1.4 The Random Access Machine

Trying to design Turing machines for different tasks, one notices that a Turing machine spends a lot of its time by just sending its read-write heads from one end of the tape to the other. One might design tricks to avoid some of this, but following this line we would drift farther and farther away from real-life computers, which have a "random-access" memory, i.e., which can access any field of their memory in one step. So one would like to modify the way we have equipped Turing machines with memory so that we can reach an arbitrary memory cell in a single step.

Of course, the machine has to know which cell to access, and hence we have to assigne addresses to the cells. We want to retain the feature that the memory is unbounded; hence we allow arbitrary integers as addresses. The address of the cell to access must itself be stored somewhere; therefore, we allow arbitrary integers to be stored in each cell (rather than just a single element of a fintie alphabet, as in the case of Turing machines).

Finally, we make the model more similar to everyday machines by making it programmable (we could also say that we define the analogue of a universal Turing machine). This way we get the notion of a *Random Access Machine* or RAM machine.

Now let us be more precise. The *memory* of a Random Access Machine is a doubly infinite sequence $\ldots x[-1], x[0], x[1], \ldots$ of memory registers. Each register can store an arbitrary integer. At any given time, only finitely many of the numbers stored in memory are different from 0.

The *program store* is a (one-way) infinite sequence of registers called *lines*. We write here a program of some finite length, in a certain programming language similar to the assembly language of real machines. It is enough, for example, to permit the following statements:

`x[`i`]:=0;` `x[`i`]:=x[`i`]+1;` `x[`i`]:=x[`i`]-1;`
`x[`i`]:=x[`i`]+x[`j`];` `x[`i`]:=x[`i`]-x[`j`];`
`x[`i`]:=x[x[`j`]];` `x[x[`i`]]:=x[`j`];`
`IF x[`i`]`$\leq$ `0 THEN GOTO` p.

Here, i and j are the addresses of memory registers (i.e. arbitrary integers), p is the address of some program line (i.e. an arbitrary natural number). The instruction before the last one guarantees the possibility of immediate access. With it, the memory behaves as an array in a conventional programming language like Pascal. The exact set of basic instructions is important only to the extent that they should be sufficiently simple to implement, expressive enough to make the desired computations possible, and their number be finite. For example, it would be sufficient to allow the values $-1, -2, -3$ for i, j. We could also omit the operations of addition and subtraction from among the elementary ones, since a program can be written for them. On the other hand, we could also include multiplication, etc.

The *input* of the Random Access Machine is a finite sequence of natural numbers written into the memory registers $x[0], x[1], \ldots$. The Random Access Machine carries out an arbitrary finite program. It stops when it arrives at a program line with no instruction in it. The *output* is defined as the content of the registers $x[i]$ after the program stops.

It is easy to write RAM subroutines for simple tasks that repeatedly occur in programs solving more difficult things. Several of these are given as exercises. Here we discuss three tasks that we need later on in this chapter.

(1.1) **Example** [Value assignment] Let i and j be two integers. Then the assignment

`x[`i`]:=`j

can be realized by the RAM program

```
x[i]:=0
x[i]:=x[i]+1;
⋮                } j times
x[i]:=x[i]+1;
```

if j is positive, and

```
x[i]:=0
x[i]:=x[i]-1;
⋮                } |j| times
x[i]:=x[i]-1;
```

if j is negative. ♢

(1.2) **Example** [Addition of a constant] Let i and j be two integers. Then the statement

`x[`i`]:=x[`i`]+`j

can be realized in the same way as in the previous example, just omitting the first row. ♢

(1.3) **Example** [Multiple branching] Let $p_0, p_1, \ldots, p_r$ be indices of program rows, and suppose that we know that the contents of register i satisfies $0 \leq x[i] \leq r$. Then the statement

`GOTO` $p_{x[i]}$

can be realized by the RAM program

```
IF x[i]≤0 THEN GOTO p_0;
x[i]:=x[i]-1:
IF x[i]≤0 THEN GOTO p_1;
x[i]:=x[i]-1:
⋮
IF x[i]≤0 THEN GOTO p_r.
```

(Attention must be paid when including this last program segment in a program, since it changes the content of xi. If we need to preserve the content of $x[i]$, but have a "scratch" register, say $x[-1]$, then we can do

```
x[-1]:=x[i];
IF x[-1]≤0 THEN GOTO p_0;
x[-1]:=x[-1]-1:
IF x[-1]≤0 THEN GOTO p_1;
x[-1]:=x[-1]-1:
⋮
IF x[-1]≤0 THEN GOTO p_r.
```

If we don't have a scratch register than we have to make room for one; since we won't have to go into such details, we leave it to the exercises. ♢

Now we show that the RAM and Turing machines can compute essentially the same functions, and their running times do not differ too much either. Let us consider (for simplicity) a 1-tape Turing machine, with alphabet $\{0, 1, 2\}$, where (deviating from earlier conventions but more practically here) let 0 stand for the blank space symbol.

