# Lecture 4

# Efficiency of algorithms

---

The development of algorithms (systematic recipes for solving problems) will be a theme that runs through the whole course. We'll be especially interested in saying rigorous-but-useful things about how much work an algorithm requires. Today's lecture introduces the standard vocabulary for such discussions and illustrates it with several examples including the Greedy Colouring Algorithm, the standard algorithm for multiplication of two square $n \times n$ matrices and the problem of testing whether a number is prime.

**Reading:**
The material in the first part of today's lecture comes from Section 2.5 of

> Dieter Jungnickel (2013), *Graphs, Networks and Algorithms*, 4th edition (available online via SpringerLink),

though the discussion there mentions some graph-theoretic matters that we have not yet covered.

---

## 4.1 Introduction

The aim of today's lecture is to develop some convenient terms for the way in which the amount of work required to solve a problem with a particular algorithm depends on the "size" of the input. Note that this is a property of the *algorithm*: it may be possible to find a better approach that solves the problem with less work. If we were being very careful about these ideas we would make a distinction between the quantities I'll introduce below, which, strictly speaking, describe the *time complexity* of an algorithm, and a separate set of bounds that say how much computer memory (or how many sheets of paper, if we're working by hand) an algorithm requires. This latter quantity is called the *space complexity* of the algorithm, but we won't worry about that much in this course.

To get an idea of the kinds of results we're aiming for, recall the standard algorithm for pencil-and-paper addition of two numbers (write one number above the other, draw a line underneath ...).

$$\begin{array}{r} 2011 \\ 21 \\ \hline 2032 \end{array}$$

The basic step in this process is the addition of two decimal digits, for example, in the first column, $1 + 1 = 2$. The calculation here thus requires two basic steps.

More generally, the number of basic steps required to perform an addition $a + b$ using the pencil-and-paper algorithm depends on the numbers of decimal digits in $a$ and $b$. The following proposition (whose proof is left to the reader) explains why it's thus natural to think of $\log_{10}(a)$ and $\log_{10}(b)$ as the sizes of the inputs to the addition algorithm.

**Definition 4.1** (Floor and ceiling)**.** *For a real number $x \in \mathbb{R}$, define $\lfloor x \rfloor$, which is read as "floor of $x$", to be the greatest integer less-than-or-equal-to $x$. Similarly, define $\lceil x \rceil$, "ceiling of $x$", to be the least integer greater-than-or-equal-to $x$. In more conventional notation these functions are given by*

$$\lfloor x \rfloor = \max \{k \in \mathbb{Z} \,|\, k \leq x\} \quad \textit{and} \quad \lceil x \rceil = \min \{k \in \mathbb{Z} \,|\, k \geq x\}\,.$$

**Proposition 4.2** (Logs and length in decimal digits)**.** *The decimal representation of a number $n > 0 \in \mathbb{N}$ has exactly $d = 1 + \lfloor \log_{10}(n) \rfloor$ decimal digits.*

In light of these results, one might hope to say something along the lines of

> *The pencil-and-paper addition algorithm computes the sum $a + b$ in $1 + \min(\lfloor \log_{10}(a) \rfloor, \lfloor \log_{10}(b) \rfloor)$ steps.*

This is a bit of a mouthful and, worse, isn't even right. Quick-witted readers will have noticed that we haven't taken proper account of carried digits. The example above didn't involve any, but if instead we had computed

$$\begin{array}{r} 1959 \\ 21 \\ \hline 1980 \end{array}$$

we would have carried a 1 from the first column to the second and so would have needed to do three basic steps: $9 + 1$, $1 + 5$, and $6 + 2$.

In general, if the larger of $a$ and $b$ has $d$ decimal digits then computing $a + b$ could require as many as $d - 1$ carrying additions. That means our statement above should be replaced by something like

> *The number of steps $N$ required for the pencil-and-paper addition algorithm to compute the sum $a + b$ satisfies the following bounds:*
>
> $$1 + \min(\lfloor \log_{10}(a) \rfloor, \lfloor \log_{10}(b)) \rfloor) \leq N \leq 1 + \lfloor \log_{10}(a) \rfloor + \lfloor \log_{10}(b) \rfloor$$

This introduces an important theme: knowing the size of the input isn't always enough to determine exactly how long a computation will take, but it may be possible to place bounds on the running-time.

The statements above are rather fiddly and not especially useful. In practice, people want such estimates so they can decide whether a problem is do-able at all. They want to know whether, say, given a calculator that can add two 5-digit numbers in one second[1], it would be possible to work out the University of Manchester's payroll in less than a month. For these sorts of questions[2] one doesn't want the cumbersome, though precise sorts of statements formulated above, but rather something semi-quantitative along the lines of:

> *If we measure the size of the inputs to the pencil-and-paper addition algorithm in terms of the number of digits, then if we double the size of the input, the amount of work required to get the answer also doubles.*

The remainder of today's lecture will develop a rigorous framework that we can use to make such statements.

