COMPLEXITY OF COMPUTATIONS

S. Winograd

Rapporteurs: Dr. A.J. Mascall
Dr. C.R. Snow
Mr. D. Wyeth

Selected from the field of "Computational Complexity", these lectures will deal with two aspects which may be termed "Arithmetic Complexity" and "Analytic Complexity". Some other aspects of the subject will be covered by Professor Rabin in a separate series of lectures during this seminar.

Arithmetic Complexity

One can justify studies of this subject with the assertion that numerical (or arithmetical) computation is very much at the centre of Computing Science activities. It therefore follows that any knowledge that can be obtained about the algorithms used must be a useful addition to Computing Science as a whole. However, the author's own interest goes rather beyond that general ideal: there are two principal reasons for studying arithmetic complexity as a special case of computational complexity. The first is the question of whether a particular algorithm can be shown to be the most efficient way possible for solving a given type of problem. Such a result will certainly be useful in that it obviates the need to search for better algorithms. Secondly, and perhaps the most satisfying aspect, is that study will reveal a completely new algorithm. One can see that we have the beginnings of a systematic way of looking at algorithms, and being able to find new ones.

Objectives of the studies

Essentially we are trying to discover how many arithmetic operations are needed to solve a given problem. In this respect such studies differ from what is termed "Analysis of Algorithms" in that we are not trying to find out how many arithmatic operations are used by a given algorithm, but rather trying to find the best way to solve a given problem, in terms of the arithmetic operations involved. In general, we are interested in discovering how many additions/subtractions, multiplications/divisions or other operations are required to compute a set of algebraic expressions.

Taking first the simple example below:

$$a_0 b_0 = a_0 b_0$$

$$a_0 b_1 + a_1 b_0 = (a_0 - a_1)(b_1 - b_0) + a_0 b_0 + a_1 b_1$$

$$a_1 b_1 = a_1 b_1$$

This example has already been discussed in the lecture by Professor J. Hopcroft, and refers possibly to the problem of doing double-precision multiplication. Evaluation by the left-hand side requires four multiplications and one addition, by the right-hand side, three multiplications and four additions. We can immediately ask ourselves the following questions:

1. Is there another way using just two multiplications?
2. Is three the minimum number of additions?
3. Is one the minimum number of additions?
4. Is there an unavoidable trade-off between the number of multiplications and additions?

These are typical of the kinds of questions we can raise when confronted by such a problem. It is hoped that the present discussion will show that there are some answers to these questions, and also that there are many more unresolved problems.

Mathematical formulation

The general problem may be stated in a rigorous fashion as follows: we have a given number of variables (indeterminates) $Z_1, Z_2, \ldots, Z_k$ and we wish to compute certain rational functions in these variables

$$\Psi_i \in Q(Z_1, Z_2, \ldots, Z_k) \quad i = 1, 2 \ldots, t$$

when we are given some part of these functions

$$B \subseteq Q(Z_1, Z_2, \ldots, Z_k).$$

In the example, what we might be given are the rational numbers themselves and the variables $Z_1, Z_2, \ldots, Z_k$. However in general we can say we are given the set B, and from this we are to build whatever it is that we are interested in computing. We then can ask the various questions listed above. The general problem as stated here is too difficult to solve, but we can first consider the linear case as a simpler sub-problem. By this we mean that the variables Z_i are partitioned into two sets, denoted $(x_1, x_2, \ldots, x_n)$ and $(y_1, y_2, \ldots, y_m)$ and we assume that the expressions to be computed are linear in the x_j. This may be stated in the form:

$$\begin{pmatrix} \Psi_1 \\ \Psi_2 \\ \vdots \\ \Psi_t \end{pmatrix} = \begin{pmatrix} \Phi_{1,1}(y_1, y_2, \ldots, y_m) \ldots \Phi_{1,n}(y_1, y_2, \ldots y_m) \\ \cdot \\ \cdot \\ \cdot \\ \Phi_{t,1}(y_1, y_2, \ldots, y_m) \ldots \Phi_{t,n}(y_1, y_2, \ldots, y_m) \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{pmatrix} + \begin{pmatrix} \varphi_1 (y_1, y_2, \ldots y_m) \\ \vdots \\ \varphi_t (y_1, y_2, \ldots y_m) \end{pmatrix}$$

That is

$$\Psi = \Phi_{t \times n}\, x + \phi$$

where the Φ_{ij} are rational functions in y_k and the elements φ_1 (the constant terms) are also functions in y_k.

We may now give the following definition of the "rank" of the matrix Φ.

Definition 1 The column (row) rank of Φ, denoted by $\rho_c(\Phi)$ $(\rho_r(\Phi))$ is the largest number of columns (rows) such that no linear combination of them yields a column (row) with all entries rational numbers.

This concept of "rank" is not that normally defined for a matrix that is, the largest number of linearly independent columns (rows): it is in fact the number of columns (rows) which will not yield a column (row), all of whose entries are constants. The weighting functions themselves must be rational numbers. Note that this definition implies that the row and column "ranks" are not necessarily equal.

We can now derive several results from this definition of rank. The first result, stated by the theorem below, allows us to obtain a lower bound for the number of multiplications/divisions required to evaluate the expression.

Theorem 1 Every algorithm for computing $\Phi x + \varphi$ from $Q(y_1, \ldots, y_m) \cup Q(x_1, \ldots, x_n)$ requires at least $\max(\rho_r(\Phi), \rho_c(\Phi))$ multiplications/divisions, even if multiplication by a rational number is not counted.

