Computational complexity is the mathematical study of efficiency. It is concerned with identifying models of efficient computation and understanding their power, their limitations, and their relationships.

In the first half of the 20th century the primary discipline concerned with computation was computability theory. The main insight of computability is that some problems like deciding whether a piece of code is eventually going to terminate are unsolvable by computer programs regardless of the resources available. By the 1960s it was realized that leaving these undecidable problems aside, there are a whole lot of other problems that are in principle solvable by methods like brute force, but for which even the best known algorithms take an enormous amount of time on reasonably sized inputs.

These insights led to the development to a theory of efficient computation, which turned out relevant not only for classifying computations, but also enabled the development of whole new areas of computer science such as cryptography, learning theory, verification, and computational game theory. Many basic insights in all these areas turn out to be about efficiency. Another subject at the forefront of scientific research today whose study was largely driven by questions of efficiency is quantum computing.

My aim in this course is to highlight those concepts and methods in computational complexity which I believe have wider significance. These should give you the ability to "see" efficient computation everywhere in the world and in your research, identify suitable models, and reason about them.

To get a sense of what is an efficient model of computation, what we might want to know about it, and what kinds of methods are "legitimate" for answering these questions, let's start with one of the simplest examples: the decision tree. But before that we need to introduce and motivate some conventions for describing computational problems.

1 Computational problems

Many of the computational problems we may be interested in can be described as functions. The function takes as its input a data item or a sequence of items as they may appear, say, in a computer's memory, hard drive, or "on the cloud". Its answer is the outcome of the computation (which we will assume always terminates) in a similar format. For example, the problem "What is the fastest route from Carleton to uOttawa?" might have maps and bus schedules as its input and your itinerary as its output.

Data items are usually described by bit sequences. Not only are bits the storage and processing unit of choice for most general-purpose computers, but even if this is not the case it is usually easy to convert whatever representation the data was stored in into bit representation. This is not where the real computational difficulty lies.[1]

We usually represent computational problems as functions that map a sequence of input bits into one or more output bits. Even in the case when the inputs and outputs are objects of another type we will think of them as being represented by sequences of bits. For example, if we study the problem of finding the prime factorization of a positive integer n, then n is given in binary, and its prime factorization is also described as a sequence of bits, one for each prime factor, with a suitable convention for representing the whole sequence as a single bit string.

A *computational problem* is a function whose domain is one set of bit sequences and whose range is another set of bit sequences. Should the domain and range be finite or infinite? This choice turns out to be surprisingly important. Let us try to motivate it by some examples.

When we talk about algorithms for "finding a shortest path" or "factoring an integer" in the abstract, we would like our algorithm to work in principle for all inputs, so an infinite domain and range seem better suited for such study. You may object, however, that in practice we are unlikely to ever see any

[1] In some parts of computational complexity (e.g., arithmetic complexity) it is more natural to work with other representations.

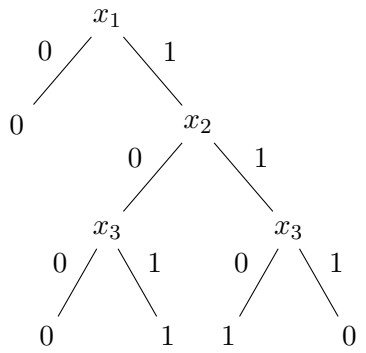
input an output that is more than say 2^{500} bits long, so shouldn't it be enough to restrict our domain and range to the sets $\{0,1\}^{2^{500}}$? While this is a reasonable objection, it is difficult to imagine anything interesting about these problems that happens only for inputs and outputs that are at most 2^{500} bits long; any algorithm designed for inputs that are 2^{500} bits long should in principle also work for arbitrarily long inputs. It is more natural to represent the domain and/or range of such problems as infinite sets, in which case the problem itself is a function from $\{0,1\}^*$ to $\{0,1\}^*$, the set of all possible bit sequences. The part of complexity theory that studies such problems is called (for reasons to be explained later) *uniform complexity*. The study of efficient algorithms, and also proofs, is more natural in the setting of uniform complexity.

