**Learning Objectives:**

- **Introduce and motivate the study of computation for its own sake, irrespective of particular implementations.**
- **The notion of an *algorithm* and some of its history.**
- **Algorithms as not just *tools*, but also *ways of thinking and understanding*.**
- **Taste of Big-$O$ analysis and the surprising creativity in the design of efficient algorithms.**

# 0
# *Introduction*

> *"Computer Science is no more about computers than astronomy is about telescopes"*, attributed to Edsger Dijkstra. [1]

> *"Hackers need to understand the theory of computation about as much as painters need to understand paint chemistry."* , Paul Graham 2003. [2]

> *"The subject of my talk is perhaps most directly indicated by simply asking two questions: first, is it harder to multiply than to add? and second, why?...I (would like to) show that there is no algorithm for multiplication computationally as simple as that for addition, and this proves something of a stumbling block."*, Alan Cobham, 1964

The origin of much of science and medicine can be traced back to the ancient Babylonians. However, the Babylonians' most significant contribution to humanity was arguably the invention of the *place-value number system*. The place-value system represents any number using a collection of digits, whereby the *position* of the digit is used to determine its value, as opposed to a system such as Roman numerals, where every symbol has a fixed numerical value regardless of position. For example, the distance to the moon is 238,900 of our miles or 259,956 Roman miles. The latter quantity, expressed in standard Roman numerals is

MMMMMMMMMMMMMMMMMMMMMMMMMMM
MMMMMMMMMMMMMMMMMMMMMMMMMMM
MMMMMMMMMMMMMMMMMMMMMMMMMMM
MMMMMMMMMMMMMMMMMMMMMMMMMMM
MMMMMMMMMMMMMMMMMMMMMMMMMMM
MMMMMMMMMMMMMMMMMMMMMMMMMMM
MMMMMMMMMMMMMMMMMMMMMMMMMMM

[1] This quote is typically read as disparaging the importance of actual physical computers in Computer Science, but note that telescopes are absolutely essential to astronomy as they provide us with the means to connect theoretical predictions with actual experimental observations.

[2] To be fair, in the following sentence Graham says "you need to know how to calculate time and space complexity and about Turing completeness". This book includes these topics, as well as others such as NP-hardness, randomization, cryptography, quantum computing, and more.

```
MMMMMMMMMMMMMMMMMMMMMMMMMMMMM
MMMMMMMMMMMMMMMMMMMDCCCCLVI
```

Writing the distance to the sun in Roman numerals would require about 100,000 symbols: a 50-page book just containing this single number!

The above means that for someone who thinks of numbers in an additive system like Roman numerals, quantities like the distance to the moon or sun are not merely large- they are *unspeakable*: cannot be expressed or even grasped. It's no wonder that Eratosthenes, who was the first person to calculate the earth's diameter (up to about ten percent error) and Hipparchus who was the first to calculate the distance to the moon, did not use a Roman-numeral type system but rather the Babylonian sexadecimal (i.e., base 60) place-value system.

The Babylonians also invented the precursors of the "standard algorithms" that we were all taught in elementary school for adding and multiplying numbers.[3] These algorithms and their variants have been of course essential to people throughout history working with abaci, papyrus, or pencil and paper, but in our computer age, do they still serve any purpose beyond torturing third graders?

To answer this question, let us try to see in what sense is the standard digit by digit multiplication algorithm "better" than the straightforward implementation of multiplication as iterated addition. Let's start by more formally describing both algorithms:

[3] For more on the actual algorithms the Babylonians used, see Knuth's paper and Neugebauer's classic book.

**Algorithm 0.1 — Multiplication via repeated addition.**
**Input:** Non-negative integers $x, y$
**Operation:**

1. Let $result \leftarrow 0$.
2. For $i = 1, \ldots, y$: set $result \leftarrow result + x$
3. Output $result$

**Algorithm 0.2 — Grade-school multiplication.**
**Input:** Non-negative integers $x, y$.
**Operation:**

1. Let $n$ be number of digits of $y$, and set $result \leftarrow 0$.
2. For $i = 0, \ldots, n-1$: set $result \leftarrow result + 10^i \times y_i \times x$, where $y_i$ is the $i$-th digit of $y$ (i.e. $y = 10^0 y_0 + 10^1 y_1 + \cdots + y_{n-1} 10^{n-1}$)
3. Output $result$.

Both algorithms assume that we already know how to add numbers, and the second one also assumes that we can multiply a number by a power of 10 (which is, after all, a simple shift) as well as multiply by a single-digit (which like addition, is done by multiplying each digit and propagating carries).

Suppose that $x$ and $y$ are two numbers of $n$ decimal digits each. Adding two such numbers takes at least $n$ single-digit additions (depending on how many times we need to use a "carry"), and so adding $x$ to itself $y$ times as per Algorithm 0.1 will take at least $n \cdot y$ single-digit additions. In contrast, the grade-school algorithm (i.e., Algorithm 0.2) reduces this problem to taking $n$ products of $x$ with a single-digit (which require up to $2n$ single-digit operations each, depending on carries) and then adding all of those together (total of $n$ additions, which, again depending on carries, would cost at most $2n^2$ single-digit operations) for a total of at most $4n^2$ single-digit operations. How much faster would $4n^2$ operations be than $n \cdot y$? Also, would this make any difference in a modern computer?

