The Complexity of Computing

John E. Savage

The Complexity of Computing

John E. Savage
Brown University

A WILEY-INTERSCIENCE PUBLICATION

John Wiley & Sons, New York • London • Sydney • Toronto

Library of Congress Cataloging in Publication Data:

Savage, John E 1939–
The complexity of computing.

"A Wiley-Interscience publication."
Bibliography: p
Includes index.
1. Machine theory. 2. Switching theory 3 Computational complexity. I Title
QA267.S28 001.6'4'0184 76-27733
ISBN 0-471-75517-6

Printed in the United States of America

10 9 8 7 6 5 4 3 2 1

Preface

The complexity of computing is one of the most important problems in computer science today. It represents an enormous challenge with which we must cope if we are to understand the complex programs and machines that we construct, and it is the principal motivation for the development of a science of computing.

There have been many responses to the complexity of computing, especially in the general area of programming. Ours is a different response. We explicitly measure the complexity of problems and then examine the resources, such as space and time, that are needed under the best possible conditions to compute functions of a given complexity on general-purpose computers. The results we derive concerning this topic are in the form of inequalities that state lower limits on possible space–time tradeoffs. The study of such tradeoffs is useful in developing intuition concerning the cost-effective use of computers.

This book serves two principal purposes. It develops the tradeoff results mentioned and gives an advanced treatment of important topics in switching and automata theory. In particular the book has two chapters (Chapters 2 and 3) on the size and depth of logic circuits and the size of formulas for Boolean functions. These chapters contain numerous interesting results, most of which are either very recent or are scattered throughout the Russian literature. The Bibliography contains an extensive list of works dealing with combinational complexity and related topics; these references are set off by asterisks.

Chapters 4 and 5 develop a number of classical topics in the theories of sequential machines and Turing machines, as well as some new results that are used to derive the tradeoff inequalities. The classical topics include reduced machines and regular expressions, universal Turing machines, unsolvability of the halting problem, and partial-recursive functions. The new topics include a product relation between the circuit size of a sequential machine, its computation time, and the circuit size of the function it is used to compute, as well as computational work and program complexity.

Chapter 6 gives an extensive treatment of the principal components of general-purpose computers, beginning with flip-flops and continuing through microprogramming to storage devices. These topics are examined from a complexity point of view. A number of interesting circuit algorithms are also given for arithmetic operations that have both small size and small depth. This chapter can serve as a good introduction to the organization of general-purpose computers.

Chapter 7 develops the computational inequalities that state lower limits on space–time tradeoffs. It also presents new results which show that the lower limits can be approached for many problems. Under the assumption that the lower limits can be achieved, we examine cost-effective operating points on the space–time boundaries under several different cost functions. We draw some interesting conclusions that are supportive of current practice.

Chapter 8 presents a sound introduction to three topics in the design and analysis of algorithms: sorting, matrix-multiplication and *NP*-complete problems. We examine a number of (nonadaptive) sorting networks as well as adaptive algorithms including Heapsort. The treatment of matrix multiplication includes the presentation of Strassen's matrix multiplication algorithm and a succinct derivation of the Fast Fourier transform algorithm. In the third section of Chapter 8, we examine a number of *NP*-complete problems as well as polynomial-time approximation algorithms for some of them. These three topics provide important results and examples of important problems that are used elsewhere in the book.

In the writing of this book I have been fortunate to have the assistance of many friends. They include Howard Elliott, Yeshoshua Imber, Roy Johnson, Edmund Lamagna, Joel Silverberg, Sowmitri Swamy, and Charles Wade, who are or recently were graduate students at Brown University and who have done a critical reading of most or much of the book. Other friends have read substantial portions of the book and/or have offered useful advice or input. They include Diedrich van Daalen, Andy van Dam, Larry Harper, Daniel Lehman, Clem McGowan, Ron Rivest, Bob Sedgewick, and Frances Yao. Many others have contributed, including my students in recent years.

The writing of the book began during my sabbatical leave in 1973–1974, which was spent in the Mathematics Department at the Technische Hogeschool Eindhoven (THE), The Netherlands. For the opportunity to spend my leave at this most hospitable institution in a most hospitable country, I thank Edsger Dijkstra and Jack van Lint. To Prof. Dr. Lunbeck I also express my thanks for a most attractive working environment.

