

Lecture 1. Our Problems, Our Model of Computation, Complexity Measures, and Complexity Classes

We start by introducing a framework that has been used in computational complexity theory.

1.1 What sort of problems are we going to discuss?

In computational complexity theory, we consider only problems that are completely and formally specified. But even among mathematically specified problems, a problem like “Determine whether Riemann’s hypothesis is correct or not” is not appropriate for discussing its computational difficulty. Instead, we consider a problem asking for an answer to a given *instance* of the problem, i.e., a problem of computing a formally specified function on a given input. For example, a problem is specified in the following way.

Max. Clique Problem

Instance: An undirected graph G .

Question: What is the size of the largest clique (i.e., complete subgraph) of G ?

Several points need to be clarified here.

1. Input instances and outputs are encoded appropriately as binary strings, i.e., finite strings over the alphabet $\{0, 1\}$.
2. For each problem, we may naturally define “input size”, a parameter that determines the size of an input instance. In our *formal* discussion, the *size* of an input instance is simply the length of binary string coding the instance. Throughout this course, we will use ℓ to denote this formal input size, while more intuitive input size (if it exists) will be denoted by n .
3. Among computational problems like the above, we will usually consider only “decision problems.” A *decision problem* is a problem to decide whether a given instance is ‘yes’ (or simply 1) or ‘no’ (simply 0). For example, instead of the problem mentioned above we can consider the following decision version.

Max. Clique Problem (decision version; abbreviated as CLIQUE)

Instance: A pair $\langle G, k \rangle$ of an undirected graph G and an integer $k \geq 1$.

Question: Does G have a clique of size $\geq k$?

We will see that (for most problems) their computational difficulty would not change so much by considering (appropriately defined) decision problem versions. For a decision problem, any binary string can be an input instance. For example, a binary string that does not represent $\langle G, k \rangle$ should be also regarded as an input instance, which should be answered ‘no’ (i.e., 0).

In summary, a problem (we will usually consider in this course) is a decision problem that is nothing but computing a function from $\{0, 1\}^*$ to $\{0, 1\}$. We are interested in an algorithm that computes such a function for *all* input instances, i.e., for *all* binary strings.

We will often denote this function by the name of the problem; e.g., $\text{CLIQUE}(\langle G, k \rangle) = 1$ or $= 0$.

1.2 A model of computation: How shall we express algorithms?

The notion of algorithm is rather abstract, and we need to consider a concrete “computational mechanism” for discussing the efficiency of a given algorithm. In fact, in order to use an algorithm A for solving a given problem, we need to implement it as a program P and run it on a computer X . In this case, the computational mechanism corresponding to the algorithm A is the computer X executing the program P . How shall we express such a computational mechanism? Throughout this course, we will use the standard “one-tape Turing machine” as our formal model of computational mechanism.

A Turing machine is quite mathematical and abstract; nevertheless, it is also a reasonable model for discussing computational complexity of algorithms. We first clarify this point.

Let us begin with the following question: What is the basis of computation in computer science that is like atoms in physics? There are several choices for a basis. Here we consider the following two sets of operations, and convince ourselves that each of them is regarded as a basis of computation.

- (A) A basis based on integer calculations by using variables for nonnegative integers¹.
 - (1) An assignment operation with (at most) single addition/subtraction over variables and/or constants on its right hand side, e.g., `a = b + c`, `d = 241`, etc.
 - (2) A conditional branch based on whether the value of a specified variable is 0 or not, e.g., `if(a == 0) { ... } else { ... }`, etc.
 - (3) An iteration until the value of a specified variable becomes 0, e.g., `while(a != 0) { ... }`, etc.
- (B) A basis based on binary string manipulations by using variables for binary strings.
 - (1) An assignment operation with basic string operations such as `car(x)`, `cdr(x)`, `concat(x,y)` on its right hand side, e.g., `a = concat(b,c)`, `b = car(a)`, `c = cdr(a)`,
`d = "0100"`, etc.
 - (2) A conditional branch based on whether the value of a specified pair of variables have the same string, e.g., `if(a == b) { ... } else { ... }`, etc.
 - (3) An iteration until the value of a specified variable becomes the empty string, e.g., `while(a != "") { ... }`, etc.

