ROCKY MOUNTAIN
JOURNAL OF ﬃATHEMATICS
Volume 17, Number 1, Winter 1987

COMPLEXITY OF COMPUTER ALGORITHMS

RONALD J. LEACH

Introduction. This paper is intended to provide an introduction to the study of complexity of computer algorithms. No special knowledge of computers is needed; the emphasis is on the ideas involved in algorithms and not on any special features of any particular computer or computer language. Our viewpoint is that a computer is a large, dumb machine capable of doing arithmetic and comparisons at an extremely rapid rate. It performs these operations according to a precisely given set of instructions called a computer program.

We will not attempt to survey the entire field of algorithm complexity in this paper; instead we will concentrate on a few algorithms indicating some current problems and directions in computer science and related problems in mathematics.

The paper is divided into four parts. Part one contains a very brief discussion of von-Neumann's model of a computer (as a sequential rather than a parallel machine). It also includes an introduction to measurement of the complexity of an algorithm. Part two is concerned with arithmetic complexity theory, i.e., minimizing the time required by an algorithm which computes something. It begins with a discussion of a simple problem: evaluation of a polynomial of degree N at a point with as few multiplications as possible. Evaluation of the polynomial at N points is then discussed briefly en route to a discussion of the Fast Fourier Transform, which is developed using the important technique of divide-and-conquer. This leads naturally to a discussion of recursion and recursive algorithms.

In part three we consider some non-arithmetic algorithms. Several sorting algorithms are discussed and a lower limit on the number of comparisons for sorting an arbitrary file is given.

Part four will include a summary and some speculations about the direction of research in complexity theory.

In this survey we will only consider a few problems and techniques; it is not the intention to provide a complete description of current work on complexity theory. For additional surveys see [**4**, **8**, **42**]; for a leisurely introduction to some of the ideas in algorithm design see [**20**]; and for the

Received by the editors on April 23, 1985 and in revised form on June 24, 1985.

state of the art of NP problems see **[16]** and the series of articles by D. Johnson **[21]** in the Journal of Algorithms.

Finally, any author purporting to show a connection between some areas of computer science and mathematics would be remiss if *s*/he did not mention the excellent series by D. Knuth [27].

1. Complexity measures and von-Neumann machines.

1.1 *Algorithms.* An algorithm is a definite procedure for solving a problem in a finite number of steps. Often the terms algorithm and program are used interchangeably. We will be considering the complexity of algorithms as being defined relatively: algorithm A is more complex than algorithm B if the time to run program A is longer than that of program B.

Note that is possible that algorithm A runs faster than algorthm B for some inputs, but runs considerably slower than B for other inputs. We will be concerned primarily with the worst case performance of the run-time on inputs of the same length, although average case and best case performance are also considered briefly. On an absolute scale, we often are interested in the run-time as a function of the length of the input. The simple example given below indicates how these ideas are used in a practical situation.

Suppose we have a set of three numbers, say 1, 3, 2 and wish to find their mean and median, and to sort them. To most mathematicians, this is an easy problem. Note that computing the mean requires slightly more work (two additions, one division and a count of the elements) instead of three comparisons. If we had 100 numbers, we would need 99 additions and one division to compute the mean but a large number of comparisons to find the median. (We will see later any method of sorting N numbers using comparisons must make at least O (N log N) comparisons.)

For small sets of numbers it is clearly faster to find the mean and median by hand than to use a computer. To find the median of the set $\{1, 2, \ldots, 99\}$ is still easier for us than it is for the computer. We can recognize that the set is given in increasing order while a computer cannot unless we have already programmed it with instructions for determining if a set is already sorted.

We assume that a computer has a memory which is an $n \times 1$ vector whose entries are binary numbers. The number of binary digits or bits in each number depends on the particular computer. All numbers, letters, symbols, instructions, etc. that the computer understands are encoded into these binary numbers. A complete collection of instructions to perform some task is called a computer program, which is stored in memory in the form of these binary numbers. The computer has a central processing unit (cpu) in which all arithmetic operations and comparisons are performed. Any two numbers in memory can be added by first moving one number to a special register, the accumulator, of the cpu, adding the second number and storing the results somewhere in memory. Thus the statement

$$x = x + 1$$

in a computer program does not indicate an equation with no solutions; rather, it indicates that the number in location x is brought to the accumulator, added to 1 and the result is stored back in location x. This use of variable names is of course different from the usual use in mathematics. (See the recent article **[40]** for an excellent short discussion of possible uses of variable names in a computer.) Comparisons between the contents of the accumulator and those of location x are done similarly. This is the classic von-Neumann model of a computer.