Every input $x_1 \dots x_n$ of the Turing machine (which is a 1–2 sequence) can be interpreted as an input of the RAM in two different ways: we can write the numbers n, $x_1, \dots, x_n$ into the registers $x[0], \dots, x[n]$, or we could assign to the sequence $x_1 \dots x_n$ a single natural number by replacing the 2's with 0 and prefixing a 1. The output of the Turing machine can be interpreted similarly to the output of the RAM.

We will consider the first interpretation first.

(1.4) **Theorem** *For every (multitape) Turing machine over the alphabet $\{0, 1, 2\}$, one can construct a program on the Random Access Machine with the following properties. It computes for all inputs the same outputs as the Turing machine and if the Turing machine makes N steps then the Random Access Machine makes $O(N)$ steps with numbers of $O(\log N)$ digits.*

Proof Let $T = \langle 1, \{0, 1, 2\}, \Gamma, \alpha, \beta, \gamma \rangle$. Let $\Gamma = \{1, \dots, r\}$, where 1 = START and r = STOP. During the simulation of the computation of the Turing machine, in register $2i$ of the RAM we will find the same number (0,1 or 2) as in the i-th cell of the Turing machine. Register $x[1]$ will remember where is the head on the tape, and the state of the control unit will be determined by where we are in the program.

Our program will be composed of parts Q_{ij} simulating the action of the Turing machine when in state i and reading symbol j ($1 \le i \le r-1, 0 \le j \le 2$) and lines P_i that jump to $Q_{i,j}$ if the Turing machine is in state i and reads symbol j. Both are easy to realize. P_i is simply

`GOTO` $Q_{i,x[1]}$`;`

for $1 \le i \le i-1$; the program part P_r consists of a single empty program line. The program parts Q_{ij} are only a bit more complicated:

`x[x[1]]:=` $\beta(i,j)$`;`
`x[1]:=x[1]+2`$\gamma(i,j)$`;`
`GOTO` $P_{\alpha(i,j)}$`;`

The program itself looks as follows.

`x[1]:=0;`
P_1
P_2
$\vdots$
P_r
$Q_{1,0}$
$\vdots$
$Q_{r-1,2}$

With this, we have described the simulation of the Turing machine by the RAM. To analyze the number of steps and the size of the number used, it is enough to note that in N steps, the

Turing machine can write anything in at most $O(N)$ registers, so in each step of the Turing machine we work with numbers of length $O(\log N)$. ∎

Another interpretation of the input of the Turing machine is, as mentioned above, to view the input as a single natural number, and to enter it into the RAM as such. This number a is thus in register $x[0]$. In this case, what we can do is to compute the digits of a with the help of a simple program, write these (deleting the 1 in the first position) into the registers $x[0], \ldots, x[n-1]$, and apply the construction described in Theorem 1.4.

(1.5) **Remark** In the proof of Theorem 1.4, we did not use the instruction $x[i] := x[i] + x[j]$; this instruction is needed when computing the digits of the input. Even this could be accomplished without the addition operation if we dropped the restriction on the number of steps. But if we allow arbitrary numbers as inputs to the RAM then, without this instruction the running time the number of steps obtained would be exponential even for very simple problems. Let us e.g. consider the problem that the content a of register $x[1]$ must be added to the content b of register $x[0]$. This is easy to carry out on the RAM in a bounded number of steps. But if we exclude the instruction $x[i] := x[i] + x[\mathrm{j}]$ then the time it needs is at least $\min\{|a|, |b|\}$. ♢

Let a program be given now for the RAM. We can interpret its input and output each as a word in $\{0, 1, -, \#\}^*$ (denoting all occurring integers in binary, if needed with a sign, and separating them by #). In this sense, the following theorem holds.

(1.6) **Theorem** *For every Random Access Machine program there is a Turing machine computing for each input the same output. If the Random Access Machine has running time N then the Turing machine runs in $O(N^2)$ steps.*

Proof We will simulate the computation of the RAM by a four-tape Turing machine. We write on the first tape the content of registers $x[i]$ (in binary, and with sign if it is negative). We could represent the content of all registers (representing, say, the content 0 by the symbol "*"). This would cause a problem, however, because of the immediate ("random") access feature of the RAM. More exactly, the RAM can write even into the register with number 2^N using only one step with an integer of N bits. Of course, then the content of the overwhelming majority of the registers with smaller indices remains 0 during the whole computation; it is not practical to keep the content of these on the tape since then the tape will be very long, and it will take exponential time for the head to walk to the place where it must write. Therefore we will store on the tape of the Turing machine only the content of those registers into which the RAM actually writes. Of course, then we must also record the number of the register in question.

