COMP 200: Elements of Computer Science
Fall 2004
Lecture 20: October 14, 2004

Introduction to Algorithmic Complexity

On the Board

Homework due today
Reading — Chapters 5 & 6

Important Properties of Algorithms

In Chapter 4, we saw a number of approaches to algorithm design — graph traversals, divide and conquer, greedy, dynamic planning. Speaking in broad terms, two properties of an algorithm should be critical concerns (in this order)

- Correctness — The algorithm should solve the problem and produce a correct answer. Correctness is the focus of Chapter 5.

  For some problems, such as phonebook lookup, the correctness criterion is simple and well defined: "Joe" is in the phonebook or he is not. Sorting (as in bubblesort, mergesort, and insertion sort) is equally well defined. For other problems, the criterion may not be well defined: navigating from Rice to the Galleria, the greedy algorithm will make locally optimal decisions that may lead to globally suboptimal solutions. Is the suboptimal solution wrong? It leads to the Galleria. It is faster than some of the paths. The greedy algorithm *approximates* the minimal-cost path with a lower cost algorithm.

  With an algorithm that produces approximate solutions, it is critical that the designer and user both understand what results the algorithm produces—in particular, whether the solution is exact or approximate. (In some cases, we can bound the distance between the approximation and the optimal solution!)

- Efficiency — How quickly can we solve the problem? Efficiency is the focus of Chapter 6.

  Going back to the first lecture, the parallel sort by first name was faster (in clock time) than the insertion sort. To achieve that speed, it used all of your brains in parallel. (Looking at total resource utilization, it may have required more resources than the insertion sort.)

Both the bubblesort algorithm and the mergesort algorithm that we derived in class perform use a number of comparisons that is roughly equivalent to the square of the number of elements being sorted. The mergesort uses about half as many comparisons as bubblesort, but both are still proportional to N^2 for a set of N names.

A good queue implementation (remember the homework that is due today) has the property that ***Add***, ***Remove***, and ***Empty*** all perform a constant amount of work each time that they are used.

Efficiency: What's the Big Deal?

Computer scientists are always concerned about the efficiency of algorithms. Why? Computers have finite resources—including memory, disk, screen resolution, and computational capacity. (What's the difference between a 1GHZ Pentium and a 2.8GHZ Pentium? Why would you want one over the other? Same question about a 20GB iPod versus a 40GB iPod.) In choosing between two algorithms, you might well consider their relative efficiency.

- How does the cost of the algorithm (number of comparisons, total number of operations) increase as the size of the input grows?
- How much memory does the algorithm require? And how does the memory requirement grow with input size?

These considerations have practical impact. The antilock brakes in your car are run by software. If the algorithm to detect a skid is too slow, you lose control of the car. If the algorithm to find the cheapest flight between Houston and Boston has a complexity that rises too quickly with the number of available flights, Travelocity goes out of business. If the algorithm that digitizes your voice in a cellphone (or that maintains the phone's connection with the network) uses too many operations, your battery life suffers.

How efficient should an algorithm be?

- Finding the largest number in a list of numbers, you would reasonably expect that we must look at each number. An algorithm that uses roughly N comparisons is about as good as we can do — common sense. In fact, $N-1$ comparisons is a lower bound for this problem.
- Finding the minimal length path from Rice to the Galleria, the approximate greedy algorithm will make a choice at each intersection. The number of comparisons (and decisions) should not be more than the number of intersections — in the worst case. In the expected case, you hope to avoid visiting every intersection (a complete tour).

Notice that the number of paths grows much faster than the number of intersections. The approximate greedy algorithm doesn't look at every path; it looks at every extension to the current set of paths. It examines many fewer paths than an exhaustive search would — defies common sense, but might be part of the approximation.

- The dynamic planning algorithm found the minimal-length path while doing a simple calculation at each intersection. Again, it takes much less time than the exhaustive algorithm would require. In effect, it discards paths that cannot lead to a minimal-length path.

Talking about Efficiency

Computer Scientists talk about algorithmic efficiency using "big-Oh" notation. The big-O complexity of an algorithm tells us, immediately, how the cost of the algorithm grows with the size of its input data. If the number of operations x that the algorithm requires can be bounded from above by

$$x < c \cdot f(N)$$

where $f(N)$ is a function that depends on the size of the input set and c is a constant, then we say that the algorithm takes $\mathbf{O}(f(N))$ time. For our max algorithm, $f(N)$ is simply N, and the max algorithm takes $\mathbf{O}(N)$ time.

When we use big-O notation, we assume that f is the smallest function that satisfies the inequality $x < c \cdot f(N)$. For the max algorithm, it is true that N is an upper bound. So is N^2, N^3, or even $N!$, but we describe the algorithm with the tightest upper bound that we can prove. Thus, we refer to max as a $\mathbf{O}(N)$ algorithm — sometimes called a *linear-time algorithm* because the running time varies in direct linear proportion to the size of the input.

Consider bubblesort. It ran over the input list once for each element. Each pass over the input list compared $N-1$ list elements. It made $N-1$ passes over the list. $(N-1) \cdot (N-1)$ is $\mathbf{O}(N^2)$. The smarter version that does half the work — starting the inner loop one element higher in the list on each iteration — does half as much work — $(N-1) + (N-2) + (N-3) + \ldots + 2$. Those numbers add up to $(N \cdot (N-1) / 2) - 1$, which is still $\mathbf{O}(N^2)$. The smarter version makes half as many comparisons as the naïve version; however, that difference simply changes the constant c.

Big-O notation describes the worst case behavior of the algorithm. That differs from the expected case behavior or the best case behavior. (Recall the greedy approximate algorithm for Rice-Galleria navigation. With the best case traffic patterns, the greedy algorithm might visit many fewer intersections than in the worst case—and incur many fewer decisions and

operations. That notwithstanding, its worst case behavior is **O**(number of intersections).

If the function, *f(N)*, in the inequality can be expressed as a polynomial, we say that the algorithm has polynomial complexity. In general, problems that have polynomial complexities are considered *tractable problems*. Many important problems have this property — in fact, most of the problems that we solve with computers are tractable. *Low-order polynomial algorithms* are considered to have efficient solutions. Multiplying two matrices — a common subtask in many scientific applications — takes $\mathbf{O}(N^3)$ time.