A von-Neumann machine is a sequential machine; it executes instructions in the sequence required by the program and it considers data one "data unit" (= one integer or one decimal representation or one character, etc.) at a time. A considerable amount of work is being done currently on parallel or distributed processing where there can be multiple processors acting simultaneously on the same "data unit" or a single processor acting upon many "data units" or even many processors working on many streams of "data units". In this paper we consider only sequential algorithms for sequential (von-Neumann) machines. The analysis of parallel algorithms is in its infancy compared to the analysis of sequential algorithms which itself is less than 20 years old. A good survey article on parallel algorithms is [7].

One final comment on parallel processing is in order. Many computer systems have the ability to perform a limited number of operations in parallel. For example, the representation of a complicated object such as the space shuttle on a computer terminal screen requires a large number of line segments just to represent the shape. Showing the effects of rotation requires many repetitions of multiplications of vectors by certain matrices. Computing the entries of the product using several processors in parallel can speed up the display so that motion is shown in real-time rather than being delayed. In general, when we say parallel processing we mean general purpose machines acting together rather than just special purpose arithmetic processors.

1.2. *Computer arithmetic.* A few comments on computer arithmetic are in order. Most computers store integers differently from the way they store decimals, or reals as they are usually called in computerese. In many personal computers an integer must have absolute value at most 2^{15} since it is stored in 16 bits or binary digits, with the first reserved for the sign. A "real" is stored in 32 bits with the first bit for the sign, the next 11 for a "base-2 characteristic" which can represent positive or negative powers, and the remaining 20 for the "base-2 mentissa". Addition requires many onebit additions and carries. Multiplication on the other hand requires many shifts (multiplication by single bits) and many additions. Overflow is checked for after the operation is performed. Thus, in general, multiplication takes longer than addition. All other things being equal, fewer multiplications mean faster algorithms.

(Knowledge of the way a computer stores numbers helps to point out the difficulties in doing arithmetic on a computer. We should not expect to find the repeating digits in the expansion of say 11/89, in the limited amount of space available for storage of numbers. Even using a computer whose design permits "double precision" numbers with a total of 64 bits for storage obviously cannot resolve all problems of this type. Rational arithmetic is therefore not usually done in hardware; rather, it is done by using software such as LISP. The list structure of LISP is well suited to the representation of rational numbers and to the representation of decimals to arbitrary precision. See **[41]** and **[40]** for a more complete discussion.)

2. Arithmetic complexity theory.

2.1. *Polynomials.* Suppose we have a polynomial P of degree N and we wish to evaluate this polynomial at many points, perhaps in order to plot its graph. A first attempt at an algorithm might be something like the following.

For each value of x between -10 and 10 with increment .1, compute

$$P = a_0$$
$$P = a_1 \cdot x + a_0$$
$$P = a_2 \cdot x^2 + a_1 \cdot x + a_0$$
$$P = a_n \cdot x^n + a_{n-1} \cdot x^{n-1} + \cdots + a_0.$$

This is written succintly in the computer language BASIC as

```
FOR X = -10   TO   10   STEP  .1
          P = 0
          For   K = 0   TO   N
                P = A(K) * X^K + P
          NEXT   K
NEXT X
```

A few words on notation are in order. Most programming languages do not accept subscripts. It is easy to get around this by writing $A(K)$ instead of a_k. The symbol "^" means "raised to the power"; tnis is a feature of the language BASIC as implemented on a particular computer, and we will follow this notation rather than the "**" notation of FORTRAN. The "*" denotes multiplication.

Notice that this algorithm requires the computation of x^k at each step. A faster algorithm for an arbitrary polynomial of degree N is

```
FOR X = -10 TO 10 STEP .1
    P = 0
    Y = 1
    FOR K = 0 TO N
        P = A(K) * Y + P
        Y = Y*X
    NEXT K
NEXT X
```

This speeds up the program by a considerable factor. In one test example involving a 10th degree polynomial, the author noted a running time decreasing from 44 to 16 seconds. An analysis of these algorithms will lead us to the important idea of an essential multiplication or division step which is developed in the next section.