On the other hand, a cryptographic function like AES-256 is an algorithm that may only take inputs that are 256 bits long and always produces outputs that are 256 bits long. It is a specific design that is believed to have certain cryptographic properties and is simply not equipped to handle inputs or produce outputs of a different length. In (applied) cryptography, it is common to fix the size of the problem to be solved (e.g., "exchange a 1024-bit key") and then design an algorithm for a problem of this size. We model such problems as functions $f\colon \{0,1\}^n \to \{0,1\}^m$ for some *a priori* fixed lengths n and m. The part of complexity theory that studies them is called *non-uniform complexity* or *circuit complexity*. Apart from cryptography, non-uniform computational models are also more common in computational learning theory. For example, if we want to train a neural network to recognize shapes, we usually decide in advance on the size of the network and then run our learning algorithm of choice to calculate its relevant parameters.

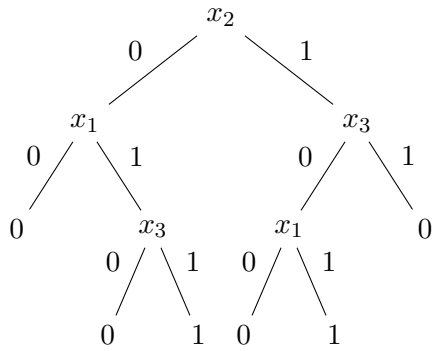
A *decision problem* is a computational problem with a yes/no answer, like "is graph G connected?". Such problems are represented by functions $f\colon \{0,1\}^* \to \{0,1\}$ and $f\colon \{0,1\}^n \to \{0,1\}$ in uniform and non-uniform complexity, respectively.

2 Decision trees

A *decision tree* for inputs of length n is a rooted binary tree whose vertices and edges are labeled as follows. Each of its internal nodes is labeled by one of the variables $x_1, \ldots, x_n$, and its two outgoing edges are labeled by the values 0 and 1 respectively. Each leaf is labeled by a 0 or by a 1. Here is an example:

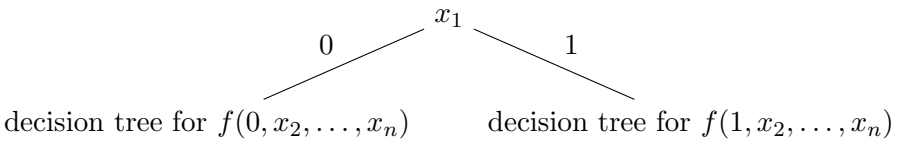


A decision tree computes a function $f\colon \{0,1\}^n \to \{0,1\}$ in a natural way: Query the variable at the root, follow the edge labeled by its value, and continue until a leaf is reached, then output its value. The above decision tree computes the function x_1 AND (x_2 XOR x_3). The following decision tree computes the same function:



The first decision tree is in some sense preferable than the second one as it is smaller. This is our first example of a *complexity measure*: The *size* of a decision tree is its number of leaves.[2]

It is not difficult to see that *any* function $f\colon \{0,1\}^n \to \{0,1\}$ can be computed by a sufficiently large decision tree like this:



The size of this decision tree is 2^n. Can we do much better?

Shannon's counting argument

We will now show that in general, the answer is "no":

Theorem 1 (Shannon's theorem for decision trees)**.** *For all n there exists a function $f\colon \{0,1\}^n \to \{0,1\}$ that requires decision tree size at least $2^n/4n$.*

Proof. First, we show that the number of decision trees of size s with n variables is at most $(n \cdot s^2)^{s-1}$. A tree of size s has $s-1$ internal nodes. To specify a decision tree it is sufficient to describe each internal node's label and the identities if its left and right children (if they exist). There are n possibilities for each label and at most s possibilities for the identity of each child (any one of the other $s-2$ internal nodes or the constants 0/1 in case of a leaf), which gives a total of $(n \cdot s^2)^{s-1}$ choices.

