Computability and Complexity

Instructor: László Babai
Date: February 25, 2021

These notes offer sketchy outlines of the basic concepts of (a) computability, developed in the 1930s, including a sketch of the proof of the undecidability of the HALTING problem, and (b) of complexity theory, developed in the early 1970s, including the concepts of Cook- and Karp-reducibility and $\mathcal{NP}$-completeness.

1 Strings, total and partial functions, languages

Definition 1.1. Let Σ be a finite set to which we refer as the **alphabet**. Σ^* denotes the set of all words over Σ, i. e., all finite strings of "letters" from the alphabet, including the empty string, usually denoted Λ. The **length** of a string X is the number of letters in it, so for instance $|\Lambda| = 0$ and if $x = 0010$ then $|x| = 4$.

Examples 1.2. Our favorite alphabets are $\Sigma_1 = \{0, 1\}$ (the Boolean alphabet), an actual alphabet like $\Sigma_2 = \{A, B, C\}$, all symbols used in Boolean formulas: $\Sigma_3 = \{x, 0, 1, [,], (,), \wedge, \vee, \neg\}$. So Σ_1^* consists of all $(0, 1)$-strings, like Λ, 0, 1, 0010, etc. Σ_3^* includes Boolean formulas like $(x[01] \vee x[02]) \wedge (\neg x[01] \vee x[11])$, and also meaningless strings like $0[\vee 110[(xx \wedge \vee$.

We shall use the questionmark as a special symbol, not in the alphabet, to mean "undefined."

Let Σ_1, Σ_2 be finite alphabets. We shall consider functions $f : \Sigma_1^* \to \Sigma_2^*$. We shall refer to such functions as **total functions**, in distinction from functions $f : \Sigma_1^* \to \Sigma_2^* \cup \{?\}$, to which we refer as **partial functions**. The domain of the partial function f is defined as $f^{-1}(\Sigma_2^*)$, and f is "undefined" on the rest of Σ_1^*.

Definition 1.3. A **language** over the alphabet Σ is a set $L \subseteq \Sigma^*$. The **complement** of this language is $\overline{L} = \Sigma^* \ L$. (We assume Σ is prespecified.) If $\mathcal{C}$ is a class of languages then we define $co\mathcal{C} = \{\overline{L} \mid L \in \mathcal{C}\}$ (the set of complements of the languages in $\mathcal{C}$).

Here "class" is just a fancy term for "set." We usually use this term when we talk about a set of sets. A language is a set of strings, so a set of languages is a set of sets; therefore we prefer to talk about a *class* of languages.

Exercise 1.4. If $\mathcal{C}$ is a class of languages then $co(co\mathcal{C}) = \mathcal{C}$.

Exercise 1.5. Let $\mathcal{C}$ and $\mathcal{D}$ be classes of languages over the alphabet Σ. Then $co\mathcal{C} \cap co\mathcal{D} = co(\mathcal{C} \cap \mathcal{D})$. In particular, if $\mathcal{C} \subseteq \mathcal{D}$ then $co\mathcal{C} \subseteq co\mathcal{D}$.

Definition 1.6. Let $L \subseteq \Sigma^*$ be a language. The **characteristic function** $\chi_L : \Sigma^* \to \{0, 1\}$ of the language L is defined as follows: for $x \in \Sigma^*$ we set

$$\chi_L(x) = \begin{cases} 1 & \text{if } x \in L \\ 0 & \text{if } x \notin L \end{cases} \tag{1}$$

2 Graphs, Boolean circuits

We can encode graphs over a finite alphabet, for instance, by naming the vertices by integers in binary, and concatenating the rows of the adjacency list by using appropriate separators. We can also get an encoding over the Boolean alphabet by concatenating the rows of the adjacency matrix. The specific encoding does not matter as long as it is "reasonable." Similarly, we can encode Boolean circuits. One of the criteria to be "reasonable" is that we can easily recongnize which strings encode graphs or Boolean formulas, and which strings do not.