In the first algorithm, the outer loop is executed 201 times since there are 201 points at which $p(x)$ is evaluated. For each one of these points, the inner loop is executed 11 times. Each pass through the inner loop means an addition to P and the computation of x^k. This computation is performed either by repeated multiplication or, more commonly, by the formula

$$x^k = \exp(k \log x)$$

which involves a fair amount of work for the computer using the built-in functions exp and log. For the purposes of this note we assume that the two methods are similar in the amount of time that a computation takes, at least for "most" small exponents. We simply call such exponentiations "multiplications". A more detailed explanation will be given later in this section.

Thus, in the first algorithm there are 201 evaluations from the outer loop, each of which requires $1 + 2 + \cdots + 10 = 55$ multiplications to compute the powers of x and 10 multiplications by the various coefficients or 65 multiplications. There are also 10 additions per evaluation so that the first algorithm requires (65) (201) or 13065 multiplications and (10) (201) or 2010 additions.

In the second algorithm, there are still 201 evaluations. The number of additions per evaluation is still 10, but there are now 22 multiplications per evaluation for a total of 22 × 201 or 4422 multiplications.

There are algorithms for evaluating polynomials of degree n at a single point using fewer than $2n + 1$ multiplications. Thus the test polynomial of degree 10 could be evaluated using fewer than 21 multiplications at each point. One such algorithm is commonly known as Horner's method, although it was originated by Newton. The polynomial is written in the form

$$\mathrm{P}(x) = (\cdots((a_nx + a_{n-1})\,x + a_{n-2})x + \cdots + a_1x) + a_0.$$

This method still requires N additions for a polynomial of degree N, but only N multiplications.

A BASIC program for this algorithm for a polynomial of degree N is:

```
FOR  X = -10  TO  10  STEP  .1
     P = A(N)
     FOR  K = N - 1  TO  0
       P = P*X + A(K)
     NEXT  K
NEXT  X
```

For our example of a polynomial of degree 10 with 201 evaluation points, we have 10 × 201 or 2010 multiplications and 10 × 201 or 2010 additions. This program took about 8 seconds to run.

Simply counting the number of multiplications indicates that the second and third algorithms should run 13065/4422 = 2.95 and 13065/2010 = 6.5 times as rapidly as the first for a 10th degree polynomial. Observed run time ratios are 44/16 = 2.75 and 44/8 = 5.5, surprisingly good agreement since an ordinary watch was used to estimate time. This reinforces the statement made in §1, that minimizing multiplications is essential to improving the runtime of the algorithm.

2.2. *Formal definition of arithmetic algorithm.* It is possible to formalize several of the ideas of the previous subsection. We note that the increase in speed seemed to be caused by reducing the number of multiplications needed. Following Winograd [**42**], we make the following definitions.

The inputs to an algorithm are a set $B = \{x_1, \ldots, x_n\}$. We suppose that both B and a field G called the field of constants both belong to a field H. An algorithm A over the set B is a finite sequence $h_1, \ldots, h_n$ of elements of H such that either

$$h_i \in B$$

or

$$h_i = h_j \circ h_k,$$

where $j, k < i$ and "$\circ$" is either $+$, $-$, $*$ or $/$. An algorithm $A = (h_1, \ldots, h_n)$ is said to compute a set $F = \{f_1, \ldots, f_t\}$ if $F \subseteq A$.

An h_i is said to be a non-multiplication—division step (non m/d step) if either

(i) $h_i \in B$;

(ii) There are $j, k < i$ so that $h_i = h_j \pm h_k$; or

(iii) There is a $j < i$ and $g \in G$ so that $h_i = g \cdot h_j$.

Otherwise h_i is called an essential m/d step.

