

# The Complexity of Algorithms

M. D. Atkinson

School of Mathematical and Computational Sciences  
North Haugh, St Andrews, Fife KY16 9SS

## Abstract

The modern theory of algorithms dates from the late 1960s when the method of asymptotic execution time measurement began to be used. It is argued that the subject has both an engineering and scientific wing. The engineering wing consists of well-understood design methodologies while the scientific wing is concerned with theoretical underpinnings. The key issues of both wings are surveyed. Finally some personal opinions on where the subject will go next are offered.

## 1 Introduction

The concept of “algorithm” is the oldest concept in Computer Science. The concept is of such antiquity in fact that it may appear presumptuous for Computer Scientists to claim it as part of their subject. However, although algorithms have been part of scientific tradition for millennia it was not until the computer era that they assumed the major role they now play in these traditions. Indeed, until a few decades ago most scientists would have been hard-pressed to give significant examples of any algorithms at all. Yet algorithms have been routinely used for centuries. Generations of school children learnt the algorithms for addition, subtraction, multiplication, and division and could use them with all the blinkered competence of a mechanical computer. Few of them would ever have stopped to wonder how it was that they allowed the computation, in a matter of moments, of quantities far beyond what could be counted or imagined. These elementary algorithms are amazingly efficient but it was not until the computer era that questions of efficiency were seriously addressed. Had they been so addressed the mathematical curriculum might have been different. For example, I can remember, as a schoolboy in the 1950s, being told how to find the greatest common divisor of two integers by finding their prime decompositions and taking the intersection of these decompositions; it was only years later that I was able to understand that this method was exponentially worse than Euclid’s algorithm which, for all its antiquity, I did not meet until graduate student days.

Nowadays this ignorance of efficiency issues has largely disappeared. For while Computer Scientists were not present at the birth they have, in the last 25 years, witnessed the childhood, adolescence, and early maturity of a subject that is now a central, some would say the central, part of computer science - the complexity of algorithms. This subject embraces the foundational questions of what efficiency means and how to measure it, the technical apparatus for designing efficient algorithms, and the classification of problems according to how efficiently they can be solved.

This chapter is about the scientific theory of algorithms but it is worth mentioning that algorithms have also had a cultural impact. The ubiquity of computational techniques has brought an awareness of the idea of algorithm to many disciplines and it is now recognised that the notion of a process scripted by a set of rules is useful beyond those processes which are scripted by computer programs. At a trivial level it embraces the idea of a boss giving instructions to a subordinate. More seriously it encompasses the causing of mechanical devices to perform their due functions through appropriate button presses and lever movements. For example, driving a car (once a few basic mechanical skills have been learnt) is essentially the diligent pursuing of the Highway Code prescriptions - and, generally, the more diligent the pursuit the more effective the outcome (*pace*, all teenage male firebrands!). Of course there are also many examples of machines in manufacturing and design which one programs to make various industrial components. Although these machines may not have the power of a universal Turing machine they are, nevertheless, carrying out algorithms. It is now common to hear terminology such as “This recipe is a good algorithm for making cakes” or “This machine is carrying out its wash, rinse, and spin dry algorithm”. The notion of algorithm has become part of our culture and for this the development of algorithms in computer science is largely responsible.

The key idea that supports most of the theory of algorithms is the method of quantifying the execution time of an algorithm. Execution time depends on many parameters which are independent of the particular algorithm: machine clock rate, quality of code produced by compiler, whether or not the computer is multi-programmed etc. Changing these parameters changes the execution time by a factor which will be common to every algorithm. Clearly, when we are trying to quantify the efficiency of an algorithm we should ignore these factors and measure what remains. Not surprisingly the execution time of an algorithm is a function of the values of its input parameters. If we know this function (at least, to within a scalar multiple determined by the machine parameters) then we can predict the time the algorithm will require without running it. (We might not wish to run it until we knew that it would complete in a reasonable time). Unfortunately this execution time function measure is generally very complicated, too complicated to know exactly. But now a number of observations come to the rescue. The first is that is very often possible to get meaningful predictive results by concentrating on one parameter alone: the total size of the input (rather than the precise values of each input parameter). The second is that

execution time functions are often the sum of a large number of terms and many of these terms make an insignificant contribution to function values: in mathematical terms, it is the asymptotic form of the function that matters rather than the precise form. The final observation is that, in practice, we wish only to know either an upper bound on the execution time (to be assured that it will perform acceptably) or a lower bound (possibly to encourage us to develop a better algorithm).

To sum up these observations: we measure the execution time of an algorithm as a function  $T(n)$  of a single quantity  $n$  (the size of the entire input), we are interested only in the asymptotic form of  $T(n)$ , and, if necessary, we shall be content with upper and lower bounds only on the asymptotic form. To express these ideas we use  $O, \Omega, \theta$  notation for upper bounds, lower bounds, exact bounds on  $T(n)$ . Thus  $T(n) = O(f(n))$  means that, apart from a multiplying constant,  $T(n)$  is bounded above by  $f(n)$  as  $n \rightarrow \infty$ . The  $\Omega$  notation is for lower bounds, and the  $\theta$  notation is for those happy situations where lower and upper bounds coincide.

The  $O$  notation was borrowed from mathematics rather than invented for the expression of upper bounds on algorithm execution times. It was adopted by algorithm designers, principally Knuth in [30], during the 1960's as the asymptotic approach to upper bounds gained acceptability. By the time Aho, Hopcroft and Ullman published their celebrated book [1] in 1974 the notation was well-established.

