Chapter 13

Introduction to Complexity

In our study of computability, the main question we were interested in was the existence of problems *that cannot be solved by a computer*, no matter how much time and space we have available. L_d was an example of a language that *cannot be recognised by any Turing Machine* at all. Not only do we currently don't know of a Turing Machine that accepts that languages, but such a Turing Machine cannot theoretically exist. We also found languages for which Turing machines that accept them exist, but a Turing machine that *decides* in a finite time whether a string belongs to the language or not cannot exist. In other words, there exists a Turing Machine that accepts such a language, but no Turing Machine that accepts it halts on all inputs. L_u and L_{halt}, as well as some less "artificial" languages such as L_{pcp}, were shown to belong to this class of problems. In this part, we investigate another crucial question of our theory - what are the problems that can be solved *efficiently* (or, as we will see, what are the languages that can be recognised efficiently). We will define precisely what do we mean by *efficiently* solving a problem, and this will allow us to define several very important classes of problems in terms of the complexity of their solution.

13.1 Languages and Problems

So far, we have mostly talked about *recognising languages*, rather than about *solving problems*. That is, we considered a computation where the input was a given string, and the outcome of the computation was a decision whether the string belongs to the language or not. The exception was when we talked about Turing machines as computers of integer functions. However, in computer science, we are in generally interested in problems such as "find all the prime factors of n", "find the product of the two numbers x and y" and "find the shortest path between the two nodes v_1 and v_2 in the graph G". The algorithms for solving these kind of problems *calculate* an output based on some input. The concept of a problem seems to be much more general than the concept of recognising languages or determining whether a string belongs to a language - the latter seems to be just a special case of the former. Therefore, it looks far from obvious how our study of languages translates into solving the problems.

In our discussion, we will restrict ourself to a class of problems that we will call the *decision problems*. A decision problem has a set of *inputs* as parameters and has a *yes* or *no* answer. That is, it accepts a set of values for parameters as an input, and produces "yes" or "no" as an output. Several examples of decision problems are

1. Does the binary string w has an equal number of 0's and 1's?

   Input: String w

   Answer: *yes* if w has an equal number of 0's and 1's and *no* otherwise.

2. Does a given finite automaton M accepts a given string w?

   Input: Finite automaton M and an input string w

   Answer: *yes* if M accepts w and *no* otherwise.

3. Does a given graph G contain a cycle?

   Input: Graph G

   Answer: *yes* if G contains a cycle and *no* otherwise.

An *instance* of a problem is a particular assignment of values to input parameters. This is when we assign some concrete values of input parameters. For example, one instance of the problem "Does the binary string w has an equal number of 0's and 1's" is obtained when we consider string $w = 000111$. For this concrete instance, it is possible to answer *yes* or *no*. It is very important to understand the difference between a problem and an instance of that problem. The difference is similar as the difference between a function and a call of a function in a programming language. A function is an object that maps inputs and outputs. But the concrete outputs are produced only when we call such a function with some concrete inputs.

If we find a suitable encoding of input parameters of a problem as a string over some alphabet Σ, the decision problem becomes a question of membership to a language. Exactly what constitutes a suitable encoding, and what encodings are considered unsuitable will be discussed later. At this point, we can consider encoding input parameters as binary strings of 0's and 1's as a suitable encoding of the values of parameters. Looking back at our examples of decision problems, we can see how they correspond to determining membership of strings to languages:

1. Does a given string w belong to the language

$$L = \{w | w \in \{0, 1\}, w \text{ has the same number of 0's and 1's}\}.$$

2. Does a given encoding of the finite automaton M and the string w belong to the language

$$L = \{x | x \text{ is an encoding of an automaton } M \text{ and string } w \text{ such that } w \in L(M)\}.$$

3. Does a given encoding of the graph G belong to the language

$$L = \{w | w \text{ is an encoding of graph } G \text{ such that } G \text{ has a cycle.}\}.$$

We can see that in each of the cases we have represented the instaces of a problem as strings, and define the language L as the set of all strings for which the answer to the problem is "yes". A *solution* to a decision problem is then a Turing Machine M that accepts the language L and that halts on all inputs. Our interest in this part is in how "fast" (by some definition of fast) this solution actually solves the instances of the problem (or again, in other words, how fast M makes a decision about the membership of some string w to the language L).

