15-451: Algorithms

Lecture 1: Introduction, Karatsuba's Algorithm

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 necessarily the best 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 handout from web page)
- What is the study of Algorithms all about?
- Why do we care about specification and proving guarantees?
- The Karatsuba multiplication algorithm.
- Strassen's matrix multiplication algorithm.

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 their 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 an 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 students with intellectual tools for designing and analyzing their own algorithms for problems they 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 and counter-intuitive notion of NP-completeness. We will also spend some time on cryptography. Cryptography is interesting from the point of view of algorithm design because one uses a problem that's

assumed to be intrinsically hard in order to construct an algorithm (e.g., an encryption method) whose security rests on the difficulty of solving that hard problem.

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 that it is nice to know at least that our algorithm produces correct solutions, 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:

Composability. A guarantee on running time gives a type of "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.

Designing better algorithms. Analyzing the asymptotic running time of algorithms has shown itself to be 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.

You can probably think of more reasons too.

4 Problems and Algorithms

The purpose of an algorithm is solve a *combinatorial problem*. Here is a list of typical examples.

1. Given a number, determine whether it is a prime.
2. Given a number, find the least prime larger that the given number.
3. Given a number, determine one of its prime factors.
4. Given a number, determine all its prime factors.
5. Given a directed graph, determine the number of its SCCs.

6. Given a directed graph, decompose it into its SCCs.
7. Given a directed graph, test whether it acyclic.
8. Given a polynomial, count its real roots.
9. Given a graph, find a perfect matching.

Needless to say, truly interesting questions such as "what will be next week's lotto numbers?" won't be considered here.

There are different types of problems one has to contend with. Here are the most important ones.

- Decision Problems

  Here we are really asking a question, and the output is YES or NO. Typically we want to know whether the input has some special property or not.

- Function Problems

  These are perhaps the most important ones in applications. We are given some problem instance x and have to compute a uniquely determined solution z.

- Search Problems

  In a search problem, a given instance is associated with a set of solutions. The goal is to find one of these solutions, or to return NO if the set of solutions is empty.

Given this taxonomy we can classify the sample problems from above as follows. Problem 1 is a decision problem:

Problem:	**Primality**
Instance:	A positive integer n.
Question:	Is n prime?

Problem 2 is a function problem, we have to calculate an integer.

Problem:	**Next Prime**
Instance:	A positive integer n.
Solution:	The least integer $m > n$ that is prime.

Problem 3 is a search problem.

Problem:	**Prime Factor**
Instance:	A positive integer $n \geq 2$.
Solution:	Any prime p that divides n.

Make sure to figure out the type of all the other problems in the list above.

Note that for any problem instance x there is always a natural notion of the *size* of x, written $|x|$: count the number of bits needed to write down the instance. The only potentially tricky item here are numbers. We will always assume that numbers are represented in binary, so to represent n we need about $\log_2 n$ bits.

4.1 Time Complexity

One important way to assess the quality of an algorithm is to measure its running time. There are two natural ways of doing this.

- Actual physical running time
- Logical running time: number of steps needed to execute the algorithm.

One step is not clearly defined, think of in simply as one CPU cycle. A little less informally, Let P be a program implementing some algorithm. Define the *running time* of P on input x as

$$T_P(x) = \#\text{ steps in the execution of } P \text{ on input } x.$$

This is a bit too complicated to deal with in general, an easier measure is *worst case running time* of all inputs of the same size:

$$T_P(n) = \max\big(T_P(x) \;\big|\; x \text{ an instance of size } n\big)$$

This is somewhat pessimistic, a better measure is *average case* running time

$$T_P^{\text{avg}}(n) = \sum_{|x|=n} p_x \cdot T_P(x)$$

Here p_x is the probability of instance x.

5 An example: Karatsuba Multiplication

One of things 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
```

[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.

```
+ 101001
-------------
  11010111010 = 1722
```

The running time here is $O(n^2)$, because we are performing n additions, each of which takes $O(n)$ time. 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 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

$$XY = 2^n AC + 2^{n/2} BC + 2^{n/2} AD + BD. \quad (1)$$

This does not yet seem so useful. If we use (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

$$T(n) = 4T(n/2) + cn, \quad (2)$$

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) and rewrite it as follows:

$$(2^n + 2^{n/2})AC - 2^{n/2}(A - B)(C - D) + (2^{n/2} + 1)BD. \quad (3)$$

It is not hard to see — you just need to multiply it out — that the formula in (3) is equivalent to the expression in (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

$$T(n) = 3T(n/2) + cn, \quad (4)$$

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.

6 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 multiply two n-by-n matrices in the usual way 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] This results in a recurrence of

$$T(n) \quad = \quad 7T(n/2) + cn^2. \tag{5}$$

(Adding two n-by-n matrices takes time $O(n^2)$.) This recurrence solves to a running time of just $O(n^{\log_2 7}) \approx O(n^{2.81})$ for Strassen's algorithm.

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!)