What we will do therefore is that whenever the RAM writes a number y into a register $x[z]$, the Turing machine simulates this by writing the string $\#\#y\#z$ to the end of its first tape. (It never rewrites this tape.) If the RAM reads the content of some register $x[z]$ then on the first tape of the Turing machine, starting from the back, the head looks up the first string of form $\#\#u\#z$; this value u shows what was written in the z-th register the last time. If it does not find such a string then it treats $x[z]$ as 0.

Each instruction of the "programming language" of the RAM is easy to simulate by an appropriate Turing machine using only the three other tapes. Our Turing machine will be a

"supermachine" in which a set of states corresponds to every program line. These states form a Turing machine which carries out the instruction in question, and then it brings the heads to the end of the first tape (to its last nonempty cell) and to cell 0 of the other tapes. The STOP state of each such Turing machine is identified with the START state of the Turing machine corresponding to the next line. (In case of the conditional jump, if $x[i] \leq 0$ holds, the "supermachine" goes into the starting state of the Turing machine corresponding to line p.) The START of the Turing machine corresponding to line 0 will also be the START of the supermachine. Besides this, there will be yet another STOP state: this corresponds to the empty program line.

It is easy to see that the Turing machine thus constructed simulates the work of the RAM step-by-step. It carries out most program lines in a number of steps proportional to the number of digits of the numbers occurring in it, i.e. to the running time of the RAM spent on it. The exception is readout, for wich possibly the whole tape must be searched. Since the length of the tape is N, the total number of steps is $O(N^2)$. ■

1.17 **Exercise** Write a program for the RAM that for a given positive number a

(a) determines the largest number m with $2^m \leq a$;

(b) computes its base 2 representation;

♢

1.18 **Exercise** Let $p(x) = a_0 + a_1 x + \cdots + a_n x^n$ be a polynomial with integer coefficients $a_0, \ldots, a_n$. Write a RAM program computing the coefficients of the polynomial $(p(x))^2$ from those of $p(x)$. Estimate the running time of your program in terms of n and $K = \max\{|a_0|, \ldots, |a_n|\}$. ♢

1.19 **Exercise** Prove that if a RAM is not allowed to use the instruction $x[i] := x[i] + x[j]$, then adding the content a of $x[1]$ to the content b of $x[2]$ takes at least $\min\{|a|, |b|\}$ steps.
♢

1.20 **Exercise** Since the RAM is a single machine the problem of universality cannot be stated in exactly the same way as for Turing machines: in some sense, this single RAM is universal. However, the following "self-simulation" property of the RAM comes close. For a RAM program p and input x, let $R(p, x)$ be the output of the RAM. Let $\langle p, x \rangle$ be the input of the RAM that we obtain by writing the symbols of p one-by-one into registers 1,2,..., followed by a symbol # and then by registers containing the original sequence x. Prove that there is a RAM program u such that for all RAM programs p and inputs x we have $R(u, \langle p, x \rangle) = R(p, x)$. ♢

1.21 **Exercise** [Pointer Machine.] After having seen finite-dimensional tapes and a tree tape, we may want to consider a machine with a more general directed graph its storage medium. Each cell c has a fixed number of edges, numbered $1, \ldots, r$, leaving it. When the head scans a certain cell it can move to any of the cells $\lambda(c, i)$ $(i = 1, \ldots, r)$ reachable from it along outgoing edges. Since it seems impossible to agree on the best graph, we introduce a new kind of elementary operation: to change the structure of the storage graph locally, around the scanning

head. Arbitrary transformations can be achieved by applying the following three operations repeatedly (and ignoring nodes that become isolated): $\lambda(c,i) :=$ New, where New is a new node; $\lambda(c,i) := \lambda(\lambda(c,j))$ and $\lambda(\lambda(c,i)) := \lambda(c,j)$. A machine with this storage structure and these three operations added to the usual Turing machine operations will be called a Pointer Machine.

Let us call RAM' the RAM from which the operations of addition and subtraction are omitted, only the operation $x[i] := x[i] + 1$ is left. Prove that the Pointer Machine is equivalent to RAM', in the following sense.

For every Pointer Machine there is a RAM' program computing for each input the same output. If the Pointer Machine has running time N then the RAM' runs in $O(N)$ steps.

For every RAM' program there is a Pointer Machine computing for each input the same output. If the RAM' has running time N then the Pointer Machine runs in $O(N)$ steps.

Find out what Remark 1.5 says for this simulation. $\diamondsuit$