This means that if we compute $\Phi x + \varphi$, starting with any possible rational functions of the $(y_1, \ldots, y_m)$ and the variables $(x_1, \ldots, x_m)$ as data for constructing the algorithm, then we shall need to perform a number of multiplications equal at least to the largest of the two ranks in order to perform the computation, and this is even if multiplication by a fixed rational number is not counted.

Similarly, the following theorem allows us to placea bound on the number of additions/subtractions, simply by subtracting the number of rows in the matrix from the number of non-zero columns.

Theorem 2 Every algorithm for computing $\Phi_{t\times m}\, x$ from $Q(y_1, \ldots, y_m) \cup Q(x_1, \ldots, x_m)$ requires at least $\bar{\rho}(\Phi) - t$ additions/subtractions, where $\bar{\rho}(\Phi)$ is the number of non-zero columns.

What do these two rather innocuous theorems tell us? Firstly, we can see that in the example given earlier we had already obtained the optimal result. We have:

$$\begin{pmatrix} a_0 b_0 \\ a_0 b_1 + a_1 b_0 \\ a_1 b_1 \end{pmatrix} = \begin{pmatrix} a_0 & 0 \\ a_1 & a_0 \\ 0 & a_1 \end{pmatrix} \begin{pmatrix} b_0 \\ b_1 \end{pmatrix}$$

so that the minimum number of multiplications is indeed three. Here we have replaced $(y_1, \ldots, y_m)$ by (a_0, a_1) and the $(x_1, \ldots, x_m)$ by (b_0, b_1). We also see that the minimum number of additions is one. However the necessary trade-off is still unknown; the algorithm simply shows us how we can achieve the theoretical result. Another result easily shown is that for forming an inner product:

$$\sum_{i=1}^{n} a_i b_i = (a_1, a_2, \ldots, a_n) \begin{pmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{pmatrix}$$

showing that the minimum number of multiplications is n, and of additions is n-1. We know already how to do this of course. Another example is a 3x4 matrix multiplied by a vector. Expressing this in the form:

$$\begin{pmatrix} a_1 & a_2 & a_3 & a_4 \\ a_5 & a_6 & a_7 & a_8 \\ a_9 & a_{10} & a_{11} & a_{12} \end{pmatrix} \begin{pmatrix} b_1 \\ b_2 \\ b_3 \\ b_4 \end{pmatrix} = \begin{pmatrix} b_1 & b_2 & b_3 & b_4 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & b_1 & b_2 & b_3 & b_4 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & b_1 & b_2 & b_3 & b_4 \end{pmatrix} \begin{pmatrix} a_1 \\ a_2 \\ \vdots \\ a_{12} \end{pmatrix}$$

we obtain 12 for the minimum number of multiplications, and 9 for the number of additions. The rearrangement allows us to obtain a matrix with all the columns linearly independent by replacing the $(y_1, \ldots, y_m)$ with the b's and the $(x_1, \ldots, x_m)$ with the a's. The normal algorithms would indeed achieve this, and we can see that in general the product of $A_{p \times q}\ \underline{b}$ requires pq multiplication and $p(q-1)$ additions.

Taking now the case of a symmetric matrix:

$$\begin{pmatrix} a_1 & a_2 & a_3 \\ a_2 & a_4 & a_5 \\ a_3 & a_5 & a_6 \end{pmatrix} \begin{pmatrix} b_1 \\ b_2 \\ b_3 \end{pmatrix} = \begin{pmatrix} b_1 & b_2 & b_3 & 0 & 0 & 0 \\ 0 & b_1 & 0 & b_2 & b_3 & 0 \\ 0 & 0 & b_1 & 0 & b_2 & b_3 \end{pmatrix} \begin{pmatrix} a_1 \\ \vdots \\ a_6 \end{pmatrix}$$

we see that the minimum number of multiplications is six. The following identity does perform the evaluation using 6 multiplications and so is optimal:

$$a_1 b_1 + a_2 b_2 + a_3 b_3 = (a_1 - a_2 - a_3)\, b_1 + a_2 (b_1 + b_2) + a_3 (b_1 + b_3)$$

$$a_2 b_1 + a_4 b_2 + a_1 b_3 = a_2 (b_1 + b_2) + (a_4 - a_2 - a_5)\, b_2 + a_5 (b_2 + b_3)$$

$$a_2 b_1 + a_5 b_2 + a_6 b_3 = a_3 (b_1 + b_2) + a_5 (b_2 + b_3) + (a_6 - a_3\ a_5)\, b_3$$

In general, $A_{n \times n}\ \underline{b}$, where A is symmetric, requires $n(n+1)/2$ multiplications.

Another example, computing the product of two complex numbers, requires a minimum of three multiplications. This can be done as follows:

$$ac - bd = ac - bd$$
$$ad + bc = (a + b)(c + d) - ac - bd$$

It is hoped by now that it should be clear that the linear case we have been studying, although a special case, still gives a good many useful results for a wide range of problems.

Another example we can examine, still using the linear case, is the evaluation of a polynomial. Here we may write:

$$P\,(x) = \sum_{i=0}^{n} a_i x^i = (1,\ x,\ x^2,\ \ldots,\ x^n) \begin{pmatrix} a_0 \\ \vdots \\ a_n \end{pmatrix}$$

Once we have the form of the right-hand-side, we see that there are n linearly independent columns, namely, the second to the (n+1)st columns. Therefore the evaluation requires at least n multiplications and n+1-1 additions for an nth order polynomial, which we know can be done using Horner's Rule. For the case of a system of polynomials we see that there is no reason to search for clever solutions; the total number of multiplications is simply the sum of the degrees, as it is also for the additions. Furthermore, for a polynomial in k variables we also have the result that $(n+1)^k-1$ multiplication and $(n+1)^k-1$ additions are needed, which can also be done by straightforward extensions of Horner's Rule. There are various other linear problems that can be examined, but it is hoped that the foregoing examples have been sufficient to convey the ideas. Incidentally, there are very few results concerning non-linear functions: one particularly interesting result, due to Straussen, is that the number of multiplications needed to compute all the symmetric functions on n variables is n log n.