Let us consider the case of multiplying 64-bit or 20-digit numbers.[4] That is the task of multiplying two numbers $x, y$ that are between $10^{19}$ and $10^{20}$. Since in this case $n = 20$, the grade-school algorithm (Algorithm 0.2) would use at most $4n^2 = 1600$ single-digit operations, while repeated addition (Algorithm 0.1) would require at least $n \cdot y \geq 20 \cdot 10^{19}$ single-digit operations. To understand the difference, consider that a human being can perform a single-digit operation in about 2 seconds, requiring just under an hour to complete the calculation of $x \times y$ using the grade-school algorithm. In contrast, even though it is more than a billion times faster than a human, a modern PC that computes $x \times y$ using naïve iterated addition would require about $10^{20}/10^9 = 10^{11}$ seconds (which is more than three millennia!) to compute the same result.

[4] This is a typical size in several programming languages; for example the `long` data type in the Java programming language, and (depending on architecture) the `long` or `long long` types in C.

**Remark 0.3 — Value vs. length of a number..** It is important to distinguish between the *value* of a number, and the *length of its representation* (i.e., the number of digits it has). There is a big difference between the two: having 1,000,000,000 dollars is not the same as having 10 dollars! When talking about running time of algorithms, "less is more", and so an algorithm that runs in time proportional to the *number of digits* of an input number (or even the number of digit squared) is much preferred to an algorithm that runs in time proportional to the *value* of the input number.

We see that computers have not made algorithms obsolete. On the contrary, the vast increase in our ability to measure, store, and

communicate data has led to much higher demand for developing better and more sophisticated algorithms that can allow us to make better decisions based on these data. We also see that in no small extent the notion of *algorithm* is independent of the actual computing device that executes it. The digit-by-digit multiplication algorithm is vastly better than iterated addition, regardless whether the technology we use to implement it is a silicon-based chip, or a third grader with pen and paper.

Theoretical computer science is concerned with the *inherent* properties of algorithms and computation; namely, those properties that are *independent* of current technology. We ask some questions that were already pondered by the Babylonians, such as "what is the best way to multiply two numbers?", but also questions that rely on cutting-edge science such as "could we use the effects of quantum entanglement to factor numbers faster?".

In Computer Science parlance, a scheme such as the decimal (or sexadecimal) positional representation for numbers is known as a *data structure*, while operations on such representations are known as *algorithms*. Data structures and algorithms have enabled amazing applications that have transformed human society, but their importance goes beyond their practical utility. Structures from computer science, such as bits, strings, graphs, and even the notion of a program itself, as well as concepts such as universality and replication, have not just found (many) practical uses but contributed a new language and a new way to view the world.

## 0.1 EXTENDED EXAMPLE: A FASTER WAY TO MULTIPLY

Once you think of the standard digit-by-digit multiplication algorithm, it seems that it is clearly the "right" way to multiply numbers. Indeed, in 1960, the famous mathematician Andrey Kolmogorov organized a seminar at Moscow State University in which he conjectured that every algorithm for multiplying two $n$ digit numbers would require a number of basic operations that is proportional to $n^2$.[5] In other words, Kolmogorov conjectured that in any multiplication algorithm, doubling the number of digits would *quadruple* the number of basic operations required.

A young student named Anatoly Karatsuba was in the audience, and within a week he found an algorithm that requires only about $Cn^{1.6}$ operations for some constant $C$. Such a number becomes much smaller than $n^2$ as $n$ grows.[6] Amazingly, Karatsuba's algorithm is based on a faster way to multiply *two-digit* numbers.

Suppose that $x, y \in [100] = \{0, \ldots, 99\}$ are a pair of two-digit numbers. Let's write $\overline{x}$ for the "tens" digit of $x$, and $\underline{x}$ for the "ones" digit, so that $x = 10\overline{x} + \underline{x}$, and write similarly $y = 10\overline{y} + \underline{y}$ for $\overline{y}, \underline{y} \in [10]$.

[5] That is, he conjectured that the number of operations would be at least some $n^2/C$ operations for some constant $C$ or, using "Big-$O$ notation", $\Omega(n^2)$ operations. See the *mathematical background* chapter for a precise definition of Big-$O$ notation.

[6] At the time of this writing, the standard Python multiplication implementation switches from the elementary school algorithm to Karatsuba's algorithm when multiplying numbers larger than 1000 bits long.

The grade-school algorithm for multiplying $x$ and $y$ is illustrated in Fig. 1.