## 4.2 Examples and issues

The three main examples in the rest of the lecture will be

- Greedy colouring of a graph $G(V, E)$ with $|V|$ vertices and $|E|$ edges.
- Multiplication of two $n \times n$ matrices $A$ and $B$.
- Testing whether a number $n \in \mathbb{N}$ is prime by trial division.

In the rest of this section I'll discuss the associated algorithms briefly, with an eye to answering three key questions:

(1) What are the details of the algorithm and what should we regard as the basic step?

(2) How should we measure the size of the input?

(3) What kinds of estimates can we hope to make?

### 4.2.1 Greedy colouring

Algorithm 3.7 constructs a colouring for a graph $G(V, E)$ by examining, in turn, the adjacency lists of each of the vertices in $V = \{v_1, \ldots, v_n\}$. If we take as our

[1] Up until the mid-1940's, a calculator was a *person* and so the speed mentioned here was not unrealistic. Although modern computing machinery doesn't work with decimal representations and can perform basic operations in tiny fractions of a second, similar reasoning still allows one to determine the limits of practical computability.

[2] Estimates similar to the ones we've done here bear on much less contrived questions such as: if there are processors that can do arithmetic on two 500-digit numbers in a nanosecond, how many of them would GCHQ need to buy if they want to be able to crack a 4096-bit RSA cryptosystem within an hour?

basic step the process of looking at neighbour's colour, then the algorithm requires a number of steps given by

$$\sum_{v \in V} |A_v| \;=\; \sum_{v \in V} \deg(v) \;=\; 2|E| \tag{4.1}$$

where the final equality follows from Theorem 1.8, the Handshaking Lemma. This suggests that we should measure the size of the problem in terms of the number of edges.

### 4.2.2 Matrix multiplication

If $A$ and $B$ are square, $n \times n$ matrices then the $i, j$-th entry in the product $AB$ is given by:

$$(AB)_{i,j} = \sum_{k=1}^{n} A_{ik} B_{kj}.$$

Here we'll measure the size of the problem with $n$, the number of rows in the matrices, and take as our basic steps the arithmetic operations, addition and multiplication of two real numbers. The formula above then makes it easy to see that it takes $n$ multiplications and $(n-1)$ additions to compute a single entry in the product matrix and so, as there are $n^2$ such entries, we need

$$n^2 \left(n + (n-1)\right) = 2n^3 - n^2 \tag{4.2}$$

total arithmetic operations to compute the whole product.

Notice that I have not included any measure of the magnitude of the matrix entries in our characterisation of the size of the problem. I did this because (a) it agrees with the usual practice among numerical analysts, who typically analyse algorithms designed to work with numbers whose machine representations have a fixed size and (b) I wanted to emphasise that an efficiency estimate depends in detail on how one chooses to measure size of the inputs. The final example involves a problem and algorithm where it does make sense to think about the magnitude of the input.

### 4.2.3 Primality testing and worst-case estimates

It's easy to prove by contradiction the following proposition.

**Proposition 4.3** (Smallest divisor of a composite number)**.** *If $n \in \mathbb{N}$ is composite (that is, not a prime), then it has a divisor $b$ that satisfies $b \leq \sqrt{n}$.*

We can thus test whether a number is prime with the following simple algorithm:

**Algorithm 4.4** (Primality testing via trial division)**.**
*Given a natural number $n \in \mathbb{N}$, determine whether it is prime.*

*(1) For each $b \in \mathbb{N}$ in the range $2 \leq b \leq \sqrt{n}$ {*

*(2) Ask: does $b$ divide $n$?*
*If Yes, report that $n$ is composite.*
*If No, continue to the next value of $b$.*

*(3) }*

*(4) If no divisors were found, report that $n$ is prime.*

This problem is more subtle than the previous examples in a couple of ways. A natural candidate for the basic step here is the computation of $(n \bmod b)$, which answers the question "Does $b$ divide $n$?". But the kinds of numbers whose primality one wants to test in, for example, cryptographic applications are large and so we might want take account of the magnitude of $n$ in our measure of the input size. If we compute $(n \bmod b)$ with the standard long-division algorithm, the amount of work required for a basic step will itself depend on the number of digits in $n$ and so, as in our analysis of the pencil-and-paper addition algorithm, it'll prove convenient to measure the size of the input with $d = \log_{10}(n)$, which is approximately the number of decimal digits in $n$.

