CMSC 251: Algorithms[1]
Spring 1998
Dave Mount

Lecture 1: Course Introduction

(Tuesday, Jan 27, 1998)

Read: Course syllabus and Chapter 1 in CLR (Cormen, Leiserson, and Rivest).

What is algorithm design? Our text defines an *algorithm* to be any well-defined computational procedure that takes some values as *input* and produces some values as *output*. Like a cooking recipe, an algorithm provides a step-by-step method for solving a computational problem.

A good understanding of algorithms is essential for a good understanding of the most basic element of computer science: *programming*. Unlike a program, an algorithm is a mathematical entity, which is independent of a specific programming language, machine, or compiler. Thus, in some sense, algorithm design is all about the mathematical theory behind the design of good programs.

Why study algorithm design? There are many facets to good program design. Good algorithm design is one of them (and an important one). To be really complete algorithm designer, it is important to be aware of programming and machine issues as well. In any important programming project there are two major types of issues, *macro issues* and *micro issues*.

Macro issues involve elements such as how does one coordinate the efforts of many programmers working on a single piece of software, and how does one establish that a complex programming system satisfies its various requirements. These macro issues are the primary subject of courses on software engineering.

A great deal of the programming effort on most complex software systems consists of elements whose programming is fairly mundane (input and output, data conversion, error checking, report generation). However, there is often a small critical portion of the software, which may involve only tens to hundreds of lines of code, but where the great majority of computational time is spent. (Or as the old adage goes: 80% of the execution time takes place in 20% of the code.) The micro issues in programming involve how best to deal with these small critical sections.

It may be very important for the success of the overall project that these sections of code be written in the most efficient manner possible. An unfortunately common approach to this problem is to first design an inefficient algorithm and data structure to solve the problem, and then take this poor design and attempt to fine-tune its performance by applying clever coding tricks or by implementing it on the most expensive and fastest machines around to boost performance as much as possible. The problem is that if the underlying design is bad, then often no amount of fine-tuning is going to make a substantial difference.

As an example, I know of a programmer who was working at Boeing on their virtual reality system for the 777 project. The system was running unacceptably slowly in spite of the efforts of a large team of programmers and the biggest supercomputer available. A new programmer was hired to the team, and his first question was on the basic algorithms and data structures used by the system. It turns out that the system was based on rendering hundreds of millions of polygonal elements, most of which were

invisible at any point in time. Recognizing this source of inefficiency, he redesigned the algorithms and data structures, recoded the inner loops, so that the algorithm concentrated its efforts on eliminating many invisible elements, and just drawing the few thousand visible elements. In a matter of two weeks he had a system that ran faster on his office workstation, than the one running on the supercomputer.

This may seem like a simple insight, but it is remarkable how many times the clever efforts of a single clear-sighted person can eclipse the efforts of larger groups, who are not paying attention to the basic principle that we will stress in this course:

Before you implement, first be sure you have a good design.

This course is all about how to design good algorithms. Because the lesson cannot be taught in just one course, there are a number of companion courses that are important as well. CMSC 420 deals with how to design good data structures. This is not really an independent issue, because most of the fastest algorithms are fast because they use fast data structures, and vice versa. CMSC 451 is the advanced version of this course, which teaches other advanced elements of algorithm design. In fact, many of the courses in the computer science department deal with efficient algorithms and data structures, but just as they apply to various applications: compilers, operating systems, databases, artificial intelligence, computer graphics and vision, etc. Thus, a good understanding of algorithm design is a central element to a good understanding of computer science and good programming.

Implementation Issues: One of the elements that we will focus on in this course is to try to study algorithms as pure mathematical objects, and so ignore issues such as programming language, machine, and operating system. This has the advantage of clearing away the messy details that affect implementation. But these details may be very important.

For example, an important fact of current processor technology is that of *locality of reference*. Frequently accessed data can be stored in registers or cache memory. Our mathematical analyses will usually ignore these issues. But a good algorithm designer can work within the realm of mathematics, but still keep an open eye to implementation issues down the line that will be important for final implementation. For example, we will study three fast sorting algorithms this semester, heap-sort, merge-sort, and quick-sort. From our mathematical analysis, all have equal running times. However, among the three (barring any extra considerations) quicksort is the fastest on virtually all modern machines. Why? It is the best from the perspective of locality of reference. However, the difference is typically small (perhaps 10–20% difference in running time).

Thus this course is not the last word in good program design, and in fact it is perhaps more accurately just the first word in good program design. The overall strategy that I would suggest to any programming would be to first come up with a few good designs from a mathematical and algorithmic perspective. Next prune this selection by consideration of practical matters (like locality of reference). Finally prototype (that is, do test implementations) a few of the best designs and run them on data sets that will arise in your application for the final fine-tuning. Also, be sure to use whatever development tools that you have, such as profilers (programs which pin-point the sections of the code that are responsible for most of the running time).

Course in Review: This course will consist of three major sections. The first is on the mathematical tools necessary for the analysis of algorithms. This will focus on asymptotics, summations, recurrences. The second element will deal with one particularly important algorithmic problem: sorting a list of numbers. We will show a number of different strategies for sorting, and use this problem as a case-study in different techniques for designing and analyzing algorithms. The final third of the course will deal with a collection of various algorithmic problems and solution techniques. Finally we will close this last third with a very brief introduction to the theory of NP-completeness. NP-complete problem are those for which no efficient algorithms are known, but no one knows for sure whether efficient solutions might exist.