The theory of algorithms has a number of goals. The practical goals are those of any engineering discipline: to provide ready made tools (algorithms for solving frequently occurring problems) and successful methodologies (algorithm design strategies) for constructing further tools and end products. In addition the subject has a purely scientific side whose goals are to create the mathematics required to produce the engineering tools and to answer foundational questions about whether algorithms of various types exist; like any pure science this is also studied for its own interest. The two wings of the subject interact with each other in many ways and it is important that researchers in the subject appreciate both of them. The next two sections survey some of the key features of the subject. Greater detail can be found in modern texts such as [16, 9, 37]; Knuth's classic volumes [30, 31, 32] still repay study; some state of the art surveys are to be found in [34].

## 2 Algorithms Engineering

### 2.1 The Blast Furnace: Design Methodologies

To illustrate the standard design methodologies and their effectiveness we shall give a number of case studies:- example problems which are solved by apply-

ing “off-the-shelf” techniques. It will be seen that most of the techniques are variations on a single theme: structured decomposition of a problem into sub-problems. The most appropriate variation in any single case can often only be chosen by an experienced algorithms engineer.

The first problem is a numerical one of a sort that often occurs in scientific programming. For simplicity we have chosen a fairly particular version but it should be clear how to extend the solution to more general cases. Let  $x_0, x_1, a, b$  be given integers and suppose that  $x_2, x_3, \dots$  are defined by the recurrence equation

$$x_n = ax_{n-1} + bx_{n-2}$$

(In the special case  $x_0 = 1, x_1 = 1, a = b = 1$  the sequence  $x_0, x_1, x_2, \dots$  is the sequence of Fibonacci numbers). The problem is to compute, for some given non-negative  $n$  which might be quite large, the value  $x_n$ . Since  $x_n$  is defined recursively a natural algorithm would be

---

**Algorithm 1** Recurrence solver

---

```

function  $X(n)$ 
  if  $n = 0$  then
    return  $x_0$ 
  else
    if  $n = 1$  then
      return  $x_1$ 
    else
      return  $aX(n - 1) + bX(n - 2)$ 
    end if
  end if

```

---

This algorithm is grossly inefficient. A very standard piece of algorithmic analysis shows that its execution time  $T(n)$  grows in proportion to the  $n$ th Fibonacci number, i.e. as  $(\frac{1+\sqrt{5}}{2})^n$ .

There is an obvious improvement one can make (obvious in this case, but deriving nevertheless from a standard design technique called dynamic programming). One computes and stores all of  $x_0, x_1, x_2, \dots, x_{n-1}$  before computing  $x_n$ . This algorithm is

---

**Algorithm 2** Improved recurrence solver

---

```

function  $X(n)$ 
  array  $Y[0..n]$ 
   $Y[0] := x_0; Y[1] := x_1$ 
  for  $i = 2$  to  $n$  do
     $Y[i] := aY[i - 1] + bY[i - 2]$ 
  end for

```

---

Clearly, the execution time of this algorithm grows in proportion to  $n$  rather

than exponentially. Stark differences like this used to be cited as evidence that recursion was inherently less efficient than iteration. Of course, it is not the overheads of recursion which make the first algorithm less efficient than the second:- it is a different algorithm, causing a different computation, which just happens to be wildly inefficient.

Usually, an improvement from an exponential to a linear running time is cause to be content but, in this case, we can do substantially better still. There is a well-known mathematical technique for solving linear recurrence equations. In this particular case it tells us that the solution has the form  $K_1\lambda_1^n + K_2\lambda_2^n$  where  $K_1$  and  $K_2$  are constants and  $\lambda_1$  and  $\lambda_2$  are the roots of  $\lambda^2 - a\lambda - b = 0$  (unless  $\lambda_1 = \lambda_2$  when the solution has the form  $(K_1n + K_2)\lambda^n$ ). Unfortunately  $\lambda_1, \lambda_2$  need not be rational (indeed, they may not even be real), and it seems dubious to introduce irrational numbers into a problem whose statement and answer involve only integers so we seek some other idea. The original recurrence may be re-written as

$$(x_{n-1}, x_n) = (x_{n-2}, x_{n-1}) \begin{pmatrix} 0 & a \\ 1 & b \end{pmatrix}$$

Put  $v_n = (x_{n-1}, x_n)$ ,  $A = \begin{pmatrix} 0 & a \\ 1 & b \end{pmatrix}$  so that  $v_1 = (x_0, x_1)$  and, for  $n > 1$ ,  $v_n = v_{n-1}A$ . Then, of course,  $v_n = v_0A^{n-1}$  so, once  $A^{n-1}$  is known,  $v_n$  (and so  $x_n$ ) can be obtained in a small number of further operations (4 multiplications and 2 additions though algorithm designers don't like to be precise about such details!). The computation of  $n$ th powers is another piece of off-the-shelf technology:

---

**Algorithm 3** Powering algorithm

---

```

function power( $A, n$ )
if  $n = 0$  then
    return identitymatrix
else
     $Y := \text{power}(A, n \div 2)$ 
    if even( $n$ ) then
         $Y^2$ 
    else
         $Y^2A$ 
    end if
end if

```

---

The powering algorithm has been obtained by the standard design technique called Divide and Conquer. Its execution time  $T(n)$  satisfies

$$T(n) \leq T(n/2) + L$$

where  $L$  is constant and so  $T(n)$  grows at worst logarithmically in  $n$ .

A doubly exponential improvement from the algorithm initially suggested by the form of the problem is very impressive but the important point is that this gain in efficiency has been achieved by calling on entirely standard machinery in algorithm design. Before leaving this example, a more pragmatic remark should be made. The possibility of integer overflow in the above problem has been ignored for simplicity. In fact, overflow may easily occur because it is possible that  $x_n$  may be very large. To overcome the overflow problem some method for representing and handling large integers must be used. Such methods have been extensively investigated and many efficient techniques are known.