Preconditioning

Remaining still with linear functions, there are other questions that we can also ask about these. Considering the evaluation of polynomials, we take a particular case of evaluating a fourth degree polynomial with many different values of the variable x:-

$$a_4 x^4 + a_3 x^3 + a_2 x^2 + a_1 x + a_0$$

$$= a_4 \left([x(x+x) + \beta]\,[x(x+\alpha) + x+\gamma] + \delta \right)$$

where $\alpha = \dfrac{a_3 - a_4}{2a_4}$

$$\beta = \frac{a_1}{a_4} - \alpha\left(\frac{a_2}{a_4} - \alpha\,(\alpha + 1)\right)$$

$$\gamma = \frac{a_2}{a_4} - \alpha(\alpha + 1) - \beta$$

$$\delta = \frac{a_0}{a_4} - \beta\gamma$$

This algorithm appears in the paper by Todd and is attributed to Motzkin. If the quantities α, β, γ, δ, depending only on the coefficients, are first evaluated and then used in the identity, only three multiplications and five additions are needed for each evaluation. Of course we must invest 7 multiplications and 7 additions for the initial computation of the α, β, γ, δ, but this is obviously worthwhile if the evaluation is to be done many times. When we are confronted with such an algorithm, the question that immediately springs to mind is whether there exists another identity which uses even fewer multiplications, or is this one minimal?

Examining this problem with the formalism we have already developed, we see that the act of computing the α, β, γ, δ effectively means that any function of the $(x_1, \ldots, x_n)$ in our general expression (where the $x_1 \ldots x_n$ have been used to represent the coefficients of the polynomial) is given to us. So now we can ask the same questions as before, only now starting with rational functions of both the $(y_1, \ldots, y_m)$ and $(x_1, \ldots x_n)$. In this case the following theorem holds:

<u>Theorem 3</u> Every algorithm for computing $\phi_{t \times n}$ from $Q(y_1, \ldots y_m) \cup Q(x_1, \ldots, x_n)$ requires at least $\rho_c(\phi)/2$ multipli-

cations/divisions and $\bar{\rho}\ (\Phi) - t$ additions/subtractions.

Thus the required number of multiplications is now the rank divided by two, the number of additions being the same as before. Initially at least, these results are rather surprising. They mean that

(a) It is possible to reduce the number of multiplications required if we are willing to precompute some of the coefficients.

(b) The number of multiplications cannot be reduced by more than one half.

(c) There is nothing that can be done to reduce the number of additions.

The result of Todd and Motzkin can now be stated as a corollary to this theorem: At least $\frac{1}{2}(n+1)$ multiplications and n additions are required to compute $P(x) = \sum_{i=0}^{n} a_i x$ even when preconditioning of the coefficients is allowed.

At this point, one could give a similar set of results as was done for the first theorem, but instead, let us just concentrate on the polynomial case we have started with, and see what results have actually been obtained with practical algorithms. We can list the following:

One algorithm uses $\frac{1}{2}(n+2)$ multiplications for $n<9$, and $\frac{1}{2}(n+1)$ multiplications for $n\geq 9$, with $n + 7$ additions. This however has the drawback that the algebraic operations involved in the preconditioning may result in complex numbers, even when the original coefficients are real, since they require taking roots of polynomials.

Another realises the computation in $\frac{1}{2}(n+4)$ multiplications and n+3 additions. This still requires operations which extract the roots of polynomials, but results in real numbers after preconditioning if the original coefficients are real.

Another algorithm shows how to do the computation in $n/2 + O(n^{\frac{1}{2}})$ multiplications and $n + O(n^{\frac{1}{2}})$ additions, using only the four arithmetic operations in the preconditioning.

But we may still have other questions about using any of these algorithms: in particular what is the effect on the numerical accuracy of these algorithms?

Taking the very specific example

$$\frac{1}{81} x^4 + \frac{5}{81} x^3 + \frac{4}{27} x^2 + \frac{11}{27} x + \frac{5}{9}$$

Horner's rule gives

$$\left(\left(\left(\frac{1}{81} x + \frac{5}{81}\right) x + \frac{4}{27}\right) x + \frac{11}{27}\right) x + \frac{5}{9} .$$

Todd and Motzkin's algorithms gives

$$\frac{1}{81} \left(\left(\left(x(x+2) + 21\right) \left(x (x+2) + x-15\right) + 360\right)\right.$$

Inspecting this last expression, we see that we are confronted with a serious problem in rounding errors when we have small x. Note that since such algorithms might well be used for building up the elementary function subroutines (such as sin x), where it is quite common to call the routine for small x, it will prove extremely unsuitable in practice. Fortunately, this is not the only way of doing the evaluation. We also have, for instance,

$$\left(\frac{1}{3} x \left(\frac{1}{3} x - \frac{2}{3}\right) + \frac{85}{81}\right) \left(\frac{1}{3} x \left(\frac{1}{3} x - \frac{2}{3}\right) + x - \frac{59}{81}\right) + \frac{8660}{6561}$$

and even better,

$$\left(\frac{1}{3} x \left(\frac{1}{3} x + \frac{1}{3}\right) + \frac{25}{27}\right) \left(\frac{1}{3} x \left(\frac{1}{3} x + \frac{1}{3}\right) + \frac{1}{3} x - \frac{1}{27}\right) + \frac{430}{729} .$$