Thus x_1/x_2, $x_2 \cdot x_7$ and x_1^2 are essentially m/d steps, while $2x_1 - 5$ and $x_1 - x_2$ are not, since they involve either addition or substraction or one of the operands is from the field of constants. The distinction between multiplications and divisions with constants and the essential m/d steps defined above is made primarily for empirical reasons. In many cases, constants are either positive or negative powers of 2. Multiplying or dividing by a power of 2 means performing a "shift" operation on the binary digits in some register, which is often faster than normal multiplication. There are often other small gains in fetching and operating on constants as opposed to operations on variables. For these and other reasons, we only consider the essential m/d steps as defined above rather than simply counting multiplications and divisions, thus distinguishing the field G of constants from H.

One aim of arithmetic complexity theory is to minimize the number of m/d steps in an algorithm. Note that this goal is in agreement with the empirical evidence observed in the previous section.

Using our new terminology, we can say that Horner' method requires n m/d steps and n additions. These are the smallest numbers possible, unless we "pre-process" the polynomial before the evaluations. As an example of this "pre-processing" consider the 4th degree polynomial

$$P(x) = a_4x^4 + \cdots + a_0.$$

An example of Motzkin and Todd (see [20]) shows that we can write

$$\begin{aligned} P(x) &= a_4((x(x + \alpha_0) + \alpha_1)(x(x + \alpha_0) + x + \alpha_2) + \alpha_3) \\ &= a_4(x^4 + (2\alpha_0 + 1)x^3 + (\alpha_1 + \alpha_2 + \alpha_0(\alpha_0 + 1))x^2 \\ &\quad + (\alpha_0\alpha_2 + \alpha_0\alpha_1 + \alpha_1)x + (\alpha_1\alpha_2 + \alpha_3)), \end{aligned}$$

where

$$\alpha_0 = (a_3 - a_4)/2a_4,$$

$$\alpha_1 = \frac{a_1}{a_4} - \frac{a_2}{a_4}(a_0^2(a_0 + 1))$$

$$\alpha_2 = \frac{a_2}{a_4} - \alpha_1 - \alpha_0(\alpha_0 + 1),$$

$$\alpha_3 = \frac{a_0}{a_4} - \alpha_1\alpha_2.$$

After the "preprocessing", the algorithm is

$$\begin{aligned} S_1 &= x(x + \alpha_0) \\ S_2 &= (S_1 + \alpha_1)(S_1 + x + \alpha_2) \\ S_3 &= a_4(S_2 + \alpha_3) \end{aligned}$$

which requires 3 m/d steps and 5 additions.

The best result for evaluating a general nth degree polynomial at a single point is (see [**20**], and [**42**])

n adds, n m/d steps	(no pre-processing)
at least n adds, $[(n + 1)/2] + 1$ m/d steps	(with pre-processing).

The lower bound given in the case of pre-processing is nearly best possible. In [**21**, vol. 2, p. 474–478], it is shown that every polynomial u of degree n (n larger than 2) can be evaluated by the scheme

$$\begin{aligned} &y = x + c, \qquad w = y^2; \\ &z = (u_n y + A_0)y + B_0 \qquad (n \text{ even}); \\ &z = u_n y + B_0 \qquad (n \text{ odd}), \\ &u(x) = (\cdots(z(w - A_1) + B_1)(w - A_2) + B_2)\cdots)(w - A_m) + B_m \end{aligned}$$

for suitably chosen c, A_k and B_k, giving a total of $[n/2] + 2$ multiplications since $m = [n/2] + 1$. The details of the computation of c, the A_k and the B_k can be found in the reference cited above.

Reducing the number of multiplications is especially important if we must evaluate the polynomial at a large number of points. If the points form an arithmetic progression, then as was pointed out in [**27**, vol. 2, p. 469], the evaluation can be reduced to addition only after the first few steps. In the next section we will consider the Fast Fourier Transform which involves evaluating terms of the form w^j for $w = \exp(2\pi i/N)$ for $0 \leqq j \leqq N - 1$. We close this subsection with an example involving the Tchebycheff polynomials and a slightly different view of pre-processing.