We saw in this example two basic design paradigms: divide and conquer, and dynamic programming. It is useful to compare the two. Both achieve their ends by defining and solving smaller subproblems which, in the most interesting cases, are smaller versions of the original problem. The divide and conquer strategy is goal-directed: each subproblem further decomposes into smaller subproblems and the entire computation is often visualised as a top-down process with recursion as the most natural implementation technique. Dynamic programming on the other hand is more commonly regarded as a bottom up process: a number of small subproblems are first solved, these are combined to solve larger subproblems which are in turn combined to solve even larger subproblems. This view of the technique more often suggests an iterative solution. However, the difference between the two techniques does not lie in the top-down/bottom-up dichotomy. What distinguishes dynamic programming from divide and conquer is the explicit tabulation of subproblem solutions. This implies that dynamic programming comes into its own for those problems where there are many duplicates in the collection of subproblems that is defined.

It might be thought from this account that dynamic programming is a more powerful design technique than divide and conquer since, with care, any algorithm discovered by divide and conquer may be re-implemented to look like the result of a dynamic programming design. But that misses the point. Divide and conquer is a conceptual tool which encourages a certain way of thinking, and this way of thinking has proved to be highly successful in a large number of designs. It is more profitable to regard the two techniques as partners in any problem which can be solved by decomposition into subproblems. The divide and conquer technique will usually be tried first. If it does lead to an algorithm to solve the problem the next step is to examine the efficiency of the algorithm. If the efficiency is unacceptable because many duplicate subproblems are being solved then the algorithm can profitably be re-engineered by dynamic programming. It is not always easy to identify precisely which subproblem solutions should be tabulated and it would be interesting to have automatic tools to aid in this analysis.

There are a number of other design paradigms which have proved useful. The first that we discuss, inductive design, could be described as a special case of both divide and conquer and dynamic programming although that does not do justice to its utility. The idea is very simple: we try to construct a solution to a

problem of size  $n$  from the solution to an  $(n - 1)$ -sized problem. Although this is just another way of solving a problem by focusing on smaller, similar sub-problems it has two advantages. The first is that we tap in to our mathematical experience of proving theorems by simple induction and this often suggests a suitable technique. Second, when the technique is successful, it tends to produce rather simpler designs than divide and conquer. The reason for this is that divide and conquer generally produces its most efficient results when the subproblems into which the original problem is partitioned are of roughly equal sizes – and this expectation is usually incorporated into any design attempt. Divide and conquer corresponds to complete rather than simple induction and so is naturally more complex. To illustrate inductive design I have chosen a problem which has been much studied as an example of the automatic derivation of algorithmic solutions from problem specifications [23, 8].

The problem is called the Maximum Segment Sum problem. One is given a sequence  $x_1, x_2, \dots, x_n$  of  $n$  numbers and is required to find two indices  $i, j$  to maximise

$$S_{ij} = x_{j+1} + \dots + x_i$$

Since there are many dependencies among the  $S_{ij}$  one would expect that brute force calculation of all of them will not be the best method. Instead we can define  $s_i = x_1 + \dots + x_i$  so that  $S_{ij} = s_i - s_j$ . The problem is therefore to compute  $m_n$  where

$$m_r = \max_{0 \leq j \leq r} (s_i - s_j)$$

The inductive design method tries to compute  $m_r$  from  $m_{r-1}$  so we need to express  $m_r$  in terms of  $m_{r-1}$ . Clearly,

$$\begin{aligned} m_r &= \max_{0 \leq i \leq r} \max_{0 \leq j \leq i} (s_i - s_j) \\ &= \max_{0 \leq i \leq r} (s_i - \min_{0 \leq j \leq i} s_j) \\ &= \max(m_{r-1}, s_r - \min_{0 \leq j \leq r} s_j) \end{aligned}$$

This equation gives hope that inductive design will be successful and it also indicates that  $t_r = \min_{0 \leq j \leq r} s_j$  will enter the calculation. The quantity  $t_r$  can also be computed by inductive design for, obviously,  $t_r = \min(t_{r-1}, s_r)$ . Finally,  $s_r$  itself has an inductive definition as  $s_{r-1} + x_r$ . Putting these observations together, with proper initialisations, the algorithm emerges as

In practice the algorithm would use single variables for each of  $s_r, t_r, m_r$ .

In addition to these decomposition techniques for algorithm design there are some heuristic techniques which are particularly useful for optimisation problems. We mention three: the greedy heuristic, simulated annealing, and genetic algorithms. The greedy heuristic is the most widely applicable of the three and is the simplest. It is characterised by the strategy of always choosing what appears

---

**Algorithm 4** Maximum segment sum algorithm

---

```
 $m_0 := 0$   
 $s_0 := 0$   
 $t_0 := 0$   
for  $r = 1$  to  $n$  do  
   $s_r := s_{r-1} + x_r$   
   $t_r := \min(t_{r-1}, s_r)$   
   $m_r := \max(m_{r-1}, s_r - t_r)$   
end for
```

---

to be the best action at every stage with no forward planning to see whether this might be inadvisable. The algorithms so produced are called greedy algorithms. This strategy sounds so naive that it scarcely deserves to be honoured by a title. Yet greedy algorithms can often produce optimal algorithms or algorithms whose departure from optimal can be quantified. Because many algorithms have been devised by this method there is a large body of experience for assessing new greedy algorithms. There is, moreover, a fairly general matroid framework due to Korte and Lovasz [33] which can sometimes demonstrate that a greedy algorithm is optimal.

Simulated annealing is a method based on randomisation techniques. The method originates from the annealing of solids as described by the theory of statistical physics and was introduced into combinatorial optimisation in [29]. The basic idea is to model an optimisation problem as a search through a space of configurations. In searching for an optimal configuration one passes from one configuration to another according to certain probabilities. In this respect the method appears similar to a random walk through configuration space as described by a Markov chain. In a “pure” Markov chain the next configuration might be worse (according to some optimisation measure) than the current one. However, in simulated annealing, there is a “temperature” parameter  $c$  which is gradually decreased and which controls the probabilities of the Markov chain. Initially, transitions to worse configurations might be quite common but, as  $c$  decreases to zero, they become rarer.

