2 Computational Complexity

So far we have classified computational problems into those which can theoretically be solved on a computer (corresponding to recursive languages and functions) and those which can’t. But right back in the introduction (remember the travelling salesman example!) we hinted that some problems may be theoretically solvable but practically unsolvable, because the algorithm which solves them simply takes too long for even the most powerful computer to perform.

The rest of the course aims to refine the classification of solvable problems into those which are easier, those which are harder and those which are solvable in practice. (It is also possible to further classify unsolvable problems; this is the main aim of more advanced computability theory but is outside the scope of this course.)

Convention. In this chapter, all Turing machines we consider will be halting, unless otherwise stated.

2.1 Time and Space

For practical purposes, there are two important features of a computation of which we have so far taken no account:

- the length of time taken to perform a computation — in the case of a TM this corresponds to the number of steps in the computation.
- the amount of space or memory required for storage — for a TM this is the number of different tape squares visited during the computation.

Of course the amount of time and space used by a TM will depend on the input with which it is supplied. It turns out to be convenient to study the amount of time and space used as a function of the length of the input.

Definitions. Let \( T, S : \mathbb{N} \rightarrow \mathbb{N} \) be monotonically increasing functions, and suppose that \( T(n), S(n) \geq n \) for all \( n \).

A TM (NDTM) \( M \) runs in time \( T(n) \) on \( \mathcal{A}^* \) if for every \( \sigma \in \mathcal{A}^* \) the (every) computation of \( M \) on input \( \sigma \) halts in \( \leq T(|\sigma|) \) steps.

A TM (NDTM) \( M \) runs in space \( S(n) \) on \( \mathcal{A}^* \) if for every \( \sigma \in \mathcal{A}^* \) the (every) computation of \( M \) on input \( \sigma \) \( M \)'s head scans \( \leq S(|\sigma|) \) distinct squares.

Remark. We usually omit mention of the alphabet \( \mathcal{A} \) when it is clear from the context (often it will be \( \{0,1\} \)).

Remark. Similar definitions apply to multitape machines; in the case of space we require that each head scans \( \leq S(|\sigma|) \) squares.

Remark. Suppose TM (or NDTM) \( M \) runs in time \( T(n) \) and space \( S(n) \). Clearly if \( T'(n) \geq T(n) \) for all \( n \) then \( M \) also runs in time \( T'(n) \), and similarly for space. Also since in a computation \( M \) cannot see more squares than the number of configurations in that computation, \( M \) also runs in space \( T(n) + 1 \).

Remark. There is a result in the other direction. Suppose that \( M \) runs in space \( S(n) \) and is halting. First notice that on input \( \sigma \), \( M \)'s head never moves outside the tape region

\[
B^{S(|\sigma|)} \sigma B^{S(|\sigma|)} - |\sigma|.
\]

This means that the tape outside this region always remains blank. Hence, if \( M \) mentions \( k \) tape symbols and \( q \) states in its rules, there are only

\[
q(2S(|\sigma|))k^{2S(|\sigma|)}
\]

distinct configurations which can occur during such a computation. So if there is a computation

\( C_1, C_2, \ldots, C_r \)
where \( r \) exceeds this number, then by the pigeon hole principle, there exist \( i < j \) such that \( C_i = C_j \). But now
\[
C_1, C_2, \ldots, C_i, C_{i+1}, \ldots, C_j, C_{i+1}, C_{i+2}, \ldots, C_j, C_{i+1}, C_{i+2}, \ldots, C_j, C_{i+1}, \ldots
\]
would be a non-halting computation of \( M \). It follows then that \( M \) must run in time
\[
q(2S(n))k^{2S(n)} \leq \beta^{S(n)}
\]
for some constant \( \beta \).

**Remark.** Any TM (NDTM) which always halts runs on \( \mathcal{A}^* \) in time \( T(n) \), and hence space \( S(n) \), for some functions \( T \) and \( S \). For example, set \( T(n) \) to be \( n + 1 \) plus the length of the longest computation of \( M \) on any of the (finitely many) words \( \sigma \in \mathcal{A}^* \) of length less than or equal to \( n \).

### 2.2 Some Previous TM\( \text{s Revisited} \)

**Example.** Let \( M \) be the TM accepting PALINDROMES from Section 1.7. Then
\[
M \text{ runs in time } n(2n+2) + 2 \leq an^2 + 2,
\]
\[
M \text{ runs in space } n + 2.
\]

**Example.** Let \( M \) be a multitape TM, and suppose \( M \) runs in time \( T(n) \) and space \( S(n) \). Let \( N \) be the equivalent 1-tape TM given in the proof of Theorem 2 (Section 1.10). Then
\[
N \text{ runs in time } 2n + 3 + T(n)(4S(n) + 3) + 1 \leq \alpha T(n)^2,
\]
\[
N \text{ runs in space } 2S(n) + n + 1 \leq 4S(n).
\]

**Exercise.** Check the above examples in detail.

**Remark.** Note that the above are mostly overestimates of the time and space requirements. To say that a machine runs in time \( T(n) \) and space \( S(n) \) means that it needs no more than \( T(n) \) steps and scans no more than \( S(n) \) tape squares.

**Exercise.** For each of the following machines find polynomial functions \( T(n) \) and \( S(n) \), with at least the degree as low as possible, such that the given machine runs in time \( T(n) \) and space \( S(n) \):

- the TM \( M \) from Section 1.6;
- the multitape machine for palindromes from Section 1.9;
- the machine in your answer to the First Coursework Test.

### 2.3 Linear Speed Up and Space Reduction

We now have a way to describe the practical difficulty (that is, the time and space requirements) of an algorithm. Intuitively, the practical difficulty of a problem should be the minimum practical difficulty of an algorithm which solves it. However, the following theorem says that in general such a minimum does not exist. One can almost always make an algorithm faster and more space efficient to a limited extent, at the price of increasing the size of the tape alphabet and number of states.

**Theorem 10.** Let \( L \) be a language accepted by a TM [NDTM] running in time \( T(n) \) and space \( S(n) \). Then for every constant \( m \), there is a TM [NDTM] accepting \( L \) and running in time \( (T(n)/m) + (n + 2)^2 + 1 \) and space \( (S(n)/m) + n + 1 \).

