Yet Another Analysis of Algorithms Laboratory

Ed C. Epp
Mathematics and Computer Science
The University of Portland
5000 North Willamette Boulevard
Portland, OR 97203-5798
(503) 283-7163
epp@uofport - bitnet

Abstract

Laboratory assignments can reinforce material given in lecture by making it come alive in practice. A prime example is the time complexity of algorithms. However, laboratories that confirm the time complexity of algorithms can be tedious exercises that do not really challenge a student to think about the algorithms they are testing. By giving students a set of unknown executables, the laboratory becomes more of a detective problem. Students are required to apply what they know about the sort algorithms so that they can identify them.

Preface

What is presented here is a piece of experience. As with all experiences, it is anecdotal and should be read with appropriate skepticism. There is no attempt in this paper to justify the value of laboratories. Many computer departments are instituting them based on a "gut" feeling for what students need and how laboratories can fulfill that need. Computer science laboratories have become one of the fads of the 90s[1]. This action involves substantial risk. There is no conclusive evidence that laboratories are accomplishing the needs they are set out to satisfy. Given that there is currently no accepted criteria for measuring the value of computer laboratories, anecdotal accounts are all we currently have to evaluate them.

A problem

Several dozen laboratory exercises have been introduced in beginning courses at the University of Portland (UP). These laboratory exercises have focused on five areas: fundamental concepts, software development, natural language writing, computer languages, and basic software tools. Empirically analyzing the complexity of algorithms is one laboratory designed to reinforce fundamental concepts. It exemplifies what laboratories are supposed to accomplish, i.e. tying fundamental concepts to "hands-on" experience.

In a "birds of a feather" session at SIGCSE's technical symposium, it was not surprising to find that many people had written laboratories demonstrating the complexity of sorting algorithms. However, it has been discouraging to see the poor response which UP students have shown to elementary laboratories that evaluate the time complexity of algorithms. The laboratory have been perceived as worthless, tedious, and boring.

Unlike McCracken's laboratories[2] designed for an algorithms course, our laboratory was designed for beginning students with less emphasis on rigorous analysis and more emphasis on building intuition. A problem with our analysis laboratory was that there is no discovery in it. A student only records, plots, reports, and forgets, with little need to really think.

Empirical Studies of the Complexity of Sorting

Computer Science I
CS261
December 4, 1991

© Edward C. Epp

Goals

Learn how to measure the time complexity of algorithms and become familiar with the time complexity of the insertion, selection, and quick sort.

Assignment

I. Copy the directory "`~cs261/labs/sorts`" into your home directory.

Run "`link`" to create the appropriate links to the data files and executables.

II. There are three sort programs: sortA, sortB, and sortC. Your task is to determine which one is an insertion, selection, and quick sort. You will also find three data files, all of which contain 10,000 integers. One of the data files is in random order, a second in order, and a third in reverse order. Run the sort programs against these data files to determine how long it takes to sort lists of different length. Then, by graphing time versus number of elements, you should be able to determine each algorithm's order of complexity.

For example, to test the "`sortB`" sort program on a list of 500 random integers, type the following command.

```
sortB 500 < random.dat
```

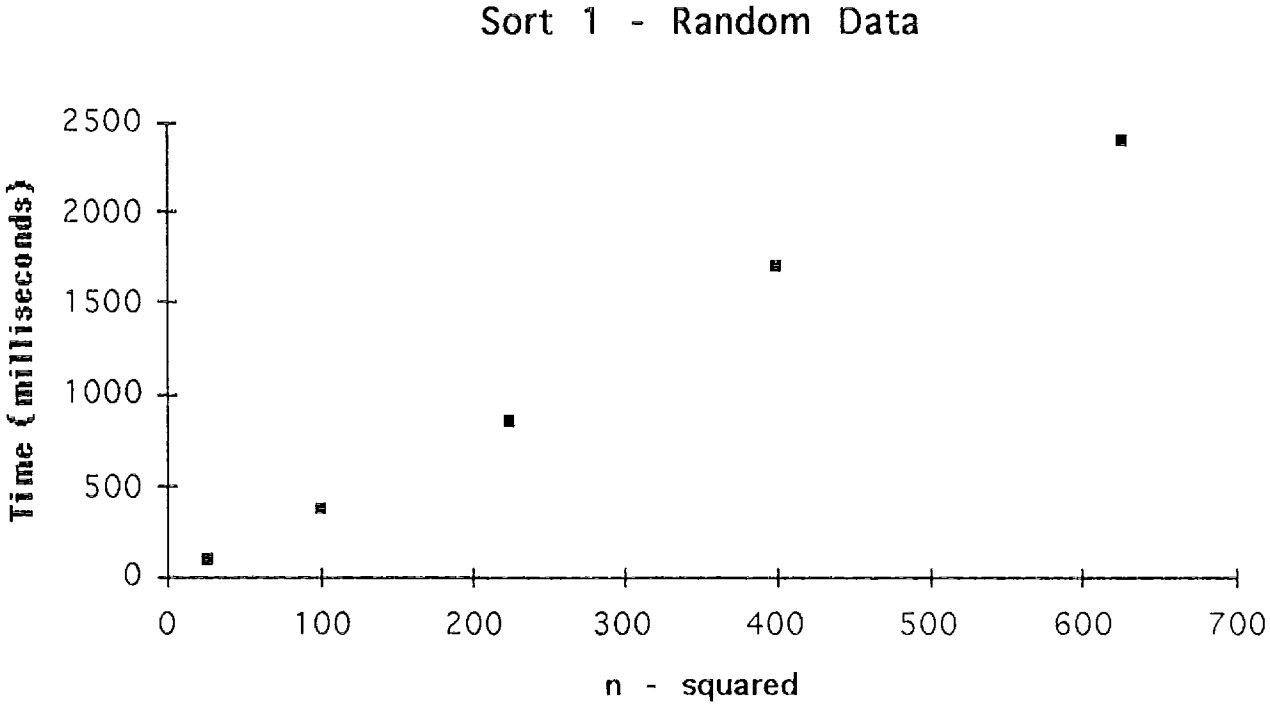
The number of microseconds of cpu time used will be displayed.

