Lecture 1

Introduction to Algorithms

1.1 Overview

The purpose of this lecture is to give a brief overview of the topic of Algorithms and the kind of thinking it involves: why we focus on the subjects that we do, and why we emphasize proving guarantees. We also go through an example of a problem that is easy to relate to (multiplying two numbers) in which the straightforward approach is surprisingly not the fastest one. This example leads naturally into the study of recurrences, which is the topic of the next lecture, and provides a forward pointer to topics such as the FFT later on in the course.

Material in this lecture:

- Administrivia (see handouts)
- What is the study of Algorithms all about?
- Why do we care about specifications and proving guarantees?
- The Karatsuba multiplication algorithm.
- Strassen's matrix multiplication algorithm.

1.2 Introduction

This course is about the design and analysis of algorithms — how to design correct, efficient algorithms, and how to think clearly about analyzing correctness and running time.

What is an algorithm? At its most basic, an algorithm is a method for solving a computational problem. Along with an algorithm comes a specification that says what the algorithm's guarantees are. For example, we might be able to say that our algorithm indeed correctly solves the problem in question and runs in time at most $f(n)$ on any input of size n. This course is about the whole package: the design of efficient algorithms, *and* proving that they meet desired specifications. For each of these parts, we will examine important techniques that have been developed, and with practice we will build up our ability to think clearly about the key issues that arise.

The main goal of this course is to provide the intellectual tools for designing and analyzing your own algorithms for problems you need to solve in the future. Some tools we will discuss are Dynamic Programming, Divide-and-Conquer, Data Structure design principles, Randomization, Network Flows, Linear Programming, and the Fast Fourier Transform. Some analytical tools we will discuss and use are Recurrences, Probabilistic Analysis, Amortized Analysis, and Potential Functions.

There is also a dual to algorithm design: Complexity Theory. Complexity Theory looks at the intrinsic difficulty of computational problems — what kinds of specifications can we expect *not* to be able to achieve? In this course, we will delve a bit into complexity theory, focusing on the somewhat surprising notion of NP-completeness. We will (may) also spend some time on cryptography. Cryptography is interesting from the point of view of algorithm design because it uses a problem that's assumed to be intrinsically hard to solve in order to construct an algorithm (e.g., an encryption method) whose security rests on the difficulty of solving that hard problem.

1.3 On guarantees and specifications

One focus of this course is on proving correctness and running-time guarantees for algorithms. Why is having such a guarantee useful? Suppose we are talking about the problem of sorting a list of n numbers. It is pretty clear why we at least want to know that our algorithm is correct, so we don't have to worry about whether it has given us the right answer all the time. But, why analyze running time? Why not just code up our algorithm and test it on 100 random inputs and see what happens? Here are a few reasons that motivate our concern with this kind of analysis — you can probably think of more reasons too:

Composability. A guarantee on running time gives a "clean interface". It means that we can use the algorithm as a subroutine in some other algorithm, without needing to worry whether the kinds of inputs on which it is being used now necessarily match the kinds of inputs on which it was originally tested.

Scaling. The types of guarantees we will examine will tell us how the running time scales with the size of the problem instance. This is useful to know for a variety of reasons. For instance, it tells us roughly how large a problem size we can reasonably expect to handle given some amount of resources.

Designing better algorithms. Analyzing the asymptotic running time of algorithms is a useful way of thinking about algorithms that often leads to nonobvious improvements.

Understanding. An analysis can tell us what parts of an algorithm are crucial for what kinds of inputs, and why. If we later get a different but related task, we can often use our analysis to quickly tell us if a small modification to our existing algorithm can be expected to give similar performance to the new problem.

Complexity-theoretic motivation. In Complexity Theory, we want to know: "how hard is fundamental problem X really?" For instance, we might know that no algorithm can possibly run in time $o(n \log n)$ (growing more slowly than $n \log n$ in the limit) and we have an algorithm that runs in time $O(n^{3/2})$. This tells us how well we understand the problem, and also how much room for improvement we have.

It is often helpful when thinking about algorithms to imagine a game where one player is the algorithm designer, trying to come up with a good algorithm for the problem, and its opponent (the "adversary") is trying to come up with an input that will cause the algorithm to run slowly. An algorithm with good worst-case guarantees is one that performs well no matter what input the adversary chooses. We will return to this view in a more formal way when we discuss randomized algorithms and lower bounds.

1.4 An example: Karatsuba Multiplication

One thing that makes algorithm design "Computer Science" is that solving a problem in the most obvious way from its definitions is often not the best way to get a solution. A simple example of this is multiplication.

Say we want to multiply two n-bit numbers: for example, 41×42 (or, in binary, 101001×101010). According to the definition of what it means to multiply, what we are looking for is the result of adding 41 to itself 42 times (or vice versa). You could imagine actually computing the answer that way (i.e., performing 41 additions), which would be correct but not particularly efficient. If we used this approach to multiply two n-bit numbers, we would be making $\Theta(2^n)$ additions. This is exponential in n even without counting the number of steps needed to perform each addition. And, in general, exponential is bad.[1] A better way to multiply is to do what we learned in grade school:

```
        101001
x       101010
-------------
       1010010
     101001
+ 101001
-------------
  11010111010 = 1722
```

More formally, we scan the second number right to left, and every time we see a 1, we add a copy of the first number, shifted by the appropriate number of bits, to our total. Each addition takes $O(n)$ time, and we perform at most n additions, which means the total running time here is $O(n^2)$. So, this is a simple example where even though the problem is defined "algorithmically", using the definition is not the best way of solving the problem.