There have been many applications of simulated annealing (see [15] for a survey). The general experience is that, while near optimal solutions can be found in many cases, the technique tends to be rather slow. This is unsurprising since the method is so general. Indeed, all simulated annealing applications can be regarded as parametrised versions of a single algorithm.

Genetic algorithms [24, 21] have much in common with simulated annealing. They are also parametrised versions of a single algorithm and they proceed iteratively with each stage being determined probabilistically from the previous stage. However, rather than keeping just one candidate configuration at each stage, the genetic algorithm keeps a whole population of candidates which it updates from one generation to the next by rules called fitness selection, mutation,

and cross-over (biological terminology is ubiquitous in the subject). In the basic genetic algorithm the configurations are represented by fixed length character strings. The next generation consists of configurations from the previous generation which have passed a fitness test (their optimisation measure was high), configurations which are mutations (small variations in the configurations of the previous generation) and, most importantly, pairs of configurations born of two parent configurations by cross-over. In cross-over two parent configuration strings  $\alpha\beta, \gamma\delta$  give rise to offspring  $\alpha\delta, \beta\gamma$ . When the genetic algorithm achieves some termination criterion the best configuration in all generations is taken as the result.

Simulated annealing and genetic algorithms have many attractions. Both offer a packaged framework that “only” requires the problem at hand to be encoded in a suitable form, both can point to a stream of successful results (of course, researchers tend to report their successes not their failures), and both have a fashionable language of discourse. The greedy technique, simulated annealing, and genetic algorithms are increasingly sophisticated methods of heuristic search for a solution in a large pool of possibilities and that fact remains even when the colourful language has been stripped away. Nevertheless, it is likely that even better heuristics exist, although they may well prove harder to analyse.

## 2.2 The tools of the trade: Standard algorithms

An early realisation in Computing was that certain tasks occur over and over again in program construction. The early libraries of machine-coded subroutines were a first attempt to categorise those tasks. Nowadays those libraries have become vastly larger collections of machine-independent software maintained by international organisations such as NAG [38]. The result is that a modern-day software designer has a huge toolbox of standard components from which to build new algorithms. It is impossible to survey with any adequacy the full range of this toolbox; creating it has probably been the major activity of algorithm designers for forty years. To give some idea of where the main efforts have been concentrated I shall give my own view of how to categorise these algorithms, with some examples from each category. I would classify the standard algorithms under three broad headings: searching, numerical, and combinatorial.

### *Searching algorithms*

In this category might be placed all the algorithms for accessing the many data structures that have been devised for representing large collections of information (search trees, heaps, hash tables, external files etc.), graph searching techniques such as are used in Artificial Intelligence, and garbage collection algorithms. The category would include also string-processing techniques such as fast pattern matching and the algorithms required for solving the special problems in molecular biology thrown up by the human genome project.

The development of algorithms of this type, more so than in any other area, has gone hand in hand with devising subtle methods of structuring data. Indeed, sometimes it is only through the invention of a new data structure that improvements in performance are obtained. Currently however, data structuring is at a cross-roads. Some designers have been reluctant to sacrifice the final ounce of efficiency and have not adopted a disciplined approach to building complex structures. The result has been a mushrooming of very specialised structures and it is difficult to discern underlying methodologies. Data structuring now needs a period of consolidation to allow the best results to come to the fore and to allow the subject to be re-engineered with fewer basic components.

Data type abstraction has not hitherto figured much as an algorithm design tool because, unlike procedural abstraction, many existing languages have few or no mechanisms to support it. Yet data type abstraction has a well-developed theory which ought to be contributing more to the design of algorithms. A re-examination of data structuring tools would also permit us to make some judgements on which abstract data types were truly fundamental just as we judge some algorithms to be more fundamental than others.

#### *Numerical algorithms*

Algorithms for numerical problems (matrix calculations, ordinary and partial differential equations, curve fitting etc.) are the earliest in computer science. Handling the mismatch between the continuous problems of numerical analysis and the discrete nature of physical computers has always presented great challenges in error analysis. These challenges are being met in two ways. Systems for performing multi-precise arithmetic have long been used to reduce propagated rounding error and the asymptotic complexity of the best algorithms for the operations of arithmetic has steadily improved. Multi-precise arithmetic is the first line of defence in the control of error propagation but, ultimately, it only delays the problem as more ambitious calculations are mounted and a more fundamental solution must be found. The forte of numerical analysis is the construction of numerically stable algorithms and, in this, it has been very successful; most common numerical problems (see [41], for example) now admit solutions where the truncation error is small and the rounding error can be controlled.

But error analysis is only a sufficient tool when numerical solutions to numerical problems are sought. Increasingly, nowadays, there is a demand for symbolic solutions. Research activity in symbolic algebra has been very intense in the last two decades and the tendency has been to produce integrated systems rather than stand-alone modules. Mathematica and Maple both offer very large libraries of routines for classical algebraic operations and an integrated programming language. For “higher” algebra, particularly group theory, CAYLEY [11] and its successor MAGMA are the most powerful systems. From a systems design perspective AXIOM [25] appears most attractive since it presents a common view of the classical and algebraic data types. It is important to have

such a common view; it brings together the two communities of researchers in computer science and algebra who have been involved in algebraic computation and makes it more likely (as has not always been the case) that they will each be aware of the other's field.

### *Combinatorial algorithms*

In this context combinatorial algorithms are those which manipulate objects that require some computer representation not automatically provided by the basic hardware. Examples are permutations, graphs, geometric objects such as polygons, finite state automata, and languages. Research under this heading is very fragmented because the various combinatorial areas are very different. Nevertheless there are some common strands.