Who do we consider only decision problems? For one, they have very easy representation as language acceptance problems. Furthermore, they are not as restrictive as one might think. In many cases, the solution to a general problem can be expressed as a solution to a number of related decision problems. Let us look at a few examples:

- *Prime factors* problem can be seen as a problem with a parameter that is a positive integer number m, and where the output is a list of the prime factors of m. Instead of solving directly this general problem, we can look at a related decision problem:

Input: A positive integer number m and prime number p

Output: *yes* if p is a factor of m, and *no* otherwise

We can solve the general problem by using the solution of a decision problem. We simply consider all of the prime numbers p smaller than m and use the solution of the decision problem to determine whether p is a factor of m. If it is, we add it to the list of factors of m and then start over for m/p.

- *Shortest path* problem is a problem where parameters are graph G, and nodes v_1 and v_2, and the output is the shortest path in G from v_1 to v_2. The output is the shortest path in G from v_1 to v_2. Again, instead of solving this general problem, we can look at a related decision problem

Input: Graph G, nodes v_1, v_2 and a node v that is neighbour to v_1.

Output: *yes* if the shortest path in G from v_1 to v_2 contains v and *no* otherwise.

Again, we can solve the more general problem if we have a solution to the decision problem. We run this solution for each neighbour of v_1 and when the answer for one of them is *yes*, we add it to the list of nodes in the shortest path from v_1 to v_2, then start over with this new node and its neighbours.

We see that in many cases we can solve a more general problem if we solve the decision problem. A further very good feature for our examples is that if the decision problem has a "fast" solution, then the more general problem also has a "fast" solution (but this is not always the case. The last reason to consider decision problem is that if we somehow show that the decision problem is hard (in that a "fast" solution to it most likely does not exist), then we know for certain that the more general problem is at least as hard. Since our main goal in this part is to identify these hard problems, this is perfectly enough for us.

13.2 Computability and Complexity

In the previous part, we have seen that there are problems for which a solution does not exist. One example was the Post Correspondence Problem which can be represented as the decision problem:

Input: Sequences of strings $A = (w_1, \ldots, w_k)$ and $B = (x_1, \ldots, x_k)$ over the alphabet $\{0, 1\}$.

Output: *yes* if there is a sequence of integers $i_1, i_2, \ldots, i_m$ such that $w_{i_1} w_{i_2} \ldots w_{i_m} = x_{i_1} x_{i_2} \ldots x_{i_m}$, and *no* otherwise.

The next question we investigate is - do all the solvable problems have *efficient* solutions, i.e. the ones that are applicable in practice? Which is to say, that the solution does not take an impractically large amount of time. Therefore, from now on, we will consider only totally decidable problems. That is, the problems for which we can find a Turing Machine that halts on all inputs and produces a correct *yes* or *no* answer for any input instance. Or, in other words, accepts the language L of encodings of all input parameters for which the answer to the problem is *yes*.

In the complexity theory, we look at what problems can be solved in a reasonable amount of time and/or that require a reasonable amount of space. That is, if we have a Turing Machine that produces a *yes* or *no* answer to any instance of the problem, how much time or space does it take to reach this answer? Is it impracticably long time or impractically large volume of space? The main distinction will be between "easy" and "hard" problems, where the easy problems are those for which fast solutions exist, and the hard problems are the ones for which we do not know any fast solution, nor do we know whether the fast solution even exists.

It may sound strange to use Turing Machines to study complexity, because Turing Machines are grossly inefficient and they also have an infinite amount of space. We have seen, for example, that the TM for subtraction of two integer numbers is not trivial at all, and this is the operation we take for granted in algorithms to be executed almost immediately. However, it turns out that the modern computers are "only" (many) orders of magnitude faster than TM's and have lots of memory. The definition of "easy" and "hard" problems that we will use is robust enough that the same problems are classified as hard regardless of whether we look at solutions in the form of Turing Machines or in the form of computer programs. That is, the definition of easy or hard problems will be independent of the constant factors that impact computation time. So the fact that the computer is, for example, 100,000,000,000 times faster than a Turing Machine will not impact what problem is classified as "hard". The problem that is hard for a Turing Machine will also be hard for a computer.

