Advanced Algorithms CS3172, 02/03

Time: Monday and Friday, 10-11.

www: Do check the website regularly.

Lecture Course comes in 2 parts:
Part I (Dix) Introduction to Complexity Classes,
Part II (Rydeheard) Specific Algorithms.

Organisation:

Part I (Dix): 8 lectures, one week free (3/7 March), one week to discuss the homework (10/14 March).
Part II (Rydeheard): 8 lectures, one week free (5/9 May), one week to discuss the homework (12/16 May).
Exam: Friday, 30. May.

Overview

1. Turing Machines
2. Complexity Classes
3. Hierarchies, Complete Problems

1 Turing Machines

1.1 The very definition

1.2 Computable languages

1.3 Modifications of TM's

1.4 Undecidability

1.1 The very definition

Computability: Consider functions over the natural numbers $\mathbb{N}$.

Which of these should we call computable?

Complexity: In order to measure the complexity of an algorithm, we need a machine model to check it against!

Which model do we choose?

Robustness: Model should not depend on small modifications.

It should be robust against such small changes.

One machine to rule them all!

This ring of the rings is the Turing machine, named after Alan Mathison Turing.

One can think of a Turing machine as a device, consisting of

Tape: A one-way infinite tape consisting of single cells (storage tape).

Head: A head which always scans exactly one cell. The head can move on the tape to the right or left and it can also print something on the cell that is currently scanned (thereby overwriting the contents of the cell).

Finite control: A mechanism that can be described by a finite table. It tells the head

1. to write a certain symbol on the current cell,
2. to move right or left and
3. to enter another state,

all this depending on the current state and the symbol scanned.

Definition 1.1 (Turing Machine, Version 1)

A deterministic Turing machine (DTM) is a tuple $\langle Q, \Sigma, \Gamma, \#, \delta, q_0 \rangle$:

- Q is a finite set (set of states).
- Γ (set of tape symbols) is a finite set with $\# \in \Gamma$,
- Σ (set of input symbols) is also a finite set with $\Sigma \subsetneq \Gamma$,
- $\#$ is a distinguished symbol (the blank) with $\# \notin \Sigma$.
- q_0 is a distinguished element of Q (start state): $q_0 \in Q$.
- δ is a function (the next move function) from $Q \times \Gamma$ to $Q \times \Gamma \times \{L, R\}$ which needs not to be defined on some inputs.

A DTM takes as input a word $w \in \Sigma^*$ and either halts after a number of moves (in which case the output is the word left on the tape) or it does not (in which case there is no output). The number of states of a DTM is $\|Q\| - 1$.

To be precise, we define the following:

ID: An **instantaneous description** of a DTM is denoted by the string $\alpha_1 q \alpha_2$ where $\alpha_1, \alpha_2 \in \Gamma^*$, $q \in Q$.

- Informally it means that the tape contains the string $\alpha_1\alpha_2$ (the first symbol of α_1 represents the first cell of the tape, and all cells beyond the last symbol of α_2 are blanks), the head is scanning the leftmost symbol of α_2 (if $\alpha_2 = \varepsilon$ then it is scanning a blank #) and the DTM is in state q.
- We also use a distinguished ID denoted by **stop:**$\alpha_1 q \alpha_2$: this means that $\alpha_1 q \alpha_2$ is the current description and the DTM stops.

An ID is like a snapshot of the current status of the DTM. It gives a complete description.

Move: For a DTM $\mathcal{M}$ we define a relation $\vdash_{\mathcal{M}}$ (a direct move) between ID's as follows. Firstly, stop:$\alpha_1 q \alpha_2 \vdash_{\mathcal{M}}$ stop:$\alpha_1 q \alpha_2$ for all $\alpha_1, \alpha_2 \in \Gamma^*$, $q \in Q$. Secondly, $X_1 \cdots X_{i-1} q X_i \cdots X_n \vdash_{\mathcal{M}}$

$$\vdash_{\mathcal{M}} \begin{cases} X_1 \cdots X_{i-1} Y p X_{i+1} \cdots X_n, & \text{if } i > 1,\ \delta(q, X_i) = (p, Y, R); \\ X_1 \cdots X_{i-2} p X_{i-1} Y X_{i+1} \cdots X_n, & \text{if } i > 1,\ \delta(q, X_i) = (p, Y, L); \\ \text{stop:} q X_1 \cdots X_n, & \text{if } i = 1,\ \delta(q, X_i) = (p, Y, L); \\ X_1 \cdots X_{i-2} p X_{i-1} Y, & \text{if } i - 1 = n > 0,\ \delta(q, \#) = (p, Y, L); \\ \text{stop:} X_1 \cdots X_{i-1} q X_i \cdots X_n, & \text{if } \delta(q, X_i) \text{ is not defined ;} \end{cases}$$

We denote by $\vdash^*_{\mathcal{M}}$ the transitive closure of $\vdash_{\mathcal{M}}$. Thus for two ID's the relation

$ID_1 \vdash^*_{\mathcal{M}} ID_2$ means that ID_2 can be reached from ID_1 in finitely many steps.

Comp: Let a partial function $f: \Sigma^* \longrightarrow \Sigma^*;\ w \mapsto f(w)$ be given

(f needs not be defined on all inputs).

We say that a DTM $\mathcal{M}$ computes f,when the following holds:

$$f(w) = w' \text{ if and only if } \#wq_0 \vdash^*_{\mathcal{M}} \text{stop:}\#w'q,$$

where $q \in Q$.

Note that a DTM might not stop on certain inputs.

Definition 1.2 ((Partial) recursive functions over Σ)

A function computed by a DTM is called partial recursive . If such a function is defined on all inputs, it is called recursive .

The last definition was introduced by Alan Turing in his ground breaking paper (Turing 1936). **It is one of the most ingenious definitions in mathematics and logic of all times.**

Church's Thesis: The intuitive notion of an algorithm is correctly reflected by a Turing machine: The input is a word written on the tape and the output also is a word obtained after the machine stops.

The intuitive notion of a computable function over $\mathbb{N}$ is that of a function computed by a Turing Machine with $\Sigma = \{1\}$. Input and output are written in unary.

Visual Turing: `http://www.cheransoft.com/vturing/`

A nice tool to develop, play and understand Turing machines.

Note the differences between our definition and visual Turing:

1. variable assignments in visual Turing are represented in our definition by introducing additional states (one for each tape symbol: when an assignment α is made, we enter a particular state so that we know to remember α and can print it later on if needed).
2. In our model we must both write on the tape and move the head (in the same step).
3. In visual Turing subprograms can be easily incorporated, whereas in our definition we have to rename the states involved to avoid confusion (in particular the start state).

