1 Motivation

Complexity Theory studies the possibilities and limits of efficient computation. Suppose there is a computational problem P we wish to solve. The first step is to design an algorithm A for P, which we can then implement in a computer program. However, not all algorithms are created equal. We might find that our algorithm takes a really long time to produce an answer on even reasonably-sized inputs, or that it exhausts the memory available to it.

The first motivation for complexity theory is that we'd like to *quantify* the performance of an algorithm, in terms of its computational resource requirements. Given our problem-solving needs and the computational resources available to us, we will have an idea of what a "reasonable level of performance" is, and we can then focus our attention on algorithms which achieve this level of performance.

This gives some motivation for the analysis of algorithms, but computational complexity is far wider in its scope. Suppose we are unable to design an efficient algorithm A for the problem P. There could be different explanations for this. Such an algorithm A might exist, but we might just not be clever enough to find it. Alternatively, such an algorithm might not exist even in principle, i.e., the problem might be *intractable*.

Clearly, it would be very useful to know which problems are efficiently solvable and which ones are intractable. Complexity theory assists us in clarifying the difficulty of solving problems. It does this in different ways - eg., unconditionally showing that a problem cannot be solved efficiently in a certain model, or showing that a wide variety of problems are equivalent in terms of their difficulty, meaning that solving one of them is tantamount to solving all of them. As much as for its theorems, complexity theory is useful in giving us a language and an arsenal of concepts for talking about the difficulty of problem-solving.

2 Languages and Machines

In complexity theory, we try to understand in an abstract setting the resource requirements of computational problems on a computational model. In order to do this, we need to formalize what we mean by "computational problem" and by "computational model".

2.1 Computational Problems: Languages

In terms of problems, we focus our attention on *decision problems*, i.e., problems with a YES/NO answer. Such problems are modelled as languages, i.e., as sets of strings over a finite alphabet, which is usually $\{0, 1\}$. Here's the equivalence between decision problems and languages: the membership of a string w in a language is interpreted as a YES answer on input w for the corresponding decision problem. One might ask why we only consider inputs that are strings, given the wide variety of types of data encountered in practice - numbers, matrices, graphs etc. The reason is that in complexity theory, the specific representation of data is irrelevant, and most natural kinds of data can be represented as strings using a natural encoding.

The restriction to YES/NO outputs is again a way of simplifying the setting without losing generality. A large part of the theory we derive for decision problems can be translated easily to the more general setting. One of the great virtues of complexity theory is that the main ideas and principles are flexible and capable of adaptation to a variety of settings.

2.2 Computational Model: Turing Machines

In terms of computational models, we study the model of the multi-tape Turing machine. Formally, a k-tape Turing machine is a tuple $(Q, \Sigma, \Gamma, \delta, q_0, q_f)$, where:

- Q is a finite set of states.
- Σ is a finite input alphabet.
- $\Gamma \supseteq \Sigma \cup B$ is a finite tape alphabet, where B is a specially designated blank symbol.
- $\delta : Q \times \Gamma^k \rightarrow Q \times \Gamma^k \times \{L, R, S\}$ is the transition function. The transition function may be undefined on some inputs.
- $q_0 \in Q$ is the start state.
- $q_f \in Q$ is the accepting state.

We now describe how this formalism is interpreted. A k-tape Turing machine consists of a finite control together with k linear tapes, divided into equal-sized tape cells. Each tape is infinite to the right, and has an attached tape head positioned at one of the tape cells. One of these tapes is a read-only input tape, and the others are read/write. At the beginning of the computation, all the tapes except the input tape are blank, meaning that each tape cell contains the blank symbol. The input tape has the input w written on the first $|w|$ cells, with the remaining cells being blank. For each tape, the tape head points to the leftmost tape cell.