The grade-school algorithm works by transforming the task of multiplying a pair of two-digit number into *four* single-digit multiplications via the formula

$$(10\overline{x}+\underline{x})\times(10\overline{y}+\underline{y}) = 100\overline{x}\overline{y} + 10(\overline{x}\underline{y}+\underline{x}\overline{y}) + \underline{x}\underline{y} \quad (1)$$

Karatsuba's algorithm is based on the observation that we can express Eq. (1) also as

$$(10\overline{x}+\underline{x})\times(10\overline{y}+\underline{y}) = (100-10)\overline{x}\overline{y}+10\left[(\overline{x}+\underline{x})(\overline{y}+\underline{y})\right]-(10-1)\underline{x}\underline{y} \quad (2)$$

which reduces multiplying the two-digit number $x$ and $y$ to computing the following three "simple" products: $\overline{x}\overline{y}$, $\underline{x}\underline{y}$ and $(\overline{x}+\underline{x})(\overline{y}+\underline{y})$.[7]

Of course, if all we wanted was to multiply two digit numbers, we wouldn't need any clever algorithms. It turns out that we can repeatedly apply the same idea, and use them to multiply 4-digit numbers, 8-digit numbers, 16-digit numbers, and so on and so forth. If we used the grade-school approach, then our cost for doubling the number of digits would be to *quadruple* the number of multiplications, which for $n = 2^\ell$ digits would result in about $4^\ell = n^2$ operations. In contrast, in Karatsuba's approach doubling the number of digits only *triples* the number of operations, which means that for $n = 2^\ell$ digits we require about $3^\ell = n^{\log_2 3} \sim n^{1.58}$ operations.

Specifically, we use a *recursive* strategy as follows:

**Algorithm 0.4 — Karatsuba multiplication.**
**Input:** nonnegative integers $x, y$ each of at most $n$ digits
**Operation:**

1. If $n \leq 2$ then return $x \cdot y$ (using a constant number of single-digit multiplications).
2. Otherwise, let $m = \lfloor n/2 \rfloor$, and write $x = 10^m\overline{x} + \underline{x}$ and $y = 10^m\overline{y} + \underline{y}$. [8]
3. Use *recursion* to compute $A = \overline{x}\overline{y}$, $B = \underline{x}\underline{y}$ and $C = (\overline{x} + \underline{x})(\overline{y}+\underline{y})$. Note that all the numbers will have at most $m+1$ digits.
4. Return $(10^n - 10^m)\cdot A + 10^m \cdot B + (1 - 10^m)\cdot C$

To understand why the output will be correct, first note that for $n > 2$, it will always hold that $m < n - 1$, and hence the recursive calls will always be for multiplying numbers with a smaller number of digits, and (since eventually we will get to single or double digit numbers)

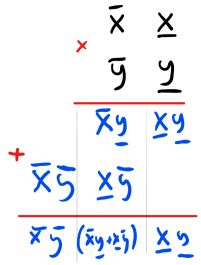

**Figure 1**: The grade-school multiplication algorithm illustrated for multiplying $x = 10\overline{x} + \underline{x}$ and $y = 10\overline{y} + \underline{y}$. It uses the formula $(10\overline{x} + \underline{x}) \times (10\overline{y} + \underline{y}) = 100\overline{x}\overline{y} + 10(\overline{x}\underline{y} + \underline{x}\overline{y}) + \underline{x}\underline{y}$.

[7] The term $(\overline{x} + \underline{x})(\overline{y} + \underline{y})$ is not exactly a single-digit multiplication as $\overline{x} + \underline{x}$ and $\overline{y} + \underline{y}$ are numbers between 0 and 18 and not between 0 and 9. As we'll see, it turns out that this does not make much of a difference, since when we use this algorithm recursively on $n$-digit numbers, this term will have at most $\lceil n/2 \rceil + 1$ digits, which is essentially half the number of digits as the original input.

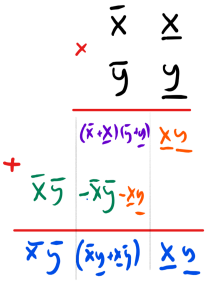

**Figure 2**: Karatsuba's multiplication algorithm illustrated for multiplying $x = 10\overline{x} + \underline{x}$ and $y = 10\overline{y} + \underline{y}$. We compute the three orange, green and purple products $\underline{x}\underline{y}$, $\overline{x}\overline{y}$ and $(\overline{x} + \underline{x})(\overline{y} + \underline{y})$ and then add and subtract them to obtain the result.

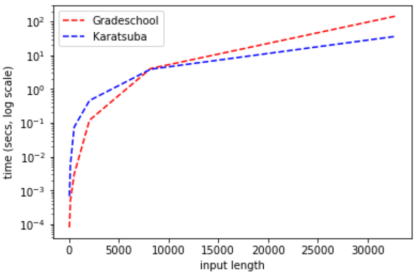


**Figure 3**: Running time of Karatsuba's algorithm vs. the grade-school algorithm. (Python implementation available online.) Note the existence of a "cutoff" length, where for sufficiently large inputs Karatsuba becomes more efficient than the grade-school algorithm. The precise cutoff location varies by implementation and platform details, but will always occur eventually.

the algorithm will indeed terminate. Now, since $x = 10^m\overline{x} + \underline{x}$ and $y = 10^m\overline{y} + \underline{y}$,

$$x \times y = 10^n\overline{x} \cdot \overline{y} + 10^m(\overline{xy} + \underline{xy}) + \underline{xy} \; . \tag{3}$$

Rearranging the terms we see that

$$x \times y = 10^n\overline{x} \cdot \overline{y} + 10^m \left[(\overline{x} + \underline{x})(\overline{y} + \underline{y}) - \underline{xy} - \overline{xy}\right] + \underline{xy} \; , \tag{4}$$

which equals $(10^n - 10^m) \cdot A + 10^m \cdot B + (1 - 10^m) \cdot C$, the value returned by the algorithm.

