August 1988

Computers, algorithms and mathematics

László Lovász

0. Introduction

The development of computers is perhaps the single most significant technological breakthrough in this century. It is natural that it has not left untouched closely related branches of science like mathematics and its education. It is also natural that whichever fields have come into contact with it, heated debates have started and very different views, extreme and moderate, progressive and conservative, have been put forth. Is algorithmic mathematics of higher value than classical, structure-oriented, theorem–proof mathematics, or does it just hide the essence of things by making them more complicated then necessary? Does teaching of an algorithm lead to a better understanding of the underlying structure, or is it a more abstract, more elegant setting that does so? Is the algorithmic way of life best (Maurer 1985), or is applied mathematics just bad mathematics (Halmos 1981)? Should computers be introduced in elementary/secondary/college education of mathematics?

I want to start with a disclaimer: I will not attempt to give an answer to all these questions. The point I will try to make is that algorithmic mathematics (put into focus by computers, but existent and important way before their development!) is not an antithesis of the "theorem–proof" type classical mathematics. Rather, it enriches several classical branches of mathematics with new insight, new kinds of problems, and new approaches to solve these. So: not algorithmic *or* structural mathematics, but algorithmic *and* structural mathematics!

The interplay between the algorithmic and structural sides of mathematics is manyfold; I will only mention the two most important lines. The design and analysis of algorithms and the study of algorithmic solvability uses deeper and deeper tools from classical structural mathematics on the one hand; and an algorithmic perspective has a more and more profound effect on the whole framework of many fields of classical mathematics on the other hand.

Let me examplify. Perhaps the first notion of an "algorithm" that was defined clearly enough so that the question of "algorithmic solvability" could be raised was the notion of a geometric construction by ruler and compass, formulated by the Greek geometers. The mathematical interest of this notion is that there are both solvable and unsolvable construction problems. The design of construction algorithms has been stimulating in geometry for a long time, and has contributed to the development of important tools (very useful also independently of construction problems) like inversion or the golden ratio. But the proof of *unsolvability* of basic construction problems (trisecting an angle, squaring a circle, doubling a cube, constructing a regular heptagon etc.) illustrates this effect more dramatically. Such negative results were inaccessible for the Greek mathematics and for quite a while later on; it required the notion of real numbers and a substantial part of modern algebra to prove them, or even to formulate them with an exactness that made them accessible to mathematical methods. In fact, modern algebra was inspired by the desire to prove such negative results (besides the non-constructability of certain configurations, the non-solvability of equations of degree at least 5 was of the same nature).

As another example, let us consider the notion of primes. These numbers were also studied by the ancient Greeks, and they proved several basic properties of them. The beautiful but very hard theory of prime numbers has been a major branch of mathematics (and a source of inspiration for many other branches) throughout the history of modern mathematics too. But only the development of computers, and even more the establishment of computational complexity theory, raised the fundamental algorithmic problem: *how to test whether a given number is prime? how to find the prime factorization if it is not?* (Mathematicians in the 18-th and 19-th century, in particular Gauss, did make extensive computations regarding primes, and developed ingenious tricks to help their work. But appearantly they did not consider their algorithms as mathematical results.)

These questions have profound applications in computing, and the simple elementary procedures to solve them are by far not satisfactory (practice requires the consideration of numbers up to several hundred digits). Over the last 10 years or so, more and more advanced methods from number theory have been applied to design more and more sophisticated and efficient algorithms to answer these questions.

Note that the interaction of mathematics and algorithms in these two examples is different: in the first, we have a certain notion of an "algorithm" (a construction procedure), and we want to prove that it does not suffice to solve certain problems. Such questions may be notoriously hard, and, as our example shows, the proof may require deep mathematics or even the development of entirely new fields. The theory of computing today is full of unsolved problems of a similar nature; there are very few methods to prove negative results about the algorithmic solvability of problems, in particular if limits are imposed on the time (or other resources) that the algorithm can use. In such cases, computer science is an "external" user of mathematics: it supplies hard problems, which need be modelled and solved just like hard problems in mechanics or astronomy.

In the second example, computer science penetrates classical mathematics, putting old questions in a new perspective. Often, this is done by requiring *constructions* where "pure" existence proofs have been supplied by the classical theories. In other cases, one requires *efficient* procedures where "theoretically finite" case-analyses have been at hand (like in prime testing). In some branches of mathematics, e.g. in graph theory, this process has provided a whole new framework for the field (cf. Lovász 1986). In my talk, I would like to focus on this second development.

