Theory of Computation

Freely using various textbooks, mainly the one by Sipser

Péter Gács

Computer Science Department
Boston University

Spring 2013

It is best not to print these slides, but rather to download them frequently, since they will probably evolve during the semester.

The class structure

See the course homepage.
In the notes, section numbers and titles generally refer to the book:
Sipser: Introduction to the Theory of Computation.

Hilbert, 1900 posed the problem of finding an algorithm to decide the solvability of Diophantine equations. By 1970, it was proved that there is no such algorithm. Hilbert could not ask for a negative result, since there was no mathematical definition of algorithm in 1900.

Gödel, 1931 arithmetized logic. This made possible to build, on the pattern of the paradox "I am a liar", a sentence saying "I am not provable".
His constructions formed the basis of the theory of computable functions.

Church and Turing (1936-37) proposed a formal notion of algorithm, and proved that some problems are unsolvable by algorithms. For example, deciding whether a certain statement of (first order) logic has a proof.

von Neumann, Eckert, Mauchly, 1947 constructed the first stored-program computer, the Eniac.

Shannon, 1952 showed that most Boolean functions need almost as large circuits to compute them as the table defining them.

Karatsuba, 1958 showed that numbers can be multiplied faster than learned in school.

Cook, Levin, Karp 1971 introduced NP-completeness theory.

Hellman, Rivest, Shamir, Adleman, 1978 public-key cryptography.

Yao, Blum, Micali, Rakoff 1985 pseudorandomness, interactive proofs.

Notation for languages

Alphabet, string, length $|\cdot|$, binary alphabet.
Empty string e.
Set Σ^* of all strings in alphabet Σ.
Lexicographical enumeration.

Elementary encodings, cardinality

Machines can only handle strings. Other objects (numbers, tuples) will be encoded into strings in some standard way.

Example Let $0, 1 \in \Sigma$, then we can encode each element (u, v) of $\Sigma^* \times \Sigma^*$ as

$$\langle u, v \rangle = 0^{|u|}1uv.$$

For example, $\langle 0110, 10 \rangle = 00001011010$.

So a pair (an abstract concept) is represented by a string (which is more concrete).
Similarly, for a natural number x, we may denote by $\langle x \rangle$ its binary representation, and for natural numbers a, b, the string $\langle a, b \rangle$ is some string encoding of the pair (a, b), for example $\langle a, b \rangle = \langle \langle a \rangle, \langle b \rangle \rangle$.
Triples, quadruples, are handled similarly: $\{a, b, c\} = \langle \langle a, b \rangle, c \rangle$.
Even, finite sequences of natural numbers.

A relation viewed as a set of pairs. Encoding it as a language.

Example Encoding the relation

$$\{\, (x, y) \in \mathbb{N}^2 : x \text{ divides } y \,\}$$

as a language

$$\{\, \langle x, y \rangle \in \{0,1\}^* : x, y \in \mathbb{N},\ x \text{ divides } y \,\}.$$

Encoding a function over strings or natural numbers: first, its graph as a relation, then this relation as a language.

Cardinality The cardinality of the set of all languages: see later.

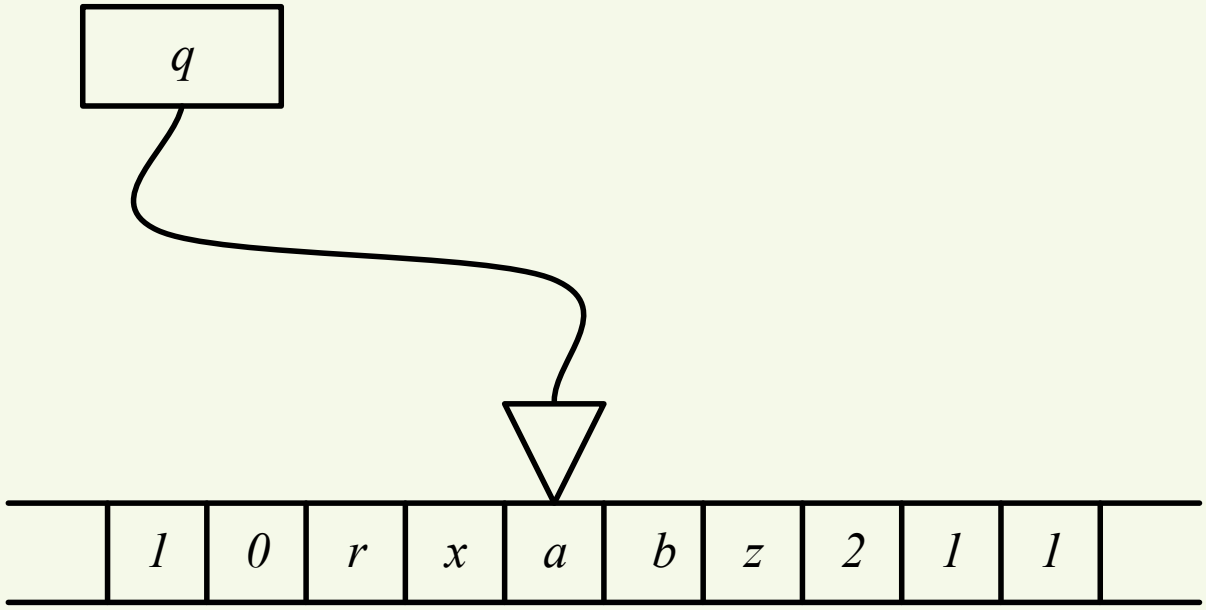
In real life, we work on very complex computers, and write programs to implement our algorithms. For theory, we do not need very complex machines: it is more important is to understand completely, how they work. At the beginning, instead of always working on one and the same machine and just writing different programs, we will devise a different machine for each task.

We need:

(state, memory, an observed memory position)

This is a configuration of the machine. Defining a machine means telling how it "works": how it changes its configuration in each clock cycle. The machine is as simple as possible, so

- the memory is just a string of symbols (a tape)
- the change in one clock cycle will be local, as described by the transition function $\delta()$ below.



- Configuration: uqv where $u = 10rx$, $v = abz211$.
- Suppose $\delta(q, a) = (q', a', d)$, then uqv yields $u'q'v'$.
  If $d = -1$ then $u' = 10r$, $v' = xa'bz211$.
  If $d = 1$ then $u' = 10rxa'$, $v' = bz211$.

More precisely:

$$M = (Q, \Sigma \subseteq \Gamma, \delta, q_{\text{start}} \in Q, F \subset Q).$$

$Q =$ set of states, F is the set of final states.

$\Sigma =$ set of tape symbols, NOT containing blank symbol $\sqcup$.

$\Gamma =$ tape alphabet containing Σ and $\sqcup$.

Head for reading and writing on the one-sided tape.

$\delta()$: $\delta(q, a) = (q', b, d)$ where $d \in \{-1, 0, 1\}$.

Configuration uqv.

Yielding: The meaning of $\boxed{uqv \text{ yields } u'q'v'}$.

The Sipser book writes $q_{\text{start}} = q_0$, $F = \{q_0, q_1\}$, and $d \in \{L, R\}$: these are unimportant variations of the definition.

Example transition table

The action of this machine is to insert the last element of the input 0-1 string at its beginning. Here, $x, y \in \{0, 1\}$. The state set is $\{q_0, q_1, r, r_0, r_1, l_0, l_1\}$.

q	a	q'	a'	d
q_0	$\sqcup$	r	$\sqcup$	1
r	x	r_x	$\sqcup$	1
r_x	y	r_y	x	1
r_x	$\sqcup$	l_x	$\sqcup$	-1
l_x	y	l_x	y	-1
l_x	$\sqcup$	q_1	x	-1

Example diagram

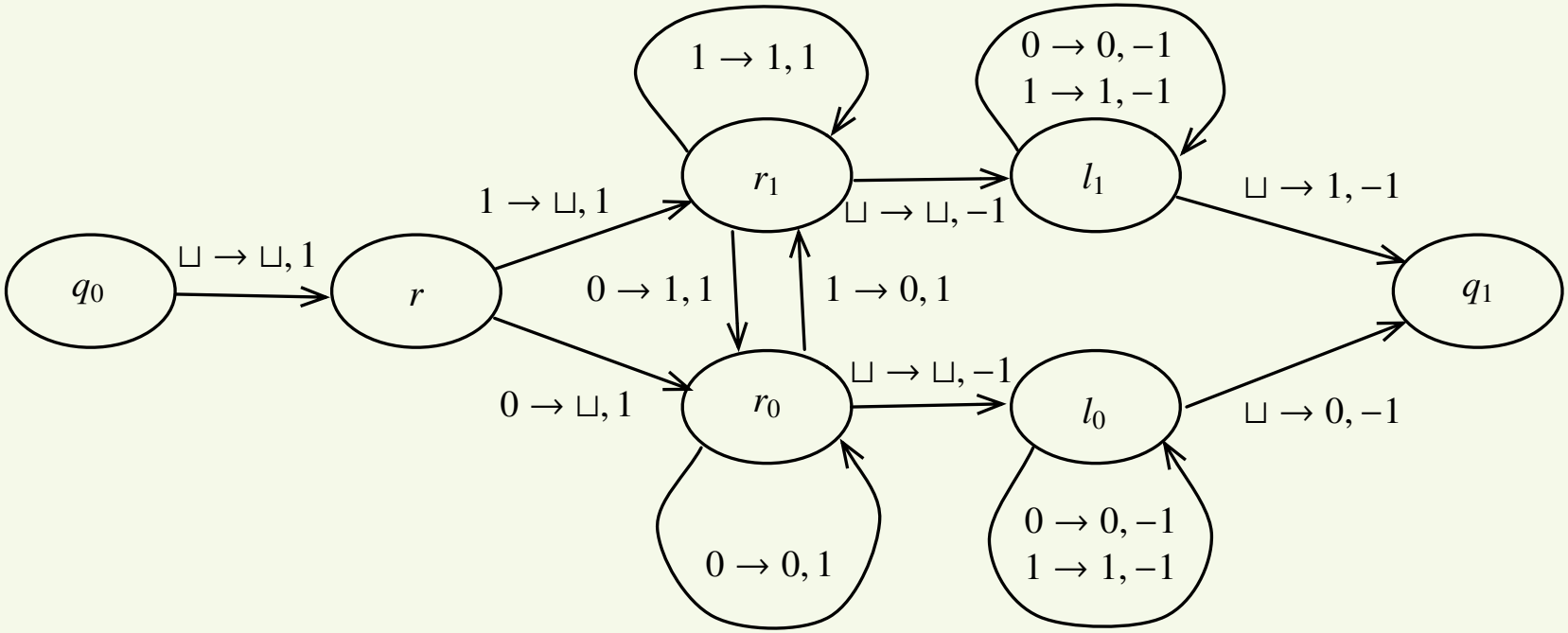


Diagram illustrating the above transition table.

Computing with a Turing machine

Input and output conventions:

Input-output content: the string of Σ^* at the beginning of the tape, after a starting blank cell at the left end (different from Sipser)

Start configuration: head at tape start, else nothing but the input-output content.

Computing a function: For machine M, $M(x)$ is the output content on input content x. $M : \Sigma^* \to \Sigma^*$. We write $M(x) = \infty$ if M does not halt on input x.

Two input arguments: $M(x, y) = M(\langle x, y \rangle)$.

Accepting a language: the set of those strings x with $M(x) = 1$ is denoted by $L(M)$, and is called the language accepted by M. (Sipser: halting in q_{acc}.) We say M recognizes $L(M)$.

Deciding versus recognizing. Turing-recognizable and Turing-decidable languages. (In some other texts, what we call "recognizing" is called "accepting".)

Machine M_2 recognizing $A = \{ 0^{2^n} : n \geqslant 0 \}$. See the Sipser book.

We introduce a machine language for Turing machines. The Sipser book uses the language of state diagrams, but I prefer a language more similar to the machine language of a real computer.
Imagine the tape symbols as being records, (objects or structures) in a programming language. So each symbol of the set Γ can consist of several parts called fields (tracks of the tape).
For example, the first field of a, called $a.io$, the input-output field, can be an element of $\Sigma = \{0, 1\}$. The other, the mark field: $a.m$ an element of $\{\circ, \bullet\}$, so

$$\Gamma = \Sigma \times \{\circ, \bullet\} = \{(0, \circ), (0, \bullet), (0, \perp), (1, \circ), (1, \bullet), (1, \perp)\}.$$

So, if $x = (1, \bullet)$ then $a.io = 1$, $a.m = \bullet$. Each field has a default value. The symbol is blank when all fields have their default value. The blank symbol is defined a $\text{␣} = (0, \circ)$.

