Algorithms and Complexity

Herbert S. Wilf
University of Pennsylvania
Philadelphia, PA 19104-6395

Internet Edition, Summer, 1994

This edition of Algorithms and Complexity is the file "`pub/wilf/AlgComp.ps.Z`" at the anonymous ftp site "`ftp.cis.upenn.edu`". It may be taken at no charge by all interested persons. Comments and corrections are welcome, and should be sent to `wilf@central.cis.upenn.edu`

CONTENTS

Chapter 0: What This Book Is About

0.1 Background 1
0.2 Hard vs. easy problems 2
0.3 A preview 4

Chapter 1: Mathematical Preliminaries

1.1 Orders of magnitude 5
1.2 Positional number systems 11
1.3 Manipulations with series 14
1.4 Recurrence relations 16
1.5 Counting 21
1.6 Graphs 24

Chapter 2: Recursive Algorithms

2.1 Introduction 30
2.2 Quicksort 31
2.3 Recursive graph algorithms 38
2.4 Fast matrix multiplication 47
2.5 The discrete Fourier transform 50
2.6 Applications of the FFT 56
2.7 A review 60

Chapter 3: The Network Flow Problem

3.1 Introduction 63
3.2 Algorithms for the network flow problem 64
3.3 The algorithm of Ford and Fulkerson 65
3.4 The max-flow min-cut theorem 69
3.5 The complexity of the Ford-Fulkerson algorithm 70
3.6 Layered networks 72
3.7 The MPM Algorithm 76
3.8 Applications of network flow 77

Chapter 4: Algorithms in the Theory of Numbers

4.1 Preliminaries 81
4.2 The greatest common divisor 82
4.3 The extended Euclidean algorithm 85
4.4 Primality testing 87
4.5 Interlude: the ring of integers modulo n 89
4.6 Pseudoprimality tests 92
4.7 Proof of goodness of the strong pseudoprimality test
4.8 Factoring and cryptography 97
4.9 Factoring large integers 99
4.10 Proving primality 100

Chapter 5: NP-completeness

5.1 Introduction 104
5.2 Turing machines 109
5.3 Cook's theorem 112
5.4 Some other NP-complete problems 116
5.5 Half a loaf 119
5.6 Backtracking (I): independent sets 122
5.7 Backtracking (II): graph coloring 124
5.8 Approximate algorithms for hard problems 128

Preface

For the past several years mathematics majors in the computing track at the University of Pennsylvania have taken a course in continuous algorithms (numerical analysis) in the junior year, and in discrete algorithms in the senior year. This book has grown out of the senior course as I have been teaching it recently. It has also been tried out on a large class of computer science and mathematics majors, including seniors and graduate students, with good results.

Selection by the instructor of topics of interest will be very important, because normally I've found that I can't cover anywhere near all of this material in a semester. A reasonable choice for a first try might be to begin with Chapter 2 (recursive algorithms) which contains lots of motivation. Then, as new ideas are needed in Chapter 2, one might delve into the appropriate sections of Chapter 1 to get the concepts and techniques well in hand. After Chapter 2, Chapter 4, on number theory, discusses material that is extremely attractive, and surprisingly pure and applicable at the same time. Chapter 5 would be next, since the foundations would then all be in place. Finally, material from Chapter 3, which is rather independent of the rest of the book, but is strongly connected to combinatorial algorithms in general, might be studied as time permits.

Throughout the book there are opportunities to ask students to write programs and get them running. These are not mentioned explicitly, with a few exceptions, but will be obvious when encountered. Students should all have the experience of writing, debugging, and using a program that is nontrivially recursive, for example. The concept of recursion is subtle and powerful, and is helped a lot by hands-on practice. Any of the algorithms of Chapter 2 would be suitable for this purpose. The recursive graph algorithms are particularly recommended since they are usually quite foreign to students' previous experience and therefore have great learning value.

In addition to the exercises that appear in this book, then, student assignments might consist of writing occasional programs, as well as delivering reports in class on assigned readings. The latter might be found among the references cited in the bibliographies in each chapter.