**Proof.** Let \( M \) be a TM (or NDTM) accepting \( L \) and running in time \( T(n) \) and space \( S(n) \) on inputs from \( \mathcal{A}^* \), and choose an integer \( k > 4m \). Let the TM \( N \) act on inputs \( a_0a_1 \ldots a_{n-1} \) from \( \mathcal{A}^* \) as follows. \( N \) first scans across the input and compresses (overlapping) blocks of \( 2k - 1 \) symbols
\[
[B^{k-1}a_0 \ldots a_{k-1}][a_1 \ldots a_{2k-1}][a_{k+1} \ldots a_{3k-1}] \ldots [a_{rk+1} \ldots a_{n-1}B^{(r+2)k-n-1}]
\]
where \( r \) is maximal such that \( rk + 1 < n \) and the \([\ldots]\) are new tape symbols (except that we identify \( B \) with \([B^{2k-1}]\)). \( N \) now enters a state \([a_0^M]\) scanning symbol \([B^{k-1}a_0a_1 \ldots a_{k-1}]\) simulating the initial configuration of \( M \) which has state \( a_0^M \) scanning the \( a_0 \).
$N$ now works as follows. Suppose at some stage $N$ is in state $[q]$ scanning $[b_{k+1} \ldots b_0 \ldots b_{k-1}]$ on a tape printed with

$$[b_{-3k+1} \ldots b_{-2k} \ldots b_{-k-1}] [b_{-2k+1} \ldots b_{-k} \ldots b_{-1}] [b_{-k+1} \ldots b_0 \ldots b_{k-1}] [b_1 \ldots b_k \ldots b_{2k-1}] [b_{k+1} \ldots b_{2k} \ldots b_{3k-1}] \ldots$$

Then at the corresponding stage $M$’s configuration is

$$\ldots b_{-3k+1} b_{-3k+2} b_{-3k+3} \ldots b_{-1} b_1 b_{-3k-3} b_{3k-2} b_{3k-1} \ldots$$
in state $q$ scanning $b_0$.

$N$’s next move is determined as follows. Continue running $M$ from this configuration. If $M$ stops without its head ever moving outside the block

$$b_{-k+1} b_{-k+2} b_{-k+3} \ldots b_{-1} b_1 b_{-3k-3} b_{3k-2} b_{3k-1}$$

then $N$ stands still and goes into its accept state if $M$ did, and some fixed other state if not. On the other hand if $M$ eventually moves out of this block, say to the right, in state $q'$ with now the tape entries in this block replaced by

$$c_{-k+1} c_{-k+2} c_{-k+3} \ldots c_{-1} c_0 c_1 \ldots c_{k-3} c_{k-2} c_{k-1}$$

then $N$ alters its scanned tape symbol to

$$[c_{-k+1} c_{-k+2} c_{-k+3} \ldots c_{-1} c_0 c_1 \ldots c_{k-3} c_{k-2} c_{k-1}]$$

remembers it (using its states), moves one left and changes the tape symbol to

$$[b_{-2k+1} b_{-2k+2} \ldots b_{-k-1} b_{-k} c_{-k+1} c_{-k+2} \ldots c_{-2} c_{-1}]$$

then moves two right and changes that tape symbol to

$$[c_1 c_2 \ldots c_{k-2} c_{k-1} b_k b_{k+1} b_{k+2} \ldots b_{2k-1}]$$

and then goes into state $[q']$ scanning this square. It should be clear that since $k$ and the number of different tape symbols that $M$ employs are finite so are the number of tape symbols employed by $N$. Also it should be clear that each of the moves we require of $N$ are determined by its state and what it scans and hence can be captured with a finite number of rules. Finally it should also be clear that $M$ and $N$ are equivalent on $A^*$.

Concerning the space requirements of $N$, once the initial $n+1$ squares have been scanned to set up the configuration simulating $M$’s initial configuration $N$ only uses one square for every $k$ squares used by $M$. Hence $N$ runs in space

$$(S(n)/k) + n + 1 \leq (S(n)/m) + n + 1.$$  

Turning now to time, the initial set up takes at most $(n + 2)^2$ steps and apart from the final step of $N$ it only takes 4 steps to simulate what must amount to at least $k$ steps of $M$. Hence $N$ runs in time

$$(n + 2)^2 + 4(T(n)/k) + 1 \leq (T(n)/m) + (n + 2)^2 + 1.$$  

\[2.4 \quad \text{Complexity Classes}\]

The previous theorem implies that it is only worth studying the time and space required to accept a language up to a constant factor. This motivates the following definitions.

**Definition.** A language $L \subseteq A^*$ is in $Time(T(n))$ if there is a (1-tape) $TM$ accepting $L$ and running in time $\alpha T(n) + \beta$ for some constants $\alpha, \beta$. The class $Time(T(n))$ is called a deterministic time complexity class.
Definition. A language $L \subseteq \mathcal{A}^*$ is in $Space(S(n))$ if there is a (1-tape) TM accepting $L$ and running in space $\alpha S(n) + \beta$ for some constants $\alpha, \beta$. The class $Space(S(n))$ is called a deterministic space complexity class.

Remark. Analogous definitions can be made for non-deterministic machines; the resulting non-deterministic time/space complexity classes are called $NTime(T(n))$ and $NSpace(S(n))$ respectively. Similarly for multitape machines.

Exercise. Show that if $T_1(n) \leq T_2(n)$ for all sufficiently large $n$ then $Time(T_1(n)) \subseteq Time(T_2(n))$. State and prove a corresponding result for space.

Exercise. Show that $Time(T(n)) = Time(\alpha T(n) + \beta)$ for any constants $\alpha, \beta > 0$. State and prove a corresponding result for space.

Example. The language of palindromes over $\{0, 1\}$ is in $Time(n^2)$ and $Space(n)$.

Important Assumptions. For technical reasons, it is often necessary to assume that functions $S(n)$ and $T(n)$ defining complexity classes are well behaved. To this end we make the following global assumptions:

- When dealing with classes $Space(S(n))$ we shall assume that there is a TM which on input of length $n$ can set markers at distance $S(n)$ apart and runs in space $\alpha S(n) + \beta$ for some constants $\alpha, \beta \geq 0$. [Such functions are called space constructible.]
- When dealing with classes $Time(T(n))$ we shall assume that there is a TM which on input of length $n$ runs for $\alpha T(n) + \beta$ steps and then halts for some constants $\alpha, \beta \geq 1$. [Such functions are called time constructible.]

Almost all functions you are likely to think of are both time and space constructible, including standard arithmetic functions such as polynomials and exponentials.

2.5 Lower Bounds and Crossing Arguments

To show that a given language is in a class $Time(T(n))$ (or $Space(S(n))$) is often straightforward — we just need to construct a TM which accepts the language in few enough steps (or scanning few enough tape squares). But it is much less obvious how to prove that a language $L$ is not in $Time(T(n))$ or $Space(S(n))$, since this requires us to consider all TMs which accept the language.

One way to prove lower bounds is to apply the pigeon hole principle to construct what are called crossing arguments. Here is an example of this technique in action:

**Theorem 11.** Let $M$ be a TM accepting the language of palindromes over the alphabet $\{0, 1\}$. Suppose that $M$ runs in time $T(n)$. Then there exists a constant $\alpha > 0$ such that $T(n) \geq \alpha n^2$ for all sufficiently large $n$.