The task of converting my jottings to typed form fell to E. E. F. M. Baselmans-Weijers, Hanny van Dongen-van Nisselrooij, Marèse van den Hurk, and H. K. van der Putten-Bosscher at THE, and to Linda and Sharon Trevitt, Joyce Oliver, and Claire Crockett at Brown. I give them

my sincere thanks for their kind cooperation, speedy service, and terrific copy. My thanks are also due S. Lowes for her valuable help in chasing down references.

To Brown University, which supported a portion of my sabbatical leave as well as much of my time since then, and to the John Simon Guggenheim Memorial Foundation and the Netherlands-America Commission for Educational Exchange (NACEE) which also supported a portion of my sabbatical leave, I express my sincere appreciation. In particular, I wish to mention Wobbina Kwast, Executive Director of NACEE, who was most effective in making the Fulbright-Hays Scholars feel at home in the Netherlands. I also thank the National Science Foundation for the support of my research which has found its way into my book.

And finally, I express my sincere gratitude to my wife, Patricia, who has been a true source of support and encouragement and who has given generously of herself to facilitate the early completion of this book. I also remember our children, Elizabeth and Kevin, for their patience.

JOHN E. SAVAGE

Providence, Rhode Island
August 1976

Contents

Chapter 1 **Introduction** **1**

1.1 Computer Science as a Science of the Artificial 2
1.2 Computational Models 4
1.3 Complexity Measures 8
1.4 Computational Inequalities 11

Chapter 2 **Combinational Complexity of Functions** **14**

2.1 Boolean Functions and Normal Forms 15
2.2 Computation Chains and Combinational Machines 20
2.3 Combinational Complexity Measures 23
2.4 Lower Bounds to Combinational Complexity 31
2.4.1 The Simple Linear Bound 31
2.4.2 Extensions of the Simple Linear Bound 33
2.4.3 Some Special Linear Lower Bounds 48
2.4.4 The Combinational Complexity of a Set of Functions 57
Appendix 59
Problems 66

Chapter 3 **Combinational Complexity and Formula Size** **69**

3.1 Functions over Complete Bases 69
3.1.1 Logical and Arithmetic Operations 70
3.1.2 Symmetric Functions 75
3.1.3 The Determinant Function 77
3.2 Functions over Incomplete Bases 80
3.2.1 Monotone Functions 80
3.2.2 Binary Sorting 81
3.2.3 The Computation of Monotone Bilinear Forms 84
3.3 Bounding Methods for Formula Size 91
3.3.1 The Krapchenko Lower Bound 91
3.3.2 The Nechiporuk Lower Bound 101
3.3.3 Formula Size of Threshold Functions 108
3.4 Asymptotic Bounds 114

3.5 Synchronous Combinational Complexity 123
Problems 129

Chapter 4 Sequential Machines 136
4.1 Finite-State Machine Model 136
4.2 Reduced Machines and Machine Equivalence 140
4.3 Regular Expressions and Sequential Machines 145
4.3.1 Regular Expressions and Their Derivatives 145
4.3.2 Regular Expressions from Sequential Machines 151
4.3.3 Sequential Machines from Regular Expressions 154
4.3.4 The Pumping Theorem for Regular Sets 156
4.4 Bounds on the Complexities of Descriptor Functions 157
4.5 Computational Work and Efficiency 164
Problems 170

Chapter 5 Turing Machines 174
5.1 The Basic Turing Machine 174
5.2 Extensions of the Basic Machine 177
5.3 Computations on Turing Machines 181
5.4 Universal Turing Machines 182
5.5 Unsolvability of the Halting Problem 185
5.6 Partial Recursive Functions—
The McCarthy Formalism 187
5.7 Program Complexity of Finite Functions 193
5.8 A Computational Inequality for Turing Machines 202
Problems 210