One such strand is the generation of combinatorial objects uniformly at random. There are two approaches to this problem. One approach is through encoding each object as an integer in a certain range, or as a sequence of such integers; this reduces the problem to generating one or more integers uniformly at random. For example, free labelled trees can be handled in this way using the Prüfer encoding, and binary trees can be handled by encoding them as a normalised bit sequence [3]. The other approach is more indirect but has led to results in other areas. It models the population from which the random object is to be chosen as a graph. Each node is an object in the population and two nodes are adjacent if they are "closely related in structure". Then, starting from some initial node, a random walk on the graph is defined. The theory of Markov chains can be used to compute the probability that a walk has reached a given node after  $t$  steps. When the probability distribution has settled down to a uniform distribution (which happens under quite mild conditions) the walk is terminated and the current node is selected. In practice the entire graph (which may be very large) is not stored, only the current node; the next node on the walk is constructed by randomly varying the previous one and it overwrites the previous node. Of course, the most important issue is how fast is the convergence to the stationary distribution and this has long been known to depend on the second largest eigenvalue of the matrix that specifies the probability transitions. During the 1980s a large number of results on "rapidly mixing" Markov chains were found which allowed the method to be applied in many situations. Examples, further details, and references are given in [26].

Another common strand is "combinatorial explosion". Combinatorial algorithms are frequently very time-consuming because they need to investigate a large space of possibilities and face the issue of NP-completeness (section 3.2).

The fastest growing area in combinatorial algorithms is undoubtedly computational geometry which has numerous applications (graphics, robotics, geographic information systems etc.). The subject addresses both classical geometric problems (such as congruence [5] and convex hulls [42]) and significant new problems (such as Voronoi diagram computation and visibility; the survey by Yao ([34],

Chapter 7) has a large bibliography of results on all these new problems). After a frenetic first 15 years of theoretical activity software systems (eg [45]) which allow geometric algorithms to be built from standard components are now being developed.

I conclude this section with an example of a problem which, when it was first posed, seemed so hard that it was proposed [10] as the basic ingredient of a cryptographic protocol (the security of such protocols depend on the supposed difficulty of carrying out particular computations). Yet, in the end, the problem proved very easy to solve merely by using standard algorithms. Unfortunately, there are now so many standard algorithms that it is virtually impossible to know of them all; a dictionary of algorithms would be very useful.

One is given a polynomial  $f(x)$  of degree  $n = rs$  (monic with no significant loss in generality) and is required to find monic polynomials  $g(x), h(x)$  (if they exist) of degrees  $r, s$  such that  $f(x) = g(h(x))$ . Let  $F(x) = x^n f(x^{-1})$  and  $H(x) = x^s h(x^{-1})$  be the reverse coefficient polynomials corresponding to  $f(x)$  and  $h(x)$ . Then it follows from  $f(x) = g(h(x))$  with a little elementary algebra that

$$F(x) = H(x)^r \pmod{x^s}$$

This type of congruence can easily be solved for  $H(x)$  by the lifting algorithm called Hensels lemma (see section 3.1). Once  $h(x)$  is known,  $g(x)$  can be determined by interpolation; one chooses  $r + 1$  values  $\alpha_0, \alpha_1, \dots, \alpha_r$  such that  $h(\alpha_0), h(\alpha_1), \dots, h(\alpha_r)$  are distinct and obtains  $g(x)$  by interpolation from the values  $g(h(\alpha_i)) = f(\alpha_i)$ . Not only do algorithms for all these steps exist they are, courtesy of the fast Fourier transform (see section 3.1), very efficient indeed (execution time  $O(n \log^2 n \log \log n)$ ). More details are given in [20].

### 3 Algorithms Science

The line between algorithms engineering and algorithms science is very blurred. The distinction I would like to make is between the development of algorithms following trusted design techniques incorporating trusted components (engineering) and the invention of techniques and novel algorithms whose existence, a priori, was not even clear (science). Of course, the latter has fewer achievements but it is crucial for the former. In this section I shall outline some discoveries that are more properly classified under Algorithms Science even though hindsight sometimes reveals that they might have been discovered by using a general technique.

### 3.1 Pinnacle algorithms

From time to time a new algorithm is discovered which changes the way a part of the subject develops. I have chosen (with great difficulty from a large pool of candidates) four such algorithms as examples of first class science. The examples show that the idea of isolated boffins working in backrooms is misleading. All these algorithms (and families of algorithms) have depended on previous researchers preparing the ground.

#### *The Newton-Raphson iteration*

The famous iterative formula

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}$$

for generating a series of numbers which, under certain widely applicable conditions, converges rapidly to a root of  $f(x) = 0$  goes back, in certain forms at least, to Sir Isaac Newton. It is a simple idea but a great one. Its importance came initially from its multitude of uses in numerical analysis where, of course, there is also a multi-variable numerical generalisation. Since then it has been recognised as the basic procedure in “Hensels Lemma”, an important technique first arising in  $p$ -adic number theory and now important in computational number theory. More recently it has been used in symbolic power series manipulation systems where one works to a polynomial modulus rather than a numeric one.

#### *The fast Fourier transform*

Fourier transforms have long been an important tool in data analysis. To calculate with them numerically the original continuous function has to be replaced by its values at  $n$  discretely spaced points and certain sums involving complex roots of 1 then have to be computed. For many years these computations were thought to be somewhat impractical since the number of operations depended on  $n^2$ . In 1967 the famous  $O(n \log n)$  algorithm was announced [14] and immediately revolutionised many applications (optics, acoustics, signal processing, etc.). Interestingly, the algorithm can be described in a classic divide and conquer framework; indeed it is now a popular example of this principle in many algorithms textbooks although it does not seem to have been discovered in this way. The fast algorithm has also had an impact within the theory of algorithms since it gives rise to an asymptotically fast method for polynomial multiplication and, thereby, many other algorithms in symbolic algebra have been improved.

