My TCS Course at Epsilon Camp

Scott Aaronson

with assistance from Anonymous, Ian Ko, Tzak Lau, and Sunetra Rao

June 2025

Contents

Lecture 1: Bits

Theoretical computer science is the math behind how computers work and what they can do and how much time and memory they need to use.

How many of you have done programming? What languages? What did you make?

This class will barely need actual computers. I might bring my laptop to demo something or other, but mostly we'll just do math—pen and paper and blackboard. I want to show you how theoretical computer science would be one of the awesomest kinds of math ever discovered *even if* we had no actual computers to try the ideas out on. But since we do have actual computers—*wow*! If any of you do any programming after this class, you're gonna be at so much of a higher level!

What's a computer? A machine that manipulates *information*. What's information? Hard to define. The easiest way is to give examples. I'm thinking of a number from one to ten. Anyone want to ask a yes/no question about my number? Is it more than 5? Yes. After you asked, you got some of the information, but not all of it. Another question? Is it more than 7? No. Is it 7? No. Is it 6? Yes. ... Now you have all the information.

The basic unit of information is the *bit* (a word popularized by Claude Shannon, who we'll meet again).

Who knows what a bit is? Right, `0` or `1`. You may have heard that computers store all information as giant strings of bits: `0011010110111101...`

Why are bits so convenient for computers? Well, they're just the simplest unit of information that works: on or off. High voltage or low voltage.

This is not the only possibility: we can use trits (`0`, `1`, `2`) or decimal digits (`0`, `1`, `2`, `3`, `4`, `5`, `6`, `7`, `8`, `9`), or quarts (`0`, `1`, `2`, `3`).

Are our computers limited because they only deal with bits? No, because we can encode anything we want using *lots* of bits!

With 2 bits, how many possibilities are there? 4: `00`, `01`, `10`, `11`.

With 3 bits how many? 8: `000`, `001`, `010`, `011`, `100`, `101`, `110`, `111`.

4 bits? 16. 5 bits? 32. The powers of 2! Who can recite more? (I *know* you kids can.)

How many bits are needed to identify every person on Earth uniquely? There are about 8 billion people, which is about 2^{33}, so let's say 34 bits to be safe.

How many bits are needed to identify every atom in the visible universe uniquely? There are about 10^{80} atoms, or very roughly 2^{300}, so let's say 300 bits (and only that!).

This is the magic of exponential growth!

Some useful terms:

Byte	8 bits	
Kilobyte	2^{10} bytes	1024 bytes
Megabyte	2^{20} bytes	1024 kilobytes
Gigabyte	2^{30} bytes	1024 megabytes
Terabyte	2^{40} bytes	1024 gigabytes
Petabyte	2^{50} bytes	1024 terabytes
Exabyte	2^{60} bytes	1024 petabytes
Zettabyte	2^{70} bytes	1024 exabytes
Yottabyte	2^{80} bytes	1024 zettabytes

What's the total amount of information in computers on earth today? 200–400 zettabytes.

How many bits are needed to represent one English letter? If you don't do anything clever, 5 bits, since $2^5 = 32$ is the first power of 2 that's at least 26. You could use the following encoding, for example:

1	00000 - A	14	01101 - N
2	00001 - B	15	01110 - O
3	00010 - C	16	01111 - P
4	00011 - D	17	10000 - Q
5	00100 - E	18	10001 - R
6	00101 - F	19	10010 - S
7	00110 - G	20	10011 - T
8	00111 - H	21	10100 - U
9	01000 - I	22	10101 - V
10	01001 - J	23	10110 - W
11	01010 - K	24	10111 - X
12	01011 - L	25	11000 - Y
13	01100 - M	26	11001 - Z

So the encoding of "EPSILON CAMP RULEZ" (with no spaces) would be

```
00100 01111 10010 01000 01011 01110 01101 00010
00000 01100 01111 10001 10100 01011 00100 11001
```

The ASCII code uses 8 bits per character. That's enough to encode uppercase and lowercase English letters, numerals, and punctuation marks, with plenty of room left over for weird other symbols.

OK, who here knows binary arithmetic?

The powers of 2, when written in binary, look like this: $1, 10, 100, 1000, 10000, \ldots$.

Adding in binary is similar to adding in decimal:

$$\begin{array}{rrrrrr} & & 1 & 1 & 1 & \\ & 1 & 0 & 1 & 0 & 1 \\ + & & & 1 & 1 & 1 \\ \hline & 1 & 1 & 1 & 0 & 0 \end{array} \quad \Longleftrightarrow \quad \begin{array}{r} 21 \\ +7 \\ \hline 28 \end{array}$$

Multiplying in binary uses the same idea!

In computers an 8-bit unsigned integer goes from 0 to 255. A 16-bit unsigned integer goes from 0 to 65,535. A 32-bit unsigned integer goes from 0 to 4,294,967,295.

A puzzle: what does half of a bit mean? Or $\frac{3}{5}$ of a bit?

How many bits do we need to represent one trit? Two: `0 = 00, 1 = 01, 2 = 10`. But this is wasteful!

How many bits do we need to represent two trits? Four, 2^4 exceeding 3^2.

So for three trits, do we need six bits? No, now only five! Since $2^5 = 32$ already exceeds $3^3 = 27$.

How many bits do we need to represent 1000 trits? Well, we're looking for the x that solves $3^{1000} = 2^x$. This turns out to be $x \approx 1585$.

In general, as N gets large, N trits should correspond to cN bits, where $3^N = 2^{cN}$. Taking the N^{th} root of both sides, we get $2^c = 3$, whose solution is $c = \log_2 3 \approx 1.585$. So, N trits are about $1.585N$ bits.

An irrational number! I can't resist: does anyone know the *proof* that $\log_2 3$ is irrational?

Right, suppose by contradiction that $\log_2 3 = A/B$, where A and B are positive integers. Then $2^{A/B} = 3$. So $2^A = 3^B$. But 2^A is even while 3^B is odd, contradiction, we're done.

One can also encode pictures in bitmap images: for example,

```
00000
01010
00000
01010
01110
```

which is a smiley face (can you see it?).

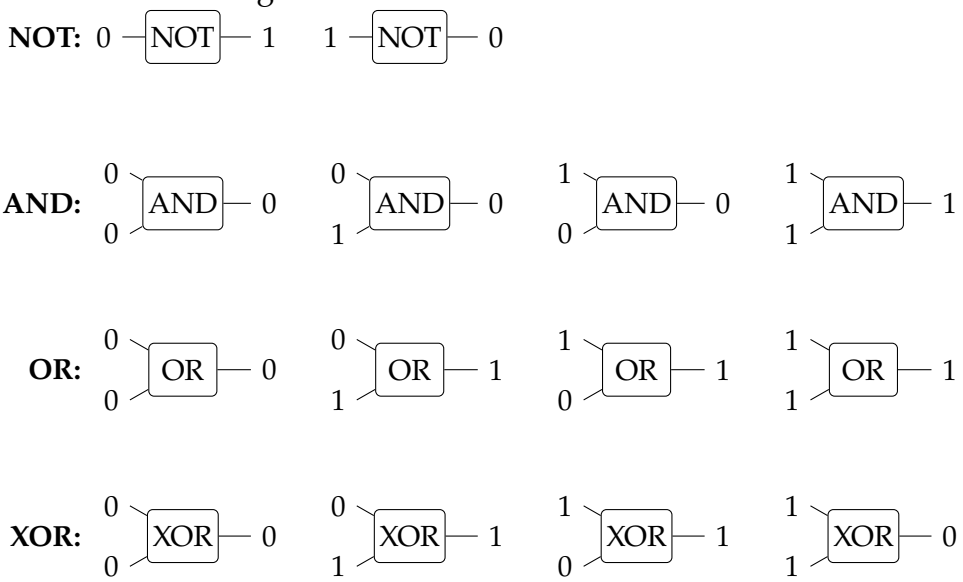
This is how we send text, images, video over the globe!

When I was little, I saw my dad use a fax machine and wondered "how did the paper get through the wire?" The answer: Information! Bits!

Lecture 2: Gates

How do computers manipulate bits? At the lowest level, using *Boolean logic gates*. George Boole was a mathematician two hundred years ago who knew there must be a way to turn logic into algebra. The way he said to do this is to set False = 0 and True = 1. Why? Well, he said, clearly False^2 = False and True^2 = True. Now, what are the solutions to the equation $x^2 = x$? Right, $x = 0$ and $x = 1$! So we get what would later be called a bit.

Now we can build gates like these:



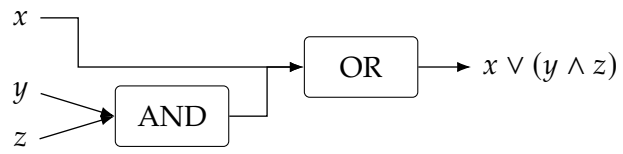
One of the greatest achievements of the last century is the transistor, which (among other things) made it easy to build these gates. Integrated circuits are made out of billions of transistors!

We can also define *Boolean functions*, using *truth tables*, like so:

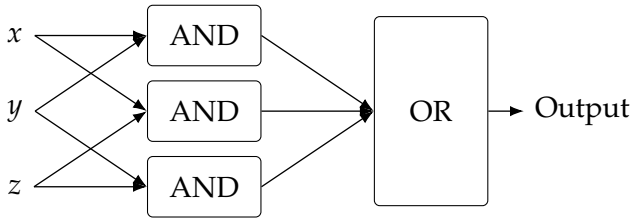
x	NOT(x)
0	1
1	0

x	y	AND(x, y)	OR(x, y)	XOR(x, y)
0	0	0	0	0
0	1	0	1	1
1	0	0	1	1
1	1	1	1	0

This is a circuit for "either x or (y and z)":



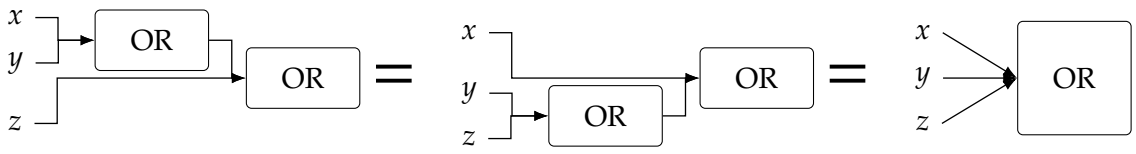
This is a circuit for majority of 3:



And this is the truth table:

x	y	z	MAJ(x, y, z)
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	1

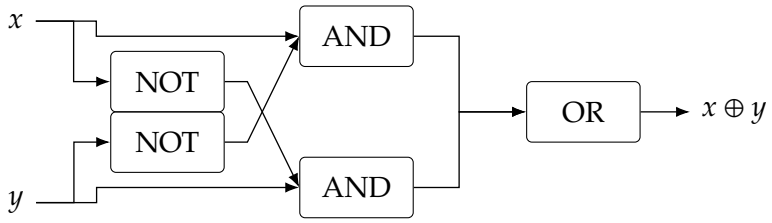
A cool fact:



This is also true for AND and XOR.

Challenge. Build a XOR gate using only AND, OR, and NOT.

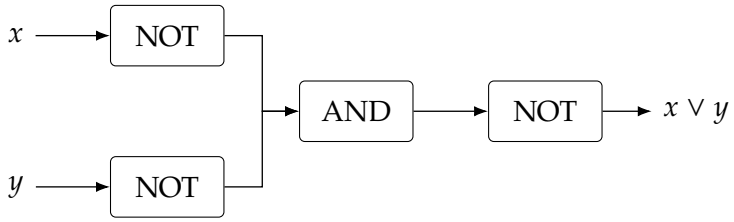
Solution.



□

Challenge. Build an OR gate using only AND, and NOT.

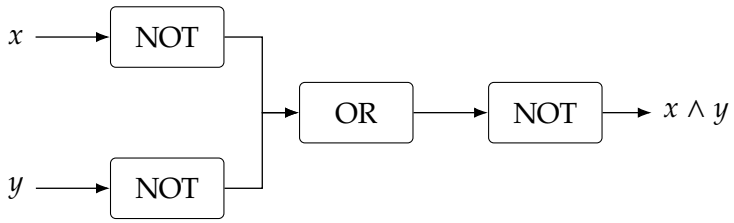
Solution.



□

Challenge. Build an AND gate using only OR and NOT.

Solution.



□

The two identities above are called "De Morgan's Laws."

Challenge. Build a NOT gate using AND and OR gates.

This is impossible! Why? Because AND and OR are *monotone*, meaning that changing an input bit from 0 to 1 can only ever change the output from 0 to 1, and never from 1 to 0. Therefore any function that you can build from AND and OR gates will be monotone as well. But NOT is non-monotone.

Challenge. Build an AND gate using NOT and XOR gates.

This is also impossible! Why? The answer involves *Boolean arithmetic*, also known as arithmetic mod 2, the kind where $1 + 1 = 0$ (and hence $x = -x$, and hence addition and subtraction are the same thing):

$\times$	0	1
0	0	0
1	0	1

$+$	0	1
0	0	1
1	1	0

We can write our logic gates in terms of Boolean arithmetic like so:

- $\text{NOT}(x) = 1 - x = 1 + x$,
- $\text{AND}(x, y) = xy$, and
- $\text{XOR}(x, y) = x + y$.

But now, it's a simple fact of algebra that with + only, we can only express *linear* functions, never *quadratic* functions like xy—and that's why we can't get AND from XOR.

Incidentally, what is $\text{OR}(x, y)$ as a formula in Boolean arithmetic? Here we can use De Morgan's Law:

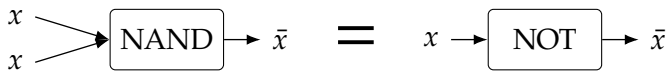
$$\text{OR}(x, y) = 1 - (1 - x)(1 - y) = 1 - (1 - x - y + xy) = x + y - xy.$$

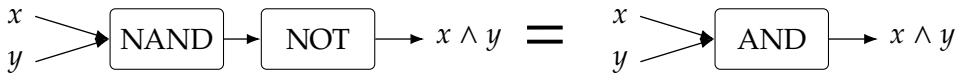
Challenge. This is the truth table for the so-called NAND function:

x	y	NAND
0	0	1
0	1	1
1	0	1
1	1	0

Use NAND gates alone to make AND, OR, and NOT.

Solution.





□

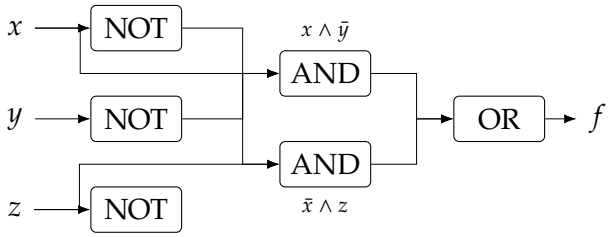
A set of logic gates is called *universal* if you can use them to make *any* Boolean function f on any number of bits, given f's truth table. We've seen that the set {AND, OR} is not universal (because it's monotone), and {NOT, XOR} is not universal either (because it's linear).

Theorem 2.1. {AND, OR, NOT} is *universal. And therefore* {AND, NOT} *and* {OR, NOT} *are also universal, as are* {NAND} *and* {NOR} *by themselves.*

Proof. The proof uses the so-called truth table method. First, you write out the truth table of the function f that you want, like so:

x	y	z	$f(x,y,z)$
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	0
1	0	0	0
1	0	1	1
1	1	0	0
1	1	1	0

Then, for every 1-input of f (that is, every x such that $f(x) = 1$), you build a circuit of AND and NOT gates to recognize f. Finally you connect all those circuits together with OR gates, like so:



This method doesn't necessarily give the *smallest* circuit—we'll talk about that later—but it will always work! □

Homework 2.1. Build a circuit (using, say, the AND, OR, and NOT gates) to add two 3-bit integers written in binary.

Homework 2.2. Show that by using only the AND and OR gates (as well as the constant inputs 0 and 1, which you can treat as free), you can express any monotone Boolean function.

A challenge now! Suppose we pick a truth table *randomly* for a Boolean function with n inputs and one output, e.g.

x	y	z	$f(x,y,z)$
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	0
1	0	0	0
1	0	1	0
1	1	0	1
1	1	1	0

Now remember that the NAND gate is universal, meaning that with enough NANDs you can make *any* function including this f. The question is: how many NAND gates

will the *smallest* circuit for f have? Do you think it will grow polynomially in n, e.g. like n^2, or exponentially, e.g. like 2^n?

Of course, in general we don't know! *Some* f's will have small circuits. All f's will have circuits of size at most $\sim 2^n$. But do any f's *require* circuits of size $\sim 2^n$? If so, what's an example?

