# ALGORITHMICS
## Theory and Practice

Gilles Brassard and Paul Bratley

# ALGORITHMICS
## Theory and Practice

Gilles Brassard and Paul Bratley
*Département d'informatique et de recherche opérationnelle*
*Université de Montréal*

PRENTICE HALL, Englewood Cliffs, New Jersey 07632

Library of Congress Cataloging-in-Publication Data

Brassard, Gilles
Algorithmics : theory and practice.

1. Recursion theory. 2. Algorithms. I. Bratley, Paul. II. Title.
QA9.6.B73 1987 511'.3 88-2326
ISBN 0-13-023243-2

Editorial/production supervision: *Editing, Design & Production, Inc.*
Cover design: *Lundgren Graphics, Ltd.*
Manufacturing buyer: *Cindy Grant*

Printed in the United States of America
10 9 8 7 6 5 4 3 2 1

ISBN 0-13-023243-2

PRENTICE-HALL INTERNATIONAL (UK) LIMITED, *London*
PRENTICE-HALL OF AUSTRALIA PTY. LIMITED, *Sydney*
PRENTICE-HALL CANADA INC., *Toronto*
PRENTICE-HALL HISPANOAMERICANA, S.A., *Mexico*
PRENTICE-HALL OF INDIA PRIVATE LIMITED, *New Delhi*
PRENTICE-HALL OF JAPAN, INC., *Tokyo*
SIMON & SCHUSTER ASIA PTE. LTD., *Singapore*
EDITORA PRENTICE-HALL DO BRASIL, LTDA., *Rio de Janeiro*

# Preface

The explosion in computing we are witnessing is arousing extraordinary interest at every level of society. As the power of computing machinery grows, calculations once infeasible become routine. Another factor, however, has had an even more important effect in extending the frontiers of feasible computation: the use of efficient algorithms. For instance, today's typical medium-sized computers can easily sort 100,000 items in 30 seconds using a good algorithm, whereas such speed would be impossible, even on a machine a thousand times faster, using a more naive algorithm. There are other examples of tasks that can be completed in a small fraction of a second, but that would require millions of years with less efficient algorithms (read Section 1.7.3 for more detail).

The Oxford English Dictionary defines *algorithm* as an "erroneous refashioning of *algorism*" and says about *algorism* that it "passed through many pseudo-etymological perversions, including a recent *algorithm*". (This situation is not corrected in the OED Supplement.) Although the Concise Oxford Dictionary offers a more up-to-date definition for the word *algorithm*, quoted in the opening sentence of Chapter 1, we are aware of no dictionary of the English language that has an entry for *algorithmics*, the subject matter of this book.

We chose the word *algorithmics* to translate the more common French term *algorithmique*. (Although this word appears in some French dictionaries, the definition does not correspond to modern usage.) In a nutshell, algorithmics is the systematic study of the fundamental techniques used to design and analyse efficient algorithms. The same word was coined independently by several people, sometimes with slightly different meanings. For instance, Harel (1987) calls algorithmics "the spirit of computing", adopting the wider perspective that it is "the area of human study, knowledge and expertise that concerns algorithms".

Our book is neither a programming manual nor an account of the proper use of data structures. Still less is it a "cookbook" containing a long catalogue of programs ready to be used directly on a machine to solve certain specific problems, but giving at best a vague idea of the principles involved in their design. On the contrary, the aim of our book is to give the reader some basic tools needed to develop his or her own algorithms, in whatever field of application they may be required.

Thus we concentrate on the techniques used to design and analyse efficient algorithms. Each technique is first presented in full generality. Thereafter it is illustrated by concrete examples of algorithms taken from such different applications as optimization, linear algebra, cryptography, operations research, symbolic computation, artificial intelligence, numerical analysis, computing in the humanities, and so on. Although our approach is rigorous and theoretical, we do not neglect the needs of practitioners: besides illustrating the design techniques employed, most of the algorithms presented also have real-life applications.