#### *Karmarkars linear programming algorithm*

The simplex method has been the workhorse of linear programming from its very early days. Although the method usually performs well it has a rare exponential worst case. In 1979 Khachian [28] published his ellipsoid method which showed that linear programming was, in principle, solvable in polynomial time; unfortunately the method has, in practice, a disappointing performance.

When Karmarkar introduced his method based on projective transformations in 1984 [27] there was general acceptance that it was a significant theoretical breakthrough but some initial scepticism about its practicality. However, it quickly became apparent that the method was very competitive with the simplex algorithm sometimes outperforming it substantially. AT&T soon released a commercial software product based on the algorithm (trademark KORBX [12]) and there have been many papers exploring its performance and studying the theory.

#### *The elliptic curve factoring algorithm*

The apparently esoteric activity of factoring very large integers became of much wider interest when the RSA encryption scheme was proposed in 1978 [44]. Its security seems to rest on the difficulty of factoring integers as do many of the later encryption schemes. At that time the best algorithms for factoring integers were developments of an old method dating back to Fermat. In 1985 Lenstra [35] introduced a method which was remarkable for its novelty, simplicity, and efficiency. The method is based on the fact that the set of points on an elliptic curve over a field form a group under an appropriate composition rule. Despite this theoretical basis the method can be implemented in just a few lines of code. It is comparatively efficient especially in the case where the integer to be factored has a relatively small divisor while not being as good as the quadratic sieve method [39] for “hard” integers (products of two primes of similar value). Nevertheless, the method was a revolution for the factoring problem and showed that new algorithms are not necessarily complex to program.

### 3.2 Unifying theories

To lay claim to being scientific any subject has to have theories which place their components in uniform contexts. We have already seen the method of stipulating execution time which is a unifying theme throughout the theory of algorithms. It is also possible to carry out the actual analysis of execution times by appealing to a theory based largely on the theory of recurrence relations. A very simple example is the execution time of a loop:

for  $i = 1$  to  $n$  do  $P(i)$

If  $P(i)$  has execution time  $S(i)$  then, clearly, the execution time of the loop satisfies

$$T(n) = T(n - 1) + S(n)$$

In this particular case the recurrence is easily unwound as a summation but things are not always so easy. Consider two procedures  $P, Q$  with a single parameter  $n \geq 0$  defined as

---

**Algorithm 5** Two recursive procedures

---

```
function  $P(n)$ 
if  $n > 0$  then
     $Q(n - 1); C; P(n - 1); C; Q(n - 1)$ 
end if
function  $Q(n)$ 
if  $n > 0$  then
     $P(n - 1); C; Q(n - 1); C; P(n - 1); C; Q(n - 1)$ 
end if
```

---

where  $C$  denotes any statements taking time independent of  $n$ . The execution times  $S(n)$  and  $T(n)$  of  $P(n)$  and  $Q(n)$  satisfy, at least approximately,

$$\begin{aligned} S(n) &= S(n-1) + 2T(n-1) + K_1 \\ T(n) &= 2S(n-1) + 2T(n-1) + K_2 \end{aligned}$$

where  $K_1$  and  $K_2$  are constants. The theory of recurrences shows that  $S(n)$  and  $T(n)$  each grow in proportion to  $\lambda^n$  where  $\lambda = \frac{3+\sqrt{17}}{2}$  is the largest eigenvalue of the matrix

$$\begin{pmatrix} 1 & 2 \\ 2 & 2 \end{pmatrix}.$$

Recurrence equations also arise commonly in analysing divide and conquer algorithms. Here, recurrences like

$$T(n) = aT(nb) + f(n)$$

are typical and, when the form of  $f(n)$  is known, can usually be solved. Further examples appear in [22].

However, the most significant unifying ideas in the theory of algorithms come from structural complexity theory which addresses efficiency issues on a much larger scale. It attempts to make statements about execution time efficiency which are valid for all algorithms which solve a particular problem and which hold no matter what computational device is used. For example, the device may not have a random access memory; the memory might be in linear form and the price of accessing the  $n$ th location might be proportional to  $n$  (or worse). Given this degree of generality it is hardly surprising that lower and upper bounds on the complexity of optimal algorithms cannot be stated with much precision. A significant realisation, due initially to Edmonds [17], is that the class  $P$  of problems which admit solutions that can be computed in polynomial time (i.e. whose execution time is bounded by a polynomial function of the input size) is an “absolute” class:- it does not depend on any particular computational model.

Structural complexity theory is concerned with such absolute classes and the relationships between them.

After  $P$ , the next most important complexity class is called  $NP$ . The formal definition of  $NP$  is a little technical but essentially  $NP$  is the class of problems for which the validity of a solution can be checked in polynomial time. The most famous example is the Travelling Salesman problem:- does a given graph have a hamiltonian circuit. This problem does lie in  $NP$  since the validity of a putative hamiltonian circuit can be confirmed in a polynomial number of steps.

The central question in the complexity of algorithms and certainly one of the most important problems in the whole of science is whether  $P = NP$ . Although  $P = NP$  appears extremely unlikely no proof is known despite intensive efforts over two decades. One of the reasons that the question is so important is that  $NP$  contains, as was first proved by Cook [13], certain “complete” problems which are provably as hard as any in  $NP$ . If any one of these  $NP$ -complete problems could be solved in polynomial time then it would follow that  $P = NP$ . Hundreds of  $NP$ -complete problems have been found in an astonishing diversity of areas [19]. The general belief is that  $P \neq NP$  and so proving that a problem is  $NP$ -complete is usually taken as an indicator that it has no efficient solution. When faced with solving an  $NP$ -complete problem in a feasible amount of time one may therefore need to resort to heuristic techniques (which is why the heuristic techniques of section 2.1 are used so extensively) or be reconciled to finding a sub-optimal solution.

