

Elements of the theory of computing: lectures

Peter Gács

Boston University

1 Introduction

1.1 The class structure

See the course homepage.

1.2 Description complexity

Showing that it is uncomputable. See the lecture note linked to the course homepage.

2 Preliminaries

2.1 TeX notation for email correspondence

“Control sequences” begin with \backslash .

$a \backslash le b$ for $a \leq b$, a_i for a_i , $a^{\wedge\{25\}}$ for a^{25} , $x \backslash in A$ for $x \in A$,
 $X \backslash cup Y$ for $X \cup Y$, $X \backslash cap Y$ for $X \cap Y$, $X \backslash setminus Y$ for
 $X \setminus Y$.

2.2 Notation for languages

Alphabet, string, length $|\cdot|$, binary alphabet.

Empty string ϵ .

Set Σ^* of all strings in alphabet Σ .

Lexicographical enumeration.

2.3 Elementary encodings, cardinality

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

Example. Let $\# \notin \Sigma$, then we can encode each element (u, v) of $\Sigma^* \times \Sigma^*$ as

$$\langle u, v \rangle = u\#v.$$

For example, $\langle 0110, 10 \rangle = 0110\#10$. ◇

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 $\langle a, b \rangle = \langle a \rangle \# \langle b \rangle$.

Triples, quadruples, are handled similarly. 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.

3 Turing machines

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.

3.1 Basic definitions

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.

Conventions:

$$M = (Q, \Sigma, \Gamma, \delta, q_{\text{start}}, q_{\text{accept}}, q_{\text{reject}}).$$

Q = set of states.

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

Γ = tape alphabet containing Σ and \sqcup .

$\delta()$: $\delta(q, a) = (q', b, d)$ where $d \in \{L, R\}$.

Read-write head. Configuration uqv . The meaning of

uqv yields $u'q'v'$.

3.2 Computing with a Turing machine

Computing a function $f : \Sigma^* \rightarrow \Sigma^*$: input and output conventions.

For machine M , $M(x)$ is the output on input x . We write $M(x) = \infty$ if M does not halt on input x .

$L(M)$ is the language of strings accepted by M . We say M recognizes $L(M)$.

Recognizing versus deciding. Turing-recognizable and Turing-decidable languages.

(In some other texts, what we call “recognizing” is called “accepting”.)

3.3 Examples of Turing machines

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

4 Variants of Turing machines

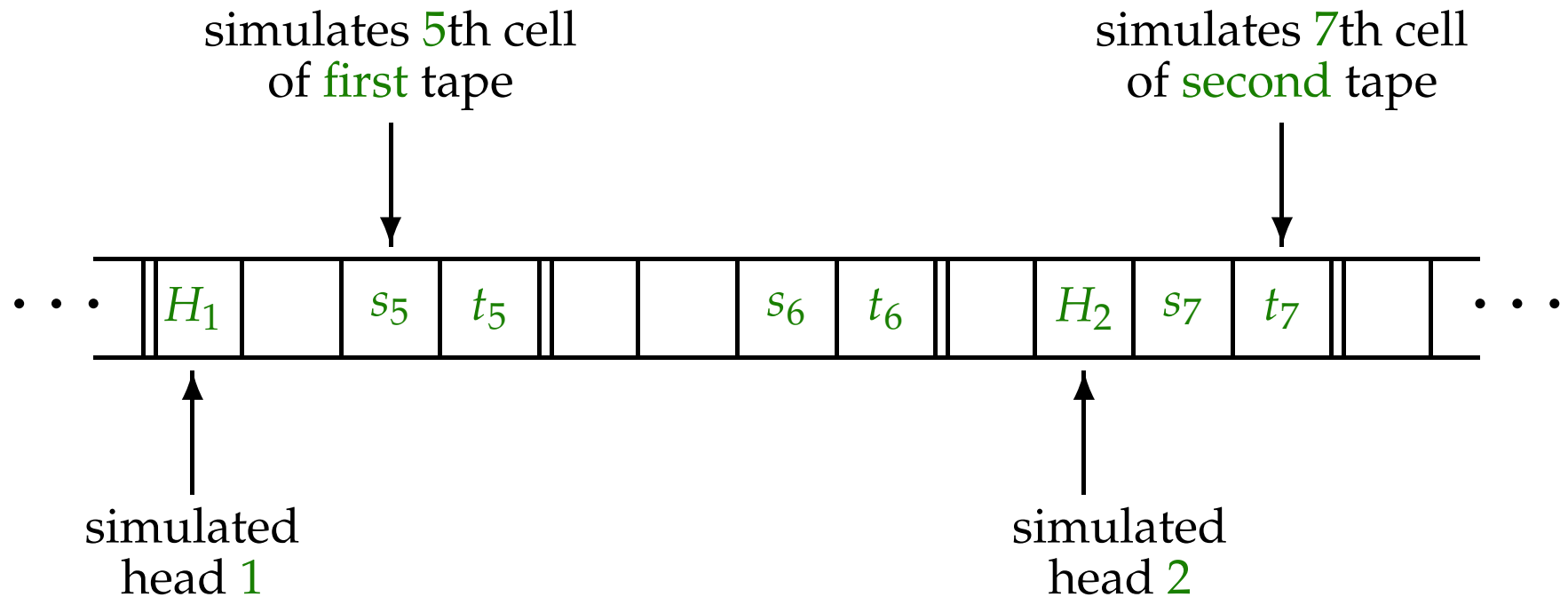
The notion of simulation: M_1 **simulates** M_2 if the input-output behavior of M_1 is the same as that of M_2 .