Let $\mathsf{3COL}$ denote the set of 3-colorable graphs (encoded as strings), and SAT the set of satisfiable Boolean formulas. These languages belong to the class $\mathcal{NP}$ (below); therefore, their complements belong to the language class $co\mathcal{NP}$. Let $\mathsf{non3COL}$ denote the set of graphs that are not 3-colorable, and let $\mathsf{nonGraph}$ denote the set of strings that do not encode graphs. We observe that the complement of $\mathsf{3COL}$ is $\mathsf{non3COL} \sqcup \mathsf{nonGraph}$ (where $\sqcup$ denotes disjoint union). Similarly, the complement of SAT is the set $\mathsf{nonSAT} \sqcup \mathsf{nonCircuit}$ where nonSAT denotes the set of non-satisfiable Boolean circuits and $\mathsf{nonCircuit}$ denotes the set of strings that do not encode Boolean circuits.

3 Computable and computably enumerable languages. Halting

By a "program," in this note we mean a program in your favorite programming language, perhaps Python, C, or, for the theoretically minded, Turing Machines (TMs). (The "program" of a TM is its transition table.) A programming language is **universal** if it can simulate Turing machines (TMs). Since Python (or C) can simulate TMs, for a programming language to be universal, it suffices if it can simulate Python (or, equivalently, C).

Let P be a program and X an input to that program. Both of these objects are strings. If we run P on X then either P stops after a finite number of steps, or it never stops. In the former case we say (P, X) is a *halting computation*; it produces an output, which we denote by $P(X)$. If (P, X) is a non-halting computation (never stops), we write $P(X) = ?$.

Definition 3.1. Let $\mathsf{HALTING}$ denote the set of those programs (in the given programming language) that, when run on an empty input, halt.

This is perhaps the most important language in computer science. While this definition depends on the arbitrary choice of programming language (e. g., Python), the theory does not depend on the particular choice of a universal programming language (including TM transition tables, which appeared in the original definition).

Definition 3.2. Let P be a program that takes strings in Σ_1^* as input, and, if it halts on input $X \in \Sigma_1^*$, produces an output $P(X) \in \Sigma_2^*$. This means P computes a partial function $\psi_P : \Sigma_1^* \to \Sigma_2^* \cup \{?\}$. A partial function $h : \Sigma_1^* \to \Sigma_2^* \cup \{?\}$ is called a **computable (partial) function** if there exists a program P such that $h = g_P$.

The traditional term for a computable function is a **recursive (partial) function**, introduced by Kurt Gödel (1906–1978), the greatest logician of all time.

Definition 3.3. Let $L \subseteq \Sigma^*$ be a language. We say that L is a **computable language** if its characteristic function is computable, i. e., there exists a program P such that $\chi_L = \psi_P$. Note that χ_L is a total function and therefore P must halt on every input. The set of computable languages is denoted $\mathcal{R}$. The traditional term for a computable language is a **recursive set** (Gödel), explaining the notation.

Exercise 3.4. The notion of computable languages does not depend on the choice of universal programming language in our definitions. (Prove that it is the same for Python as for C.)

Exercise 3.5. Prove: $\mathcal{R} = co\mathcal{R}$. In other words, you need to prove that the complement of every computable language is computable.

Definition 3.6. A **decision problem** is a problem that has a YES/NO answer. The typical decision problem we need to deal with is the **membership problem** in a given language $L \subseteq \Sigma^*$: given a string $X \in \Sigma^*$, decide whether $X \in L$.

Example 3.7. Deciding whether a graph is 3-colorable is the membership problem in $\mathsf{3COL}$.

Definition 3.8. We say that the membership problem in the language L is **decidable** if $L \in \mathcal{R}$, and **undecidable** otherwise.

Exercise 3.9. Prove: $\mathsf{3COL} \in \mathcal{R}$. In other words, show that 3-colorability of a graph is decidable.

