Great Ideas in Computer Science

A Gentle Introduction

Alan W. Biermann

Great Ideas in Computer Science

A Gentle Introduction

Alan W. Biermann

The MIT Press
Cambridge, Massachusetts
London, England

AEW 10/01

Preface

This is a book about computers—what they are, how they work, what they can do and what they cannot do. It is written for people who read about such topics as silicon chips or artificial intelligence and want to understand them, for people who need to have data processed on the job and want to know what can and cannot be done, and for people who see the proliferation of computers throughout society and ask about the meaning of it all. It is written for doctors, lawyers, preachers, teachers, managers, students, and all others who have a curiosity to learn about computing. It is also written for computer science students and professionals whose education may not have covered all of the important areas and who want to broaden themselves.

I was asked in 1985 to create a course in computer science for liberal arts students, and I decided that it was my job to present, as well as I could, the great intellectual achievements of the field. These are the "great ideas" that attract the attention of everyone who comes near and that, when collected together, comprise the heart of the field of computer science. I have spent considerable time in the succeeding years gathering together materials that capture these results.

What are the "great ideas" of computer science? The first and most important is the idea of the *algorithm*—a procedure or recipe that can be given to a person or machine for doing a job. The other great ideas revolve around this central one; they give methodologies for coding algorithms into machine readable form, and they describe what can be and what cannot be coded for machine execution. They show how concepts of everyday life can be meaningfully represented by electrical voltages and currents that are manipulated inside a machine, and they show how to build mechanisms to do these computations. They also show how to translate languages that people can use comfortably into languages that machines use so that the machine capabilites are accessible. They show how human-like reasoning processes can be programmed for machine execution, and they help us to understand what the ultimate capabilities of machines someday may become.

But it would seem that these great ideas are too complex and too technical to be understood by nonspecialists. Typically, a computer science major studies several years of mathematics and a long list of computer courses to learn these things, and we should not expect ordinary people to pick them up reading a single book. How can we meaningfully condense such extensive studies into a volume that many people can understand?

The answer is that the ideas must be reformulated in substantial ways, huge amounts of nonessential detail must be removed, and the vocabulary of the studies must be chosen carefully. Consider, for example, the traditional coverage of computer programming in a computer science curriculum. The student is taught all of the syntactic features of some programming language, numerous implementation details, and a variety of applications. We know that if we teach all of these things, there will be no time in the course for anything else. Students who want a broader view of the field than just programming will be frustrated. The treatment in this book teaches only a few of the features of Pascal, and all programs are restricted to those constructions. Most of the important lessons in programming can be taught within these limitations, and the reader's confusion from broad syntactic variety is eliminated. As another example, the traditional treatment of switching circuit design involves extensive study of Boolean algebra—equations, minimization, and circuit synthesis. But one can teach the most important ideas without any Boolean algebra at all. One can still address a design problem, write down a functional table for the target behavior, and create a nonminimal switching circuit to do the computation. The whole issue of circuit minimization that electrical engineers spend so much time on need not concern the general reader who simply wants to learn something about computers. Similar rather major revisions have been made to the traditional treatment of all computer topics. Thus we have Pascal without pointers, transistor theory without potential barriers, compilation without code optimization, computability theory without Turing machines, artificial intelligence without LISP, and so forth.

But all of these revisions have been made for a good reason. The goal is to give readers access to the essentials of the great ideas in one book. Readers learn to write a variety of programs in Pascal, design switching circuits, learn the essential mechanisms of transistor theory and their implementation in VLSI, study a variety of von Neumann and parallel architectures, hand simulate a compiler to see how it works, learn to classify various computations as tractable or intractable, gain an understanding of the concept of noncomputability, and come to grips with many of the important issues in artificial intelligence.

Of course, the presentation to nonspecialists must be done carefully in other ways as well. For example, it is important in the early chapters to introduce topics with motivating material. Each chapter on programming begins with a computational problem, and the lessons in programming are presented as a way of solving the problem. Also, much attention has been given to the choice of vocabulary. For example, computer scientists tend to use the word "move" in places where ordinary English speakers would say "copy." So the word "move" can cause confusion and has been banned from the book except where it is the only correct word. Dozens of other common vocabulary words have been similarly filtered where they might cause confusion, or they have been carefully defined when they are specifically required.