The key observation is that (4) reduces the task of computing the product of two $n$-digit numbers to computing *three* products of $m$-digit numbers where $m = \lfloor n/2 \rfloor + 1$ is roughly equal to $n/2$ (in fact, $m$ is always at most $n/2 + 1$). Specifically, we can compute $x \times y$ from the three products $\overline{xy}$, $\underline{xy}$ and $(\overline{x} + \underline{x})(\overline{y} + \underline{y}))$, using a constant number (in fact eight) of additions, subtractions, and multiplications by $10^n$ or $10^{\lfloor n/2 \rfloor}$. (Multiplication by a power of ten can be done very efficiently as it corresponds to simply shifting the digits.) Intuitively this means that as the number of digits *doubles*, the cost of performing a multiplication via Karatsuba's algorithm *triples* instead of quadrupling, as happens in the naive algorithm.

In particular, by iteratively repeating this reasoning, we can see that multiplying numbers of $n = 2^\ell$ digits via Karatsuba's Algorithm will cost about $3^\ell = n^{\log_2 3} \sim n^{1.585}$ operations. Exercise 0.4 shows that the number of single-digit operations that Karatsuba's algorithm uses for multiplying $n$ digit integers is at most $O(n^{\log_2 3})$ (see also Fig. 2).

[8] Recall that for a number $x$, $\lfloor x \rfloor$ is obtained by "rounding down" $x$ to the largest integer smaller or equal to $x$.

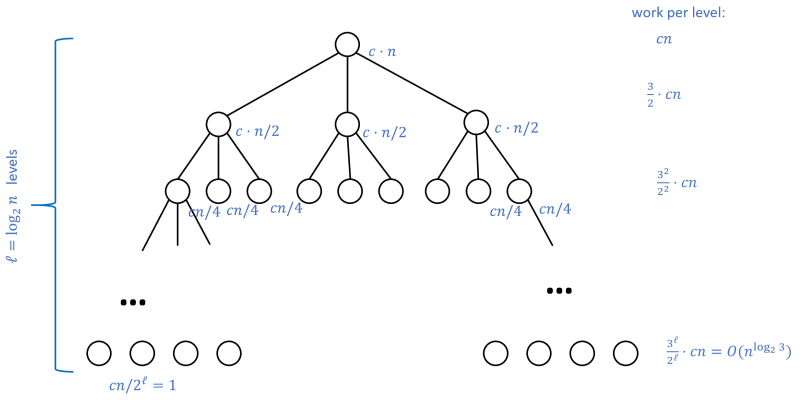


**Figure 4**: Karatsuba's algorithm reduces an $n$-bit multiplication to three $n/2$-bit multiplications, which in turn are reduced to nine $n/4$-bit multiplications and so on. We can represent the computational cost of all these multiplications in a 3-ary tree of depth $\log_2 n$, where at the root the extra cost is $cn$ operations, at the first level the extra cost is $c(n/2)$ operations, and at each of the $3^i$ nodes of level $i$, the extra cost is $c(n/2^i)$. The total cost is $cn\sum_{i=0}^{\log_2 n}(3/2)^i \leq 10cn^{\log_2 3}$ by the formula for summing a geometric series.

R

**Remark 0.5 — Ceilings, floors, and rounding.** One of the benefits of using Big-$O$ notation is that we can allow ourselves to be a little looser with issues such as rounding numbers. For example, the natural way to describe Karatsuba's algorithm's running time is via the following recursive equation

$$T(n) = 3T(n/2) + O(n) \ . \tag{5}$$

But if $n$ is not even then we cannot recursively invoke the algorithm on $n/2$-digit integers. Rather, the true recursive equation is

$$T(n) = 3T(\lfloor n/2 \rfloor + 1) + O(n) \ . \tag{6}$$

However, you can verify that the difference between (5) and (6) is insignificant when we are only concerned with "Big Oh analysis". These kinds of tricks work not just in the context of multiplication algorithms but in many other cases as well. Thus most of the time we can safely ignore "rounding issues" in the lengths of the input in the analysis of algorithms.

### 0.1.1 Beyond Karatsuba's algorithm

Toom and Cook extended in the ideas of Karatsuba to get even faster multiplication algorithms, but this was not the end of the line. In 1971, Schönhage and Strassen improved on the Toom-Cook algorithm using the *Fast Fourier Transform*; their idea was to somehow treat integers as "signals" and do the multiplication more efficiently by moving to the Fourier domain.[9] The latest asymptotic improvement was given by Fürer in 2007 (though it only starts beating the Schönhage-Strassen algorithm for truly astronomical numbers). Yet, despite all this progress, we still don't know whether or not there is an $O(n)$ time algorithm for multiplying two $n$ digit numbers!

[9] The *Fourier transform* is a central tool in mathematics and engineering, used in a great many applications. If you have not seen it yet, you will hopefully encounter it at some point in your studies.

R

**Remark 0.6 — Matrix Multiplication (advanced note).** (This book contains many "advanced" or "optional" notes and sections. These may assume background that not every student has, and can be safely skipped over as none of the future parts depends on them.)

Ideas similar to Karatsuba's can be used to speed up *matrix* multiplications as well. Matrices are a powerful way to represent linear equations and operations, widely used in a great many applications of scientific computing, graphics, machine learning, and many many more.