Definition 3.10. A language $L \subseteq \Sigma^*$ is **computably enumerable** if there exists a computable function of which L is the domain. In other words, L is computably enumerable exactly if there exists a program that takes inputs from Σ^* and halts on input $X \in \Sigma^*$ if an only if $X \in L$. The class of computably enumerable languages is denoted $\mathcal{RE}$. The traditional term is **recursively enumerable sets**.

Exercise 3.11. Prove: a language $L \subseteq \Sigma^*$ is computably enumerable if and only if either $L = \emptyset$ or L is the range of a computable total function.

Exercise 3.12. Prove: $\mathcal{R} \subseteq \mathcal{RE}$.

Exercise 3.13. Infer from the previous exercise and another exercise earlier (which one?) that $\mathcal{R} \subseteq \mathcal{RE} \cap co\mathcal{RE}$.

Exercise 3.14. Prove: $\mathcal{R} = \mathcal{RE} \cap co\mathcal{RE}$.

4 HALTING is undecidable

Exercise 4.1. Prove: $\mathsf{HALTING} \in \mathcal{RE}$.

How difficult is the HALTING problem?

Probably the most famous problem of mathematics, "Fermat's Last Theorem (FLT)," was first stated by Pierre de Fermat around 1637, and not proved until three and a half centuries later by Andrew Wiles (1995)[1].

FLT states that if x, y, z, n are positive integers and $n \geq 3$ then $x^n + y^n \neq z^n$.

Exercise 4.2. Imagine that FLT is still an open question. Design a simple program P such that P halts on empty input if and only if FLT is false.

This shows that deciding whether your program P halts is not easier than solving FLT.

Another mathematical problem that acquired great fame among the general public was the 4-color theorem (4CT), which states that *every planar graph is 4-colorable.* The question was first proposed by Francis Guthrie, a cartographer, in 1852. More than 120 years later, in 1976, Kenneth Appel and Wolfgang Haken designed a program P about which they were able to prove that the 4CT is true if and only if P halts. Then they ran the program on a computer at UIUC. They waited and waited, not knowing whether the program will ever halt. It did!, within two weeks. Thus the theorem was proved. (Not a very elegant proof, though. Over a thousand cases checked. But the number of cases was not known in advance—could have been infinite.)

Actually, the HALTING problem is much harder even than suggested by these two examples.

Next we sketch the proof of the most fundamental result in computer science.

Theorem 4.3 (Undecidability of the HALTING problem)**.** $\mathsf{HALTING} \notin \mathscr{R}$.

In other words, there is **no algorithm** that could decide, whether a given program halts on empty input.

Proof. Assume for a contradiction that $\mathsf{HALTING}$ is decidable, i. e., there exists a hlting program H such that given a program P, we have

$$H(P) = \begin{cases} 1 & \text{if } P \text{ halts on empty input} \\ 0 & \text{if } P \text{ does not halt on empty input} \end{cases} \tag{2}$$

Exercise 4.4. Given H, construct a halting program H_1 with the following property. Let P be a program and X an input to P. H_1 accepts the pair (P, X) as input and

$$H_1(P, X) = \begin{cases} 1 & \text{if } P \text{ halts input } X \\ 0 & \text{if } P \text{ does not halt on input } X \end{cases} \tag{3}$$

The idea is to encode the input X into the program P; this step produces a program P_1 such that the behavior of P_1 on empty input is the same as the behavior of P on input X; and then apply H to P_1. The program H_1 can accomplish this.

[1]For light entertainment centered around the human drama leading to the proof, watch the musical "Fermat's Last Tango." Among the characters featured are Pythagoras, Euclid, Newton, and Gauss, inhabitants of the *Aftermath.*

Our next step is to construct a program A that accepts pairs (P, X) as input, where P is a program and X is an input to P, with the property that

$$A(P, X) = \begin{cases} ? & \text{if } H_1(P, X) = 1 \\ 0 & \text{if } H_1(P, X) = 0 \end{cases} \tag{4}$$