Another issue concerns the use of mathematical notations when the reader may not be experienced mathematically. The philosophy followed here is that such notations are essential to the study and that they must be included. The reason that a person reads a book in computer science is to study the field, and many of the things worth studying are notations. One of the great

benefits of the study may be that a person who feels alien to formal notations, in fact, may become quite comfortable with them. The book introduces relatively few notations, explains them in considerable detail, and uses them repetitively so the reader can become comfortable with them.

Finally, one might ask whether the "great ideas" presented here are the same as those that would be chosen by other authors. In fact, while one would expect some variations in opinions, there is considerable agreement about what constitutes the field of computer science. The central themes presented here are probably the same as would be chosen by most experts. As an illustration, one can examine the "intellectual framework for the discipline of computing" as given by the Task Force on the Core of Computer Science: "Computing as a Discipline" by Peter J. Denning (Chairman), Douglas E. Comer, David Gries, Michael C. Mulder, Allen Tucker, A. Joe Turner, and Paul Young.* This report presents a view of the field and makes recommendations related to proper computer science education. Among the contributions of the report is a description of nine subareas that the authors propose cover the field. They are

1. Algorithms and data structures
2. Programming languages
3. Architecture
4. Numerical and symbolic computation
5. Operating systems
6. Software methodology and engineering
7. Database and information retrieval systems
8. Artificial intelligence and robotics
9. Human-computer communication

This book presents an introduction to most of the nine subareas. The notable exceptions are "operating systems" and some issues related to "human-computer communication." Space limitations prevented the inclusion of operating systems except that they are mentioned, and their function is briefly described in chapter 10. Human-computer communication via natural language is discussed but no mention of computer graphics is included. But except for these few omissions, the contents of this book are in agreement with the views of the committee.

Instructors of classes may find that this volume covers more material than they can fit into a single course. In this case, coverage can be limited, for example, to the first two thirds of the book with only one or two lectures allocated at the end for overviewing advanced topics. This yields a course on programming and a study of how computers work. Another way to accelerate the study is to cover the switching, transistor studies, and VLSI in a single lecture, and then to spend about half of the course on chapters 9 through 14. There is a third way to use the book that is

---

* Peter J. Denning (Chairman), Douglas E. Comer, David Gries, Michael C. Mulder, Allen Tucker, A. Joe Turner, and Paul R. Young, "Computing as a Discipline," *Communications of the ACM*, Volume 32, Number 1, January, 1989; also in *Computer*, Volume 22, Number 2, February, 1989.

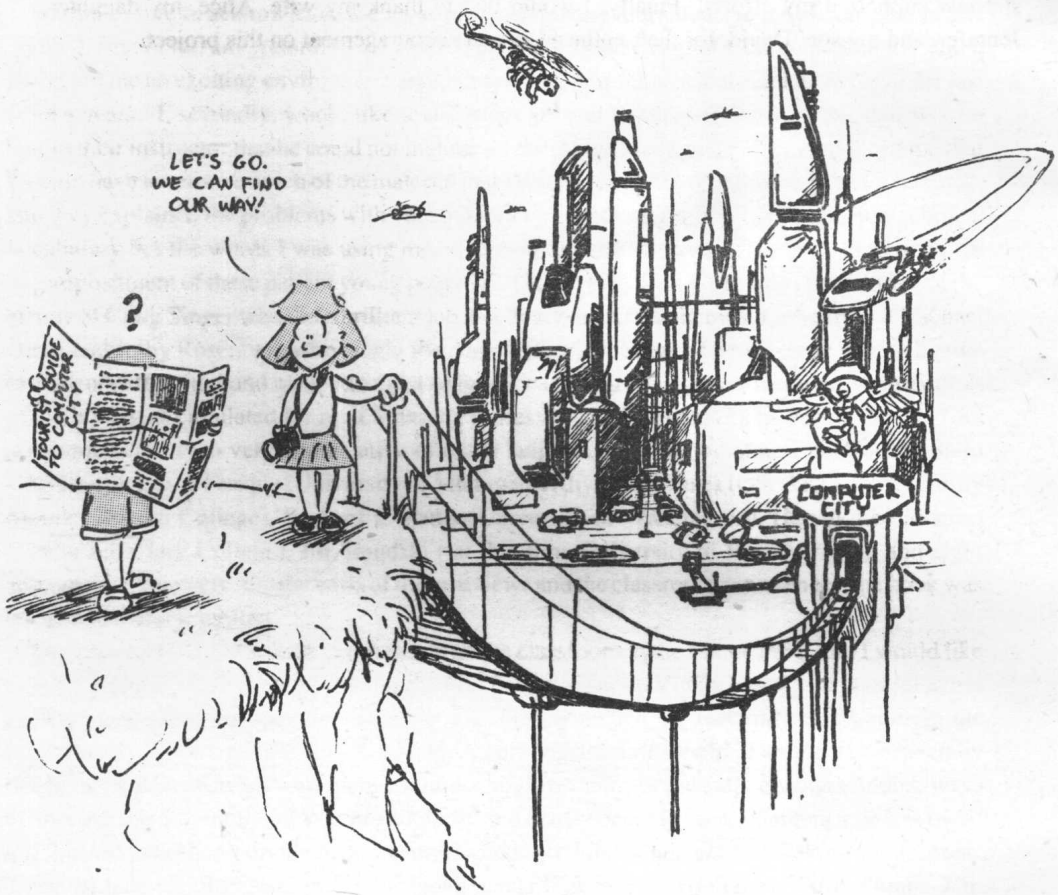
applicable when students have already learned programming from another source. In this case, coverage can begin at chapter 5 and proceed to the end. I have used many variations of these strategies in my own course. I usually do not cover recursion or the C-ranked sections of the translation and noncomputability chapters because they are difficult for my students.