One of the basic operations one can do with two matrices is to *multiply* them. For example, if $x = \begin{pmatrix} x_{0,0} & x_{0,1} \\ x_{1,0} & x_{1,1} \end{pmatrix}$ and $y = \begin{pmatrix} y_{0,0} & y_{0,1} \\ y_{1,0} & y_{1,1} \end{pmatrix}$ then the product of $x$ and $y$ is the matrix $\begin{pmatrix} x_{0,0}y_{0,0} + x_{0,1}y_{1,0} & x_{0,0}y_{0,1} + x_{0,1}y_{1,1} \\ x_{1,0}y_{0,0} + x_{1,1}y_{1,0} & x_{1,0}y_{0,1} + x_{1,1}y_{1,1} \end{pmatrix}$. You can see that we can compute this matrix by *eight* products of numbers.

Now suppose that $n$ is even and $x$ and $y$ are a pair of $n \times n$ matrices which we can think of as each composed of four $(n/2) \times (n/2)$ blocks $x_{0,0}, x_{0,1}, x_{1,0}, x_{1,1}$ and $y_{0,0}, y_{0,1}, y_{1,0}, y_{1,1}$. Then the formula for the matrix product of $x$ and $y$ can be expressed in the same way as above, just replacing products $x_{a,b}y_{c,d}$ with *matrix* products, and addition with matrix addition. This means that we can use the formula above to give an algorithm that *doubles* the dimension of the matrices at the expense of increasing the number of operation by a factor of 8, which for $n = 2^\ell$ results in $8^\ell = n^3$ operations.

In 1969 Volker Strassen noted that we can compute the product of a pair of two-by-two matrices using only *seven* products of numbers by observing that each entry of the matrix $xy$ can be computed by adding and subtracting the following seven terms: $t_1 = (x_{0,0} + x_{1,1})(y_{0,0} + y_{1,1})$, $t_2 = (x_{0,0} + x_{1,1})y_{0,0}$, $t_3 = x_{0,0}(y_{0,1} - y_{1,1})$, $t_4 = x_{1,1}(y_{0,1} - y_{0,0})$, $t_5 = (x_{0,0} + x_{0,1})y_{1,1}$, $t_6 = (x_{1,0} - x_{0,0})(y_{0,0} + y_{0,1})$, $t_7 = (x_{0,1} - x_{1,1})(y_{1,0} + y_{1,1})$. Indeed, one can verify that $xy = \begin{pmatrix} t_1 + t_4 - t_5 + t_7 & t_3 + t_5 \\ t_2 + t_4 & t_1 + t_3 - t_2 + t_6 \end{pmatrix}$.

Using this observation, we can obtain an algorithm such that doubling the dimension of the matrices results in increasing the number of operations by a factor of 7, which means that for $n = 2^\ell$ the cost is $7^\ell = n^{\log_2 7} \sim n^{2.807}$. A long sequence of work has since improved this algorithm, and the current record has running time about $O(n^{2.373})$. However, unlike the case of integer multiplication, at the moment we don't know of any algorithm for matrix multiplication that runs in time linear or even close to linear in the size of the input matrices (e.g., an $O(n^2 polylog(n))$ time algorithm). People have tried to use group representations, which can be thought of as generalizations of the Fourier transform, to obtain faster algorithms, but this effort has not yet succeeded.

## 0.2 ALGORITHMS BEYOND ARITHMETIC

The quest for better algorithms is by no means restricted to arithmetical tasks such as adding, multiplying or solving equations. Many *graph algorithms*, including algorithms for finding paths, matchings, spanning trees, cuts, and flows, have been discovered in the last several decades, and this is still an intensive area of research. (For example, the last few years saw many advances in algorithms for the *maximum flow* problem, borne out of unexpected connections with electrical circuits and linear equation solvers.) These algorithms are being used not just for the "natural" applications of routing network traffic or GPS-based navigation, but also for applications as varied as drug discovery through searching for structures in gene-interaction graphs to computing risks from correlations in financial investments.

Google was founded based on the *PageRank* algorithm, which is an efficient algorithm to approximate the "principal eigenvector" of (a dampened version of) the adjacency matrix of the web graph. The *Akamai* company was founded based on a new data structure, known as *consistent hashing*, for a hash table where buckets are stored at different servers. The *backpropagation algorithm*, which computes partial derivatives of a neural network in $O(n)$ instead of $O(n^2)$ time, underlies many of the recent phenomenal successes of learning deep neural networks. Algorithms for solving linear equations under sparsity constraints, a concept known as *compressed sensing*, have been used to drastically reduce the amount and quality of data needed to analyze MRI images. This made a critical difference for MRI imaging of cancer tumors in children, where previously doctors needed to use anesthesia to suspend breath during the MRI exam, sometimes with dire consequences.

Even for classical questions, studied through the ages, new discoveries are still being made. For example, for the question of determining whether a given integer is prime or composite, which has been studied since the days of Pythagoras, efficient probabilistic algorithms were only discovered in the 1970s, while the first deterministic polynomial-time algorithm was only found in 2002. For the related problem of actually finding the factors of a composite number, new algorithms were found in the 1980s, and (as we'll see later in this course) discoveries in the 1990s raised the tantalizing prospect of obtaining faster algorithms through the use of quantum mechanical effects.