Chapter 6 General-Purpose Computers 212
6.1 Organization of General-Purpose Computers 212
6.2 Storage Media 214
6.2.1 Flip-Flops and Registers 214
6.2.2 Random Access Memory 220
6.2.3 Tape Memory 225
6.2.4 Drum and Disk Memories 230
6.2.5 Associative Memory 235
6.3 Arithmetic and Logical Functions 237
6.3.1 Vector Operations 237
6.3.2 Shifting 237
6.3.3 Comparison Tests 239
6.3.4 Delay Complexity of Overflow Tests 241
6.3.5 Addition 244
6.3.6 Subtraction 247
6.3.7 Multiplication 250

6.3.8 Division 255
6.3.9 Floating-Point Arithmetic 259
6.4 Central Processing Unit 261
6.4.1 Microinstructions for a Simple CPU 262
6.4.2 Microprograms 266
6.4.3 Microprogrammed Control 269
6.4.4 Machine Language Programming 271
6.4.5 Other Features of General-Purpose Computers 275
Appendix 276
Problems 284

Chapter 7 Storage–Time Tradeoffs 287

7.1 Problems, Machines, Programs, and Computations 287
7.2 Computational Inequalities of the First Kind 289
7.3 Computational Inequalities of the Second Kind 292
7.4 Limits on the Performance of Computers 300
7.4.1 Summary of Computational Inequalities 301
7.4.2 Space–Time Boundaries for M_{RAM} and M_{CAM} 302
7.4.3 Realizable Storage–Time Tradeoffs 304
7.4.4 Computational Efficiency 312
7.4.5 Limits on the Performance of a Two-Memory Machine 316
Problems 318

Chapter 8 Combinatorial and Algebraic Problems 320

8.1 Sorting 320
8.1.1 Sorting Networks 321
8.1.2 Adaptive Sorting 326
8.2 Matrix Multiplication 334
8.2.1 The Multiplication Problem 335
8.2.2 Strassen's Matrix Multiplication Algorithm 337
8.2.3 Matrix-Vector Multiplication 338
8.2.4 The Fast Fourier Transform 342
8.3 Difficult Combinatorial Problems 347
8.3.1 Nondeterministic Turing Machines 348
8.3.2 The Classes *P* and *NP* 349
8.3.3 The Satisfiability Problem is *NP*-Complete 352
8.3.4 Some *NP*-Complete Problems 356
8.3.5 Polynomial-Time Approximations to *NP*-Complete Problems 364
Problems 368

***Bibliography* 373**
***Index* 387**

Chapter 1

Introduction

. . . the world is waiting to hear the answers to such questions as
- *What measurements exist?*
- *What efficiencies do they indicate?*
- *How can operations be improved?*
- *And at what degree of predictability?*

To me this is the essence of science and engineering in any field of endeavor.

WALTER M. CARLSON
Reflections on Ljubljana
CACM, October 1971.

As we succeed in broadening and deepening our knowledge—theoretical and empirical—about computers, we shall discover that in large part their behavior is governed by simple general laws, that what appeared as complexity in the computer program was, to a considerable extent, complexity of the environment to which the program was seeking to adapt its behavior.

HERBERT A. SIMON
The Sciences of the Artificial
MIT Press, 1968.

Many important computing systems and programs are sufficiently complex that they tax our powers of comprehension, and yet we must attain some satisfactory understanding of these systems and programs before we can be reasonably confident that we are making efficient use of them. This book explicitly recognizes the complexity of computing and attempts to respond to the first question raised by Walter Carlson in the manner suggested by Herbert Simon, namely, by moving to the proper level of abstraction. At this level we examine computing systems and programs in terms of a few macroscopic parameters such as space and time, and we make an effort to establish the fundamental relationships that exist between these parameters. For this purpose, we examine in detail a number of analytical tools that have a great deal of interest in their own right as well as for the role they play in the development of this new level of understanding.

1.1. COMPUTER SCIENCE AS A SCIENCE OF THE ARTIFICIAL

The handiwork of man, the artificial, is evident everywhere in our daily lives. It is seen in our clothing, the temperature of the air we breathe, the social and economic systems in which we participate, and in the composition of this book. Some of our artificial systems are fairly easy to understand and/or are not deserving of detailed analysis. Others, in particular, computer systems and programs, have such complexity and economic importance that they must be subjected to close analysis and comprehended at each of several levels.

