search and nothing else).
- Depending on the size of your memory, the speed of your CPU, and the accuracy of your timer, there's a small chance that you will run out of memory before achieving execution times that are significant enough to show asymptotic trends. If this happens to you, contact me for suggestions.

Coding clearly: You should make an effort to write code that is easy for other people to read. In the real world, this is an important skill that is valued higher than being clever or using the least number of lines possible. In this class, it is important so that we can understand your program and grade it fairly. By reading through your code and comments, we should be able to figure out how your code is organized and why it works. If we can't, we reserve the right to deduct points.

Writing clear code involves commenting your code and avoiding statements that are unnecessarily complicated. Comments should be used to explain (a) what code each file contains, (b) what each major function does, (c) what any non-obvious lines of code do. Overly complicated statements are things like:

```
if ((i = myfunc(counter++)) < num_iterations) {
   ...
}
```

that could be easily rewritten in a clearer manner, such as:

```
i = myfunc(counter);
counter++;
if (i < num_iterations) {
  ...
}
```

Turn-in: Before class on Friday, e-mail a copy of your code to `c373@ms.washington.edu`. Please use attachments to include separate files if your mailer supports them (most do). Be sure to include your name and the assignment number in the subject line or towards the top of the mail. When you turn in your written assignment on Friday, include: (a) all of your code, (b) sample inputs and outputs that demonstrate how your program works and that it works correctly, (c) your mini-lab write-up as described above.

10. **Approximately how much time did you spend on this assignment?** (this question will not be graded...)