This new perspective on several issues in classical mathematics is, of course, a challenge to mathematics education. The introduction of computers (at whatever level) is only a very partial answer. I will give some remarks about how I think this challenge can be met; but I believe that it will take much more work — experimental and theoretical — before the contours of the answer will be clear.

1. Some old results from a new perspective

The approximation of irrational numbers by rational ones has a long history. Suppose that we have a real number α, that we want to approximate by a rational number with a small denominator. The first, trivial approach is to round: take any positive integer q, and round the product $q\alpha$ to the nearest integer p. Then p/q is a reasonably good rational approximation of α; in fact, $|\alpha - \frac{p}{q}| \le \frac{1}{2q}$.

Can we do better, with some positive integer Q in place of the 2 here? More precisely, can we find, for a given real number α and positive integer Q, a rational number p/q such that $|\alpha - \frac{p}{q}| \le \frac{1}{Qq}$? It is clear that in this case q cannot be chosen arbitrarily any more; but how large does it have to be?

A theorem of Dirichlet states that *for every given real number α and positive integer Q, we can find integers p and q such that* $0 < q \le Q$ *and* $|\alpha - \frac{p}{q}| \le \frac{1}{Qq}$. In other words, $|q\alpha - p| \le \frac{1}{Q}$.

There are two basic proofs of this fundamental result, and I would like to discuss them here to illustrate the difference between the algorithmic and non-algorithmic approaches.

First proof: Consider the $Q+1$ numbers

$$0\alpha - \lfloor 0\alpha \rfloor, \quad 1\alpha - \lfloor 1\alpha \rfloor, \quad , \ldots, \quad Q\alpha - \lfloor Q\alpha \rfloor$$

These numbers all lie in the interval $[0,1)$, so some two of them, say $k\alpha - \lfloor k\alpha \rfloor$ and $l\alpha - \lfloor l\alpha \rfloor$ $(0 \le k < l \le Q)$ are closer to each other than $1/Q$. Let $q = l - k$ and $p = \lfloor l\alpha \rfloor - \lfloor k\alpha \rfloor$. Then $q \le Q$ and

$$|q\alpha - p| = |(l-k)\alpha - (\lfloor l\alpha \rfloor - \lfloor k\alpha \rfloor)| = |(k\alpha - \lfloor k\alpha \rfloor) - (l\alpha - \lfloor l\alpha \rfloor)| < \frac{1}{Q}.$$

So the rational number p/q proves Dirichlet's Theorem.

Second proof (sketch): It is well known that every real number α can be expressed as a *continued fraction*

$$\alpha = a_0 + \cfrac{1}{a_1 + \cfrac{1}{a_2 + \cfrac{1}{a_3 + \ldots}}}.$$

This expansion may be finite (if α is rational) or infinite (if α is irrational). For example, we have the expansions

$$\frac{11}{8} = 1 + \frac{3}{8} = 1 + \cfrac{1}{2 + \cfrac{2}{3}} = 1 + \cfrac{1}{2 + \cfrac{1}{1 + \cfrac{1}{2}}}$$

and

$$\sqrt{2} = 1 + (\sqrt{2} - 1) = 1 + \frac{1}{\sqrt{2}+1} = 1 + \frac{1}{2 + (\sqrt{2} - 1)} = 1 + \cfrac{1}{2 + \cfrac{1}{2 + \cfrac{1}{2 + \ldots}}}.$$

Now if we stop in a continued fraction expansion in the k-th step, we get a rational number

$$\frac{p_k}{q_k} = a_0 + \cfrac{1}{a_1 + \cfrac{1}{a_2 + \ldots + \cfrac{1}{a_k}}}.$$

There are a number of basic facts that can be proved about this rational number, called the k-th *convergent* of α. What we need here is the fact that p_k/q_k is a very good approximation of α:

$$|\alpha - \frac{p_k}{q_k}| \le \frac{1}{q_k q_{k+1}}.$$

Hence, if we let k be the largest subscript for which $q_k \le Q$ (it is known that q_k tends to infinity, in fact exponentially fast), then

$$|\alpha - \frac{p_k}{q_k}| \le \frac{1}{q_k Q},$$

i.e. the rational number p_k/q_k proves Dirichlet's Theorem.

Which of these proofs is "better"? There is little doubt that the first one is much simpler: not only is it shorter but it is self-contained, while the second uses quite a bit from the theory of continued fractions.