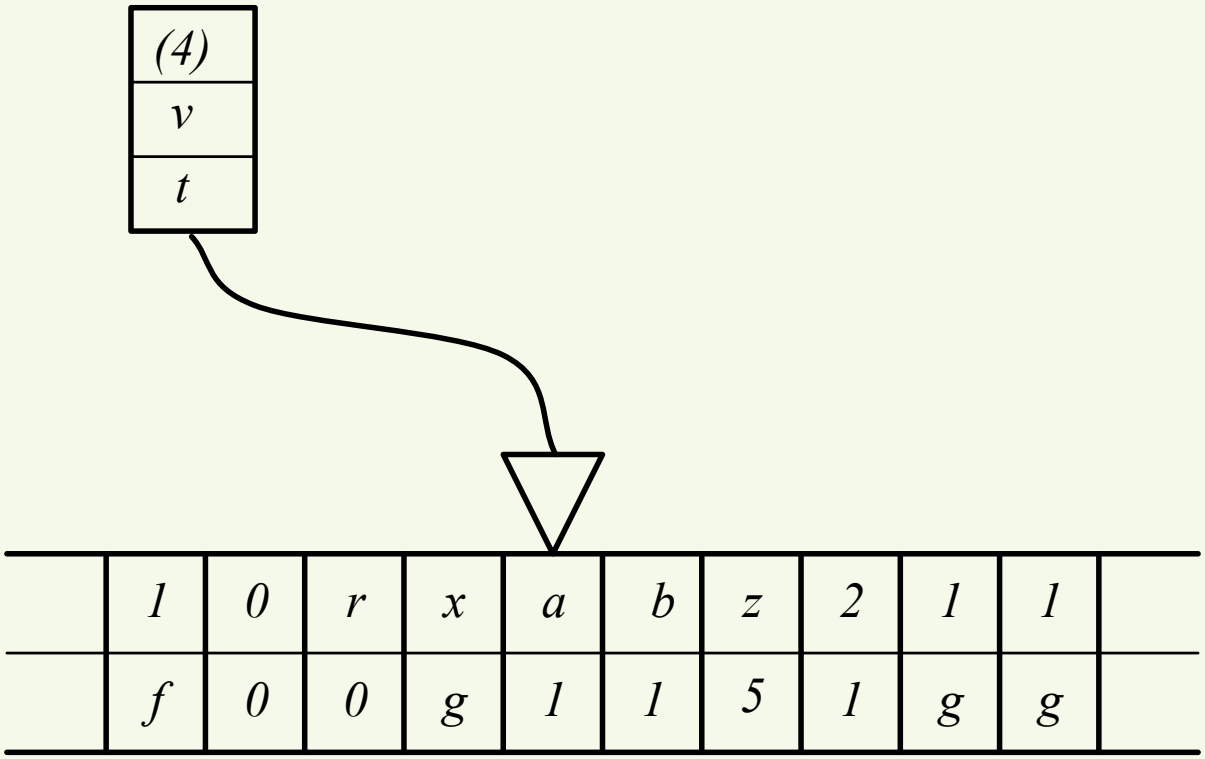
Fields of a state

The machine states can also have fields. For example, we could have

$$Q = \{(\text{acc}), (\text{acc}), (1), (2), (3), (4), (5)\} \times \Sigma.$$

The first field of the state q could be called $q.l$ for an instruction label, the second part $q.info$. The default value of $q.l$ is (1).
The state is the starting state if all fields have their default value. The states with $q.l =\in \{(\text{acc}), (\text{acc})\}$ are the final (halting) states.

Turing machine with fields



A Turing machine with three fields of the state: here $q = ((4), v, t)$.
They could have names like $q.label = (4)$, $q.mark = v$, and so on.
The tape symbols also have two fields, so the tape has two tracks.

Instructions

- A program is a list of instructions, labeled with $(\mathrm{acc}), (\mathrm{acc}), (1), (2), \ldots$. Each instruction has the form

> **if** condition **then** action

- The condition involves the state q and the tape symbol a.
- The action can change the state and the tape symbol, and can move the head.
- The field $q.l$ shows the label of the program instruction which is being read. So the action **goto** (8) means $q.l \leftarrow (8)$.

As an example, we implement Sipser's program for machine M_2 recognizing $A = \{ 0^{2^n} : n \geqslant 0 \}$. We generalize it to recognize any $x \in \{0, 1\}^*$ which contains exactly 2^n 0's for some n.
On input string w, here is Sipser's description:

1. Sweep right, marking every 0.
2. If in stage 1 the tape contained a single 0, accept.
3. If in stage 1 the tape contained more than a single 0 and the number of 0's was odd, reject
4. Return the head to the left end.
5. Go to stage 1.

Implementation as a machine:

- Fields of a tape symbol: $io \in \{0, 1\}$, $m \in \{\circ, \bullet, \bot\}$, with default values $m = \circ$, $io = 0$.
  So a tape symbol x is a pair $(a.io, a.m)$. The blank tape symbols a have $a.m = \bot$). The value $a.m = \bullet$ says that the symbol is marked.
- Fields of the state:

$$l \in \{(\text{acc}), (\text{acc}), (1), (2), (3)\}, \; n \in \{0, 1, 2\}, \; p \in \{0, 1\},$$

  with default values $l = (1)$, $n = 0$, $p = 0$. So a state is a 3-tuple $(q.l, q.n, q.p)$.
- The instruction **accept** will mean setting $q.l \leftarrow (\text{acc})$: halting in an accepting state. Similarly with **reject**.

```
(1) move right
(2) // Right sweeping loop until blank:
    if x = ␣ then move left
    else
        if a.io = 0 and a.m = ∘ then
            if q.n < 2 then q.n++   // Count unmarked 0's up to 2
            q.p ← 1 − q.p
            if q.p = 0 then a.m ← •   // Mark even unmarked 0's
        move right; goto (2)
(3) // Left sweeping loop until blank:
    if x ≠ ␣ then
        move left; goto (3)
    // Decision at end of left sweep:
    else if q.n = 0 or q.p = 1 then reject
    else if q.n = 1 then accept
    else
        q.n ← 0; move right; goto (2)
```

In form of a "transition table", where u, v, x, y means "anything". (For a pure table, expand each line by substituting all possible values into u, v, x, y.)

$q.l$	$q.n$	$q.p$	$a.io$	$a.m$	$q'.l$	$q'.n$	$q'.p$	$a'.io$	$a'.m$	d
(1)	0	0	0	$\perp$	(2)	0	0	0	$\perp$	1
(2)	u	v	x	$\perp$	(3)	u	v	x	$\perp$	-1
(2)	u	v	1	x	(2)	u	v	1	x	1
(2)	u	v	0	$\bullet$	(2)	u	v	0	$\bullet$	1
(2)	0	0	0	$\circ$	(2)	1	1	0	$\bullet$	1
(2)	1	1	0	$\circ$	(2)	2	0	0	$\bullet$	1
(2)	2	v	0	$\circ$	(2)	2	$1 - v$	0	$\bullet$	1
(3)	u	v	x	$y \neq \perp$	(3)	u	v	x	y	-1
(3)	0	v	x	$\perp$	(acc)	0	v	x	$\perp$	0
(3)	$u > 0$	1	x	$\perp$	(acc)	u	1	x	$\perp$	0
(3)	1	0	x	$\perp$	(acc)	1	0	x	$\perp$	0
(3)	2	0	x	$\perp$	(2)	0	0	x	$\perp$	1

This is when we really will talk about:

- Representing a pair of strings.
- Functions with 2 or more arguments.
- Recognition of a language.

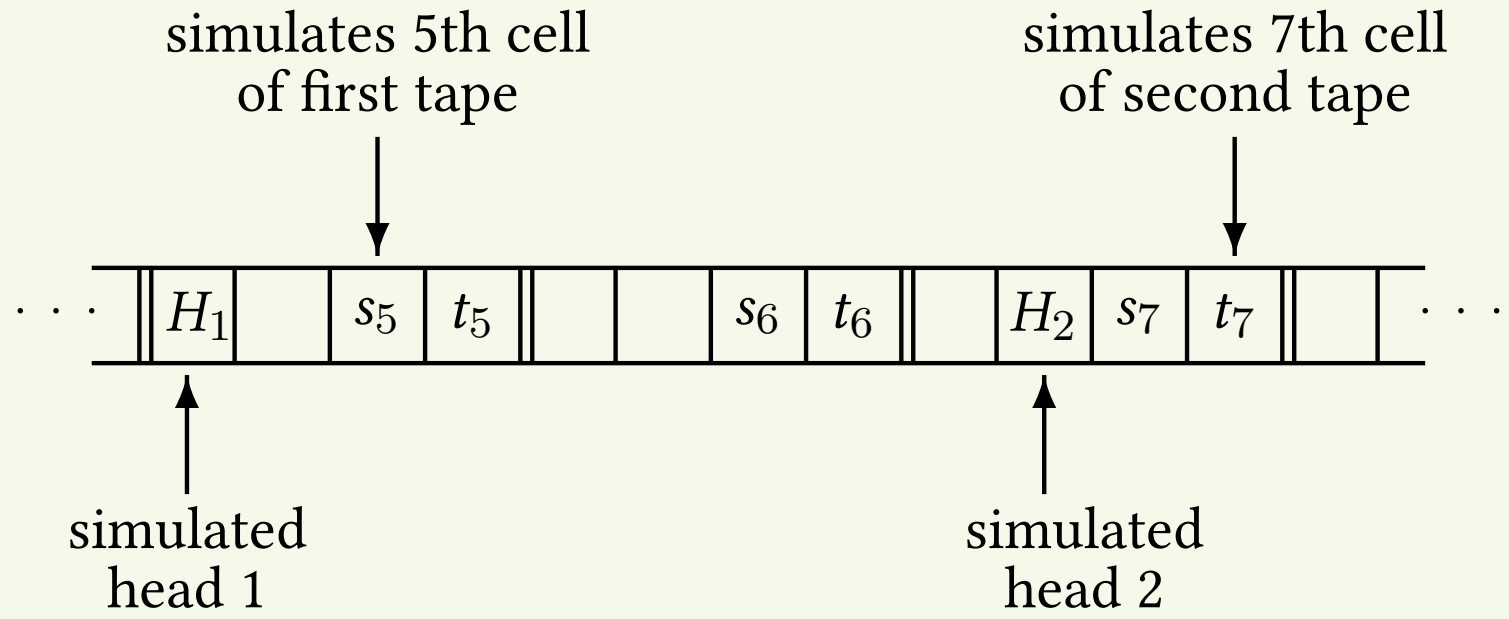
- Definition: the natural one.
- Many operations are easier to implement with 2 or more tapes.

Example (Binary addition)

- Input, numbers x, y represented by binary strings, on tapes 1 and 2. (If it is given as $\langle x, y \rangle$ then first copy y over from tape 1 to tape 2.)
- Output, $M(x, y) = x + y$, say on tape 3.

Simulation: M' simulates M if the input-output behavior of M' is the same as that of M.

In practice: representing one "data structure" in another:
Configuration C of M is represented by a configuration C' of M'.
Each step of M will be simulated by several steps of M'. If a step of M carries C to D then the corresponding steps of M' carry C' to D'.



If $s(x)$ is the memory requirement and $t(x)$ is the time requirement of the 2-tape machine on input x then the time requirement of the 1-tape simulation is $O(s(x)t(x)) = O(t^2(x))$.

2-dimensional Turing machine

Representing a 2-dim tape of a machine M on two 1-dim tapes of some machine M':

- Address on M: a pair of numbers (u, v), represented in binary, length $\lceil \log u \rceil + \lceil \log v \rceil + 2$.
- Tape 1 of M' contains a sequence of (address, content) pairs of M, in arbitrary order.
  Tape 2 contains the (address, content) pair currently observed.
- Simulating one step of M: Apply the transition function to the content on tape 2. Compute the new address on tape 2 (changing one coordinate by ± 1).
  Look up this address on tape 1 (find match). This may need to scan the whole unblank part of tape 1 of M', length $< 2t \log t$.
  And so on.
- t steps of M are simulated by $O(t^2 \log t)$ steps of M'.

The random access machine

Memory a (potentially) infinite sequence $x[0], x[1], x[2], \ldots$ of memory registers each containing an integer.

Program store a (potentially) infinite sequence of registers containing instructions.

$$x[i] := 0; \qquad x[i] := x[i] + 1; \qquad x[i] := x[i] - 1;$$
$$x[i] := x[i] + x[j]; \qquad x[i] := x[i] - x[j];$$
$$x[i] := x[x[j]]; \qquad x[x[i]] := x[j];$$
$$\textbf{if } x[i] \leqslant 0 \textbf{ then goto } p.$$

Input-output conventions.

How to define running time?

Simulations between the RAM and Turing machines. There is at most a $t \mapsto t^2$ slowdown.

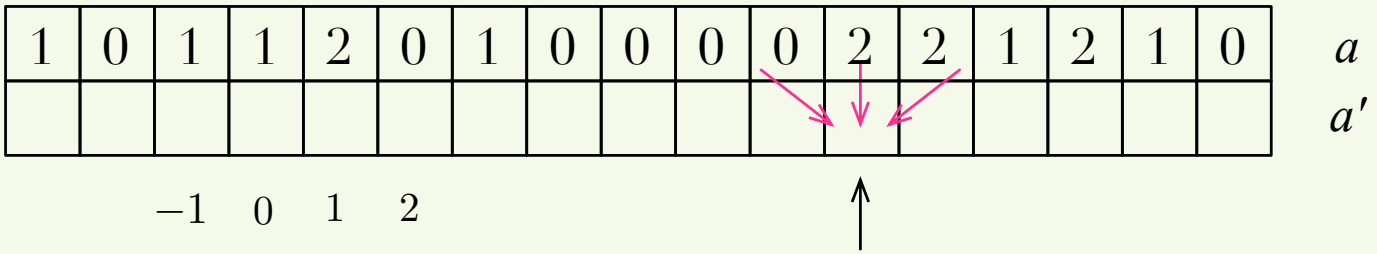
Cellular automata

Let us also see a parallel computation model. It looks like a doubly infinite Turing machine tape, except that each cell is active: it serves both as a tape symbol and as a state.

- A 1-dimensional cellular automaton is given by an alphabet Γ, and a transition function $\delta : \Gamma^3 \to \Gamma$.
- A configuration is a doubly infinite string a. The content (state) of cell n is written as $a[n]$.
- Here is how the machine computes. If the configuration at a given time is a, then the configuration a' at next time is given by

$$a'[n] = \delta(a'[n-1], a'[n], a'[n+1]).$$

So there is a little processor at each site n with transition function δ whose inputs are the states of itself and of its two neighbors, and whose output is its state in the next clock cycle.

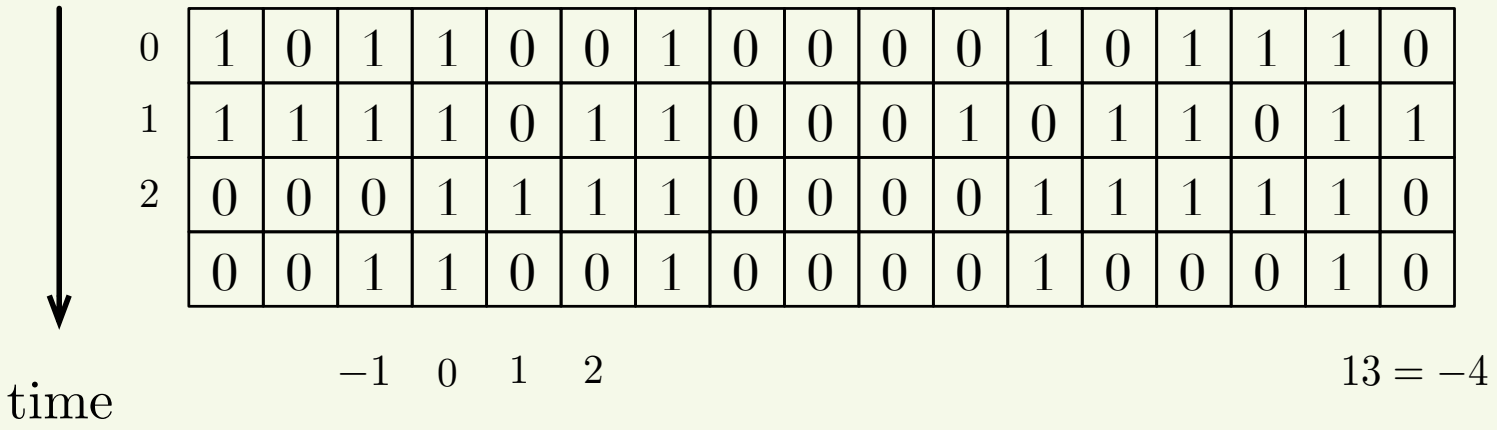


$$a'[8] = \delta(a[7], a[8], a[9])$$

Example rule:

x, y, z	000	001	010	011	100	101	110	111
$\delta(x, y, z)$	0	1	1	1	0	1	1	0

A sample history. Here the space is not infinite, only a cycle of length 16: cell 13 is made the left neighbor of cell -3.