I should reach this point at the end of the first lecture. Finish with playing around with the visual Turing program.

Here is the definition of visual Turing's Left# wrt. $\Gamma = \{\#, a, b\}$
(Go to the first blank on the left of the current position of the head).

Let $Q := \{q_0, q_1\}$, $\Sigma := \{a, b\}$, and δ be defined as follows:

state	#	a	b
q_0	$\langle q_1, \#, L\rangle$	$\langle q_1, a, L\rangle$	$\langle q_1, b, L\rangle$
q_1	$-$	$\langle q_1, a, L\rangle$	$\langle q_1, b, L\rangle$

- It is easy to design DTMs for addition, subtraction, multiplication, copying of strings etc. It might be tedious at first.
- DTMs can also simulate any type of subroutine, including recursive procedures and parameter passing mechanisms.
- I doubt whether you can come up with any number theoretic function that is not computable by a DTM. **Anything goes with a DTM.**

Homework 1 (Number of DTMs)

a) Given numbers n, m, exactly how many Turing machines with up to n states and m tape symbols are there?

b) How many Turing machines are there at all (no restriction on the number of states)?

c) Do the above answers change if we allow the set Σ to be countably infinite?

d) How many instantaneous descriptions are there for a DTM working on a part of the tape of length n?

e) How many Turing machines or equivalents have actually been built from 1936-1986? I mean machines that can compute all partial recursive functions in the sense of Definition 1.2. Just make a guess.

Example 1.1 (Printing n on the tape: $\mathrm{Print_n}$)
For each $n \in \mathbb{N}$, let us design a machine that prints exactly n 1's on the tape, starting with the empty tape standing on the second blank (from the left). We try to do it with as few states as possible. We use the machine copy. For given n, we first write $\lfloor \frac{n}{2} \rfloor$ many 1's on the tape. This can be done with $\lfloor \frac{n}{2} \rfloor$ many states. We then copy this number (this can be done with a constant number of states, namely 7). Then, we replace the separating # with a 1 and, depending on whether n is odd or even, we leave it as is or we replace the last 1 by a #. To do this we need another 2 states.
In total, we can write n 1s with only $\lfloor \frac{n}{2} \rfloor + 9$ states.

The machine Copy, using seven states:

state	q_0	q_1	q_2	q_3	q_4	q_5	q_6	q_7
#	$\langle q_1,\#,L\rangle$	$\langle q_2,\#,R\rangle$	$\langle q_7,\#,R\rangle$	$\langle q_4,\#,R\rangle$	$\langle q_5,1,L\rangle$	$\langle q_6,\#,L\rangle$	$\langle q_2,1,R\rangle$	$-$
1	$-$	$\langle q_1,1,L\rangle$	$\langle q_3,\#,R\rangle$	$\langle q_3,1,R\rangle$	$\langle q_4,1,R\rangle$	$\langle q_5,1,L\rangle$	$\langle q_6,1,L\rangle$	$\langle q_7,1,R\rangle$

1.2 Acceptable languages

In the last section we viewed a DTM as a computing device : given an input, it computes an output. Often, however, it is conceptually simpler to view a DTM as a device getting an input and returning *Yes* or *No*: this is called an **acceptor**.

A DTM under this viewpoint gets an input, runs and either stops in an accepting state, stops in an non-accepting state or runs forever. We therefore have to add a set of accepting states.

Definition 1.3 (Turing Machine, Version 2 (Acceptor))
A deterministic Turing machine (DTM) M is a seven-tuple

$$\langle Q, \Sigma, \Gamma, \#, \delta, q_0, F \rangle, \text{ where}$$

- Q is a finite set (set of states).
- Γ (set of tape symbols) is a finite set with $\# \in \Gamma$,
- Σ (set of input symbols) is also a finite set with $\Sigma \subsetneq \Gamma$,
- # is a distinguished symbol (the blank) with $\# \notin \Sigma$.
- q_0 is a distinguished element of Q (start state): $q_0 \in Q$.

- δ is a function (the next move function) from $Q \times \Gamma$ to $Q \times \Gamma \times \{L, R\}$ which needs not to be defined on some inputs.
- $F \subseteq Q$, F is the set of final states .

A DTM takes as input a word $w \in \Sigma^*$ and either (1) halts in an accepting state, or (2) halts in a non-accepting state, or (3) does not halt at all.

The only modification we have to make for the definition of a move (see Slide 10) is the following:

If $q \in F$, then $X_1 \cdots X_{i-1} q X_i \cdots X_n \vdash_{\mathcal{M}}$ stop:$X_1 \cdots X_{i-1} q X_i \cdots X_n$.

Definition 1.4 (Language accepted by a DTM)
The language $\mathcal{L} \subseteq \Sigma^*$ accepted by a DTM M, denoted by $\mathcal{L}(\mathrm{M})$, is the set of words in Σ^* that cause M to enter a final state (when started on the string "#w" with the head at the first blank to the right of w).

Note that for non-accepted words, a DTM needs not halt!

Why are we talking about accepting languages? Because it is a nice unifying framework and makes the introduction of complexity notions so much easier!

Example 1.2 (Instances of Definition 1.4)
The following examples will be discussed and illustrated in greater depth later.

Numeric: $\Sigma_{\text{num}} := \{1\}$, $\mathcal{L}_{\text{primes}} := \{p : \ p \text{ is a prime }\}$. $\overline{\mathcal{L}_{\text{primes}}}$: the set of non-primes. Here we use unary notation.

One could also code integers in n-ary notation. We need an input alphabet $\Sigma := \{0, 1, \ldots, n-1\}$ and code an integer according to its value to the base n. For example, in 2-ary notation (binary), the integer 5 would be written as "101", whereas in unary notation it would be "11111".

Writing numbers in n-ary notation ($n > 1$) saves storage tape: instead of m cells we only need $\log_n m$ many.

However, there is only a gain when switching from unary to binary notation (from m to $\lg m$). Switching further to $\log_n m$ saves only a (linear) constant, but nothing more in the limit.

Formulae: $\Sigma_{\text{sat}} := \{\wedge, \vee, \neg, (,), x, 0, 1\}$. This is enough to define boolean expressions w over variables $\{x_1, x_2, \ldots\}$. A variable x_i is denoted by the string "x" followed by the binary notation of i written as 0s and 1's.

As usual, when the variables of such a boolean expression are set to t (true) or f (false), the whole expression evaluates to either true or false.

The satisfiability problem is:

Given an expression w, is it satisfiable?

Suppose we are given a boolean expression (in the usual form) with n symbols. What is the length of its coded version in Σ_{sat}?