Here the evenly-balanced coefficients greatly reduce the rounding error problems incurred by the Todd and Motzkin algorithm. Thus we see that there is still a large freedom of choice among algorithms with the same number of arithmetic operations, but having different numerical accuracy. But the freedom is even larger than this example indicates: in most cases we have rational functions approximating to the elementary functions, which adds an extra degree of freedom to the choices we can make. In general, it was found that where one has a system of polynomials, like the case of a rational function, where one has two polynomials, the freedom of choice for selecting good algorithms increases with the number of polynomials, and one can find very clever ways for optimising the algorithms.

In one practical study, Rabin and Winograd were able to find a preconditioned way of evaluating one of the elementary functions in a FORTRAN compiler for which the coefficients were given by the manufacturer in the manual. The preconditional method and the original method both resulted in the same numerical accuracy.

Finally, one should note that this method of preconditioning is generally useful wherever one has the problem of continually re-evaluating a set of expressions where some of the parameters remain fixed. Hoffmann and Winograd have looked at the problem of finding the minimum path between any two nodes in a graph, and succeeded in reducing the number of additions from $O(n^3)$ to $O(n^{5/2})$ using preconditioning. It can be said that one important concept that this area of investigation has brought to light is that of preconditioning, which has found more application than even the theory may indicate.

Discussion

Professor Hopcroft asked how it had been possible to discover that different preconditioned methods existed for the problem quoted in the lecture. Dr. Winograd replied that by studying the proof of the theorem one was able to obtain guidelines on how to search for other algorithms, even though there was not a mechanical way of looking for them. In the case cited the speaker had found three other algorithms in addition to the three that were illustrated.

Professor Page asked for an indication of the degree of difficulty and length of the proofs for which the speaker had quoted so many interesting and useful corollaries. Dr. Winograd decided that since there was still a little time remaining for this particular lecture he would sketch a partial proof of Theorem 1. The theorem is proved in two parts; firstly one proves that the number of multiplications needed is at least equal to the column rank. The first part of the proof is somewhat easier than the second and is as follows:

If $p_1, p_2, \ldots p_k$ are the k multiplications/divisions in an algorithm for computing $\Phi x + \phi$ then the value computed by every step of the algorithm can be expressed as

$$\sum_{i=1}^{k} r_i p_i + \sum_{i=1}^{k} s_i x_i + f(y_1, \ldots, y_m) \text{ where } r_i, s_i \in Q \text{ and}$$

$f(y_1, \ldots, y_m) \in Q(y_1, \ldots y_m)$.

Therefore $\Phi x + \phi = R.\, p + Sx + F$ where R is a t x k matrix of scalars, S is a t x n matrix of scalars, F is a column vector of functions in $Q(y_1, \ldots, y_m)$, and p is the (column) vector $(p_1, p_2, \ldots, p_k)$. It is easily seen that it suffices to consider the case $p_r\ (\Phi) = t$. Assume $k < t = p_r\ (\Phi)$. There exists a row vector v such that $vR = 0$ and therefore $v\Phi x = vRp + vSx + v(F-\phi) = vSx + v(F-\phi)$. But the entries of $v\Phi$ are not all scalars, and by equating coefficients we get a contradiction to the assumption that $k < p_r\ (\Phi)$.

Bi-linear Forms

Continuing on the theme of Arithemetic Complexity, we will now consider some special functions, a system of bi-linear forms. These are functions of the form

$$\Upsilon = \sum_{i=1}^{s} \sum_{i=1}^{r} a_{ijk}\, x_i y_j \quad , \qquad k = 1, 2, \ldots t,$$

and we are interested in the arithmetic complexity involved in computing the values of bi-linear forms. Naturally all the general results given earlier apply, since the function is linear both in the x's and in the y's, but we ask the question: Is it possible to achieve more by considering the fact that it is bi-linear? The answer to this question is "yes" as we shall see.

The first theorem we consider is:

Theorem 4 The minimum number of multiplications/divisions required to compute $\{\Psi_k\}$ is the same as the minimum number of multiplications.

In other words, we can say that any divisions used in the calculation of $\{\Psi_k\}$ are unnecessary, since we can do as well using only multiplications. We may ask why this theorem is even considered, since the original expression of a bi-linear form contains no divisions, but two examples will show that in some circumstances, the introduction of some divisions can reduce the number of multiplications/divisions required. For example

a) x^{31} - can be computed using 6 multiplications/divisions but requires 7 multiplications.

b) det(A) - the determinant of the matrix A can be calculated using only multiplications, but a more efficient way of calculating det(A) is by using Gaussian Elimination which of course uses divisions.

Theorem 5 The minimum number of multiplications/divisions required to compute $\{\Psi_k\}$ does not change if we demand that each product is a product of two linear forms.

Theorem 6 Let n be the minimum number of multiplications required to compute $\{\Psi_k\}$, and $\bar{n}$ the number if the commutative law is not used. Then

$n \le \bar{n} \le 2n.$

This theorem may seem a little odd, since we have been considering computations using the commutative law and now we are wondering about an algebra without the commutative law. Let us first consider an example where the commutative law helps us.