To profit fully from this book, you should have some previous programming experience. However, we use no particular programming language, nor are the examples for any particular machine. This and the general, fundamental treatment of the material ensure that the ideas presented here will not lose their relevance. On the other hand, you should not expect to be able to use the algorithms we give directly: you will always be obliged to make the necessary effort to transcribe them into some appropriate programming language. The use of Pascal or similarly structured language will help reduce this effort to the minimum necessary.

Some basic mathematical knowledge is required to understand this book. Generally speaking, an introductory undergraduate course in algebra and another in calculus should provide sufficient background. A certain mathematical maturity is more important still. We take it for granted that the reader is familiar with such notions as mathematical induction, set notation, and the concept of a graph. From time to time a passage requires more advanced mathematical knowledge, but such passages can be skipped on the first reading with no loss of continuity.

Our book is intended as a textbook for an upper-level undergraduate or a lower-level graduate course in algorithmics. We have used preliminary versions at both the Université de Montréal and the University of California, Berkeley. If used as the basis for a course at the graduate level, we suggest that the material be supplemented by attacking some subjects in greater depth, perhaps using the excellent texts by Garey and Johnson (1979) or Tarjan (1983). Our book can also be used for independent study: anyone who needs to write better, more efficient algorithms can benefit from it. Some of the chapters, in particular the one concerned with probabilistic algorithms, contain original material.

It is unrealistic to hope to cover all the material in this book in an undergraduate course with 45 hours or so of classes. In making a choice of subjects, the teacher should bear in mind that the first two chapters are essential to understanding the rest of the book, although most of Chapter 1 can probably be assigned as independent reading. The other chapters are to a great extent independent of one another. An elementary course should certainly cover the first five chapters, without necessarily going over each and every example given there of how the techniques can be applied. The choice of the remaining material to be studied depends on the teacher's preferences and inclinations.The last three chapters, however, deal with more advanced topics; the teacher may find it interesting to discuss these briefly in an undergraduate class, perhaps to lay the ground before going into detail in a subsequent graduate class.

Each chapter ends with suggestions for further reading. The references from each chapter are combined at the end of the book in an extensive bibliography including well over 200 items. Although we give the origin of a number of algorithms and ideas, our primary aim is not historical. You should therefore not be surprised if information of this kind is sometimes omitted. Our goal is to suggest supplementary reading that can help you deepen your understanding of the ideas we introduce.

Almost 500 exercises are dispersed throughout the text. It is crucial to *read the problems*: their statements form an integral part of the text. Their level of difficulty is indicated as usual either by the absence of an asterisk (immediate to easy), or by the presence of one asterisk (takes a little thought) or two asterisks (difficult, maybe even a research project). The solutions to many of the difficult problems can be found in the references. No solutions are provided for the other problems, nor do we think it advisable to provide a solutions manual. We hope the serious teacher will be pleased to have available this extensive collection of unsolved problems from which homework assignments can be chosen. Several problems call for an algorithm to be implemented on a computer so that its efficiency may be measured experimentally and compared to the efficiency of alternative solutions. It would be a pity to study this material without carrying out at least one such experiment.

The first printing of this book by Prentice Hall is already in a sense a second edition. We originally wrote our book in French. In this form it was published by Masson, Paris. Although less than a year separates the first French and English printings, the experience gained in using the French version, in particular at an international summer school in Bayonne, was crucial in improving the presentation of some topics, and in spotting occasional errors. The numbering of problems and sections, however, is not always consistent between the French and English versions.

