CS 161 (Stanford, Winter 2025) Lecture 1

Adapted From Virginia Williams' lecture notes. Additional credits: J. Su, W. Yang, Gregory Valiant, Mary Wootters, Aviad Rubinstein.
Please direct all typos and mistakes to Nima Anari and Moses Charikar.

Introduction

1 Logistics

The class website is at `https://cs161.stanford.edu`. All course information is available on the website.

2 Why are you here?

Many of you are here because the class is required. But why is it required?

1. **Algorithms are fundamental to all areas of CS:** Algorithms are the backbone of computer science. Wherever computer science reaches, an algorithm is there, and the classical algorithms algorithmic design paradigms that we cover re-occur throughout all areas of CS. For example, CS 140 and 143 (operating systems and compiles) leverages scheduling algorithms and efficient data structures, CS 144 (networking) crucially uses shortest-path algorithms, CS 229 (machine learning) leverages fast geometric algorithms and similarity search, CS 255 (cryptography) leverages fast number theoretic and algebraic algorithms, and CS 262 (computational biology) leverages algorithms that operate on strings–and often employs the dynamic programming paradigm. We'll discuss applications to all these subjects in this class.

   Algorithms and the computational perspective (the "computational lens") has also been fruitfully applied to other areas, such as physics (e.g. quantum computing), economics (e.g. algorithmic game theory), and biology (e.g. for studying evolutions, as a surprisingly efficient algorithm that searches the space of genotypes).

2. **Algorithms are useful:** Much of the progress that has occurred in tech/industry is due to the dual developments of improved hardware (a la "Moore's Law"—a prediction made in 1965 by the co-founder of Intel that the density of transistors on integrated circuits would double every year or two), and improved algorithms. In fact, the faster computers get, the bigger the discrepancy is between what can be accomplished with fast algorithms vs what can be accomplished with slow algorithms. . . . Industry needs to continue developing new algorithms for the problems of tomorrow, and you can help contribute.

3. **Algorithms are fun!** The design and analysis of algorithms requires a combination of

creativity and mathematical precision. It is both an art and a science, and hopefully at least some of you will come to love this combination. One other reason it is so much fun is that algorithmic surprises abound. Hopefully this class will make you re-think what you thought was algorithmically possible, and cause you to constantly ask “is there a better algorithm for this task?”. Part of the fun is that Algorithms is still a young area, and there are still many mysteries, and many problems for which (we suspect that) we still do not know the best algorithms. This is what makes research in Algorithms so fun and exciting, and hopefully some of you will decide to continue in this direction.

3 Karatsuba Integer Multiplication

3.1 The problem

Suppose you have two large numbers, and you want to multiply them. Of course, you all know how to solve this problem: you learned an algorithm (which we'll call the “grade-school algorithm") when you were in grade school. The question is, **can we do better?**

In order to understand this, we need to talk at least a little bit about what we mean by better. How do we measure the running time of an algorithm? It's tempting to measure it in units of time—say, milliseconds on a computer. However, this doesn't really capture the running time of an *algorithm.* Rather, it captures the running time of an algorithm, with a particular implementation, on a particular piece of hardware. For example, grade-school multiplication is much faster on a computer than by hand, but it's still the same algorithm in both cases.

Instead, we'll focus on how fast the running time *scales* as a function of the input. We will be a bit more precise about this in the next lecture, but for now, we'll define this notion by example. Suppose we use the grade-school algorithm to multiply two n-digit numbers. The bulk of the work is taken up by multiplying every pair of digits together. For example, in 1234×6789, we have to multiply $9 \times 4, 9 \times 3, 9 \times 2, 9 \times 1, 8 \times 3$, etc. There are n^2 such pairs, so we'll say that this algorithm has a running time that scales like n^2.

Why should we care about this measure of complexity? We'll talk about this more next lecture, but intuitively, this scaling behavior is the thing that really matters as n gets large. Suppose we had two algorithms, one of which had running time that scaled like n^2 and one which scaled like $n^{1.6}$. Suppose that running an algorithm by hand is 10000 times slower than running an algorithm on a computer. For large enough n, $10000n^{1.6} < n^2$, and intuitively this means it would actually be faster to run the $n^{1.6}$ algorithm by hand than the n^2 algorithm on a computer. So we can definitively say that the $n^{1.6}$ algorithm is “faster," because for large n, it will be faster, no matter how the algorithm is implemented and no matter what hardware it's running on.