But suppose that we also want to *find* the rational number in question. Before going into a discussion of this, we have to clarify one thing: how is α given? Since we want to have an algorithm now, we better assume that α is represented in some finite explicit form; let us assume that it is rational, say $\alpha = a/b$. Of course, we have to assume then that $Q < b$, else the approximation problem is trivial.

Which algorithm can we derive from the first proof? It tells us that we should form all the numbers $k\alpha - \lfloor k\alpha \rfloor$, then find two, which are close, then ... But if we want to form all these numbers anyway, we can check with the same amount of work which of them is less that $1/Q$. *So the first proof of the theorem does not provide us with any non-trivial algorithm to find the rational number whose existence it certifies;* it is a "pure existence proof" in this sense.

(It could be argued that the structural insight gained from the first proof can be developed further to obtain an efficient algorithm. This, however, takes further work and deeper insight.)

Is the second proof better from this point of view? How much work does it take to compute the continued fraction expansion? We show that not only is it easy to obtain this expansion, but also that there is nothing mystical about its use in an approximation problem.

Suppose that we want to find a good rational approximations of a number α. As a first approach, we could use the rational number $a_0/1$, where $a_0 = \lfloor \alpha \rfloor$. To obtain a better approximation, we have to approximate the error, i.e., $x_1 = \alpha - a_0$. Since this number is less than 1, it is a natural idea to replace it with its reciprocal, and approximate this by its integer part $a_1 = \lfloor 1/x_1 \rfloor$. Now this approximation again has an error $x_2 = 1/x_1 - a_1$. Take the reciprocal again etc. The positive integers $a_0, a_1, \dots$ obtained this way are just the coefficients in the continued fraction expansion of α.

If $\alpha = a/b$ then we obtain a_0 by dividing a by b with remainder; then a_0 is the quotient and if r is the remainder then the error $x_1 = r/b$. So $1/x_1$ is the rational number b/r and we repeat the procedure with this in place of a/b. It is clear that this is a finite procedure, and so the continued fraction expansion of α can be obtained. The rest of the proof gives a simple recipe to obtain the approximating rational number.

(Some of my readers may have observed at this point that to obtain the continued fraction expansion of a rational number a/b, we carry out exactly the same arithmetic operations as in the euclidean algorithm used to compute the greatest common divisor of a and b.)

So the second proof is just the analysis of a very natural iterative algorithm to find better and better approximations of a number.

But is this algorithm any better than the trivial one derived from the first proof? The answer is: much better! The number of steps in the continued fraction expansion of a/b (equivalently, the number of steps in the euclidean algorithm to compute g.c.d.(a,b)), is proportional to the number of digits of a and b; the number of steps in the first trivial algorithm is proportional to Q, which may be as large as b itself. If a, b and Q are 100 digit numbers, then the first algorithm takes 10^{100} steps while the second, less than 500.

The usual way to measure the running time of an algorithm is to compare the number of bit-operations with the number of bits necessary to write down the input. So a k-bit integer (having k bits in its base 2 expansion) contributes k to the input; in the diophantine approximation problem, the size of the input is the number of bits in a, b and Q, which is essentially $\log_2 a + \log_2 b + \log_2 Q$. In this model of computation, the length of the numbers also influences the time spent on a single arithmetic operation. For example, multiplication of two k-bit integers by the method tought at school takes about k^2 bit-operations, so such an operation contributes k^2 to the running time etc.

An important distinction to make is whether the running time grows as a polynomial of the input length or faster. Polynomial algorithms tend to be mathematically interesting and usually — though not always — practically feasible. Brute force case-distinctions often lead to exponentially many cases to distinguish (all subsets of a set, etc.) and thereby to exponentially growing running times. The algorithm derived from the first proof needs exponential time; the algorithm derived from the second is polynomial.

So we see that:

– The first proof is an existence proof; it is elegant, short, but does not give an algorithm to find the approximation. It takes more work and further ideas to develop it into an efficient algorithm.

– The second proof is just an analysis of an elegant, natural and efficient algorithm to construct good approximations. It takes work to analyse how good these approximations are.

The first proof also generalizes easily to the problem of simultaneously approximating several numbers by rationals with a common denominator. The algorithmic proof (continued fractions) also has some generalizations to this case, but those are substantially less elegant and do not yield as good approximations as the generalization of the first proof. To find a polynomial time algorithm for this simultaneous diophantine approximation problem, which would *find* the approximating rational numbers whose *existence* is guaranteed by the almost straightforward extension of the first proof is unsettled!