There can only be $\lceil \frac{n}{2} \rceil$ different variables and each requires no more than $1 + \lceil \lg n \rceil$ symbols: this gives a bound of $n\lceil \lg n \rceil$. All our forthcoming complexity results will not depend on whether we are using n or $n \lg n$ as the length of the input (this will be explained later).

We then define:

$$\mathcal{L}_{\text{sat}} := \{w \in \Sigma^*_{\text{sat}} : \quad \text{there is an assignment of } x_i \text{ such that formula } w \text{ evaluates to true}\}$$

Graphs: Let $\mathcal{G} = \langle V, E \rangle$ be a directed graph (V denotes the vertices and $E \subseteq V \times V$ the edges of the graph).

Is there a path leading from vertex 1 to vertex n?

We choose $\Sigma_{\text{graph}} := \{v, e, 0, 1, (,)\}$. v_i is represented as the string "v" followed by the binary notation of i written as 0s and 1's. An edge $e_{i,j}$ is represented as the string "(string1#string2)" where string1 stands for the binary representation of i and string2 stands for the binary representation of j. We define:

$$\mathcal{L}_{\text{reach}} := \{w \in \Sigma^*_{\text{graph}} : \text{there is a path in graph } w \text{ leading from the first vertex } v_1 \text{ to the last one } v_n\}$$

Integer Linear Programming: Given an $m \times n$ matrix of integers $\mathbf{A}$ and a column vector $\mathbf{b}$.

Does there exist a column vector $\mathbf{x}$ such that $\mathbf{Ax} \geq \mathbf{b}$?

Representation is straightforward: words of the language are the entries of $\mathbf{A}$ and $\mathbf{b}$ written in binary.

$$\mathcal{L}_{\mathsf{ILP}} := \{w \in \Sigma^*_{\mathsf{ILP}} : \quad w \text{ represents an ILP problem } \langle \mathbf{A}, \mathbf{b} \rangle \text{ such that there is } \mathbf{x} \text{ with } \mathbf{Ax} \geq \mathbf{b}.\}$$

This is the point reached after the second lecture.

Is the language $\{ww^R : w \in \Sigma^*\}$ acceptable? Is the language $\emptyset$ acceptable?

What language does the following DTM accept:

state	#	a	b
q_0	$-$	$-$	$-$

What does the following DTM do:

state	#	1
q_0	$\langle q_1, 1, R\rangle$	$-$
q_1	$\langle q_2, 1, R\rangle$	$-$
q_2	$-$	$\langle q_1, a, L\rangle$

How many DTM's with 0 states are there?

Σ^*: Let Σ be finite or countable infinite. Then Σ^* is countably infinite. Each finite string can be represented as a natural number by using the prime factor decomposition.

Computable functions: There are as many computable functions as natural numbers.

$f : \mathbb{N} \to \{0,1\}$: There are uncountably many such functions. Given $f_1, \ldots f_n, \ldots$ we construct

$$C : \mathbb{N} \to \mathbb{N}; n \mapsto \begin{cases} 1, & \text{if } f_n(n) = 0; \\ 0, & \text{else.} \end{cases}$$

C is certainly different from all f_i (why?).

Subsets of $\mathbb{N}$: There are uncountably many subsets of $\mathbb{N}$.

Therefore there must be non-computable functions.

1.3 Modifications of DTM's

Various Modifications of DTMs have been defined. They are all equivalent when it comes to the class of computable functions or acceptable languages they lead to.

1.3.1 Modifying the tape

Two-way infinite tape: we can allow the tape to be two-way infinite. This adds even more storage and the head cannot *fall off* when too far left.

Multitape DTM: We can allow multiple tapes (all of them two-way infinite). One can be distinguished as the input tape. In each step, all the heads on all tapes have to be moved (independently).

Theorem 1.1 (Two-way infinite = one-way infinite)

DTMs with a two-way infinite tape or with multiple such tapes are equivalent to ordinary DTMs: the set of acceptable languages is the same.

Proof:

two-way infinite: Obviously, a two-way infinite tape DTM can simulate an ordinary DTM: it just has to mark the left of its tape (the second blank to the left of the input) and can then simulate the programme of the DTM.

We have to show how to simulate a two-way infinite DTM on a DTM:

use all even numbered cells of the one-way tape to store the contents of the left hand side of the two-way tape, and all odd numbered cells for the right hand side.

We need some more states to remember on which tape we are and the simulation has to do two moves (to stay on the right tape) rather than just one. For each state q of the two-way infinite machine, we need two states $(q, even), (q, odd)$.

Multiple tapes: We have to show how to simulate a k-tape DTM on an ordinary one. As in the case of the two-way infinite tape, we split the tape into a number of tracks, namely $2k$ many.

Each tape of the k-tape DTM corresponds to 2 tracks: one just indicates the position of the head, and the other track corresponds to the contents of the tape.

It is quite obvious how this new DTM has to act to simulate the k-tape DTM (note that k is fixed: this information is stored in the states of the DTM).

Consider the language $\{ww^R : w \in \Sigma^*\}$ where w^R is the reverse of w.

1-tape DTM: How can this language be decided by a 1-tape DTM?
How many moves is the head doing?

2-tape DTM: How can this language be decided by a 2-tape DTM?
How many moves are the heads doing?

This technique shows that for m moves of the k-tape DTM, the 1-tape DTM needs a number of moves that is **quadratic in** m (actually, $6m^2$).

I should have reached this point after the third lecture.

34-1

1.3.2 Indeterminism

Our Definition 1.3 is deterministic in the sense that the move function δ is always uniquely determined (or undefined).

A nondeterministic version allows the machine to go into a finite number of successor states.

Definition 1.5 (Turing Machine, Nondeterministic Acceptor)
A nondeterministic Turing machine (NDTM) M is a seven-tuple $\langle Q, \Sigma, \Gamma, \#, \delta, q_0, F \rangle$ as in Definition 1.3, where δ is modified

- δ is a function (the next move function) from $Q \times \Sigma$ to $2^{(Q \times \Sigma \times \{L,R\})}$ which needs not to be defined on some inputs.

A NDTM takes as input a word $w \in \Sigma^*$ and either (1) halts in an accepting state, or (2) halts in a non-accepting state, or (3) does not halt at all.

The language $\mathcal{L} \subseteq \Sigma^*$ accepted by a NDTM M, denoted by $\mathcal{L}(\mathrm{M})$, is the set of words in Σ^* such that there is a series of moves that cause M to enter a final accepting state (when placed on the empty tape with the head at the first blank to the right).

Why are NDTM so important? The precise answer will be given in Section 3. But let us consider Example 1.2, the instance with the satisfiability problem.

Given a formula, to find whether it is satisfiable or not, we have to guess the right assignment of truth values for the variables.