Writing this book would have been impossible without the help of many people. Our thanks go first to the students who have followed our courses in algorithmics over the years since 1979, both at the undergraduate and graduate levels. Particular thanks are due to those who kindly allowed us to copy their course notes: Denis Fortin, Laurent Langlois, and Sophie Monet in Montréal, and Luis Miguel and Dan Philip in Berkeley. We are also grateful to those people who used the preliminary versions of our book, whether they were our own students, or colleagues and students at other universities. The comments and suggestions we received were most valuable. Our warmest thanks, however, must go to those who carefully read and reread several chapters of the book and who suggested many improvements and corrections: Pierre

Beauchemin, André Chartier, Claude Crépeau, Bennett Fox, Claude Goutier, Pierre L'Ecuyer, Pierre McKenzie, Santiago Miro, Jean-Marc Robert, and Alan Sherman.

We are also grateful to those who made it possible for us to work intensively on our book during long periods spent away from Montréal. Paul Bratley thanks Georges Stamon and the Université de Franche-Comté. Gilles Brassard thanks Manuel Blum and the University of California, Berkeley, David Chaum and the CWI, Amsterdam, and Jean-Jacques Quisquater and Philips Research Laboratory, Bruxelles. He also thanks John Hopcroft, who taught him so much of the material included in this book, and Lise DuPlessis who so many times made her country house available; its sylvan serenity provided the setting and the inspiration for writing a number of chapters.

Denise St.-Michel deserves our special thanks. It was her misfortune to help us struggle with the text editing system through one translation and countless revisions. Annette Hall, of Editing, Design, and Production, Inc., was no less misfortuned to help us struggle with the last stages of production. The heads of the laboratories at the Université de Montréal's Département d'informatique et de recherche opérationnelle, Michel Maksud and Robert Gérin-Lajoie, provided unstinting support. We thank the entire team at Prentice Hall for their exemplary efficiency and friendliness; we particularly appreciate the help we received from James Fegen. We also thank Eugene L. Lawler for mentioning our French manuscript to Prentice Hall's representative in northern California, Dan Joraanstad, even before we plucked up the courage to work on an English version. The Natural Sciences and Engineering Research Council of Canada provided generous support.

Last but not least, we owe a considerable debt of gratitude to our wives, Isabelle and Pat, for their encouragement, understanding, and exemplary patience — in short, for putting up with us — while we were working on the French and English versions of this book.

Gilles Brassard
Paul Bratley

# Contents

# 1 Preliminaries

## 1.1 WHAT IS AN ALGORITHM?

The Concise Oxford Dictionary defines an algorithm as a "process or rules for (esp. machine) calculation". The execution of an algorithm must not include any subjective decisions, nor must it require the use of intuition or creativity (although we shall see an important exception to this rule in Chapter 8). When we talk about algorithms, we shall mostly be thinking in terms of computers. Nonetheless, other systematic methods for solving problems could be included. For example, the methods we learn at school for multiplying and dividing integers are also algorithms. The most famous algorithm in history dates from the time of the Greeks: this is Euclid's algorithm for calculating the greatest common divisor of two integers. It is even possible to consider certain cooking recipes as algorithms, provided they do not include instructions like "Add salt to taste".

When we set out to solve a problem, it is important to decide which algorithm for its solution should be used. The answer can depend on many factors: the size of the instance to be solved, the way in which the problem is presented, the speed and memory size of the available computing equipment, and so on. Take elementary arithmetic as an example. Suppose you have to multiply two positive integers using only pencil and paper. If you were raised in North America, the chances are that you will multiply the multiplicand successively by each figure of the multiplier, taken from right to left, that you will write these intermediate results one beneath the other shifting each line one place left, and that finally you will add all these rows to obtain your answer. This is the "classic" multiplication algorithm.