I am indebted first of all to the students on whom I worked out these ideas, and second to a number of colleagues for their helpful advice and friendly criticism. Among the latter I will mention Richard Brualdi, Daniel Kleitman, Albert Nijenhuis, Robert Tarjan and Alan Tucker. For the no-doubt-numerous shortcomings that remain, I accept full responsibility.

This book was typeset in TEX. To the extent that it's a delight to look at, thank TEX. For the deficiencies in its appearance, thank my limitations as a typesetter. It was, however, a pleasure for me to have had the chance to typeset my own book. My thanks to the Computer Science department of the University of Pennsylvania, and particularly to Aravind Joshi, for generously allowing me the use of TEX facilities.

Herbert S. Wilf

Chapter 0: What This Book Is About

0.1 Background

An algorithm is a method for solving a class of problems on a computer. The complexity of an algorithm is the cost, measured in running time, or storage, or whatever units are relevant, of using the algorithm to solve one of those problems.

This book is about algorithms and complexity, and so it is about methods for solving problems on computers and the costs (usually the running time) of using those methods.

Computing takes time. Some problems take a very long time, others can be done quickly. Some problems *seem* to take a long time, and then someone discovers a faster way to do them (a 'faster algorithm'). The study of the amount of computational effort that is needed in order to perform certain kinds of computations is the study of computational *complexity*.

Naturally, we would expect that a computing problem for which millions of bits of input data are required would probably take longer than another problem that needs only a few items of input. So the time complexity of a calculation is measured by expressing the running time of the calculation *as a function of* some measure of the amount of data that is needed to describe the problem to the computer.

For instance, think about this statement: 'I just bought a matrix inversion program, and it can invert an $n \times n$ matrix in just $1.2n^3$ minutes.' We see here a typical description of the complexity of a certain algorithm. The running time of the program is being given as a function of the size of the input matrix.

A faster program for the same job might run in $0.8n^3$ minutes for an $n \times n$ matrix. If someone were to make a really important discovery (see section 2.4), then maybe we could actually lower the exponent, instead of merely shaving the multiplicative constant. Thus, a program that would invert an $n \times n$ matrix in only $7n^{2.8}$ minutes would represent a striking improvement of the state of the art.

For the purposes of this book, a computation that is guaranteed to take at most cn^3 time for input of size n will be thought of as an 'easy' computation. One that needs at most n^{10} time is also easy. If a certain calculation on an $n \times n$ matrix were to require 2^n minutes, then that would be a 'hard' problem. Naturally some of the computations that we are calling 'easy' may take a very long time to run, but still, from our present point of view the important distinction to maintain will be the polynomial time guarantee or lack of it.

The general rule is that if the running time is at most a polynomial function of the amount of input data, then the calculation is an easy one, otherwise it's hard.

Many problems in computer science are known to be easy. To convince someone that a problem is easy, it is enough to describe a fast method for solving that problem. To convince someone that a problem is hard is hard, because you will have to prove to them that it is *impossible* to find a fast way of doing the calculation. It will *not* be enough to point to a particular algorithm and to lament its slowness. After all, *that* algorithm may be slow, but maybe there's a faster way.

Matrix inversion is easy. The familiar Gaussian elimination method can invert an $n \times n$ matrix in time at most cn^3.

To give an example of a hard computational problem we have to go far afield. One interesting one is called the 'tiling problem.' Suppose* we are given infinitely many identical floor tiles, each shaped like a regular hexagon. Then we can tile the whole plane with them, *i.e.*, we can cover the plane with no empty spaces left over. This can also be done if the tiles are identical rectangles, but not if they are regular pentagons.

In Fig. 0.1 we show a tiling of the plane by identical rectangles, and in Fig. 0.2 is a tiling by regular hexagons.

That raises a number of theoretical and computational questions. One computational question is this. Suppose we are given a certain polygon, not necessarily regular and not necessarily convex, and suppose we have infinitely many identical tiles in that shape. Can we or can we not succeed in tiling the whole plane?

That elegant question has been *proved** to be computationally unsolvable. In other words, not only do we not know of any fast way to solve that problem on a computer, it has been *proved* that there isn't *any*

---

* See, for instance, Martin Gardner's article in *Scientific American*, January 1977, pp. 110-121.