On the other hand, the number of functions from $\{0,1\}^n$ to $\{0,1\}$ is 2^{2^n}. If 2^{2^n} were larger than $(n \cdot s^2)^{s-1}$, or equivalently if 2^n were larger than $(s-1)\log(n \cdot s^2)$, then there would be at least one function that is not computed by any decision tree of size s. You can verify that this is the case when $s = 2^n/2n$. □

This type of proof, invented by Shannon over 70 years ago, is called a counting argument. It has several appealing features. First, it applies to virtually any non-uniform model of computation that one can think of. Indeed, the proof uses very little about decision trees.[3] Second, it is quite insensitive to details: If I didn't distinguish between internal nodes and leaves the end result would have been almost the same. Third, there is a variation of the counting argument which not only tells us that there exists a "hard" function for decision trees, but also that a vast majority of such functions are hard:

Theorem 2. *For all n the probability that a random function $F\colon \{0,1\}^n \to \{0,1\}$ can be computed by a decision tree of size $2^n/4n$ is at most $2^{-2^{n-1}}$.*

[2] We could have also defined size as number of nodes, which is twice the number of leaves minus one. Working with leaves will be more convenient for us.

[3] In fact, for decision trees the count can be improved to prove a better bound of $\Omega(2^n/\log n)$.

Proof. To bound the probability that a random function is computable by some decision tree of size s, we apply a union bound:

$$\Pr_F[\text{some decision tree of size } s \text{ computes } F] \leq \sum_T \Pr[T \text{ computes } F]$$

where the summation ranges over all decision trees T of size s. The probability of any single such tree computing F is exactly 2^{-2^n}, as the value of the function computed by T must match the value of a random function at all points. From the proof of Theorem 1 the number of decision trees of size s is at most $(n \cdot s^2)^{s-1}$, so the expression on the right is at most $(n \cdot s^2)^{s-1} \cdot 2^{-2^n} = 2^{(s-1)\log(n \cdot s^2) - 2^n}$. When $s = 2^n/4n$, $(2s-1)\log(n \cdot (2s)^2)$ is at most 2^{n-1} by a very similar calculation as in the proof of Theorem 1 and we get the desired result. □

The main drawback of the counting argument is that it does not give us hold of an *explicit* function f that is hard for decision trees. Intuitively, "explicit" means that despite the existence of f with the desired property, we are at a loss when it comes to "writing it down". The word "explicit" has a precise meaning in complexity theory, and we will define it properly once we become more comfortable with complexity-centric ways of thinking about computation.

An explicit hard function

The *parity function* on n bits is the function

$$PARITY(x_1, x_2, \ldots, x_n) = x_1 \text{ XOR } x_2 \text{ XOR } \cdots \text{ XOR } x_n.$$

It takes value zero when an even number of its inputs are ones, and value one otherwise.

Theorem 3. *$PARITY$ requires decision tree size 2^n.*

Proof. Let T be any decision tree for $PARITY$. We claim that T cannot contain any path of length strictly less than n: Regardless of the values that the fewer than n inputs along this path take, the value of the $PARITY$ function is undetermined after reading them, so the leaf of the path cannot be labeled either 0 or 1. As all n-bit edge label prefixes along paths must all lead to different leaves, T must have at least 2^n leaves. □

Theorem 3 is in some sense more satisfying than Theorem 1: Here is an explicit function, one that we can write down and calculate at will, that cannot be computed by small decision trees. (The lower bound in Theorem 3 also better than the one in Theorem 1, but that is an unusual feature of this specific example.) What more could be possibly want? Well, one drawback of this argument is that it is specifically tailored to the $PARITY$ function. What if we want to know, say, the decision tree size of some other function, like $MAJORITY$ (assuming n is odd)?