**Proof.** Given $n$ let $k$ be the integer part of $n/3$ and let $m = n - 2k$, so

$$k + 2 \geq m \geq n/3 \geq k \geq (n - 2)/3. \quad (1)$$

We shall look at inputs of the form $\sigma 0^m \sigma^R$ with $|\sigma| = k$. Let $1 \leq i \leq m$ and imagine the computation of $M$ on $\sigma 0^m \sigma^R$ and an observer sitting on the line between the $\sigma 0^i$ and $0^{m-i} \sigma^R$ blocks. During this computation suppose that $M$’s head moves across this interface going into states $q_1, q_2, \ldots, q_t$ in that order. In particular then $q_1$ was the state $M$ was entering the first time it crossed the interface, necessarily from left to right, $q_2$ was the state it was entering when it next crossed back and so forth. Call this sequence $q_1, q_2, \ldots, q_t$ the $i$th crossing sequence on input $\sigma 0^m \sigma^R$ and denote it $C_i^\sigma$. Write $|C_i^\sigma|$ for the number of states in $C_i^\sigma$. Notice that

$$T(n) \geq \text{time taken by } M \text{ on } \sigma 0^m \sigma^R \geq \sum_{i=1}^m |C_i^\sigma|.$$  

Hence there must be some $1 \leq i_\sigma \leq m$ such that

$$|C_{i_\sigma}^\sigma| \leq T(n)/m \quad (2)$$

30
or else the sum of these lengths would exceed \( T(n) \). Now suppose \( M \) has \( s \) states. Then the number of distinct possibilities for \( C^\sigma_i \) as \( \sigma \) ranges over words in \( \{0, 1\}^* \) of length \( k \) is, by (2), at most

\[
\sum_{r \leq T(n)/m} s^r \leq s^{(T(n)/m)+1}. 
\]

(3)

where \( s \) is the number of states in \( M \) and we assume (as we clearly can) that \( s \geq 2 \).

Now suppose for a contradiction that \( \tau \in \{0, 1\}^* \), \(|\tau| = |\sigma| = k \), \( \tau \neq \sigma \) and \( C^\tau_j = C^\sigma_i = q_1, q_2, \ldots, q_t \) for some \( 0 < i, j \leq m \). Consider the following:

<table>
<thead>
<tr>
<th>Confis of ( M ) on ( \sigma 0^m \sigma^R )</th>
<th>Confis of ( M ) on ( \tau 0^m \tau^R )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \sigma 0^i</td>
<td>0^{m-i} \sigma^R ) initial config.</td>
</tr>
<tr>
<td>( \ldots )</td>
<td>( \ldots )</td>
</tr>
<tr>
<td>( \nu_i</td>
<td>0^{m-i} \sigma^R ) 1st crossing into state ( q_1 )</td>
</tr>
<tr>
<td>( \ldots )</td>
<td>( \ldots )</td>
</tr>
<tr>
<td>( \nu_i</td>
<td>\xi_1 ) 2nd crossing into state ( q_2 )</td>
</tr>
<tr>
<td>( \ldots )</td>
<td>( \ldots )</td>
</tr>
<tr>
<td>( \nu_2</td>
<td>\xi_1 ) 3rd crossing into state ( q_3 )</td>
</tr>
<tr>
<td>( \ldots )</td>
<td>( \ldots )</td>
</tr>
<tr>
<td>( \ldots )</td>
<td>( \ldots )</td>
</tr>
</tbody>
</table>

In all cases \( M \) eventually crosses for the last time into state \( q_t \) and subsequently accepts since \( \sigma 0^m \sigma^R \) and \( \tau 0^m \tau^R \) are both palindromes. But then \( M \) must also accept \( \sigma 0^i 0^{m-j} \tau^R = \sigma 0^{i+m-j} \tau^R \), which is clearly not a palindrome, giving a contradiction!

We conclude that our assumption that we could have \( C^\tau_j = C^\sigma_i \) must be false. Thus we have shown that if \( \sigma, \tau \in \{0, 1\}^* \), \(|\tau| = |\sigma| = k \), \( \tau \neq \sigma \) then \( C^\tau_j \neq C^\sigma_i \) for \( 1 \leq i, j \leq m \). In particular then the map

\( \sigma \mapsto C^\sigma_i \)

for \( |\sigma| = k \) must be injective. Since the set of such \( \sigma \) has \( 2^k \) elements and the set of possible \( C^\sigma_i \) has at most \( s^{(T(n)/m)+1} \) elements we conclude, by the pigeon-hole principle, that

\[ 2^k \leq s^{(T(n)/m)+1}. \]

Taking logs to base 2 gives

\[ k \leq ((T(n)/m) + 1) \log(s), \]
so, since $s > 1$, we have

$$T(n) \geq \frac{km}{\log(s)} - m \geq \frac{(n - 2)^2}{9 \log(s)} - \frac{n}{3} - 2$$

by (1) and sorting this out gives that

$$T(n) \geq n^2/9 \log(s) - \beta n - \gamma$$

for some $\beta, \gamma$ and hence that

$$T(n) \geq n^2/10 \log(s)$$

for all $n$ eventually. \qed

**Corollary 12.** The language of palindromes over $\{0,1\}$ is in

$$\text{Time}(n^2) \setminus \text{Time}(n^{2-\epsilon})$$

for every $\epsilon > 0$.

**Remark.** The theorem says that the language of palindromes cannot be recognised by a 1-tape TM in less than quadratic time, but we have seen that there is a 2-tape TM which recognises palindromes in linear time. We saw in Section 2.2 that our 1-tape simulation of a multitape TM roughly squares the time requirement; the above implies that we cannot significantly improve on this in general.

### 2.6 Hierarchy Theorems

Intuition and the last section both suggest that the more time and space we are allowed, the more problems we can solve. The following theorem is a precise mathematical statement of this intuition in the case of space.

**The Space Hierarchy Theorem.** Let $S_1(n)$ and $S_2(n)$ be space constructible functions such that

$$\lim_{n \to \infty} \frac{S_1(n)}{S_2(n)} = 0. \quad (4)$$

Then $\text{Space}(S_1(n)) \subsetneq \text{Space}(S_2(n))$.

**Proof.** We construct a multitape TM $K$ which acts as follows. On input $\sigma \in \{0,1\}^*$ $K$ first checks that $\sigma$ has the form $0^k \sigma_M$ where $\sigma_M$ is the code for a TM $M$; the start of the word $\sigma$ is easily found since codes for Turing machines start with 01, and it is an easy exercise to show that this can be done in space $\alpha n + \beta$ for some $\alpha$ and $\beta$. If not $K$ rejects. Otherwise, for $n = |\sigma|$ $K$ writes

$$@B^{S_2(n)} @@$$

on tape 2 and

$$@B^{S_2(n)} \sigma_M 1111110^k \sigma_M B^{S_2(n)} @@$$

on tape 3. This is possible in space $\alpha S_2(n) + \beta$ for some $\alpha, \beta$, because of the assumption that the function $S_2(n)$ is space constructible.