The nth Tchebycheff polynomial $T_n(x)$ on the interval $[-1, 1]$ is defined by

$$T_n(x) = \cos(n \arccos x).$$

The first few polynomials are

$$\begin{aligned}
T_0(x) &= 1,\\
T_1(x) &= x,\\
T_2(x) &= 2x^2 - 1,\\
T_3(x) &= 4x^3 - 3x,\\
T^4(x) &= 8x^4 - 8x^2 + 1.
\end{aligned}$$

Suppose we wish to evaluate the polynomial

$$p(x) = ax^4 + bx^3 + cx^2 + dx + e$$

at a number of points on the interval $[-1, 1]$. Solving of the system of T_n's for $1, x, x_2, x_3, x_4$ gives

$$\begin{aligned}
p(x) = {} & (1 + c/2 + 3a/8)\ T_0(x)\\
& + (d + 3b/4)\ T_1(x)\\
& + (a/2 + c/2)\ T_2(x)\\
& + (b/4)\ T_3(x)\\
& + (a/8)\ T_4(x).
\end{aligned}$$

Note that all the multiplication of the T_n's are by constants and hence these are not essential m/d steps in the sense of Winograd. The difficulty is in the evaluation of the $T_n(x)$. Recall that, on the interval $[-1, 1]$, $T_n(x) = \cos(n \arccos x)$. Evaluation of cosines and inverse cosines can be quite time consuming. However, if we know in advance the values of x which we will use to evaluate p, then a table of values of cosines and inverse cosines can be set up in advance with the desired accuracy. The value of arccos x is obtained from the table. Values of n arccos x are then computed, which involves either multiplication by constants or repeated addition. Neither of these relatively fast computations involves an essential m/d step. The value of $T_n(x)$ is then obtained by looking it up in a table of cosines. Finally, $p(x)$ is computed by using the expansion into the T_n's. The computation of $p(x)$ involves the construction of two tables, multiplications by constants and additions, but no essential m/d steps. This situation is highly dependent on the interval $[-1, 1]$ and has obvious round-off and truncation errors which can be easily estimated for any polynomial $p(x)$. We invite the reader to experiment with this and similar algorithms for polynomial evaluation.

2.3. *The Fast Fourier Transform.* From §2.2, we see that evaluation of a polynomial P of degree N at N points can be performed in $O(N^2)$ operations. An extremely important special case occurs when P has terms of the form w^j for $w = \exp(2\pi i/N)$, $0 \leqq j \leqq N - 1$.

The discrete Fourier transform (DFT) of a function a with sample points $a_0, a_1, \ldots, a_{N-1}$ is defined by

$$A_j = \sum_{0 \leqq k \leqq N-1} a_k \exp(2\pi ijk/N), \quad 0 \leqq j \leqq N-1,$$

with the inverse transform given by

$$a_k = (1/N) \sum_{0 \leqq j \leqq N-1} A_j \exp(-2\pi ijk/N), \quad 0 \leqq k \leqq N-1.$$

The evaluation of a discrete Fourier transform is precisely the same as the evaluation of a polynomial at N distinct points and thus by Horner's method can be done in $O(N^2)$ operations. The Fast Fourier Transform (FFT) is a method for computing the DFT in $O(N \log N)$ operations, an incredible speedup. The FFT came into common use in 1965 when it was popularized by Cooley and Tukey [**10**]. See [**9**] for a history of the FFT.

We describe the procedure for computing the FFT. The algorithm is written in a language similar to Pascal. The feature of the language we will use is recursion—ability to define a function or procedure in terms of itself. For a more complete discussion of these ideas, see [**20**, Chap. 9] from which most of this material is taken. The article [**40**] contains an elegant example of a recursive program written in LISP.

We assume for the moment that N is a power of 2. Note that if $N = 2n$ and w is a primitive N-th root of unity, then $-w^j = w^{j+n}$.

We will use the technique of "divide-and-conquer":

1. Divide the problem into simpler sub-problems.

2. Continue the subdivision until all sub-problems can be solved.

3. Generate a solution to the original problem from the solutions to the sub-problems.

Let $a_{N-1}, \ldots, a_0$ be the given coefficients. Then we break up a polynomial a with coefficients a_j as follows:

$$\begin{aligned} a(x) &= a_{N-1}x^{N-1} + a_{N-2}x^{N-2} + a_{N-3}x^{N-3} + \cdots + a_0) \\ &= [a_{N-1}x^{N-1} + a_{N-3}x^{N-3} + \cdots + a_1x] \\ &\quad + [a_{N-2}x^{N-2} + a_{N-4}x^{N-4} + \cdots + a_2x^2 + a_0] \\ &= [a_{N-1}y^{n-1} + a_{N-3}y^{n-2} + \cdots + a_1] \cdot x \\ &\quad + [a_{N-2}y^{n-1} + a_{N-4}y^{n-2} + \cdots + a_1y + a_0] \\ &= [c(y)] \cdot x + b(y), \end{aligned}$$