Amazing claim: Even though I can't give you an example, I can prove that *almost all* f*'s* require nearly 2^n gates!

How? Well, firstly, how many n-bit Boolean functions f are there? Are there 2^n? No, think much bigger!

Right, there are 2^{2^n} of them. Why? Because to specify even a *single* n-bit Boolean function, you need to specify all 2^n bits of its truth table. This gives you 2^{2^n} possible choices.

But now, how many circuits are there with T NAND gates or fewer?

To give a crude upper bound, we can say there are at most n^2 choices for the two inputs to the first NAND gate, then at most $(n+1)^2$ choices for the two inputs to the second NAND gate (since the output of the first NAND gate can now be an input), then $(n+2)^2$ choices for the two inputs to the third NAND gate (since the outputs of the first two NAND gates can now be inputs), and so on. This gives us

$$n^2(n+1)^2(n+2)^2 \cdots (n+T-1)^2 \leq [(n+T)^2]^T = (n+T)^{2T}$$

choices overall. (Incidentally, this count leaves more than enough room to encode the possibility that the circuit "stops" before the T^{th} NAND gate, leaving fewer than T gates overall.)

But now, notice that each circuit can represent only one Boolean function! So, if we want to represent all 2^{2^n} functions using circuits with at most T gates, then we need

$$(n+T)^{2T} \geq 2^{2^n}.$$

Or taking the log of both sides:

$$2T \log(n+T) \geq 2^n.$$

Solving for T—well, there's no closed-form answer, but we find that T needs to be at least *approximately* $2^n/n$ before the above inequality holds. Furthermore, if T is significantly smaller than $2^n/n$, then $(n+T)^{2T}$ will be much, *much* smaller than 2^{2^n}. So, our conclusion is that *almost all* n-bit Boolean functions must require at least $\sim 2^n/n$ NAND gates.

This is Shannon's counting argument: there are too many Boolean functions, and not enough small circuits to go around! Thus, even though we haven't given a single example of an exponentially hard function, we've proved that such hardness is the norm, *not* the exception!

In theoretical computer science, we'll care a lot about the *tiny subset* of n-bit Boolean functions that have circuits with "reasonable" numbers of gates (like $100n$ or n^2 or whatever), even though it will turn out to be *staggeringly* hard to determine exactly what is and isn't in that subset. But more about that later!

Lecture 3: Finite Automata

We saw last time that the AND, OR, NOT gates (or even NAND) are enough to make *any* Boolean function, on *any* number of inputs, including MAJORITY, Addition, Multiplication, Primality Testing, Minecraft, . . .

So why aren't we done? What do you think?

Well, I'd say the biggest limitation of circuits is that each one can only handle a fixed number of bits! As soon as there are more bits, you need a brand-new circuit!

What we want is a single program that can handle inputs with *any* number of bits. Even if the input is much longer than the program itself!

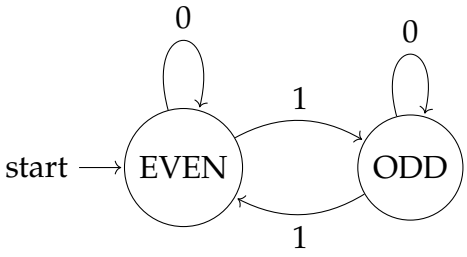
Example. A program to compute $n!$:

```
1  Input: n
2  f ← 1
3  for i = 1 to n
4          f ← f × i
5  Print f
```

This program works for any n. (Well, at least until n gets too big to fit in memory.)

Today we'll learn about an especially simple type of program called a *finite automaton*. A finite automaton reads a string of bits from left to right, maintains an internal state, and updates according to a rule.

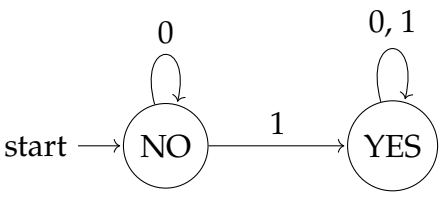
Example. Suppose we have a stream of bits and we need to know whether the number of `1`'s is odd or even; for example the number of `1`'s in `0101100` is odd and the number of `1`'s in `1010110` is even. Then we can use this machine:



Do an execution trace.

Challenge. Build a finite-state machine that checks if there are *any* `1`'s in the input.

Solution.



□

Challenge. Build a finite-state machine that checks if there are three consecutive `1`'s in the input—e.g., it should accept `0101110` and reject `0110110`.

Solution.

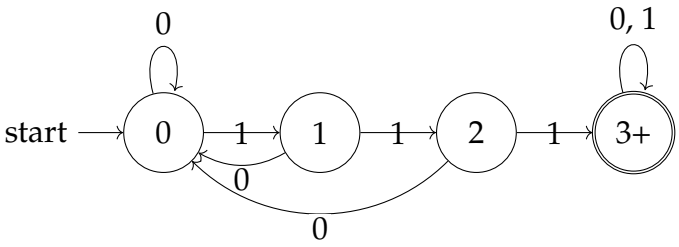
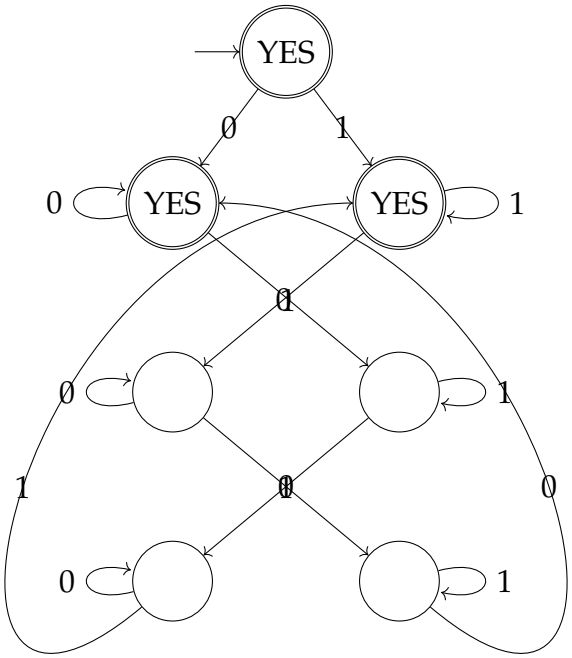


Figure 1: This tracks the streaks of 1s. If a 0 appears, it breaks the streak and returns back to the start state (0)

□

Challenge. Build a finite-state machine that checks if the number of *changes* between `0`'s and `1`'s is a multiple of 3.

Solution.



□

Is there anything finite-state machines *can't* do?

Challenge. Build a finite-state machine that checks if the input is a palindrome—i.e., is the same backwards and forwards, like 11011 but *not* like 11001.

This seems hard, because "how do you remember the first half?" You seem to need more and more states! But can we *prove* it's impossible?

We can ... by considering only inputs of the form 10^a10^b1, where the notation 0^a means a 0's in a row! For example 101001, 1001001, etc. Note that 10^a10^b1 is a palindrome if and only if $a = b$.

The key insight is that there must be some $a \neq b$ such that our machine ends in the same state after reading 10^a1 and 10^b1. Why? Because there are only finitely many states!

So then 10^a10^a1 and 10^b10^a1 must either be both accepted or both rejected. But one of these is a palindrome and the other isn't! QED.

Next we introduce *regular expressions*: a way to specify a set of strings that's "totally unrelated" to finite-state machines! In a regular expression, you're allowed the alphabet symbols 0 and 1, which you can string together from left to right, but you're also allowed the following extra symbols:

- Parentheses, which tell you in which order to evaluate things.
- |, meaning OR. For example, the regular expression 0|1 means the set of strings $\{0, 1\}$. Likewise, 00|11 means $\{00, 11\}$. 1(0|10)0 means $\{100, 1100\}$. (0|1)(000|111) means $\{0000, 0111, 1000, 1111\}$.
- *, meaning "you can repeat this as many times as you like (*including* zero times)." For example, 0^* means $\{\epsilon, 0, 00, 000, \ldots\}$, where ϵ is the empty string (the string of length 0). $(01)^*$ means $\{\epsilon, 01, 0101, 010101, \ldots\}$.

Now we can get creative and combine the symbols: for example, $(0|1)^*$ means the set of all possible strings (why?).

Meanwhile, $(00|11)^*$ means the set of strings

$$\{\epsilon, 00, 11, 0000, 0011, 1100, 1111, \ldots\}$$

Challenge. Give a regular expression for the set of all strings with at least one 1.

Solution. $(0|1)^*1(0|1)^*$. Note that $(0|1)^*10^*$ also works—do you see why? □

Challenge. Given a regular expression for the set of strings with an even number of 1's.

Solution. $(0^*10^*1)^*0^*$. (Would $(0^*10^*10^*)^*$ also work? Why or why not?) □

Challenge. Give a regular expression for the set of all palindromes.

This one seems harder! For how do you enforce that the second half of the string should be the reverse of the first half, using | and * only?

Hmm, could there a connection to finite-state machines after all? Alright, I'll tell you the punchline:

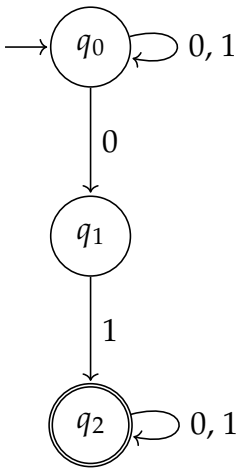
Surprising Theorem. *Every set of strings S has a regular expression if and only if it has a finite-state machine.*

Note that *both* directions here are interesting—and, as it happens, *both* can incur exponential blowup in size.

The Surprising Theorem, in turn, has all sorts of interesting consequences. For example, since finite-state machines can clearly "NOT" their outputs, regular expressions must be able to do the same, even though we didn't include any "NOT" symbol when we listed the rules for forming regular expressions! In other words:

Nontrivial Corollary. *If there's a regular expression for the set of all strings with property P, then there's also a regular expression for the set of all strings with property* NOT(P).

How do we prove the Surprising Theorem? Just to give you a little taste, the proof involves introducing yet a third concept, called "nondeterministic" finite-state machines—the machines we saw earlier being "deterministic." A nondeterministic machine is allowed to follow *multiple* paths from the same state when reading the same symbol, like so:



The rule is that the machine "accepts" a string if and only if there *exists* a path that you could follow on that string from the start state that ends in an accept state. For example, the machine above accepts the strings 01 and 101, but does not accept the string 1110.

To prove the Surprising Theorem, the first step is to show that, whenever a set of strings S has a nondeterministic machine, S also has a deterministic machine! This might be a little less shocking, once I tell you that if the nondeterministic machine has n states, then the deterministic machine could have as many as 2^n states. With that hint, maybe you can see how the simulation goes?

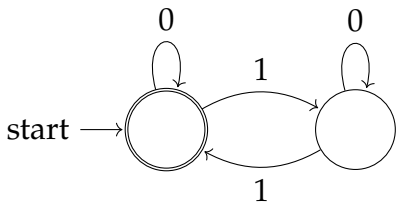
Anyway, the next step is to show that every regular expression has an equivalent nondeterministic machine, and the third and final step is to show that every nondeterministic machine has an equivalent regular expression.

I offer the following, *only* for those who want a serious extra credit challenge:

Homework 3.1. Fill in the details of the above proof sketch.

Lecture 4: Turing Machines

We saw finite-state machines last time. Their big advantage over circuits was that a single machine could handle infinitely many different inputs, e.g.



checks whether *any* string has an even or odd number of 1's. Their big disadvantage is that they're weak! They can't even decide whether a string is a palindrome—something that should be trivial for computers. What do you think makes finite-state machines so weak?

- They can only move to the right on the input, not left.
- On an n-bit input, they can only do n steps.
- They don't have an external memory.
- They can only read bits, not write them.

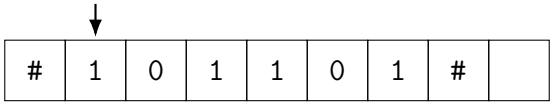
Today we'll see a machine that solves all these interrelated problems at once . . . the **Turing Machine**!

The Turing machine was invented by Alan Turing in 1934 when he was a 22-year-old college student. (No pressure!) A few years later, Turing went on to help break the Nazi codes and win World War II—but more about that later!

For now, the thing to know is that ever since the 1930s, Turing machines have been our main mathematical model of what we mean by "a computer," or something being "computable."

The Turing machine is equipped with a one-dimensional paper tape divided into squares, infinite in both directions.

Either a `0` or a `1` is written on each square. (We'll sometimes also allow more symbols, like `#` for "STOP.") The tape could initially be all `0`'s or could have an input written on it, like this:



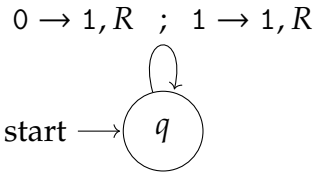
We'll often want our Turing machine to *decide* something about its input: for example, "is the string between the #'s a palindrome, or not?" (In the example above, of course, it is.)

The Turing machine has a "tape head" which points to a single square and can do the following:

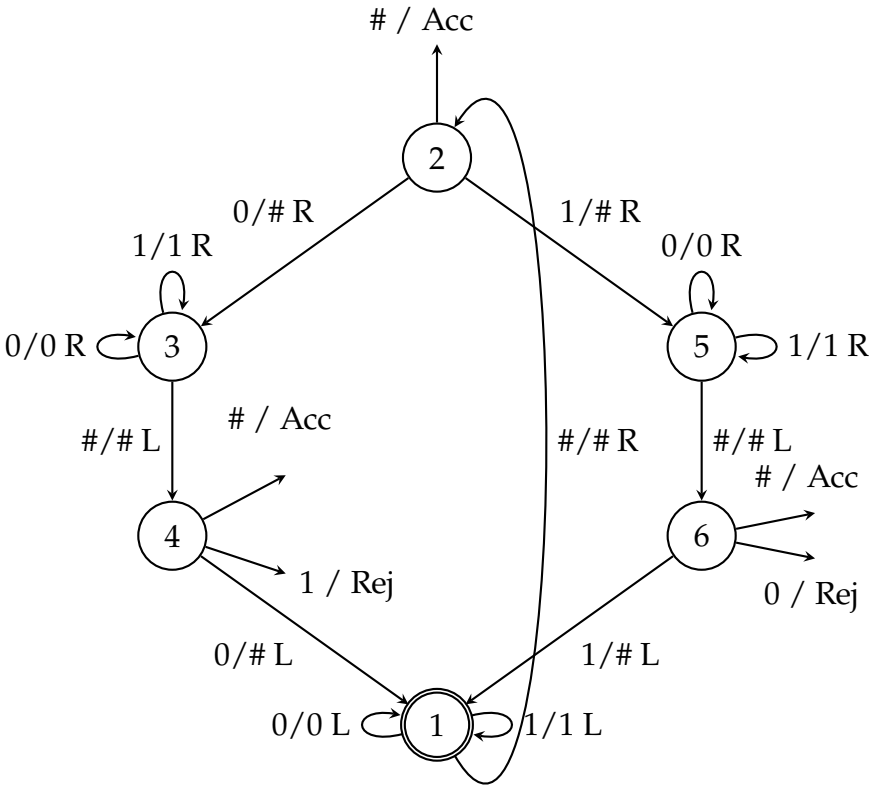
- Read what symbol is on the square (0, 1, #).
- Overwrite the symbol with a different symbol.
- Move one square to the left or the right on the tape.
- Transition to a new internal state.
- Halt, and either accept or reject the input.

Crucially, the current internal state, plus the symbol currently being read, determine all of these choices.

As an example, here's a 1-state Turing machine that just prints 1's on its tape and moves to the right forever:



And here's a 7-state Turing machine that solves the palindrome problem:



(How does it work?)

Homework 4.1. Design a two-state Turing machine over the {0, 1} alphabet, that on an initially all 0 tape runs for the maximum number of steps other than infinity.

Homework 4.2. Design a Turing machine to check whether the input has more 1's than 0's.

Take this on faith: there are Turing machines that

- Add two integers in binary.
- Multiply two integers in binary.
- Check whether a given integer is prime.
- Run Super Mario Bros.

In fact there's a "universal" Turing Machine—a single Turing Machine that simulates any other Turing Machine described in a coded form on its input tape. In other words: we have . . . programs! Apps!

Stepping back, how is a computer different from a toaster, vacuum cleaner, etc.? A toaster toasts bread. A vacuum cleaner vacuums. But a computer does email, web browsing, Minecraft, calendars, word processing, weather prediction and other things that haven't even been invented yet.