With that in mind, our question is now this: can we multiply two n-digit integers faster than the grade-school algorithm? That is, with a running time that scales faster than n^2?

One try might be to store the answers ahead of time, or at least store partial answers. For example, we could store the products of all pairs of n-digit numbers, and then just look up the pair we need. This does result in performance gains, however, and also leads to exponential storage costs. (For example, if $n = 100$, we would need to store a table of $10^{2n} = 10^{200}$ products. Note that the number of atoms in the universe is only $\approx 10^{80}$....) So we'll have to do something more clever.

3.2 Divide and conquer

The "Divide and Conquer" algorithm design paradigm is a very useful and widely applicable technique. We will see a variety of problems to which it can be fruitfully applied. The high-level idea is just to split a given problem into smaller pieces and then solve the smaller ones, often recursively.

How can we apply Divide and Conquer to integer multiplication? Let's try splitting up the numbers. For example, if we were multiplying 1234×5678, we could express this as $((12 \cdot 100) + 34) \cdot ((56 \cdot 100) + 78)$. In general, if we are multiplying two n-digit numbers x and y, we can write $x = 10^{n/2} \cdot a + b$ and $y = 10^{n/2} \cdot c + d$. So

$$x \cdot y = (10^{n/2}a + b) \cdot (10^{n/2}c + d) = 10^n ac + 10^{n/2}(ad + bc) + bd.$$

Now we can split this problem into four subproblems, where each subproblem is similar to the original problem, but with half the digits. This gives rise to a recursive algorithm.

Interestingly enough, **this algorithm isn't actually better!** Intuitively this is because if we expand the recursion, we still have to multiply every pair of digits, just like we did before. But in order to prove this formally, we need to formally define the runtime of an algorithm and prove that these algorithms are not very different in runtime.

3.3 Recurrence relation

We can analyze the runtime of the algorithm as follows.

Let $T(n)$ be the runtime of the algorithm, given an input of size n (two n-digit numbers). Because we are breaking up the problem into four subproblems with half the digits, plus some addition with linear cost, we have the equation $T(n) = 4T(n/2) + O(n)$. (Don't worry if you haven't seen big-O notation before; we'll go over this in detail in the coming lectures.)

Although **in general you should pay attention to the $O(n)$ term**, today we will just ignore it because the term doesn't matter in this case.

By repeatedly breaking up the problem into subproblems, we find that

$$T(n) = 4T(n/2) = 16T(n/4) = \ldots = 2^{2t}T\left(\frac{n}{2^t}\right) = n^2T(1).$$

Since $T(1)$ is the time it takes to multiply two digits, we see that the above suggestion does not reduce the number of 1-digit operations.

Note: In the lecture slides, we'll consider a slightly different argument, which analyzes a recursion tree. It's a good exercise to understand both arguments! Again, we'll discuss both techniques more in coming lectures.

3.4 Divide and conquer (take 2)

Karatsuba found a better algorithm (in 1960, published in 1962) by noticing that we only need the sum of ad and bc, not their actual values. So he improved the algorithm by computing ac and bd as before, and computing $(a+b)\cdot(c+d)$. It turns out that if $t=(a+b)\cdot(c+d)$, then $ad+bc=t-ac-bd$. Now instead of solving four subproblems, we only need to solve three! This idea goes back to Gauss, who found a similar efficient way to multiply two complex numbers.