Note that, thanks to H, we were able to switch halting and non-halting: $A(P, X)$ will halt if and only if $P(X)$ does not halt.
Here is a pseudocode for A, with reference to the halting program H_1.

```
procedure A
Input: $(P, X)$
$z := H_1(P, X)$,    $u := 1$
if $z = 1$ then
    while $u > 0$ do $u := u + 1$
    endwhile
else return 0, halt
```

Finally, let us consider the program B that takes as input any program P and is defined as $B(P) := A(P, P)$. Then, according to Eq. (4), for all programs P we have

$$B \text{ halts on input } P \text{ if and only if } P \text{ does not halt on input } P. \tag{5}$$

Since this is true for every program P, it is true in particular for $P := B$. Then it says,

$$B \text{ halts on input } B \text{ if and only if } B \text{ does not halt on input } B, \tag{6}$$

which is absurd. This contradiction proves that no such H exists. □

This is an example of the **diagonal argument** that descends from the ancient Greek paradox "all Cretans lie" (told by Cretan philosopher Epimenides, cca. 600 BCE) and was first employed as a template of a mathematical proof by Georg Cantor (1845–1918), creator of set theory, in his famous proof that there are more real numbers than there are integers.

Exercise 4.5. Prove: $\mathsf{HALTING} \notin co\mathcal{RE}$.

Exercise 4.6. (a) Prove: $\mathcal{RE} \neq \mathcal{R}$.
(b) Prove: $\mathcal{RE} \neq co\mathcal{RE}$.

5 Polynomial time, Cook reduction

Definition 5.1. Let P be a halting program. We say that P runs in **polynomial time** if there exists a polynomial p such that on every input x the number of steps the program takes is $\leq p(|x|)$. We say that a total function $f : \Sigma_1^* \to \Sigma_2^*$ is **polynomial-time computable** if there is a polynomial-time program that computes f.

Examples 5.2. The following functions are polynomial-time computable.
Multiplication of integers, multiplication of integral matrices (matrices with integer entries), computing the matching number of a bipartite graph (Kőnig, 1931) and of a general graph (Edmonds, 1965), computing the determinant and the rank of an integral matrix (Edmonds, 1965), solving a nonsingular $n \times n$ system of linear equations with integer coefficients, modular exponentiation.

Exercise 5.3. Given the positive integers n and m, prove that (F_n mod m) is computable in polynomial time. Estimate the degree of the polynomial. (Here F_n is the n-th Fibonacci number, and (a mod m) denotes the smallest non-negative remainder of a modulo m. For instance, (38 mod 9) = 2 and (−38 mod 9) = 7.)

Definition 5.4. The language class $\mathcal{P}$ is the set of polynomial-time computable languages (i. e., languages of which the characteristic functions can be computed in polynomial time).

Exercise 5.5. None of Examples 5.2 belongs to $\mathcal{P}$.

Examples 5.6. The following languages belong to $\mathcal{P}$: all finite languages, all cofinite languages, the set of bipartite graphs, the set of graphs with a perfect matching (Edmonds 1965), the set of planar graphs, the set of connected graphs, the set of even numbers, the set of squares $(1, 4, 9, 16, \dots)$, the set of prime numbers (Agrawal–Kayal–Saxena 2002).

Definition 5.7. A polynomial-time Turing reduction is called a Cook reduction. Notation: $f_1 \prec_{\text{Cook}} f_2$. If two functions are mutually Cook reducible to each other then we say that they are Cook equivalent.

Exercise 5.8. If $f \prec_{\text{Cook}} f_2$ and f_2 can be computed in polynomial time then f_1 can be computed in polynomial time.

Exercise 5.9. Every language is Cook equivalent to its complement.

6 Optimization vs. decision

6.1. It is an important observation that under rather general circumstances, discrete optimization problems are Cook equivalent to their decision versions. The key condition is that the optimum value is an integer with a polynomially bounded number of digits.