How? Stored programs! Software! It all traces back to Turing's universal Turing machine.

But how do we prove the existence of a universal Turing Machine? At a high level, it's "just" another programming exercise.

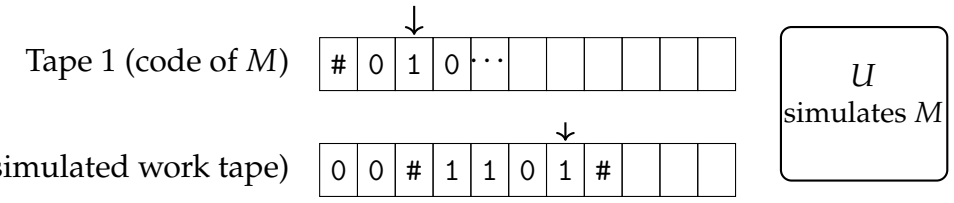
First we need a way to encode any Turing machine by a string of symbols (say, `0`, `1`, `#`). We can make a table of all the states and transitions, like so:

State	Read	Action	Next
q	`0`	write `1`, move R	q
q	`1`	write `1`, move R	q
q	`#`	halt	—

Next, we encode this table as a string that could itself be written on a Turing machine tape. The details are quite boring.

But given a Turing machine M, how do we simulate M's behavior on an arbitrary tape, if a description of M itself must always be written on the tape?

One solution is to define a *two-tape* Turing machine. The first tape will store M's code, plus which state M is currently in. The second tape is where our universal machine, U, will do whatever M would do.



In his 1936 paper, Turing showed a single such U that (slowly) simulates any M described on its first tape. His program had a bug, which he had to fix in 1937!

Analogy: Once you can code in Python, one of the many things you can code is a C compiler, or even another Python compiler!

But there's a problem: we needed to add a tape! The solution is to give a general simulation of two-tape Turing machines by one-tape Turing machines, e.g. by interlacing the symbols. Again, I won't inflict the details on you.

So Turing machines get us over the "threshold of universality." Except for being slow and lacking the right peripherals (graphics, Internet access, etc.), they're basically modern computers!

Is there anything that Turing machines *can't* do?

It's said that theoretical computer science is one of the only fields that was born with knowledge of its own limitations.

In the same 1936 paper, Turing showed that some math problems can't be solved even by a Turing machine.

One way to see this is with an analogue of Shannon's counting argument—the thing that showed that there aren't enough small circuits to represent all n-bit Boolean functions. Here, though, rather than comparing two numbers, like $(n+T)^{2T}$ and 2^{2^n}, we need to compare two different degrees of infinity!

Since I'm told you guys learned what this stuff means last summer: the infinity of possible Turing machines is "merely" countable, or $\aleph_0$. Why? Right, because each Turing machine can be specified by a finite string of bits, which can in turn be encoded as a positive integer.

By contrast, I claim that the infinity of possible computational problems is *uncountable*. Why? Because for every possible *set* S of finite strings, we get another problem: namely, the problem of deciding whether or not our Turing machine's input x belongs to S! But a single set S of finite strings already takes infinity bits to specify, since for every string x, we need to specify whether or not $x \in S$. So the infinity of all these possible S's is actually $2^{\aleph_0}$, also called the Continuum—just like the infinity of real numbers, or like the infinity of *infinite* binary strings. And Cantor's Theorem, one of the greatest theorems in all of mathematics, tells us that $2^{\aleph_0} > \aleph_0$.

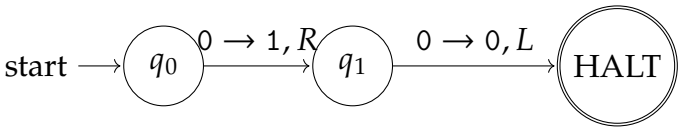
In conclusion, then, some problems *must* lack a Turing machine, simply because "there aren't enough Turing machines to go around"! Or rather, because the infinity of problems is greater than the infinity of Turing machines.

Notice that, just like Shannon's counting argument, this abstract argument failed to produce a single *example* of an uncomputable problem: it merely showed that "almost all" problems must be uncomputable!

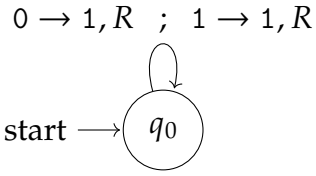
In this case, however, we can *also* give a very nice example of an uncomputable problem.

The Halting Problem: Given as input a Turing machine M (described by a string of symbols), decide whether M halts eventually when run on a tape of all 0's.

For example



halts after two steps but



runs forever.

A solution to the Halting Problem would be super useful: you could tell whether your program will get stuck in an infinite loop!

But there are hints that it might be super hard. Consider for example the following program:

```
1  for n = 4, 6, 8, 10, 12, ...
2      z ← 0
3      for a = 1, 2, 3, ..., n − 1
4          if a and n − a are both prime
5              z ← 1
6      if z = 0
7          Halt.
```

Does this program halt? Well, it does if and only if the *Goldbach Conjecture* is false!

That's the famous conjecture from the 1700s that says that every even number greater than or equal four can be written as a sum of 2 primes, e.g.

$$\begin{aligned} 4 &= 2 + 2 \\ 6 &= 3 + 3 \\ 8 &= 5 + 3 \\ 10 &= 5 + 5 = 7 + 3 \\ 12 &= 7 + 5 \\ 14 &= 7 + 7 = 11 + 3 \end{aligned}$$

So solving the Halting Problem even for this one program would require settling Goldbach's Conjecture! And likewise for many other famous unsolved math problems.

But that doesn't *prove* that there couldn't be a solution. Turing's impossibility proof is one of the greatest in the history of math. Are you ready for it?

Suppose by way of contradiction that P is a program (Turing machine) to solve the Halting Problem. That is, for all Turing machines M, we have $P(\langle M \rangle) = \textsf{Accept}$ if M halts on a blank tape, and $P(\langle M \rangle) = \textsf{Reject}$ if M runs forever on a blank tape. Here $P(x)$ means the output of P when x is written on P's input tape, and $\langle M \rangle$ means the finite string of symbols that encodes M.

Then it's easy to modify P to get a new program Q such that for all Turing machines M:

- $Q(\langle M \rangle)$ runs forever if $M(\langle M \rangle)$ halts, and
- $Q(\langle M \rangle)$ halts if $M(\langle M \rangle)$ runs forever.

Here we've made two changes: we look at $M(\langle M \rangle)$ instead of M on the all-zero tape, and we halt or run forever instead of accepting or rejecting.

Now consider $Q(\langle Q \rangle)$: Q run on its own description! If $Q(\langle Q \rangle)$ halts, then $Q(\langle Q \rangle)$ runs forever. If $Q(\langle Q \rangle)$ runs forever, then $Q(\langle Q \rangle)$ halts. BOOM! Contradiction! We conclude that Q couldn't have existed, which means that P couldn't have existed either. So the Halting Problem can't be solved by any Turing machine.

Homework 4.3. Show that there's no Turing machine that takes as input a Turing machine description $\langle M \rangle$, and that accepts if $M()$ accepts, rejects if $M()$ rejects, and either accepts or rejects (but in any case, halts) if $M()$ runs forever.

Homework 4.4. Show there's a Turing machine that runs for infinite time, and that lists all Turing machines that eventually halt.

Homework 4.5. Show that, if there's a Turing machine to list all the elements of S (a subset of $(0|1)^*$), and another Turing machine to list all the elements of S's complement $\overline{S} = \{x : x \notin S\}$, then there's a Turing machine to decide membership in S.

Homework 4.6. By combining the previous two problems, show that there's no Turing machine to list all the Turing machines that run forever.

Lecture 5: Big Numbers

We saw Turing machines—how they can solve problems like palindromes (and arithmetic, and everything else needed to run a modern computer), and how there's a single Turing machine to simulate any *other* Turing machine. We also saw that even Turing machines have limits—e.g., they can't solve the Halting Problem.

Today we'll see how to use Turing machines to define what I'm almost certain are the largest numbers you've ever seen.

First, the biggest number contest! You'll each have 30 seconds to write on your card. The rules:

- You must specify a unique positive integer—a mathematician looking only at your card should know which one you intended.
- No infinities.
- No "1 + largest number anyone else wrote down," or anything like that—you can't refer to other cards!

Are the rules clear? Ready, set, GO!

What do we have? 999999? $9^{9^{9^9}}$? Better! Even 111111111 would've been better, since 1's are faster to write than 9's. See, naming huge numbers is not about how fast you can write. It's about having the right *notation*. Exponentials are an example of a powerful notation. Writing "10^{1000}" is a lot faster than writing a 1 followed by a thousand 0's. Stacking exponents gives even more power: $10^{10^{10^{10}}}$.

Multiplication is repeated addition and exponentiation is repeated multiplication. But what's repeated exponentiation? We can give it a name—*tetration*—as well as a notation. Let's say that ${}^4 3$ means $3^{3^{3^3}}$, that ${}^5 3$ means $3^{3^{3^{3^3}}}$, and so on. We could then write ${}^{1000000}10$ or even ${}^{{}^{10}10}10$. But any time we're writing something over and over, we should invent a notation so we don't have to. Thus, let "pentation" mean repeated tetration. "10 pentated to the 7" means ${}^{{}^{{}^{{}^{{}^{10}10}10}10}10}10$, and so on. Then we can let "hextation" mean repeated pentation, etc.

But now we can invent something wilder than any of these: the *Ackermann sequence*, from 1928!

$$A(1) = 1 + 1 = 2$$
$$A(2) = 2 \times 2 = 4$$
$$A(3) = 3^3 = 27$$
$$A(4) = {}^4 4 = 4^{4^{4^4}} = 4^{4^{256}} = \ldots \text{ too many digits to write down}$$
$$A(5) = 5 \text{ pentated to the } 5 = {}^{{}^{{}^{{}^5 5}5}5}5 = \text{WHOA...}$$

Now imagine that in the Biggest Number Contest, you wrote on you card $A(100)$.

And yet the Ackermann numbers are *puny* compared to what I'm going to show you next. After all, there's still a Turing machine that given any n written on its tape, calculates $A(n)$ and then halts. That won't be true for the next thing.

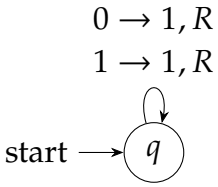
In 1962, a mathematician named Tibor Radó invented what he called the Busy Beaver sequence. Here's how it works. We pick some number n. We consider all possible Turing machines with n states and the $\{0, 1\}$ alphabet, which just run forever or halt (they don't accept or reject). How many such machines are there?

Well, for each state and each symbol (0 or 1), you could halt, but if you don't there are two choices for which symbol to write (0 or 1), two choices for which way to go (left or right), and n choices for the next state. So, $4n + 1$ choices overall. Multiplying across all states and symbols, this gives us $(4n + 1)^{2n}$ choices. Of course many will secretly be equal to each other, with e.g. different labels for the states, but at any rate it isn't *more* than that. This is a *finite* number.

Among these $(4n + 1)^{2n}$ machines, what does each one do, when run on a tape that's initially all 0's? Well, some of then run forever—they either don't have halt instructions, or these instructions never get triggered. We throw away all the ones that run forever and keep only the ones that halt.

Among the ones that halt, some might halt after only a few steps (like one or two), others only after a lot of steps. But there must be some machine that takes the *maximum* number of steps among all n-state machines that eventually halt. Whichever machine that is, we call it an n-state Busy Beaver. And the number of steps it runs is $\mathsf{BB}(n)$: the n^{th} Busy Beaver number.

What's $\mathsf{BB}(1)$? It's 1! If a 1-state doesn't halt as its very first step, it must go into an infinite loop, like this:



What's $\mathsf{BB}(2)$? That was your homework yesterday. What did you find? It's not too hard to see that $\mathsf{BB}(2) = 6$ just by playing around with various 2-state machines.

What's $\mathsf{BB}(3)$? A guy named Shen Lin figured out for his PhD thesis in 1963 that $\mathsf{BB}(3) = 21$. That was tricky: there are $(4 \cdot 3 + 1)^{2 \cdot 3} = 13^6 = 4,826,809$ 3-state machines. And it's not enough to find one that runs for 21 steps. Lin needed to *prove* that every machine that runs for more than 21 steps actually runs forever! Why is that hard? Right, it's the Halting Problem, which we proved yesterday has no general algorithm that works for every machine! So Lin needed to go on a case-by-case basis.

What's $\mathsf{BB}(4)$? A guy named Allan Brady worked on this for more than a decade before he finally proved in 1983 that $\mathsf{BB}(4) = 107$. (Looks pretty slow so far! But fee fi fo fum...)

What's $\mathsf{BB}(5)$? In 1990, Marxen and Buntrock discovered a 5-state machine that runs for 47,176,870 steps before halting, proving that $\mathsf{BB}(5)$ is *at least* that number. But it took another 34 years to prove that $\mathsf{BB}(5) = 47,176,870$. I wrote an article

5 years ago that ignited interest in the question. It inspired a worldwide team to examine all different 5-state machines with the help of computers—finally showing (via a computer-checked proof) that all 5-state Turing machines that run for more than 47,176,870 steps, actually run forever.

What's BB(6)? It was known for a while that $\mathsf{BB}(6) > 10^{30000}$. To be clear, no one actually ran the machine that many steps! But they found a 6-state machine that they could prove *would* halt after that many steps (way more than the number of nanoseconds since the Big Bang). Then a few years ago Kropitz showed that

$$\mathsf{BB}(6) > 10^{10^{10^{10^{10^{10^{10^{10^{10^{10^{10^{10^{10^{10^{10}}}}}}}}}}}}}} = {}^{15}10$$

Recently, someone improved this to $\mathsf{BB}(6) > {}^{10,000,000}10$. Then he improved it further, to $\mathsf{BB}(6) > {}^{9_2}2$. Maybe that's *still* not the limit! I don't know if humans will ever know for sure what's BB(6). Maybe AI can help figure it out for us.

BB(7)? Forget it. What is known is that $\mathsf{BB}(()7) > A(7)$, and even $\mathsf{BB}(()7) > A(11)$—by this point we've left the Ackermann function in the dust.

Nine years ago, my student and I proved that BB(8000) can't be determined from the usual axioms of math. That was later improved to show that BB(649) can't be so determined. I now suspect that maybe BB(7) or BB(8) can't be.

Anyway, if you're playing the biggest number contest against someone who's never seen Busy Beaver, you can *destroy* them by just writing BB(100) or whatever!

What makes these numbers grow so fast? I claim they grow faster than *any* computable sequence of numbers! Can anyone say why?

It's a consequence of something we already proved: the unsolvability of the halting problem. Suppose there were a program that took n as input, and outputted a number $f(n)$ after finite time such that $f(n) \geq \mathsf{BB}(n)$. I claim we could use that program to solve the Halting Problem . . . which would contradict Turing's famous theorem!

How would we do it? Let's say you're given a Turing Machine M with n states and you want to know whether M halts on an initially all-0 tape.

First, compute $f(n)$. Then run M for up to $f(n)$ steps. If M halts within that time, it halts. If it doesn't, then it must run forever—since if it was ever going to halt, it would've done so in at most $\mathsf{BB}(n) \leq f(n)$ steps.

So there couldn't have been such a computable f. So $\mathsf{BB}(()n)$ must grow *uncomputably fast*.

Does anything grow way faster even than $\mathsf{BB}(n)$? The answer will introduce another central concept of theoretical computer science: *oracles*.

What was an oracle in ancient Greece? Right, someone who predicted the future by consulting the gods. You'd bring them questions and they'd answer them.

This term was introduced to computer science by Alan Turing in 1939 right before he left to join Bletchley Park! An oracle is a magic box that solves a certain problem, e.g. the Halting problem, for free. A Turing machine with an oracle has an extra tape and it can write a string there and say "O mighty oracle!" and it will learn whether that string is included in a certain language or not in a single time step. By the notation

$M^A(x)$, we'll mean "the Turing machine M with the oracle A, run on input x." E.g., $M^{\text{HALT}}(\langle P \rangle)$ means "M with an oracle for the Halting Problem, run on program $\langle P \rangle$ as input."

We often ask: *if* you had an oracle for *this* problem which other problems could you solve? I claim: if you had an oracle for HALT, then you could compute the Busy Beaver function, and conversely, if you had an oracle for BB() you could solve HALT. (Why?) Thus, HALT and BB(n) are "computably equivalent." They have the same "Turing degree."