We can model this very easily with a NDTM: whenever the NDTM reaches a variable, it makes two possible moves: one where the variable is set to true, the other where it is set to false.

Obviously, the formula is satisfiable if and only if there is a sequence of moves leading to an assignment making the whole formula true.

We want to construct a machine which checks whether the input wrt. $\Sigma = \{a,b\}$ is of the form ww, with $w \in \Sigma^*$.

The idea is to

1. find the middle position of the input, and
2. then successively replace the appropriate symbols (in the left word by #, in the right word by a new symbol c), and
3. checking whether they match.

We end up with a tape which consists of only cs (in which case the input had this form) or not (in which case the input was not of this form).

Here is the deterministic part of the program describing δ:

state	q_1	q_2	$\mathbf{q_3}$	$\mathbf{q_4}$	$\mathbf{q_5}$	$\mathbf{q_6}$	q_7	q_8
#	$\langle q_2,\#,R\rangle$		$\langle q_5,\#,R\rangle$	$\langle q_6,\#,R\rangle$				
a	$\langle q_2,a,R\rangle$	$\langle q_3,\mathbf{c},L\rangle$	$\langle q_3,a,L\rangle$	$\langle q_4,a,L\rangle$	$\langle q_7,\#,R\rangle$		$\langle q_7,a,R\rangle$	$\langle q_1,a,L\rangle$
b	$\langle q_2,b,R\rangle$	$\langle q_4,\mathbf{c},L\rangle$	$\langle q_3,b,L\rangle$	$\langle q_4,b,L\rangle$		$\langle q_7,\#,R\rangle$	$\langle q_7,b,R\rangle$	$\langle q_1,b,L\rangle$
c			$\langle q_3,c,L\rangle$	$\langle q_4,c,L\rangle$			$\langle q_8,c,R\rangle$	$\langle q_8,c,R\rangle$

How can you make this into a NDTM that correctly solves our problem?

Theorem 1.2 (DTM and NDTM are equivalent)
A language accepted by a NDTM is also accepted by some DTM.

Proof: Suppose we are given a NDTM. In order to construct an equivalent DTM, we have to make sure that all possible moves are simulated in a systematic manner. Note that there is a maximal number r of possible next-moves for our NDTM (determined by δ).

The simulation will be done on a 3-tape DTM. The first tape is for the input. The second tape systematically generates finite sequences of numbers from $1,\ldots,r$. The sequence $1,5,6,3,6$ means: in the simulation of the NDTM, when coming to the first choicepoint use possibility 1, for the second use 5, etc. for the fifth choicepoint use 6.

The third tape is for the actual simulation, where the choices are dictated by tape 2. ∎

1.3.3 Restrictions

Can we restrict the number of states or the number of symbols on the tape alphabet?

Definition 1.6 (Off-Line (N)DTM)
An off-line (N)DTM is a k-tape (N)DTM where one tape is the input tape. We assume the input is enclosed between two blanks #. The head of this tape can only move on the input and the enclosing blanks (to determine that the end of the input is reached), but not write or go farther to the left or right of the input.

Theorem 1.3 (Minimal number of tapes/Minimal alphabet)
Any acceptable language can be accepted by an off-line (N)DTM with one storage tape the alphabet of which is $\{\#, 1\}$.

Any acceptable language can be accepted by an off-line (N)DTM with one storage tape and 3 states.

1.4 Undecidability

As mentioned on Slide 15, it is not easy to think of a function that is not computable, without sorting to very artificial examples. Here is one dating back to (Rado 1962; Lin and Rado 1965).

Define $BB : \mathbb{N} \to \mathbb{N}$ as follows

$n \mapsto f(n) :=$ the maximal number of 1s a DTM (as in Definition 1.1 defined over $\Gamma = \{\#, 1\}$) with maximal n states can print on an empty tape and halt.

It is easy to compute BB for $1, 2$: what results do you get?

The classical busy beaver is for two-way infinite tapes and final states (Definition 1.3). The only known values are (in our terminology): $BB(1) = 1, BB(2) = 4, BB(3) = 13$. Already $BB(5)$ is not known. And $BB(6)$ is greater than $1.29 * 10^{865}$.

Theorem 1.4 (BB is not computable)
The busy beaver increases too much to be computable: there is no DTM computing BB.

Proof: As one can always add 1 state to print one more 1:

- BB is strictly monotonically increasing.

Suppose there is a DTM BB computing BB. Let it have n_0 states.

We build a series of other machines which write $BB(m)$ 1s on the empty tape and use less than m states. This is a contradiction to our observation above.

Consider the compound machines $\mathrm{Comp_m} := \mathrm{BB} \circ \mathrm{Print_m}$, i.e. firstly m 1s are printed on the empty tape (see Example 1.1 on page 17) and then, secondly, BB runs to compute $BB(m)$. These machines have $n_0 + \lfloor \frac{m}{2} \rfloor + 9$ states. Now choose m_0 such that

$$m_0 > n_0 + \lfloor \frac{m_0}{2} \rfloor + 9.$$

Then for all $m \geq m_0$, Comp_m writes $BB(m)$ 1s on the tape and works with strictly less than m states. ∎

This is the point reached after the fourth lecture.

2 Complexity Classes

2.1 Time/Space Complexity

2.2 Speed up

2.3 Relations between Time/Space

2.1 Time/Space Complexity

Definition 2.1 (NTIME$(T(n))$, DTIME$(T(n))$)
We consider as base model a multitape TM M with k two-way infinite tapes, one of which contains the input. If for every word of length n as input, M makes at most $T(n)$ moves, then M is called $T(n)$ time bounded.

The language accepted by M is said to be of time complexity $T(n)$ (actually we mean $\max(n+1, \lceil T(n) \rceil)$).

- **DTIME**$(T(n))$ is the class of languages accepted by $T(n)$ time bounded deterministic DTMs.
- **NTIME**$(T(n))$ is the class of languages accepted by $T(n)$ time bounded nondeterministic NDTMs.

Definition 2.2 (NSPACE (S(n)), DSPACE$(S(n))$)
We consider as base model an offline TM M with k one-way infinite tapes and a special input tape. If for every word of length n as input, M scans at most $S(n)$ cells on the storage tapes, then M is called $S(n)$ space bounded.

The language accepted by M is said to be of space complexity $S(n)$ (actually we mean $\max(1, \lceil S(n) \rceil)$).

- **DSPACE**$(S(n))$ is the class of languages accepted by $S(n)$ space bounded deterministic DTMs.
- **NSPACE** $(S(n))$ is the class of languages accepted by $S(n)$ space bounded nondeterministic NDTMs.

Why offline TM?