To see, for example, that the set (A) of operations is enough to express any sort of computation, let us consider any program written in programming language C. Precisely speaking, it is a part of C program defining some function F mapping from the set of nonnegative integers to $\{0, 1\}$ that solves a given decision problem. To be concrete, let us assume that x is the input variable of F , and that F locally uses three variables a , b , y of `int` type, and that the return value is always computed at variable y . Then it is not

¹Any binary string can be expressed as a nonnegative integer by adding ‘1’ in its front; then it may not be so hard to imagine that nonnegative integers are enough to express all sorts of data and that the `int` type is enough for the basic data type. We will see at the next lecture how to handle arrays.

so hard to see that the program can be restated as a program with a simple structure of Figure 1.1. We will refer this simple structure as the *standard form*.

```

int F(int x) {
    int a, b, y;
    int line;
    while line != 0 {
        switch (line) {
            case 1:  a = b + x;  line = 3;  break;
            case 2:  if(a == 0) { y = 0; pc = 0; } else { line = 4; }  break;
            :
            case K:  b = a - x;  line = 1;  break;
        }
    }
    return y;
}

```

Figure 1.1 A standard form C program

In this standard form, each case statement consists of an operation of type (1) or (2) followed by an assignment to `line` like “`line = 3;`”, which determines the next statement executed. Thus, it is clear from this observation that (A) is enough (and in fact, an operation of type (3) is used only once). Furthermore, we can also imagine that its computation time is not so large (say, at most within 1000 times) compared with the original one. Then for a program of such a simple structure, it is not so difficult to see that a Turing machine (defined in the next) can simulate its computation without reducing its efficiency so much.

1.3 A Turing machine

We follow [1] and use one-tape Turing machines as our model of computation. The following definition is from [1:Def. 3.3].

Definition 1.1 A *one-tape Turing machine* \mathbb{M} is a seven tuple $(Q, \Sigma, \Gamma, \delta, q_0, q_{\text{accept}}, q_{\text{reject}})$, where

- Q is the finite set of states,
- Σ is the input alphabet (which we assume $\{0, 1\}$ throughout this course),
- Γ is the finite tape alphabet, where $\Sigma \subseteq \Gamma$ and $\sqcup \in \Gamma$,
- δ is the transition function mapping $Q \times \Gamma$ to $Q \times (\Gamma \setminus \{\sqcup\}) \times \{L, R\}$,
- $q_0 \in Q$ is the *start state*,
- $q_{\text{accept}} \in Q$ is the *accept state*, and
- $q_{\text{reject}} \in Q$ is the *rejecting state*.

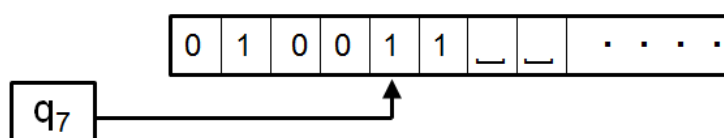


Figure 1.2 A snapshot of the execution of a Turing machine

Here we omit the explanation on this Turing machine model, which can be found in [1] (Chapter 3) except for the following very basic points. Consider any Turing machine M . Its execution starts with (i) an input instance x (encoded as a binary string) on its tape, (ii) its tape head looking at the leftmost cell, and (iii) its state being the initial state q_0 . Then the machine changes its state, the tape head position, a symbol on a cell that the tape head currently looking at following the transition function δ . This change is regarded as *one move* during the execution of M . Note that if the tape head looks at some cell with the blank symbol \sqcup , this blank symbol must be changed to another symbol by the next move. We say that a cell is *used* if it has a symbol rather than \sqcup . The execution of M terminates when its state becomes either the accept state or the reject state. We say that M *accepts* (resp., *rejects*) input x if it terminates with the accept state (resp., rejecting state). We also say that M *outputs* 1 (resp., 0) on x (write as $M(x) = 1$ (resp., $M(x) = 0$)) if it terminates with the accept state (resp., reject state). The notation “ $M(x)$ ” will be used also as an abbreviation of “the computation of M on input x .”