In practice: **representing** one “data structure” in another:

Configuration C of M_2 is represented by a configuration C' of M_1 .

“Programming” the update: Each step of M_2 will be simulated by *several steps* of M_1 . If the step of M_2 carries C to D then the corresponding steps of M_1 carry C' to D' .

4.1 Simulating 2 tapes by one tape



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 $O(s(x)t(x)) = O(t^2(x))$.

4.2 2-dimensional Turing machine

What is a “good” representation of a 2-dim tape on a 1-dim tape?
(address, content) pairs. Slowdown is similar to the 2-tape case:
 $t \mapsto t^2 \log t$.

4.3 Random access machine

(After Lewis-Papadimitriou. Not studied in detail this semester.)

Fixed number K of registers $R_j, j = 1, \dots, K$. Memory: one-way infinite tape: cell i contains natural number $T[i]$ of arbitrary size.

Program: a sequence of instructions, in the “program store”: a (potentially) infinite sequence of registers containing instructions. A program counter.

read	j	$R_0 = T[R_j]$	(this is random access)
write	j		
store	j	$R_j = R_0$	
load	j		
add	j	$R_0 += R_j$	
add	=c	$R_0 += c$	
sub	j	$R_0 = R_0 - R_j ^+$	
sub	=c		
half		$R_0 /= 2$	
jump	s		
jpos	s	if $R_0 > 0$ then jump s	
jzero	s		
halt			

No multiplication: that would make the machine too fast, and we can simulate it anyway.

Input-output conventions.

How to define **running time**?

Simulations between the RAM and Turing machines.

Simulating a Turing machine by RAM: transition table \rightarrow RAM program jumps.

Simulating a RAM by a Turing machine.

Representation on a TM: a separate tape for each register. A tape for the memory: list of (address, value) pairs. At most $t \mapsto t^2$ slowdown.

5 Universal Turing machines

5.1 What is a universal machine

Let Σ be some alphabet, called an **input/output** alphabet. Let U be a two-tape Turing machine whose tape alphabet contains Σ . 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 2-tape machine U is **universal** for machines in the set $\mathcal{T}_1(\Sigma)$ if for all machines $M \in \mathcal{T}_1(\Sigma)$ there is a string $p_M \in \Sigma^*$ such that for all inputs $x \in \Sigma^*$ we have

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

Theorem 1. *For every Σ there is a machine U universal for $\mathcal{T}_1(\Sigma)$. Also, $U(p_M, x)$ accepts iff $M(x)$ accepts.*

5.2 Constructing the universal machine

Encoding the states: elements of $q\{0,1\}^*$.

Encoding the tape symbols: elements of $a\{0,1\}^*$.

Assume $i = \lceil \log |Q| \rceil$, then Q will be encoded by $q\{0,1\}^i$, and so on.