Progress on proving that  $P \neq NP$  has been slow and it is unlikely to be settled in the next few years. The most striking contribution was made by Razborov [43] for which, in 1990, he was awarded the Nevanlinna medal. One can represent a computable function by a Boolean circuit - an acyclic network of gates representing simple Boolean operators such as OR and NOT. Since AND, OR, NOT are a basis of Boolean functions it follows that functions which are computable in polynomial time can be represented by circuits with a polynomial number of gates of these types. Thus if we could show that a certain function necessarily required a superpolynomial number of gates we would know that it was not polynomial time computable. Razborov, using a method of approximation, managed to prove that some functions associated with NP-complete problems could not be computed in a polynomial number of gates of a monotone circuit (where only AND, OR are allowed). His method has subsequently been refined by other researchers to show that some  $NP$ -complete problems require an exponential number of gates of a monotone circuit [2].

## 4 The Crystal Ball

There are thousands of researchers developing new algorithms in numerous specialised areas. The algorithms they invent will depend much more on their

specialised areas than on the discovery of new central tenets in the theory of algorithms. It may therefore seem that the direction of general algorithmic theory will be driven by applications rather than from within the subject; that is, the engineering side will dominate the scientific side. Nevertheless, it is important that the scientific side flourishes for past experience is that it has nourished the engineering aspects in an essential way. Unquestionably, work on the  $P = NP$  question will continue but it is impossible to be so categorical about other strands of the subject. Despite this I would like to offer a personal view on questions the general theory should next address.

I have hinted already that abstract data typing should become a more serious tool in algorithm design. Of course it has long been promoted as an important program development tool. In addition there is also an elaborate theory of algebraic specification [18] which, at the very least, demonstrates that abstract data types (ADTs) are worthy objects of mathematical study. But I think there is more to abstract data types than this.

There is a strong analogy between ADTs and abstract algebra. ADTs are defined in terms of a permitted set of operations on the instances of the data type in question. Already this recalls abstract algebra. Indeed any category of algebraic objects can lay just claim to being an ADT. Thus, groups are sets which “support” a 0-ary, 1-ary, and 2-ary operation (with certain important additional properties). This analogy is clearly visible in modern symbolic algebra systems; for example, the AXIOM system reference book [25] treats both ADTs and algebraic objects on an equal footing as “domains”. There is also an interesting parallel having to do with representation. ADTs might be concretely represented in a number of ways (a priority queue as a heap or as a binomial queue, for example) just as the objects of abstract algebra might have one or more concrete representations (a group as a set of permutations or a set of matrices, for example). One might argue that the realisation that data types should be defined independently of their representation was as profound as the nineteenth century realisation that algebraic objects did not have to be studied in any fixed representation. Abstract definition in algebra was the first step in an impressive development throughout this century which led to fundamental applications in quantum theory, crystallography, and relativity [36, 46]. The long-term benefits of a study of properties of ADTs are, of course, impossible to foresee but the historical parallel with algebra is very encouraging.

Despite there being an infinite number of data types there is a small number only of them which recur frequently in software and algorithm design (stacks, queues, arrays, dictionaries etc) suggesting that some data types are more fundamental than others. Exactly the same phenomenon occurs in algebra and, significantly, the natural algebraic systems (groups, rings, modules etc) have very rich theories. It is at least plausible that the commonly occurring data types will also have very rich theories. Hitherto, ADTs have been primarily used as a software design tool but now that a cohort of fundamental ADTs has emerged the time is ripe for these to be studied for themselves. Even before

the idea of abstract data type emerged Knuth, Tarjan and Pratt were considering the functional behaviour of stacks and queues [[30, 47, 40] viewing them as restricted computers and investigating their power. More recently Beals and I [4] have begun a similar study of priority queues and have obtained some very promising results. This work is currently being extended at St Andrews to other data types and has already suggested a new approach to studying non-determinism in computer networks [6]. Our hope is that, ultimately, through a taxonomy of properties of fundamental data types the software designer will know which are most likely to be of use in specific applications just as a taxonomy of algorithm design techniques (divide and conquer, greedy heuristics, dynamic programming, etc) is a guide to inventing appropriate algorithms.

## References

- [1] A.V. Aho, J.E. Hopcroft, J.D. Ullman, *The Design and Analysis of Computer Algorithms*, Addison-Wesley 1974.
- [2] A.A. Andreev, A method of proving lower bounds of individual monotone functions, *Soviet Math. Dokl.* 31 (1985) 530–534.
- [3] M.D. Atkinson, J.-R. Sack, Generating binary trees at random, *Information Processing Letters* 41 (1992), 21–23.
- [4] M.D. Atkinson and R. Beals: Priority queues and permutations, *SIAM J. Comput.* (to appear).
- [5] M.D. Atkinson, An optimal algorithm for geometrical congruence, *J. Algorithms* 8 (1987), 159–172.
- [6] M.D. Atkinson, M.J. Livesey, D. Tulley, Networks of bounded buffers, in preparation.
- [7] J.L. Balcazr, J. Diz, J. Gabarr, *Structural Complexity I*, Springer-Verlag 1988.
- [8] J.L. Bates, R.L. Constable: Proofs as programs, *ACM Trans. Program. Lang. Syst.* 7 (1985), 113–136.
- [9] G. Brassard, P. Bratley, *Algorithmics: Theory and Practice*, Prentice-Hall, 1988.
- [10] J.J. Cade, A new public-key cipher which allows signatures, *Proc. 2nd SIAM Conf. on Applied Linear Algebra*, Raleigh NC, 1985.
- [11] J.J. Cannon, An introduction to the group theory language Cayley, in *Computational Group Theory*, (Ed. M.D. Atkinson), Academic Press 1984.