New question: would any problem still be uncomputable even if you had an oracle for HALT? Well, how about the Halting Problem for Turing machines with HALT oracles, what we could call the SUPERHALT problem! Why is SUPERHALT uncomputable even given a HALT oracle? Because Turing's proof "relativizes." In other words, it still works the same as before if all machines in question have access to HALT oracles. That is, let P be a machine such that P^{HALT} solves SUPERHALT. Then we could modify P to a machine Q, such that when we run $Q^{\text{HALT}}(\langle Q \rangle)$, it does the opposite of whatever it does. Therefore Q can't have existed, and therefore P can't have existed either.

We can keep going further, to the Halting Problem for Turing machines with SUPERHALT oracles (SUPERDUPERHALT?), then SUPERDUPERPOOPERHALT, and so on. Just like there's no biggest number, so there's no hardest computational problem. For every problem X, an even harder problem—one that can't be solved even with an X oracle—is the Halting Problem for Turing machines with X oracles. This is called the "Turing jump."

We can use this understanding to define even bigger numbers than we could with the BB() function. What's a function that grows uncomputably fast *even for Turing machines with oracles for BB()*? Why, the Busy Beaver function for Turing machines with oracles for BB() (or equivalently, for HALT).

We can continue the sequence

$$\text{HALT}, \text{SUPERHALT}, \text{SUPERDUPERHALT}, \text{SUPERDUPERPOOPERHALT}, \ldots$$

all the way into infinite ordinals. Indeed, when carried out between two professionals, the biggest-number contest will quickly degenerate into an argument about which ordinals are well-defined, which in turn become an argument about the axioms of set theory!

Now, why can't you destroy even Busy Beaver *and everything else like it* in the biggest number contest, using the following trick?

> THE LARGEST NUMBER THAT CAN BE SPECIFIED USING AT MOST ONE HUNDRED WORDS

This seems OK: there are only finitely many possible combinations that specify a number. Surely one of the numbers must be the largest.

Ah, but if we could define *that*, then why not *this*?

> ONE PLUS THE LARGEST NUMBER THAT CAN BE SPECIFIED USING AT MOST ONE HUNDRED WORDS

Anyone see the problem here?

Right, this number, by definition, *can't* be specified using at most one hundred words. But I just specified it using only (count them) sixteen words.

What gives?! This is called the "Berry Paradox." Logicians say the solution is that there's no mathematical definition of what it means to specify a number using English words. And how do we know that? Because, if there were, then the Berry Paradox would create a contradiction in math!

You can think of the Busy Beaver numbers as a way to get around the Berry Paradox by specifying huge numbers using a language that we *know* is well-defined: not English, but Turing machines (or computer programs). That gives us numbers that are mind-bogglingly enormous—in fact, *uncomputably* enormous—but not *so* enormous that they no longer even make sense.

Homework 5.1. Show that for every computable function f, we have $f(n) > \mathsf{BB}(n)$ for at most finitely many values of n.

Solution. Consider a computable function f and a Turing machine F with p states that computes f with input and output in binary. For every n we can build a Turing machine with $C \log n$ states that writes the binary representation of n on the tape. Call this Turing machine A_n. We can also build a Turing machine B that writes a number m in binary form from the tape and runs for $m + 1$ steps after reading the machine. Suppose the machine B has q states. If we chain the machines A_n, F and B we will get a machine that runs for at least $f(n) + 1$ steps. This machine has $p + g + C \log n$ states. For large enough n this will be less than n so $\mathsf{BB}(n) \geq f(n) + 1 > f(n)$. □

Besides the Busy Beaver function, a different way to make computability theory quantitative—one that I can't resist telling you about—is Kolmogorov complexity!

Here's something you may have wondered about: Which sequence of coin tosses is more random:

#1 HTHTHTHTHTHTHTHT, or

#2 HTHTTTHHTTTHHHTT?

Is it #1? But if I flip a coin sixteen times, #1 and #2 are exactly equally likely! They both have 2^{-16} probability! But, you say, #2 has a pattern while #1 doesn't! Well then, what do we mean by a "pattern"?

A modern answer to this question is that #2 is describable by a short computer program. Let x be an n-bit string, and fix a programming language P (such as C, Python, or Turing machines). Then $K_{\mathrm{P}}(x)$ is the length of the shortest P-program whose output on a blank input is x. That is, $K_{\mathrm{P}}(x)$ is the "best possible computable compression" of x.

First claim: For every n-bit string x, we have

$$K_{\mathrm{P}}(x) \leq n + C_P,$$

for some constant C_P depending only on P. Why? Because there's always the program

print "HTHTTTHHTTT..."

But some x's have much smaller K_{P}—e.g.,

```
1  for i = 1 to n
2          print "HT"
```

for the "HTHTHTHT..." string. Since the above program encodes only the positive integer n plus a constant amount of additional stuff, we have

$$K_{\mathrm{P}}(\mathrm{HTHT}\ldots\mathrm{HT}) \leq C + \log_2 n,$$

where n is the number of repetitions.

Second claim: For *almost all* n-bit strings $x \in \{0,1\}^n$ (in particular, for at least a $1 - 2^{-C}$ fraction of them), we have $K_{\mathrm{P}} \geq n - C$.

Why? Because of yet another counting argument. There are 2^n different n-bit strings, but at most $2^{n-C} - 1$ different programs of fewer than $n - C$ bits, and each program can generate at most one string. So, there just aren't enough short programs to go around!

Next question: How much could the choice of P matter?

Interesting Theorem. *For all programming languages P and Q (such as C++ or Python), there's a constant $C_{\mathrm{P,Q}}$, depending only on P and Q, such that $|K_{\mathrm{P}}(x) - K_{\mathrm{Q}}(x)| \leq C_{P,Q}$ for all x.*

What's the proof? It's simply to write a P-interpreter in Q, and a Q-interpreter in P! This is what justifies being sloppy, and writing just $K(x)$ rather than $K_P(x)$.

Homework 5.2. Show there's no algorithm that given x, computes $K(x)$. *Hint:* What strategy should you use? You *could* try to show that given an oracle for K, you could solve the Halting Problem. This turns out to be possible, and it *does* imply that K is uncomputable, but I don't recommend it, because it's hard! An easier way: show that if K was computable, you could have a program (let's say) n bits long that found and output a string with Kolmogorov complexity $2n$—and that would be a contradiction, because if the Kolmogorov complexity is $2n$, then that's how long the program needs to be!

Finally, here's the most amazing Kolmogorov complexity fact of all. Pick a random string x with, say, 10000 bits. Then with overwhelming probability, $K(x) \approx 10000$ *but that fact is unprovable in ZFC.* (Proof: extra credit for you!)

Last time, we touched on Gödel's Incompleteness Theorem, when we talked about how the value of BB(650) is now known to be independent of ZFC. It would take us too far afield to state and prove Gödel's Theorem as Gödel himself did it in 1931.

But once we have Turing's theorem about the uncomputability of the Halting Problem, there's an excellent version of Gödel's Incompleteness Theorem that falls out basically for free. Namely, I claim there's no system for proving theorems such that

1. Whenever a proof is valid, the theorem is true. ("Soundness.")

2. Whenever a Turing machine runs forever, there's a valid proof of that fact (so in particular, there are proofs of Goldbach's Conjecture, the Riemann Hypothesis, etc.). ("Completeness.")

3. There's a computer program to check whether proofs are valid. ("Computability.")

Why can't such a system exist? Because if it did, we could use it to solve the Halting Problem! Given a Turing machine M, we would alternate between running M and examining possible proofs that M runs forever, secure in our knowledge that one of these processes must eventually halt, and when it does, we'll know the answer. But we already know the Halting Problem is uncomputable, so no such proof system can exist.

Lecture 6: Complexity, or Number of Operations

I hope you enjoyed last week's tour of various models of computation (circuits, finite automata, Turing machines), and of what they can and can't compute! But the truth is, when people started building and using *actual* computers 60 or 70 years ago, they quickly realized that *what can be computed* often isn't the most important question. Often it's *obvious* that something can be computed—the problem is "only" that any program we know how to write would take longer than the age of the universe!

Example. Factoring a number into primes, like $21 = 3 \times 7$ or $1591 = 37 \times 43$ or $13,589 = 107 \times 127$.

How do we know factoring is computable? Right, given N you only need to check possible divisors from 2 up to $N - 1$—or better yet 2 up to $\sqrt{N}$. (Why?)

But even with the fastest factoring algorithm anyone has published running on the biggest supercomputer anyone has built, factoring a $10,000$-digit number would in general take longer than the age of the universe.

Why do we care about factoring so much? Well, have any of you heard of RSA? Right, that's the secret code that (along with its cousins) protects almost the entire internet—chat messages, your parents' credit card numbers. And it depends on factoring huge numbers being a hard problem. If you discovered a fast way to factor 5000-digit numbers, you could read most of the secrets on the Internet (and among other things, steal tens of billions of dollars' worth of Bitcoin left by Bitcoin's inventor Satoshi Nakamoto!)

More generally, secret codes are one *major* reason why people care about how much time problems take to solve, and we'll come back to them in a few days.

For now, though, I simply want to talk about how could there be faster or slower, better or worse, algorithms for the same problem. It's easiest if we see some examples.

Probably some of you have played the game "I'm thinking of a number." So, I'm thinking of a whole number from 1 to 1000. You can ask me yes-or-no questions like "is your number 35?" and also like "is it bigger than 100?" Your goal is to learn my number by asking me as few questions as possible. So, let's play!

What would be a bad strategy in this game? "Is it 1?" "Is it 2?" "Is it 3?" Only slightly better: "Is it less than 10?" "Is it less than 20?" "Is it less than 30?" And then: "21?" "22?" "23?"

Why are these strategies bad? The first one could take 1000 questions in the worst case (500 on average). Even the second one takes 110 questions in the worst case (55 on average). Even worse though is how these strategies *scale*. If my number was from 1 to a million, I'd need a million or 100000 questions. Every time I double the range of the numbers, I double how many questions you need to ask.

So what's a better strategy?

Is it more than 500?

Is it more than 750?

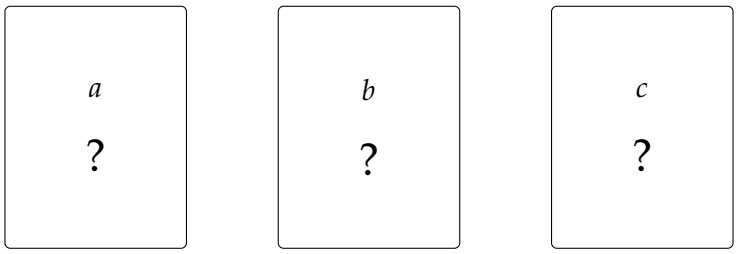
Is it more than 625? etc.

This is called *binary search*. How many questions do you need to ask with *this* strategy? Only about 10 questions. Why 10? Because $2^{10} = 1024 > 1000$. Good way to think about it: with each question you learn another bit of the number's *binary representation*.

Binary search has excellent *scaling*: when you double the range of the numbers (say, from 1000 to 2000) the number of questions *doesn't* double; in fact it only increases by one! How many questions does binary search require to determine a number from 1 to N? $\log_2 N$. The power two has to be raised to to reach N (or the next integer close to that).

Is binary search *the best possible* strategy in guess-the-number? What do you think? Well I claim that *any* strategy will need to ask at least $\log_2 N$ questions. Why? because the secret number is $\log_2 N$ bits long, and each yes-or-no question can only supply one new bit of information. So yes: binary search is optimal!

Now let's try a more interesting example. On the back of each these three cards, I've written a number.



You may compare any two cards.

You can point to any two cards and ask me which number is bigger, and I'll tell you. Your goal: sort the numbers from smallest to largest, by asking as few questions as possible.

Okay, so it's possible to do it with three questions. Is there a strategy that always wins and that asks me only two questions? No? Why not?

Right: after question one, we have (let's suppose) $a < b$. Then we need to compare c to either a or b. But if $a < c$ or $c < b$, then we need to do a third comparison.

Now how about with four cards? Here I claim that five questions are enough! I.e., you can save one compared to asking all $\binom{4}{2} = 6$ possible questions.

How? MergeSort! Given a, b, c, d, first I sort a, b, c, which we already showed takes 3 comparisons. Suppose for example that the result is $b < c < a$. Then next I use two additional comparisons to place d in the right place: first I compare d to c (the element currently in the middle), and then I compare d to either b or a depending on the result.

Can anyone prove that 5 comparisons are needed?

Here's a cool strategy: how many possible ways are there to sort 4 elements? Why,

$$4! = 4 \times 3 \times 2 \times 1 = 24.$$

Each comparison answers a single yes-or-no question, so gives one bit of new information about the correct order. How many bits do we need? $2^4 = 16$ and $2^5 = 32$, so $4 < \log_2 24 < 5$. So we need at least 5 bits and hence 5 comparisons.

Now let's generalize to *any* number of cards! If we compared every card to every other card, how many comparisons would we make? $\binom{n}{2} = \frac{n(n-1)}{2}$, which grows like n^2. What if we use MergeSort?

$$\lceil \log_2 1 \rceil + \lceil \log_2 2 \rceil + \lceil \log_2 3 \rceil + \cdots + \lceil \log_2 n \rceil \leq n \lceil \log_2 n \rceil$$

So it grows like $n \log n$, which is better than n^2 for sure!

But is that optimal? Could there be a way to sort n cards, using a number of comparisons that only grows like n times a constant ($5n$, $10n$, etc.)? No? Why not?

How many possible sortings are there? Right,

$$n! = n(n-1)(n-2)\cdots 3 \cdot 2 \cdot 1 \geq (\frac{n}{2})^{n/2}.$$

So

$$\log_2(n!) \geq \log_2((\frac{n}{2})^{n/2}) = \frac{n}{2} \log_2 \frac{n}{2}$$

which grows roughly like $n \log n$.

Alternatively, *Stirling's Approximation* says that as n gets large,

$$n! \approx \sqrt{2\pi n}(\frac{n}{e})^n.$$

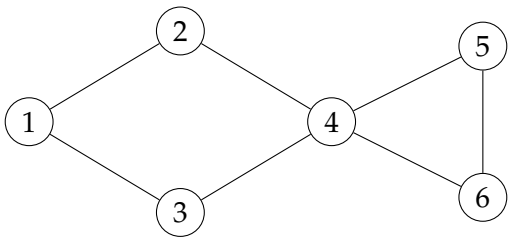
(Yes, really! Try plugging in some n's and see!) In this formula π is pi and $e = 1 + \frac{1}{2!} + \frac{1}{3!} + \cdots \approx 2.71828\ldots$ is the base of natural logarithms. The logarithm of $\sqrt{2\pi n}(\frac{n}{e})^n$ grows like $n \log n$, so again, any method to sort n cards must use a number of comparisons that grows like $n \log n$.

(Don't worry if you didn't follow these calculations.)

Homework 6.1. Suppose you have two identical eggs, which can be dropped at any integer height from 1 to 100. They'll break if dropped from an unknown height X or higher. Once an egg breaks, that egg can no longer be used. Give a strategy to determine X, using as few egg drops as possible (in the worst case).

Homework 6.2. Show that Euclid's gcd algorithm – wherein you repeatedly replace $\gcd(a, b)$ (for $a < b$) by $\gcd(b \mod a, a)$ until a divides b—manages to calculate the gcd of two n-bit integers using a number of iterations that's at most linear in n. (Hint: Show that the "worst possible example," the slowest one to converge, involves the Fibonacci sequence.)

Alright, here's another famous example of a basic algorithm that I can't resist showing you. How many of you use Google Maps? How many of you know what a graph is?



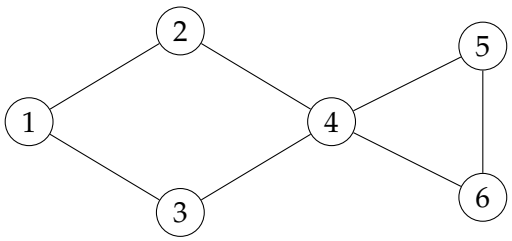
A graph is just a collection of points, called *vertices*, some of which are connected to each other by links, called *edges*. You could specify a graph to a computer program via its *adjacency matrix*: that is, a table where the rows and columns correspond to the vertices, and there's a 1 wherever there's an edge between two vertices and a 0 otherwise. For example, the graph above would have the following adjacency matrix:

	1	2	3	4	5	6
1	0	1	1	0	0	0
2	1	0	0	1	0	0
3	1	0	0	1	0	0
4	0	1	1	0	1	1
5	0	0	0	1	0	1
6	0	0	0	1	1	0

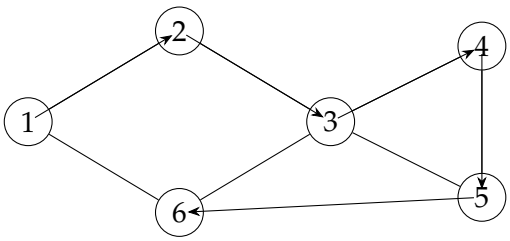
Given an n-vertex graph, one of the most basic questions you could ask is, *is it connected*? In other words, is every vertex reachable from every other? How could you decide that? Intuitively, pick some vertex (say the first one) and then start exploring, counting how many vertices you've reached so far. If you ever reach all n of them, the graph is connected. If you don't reach n vertices . . . well, how do you know when to stop? You can stop when there are no new vertices left to explore. But how do you know if you've tried all possibilities? After all, there are exponentially many possible paths! So you need to explore the vertices in some systematic order. There are two famous ways to do this: *breadth-first search* and *depth-first search*.