2. A glimpse of complexity theory

Most of us have met problems sounding like "Characterize those sets (numbers,...) with the property that $***$.", and also the student who gives, provocatively, the answer "A set has property $***$ if and only it has property $***$." What is wrong with this answer? And what happens if the student hides the triviality by slightly re-phrasing the property $***$? When does the answer begin to be non-trivial and thereby acceptable? Or should we simply ban this kind of problems as meaningless?

One of the great successes of the theory of computing is that *it is able to define in a mathematically exact way which characterizations are "good" and which are more-or-less just rephasing the question,* at least for a large class of structures and properties. There is no room in this talk, of course, to develop this theory. But I try to illustrate the idea on an important example.

Consider a system of inequalities

$$\begin{aligned}
a_{11}x_1 + a_{12}x_2 + \cdots + a_{1n}x_n &\le b_1,\\
a_{21}x_1 + a_{22}x_2 + \cdots + a_{2n}x_n &\le b_2,\\
&\vdots\\
a_{m1}x_1 + a_{m2}x_2 + \cdots + a_{mn}x_n &\le b_m.
\end{aligned} \tag{1}$$

Since we are interested in algorithmic aspects, we assume that the inequalities have rational coeffiecients.

When does (1) have a solution? Let us investigate the following two answers to this question:

Theorem A. *The system of inequalities (1) has a solution if and only if the system*

$$\begin{aligned}
a_{11}(u_1 - v_1) + a_{12}(u_2 - v_2) + \cdots + a_{1n}(u_n - v_n) &\le b_1,\\
a_{21}(u_1 - v_1) + a_{22}(u_2 - v_2) + \cdots + a_{2n}(u_n - v_n) &\le b_2,\\
&\vdots\\
a_{m1}(u_1 - v_1) + a_{m2}(u_2 - v_2) + \cdots + a_{mn}(u_n - v_n) &\le b_m
\end{aligned} \tag{2}$$

has a non-negative solution.

Theorem B. *The system of inequalities (1) has a solution if and only if the system*

$$\begin{aligned}
a_{11}y_1 + a_{21}y_2 + \dots + a_{m1}y_m &= 0,\\
a_{12}y_1 + a_{22}y_2 + \dots + a_{m2}y_m &= 0,\\
&\vdots\\
a_{1n}y_1 + a_{2n}y_2 + \dots + a_{mn}y_m &= 0,\\
b_1y_1 + b_2y_2 + \dots + b_my_m &= -1
\end{aligned} \tag{3}$$

has no non-negative solution.

(Note that a solution of (3) can be viewed as a linear combination of the inequalities in (1) with non-negative multipliers — the y_i — that yields the trivially non-solvable inequality $0x_1 + \ldots + 0x_n \le -1$.)

Both theorems give necessary and sufficient conditions for the solvability of (1). But Theorem A is an essentially trivial trick to show that solvability is easily transformed into non-negatyive solvability, while Theorem B is an important result (called Farkas' Lemma). What makes the difference?

Assume that I want to use a concrete case of (1) in this talk as an illustration of a system of solvable linear inequalities. How can I convince you that it is indeed solvable? Easy: I just show you a solution. Suppose now that I want to use another concrete case of (1) to illustrate an unsolvable system. How can I convince you that it is indeed unsolvable? To try all possible values for the variables? There is no easy way at hand.

Do the necessary and sufficient conditions formulated in the two theorems above help? If we apply Theorem A, I have to exhibit that (2) *does not* have a non-negative solution — this is not any easier than the original task. But if we apply Theorem B, it suffices to exhibit that (3) *does* have a non-negative solution — and this I can do by showing you one. So the condition in Theorem B does have an entirely different logical structure from the original property and from the condition given in Theorem A.

It turns out that a large part of graph theory, optimization, number theory etc. has an analogous structure. Important properties of graphs, numbers etc. have the feature that if they are present, there is an easy way to exhibit this (e.g. composite numbers, 4-colorable graphs etc.) Basically, these properties are defined in terms of the *existence* of a certain object (a number is composite if it has a proper divisor; a graph is 4-colorable if it has a proper coloring with 4 colors etc.) If I want to convince you that the number

$$617532176011725742711064069114397791170668109866281$$

is composite, all I have to do is to show you the divisor

$$7858321551080267055879091$$

(Of course you still have to verify that this is a divisor; but, as we know, this takes only polynomial time.)