Consider the system of bi-linear forms

$$\begin{pmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{pmatrix}\begin{pmatrix} b_{11} & b_{12} & b_{13} & b_{14} & b_{15} \\ b_{21} & b_{22} & b_{23} & b_{24} & b_{25} \end{pmatrix}$$

and the following theorem of Hopcroft and Kerr:

<u>Theroem 7</u> Every algorithm for computing $A_{2\times 2} \times B_{2\times n}$ without commutativity requires at least $\left\lceil \frac{7n}{2} \right\rceil$ multiplications.

Thus applying this theorem to our example we see that without commutativity we require 18 multiplications, wheareas using the commutative law, the following algorithm gives us the result in only 17 multiplications.

$$\begin{aligned}
a_{11}\,b_{11} + a_{12}b_{21} &= (a_{11} + b_{21})(a_{12} + b_{11}) - a_{11}a_{12} - b_{11}b_{21} \\
a_{11}\,b_{12} + a_{12}b_{22} &= (a_{11} + b_{22})(a_{12} + b_{12}) - a_{11}a_{12} - b_{12}b_{22} \\
a_{11}\,b_{13} + a_{12}b_{23} &= (a_{11} + b_{23})(a_{12} + b_{13}) - a_{11}a_{12} - b_{13}b_{23} \\
a_{11}\,b_{14} + a_{12}b_{24} &= (a_{11} + b_{24})(a_{12} + b_{14}) - a_{11}a_{12} - b_{14}b_{24} \\
a_{11}\,b_{15} + a_{12}b_{25} &= (a_{11} + b_{25})(a_{13} + b_{15}) - a_{11}a_{12} - b_{15}b_{25} \\
a_{21}\,b_{11} + a_{22}b_{21} &= (a_{21} + b_{21})(a_{22} + b_{11}) - a_{21}a_{22} - b_{11}b_{21} \\
a_{21}\,b_{12} + a_{22}b_{22} &= (a_{21} + b_{23})(a_{22} + b_{12}) - a_{21}a_{22} - b_{12}b_{22} \\
a_{21}\,b_{13} + a_{22}b_{25} &= (a_{21} + b_{23})(a_{22} + b_{13}) - a_{21}a_{22} - b_{13}b_{23} \\
a_{21}\,b_{14} + a_{22}b_{24} &= (a_{21} + b_{24})(a_{22} + b_{14}) - a_{21}a_{33} - b_{14}b_{24} \\
a_{21}\,b_{15} + a_{22}b_{25} &= (a_{21} + b_{25})(a_{22} + b_{15}) - a_{21}a_{22} - b_{15}b_{25}
\end{aligned}$$

<u>Theorem 8</u> The minimum number of multiplications required to compute $\{\Psi_k\}$ is $\bar{n}$, even if we demand that each product is a product of a linear form in the x's and a linear form in the y's.

This is simply stating that without the use of the commutative law, no improvement can be obtained by mixing the x's with the y's.

<u>Theorem 9</u> If $\{\Psi_k\}$ are bi-linear forms such that $\Psi_1, \ldots \Psi_k$ are linearly independent and each of them can be computed in one product,

but it can be shown that the number of additions also grows by

$$An \log_2 7 \ .$$

This is an illustration of a property of many applications of matrices as stated in the following theorem.

<u>Theorem 10</u> The number of mult/div's required to compute $A \times B < a\, n^{\alpha}$

$\Leftrightarrow$	"	"	"	operations	"	"	"	$A \times B < b\, n^{\alpha}$
$\Leftrightarrow$	"	"	"	mult/div's	"	"	"	$A^2 < c\, n^{\alpha}$
$\Leftrightarrow$	"	"	"	operations	"	"	"	$A^2 < d\, n^{\alpha}$
$\Leftrightarrow$	"	"	"	mult/div's	"	"	"	$AB + BA < e\, n^{\alpha}$
$\Leftrightarrow$	"	"	"	operations	"	"	"	$AB + BA < f\, n^{\alpha}$
$\Leftrightarrow$	"	"	"	mult/div's	"	"	"	$A^{-1} < g\, n^{\alpha}$
$\Leftrightarrow$	"	"	"	operations	"	"	"	$A^{-1} < h\, n^{\alpha}$

where A, B are n x n matrices.

We hope that by now the reader will be convinced that bi-linear forms are as interesting and worthy of study, and that the value of non-commutativity has also been demonstrated.

Now let us take a system of bi-linear forms $\{\Psi_K\}$ and associate with it a single tri-linear form

$$T(\Psi_K) = \sum_{K=1}^{t} \sum_{j=1}^{s} \sum_{i=1}^{r} a_{ijK}\, x_i y_i z_K$$

Here we have introduced the dummy variables z_K to construct the tri-linear form.

<u>Theorem 11</u> The minimum number of multiplications required to compute $\{\Psi_K\}$ without commutativity is the smallest integer n such that $T(\Psi_K) = \sum_{i=1}^{n} L_i\,(x)\; M_i\,(y)\;\; N_i\,(z)$ where L_i , M_i and N_i are linear forms.

The proof is so simple as to be little more than an observation

$$\Psi_K = \sum_{i=1}^{n} c_{Ki}\; p_i$$ where $p_1 \ldots p_n$ are the products required in the computation of $\{\Psi_K\}$

$$\Leftrightarrow \Psi_k = \sum_{i=1}^{n} c_k L_i(x) M_i(y) \quad \text{by a previous theorem}$$

$$\Leftrightarrow T(\Psi_k) = \sum_{i=1}^{n} L_i(k) M_i(y) N_i(z)$$

$$\text{(where } N(z) = \sum_{k=1}^{t} c_k z_k\text{)}$$

From this we have

Corollary If $\Psi_k = \sum_{j=1}^{s} \sum_{i=1}^{r} a_{ijk} x_i y_j$, $k = 1 \ldots t$,

and $\bar{\Psi} = \sum_{k=1}^{t} \sum_{j=1}^{s} b_{ijk} y_j z_k$, $i = 1 \ldots r$,

are such that $a_{ijk} = b_{ijk}$ for all i,j,k then both systems require the same number of multiplications, and an algorithm for computing one system can be "transposed" into an algorithm for computing the other.