Despite all this progress, there are still many more questions than answers in the world of algorithms. For almost all natural problems, we do not know whether the current algorithm is the "best", or whether a significantly better one is still waiting to be discovered. As

we already saw, even for the classical problem of multiplying numbers we have not yet answered the age-old question of **"is multiplication harder than addition?"** .

But at least we now know the right way to *ask* it.

## 0.3 ON THE IMPORTANCE OF NEGATIVE RESULTS.

Finding better multiplication algorithms is undoubtedly a worthwhile endeavor. But why is it important to prove that such algorithms *don't* exist? What useful applications could possibly arise from an impossibility result?

One motivation is pure intellectual curiosity. After all, this is a question even Archimedes could have been excited about. Another reason to study impossibility results is that they correspond to the fundamental limits of our world. In other words, they are *laws of nature*. In physics, the impossibility of building a *perpetual motion machine* corresponds to the *law of conservation of energy*. The impossibility of building a heat engine beating Carnot's bound corresponds to the second law of thermodynamics, while the impossibility of faster-than-light information transmission is a cornerstone of special relativity.

In mathematics, while we all learned the solution for quadratic equations in high school, the impossibility of generalizing this to equations of degree five or more gave birth to *group theory*. Another example of an impossibility result comes from geometry. For two millennia, mathematicians tried to show that Euclid's fifth axiom or "postulate" could be derived from the first four. (This fifth postulate was known as the "parallel postulate", and roughly speaking it states that every line has a unique parallel line of each distance.) It was shown to be impossible using constructions of so called "non-Euclidean geometries", which turn out to be crucial for the theory of general relativity.[10]

In an analogous way, impossibility results for computation correspond to "computational laws of nature" that tell us about the fundamental limits of any information processing apparatus, whether based on silicon, neurons, or quantum particles.[11] Moreover, computer scientists have recently been finding creative approaches to *apply* computational limitations to achieve certain useful tasks. For example, much of modern Internet traffic is encrypted using the RSA encryption scheme, which relies on its security on the (conjectured) impossibility of efficiently factoring large integers. More recently, the Bitcoin system uses a digital analog of the "gold standard" where, instead of using a precious metal, new currency is obtained by "mining" solutions for computationally difficult problems.

[10] It is fine if you have not yet encountered many of the above examples of impossiblity results. I hope however that they spark your curiosity! See the "Biliographical Notes" section (Section 0.6) for some references.

[11] Indeed, some exciting recent research is focused on trying to use computational complexity to shed light on fundamental questions in physics such understanding black holes and reconciling general relativity with quantum mechanics.

**Lecture Recap**

- The history of algorithms goes back thousands of years; they have been essential much of human progress and these days form the basis of multi-billion dollar industries, as well as life-saving technologies.
- There is often more than one algorithm to achieve the same computational task. Finding a faster algorithm can often make a much bigger difference than improving computing hardware.
- Better algorithms and data structures don't just speed up calculations, but can yield new qualitative insights.
- One question we will study is to find out what is the *most efficient* algorithm for a given problem.
- To show that an algorithm is the most efficient one for a given problem, we need to be able to *prove* that it is *impossible* to solve the problem using a smaller amount of computational resources.

## 0.4 ROADMAP TO THE REST OF THIS BOOK

Often, when we try to solve a computational problem, whether it is solving a system of linear equations, finding the top eigenvector of a matrix, or trying to rank Internet search results, it is enough to use the "I know it when I see it" standard for describing algorithms. As long as we find some way to solve the problem, we are happy and don't care so much about formal descriptions of the algorithm. But when we want to answer a question such as "does there *exist* an algorithm to solve the problem $P$?" we need to be much more precise.

In particular, we will need to (**1**) define exactly what it means to solve $P$, and (**2**) define exactly what an algorithm is. Even (**1**) can sometimes be non-trivial but (**2**) is particularly challenging; it is not at all clear how (and even whether) we can encompass all potential ways to design algorithms. We will consider several simple *models of computation*, and argue that, despite their simplicity, they do capture all "reasonable" approaches to achieve computing, including all those that are currently used in modern computing devices.

Once we have these formal models of computation, we can try to obtain *impossibility results* for computational tasks, showing that some problems *can not be solved* (or perhaps can not be solved within the resources of our universe). Archimedes once said that given a fulcrum and a long enough lever, he could move the world. We will see how *reductions* allow us to leverage one hardness result into a slew of a great many others, illuminating the boundaries between

the computable and uncomputable (or tractable and intractable) problems.

Later in this course we will go back to examining our models of computation, and see how resources such as randomness or quantum entanglement could potentially change the power of our model. In the context of probabilistic algorithms, we will see a glimpse of how randomness has become an indispensable tool for understanding computation, information, and communication. We will also see how computational difficulty can be an asset rather than a hindrance, and be used for the "derandomization" of probabilistic algorithms. The same ideas also show up in *cryptography*, which has undergone not just a technological but also an intellectual revolution in the last few decades, much of it building on the foundations that we explore in this course.

Theoretical Computer Science is a vast topic, branching out and touching upon many scientific and engineering disciplines. This book provides a very partial (and biased) sample of this area. More than anything, I hope I will manage to "infect" you with at least some of my love for this field, which is inspired and enriched by the connection to practice, but is also deep and beautiful regardless of applications.