However, here is quite a different algorithm for doing the same thing, sometimes called "multiplication *à la russe*". Write the multiplier and the multiplicand side by side. Make two columns, one under each operand, by repeating the following rule until the number under the multiplier is 1: divide the number under the multiplier by 2, ignoring any fractions, and double the number under the multiplicand by adding it to itself. Finally, cross out each row in which the number under the multiplier is even, and then add up the numbers that remain in the column under the multiplicand. For example, multiplying 19 by 45 proceeds as in Figure 1.1.1. In this example we get $19+76+152+608 = 855$. Although this algorithm may seem funny at first, it is essentially the method used in the hardware of many computers. To use it, there is no need to memorize any multiplication tables: all we need to know is how to add up, and how to double a number or divide it by 2.

| | | |
|---|---|---|
| 45 | 19 | 19 |
| 22 | 38 | --- |
| 11 | 76 | 76 |
| 5 | 152 | 152 |
| 2 | 304 | ----- |
| 1 | 608 | 608 |
| | | 855 |

**Figure 1.1.1.** Multiplication *à la russe*.

We shall see in Section 4.7 that there exist more efficient algorithms when the integers to be multiplied are very large. However, these more sophisticated algorithms are in fact slower than the simple ones when the operands are not sufficiently large.

At this point it is important to decide how we are going to *represent* our algorithms. If we try to describe them in English, we rapidly discover that natural languages are not at all suited to this kind of thing. Even our description of an algorithm as simple as multiplication *à la russe* is not completely clear. We did not so much as try to describe the classic multiplication algorithm in any detail. To avoid confusion, we shall in future specify our algorithms by giving a corresponding *program*. However, we shall not confine ourselves to the use of one particular programming language: in this way, the essential points of an algorithm will not be obscured by the relatively unimportant programming details.

We shall use phrases in English in our programs whenever this seems to make for simplicity and clarity. These phrases should not be confused with comments on the program, which will always be enclosed within braces. Declarations of scalar quantities (integer, real, or Boolean) are usually omitted. Scalar parameters of functions and procedures are passed by value unless a different specification is given explicitly, and arrays are passed by reference.

The notation used to specify that a function or a procedure has an array parameter varies from case to case. Sometimes we write, for instance

```
procedure proc1(T : array)
```

or even

**procedure** *proc*2(*T*)

if the type and the dimensions of the array $T$ are unimportant or if they are evident from the context. In such a case $\#T$ denotes the number of elements in the array $T$. If the bounds or the type of $T$ are important, we write

**procedure** *proc*3($T[1..n]$)

or more generally

**procedure** *proc*4($T[a..b]$ : *integers*) .

In such cases $n$, $a$, and $b$ should be considered as formal parameters, and their values are determined by the bounds of the actual parameter corresponding to $T$ when the procedure is called. These bounds can be specified explicitly, or changed, by a procedure call of the form

*proc*3($T[1..m]$) .

To avoid proliferation of **begin** and **end** statements, the range of a statement such as **if**, **while**, or **for**, as well as that of a declaration such as **procedure**, **function**, or **record**, is shown by indenting the statements affected. The statement **return** marks the dynamic end of a procedure or a function, and in the latter case it also supplies the value of the function. The operators **div** and **mod** represent integer division (discarding any fractional result) and the remainder of a division, respectively. We assume that the reader is familiar with the concepts of recursion and of pointers. The latter are denoted by the symbol "↑". A reader who has some familiarity with Pascal, for example, will have no difficulty understanding the notation used to describe our algorithms. For instance, here is a formal description of multiplication *à la russe*.

```
function russe (A, B)
  arrays X, Y
  {initialization}
  X[1] ← A ; Y[1] ← B
  i ← 1
  {make the two columns}
  while X[i] > 1 do
    X[i+1] ← X[i] div 2
    Y[i+1] ← Y[i] + Y[i]
    i ← i + 1
  {add the appropriate entries}
  prod ← 0
  while i > 0 do
    if X[i] is odd then prod ← prod + Y[i]
    i ← i − 1
  return prod
```