Definition 6.2. A **discrete optimization problem** has the following input: finite descriptions of functions $f, g_1, \dots, g_k : \mathbb{Z}^n \to \mathbb{Z}$. The problem is to find

$$\max_{x \in \mathbb{Z}^n} \{ f(x) \mid (x \in \mathbb{Z}^n) \text{ and } (\forall j)(g_j(x) \geq 0) \} . \tag{7}$$

(The maximum is either an integer or $\pm\infty$.)
The g_j are called the **constraints** and f is the **objective function**. We say that an assignment $x \in \mathbb{Z}^n$ is **feasible** if it satifies the constraints. We say that the system is **feasible** if there is a feasible point $x \in \mathbb{Z}^n$. The optimum of an infeasible system is $-\infty$ (maximum of the empty set).
The **decision problem** corresponding this optimization problem takes the same input plus an integer N and asks the question

$$(\exists ? x \in \mathbb{Z}^n)((\forall j)(g_j(x) \geq 0) \text{ and } f(x) \geq N) . \tag{8}$$

Exercise 6.3. (a) The decision problem Eq. (8) can be solved by a single query to an oracle for the optimization problem Eq. (7).
(b) Assume we are given an integer N_0 and a guarantee that for any feasible $x \in \mathbb{Z}^n$ we have $|f(x)| \le N_0$. Then we can solve the optimization problem Eq. (7) by making at most $\lceil \log_2(2N_0 + 1) \rceil$ queries to an oracle to the decision problem Eq. (8).
In both cases, state, what additional computation you need to do in addition to making the oracle queries.

Hint. (a) is obvious. For (b), use binary search on N, starting with $-N_0$. The answer to the first question should tell whether the optimization problem is feasible; what is the first question?

6.4. In discrete optimization problems we often require the variables to be Boolean (take values 0 or 1). This can be ensured by the constraints $x_{\ge}0$ and $1 - x_i \ge 0$. When we say "the variables are Boolean," we assume these constaints without actually listing them.

Example 6.5. The **integer Knapsack problem** (all weights and values are integers) is an example of a discrete optimization problem with Boolean variables, used to select the items. The input is a list $(w_1, \dots, w_n, v_1, \dots, v_n, W)$ of positive integers (the weights, the values, and the weight limit). We have n Boolean variables (the selectors), listed as $x = (x_1, \dots, x_n)$. The constraints are that the variables are Boolean and $\sum_{i=1}^n x_i w_i \le W$. The objective function is $\sum_{i=1}^n x_i v_i$. So here is a description of the problem:

$$\max_{x \in \{0,1\}^n} \left\{ \sum_{i=1}^n x_i v_i \;\middle|\; \sum_{i=1}^n x_i w_i \le W \right\} . \tag{9}$$

The corresponding decision problem has an additional input item, N, the target value, and the question is, does there exist an input $(x_, \dots, x_n)$ that satisfies the constraints and additionally satisfies the inequality $\sum_{i=1}^n x_i v_i \ge N$.

Exercise 6.6. Let KNAP denote the set of inputs to the integer Knapsack problem (weights, values, weight limit) that can achieve value $\ge N$. (KNAP is the language corresponding to he decision problem described in the preceding exercise.) Show that the integer Knapsack problem is Cook-reducible to the language KNAP.

Definition 6.7. The **Factoring problem** takes a positive integer x as input and returns its prime factorization: the list of prime factors in non-decreasing order. For instance, if $x = 360$ then the output is $(2, 2, 2, 3, 3, 5)$. If $x = 1$ then the output is $()$.
The **decision version** of the Factoring problem is the language

$$\mathsf{FACT} = \{(x, s) \mid x, s \ge 1 \text{ are integers and } (\exists d)(2 \le d \le s \wedge d \mid x)\} . \tag{10}$$

Exercise 6.8. Prove that the Factoring problem is Cook equivalent to the language FACT. For the reduction a Factoring to FACT, state the number of oracle queries; make it asymptotically as small as you can.

