Lecture 2. More on Computation Models

We introduce other important computation models.

2.1 Boolean circuits and circuit complexity (Only briefly explained today)

We use the following definition, which is slightly modified from [1:Def. 9.21].

Definition 2.1 A *Boolean circuit* (in short, circuit) is a collection of *gates* and *nodes* connected by *wires*. Cycles are not permitted. Computing gates take three forms: AND gates, OR gates, and NOT gates. Each computing gate take values from *input nodes* and yields a value to its *output node* which could be used as an *input node* of the other gate. An *input gate* is an input node that is not the output node of any computing gate. An *output gate* is an output node that is not the input node of any computing gate.

The following is an example of circuits. See the explanation after [1:Def. 9.21] for the way to compute a value when Boolean values are given to input gates x and y. (In this course note, circuits are drawn upside-down compared with [1], with input gates at the bottom

x	y	$\oplus$
0	0	0
0	1	1
1	0	1
1	1	0

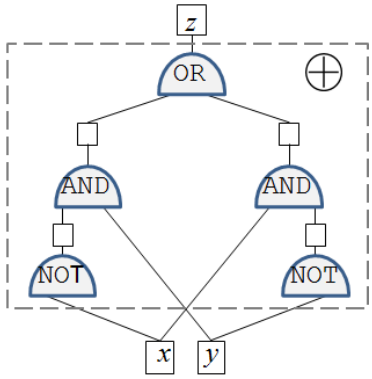


Figure 2.1 A circuit computing the XOR function $\oplus$

In general we may consider a circuit as a device for computing a fixed length binary string from a fixed length binary string. For example, the circuit $\mathtt{C}$ of Figure 2.2 is regarded as a device computing a 3 bit binary string $z = z_1z_2z_2z_3$ from 6 bit binary string $xy = x_1x_2x_3y_1y_2y_3$, which we denote as $\mathtt{C}(xy) = z$.

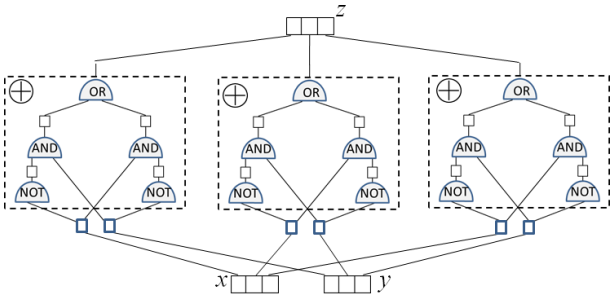


Figure 2.2 A circuit computing the bit-wise XOR function (for 3 bit strings)

Since each circuit has fixed number of input gates, it can handle only fixed size binary strings. Therefore, we use a family $\{\mathtt{C}_\ell\}_{\ell\geq 1}$ of circuits for computing a given function f, where each $\mathtt{C}_\ell$ is a circuit for computing size ℓ input strings (see also [1:Def. 9.27]).

Definition 2.2 For a given problem L, we say that a circuit family $\{\mathtt{C}_\ell\}_{\ell\geq 1}$ *computes* L if for every $\ell \geq 1$, we have $\mathtt{C}_\ell(x) = L(x)$ for all $x \in \{0,1\}^\ell$.
Remark: Recall that problems we consider in this course are decision problems over $\{0,1\}$, that is, problems of computing Boolean functions on $\{0,1\}^*$.

The *size* of a circuit is the number of computing gates of the circuit, which is the main computational cost of a circuit. Note that for a given problem L, it is possible to determine a *minimal size circuit family* as follows, which allows us to define the *circuit size complexity measure for the problem* L directly.

Definition 2.3 Let L be any problem.
(1) A *minimal circuit family* $\{\mathtt{C}_\ell\}_{\ell\geq 1}$ *for* L is a circuit family $\{\mathtt{C}_\ell\}_{\ell\geq 1}$ such that (i) it computes L, and (ii) for every $\ell \geq 1$, the size of $\mathtt{C}_\ell$ is the smallest among all circuits computing L on $\{0,1\}^\ell$.
(2) Let $\mathtt{C} = \{\mathtt{C}_\ell\}_{\ell\geq 1}$ be a minimal circuit family for L. Then the circuit size complexity of L is defined by

$$\mathrm{size}_L(\ell) \;=\; \text{the size of } \mathtt{C}_\ell.$$

Remark: (i) There is no notion of "worst-case instance" when discussing the circuit size. (ii) For any circuit family $\mathtt{C}$, we would also use $\mathrm{size}_{\mathtt{C}}(\ell)$ to denote the size of the circuit $\mathtt{C}_\ell$.