Our purpose in this book is to lay the groundwork for such analysis and comprehension. But are we not doomed to failure? Isn't it impossible to develop a science of the artificial? These are questions that have attracted Herbert Simon in the context of management science, psychology, computer systems, and engineering education, and he offers convincing arguments that they will be settled in the negative. Speaking of natural science he says (Simon, 1969)

> The central task of a natural science is to make the wonderful commonplace: to show that complexity, correctly viewed, is only a mask for simplicity.

Should that not also be the objective of a science of the artificial?

If the need to understand complex man-made systems exists, where do we begin? Simon (1969) suggests an answer in his definition of an artifact, a man-made object or system:

> An artifact can be thought of as a meeting point—an "interface" in today's terms—between an inner environment, the substance and organization of the artifact itself, and an outer environment, the surroundings in which it operates.

Thus, the place to look for such an understanding is at the interface, at the point where the inner and outer environment meet, the point at which the stated goals of the system meet the internal constraints set by the inner environment.

In computing systems we suggest that one important interface is the point at which a computer program meets the physical computer. Here the physical limitations on space, time, and the capabilities of a central processor must live with the objective of the program, namely, to compute a specified function or complete some specified task. This is a profitable point at which to study programs and computers, as we see in Chapter 7.

But let us challenge this hypothesis in several ways before we accept it as a reasonable approach to the understanding of complex computer systems. We ask whether the multiplicity of computer organizations (Bell and Newell, 1971, have identified more than 1000 different such organizations) precludes the development of knowledge that is useful in studying the majority of these organizations. Despite the differentiation that has been observed in the above-mentioned taxonomy, there is a surprising degree of commonality among computer systems, and they can be grouped into a small number of classes of like systems. Thus, the interface is less complex than it would seem from this point of view. Then there is complexity in computer programs. Here we have learned recently (Dahl et al., 1972) of the importance of structured programs, programs that observe a hierarchical structure. Such programs are constructed with a few simple rules, and when approached this way they are much easier to design, understand, prove correct, and make efficient. Programs organized or viewed in this manner lose a great deal of their complexity at the interface, where they are characterized primarily by their goals. Again, it seems reasonable to examine programs and computers at this level of abstraction.

Given that it is reasonable to examine computing systems at the interface between their inner and outer environments, we ask how we might characterize the interface. A clue is found in the opinion voiced by Minsky (1970) in his Turing lecture.

> We have had misconceptions about the possible exchanges between time and memory, tradeoffs between time and program complexity, software and hardware, digital and analog circuits, serial and parallel computations, associative and addressed memory, and so on.

We are led to believe by this statement and personal programming experience that a most natural characterization of the interface is in terms of tradeoff relations. Minsky develops analogies with the natural sciences and argues that "the recognition of exchanges is often the conception of a science, if quantifying them is its birth." We are almost ready to accept this point of view, but further clarification is needed.

Computing systems are not only made by men, they are used by men. Therefore, if the interface is described by exchange relations of the equality variety, such as the conservation laws of physics, whatever invariants characterize the interface must be satisfied by all users. Because of the variety in the human race, it is unlikely that any such equality relation of any significance can hold. Thus, we are led to expect exchange relations of the inequality variety, such as the second law of thermodynamics or the Heisenberg uncertainty principle. Here the interface is characterized by inequalities that define lower limits on exchanges of space for time, for example, and explicitly permit users and programs to operate above the lower limits. One expects that the limits embody the essentials of the interface—they should be stated in terms of physical properties of computers, such as space and time, and they should reflect the nature of the program's objective. Such exchange relations are derived in Chapter 7, and the inner environment is reflected in the complexity of the program objective as measured in two different ways.

The development of computational inequalities that define the interface between programs and computers requires considerable preparation. We develop models to represent machines and computations, and we define and examine complexity measures. And finally we introduce some new concepts in the process of deriving our inequalities. The development of these ideas and tools is interesting in its own right and has occupied scholars for decades. We outline these methods and concepts in succeeding sections.

