space as well, constants don't count. Its proof is an easy variant of that proof (see Problem 2.8.14).

Theorem 2.3: Let L be a language in **SPACE**$(f(n))$. Then, for any $\epsilon > 0$, $L \in$ **SPACE**$(2 + \epsilon f(n))$. □

2.6 RANDOM ACCESS MACHINES

We have already seen a number of examples and results suggesting that Turing machines, despite their weak data structure and instruction set, are quite powerful. But can they really be used to implement arbitrary algorithms?

We claim that they can. We shall not be able to prove this claim rigorously, because the notion of an "arbitrary algorithm" is something we cannot define (in fact, the Turing machine is the closest that we shall come to defining it in this book). The classical version of this informal claim, known as *Church's Thesis*, asserts that any algorithm can be rendered as a Turing machine. It is supported by the experiences of mathematicians in the early twentieth century while trying to formalize the notion of an "arbitrary algorithm." Quite independently, and starting from vastly different points of view and backgrounds, they all came up with models of computation that were ultimately shown equivalent in computational power to the Turing machine (which was the entry by one of them, Alan M. Turing; see the references for more on this fascinating story).

In the decades since those pioneering investigations, computing practice and the concern of computer scientists about time bounds has led them to another widespread belief, which can be thought of as a quantitative strengthening of Church's Thesis. It states that *any reasonable attempt to model mathematically computer algorithms and their time performance is bound to end up with a model of computation and associated time cost that is equivalent to Turing machines within a polynomial.* We have already seen this principle at work, when we noticed that multistring Turing machines can be simulated by the original model with only a quadratic loss of time. In this section we will present another, even more persuasive argument of this sort: We shall define a model of computation that reflects quite accurately many aspects of actual computer algorithms, and then show that it too is equivalent to the Turing machine.

A *random access machine*, or *RAM*, is a computing device which, like the Turing machine, consists of a program acting on a data structure. A RAM's data structure is an array of *registers*, each capable of containing an *arbitrarily large integer*, possibly negative. RAM instructions resemble the instruction set of actual computers; see Figure 2.6.

Formally, a *RAM program* $\Pi = (\pi_1, \pi_2, \ldots, \pi_m)$ is a finite sequence of *instructions*, where each instruction π_i is of one of the types shown in Figure 2.6. At each point during a RAM computation, Register i, where $i \geq 0$, contains an integer, possibly negative, denoted r_i, and instruction π_κ is executed. Here

Instruction	**Operand**	**Semantics**
READ	j	$r_0 := i_j$
READ	$\uparrow j$	$r_0 := i_{r_j}$
STORE	j	$r_j := r_0$
STORE	$\uparrow j$	$r_{r_j} := r_0$
LOAD	x	$r_0 := x$
ADD	x	$r_0 := r_0 + x$
SUB	x	$r_0 := r_0 - x$
HALF		$r_0 := \lfloor \frac{r_0}{2} \rfloor$
JUMP	j	$\kappa := j$
JPOS	j	**if** $r_0 > 0$ **then** $\kappa := j$
JZERO	j	**if** $r_0 = 0$ **then** $\kappa := j$
JNEG	j	**if** $r_0 < 0$ **then** $\kappa := j$
HALT		$\kappa := 0$

j is an integer. r_j are the current contents of Register j. i_j is the jth input. x stands for an operand of the form j, "$\uparrow j$", or "$= j$". Instructions READ and STORE take no operand "$= j$". The value of operand j is r_j; that of "$\uparrow j$" is r_{r_j}; and that of "$= j$" is j. κ is the program counter. All instructions change κ to $\kappa + 1$, unless explicitly stated otherwise.

Figure 2.6. RAM instructions and their semantics.

κ, the "program counter," is another evolving parameter of the execution. The execution of the instruction may result in a change in the contents of one of the registers, and a change in κ. The changes, that is, the *semantics* of the instructions, are those one would expect from an actual computer, and are summarized in Figure 2.6.

Notice the special role of Register 0: It is the *accumulator*, where all arithmetic and logical computation takes place. Also notice the three *modes of addressing*, that is, of identifying an operand: Some instructions use an *operand* x, where x may be any of three types: j, "$\uparrow j$", or "$= j$". If x is an integer j, then x identifies the contents of Register j; if x is "$\uparrow j$", then it stands for the contents of the register *whose index is contained* in Register j; and if x is "$= j$", this means the integer j itself. The number thus identified is then used in the execution of the instruction.