It is a pleasure to acknowledge the contributions of many individuals to the preparation of this book. I, first of all, am grateful to the Duke University Department of Computer Science which has given me an exciting environment and plenty of support for scholarly endeavors over the last fifteen years. I, secondly, would like to thank my several hundred students in this course who taught their instructor that he could not include all the things he wanted. They convinced me that I would have to remove much of the material that I dearly loved if any of it were to be understood, and they explained the problems with vocabulary: I might think I was using simple nontechnical vocabulary but the words I was using meant something else to them. This book is as much an accomplishment of these patient young people as it is of mine. I am especially appreciative of the efforts of Craig Singer who did a brilliant job as a Teaching Assistant over two years and Michael Hines and Jothy Rosenberg who taught the course other semesters. These people were sensitive to student difficulties and made excellent suggestions for improving the coverage. An early draft of the book was circulated for review among professors at other institutions during the 1987-88 academic year. I am very appreciative of many helpful comments by Shan Chi (Northwestern University), David Frisque (University of Michigan), Rhys Price Jones (Oberlin College), Emily Moore (Grinnel College), Richard E. Pattis (University of Washington), Harvey Lee Shapiro (Lewis and Clark College), Jill Smudski (at that time University of Pennsylvania), and eight anonymous reviewers. On the basis of these reviews and the classroom experiences, the book was reorganized and rewritten.

The new version of the book was brought to the classroom in the fall of 1988, and I would like to thank Ronnie Smith for contributing the excellent chapter on VLSI. I am again grateful to my students who filled out questionnaires on four occasions helping me find weak points in the explanations and to my Teaching Assistant, Albert Nigrin, for his help. I would also especially like to thank Elina Kaplan who spent countless hours on some of the early chapters finding ways to improve the presentation. Where simplicity and clarity occur in these chapters, much is owed to Elina. Many other individuals have contributed by reading chapters and making suggestions. These include Heidi Brubaker, Dania Egedi, Linda Fineman, Chris Gandy, Curry Guinn, Tim Gegg-Harrison, Barry Koster, Anselmo Lastra, Ken Lang, Albert Nigrin, Lorrie Tomek, Tom Truscott, and Doreen Yen. I am especially appreciative of errors found and suggestions made by David M. Gordon, Henry Greenside, Donald Loveland, and Charlie Martin. Many other friends have made suggestions and commented on the chapters.

The book has been given much of its personality by Matt Evans who created the cartoons at the beginnings of the chapters. I am extremely appreciative of his efforts. I am tremendously indebted to Ann Davis who typed the manuscript from my handwritten pages. Her diligence and accuracy greatly eased the burden of creating the book. I would also like to thank Marie Cunningham for typing some of the chapters and Denita Thomas for preparing the index. Barry Koster was kind

enough to generate a large number of the figures, Eric Smith helped me on numerous occasions with library work, and Lorrie LeJeune of The MIT Press did the excellent job of typesetting. I would also like to express my heartfelt thanks to Robert Prior, Harry Stanton, and the other editors at The MIT Press who understood the dream of my book from the beginning and who have strongly supported my efforts. Finally, I would like to thank my wife, Alice, my daughter, Jennifer, and my son, David, for their enthusiasm and encouragement on this project.