1.2. COMPUTATIONAL MODELS

To study programs and computers at their interface we must develop models for computation. In this section, we give a brief introduction to the three models we use and indicate the roles they play in modeling general-purpose computers. The three models are logic circuits, sequential machines, and Turing machines. We see that the first and last of these models play a second role, namely, providing a basis for measuring the complexity of functions.

A logic circuit, which we also call a combinational machine, is an assemblage of logic elements each of which realizes a Boolean function. (Precise definitions of circuits and functions are given in Chapter 2.) We assume that the reader has some familiarity with logic circuits and with the two-input Boolean functions of AND, OR, and NOT. (Readers not satisfying this condition can proceed directly to Chapter 2.) Denoting AND and OR by $\cdot$ and $+$, respectively, in a circuit and NOT by a small circle, we now illustrate the concept of a logic circuit by example. We do this for two problems which are instances of important problems studied later.

Given an undirected graph on four nodes, we ask whether it contains a subset of three nodes such that every pair of nodes in the subset is connected by an edge. In a graph of n nodes, a subset of k nodes that satisfies this property is called a k-clique. The "k-clique problem," which is to determine for arbitrary n and $k \leq n$ whether an arbitrary graph on n nodes has a k-clique, is very difficult and is an example of an NP-complete problem (see Chapter 8). These problems are all either of exponential or polynomial running time, but their best known algorithms are exponential. The Traveling Salesman problem and the 0-1 integer programming problem belong to this class.

The 3-clique problem on four nodes, as with the more general problem, can be characterized by a Boolean function, a function on 0-1 valued variables whose value is 0 or 1. Figure 1.2.1*a* shows a graph on four nodes with six edges labeled $y_1, y_2, \ldots, y_6$. Here y_i is a variable that has value 1 if the indicated edge is present and is equal to 0 otherwise. Thus, $(y_1, y_2, \ldots, y_6)$ is a binary 6-tuple that characterizes a graph on four nodes. Such a graph has a 3-clique if for one of the following sets of three nodes, every pair of nodes is connected by an edge: $\{a, b, c\}$, $\{b, c, d\}$, $\{c, d, a\}$, $\{d, a, b\}$. This condition holds for $\{a, b, c\}$ if $y_1 = y_2 = y_6 = 1$, that is, if $y_1 \cdot y_2 \cdot y_6 = 1$ where $\cdot$ denotes AND. Similar products can be formed for each set of three nodes, and if $f_{3-cl}(y_1, y_2, \ldots, y_6)$ is a Boolean function that has value 1 if and only if $(y_1, y_2, \ldots, y_6)$ characterizes a graph with a 3-clique, f_{3-cl} can be realized as the OR of these four products. Figure 1.2.1*b* shows a logic circuit that realizes this function. It is not known whether this circuit has a minimal number of elements or how many elements are needed in general for the *k*-clique problem.

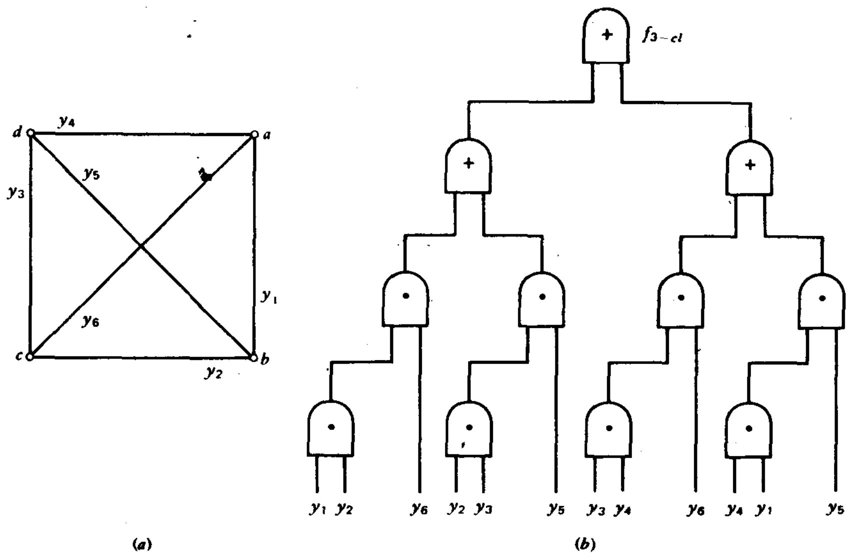
d
a
b
c
y_1
y_2
y_3
y_4
y_5
y_6
(a)
f_{3-cl}
+
+
+
y_1 y_2 y_6 y_2 y_3 y_5 y_3 y_4 y_6 y_4 y_1 y_5
(b)