Is the above method the fastest way to multiply two numbers? It turns out it is not. Here is a faster method called Karatsuba Multiplication, discovered by Anatoli Karatsuba, in Russia, in 1962. In this approach, we take the two numbers X and Y and split them each into their most-significant

[1] This is reminiscent of an exponential-time sorting algorithm I once saw in Prolog. The code just contains the definition of what it means to sort the input — namely, to produce a permutation of the input in which all elements are in ascending order. When handed directly to the interpreter, it results in an algorithm that examines all $n!$ permutations of the given input list until it finds one that is in the right order.

half and their least-significant half.

```
                                 ---------------
X = A*2^{n/2} + B                |  A   |   B  |
                                 +--------------+
Y = C*2^{n/2} + D                |  C   |   D  |
                                 ---------------
```

We can now write the product of X and Y as

$$XY = 2^n AC + 2^{n/2}BC + 2^{n/2}AD + BD. \tag{1.1}$$

This does not yet seem so useful: if we use (1.1) as a recursive multiplication algorithm, we need to perform four $n/2$-bit multiplications, three shifts, and three $O(n)$-bit additions. If we use $T(n)$ to denote the running time to multiply two n-bit numbers by this method, this gives us a recurrence of

$$T(n) = 4T(n/2) + cn, \tag{1.2}$$

for some constant c. (The cn term reflects the time to perform the additions and shifts.) This recurrence solves to $O(n^2)$, so we do not seem to have made any progress. (In the next lecture we will go into the details of how to solve recurrences like this.)

However, we can take the formula in (1.1) and rewrite it as follows:

$$(2^n - 2^{n/2})AC + 2^{n/2}(A+B)(C+D) + (1-2^{n/2})BD. \tag{1.3}$$

It is not hard to see — you just need to multiply it out — that the formula in (1.3) is equivalent to the expression in (1.1). The new formula looks more complicated, but, it results in only *three* multiplications of size $n/2$, plus a constant number of shifts and additions. So, the resulting recurrence is

$$T(n) = 3T(n/2) + c'n, \tag{1.4}$$

for some constant c'. This recurrence solves to $O(n^{\log_2 3}) \approx O(n^{1.585})$.

Is *this* method the fastest possible? Again it turns out that one can do better. In fact, Karp discovered a way to use the Fast Fourier Transform to multiply two n-bit numbers in time $O(n \log^2 n)$. Schönhage and Strassen in 1971 improved this to $O(n \log n \log\log n)$, which is, asymptotically, the fastest algorithm known. We will discuss the FFT later on in this course.

Actually, the kind of analysis we have been doing really is meaningful only for very large numbers. On a computer, if you are multiplying numbers that fit into the word size, you would do this in hardware that has gates working in parallel. So instead of looking at sequential running time, in this case we would want to examine the size and depth of the circuit used, for instance. This points out that, in fact, there are different kinds of specifications that can be important in different settings.

1.5 Matrix multiplication

It turns out the same basic divide-and-conquer approach of Karatsuba's algorithm can be used to speed up matrix multiplication as well. To be clear, we will now be considering a computational

model where individual elements in the matrices are viewed as "small" and can be added or multiplied in constant time. In particular, to multiply two n-by-n matrices in the usual way (we take the ith row of the first matrix and compute its dot-product with the jth column of the second matrix in order to produce the entry ij in the output) takes time $O(n^3)$. If one breaks down each n by n matrix into four $n/2$ by $n/2$ matrices, then the standard method can be thought of as performing eight $n/2$-by-$n/2$ multiplications and four additions as follows:

$$\begin{array}{|c|c|}\hline A & B \\ \hline C & D \\ \hline \end{array} \times \begin{array}{|c|c|}\hline E & F \\ \hline G & H \\ \hline \end{array} = \begin{array}{|c|c|}\hline AE+BG & AF+BH \\ \hline CE+DG & CF+DH \\ \hline \end{array}$$

Strassen noticed that, as in Karatsuba's algorithm, one can cleverly rearrange the computation to involve only *seven* $n/2$-by-$n/2$ multiplications (and 14 additions).[2] Since adding two n-by-n matrices takes time $O(n^2)$, this results in a recurrence of

$$T(n) \quad = \quad 7T(n/2) + cn^2. \tag{1.5}$$

This recurrence solves to a running time of just $O(n^{\log_2 7}) \approx O(n^{2.81})$ for Strassen's algorithm.[3]

Matrix multiplication is especially important in scientific computation. Strassen's algorithm has more overhead than standard method, but it is the preferred method on many modern computers for even modestly large matrices. Asymptotically, the best matrix multiply algorithm known is by Coppersmith and Winograd and has time $O(n^{2.376})$, but is not practical. Nobody knows if it is possible to do better — the FFT approach doesn't seem to carry over.

---

[2]In particular, the quantities that one computes recursively are $q_1 = (A+D)(E+H)$, $q_2 = D(G-E)$, $q_3 = (B-D)(G+H)$, $q_4 = (A+B)H$, $q_5 = (C+D)E$, $q_6 = A(F-H)$, and $q_7 = (C-A)(E+F)$. The upper-left quadrant of the solution is $q_1+q_2+q_3-q_4$, the upper-right is q_4+q_6, the lower-left is q_2+q_5, and the lower right is $q_1-q_5+q_6+q_7$. (feel free to check!)

[3]According to Manuel Blum, Strassen said that when coming up with his algorithm, he first tried to solve the problem mod 2. Solving mod 2 makes the problem easier because you only need to keep track of the parity of each entry, and in particular, addition is the same as subtraction. One he figured out the solution mod 2, he was then able to make it work in general.