Representation of tuples and strings: if $w = a_1 \dots a_n$ then

$$\langle w \rangle = \langle a_1, \dots, a_n \rangle = \langle a_1 \rangle \# \langle a_2 \rangle \# \dots \# \langle a_n \rangle.$$

Representation of M : $\langle M \rangle = E_1 \#\# E_2 \#\# \dots \#\# E_{|Q| \cdot |\Gamma|}$ where each E_i is an entry of the table of the transition function δ : $E_i = \langle q, a, q', a', d \rangle$ where $(q, a) \mapsto (q', a', d)$.

First, we will only construct a U_1 with

$$\langle M(w) \rangle = U_1(\langle M \rangle, \langle w \rangle).$$

Then, to construct U , we add to U_1 some preprocessing, encoding the input w into $\langle w \rangle$ and some postprocessing, decoding the output $\langle M(w) \rangle$ into $M(w)$.

To construct U_1 we first construct U_2 with 3 tapes: this will then be simulated on two tapes.

Tape 1: repr. the tape contents and head position of M .

Tape 2: $\langle M \rangle$.

Tape 3: repr. of the current state of M .

Simulation: in the obvious way.

Discussion 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.

5.3 A useful picture

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

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

6 The concept of an algorithm – Church's thesis

History: different formal definitions by Church (lambda calculus), Gödel (general 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).

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).

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.

7 Analyzing the running time of an algorithm

7.1 Measuring the running time of a program

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` and `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.

7.2 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 general 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 csb.



7.3 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.

7.4 Tricky examples

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

```
int min(vector<int> A) {
    int i;
    int n = A.size();
    int m = A[0];
    for (i = 0;
        i < n; i++)
        if (m > A[i]) m = A[i];
    return m;
}
```

1
1
1
1
The loop repeats n times
2 for each i
1 or 2 for each i
1

Total number of steps: $3 + n \cdot c$ where $3 \leq c \leq 4$. 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 we cannot determine it and it would not be too informative. (Different steps have different costs, too.)

For example, `min(A)` takes at most $5 + 2n \leq 7n$ 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:

```
(1) for (i = 0; i < n; i++)  
      B[i] = min(A) * (i + 1);  
(2) m = min(A);  
    for (i = 0; i < n; i++)  
      B[i] = m * (i + 1);
```

Compare the number of operations taken by parts (1) and (2) as a function of n .

The first one can only be said to be $O(n^2)$, since `min(A)` recomputes the minimum every time it is called, the second one is $O(n)$.

The big-O notation directs our attention to these big differences, by ignoring the small ones.

7.5 Worst-case analysis

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

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

7.6 Average case analysis

Example.

```
find_root(vector<int> A) { // returns -1 if there is no root.
    int n = A.size();
    for (i = 0; i < n, i++) {
        if (0 == A[i]) 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. ◇

8 Complexity classes

8.1 What kinds of problem?

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 .

8.2 Upper and lower bounds

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

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

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

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

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



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

8.3 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.

8.4 Why are we concentrating 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: $x \cdot x \cdots x$ (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.

8.5 Eliminating duplicates: an analysis

What is the running time?

```
1 void rm_dupl_1(vector<int> &x)
2 // Removes all duplicates from x, without changing the order.
3 {
4     if (2 > x.size()) return;
5     for (size_t i = 0; i < x.size() - 1; i++)
6         for (size_t j = i + 1; j < x.size(); j++)
7             if (x[i] == x[j]) {
8                 erase(x, j);
9                 j--;
10            }
11 }
```

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.*)