A further subtlety is that because the algorithm reports an answer as soon as it finds a factor, the amount of work required varies wildly, even among $n$ with the same number of digits. For example, half of all 100-digit numbers are even and so will be revealed as composite by the very first value of $b$ we'll try. Primes, on the other hand, will require $\lfloor\sqrt{n}\rfloor - 1$ tests. A standard way to deal with this second issue is to make estimates about the *worst case* efficiency: in this case, that's the running-time required for primes. A much harder approach is to make an estimate of the *average case* running-time obtained by averaging over all inputs with a given size.

## 4.3 Bounds on asymptotic growth

As the introduction hinted, were really after statements about how quickly the amount of work increases as the size of the problem does. The following definitions provide a very convenient language in which to formulate such statements.

**Definition 4.5** (Bounds on asymptotic growth)**.** *The rate of growth of a function $f : \mathbb{N} \to \mathbb{R}^+$ is often characterised in terms of some simpler function $g : \mathbb{N} \to \mathbb{R}^+$ in the following ways*

- $f(n) = O(g(n))$ *if* $\exists\, c_1 > 0$ *such that, for all sufficiently large $n$, $f(n) \leq c_1 g(n)$;*
- $f(n) = \Omega(g(n))$ *if* $\exists\, c_2 > 0$ *such that, for all sufficiently large $n$, $f(n) \geq c_2 g(n)$;*
- $f(n) = \Theta(g(n))$ *if $f(n) = O(g(n))$ and $f(n) = \Omega(g(n))$.*

Notice that the definitions of $f(n) = O(g(n))$ and $f(n) = \Omega(g(n))$ include the phrase "for all sufficiently large $n$". This is equivalent to saying, for example,

> $f(n) = \Omega(g(n))$ if there exist some $c_1 > 0$ and $N_1 \geq 0$ such that for all $n \geq N_1$, $f(n) \leq c_1 g(n)$.

The point is that the definitions are only concerned with *asymptotic* bounds—they're all about the limit of large $n$.

## 4.4 Analysing the examples

Here I'll apply the definitions from the previous section to our three examples. In practice people are mainly concerned with the $O$-behaviour of an algorithm. That is, they are mainly interested in getting an asymptotic *upper* bound on the amount of work required. This makes sense when one doesn't know anything special about the inputs and is thinking about buying computer hardware, but as an exercise I'll obtain $\Theta$-type bounds where possible.

### 4.4.1 Greedy colouring

**Proposition 4.6** (Greedy colouring)**.** *If we take the process of checking a neighbour's colour as the basic step, greedy colouring requires $\Theta(|E|)$ basic steps.*

We've already argued that the number of basic steps is $2|E|$, so if we take $c_1 = 3$ and $c_2 = 1$ we have, for all graphs, that the number of operations $f(|E|) = 2|E|$ required for the greedy colouring algorithm satisfies

$$c_2|E| \leq f(|E|) \leq c_1|E| \qquad \text{or} \qquad |E| \leq 2|E| \leq 3|E|$$

and so the algorithm is both $O(|E|)$ and $\Omega(|E|)$ and hence is $\Theta(|E|)$.

### 4.4.2 Matrix multiplication

**Proposition 4.7** (Matrix multiplication)**.** *If we characterise the size of the inputs with $n$, the number of rows in the matrices, and take as our basic step the arithmetic operations of addition and multiplication of two matrix entries, then matrix multiplication requires $\Theta(n^3)$ basic steps.*

We argued in Section 4.2.2 that the number of arithmetic operations required to multiply a pair of $n \times n$ matrices is $f(n) = 2n^3 - n^2$. It's not hard to see that for $n \geq 1$,

$$n^3 \leq 2n^3 - n^2 \qquad \text{or equivalently} \qquad 0 \leq n^3 - n^2$$

and so $f(n) = \Omega(n^3)$. Further, it's also easy to see that for $n \geq 1$ we have

$$2n^3 - n^2 \leq 3n^3 \qquad \text{or equivalently} \qquad -n^3 - n^2 \leq 0,$$

and so we also have that $f(n) = O(n^3)$. Combining these bounds, we've established that

$$f(n) = \Theta(n^3),$$

which is a special case of a more general result. One can prove—see the Problem Sets—that if $f : \mathbb{N} \to \mathbb{R}^+$ is a polynomial in $n$ of degree $k$, then $f = \Theta(n^k)$. Algorithms that are $O(n^k)$ for some $k \in \mathbb{R}$ are often called *polynomial time algorithms*.