One of the important features of circuits is that it can computes values in parallel. The parallel computation time is measured by the *depth* of a circuit, that is, the number of computing gates on the longest path from its input gate to its output gate. We define $\mathrm{depth}_L(\ell)$ and $\mathrm{depth}_{\mathtt{C}}(\ell)$ in the same way as size.

Now we define the following basic circuit complexity classes.

Definition 2.4 For any $s, d : \mathcal{Z}^+ \to \mathcal{Z}^+$, the circuit size and size&depth complexity classes, SIZE($s(\ell)$) and SIZE&DEPTH($s(\ell), d(\ell)$), are defined as follows.

$$\mathrm{SIZE}(s(\ell)) \;=\; \{\, L \,|\, \mathrm{size}_L(\ell) = O(t(\ell)) \,\}.$$
$$\mathrm{SIZE\&DEPTH}(s(\ell), d(\ell)) \;=\; \{\, L \,|\, \mathrm{size}_L(\ell) = O(t(\ell)) \text{ and } \mathrm{depth}_L(\ell) = O(d(\ell)) \,\}.$$

2.2 Circuits and Turing machines

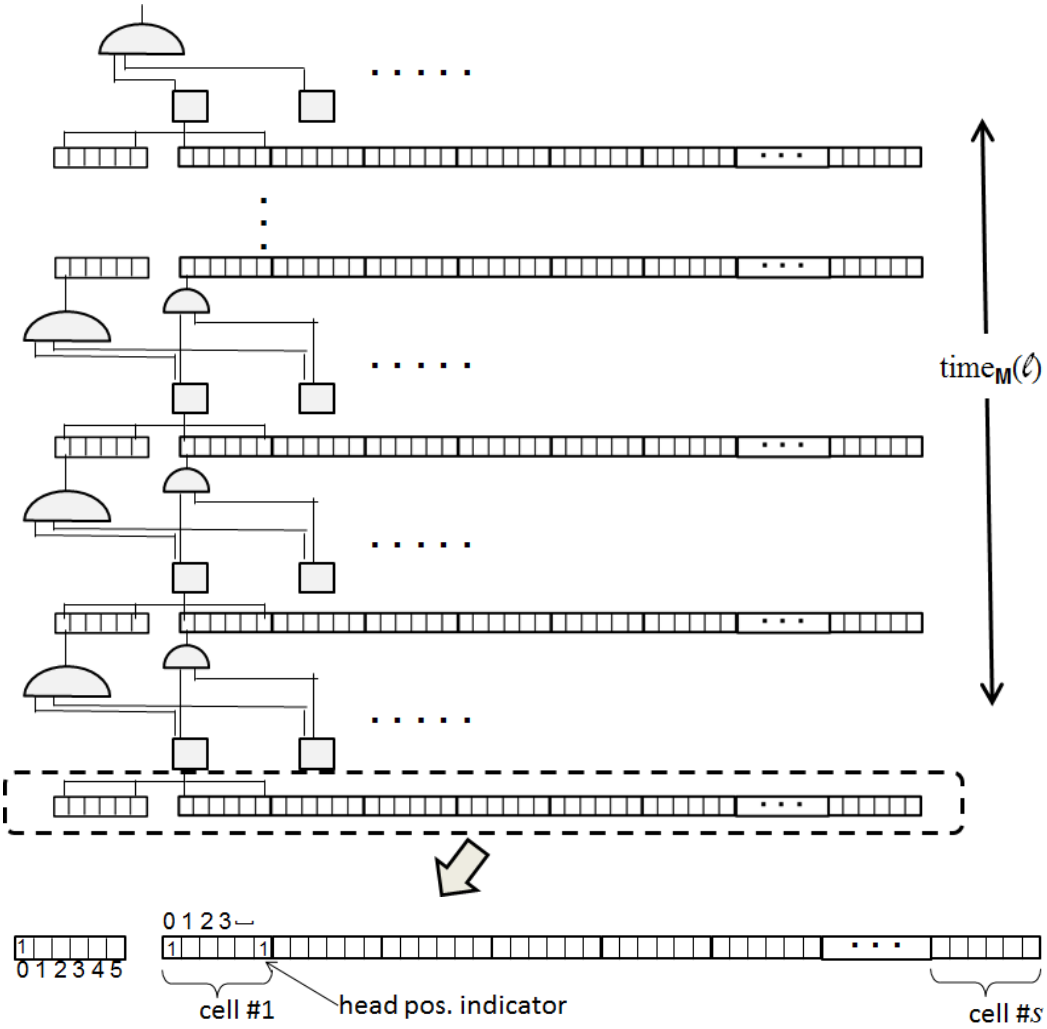
We consider the relation between circuits and Turing machines. First we show a relatively easy way to design a circuit family for simulating a given Turing machine. (Recall that we are using one-tape Turing machines for our computation model.)