* R. Berger, The undecidability of the domino problem, *Memoirs Amer. Math. Soc.* **66** (1966), Amer.

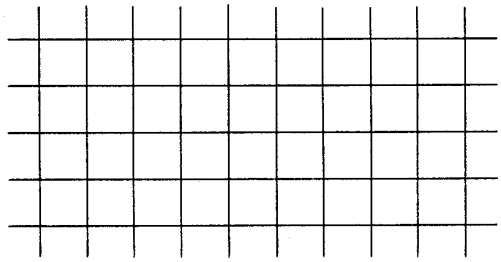

Fig. 0.1: Tiling with rectangles

Fig. 0.2: Tiling with hexagons

way to do it, so even looking for an algorithm would be fruitless. That doesn't mean that the question is hard for every polygon. Hard problems can have easy instances. What has been proved is that no single method exists that can guarantee that it will decide this question for every polygon.

The fact that a computational *problem* is hard doesn't mean that every instance of it has to be hard. The *problem* is hard because we cannot devise an algorithm for which we can give a *guarantee* of fast performance for *all* instances.

Notice that the amount of input data to the computer in this example is quite small. All we need to input is the shape of the basic polygon. Yet not only is it impossible to devise a fast algorithm for this problem, it has been proved impossible to devise any algorithm at all that is guaranteed to terminate with a Yes/No answer after finitely many steps. That's *really* hard!

0.2 Hard vs. easy problems

Let's take a moment more to say in another way exactly what we mean by an 'easy' computation vs. a 'hard' one.

Think of an algorithm as being a little box that can solve a certain class of computational problems. Into the box goes a description of a particular problem in that class, and then, after a certain amount of time, or of computational effort, the answer appears.

A 'fast' algorithm is one that carries a guarantee of fast performance. Here are some examples.

Example 1. *It is guaranteed that if the input problem is described with B bits of data, then an answer will be output after at most $6B^3$ minutes.*

Example 2. *It is guaranteed that every problem that can be input with B bits of data will be solved in at most $0.7B^{15}$ seconds.*

A performance guarantee, like the two above, is sometimes called a 'worst-case complexity estimate,' and it's easy to see why. If we have an algorithm that will, for example, sort any given sequence of numbers into ascending order of size (see section 2.2) it may find that some sequences are easier to sort than others.

For instance, the sequence 1, 2, 7, 11, 10, 15, 20 is nearly in order already, so our algorithm might, if it takes advantage of the near-order, sort it very rapidly. Other sequences might be a lot harder for it to handle, and might therefore take more time.

Math. Soc., Providence, RI

So in some problems whose input bit string has B bits the algorithm might operate in time $6B$, and on others it might need, say, $10B \log B$ time units, and for still other problem instances of length B bits the algorithm might need $5B^2$ time units to get the job done.

Well then, what would the warranty card say? It would have to pick out the worst possibility, otherwise the guarantee wouldn't be valid. It would assure a user that if the input problem instance can be described by B bits, then an answer will appear after at most $5B^2$ time units. Hence a performance guarantee is equivalent to an estimation of the worst possible scenario: the longest possible calculation that might ensue if B bits are input to the program.

Worst-case bounds are the most common kind, but there are other kinds of bounds for running time. We might give an *average* case bound instead (see section 5.7). That wouldn't *guarantee* performance no worse than so-and-so; it would state that if the performance is averaged over all possible input bit strings of B bits, then the average amount of computing time will be so-and-so (as a function of B).

Now let's talk about the difference between easy and hard computational problems and between fast and slow algorithms.

A warranty that would *not* guarantee 'fast' performance would contain some function of B that grows faster than *any* polynomial. Like e^B, for instance, or like $2^{\sqrt{B}}$, etc. *It is the polynomial time vs. not necessarily polynomial time guarantee that makes the difference between the easy and the hard classes of problems, or between the fast and the slow algorithms.*

It is highly desirable to work with algorithms such that we can give a performance guarantee for their running time that is at most a polynomial function of the number of bits of input.

An algorithm is *slow* if, whatever polynomial P we think of, there exist arbitrarily large values of B, and input data strings of B bits, that cause the algorithm to do more than $P(B)$ units of work.