Breadth-first search means that, starting from the first vertex, you first visit all of its neighbors, then you visit all of *their* neighbors, and so on, making expanding bubbles, and keeping a list of all the vertices that have already been visited so you never need to visit the same vertex twice. Depth-first search, by contrast, means that you keep visiting a neighbor that hasn't yet been visited, for as long as you can until you get stuck, and then you retrace your steps until you find a path that you didn't try earlier, and then you explore *that* path until you get stuck, and so on recursively until you've visited every vertex you possibly can. Of course this *also* requires keeping a list of which vertices have already been visited and which haven't.

BFS (from 1)



DFS (one possible run)



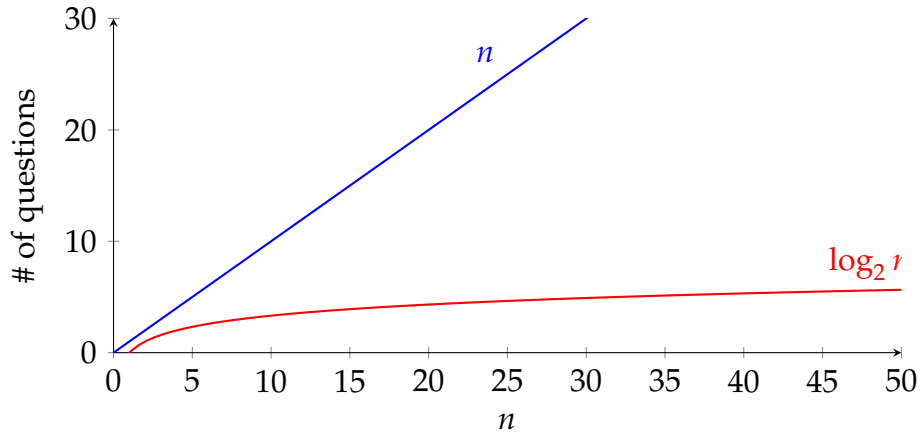
Crucially, both breadth-first search and depth-first search use a number of steps that's only linear in the size of the graph—if, for example, the graph is given as an $n \times n$ adjacency matrix, then $O(n^2)$ steps. Why? Because they never need to traverse

the same edge more than once! That's the point of remembering which vertices have already been visited—like Hansel and Gretel leaving breadcrumbs.

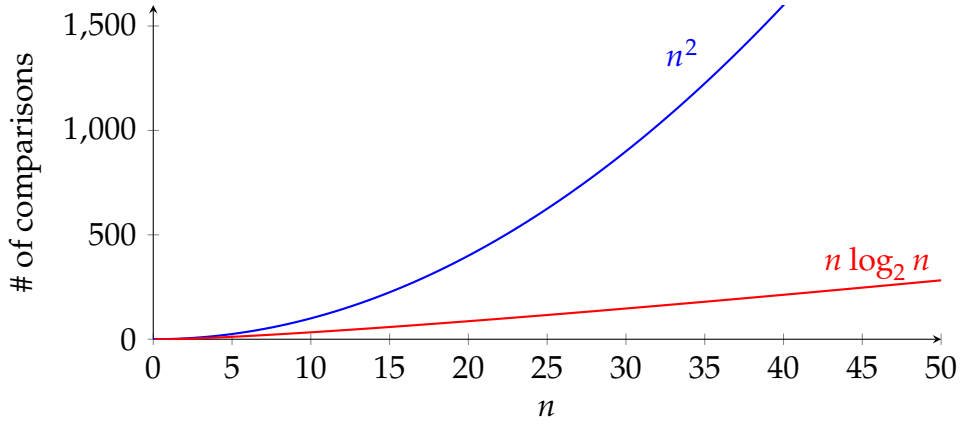
You can easily adapt these algorithms to, for example, find the *shortest* path between two vertices, as hundreds of millions of people ask Google Maps to do every day, or lots of other related tasks. There's much more to say about graph search (and you'll learn it in any introductory computer science class), but that's all we'll need for an application that we'll make of it tomorrow.

Lecture 7: Polynomial or Exponential

Yesterday, we saw how binary search has better scaling than the dumb linear search strategy, in the number of yes/no questions needed to learn an unknown integer from 1 to n:



And likewise, how MergeSort has better scaling than the dumb strategy ("BubbleSort") in the number of comparisons needed to sort a list of n items:



But the truth is, there's one difference in growth rates that we care about more than any other in computer science. It's the difference between *polynomial growth* and *exponential growth*. Who knows what *polynomials* are? Functions like $f(n) = n^2 - 3n + 5$. A sum of terms, where each term involves n to some power. In computer science, if n is the number of input items, a *polynomial-time algorithm* is a one that uses a number of steps that grows at most like n raised to some power.

Examples:

- n^2 is a polynomial.
- $n \log n$ is *not* a polynomial, but there are powers of n that it's less than like n^2 or even $n^{1.1}$ or $n^{1.001}$, once n gets large enough. (Why?) So we still count it as polynomial time.

- $n^{\log n}$ is "slightly superpolynomial." It grow faster than any polynomial but slower than exponentials.
- 2^n and 3^n are of course exponential.
- In computer science, we also count 2^{n^2} and $2^{\sqrt{n}}$ (for example) as "exponential growth." Likewise $n!$, which grows like $\exp(n \log n)$.
- There are "half-exponential" functions—functions f such that $f(f(n))$ grows like 2^n—although there are provably no closed-form formulas for such functions. (I won't prove that here.)

Adding two numbers that are n digits long—how does the number of steps scale?

$$\begin{array}{r} 1 \\ 456 \\ +783 \\ \hline 1239 \end{array}$$

Right, linearly! If there are 6 digits, it takes about twice as long. Remember our circuit for adding two n-bit numbers? It used a number of AND, OR, and NOT gates that grew *linearly* with n (check this!).

What about *multiplying* two n-digit numbers?

$$\begin{array}{r} 123 \\ \times 456 \\ \hline 738 \\ 615 \\ +492 \\ \hline 56088 \end{array}$$

Each digit of the first number needs to be multiplied by each digit of the second! The total number of one-digit multiplications is n^2, and we also need $\sim n^2$ one-digit additions. So this is a quadratic or n^2-time algorithm. Double n and it takes about 4 times as long.

Is multiplying *inherently* harder than adding? In the 1960s, a student named Karatsuba discovered a clever way to multiply n-digit numbers using only $\sim n^{\log_2 3}$ steps. Recall that $\log_2 3 \approx 1.585$. So, less than quadratic but more than linear!

Here's the idea: let X and Y be two n-digit numbers that we want to multiply, and assume for simplicity that $n = 2^k$ is a power of 2 (if it isn't, we can just pad X and Y out with leading zeroes). We then break up X as $10^{n/2}A + B$ and Y as $10^{n/2}C + D$, where A, B, C, D are numbers with $n/2 = 2^{k-1}$ digits each. For example, if $X = 6742$ and $Y = 9381$, then $A = 67, B = 42, C = 93, D = 81$.

Then

$$X \times Y = (10^{n/2}A + B) \times (10^{n/2}C + D) = 10^n AC + 10^{n/2}(AD + BC) + BD.$$

So far this doesn't seem to help! But now notice that

$$(A + B) \times (C + D) = AC + AD + BC + BD$$

so

$$(A + B) \times (C + D) - AC - BD = AD + BC.$$

So to summarize, by computing multiplying just *three* pairs of $n/2$-digit numbers—namely, AC, BD, and $(A + B)(C + D)$—and then doing the above subtraction, we can get all three of the terms AC, BD, and $AD + BC$ that we need to express $X \times Y$.

OK, but how do we do the three $n/2$-digit multiplications? By running the same algorithm recursively! In other words, we do each $n/2$-digit multiplication by breaking it up into three $n/4$-digit multiplications, and then putting the pieces back together. We do each $n/4$-digit multiplication by breaking it up into three $n/8$-digit multiplications. And so on.

Is this actually a win? When n is small, it just makes everything more complicated and confusing than using the grade-school method! But I claim that the bigger n gets, the more this new method wins out over the grade-school one. Why? Well, let $T(n)$ be the total number of arithmetic operations that we use to multiply two n-digit numbers using Karatsuba's procedure. Then we get

$$T(n) \leq 3T(n/2) + cn$$

for some constant c, since the additions and subtractions take time linear in n. And then, of course, once n becomes small enough, we can just revert to the grade-school method, using a constant number of operations.

As I invite you to check, the solution to this has the form $T(2^k) \sim 3^k$, or in other words, $T(n) \sim n^{\log_2 3}$, as promised.

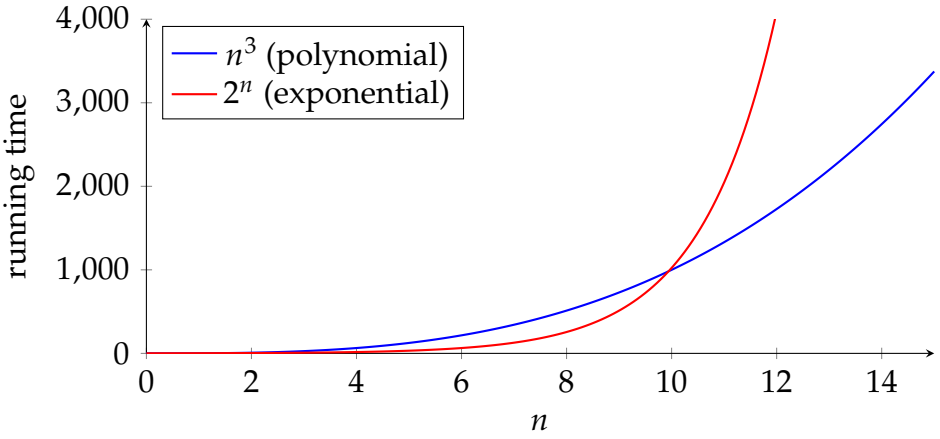
Today, there are super-advanced methods for multiplying n-digit numbers using only about $n \log n$ steps (just like for sorting), which is very close to linear in n. Can it be exactly linear? Still no one knows!

By contrast, as we said before, an exponential-time algorithm uses a number of steps that grows like 2^n or 3^n.

Example. We saw one yesterday! *Factoring* an n-digit number X into primes (the problem so much encryption is based on). If we use trial division how many steps does this take? It could be anywhere from 0 to $2^n - 1$. So if we go all the way up to X, then up to $\sim 2^n$ trial divisions, each taking $\sim n$ or $\sim n^2$ time. If we only go up to $\sqrt{X}$, then "only" $\sqrt{2^n} = 2^{n/2}$ trial divisions. This is still exponential!

Tomorrow we'll see more examples of problems where the fastest known algorithms are exponential.

What's wrong with exponential growth, and why do we like polynomial growth so much more?



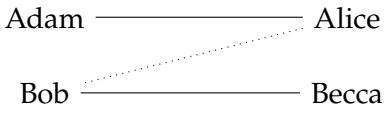
It's true that, for reasonable values of n (like 1000), the "polynomial" running time $n^{1000000}$ is much, *much* larger than the "exponential" running time 1.000001^n. But, we usually don't see running times like those! Usually, they're more like n^2, n^3, or 2^n. So polynomial vs. exponential is a rough way to classify which algorithms are "efficient" and which are "inefficient."

Is it easy to tell which problems have polynomial algorithms and which require exponential ones? Let me show you one of my favorite examples.

We have a list of men and another list of equally many women. Our goal is to marry everyone off. But how? Each man has ranked off the women in order of how much he likes them (no ties), and each woman has likewise ranked all the men. For example:

Adam:	Alice > Becca > Claire	**Alice:**	Bob > Adam > Charlie
Bob:	Alice > Claire > Becca	**Becca:**	Adam > Bob > Charlie
Charlie:	Becca > Alice > Claire	**Claire:**	Adam > Charlie > Bob

Our goal is a *stable marriage*: one when no man and woman who are *not* married to each other, both prefer each other to their spouses. For instance



is unstable. Why? Right, because Alice prefers Bob to Adam, *and* Bob prefers Alice to Becca.

First question: does a stable marriage always exist? Second question: if so, how do we find it?

Let's actually tackle the second question first! An obvious strategy would be to try every possible pairing, looking for one that's stable. How many ways are there to marry off three men and three women? Right! Six! How about four men and women? 24. In general, with n men and women, there are

$$n! = n \times (n-1) \times \cdots \times 3 \times 2 \times 1$$

ways. And remember, $n!$ grows like $\sqrt{2\pi n}(\frac{n}{2})^n$, or more roughly like n^n. We call this exponential time—at any rate, it's certainly not polynomial time. It's a brute-force

algorithm, which with a hundred men and women would take longer than the age of the universe.

Can anyone suggest a faster method? Just try pairing them and see what happens? Let me show you a method that only takes $\sim n^2$ steps. Note that $\sim n^2$ is the amount of data you've been given ($2n$ people each have a preference list of size n), so this actually a *linear*-time algorithm, in terms of the amount of input. And as a byproduct, this algorithm will prove that a stable marriage always exists . . . by *finding* it!

The idea is to do what would happen in a romantic comedy. In the old days, when this algorithm was published, men proposed to women, although of course the algorithm would work equally well the other way. So let's start with Adam. Who does he propose to? His first choice, Alice. Does Alice accept Adam? *For now.* But what happens next? Bob's turn to propose! Who does he propose to first? Also Alice! Who does Alice prefer? She prefers Bob! So now Alice and Bob are together and Adam is kicked out. Adam is single again. We'll deal with him later. For now, let's move on to Charlie. Who does he propose to? Becca, who accepts Charlie for now.

Now we circle back to Adam. Who does Adam ask next? Right, Becca! Becca now has to choose between Adam and Charlie. She chooses . . . right, Adam! So Charlie is single again.

Charlie next proposes to Alice, who says "no chance." So he moves on to Claire, who accepts him. And we're done. Is this a stable marriage? Yes.

Question. Will this procedure always finish, or could it get into an infinite loop?

Answer. It will always finish. Why? Because each man asks each woman at most once!

Question. How long does it take to finish?

Answer. At most n^2 steps—the number of men times the number of women. So this is a polynomial (and even linear) time algorithm.

Question. When it finishes, is everyone always paired off?

Answer. Yes! Why? Suppose a man and woman were still single. Eventually, that man would ask that woman and if she was single she'd say yes and she'd stay with him.

Question. Will the pairing be *stable*?

Answer. Yes! For suppose not. Then some man and woman *both* prefer each other to their spouses. But then that man would've asked that woman before his spouse and she would've chosen him over *her* spouse. So this never happens, QED.

By thinking about it, we improved $n!$ steps to n^2, *and* proved that a solution always exists!

Lecture 8: The P vs. NP Problem

Today I want to tell you about the biggest unsolved problem in all of theoretical computer science—maybe even the biggest unsolved problem in all of math.

It's about what we talked about yesterday: which problems are solvable by some method that uses *polynomial time* (that is, a number of steps that grows at most like the size of the input raised to some fixed power), and which problems require *exponential time*.

In computer science, we like to organize problems into classes, and to give those classes names involving capital letters. Maybe our most important class is called P which stands for Polynomial Time. P is basically just the class of all the problems that are solvable by some polynomial-time algorithm (linear, quadratic, etc.) Technically, they have to be decision problems—that is, problems with a yes-or-no answer—although sometimes we get sloppy and ignore that. The algorithm has to work for every input, and it has to work on a standard computer—that is, a Turing machine—not a quantum computer or anything else.

What are examples of problems in P?

Palindromes, from before! This takes quadratic time on a Turing machine, though only linear time on a more realistic computer.