In order to discuss the computational efficiency of M , we introduce the following two functions:

$$\begin{aligned} \text{time}_M(x) &= \text{the number of moves of } M \text{ on } x \text{ until it terminates, and} \\ \text{space}_M(x) &= \text{the number of tape cells that } M \text{ uses on } x. \end{aligned}$$

Note that $\text{time}_M(x)$ is undefined if the execution of M on x does not halt.

For understanding the connection to standard form programs more clearly, it may be better to consider multi-tape Turing machines. A *multi-tape Turing machine* is almost the same as a one-tape Turing machine except that it can use more than one tape. Note that one can use any number of tapes, but that the number of tapes must be fixed for each Turing machine.

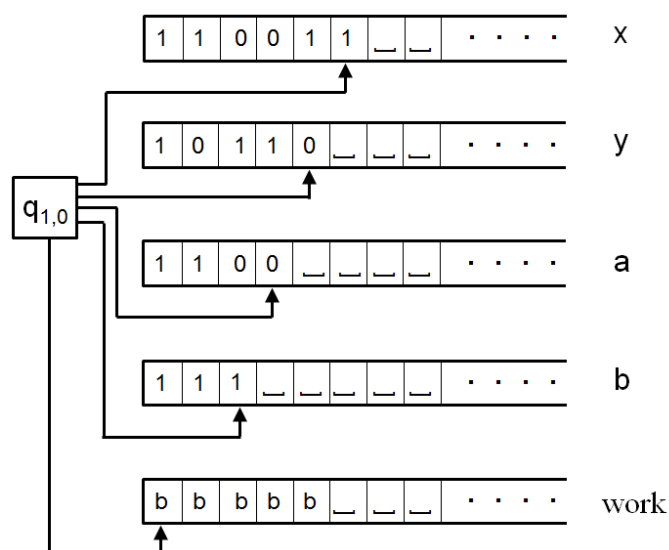


Figure 1.3 A snapshot of the execution of a multi-tape Turing machine

By using this multi-tape Turing machine model, we can naturally simulate a standard form program. For example, for the standard form program F of Figure 1.1, we may use the above Turing machine. It has a tape corresponding to each variable of F for keeping its current value, and it also has one (or maybe two) work tape for computing a required

arithmetic operation at each statement. For each value s of variable **line** of F , the Turing machine has states $q_{s,i}$ for $i \in \{0, 1, \dots, k_s\}$, and each s th statement of F is simulated by starting from the state $q_{s,0}$. Roughly speaking, it is enough to use some $c_1 \ell_*$ moves to simulate one statement, where ℓ_* is the max. number of bits stored in the variables of F and c_1 is a constant independent from the choice of standard form programs.

For the relation between the power of one-tape and multi-tape Turing machines, we have the following theorem. (A proof idea, which is slightly different from the one in [1], is illustrated in Figure 1.4, and it is explained in the class.)

Theorem 1.1 There exists some constant $c_{1.1}$ with the following property. For any multi-tape Turing machine M , we can construct a one-tape Turing machine M' that simulates M and satisfies the following for any input x :

$$\text{time}_{M'}(x) \leq c_{1.1}(\text{time}_M(x))^2 \text{ and } \text{space}_{M'}(x) \leq c_{1.1}\text{space}_M(x).$$

Remark: For any Turing machines M and M' , we say that M' *simulates* M if M computes the same function as M .

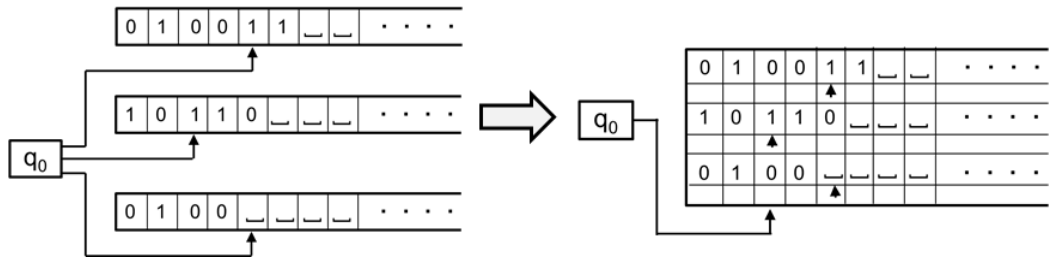


Figure 1.4 An idea of the simulation

By using the method outlined above, we cannot improve the squared time bound. Interestingly, we have a much more efficient simulation by using two-tapes.