Using cellular automata

Blank state We will assume that one of the states is called the blank state $␣$, and $\delta(␣, ␣, ␣) = ␣$. We will only consider configurations in which all but a finite number of sites contain a the blank state.

Input-output alphabet We again assume that there is an input-output alphabet $\Sigma \subset \Gamma$ not containing $␣$.

Starting and halting Cellular automata never halt, but we can assume a distinguished starting state q_{start}, and halting state q_{halt}. The machine is said to halt when one of the cells reaches q_{halt}.

Computing a function $f : \Sigma^* \to \Sigma^*$. Write input-output string $x_1, \ldots, x_n$ into cells $1, 2, \ldots, n$. Write q_{start} into cell 0, and blank everywhere else. Compute until halt, then read off the result.

Simulating a Turing machine by a CA

Let us simulate a Turing machine M with state space Q and tape alphabet Γ, using a cellular automaton C with state set

$$\Gamma' = (Q \cup \{*\}) \times \Gamma.$$

- Each cell x of C has fields called *state*, *tape*.
  $x.tape[n]$ will represent the symbol $a[n]$ of the tape.
  $x.state[n] = * \notin Q$ if the head is not on tape cell n. If it is then $x.state[n] = q$, the state of the Turing machine.
- Transition function $\delta'(\cdot, \cdot, \cdot)$ of C is easy to define using $\delta(q, a)$.
  Let $\delta'((p, a), (q, b), (r, c)) = (q', b')$.
  If $p = q = r = *$ then $q' = *$, $b' = b$.
  If $p = q = *$, $r \in Q$, $\delta(r, c) = (r', a', -1)$ then $q' = r'$, $b' = b$.
  And so on.

Simulating a CA by a one-tape Turing machine

This will be homework.

Let Σ be some alphabet, called an input/output alphabet. Let $\mathcal{T}_1(\Sigma)$ be the set of all 1-tape Turing machines M whose alphabet contains Σ, and such that whenever $M(x)$ halts on some $x \in \Sigma^*$, we have $M(x) \in \Sigma^*$.
We say that the 1-tape machine U in $\mathcal{T}_1(\Sigma)$ with tape alphabet Γ is universal for machines in $\mathcal{T}_1(\Sigma)$ if for all machines $M \in \mathcal{T}_1(\Sigma)$ there is a string $p_M \in \Gamma^*$ such that for all inputs $x \in \Sigma^*$ we have

$$M(x) = U(p_M, x).$$

Theorem For every Σ there is a machine U universal for $\mathcal{T}_1(\Sigma)$. Also, $U(p_M, x)$ accepts iff $M(x)$ accepts.

Constructing the universal machine

For simplicity, let $\Sigma = \{0, 1\}$, $k = \lceil \log |Q| \rceil$, $l = \lceil \log |\Gamma| \rceil$.

Encoding the states into $\langle q \rangle \in \{0, 1\}^k$, and the the tape symbols into $\langle a \rangle \in \{0, 1\}^l$. For example, $\Gamma = \{0, 1, \#, *, \textvisiblespace\}$ can be encoded into

$$\langle 0 \rangle = 000, \ \langle 1 \rangle = 001, \ \langle \# \rangle = 010, \ \langle * \rangle = 011, \ \langle \textvisiblespace \rangle = 100.$$

Encode directions $d \in \{-1, 0, 1\}$ into $\langle d \rangle \in \{10, 00, 01\}$.

Encoding tuples: Entry $E = (q, a) \mapsto (q', a', d)$, of the table of the transition function δ into

$$\langle E \rangle = \langle q \rangle \langle a \rangle \langle q' \rangle \langle a' \rangle \langle d \rangle.$$

Representation of M:

$$\langle M \rangle = 0^k 1 0^l 1 \langle E_1 \rangle \langle E_2 \rangle \cdots \langle E_m \rangle.$$

We first construct U_3 with 3 tapes: this will then be simulated on a one-tape machine U.

Preprocessing: encode the input $x_1 \dots x_n$ into $\langle x_1 \rangle \dots \langle x_n \rangle$, onto Tape 1. For example if $x_i \in \{0, 1\}$ and $\langle 0 \rangle = 000$, $\langle 1 \rangle = 001$ then $\langle x_1 \rangle \dots \langle x_n \rangle = 00x_1 00x_2 \dots 00x_n$.

Tape 1 represents the tape contents and head position of M.

Tape 2: $\langle M \rangle$.

Tape 3: $0^k 10^l 1 \langle q \rangle$, where q is the current state of M.

Simulation: for each simulated step of M, look up the needed entry in the transition table represented by $\langle M \rangle$.

Postprocessing: decode the output $\langle y_1 \rangle \dots \langle y_{n'} \rangle$ into $M(w) = y_1 \dots y_{n'}$: for example $00y_1 00y_2 \dots 00y_{n'}$ into $y_1 \dots y_{n'}$.

How long does it take? The inefficiency of having to represent a whole transition table. It can be made faster if for example the transition function is computed by a logic circuit.

A useful picture

Imagine $(M, w) \mapsto M(w)$ in a matrix with rows indexed by $\langle M \rangle$ and columns indexed by w: at position $(\langle M \rangle, w)$ sits the result $M(w)$, if it is defined, namely if the computation of M halts on input w. Let us put ∞ where it does not.

	$w_0 = e$	$w_1 = 0$	$w_2 = 1$	$w_3 = 00$	$\dots$
$\langle M_1 \rangle$	e	∞	0001	e	$\dots$
$\langle M_2 \rangle$	1101	0	1	1	
$\langle M_3 \rangle$	$M_3(e) = 111$	$M_3(0) = 010$	$M_3(1) = \infty$	$M_3(00) = \infty$	$\dots$
$\langle M_4 \rangle$					
$\vdots$					$\ddots$

The concept of an algorithm – Church's thesis

This is a "thesis", not a theorem, since it says that a certain informal concept (algorithmically computable) is equivalent to a formal one (Turing computable).

History different formal definitions by Church (lambda calculus), Gödel (recursive functions), Turing (you know what), Post (formal systems), Markov (a different kind of formal system), Kolmogorov (spider machine on a graph) all turned out all to be equivalent.

Algorithm any procedure that can be translated into a Turing machine (or, equivalently, into a program on a UTM).

Two possible uses of Church's Thesis

Justified Proving that something is not computable by Turing machines, we conclude that it is also not computable by any algorithm.

Unjustified Giving an informal algorithm for the solution of a problem, and referring to Church's thesis to imply that it can be translated into a Turing machine. It is your responsibility to make sure the algorithm is implementable: otherwise, it is not really an algorithm. Informality can be justified by common experience between writer and reader, but not by Church's Thesis.

In the Data Structures and Algorithms courses, you must have learned how to measure the runtime of actual programs. For example, the Unix commands

```
time binom_dumb 28 13
times binom_dumb 28 13
```

measure the time spent by the computer on a command. It is even more convenient to find this information from inside a C++ program, since then you can store it, tabulate it, run large-scale experiments with different inputs. You can use the `clock` or `times` library functions for this.

Ignoring constant factors

An algorithm is not a program: it can be implemented in programs in various programming languages.
When we run the implementation, we can measure the time, and it will depend on the implementation. The implementation generally does not change the overall way in which the running time depends on different inputs. On one machine, for input sizes $n = 1, 2, 3, 4$ it may be

$$1, 4, 9, 16,$$

on another machine: $10, 40, 90, 160$. The difference is only by a constant factor (even if large).

Example `binom_dumb` run on my laptop or on csa3.

What is a step?

It seems possible to define running time in an abstract way that depends only on the algorithm, not on the implementation. The running time is the number of "elementary steps" taken.
On a Turing machine, it is simply a step. In C++, we can be a little more liberal:

- assignment of primitive data types (integer, double, and so on).
- simple arithmetic operations, though some are more expensive than others.
- comparisons of primitive data types.
- function call (but not its execution).
- following a pointer.
- indexing into an array.

Let $\min(A)$ be a function that finds the minimum of an integer vector A of length n:

```
n ← A.size()                                  // 1
m ← A[0]                                      // 1
                        // The loop repeats n times
for i = 0 to n − 1 do
                        // Loop control: 2 steps for each i
    if m > A[i] then m ← A[i]          // 1 or 2 for each i
    return m                                                  // 1
```

Total number of steps: $3 + n \cdot c$ where $3 \leqslant c \leqslant 4$. We say that the number of steps is $O(n)$ if it is $< Cn$ for some constant C. We frequently do not bother about the exact value of C, since it would be too machine-dependent, just as the cost of the different kinds of step.

For example, $\min(A)$ takes at most $5 + 2n \leqslant 7n$ steps, so it takes $O(n)$ steps; we thankfully suppress the irrelevant detail in the formula $O(n)$.
We are interested in bigger differences. Compare the following two program parts:
Part (1):

```
for i = 0 to n − 1 do
    B[i] ← min(A) · (i + 1)
```

Part (2):

```
m ← min(A)
for i = 0 to n − 1 do
    B[i] ← m · (i + 1)
```

Part (1) is $O(n^2)$, since $\min(A)$ recomputes the minimum every time it is called. Part (2) is $O(n)$. The big-O notation directs attention to these big differences, by ignoring the small ones.

Machine M terminates in worst-case-time $O(f(n))$ if

$$(\exists c > 0)(\forall n) \max_{x \in \Sigma^n} \text{Time}_T(x) \leqslant c \cdot f(n).$$

Example Consider the following function called find_root(A). It returns -1 if there is no root.

```
n ← A.size
for i = 0 to n − 1 do
    if 0 = A[i] then return i
return −1
```

The worst case and best case running time are very different here. We are generally interested in the worst case. The "average case" may also be interesting, but is harder to say what it is.

Example The linear programming problem (solving a set of linear inequalities). For the simplex algorithm, the worst case is exponential, but the average case is linear (number of "iterations").

Example Deterministic quicksort. Worst case, n^2. Average case, $n \log n$. But how natural is here to consider average? It is quite likely that our input will be an array that is already (maybe almost) sorted.

Complexity of a problem (informally): the complexity of the best algorithm solving it.

Problems

- Compute a function
- Decide a language
- Given a relation $R(x, y)$, for input string x find an output string y for which $R(x, y)$ is true. Example: $R(x, y)$ means that the integer y is a proper divisor of the integer x.

$\mathrm{DTIME}(f(n))$: a class of languages.

Upper bound given a language L and a time-measuring function $g(n)$, showing $L \in \mathrm{DTIME}(g(n))$.

Lower bound given a language L and a time-measuring function $g(n)$, showing $L \notin \mathrm{DTIME}(g(n))$.

Example Let $\mathrm{DTIME}(\cdot)$ be defined using 1-tape Turing machines, and let $L_1 = \{ uu : u \in \Sigma^* \}$. Then it can be proved that

$$L_1 \notin \mathrm{DTIME}(n^{1.5}).$$

The difficulty of proving a lower bound: this is a statement about all possible algorithms.

Why complexity classes?

Why we are just speaking about complexity classes, rather than the complexity of a particular problem.
The difficulty of defining “the complexity” of a problem.
Speedup theorems.

Why concentrate on language classes?

The complexity of computing a function is just as interesting. But sometimes, there are trivial lower bounds for functions: namely, $|f(x)|$ (the length of $f(x)$) is a lower bound.

Example $f(x, y) = x^y$ where the binary strings x, y are treated as numbers.

Naive algorithm $x \cdot x \cdot \cdots \cdot x$ (y times). This takes y multiplications, so it is clearly exponential in the length of y.

Repeated squaring now the number of multiplications is polynomial in $|y|$.

But no matter what we do, the output length is $|x^y| \approx y \cdot |x|$, exponential in $|y|$.

If the function values are restricted to $\{0, 1\}$ (like when deciding a language) then we cannot have such trivial lower bounds.

Eliminating duplicates: an analysis

Remove all duplicates from vector x, without changing the order. It uses the function erase(x, j), that erases the jth element of a vector, shifting over the others accordingly.
What is the running time of this algorithm:

```
if 2 > x.size then return
for i = 0 to x.size − 2 do
    for j = i + 1 to x.size − 1 do
        if x[i] = x[j] then
            erase(x, j)
            j−−;
```

Easy upper bound: n^3. But is it really? Better analysis shows at most n^2. (How many times can erase() really be called?)

Better organization avoids erase() altogether. Here, only the number of comparisons grows like n^2, the number of assignments is at most n. (You can safely skip this.)

```
n ← x.size
if 2 > n then return
current_end ← 1
for compared1 = 1 to n − 1 do
    found ← false
    for compared2 = 0 to current_end − 1 do
        if x[compared2] = x[compared1] then
            found ← true
            break                    // from inner loop
    if found then continue                    // outer loop
    else
        if compared1 > current_end then
            x[current_end] ← x[compared1]
            current_end++
for i = n downto current_end + 1 do x.pop_back()
```

Sorting brings the complexity down to $n \log n$. It also features an important problem-solution method: try to bring some additional order into the situation, even if it does not seem immediately required.