### 0.4.1 Dependencies between chapters

This book is divided into the following parts ,see Fig. 5.

- **Preliminaries:** Introduction, mathematical background, and representing objects as strings.
- **Part I: Finite computation:** Boolean circuits / straight-line programs. Universal gate sets, existence of a circuit for every function, representing circuits as strings and the universal circuit, and the counting lower bound.
- **Part II: Uniform computation:** Turing machines / programs with loops. Equivalence of models (including RAM machines and $\lambda$ calculus), universal Turing machine, uncomputable functions the Halting problem and Rice's Theorem, Gödel's incompleteness theorem, restricted models (regular and context free languages).
- **Part III: Efficient computation:** Definition of running time, time hierarchy theorem, **P** and **NP**, **NP** completeness, space bounded computation.
- **Part IV: Randomized computation:** Probability, randomized algorithms, **BPP**, amplification, **BPP** $\subseteq \mathbf{P}_{/poly}$, pseudrandom generators and derandomization.

- **Part V: Advanced topics:** Cryptography, proofs and algorithms (interactive and zero knowledge proofs, Curry-Howard correspondence), quantum computing.

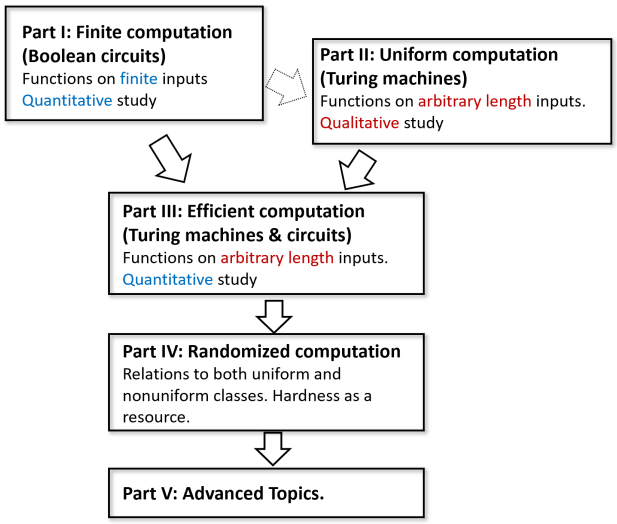


**Figure 5**: The dependency structure of the different parts. Part I introduces the model of Boolean circuits to study *finite functions* with an emphasis on *quantitative* questions (how many gates to compute a function). Part II introduces the model of Turing machines to study functions that have *unbounded input lengths* with an emphasis on *qualitative* questions (is this function computable or not). Much of Part II does not depend on Part I, as Turing machines can be used as the first computational model. Part III depends on both parts as it introduces a *quantitative* study of functions with unbounded input length. The more advancec parts IV (randomized computation) and V (advanced topics) rely on the material of Parts I, II and III.

The book largely proceeds in linear order, with each chapter building on the previous ones, with the following exceptions:

- Part II (Uniform Computation) does not have a strong dependency on Part I (Finite computation) and it should be possible to teach them in the reverse order.
- All chapters in Part V (Advanced topics) are independent of one another and can be covered in any order.
- Chapter 10 (Gödel's incompleteness theorem), Chapter 9 (Restricted computational models), and Chapter 16 (Space bounded computation), are not used in following chapters. Hence you can choose whether to cover or skip any of them them.

A course based on this book can use all of Parts I, II, and III (possibly skipping over some or all of Chapter 10, Chapter 9 or Chapter 16), and then either cover all or some of Part IV (randomized computation), and add a "sprinkling" of advanced topics from Part V based on student or instructor interest.

## 0.5 EXERCISES

**Exercise 0.1** Rank the significance of the following inventions in speeding up multiplication of large (that is 100-digit or more) numbers. That is, use "back of the envelope" estimates to order them in terms of the speedup factor they offered over the previous state of affairs.

a. Discovery of the grade-school digit by digit algorithm (improving upon repeated addition)

b. Discovery of Karatsuba's algorithm (improving upon the digit by digit algorithm)

c. Invention of modern electronic computers (improving upon calculations with pen and paper).

■

**Exercise 0.2** The 1977 Apple II personal computer had a processor speed of 1.023 Mhz or about $10^6$ operations per seconds. At the time of this writing the world's fastest supercomputer performs 93 "petaflops" ($10^{15}$ floating point operations per second) or about $10^{18}$ basic steps per second. For each one of the following running times (as a function of the input length $n$), compute for both computers how large an input they could handle in a week of computation, if they run an algorithm that has this running time:

a. $n$ operations.

b. $n^2$ operations.

c. $n \log n$ operations.

d. $2^n$ operations.

e. $n!$ operations.

■

**Exercise 0.3 — Usefulness of algorithmic non-existence.** In this chapter we mentioned several companies that were founded based on the discovery of new algorithms. Can you give an example for a company that was founded based on the *non existence* of an algorithm? See footnote for hint.[12]

■

**Exercise 0.4 — Analysis of Karatsuba's Algorithm.** a. Suppose that $T_1, T_2, T_3, \ldots$ is a sequence of numbers such that $T_2 \leq 10$ and for every $n$, $T_n \leq 3T_{\lfloor n/2 \rfloor + 1} + Cn$ for some $C \geq 1$. Prove that $T_n \leq 20Cn^{\log_2 3}$ for every $n > 2$.[13]

b. Prove that the number of single-digit operations that Karatsuba's algorithm takes to multiply two $n$ digit numbers is at most $1000n^{\log_2 3}$.