LET'S GO.
WE CAN FIND
OUR WAY!
?
COMPUTER
CITY

Studying Academic Computer Science: An Introduction

Rumors

Computers are the subject of many rumors, and we wonder what to believe. People say that computers in the future will do all clerical jobs and even replace some well-trained experts. They say computers are beginning to simulate the human mind, to create art, to prove theorems, to learn, and to make careful judgments. They say that computers will permeate every aspect of our jobs and private lives by managing communication, manipulating information, and providing entertainment. They say that even our political systems will be altered—that in previously closed societies, computers will bring universal communication that will threaten the existing order, and in free societies, they will bring increased monitoring and control. On the other hand, there are skeptics who point out that computer science has many limitations and that the impact of machines has been overemphasized.

Some of these rumors are correct and give us fair warning of things to come. Others may be somewhat fanciful, leading us to worry about the future more than is necessary. Still others point out questions that we may argue about for years without finding answers. Whatever the case, we can be sure that there are many important issues related to computers that are of vital importance, and they are worth trying to understand.

We should study computer science and address these concerns. We should get our hands on a machine and try to make it go. We should control the machine; we should play with it; we should harness it; and most important, we should try to understand it. We should try to build insights from our limited experiences that will illuminate answers to our questions. We should try to arm ourselves with understanding because the computer age is upon us.

This book is designed to help people understand computers and computer science. It begins with a study of programming in the belief that using, controlling, and manipulating machines is an essential avenue to understanding them. Then it takes the reader on a guided tour of the machine internals, exploring all of its essential functioning from the tiniest movements of electrons through semiconductors to the architecture of the machine and the software that drives it. Finally, the book explores the limitations of computing, the frontiers of the science as they are currently understood.

In short, the book attempts to give a thorough introduction to the field with an emphasis on the fundamental mechanisms that enable computers to work. It presents many of the "great ideas" of computer science, the intellectual paradigms that scientists use to understand the field. These ideas provide the tools to help the reader comprehend and live with machines in the modern world.

Studying Computer Science

Computer science is the study of recipes and ways to carry them out. A *recipe* is a procedure or method for doing something. The science studies kinds of recipes, the properties of recipes, languages for writing them down, methods for creating them, and the construction of machines that will carry them out. Of course, computer scientists want to distinguish themselves from chefs, so they have their own name for recipes—they call them *algorithms*. But we will save most of the technical jargon for later.

If we wish to understand computer science, then we must study recipes, or algorithms. The first problem relates to how to conceive of them and how to write them down. For example, one might want a recipe for treating a disease, for classifying birds on the basis of their characteristics, or for organizing a financial savings program. We need to study some example recipes to see how they are constructed, and then we need practice writing our own. We need experience in abstracting the essence of real-world situations and in organizing this knowledge into a sequence of steps for getting our tasks done.

Once we have devised a method for doing something, we wish to *code* it in a computer language in order to communicate our desires to the machine. Thus, it is necessary to learn a computer language and to learn to translate the steps of a recipe into commands that can be carried out by a machine. This book will introduce a language called *Pascal* which is easy to learn and quite satisfactory for our example programs.

The combination of creating the recipe and coding it into a computer language is called *programming*, and this is the subject of the first third of the book, chapters 1 to 5. These chapters give a variety of examples of problem types, their associated solution methods, and the Pascal code, the *program*, required to solve them. The final chapter in the sequence discusses the problems related to scaling up the lessons learned here to industrial sized programming projects.

While the completion of the programming chapters leads to an ability to create useful code, the resulting level of understanding will still fall short of our deeper goals. The programmer's view of a computer is that it is a magic box that efficiently executes commands, and the internal mechanisms may remain a mystery. However, as scholars of computer science, we must know something of these mechanisms so that we can comprehend why a machine acts as it does, what its limitations are, and what improvements can be expected. The second third of the book addresses the issue of how and why computers are able to compute.

Chapter 6 shows methods for designing electric circuits and how to employ these techniques to design computational mechanisms. For example, the reader is shown how to build a circuit for adding numbers. Modern machines are constructed using semiconductor technologies, and chapters 7 and 8 tell how semiconductor devices operate and how they are assembled to produce

application circuits. Chapter 9 describes computer architecture and the organization of typical computers. Chapter 10 addresses the problem of translation of a high level computer language like Pascal into machine language, so that it can be run on the given architecture. An example at the end of chapter 10 traces the significant processing that occurs in the execution of a Pascal language statement from the translation to machine language, through the detailed operations of the computational circuits, to the migration of electrons through the semiconductors.