III. Build tables of runs as follows (the size of n may be different but should not exceed 10,000.)

insertion sort	on random number
n - items sorted	time in milliseconds
400	
600	
800	
900	
1000	

IV. Neatly graph each table using a full sheet of graph paper for each graph. By varying the order of n, the time complexity of the algorithm can be determined. For example, if you want to show that an algorithm is $O(n^2)$, you will get a straight line when graphing the time against the square of n. Given the data below, you should create the following graph. Since the graph is a straight line when graphed against n^2 the algorithm is $O(n^2)$.

n	n^2	time
5	25	95
10	100	380
15	225	850
20	400	1700
25	625	2400



V. For each sort (sortA, sortB, and sortC) write down whether it is an insertion, selection, or quick sort. Write down the clues (e.g., it is O(n) for ordered lists). The more good clues you give, the higher your grade will be. Turn in the graphs which determined the order of complexity of each algorithm.

A solution

One approach for adding some fun and a little thinking into the laboratory is to give the sort algorithms to the students, as one would give a chemical unknown sample. The student's role is to identify each mystery sort based on its complexity signature. This twist to the laboratory forces the student to distinguish between the selection sort and insertion sort (based on how they behave in the best case) and between the quick sort and heap sort (based on how they behave in the worst case). An example of this approach to an analysis laboratory is given at the end of this paper.

Mechanics

Each student can be given a different set of algorithms. As a result, students will have to rely on their own detective work to discover which algorithms they received.

Test data selection needs to be discussed. Students need to understand that the size of the unsorted list must be large enough to create run-times that are substantially longer than the timer granularity. The granularity may be surprisingly large, e.g., the "clock" function on a DecStation 3100 under Ultrix returns time in microseconds with a resolution of 16.7 milliseconds. Some students may not question the fact that all their runs are exactly 16.7 milliseconds long. In addition, the spacing between data points is important. Student intuition may not be well developed here, e.g., they tend to select evenly spaced data points or a points spacing based on squares (e.g. 100, 400, 900, and 1600 items in a list). Students need help developing a criteria for selecting the interval between sample points.

The quick sort algorithm is written so that the pivot point is the first element in the unsorted list. This makes choosing worst case data easier. Students will be surprised when the quick sort exhausts memory in the worst case for large data sets. This brings home the fact that each recursive call to the quick sort allocates an activation record.

Conclusion

Many laboratory exercises have been written in which students mechanically fill in the blanks and then forget what they have done. This little twist to an analysis laboratory represents a strategy that can require some analytical skills. The only proof of its value is that students asked appropriate questions during the laboratory and did not complain about the analysis laboratory this year.

---

1 **The Papers of the Twenty-Third SIGCSE Technical Symposium on Computer Science Education**, SIGCSE Bulletin, vol. 24, no. 1 (March 1992).

2 Daniel D. McCracken, *Three "Lab Assignments" for an Algorithms Course*, **SIGCSE Bulletin**, vol. 21, no. 2, (June 1989), pp 61-64.

****
WRITING IN CS1-- continued from page 10

Hartman, J., & White, C.M. (1990). "Real World" Skills vs. "School Taught" Skills for the Undergraduate Computer Major. SIGCSE Bulletin, 22(1), 216-218.

Jackowitz, P.M., Plishka, R.M., & Sidbury, J.R. (1990). Teach Writing and Research Skills in the Computer Science Curriculum. SIGCSE BULLETIN, (22)1, 212-215.

Pesante, L. H. Integrating Writing into Computer Science Courses. (1991). SIGCSE BULLETIN, (23)1, 205-209.

Soloway, Elliot. (1986). Learning to Program = Learning to Construct Mechanisms and Explanations. Communications of the ACM, 29(9), 850-858.