Truthfully, most of what we do with computers on a day-to-day basis is in P. All of basic arithmetic, for example. Or, finding the shortest route between two points on a map (what Google Maps does), or let's say, deciding whether it's *possible* to escape from a given maze. And some much less obvious examples:

- Reducing a fraction to lowest terms, *e.g.* $\frac{8}{12} \rightarrow \frac{2}{3}$. (The polynomial-time algorithm was given by Euclid in 300 BC.)
- Deciding whether a number is prime—that was only proven to be in P in 2002! (Note: I did *not* say factoring the number, if it turns out to be composite!)
- Stable Marriage. Well, there's no decision problem here. The answer is always "yes, a stable marriage exists." But related problems are in P, like "given which men and women are willing to marry each other, can everyone be paired off with someone they like?"

Besides P, there's another extremely important class called NP. What do you think NP stands for? Not Polynomial? It actually stands for Nondeterministic Polynomial Time. NP contains all the problems for which if the answer is "yes," then a solution can be *checked* in polynomial time. In other words, there's a proof, which is a string with a polynomial (linear, quadratic, etc.) number of bits. And there's a verifier, which is a polynomial-time algorithm. If the answer is yes, then *some* string must make the verifier output "yes." If the answer is no, then *no* string must make the verifier output "yes."

What are examples of problems in NP?

We already saw one: FACTORING! Or to phrase FACTORINGas a decision problem: given, say, an integer N written in binary, does N have a prime factor ending in a 1? Does N have at least three prime factors?

Why are these problems in NP? If the answer is yes, you can prove it by just giving the prime factorization. Multiplying the purported factors to check that you get N is in P! And if you need it, there are also polynomial-time algorithms to check that the factors are indeed prime.

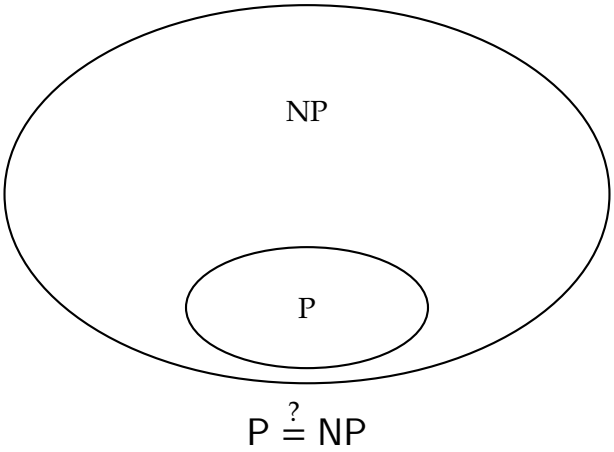
But there are many other examples of NP problems:

- GRAPHISOMORPHISM: Given as input two n-vertex graphs G and H (say, via lists of their vertices and edges), are G and H equivalent under some relabeling of the vertices?
- CLIQUE: Given a graph G and a positive integer k, does G contain at least k vertices that are all connected to each other by edges?
- TRAVELINGSALESMAN: Given a list n cities, the travel time between each pair of cities, and a total travel budget B, is there a way to visit all of the cities in at most B hours?
- PACKING: Given the dimensions of each of a set of cubical boxes, and the dimensions of the trunk of your car, can all the boxes be fit into the trunk?
- 3-COLORING: Given an n-vertex graph G, is there a way to color each vertex red, green, or blue, in such a way that no two neighboring vertices are colored the same?
- CIRCUITSAT: Given a Boolean circuit C (say, made of AND, OR, and NOT gates), with n input bits and 1 output bits, is there a setting x of the input bits that causes the output $C(x)$ to be 1?
- 3-SAT: Given a list of n Boolean variables $x_1, \ldots, x_n$, together with a list of constraints each of which involves at most 3 of the variables (for example, $x_1 \vee x_2 \vee \overline{x_3}$, saying that if x_1 and x_2 are both TRUE then x_3 must be FALSE), is there a setting of the variables that satisfies all of the constraints?

For each of these problems, if the answer is "yes, a short witness exists" then it's easy to check the witness. But it's not obvious how to *find* a witness, or decide whether there is one, in less than exponential time.

I claim that $\mathsf{P} \subseteq \mathsf{NP}$. (Why? Because, if a problem is in P, then it satisfies the criterion for being in NP even if you leave the witness empty, and have the verifier just solve the problem for itself.)

The literally million-dollar question is: does $\mathsf{P} = \mathsf{NP}$, i.e. is $\mathsf{NP} \subseteq \mathsf{P}$? This is one of the seven Clay Millennium Problems, for which a solution carries a $1M prize from the Clay Mathematics Institute, but that's honestly the least of it.



Suppose P = NP via a "practical" algorithm (e.g. n rather than n^{100}). Then we could program our computer to solve all the other Clay problems! For we could ask, for example: "Is there a proof of the Riemann Hypothesis in ZFC set theory, that's at most five hundred million symbols long?" If such a proof existed, then it could quickly be checked—which means that, in this P = NP world, it could also quickly be found! Essentially this was pointed out by Kurt Gödel in a letter to John von Neumann in 1956. If P = NP then, as Gödel put it, "the mental effort of the mathematician could be completely replaced by machine, apart from the postulation of axioms."

What else could you do in the P = NP world? You could mine *all* the remaining Bitcoin. In fact you could break essentially all the encryption that protects the Internet—even the kinds we believe to be safe against quantum computers. I'll discuss this more in a future lecture.

What else? Could you solve the halting problem or compute Kolmogorov complexity? No, those would still be uncomputable (after all, we proved that unconditionally). But you could find the shortest program that *quickly* produces all the text on the Internet. In other words, you could do the perfect version of what OpenAI, DeepMind, Anthropic, and the other AI companies are doing today—and plausibly without requiring *anywhere* near as much computing power.

What do we believe? The majority of computer scientists believe that P ≠ NP. A few say they believe that P = NP, but it's not clear whether they're just trying to troll the rest of us. I've personally given a 97% chance that P ≠ NP. Others give 75% or 99.9%. I like to say that if we were physicists, we would've declared P ≠ NP a law of nature, and given ourselves Nobel Prizes for its discovery! And if it later turned out that P = NP, we'd give ourselves more Nobel Prizes for the law's overthrow. It's only because theoretical computer scientists are basically mathematicians that we need to call this an open problem.

Why, then, is it so hard to prove P ≠ NP? There's an enormous amount one can say about this question—a decade ago, I wrote a 120-page survey article about it!—but one short answer would be, there really *are* many clever efficient algorithms. "Obviously" testing primality requires exponentially slow trial division... except that it doesn't! "Obviously" it takes n^2 time to multiply two n-bit integers... except that it doesn't! "Obviously" Stable Marriage requires . . . well, you get the idea.

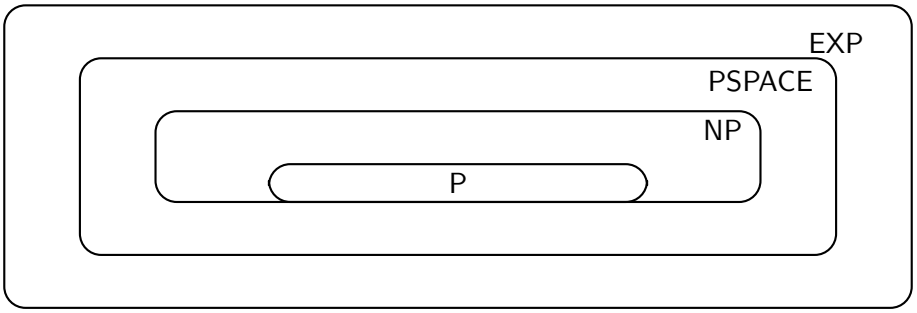
Every week, I get emails from people who think they've proved P ≠ NP. They pick some NP problem, and give a complicated argument that it requires exponential time, which always has a step of the form "there are 2^n possibilities, and it takes constant time to check each one, therefore obviously we need 2^n time." And it's like, dude! If that argument worked it would also show that stable marriage required $n!$ time! But it *doesn't* require $n!$ time.

Homework 8.1. Show that if P = NP, then you can *find* solutions to yes-instances of NP problems in polynomial time.

Solution. Suppose P = NP and let $L \in$ NP and we know that $x \in L$. First note that if L is in NP and w is a bit-string the language defined by "x is in L and has a witness starting with w is also in NP. We want to find the witness for x being in L. The witness is bounded in size by a polynomial $p(|x|)$ and we need to determine all the bits of the witness w. We proceed in $p(|x|)$ steps. At each step i we first guess that the ith bit of w is 0. We check whether x has a witness starting with the determined bits plus 0 (which is in NP and hence in P by assumption). If yes we choose 0 for the ith bit, if not we put 1. Each of the $p(|x|)$ steps only requires a call to a subroutine which is in NP and hence in P and therefore the whole procedure ends in polynomial time. □

Beyond P and NP, I'll tell you about two other important classes: PSPACE and EXP. PSPACE is the class of all decision problems that are solvable by a Turing machine that runs in polynomial space, but can reuse that space over and over for as much time as needed. EXP, meanwhile, is the class of all decision problems that are solvable by a Turing machine that runs in exponential time, by which we mean $2^{p(n)}$ time for some polynomial p (for example, 2^{n^2} would count).

To illustrate, consider the Chess problem of deciding whether White has a win from a given position on a chessboard. To make this interesting, we first need to generalize chess from an 8×8 board to an $n \times n$ board, with as many pieces as desired—since otherwise the problem will be solvable in *constant* ($O(1)$) time, with all the astronomical costs hidden in the constant! Once we generalize chess to $n \times n$, we then face a further choice: namely, do we limit the games to last for at most $p(n)$ moves for some polynomial p (which models what might happen in real tournament play with timers), or do we let games continue for an exponential in n number of moves? It turns out that the Chess problem is known to be in PSPACE under the former choice, but is only known to be in EXP under the latter choice. (In fact, the two versions of Chess are PSPACE-complete and EXP-complete respectively—concepts that we'll define tomorrow!)



Meanwhile, here's what we know about the relationships among these classes:

$$\mathsf{P} \subseteq \mathsf{NP} \subseteq \mathsf{PSPACE} \subseteq \mathsf{EXP}.$$

Meanwhile, the only *proven* separation among these classes is $\mathsf{P} \neq \mathsf{EXP}$. This follows from the famous "Time Hierarchy Theorem": a sort of scaled-down version of the unsolvability of the halting problem, which shows (for example) that there are problems solvable in n^3 time but not in n^2 time, and other problems solvable in n^4 time but not in n^3 time, and so on for every pair of reasonable time bounds. (What *are* those problems? They tend to look like: "given as input a description of a Turing machine M as well as an integer n, does M halt in at most n^2 steps when started on a blank tape, or not?") More generally, proving separations between classes tends to be extremely easy when you're comparing different amounts of the *same* resource—n^2 time versus n^3 time, for example, or 2^n space versus 3^n space—and extremely hard when you're comparing *different* resources, like P versus NP or P versus PSPACE or NP versus PSPACE.

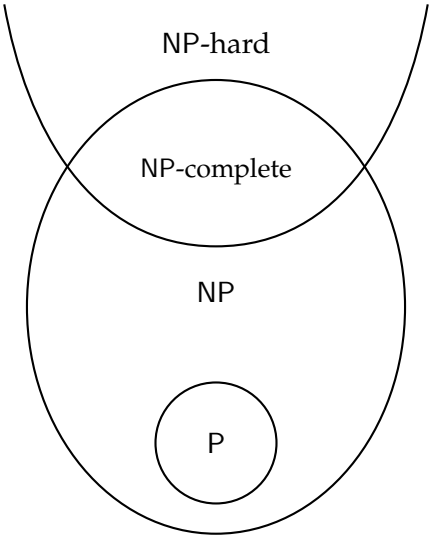
Anyway, from the Time Hierarchy Theorem ($\mathsf{P} \neq \mathsf{EXP}$), we reach an amusing conclusion about the list of containments above: namely, that either $\mathsf{P} \neq \mathsf{NP}$ *or* $\mathsf{NP} \neq \mathsf{PSPACE}$ *or* $\mathsf{PSPACE} \neq \mathsf{EXP}$ (since otherwise we'd have $\mathsf{P} = \mathsf{EXP}$). We conjecture that all three of these separations are true, but to this day, we can only prove that at least one of them is!

Lecture 9: NP-completeness

There's one further concept that plays a central role in this story: NP-*completeness*.

First, remember the concept of an *oracle* for the set A, which tells you for free whether any given input x belongs to A? By P^A, we mean the class of all problems solvable in polynomial time given access to an oracle for A. Then, we say that a problem A is NP-*hard* if $\mathsf{NP} \subseteq \mathsf{P}^A$: in other words, if an oracle for A would let you solve any NP problem in polynomial time. It shouldn't be obvious that NP-hard problems even exist! But as we'll see, many do.

Next, a problem is NP-*complete* if it's both NP-hard and *in* NP. I.e., NP-complete problems are the hardest problems in NP.



The huge discovery of the 1970s was that tons of practical problems turned out to be NP-complete. For starters, there's what I call the "DUH" problem. Given $\langle M \rangle$ and a string of T 0's is there a T-bit string that M accepts in less than or equal T steps? the DUH problem is clearly in NP and is NP-hard essentially by definition. (Why?)

But once we have *any* NP-complete problem A we can prove other problems NP-complete by reducing A to them.

Example. CircuitSAT. Given as input a description of a Boolean circuit C mapping n bits to one, is there an n-bit string x such that $C(x) = 1$?

Clearly CircuitSAT $\in$ NP. (Why?) To show that CircuitSAT is NP-hard (and, therefore, NP-complete), we show DUH $\leq_{\mathsf{P}}$ CircuitSAT: in other words, DUH is polynomial-time reducible to CircuitSAT, or in P with a CircuitSAT oracle. This just involves building a circuit to simulate the actions of a Turing machine with at most polynomial blowup! Pretty boring to be honest.

Next let's consider 3Sat. Recall that this is where you're given n Boolean variables, along with a list of constraints affecting at most three variables each, and your goal is to figure out whether the variables can be set so that all the constraints are satisfied. Here's an example of a 3Sat instance:

$$a \vee b \vee c, \bar{a} \vee \bar{b}, \bar{b} \vee \bar{c}, \bar{a} \vee \bar{c}.$$

where $\bar{a}$ means NOT(a). Can this be satisfied? Sure it can! For example, by $a = 1$ and $b = c = 0$ (and by what else?).

The famous *Cook-Levin Theorem*, proved in 1971, says that 3Sat is NP-complete. To prove that, we need to show two things:

- 3Sat $\in$ NP, and
- 3Sat is NP-hard.

The first part is easy. For the second part, we show CircuitSAT $\leq_P$ 3Sat: that is, given any instance of CircuitSAT, in polynomial time we can convert it to an instance of 3Sat that has the same answer. Since we already know that CircuitSAT is NP-hard, this proves that 3Sat is NP-hard also.

How do we reduce CircuitSAT to 3Sat? The basic idea is to define a new variable for each AND, OR, and NOT, gate, then add clauses relating each output to its inputs. These clauses will have at most $2 + 1 = 3$ variables each. Finally, enforce that the overall output is true.

Homework 9.1. Show that 2Sat $\in$ P.

Next example: k-coloring of graphs. I claim that 2-Coloring is in P. Why? Right, because once you choose the color of a single country (say, red), your "hand is forced" thereafter: each of its neighboring countries must have the opposite color (say, blue), then each of *their* neighbors must be red, and so on. So, either you 2-color the entire map this way, or else you discover that you can't, and either way you know the answer.

By contrast, 3-Coloring turns out to be NP-complete. We prove this by reducing 3Sat to 3-Coloring which in turn requires building suitable "clause gadgets."

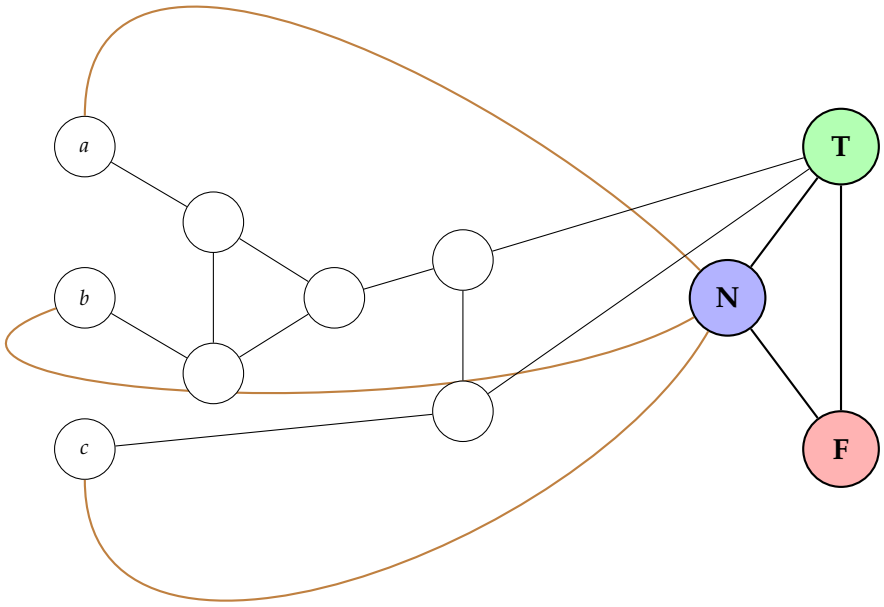


Figure 2: A 3-coloring gadget for the clause $(a \vee b \vee c)$. The literal vertices are connected to N, so each literal must be colored either T or F. The gadget is 3-colorable if and only if at least one of a, b, c is colored T. Internal gadget vertices are not connected to N.