For example:

$$\begin{aligned} x_1y_1 + x_2y_2 &= (x_1 - x_2)y_1 + x_2(y_1 + y_2) \\ x_2y_1 + x_3y_2 &= x_2(y_1 + y_2) + (x_3 - x_2)y_2 \end{aligned}$$

and

$$\begin{aligned} x_1y_1 &= x_1y_1 \\ x_1y_2 + x_2y_1 &= (x_1 + x_2)(y_1 + y_2) - x_1y_1 - x_2y_2 \\ x_2y_2 &= x_2y_2 \end{aligned}$$

are "transposes" of one another, and so are the algorithms used to compute them.

A further example is that of multiplying together the matrices A and B. We observe that the tri-linear form can represent six different systems of bi-linear forms, since the x's, y's or the z's can be regarded as the dummy variable and the remaining two can be taken in either order. If we use the notation $\sim$ to mean "requires the same number of multiplications as" we have

$$A_{m \times n} \times B_{n \times p} \sim A_{p \times n} \times B_{n \times m}$$
$$\sim A_{n \times p} \times B_{p \times m} \sim A_{m \times p} \times B_{p \times n}$$
$$\sim A_{p \times m} \times B_{m \times n} \sim A_{n \times m} \times B_{m \times p}$$

Given a good algorithm for one of these problems, mechanically a good algorithm can be produced for the others.

We conclude by citing an actual example in which all these techniques are employed to obtain the solution. This example is that of a digital filter. We have an infinite sequence of incoming symbols

$$x_1, x_2, x_3 \ldots\ldots\ldots$$

and a set of weights $y_1 y_2 \ldots y_k$. We wish to compute the cross products

$$z_1 = \sum_{i=1}^{k} x_i y_i$$

$$z_2 = \sum_{i=1}^{k} x_{i+1} y_i$$

etc.

The problem may be written down in matrix form as (for k = 32):

$$\begin{pmatrix} x_1 & x_2 & x_3 & \cdots & x_{32} \\ x_2 & x_3 & x_4 & \cdots & x_{33} \\ x_3 & x_4 & x_5 & \cdots & x_{34} \\ & & \vdots & & \end{pmatrix} \begin{pmatrix} y_1 \\ y_2 \\ y_3 \\ \vdots \\ y_{32} \end{pmatrix}$$

If we consider 1024 rows, we can part the matrix so that we need to find a efficient algorithm which makes no use of the commutative law for computing

$$\begin{pmatrix} x_1 & x_2 \\ x_2 & x_3 \\ x_3 & x_4 \\ x_4 & x_5 \end{pmatrix} \begin{pmatrix} y_1 \\ y_2 \end{pmatrix}$$

where each x_i represents a 256 x 16 matrix and each y_j a 16 x 1 vector. The first result we are to use states that at least five multiplications will be required. Now we must look to see if we can find an algorithm to do the computation in exactly five multiplications. Making use (implicitly) of all the results stated earlier, we produce the following algorithm:

$$\begin{aligned}
m_1 &= (2x_1 - x_2 - 2x_3 + x_4)y_1/2 \\
m_2 &= (2x_2 + x_3 - x_4)(y_1 + y_2)/2 \\
m_3 &= (2x_2 - 3x_3 + x_4)(y_2 - y_1)/6 \\
m_4 &= (-x_2 + x_4)(2y_2 + y_1)/6 \\
m_5 &= (2x_2 - x_3 - 2x_4 + x_5)\,y_2
\end{aligned}$$

We may disregard operations on the y's as these can be done once only at the beginning of the computation. From these products we have

$$\begin{aligned}
x_1 y_1 + x_2 y_2 &= m_1 + m_2 + m_3 + m_4 \\
x_2 y_1 + x_2 y_2 &= m_2 - m_2 + 2m_4 \\
x_3 y_1 + x_4 y_2 &= m_2 + m_3 + 4m_4 \\
x_4 y_1 + x_5 y_2 &= m_2 - m_3 + 8m_4 + m_5
\end{aligned}$$

Thus for the computation of the 1024 x 32 case approximately three multiplications per row are required.

Professor Seegmüller asked whether anything was known about the loss of precision involved in the techniques described for matrix multiplication. Dr. Winograd replied by citing some work by Richard Brent of Stanford University who had taken three methods for matrix multiplication, the straightforward method, a method related to the $A_{2x2} \times B_{2x5}$ method described earlier which took $\frac{1}{2}n^3$ multiplications, and the method described related to Strassen's method. This study was carried out on a large number of sets of data, and the methods compared for speed and for numerical accuracy, and his final conclusion taking numerical accuracy into account was that the method taking $\frac{1}{2}n^3$ multiplications was superior. This, however, was a purely experimental study with no theoretical investigation.

Analytic Complexity

Analytic complexity addresses the question "How much computation has to be performed to obtain a result with a given degree of accuracy?". Interest focuses on those computational processes which in a certain sense never end. The process is interrupted at some point and, if the current value of the result lies within some error bound, that value is accepted, otherwise the computation is continued until a satisfactory result is obtained.