Theorem 1.2 There exists some constant $c_{1.2}$ with the following property. For any multi-tape Turing machine M , we can construct a two-tape Turing machine M' that simulates M and satisfies the following for any input x :

$$\text{time}_{M'}(x) \leq c_{1.2}\text{time}_M(x) \log(\text{time}_M(x)) \text{ and } \text{space}_{M'}(x) \leq c_{1.2}\text{space}_M(x).$$

Because of this difference, in computational complexity theory the two-tape Turing machine model is usually used. But for simplicity we will follow [1] and use the one-tape Turing machine model. Below by “Turing machine” we mean one-tape Turing machines.

1.4 Complexity measures and basic complexity classes

Now we are ready to define complexity measures. In general a complexity measure is a measure for evaluating the computational efficiency of a given Turing machine M , and it is a function mapping an input size ℓ to the computational cost of M on “some” input of size ℓ . In this course, we mainly consider the following “worst-case” complexity measures defined by worst-case inputs [1:Def. 7.1].

Definition 1.2 For any M , its time and space complexity measures are defined as follows:

$$\begin{aligned}\text{time}_M(\ell) &= \max_{x \in \{0,1\}^\ell} \text{time}_M(x), \text{ and} \\ \text{space}_M(\ell) &= \max_{x \in \{0,1\}^\ell} \text{space}_M(x).\end{aligned}$$

We want to compare the efficiency of algorithms, and we need to compare, e.g., the time complexity measures of two Turing machines M_1 and M_2 . Note that complexity measures are functions from \mathcal{Z}^+ to \mathcal{Z}^+ , where \mathcal{Z}^+ is the set of nonnegative integers. Here we use the following way to compare two functions on \mathcal{Z}^+ [1:Def. 7.2].

Definition 1.3 For any functions $f, g : \mathcal{Z}^+ \rightarrow \mathcal{Z}^+$, we write $f = O(g)$ (or more specifically, write $f(n) = O(g(n))$ with an argument n) if there exist integer $c > 0$ and $n_0 \geq 0$ such that

$$\forall n \geq n_0 [f(n) \leq cg(n)].$$

For a Turing machine M , if $\text{time}_M(\ell) = O(t(\ell))$ (resp., $\text{space}_M(\ell) = O(s(\ell))$), then we say that M runs in $O(t(\ell))$ -time (resp., in $O(s(\ell))$ -space) and also call M an $O(t(\ell))$ -time Turing machine (resp., an $O(s(\ell))$ -space Turing machine).

Although complexity measures are for evaluating the efficiency of an algorithm represented by a Turing machine, we may use them to estimate the easiness/hardness of a given problem in the following way.

Definition 1.4

- (1) For a given problem L , we say that a Turing machine M *solves* L if $M(x) = L(x)$ for all strings $x \in \{0,1\}^*$.
- (2) For any function $t : \mathcal{Z}^+ \rightarrow \mathcal{Z}^+$, we say that L is $O(t(\ell))$ -time solvable if there exists an $O(t(\ell))$ -time Turing machine solving it. Similarly, for any function $s : \mathcal{Z}^+ \rightarrow \mathcal{Z}^+$, we say that L is $O(s(\ell))$ -space solvable if there exists an $O(s(\ell))$ -space Turing machine solving it.

Finally, define the following basic complexity classes [1:Def. 7.7, Def. 8.2].

Definition 1.5 For any $t, s : \mathcal{Z}^+ \rightarrow \mathcal{Z}^+$, the time and space complexity classes, $\text{TIME}(t(\ell))$ and $\text{SPACE}(s(\ell))$, are defined as follows.

$$\begin{aligned}\text{TIME}(t(\ell)) &= \{ L \mid L \text{ is an } O(t(\ell))\text{-time solvable problem} \}. \\ \text{SPACE}(s(\ell)) &= \{ L \mid L \text{ is an } O(s(\ell))\text{-space solvable problem} \}.\end{aligned}$$

Reference:

1. M. Sipser, Introduction to the Theory of Computation (2nd edition), Thomson Course Technology, 2006, ISBN:0-534-95097-3.