Such properties are called NP-*properties* (named after a technical definition involving **N**on-deterministic **P**olynomial-time Turing-machines, whose detailes I don't have to give here). Sometimes (but not always!) the negation of NP-properties is again an NP-property. Theorems establishing such equivalences are often among the most important results in the field (like the Farkas Lemma above). They are sometimes called *good characterizations.*

This classification of properties is also closely related to algorithms. Most properties for which we would like to find a polynomial-time algorithm to decide them belong to NP in a natural way. If a property can be decided by a polynomial-time algorithm, then both the property and its negation are NP-properties, i.e., it has a good characterization. The converse may not be true: it is not known whether every well-characterized property can be decided in polynomial time (probably not, but as I remarked before, we do not have the means to prove such negative results in this area). But, usually, to find a good characteriztion is an important step towards the construction of a polynomial time algorithm. The example of the Farkas Lemma is very illustrative: almost a century after the proof of the lemma, a polynomial time algorithm to decide if (1) has a solution was given by Khachiyan in 1978 (the famous Ellipsoid Method).

Of course, this article cannot discuss the theory of algorithms in any reasonable detail; we only sketched what was necessary to support our arguments on a possible new framework for various fields in mathematics. (For an introduction to the theory of algorithms, see e.g. Sedgewick 1983.)

3. What does this imply in math education?

Whatever it implies, should be regarded with utmost caution and moderation. I feel that math education should follow what happens in math research, at least to a certain extent, in particular those (rare) developments there that fundamentally change the whole framework of the subject. Algorithmic mathematics is one of these. However, the range of the penetration of an algorithmic perspective in classical mathematics is not yet clear at all, and varies very much from subject to subject (as well as from lecturer to lecturer). Graph theory and optimization, for example, have been thoroughly re-worked from a computational complexity point of view; number theory and parts of algebra are studied from such an aspect, but many basic questions are unresolved; in analysis and differential equations, such an approach may or may not be a great success; set theory does not appear to have much to do with algorithms at all.

Our experience with "New Math", the adaptation of the set-theoretic foundations of mathematics in lower level mathematics education, warns us that drastic changes may be disastrous even if the new framework is well established in research and college mathematics. So let me just raise some ideas on the teaching of algorithms on various levels, emphasizing that they must be carefully discussed and tried out before any large scale implementation is attempted.

Basically, some algorithms and their analysis could be taught about the same time when theorems and their proofs first occur, perhaps around the age of 14. Of course, certain algorithms (for multiplication and division etc.) occur quite early in the curriculum. But these are more recipes than algorithms; no correctness proofs are given (naturally), and the efficiency is not analysed. The children have to learn (and practice) how to carry out these simple algorithms. This is like teaching theorems (axioms) without proofs, or teaching empirical facts in the sciences without experiments: necessary but not leading to really deep understanding.

What I would consider as the beginning of learning "algorithmics" is to learn to **design**, *rather than execute, algorithms.* (For an elaboration of this idea, see e.g. Maurer 1984). The euclidean algorithm, for example, is one that can be "discovered" by students in class. In time, a collection of "algorithm design problems" will arise (just as there are large collections of problems and exercises in algebraic identities, geometric constructions or elementary proofs in geometry). Along with these concrete algorithms, the students should get familiar with basic notions of the theory of algorithms: input-output, correctness and its proof, analysis of running time and space, good characterizations read off from algorithms, algorithms motivated by good characterizations etc.

Some possible types of algorithm-design problems, suitable probably already on the high school level: enumeration problems where no closed formula exists; elementary optimization problems in graph theory (e.g. maximum independent sets in trees, shortest paths, listing of cliques, circuits etc.); sorting and searching; simple (though inefficient) methods for primality testing, factorization, and many other problems in number theory; Gaussian elimination and other manipulations in linear algebra; convex hull and other elemantary plane geometry constructions.

In college, the shift to a more algorithmic presentation of the material should, and will, be easier and faster. Already now, some subjects like graph theory are taught in many colleges quite algorithmically: shortest spanning tree, maximum flow and maximum matching algorithms are standard topics in most graph theory courses. This is quite natural since, as I have remarked, computational complexity theory provides a unifying framework for many of the basic graph-theoretic results. In other fields this is not quite so at the moment; but some topics like primality testing or cryptographic protocols provide nice applications for a large part of classical number theory.