```
1 void rm_dupl_2(vector<int> &x){
2 // Removes all duplicates from x, without changing the order.
3   size_t n = x.size();
4   if (2 > n) return;
5   size_t current_end = 1;
6   for (size_t compared1 = 1; compared1 < n; compared1++){
7       bool found = false;
8       for (size_t compared2 = 0; compared2 < current_end;
9           compared2++){
10          if (x[compared2] == x[compared1]) {
11              found = true;
12              break; // from inner loop
13          }
14          if (found) continue; // outer loop
15          else {
16              if (compared1 > current_end)
17                  x[current_end] = x[compared1];
18              current_end++;
19          }
20      }
21      for (size_t i = n; i > current_end; i--)
22          x.pop_back();
23 }
```

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.

9 Asymptotic analysis

$O()$, $o()$, $\Omega()$, $\Theta()$. 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 $\log 3$.

9.1 Examples

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

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:

$$n^2 - 3 \log \log n \sim n^2,$$

$$\log n / n,$$

$$\log \log n,$$

$$n \log^2 n,$$

$$3 + 1/n \sim 1,$$

$$\sqrt{(5n)}/2^n,$$

$$(1.2)^{n-1} + \sqrt{n} + \log n \sim (1.2)^n.$$

Solution:

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

9.2 Sums: the art of simplification

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:

$$1 + 2^3 + 3^3 + \dots + n^3 = \Theta(n^4),$$
$$1/3 + 2/3^2 + 3/3^3 + 4/3^4 + \dots < \infty.$$



Example.

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

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

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

Lower bound:

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



9.3 Fast polynomial multiplication

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

$$\begin{aligned} f(x) &= a_0 + a_1x + \cdots + a_{m-1}x^{m-1} + x^m(a_m + \cdots + a_{2m-1}x^{m-1}) \\ &= f_0(x) + x^m f_1(x). \end{aligned}$$

Similarly for $g(x)$. So,

$$fg = f_0g_0 + x^m(f_0g_1 + f_1g_0) + x^{2m}f_1g_1.$$

In order to compute fg , we need to compute

$$f_0g_0, \quad f_0g_1 + f_1g_0, \quad f_1g_1.$$

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

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

which suggests that we need n^2 multiplications.

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

$$f_0g_1 + f_1g_0 = (f_0 + f_1)(g_0 + g_1) - f_0g_0 - f_1g_1. \quad (1)$$

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

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

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.)

9.4 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.

10 Polynomial time

10.1 Invariance with respect to machine model

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.

10.2 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”.

10.3 Shortest path

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.

10.4 Algorithms on integers

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 2. $\gcd(a, b) = \gcd(b, a \bmod b)$

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

10.5 Extended Euclidean algorithm

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

$$a' = x_1a + y_1b,$$

$$b' = x_2a + y_2b,$$

$$r' = a - qb' < b'$$

then

$$r' = (x_1 - ax_2)a + (y_1 - qy_2)b.$$

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.

10.6 Longest common subsequence

The `diff` program in Unix. Given sequences x, y of length n , it is possible to find the shortest 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>

You must have learned the method, **dynamic programming**, in your Algorithms class. A recursive solution to subproblems.

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

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)**.

10.7 Gaussian elimination

(Skip this 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**.

11 The class NP

11.1 Examples

HAMILTON CYCLE, TRAVELING SALESMAN PROBLEM.
EULER CYCLE.

11.2 Witness relation

Example. $R_1(x, y)$ is true if, x, y are integers and y is a proper divisor of x . ◇

Example. $R_2(G, C)$ is true if x 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. ◇

11.3 Polynomial-verifiable witness relations

The witness relation R must be

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

All this makes sense after integers, graphs, etc. are **encoded** into strings.

Relation $R \subset \Sigma^* \times \Sigma^*$, encoded as a **language**:

$$\langle R \rangle = \{ \langle x, y \rangle : R(x, y) \}$$

$$R(x, y) (= \text{true}) \Leftrightarrow (x, y) \in R \Leftrightarrow \langle x, y \rangle \in \langle R \rangle.$$

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

11.4 Definition of NP

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

$$L = \{ x \in \Sigma^* : \exists y \in \Sigma^* R(x, y) \}.$$

We associate **two problems** with a witness relation R :