$O(), o(), \Omega(), \Theta()$. More notation: $f(n) \ll g(n)$ for $f(n) = o(g(n))$, $f(n) \overset{*}{<} g(n)$ for $f(n) = O(g(n))$ and $\sim$ for ($\overset{*}{<}$ and $\overset{*}{>}$).
The most important function classes: log, logpower, linear, power, exponential.
Some simplification rules.

- Addition: take the maximum. Do this always to simplify expressions. *Warning*: do it only if the number of terms is constant!
- An expression $f(n)^{g(n)}$ is generally worth rewriting as $2^{g(n)\log f(n)}$. For example, $n^{\log n} = 2^{(\log n)\cdot(\log n)} = 2^{\log^2 n}$.
- But sometimes we make the reverse transformation:

$$3^{\log n} = 2^{(\log n)\cdot(\log 3)} = (2^{\log n})^{\log 3} = n^{\log 3}.$$

The last form is easiest to understand, showing n to a constant power $\log 3$.

$$n/\log\log n + \log^2 n \sim n/\log\log n.$$

Indeed, $\log\log n \ll \log n \ll n^{1/2}$, hence
$n/\log\log n \gg n^{1/2} \gg \log^2 n$.

Order the following functions by growth rate:

$$
\begin{aligned}
& n^2 - 3\log\log n && \sim n^2, \\
& \log n/n, && \\
& \log\log n, && \\
& n\log^2 n, && \\
& 3 + 1/n && \sim 1, \\
& \sqrt{(5n)/2^n}, && \\
& (1.2)^{n-1} + \sqrt{n} + \log n && \sim (1.2)^n.
\end{aligned}
$$

Solution:

$$
\begin{aligned}
\sqrt{(5n)/2^n} \ll \log n/n \ll 1 \ll \log\log n \\
\ll n/\log\log n \ll n\log^2 n \ll n^2 \ll (1.2)^n.
\end{aligned}
$$

Arithmetic series

Geometric series its rate of growth is equal to the rate of growth of its largest term.

Example

$$\log n! = \log 2 + \log 3 + \cdots + \log n = \Theta(n \log n).$$

Indeed, upper bound: $\log n! < n \log n$.
Lower bound:

$$\begin{aligned} \log n! &> \log(n/2) + \log(n/2+1) + \cdots + \log n > (n/2)\log(n/2) \\ &= (n/2)(\log n - 1) = (1/2)n \log n - n/2. \end{aligned}$$

Example

Prove the following, via rough estimates:

$$1 + 2^3 + 3^3 + \cdots + n^3 = \Theta(n^4),$$

$$1/3 + 2/3^2 + 3/3^3 + 4/3^4 + \cdots < \infty.$$

Example

$$1 + 1/2 + 1/3 + \cdots + 1/n = \Theta(\log n).$$

Indeed, for $n = 2^{k-1}$, upper bound:

$$1 + 1/2 + 1/2 + 1/4 + 1/4 + 1/4 + 1/4 + 1/8 + \ldots$$
$$= 1 + 1 + \cdots + 1 \ (k \text{ times}).$$

Lower bound:

$$1/2 + 1/4 + 1/4 + 1/8 + 1/8 + 1/8 + 1/8 + 1/16 + \ldots$$
$$= 1/2 + 1/2 + \cdots + 1/2 \ (k \text{ times}).$$

For simplicity, assume n is a power of 2 (otherwise, we pick $n < n' \leqslant 2n$ that is a power of 2). Let $m = n/2$, then

$$\begin{aligned} f(x) &= a_0 + a_1 x + \cdots + a_{m-1} x^{m-1} + x^m (a_m + \cdots + a_{2m-1} x^{m-1}) \\ &= f_0(x) + x^m f_1(x). \end{aligned}$$

Similarly for $g(x)$. So,

$$fg = f_0 g_0 + x^m (f_0 g_1 + f_1 g_0) + x^{2m} f_1 g_1.$$

In order to compute fg, we need to compute

$$f_0 g_0, \quad f_0 g_1 + f_1 g_0, \quad f_1 g_1.$$

How many elementary multiplications does this need? (We do not count additions.) If we compute $f_i g_j$ separately for $i, j = 0, 1$ this would just give the recursion

$$M(2m) \leqslant 4M(m),$$

which suggests that we need n^2 multiplications.

Trick (found by Karatsuba) that saves us a (polynomial) multiplication:

$$f_0 g_1 + f_1 g_0 = (f_0 + f_1)(g_0 + g_1) - f_0 g_0 - f_1 g_1. \tag{1}$$

This gives $M(2m) \leqslant 3M(m)$, saving a lot more when we apply it recursively.

$$M(2^k) \leqslant 3^k M(1) = 3^k.$$

So, if $n = 2^k$, then $k = \log n$,

$$M(n) < 3^{\log n} = 2^{(\log n)\cdot(\log 3)} = n^{\log 3}.$$

Since $\log 4 = 2$, so $\log 3 < 2$ and hence $n^{\log 3}$ is a smaller power of n than n^2.
(It is possible to do much better than this for polynomial multiplication.)

Fast multiplication of numbers

Taking additions in account in polynomial multiplication. The result will not change, since the recursion is controlled by the multiplications.

Numbers. What we count here are the elementary steps (bit operations) of a Turing machine (or other machine). The analysis is similar to polynomial multiplications, but the carry needs to be taken into account.

We have seen that 2-tape Turing machines and even 2-dimensional and random-access machines can be simulated by 1-Tape Turing machines, with a slowdown similar to $t \mapsto t^2$. Therefore to some questions ("is there a polynomial-time algorithm to compute function f?") the answer is the same on all "reasonable" machine models.

Does it capture “practical”?

- The polynomial time requirement is too weak: in many situations, (data mining) only linear-time algorithms are affordable. Moreover, sometimes only logarithmic-time algorithms can be allowed.
- It may miss the point. On small data, an $0.001 \cdot 2^{0.1n}$ algorithm is better than a $1000n^3$ algorithm.

Still, in typical situations, the lack of a polynomial-time algorithm means that we have no better idea for solving our problem than “brute force”: a run through “all possibilities”.

PATH between points s and t in a graph (remember the algorithms course CS330). Breadth-first search.
The same problem, when the edges have positive integer lengths. Reducing it to PATH in the obvious way (each edge turned into a path consisting of unit-length edges) may result in an exponential algorithm (if edge lengths are large). But Dijkstra's algorithm works in polynomial time also with large edge lengths.

Every algorithm $(a, b) \mapsto a^b$ over positive integers is at least exponential: look at the length of the output.
Repeated squaring trick: now the number of multiplications is polynomial, but these will be performed, eventually, on very large numbers. But: this gives a polynomial algorithm for computing $(a, b, m) \mapsto a^b \bmod m$.
The customary algorithm for deciding whether a number is prime, is exponential (in the length of input).
The greatest common divisor of two numbers can be computed in polynomial time, using:

Theorem $\gcd(a, b) = \gcd(b, a \bmod b)$

This gives rise to Euclid's algorithm. Why polynomial-time?

Gives us numbers x, y with

$$\gcd(a, b) = xa + yb.$$

For this, simply maintain such a form for all numbers computed during the algorithm. If

$$\begin{aligned} a' &= x_1 a + y_1 b, \\ b' &= x_2 a + y_2 b, \\ r' &= a - qb' < b' \end{aligned}$$

then

$$r' = (x_1 - qx_2)a + (y_1 - qy_2)b.$$

Analysis of the Euclidean algorithm on numbers of length n, it finishes in $2n$ iterations. So, there is a simple polynomial algorithm to decide whether two numbers are relatively prime.

Primality It is much harder to decide about a number x whether it is prime. The simple-minded algorithm of trying all numbers less than x (or, even only $\sqrt{x}$) is exponential.

By now, a polynomial algorithm has been found for prime testing, but that algorithm is quite complex. There is a much faster, randomized algorithm. Both of these algorithms rely on deeper ideas than our simple-minded enumeration.

Longest common subsequence

The `diff` program in Unix. Given sequences x, y of length n, it is possible to find the longest common subsequence of x and y in $O(n^2)$ steps. (This needs insight: the simple-minded algorithm of trying all subsequences is exponential.)

```
X  =  <A,B,   C,   B,D,A,B>
Y  =  <   B,D,C,A,B,   A>
      <   B,   C,   B,   A>
```

Formally, if $x = (x_1, \ldots, x_n)$ and $y = (y_1, \ldots, y_n)$ then a common subsequence of length k is given by $a_1 < \cdots < a_k$, $b_1 < \cdots < b_k$ with $x_{a_i} = y_{b_i}$ for $i = 1, \ldots, k$.

You must have learned the method, dynamic programming, in your Algorithms class. A recursive solution to subproblems. Let $c[i, j]$ be the length of the longest common subsequence of $(x_1, \ldots, x_i)$ and $(y_1, \ldots, y_j)$.

$$c[i,j] = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0, \\ c[i-1, j-1] + 1 & \text{if } i, j > 0 \text{ and } x_i = y_j, \\ \max(c[i, j-1], c[i-1, j]) & \text{otherwise.} \end{cases}$$

Non-recursive solution: compute the table $c[i, j]$ bottom-up. Also, store the value $b[i, j] = 1, 2$ or 3 depending on whether the optimum is $c[i, j-1], c[i-1, j]$ or $c[i-1, j-1] + 1$. (Can be represented by left, up and back-diagonal arrows in a diagram of the table.)
Recursive solution with saving the results in a table, called caching (memoization).

(Skip in CS332.)
Solving a set of linear equations. The problem of round-off errors.
Rational inputs, exact rational solution.
How large can the numerators and denominators grow?
Determinant, properties:

1. It is a polynomial of its entries.
2. Row operations do not change it.
3. Interpreting it as volume, hence upper bound: product of vector lengths.

Expressing entries during Gaussian elimination as a quotient of determinants. Hence, bound on the size of numerator/denominator; hence, polynomial algorithm.

Hamilton cycle, traveling salesman problem.
Euler cycle.

Witness verifying relation

Example $V_1(x, w)$ is true if, x, w are integers and w is a proper divisor of x.

Example $V_2(G, C)$ is true if G is a graph and C is a sequence of edges of G that is a Hamiltonian cycle of G.

Example Connectivity of a graph.

Example Non-connectivity of a graph.

Polynomial-verifiable witness relations

The witness verifying relation V must be

- polynomially bounded ($|w|$ is polynomial in $|x|$ when $V(x, w)$ is true).
- polynomial-time computable.

All this makes sense after integers, graphs, and so on, are encoded into strings.
Relation $V \subset \Sigma^* \times \Sigma^*$, encoded as a language:

$$\langle V \rangle = \{ \langle x, w \rangle : V(x, w) \}$$
$$V(x, w) (= \text{true}) \Leftrightarrow (x, w) \in V \Leftrightarrow \langle x, w \rangle \in \langle V \rangle.$$

So, instead of a witness relation, we can talk about witness language.

Example Take a set of linear equations

$$a_{11}x_1 + \cdots + a_{1n}x_n = b_1,$$
$$\ddots$$
$$a_{n1}x_1 + \cdots + a_{nn}x_n = b_n.$$

Assume a_{ij}, b_i are integers. Consider the verification relation $V(\{a_{ij}, b_i\}, x)$, which is true if the sequence of fractions $x = (x_1, \ldots, x_n)$, $x_i = y_i/z_i$ (now the witness) is a solution.

Is this relation polynomial-time computable? Yes, but not obviously, since rounding is not allowed! Fortunately, we can compute a common denominator $z_1 z_2 \cdots z_n$: big, but still polynomial-length.

Is it polynomially bounded? Yes, but much less obviously! It is possible to prove that if there is a solution then there is one in which the numerators and denominators are not too large ("Cramer's rule").

Definition of NP

$L \in$ NP if there is a polynomial-verifiable witness relation R such that

$$L = \{\, x \in \Sigma^* : \exists w \in \Sigma^* \; V(x, w) \,\}.$$

We associate two problems with a witness relation V. Given input (instance) string $x \in \Sigma^*$,

Decision problem Decide whether $x \in L$.

Search problem As above, but if yes, also find w with $V(x, w)$.

One and the same language L can be associated with several witness relations.

Example

$$V_1(x, w) \Leftrightarrow x \text{ is odd, } w \text{ is a proper divisor of } x.$$
$$V_2(x, \langle a, b\rangle) \Leftrightarrow x \text{ is odd, } x = a^2 - b^2, \quad a - b > 1.$$

- If $V_1(x, w)$ then $x = v \cdot w$, where $v, w > 1$ are odd. Let $a = (v + w)/2$, $b = (v - w)/2$, then $a - b = w > 1$ and $a^2 - b^2 = v \cdot w = x$, so $V_2(x, \langle a, b\rangle)$ holds.
- If $V_2(x, \langle a, b\rangle)$ then $a - b > 1$ and $x = a^2 - b^2 = (a + b)(a - b)$. Let $w = a - b$, then $V_1(x, w)$ holds.

Maximum clique, minimum node cover, traveling salesman problem.
In general, a question of the sort:

$$\textit{given } x, \textit{ maximize } f(x, y)$$

where $f(x, y)$ is polynomial-time computable.
Turning an optimization problem into a witness relation:

$$V(\langle x, k\rangle, w) \Leftrightarrow \exists w\ f(x, w) \geqslant k.$$

Example Given graph G and integer k, does G have an independent set of size $\geqslant k$?

Example Example graph in Figure 6-9 of Lewis-Papadimitriou: find maximum independent set, maximum clique, minimum node cover.

(Skip in CS332)
Recall: notion of language recognized by a Turing machine.
Definition of a langage enumerated by a Turing machine.

Theorem A language is Turing recognizable iff it is Turing enumerable.

Definition of a nondeterministic Turing machine.
Not a real machine: just another way of speaking about witnesses.
A language recognized by a nondeterministic Turing machine. (We do not define the notion of a language decided by a nondeterministic machine.)

Theorem Language L is in NP iff it is recognized by some nondeterministic polynomial-time bounded Turing machine.

Rewrite rules and grammars

(Skip in cs332)

Grammars also illustrate nondeterminism.

Fix an alphabet Σ.

Rewrite rule (production) $P : u \to v$ for $u, v \in \Sigma^*$.

A rewrite process is a finite set Π of productions. The meaning of $u \Rightarrow_\Pi v$ and $u \Rightarrow_\Pi^* v$. The word problem of a rewrite process: to decide, given Π, u, v, whether $u \Rightarrow_\Pi^* v$.