13.3 Time and Space Complexity of Turing Machines

Complexity can be considered with respect to any metric, but it is most common to consider it with respect to time it takes to complete a computation or the amount of space required for this completion. *Time* is measured as the number of *steps* required for a Turing Machine to decide on a particular instance of the problem. One step consists of replacing a symbol which the tape head is scanning, moving the tape head one cell left or right on the tape and changing the state. *Space* is the number of different cells visited by a tape head during the computation of a Turing Machine on a particular instance of the problem.

We will focus our study on time, but it is important to know that many considerations will be generic and independent of what metric do we use, provided the metric is well defined in terms of Turing Machines. Furthermore, it is obvious that the space required for a Turing Machine to decide on a particular instance of the problem is at most the number of steps it takes to decide that instance, since at most one cell is visited in each step. Therefore, if we derive an upper bound for some instance on some Turing Machine, the same upper bound will also hold for space.

Let us assume that we have a Turing Machine M that solves a given decision problem. Will try to answer the questions such as *what is the number of steps that this Turing Machine takes to compute an answer to a*

particular instance of a problem, *what is the number of step that it takes to compute an answer to an instance of a problem of size n* in the worst case and *how do we classify problems based on the time it takes for any Turing Machine to solve their instances*.

13.3.1 Time Complexity

Let us start with an example of a Turing Machine $M = (Q, \Sigma, \Gamma, \delta, q_0, B, F)$ that accepts the language $L = \{0^n 1^n | n \geq 0\}$. M is given with

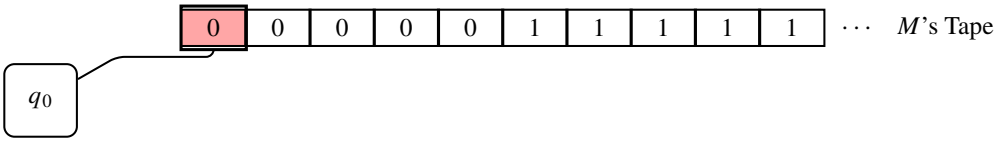
- $Q = \{q_0, q_1, q_2, q_3, q_4\}$,
- $\Sigma = \{0, 1\}$,
- $\Gamma = \{0, 1, X, Y, B\}$,
- $F = \{q_4\}$.

δ	q_0	q_1	q_2	q_3	q_4
0	(q_1, X, R)	$(q_1, 0, R)$	$(q_2, 0, L)$	–	–
1	–	(q_2, Y, L)	–	–	–
X	–	–	(q_0, X, R)	–	–
Y	(q_3, Y, R)	(q_1, Y, R)	(q_2, Y, L)	(q_3, Y, R)	–
B	–	–	–	(q_4, B, R)	–

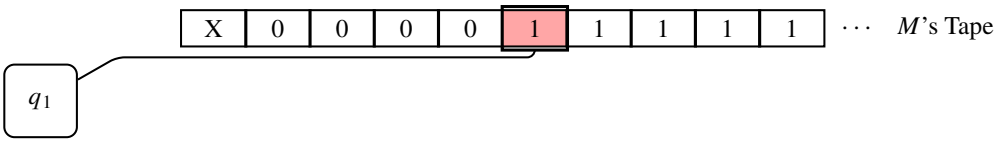
As we have seen, this Turing Machine works by replacing the first 0 on the left by X, then searching to the right for a 1 and replacing it by Y, going back left to find the first 0 and then repeating this process. The states serve the following purpose:

- q_0 is the state where we found the leftmost 0 or where we determined that there are no more 0's before the first 1.
- In q_1 we look for the first 1, going over all 0's and Y's. When we find the first 1, we replaced it by Y and go left.
- In q_2, we find the last X that replaced 0 and go to the right of it, looking for the next 0.
- In q_3 we found out that there are no more 0's, so we go over the string, checking that all of the 1's have been replaced by Y's. If we reach the B symbol, the end of the input, we halt and accept. Otherwise, if we reach another 0 or 1, we halt without rejecting.
- q_4 is the accepting state.