One of the simplest examples of analytic complexity is a study of algorithms for extracting the square root of a number. The variety of known algorithms is illustrated by six examples contained in Table 1. To facilitate a comparison of these methods the numbers of arithmetic operations used in one iteration of each algorithm are tabulated. Note that scalar multiplications have been counted separately.

The first example is Newton's method, which has a power of convergence of two. This means that on each iteration the number of significant digits in the result is roughly doubled. The second method, basically Newton's method in disguise, has three multiplications per iteration but no divisions. In certain situations, for instance a machine on which division takes very much longer than multiplication, this may be a very attractive method. The division $^1/a$ has to be performed once at the start of the computation, but thereafter no divisions are necessary.

Example 4 is a somewhat unusual method which will extract the square roots of numbers between 0 and 2. It involves essentially double iteration but it is the value of X_n which approaches the square root. The last method has a power of convergence of four, so that on each iteration one obtains four times the number of significant digits. In fact, this last formula is obtained by applying Newton's method twice, and therefore it is not surprising that the power of convergence is four, which is the square of the power of convergence of Newton's method, and the number of multiplications and divisions is two. In general, it is possible to take any of these methods and produce new algorithms in a similar manner.

Table 1

1. $X_{n+1} = \frac{1}{2}(X_n + a/X_n)$ $\qquad p = 2 \quad d = 1$

   $sm = 1 \quad a/s = 1$

2. $X_{n+1} = \frac{1}{2}(3X_n - (\frac{1}{a})X_n^3)$ $\qquad p = 2 \quad m = 3$

   $sm = 1 \quad a/s = 3$

3. $X_{n+1} = \dfrac{X_n X_{n-1} - a}{X_n + X_{n-1}}$ $\qquad p = (1+\sqrt{5})/2 \approx 1.618$

   $m/d = 2 \quad a/s = 2$

4. $X_0 = a, \; y_0 = \dfrac{a-1}{2}$ $\qquad p = 2$

   $m = 3 \quad a/s = 2$

   $X_{n+1} = X_n(1 - y_n), \quad y_{n+1} = y_n^2(1.5 + y_n)$

5. $X_{n+1} = \dfrac{1}{8X_n^3}(X_n^2 + a)\;(5X_n^2 + a)$ $\qquad p = 3$

   $m/d = 4 \quad a/s = 5$

6. $X_{n+1} = \frac{1}{4}(X_n + a/X_n) + \dfrac{a}{X_n + a/X_n}$ $\qquad p = 4 \quad m/d = 2$

   $sm = 1 \quad a/s = 2$

p	power of convergence
d	number of divisions/iteration
m	number of multiplications/iteration
m/d	number of multiplications and divisions/iteration
sm	number of scalar multiplications/iteration
a/s	number of additions or subtractions/iteration

Algorithms for extracting the square root of a number

If one of these methods is applied say three times and the resulting formula used to extract the square root, the number of multiplications will have increased threefold and the power of convergence will have risen by the power three. In any comparison of these methods, one would want to treat the first and last examples as being the same because they are both applications of the same algorithm. The measure used to compare methods of extracting the square root of a number is

$$\gamma = \frac{\log_2 \text{(power of convergence)}}{\text{number of multiplications and divisions/iteration}}$$

Ideally, one would like to use an algorithm which makes γ as large as possible; a combination of a large power of convergence and a small number of multiplications or divisions/iteration. Unfortunately, the following result by Patterson indicates that there is an upper limit on γ.

Theorem 12 For every iterative method for finding the square root of a number, $\gamma \leq 1$.

Newton's method, for which $\gamma = 1$, achieves this bound.

This illustration of one area of analytic complexity attempts to give a flavour of the questions which are being asked. A similar example is that of general iterative methods; one is given a function f and is required to find its root. Table 2 gives three examples of such methods.

The first method has a power of convergence 1.618 and possesses the useful property that it requires no knowledge of the derivative of the function in question. So, this algorithm is applicable in cases where it is difficult to calculate the derivative of a function. Newton's method is the second example, which has a power of convergence of two and uses the first derivative. In general, if one studies the literature one finds that there are iterative methods with arbitrary high powers of convergence, but they require higher and higher derivatives. This observation leads one to ask if there is some kind of relation between the power of convergence and the number

Table 2

1. $X_{n+1} = \{X_{n-1}\, f(X_n) - X_n\, f(X_{n-1})\}/(X_n - X_{n-1})$

power of convergence = $(1 + \sqrt{5})/2 \simeq 1.618$; uses no derivatives.

2. $X_{n+1} = X_n - f(X_n)/f'(X_n)$

power of convergence = 2; uses first derivative.

3. $X_{n+1} = X_n - \frac{1}{2}\frac{f(X_n)}{f'(X_n)}\ (1 + f(X_n)\ f''(X_n))/\{f'(X_n)\}^2$

power of convergence = 3; uses second derivative.

of derivatives that have to be found. The following theorem gives a bound on the power of convergence in terms of the highest derivative used.

Theorem 13 Every iterative method using derivatives up to order d has a power of convergence of at most $d + 2$.

It is immediate from this theorem that, if no derivatives are used, the largest attainable power of convergence is two. In practice, the bound $d + 2$ can be approached, since for every $\epsilon > 0$ there exists an iterative method whose power of convergence exceeds $d + 2 - \epsilon$.