The final chapters of the book examine the limitations of computers and the frontiers of the science as it currently stands. Chapter 11 discusses problems related to program execution time and computations that require long processing times. Chapter 12 describes an attempt to speed up computers to do larger problems, the introduction of parallel architectures. Chapter 13 discusses the existence of so called *noncomputable* functions, and chapter 14 gives an introduction to the field of *artificial intelligence*.

An Approach for Nonmathematical Readers

A problem arises in the teaching of computer science in that the people who understand the field tend to speak their own language and use too much mathematical notation. The difficulties in communication lead the instructors to the conclusion that ordinary people are not able to understand the field. Thus, books and the university courses often skirt the central issues, and instead, teach the operation of software packages and the history and sociology of computing.

This book was written on the assumption that intelligent people can understand every fundamental issue of computer science if the preparation and explanation are adequate. No important topics have been omitted because of their difficulty. However, tremendous efforts were made to prune away unnecessary detail from the topics covered and to remove special vocabulary except where careful and complete definitions could be given.

Because casual readers may not wish to read all the chapters, the book is designed to encourage dabbling. Readers are encouraged to jump to any chapter at any time and read as much as is of interest. Of course, most chapters use some concepts gathered from earlier pages and where this occurs, understanding will be reduced. The programming chapters 1 through 4 are highly dependent on each other, and the architecture chapter (9) should be read before the translation chapter (10). Also, some of the advanced chapters (11 through 14) use concepts of programming from the early chapters (1 to 4). Except for these restrictions, the topics can probably be covered in any order without much sacrifice.

All of the chapter sections are classified as either A, B, or C again to encourage readers to taste much and devour only to the extent desired. Chapter sections labelled A include only introductory material and make few demands on the reader. One can get an overview of the book in a single evening by reading all the A sections. The B sections are the primary material of the book and may require substantial time and effort to read. The reader who completes the B material in a chapter will understand the major lessons on that topic and need feel no guilt about stopping at that point. The C material answers questions that careful readers may ask and supplements the main portions of the book.

Readings

For overview of computer science:

Brookshear, J. G., *Computer Science, An Overview,* Second Edition, Benjamin/Cummings Publishing Company, Menlo Park, California, 1988.

Goldschlager, L., and Lister, A., *Computer Science, A Modern Introduction,* Prentice-Hall, New York, 1988.

Schaffer, C., *Principles of Computer Science,* Prentice-Hall, Englewood Cliffs, New Jersey, 1988.

Pohl, I., and Shaw, A., *The Nature of Computation: An Introduction to Computer Science,* Computer Science Press, Rockville, Maryland, 1981.

For philosophical discussion:

Bolter, J. D., *Turing's Man,* University of North Carolina, Chapel Hill, North Carolina, 1984.

Contents

* Each section is labeled A, B, or C depending on whether it contains introductory, regular or optional material.

1

An Introduction to Programming: Coding Decision Trees

Good News (A)*

In the old days before computers, if we wanted to do a job, we had to do the job. But with computers, one can do many jobs by simply writing down what is to be done. A machine can do the work. If we want to add up some numbers, search for a given fact, carefully format and print a document, distribute messages to colleagues, control an industrial process, or other tasks, we can write down a recipe for what is to be done and walk away while a machine obediently and tirelessly carries out our instructions. If we wish, the machine will continue working while we sleep or go on vacation or do other jobs. Our recipe could even be distributed to many computers, and they could all work together, carrying out the instructions. Even after we retire from this life, computers may still be employed to do the same jobs following the commands that we laid down.

The preparation and writing of such "recipes" is called *programming*, and it implements a kind of "work amplification" that is revolutionizing the society of man. It enables a single person to do a finite amount of work, the preparation of a computer program, and to achieve, with the help of computers, an unbounded number of results. Thus, our productivity is no longer simply a function of the number of people; it is a function of the number of people and the number of machines we have.

There is even more good news: computers are relatively inexpensive, and their costs are continuously decreasing. Machines with 64,000 word memories and 1 microsecond instruction times cost $1 million two decades ago, $100,000 a few years ago, and $1000 or $2000 now. For the cost of one month of a laborer's time, we can purchase a machine that can do some tasks faster than a thousand people working together. The power and importance of the computer revolution are clear.

We wish to study computer programming in this book so that we can experience the work amplification that computers make possible. We will study programming by learning fundamental information structures and processing techniques. We will do problem solving using these ideas and develop expertise in abstracting the essence of problem situations into machine code so that our jobs can be done for us automatically.

* Each section is labeled A, B, or C depending on whether it contains introductory, regular or optional material.