Grammar: $\Gamma = (\Sigma_0 \subset \Sigma, S \in \Sigma \setminus \Sigma_0)$.

The language $L(\Gamma) = \{ w \in \Sigma_0^* : S \Rightarrow^* w \}$.

Grammars are also more naturally related to nondeterministic computations than to deterministic ones.

Given $a_1, \ldots, a_n, b$, are there $x_1, \ldots, x_n \in \{0, 1\}$ with

$$a_1 x_1 + \cdots + a_n x_n = b?$$

Example: $\{4, 11, 16, 21, 27\}, 25$.
A dynamic programming algorithm (Theorem 5.5.7 of Sipser). Let S_i be the set of numbers $\leqslant b$ of form $a_1 x_1 + \cdots + a_i x_i$. Then

$$S_i = S_{i-1} \cup (a_i + S_{i-1}), \text{ deleting all numbers } > b.$$

We are working with subsets of $\{0, 1, \ldots, b-1, b\}$, so the complexity of this algorithm is

$$O(b \cdot n)$$

times the cost of additions involved.
Is this polynomial?

Partition similar to subset sum. Example: for $38, 17, 52, 61, 21, 88, 25$. the subset $\{38, 52, 61\}$ will do.

Example application: divide some manuscripts between two secretaries.

Maximum cut Proof of NP-completeness (not in cs332)? (See an earlier assignment on "diversity clauses".)

Logic formulas If this is new to you, review it from your discrete math book, or from the course cs210.

- Boolean variables $x_i \in \{0, 1\}$, where 0 stands for false, 1 for true. A logic expression is formed using the connectives $x \wedge y = x \cdot y$ (conjunction), $\neg x = 1 - x$ (negation), $x \vee y = \neg(\neg x \wedge \neg y)$ (disjunction), for example

$$F(x_1, x_2, x_3, x_4) = (x_1 \vee \neg x_2) \wedge (x_2 \vee \neg x_3 \vee x_4).$$

  Other connectives: say $x \Rightarrow y = \neg x \vee y$, $x \oplus y = x + y \bmod 2$ (XOR).
- A (truth-) assignment (say $x_1 = 0, x_2 = 0, x_3 = 1, x_4 = 0$) allows to compute a value (in our example, $F(0, 0, 1, 0) = 0$).

Associativity, commutativity of $\vee$, $\wedge$ and $\oplus$: $a \wedge b = b \wedge a$, $a \vee (b \vee c) = (a \vee b) \vee c$.

Distributivity of $\wedge$ over $\vee$ and of $\oplus$, of$\vee$ over $\wedge$.

De Morgan rules move the negation inside: $\neg(a \vee b) = \neg a \wedge \neg b$.

Conjunctive normal form (CNF) $F(x_1, \ldots, x_n) = C_1 \wedge \cdots \wedge C_k$ where each C_i is a clause, with the form $C_i = \tilde{x}_{j_1} \vee \cdots \vee \tilde{x}_{j_r}$. Here each $\tilde{x}_j$ is either x_j or $\neg x_j$, and is called a literal.

Disjunctive normal form (DNF) Similar, with disjunctions outside and conjunctions inside.

Transform a formula into CNF by distributing $\vee$ over $\wedge$, or into DNF by distributing $\wedge$ over $\vee$.

Theorem Expressing every Boolean function $f : \{0,1\}^n \to \{0,1\}$ via a Boolean formula.

Example $\Delta(x, y, z) = 1$ if x, y, z are not all equal.

Disjunctive normal form (DNF)

$$\Delta(x, y, z) = (x \wedge y \wedge \neg z) \vee (x \wedge \neg y \wedge z) \vee (x \wedge \neg y \wedge \neg z) \\ \vee (\neg \wedge y \wedge z) \vee (\neg x \wedge \neg y \wedge z) \vee (\neg x \wedge \neg y \wedge z).$$

(List all cases in which $\Delta(x, y, z) = 1$.)

Conjunctive normal form (CNF)

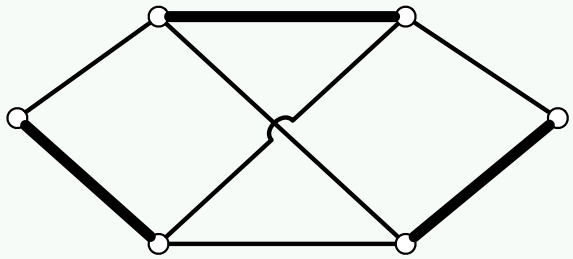
$$\Delta(x, y, z) = (x \vee y \vee z) \wedge (\neg x \vee \neg y \vee \neg z).$$

(More interesting for satisfiability, since it is a list of constraints.)

Satisfiability

- For a given formula, witness is a satisfying assignment.
- Tautologies.
- Special case of satisfiability problem for conjunctive normal forms: SAT.
- A 3-CNF is a conjunctive normal form in which each clause contains at most 3 literals—gives rise to 3SAT.
- 2SAT: as will be seen, solvable in polynomial time.

Example Matching in an undirected graph.



We reduce to SAT, for graph $G = (V, E)$. To each edge $\{u, v\} \in E$ ($u \neq v$, order does not matter) we assign a variable $x_{\{u,v\}}$, with the meaning that this edge is selected into the matching. The formula Φ_G to which translate the graph is the conjunction of a list of constraints. To each point $u \in V$, belong two kinds of constraint.

- For pairs of edges $\{u, v\}, \{u, w\} \in E$, we write $\neg x_{\{u,v\}} \vee \neg x_{\{u,w\}}$, since each point is in at most 1 matching edge.
- If $\{u, v_1\}, \ldots, \{u, v_k\}$ are all the edges adjacent to point u then we write $x_{\{u,v_1\}} \vee \cdots \vee x_{\{u,v_k\}}$, since each point is in at least 1 matching edge.

Logic circuits

Logic formulas can be generalized to logic circuits.

- Acyclic directed graph, where some nodes and edges have labels. Nodes with no incoming edges are input nodes, each labeled by some logic variable $x_1, \ldots, x_n$.
  Nodes with no outgoing edges are output nodes.
- Some edges have labels $\neg$. Non-input nodes are labeled $\vee$ or $\wedge$.
- If there is just one output node, the circuit C defines some Boolean function $f_C(x_1, \ldots, x_n)$. Circuit satisfiability is the question of satisfiability of this function.
- Assume also that every non-input node has exactly two incoming edges.

- The circuit satisfiability problem.
- Specialize: formula satisfiability problem.
- Specialize: CNF satisfiability, that is SAT.
- Specialize: 3SAT.

On the other hand:

Theorem Circuit satisfiability can be reduced to 3SAT.

This is our first nontrivial example of reduction now.

Proof. Introduce a new variable y_i for the output of each gate. The relation of each gate output to its inputs can be expressed by a formula of at most 3 variables: for example $y_i \Leftrightarrow x_j \wedge y_k$. Transform this into a 3CNF G_i. The conjunction of all these gives a 3CNF

$$F(x_1, \ldots, x_n, y_1, \ldots, y_m) = G_1 \wedge \cdots \wedge G_m,$$

where y_m is the output. The satisfiability of the circuit is equivalent to the satisfiability of $F(x_1, \ldots, x_n, y_1, \ldots, y_m) \wedge y_m$ □

Many-one reduction.
Reduction of the decision problems, of the search problems.

Example Shortest path with edge length to shortest path with unit edge length. Why not a reduction?

Example Euler circuit to Hamilton circuit. This does not show that Euler circuit is as difficult as Hamilton circuit; only that Hamilton circuit is as difficult as Euler circuit.

Example CLIQUE ≡ INDEPENDENT

Example INDEPENDENT ≡ NODE COVER

NP-hard problems (Some kind of reduction is understood.)

Example All NP-complete problems given below. Also the halting problem. Also: given a graph, does it have at least k matchings?

NP-complete problems We will see many examples.

Theorem (Cook-Levin) SAT is NP-complete.

Proof. Let $V(x, w)$ be a witness verification relation, computable in time $|x|^c$ on, say, a cellular automaton, over alphabet Γ, with transition function $\delta(a, b, c)$.
Represent the space-time history by a table of size $|x|^c \times 2|x|^c$, where $y_{p,t}$ is the state of cell p at time t. At time $t = 0$, initial condition: $y_{-p,0} = x_p$ for $p \leqslant |x|$, and $y_{p,0} = w_p$ for $p \leqslant |x|^c$.
Convention: $V(x, w) = y_{0,|x^c|}$.
Represent the elements of Γ as binary strings of length $\log |\Gamma|$. Then $\delta(a, b, c)$ is implementable by a constant-size logic circuit. Put together the copies of this little circuit, to compute each $y_{p,t+1}$ from $y_{p-1,t}$, $y_{p,t}$, $y_{p+1,t}$, into a big circuit. Hardwire the x part of the initial condition, then we got a circuit C_x.
This reduces the problem $\exists w V(x, w)$ to the satisfiability problem of the circuit: is there a witness w producing output $C_x(w) = 1$? □

Combinatorial meaning of SAT. Translate the constraints into the independent set problem of a graph.

Completeness of subset sum

Set of linear equations for variable $x_j \in \{0, 1\}$:

$$a_{i1}x_1 + \cdots + a_{in}x_n = b_i, \qquad a_{ij}, b_i \geqslant 0, \quad i = 1, \ldots, m.$$

We reduce 3SAT to the solvability of such a system. Let $F = C_1 \wedge \cdots \wedge C_m$ be a 3CNF. Say $C_i = x_1 \vee \neg x_2 \vee \neg x_5$. The values of x_1, x_2, x_5 satisfy this if and only if the 0-1 equations

$$x_1 + x_2' + x_5' + y_{i1} + y_{i2} = 3,$$
$$x_2 + x_2' = 1, \quad x_5 + x_5' = 1$$

are solved by them, with appropriate values given to the new variables $x_2', x_5', y_{i1}, y_{i2}$. This reduces satisfiability of F to the solvability of a set of equations with 0-1 coefficients a_{ij}. Also, at most 5 of these are different from 0, in each equation.

Reduce the solvability of the above linear 0-1 equations with 0-1 coefficients, at most 5 of which are different from 0, to the solvability of a single equation

$$A_1 x_1 + \cdots + A_n x_n = B.$$

For this, sum up the above equations, multiplying the ith one by 6^{i-1}, that is for example $A_2 = a_{12} + a_{22} \cdot 6 + \cdots + a_{m2} \cdot 6^{m-1}$, $B = b_1 + b_2 \cdot 6 + \cdots + b_m \cdot 6^{m-1}$.
Why did we multiply equation i by the coefficient 6^{i-1} before summing up?

Reducing the SAT to dHAMPATH, the problem of directed Hamilton paths.

- Points $v_{\text{start}}, v_{\text{end}}$, and one point for each of the m clauses C_j.
- For each of the n variables x_i, a doubly linked chain $X_i = v_{i,0} \leftrightarrow v_{i,1} \leftrightarrow \cdots \leftrightarrow v_{i,3m-1} \leftrightarrow v_{i,3m}$.
- $v_{\text{start}} \to v_{1,0}, v_{1,3m}$; $v_{n,0}, v_{n,3m} \to v_{\text{end}}$.
- $v_{i,0}, v_{i,3m} \to v_{i+1,0}, v_{i+1,3m}$ if $i < n$.
- If x_i occurs in C_j then $v_{i,3j-2} \to C_j \to v_{i,3j-1}$.
  If $\neg x_i$ occurs in C_j then $v_{i,3j-1} \to C_j \to v_{i,3j-2}$.

Making x_i true corresponds to traversing X_i from left to right.

Recall the picture after the discussion of a universal Turing machine. A matrix with rows indexed by $\langle M \rangle$ and columns indexed by w: at position $(\langle M \rangle, w)$ sits the result $M(w)$, if it is defined, namely if the computation of M halts on input w. Let us put ∞ where it does not.

	$w_0 = e$	$w_1 = 0$	$w_2 = 1$	$w_3 = 00$	$\ldots$
$\langle M_1 \rangle$	e	∞	0001	e	$\ldots$
$\langle M_2 \rangle$	1101	0	1	1	
$\langle M_3 \rangle$	$M_3(e) = 111$	$M_3(0) = 010$	$M_3(1) = \infty$	$M_3(00) = \infty$	$\ldots$
$\langle M_4 \rangle$					
$\vdots$					$\ddots$

Let us define a non-computable function $D(w)$ explicitly. Recall that $w_1, w_2, \ldots$ lists all binary strings, and $M_1, M_2, \ldots$ lists all Turing machines, with corresponding codes $\langle M_i \rangle$.

$$D(w_i) = \begin{cases} 1 & \text{if } M_i(w_i) = 0, \\ 0 & \text{otherwise.} \end{cases}$$

This function is not computable, since at position w_i we made it different from $M_i(w_i)$, and $M_1(w), M_2(w), \ldots$ is a list of all computable functions.
The construction of $D(\cdot)$ is called the diagonal method, since we defined $D(\cdot)$ as the "complement" of the diagonal of the above table.

The above definition of $D(w)$ sounds like a presciption to compute it, but we just proved that $D(w)$ is not computable. Where is the rub? The condition $M_i(w_i) = 0$ is not simple to test, since M_i may not halt on input w_i. In fact, let us define the language

$$H = \{ (\langle M \rangle, w) : M \text{ halts on input } w \}$$

called the Halting Problem. If there was an algorithm to decide the question $(\langle M \rangle, w) \in H$ for all pairs $(\langle M \rangle, w)$ then we could compute $D(w)$. But we know $D(w)$ is uncomputable. So we proved

Theorem The halting problem is undecidable.

This proof is an example of a reduction: the computation problem of $D(\cdot)$ was reduced to that of H. (Since $D(\cdot)$ is uncomputable, so is H.)

Computable enumerability

Narrower class Computable. Other names for the same notion: decidable, recursive.

Wider class A language is called computably enumerable if it is recognized by some (deterministic) Turing machine.
Other names for the same notion: computably enumerable, recursively enumerable, semidecidable, recognizable.

Theorem A language is computably enumerable if and only if there is a Turing machine that lists its elements on its output tape.

We had this in a homework problem.

You need the following theorem if you learn about nondeterministic machines:

Theorem The languages recognizable by non-deterministic Turing machines are just the computably enumerable languages.