(tape bounds of less than linear growth)

To which time/space complexity class belongs

$$\mathcal{L}_{\text{mirror}} := \{wcw^R : \ w \in (0+1)^*\},$$

i.e. the set of words that can be mirrored on the middle letter c?

Time: **DTIME**$(n+1)$. Just copy the input to the right of c in reverse order on another tape. Once a c is found, just compare the remaining part (the w) with the copy of w on the tape.

Space: **DSPACE**$(\lg n)$. The machine just described gives us a bound of **DSPACE**(n). But we can do better. We use two tapes as binary counters. Firstly the input is checked for the occurrence of just one c and an equal number of symbols to the left and right of c. This needs only constant space, resp. it can be done with a number of states (and thus needs no space at all). Secondly we check the right and left part symbol by symbol: to do this we just have to keep in mind the two positions to be checked (for equality) (and they are coded on the two tapes).

A few words on terminology.

computable**:** A function is computable, if, by definition, there is a DTM computing it (given the input n, the DTM computes $f(n)$). The function can be partial or not. We also say the function is **partial recursive** (see Slide 11).

accepted**:** A language is accepted (or recognised), if there is a DTM accepting it (given an input w, the DTM stops in an accepting state if and only if w is in the language).

decided**:** We say a DTM decides a language if there is a DTM that accepts it and it always terminates. The language is then called decidable.

decidable**:** We say a problem is **decidable** if there is a DTM that decides it. A problem can always be put in the form "Is $w \in \mathcal{L}$" for an appropriate language $\mathcal{L}$.

Recall that we also call a function **recursive** if it is partial recursive and total.

If a language is decidable, then its complement is as well. This is not true for acceptance.

Time: Is any language in **DTIME**$(f(n))$ decided by a DTM?

Space: Is any language in **DSPACE**$(f(n))$ decided by a DTM?

Time/Space: Same questions about **NTIME**$(,)$ **NSPACE**.

Homework 2 (Acceptability/Decidability)
Comment on the following statements.

1. The traditional algorithm for checking whether a number is prime, is decidable.
2. Any finite language is accepted by a DTM.
3. Any infinite language is accepted by a NDTM.
4. Any finite language is decided by a DTM.
5. There are algorithms that are undecidable.
6. If $\mathcal{L}$ is decidable, then its complement is decidable as well.

7. There is at least one DTM that is decidable.
8. There is at least one NDTM that is undecidable.
9. The following function is decidable:

$$f : \mathbb{N} \longrightarrow \mathbb{N}, n \mapsto \begin{cases} 1, & \text{if cricket is a sport for stupid people;} \\ 0, & \text{otherwise.} \end{cases}$$

2.2 Speed up

The aim of this section is to illustrate that only the functional rate of growth of a function matters in a complexity class: constant factors have to be ignored.

Theorem 2.1 (Tape compression)
For any $c > 0$ and space function $S(n)$:

$$\textbf{DSPACE}(S(n)) = \textbf{DSPACE}(cS(n))$$
$$\textbf{NSPACE}(S(n)) = \textbf{NSPACE}(cS(n))$$

Note that one direction is trivial. The proof for the other is by representing a fixed number r ($> \frac{2}{c}$) of adjacent tape cells by a new symbol. The states of the new machine keep track which of the many cells represented is actually scanned during simulation.

Theorem 2.2 (Time speed up)

For any $c > 0$ and time function $T(n)$ with $\inf_{n\to\infty} \frac{T(n)}{n} = \infty$:

$$\mathbf{DTIME}(T(n)) = \mathbf{DTIME}(cT(n))$$
$$\mathbf{NTIME}(T(n)) = \mathbf{NTIME}(cT(n))$$

Again one direction is trivial. The proof for the other is also by representing a fixed number r ($> \frac{16}{c}$) of adjacent tape cells by a new symbol (the states of the new machine keep track which of the many cells represented is actually scanned (when simulating the old machine)).

When simulating the old machine, the new one only needs to make 8 moves instead of r: 4 to check the immediate neighbours and another 4 to modify them.

What happens if we reduce the number of tapes? Let us consider again $\mathcal{L}_{\text{mirror}}$ from Slide 50. The linear complexity does no more hold if there is only one tape available. However, the following holds.

Theorem 2.3 (Reduction of tapes (1))

- If $\mathcal{L} \in \mathbf{DTIME}(T(n))$, then $\mathcal{L}$ is accepted in time $T^2(n)$ by a one-tape DTM.
- If $\mathcal{L} \in \mathbf{NTIME}(T(n))$, then $\mathcal{L}$ is accepted in time $T^2(n)$ by a one-tape NDTM.

The proof is simple. Remember that we need $6T^2(n)$ steps to simulate the k-tape DTM using a 1-tape DTM (see slide 34). Now we speed it up by $\frac{1}{\sqrt{6}}$.

The last theorem also holds for space bounded functions:

Theorem 2.4 (Reduction of tapes (2))

- If $\mathcal{L} \in \mathbf{DSPACE}(S(n))$, then $\mathcal{L}$ is accepted in space $S(n)$ by a one-tape DTM.
- If $\mathcal{L} \in \mathbf{NSPACE}(S(n))$, then $\mathcal{L}$ is accepted in space $S(n)$ by a one-tape NDTM.

This proof is as simple as the last one. Note that in simulating a k-tape TM with a 1-tape TM we need the same number of storage cells. So we do not even need to speed up to get our result.

I should have reached this point after the fifth lecture.

59-1

2.3 Relations between Time/Space

Theorem 2.5 (Time versus Space)

- $\mathbf{DTIME}(f(n)) \subseteq \mathbf{DSPACE}(f(n))$.
- If $f(n) \geq \lg n$, and $\mathcal{L} \in \mathbf{DSPACE}(f(n))$, then there is a $c > 0$ s.t. $\mathcal{L} \in \mathbf{DTIME}(c^{f(n)})$.
- If $\mathcal{L} \in \mathbf{NTIME}(f(n))$, then there is a $c > 0$ s.t. $\mathcal{L} \in \mathbf{DTIME}(c^{f(n)})$.

Proof:

- Obvious.
- Suppose we have s states and t tape symbols. By using at most $f(n)$ cells, the number of different IDs on an input of length n is bounded by $s(n+2)(f(n)+1)t^{f(n)}$ (we assume in view of Theorem 2.4 that we are dealing with an offline DTM with just one storage tape). Because of $f(n) \geq \lg n$, there is a c such that for $n \geq 1$: $c^{f(n)} \geq s(n+2)(f(n)+1)t^{f(n)}$. We can construct a 3-tape DTM: one tape is used to count up to $c^{f(n)}$, the other two to simulate the old machine. When no accepting state is reached until the maximal count, it will never accept (the old machine actually loops): we then simply terminate in a non accepting state. If an accepting state is reached, the new machine accepts as well.