- **decision problem**: given $x \in \Sigma^*$ **decide** whether $x \in L$.
- **search problem**: given $x \in \Sigma^*$ **find** y with $R(x, y)$.

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

Example.

$$R_1(x, y) \Leftrightarrow x \text{ is odd, } y \text{ is a proper divisor of } x.$$

$$R_2(x, \langle a, b \rangle) \Leftrightarrow x \text{ is odd, } x = a^2 - b^2, a > b + 1.$$



11.5 Optimization problems

MAXIMUM CLIQUE, MINIMUM NODE COVER, TRAVELING SALESMAN PROBLEM.

In general, a question of the sort:

given x , maximize $f(x, y)$

where $f(x, y)$ is polynomial-time computable.

Turning an optimization problem into a witness relation:

$$R(\langle x, k \rangle, y) \Leftrightarrow \exists y f(x, y) \geq k.$$

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

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

12 Nondeterministic computations

12.1 Recognition and enumeration

Recall: notion of language **recognized** by a Turing machine. Definition of a language **enumerated** by a Turing machine.

Theorem 3. *A language is Turing recognizable iff it is Turing enumerable.*

12.2 Nondeterminism

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

12.3 Rewrite rules and grammars

(You can safely skip this, not covered in cs332.)

Grammars also illustrate nondeterminism.

Fix an alphabet Σ .

Rewrite rule (production) $P : u \rightarrow v$ 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.

13 More examples of NP problems

13.1 Subset sum

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

$$a_1x_1 + \dots + a_nx_n = b.$$

Example: $\{4, 11, 16, 21, 27\}, 25$.

A dynamic programming algorithm (Theorem 5.5.7). Let S_i be the set of numbers of form $a_1x_1 + \dots + a_ix_i$. Then

$$S_i = S_{i-1} \cup (a_i + S_{i-1}).$$

The complexity of this algorithm can be bounded by

$$\left(\sum_i a_k\right) \cdot n$$

times the cost of the algebraic operations involved.

Is this polynomial?

Similar problem: PARTITION.

Example: 38, 17, 52, 61, 21, 88, 25. The subset {38, 52, 61} will do.

13.2 MAXCUT

Definition.

14 Satisfiability

14.1 Propositional logic

Logic formulas. Conjunctive and disjunctive normal form. *If this is new to you, review it from your discrete math book, or from the course cs210.*

Example:

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

Truth assignment

Definitions of \Rightarrow , \Leftrightarrow , \oplus (XOR). Properties:

- associativity, commutativity of \vee , \wedge and \oplus .
- distributivity: \wedge into \vee , \vee into \wedge , \wedge into \oplus .
- de Morgan rules, help moving the negation inside.

Conjunctive normal form. Literals. Clauses.

Theorem 5. *Expressing every Boolean function via a Boolean formula.*

14.2 Satisfiability

Satisfying assignment. Special case for conjunctive normal forms.

Tautologies.

Satisfiability problem general and for CNF: **SAT**.

Example. Matching in graphs, reduced to SAT.



SAT reduced to **3-SAT** via logic circuits.

15 Completeness

15.1 Reductions

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 \equiv INDEPENDENT ◇

Example. INDEPENDENT \equiv NODE COVER ◇

15.2 Hardness and completeness

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

Example. All **NP**-complete problems below. Also the halting problem.

Also: given a graph, does it have at least k matchings?



NP-complete problems.

We will see many examples.

15.3 The NP-completeness of SAT

Describe the **constraints** of the **space-time history** of a (deterministic or nondeterministic) Turing machine computation by a conjunctive normal form.

16 Other NP-complete problems

16.1 Independent sets

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

16.2 Hamiltonian path

Follow the reduction in the Sipser book. Are the separator points needed?

17 Undecidable problems

17.1 Recursive enumerability

Computable/decidable/recursive.

Weaker notion: semidecidable/recognizable/recursively enumerable.

A language is called **recursively enumerable** if it is recognized by some (deterministic) Turing machine.

Theorem 6. *The languages recognizable by non-deterministic Turing machines are just the recursively enumerable languages.*

Proof. Det. \Rightarrow nondet.: this direction is easy.

Nondet. recognized by $M \Rightarrow$ det.: breadth-first search over all possible computations of M . This is also called **dovetailing**. \square

17.2 Halting problem

Recall the infinite matrix containing $M(w)$, with rows indexed by $\langle M \rangle$, and columns indexed by inputs w . We introduced this matrix in connection with the universal Turing machines.

The **halting problem** is to decide whether the entry in position $(\langle M \rangle, w)$ is ∞ . Let us define the language

$$A_{\text{TM}} = \{ \langle M, w \rangle : M \text{ accepts } w \}.$$

By definition, it is a semidecidable (same as Turing recognizable, recursively enumerable) language.

Theorem 7. A_{TM} is not decidable.