7 The class $\mathcal{NP}$

7.1. We formalize the notion of efficiently verifiable puzzles. Think of a puzzle like Sudoku, which may or may not have a solution and may have multiple solutions. Solutions may be difficult to find (if they exist at all), but if soeone presents a purported solution, it is easy to verify, i. e., decide whether it is indeed a solution. A solution is a *witness of solvability*, and solvability is our main concern. An example of such puzzles is 3-colorability of graphs; a solution, which may or may not exist, is a legal 3-coloring of the vertices.

Definition 7.2. The language class $\mathcal{NP}$ is defined as follows.
Let $L \subseteq \Sigma^*$ be a language. We say that $L \in \mathcal{NP}$ if the following condition holds:

$$\begin{gathered}(\exists \text{ polynomial } p)(\exists A \in \mathcal{P})(\exists \text{ finite set } \Sigma_1) \\ (\forall x \in \Sigma^*)(x \in L \Leftrightarrow (\exists w \in \Sigma_1^*)(|w| \le p(|x|) \wedge (x, w) \in A)\,. \end{gathered} \tag{11}$$

If such a w exists, we call w a *witness* of the membership $x \in L$.

Example 7.3. $\mathsf{3COL} \in \mathcal{NP}$. Witness: a legal 3-coloring of x.
Usually, naming the witness should suffice to indicate the proof, the required properties of the witness (not too long, verifiable in polynomial time) being evident.

7.4. We ignored the issue when $x \in \Sigma^*$ is a string that does not encode a graph. Under any reasonable encoding of graphs, this should be polynomial-time decidable, so we can leave it to the judge. In this case, the judge would reject all witnesses.

Exercise 7.5. Prove that each of the following languages belong to $\mathcal{NP}$:

1. 3COL (the set of 3-colorable graphs)
2. 4COL, etc.
3. Hamiltonian graphs (graphs that have a Hamilton cycle)
4. KNAP (the decision version of the integer Knapsack problem)
5. FACT (the decision version of the Factoring problem)
6. SAT (Boolean circuit satisfiability)
7. 3SAT (satisfiability of 3-CNF formulas [3 literals per clause])

In each case, just name the witness.

Exercise 7.6. Define the class $co\mathcal{NP}$. (Check Def. 1.3.)

Exercise 7.7. Prove: $\mathcal{P} \subseteq \mathcal{NP}$. Find the simplest possible witnesses.

7.8. One of the greatest mathematical **conjecture**s of our time is that $\mathcal{P} \neq \mathcal{NP}$. This is one of the seven “Millennium problems of mathematics.” The Clay Foundation offers a \$1 million prize for a proof or disproof.

7.9. Another related conjecture states that $\mathcal{NP} \neq co\mathcal{NP}$.

Exercise 7.10. Consider the two conjectures stated above:

(a) $\mathcal{P} \neq \mathcal{NP}$

(b) $\mathcal{NP} \neq co\mathcal{NP}$

Prove one of the implications $(a) \Rightarrow (b)$ and $(b) \Rightarrow (a)$. Clearly state, which direction you are proving.

Exercise 7.11. Infer from Ex. 7.7 that $\mathcal{P} \subseteq \mathcal{NP} \cap co\mathcal{NP}$.

7.12. Yet another related **conjecture** states that $\mathcal{P} \neq \mathcal{NP} \cap co\mathcal{NP}$.

Exercise 7.13. Let (c) be the conjecture $\mathcal{P} \neq \mathcal{NP} \cap co\mathcal{NP}$. Prove one of the following implications:
$(a) \Rightarrow (c)$, $(b) \Rightarrow (c)$, $(c) \Rightarrow (a)$, $(c) \Rightarrow (b)$. Here (a) and (b) refer to Ex. 7.10.

Exercise 7.14. Prove: $\mathsf{FACT} \in co\mathcal{NP}$. Explicitly use AKS (Agrawal–Kayal–Saxena).

Exercise* 7.15. Prove: $\mathsf{FACT} \in co\mathcal{NP}$. Do not use AKS.
This result was known nearly three decades before AKS (Vaughan Pratt, 1975).