- Similar to the last case, but now we have to take into account the number k of tapes as well (and that it is a regular k-tape NDTM). Number of IDs is bounded by ... We construct a multitape DTM to simulate the old NDTM. Our machine first constructs a list L of all accessible IDs (from the initial input): This can be done in time bounded by $\text{length}^2(L)$ (why?) and we have $\text{length}(L) \leq \ldots$ and therefore $\leq c^{f(n)}$. It then checks whether any of the IDs leads to an accepting state or not ...

In the following we want to state some more relations between complexity classes. Unfortunately they do not hold for all time functions $T(n)$ or space functions $S(n)$, but for almost all that do occur naturally. We therefore define the following space of functions.

Definition 2.3 (Well-behaved functions)
We consider the vector space of functions from $\mathbb{N}$ into $\mathbb{N}$ containing $\log_a n, n^k, 2^n, n!$ and closed under multiplication, exponentiation and composition. We call such functions well-behaved.

Theorem 2.6 (Det. versus Non-Det. Space)
Let $S(n)$ be well-behaved. Then:

$$\mathbf{NSPACE}(S(n)) \subseteq \mathbf{DSPACE}(S^2(n)).$$

Theorem 2.7 (Time/Space Hierarchies)
Let $S_1(n), S_2(n)$ and $T_1(n), T_2(n)$ be well-behaved. We assume further that $S_1(n) < S_2(n)$ and $T_1(n) < T_2(n)$ for all $n > n_0$ for a $n_0 \in \mathbb{N}$.

1. If $\inf_{n\to\infty} \frac{S_1(n)}{S_2(n)} = 0$ then $\mathbf{DSPACE}(S_1(n)) \subsetneq \mathbf{DSPACE}(S_2(n))$.
2. If $\inf_{n\to\infty} \frac{T_1(n)\lg T_1(n)}{T_2(n)} = 0$ then $\mathbf{DTIME}(T_1(n)) \subsetneq \mathbf{DTIME}(T_2(n))$.

The last theorem implies the following hierarchies:

$$\mathbf{DSPACE}(n) \subsetneqq \mathbf{DSPACE}(n^2) \subsetneqq \ldots \subsetneqq \mathbf{DSPACE}(n^r) \subsetneqq \ldots$$

and

$$\mathbf{DTIME}(n) \subsetneqq \mathbf{DTIME}(n^2) \subsetneqq \ldots \subsetneqq \mathbf{DTIME}(n^r) \subsetneqq \ldots$$

as well as

$$\mathbf{DSPACE}(\log n) \subsetneqq \mathbf{DSPACE}(\log^2 n) \subsetneqq \ldots \subsetneqq \mathbf{DSPACE}(\log^r n) \subsetneqq \ldots$$

and

$$\mathbf{DTIME}(\log n) \subsetneqq \mathbf{DTIME}(\log^2 n) \subsetneqq \ldots \subsetneqq \mathbf{DTIME}(\log^r n) \subsetneqq \ldots$$

What about similar results for **nondet.** space and time?

Theorem 2.8 (Nondeterministic Hierarchies)
Let $S_1(n), S_2(n)$ and $f(n)$ be well-behaved. We assume further that $S_2(n) \geq n$ and $f(n) \geq n$ for all $n \in \mathbb{N}$.

1. $\mathbf{NSPACE}(S_1(n)) \subseteq \mathbf{NSPACE}(S_2(n))$ implies $\mathbf{NSPACE}(S_1(f(n))) \subseteq \mathbf{NSPACE}(S_2(f(n)))$.
2. $\mathbf{NTIME}(S_1(n)) \subseteq \mathbf{NTIME}(S_2(n))$ implies $\mathbf{NTIME}(S_1(f(n))) \subseteq \mathbf{NTIME}(S_2(f(n)))$.

This theorem is applied as follows. Suppose $\mathbf{NSPACE}(n^4) \subseteq \mathbf{NSPACE}(n^3)$. We then apply the theorem for n^3, n^4 and n^5 separately and get: $\mathbf{NSPACE}(n^{20}) \subseteq \mathbf{NSPACE}(n^9)$. By Theorem 2.6 we know that $\mathbf{NSPACE}(n^9) \subseteq \mathbf{DSPACE}(n^{18})$ and by Theorem 2.7 $\mathbf{DSPACE}(n^{18}) \subsetneq \mathbf{DSPACE}(n^{20})$, which is a contradiction.

The last theorem implies the following hierarchies:

$$\mathbf{NSPACE}(n) \subsetneqq \mathbf{NSPACE}(n^2) \subsetneqq \ldots \subsetneqq \mathbf{NSPACE}(n^r) \subsetneqq \ldots$$

and

$$\mathbf{NTIME}(n) \subsetneqq \mathbf{NTIME}(n^2) \subsetneqq \ldots \subsetneqq \mathbf{NTIME}(n^r) \subsetneqq \ldots$$

as well as

$$\mathbf{NSPACE}(\log n) \subsetneqq \mathbf{NSPACE}(\log^2 n) \subsetneqq \ldots \subsetneqq \mathbf{NSPACE}(\log^r n) \subsetneqq \ldots$$

and

$$\mathbf{NTIME}(\log n) \subsetneqq \mathbf{NTIME}(\log^2 n) \subsetneqq \ldots \subsetneqq \mathbf{NTIME}(\log^r n) \subsetneqq \ldots$$

Homework 3 (Simple Relations)

a) Consider the problem of testing whether a given natural number is prime. Use unary representation. Is there a DTM solving this problem in polynomial time? I.e. is this problem in $\mathbf{DTIME}(n^r)$ for a $r \in \mathbb{N}$? Note that you do not have to actually construct a DTM, it suffices if you can argue convincingly.

b) Complete the third part of the proof of Theorem 2.5.

c) Discuss the two notions of decidability versus acceptability of a language wrt. NDTM's and DTM's. What happens if we take time bounds into consideration? If a language is acceptable in time $T(n)$, it is also decidable in $T(n)$?

d) What, if any, is the relationship between the following pairs of complexity classes?

1. **DSPACE**(n^2) and **DSPACE**$(f(n))$ where $f(n) := n$ for odd n and $f(n) := n^3$ for even n.
2. **DTIME**(134^n) and **DTIME**$((\ln n)^n)$.
3. **DTIME**(2^n) and **DTIME**(3^n).
4. **NSPACE** (2^n) and **DSPACE**(5^n).
5. **DSPACE**(n) and **DTIME**$(\lceil \lg n \rceil^n)$.

This is what I did until the end of the sixth lecture.

