Algorithms: Design and Analysis

Note from Editor

The following contents have been taken from the EdX page of Stanford Online 's course titled "Algorithms: Design and Analysis, Part I". The contents of this book are a compilation of the transcripts of the various lectures in the course. It also includes some of the written overviews provided with the course. Certain irrelevant parts may be left out, however a copy without these deletions can be found from the same source as this one, on GitHub.

Introduction

Overview

WELCOME: Welcome to Algorithms: Design and Analysis, Part I! Here's an overview of the first few sections of material.

INTRODUCTION: The first set of lectures for this week is meant to give you the flavor of the course, and hopefully get you excited about it. We begin by discussing algorithms in general and why they're so important, and then use the problem of multiplying two integers to illustrate how algorithmic ingenuity can often improve over more straightforward or naive solutions. We discuss the Merge Sort algorithm in detail, for several reasons: it's a practical and famous algorithm that you should all know; it's a good warm-up to get you ready for more intricate algorithms; and it's the canonical introduction to the "divide and conquer" algorithm design paradigm. These lectures conclude by describing several guiding principles for how we'll analyze algorithms in this course.

ASYMPTOTIC ANALYSIS: The second set of lectures for this week is an introduction to big-oh notation and its relatives, which belongs in the vocabulary of every serious programmer and computer scientist. The goal is to identify a "sweet spot" of granularity for reasoning about algorithms --- we want to suppress second-order details like constant factors and lower-order terms, and focus on how the running time of an algorithm scales as the input size grows large.

DIVIDE AND CONQUER ALGORITHMS: The final set of lectures for this week discusses three non-trivial examples of the divide and conquer algorithm design paradigm. The first is for counting the number of inversions in an array. This problem is related to measuring similarity between two ranked lists, which in turn is relevant for making good recommendations to someone based on your knowledge of their and others' preferences ("collaborative filtering"). The second algorithm is Strassen's mind-blowing recursive algorithm for matrix multiplication, which improves over the obvious iterative method. The third algorithm, which is more advanced and is optional material, is for computing the closest pair of points in the plane.

PREREQUISITES: This course is not an introduction to programming, and it assumes that you have basic programming skills in a language such as Python, Java, or C. There are several outstanding free online courses that teach basic programming. We also use mathematical analysis as needed to understand how and why algorithms and data structures really work. If you need a refresher on the basics of proofs (induction, contradiction, etc.), I recommend the lecture notes "Mathematics for Computer Science" by Lehman and Leighton (see separate Resources pages).

DISCUSSION FORUMS: The discussion forums play a crucial role in massive online courses like this one, which is an all-volunteer effort. If you have trouble understanding a lecture or completing an assignment, you should turn to the forums for help. After you've mastered the lectures and assignments for a given week, I hope you'll contribute to the forums and help out your fellow students. While I won't have time to carefully monitor the discussion forums, I'll check in and answer questions whenever I find the time.

VIDEOS AND SLIDES: Videos can be streamed or downloaded and watched offline (recommended for commutes, etc.). We are also providing PDF lecture slides (typed versions of what's written in the lecture videos), as well as subtitle files. And if you find yourself wishing that I spoke more quickly or more slowly, note that you can adjust the video speed to accommodate your preferred pace.

Why Study Algorithms?

Now, I imagine many of you are already clear on your reasons for taking this course. But let me begin by justifying this course's existence. And giving you some reasons why you should be highly motivated to learn about algorithms.

So, what is an algorithm anyways? Basically, it's a set of well-defined rules, a recipe in effect for solving some computational problem. Maybe you have a bunch of numbers and you want to rearrange them so that they're in sorted order. Maybe you have a roadmap and an origin and a destination and you want to compute the shortest path from that origin to that destination. Maybe you face a number of different tasks that need to be completed by certain deadlines and you want to know in what order you should accomplish the task. So that you complete them all by their respective deadlines.

So why study algorithms? Well first of all, understanding the basics of algorithms and the related field of data structures is essential for doing serious work in pretty much any branch of computer science. This is the reason why here at Stanford, this course is required for every single degree that the department offers. The bachelor's degree the master's degree and also the PHD. To give you a few examples routing and communication networks piggybacks on classical shortest path algorithms. The effectiveness of public key cryptography relies on that of number-theoretic algorithms. Computer graphics needs the computational primitives supplied by geometric algorithms. Database indices rely on balanced search tree data structures. Computational biology uses dynamic programming algorithms to measure genome similarity. And the list goes on.

Second, algorithms play a key role in modern technological innovation. To give just one obvious example, search engines use a tapestry of algorithms to efficiently compute the relevance of various webpages to its given search query. The most famous such algorithm is the page rank algorithm currently in use by Google. Indeed, in a December 2010 report to the United States White House, the President's council of advisers on science and technology argued that in many areas' performance gains due to improvements in algorithms have vastly exceeded even the dramatic performance gains due to increased processor speeds.

Third, although this is outside of the score, the scope of this course. Algorithms are increasingly being used to provide a novel lens on processes outside of computer science and technology. For example, the study of quantum computation has provided a new Computational viewpoint on quantum mechanics. Price fluctuations in economic markets can be fruitfully viewed as an algorithmic process and even evolution can be usefully thought of as a surprisingly effect search algorithm.

The last two reasons for studying algorithms might sound flippant but both have more than a grain of truth to them. I don't know about you, but back when I was a student, my favorite classes were always the challenging ones that, after I struggled through them, left me feeling a few IQ points smarter than when I started. I hope this course provides a similar experience for many of you. Finally, I hope that by the end of the course I'll have converted some of you to agree with me that the design and analysis of algorithms is simply fun. It's an endeavor that requires a rare blend of precision and creativity. It can certainly be frustrating at times, but it's also highly addictive.