An instruction also changes κ. Except for the last five instructions of Figure 2.6, the new value of κ is simply $\kappa+1$, that is, the instruction to be executed next is the next one. The four "jump" instructions change κ in another way, often dependent on the contents of the accumulator. Finally, the HALT instruction stops the computation. We can think that any *semantically wrong* instruction, such as one that addresses Register -14, is also a HALT instruction. Initially all registers are initialized to zero, and the first instruction is about to be executed.

The *input* of a RAM program is a finite sequence of integers, contained in a finite array of *input registers*, $I = (i_1, \ldots, i_n)$. Any integer in the input can be transferred to the accumulator by a READ instruction. Upon halting, the output of the computation is contained in the accumulator.

The RAM executes the first instruction and implements the changes dictated by it, then it executes instruction π_κ, where κ is the new value of the program counter, and so on, until it halts. RAM computations can be formalized by defining the RAM analog of a configuration. A *configuration of a RAM* is a pair $C = (\kappa, R)$, where κ is the instruction to be executed, and $R = \{(j_1, r_{j_1}), (j_2, r_{j_2}), \ldots, (j_k, r_{j_k})\}$ is a finite set of *register-value pairs*, intuitively showing the current values of all registers that have been changed so far in the computation (recall that all others are zero). The initial configuration is $(1, \emptyset)$.

Let us fix a RAM program Π and an input $I = (i_1, \ldots, i_n)$. Suppose that $C = (\kappa, R)$ and $C' = (\kappa', R')$ are configurations. We say that (κ, R) *yields in one step* (κ', R'), written $(\kappa, R) \xrightarrow{\Pi,I} (\kappa', R')$, if the following holds: κ' is the new value of κ after the execution of π_κ, the κth instruction of Π. R', on the other hand, is the same as R in the case the κth instruction is one of the last five instructions in Figure 2.6. In all other cases, some Register j has a new value x, computed as indicated in Figure 2.6. To obtain R' we delete from R any pair (j, x') that may exist there, and add to it (j, x). This completes the definition of the relation $\xrightarrow{\Pi,I}$. We can now, as always, define $\xrightarrow{\Pi,I^k}$ (*yields in k steps*) and $\xrightarrow{\Pi,I^*}$ (*yields*).

Let Π be a RAM program, let D be a set of finite sequences of integers, and let ϕ be a function from D to the integers. We say that Π *computes* ϕ if, for any $I \in D$, $(1, \emptyset) \xrightarrow{\Pi,I^*} (0, R)$, where $(0, \phi(I)) \in R$.

We next define the time requirements of RAM programs. Despite the fact that the arithmetic operations of RAMs involve arbitrarily large integers, *we shall still count each RAM operation as a single step.* This may seem too liberal, because an ADD operation, say, involving large integers would appear to "cost" as much as the *logarithm* of the integers involved. Indeed, with more complicated arithmetic instructions the unit cost assumption would lead us to a gross underestimate of the power of RAM programs (see the references). However, our instruction set (in particular, the absence of a primitive multiplication instruction) ensures that the unit cost assumption is a simplification that is otherwise inconsequential. The mathematical justification for this choice will come with Theorem 2.5, stating that RAM programs, even with this extremely liberal notion of time, can be simulated by Turing machines with only a polynomial loss of efficiency.

We do need, however, logarithms in order to account for the size of the input. For i an integer, let $\mathrm{b}(i)$ be its *binary representation*, with no redundant leading 0s, and with a minus sign in front, if negative. The *length* of integer i is $\ell(i) = |\mathrm{b}(i)|$. If $I = (i_1, \ldots, i_k)$ is a sequence of integers, its *length* is defined as $\ell(I) = \sum_{j=1}^{k} \ell(i_j)$.