4. Computers and algorithms

At this point, I have to comment on the use of computers in the teaching of these topics. One should distinguish between an algorithm and its implementation as a computer program. The algorithm itself is a mathematical object; the program depends on the machine and/or on the programming language. It is of course necessary that the students see how an algorithm leads to a program that runs on a computer; but it is not necessary that every algorithm they learn about or they design be implemented. The situation is (again) analoguous to that of geometric constructions with ruler and compass: some constructions have to be carried out on paper, but for some more, it may be enough to give the mathematical solution (since the point is not to learn to draw but to provide a field of applications for a variety of geometric notions and results).

Let me insert a warning about the shortcomings of algorithmic language. There is no generally accepted form of presenting an algorithm, even in the research literature (and as far as I see, computer science text books for secondary schools are even less standardized and often even more extravagant in handling this problem.) The practice ranges from an entirely informal description to programs in specific programming languages. There are good arguments in favor of both solutions; I am leaning towards informality, since I feel that implementation details often cover up the mathematical essence. Let me illustrate this by two examples.

An algorithm may contain a step "Select any element of set S". In an implementation, we have to specify which element to choose, so this step necessarily becomes something like "Select the first element of set S". But there may be another algorithm, where it is important the we select the first element; turning both into programs hides this important detail. (Also, it may turn out that there is some adventage in selecting the *last* element of S. Giving an informal description leaves this option open, while turning the algorithm into a program forbids it.)

To give my second example, recall that the Fibonacci numbers are defined by the recurrence

$$F_{k+1} = F_k + F_{k-1}$$

(and by $F_0 = 0$ and $F_1 = 1$). This recurrence provides a trivial algorithm to compute these numbers. Turning this recurrence into a program, we would get something like this: if F is the current Fibonacci number, and G is the previous, then (with an auxiliary variable H)

$$H := F, \quad F := G + F, \quad G := H.$$

We see that the program contains a number of details that do not belong *mathematically* to the procedure of computing Fibonacci numbers: it stores F_{k+1} in the same register where F_k used to sit, but to do so, it has to "salvage" F_k since its value will be needed in the next step, and to do so, we need an "auxiliary variable" etc.*

To show the other side of the coin, the main problem with the informal presentation of algorithms is that the "running time" or "number of steps" is difficult to define — as we have experienced above. Unfortunately, this depends on the details of implementation (down to a level below the programming language; it depends on the data representation and data structures used). Sometimes there is a way out by specifying which steps are counted (e.g. comparisons in a sorting algorithm, or arithmetic operations in an algebraic algorithm); but this is "cheating" in a sense since we disregard the time needed to handle the data, which may be as much as, or even more than, the time used by the "mathematical essential" steps. It should be mentioned that the polynomiality of an algorithm is "robust", i.e., it does not depend on implementation (although different implementations may have different polynomials in the bound on their running time).

The route from the mathematical idea of an algorithm to a computer program is long. It takes the careful design of the algorithm; analysis and improvements of running time and space requirements; selection of (sometimes mathematically very involved) data structures; and programming. In college, to follow this route is very instructive for the students. But even in secondary school mathematics, at least the mathematics and implementation of an algorithm should be distinguished.

An important task for mathematics educators of the near future (both in college and high school) is to develop a smooth and unified style of describing and analysing algorithms.

---

* I am grateful to Jack Edmonds for these examples and arguments — and regret not to have given a more detailed exposition of them.

A style that shows the mathemetical ideas behind the design; that facilitates analysis; that is concise and elegant would also be of great help in overcoming the contempt against algorithms that is felt nowadays both from the side of the teacher and of the student.

References:

P. R. Halmos (1981), Applied mathematics is bad mathematics, in *Mathematics Tomorrow* (ed. L. A. Steen), Springer, 9-20.

L. Lovász (1986), *An algorithmic Theory of Numbers, Graphs, and Convexity,* CBMS-NSF Reg. Conf. Series in Appl. Math. **50**; SIAM.

S. Maurer (1984), Two meanings of algorithmic mathematics, *Mathematics Teacher* 430-435.

S. Maurer (1985), The algorithmic way of life is best; reflexions by R. G. Douglas, B. Korte, P. Hilton, P. Renz, C. Smorynski, J. M. Hammersley and P. R. Halmos; *College Math. Journal* **16**, 2-18.

R. Sedgewick (1983): *Algorithms*, Addison-Wesley.