$$MAJORITY(x_1, \ldots, x_n) = \begin{cases} 1, & \text{if the input has more 1s than 0s,} \\ 0, & \text{if the input has more 1s than 1s.} \end{cases}$$

A more general approach for proving lower bounds on decision tree size is to identify a relevant property that all small decision trees satisfy, but the "hard" function of interest does not satisfy. In the case of the PARITY function, the crucial property is that no assignment to fewer than n inputs determines the value of the function. Let us give this property a name: Say f is *k-undetermined* if no partial assignment to any k of the inputs fixes the value of the function. Thus $PARITY$ is $(n-1)$-undetermined, while $MAJORITY$ is $(n-1)/2$-undetermined. The proof of Theorem 3 readily generalizes to the following statement:

Theorem 4. *Any k-undetermined function requires decision tree size 2^{k+1}.*

As a corollary, $MAJORITY$ of n inputs requires decision tree size $2^{(n+1)/2}$. It is not difficult to prove an even better bound of $2^{n-o(n)}$. Instead of doing this, let us introduce a different method that will yield a weaker bound for $MAJORITY$ but will enable us to analyze more powerful models of computation in the next lecture.

3 Random restrictions

A *restriction* of the variables x_1 up to x_n is a partial assignment that gives each of the variables the value 0, the value 1, or leaves it unassigned. Such a restriction can be succinctly represented as a string ρ in $\{0,1,\star\}^n$, where a $\star$ means that the corresponding variable is unassigned; for example, $01\star0\star$ is the restriction $x_1 = 0$, $x_2 = 1$, $x_4 = 0$, x_3 and x_5 unassigned.

Given a function $f\colon \{0,1\}^n \to \{0,1\}$ and a restriction $\rho \in \{0,1,\star\}^n$, the restricted function $f|_\rho(x)$ is obtained by substituting the variables assigned by ρ with the corresponding constants, for example

$$f|_{01\star0\star}(x_3, x_5) = f(0, 1, x_3, 0, x_5).$$

The input size of $f|_\rho$ equals the number of stars in ρ.

A *δ-random restriction* is a restriction that sets each coordinate independently to $\star$ with probability $\delta \in [0,1]$ and 0 and 1 with probability $(1-\delta)/2$ each. Some very simple functions, like the AND of n bits, are typically "killed" by random restrictions when n is large: If any of the input is restricted to zero the AND function vanishes, so the probability that AND does *not* vanish is at most $(1/2 + \delta/2)^n$, which is exponentially small in n for say $\delta = 1/10$. By the same reasoning, the OR of n bits is also typically killed by a random restriction, and so are ORs and ANDs of *literals* (possibly negated variables), such as

$$x_1 \text{ AND } (\text{NOT } x_2) \text{ AND } (\text{NOT } x_3) \text{ AND } x_4 \text{ AND } (\text{NOT } x_5) \cdots \text{ AND } x_n.$$

All these functions have decision trees of size n. Is it the case that random restrictions kill decision trees as well? Not always so, but they do simplify them greatly:

Theorem 5. *For every f computable by a decision tree of size s and a δ-random restriction ρ, $f|_\rho$ has a decision tree of depth less than d except with probability at most $s \cdot (1/2 + \delta/2)^d$.*

The *depth* of a decision tree is the maximum length of a path from root to leaf. This theorem reduces the problem of showing that f has no small decision tree to showing that the related function $f|_\rho$ has no *shallow* decision tree (with sufficient probability). The upside is that depth is easier to analyze than size.

For example, it is immediate that the decision tree depth of $MAJORITY$ on n bits is at least $(n+1)/2$: any shorter root-to-leaf path can query at most half the inputs, which is insufficient to determine their majority value. In fact, the decision tree depth must be n: If the inputs "seen" by the decision tree are a 0, then a 1, then a 0, then a 1, and so on, then the majority cannot be determined until all n bits are read.