Proof of Theorem 7. Suppose that A_{TM} is decidable. Then the language $K = \{ \langle M \rangle : M \text{ accepts } \langle M \rangle \}$ is also decidable, and so there is a Turing machine T that for all w , accepts w iff w is **not** in K . We designed T in such a way that on the diagonal

$$\{ (\langle M \rangle, \langle M \rangle) : M \text{ runs through all machines } \}$$

of our matrix (indexed with $(\langle M \rangle, w)$), it behaves always differently from the way M behaves on w : it accepts w exactly when M does not accept w . But the table also contains a row for $M = T$. This leads to a contradiction at position $(\langle T \rangle, \langle T \rangle)$. □

17.3 A Turing unrecognizable language

The language A_{TM} introduced above is Turing recognizable: indeed, the universal Turing machine can simulate M on w to accept exactly whenever M accepts w .

Let us show that the complement of A_{TM} is not Turing recognizable.

Theorem 8. *A language is decidable iff both it and its complement is Turing recognizable.*

Theorem 9. *The complement of A_{TM} is not Turing recognizable.*

17.4 Reductions

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 . More restricted definition, called “mapping 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.

Theorem 10. *The halting problem is undecidable.*

Proof: we reduce the acceptance problem A_{TM} to the halting problem.

Let $E_{\text{TM}} = \{ \langle M \rangle : L(M) = \emptyset \}$.

Theorem 11. E_{TM} is undecidable.

Proof. Reduce A_{TM} to E_{TM} . **Input** of A_{TM} : a pair $\langle M, w \rangle$. **Output**: decision about whether M accepts w .

Input of E_{TM} : a machine T . Output: decision about whether T accepts **anything**.

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

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

Then

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

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

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

Theorem 12. EQ_{TM} is undecidable.

We will solve this in class.

17.5 A simple undecidable problem

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

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

Theorem 13. *PCP is undecidable.*

Proof. MPCP: a distinguished first tile. Reducing A_{TM} to MPCP. Given M, w . (Assume M never tries to move left from the left end.)

Parts of the kit:

1. $\left[\frac{\#}{\#q_0w_1\dots w_n\#} \right]$
2. Head moving right: $(q, a) \mapsto (r, b, R)$. Tile $\left[\frac{qa}{br} \right]$.
3. Head moving left: $(q, a) \mapsto (r, b, L)$. All tiles $\left[\frac{cqa}{rcb} \right]$.
4. $\left[\frac{a}{a} \right]$ for all a in Γ .

Example. $\Gamma = 0, 1, 2, \sqcup,$

$$(q_0, 0) \rightarrow (q_7, 2, R),$$

$$(q_7, 1) \rightarrow (q_5, 0, R),$$

$$(q_5, 0) \rightarrow (q_9, 2, L).$$



5. $\left[\frac{\#}{\#} \right], \left[\frac{\#}{\sqcup\#} \right]$. This extends the tape if needed.
6. $\left[\frac{aq_{\text{accept}}}{q_{\text{accept}}} \right], \left[\frac{q_{\text{accept}}a}{q_{\text{accept}}} \right]$. This eats up the rest of the tape.
7. $\left[\frac{q_{\text{accept}}\#\#}{\#} \right]$ completes the match.

Reducing MPCP to PCP.

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

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

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


Convert an instance $\left\{ \left[\frac{t_1}{b_1} \right], \dots, \left[\frac{t_n}{b_n} \right] \right\}$ of MPCP into the following instance of PCP:


$$\left[\frac{*t_1}{*b_1*} \right],$$
$$\left[\frac{*t_1}{b_1*} \right], \dots, \left[\frac{*t_n}{b_n*} \right],$$
$$\left[\frac{*@}{@} \right].$$

18 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). 

18.1 Language

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 $\mathcal{L} = (L, \text{Neg})$.

Example. For logical language $\mathcal{L}_1 = (L_1, \text{Neg})$, 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: $\text{Neg}(l(a, b)) = l'(a, b)$, $\text{Neg}(l'(a, b)) = l(a, b)$. ◇

18.2 Formal system

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

$$V(P, T)$$

(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

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

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, \dots, 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 $a < b$.



18.3 Consistency and completeness

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 $a \geq b$. ◇

Completeness is not always desirable.

Example. The language of **group theory** consists of expressions formed from the following: variable symbols, an abstract binary operation $*$ called **multiplication**. Further we have the equation symbol $=$, parentheses, and logical symbols $\wedge, \vee, \neg, \exists, \forall$. The formal system is given by 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, spelling out all properties of groups. To this are added a few general axioms related to logic and equality, and some general deduction rules of logic. Such a theory is not meant to be complete. It should describe the group of permutations as well as the group of invertible matrices and the group of integers when $*$ is interpreted as addition. In this theory the sentence $\forall a \forall b (a * b = b * a)$ is undecidable. It is true for the addition of natural numbers, but false for the multiplication of permutations. Thus, an incomplete theory is *more general* than the theory obtained when we make it complete (if we can). Its theorems have wider validity. \diamond