Suppose now that a RAM program Π computes a function ϕ from a domain D to the integers. Let f be a function from the nonnegative integers to the nonnegative integers, and suppose that, for any $I \in D$, $(1, \emptyset) \xrightarrow{\Pi, I}{}^{k} (0, R)$, where $k \leq f(\ell(I))$. Then we say that Π *computes ϕ in time $f(n)$*. In other words, we express the time required by a RAM computation on an input as a function of the total length of the input.

```
1.    READ 1
2.    STORE 1    (Register 1 contains $i_1$; during the $k$th iteration,
3.    STORE 5      Register 5 contains $i_1 2^k$. Currently, $k = 0$.)
4.    READ 2
5.    STORE 2    (Register 2 contains $i_2$)
6.    HALF       ($k$ is incremented, and $k$th iteration begins)
7.    STORE 3    (Register 3 contains $\lfloor i_2/2^k \rfloor$)
8.    ADD 3
9.    SUB 2
10.   JZERO 14
11.   LOAD 4     (add Register 5 to Register 4
12.   ADD 5        only if the $k$th least significant bit of $i_2$ is zero)
13.   STORE 4    (Register 4 contains $i_1 \cdot (i_2 \bmod 2^k)$)
14.   LOAD 5
15.   ADD 5
16.   STORE 5    (see comment of instruction 3)
17.   LOAD 3
18.   JZERO 20   (if $\lfloor i_2/2^k \rfloor = 0$ we are done)
19.   JUMP 5     (if not, we repeat)
20.   LOAD 4     (the result)
21.   HALT
```

Figure 2.7. RAM program for multiplication.

Example 2.7: Conspicuous in Figure 2.6 is the absence of the *multiplication* instruction MPLY. This is no great loss, because the program in Figure 2.7 computes the function $\phi : \mathbf{N}^2 \mapsto \mathbf{N}$ where $\phi(i_1, i_2) = i_i \cdot i_2$. The method is ordinary binary multiplication, where the instruction HALF is used to recover the binary representation of i_2 (actually, our instruction set contains this unusual instruction precisely for this use).

In more detail, the program repeats $\lceil \log i_2 \rceil$ times the iteration in instructions 5 through 19. At the beginning of the kth iteration (we start with iteration 0) Register 3 contains $\lfloor i_2/2^k \rfloor$, Register 5 contains $i_1 2^k$, and register 4 contains $i_1 \cdot (i_2 \bmod 2^k)$. Each iteration strives to maintain this invariant (the reader is welcome to verify this). At the end of the iteration, we test whether Register 3 contains 0. If it does, we are done and ready to output the contents of Register 4. Otherwise, the next iteration is started.

It is easy to see that this program computes the product of two numbers in $\mathcal{O}(n)$ time. Recall that by $\mathcal{O}(n)$ time we mean *a total number of instructions that is proportional to the logarithm of the integers contained in the input.* This follows from the number of iterations of the program in Figure 2.7, which is at most $\log i_2 \leq \ell(I)$; each iteration entails the execution of a constant number of instructions.

Notice that, once we have seen this program, we can freely use the instruction MPLY x in our RAM programs, where x is any operand (j, $\uparrow j$, or $= j$). This can be simulated by running the program of Figure 2.7 (but possibly at a totally different set of registers). A few more instructions would take care of the case of negative multiplicands. □

Example 2.8: Recall the REACHABILITY problem from Section 1.1. The RAM program in Figure 2.8 solves REACHABILITY. It assumes that the input to the problem is provided in $n^2 + 1$ input registers, where Input Register 1 contains the number n of nodes, and Input Register $n(i-1)+j+1$ contains the (i,j)th entry of the adjacency matrix of the graph, for $i, j = 1, \ldots, n$. We assume that we wish to determine whether there is a path from node 1 to node n. The output of the program is simply 1 if a path exists (if the instance is a "yes" instance) and 0 otherwise.

The program uses Registers 1 through 7 as "scratch" storage. Register 3 always contains n. Register 1 contains i (the node just taken off S, recall the algorithm in Section 1.1), and Register 2 contains j, where we are currently investigating whether j should be inserted in S. In Registers 9 through $n+8$ we maintain the *mark bits* of the nodes: One if the node is marked (has been in S at some time in the past, recall the algorithm in Section 1.1), zero if it is not marked. Register $8+i$ contains the mark bit of node i.