To apply this theorem, we need to analyze the decision tree depth not of $MAJORITY$ itself, but of its random restrictions. Before we do so let's prove Theorem 5.

Proof of Theorem 5. Let T be a decision tree of size s for f and p be a root-to-leaf path in T. We will assume, without loss of generality, that all variables that appear along p are distinct. We will say that p is killed by ρ if there exists a variable along p that ρ fixes to a value different from the one on its outgoing edge. For example, the root-to-leaf path

$$x_2 \xrightarrow{0} x_4 \xrightarrow{0} x_1 \xrightarrow{1} 0$$

is killed by the restriction $0\star00$ as the value of x_1 in ρ is inconsistent with the value on its outgoing edge. After substituting the non-starred variables in ρ by their values in T and removing all the paths that are killed, we obtain a decision tree for the function $f|_\rho$.

To conclude that the decision tree depth of $f|_\rho$ is less than d, it is therefore enough to show that all paths of length d or more are killed by the restriction. The probability that a given path p is *not* killed by ρ equals $(1/2 + \delta/2)^{\text{length of } p}$, as for the path to survive each value along p must be either unfixed by ρ or fixed to the value along its outgoing edge. As there are at most s such paths, we can upper bound the

probability of the complement event by a union bound:

$$\begin{aligned}\Pr[f|_\rho \text{ requires decision depth at least } d] &\leq \Pr[\text{some length} \geq d \text{ path of } T \text{ is not killed by } \rho] \\ &\leq \sum_{\text{paths } p \text{ in } T \text{ of length} \geq d} \Pr[p \text{ is not killed by } \rho] \\ &\leq s \cdot (1/2 + \delta/2)^d. \end{aligned}$$

□

On the other hand, even after a random restriction, $MAJORITY$ is not too likely to have small decision tree depth:

Claim 6. *Assuming $(1-\delta)n/2$ is an integer, with probability $\Omega(1/n^2)$, $MAJORITY|_\rho$ on n inputs has decision tree depth at least δn.*

Proof sketch. Consider the event that exactly $(1-\delta)n/2$ of the inputs are fixed to 0, exactly $(1-\delta)n/2$ are fixed to 1, and the remaining δn are unfixed by ρ. If this is the case, $MAJORITY|_\rho$ is a majority of its unfixed δn inputs, and so it has decision tree depth (at least) δn. What is the probability that ρ has exactly $a = (1-\delta)n/2$ zeros, $b = (1-\delta)n/2$ ones, and $c = \delta n$ stars? Among all possible values of a, b, and c this one has the maximum likelihood. As there are $\binom{n+2}{2}$ possible choices for the triplet of numbers (a, b, c), the probability is at least $1/\binom{n+2}{2} = \Omega(1/n^2)$. (A more precise estimate using Stirling's formula gives the stronger lower bound $\Omega(1/n)$.) □

From Theorem 5 and Claim 6 it follows that if $MAJORITY$ on n bits has a decision tree of size s then $s \cdot ((1+\delta)/2)^{\delta n} = \Omega(1/n^2)$, so $s = \Omega((2/(1+\delta))^{\delta n}/n^2)$. When $\delta = 0.46$ (any δ works but this is roughly the best value), this yields a lower bound of about $s = \Omega(1.156^n)$.

4 Disjunctive normal form

A formula in *disjunctive normal form* (DNF) of size s is an OR of s *clauses*, each of which is an AND of (distinct) literals. For example, distinctness of two n-bit strings $x, y \in \{0,1\}^n$ can be expressed as a DNF of size $2n$:

$$\begin{aligned}DISTINCT(x,y) = (x_1 \text{ AND NOT } y_1) \text{ OR } (\text{NOT } x_1 \text{ AND } y_1) \\ \text{OR } \cdots \text{ OR } (x_n \text{ AND NOT } y_n) \text{ OR } (\text{NOT } x_n \text{ AND } y_n).\end{aligned}$$

Theorem 7. *If f has a decision tree of size s, then it has a DNF of size at most s.*