If you are an experienced programmer, you will probably have noticed that the arrays $X$ and $Y$ are not really necessary, and that this program could easily be simplified. However, we preferred to follow blindly the preceding description of the algorithm, even if this is more suited to a calculation using pencil and paper than to computation on a machine. The following APL program describes exactly the same algorithm (although you might reasonably object to a program using logarithms, exponentiation, and multiplication by powers of 2 to describe an algorithm for multiplying two integers ...).

```
    ∇ R←A RUSAPL B;T
[1] R←+/(2|⌊A÷T)/B×T←1,2*⍳⌊2*A ∇
```

On the other hand, the following program, despite a superficial resemblance to the one given previously, describes quite a different algorithm.

```
function not-russe (A, B)
   arrays X, Y
   {initialization}
   X[1] ← A ; Y[1] ← B
   i ← 1
   {make the two columns}
   while X[i] > 1 do
      X[i+1] ← X[i] − 1
      Y[i+1] ← B
      i ← i + 1
   {add the appropriate entries}
   prod ← 0
   while i > 0 do
      if X[i] > 0 then prod ← prod + Y[i]
      i ← i − 1
   return prod
```

We see that different algorithms can be used to solve the same problem, and that different programs can be used to describe the same algorithm. It is important not to lose sight of the fact that in this book we are interested in *algorithms*, not in the *programs* used to describe them.

## 1.2 PROBLEMS AND INSTANCES

Multiplication *à la russe* is not just a way to multiply 45 by 19. It gives a general solution to the *problem* of multiplying positive integers. We say that (45, 19) is an *instance* of this problem. Most interesting problems include an infinite collection of instances. Nonetheless, we shall occasionally consider finite problems such as that of playing a perfect game of chess. An algorithm must work correctly on every instance of the problem it claims to solve. To show that an algorithm is incorrect, we need only find one instance of the problem for which it is unable to find a correct answer. On the other hand, it is usually more difficult to prove the correctness of an algorithm. When we specify a problem, it is important to define its *domain of definition*, that is, the set of instances to be considered. Although multiplication *à la russe* will not work if the first operand is negative, this does not invalidate the algorithm since (−45, 19) is *not* an instance of the problem being considered.

Any real computing device has a limit on the size of the instances it can handle. However, this limit cannot be attributed to the algorithm we choose to use. Once again we see that there is an essential difference between programs and algorithms.

## 1.3 THE EFFICIENCY OF ALGORITHMS

When we have a problem to solve, it is obviously of interest to find several algorithms that might be used, so we can choose the best. This raises the question of how to decide which of several algorithms is preferable. The *empirical* (or a posteriori) approach consists of programming the competing algorithms and trying them on different instances with the help of a computer. The *theoretical* (or a priori) approach, which we favour in this book, consists of determining mathematically the quantity of resources (execution time, memory space, etc.) needed by each algorithm *as a function of the size of the instances considered.*

The *size* of an instance $x$, denoted by $|x|$, corresponds formally to the number of bits needed to represent the instance on a computer, using some precisely defined and reasonably compact encoding. To make our analyses clearer, however, we often use the word "size" to mean any integer that in some way measures the number of components in an instance. For example, when we talk about sorting (see Section 1.7.1), an instance involving $n$ items is generally considered to be of size $n$, even though each item would take more than one bit when represented on a computer. When we talk about numerical problems, we sometimes give the efficiency of our algorithms in terms of the *value* of the instance being considered, rather than its size (which is the number of bits needed to represent this value in binary).

The advantage of the theoretical approach is that it depends on neither the computer being used, nor the programming language, nor even the skill of the programmer. It saves both the time that would have been spent needlessly programming an inefficient algorithm and the machine time that would have been wasted testing it. It also allows us to study the efficiency of an algorithm when used on instances of any size. This is often not the case with the empirical approach, where practical considerations often force us to test our algorithms only on instances of moderate size. This last point is particularly important since often a newly discovered algorithm may only begin to perform better than its predecessor when both of them are used on large instances.

It is also possible to analyse algorithms using a *hybrid* approach, where the form of the function describing the algorithm's efficiency is determined theoretically, and then any required numerical parameters are determined empirically for a particular