Figure 1.2.1. Graph on four nodes and circuit for f_{3-cl}.

As a second example of problem and circuit consider whether $\mathbf{x} = (x_1, x_2)$ and $\mathbf{y} = (y_1, y_2)$, $x_i, y_i \in \{0, 1\}$, denoting integers $|\mathbf{x}| = x_1 \cdot 2 + x_0$ and $|\mathbf{y}| = y_1 \cdot 2 + y_0$ satisfy $|\mathbf{x}| > |\mathbf{y}|$. Let $f_{\text{COMP}}(\mathbf{x}, \mathbf{y})$ be the Boolean function defined as follows:

$$f_{\text{COMP}}(\mathbf{x}, \mathbf{y}) = \begin{cases} 1 & |\mathbf{x}| > |\mathbf{y}| \\ 0 & \text{otherwise} \end{cases}$$

This function is examined closely in Section 2.4.3. We observe that

$$f_{\text{COMP}}(\mathbf{x}, \mathbf{y}) = \begin{cases} 1 & \text{if } x_1 > y_1 \quad \text{or } x_1 = y_1 \quad \text{and } x_0 > y_0 \\ 0 & \text{otherwise} \end{cases}$$

Now if $\bar{\ }$ denotes NOT in a formula, then $x_1 \cdot \bar{y}_1$ is 1 when $x_1 > y_1$, and $x_1 \cdot y_1 + \bar{x}_1 \bar{y}_1$ is 1 when $x_1 = y_1$. Thus, f_{COMP} can be realized by the circuit shown in Figure 1.2.2. We show in Section 2.4.3 that this circuit has a minimal number of AND's and OR's.

We turn now to the second computational model, the sequential machine. A sequential machine has memory and feedback, and at any one point in time it is in one of several states. It is given an input, then makes a state transition and produces an output. The successor state is determined

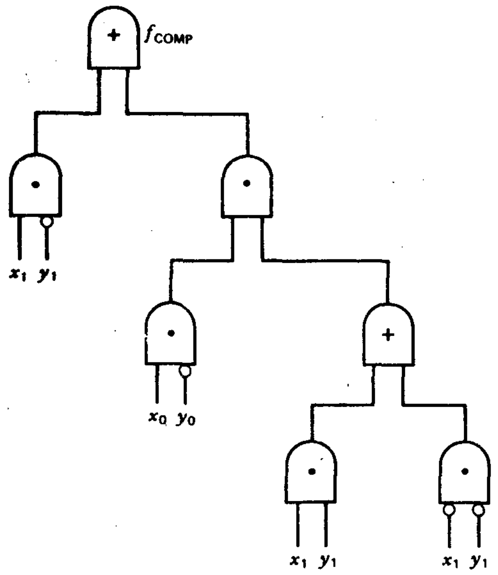

Figure 1.2.2. Circuit for f_{COMP}.

by the current state and input. Most sequential machines are clocked; that is, state transitions occur at time instants that are determined by a central clock. However, the definition of a sequential machine is not dependent on whether or not it is clocked. Sequential machines are the subject of Chapter 4.

Shown in Figure 1.2.3 are the state diagrams for two simple sequential machines. The states are numbered q_0, q_1, q_2, q_3, and the inputs are 0 and 1 for the first machine (*a*) and 0, 1, 2 for the second (*b*). The labels on arrows between states indicate the state transitions that are made on individual inputs; thus in state q_0 each machine moves to q_1 on an input of 1. The first machine produces an output of 0 in each state except q_3 in which an output of 1 is produced. If the initial state is q_0, the machine reaches this state only after receiving three 1's as input. Thus, it computes the threshold function of threshold 3.