Theorem 2.1 There exists some constant $c_{2.1}$ with the following property: For any Turing machine $\mathtt{M} = (Q, \Sigma, \Gamma, \delta, \mathtt{q}_0, \mathtt{q}_{\mathrm{accept}}, \mathtt{q}_{reject})$, we can construct a class of circuits $\mathtt{C} = \{\mathtt{C}_\ell\}_{\ell\geq 1}$ that simulates $\mathtt{M}$ and satisfies the following for any input x of length ℓ:

$$\mathrm{size}_{\mathtt{C}}(\ell) \;\leq\; c_{2.1} c_{\mathtt{M}} (\mathrm{time}_{\mathtt{M}}(\ell))^2,$$

where $c_{\mathtt{M}}$ is $(|Q| + |\Gamma|) \cdot |Q| \cdot |\Gamma|$.
Remark: For any finite set, we use $|A|$ to denote the number of elements in the set A. On the other hand, for any string $w \in \{0,1\}^*$, we also use $|w|$ to denote the length of w.

Consider the computation of $\mathtt{M}$ on inputs of length ℓ, and design the circuit $\mathtt{C}_\ell$ simulating it. Let $s \leq \mathrm{time}_{\mathtt{M}}(\ell)$ denote the max. number of tape cells used by $\mathtt{M}$ during the computation. Intuitively, $\mathtt{C}_\ell$ is designed as a layer of gates as shown in Figure 2.3. The simulation idea is to express the configuration of $\mathtt{M}(x)$ after each move by a sequence of input nodes of a sequence of gates at each layer of the circuit that computes the next configuration on their output nodes. More specifically, two sets of input nodes are used; one is the set of input nodes to express the current state of $\mathtt{M}$ and another is the set of input nodes to express the tape symbols currently written on tape cells and $\mathtt{M}$'s tape head position.



The structure of the circuit $\mathtt{C}_\ell$ for simulating $\mathtt{M}$ on any input of length ℓ. We assume here that $\mathtt{M}$ uses 6 finite states $q_0, q_1, \ldots, q_5$ and 5 tape symbols $\mathtt{0}, \mathtt{1}, \mathtt{2}, \mathtt{3}$, and $\mathtt{\sqcup}$.

Figure 2.3 The structure of the circuit $\mathtt{C}_\ell$

Note that circuits do not have the "if branch" mechanism. On the other hand, we may use its parallel computation mechanism. That is, we prepare input nodes and gates for

all possible situations that are computed in parallel, where only the gates with the input nodes that have meaningful values (corresponding to the current configuration) would be used for the simulation .

It is easy to see that the number of input nodes for representing a configuration at each level is bounded by $c_1 \cdot s$ for some constant $c_1 = O(|Q| + |\Gamma|)$. Then it is clear that the number of gates at each level is also bounded by $c_2 \cdot c_1 \cdot s$, where c_2 is a constant determined by the complexity of $\mathtt{M}$'s transition function, which is $O(|Q| \cdot |\Gamma|)$. Since there are $\mathrm{time}_{\mathtt{M}}(\ell)$ number of levels and $s \leq \mathrm{time}_{\mathtt{M}}(\ell)$, the total number of gates in $\mathtt{C}_\ell$ is bounded by $c_{2.1} c_{\mathtt{M}} (\mathrm{time}_{\mathtt{M}}(\ell))^2$ for some constant $c_{2.1} \geq 1$ (independent from the choice of $\mathtt{M}$) and $c_{\mathtt{M}} = (|Q| + |\Gamma|) \cdot |Q| \cdot |\Gamma|$. Then the theorem follows.

