# CSE 373 -- Autumn 1999 -- Programming Assignment #1

(***due Friday, October 8*** )

**Goals:** This programming assignment has two goals: (1) to make sure that everyone's programming environment and skills are up-to-speed before we reach assignments that are conceptually more difficult; and (2) to experimentally observe primary and secondary performance effects as discussed in lecture.

**Your Mission:** You will write a program that searches for a target value within a pre-sorted input array using a variety of search algorithms that have different primary and secondary performance effects. You will time the worst case running time for each of these search algorithms across a range of problem sizes to demonstrate their asymptotic behavior. We will provide you with a set of timing routines with which to complete this task. Your primary job is to write the search algorithms, gather the timing data, and see if it matches your predictions. Along the way you will have to use file I/O and dynamic memory allocation -- skills that we'll be relying on throughout the quarter.

**The Program:** Your program should work as follows:

- Take as input several values: a problem size *n*, a filename, the target value to search for, and some indication of which search algorithm to use. These input values can be supplied on the command line, read from the console, or read from a file, whichever you prefer.

- Dynamically allocate an array of *n* integers and initialize it by reading the first *n* values from the specified file. You may assume that the file contains at least *n* values already in sorted order (*i.e.*, under no circumstances should you write a sorting routine).

- The program should then reset the timer and run the search algorithm specified by the programmer. Your program should implement **four** of the following search algorithms (you may implement more than four if you want):

    - *iterative binary search* (given in the textbook)
    - *recursive binary search* (same, but recursively)

    - *iterative linear search* (search the array from low to high until you find the target or reach an element greater than the target)
    - *recursive linear search* (same, but recursively)

    - *iterative ternary search* (similar in spirit to binary search except that the target is compared against *two* values in the array at each step, causing the array to be cut into *thirds*)
    - *recursive ternary search* (same, but recursively)

    - *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).

- Once the search has completed, your program should check the timer and print out the elapsed time. It should also print out the position of the target value if it was found or an appropriate message if it was not found. Be sure that you are **only** timing the search algorithm itself and not reading the input parameters, reading the array from disk, or printing the results.

**The Experiment:** Once you've verified that your search algorithms are working, perform runs of your program that exhibit the worst-case behavior for each of your algorithms (note that different algorithms *may* have different worst-case inputs). Do this for a variety of increasing problem sizes until you reach one that either takes a painfully long time to complete (say, a minute) or causes you to run out of memory (your call to `new` returns `NULL`).

**Experimental Write-up:** Write this up as though it was an informal lab:

> *Setup:* Give the details of your experimental setup: List the algorithms you implemented, their expected asymptotic running times, the problem sizes you ran, the worst case input you used, and anything else relevent to your approach.
>
> *Results:* Plot your results on a graph showing time vs. problem size. This does not need to be done in any fancy way -- graph paper and pencil will suffice if you're careful about it.
>
> *Observations:* Did the algorithms demonstrate the expected asymptotic behaviors? Did secondary effects cause differences between algorithms whose asymptotic behaviors were the same? 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, or the number of pages in your writeup. In most respects, a neat, concise report is much more impressive than a huge sprawling one. If you have any questions about what we expect or if you need help with this process, don't hesitate to ask us.

**Timing Routines:** We're providing you with a set of timing routines for 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 C 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). Here are some general guidelines for gathering experimental timings on computers:

- 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).

**Tip:** You only need a single data file, and it can be something as simple as the integers from 1 to the largest problem size you plan on using. Creating such a file should be about a 12-line C++ program. You can then use this data file for any problem size since your program can just read the first *n* values and ignore the rest.

**Turn-in:** Follow the general Programming Assignment Guidelines for turning in your code. Please flag the following sections of your code with a marker:

1. Where you get the input parameters
2. Where you dynamically allocate the array
3. Where you read the input array from the file
4. Where you reset the timer
5. Where you check the timer
6. Your various search routines (mark each one "6A", "6B", "6C", etc.)