$K$ now acts on tape 3 (and some other working tapes) like the multitape universal TM $U$ from Section 1.14, simulating the operation of $M$ on input $\sigma = 0^k \sigma_M$. However, each time it simulates a move of $M$, it increments the binary number stored between the @’s on tape 2. [This is necessary to ensure that if $M$ never halts then $K$ limits how long it keeps up the simulation.]

If ever the head on tape 3 sees a @ then $K$ halts without accepting. Similarly, if when incrementing the number of tape 2, it discovers a @ in the way at the left-hand end, then $K$ halts without accepting. Note that the latter must occur if $K$ is still running after simulating $2^{S_2(|\sigma|)}$ steps of $M$, so this guarantees that $K$ is a halting TM. Alternatively, if $M$ halts before either of these things happen, then $K$ halts, accepting exactly if $M$ does not accept.
Now let $L \subseteq \{0, 1\}^*$ be the language accepted by $K^7$. Clearly, $K$ runs in (multitape) space $\alpha S_2(n) + \beta$ for some constants $\alpha$ and $\beta$. It follows by the observation in Section 2.2 that $L \in \text{Space}(S_2(n))$. It remains to show that $L \notin \text{Space}(S_1(n))$.

Suppose on the contrary that $M$ was a TM running in space $\alpha M S_1(n) + \beta_M$ and accepting $L$. Take $k$ very large$^8$ so $n$ is also very large, and consider the result of running $M$ on $0^k \sigma_M$. Since $M$ runs in space $\alpha M S_1(n) + \beta_M$ we know that it runs in time $\gamma S_1(n)$ for some constant $\gamma$. By (4) for such large $n$ we have that, $2^S_2(n) > \gamma S_1(n)$. Hence $K$ does not stop because it reaches $2S_2(n)$ on tape 2.

Since $M$ runs in space $\alpha M S_1(n) + \beta_M$, when simulating it $U$ runs in space $\gamma (\alpha M S_1(n) + \beta_M)$ for some $\gamma$ (see Exercise Sheet 2). Using (4) again, this is less than $S_2(n)$ for large enough $n$. Hence $K$ does not stop because it sees a @ on tape 3. The only possibility left is that it halts because $M$ halts. But then it accepts just if $M$ does not accept, which contradicts $M$ being equivalent to $K$.

**Remark.** Compare this proof with the method we first used to construct a non-recursive language (Section 1.13). The two proofs have the same essential idea at heart: show that a language $L$ is *not* accepted by a TM (or a TM in some particular class), by associating to each TM a word which is specially constructed to be in $L$ if and only if it is *not* accepted by the machine. This technique is called diagonalization.$^9$

**Remark.** The language $L$ in the proof does not actually depend on the function $S_1(n)$. Thus, we see that there is a single language $L \in \text{Space}(S_2(n))$ which isn’t in $\text{Space}(S(n))$ for *any* $S(n)$ with

$$\lim_{n \to \infty} \frac{S(n)}{S_2(n)} = 0.$$ 

When we turn our attention to time, the situation is not quite so clear-cut. It turns out that the most obvious analogue of the Space Hierarchy Theorem does *not* hold, but it is still possible to establish a related result.

**The Time Hierarchy Theorem.** Let $T_1(n)$ and $T_2(n)$ be time constructible functions such that

$$\lim_{n \to \infty} \frac{T_1(n)}{T_2(n)} = 0. \quad (5)$$

Then $\text{Time}(T_1(n)) \subset \text{Time}(T_2(n)^2)$.

**Proof.** By the time constructibility assumption on $T_2(n)$ there is a TM, $R$ say, which on input $\sigma$ runs for $\alpha_R T_2(|\sigma|) + \beta_R$ steps ($\alpha_R, \beta_R \geq 1$) and then stops.

We describe a multitape TM $K$. On input $\tau = 0^k \sigma \in \{0, 1\}^*$ where $\sigma$ starts with 01, $K$ writes $\sigma|111110^k \sigma$ on tape 2. If no such $\sigma$ is found then $K$ stops without accepting. Otherwise, $K$

- runs like $U$ (the multitape universal TM from Section 1.14 on tape 2 (and the other tapes $U$ needs); and

- simultaneously runs like $R$ on tape 1.

If $L$ halts before $U$ then $K$ halts and does not accept (say). On the other hand if $U$ halts first then $K$ accepts if and only if $U$ does *not* accept.

Let $L$ be the language accepted by $K$. Clearly by the standing assumptions on $T_2(n)$ and the “clock” provided by $R$, $K$ runs in time $\alpha_K T_2(n) + \beta_K$ for some $\alpha_K, \beta_K$. Hence, after moving to a single tape TM, $L \in \text{Time}(T_2(n)^2)$.

---

$^7$It is possible, but not especially illuminating, to exactly describe $L$. For what it is worth, $L$ is the language of all words of the form $\sigma | 0^k \sigma_M$ where $\sigma_M$ codes a TM $M$, and $M$ does not accept $0^k \sigma_M$ in fewer than $2^S_2(|\sigma|)$ steps and without moving the tape head more than $S_2(|\sigma|)$ squares to the left of the start, or $S_2(|\sigma|) + 6 + |\sigma|$ to the right.

$^8$If you are not happy with this kind of argument, you can work out a more precise statement of exactly *how* large $k$ needs to be by examining the subsequent argument.

$^9$To understand the terminology, think of a table with TM’s ranged along the top, the corresponding words down the left-hand-side in the appropriate order, and the table entries indicating whether each TM “agrees with” $L$ on each word. Also, compare with Cantor’s proof that the real numbers are not countable.
We claim that $L \notin \text{Time}(T_1(n))$, from which the required result obviously follows. Suppose for a contradiction that it was and let $M$ be a TM running in time $\alpha_MT_1(n) + \beta_M$ and accepting $L$. Let $k$ be large and consider the computation of $K$ on input $0^k\sigma_M$ where as usual $\sigma_M$ is a code for $M$. In simulating $M$ on input $0^k\sigma_M$, $K$ uses time $\gamma(\alpha_MT_1(|0^k\sigma_M|) + \beta_M)$ for some $\gamma$ which depends on $\sigma_M$ but not $k$ (see Exercise Sheet 2).