Now consider the other direction of Theorem 2.1. Can we simulate a family of circuits by a Turing machine? The answer is 'No' due to the "nonuniformity", which is another important feature of the circuit model. Consider any circuit family $\mathtt{C} = \{\mathtt{C}_\ell\}_{\ell \geq 1}$ for solving a given problem L. Our definition does not require the relation between members $\mathtt{C}_1, \mathtt{C}_2, \ldots$ of $\mathtt{C}$ (except that each $\mathtt{C}_\ell$ must solve L on $\{0,1\}^\ell$). Thus, they may be quite unrelated. In particular, we do not have to show how to construct each $\mathtt{C}_\ell$ for a given ℓ. Therefore, for example, the following fact is quite easy to show.

Fact For any function $t : \mathcal{Z}^+ \to \mathcal{Z}^+$, there exists a circuit family $\{\mathtt{C}_\ell\}_{\ell \geq 1}$ that solves a problem $L \notin \mathrm{TIME}(t(\ell))$.

Notice, however, that we usually consider a quite constructive way to define circuits when we design a circuit family for solving a given problem. For example, a circuit family explained above for simulating a given $\mathtt{M}$ can be constructed quite efficiently from the description of $\mathtt{M}$.

2.3 The RAM model

Although we will mainly use the Turing machine model and the circuit model as a model of computation, let us see one more computation model, "Random Access Machine" (abbrev. RAM). This model is intuitively much closer to standard computers executed by machine codes. Here we consider a RAM model similar to the model used in [2].

Definition 2.5 A *random access machine* $\mathtt{RM}$ consists of *registers* and *memory cells* as in Figure 2.4. It is specified by a three tuple $(k, \ell_{\mathrm{wd}}(\ell), \mathtt{P})$, where

- k is the number of registers.
- $\ell_{\mathrm{wd}} : \mathcal{Z}^+ \to \mathcal{Z}^+$ is the *word size bound*, i.e., the max. number of bits stored in a memory cell or a register.
- $\mathtt{P}$ is a program in an assembly language.

Although we do not define our assembly language precisely, we note several important points.

1. RAMs are in general used to compute functions mapping $\mathcal{Z}^+$ to $\mathcal{Z}^+$. For using a RAM for solving a decision problem on $\{0,1\}^*$, we assume that an input string is converted to an integer by adding `1` to its front and that the machine always output either `0` or `1`.

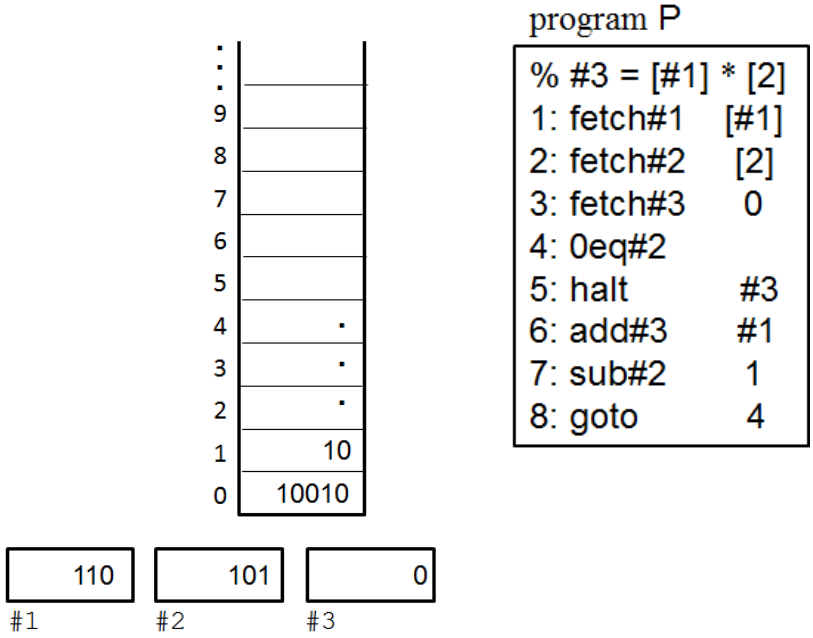
2. The execution of a RAM starts from the first line of its program with an input integer in the first memory cell, i.e., the cell with address 0. The execution terminates when it

executes the halt statement, and we regard the argument of the halt statement as the output.