69-1

3 Hierarchies, P/NP

3.1 The Structure of PSPACE

3.2 Completeness, Hardness

3.3 Examples

3.1 The structure of PSPACE

We define the most important complexity classes and look into their structure.

Definition 3.1 (P, NP, PSPACE, NSPACE)
We define the following complexity classes:

$$\begin{aligned}
\mathbf{P} &:= \bigcup_{i\geqslant 1} \mathbf{DTIME}(n^i) \\
\mathbf{NP} &:= \bigcup_{i\geqslant 1} \mathbf{NTIME}(n^i) \\
\mathbf{PSPACE} &:= \bigcup_{i\geqslant 1} \mathbf{DSPACE}(n^i) \\
\mathbf{NSPACE} &:= \bigcup_{i\geqslant 1} \mathbf{NSPACE}(n^i)
\end{aligned}$$

The intuition is as follows:

- Problems in **P** are efficiently solvable,
  whereas those in **NP** require exponential time.
- **PSPACE** is a huge class, way above **P** and **NP**.
- **DSPACE**$(\log n)$ is a small class within **P** and very small in **PSPACE**.

What are the precise relations between the complexity classes introduced in Definition 3.1?

Using Theorem 2.6 it is obvious that (remember the hierarchies introduced on Slide 65)

$$\begin{array}{rcl} \textbf{PSPACE} & = & \textbf{NSPACE} \\ \bigcup_{i\geqslant 1}\textbf{NSPACE}(\log^i n) & = & \bigcup_{i\geqslant 1}\textbf{DSPACE}(\log^i n) \end{array}$$

This gives us (using Theorem 2.7)

$$\textbf{DSPACE}(\log n) \subseteq \textbf{P} \subseteq \textbf{NP} \subseteq \textbf{PSPACE}$$

where at least one containment is proper (but it is unknown which).

- How do we show that a particular problem is in a certain class?

  Reduce it to a known one!

  But we have to have one to start with! Yes, and this is the most difficult step. A suitable problem is SAT introduced in Example 1.2.

- Can we find problems in a class that are the most difficult ones in that class?

  There are several ways to define a most difficult problem. They depend on which notion of reducibility we use.

Based on a particular reducibility notion, the answer to our question is Yes: such problems are called **complete problems** for the given class under the chosen reducibility notion.

Definition 3.2 (polynomial-time-, log-space reducibility)
Let $\mathcal{L}_1, \mathcal{L}_2$ be languages.

poly-time: We say that $\mathcal{L}_2$ is poly-time reducible to $\mathcal{L}_1$ if there is a polynomial-time bounded DTM that produces for each input w an output $f(w)$ such that

$$w \in \mathcal{L}_2 \text{ if and only if } f(w) \in \mathcal{L}_1$$

log-space: We say that $\mathcal{L}_2$ is log-space reducible to $\mathcal{L}_1$ if there is a $\log n$ space bounded offline DTM that always halts and produces for each input w an output $f(w)$ such that

$$w \in \mathcal{L}_2 \text{ if and only if } f(w) \in \mathcal{L}_1$$

The output tape is write-only and the head never moves left.
Space on the output tape is not counted.

The notions just introduced are important because of the following

Lemma 3.1 (Properties of poly-time and log-space reductions)

1. Let $\mathcal{L}_2$ be poly-time reducible to $\mathcal{L}_1$. Then

$\mathcal{L}_2$ is in **NP**	if	$\mathcal{L}_1$ is in **NP**
$\mathcal{L}_2$ is in **P**	if	$\mathcal{L}_1$ is in **P**

2. Let $\mathcal{L}_2$ be log-space reducible to $\mathcal{L}_1$. Then

$\mathcal{L}_2$ is in **P**	if	$\mathcal{L}_1$ is in **P**
$\mathcal{L}_2$ is in **DSPACE**$(\log^k n)$	if	$\mathcal{L}_1$ is in **DSPACE**$(\log^k n)$
$\mathcal{L}_2$ is in **NSPACE** $(\log^k n)$	if	$\mathcal{L}_1$ is in **NSPACE** $(\log^k n)$

3. The composition of two log-space (resp. poly-time) reductions is itself a log-space (resp. poly-time) reduction.

4. Log-space reducibility implies poly-time reducibility.

The next slide shows a log-space reduction of $\mathcal{L}_{\text{sat}}$ to $\mathcal{L}_{\text{ILP}}$.

I am here after the seventh lecture.

78-1

Recall the integer linear programming problem $\mathcal{L}_{\text{ILP}}$ from Slide 28. How is this problem related to $\mathcal{L}_{\text{sat}}$?

Suppose we are given a formula $\phi : \psi_1 \wedge \cdots \wedge \psi_l$ where each ψ_i is a disjunction of literals (over variables $x_1, \ldots, x_n$).

We construct an ILP problem $\langle \mathbf{A}, \mathbf{b} \rangle$ as follows:

Matrix A: $2n$ columns of $\mathbf{A}$ correspond to $x_1, \overline{x_1}, \ldots, x_n, \overline{x_n}$. The first n rows each contain exactly two consecutive 1's (starting from position i at row i). The next n rows each contain exactly two consecutive -1's (starting from position i at row $2i$). The next l rows contain 1' at exactly the positions determined by the clauses ψ_i. Finally, we have n rows containing only one 1 (at position x_i) and another n rows containing only one 1 (at position $\overline{x_i}$).

Vector b: The first n entries are 1, the next n entries are -1, the next l (number of clauses) are 1. Finally, we have $2n$ entries which are 0.

$$\phi \in \mathcal{L}_{\text{sat}} \text{ if and only if } \langle \mathbf{A}, \mathbf{b} \rangle \in \mathcal{L}_{\text{ILP}}$$

Why? Just split off the matrix multiplication into its many inequalities and find out what they really mean.

Theorem 3.1 (Characterisation of NP)
A language $\mathcal{L}$ is in **NP** if and only if there is a language $\mathcal{L}'$ in **P** and there is $k \geq 0$ such that for all $w \in \Sigma$

$$w \in \mathcal{L} \text{ if and only if there is } c : \langle w, c \rangle \in \mathcal{L}' \text{ and } |c| < |w|^k.$$

c is called a witness (or certificate) of w in $\mathcal{L}$. A DTM accepting the language $\mathcal{L}'$ is called a verifier of $\mathcal{L}$.

A decision problem is in **NP** if and only if every yes-instance has a succinct certificate (i.e. its length is polynomial in the length of the input) which can be verified in polynomial time.

E.g. checking "nonprimeness", graphs being hamiltonian, satisfiability.