Since $k$ is large, by (5),

$$\gamma(\alpha_MT_1(|0^k\sigma_M|) + \beta_M) < \alpha_RT_2(|0^k\sigma_M|) + \beta_R.$$ 

This means then that $K$ completes the simulation of $M$ on this input before the clock given by $R$ runs down. But in that case $K$ accepts if and only if $M$ does not accept, contradicting the assumption that $M$ and $K$ accept the same language.

\[ \square \]

**Corollary.** For every $p > 0$ there are languages in $\text{Time}(n^{2p+2}) \setminus \text{Time}(n^p)$.

**Remark.** There are similar hierarchy results for $N\text{Time}$ and $N\text{Space}$.

### 2.7 $\mathcal{P}$, $N\mathcal{P}$ and Tractability

At this point we might ask what kind of upper bound is required to establish that a problem is *practically* solvable.

**Definition.** The class $\mathcal{P}$ of *polynomial time languages* is defined by

$$\mathcal{P} = \bigcup_{p(x) \in \mathbb{N}[x] \setminus \{0\}} \text{Time}(p(n))$$

where $\mathbb{N}[x]$ is the set of polynomials in the variable $x$ with coefficients from $\mathbb{N}$.

**Example.** The language of palindromes over $\{0, 1\}$ (or any other alphabet) is in $\mathcal{P}$.

**Remark.** Any non-zero polynomial with non-negative coefficients is time (and space) constructible and monotonically increasing on the non-negative integers, so this does make sense.

Empirical evidence has led to the formulation of the following claim.

**The Tractability Thesis**\(^\text{10}\). A language which arises in practice lies in $\mathcal{P}$ if and only if its membership problem is *practically* solvable (or tractable).

**Remark.** The Tractability Thesis plays a role for complexity theory which the Church-Turing Thesis plays for computation theory. They each state a presumed connection between an abstract mathematical entity (TM’s in one case, the complexity class $\mathcal{P}$ in the other) and a real world phenomenon which we want it to model (algorithms in general, practical algorithms).

Why should we believe the Tractability Thesis? It is in the same bracket as the Church-Turing Thesis, as a philosophical or physical statement rather than a mathematical one. It differs, however, in that the evidence presented for it is largely empirical, rather than intuitive or philosophical.

From what we have seen so far, there is no obvious reason to believe either implication; indeed, quite the reverse! Firstly, by the corollary to the Time Hierarchy Theorem (Section 2.6), there exist polynomial time problems, but where the degree of the polynomial governing the number of steps is arbitrarily high. It is pretty clear that an algorithm which takes $n^{2018}$ steps on input of size $n$ will not be practical even when $n = 2$; in fact anything more than $n^4$ usually causes major problems. Conversely, an algorithm which takes the integer part of $2^{(n/1000000)}$ steps may be practical for quite large $n$, even though this function will eventually exceed any polynomial.

However, while it is quite easy to cook up “laboratory counterexamples” to the Thesis, these counterexamples all seem to be problems of a very esoteric nature which don’t actually turn up in any real world.

\(^\text{10}\)This name is not standard. The stronger claim that an arbitrary (rather than real-world) language lies in $\mathcal{P}$ if and only if its membership problem is tractable is often called *Cobham’s Thesis* or the *Cook-Karp Thesis*. The related claim that a Turing machine is as efficient as any physical means of computation is sometimes called the *Extended Church-Turing Thesis*. 

34
scenario. Experience suggests that polynomial time gives a good approximation of feasibility in the real world, and most computer scientists accept the existence [non-existence] of a polynomial time algorithm as proof (or at least strong evidence) that a problem can [respectively, cannot] be solved in practice. As mathematicians, however, you may glad to hear that we shall not place any reliance on the correctness of the thesis!

It is widely believed, then, that the real-world problems we can solve lie in the class $P$ of deterministic polynomial time problems. Unfortunately, many of the important problems we would like to solve are only known to be solvable in non-deterministic polynomial time, that is, in $NTime(p(n))$ for some polynomial $p(n)$.

**Definition.** The class $NP$ of non-deterministic polynomial time languages is defined by

$$NP = \bigcup_{p(x) \in \mathbb{N}[x] \backslash \{0\}} NTime(p(n)).$$

So why does $NP$ contain so many of the problems we would like to solve in practice? Intuitively, non-deterministic algorithms have the ability to perform certain exhaustive search processes very efficiently, by guessing the correct place to look. So many decision problems which admit solution by exhaustive search lie in the class $NP$.

**Example.** Let $FARE^{11}$ be the language

$$\{ \overline{a_1} \oplus \overline{a_2} \oplus \ldots \oplus \overline{a_k} \oplus \overline{b} \mid a_1, \ldots, a_k, b \in \mathbb{N} \text{ and for some } S \subseteq \{1, 2, \ldots, k\}, \sum_{i \in S} a_i = b \}$$

We claim that $FARE \in NP$; to show this we shall describe an NDTM to accept it. The machine first finds the right-hand end of the input, and then moves back over $\overline{b}$ to find the rightmost $\oplus$. It now moves back to the left, but at each $\oplus$ it makes a (non-deterministic) choice whether to pass over the next $\overline{a_i}$ leaving it unchanged, or whether to scrub it out replacing it with more $\oplus$ symbols.

Thus, it arrives back at the left with $\overline{b}$ and some of the $\overline{a_i}$’s intact, but some of the $\overline{a_i}$’s removed. Clearly, every possible combination of $\overline{a_i}$’s removed and remaining (that is, every choice of $S$) will be considered in some computation. At this point, the machine checks (deterministically) whether the remaining $a_i$’s sum to $b$. If so, it accepts; otherwise this computation halts without accepting.

Clearly, the initial pass back and forth to delete some of the $a_i$’s can be done in $2n + 4$ steps, while the (deterministic) task of checking whether the remaining $a_i$’s sum to $b$ can be also be implemented in polynomial time (see Exercise Sheet 2). Thus, the machine runs in (non-deterministic) polynomial time, and so $FARE \in NP$.

**Remark.** For any polynomial

$$p(x) = m_r x^r + m_{r-1} x^{r-1} + \ldots + m_1 x + m_0$$

in $\mathbb{N}[x] \backslash \{0\}$ we have

$$p(n) \leq (m_r + m_{r-1} + \ldots + m_1)n^r + m_0$$

for all $n \in \mathbb{N}$, so that

$$P = \bigcup_{r \in \mathbb{N}} Time(n^r) \quad \text{and} \quad NP = \bigcup_{r \in \mathbb{N}} NTime(n^r).$$

**Remark.** Since multitable machines are equivalent to single tape machines with, essentially, just a squaring of the time, we could replace “1-tape time” here by multitable time without affecting the classes defined.

**Remark.** Exactly analogously, we can define the classes of polynomial space decision problems ($PSpace$) and non-deterministic polynomial space decision problems ($NPSpace$).