3. Each line of RAM's program is a simple instruction, including an arithmetic operation to one of its registers. Arithmetic operations are limited to the addition and subtraction, and the general multiplication or division is not allowed so that we can guarantee that the bit length of an integer in a register increases at most t after executing t lines.

4. Each memory cell is given a unique integer address, which is used to refer this memory cell. For example, the instruction "`add#1 [#2]`" means to add to the register #1 the value of the memory cell whose address is specified by the register #2. It is easy to see that an array can be implemented by using this address system.

5. One big difference from ordinary computers is that the word size bound is not fixed in advance; in fact, we can use a function on the input size to determine it. But since the bit length of each data would not increase so rapidly, we do not have to use a large function for the word size bound (see Remark of Theorem 2.2).



A random access machine RM specified by $(3, \ell_{\mathrm{wd}}(\ell), \mathtt{P})$, where $\ell_{\mathrm{wd}}(\ell)$ is the word length determined by the input length ℓ.

Figure 2.4 A random access machine RM

For any RAM RM, in order to discuss its computational efficiency, we introduce the following two functions:

$$\begin{aligned} \mathrm{ram\_time}_{\mathtt{RM}}(x) &= \text{the number of lines of RM on } x \text{ executed until it terminates, and} \\ \mathrm{ram\_space}_{\mathtt{RM}}(x) &= \text{the number of memory cells that RM uses on } x. \end{aligned}$$

Then we have the following relation between Turing machines and RAMs.

Theorem 2.2 There exists some constant $c_{2.2}$ with the following property. For any RAM $\mathtt{RM} = (k, \ell_{\mathrm{wd}}(\ell), \mathtt{P})$, we can construct a Turing machine M that simulates RM and satisfies

the following for any input x:

$$\begin{aligned} \mathrm{time}_{\mathtt{M}}(x) &\leq c_{2.1}(\ell_{\mathrm{wd}}(|x|)\mathrm{ram\_time}_{\mathtt{RM}}(x)) \cdot \mathrm{ram\_time}_{\mathtt{RM}}(x), \quad \text{and} \\ \mathrm{space}_{\mathtt{M}}(x) &\leq c_{2.1}\ell_{\mathrm{wd}}(|x|)\mathrm{ram\_space}_{\mathtt{RM}}(x). \end{aligned}$$

Remark: Due to our restriction of the instruction set of RAM programs, we can show that the bit-length of numbers in registers and memory cells in the execution of $\mathtt{RM}(x)$ is at most $|x| + \mathrm{ram\_time}_{\mathtt{RM}}(x)$. Thus, it does not make sense to consider $\ell_{\mathrm{wd}}(\ell)$ larger than $\ell + \mathrm{ram\_time}_{\mathtt{RM}}(\ell)$. Therefore, the above time bound at most $O(\mathrm{ram\_time}_{\mathtt{RM}}(\ell)^3)$.

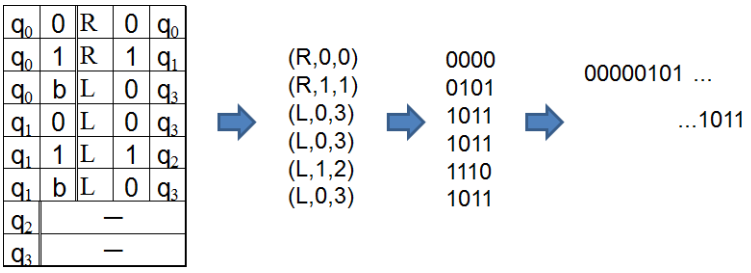
2.4 Universaly Turing machine or Interpreter

One of the important features common in these computation models is that there is a way to encode each machinery by a binary sequence, which we usually call a *program* (or, a *program description*).

For example, as we discussed in the last lecture, any Turing machine $\mathtt{M}$ is specified by $\mathtt{M} = (Q, \Sigma, \Gamma, \delta, \mathtt{q}_0, \mathtt{q}_{\mathrm{accept}}, \mathtt{q}_{reject})$, where Q is a finite set of states, Σ is a finite set of input symbols, ... and so on. It is not so difficult to see that this can be expressed by a string of some finite length, which is encoded in a string in $\{0,1\}^*$. This can be regarded as a "program" expressing $\mathtt{M}$. Note that "program" can be treated separately in the RAM model while a whole machine (or a circuit family) is considered as a program in the Turing machine (resp., the circuit) model.