3.2 Completeness, Hardness

Definition 3.3 (Completeness, Hardness)
A language $\mathcal{L}$ is called **NP complete** if $\mathcal{L} \in \mathbf{NP}$ and every language $\mathcal{L}' \in \mathbf{NP}$ is log-space reducible to $\mathcal{L}$.

A language $\mathcal{L}$ is called **NP hard** if every language $\mathcal{L}' \in \mathbf{NP}$ is log-space reducible to $\mathcal{L}$.

A language $\mathcal{L}$ is called **PSPACE complete** if $\mathcal{L} \in \mathbf{PSPACE}$ and every language $\mathcal{L}' \in \mathbf{PSPACE}$ is poly-time reducible to $\mathcal{L}$.

A language $\mathcal{L}$ is called **NSPACE hard** if every language $\mathcal{L}' \in \mathbf{PSPACE}$ is poly-time reducible to $\mathcal{L}$.

It is worth noting the following:

- If a **NP** hard problem is shown to be in **P**, then **P** =**NP**.
- If a **PSPACE** hard problem is shown to be in **P**, then **P** =**PSPACE**.
- If **P** $\neq$ **NP**, then no **NP** complete problem is solvable in polynomial time.

Complexity classes defined on deterministic Turing machines are closed under complementation: If a language $\mathcal{L}$ belongs to it, then its complement belongs to it as well (just run the old machine and swap the answers).

Does that hold for NP as well?

Nobody knows!!

The last complexity class we introduce is **co-NP**.

Definition 3.4 (co-NP)
co- **NP** is the class of languages $\mathcal{L}$ whose complements $\overline{\mathcal{L}}$ $(= \Sigma^* \setminus \mathcal{L})$ are in **NP**.

The following are **unknown** to science:

1. **P** $\neq$ **NP**.
2. **NP** $\neq$ **co-NP**.
3. **P** $\neq$ **PSPACE**.
4. **NP** $\neq$ **PSPACE**.

3.3 Some reductions

SAT, 3-CNF are NP complete

Definition 3.5 (SAT, k-CNF, k-DNF)
We have already introduced the satisfiability problem in Example 1.2 on Slide 25. A literal l_i is a variable x_i or its negation $\neg x_i$. A clause is a disjunction of literals. Let us define the following:

DNF: A formula is in disjunctive normal form (DNF) if it is of the form $(l_{11} \wedge \ldots \wedge l_{1n_1}) \vee \ldots \vee (l_{m1} \wedge \ldots \wedge l_{mn_m})$.

CNF: A formula is in conjunctive normal form (CNF) if it is of the form $(l_{11} \vee \ldots \vee l_{1n_1}) \wedge \ldots \wedge (l_{m1} \vee \ldots \vee l_{mn_m})$.

***k*-DNF:** A formula is in *k*-DNF if it is in DNF and every disjunct contains exactly *k* literals.

***k*-CNF:** A formula is in *k*-CNF if it is in CNF and every conjunct contains exactly *k* literals.

We also use the notation k-SAT for k-CNF: each clause contains at most k literals (they can of course vary from clause to clause, so that the whole formula can contain an arbitrary number of literals).

DNF: What is the time complexity of the satisfiability problem for formulae in DNF?

2**-CNF:** What is the time complexity for the satisfiability problem for 2-CNF?

Reducibility: How can we reduce a satisfiability problem in CNF to one into DNF and vice versa? What does this imply about the poly-time or log-space reducibility of SAT wrt to DNF and SAT wrt. CNF to each other?

Theorem 3.2 (NP complete Problems)
The following are **NP** complete problems: $\mathcal{L}_{\text{sat}}$, $\mathcal{L}_{3\text{-sat}}$, integer linear programming, the problem whether a graph contains a hamiltonian cycle.

Proof: not trivial! ∎

3.4 Graph Reachability, Vertex Cover

Theorem 3.3 (Completeness of $\mathcal{L}_{\text{reach}}$)
$\mathcal{L}_{\text{reach}}$ (graph reachability) is log-space complete for **NSPACE**$(\log n)$.

Theorem 3.4 (Completeness of Vertex Cover)
Vertex Cover is **NP** complete.

Homework 4 (Reducibilities, P/NP)

1. Suppose there is a function $f : \mathbb{N} \to \mathbb{N}$ mapping integers of length k onto integers of length k such that

   (a) $f \in \mathbf{P}$,

   (b) $f^{-1} \notin \mathbf{P}$.

   Show that

$$\{\langle x,y\rangle : \ f^{-1}(x) < y\} \in (\mathbf{NP} \cap \mathbf{co\text{-}\ NP}) \setminus \mathbf{P}$$

2. Suppose we have a DTM M that decides a language $\mathcal{L}$. If $x \in \mathcal{L}$, M says yes in polynomial time. If $x \notin \mathcal{L}$, M says no in exponential time. Is $\mathcal{L} \in \mathbf{P}$?

3. Classify the following problems to t (true), f (false), or u (unknown to science, i.e. depending on the P/NP problem).
   - SAT $\in \mathbf{P}$.
   - If a problem in **NP** can be solved in polynomial time, then $\mathbf{P} = \mathbf{NP}$.
   - If a problem in **NP** can be shown not to be solvable in polynomial time, then $\mathbf{P} \neq \mathbf{NP}$.
   - SAT can be polynomially reduced to 1-CNF.
   - All problems in **P** can be polynomially reduced to each other.
   - Some problems in **NP** can not be polynomially reduced to each other.

References

Hopcroft, J. and J. Ullman (1979). *Introduction to Automata Theory, Languages, and Computation*. Reading, MA: Addison Wesely.

Lin, S. and T. Rado (1965). Computer studies of turing machine problems. *Journal of the ACM 12*(2), 196–212.

Papadimitriou, C. (1994). *Computational Complexity*. Addison-Wesley.

Rado, T. (1962). On non-computable functions. *The Bell System Technical Journal XLI*(3), 877–884.

Turing, A. M. (1936). On Computable Numbers with an Application to the Entscheidungsproblem. *Proc. London Mathematical Soc., series 2 42*, 230–265. corrections ibid., 43:544–546.

References

Hopcroft, J. and J. Ullman (1979). *Introduction to Automata Theory, Languages, and Computation*. Reading, MA: Addison Wesely.

Lin, S. and T. Rado (1965). Computer studies of turing machine problems. *Journal of the ACM 12*(2), 196–212.

Papadimitriou, C. (1994). *Computational Complexity*. Addison-Wesley.

Rado, T. (1962). On non-computable functions. *The Bell System Technical Journal XLI*(3), 877–884.

Turing, A. M. (1936). On Computable Numbers with an Application to the Entscheidungsproblem. *Proc. London Mathematical Soc., series 2 42*, 230–265. corrections ibid., 43:544–546.