At each step in the computation, the finite control is in some state $q \in Q$, with the state at the beginning being q_0. The computation evolves as follows.At time t, assume the finite control is in state q_t and the symbols being read on the k read/write tapes are $a_1, a_2 \ldots a_k \in \Gamma$. Let $\delta(q_t, a_1, a_2 \ldots a_k) = (q', a'_1, a'_2 \ldots a'_k, b_1, b_2 \ldots b_k)$, where each $b_i \in L, R, S$. Then the state q_{t+1} of the finite control at time $t + 1$ is q', and for each $i, 1 \leq i \leq k$, the symbol a_i is overwritten by a'_i on the i'th tape. Moreover, for each i, depending on whether $b_i = L, R$ or S, the tape head on the i'th tape either moves one place left, one place right, or remains stationary.

Starting at time 0, the computation continues to evolve in this way until one of two things happens. If the finite control attains state q_f at any point, the computation halts and the Turing machine accepts. If at any time t, $\delta(q_t, a_1, a_2 \dots a_k)$ is undefined, the computation halts and the Turing machine rejects.

We say the Turing machine M accepts an input w if it halts and accepts on w, otherwise we say it rejects on w. We say that a Turing machine M decides a language $L \subseteq \Sigma^*$ if for each $w \in \Sigma^*$, $w \in L$ iff M accepts w.

On occasion, the k-tape Turing machine is equipped with an extra write-only tape called the output tape. A Turing machine of this kind is called a *transducer*. Just as regular Turing machines decide languages, transducers compute functions from Σ^* to Γ^*. The output of a transducer M on input x is the string y that is present on the output tape when M halts. Note that, with transducers, the distinction between acceptance and rejection is no longer relevant. Note also that a transducer might not always halt on every input, in which case it computes a partial function.

We will require a standard notation for configurations of a Turing machine. A configuration C is of the form $(q, i, (u_1, v_1), (u_2, v_2) \dots (u_{k-1}, v_{k-1}))$, where $q \in Q$, $i \in \mathbb{N}$ and for each $i, 1 \leq i \leq k-1$, $u_i, v_i \in \Gamma^*$. This is interpreted to mean that the Turing machine is in state q, the input head is reading the i'th cell on the input tape, and for each $i, 1 \leq i \leq k-1$, the contents of the $i+1$'th tape are $u_i v_i$, with the tape head reading the $|u_i|+1$'th cell. A configuration C' succeeds a configuration C if the Turing machine moves in one time step from configuration C to C'. A computation of a Turing machine is a sequence of configurations $C_0, C_1 \dots C_T$ such that C_0 is the initial configuration, C_T is a halting configuration, and for each $i, 1 \leq i \leq T$, configuration C_i succeeds C_{i-1}.

2.3 Why Turing Machines?

The choice of the multi-tape Turing machine as our model of computation might seem rather arbitrary, as it does not seem to correspond to any model of computation we encounter in the real world. However, the multi-tape Turing machine is in fact *universal*, in the sense that any "reasonable" abstract model of computation we might define (where "reasonable" means that the model is finitistic, with computation proceeding according to local rules) is simulated by the Turing machine. This is not a theorem, but rather a widely believed empirical fact, known as the "Church-Turing thesis". A stronger version of this, known as the "Polynomial-time Church-Turing thesis" appears to be true, and this is particularly relevant to complexity theory. The Polynomial-time Church-Turing thesis states that any reasonable model of computation can be simulated *efficiently* by a Turing machine, where here "efficient" means that there is at most a polynomial slowdown in time. Thus the notion of efficient computation is robust with respect to the choice of model.

In this context, the multi-tape Turing machine is a good model for two reasons. First, different computational resources such as time, space, non-determinism and randomness have natural interpretations in this model (or in minor variants of this model). Second, while it can be complicated and tedious to design Turing machines to solve specific problems (somewhat akin to writing assembly language code), the conceptual simplicity of the model makes it easier to prove theorems about. There is a trade-off between the ease of problem-solving in a model and the ease of proving theorems about the model. Models with lots of features are easier to program in, but simpler, more limited models are more convenient to study theoretically.

You should not take the Polynomial-time Church-Turing thesis on faith. I encourage you to dream up other models of computation, or to take existing ones such as register machines or your favorite programming language, and verify that they can be simulated efficiently on multi-tape Turing machines.