Here we can naturally consider an "interpreter", a program that interprets a given program description and execute it on a given input (to the program). In the Turing machine model, it is usually called a *universal Turing machine*. This is one of the most fundamental notion in Theory of Computing and even in Computer Science and Engineering in general.

Let us consider this notion more carefully. For this we need to fix our way to encode programs, i.e., Turing machines. Consider any Turing machine $\mathtt{M} = (Q, \Sigma, \Gamma, \delta, \mathtt{q}_0, \mathtt{q}_{\mathrm{accept}}, \mathtt{q}_{reject})$. We can simply use nonnegative integers (encoded by the binary representation) for members of Q and Γ; thus, we only need to specify $q = |Q|$ and $g = |\Gamma|$. We may assume that $\Sigma = \{0,1\}$, $\mathtt{q}_0$ is the first state, i.e., state 0, and $\mathtt{q}_{\mathrm{accept}}$ and $\mathtt{q}_{reject}$ are the last two states respectively. Since δ is a function from $(\, Q - \{\mathtt{q}_{\mathrm{accept}}, \mathtt{q}_{\mathrm{reject}}\}\,) \times \Gamma$ to $(\,\Gamma - \{\textvisiblespace\}\,) \times \{\mathrm{L}, \mathrm{R}\}$, we can encode it by $\{0,1\}^*$ as illustrated in Figure 2.5. Let $\overline{\delta}$ denote this binary encoding of δ. In summary, $\mathtt{M}$ is encoded as $\langle q, g, \overline{\delta}\rangle$ in $\{0,1\}^*$, which is referred as $\overline{\mathtt{M}}$ in the following. (More specifically, we assume some simple encoding of a tuple of binary strings as a single binary string. For example, $\langle \mathtt{010}, \mathtt{10}, \mathtt{0101}\rangle$ can be expressed as $\mathtt{001100101100100101}$.)



An encoding of $\mathtt{M}$ with $q = |Q| = 4$ and $g = |\Gamma| = 3$. We assume that q_2 and q_3 are accepting and rejecting states (i.e., halting states) respectively, and hence $\delta(q_2, -, -)$ and $\delta(q_3, -, -)$ are undefined. Since $|\Gamma| = 3$, we can encode each nonblank symbol by 1 bit. Similarly, each state is encoded by 2 bits. R and L are denoted as 0 and 1 respectively.

Figure 2.5 An encoding of δ in $\{0,1\}^*$

Now we are ready to define the notion of "universal Turing machine."

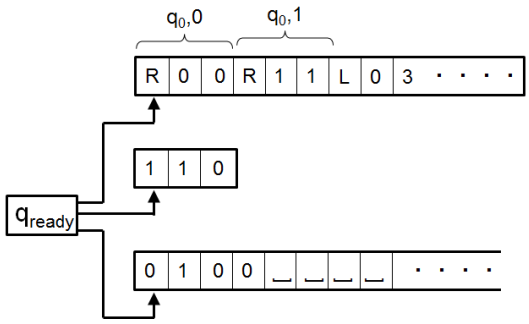
Definition 2.6 A *universal Turing machine* is a Turing machine $\mathtt{M}_{\text{eval}}$ that takes $\langle \overline{\mathtt{M}}, x \rangle$, $x \in \{0,1\}^*$ as an input and simulate the execution of $\mathtt{M}$ on x.
Remark: If $\mathtt{M}(x)$ does not terminate, then so does $\mathtt{M}_{\text{eval}}$ on $\langle \overline{\mathtt{M}}, x \rangle$.

Theorem 2.3 There exists a universal Turing machine $\mathtt{M}_{\text{eval}}$ such that for any Turing machine $\mathtt{M}$ and for any $x \in \{0,1\}^*$, it simulates $\mathtt{M}(x)$ with the following efficiency for a constant $c_{\mathtt{M}}$ determined by $\mathtt{M}$:

$$\text{time}_{\mathtt{M}_{\text{eval}}}(\langle \overline{\mathtt{M}}, x \rangle) \ \leq \ c_{\mathtt{M}} \text{time}_{\mathtt{M}}(x).$$

Remark: Interestingly, it seems difficult (maybe impossible) to get a similar linear bound in the case of RAM model.

The following figure illustrates the idea of the proof. Although the machine uses three tapes for the sake of simplicity, we can show that the simulation can be executed by using a single tape by increasing the running time by only a constant factor.