### 4.4.3 Primality testing via trial division

**Proposition 4.8** (Primality testing)**.** *If we characterise the size of the input with $d = \log_{10}(n)$ (essentially the number of decimal digits in $n$) and take as our basic step the computation of $n \bmod b$ for a number $2 \leq b \leq \sqrt{n}$, then primality testing via trial division requires $O(10^{d/2})$ steps.*

In the most demanding cases, when $n$ is actually prime, we need to compute $n \bmod b$ for each integer $b$ in the range $2 \leq b \leq \sqrt{n}$. This means we need to do $\lfloor \sqrt{n} \rfloor - 1$ basic steps and so the number of steps $f(d)$ required by the algorithm satisfies

$$\begin{aligned} f(d) &\leq \lfloor \sqrt{n} \rfloor - 1 \\ &\leq \lfloor \sqrt{n} \rfloor \\ &\leq \sqrt{n}. \end{aligned}$$

To get the righthand side in terms of the problem size $d = \lfloor \log_{10}(n) \rfloor + 1$, note that, for all $x \in \mathbb{R}$, $x < \lfloor x \rfloor + 1$ and so

$$\log_{10}(n) < d.$$

Then

$$\sqrt{n} \;=\; n^{1/2} \;=\; \left(10^{\log_{10}(n)}\right)^{1/2}$$

which implies that

$$\begin{aligned} f(d) &\leq \sqrt{n} \\ &\leq \left(10^{\log_{10}(n)}\right)^{1/2} \\ &\leq \left(10^{d}\right)^{1/2} \\ &\leq 10^{d/2}, \end{aligned}$$

and thus that primality testing via trial division is $O(10^{d/2})$.

Such algorithms are often called *exponential-time* or just *exponential* and they are generally regarded as impractical for, as one increases $d$, the computational requirements can jump abruptly from something modest and doable to something impossible. Further, we haven't yet taken any account of the way the sizes of the numbers $n$ and $b$ affect the amount of work required for the basic step. If we were to do so—if, say, we choose single-digit arithmetic operations as our basic steps—the bound on the operation count would only grow larger: trial division is *not* a feasible way to test large numbers for primality.

## 4.5 Afterword

I chose the algorithms discussed above for simplicity, but they are not necessarily the best known ways to solve the problems. I also simplified the analysis by judicious choice of problem-size measurement and basic step. For example, in practical matrix multiplication problems the multiplication of two matrix elements is more computationally expensive than addition of two products, to the extent that people normally just ignore the additions and try to estimate the number of multiplications. The standard algorithm is still $\Theta(n^3)$, but more efficient algorithms are known. The basic idea is related to a clever observation about the number of multiplications required to compute the product of two complex numbers

$$(a + ib) \times (c + id) = (ac - bd) + i(ad + bc). \tag{4.3}$$

The most straightforward approach requires us to compute the four products $ac$, $bd$, $ad$ and $bc$. But Gauss noticed that one can instead compute just three products, $ac$, $bd$ and

$$q = (a + b)(c + d) = ac + ad + bc + bd,$$

and then use the relation

$$(ad + bc) = q - ac - bd$$

to compute the imaginary part of the product in Eqn. (4.3). In 1969 Volker Strassen discovered a similar trick whose simplest application allows one to compute the product of two $2 \times 2$ matrices with only 7 multiplications, as opposed to the 8 that the standard algorithm requires. Building on this observation, he found an algorithm that can compute all the entries in the product of two $n \times n$ matrices using only $O(n^{\log_2(7)}) \approx O(n^{2.807})$ multiplications[3].

More spectacularly, it turns out that there is a polynomial-time algorithm for primality testing. It was discovered in the early years of this century by Agrawal, Kayal and Saxena (often shortened to AKS)[4]. This is particularly cheering in that two of the authors, Kayal and Saxena, were undergraduate project students when they did this work.

[3] I learned about Strassen's work in a previous edition of William H. Press, Saul A. Teukolsky, William T. Vetterling and Brian P. Flannery (2007), *Numerical Recipes in C++*, 3rd edition, CUP, Cambridge. ISBN: 978-0-521-88068-8, which is very readable, but for a quick overview of the area you might want to look at Sara Robinson (2005), Toward an optimal algorithm for matrix multiplication, *SIAM News*, **38**(9).

[4] See: Manindra Agrawal, Neeraj Kayal and Nitin Saxena (2004), PRIMES is in P, *Annals of Mathematics*, **160**(2):781–793. DOI: 10.4007/annals.2004.160.781. The original AKS paper is quite approachable, but an even more reader-friendly treatment of their proof appears in Andrew Granville (2005), It is easy to determine whether a given integer is prime, *Bulletin of the AMS*, **42**:3–38. DOI: 10.1090/S0273-0979-04-01037-7.