Thus general iterative methods provide another example where one is trying to compare how much computation has to be performed in order to achieve a certain accuracy. A final example concerns the problem of parallelism. Assuming one has k processors working in parallel what is the resulting gain in speed in execution of an algorithm that can be expected over the single processor case? Here we disregard problems of synchronisation, availability of information etc., and assume such problems can be solved.

If one considers an iterative method, but this time with k processors, it is possible to compute the k next approximations, from which one can choose the most satisfactory. These k approximations are evaluated in parallel but do not cause any conflict. The power of convergence increases from $d + 2$ to $k(d + 1) + 1$, an increase of almost k-fold. This appears to be a useful improvement but consider what is meant by the power of convergence. Assume that the initial approximation has error δ, and that one wants to continue to iterate until an approximation to the solution is obtained within ϵ of the solution (hopefully $\epsilon \ll \delta$), then the number of iterations which has to be performed is given by

$$n \sim \log_p \log_{1/\delta} (1/\epsilon)$$

where p is the power of convergence.

If $(d+2)$ and then $k(d+1)+1$ are substituted for p, the reduction in the number of iterations, which is the gain in speed of the parallel case over the single processor case, is only logarithmic in the number of processors rather than being linear. Thus if 100 processors were used in parallel the number of iterations necessary would be reduced

10-fold not by 100-fold.

Discussion

Professor Dijkstra pointed out that the fourth example in the methods of extracting square roots was of a different nature to the others. To obtain a certain precision with this method it is necessary to maintain that precision from the start, whereas the other algorithms permit the first step to be computed in small precision. Dr. Winograd explained that this method had been included to illustrate the wide variety of possible methods. It does have the unfortunate property of propagating any errors, whereas the other methods all include a in the formula and this has a correcting effect. This lecture concentrated on the relation between the amount of computation that has to be done to achieve a certain amount of accuracy; the properties of the algorithms which are therefore of interest are the power of convergence, which indicates how fast the error reduces, and the number of arithmetic operations per iteration. The six methods have very different properties from the point of view of stability.

Professor Dijkstra also expressed concern that the amount of computation required to evaluate a square root was measured in terms of the number of multiplications or divisions required per iteration of the algorithm, as though this was independent of the precision with which the computation had to be performed. This concern was shared by Dr. Winograd who repeated the remark of Dr. Rabin that we prove what we know how to prove. The study of analytic complexity has only provided a partial answer to the question of the amount of computation that has to be performed to achieve a square root to a given accuracy

In connection with the general iterative methods, Professor Page suggested that a possible way to obtain a higher rate of convergence would be to apply the formula twice. So if the original formula involved X_n and X_{n-1}, substitute for X_{n-1} a formula involving X_{n-2}.
Dr. Winograd explained that if f is known explicitly, for example $f(x) = x^2 - a$, this can sometimes be done (as was done in the sixth square root example to obtain a power of convergence of four) but in general it is not possible. Consider Newton's method:

$$X_n = X_{n-1} - \frac{f(X_{n-1})}{f'(X_{n-1})}$$

$$X_{n-1} = X_{n-2} - \frac{f(X_{n-2})}{f'(X_{n-2})}$$

Substituting for X_{n-1}

$$X_n = X_{n-2} - \frac{f(X_{n-2})}{f'(X_{n-2})} - \frac{f(?)}{f'(?)}$$

The problem arises in the substitution for X_n in $f(X_n)$ and $f'(X_n)$. Since f is not known as a function of variables, nothing is gained by making such a substitution. If f is known explicitly, it might be possible to compute $f(X_n)$ and $f'(X_n)$ in terms of X_{n-1}

Professor Michaelson asked whether the last result, concerning parallelism, included the potential increase in speed of a single iteration by a factor of k, due to the use of k processors. Dr. Winograd said that it did not and explained that if the gain in speed for doing arithmetic operations is considered the situation is quite different. Combining results from the two areas suggests that if one has k processors and an iterative method which is to be executed, the best way to employ the k processors is to leave the overall method as a sequential one, but within each iteration do the arithmetic in parallel. This corresponds roughly to doing k separate jobs in parallel. On average there would be an improvement by a factor of k even though no one job would have such an improvement.

Teaching Aspects

In summary, we ask whether any of the material which has been presented in this series of lectures should be incorporated in a University Computing Science course at some level. We believe that there are four reasons why the Complexity of Computation has a place in a Computer Science Curriculum.

(a) It raises some interesting mathematical questions.

(b) Although interest, like beauty, is in the eye of the beholder, most Computer Scientists would agree that some interesting Computer Science questions are raised.

(c) During these lectures we have been introduced to new concepts or new ways of thinking.

(d) These new concepts have given rise to new algorithms, that is, the new ways of thinking about a problem ha led to new ways of doing the problem.

Discussion

Professor Hoare made the comment that from what he had seen in these lectures, it seemed that FORTRAN was the worst possible tool available to people interested in numerical computation. On being asked to explain further he said that it was quite clear from what Dr. Winograd had said that an understanding of recursion was essential to the understanding of these techniques, and naturally, programmers, and students in particular, who write in FORTRAN have a very limited grasp of the principles of recursion.

References

J. Todd, "Motivation for Working in Numerical Analysis", Communications in Pure and Applied Mathematics (1955).

T.S. Motzkin, "Evaluation of Polynomials and Evaluation of Rational Functions", Bulletin of the American Mathematical Society (1955).

J.D. Hopcroft and L.R. Kerr, "Some Techniques for Proving Certain Simple Programs Optimum", (1969) Tenth Annual Symposium on Switching and Automata Theory.

R.P. Brent, "Algorithms for Matrix Multiplication", Stanford University Report STAN-CS-70-157.