A snapshot of $\mathtt{M}_{\text{eval}}$ on $\langle \overline{\mathtt{M}}, x \rangle$, where $\overline{\mathtt{M}} = \langle q, g, \overline{\delta} \rangle$ and $x = 0100$. For a simple explanation, three tapes are used; the first one for keeping $\overline{\delta}$ of $\mathtt{M}$, the second one for small counting, and the third one for the actual simulation. This snapshot shows the situation when $\mathtt{M}_{\text{eval}}$ has separated $\overline{\mathtt{M}}$ from the input and it is ready for the simulation.

Figure 2.6 The idea of the universal Turing machine $\mathtt{M}_{\text{eval}}$

A time bounded universal Turing machine

We often need a Turing machine interpreter that can stop the computation when the number of moves exceeds a given time bound.

Theorem 2.4 There exists a Turing machine $\mathtt{M}_{\text{eval\_in\_time}}$ such that for any Turing machine $\mathtt{M}$, for any $x \in \{0,1\}^*$, and for any $t \geq 1$, (i) $\mathtt{M}_{\text{eval\_in\_time}}(\langle \overline{\mathtt{M}}, 0^t, x \rangle)$ simulates $\mathtt{M}(x)$ up to t

moves (and rejects the input if $\mathtt{M}(x)$ does not terminate in t moves), and (ii) it has the following efficiency for a constant $c_{\mathtt{M}}$ determined by $\mathtt{M}$:

$$\mathrm{time}_{\mathtt{M}_{\mathrm{eval\_in\_time}}}(\langle \overline{\mathtt{M}}, 0^t, x\rangle) \ \leq \ c_{\mathtt{M}} \min\{\, \mathrm{time}_{\mathtt{M}}(x), t \,\} \log t.$$

The idea is to introduce a counter that is set t and decremented by 1 after the simulation of each move of $\mathtt{M}$. Again for simplicity, we may keep it as an additional work tape. If we keep it in the unary representation, then the decrement can be done in constant steps. But we cannot (so far) simulate the computation by a one-tape Turing machine better than Theorem 1.1, and the above time bound cannot be obtained. On the other hand, if we keep it in the binary representation, we need $O(\log t)$ steps for the decrement. But we can express the counter by $O(\log t)$ bits, which enable us to implement the time bounded computation in $O(\mathrm{time}_{\mathtt{M}}(x) \cdot \log t)$ moves.

Lecture 3. Complexity Classes and Time/Space Hierarchy Theorems

We discuss the first basic theorem in computational complexity theory, namely, Time and Space Hierarchy Theorems.

3.1 Introduction

Let us recall the following basic complexity classes [1:Def. 7.7, Def. 8.2].

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

$$\begin{array}{lcl} \mathrm{TIME}(t(\ell)) & = & \{\, L \,|\, L \text{ is an } O(t(\ell))\text{-time solvable problem } \}. \\ \mathrm{SPACE}(s(\ell)) & = & \{\, L \,|\, L \text{ is an } O(s(\ell))\text{-space solvable problem } \}. \end{array}$$

Below we use the "small ω" notation, which is defined as follows: for any functions $f, g : \mathcal{Z}^+ \to \mathcal{Z}^+$, we say $g(n) = \omega(f(n))$ if we have

$$\forall c > 0,\ \exists n_c,\ \forall n > n_c \ [\ g(n) > cf(n)\ ].$$

Now we are ready to state Time Hierarchy Theorems. (Space Hierachy Theorem is similar but omitted here.)

Theorem 3.1 (Time Hierarchy Theorem)
For any functions t_1, t_2: $\mathcal{Z}^+ \to \mathcal{Z}^+$, if $t_2(\ell) = \omega(t_1(\ell) \log t_1(\ell))$, then we have

$$\mathrm{TIME}(\, t_1(\ell)\,) \underset{\neq}{\subseteq} \mathrm{TIME}(\, t_2(\ell)\,).$$

Remark: Precisely speaking, we require t_1 to be "time constructible." Intuitively, a time constructible function is a natural function for a time bound. We omit explaining this notion; see [1:Def. 9.8].

Rererences:

1. M. Sipser, Introduction to the Theory of Computation (2nd edition), Thomson Course Technology, 2006, ISBN:0-534-95097-3.
2. 渡辺治，今度こそわかる P ≠ NP 予想，講談社，2014，ISBN:978-4-06-156600-2.