Complete theories have a desirable algorithmic property.


Theorem 14. *If a formal system \mathbf{T} is complete then there is an algorithm that for each sentence S finds in \mathbf{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 \leq . Further we have the equation symbol $=$, parentheses, and logical symbols $\wedge, \vee, \neg, \exists, \forall$. The formal system is given by some axioms

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

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. 

18.4 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 **recursively enumerable, non-recursive** set of integers. An arithmetical theory \mathbf{T} is **minimally adequate** if there is a computable function that to each number n , assigns a sentence φ_n such that $n \in L$ iff $\varphi_n \in \mathbf{T}$.

A reasonable formal system of natural numbers should be minimally adequate: 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 15 (Gödel's first incompleteness theorem). *Every minimally adequate formal system is incomplete.*

Proof. If the formal system were complete then, according to Theorem 14 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.

18.5 More arithmetization, consistency

For any formal system \mathbf{F} , it is possible to encode the proof verification process into arithmetical formulas. If \mathbf{F} is “sufficiently strong”, then there is a formula $\Gamma_{\mathbf{F}}$ expressing the fact that \mathbf{F} is consistent.

With an appropriate definition of *sufficiently strong*, then the following theorem holds (we will not prove it):

Theorem 16 (Gödel’s second incompleteness theorem). *If the system \mathbf{F} is sufficiently strong and consistent then $\Gamma_{\mathbf{F}}$ is undecidable in it.*

Thus, the consistency of a sufficiently strong system \mathbf{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*.

19 Approximations

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 $\text{obj}(x, y)$ for $x, y \in \{0, 1\}^n$, the optimum is

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

where y runs over the possible .

For $1 \leq k < \infty$, an algorithm $A(x)$ is a **k -approximation** if


$$\text{obj}(x, A(x)) \geq M(x)/k.$$

Analogous concept for minimization.

19.1 Greedy algorithms

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 17. *If you cannot improve anymore with this algorithm then you are within a factor 2 of the optimum.*

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

19.2 Less greed is sometimes better

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

Approx_Vertex_Cover(G)

$C \leftarrow \emptyset$

$E' \leftarrow E[G]$

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 18. *Approx_Vertex_Cover* has a ratio bound of 2.

19.3 Approximation classes

(1) **Fully approximable**: for every ε , there is a $1 + \varepsilon$ -approximation.

Example: appropriate version of KNAPSACK

(2) **Partly approximable**: there is an constant lower bound $k_{\min} > 1$ on the achievable approximation ratio.

Example: MAXIMUM CUT, VERTEX COVER, MAX-SAT.

(3) **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.

20 Randomization

20.1 Two uses of randomness in complexity

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 (which generate their own randomness) are called **randomized**.

20.2 Randomized primality tests

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”.

- (1) If x is prime then $A(x, r) = \text{“yes”}$.
- (2) If x is composite then $A(x, r) = \text{“no”}$ with probability $\geq 1/2$. In this case, r serves as a **witness** of nonprimality.

After repeating the test k times, the probability of “no” for composite x is increased to $1 - (1/2)^k$.

The witness above is **not a factor** of x , and does not help in factorization.

20.3 Polynomial identity

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.

$$\det(A_1x_1 + \cdots + A_kx_k + A_{k+1})$$

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

Lemma 1. *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 2 (Schwartz). Let $p(x_1, \dots, x_m)$ be a nonzero polynomial, with each variable having degree at most d . If r_1, \dots, r_m are selected randomly from $\{1, \dots, f\}$ then the probability that $p(r_1, \dots, r_m) = 0$ is at most md/f .

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

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

□

20.4 Branching programs

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