Proof. Deterministic $\Rightarrow$ nondeterministic: this direction is easy. Recognized by nondeterministic $M \Rightarrow$ recognized by deterministic one: breadth-first search over all possible computations of M. This is also called dovetailing. $\square$

A not computably enumerable language

The language H (halting problem) is computably enumerable: indeed, the universal Turing machine can simulate M on w to accept exactly whenever M halts on w.

Theorem A language is decidable iff both it and its complement is computably enumerable.

Hence:

Theorem The complement of H is not Turing recognizable.

Reduction for the sake of proving undecidability is similar to reduction for the sake of proving NP-completeness.
Problem A can be reduced to problem B if a solution of problem B can be used to solve problem A. We will use a more restricted definition, called "mapping reducibility", or many-one reducibility, similar to the polynomial reducibility of NP theory.
Reduction enables us to prove many problems undecidable, even problems that have nothing to do with Turing machines.

Example We have shown that the halting problem H is undecidable by reducing the computation of the diagonal function $D(w)$ to it.

For a machine M, let $L(M)$ denote the set of strings accepted by M. Let $E = \{ \langle M \rangle : L(M) = \emptyset \}$.

Theorem E is undecidable.

Proof. Reduce the halting problem to E.

Input of H: a pair $(\langle M \rangle, w)$.

Output decision about whether M halts on w.

Input of E: a machine T.

Output decision about whether T accepts anything.

From pair $(\langle M \rangle, w)$, we create a machine $T_{M,w}$. On input x:

- if $x \neq w$ reject.
- else simulate M on x; accept if M does.

$$L(T_{M,w}) = \begin{cases} \emptyset & \text{if } M \text{ does not accept } w, \\ \{w\} & \text{if } M \text{ accepts } w. \end{cases}$$

We do not run $T_{M,w}$, just create it. Now, if a TM S could decide whether $L(T_{M,w}) = \emptyset$ then using S, we could decide whether M accepts w. □

Let $\text{EQ} = \{ \langle M_1, M_2 \rangle : L(M_1) = L(M_2) \}$.

Theorem EQ is undecidable.

We will solve this in class.

Post Correspondence Problem (PCP), see the Sipser book.

Example

Kit $\left\{\left[\frac{\mathrm{b}}{\mathrm{ca}}\right], \left[\frac{\mathrm{a}}{\mathrm{ab}}\right], \left[\frac{\mathrm{ca}}{\mathrm{a}}\right], \left[\frac{\mathrm{abc}}{\mathrm{c}}\right]\right\}$.

Match $\left[\frac{\mathrm{a}}{\mathrm{ab}}\right], \left[\frac{\mathrm{b}}{\mathrm{ca}}\right], \left[\frac{\mathrm{ca}}{\mathrm{a}}\right], \left[\frac{\mathrm{a}}{\mathrm{ab}}\right], \left[\frac{\mathrm{abc}}{\mathrm{c}}\right]$.

Theorem PCP is undecidable.

Let MPCP be the problem in which there is a distinguished tile that is required to come first. Let A be the set of pairs $(\langle M \rangle, w)$ such that M accepts w. This is undecidable. We will reduce A to MPCP. Given M, w (assume M never tries to move left from the left end).

Parts of the kit:

1. $\left[\frac{\#}{\#q_0 w_1 \ldots w_n \#}\right]$
2. Head moving right: $(q, a) \mapsto (r, b, R)$. For each pair (q, a) a tile $\left[\frac{qa}{br}\right]$.
3. Head moving left: $(q, a) \mapsto (r, b, L)$. For each triple (q, a, c) a tile $\left[\frac{cqa}{rcb}\right]$.
4. $\left[\frac{a}{a}\right]$ for all a in Γ.
5. $\left[\frac{\#}{\#}\right], \left[\frac{\#}{\sqcup\#}\right]$. This extends the tape if needed.
6. $\left[\frac{a q_{accept}}{q_{accept}}\right], \left[\frac{q_{accept} a}{q_{accept}}\right]$. This eats up the rest of the tape.
7. $\left[\frac{q_{accept}\#\#}{\#}\right]$ completes the match.

Example $\Gamma = 0, 1, 2, \text{␣}$

$$(q_0, 0) \to (q_7, 2, R),\ (q_7, 1) \to (q_5, 0, R),\ (q_5, 0) \to (q_9, 2, L),$$

where $q_9 = q_{\text{acc}}$. Starting configuration q_0010. Kit:

$$\left[\frac{\#}{\#q_0010\#}\right], \left[\frac{0}{0}\right], \left[\frac{1}{1}\right], \left[\frac{2}{2}\right], \left[\frac{\#}{\#}\right], \left[\frac{\#}{\text{␣}\#}\right], \left[\frac{q_00}{2q_7}\right], \left[\frac{q_71}{0q_5}\right], \left[\frac{2q_50}{q_920}\right],$$
$$\left[\frac{q_90}{q_9}\right], \left[\frac{q_92}{q_9}\right], \left[\frac{2q_9}{q_9}\right], \left[\frac{q_9\#\#}{\#}\right]$$

A correspondence:

$$\left[\frac{\#}{\#q_0010\#}\right]\left[\frac{q_00}{2q_7}\right]\left[\frac{1}{1}\right]\left[\frac{0}{0}\right]\left[\frac{\#}{\#}\right]\left[\frac{2}{2}\right]\left[\frac{q_71}{0q_5}\right]\left[\frac{0}{0}\right]\left[\frac{\#}{\#}\right]\left[\frac{2}{2}\right]\left[\frac{0q_50}{q_902}\right]\left[\frac{\#}{\#}\right]$$
$$\left[\frac{2q_9}{q_9}\right]\left[\frac{0}{0}\right]\left[\frac{2}{2}\right]\left[\frac{\#}{\#}\right]\left[\frac{q_90}{q_9}\right]\left[\frac{2}{2}\right]\left[\frac{\#}{\#}\right]\left[\frac{q_92}{q_9}\right]\left[\frac{\#}{\#}\right]\left[\frac{q_9\#\#}{\#}\right]$$

Reducing MPCP to PCP. Notation:

$$*u = *u_1 * u_2 * \cdots * u_n,$$
$$u* = u_1 * u_2 * \cdots * u_n*,$$
$$*u* = *u_1 * u_2 * \cdots * u_n * .$$

Convert an instance $\left\{\left[\frac{\mathtt{t}_1}{\mathtt{b}_1}\right], \ldots, \left[\frac{\mathtt{t}_n}{\mathtt{b}_n}\right]\right\}$ of MPCP into the following instance of PCP:

$$\left[\frac{*\mathtt{t}_1}{*\mathtt{b}_1*}\right],$$
$$\left[\frac{*\mathtt{t}_1}{\mathtt{b}_1*}\right], \ldots, \left[\frac{*\mathtt{t}_n}{\mathtt{b}_n*}\right],$$
$$\left[\frac{*@}{@}\right].$$

This will force $\left[\frac{*\mathtt{t}_1}{*\mathtt{b}_1*}\right]$ to be the first tile used, without requiring it specially.

- In case of NP problems, the approximation question makes sense for optimization. We will formulate it only for maximization problems, where we maximize a positive function. For object function $\mathrm{obj}(x, y)$ for $x, y \in \{0,1\}^n$, the optimum is

$$M(x) = \max_y \mathrm{obj}(x, y)$$

  where y runs over the possible .
- For $1 \leqslant k < \infty$, an algorithm $A(x)$ is a k-approximation if

$$\mathrm{obj}(x, A(x)) \geqslant M(x)/k.$$

  Analogous concept for miminization.

Given an undirected graph $G = (V, E)$ with edge length $w_{ij} \geqslant$ for each edge (i, j). If (i, j) is not an edge then $w_{ij} = 0$. Special case: unit edgelengths: $w_{ij} = 1$.

- A cut: a nonempty set $S \subset V$. The value of the cut: $\text{val}(S) = \sum_{i \in S, j \notin S} w_{ij}$.
- We want to find S with maximal $\text{val}(S)$.

Example The nodes in G are points in some space, say a many-dimensional space corresponding data about customers. The edge lengths are distances.
A maximum cut divides the nodes into two groups (clusters) with the property that in general, elements in the same cluster are close to each other and elements in different clusters are far from each other.

Try local improvements as long as you can.

Example (MAXIMUM CUT) Repeat: find a point on one side of the cut whose moving to the other side increases the cutsize.

Theorem If you cannot improve anymore with this algorithm then you are withing a factor 2 of the optimum.

Proof. The unimprovable cut contains at least half of all edges. □

Maximum cut for weighted edges

- The greedy algorithm brings within factor 2 of the optimum also in the weighted case. But does it take a polynomial number of steps?
  - Yes, when edge weights are 0 or 1 (you must know why).
  - No, in the general case: there are examples when it can take an exponential number of steps (what is an upper bound?).
- New idea: decide each "$v \in S$?" question by tossing a coin. (See later.) The expected weight of the cut is $\frac{1}{2} \sum_e w_e$, since each edge is in the cut with probability $1/2$.
- Semidefinite programming increases the ratio of expected cut further.

Less greed is sometimes better

What does the greedy algorithm for vertex cover say? The following, less greedy algorithm has better performance guarantee.

$C \leftarrow \emptyset$
$E' \leftarrow E[G]$
while $E' \neq \emptyset$ **do**
    let (u, v) be an arbitrary edge in E'
    $C \leftarrow C \cup \{u, v\}$
    remove from E' every edge incident on u or v
return C

Theorem The above algorithm has a ratio bound of 2.

Indeed, each added pair of nodes belongs to an edge that is disjoint from all edges chosen earlier. So even the optimal cut must contain at least one of each of these node pairs.

This algorithm does not help at all in the approximation of the size of the maximum independent set, even though finding it is equivalent to findind a minimum vertex cover.
Indeed, consider an example where a maximum independent set has size $n^{1/2}$. A minimum vertex cover, of size $n - n^{1/2}$, is approximated within 2 even when we choose *all points*: a vertex cover of size n. Its complement, of size 0 (but even if we choose a set of size 1) is not a good approximation to $n^{1/2}$.

Fully approximable version of knapsack

The knapsack problem is defined as follows.
Given: integers $b \geqslant a_1 \geqslant \ldots \geqslant a_n$, and integer weights
$w_1 \geqslant \cdots \geqslant w_n$.
Maximize $\sum_j w_j x_j$, subject to $\sum_j a_j x_j \leqslant b$, with $x_j \in \{0, 1\}$.

- Example: a thief, items with volumes a_j and values w_j, knapsack of volume b.
- NP-hard, since we get SUBSET SUM by setting $w_j = a_j$ and asking whether the optimum is b.
- For every ε, we show a polynomial, $(1 + \varepsilon)$-approximation algorithm.
- Even in the subset sum case, the equivalent problem: minimizing $b - \sum_i x_i a_i$, is not approximable, for a trivial reason. Can be 0, and no non-zero approximation is within constant factor of 0.

Dynamic programming: For $1 \leqslant k \leqslant n$,

$$A_k(p) = \min\{\, \boldsymbol{a}^T \boldsymbol{x} : \boldsymbol{w}^T \boldsymbol{x} \geqslant p,\ x_{k+1} = \cdots = x_n = 0 \,\}.$$

If the set is empty the minimum is ∞, and let $A_k(x) = 0$ for $x \leqslant 0$. Let $w = w_1 + \cdots + w_n$. The vector $(A_{k+1}(0), \ldots, A_{k+1}(w))$ can be computed by a simple recursion from $(A_k(0), \ldots, A_k(w))$.

$$A_{k+1}(p) = \min\{\, A_k(p), a_{k+1} + A_k(p - w_{k+1}) \,\}.$$

The optimum is $\max\{\, p : A_n(p) \leqslant b \,\}$.
Complexity: roughly $O(nw)$ steps.
Why is this not a polynomial algorithm?

Idea for approximation: break each w_i into a smaller number of big chunks, and use dynamic programming. Let $r > 0$, $w'_i = \lfloor w_i / r \rfloor$.

$$\begin{array}{ll} \text{maximize} & (\boldsymbol{w}')^T \boldsymbol{x} \\ \text{subject to} & \boldsymbol{a}^T \boldsymbol{x} \leqslant \quad b, \\ & x_i = 0, 1, \ i = 1, \ldots, n. \end{array}$$

For the optimal solution $\boldsymbol{x}'$ of the changed problem, estimate $\frac{\boldsymbol{w}^T\boldsymbol{x}'}{\text{opt}} = \frac{\boldsymbol{w}^T\boldsymbol{x}'}{\boldsymbol{w}^T\boldsymbol{x}^*}$. We have

$$\begin{aligned}\boldsymbol{w}^T\boldsymbol{x}'/r \geqslant (\boldsymbol{w}')^T\boldsymbol{x}' \geqslant (\boldsymbol{w}')^T\boldsymbol{x}^* \geqslant (\boldsymbol{w}/r)^T\boldsymbol{x}^* - n,\\ \boldsymbol{w}^T\boldsymbol{x}' \geqslant \text{opt} - r \cdot n = \text{opt} - \varepsilon w_1,\end{aligned}$$

where we set $r = \varepsilon w_1/n$. This gives

$$\frac{(\boldsymbol{w})^T\boldsymbol{x}'}{\text{opt}} \geqslant 1 - \frac{\varepsilon w_1}{\text{opt}} \geqslant 1 - \varepsilon.$$

With $w = \sum_i w_i$, the amount of time is of the order of

$$nw/r = n^2 w/(w_1\varepsilon) \leqslant n^3/\varepsilon,$$

which is polynomial in $n, (1/\varepsilon)$.

Approximation classes

Fully approximable For every ε, there is a $1 + \varepsilon$-approximation.
Example: appropriate version of KNAPSACK

Partly approximable There is an constant lower bound $k_{\min} > 1$ on the achievable approximation ratio.
Example: MAXIMUM CUT, VERTEX COVER, MAX-SAT.

Inapproximable Example: INDEPENDENT SET (deep result). The approximation status of this problem is different from VERTEX COVER, despite the close equivalence between the two problems.