The second machine in Figure 1.2.3 has three input symbols. Application of input 2 restores the machine to state q_0, which we take as the initial state. Otherwise the state advances on receipt of a 1, returning to q_0 on receiving a multiple of four 1's. Furthermore, the output associated with a state is a pair $\mathbf{a} = (a_1, a_0)$ which is the subscript of the state written in binary; that is, if $|\mathbf{a}| = a_1 2 + a_0$, then $\mathbf{a}$ is the output when the machine is in state $q_{|\mathbf{a}|}$. This sequential machine adds modulo 4 and can be used for three consecutive cycles and reset to q_0 in order to realize a Full Adder, a component in an adder for binary numbers (see Section 2.2).

An important point to be made here is that sequential machines compute functions, just as do logic circuits. However, since sequential

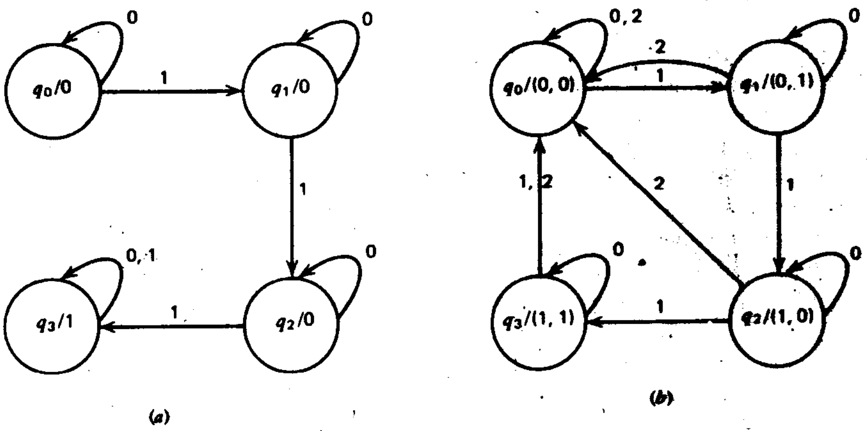

Figure 1.2.3. Two sequential machines.

machines use their memory to reuse their logic circuitry, they can realize functions with less circuitry than a no-memory machine but at the expense of time. This observation, which is elevated to the level of a theorem in Chapter 4, is one basis for the derivation of computational inequalities.

General-purpose computers come in many sizes and shapes, as indicated by the preceding. Nonetheless, they typically have the principal component shown in Figure 1.2.4. This consists of a central processing unit (CPU) and a random-access memory (RAM), the latter being a device holding an array of indexed words for which each word is accessible in one unit of time (the cycle of the RAM) by specifying its index or address. The CPU can address the RAM and store or fetch a word from it. It also executes instructions fetched from the RAM and has access to an outside world. The CPU typically can execute arithmetic operations and logical operations on words such as tests on the sign of a number, comparisons of numbers, and so forth. These operations are implemented in logic circuits, and the overall operation of the CPU and RAM can be modeled quite well as a pair of sequential machines. The component parts and functions of general-purpose computers are examined in considerable detail in Chapter 6.

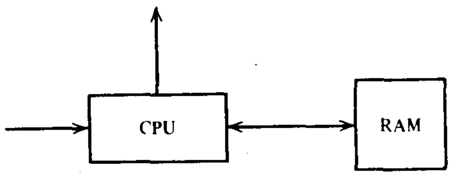

Figure 1.2.4. Model of a general-purpose computer.

Our third computational model is the Turing machine. This consists of a Control, which is a sequential machine, and one or more potentially infinite tapes the heads of which are driven by the Control. It is not an especially good model for general-purpose computers, but it is a classical computational model that explicitly permits unlimited storage. In terms of the functions it can compute, no more general model has been found and yet it cannot solve some easily described problems. The Turing machine, which is described in Chapter 5, provides a complexity measure that is a second basis for the derivation of computational inequalities.

1.3. COMPLEXITY MEASURES

We make use of two basic types of complexity measures, one related to the size and depth of logic circuits, and another that is the length of a