```
1 L1:      x_1 ? goto L3 : goto L2
2 L2:      x_2 ? goto L6 : goto L4
3 L3:      x_3 ? goto L7 : goto L5
4 L4:      x_3 ? goto L7 : goto L6
5 L5:      x_2 ? goto L7 : goto L6
6 L6:      return 0
7 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 19. *The **equivalence problem** for branching programs is co-NP-complete.*

A branching program is **read-once** (ROBP) if each variable is read 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 20. 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 :

1 to the input node

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

20.5 Randomized complexity classes

Randomized Turing machine, two possible views.

1. Nondeterministic machine, with coin-toss choices
2. Deterministic machine, with extra tape containing random bits

RP and BPP.

The examples we have seen are for RP. I do not know of simple and natural examples in $BPP \setminus RP$ (maybe the approximate computation of some multidimensional volume).

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 machine M decide whether input x is in L , with error probability $q < 1/2$. Construct another machine M' that recognizes L with a **much smaller** probability $(4q(1 - q))^n$ where $n = |x|$. (Show that this is always less than 1!) So, the error probability can be made even **exponentially small**.

Machine M' , on input x of length n : repeats $2n$ times the calculations of $M(x)$, then takes the majority.

(There are $2n + 1$ possibilities for the number m of accepting computatons: $0, 1, \dots, 2n$. Accept if $m \geq n$.)

$\mathbf{P}[M'$ accepts when it should not] =

$$\mathbf{P}[m \geq n] = \sum_{i=n}^{2n} \mathbf{P}[m = i] = \sum_{i=n}^{2n} \binom{2n}{i} q^i (1 - q)^{2n-i}.$$

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

$$\mathbf{P}[m \geq n] \leq \sum_{i=0}^{2n} \binom{2n}{i} q^n (1 - q)^n = 2^{2n} q^n (1 - q)^n = (4q(1 - q))^n.$$

Similarly, of course,

$$\mathbf{P}[M' \text{ rejects when it should not}] \leq (4q(1 - q))^n.$$

21 Cryptography

Cryptography in history.

Example. Caesar cypher. ◇

Secret algorithm, or secret key. Example: DES (Data Encryption Standard).

Example. 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.

Cryptanalysis in WWII. The Enigma, the Magic, Turing etc.

Commercial needs in the present. Systematic cryptology based on complexity theory, started in mid 1970s.

Private-key versus **public-key** cryptosystems.

Notation: M is the message, C is the cyphertext.

21.1 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.

21.2 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.

21.3 One-way functions

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

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

$$\mathbf{P}[f(M(f(w))) = f(w)] \leq n^{-c}.$$

It is not known whether one-way functions exist. (If they do then $\mathbf{P} \neq \mathbf{NP}$.) Some functions seem 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.

21.4 Trap-door functions

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

- Length-preserving polynomial-time encoding function

$$E : (\Sigma^*)^2 \rightarrow \Sigma^*.$$

- Polynomial-time inverting function $D : (\Sigma^*)^2 \rightarrow \Sigma^*$.

- Auxiliary probabilistic polynomial time Turing machine G , produces (key, trapdoor) pair $\langle e, d \rangle$.

Requirements:

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

$$\mathbf{P}[E(e, v) = E(e, w)] \leq n^{-c}.$$

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

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

22 Some number theory

22.1 Modular arithmetic

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 21. *Given integers a, e, m , it takes only polynomial time to compute*

$$a^e \bmod m.$$

22.2 Division modulo m

Theorem 22. 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, $\mathbb{Z}_m^* = \{1, \dots, m - 1\}$. In this case, we can “divide” in \mathbb{Z}_m by any nonzero element, so \mathbb{Z}_m is a **field**.

22.3 Fermat's theorem

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

Theorem 23 (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, \dots, 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 24 (Euler).

- (a) *The number of integers in $\{1, \dots, m - 1\}$ relatively prime to m is $\phi(m)$.*
- (b) *If $\gcd(a, m) = 1$ then*

$$a^{\phi(m)} \equiv 1 \pmod{m}.$$

23 The RSA cryptosystem

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 $\phi(m)$.

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

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

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?

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

- Find two large primes, p, q , of length n . For this, choose large numbers repeatedly and test them for primality.
- 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.
- Compute the multiplicative inverse d of e modulo $\phi(m)$.
- Output $\langle\langle m, e \rangle, \langle m, d \rangle\rangle$ 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 .