---

11So called because it is the problem faced when deciding whether the coins in one’s pocket will allow one to pay an exact bus fare! However, the names of these languages are not entirely standardised. Sometimes this problem is called $KNAPSACK$, but we shall use the term $KNAPSACK$ for something else later on (in Section 2.9).
Remark. Given what we know at this point we can show (see Exercise Sheet 2) that

\[ \mathcal{P} \subseteq \mathcal{NP} \subseteq \mathcal{P}Space. \]

Despite millions of man-hours of research, nothing more is known here. In particular, we don’t know if either of the above inclusions are strict.

Remark. Given the above discussion, the question of whether \( \mathcal{P} = \mathcal{NP} \) is particularly important and intriguing. A “generally held” believe among the cognoscenti is that \( \mathcal{P} \neq \mathcal{NP} \). Partly this is based on pure intuition, but there is also an argument as follows: proving that \( \mathcal{P} = \mathcal{NP} \) would involve proving an upper bound, while proving that \( \mathcal{P} \neq \mathcal{NP} \) would involve proving a lower bound. Because proving upper bounds is usually much easier than proving lower bounds, it is far more plausible that everyone has failed to do the latter, than that everyone has failed to do the former.

Remark. It is generally believed that replacing the Turing machines in our definition with a different idealized notion of a language acceptor would not dramatically change things, and in particular would not alter \( \mathcal{P} \) and \( \mathcal{NP} \). It may be that probabilistic computers and/or quantum computers are in some sense an exception; however, they are not directly comparable in the same framework, since they work by producing a correct answer with high probability instead of with certainty.

2.8 Basic Properties of \( \mathcal{P} \) and \( \mathcal{NP} \)

**Proposition 13.** Let \( L_1, L_2 \subseteq \mathcal{A}^* \) be languages in \( \mathcal{P} \). Then \( L_1 \cap L_2, L_1 \cup L_2 \) and \( \mathcal{A}^* \setminus L_1 \) are in \( \mathcal{P} \).

**Proof.** Suppose \( M_i = \) a TM running in polynomial time \( p_i(n) \) and accepting \( L_i \) for \( i = 1, 2 \). Let \( M \) be the 3TM which on input \( \sigma \) copies \( \sigma \) on tapes 2 and 3 and then runs \( M_1 \) on tape 2, \( M_2 \) on tape 3 until they both stop, accepting just if they both do. Then \( M \) clearly accepts \( L_1 \cap L_2 \). It runs in (multitape) time \( 2n + 2 + p_1(n) + p_2(n) + 1 \), and hence can be simulated by 1-tape TM running in polynomial time.

The proof for complement \( \mathcal{A}^* \setminus L_1 \) is even easier. Just as in the more formal proof of Theorem 1 (Section 1.8) we construct a machine which runs like \( M_1 \) until it halts and then give the opposite answer — this takes only one extra step, and so runs in time \( p_1(n) + 1 \).

The proof for \( L_1 \cup L_2 \) follows from the other two. \( \square \)

**Proposition 14.** Let \( L_1, L_2 \subseteq \mathcal{A}^* \) be languages in \( \mathcal{NP} \). Then \( L_1 \cap L_2 \) and \( L_1 \cup L_2 \) are in \( \mathcal{NP} \).

**Proof.** The proof for \( L_1 \cap L_2 \) is just as for \( \mathcal{P} \), and the proof for \( L_1 \cup L_2 \) is similar. \( \square \)

**Remark.** Notice that Proposition 14 omits to say whether \( \mathcal{A}^* \setminus L_1 \) is in \( \mathcal{NP} \). Indeed it is an open question whether this always holds, that is, whether \( \mathcal{NP} \) is closed under complement. This leads to the following definition.

**Definition.** \( \mathcal{NP}^c \) (sometimes called co\(\mathcal{NP} \)) is the class of languages \( L \subseteq \mathcal{A}^* \) such that \( (\mathcal{A}^* \setminus L) \in \mathcal{NP} \).

**Remark.** The (open) question of whether \( \mathcal{NP} \) is closed under negation is clearly equivalent to the question of whether \( \mathcal{NP}^c \subseteq \mathcal{NP} \). We shall see shortly that (Proposition 16) that this is also equivalent to whether \( \mathcal{NP} = \mathcal{NP}^c \).

**Proposition 15.** \( \mathcal{P} \subseteq \mathcal{NP} \cap \mathcal{NP}^c \).

**Proof.** Given what we already know, we only need check that if \( L \subseteq \mathcal{A}^* \) and \( L \in \mathcal{P} \) then \( L \in \mathcal{NP} \cap \mathcal{NP}^c \). But in this case \( \mathcal{A}^* \setminus L \in \mathcal{P} \subseteq \mathcal{NP} \) so directly \( L \in \mathcal{NP} \).

---

\(^{12}\)A survey of experts in 2002 found 61 thought \( \mathcal{P} \neq \mathcal{NP} \), 9 thought \( \mathcal{P} = \mathcal{NP} \) and 22 had no idea.

\(^{13}\)It is widely assumed that the question must be resolvable one way or the other by mathematical means, but there is (to your lecturer’s knowledge) no entirely satisfactory argument excluding the third possibility that the statement \( \mathcal{P} = \mathcal{NP} \) is independent of the axioms of set theory. In the survey mentioned above, 4 experts thought this would be the case.

\(^{14}\)As usual, it doesn’t matter which overlying alphabet \( A \) we take here. If \( L \subseteq \mathcal{A}^*_1 \cup \mathcal{A}^*_2 \) then

\[ \mathcal{A}^*_2 \setminus L = ((\mathcal{A}^*_1 \setminus L) \cap \mathcal{A}^*_2) \cup (\mathcal{A}^*_2 \setminus \mathcal{A}^*_1). \]

Since \( \mathcal{A}^*_1 \) and \( \mathcal{A}^*_2 \) are both clearly in \( \mathcal{P} \), by Proposition 13 so is \( \mathcal{A}^*_2 \setminus \mathcal{A}^*_1 \), so it follows by Proposition 14 that \( \mathcal{A}^*_1 \setminus L \in \mathcal{NP} \) if and only if \( \mathcal{A}^*_2 \setminus L \in \mathcal{NP} \).
Remark. It is an open question whether in fact \( \mathcal{P} = \mathcal{NP} \cap \mathcal{NP}^c \). Until recently, it seemed possible that the language \textsc{Primes} (see Section 1.2), which was known to be in \( \mathcal{NP} \cap \mathcal{NP}^c \), might separate these classes. However \textsc{Primes} has recently been shown to be in \( \mathcal{P} \). It now seems that the only remaining natural decision problem which is known to be in \( \mathcal{NP} \cap \mathcal{NP}^c \) but not known to be in \( \mathcal{P} \) is the problem of deciding whether two given finite graphs are isomorphic.\(^{15}\)

**Proposition 16.**

(i) \( \mathcal{NP} \subseteq \mathcal{NP}^c \iff \mathcal{NP} = \mathcal{NP}^c \)