7.16. Note that by Ex. 7.5, this means $\mathsf{FACT} \in \mathcal{NP} \cap co\mathcal{NP}$.

Exercise 7.17. Prove: if Conjecture 7.12 fails then the RSA cryptosystem can be broken in polynomial time.

8 Karp reducibility, $\mathcal{NP}$-completeness

Definition 8.1. A **Karp-reduction** is a polynomial-time many-one reduction.
In other words, let $L_1 \subseteq \Sigma_1$ and $L_2 \subseteq \Sigma_2$. A Karp-reduction from L_1 to L_2 is a polynomial-time computable function $f : \Sigma_1^* \to \Sigma_2^*$ such that

$$(\forall x \in \Sigma_1^*)(x \in L_1 \Leftrightarrow f(x) \in L_2) \tag{12}$$

We say that L_1 is **Karp-reducible** to L_2 if such an f exists. This circumstance is denoted $L_1 \prec_{\text{Karp}} L_2$.

Exercise 8.2. Prove: If $L_1 \prec_{\text{Karp}} L_2$ then $L_1 \prec_{\text{Cook}} L_2$.

Exercise 8.3. Prove: If the converse of the preceding exercise holds then $\mathcal{NP} = co\mathcal{NP}$.

Exercise 8.4. Prove: If $L_1 \prec_{\text{Karp}} L_2$ and $L_2 \prec_{\text{Karp}} L_3$ then $L_1 \prec_{\text{Karp}} L_3$. The definition of Karp-reducibility involves a polynomial (the reduction is a polynomial-time computation). Let d_{ij} denote the degree of the polynomial for the $L_i \prec_{\text{Karp}} L_j$ reduction. Give your best upper bound on d_{13} in terms of d_{12} and d_{23}.

Exercise 8.5. Prove: $\mathsf{3COL} \prec_{\text{Karp}} \mathsf{HALTING}$.

Exercise 8.6. Prove: $\mathsf{non3COL} \prec_{\text{Karp}} \mathsf{HALTING}$.

Exercise 8.7. Give a very simple direct proof of the fact that $\mathsf{3COL} \prec_{\text{Karp}} \mathsf{4COL}$. Do not use the fact that $\mathsf{4COL}$ is $\mathcal{NP}$-complete.

Definition 8.8. We say that the language L is **$\mathcal{NP}$-complete** if

(a) $L \in \mathcal{NP}$

(b) $(\forall\Sigma)(\forall M \subseteq \Sigma^*)(M \in \mathcal{NP} \Rightarrow M \prec_{\text{Karp}} L)$.

The class of $\mathcal{NP}$-complete languages is denoted $\mathcal{NPC}$.

By definition, $\mathcal{NPC} \subseteq \mathcal{NP}$.

Theorem 8.9 (Steven Cook and Leonid Levin, 1972/73)**.** SAT *is $\mathcal{NP}$-complete.*

Exercise 8.10. Prove that the following two conjectures are equivalent:

(a) $\mathcal{P} \neq \mathcal{NP}$

(b) $\mathcal{NPC} \cap \mathcal{P} = \emptyset$

Do not use the Cook–Levin Theorem.

Exercise 8.11. Prove that the following two conjectures are equivalent:

(a) $\mathcal{NP} \neq co\mathcal{NP}$

(b) $\mathcal{NPC} \cap co\mathcal{NP} = \emptyset$

State, where you are using the Cook–Levin Theorem.

Exercise 8.12. Prove: If $L \in \mathcal{NP}$ and $\mathsf{SAT} \prec_{\text{Karp}} L$ then $L \in \mathcal{NPC}$.

Exercise 8.13. Prove: If $\mathsf{SAT} \prec_{\text{Karp}} \mathsf{FACT}$ then $\mathcal{NP} = co\mathcal{NP}$.

Exercise$^+$ 8.14. Prove: If $\mathsf{SAT} \prec_{\text{Cook}} \mathsf{FACT}$ then $\mathcal{NP} = co\mathcal{NP}$.