We should also discuss the problem of *coloring* planar graphs—graphs that you can draw on a plane with no intersecting edges. Two-coloring planar graphs is in P, three-coloring planar graphs can be shown to be already NP-complete, but what about *four*-coloring planar graphs? Well, that one is again in P, but this time, only because of the Four-Color Theorem (proved by Appel and Haken in 1976), which says that *every* planar graph is four-colorable! So we can solve the problem in *constant* time, via the algorithm "always output yes."

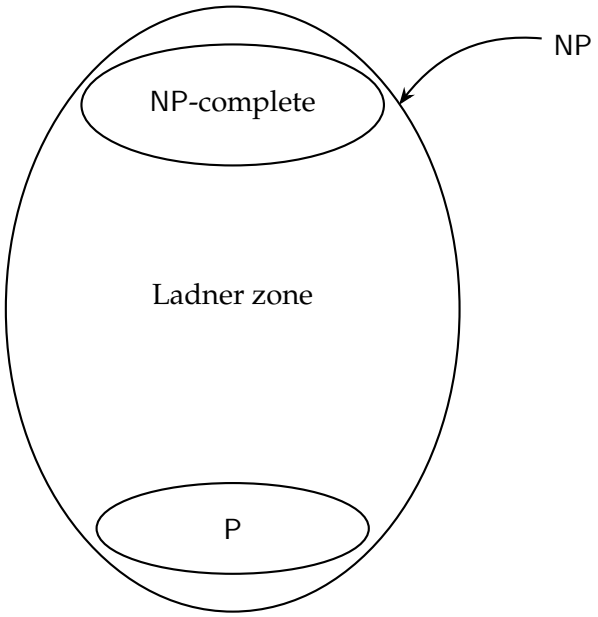
The more problems are known to be NP-complete, the more the floodgates open and the easier it is to show *more* problems are NP-complete.

All NP-complete problems are "secretly the same problem," despite their surface differences, in the following sense. If any of them is in P then P = NP and all the others are in P as well. Conversely, if any NP-complete problem is *not* in P, then P ≠ NP and none of them are in P.

So far, what we know about NP is that

1. it contains P,
2. it contains the NP-complete problems,
3. if P ≠ NP then these two subsets of NP are disjoint from each other, and
4. if P = NP then they're equal to each other and to NP itself.

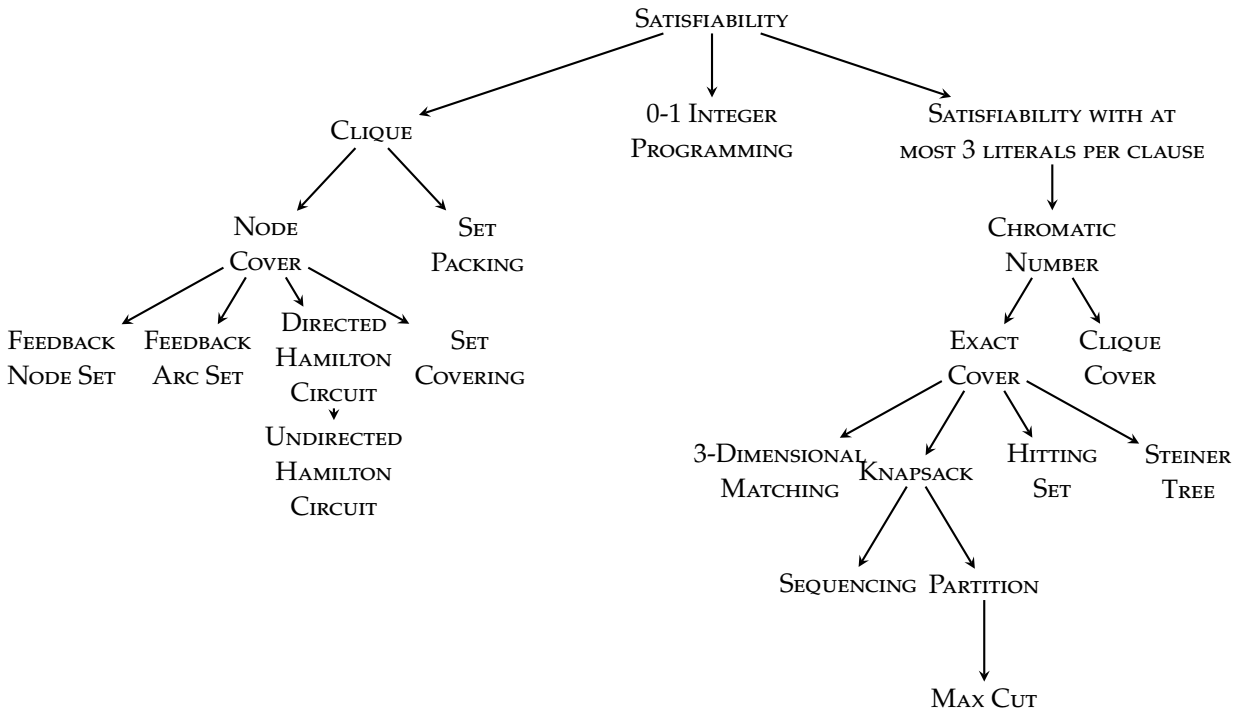
But can we say anything more about case 3? For example, could it be that every NP problem is *either* in P, or else NP-complete: that these are the only two possibilities?



Ladner's Theorem (which we won't prove!) says that, if P ≠ NP, then there must also be a "no-man's land" in between: that is, NP problems that are neither in P nor NP-complete. But the problems produced by the proof are extremely artificial.

Actually, Factoring and GraphIsomorphism are two natural problems conjectured to inhabit this "no-man's land." They're in NP, they're not known to be in P, and they have special structure that *seems* to prevent them from being NP-complete.

Let me end today's lecture by mentioning a core reason why most computer scientists came to believe that P ≠ NP: namely, a phenomenon that I've dubbed the "invisible electric fence."



Karp's 1972 paper on NP-Complete Problems

Today there are thousands of problems proven to be NP-complete, and thousands more problems proven to be in P. If even one had been discovered to be both, that would've meant that P = NP! In other words, the P ≠ NP hypothesis has had thousands of separate opportunities to be "falsified by observation." Yet somehow this has never happened, in more than a half-century of research. You might say it's as though there's an invisible electric fence separating the problems in P, like 2-Coloring, from the NP-complete problems, like 3-Coloring. The P versus NP problem, of course, is to prove or disprove that that fence is real.

Homework 9.2. Consider the following problem: you're given as input a list of integers, such as 36425, and asked to find the *longest increasing subsequence*: in this case, 345. This feels like it might be yet another NP-complete problem. In this case, however, it isn't: the problem turns out to be in P! Why? What's a polynomial-time algorithm to solve it?

Lecture 10: Foundations of Cryptography

People have been making codes for thousands of years, e.g.

FQTJMPO DBNQ SVMFT

Who can decode this? (Hint: it's just a cyclic shift of letters.)

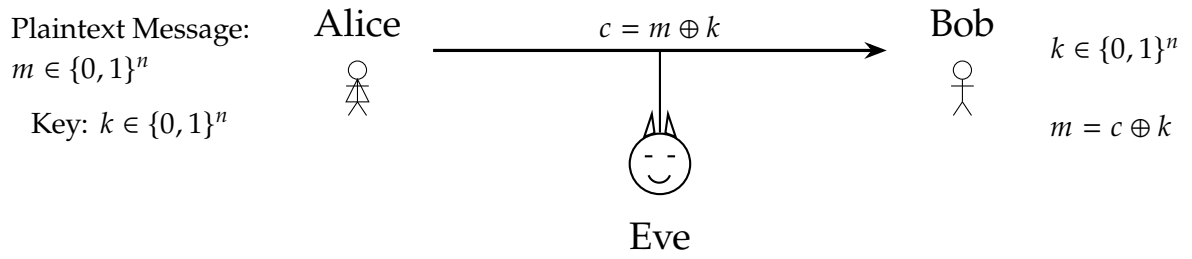
More generally, we have letter substitution ciphers:

$$A \to K \quad B \to M \quad C \to E \quad D \to D \quad \cdots$$

What's the security problem with these? Where to begin? Can you guess what the most frequent letter is?

Yet incredibly, elaborations on this sort of thing were used all the way through World War II—which is why Alan Turing and his friends were able to break the Nazis' Enigma code by building computers to cycle through the possible keys!

It wasn't until 1926—less than 100 years ago—that someone finally invented a cryptosystem that's secure by modern standards if used correctly. This is Vernam's One-Time Pad.



We have a plain-text message $m \in \{0,1\}^n$, an n-bit string. We randomly choose a key $k \in \{0,1\}^n$, another n-bit string, and assume k is shared between the sender Alice and the receiver Bob. Then the ciphertext is $c = m \oplus k$, where $\oplus$ means bitwise XOR (i.e., addition in binary with without carrying). Then, to decrypt, Bob simply computes $m = c \oplus k$: decryption is the same operation as encryption!

For example if the message is `100101` and the key is `011101`, we have

$$\begin{array}{cccccccc} & m & 1 & 0 & 0 & 1 & 0 & 1 \\ \oplus & k & 0 & 1 & 1 & 1 & 0 & 1 \\ \hline & c & 1 & 1 & 1 & 0 & 0 & 0 \end{array}$$

But each key *must* be used only once! Why is this so important? Suppose we encrypted two messages using the same key: $c_1 = m_1 \oplus k$ and $c_2 = m_2 \oplus k$. What could the eavesdropper Eve do, after seeing c_1 and c_2? Right, she could calculate

$$\begin{aligned} c_1 \oplus c_2 &= (m_1 \oplus k) \oplus (m_2 \oplus k) && (1) \\ &= (m_1 \oplus m_2) \oplus (k \oplus k) && (2) \\ &= m_1 \oplus m_2, && (3) \end{aligned}$$

where line (3) used the fact that any k bitwise-XORed with itself is just the all-0 string.

Why is this bad? Right, because then Eve can learn something about m_1 and m_2! She doesn't immediately get m_1 and m_2 themselves, but even learning their XOR could be enough to cause trouble.

Why? Well, imagine m_1 and m_2 are bitmap images that are mostly `0`'s..., representing blank space. Then Eve sees the `1`'s superimposed! In any case, information about m_1 and m_2 has now leaked.

Notably, something like this actually happened during the Cold War. The Soviets used a one-time pad, but sometimes they messed up and used the same key twice. This is apparently how the atomic spies Julius and Ethel Rosenberg were caught, a major event in American history. The decryption project was called VENONA; it was carried out by the forerunner of the NSA and declassified in the 1990s.

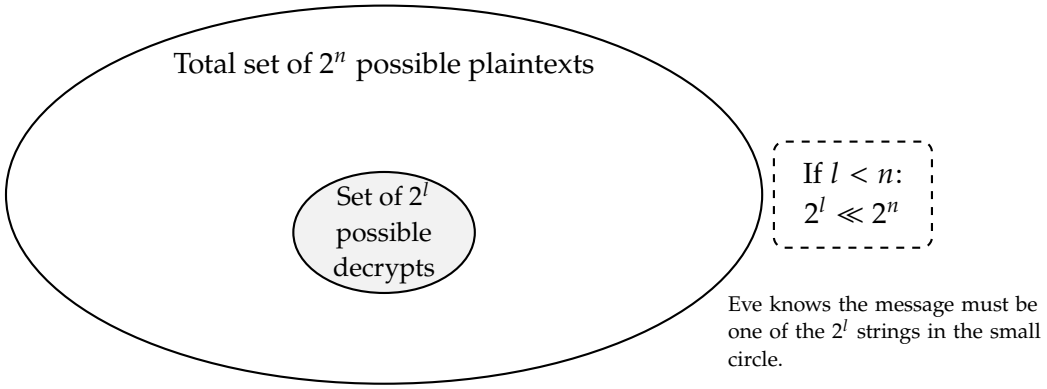
So, the one-time pad is massively inconvenient, because of the need to maintain these gigantic shared keys, each of which needs to be discarded after use. What we'd really like is to use a *short* key to encrypt a long message (or many messages).

Now we ask a question: can we do that, while still getting the perfect security of the one-time pad?

Here our old friend Claude Shannon swoops in once again, to explain why the answer is no.

Theorem 10.1 (Shannon 1940s)**.** *Suppose Alice wants to send Bob a plaintext message m (or many plaintext messages concatenated together), which has n bits in total. Suppose also Alice and Bob share a secret key k that's l bits long. Then to encrypt with perfect security (i.e., so that the eavesdropper Eve learns nothing about m regardless of her computational power), we must have $l \geq n$.*

Proof. Suppose $l < n$, and suppose Eve has unlimited computation time. Eve sees the ciphertext message $c = c(m, k)$. She can then try all 2^l possible decryptions of c with all 2^l possible keys. All $2^n - 2^l$ possible plaintexts other than those can be eliminated! So, Eve has learned something about m. □



So, if we want cryptography with short keys, then we need to rely on computational hardness!

For most cryptosystems, you can *recognize* it when you guess a correct key. Which means decryption is an NP problem! Which means it's easy if P = NP!

Is the converse true? If $\mathsf{P} \neq \mathsf{NP}$, is cryptography with small keys possible? That's an open problem!

We do know that cryptography schemes with small keys is possible if there are *pseudorandom generators.* A pseudorandom generator means a function g mapping n bits to (say) $2n$ bits, such that for all polynomial-time algorithms A, if we choose an n-bit x randomly, the probability that A accepts $g(x)$ is approximately the same as the probability that A accepts a $2n$-bit random string y. By approximately, we mean for example within 2^{-n} of each other.

If $\mathsf{P} = \mathsf{NP}$, then all pseudorandom generators can be broken. (Why?)

If all pseudorandom generators can be broken, then does $\mathsf{P} = \mathsf{NP}$? No one knows!

How can we build a cryptosystem from a pseudorandom generator? Given a short key k shared by Alice and Bob, as well as a pseudorandom generator g, Alice encrypts her plaintext message m by sending Bob the ciphertext message

$$c = m \oplus g(k).$$

Suppose now that Eve could distinguish this c from pure randomness. If $g(k)$ were truly random, then this would be the one-time pad, so she couldn't. This means that Eve must be distinguishing $g(k)$ from a truly random key—i.e., breaking the pseudorandom generator!

One example of a candidate pseudorandom generator is to start with a large composite integer N and a smaller integer x, then take the least significant bits of

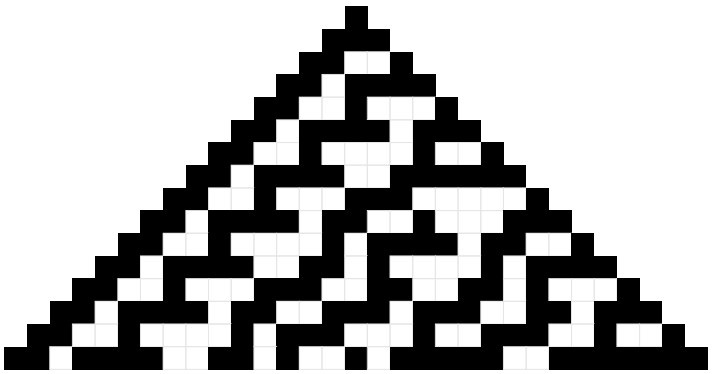
$$x, x^2 \mod N, (x^2)^2 \mod N, ((x^2)^2)^2 \mod N, \ldots$$

Or: Wolfram's Rule 30. We start from a string of zeros and ones and at each step we change the value of each bit according to its value and the value of its neighbors based on the following rules:

0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	1
1	0	0	1
1	0	1	0
1	1	0	0
1	1	1	0

For example,

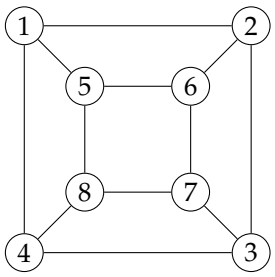
Rule 30 Evolution:



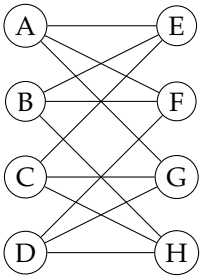
Let x be the initial state, run, and let $g(x)$ be the middle column. More generally, almost any "generic, scrambling" operation is plausibly fine!