(ii) \( \mathcal{NP}^c \subseteq \mathcal{NP} \iff \mathcal{NP} = \mathcal{NP}^c \)

**Proof.** For (i) assume \( \mathcal{NP} \subseteq \mathcal{NP}^c \) and let \( L \subseteq \mathcal{A}^* \) be such that \( L \in \mathcal{NP}^c \). Then by definition \( \mathcal{A}^* \setminus L \in \mathcal{NP} \). From the assumption that \( \mathcal{NP} \subseteq \mathcal{NP}^c \) we have \( \mathcal{A}^* \setminus L \in \mathcal{NP}^c \), but now \( L = \mathcal{A}^* \setminus (\mathcal{A}^* \setminus L) \in \mathcal{NP} \). We have shown \( \mathcal{NP}^c \subseteq \mathcal{NP} \) and with our assumption it follows that \( \mathcal{NP} = \mathcal{NP}^c \). The converse is immediate, and case (ii) is very similar. \( \Box \)

### 2.9 Polynomial Time Reducibility

Right back at the start of the course (remember factorisation and primality testing?) we saw some examples where a solution for one problem seemed to lead easily to a solution for another. Our next objective is to formalise this idea.

**Definition.** A function \( f : \mathcal{A}_1^* \rightarrow \mathcal{A}_2^* \) is said to be polynomial time computable if there is a TM \( M \) which computes \( f \) and runs in time \( p(n) \) for some polynomial \( p(x) \in \mathbb{N}[x] \).

**Remark.** As always, the definition extends via codings to functions \( f : \mathcal{N} \rightarrow \mathcal{N} \), \( f : \mathcal{N}^k \rightarrow \mathcal{N} \) and so forth.

**Exercise.** Show that the addition function

\[
+ : \mathcal{N}^2 \rightarrow \mathcal{N}, \quad (x, y) \mapsto x + y
\]

is polynomial time computable.

**Example.** For any \( p(x) \in \mathbb{N}[x] \) the function which on input \( \sigma \in \mathcal{A}^* \) takes value \( 1^{p(|\sigma|)} \in \{0, 1\}^* \) is polynomial time computable - see Exercise Sheet 2.

**Definition.** Let \( L_1 \subseteq \mathcal{A}_1^* \) and \( L_2 \subseteq \mathcal{A}_2^* \). We say that \( L_1 \) is polynomial time reducible to \( L_2 \), written \( L_1 \leq_p L_2 \), if there is a polynomial time computable function \( f : \mathcal{A}_1^* \rightarrow \mathcal{A}_2^* \) such that for all \( \sigma \in \mathcal{A}_1^* \),

\[
\sigma \in L_1 \iff f(\sigma) \in L_2,
\]

The function \( f \) is called a reduction; we say that \( f \) reduces \( L_1 \) to \( L_2 \) (in polynomial time) and write \( f : L_1 \leq_p L_2 \).

**Remark.** Whether or not \( L_1 \leq_p L_2 \) does not depend on the overlying \( \mathcal{A}_1^* \) and \( \mathcal{A}_2^* \), provided that \( L_1 \neq L_2 \) — see Exercise Sheet 2.

**Example.** Let \textsc{Knapsack}\(^{16}\) be the language

\[
\{ \overline{a_1 \oplus a_2 \oplus \ldots \oplus a_k} \mid a_1, \ldots, a_k \in \mathcal{N} \ \text{and} \ \sum_{i \in S} a_i = \sum_{i \notin S} a_i \}
\]

**Proposition.** \textsc{Knapsack} \( \leq_p \textsc{FARE} \).

\(^{15}\)By which of course we really mean something like “the language of all words \( \sigma @ \tau \) such that \( \sigma, \tau \in \{0, 1\}^* \) are codings of graphs (with some suitable coding system) and the graphs they code are isomorphic”. But we will increasingly allow ourselves to speak of decision problems in the abstract, ignoring the issue of how precisely to code them as languages.

\(^{16}\)This is the problem of deciding whether the items to be taken on a hike can be divided between the knapsacks of two walkers, in such a way that each carries the same weight! Called \textsc{Set Partition} by more sedentary types.
Proof. We need to define a function \( f : \{0, 1, \underline{a}\}^* \to \{0, 1, \underline{a}\}^* \). First of all, we pick some arbitrary element \( \tau \) of \( \{0, 1, \underline{a}\}^* \setminus FARE \), and define \( f(\sigma) = \tau \) whenever \( \sigma \) is not of the form\(^{17}\) \( \mathbf{a}_1 \underline{a}_2 \underline{a}_3 \ldots \underline{a}_k \) for some \( a_1, a_2, \ldots, a_k \in \mathbb{N}. \) Otherwise set
\[
f(\mathbf{a}_1 \underline{a}_2 \underline{a}_3 \ldots \underline{a}_k) = 2\mathbf{a}_1 \underline{a}_2 \underline{a}_3 \ldots \underline{a}_k \underline{a}(\sum_{i=1}^{k} a_i).
\]

Then \( f \) is polynomial time computable and there is some choice of 2a’s from 2a1, 2a2, ..., 2ak which sum to \( \sum_{i=1}^{k} a_i \) exactly if there is a choice from \( a_1, a_2, \ldots, a_k \) which sum to \( (\sum_{i=1}^{k} a_i)/2 \) (equivalently which sums to the same total as its complement). Hence \( f : K N A S P A C K \subseteq_p F A R E. \)

**Proposition.** \( F A R E \subseteq_p K N A S P A C K. \)

**Proof.** To see this define \( g \) on \( \mathbf{a}_1 \underline{a}_2 \underline{a}_3 \ldots \underline{a}_k \underline{b}_0 \), where \( a_1, \ldots, a_k, b \in \mathbb{N} \) (recall recent footnote!) by
\[
g(\mathbf{a}_1 \underline{a}_2 \underline{a}_3 \ldots \underline{a}_k \underline{b}) = \mathbf{a}_1 \underline{a}_2 \underline{a}_3 \ldots \underline{a}_k \underline{a}(c+2b+1)\underline{a}(2c+1)
\]
where \( c = \sum_{i=1}^{k} a_i. \) Clearly \( g \) is polynomial time computable. Also
\[
gr(\mathbf{a}_1 \underline{a}_2 \underline{a}_3 \ldots \underline{a}_k \underline{a}(c+2b+1)\underline{a}(2c+1)) \in K N A S P A C K
\]
\[
\iff 2c+1 + \sum_{i \in S} a_i = c + 2b + 1 + \sum_{i \not\in S} a_i
\]
for some \( S \subseteq \{1, 2, \ldots, k\} \) (since \( 2c+1 \) and \( c + 2b + 1 \) cannot both be on the same side)
\[
\iff 2c+1 + \sum_{i \in S} a_i = c + 2b + 1 + c - \sum_{i \not\in S} a_i
\]
for some \( S \subseteq \{1, 2, \ldots, k\} \)
\[
\iff \sum_{i \in S} a_i = b \text{ for some } S \subseteq \{1, 2, \ldots, k\}
\]
\[
\iff \mathbf{a}_1 \underline{a}_2 \underline{a}_3 \ldots \underline{a}_k \underline{b} \in F A R E,
\]
so \( g : F A R E \subseteq_p K N A S P A C K \), as claimed.