■

[12] As we will see in Chapter Chapter 20, almost any company relying on cryptography needs to assume the *non existence* of certain algorithms. In particular, RSA Security was founded based on the security of the RSA cryptosystem, which presumes the *non existence* of an efficient algorithm to compute the prime factorization of large integers.

[13] **Hint:** Use a proof by induction - suppose that this is true for all $n$'s from 1 to $m$ and prove that this is true also for $m + 1$.

**Exercise 0.5** Implement in the programming language of your choice functions `Gradeschool_multiply(x,y)` and `Karatsuba_multiply(x,y)` that take two arrays of digits `x` and `y` and return an array representing the product of `x` and `y` (where `x` is identified with the number `x[0]+10*x[1]+100*x[2]+...` etc..) using the grade-school algorithm and the Karatsuba algorithm respectively. At what number of digits does the Karatsuba algorithm beat the grade-school one?

■

**Exercise 0.6 — Matrix Multiplication (optional, advanced).** In this exercise, we show that if for some $\omega > 2$, we can write the product of two $k \times k$ real-valued matrices $A, B$ using at most $k^\omega$ multiplications, then we can multiply two $n \times n$ matrices in roughly $n^\omega$ time for every large enough $n$.

To make this precise, we need to make some notation that is unfortunately somewhat cumbersome. Assume that there is some $k \in \mathbb{N}$ and $m \leq k^\omega$ such that for every $k \times k$ matrices $A, B, C$ such that $C = AB$, we can write for every $i, j \in [k]$:

$$C_{i,j} = \sum_{\ell=0}^{m} \alpha_{i,j}^{\ell} f_\ell(A) g_\ell(B) \tag{7}$$

for some linear functions $f_0, \ldots, f_{m-1}, g_0, \ldots, g_{m-1} : \mathbb{R}^{n^2} \to \mathbb{R}$ and coefficients $\{\alpha_{i,j}^{\ell}\}_{i,j \in [k], \ell \in [m]}$. Prove that under this assumption for every $\epsilon > 0$, if $n$ is sufficiently large, then there is an algorithm that computes the product of two $n \times n$ matrices using at most $O(n^{\omega+\epsilon})$ arithmetic operations.[14]

■

[14] *Hint:* Start by showing this for the case that $n = k^t$ for some natural number $t$, in which case you can do so recursively by breaking the matrices into $k \times k$ blocks.

## 0.6 BIBLIOGRAPHICAL NOTES

For a brief overview of what we'll see in this book, you could do far worse than read Bernard Chazelle's wonderful essay on the Algorithm as an Idiom of modern science. The book of Moore and Mertens [MM11] gives a wonderful and comprehensive overview of the theory of computation, including much of the content discussed in this chapter and the rest of this book. Aaronson's book [Aar13] is another great read that touches upon many of the same themes.

Many of the algorithms we mention in this chapter are covered in algorithms textbooks such as those by Cormen, Leiserson, Rivert, and Stein [Cor+09], Kleinberg and Tardos [KT06], and Dasgupta, Papadimitriou and Vazirani [DPV08].

The story of Karatsuba's discovery of his multiplication algorithm is recounted by him in [Kar95]. As mentioned above, further improvements were made by Toom and Cook [Too63; Coo66], Schönhage

and Strassen [SS71], and Fürer [Für07]. These last papers crucially rely on the *Fast Fourier transform* algorithm. The interesting of the (re)discovery of this algorithm by John Tukey in the context of the cold war is recounted in [Coo87]. (We say re-discovery because it later turned out that the algorithm dates back to Gauss [HJB85].) The Fast Fourier Transform is covered in some of the books mentioned below, and there are also online available lectures such as Jeff Erickson's. See also this popular article by David Austin. Fast *matrix* multiplication was discovered by Strassen [Str69], and since then this has been an active area of research. [Blä13] is a recommended self-contained survey of this area.

The *Backpropagation* algorithm for fast differentiation of neural networks was invented by Werbos [Wer74]. The *Pagerank* algorithm was invented by Larry Page and Sergey Brin [Pag+99]. It is closely related to the *HITS* algorithm of Kleinberg [Kle99]. The *Akamai* company was founded based on the *consistent hashing* data structure described in [Kar+97]. *Compressed sensing* has a long history but two foundational papers are [CRT06; Don06]. [Lus+08] gives a survey of applications of compressed sensing to MRI; see also this popular article by Ellenberg [Ell10]. The deterministic polynomial-time algorithm for testing primality was given by Agrawal, Kayal, and Saxena [AKS04].

We alluded briefly to classical impossibility results in mathematics, including the impossibility of proving Euclid's fifth postulate from the other four, impossibility of trisecting an angle with a straightedge and compass and the impossibility of solving a quintic equation via radicals. A geometric proof of the impossibility of angle trisection (one of the three geometric problems of antiquity, going back to the ancient Greeks) is given in this blog post of Tao. The book of Mario Livio [Liv05] covers some of the background and ideas behind these impossibility results.