Proof. Let T be a decision tree for f of size s. For every path p of T that leads to a 1-leaf introduce a clause c_p that looks like this: Each variable x_i in p appears in c_p as literal x_i if its outgoing edge is a 1-edge and as literal NOT x_i if its outgoing edge is a 0-edge. The resulting DNF accepts exactly those inputs that lead to 1-leaves of T, so it computes the function f. □

So DNFs are at least as strong a model of computation as decision trees. Are they stronger? To answer this question in the positive, we need to come up with a function that requires a large decision tree but admits a small DNF. It turns out that our usual suspects from the previous section — "most functions", $PARITY$, and $MAJORITY$ — are not of much help here. A good exercise which may appear on your homework is to prove a Shannon theorem for DNF, namely to show that most functions require DNF size $\Omega(2^n/n)$. As for $PARITY$, and $MAJORITY$, I suggest that you try writing small DNFs for these functions to get some intuition why this is so. In the next lecture, we will in fact show that the PARITY and MAJORITY functions cannot be represented by DNFs of size sub-exponential in the input length n.

Could it then be the case that DNFs of size s are *equivalent* to decision trees of size s, or maybe that they can be represented by decision trees of somewhat larger size, say s^2 or s^{10}? Not so: In the homework you will show that $DISTINCT$ (for input size $2n$) requires decision trees of size 2^n or more. This example

shows that there can be an *exponential gap* in the size of the smallest DNF and the smallest decision tree for the same function. In the homework you will also show that the gap can never be more than exponential.

To conclude, today we saw two examples of non-uniform models of computation, namely decision trees and DNFs. We saw several ways to argue that certain functions that cannot be computed by small decision trees. We then stated a *separation* between these two models: DNFs of a given size are at least as powerful as decision trees of that size, but in the other direction there exist DNFs of size s that require decision tree size $2^{\Omega(s)}$ for infinitely many s.

5 A bit of history and a bit of motivation

Computational complexity is a demanding subject with many questions but few definite answers. There has been no meaningful progress on the important open problems in the last three decades. Even minor results sometimes require inventing new mathematics and much trial and error. Is studying the subject worth the effort? I believe that it is. Let me try to illustrate the importance of complexity in two "modern" subjects: machine learning and quantum computing.

Recent spectacular progress in machine learning leaves one with the impression that there is little that neural networks are unable to accomplish: they can translate from Chinese, tell apart cats and dogs, write essays, and so on. Yet there are areas where machine learning has had limited impact, for example in digital crime. Online shopping today does not feel any less secure than it was a decade ago. Should we worry that as machine learning gets even better it will be able to breach encryptions and steal confidential information?

This question has to do with computational complexity, and the dominant belief among experts is that the answer is no. The reason is that e-commerce security is based on specially designed problems no algorithm, including the best-trained neural network, can solve in a reasonable amount of time. Later in the course I hope to show you what some of these problems look like. These were discovered by researchers in complexity and related fields like cryptography and learning theory in the 1970s and 1980s and remain relevant to this day.

The main challenge in quantum computing is to build a quantum computer. This is a device envisioned in the 1980s for the purpose of simulating quantum-mechanical systems. Thanks to the work of complexity theorists in the 1990s it was discovered that a quantum computer would be much faster at solving purely classical problems that have nothing to do with quantum mechanics like searching databases and factoring numbers. I will explain why in Lecture 4.

Building a full-fledged quantum computer has proven challenging. The current ambition is to design a special device that can do *any* task that no classical computer can perform efficiently. Coming up with tasks suitable for this purpose is the business of complexity theory. There are skeptics who believe that a quantum computer is impossible to build for reasons that have again to do with complexity theory, such as robustness of computation to noise.

These and many other challenges of computer science are closely related to questions of computational complexity. To understand the limits of current technology and to get a sense where progress might happen and where it is unlikely I believe that having a working knowledge of the subject is helpful.