Sure, we need to do more additions, but again it turns out that additions are pretty cheap. To do a quick-and-dirty analysis of the number of operations required by Karatsuba multiplication, first assume that $n=2^s$ for some integer s. (Note that we can always add 0's to the front of a number until the length is a power of two, so this assumption holds without loss of generality.) Letting $T(n)$ denote the number of multiplications of pairs of 1-digit numbers required to compute the product of two n-digit numbers, Karatsuba's algorithm gives $T(n)=3T(n/2)$, since we've divided the problem into three recursive calls to multiplication of length $n/2$ numbers. [Note, we are cheating a bit here, since $(a+b)$ and $(c+d)$ might actually be $n/2+1$ digit numbers, but lets ignore this for now...] Hence we have the following:

$$T(n)=3T(n/2)=3^2T(n/4)=\ldots=3^sT(n/2^s).$$

Since we assumed $n=2^s$, we have that $T(n/2^s)=T(1)=1$, since multiplying two 1-digit numbers counts as 1 basic operation. Hence $T(n)=3^s$, where $n=2^s$. Solving for s yields $s=\log_2 n$, and hence we get

$$T(n)=3^{\log_2 n}=2^{(\log_2 3)(\log_2 n)}=n^{\log_2 3}\leq n^{1.6}.$$

We were pretty sloppy with the above argumentation in a lot of ways. However, we'll see a much more principled way of analyzing the runtime of recursive algorithms in the coming classes, so we won't sweat about it too much now. The point is that (even if you do it correctly) the running time of this algorithm scales like $n^{1.6}$. Thus is **much** better than the n^2 algorithm that we learned in grade school!

3.5 Can we do better?

Progress on efficient algorithms for multiplication of n-digit numbers continued beyond Karatsuba's algorithm. Although you don't need to know these algorithms for CS 161, it is interesting to review the history of progress on this problem. Toom and Cook (1963) developed an algorithm that ran in time $O(n^{1.465})$ by showing how a single n-sized problem could be broken up into five $n/3$-sized problems. Schönage and Strassen (1971) developed an algorithm that runs in time $O(n \log(n) \log\log(n))$. More than 35 years later, Fürer (2007) developed an algorithm that ran in time $n \log(n) 2^{\log^*(n)}$. In case you are wondering what that weird function $\log^*(n)$ (read "log star") is, it is the number of times you have to apply the logarithm function log() iteratively to n in order to get down to something ≤ 1. For all values of n less than the estimated number of atoms in the universe, the value of $\log^*(n)$ (with base 2) is less than 5. So $\log^*(n)$ is a really really *really* slowly growing function of n. Finally, Harvey and van der Hoeven (2019) gave an algorithm that runs in time $O(n \log n)$. This is conjectured to be optimal. It is quite amazing that the seemingly simple (and old) question of multiplying two numbers has proved to be so mysterious and has seen new research advances as recently as 2019. This is what makes the study of algorithms so exciting!

4 Etymology of the word "Algorithm"

As a round-about way of describing the etymology of the word "Algorithm", pause for a minute and consider how remarkable it is that 3rd graders can actually multiply large numbers. It's really amazing that anyone, let alone an 8-yr old, can multiply two 10-digit numbers. One reason multiplication is so easy for us is that we have a great data structure for numbers—we represent numbers using base-10 (Arabic) numerals, and this lends itself to easy arithmetic.

Why were Romans so bad at multiplication? Well, imagine multiplying using roman numerals. What is LXXXIX times CM? The only way I can imagine computing this is to first translate the numbers into Arabic numerals [$LXXXIX = 50 + 10 + 10 + 10 + (-1) + 10 = 89$, and $CM = (-100) + 1000 = 900$] then multiply those the standard way. Roman numerals seem like a pretty lousy data structure if you want to do arithmetic.

The word "Algorithm" is a mangled transliteration of the name "al-Khwarizmi." Al-Khwarizmi was a 9th-century Persian polymath, born in present-day Uzbekistan, who studied and worked in Baghdad during the Abbassid Caliphate; around 820 AD he was appointed as the astronomer and head of the library of the House of Wisdom in Baghdad. He wrote several influential books, including one with the title [something like] "On the Calculation with Hindu Numerals", which described how to do arithmetic using Arabic numerals (aka Arabic-Hindu, or just Hindu numerals). The original manuscript was lost, though a Latin translation from the 1100s introduced this number system to Europe, and is responsible for why we use Arabic numerals today. (You can imagine how happy a 12th-century tax collector would have been with this new ability to easily do arithmetic....) [The old French word *algorisme* meant "the Arabic numerals system", and only later did it come to mean a general recipe for solving computational problems.]