Random variables, expected value

I assume that you learned about random variables already in an introductory course (Probability in Computing, or its equivalent). For a random variable X with possible values $a_1, \ldots, a_n$, its expected value $\mathbb{E}X$ is defined as

$$a_1 \mathbb{P}\{X = a_1\} + \cdots + a_n \mathbb{P}\{X = a_n\}.$$

Example: if Z is a random variable whose values are the possible outcomes of a toss of a 6-sided die, then

$$\mathbb{E}Z = (1 + 2 + 3 + 4 + 5 + 6)/6 = 3.5.$$

Example: If Y is the random variable that is 1 if $Z \geqslant 5$, and 0 otherwise, then

$$\mathbb{E}Y = 1 \cdot \mathbb{P}\{Z \geqslant 5\} + 0 \cdot \mathbb{P}\{Z < 5\} = \mathbb{P}\{Z \geqslant 5\}.$$

Theorem For random variables X, Y (on the same sample space):

$$\mathbb{E}(X+Y) = \mathbb{E}X + \mathbb{E}Y.$$

Example For the number X of spots on top after a toss of a die, let A be the event $2|X$, and B the event $X > 1$. Dad gives me a dime if A occurs and Mom gives one if B occurs. What is my expected win?
Let I_A be the random variable that is 1 if A occurs and 0 otherwise.

$$\mathbb{E}(I_A + I_B) = \mathbb{E}I_A + \mathbb{E}I_B = \mathbb{P}(A) + \mathbb{P}(B) = 1/2 + 5/6 \text{ dimes}.$$

Chebyshev-Markov inequality

The following theorem helps draw implications about probabilities with the help of the expected value.

Theorem (Chebyshev-Markov) If X is a nonnegative random variable then

$$\mathbb{P}\{\, X > c \cdot \mathbb{E}(X) \,\} < 1/c.$$

```
Quicksort(A)

    Partition(b,A)              // into arrays A1,A2
    Quicksort(A1)
    Quicksort(A2)
```

Average case analysis Assume that b is chosen as $A[1]$. Worst case $\Theta(n^2)$ comparisons. Average case (over all possible orders of A): only $O(n \log n)$ comparisons.

Randomization Choose b randomly. Then for every order of A, the average number of comparisons (over all possible choices of b in all recursive calls) is only $O(n \log n)$.

We are now interested in the second kind of use of randomness: algorithms of this kind are called randomized.

This works both for average case and for randomization.
Let the sorted order be $z_1 < z_2 < \cdots < z_n$. If $i < j$ then let

$$Z_{ij} = \{z_i, z_{i+1}, \ldots, z_j\}.$$

Let $C_{ij} = 1$ if z_i and z_j will be compared sometime during the sort, and 0 otherwise.
The only comparisons happen during a partition, with the pivot element. Let π_{ij} be the first (random) pivot element entering Z_{ij}. A little thinking shows:

Lemma We have $C_{ij} = 1$ if and only if $\pi_{ij} \in \{z_i, z_j\}$. Also, for every $x \in Z_{ij}$, we have

$$\mathbb{P}\left\{\pi_{ij} = x\right\} = \frac{1}{j - i + 1}.$$

It follows that $\mathbb{P}\left\{C_{ij}=1\right\}=\mathbb{E}C_{ij}=\frac{2}{j-i+1}$. The expected number of comparisons is, with $k=j-i+1$:

$$\sum_{1\leqslant i<j\leqslant n}\mathbb{E}C_{ij}=\sum_{1\leqslant i<j\leqslant n}\frac{2}{j-i+1}=2\sum_{k=2}^{n}\sum_{i=1}^{n-k+1}\frac{1}{k}$$
$$=2\sum_{k=2}^{n}\frac{n-k+1}{k}<2(n-1)\left(\frac{1}{2}+\frac{1}{3}+\cdots+\frac{1}{n}\right).$$

From analysis we know $1+\frac{1}{2}+\frac{1}{3}+\cdots+\frac{1}{n}=\ln n+O(1)$. Hence the average complexity is $O(n\log n)$.

- The Chebyshev-Markov inequality helps now estimate the probability that the number of comparisons is larger than, say, $10n\log n$.
- Analysis of the variance of the random cost can also give lower bounds.

The factoring problem seems very hard. But to test a number for having factors is much easier than to find them.
Recently, a polynomial algorithm has been found for testing primes. But the randomized algorithms found around 1980 are still much faster. The book shows how it is done, I discuss it here only on a high level. It is an algorithm $A(x, r)$ that, given number x and some coin-toss sequence r, says "yes"/"no".

- If x is prime then $A(x, r) =$ "yes".
- If x is composite then $A(x, r) =$ "no" with probability $\geqslant 1/2$. In this case, r serves as a witness of nonprimality.

After repeating the test k times, the probability of "yes" for composite x is decreased to $(1/2)^k$.
The witness above is not a factor of x, and does not help in factorization.

Given a function $f(x)$, is it 0 for *all* input values x?
The function may be given by a formula, or by a complicated program.

Example

$$\det(A_1 x_1 + \cdots + A_k x_k + A_{k+1})$$

where the A_i are $n \times n$ matrices.

Lemma A degree d polynomial of one variable has at most d roots.

Proof. See book, this is very well-known. □

So, if we find $P(r) = 0$ on a random r, this can only happen if r hits one of the d roots.

Lemma (Schwartz) Let $p(x_1, \ldots, x_m)$ be a nonzero polynomial, with each variable having degree at most d. If $r_1, \ldots, r_m$ are selected randomly from $\{1, \ldots, f\}$ then the probability that $p(r_1, \ldots, r_m) = 0$ is at most md/f.

Proof. Induction on m. $p(x_1, \ldots, x_m) = p_0 + x_1 p_1 + \cdots + x_1^d p_d$, where at least one of the p_i, say p_j, is not 0. Let $p'(x_1) = p(x_1, r_2, \ldots, r_m)$.

$$\begin{array}{rcrcl} p_j(r_2, \ldots, r_m) = 0 & & p'(r_1) = 0 & & \\ (m-1)d/f & + & d/f & = & md/f. \end{array}$$

□

Randomized complexity classes

Benefits of a randomized algorithm? Some possibilities:

Expected running time The result is always correct, but only a polynomial expected running time is guaranteed. Example: Quicksort

Optimization Runs in polynomial time, and the expected value of the result is close to the optimum. Example: Maximum cut.

Deciding a language, one-sided Runs in polynomial time, to answer the question "$x \in L$?" If the answer is "yes", it is never wrong. If the answer is "no", it is wrong probability $\leqslant 1/2$. Example 1: L is the set of composite numbers, and the algorithm is the randomized primality test. Example 2: Polynomial non-identity. We say that L is in the class **RP** if there is such an algorithm.

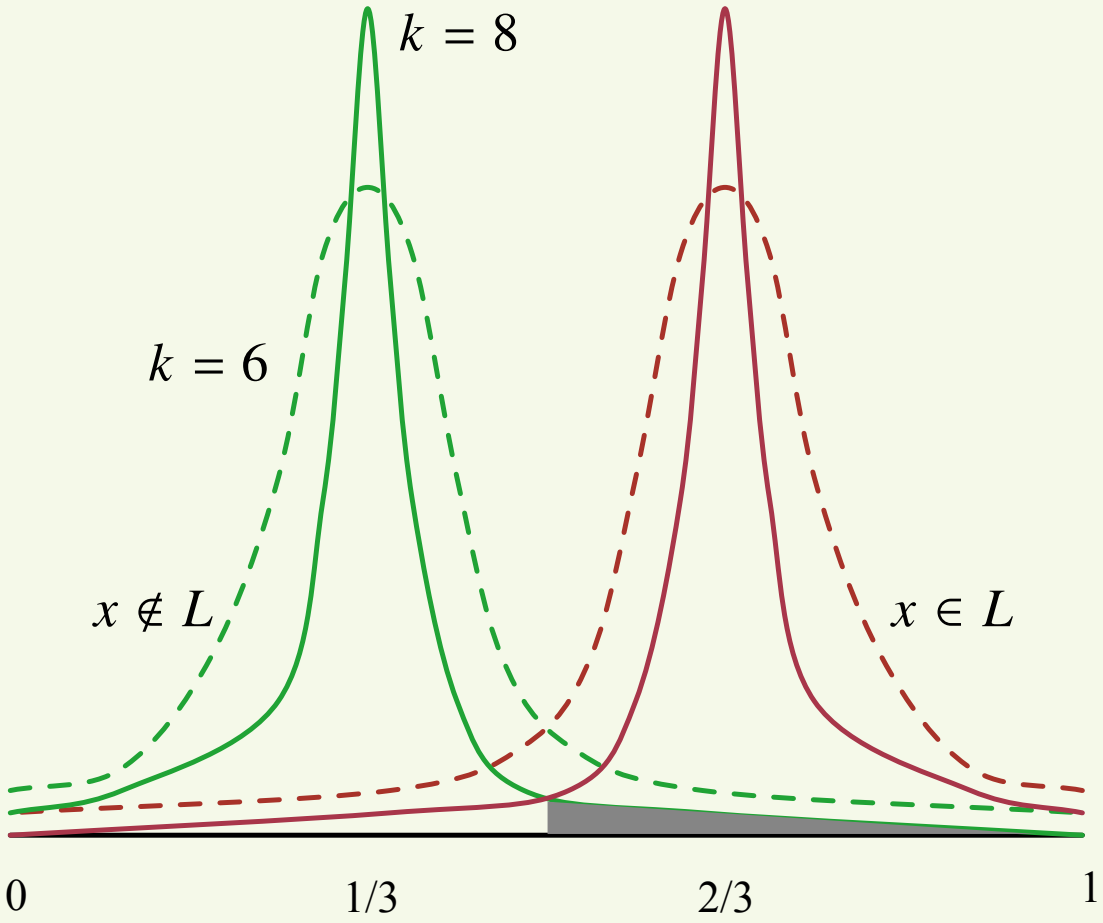
Deciding a language, two-sided Runs in polynomial time, to answer the question "$x \in L$?" The answer can be wrong with probability $\leqslant 1/3$. We say that L is in the class **BPP** if there is such an algorithm. Example: I do not have a natural one...

RP versus BPP

- The examples we have seen are for **RP**. I do not know of simple and natural examples of a language that is in **BPP** but probably not in **RP**. Here is an artificial one: $L = L_1 \setminus L_2$, where $L_i \in \mathbf{RP}$.
- Still, **BPP** is a very important class: the class of languages decidable in randomized polynomial time.

Amplification: decreasing the error probability

Very simple for **RP**: just repeat the experiment.
For **BPP**: repeat and take majority. The proof is a version of the Law of Large Numbers.
Given a language L, let algorithm M decide whether input x is in L, with error probability $q < 1/2$. Construct another algorithm M' that recognizes L with a much smaller probability $(4q(1-q))^k$.
(Show that this is always less than 1!)
So, the error probability can be made even exponentially small.
Algorithm M', on input x repeats $2k$ times the calculations of $M(x)$, then takes the majority.



The value of each curve shows (for $x \notin L$, $x \in L$, $k = 6$, $k = 8$), at position i/k, the probability of i successes after k repetitions. The grey area is the probability that with $x \notin L$, the majority decision is acceptance after $k = 8$ repetitions.

There are $2k+1$ possibilities for the number m of accepting computatons: $0, 1, \ldots, 2k$. Accept if $m \geqslant k$.

$$\mathbb{P}\{\, M' \text{ accepts when it should not} \,\} =$$

$$\mathbb{P}\{\, m \geqslant k \,\} = \sum_{i=k}^{2k} \mathbb{P}\{\, m = i \,\} = \sum_{i=k}^{2k} \binom{2k}{i} q^i (1-q)^{2k-i}.$$

Since $i \geqslant k$ and $q < 1/2$, we have $q^i(1-q)^{2k-i} < q^k(1-q)^k$, so

$$\mathbb{P}\{\, m \geqslant k \,\} \leqslant \sum_{i=0}^{2k} \binom{2k}{i} q^k (1-q)^k = 2^{2k} q^k (1-q)^k = (4q(1-q))^k.$$

Similarly, of course,

$$\mathbb{P}\{\, M' \text{ rejects when it should not} \,\} \leqslant (4q(1-q))^k.$$

Example that is hard for a deterministic algorithm but easy for a randomized one.

```
L1:     x_1 ?  goto L3 : goto L2
L2:     x_2 ?  goto L6 : goto L4
L3:     x_3 ?  goto L7 : goto L5
L4:     x_3 ?  goto L7 : goto L6
L5:     x_2 ?  goto L7 : goto L6
L6:     return 0
L7:     return 1
```

Each line outputs 0 or 1, or reads a variable and depending on its value, performs a GOTO to some later lines.
Equivalent description: a directed acyclic graph. Each node corresponds to a program line, outgoing edges are labeled by 0/1. At each node it is shown which variable it reads.
Draw the graph corresponding to the above program.

Theorem The equivalence problem for branching programs is co-NP-complete.

A branching program is read-once (ROBP) if each variable is read at only once in each execution (directed path from start to finish). Every Boolean function can be computed by a read-once branching program. Language

$$E_{\text{ROBP}} = \{ (B_1, B_2) : B_1 \text{ and } B_2 \text{ are equivalent ROBP} \}.$$

Theorem E_{ROBP} is in **BPP** (decidable in randomized polynomial time).

Can testing on some random inputs help? It is hard to get guarantees when x_i is chosen from $\{0, 1\}$.

Arithmetization View Boolean expressions as polynomials. Namely, assign polynomials to each node recursively.

Nodes The input node gets 1.

Each non-input node gets the sum of polynomials on its incoming edges

Edges If a node testing x has polynomial p, assign

px to the 1 out-edge

$p(1-x)$ to the 0 out-edge

Represent as sum of products over all paths. Example product:

$$x_1 x_2 (1 - x_3)(1 - x_5).$$

Make sure each variable appears in each product:

$$x_1 x_2 (1 - x_3) x_4 (1 - x_5) + x_1 x_2 (1 - x_3)(1 - x_4)(1 - x_5).$$

Like DNF, but $+$ in place of $\vee$, and x_i is not restricted to $\{0, 1\}$.

Lemma Two read-once programs are equivalent if and only if their polynomials are equal.

Gödel's incompleteness theorem

The topic of these lectures is a new kind of undecidability. In computation theory, undecidability is the property of a language L. This language gives rise to a family of questions of the type $x \in L$?. The language is undecidable if there is no Turing machine that answers the whole family of such questions.
Most mathematics can be represented formally. Mathematical proof, when written out in all formal detail, can be checked algorithmically. A theory is a method to prove theorems, and a sentence S is undecidable if neither S nor its negation can be proved. So, in logic/mathematics, undecidability is the property of a sentence in relation to a theory.