Homework 10.1. Let G and H be two n-vertex graphs. Suppose they're not isomorphic. How can a computationally unbounded Merlin convince a polynomial-time bounded Arthur of this fact, with high statistical confidence, via an interactive protocol? (Hint: How could you prove to someone that Coke tastes different from Pepsi, without needing to say anything about their chemical formulas?)

Solution. At each step Arthur chooses one of the graphs G or H at random, shuffles the vertex labels, and sends the resulting graph to Merlin. Since the graphs are not isomorphic, the computationally unbounded Merlin can find the vertices of which graph was shuffled. Merlin would find this out and tell it to Arthur. □



Graph G



Graph H

Homework 10.2. Suppose Alice and Bob want to flip a fair coin, but can only communicate via email. Neither trusts the other, but as long as either one of them wants the outcome to be fair, it should be. How can they accomplish this using cryptography?

Solution. Alice chooses a random string x, applies a pseudorandom generator g to x, and sends the result $y = g(x)$ to Bob. (Note: technically we need g to be injective—that is, there should be no $x \neq y$ such that $g(x) = g(y)$—but most pseudorandom generators with sufficient stretch will have that property in practice.) Then Bob chooses a random

bit b and sends it to Alice. Only now does Alice reveal her original x to Bob, whereupon Bob checks that indeed $y = g(x)$.

Let x_1 be the first bit of x. Then the output of this protocol is $x_1 \oplus b$. Note that Alice doesn't know b when she chooses x because Bob hasn't sent it yet, while Bob doesn't know x_1 when he chooses b because he can't distinguish $g(x)$ from a uniformly random string. On the other hand, Alice can't change x later because, by sending Bob $g(x)$, she *committed* herself to its value. Thus, the output $x_1 \oplus b$ will be uniformly random as long as *either* Alice or Bob wants it to be. □

Lecture 11: Public-Key Cryptography and Quantum Computing

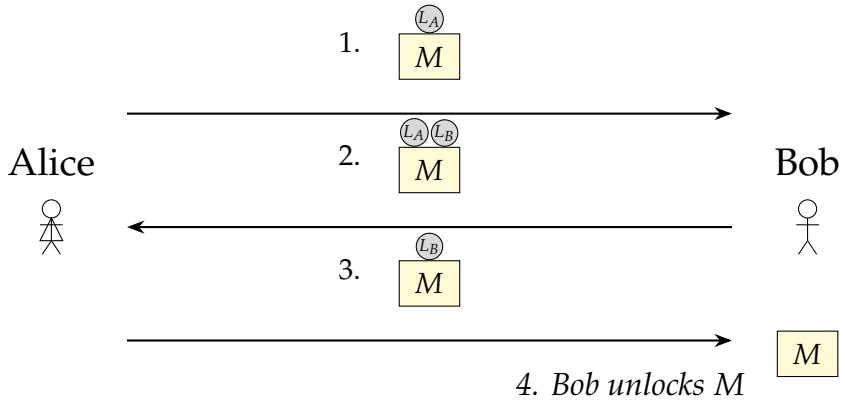
In the last lecture we learned about the one-time pad. But once we got the Internet, it became clear that *any* form of encryption that required agreeing on a secret key in advance was going to be impractical! Should you meet someone from Amazon in a dark alley at 2AM to say "I'm going to send you my credit card number using the key ROSEBUD?"

The revolutionary cryptographic discovery of the 1970's was *public-key cryptography*, also called *asymmetric* cryptography. For the first time in history, it let Alice send a secret message to Bob even if they shared no secret key in advance, and could communicate *only* in the clear.

The two most famous public-key cryptosystems are based on hard but structured problems in number theory: Diffie-Hellman is based on discrete logarithms, while RSA (named for its discoverers, Rivest, Shamir, and Adleman) is based on factoring. Actually, we now know that both systems were discovered first by mathematicians at GCHQ (the British signals intelligence agency), but they kept the discovery secret.

But how could encryption be possible between two people with no shared secret? Wouldn't the eavesdropper know everything they know? That's certainly what I thought when I first heard about this!

Here's a fun story, to build intuition about how public-key encryption *might* be possible. Alice wants to mail Bob a package, but spies in the mail system will open the package if it isn't padlocked. Alice and Bob both have padlocks. The problem is that only Alice can open Alice's padlock, while only Bob can open Bob's. So what can they do?



Here's the clever solution, if you've never seen this before. First Alice sends the package to Bob, with her padlock on it. Bob then *adds* his padlock and returns the package to Alice. Next, Alice removes her padlock and returns the package to Bob. Finally, Bob removes his padlock and opens the package. Note that, every time the spies have access to the package, it has at least one padlock on it.

Diffie-Hellman is likewise based on "multiple padlocks," to keep Eve (the "monkey in the middle") in the dark. To achieve this, we use the so-called *multiplicative group* mod p, denoted $\mathbb{Z}_p^\times$, where p is a large random prime number.

Who knows what that is? It's easiest to give an example: in $\mathbb{Z}_5^\times$ the elements are $1, 2, 3, 4$ and we do addition and multiplication mod five. For example $3 \times 4 = 2 \pmod 5$, etc. Here's the full multiplication table:

$\times$	1	2	3	4
1	1	2	3	4
2	2	4	1	3
3	3	1	4	2
4	4	3	2	1

In general, $\mathbb{Z}_p^\times$ consists of all integers from 1 to $p-1$, under the operation of multiplication $\pmod p$. Then, for a given prime p and a base $g \in \mathbb{Z}_p^\times$, Diffie-Hellman encryption will be based around the function $f(a) = g^a \pmod p$, where we multiply g by itself a times.

Claim. We can calculate $g^a \pmod p$ in poly(n) time, when g, a and p are each n bits long. We don't literally need to do $g \cdot g \cdot g \cdots g$, a times.

How? By *repeated squaring*: as an example,

$$\begin{aligned} g^{21} \pmod p &= g^{16+4+1} \pmod p && (4)\\ &= g^{16} g^4 g^1 \pmod p && (5)\\ &= (((g^2)^2)^2)^2 (g^2)^2 g \pmod p. && (6) \end{aligned}$$

(Note also that all the intermediate results are stored $\pmod p$, which prevents the intermediate numbers from blowing up to have too many digits to write down.)

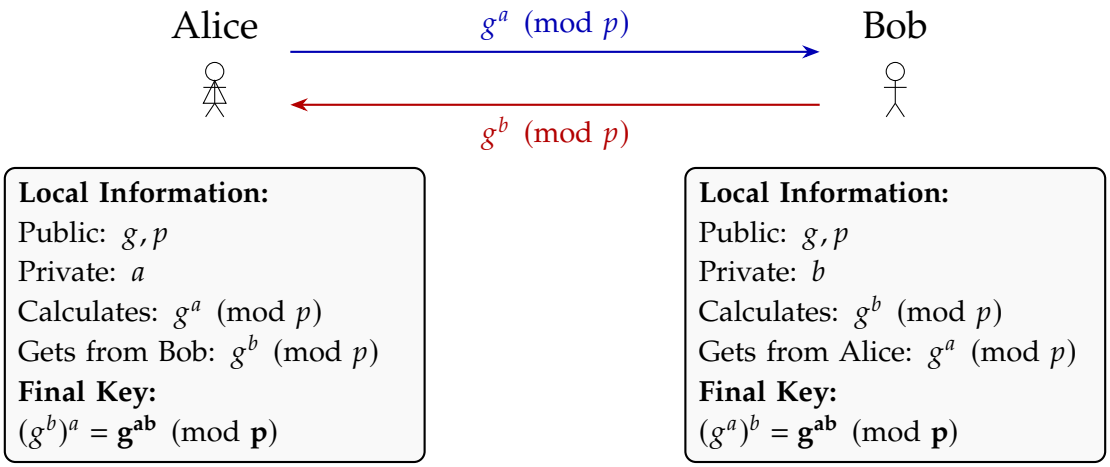
Notice how we reduced the number of multiplications needed to get g^{21} from 20 all the way down to 6 (provided we're careful to store $(g^2)^2$ for later reuse, along the way to computing $(((g^2)^2)^2)^2$)! In general, if the exponent a is n bits long, this method only requires $O(n)$ multiplications $\pmod p$, a vast improvement over the naïve 2^n.

With that out of the way, here's how Diffie-Hellman works. First Alice chooses the prime p, the base g, and an exponent a, and computes $g^a \pmod p$. She then sends g, p, and $g^a \pmod p$ to Bob (while keeping a itself secret).

Next Bob chooses an exponent b, and computes $g^b \mod p$, which he sends back to Alice (while keeping b itself secret).

Finally, Alice and Bob both compute $g^{ab} \pmod p$: Alice as $(g^b)^a = g^{ab} \pmod p$, and Bob as $(g^a)^b = g^{ab} \pmod p$.

Meanwhile, what has the eavesdropper Eve seen? She's seen p, g, $g^a \pmod p$, and $g^b \pmod p$. It's not obvious how, from that information alone, she could compute $g^{ab} \pmod p$ in any reasonable amount of time. If she can't, then Alice and Bob now have a shared secret that *isn't* shared by Eve, and which they can use as a cryptographic key! They've successfully beat the monkey in the monkey-in-the-middle game!



But this all depends on the assumption that given p, g, g^a, and g^b, determining g^{ab} really *is* intractable. Note that this wouldn't be true if (for example) Eve could efficiently solve the equation $g^a = y \pmod p$ for a, which is called the DiscreteLog problem! Like Factoring, DiscreteLog is solvable in $\sim \exp(n^{1/3})$ time, but it's not known to be in P.

RSA is a different public-key cryptosystem, based on the belief that Factoring is hard, and based on multiplicative groups modulo *composite* numbers. I'd explain RSA if I had time.

In 1995, the computer scientist Russell Impagliazzo named five different worlds that we could be living in:

- Algorithmica—where P = NP.
- Heuristica—where P ≠ NP but all NP problems are easy in the average case (that is, for most inputs sampled by a polynomial-time algorithm).
- Pessiland—where hard-on-average NP problems do exist, but we can't base any cryptographic codes on them (the "worst of both worlds").
- Minicrypt—where symmetric-key encryption is possible with small keys, but not public-key encryption.
- Cryptomania—where public-key cryptography is possible.

As far as we can tell, we *seem* to live in Cryptomania!

If we're indeed in Cryptomania—or even Minicrypt—all sorts of interesting things are possible. As one example, which I gave as homework yesterday, you can securely flip coins over the Internet! E.g., if Alice and Bob are getting a divorce and need to flip a coin to see who gets the car, they can do it, even without meeting in person, without relying on any third parties, and with neither one trusting the other.

Another example: you know a Hamiltonian cycle in a certain graph. How can you convince a skeptic that it exists, without doing anything to help them *find* the cycle?

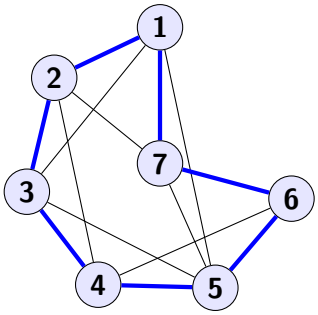


Figure: A Hamiltonian Cycle in a 7-node graph

Solution. First, randomly permute G's vertices. Then, for each pair of vertices v and w, send an encryption of the bit 1 if there is an edge between v and w, or an encryption of the bit 0 otherwise. Then give the skeptic a choice: either you'll decode all the encrypted messages, and also reveal the permutations, back to the original graph G, or else you'll decrypt only the n messages that reveal a Hamiltonian cycle in the permuted graph, showing that the graph must be Hamiltonian (but giving no clue how it relates to the Hamiltonian cycle in the original G).

Since Hamiltonian cycle is NP-complete, you could also use this to convince skeptics that you had, *e.g.*, a ZFC proof of the Riemann hypothesis, without revealing anything about the proof! □

Finally, a few words about quantum computing!

More than thirty years ago, it became clear that quantum mechanics would substantially change the picture of what's efficiently computable in the physical world—and in particular, of which cryptosystems are secure.

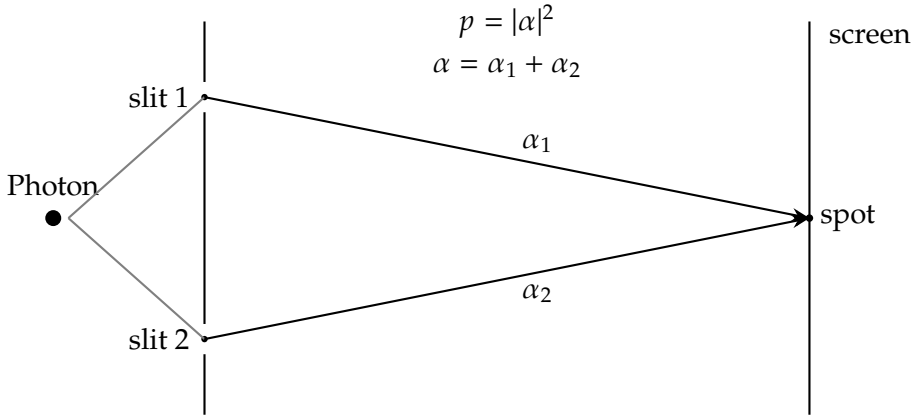
To make a long story short, BQP is the generalization of P to allow quantum algorithms (which need only succeed with, say, 90% probability).

In 1994, Peter Shor showed that BQP contains (the decision versions of) Factoring and DiscreteLog. For example, quantum computers can factor an n-bit integer in $\sim O(n^2)$ steps. As a consequence, a large quantum computer, if built, could be used to break essentially all the public-key cryptography that protects the modern Internet: RSA, Diffie-Hellman, and as it turns out elliptic curve cryptography too.

There are other public-key encryption codes—for example, those based on problems involving high-dimensional lattices—that are believed to be resistant to quantum computers. So, there's now a big push to migrate to these systems. But most of the world still relies on the systems that quantum computers can break.

But what *is* a quantum computer? I'll first need to say a few words about what's quantum mechanics. Imagine firing photons—particles of light—one by one at a barrier with two narrow openings, and a screen behind it to record where they land. You might expect that, if a photon can go through either slit 1 or slit 2, then the chance of it landing somewhere on the screen should just be the sum of the two separate chances. In particular, opening more slits can only *increase*, not *decrease*, the probability that the photon reaches a certain spot. *Obviously*, this is knowable *a priori*, with no need for any details about physics!

Except . . . it's empirically false! Here's what *actually* happens:



where α_1 and α_2 are *complex* numbers, which we call "amplitudes," and which are related to probabilities via $p = |\alpha|^2$.

For example we could have $\alpha_1 = \frac{1}{2}$ and $\alpha_2 = -\frac{1}{2}$ in which case they "interfere destructively" and cancel each other out, so $\alpha = 0$ and $p = |0|^2 = 0$, and the photon is never observed in that spot at all! Yet if we close one of the slits, α will be $\frac{1}{2}$ or $-\frac{1}{2}$, so $p = |\alpha|^2$ will be $\frac{1}{4}$, and the photon *can* be observed there. Also, if we measure to see *which* slit, we just revert to the first picture, with two probabilities that add. Interference is something that the photons only like to do in private.

A quantum computer is a device that would exploit this change to the rules of probability. The idea, with every quantum algorithm, is to choreograph a pattern of interference so that for each possible wrong answer—there might be exponentially many—the exponentially many possible paths leading to that wrong answer point every which way in the complex plane, so they "interfere destructively" and cancel each other out. Meanwhile, the contributions to the right answer all point in more or less the same direction, so they reinforce.

If you can arrange that, then when you measure, you'll see the right answer with a high probability. The tricky part is that you need to do this even though you don't know in advance which answer *is* the right one. If you did, what would be the point?

This is what Shor showed how to do for the specific problems of Factoring and DiscreteLog.

There's vastly more to say about quantum computing, about cryptography, and about theoretical computer science in general. You could spend your entire life learning more about these things if you liked! But I hope I've given you a little taste, just enough to leave you wanting more. Thanks for listening.