Set S is maintained as a *stack;* as a result, the search will be depth-first. Register $n+9$ points to the top of the stack, that is, to the register containing the last node that was added to the stack. If Register $n+9$ is ever found to point to a register containing a number bigger than n (that is, to itself), then we know the stack is empty. For convenience, Register 8 contains the integer $n+9$, the address of the pointer to the top of the stack.

```
 1.   READ 1
 2.   STORE 3       (Register 3 contains n)
 3.   ADD = 9
 4.   STORE 8       (Register 8 contains the address of Register n + 9,
 5.   ADD = 1         where the address of the top of the stack is stored)
 6.   STORE ↑ 8     (Initially, the stack will contain only one element, node 1)
 7.   STORE 7
 8.   LOAD = 1      (push node 1 to the top of the stack;
 9.   STORE ↑ 7       this ends the initialization phase)
10.   LOAD ↑ 8      (an iteration begins here)
11.   STORE 6
12.   LOAD ↑ 6      (load the top of the stack;
13.   SUB 3           if it is a number bigger than n,
14.   JPOS 52         the stack is empty and we must reject)
15.   STORE 1       (Register 1 contains i)
16.   LOAD 6
17.   SUB = 1
18.   STORE ↑ 8     (the top of the stack goes down by one)
19.   LOAD = 1
20.   STORE 2       (Register 2 contains j, initially 1)
21.   LOAD 1
22.   SUB = 1
23.   MPLY 3        (the multiplication program, Figure 2.7, is used here)
24.   ADD 2         (compute (i − 1) · n + j, the address of the
25.   ADD = 1         (i, j)th element of the adjacency matrix)
26.   STORE 4
27.   READ ↑ 4      (the (i, j)th element is read in)
28.   JZERO 48      (if zero, nothing to do)
29.   LOAD 2
30.   SUB 3
31.   JZERO 54      (if j = n we must accept)
32.   LOAD 2
33.   ADD = 8       (compute the address of the mark bit of j)
34.   STORE 5
35.   LOAD ↑ 5
36.   JPOS 48       (if the mark bit of j is one, nothing to do)
37.   LOAD = 1
38.   STORE ↑ 5     (the mark bit of j is now one)
39.   LOAD ↑ 8      (continued next page...)
```

Figure 2.8. RAM program for REACHABILITY.

```
40.   ADD = 1
41.   STORE 7
42.   STORE ↑ 8     (the top of the stack goes up by one)
43.   LOAD 2
44.   STORE ↑ 7     (push j onto the stack)
45.   LOAD 2
46.   SUB 3
47.   JZERO 10      (if j = n, then a new iteration must begin)
48.   LOAD 2
49.   ADD = 1
50.   STORE 2
51.   JUMP 21       (otherwise, j is incremented, and we repeat)
52.   LOAD = 0
53.   HALT          (and reject)
54.   LOAD = 1
55.   HALT          (and accept)
```

Figure 2.8 (Continued). RAM program for REACHABILITY.

Undoubtedly, this program keeps little of the original simplicity and elegance of the search algorithm, but it is an accurate rendering of that algorithm (you are invited to check). The number of steps is $\mathcal{O}(n^2 \log n)$: The work done per each READ instruction (each probe of the adjacency relation) is at most a constant plus $\log n$ for the MPLY instruction, and each entry is probed only once. To express this as a function of the length of the input (as we should for RAM programs), the length of the input $\ell(I)$ is about $(n^2 + \log n)$, which is $\Theta(n^2)$. Thus, the program works in time $\mathcal{O}(\ell(I) \log \ell(I))$.

If we wish to implement breadth-first search, this can be accomplished by realizing a *queue* instead of a stack; it would only be a minor modification of the RAM program of Figure 2.8. □

As a further illustration of the power of RAM programs, the next theorem states the rather expected fact that any Turing machine can be easily simulated by a RAM program. Suppose that $\Sigma = \{\sigma_1, \ldots, \sigma_k\}$ is the alphabet of the Turing machine. Then we let D_Σ be this set of finite sequences of integers: $D_\Sigma = \{(i_1, \ldots, i_n, 0) : n \geq 0, 1 \leq i_j \leq k, j = 1, \ldots, n\}$. If $L \subset (\Sigma - \{\sqcup\})^*$ is a language, define ϕ_L to be the function from D_Σ to $\{0, 1\}$ with $\phi_L(i_1, \ldots, i_n, 0) = 1$ if $\sigma_{i_1} \ldots \sigma_{i_n} \in L$, and 0 otherwise. In other words, computing ϕ_L is the RAM equivalent of deciding L. The last 0 in the input sequence helps the RAM program "sense" the end of the Turing machine's input.

Theorem 2.4: Suppose that language L is in $\textbf{TIME}(f(n))$. Then there is a RAM program which computes the function ϕ_L in time $\mathcal{O}(f(n))$.