A computational problem is *tractable* if there is a fast algorithm that will do all instances of it.

A computational problem is *intractable* if it can be proved that there is no fast algorithm for it.

Example 3. Here is a familiar computational problem and a method, or algorithm, for solving it. Let's see if the method has a polynomial time guarantee or not.

The problem is this. Let n be a given integer. We want to find out if n is *prime*. The method that we choose is the following. For each integer $m = 2, 3, \ldots, \lfloor \sqrt{n} \rfloor$ we ask if m divides (evenly into) n. If all of the answers are 'No,' then we declare n to be a prime number, else it is composite.

We will now look at the *computational complexity* of this algorithm. That means that we are going to find out how much work is involved in doing the test. For a given integer n the work that we have to do can be measured in units of divisions of a whole number by another whole number. In those units, we obviously will do about $\sqrt{n}$ units of work.

It seems as though this is a tractable problem, because, after all, $\sqrt{n}$ is of polynomial growth in n. For instance, we do less than n units of work, and that's certainly a polynomial in n, isn't it? So, according to our definition of fast and slow algorithms, the distinction was made on the basis of polynomial vs. faster-than-polynomial growth of the work done with the problem size, and therefore this problem must be easy. Right? Well no, not really.

Reference to the distinction between fast and slow methods will show that we have to measure the amount of work done *as a function of the number of bits of input to the problem.* In this example, n is not the number of bits of input. For instance, if $n = 59$, we don't need 59 bits to describe n, but only 6. In general, the number of binary digits in the bit string of an integer n is close to $\log_2 n$.

So in the problem of this example, testing the primality of a given integer n, the length of the input bit string B is about $\log_2 n$. Seen in this light, the calculation suddenly seems very long. A string consisting of a mere $\log_2 n$ 0's and 1's has caused our mighty computer to do about $\sqrt{n}$ units of work.

If we express the amount of work done as a function of B, we find that the complexity of this calculation is approximately $2^{B/2}$, and that grows much faster than any polynomial function of B.

Therefore, the method that we have just discussed for testing the primality of a given integer is slow. See chapter 4 for further discussion of this problem. At the present time no one has found a fast way to test for primality, nor has anyone proved that there isn't a fast way. Primality testing belongs to the (well-populated) class of seemingly, but not provably, intractable problems. ■

In this book we will deal with some easy problems and some seemingly hard ones. It's the 'seemingly' that makes things very interesting. These are problems for which no one has found a fast computer algorithm,

but also, no one has proved the impossibility of doing so. It should be added that the entire area is vigorously being researched because of the attractiveness and the importance of the many unanswered questions that remain.

Thus, even though we just don't know many things that we'd like to know in this field , it isn't for lack of trying!

0.3 A preview

Chapter 1 contains some of the mathematical background that will be needed for our study of algorithms. It is not intended that reading this book or using it as a text in a course must necessarily begin with Chapter 1. It's probably a better idea to plunge into Chapter 2 directly, and then when particular skills or concepts are needed, to read the relevant portions of Chapter 1. Otherwise the definitions and ideas that are in that chapter may seem to be unmotivated, when in fact motivation in great quantity resides in the later chapters of the book.

Chapter 2 deals with recursive algorithms and the analyses of their complexities.

Chapter 3 is about a problem that seems as though it might be hard, but turns out to be easy, namely the network flow problem. Thanks to quite recent research, there are fast algorithms for network flow problems, and they have many important applications.

In Chapter 4 we study algorithms in one of the oldest branches of mathematics, the theory of numbers. Remarkably, the connections between this ancient subject and the most modern research in computer methods are very strong.

In Chapter 5 we will see that there is a large family of problems, including a number of very important computational questions, that are bound together by a good deal of structural unity. We don't know if they're hard or easy. We do know that we haven't found a fast way to do them yet, and most people suspect that they're hard. We also know that if any one of these problems is hard, then they all are, and if any one of them is easy, then they all are.

We hope that, having found out something about what people know and what people don't know, the reader will have enjoyed the trip through this subject and may be interested in helping to find out a little more.