**Proposition 17.** The ordering \( \subseteq_p \) is reflexive and transitive.

**Proof.** Reflexivity is immediate, since if \( L \subseteq A^* \) then the identity function \( \iota : A^* \to A^* \) clearly reduces \( L \) to itself in polynomial time.

To show transitivity suppose \( L_i \subseteq A_i^* \) for \( i = 1, 2, 3 \) and \( f : L_1 \subseteq_p L_2 \) and \( g : L_2 \subseteq_p L_3 \) with \( f \) and \( g \) functions computable in polynomial times \( T_f(n) \) and \( T_g(n) \) by TM’s \( M_f \) and \( M_g \) respectively. Then for \( \sigma \in A_i^* \),
\[
\sigma \in L_1 \iff f(\sigma) \in L_2 \iff g f(\sigma) \in L_3
\]
so it will suffice to show that the composition \( g f \) is computable in polynomial time.

To see how this can be done, imagine a machine \( M_{g f} \) which first runs like \( M_f \) on input \( \sigma \) to replace it with \( f(\sigma) \), and then runs like \( M_g \) to replace \( f(\sigma) \) with \( g f(\sigma) \). The first stage clearly takes \( \leq T_f(|\sigma|) \) steps, while the second takes \( \leq T_g(|f(\sigma)|) \) steps. Now clearly \( M_f \) cannot increase the length of \( \sigma \) by more than the number of steps it runs for, so we have \( |f(\sigma)| \leq |\sigma| + T_f(|\sigma|) \). Since \( T_g \) is monotonically increasing this means that \( T_g(|f(\sigma)|) \leq T_g(|\sigma| + T_f(|\sigma|)) \) and hence the total number of steps taken is bounded above by
\[
T_f(|\sigma|) + T_g(|\sigma| + T_f(|\sigma|)).
\]
Since, \( T_f \) and \( T_g \) are polynomial functions of \( |\sigma| \), as required.

\(^{17}\)Because of the simple device such as given here it is standard practice to only describe how the polynomial time reduction function is defined on inputs which already satisfy some basic form which is easily seen to be decidable in polynomial time, in this case having the form \( \mathbf{a}_1 \underline{a}_2 \underline{a}_3 \ldots \underline{a}_k \). From now on we too will often only consider the “non-trivial” part of the definition of such reduction functions.
Theorem 18. Let \( L_1 \subseteq A^*_1 \), \( L_2 \subseteq A^*_2 \) and \( L_1 \preceq_p L_2 \). Then

(i) \( L_2 \in \mathcal{P} \Rightarrow L_1 \in \mathcal{P} \).

(ii) \( L_2 \in \mathcal{NP} \Rightarrow L_1 \in \mathcal{NP} \).

(iii) \( L_2 \in \mathcal{NP}^c \Rightarrow L_1 \in \mathcal{NP}^c \).

(iv) \( L_2 \in \text{PSpace} \Rightarrow L_1 \in \text{PSpace} \).

Proof. We prove (ii), and leave the other parts as exercises.

Suppose \( f : L_1 \preceq_p L_2 \) with \( f \) computable in polynomial time \( T_f(n) \) and \( L_2 \) accepted by an NDTM \( M \) in polynomial time \( T(n) \). We devise an NDTM \( N \) working as follows. On input \( \sigma \in A^*_1 \) it first computes \( f(\sigma) \). This takes time bounded by \( T_f(|\sigma|) \) and produces an output, \( f(\sigma) \), of length at most \(|\sigma| + T_f(|\sigma|)\). Now \( N \) runs like \( M \) on \( f(\sigma) \) and accepts if and only if \( M \) does. This step takes time bounded above by

\[
T(|f(\sigma)|) \leq T(|\sigma| + T_f(|\sigma|))
\]

which is still polynomial, so \( N \) runs in polynomial time. Also,

\[
\sigma \in L_1 \iff f(\sigma) \in L_2 \iff M \text{ accepts } f(\sigma)
\]

so \( N \) accepts \( L_1 \), as required. \( \square \)

Example. We have seen that \( \text{FARE} \in \mathcal{NP} \) and that \( \text{KNAPSACK} \preceq_p \text{FARE} \) so by the Theorem we have \( \text{KNAPSACK} \in \mathcal{NP} \).

Appendix B: Landau Notation

The material in this appendix is for interest and completeness only. It does not form part of the syllabus and will not be examined.

Landau notation (also called Bachmann-Landau or big-O notation) is a way of expressing relationships between the asymptotic growth rates of different functions. We won’t use it in this course, but you’ll encounter it very quickly if you start reading more widely. So here is a very quick guide to the aspects most relevant to complexity.

Definition. Let \( f, g : \mathbb{N} \rightarrow (\mathbb{N} \setminus \{0\}) \) be functions. Then we say “\( f(n) = O(g(n)) \)” to mean that “there exist positive constants \( k \) and \( c \) such that for all \( n > k \) we have \( f(n) \leq cg(n) \)”.

Warning. The use of the equals sign (=) here is a notational convention completely distinct from its normal meaning: the statement “\( f(n) = O(g(n)) \)” does not assert that anything is equal to anything else!

In our terminology, a problem is in \( \text{Time}(g(n)) \) if and only if it admits an algorithm running in time \( f(n) \) for some function \( f(n) \) such that \( f(n) = O(g(n)) \), and similarly for space. People often abbreviate this, saying for example that a problem “solvable in \( O(n^2) \) time” or even just “in \( O(n^2) \)”, which in our terminology means the problem is in \( \text{Time}(n^2) \).

Note. The definition above is for positive-valued functions, which are all we generally need for complexity. In other areas of mathematics the notation is used with more general real-valued functions, but this requires a slightly more complex definition involving absolute values.

Note. You may also encounter similar notation with the \( O \) replaced by \( o, \theta, \omega \) or \( \Omega \). Each of these is used in a similar way to express a different relationship between the asymptotic behaviours of the two functions concerned.

Remark. You’ll also see people using Landau notation for functions of more than one variable, for example writing things like \( f(x, y) = O(x^2y^2) \). This is most often taken to mean that there are constants \( c \) and \( k \) such that for all \( x, y > k \) we have \( f(x, y) \leq cx^2y^2 \). (If you think about it, there are a couple of other, equally plausible but subtly different, things it could mean, and some authors don’t seem to be entirely clear on which they are using!)