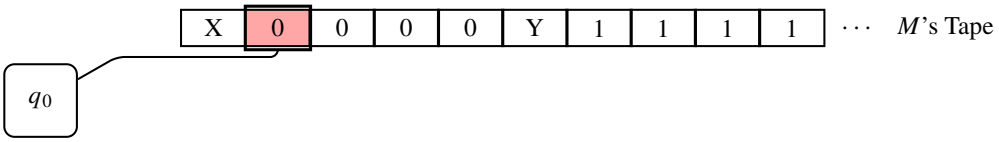
M obviously halts on all inputs and accepts the language $0^n 1^n$. Let us determine how many steps does M take to decide on input 0000011111. The starting configuration is



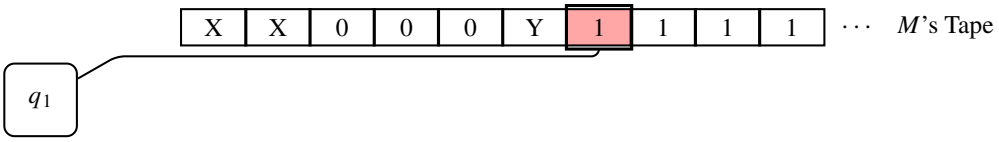
Replacing the first 0 by X and going over all the following 0's until the first 1 is reached takes 5 steps. We reach the configuration



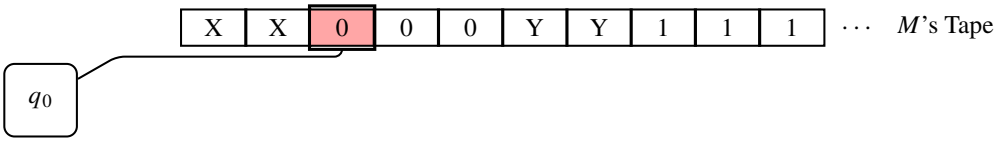
Replacing the 1 scanned by the tape head by Y, going to the left over all the 0's until X is found, then moving the tape head one cell to the right takes 6 steps. The total number of steps taken so far is 11 and we reach the configuration



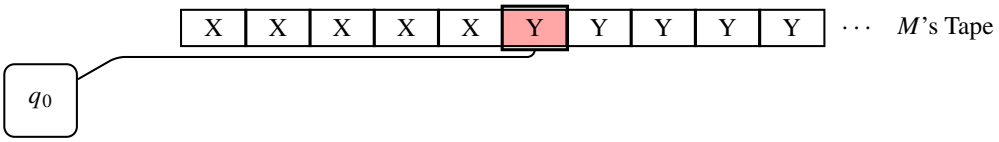
Replacing the next 0 by X and going over all the 0's and Y's until we reach the first 1 takes another 5 steps. The total number of steps taken so far is 16 and the configuration reached is



Replacing the next 1 by Y, going to the left over all Y's and 0's until the first X is reached, and then going one cell to the right takes another 6 steps. The number of steps taken so far is 22 and the configuration reached is



We can see that it takes 11 steps to match one 0 with one 1 and move the tape head to the next 0. Since there are 6 0's to match, matching all the 0's with 1's will take $5 \cdot 11 = 55$ steps. The configuration reached is



The computation ends by going over all the Y's in the state q_3 to make sure no 1's are left on the tape, once B is reached, M moves to the accepting state and the computation ends. This takes further 5 steps. Therefore, the total number of steps is $5 \cdot 11 + 5 = 60$. This can also be written as $5 \cdot (5 + 6) + 5$, if we split the 11 steps into 5 steps it takes to replace a 0 with X, go over all 0's and Y's to reach the first 1, and 6 steps to replace the first 1 with Y, go left over all the Y's and 0's until X is reached, and then go one cell to the right.

What about the input $0^6 1^6$? We can easily see, using the same reasoning, that the number of steps taken to decide on this string is $6 \cdots (6 + 7) + 6$.

It is obviously very impractical to consider the time for every instance of the problem. For one, most likely there will be infinitely many instances of the problem and it is unclear what the raw numbers for time would even tell us. Is the Turing Machine from the previous example efficient or not? We don't know. So, let us instead look at a more general question: What is the number of steps that it takes to solve an instance of a problem of size n? That is, represent the time it takes to solve an instance of the problem as a function of the size of the instance n. This is a much more useful measure, as we can see how the amount of time required for solving an instance of the problem changes as the instance size increases. This can be seen from the rate of growth of the function for time based on the size of the input instance. But this immediately gives rise to two questions:

- *What do we mean by "input instance of size n"?* Is there a precise way to define this?
- *Different problems of size n can take different amount of time to solve. How do we deal with this?* For any reasonable definition of the input instance size, there will be many different instances of that size, and for each of these sizes the Turing Machine can take a different number of steps. So, we will have many different times for the same input size.