So, let's descend from these lofty generalities and get much more concrete. And let's remember that we've all been learning about and using algorithms since we were little kids.

Integer Multiplication

Sometime when you were a kid, maybe around third grade, you learned an algorithm for multiplying two numbers. Maybe your third-grade teacher didn't call it that, maybe that's not how you thought about it. But you learned a well-defined set of rules for transforming input, namely two numbers, into an output, namely their product. So, that is an algorithm for solving a computational problem.

Let's pause and be precise about it. Many of the lectures in this course will follow a pattern. We'll define a computational problem. We'll say what the input is, and then we'll say what the desired output is. Then we will proceed to giving a solution, to giving an algorithm that transforms the input to the output.

When the integer multiplication problem, the input is just two n-digit numbers. So, the length, n, of the two input integers x and y could be anything, but for motivation, you might want to think of n as large, in the thousands or even more, perhaps we're implementing some kind of cryptographic application which has to manipulate very large numbers. We also need to explain what the desired output is; in this simple problem, it's simply the product x times y.

A quick digression: back in 3rd grade, around the same time I was learning the Integer Multiplication Algorithm, I got a C in penmanship, and I don't think my handwriting has improved much since. Many people tell me by the end of the course. They think of it fondly as a sort of acquired taste, but if you're feeling impatient, please note there are typed versions of these slides. Which I encourage you to use as you go through the lectures if you don't want to take the time deciphering the handwriting.

Returning to the Integer Multiplication problem, having now specified the problem precisely, the input, the desired output, we'll move on to discussing an algorithm that solves it, namely, the same algorithm you learned in third grade. The way we will assess the performance of this algorithm is through the number of basic operations that it performs.

And for the moment, let's think of a basic operation as simply adding two single-digit numbers together or multiplying two single-digit numbers. We're going to then move on to counting the number of these basic operations performed by the third-grade algorithm. As a function of the number n of digits in the input.

Here's the integer multiplication algorithm that you learned back in third grade, illustrated on a concrete example. Let's take, say, the numbers 1, 2, 3, 4, and 5, 6, 7, 8. As we go through this algorithm quickly, let me remind you that our focus should be on the number of basic operations this algorithm performs. As a function of the length of the input numbers, which, in this particular example, is four digits long.

So, as you'll recall, we just compute one partial product for each digit of the second number. We start by just multiplying 4 times the upper number 5, 6, 7, 8. So, you know, 4 times 8 is 32, 2 carry to 3, 4 times 7 is 28, with the 3 that's 31, write down the 1, carry the 3, and so on. When we do the next partial product, we do a shift effectively, we add a 0 at the end, and then we just do exactly the same thing. And so on for the final two partial products. And finally, we just add everything up.

What you probably realized back in third grade is that this algorithm is what we would call correct. That is, no matter what integers x and y you start with if you carry out this procedure, this algorithm. And all of your intermediate computations are done properly. Then the algorithm will eventually terminate with the product, x times y, of the two input numbers. You're never going to get a wrong answer. You're always going to get the actual product.

What you probably didn't think about was the amount of time needed to carry out this algorithm out to its conclusion, to termination. That is the number of basic operations, additions or multiplications of single-digit numbers needed before finishing.

So, let's now quickly give an informal analysis of the number of operations required as a function of the input length n. Let's begin with the first partial product, the top row. How did we compute this number 22,712? Well, we multiplied 4 times each of the numbers 5, 6, 7, and 8. So that was four basic operations. One for each digit at the top number, plus we had to do these carries. So those were some extra additions. But in any case, this is at most twice times the number of digits in the first number. At most two n basic operations to form this first partial product.

And if you think about it, there's nothing special about the first partial product. The same argument says that we need at most 2n operations to form each of the partial products of which there are again n, one for each digit of the second number. Well, if we need at most two n operations to compute each partial product and we have n partial products. That's a total of at most 2n^2 operations to form all of these blue numbers, all of the partial products.

Now we're not done at that point. We still have to add all of those up to get the final answer, in this case 7,006,652. And that final addition requires a comparable number of operations. Roughly, another says two n^2, at most operations.

So, the upshot, the high-level point that I want you to focus on, is that as we think about the input numbers getting bigger and bigger. That is as a function of n the number of digits in the input numbers. The number of operations that the Grade-School Multiplication Algorithm performs, grows like some constant. Roughly 4 say times n^2. That is, it's quadratic in the input length n.

For example, if you double the size of the input, if you double the number of digits in each of the two integers that you're given. Then the number of operations you will have to perform using this algorithm has to go up by a factor of four. Similarly, if you quadruple the input length, the number of operations going, is going to go up by a factor of 16, and so on.

Now, depending on what type of third-grader you were. You might well have accepted this procedure as the unique or at least the optimal way of multiplying two numbers together to form their product.

Now if you want to be a serious algorithm designer. That kind of obedient timidity is a quality you're going to have to grow out of. And an early and extremely important textbook on the design and analysis of algorithms was by Aho, Hopcroft, and Ullman. It's about 40 years old now. And there's the following quote, which I absolutely adore.

So, after iterating through a number of the algorithm design paradigms covered in the textbook. They say the following, perhaps the most important principle of all, for the good algorithm designer is to refuse to be content. And I think this is a spot-on comment.

I might summarize it a little bit more succinctly. As, as an algorithm designer you should adopt as your Mantra the question, can we do better? This question is particularly apropos when you're faced with a naive or straight-forward solution to a computation problem. Like for example, the third-grade algorithm for integer multiplication. The question you perhaps did not ask yourself in third grade was, can we do better than the straight forward multiplication algorithm? And now is the time for an answer.