Example The continuum hypothesis: the statement that there is no cardinality between the size of the set of natural numbers and the size of the set of real numbers. There is a logical theory, ZFC (the Zermelo-Fraenkel set theory with the axiom of choice) suitable for deriving essentially all known theorems of mathematics. It is known (by theorems of Gödel and Cohen) that the continuum hypothesis is undecidable in ZFC.

Example Euclidean geometry is the first example of a mathematical theory. The so-called axiom of parallels is known to be undecidable in the rest of the theory (independent from it).

Mathematics deals with sentences (statements about some mathematical objects). For a while, let us abstract away from the structural content of these sentences: they are just strings in some finite alphabet. Assumptions:

a. Permitted sequences form a decidable language L: they are distinguishable from (formal) nonsense.
b. There is a computable function Neg assigning to each sentence ϕ, another sentence ψ called its negation.

So, a logical language is a pair $\mathcal{L} = (L, \mathrm{Neg})$.

Example For logical language $\mathcal{L}_1 = (L_1, \mathrm{Neg})$, the set L_1 consists of all expressions of the form $l(a, b)$ and $l'(a, b)$ where a, b are natural numbers (in decimal representation). So, $l(2, 4)$ and $l(15, 0)$ are examples of sentences. The sentences $l(a, b)$ and $l'(a, b)$ are each other's negations: $\mathrm{Neg}(l(a, b)) = l'(a, b)$, $\mathrm{Neg}(l'(a, b)) = l(a, b)$.

A formal system $\mathbf{F} = (\mathcal{L}, V)$ is given by a logical language $\mathcal{L}$ and a decidable relation

$$V(T, P)$$

(the proof verifying relation). A sentence T for which there is a proof in $\mathbf{F}$ is called a theorem of $\mathbf{F}$. The set of theorems of system $\mathbf{F}$ is called the theory of $\mathbf{F}$, written as

$$\mathrm{Th}(\mathbf{F}) = \{\, T \in L : \exists P \; V(T, P) \,\}.$$

Proofs are like witnesses in the NP framework.

Example A simple formal system $\mathbf{T}_1$ based on the language $\mathcal{L}_1$ above. Let us call axioms all $l(a, b)$ where $b = a + 1$. A proof is a sequence $S_1, \ldots, S_n$ of sentences with the following property. S_i is either an axiom or there are $j, k < i$ and a, b, c with $S_j = l(a, b)$, $S_k = l(b, c)$ and $S_i = l(a, c)$. This is a proof for S_n.
This system has a proof for all formulas of the form $l(a, b)$ where $a < b$.

A formal system (or its theory) is called consistent if for no sentence can both it and its negation be a theorem. Inconsistent formal systems are uninteresting, but sometimes we do not know whether a formal system is consistent.
A sentence S is called undecidable in a theory $\mathcal{T}$ (or, equivalently, independent of this theory) if neither S nor its negation is a theorem in $\mathcal{T}$. A consistent formal system (or its theory) is complete if it has no undecidable sentences.

Example The toy formal system $\mathbf{T}_1$ above is incomplete since it has no proof of either $l(5,3)$ or $l'(5,3)$. It becomes complete, say, by adding as new axioms all formulas of the form $l'(a,b)$ with $a \geqslant b$.

Completeness is not always desirable.

Example The language of group theory consists of expressions formed from: variable symbols, binary operation symbol $*$ called multiplication, equation symbol $=$, parentheses, and logical symbols $\wedge, \vee, \neg, \exists, \forall$. The formal system: some axioms like

$$\forall a \forall b \forall c \ (a * (b * c) = a * (b * c)), \quad \exists e \forall a \ (a * e = a),$$

and so on, all needed properties of groups. Further, general axioms related to logic and equality, and general deduction rules of logic. Such a theory is not meant to be complete: should describe the group of permutations as well as the group of invertible matrices, and also the integers when $*$ is interpreted as addition. In this theory the sentence $\forall a \forall b (a * b = b * a)$ is undecidable: true for addition of natural numbers, false for multiplication of permutations.

An incomplete theory is more general than any theory containing it: its theorems have wider validity.

Complete theories have a desirable algorithmic property.

Theorem If a formal system **T** is complete then there is an algorithm that for each sentence S finds in **T** a proof either for S or for its negation.

Thus, if there are no logically undecidable sentences then the set of theorems is algorithmically decidable.

Proof. The algorithm starts enumerating all possible finite strings P and checks whether P is a proof for S or a proof for the negation of S. Sooner or later, one such proof turns up, since it exists. □

Example The language of real numbers consists of expressions formed from the following: variable symbols, an abstract binary operations $*$, $+$, the relation $\leqslant$. Further we have the equation symbol $=$, parentheses, and logical symbols $\wedge$, $\vee$, $\neg$, $\exists$, $\forall$. The formal system is given by some axioms

$$\forall a \forall b \forall c\ (a*(b*c) = a*(b*c)), \quad \forall a \forall b\ (a+b = b+a), \quad \forall a\ (a*a \geqslant 0),$$

and so on, spelling out all important properties of real numbers. To this are added a few general axioms related to logic and equality, and some general deduction rules of logic. Tarski proved that this theory is complete. This implies that there is a decision algorithm to decide every such sentence about real numbers.

Arithmetization and Gödel's Theorem

We may feel that there is only one set of natural numbers, so we may want a system for it that is as complete as possible. Suppose that we want to develop such a formal system. Strings, tables, and so on can be encoded into natural numbers, so this formal system must express all statements about such things as well. Turing machine transitions and configurations are just tables and strings: using natural numbers we can therefore speak about a Turing machine, and about whether it halts. These actions of encoding everything into numbers is called arithmetization.

Let L be some fixed computably enumerable, non-computable set of integers. An arithmetical theory $\mathbf{T}$ is rich if there is a computable function that to each number n, assigns a sentence ϕ_n such that $n \in L$ iff $\phi_n \in \mathbf{T}$.

A reasonable formal system of natural numbers should be rich: since $n \in L$ is checkable by computation, there should be also a proof for this!

Here is one of the most famous theorems of mathematics, which has not ceased to exert its fascination on people with philosophical interests:

Theorem (Gödel's first incompleteness theorem) Every rich formal system is incomplete.

Proof. If the formal system were complete then, according to the above theorem, it would give a procedure to decide all sentences ϕ_n. This could be used to decide $n \in L$, which is impossible. □

How can the theory of real numbers be complete and the theory of natural numbers not? Are not all natural numbers real?
They are. But in the formal system of real numbers there is no expression that is satisfied only by natural numbers: the concept of an arbitrary natural number is not expressible in that system.

For any formal system **F**, it is possible encode the proof verification process into arithmetical formulas. If **F** is "sufficiently strong", then there is a formula $\Gamma_{\mathbf{F}}$ expressing the fact that **F** is consistent.
With an appropriate definition of sufficiently strong, then the following theorem holds (we will not prove it):

Theorem (Gödel's second incompleteness theorem) If the system **F** is sufficiently strong and consistent then $\Gamma_{\mathbf{F}}$ is undecidable in it.

Thus, the consistency of a sufficently strong system **F** of natural numbers cannot be proved by tools that do not go "beyond" it: we have to rely on our "intuition". This theorem showed the failure of the so-called Hilbert program.

Cryptography in history.

- Caesar cypher.
- Cryptanalysis in WWII. The Enigma, the Magic, Turing.
- Commercial and privacy needs in the present.
- Systematic cryptology based on complexity theory started in mid 1970s.

Varieties of cryptography:

Secret algorithm, or secret key Examples: DES (Data Encryption Standard). The mask in Jules Verne's Sándor Mátyás.

One-time pad Shannon showed that this is the only way to achieve information-theoretic (the strongest) security.

Private-key versus public-key cryptosystems Notation: M is the message, C is the cyphertext.

Private key

Sender and receiver share the secret key k:

$$E(k, M) = C, \quad D(k, C) = M.$$

A participant must have a different key for each partner: n participants need $n^2/2$ keys.

Public key

Let e be the public encryption key, d the private decryption key.

$$E(e, M) = C, \quad D(d, C) = M.$$

For n participants, only $2n$ keys are needed. The public keys can be published in a directory.
This assumes $D(d, E(e, M)) = M$. Suppose that the functions are also inverse in the other direction:

$$E(e, D(d, M)) = M.$$

Then the system can also be used for signatures. ("I do.")

Questions:

What are the mathematical requirements? There are many, we will see the definition of one-way functions and trap-door functions.

What are the proposed implementations? We will see RSA.

Are the proposed implementations proved to satisfy those requirements? No.

What useful weaker statements are proved? Some of the schemes have the property that breaking them would give mathematical algorithms that have been long sought.

Easy to compute, hard to invert. Function f is one-way if:

- Computable in polynomial time
- Inverting it is hard: For every probabilistic polynomial-time algorithm M, every c and sufficiently large n, random w of length n:

$$\mathbb{P}\{f(M(f(w))) = f(w)\} \leqslant n^{-c}.$$

It is not known whether one-way functions exist. (If they do then P ≠ NP.) Some functions seem to be one-way.

Example Input: large primes p, q. Output: pq.

Nice cryptographic applications.

- A provably secure private-key encryption scheme.
- Pseudorandom generators
- Password system, where each password is encrypted via a one-way function and stored in this form.

Trap-door functions

Like a one-way function, but knowing a "trapdoor" key allows to invert it.

- Length-preserving polynomial-time encoding function $E : (\Sigma^*)^2 \to \Sigma^*$.
- Polynomial-time inverting function $D : (\Sigma^*)^2 \to \Sigma^*$.
- Auxiliary probabilistic polynomial time algorithm G, produces (key, trapdoor) pair (e, d).

Requirements:

- Inverting without the trapdoor key is hard: For every probabilistic polynomial-time computable function f, every constant c and sufficiently large n, random output (e, d) of G on 1^n, and random w of length n, with $v = f(E(e, w))$, we have

$$\mathbb{P}\{ E(e, v) = E(e, w) \} \leqslant n^{-c}.$$

- Inverting with the trapdoor key is easy: For every n, every w of length n, every output (e, d) of G that occurs with nonzero probability, with $v = D(d, E(e, w))$,

$$E(e, v) = E(e, w).$$

The notation

$$a \equiv b \pmod{m}.$$

This is called congruence. Its meaning is the same as

$$a \bmod m = b \bmod m.$$

If the modulus is the same, you can add, subtract or multiply two congruences. We will be interested in arithmetical operations with respect to a fixed modulus.

Theorem Given integers a, e, m, it takes only polynomial time to compute

$$a^e \bmod m.$$

Division modulo m

Theorem For every natural number $m > 1$ and every u relatively prime to m, there is a v with

$$uv \equiv 1 \pmod{m}.$$

This v can be found in polynomial time.

Proof. Use the Extended Euclidean Algorithm to solve $ux + my = 1$. □

The set of remainders modulo m is called $\mathbb{Z}_m$. It is a ring ($+$, $*$, usual rules apply to these operations).. The set of remainders modulo m relatively prime to m is called $\mathbb{Z}_m^*$. This is a group with respect to multiplication (every element has an inverse).
If m is a prime then, of course, $\mathbb{Z}_m^* = \{1, \ldots, m-1\}$. In this case, we can "divide" in $\mathbb{Z}_m$ by any nonzero element, so $\mathbb{Z}_m$ is a field.

Fermat's theorem

For $a \in \mathbb{Z}_p^*$, look at the sequence $a, a^2, \ldots,$. Eventually, it begins to repeat. First, it becomes 1. The smallest number n such that $a^n \equiv 1$ is called the order of n. It is easy to show (exercise) that if $a^k \equiv 1m$ then the order of a divides k.

Theorem (Fermat) Let p be a prime and a a number not divisible by p. Then $a^{p-1} \equiv 1 \pmod{p}$.

Generalization: Given a number m with prime divisors $p_1, \ldots, p_k$, we denote

$$\phi(m) = m(1 - 1/p_1) \cdots (1 - 1/p_k).$$

For example, if $m = pq$ then $\phi(m) = (p-1)(q-1)$.

Theorem (Euler)

a. The number of integers in $\{1, \ldots, m-1\}$ relatively prime to m is $\phi(m)$.

b. If $\gcd(a, m) = 1$ then $a^{\phi(m)} \equiv 1 \pmod{m}$.

Given a composite modulus m (say, the product of two large primes), and an integer exponent e, it is easy to compute $a^e \bmod m$ but, this operation is, in general, hard to invert: to find a given a^e. However, this becomes easy if

a. $\gcd(a, m) = 1$,
b. $\gcd(e, \phi(m)) = 1$,
c. we know $\phi(m)$.

Just find d with $ed \equiv 1 \pmod{\phi(m)}$ and compute

$$(a^e)^d = a^{ed} = a^{k\phi(m)+1} = a \cdot a^{k\phi(m)} \equiv a \pmod{m}.$$

Without knowing $\phi(m)$ the inversion seems hard. If $m = pq$ for primes p, q then finding $\phi(m)$ is equivalent to factoring m.
Why is it easy for real numbers?

A polynomial algorithm G generating the keys from 1^n:

1. Find two large primes, p, q, of length n. For this, choose large numbers repeatedly and test them for primality.
2. Compute $m = pq$ and $\phi(m)$. Select a number e relatively prime to $\phi(m)$. For this, try numbers $< \phi(m)$ repeatedly and test for relative primality. It can be shown that that this process succeeds soon.
3. Compute the multiplicative inverse d of e modulo $\phi(m)$.
4. Output $(m, e), (m, d)$ as the public and private keys.

Our encryption works on numbers, not on strings. To encrypt strings, we have to break up the string into smaller segments and convert the segments into numbers first. Let w be such a number, we will view it as our message. The encryption function is

$$E(\langle m, e\rangle, w) = w^e \bmod m.$$

The inverting function is

$$D(\langle m, d\rangle, v) = v^d \bmod m.$$

Compare with the definition of trapdoor functions!

It is important to assume that the solution of the equation

$$x^b \equiv c \pmod{m}$$

is not easy even for, say, 1% of the numbers c. Otherwise, a randomized polynomial-time inversion algorithm would find x for all numbers c.