- [12] Y.C. Cheng, D.J. Houck, J.M. Liu, M.S. Meketon, L. Slutsman, R.J. Vanderbei, P.Wang, The AT & T KORBX system, *AT & T Tech. J.* 68 (1989), 7–19.
- [13] S. Cook, The complexity of theorem proving procedures, *Proceedings of third annual ACM symposium on Theory of Computing*, 1971, 151–158.
- [14] J. W. Cooley, J.W. Tukey, An algorithm for the machine calculation of complex Fourier series, *Math. Comp.* 19 (1965), 297–301.
- [15] N.E. Collins, R.W. Eglese, B.L. Golden, Simulated annealing - an annotated bibliography, *Amer. J. Math. Management Sci.* 8 (1988), 209–307.
- [16] T.H. Cormen, C.H. Leiserson, R.L. Rivest, *Algorithms*, McGraw-Hill 1992.
- [17] J. Edmonds, Paths, trees, and flowers, *Canad. J. Math.* 17 (1965), 449–467.
- [18] H. Ehrig, B. Mahr, I. Classen, F. Orejas: Introduction to algebraic specification. Part 1 and Part 2: Formal methods for software development and From classical view to foundations of systems specifications, *Computer Journal* 35 (October 1992), 451–459 and 460–467.
- [19] M.R. Garey, D.S. Johnson, *Computers and Intractability: A guide to the theory of NP-Completeness*, Freeman 1979.
- [20] J. von zur Gathen, Functional decomposition of polynomials: the tame case, *J. Symb. Comp.* 9 (1990), 281–299.
- [21] D.E. Goldberg, *Genetic Algorithms in Search, Optimization, and Machine Learning*, Addison-Wesley 1989.
- [22] D.H. Greene, D.E. Knuth, *Mathematics for the Analysis of Algorithms*, Birkhuser, 1982.
- [23] D.Gries, The maximum-segment-sum problem, in *Formal Development of Programs and Proofs* (ed. E.W. Dijkstra), Addison-Wesley (Reading, Mass.) 1990, 33–36.
- [24] J.H. Holland, *Adaptation in Natural and Artificial Systems*, University of Michigan Press 1975.
- [25] R.D. Jenks, R.S. Sutor: *AXIOM: The Scientific Computer System*, Springer-Verlag 1992
- [26] M.R. Jerrum, A. Sinclair, Approximate counting, uniform generation, and rapidly mixing Markov chains, *Infor. and Control* 82 (1989), 93–133.
- [27] N. Karmarkar, A new polynomial time algorithm for linear programming, *Combinatorica* 4 (1984), 373–395.
- [28] L.G. Khachian, A polynomial time algorithm in linear programming, *Soviet Mathematics Doklady* 20 (1979), 191–194

- [29] S. Kirkpatrick, S.D. Gelatt, M.P. Vecchi, Optimization by simulated annealing, *Science* 220 (1983), 671–680.
- [30] D.E. Knuth, *Fundamental Algorithms*, vol. 1 of *The Art of Computer Programming*, Addison-Wesley 1973 (2nd edition).
- [31] D.E. Knuth, *Seminumerical Algorithms*, vol. 2 of *The Art of Computer Programming*, Addison-Wesley 1981 (2nd edition).
- [32] D.E. Knuth, *Sorting and Searching*, vol. 3 of *The Art of Computer Programming*, Addison-Wesley 1973.
- [33] B. Korte, L. Lovasz, Mathematical structures underlying greedy algorithms, in F. Gecseg (Editor), *Fundamentals of Computation Theory, Lecture Notes in Computer Science* 117, 205–209, Springer-Verlag, 1981.
- [34] J. van Leeuwen (Ed.), *Handbook of Theoretical Computer Science*, MIT Press/Elsevier 1992.
- [35] H.W. Lenstra, Factoring integers with elliptic curves, *Ann. Math.* 126 (1987), 649–673.
- [36] E.M. Loeb: *Group Theory and its Applications*, Academic Press 1968.
- [37] U. Manber, *Introduction to Algorithms; A Creative Approach*, Addison-Wesley 1989.
- [38] The Numerical Algorithms Group, Ltd, Oxford, UK, and Numerical Algorithms Group, Inc., Downers Grove, Illinois, USA.
- [39] C. Pomerance, The quadratic sieve factoring algorithm, in *Advances in Cryptology* (Eds T. Beth, N. Cot, I. Ingemarrson), *Lecture Notes in Computer Science* 209 (1984) 169–182.
- [40] V.R. Pratt, Computing permutations with double-ended queues, parallel stacks and parallel queues, *Proc. ACM Symp. Theory of Computing* 5 (1973), 268–277.
- [41] W.H. Press, S.A. Teukolsky, W.T. Vetterling, B.P. Flannery, *Numerical Recipes in Fortran*, Cambridge University Press 1992 (2nd edition).
- [42] F.P. Preparata, M.I. Shamos, *Computational Geometry*, Springer-Verlag, 1985.
- [43] A.A. Razborov, Lower bounds on the monotone complexity of some Boolean functions, *Soviet Math. Dokl.* 31 (1985), 354–357.
- [44] R. L. Rivest, A. Shamir, L.M. Adleman, A method for obtaining digital signatures and public-key cryptosystems, *Communications of the ACM* 21 (1978), 120–126.

- [45] J.-R. Sack, A. Knight, P. Epstein, J. May, T. Nguyen, A workbench for computational geometry, *Algorithmica* (to appear).
- [46] R.L.E. Schwarzenberger: *N-dimensional Crystallography*, Pitman 1981.
- [47] R.E. Tarjan, Sorting using networks of queues and stacks, *JACM* 19 (1972), 341–346.