where $y = x^2$ and $N = 2n$. Thus we have written $a(x)$ as the sum of two polynomials. We have

$$a(w^j) = c(w^{2j})w^j + b(w^{2j})$$

and

$$\begin{aligned} a(w^{j+n}) &= c(w^{2(j+n)})w^{j+n} + b(w^{2(j+n)}) \\ &= c((-w^j)^2)\,(-w^j) + b((-w^j)^2) = -c(w^{2j})w^j + b(w^{2j}) \end{aligned}$$

for $0 \leqq j \leqq n-1$ since $-w^j = w^{j+n}$.

Hence in the case that N is even, $N = 2n$, we can reduce the evaluation of the N values of the A_k to the evaluation of the $N/2$ values of similar expressions for the polynomials $b(x)$ and $c(x)$. If N is a power of 2, this procedure can be continued until the polynomials $b(x)$ and $c(x)$ have degree 1.

A program to use this divide-and-conquer technique is now given. Since divide-and-conquer is naturally suited to recursion, the program is written recursively. Note also that the program is written requires that N is a power of 2. Notation is as follows: $a(x)$ is the polynomial mentioned above, w is a root of unity and A is an array to hold the coefficients of $a(x)$.

```
Program FFT ($N$, $a(x)$, $w$, $A$)
BEGIN
        IF $N = 1$ THEN $A(0) = a_0$
        ELSE
            BEGIN
                $n = N/2$
                $b(x) = a_{N-2}x^{n-1} + \cdots + a_2x + a_0$
                $c(x) = a_{N-1}x^{n-1} + \cdots + a_3x + a_1$
                CALL FFT ($n$, $b(x)$, $w^2$, $B$)
                CALL FFT ($n$, $c(x)$, $w^2$, $C$)
                $WP(-1) = 1/w$
                FOR $j = 0$ TO $n$ DO
                  BEGIN
                    $WP(j) = W * WP(j-1)$
                    $A(j) = B(j) + WP(j) * C(j)$
                    $A(j+n) = B(j) - WP(j) * C(j)$
                  END
        END
END.
```

The BEGIN-END pairs indicate logically grouped blocks of statements.

Let us examine this program line by line.

1–4. We are given N and the a_j and wish to find the A_k. If $N = 1$, the problem is trivial. A is an array which will hold the coefficients of $a(x)$.

6–8. Split the evaluation into two parts.

9–10. Call the procedure FFT again to act on the smaller inputs $b(x)$, $c(x)$. B and C are arrays that will hold the coefficients of $b(x)$ and $c(x)$

11–17. This loop actually evaluates the coefficients. Notice that the first time through the loop $n = N/2$ and so lines 15 and 16 refer to the first and second halves of the array $A(1), \ldots, A(N)$. The main idea being used here is that, for $N = 2n$, if w^j, $0 \leqq j \leqq N - 1$ are the primitive N-th roots of unity, then w^{2j}, $0 \leqq j \leqq n - 1$ are the primitive n-h roots of unity. The recursion proceeds until the procedure is invoked with $n = 1$. At that point, line 3 applies and the program continues until the FFT is evaluted.

How do we measure the computing time for this algorithm? The answer is obtained by noting the recursive structure of the program.

Let $T(n)$ be the time for the algorithm to work given n inputs. We wish to find $T(N)$ in terms of the initial number N of points. Let us examine the key step of dividing a problem of size n into two problems of size $n/2$. Let $b(x)$ and $c(x)$ be the two polynomials given by this division of a polynomial of degree n. Then $b(x)$ and $c(x)$ have degree $n/2$ and hence, by Horner's method, the number of essential m/d steps is $2(n/2)$ for evaluation of both $b(x)$ and $c(x)$. The number of m/d for the loop in lines 12–17 is n. Thus

$$T(n) = 2T(n/2) + (\text{time for } b(x),\ c(x),\ \text{loop}).$$

The second term in the sum is no larger than a constant times $\sum_{k=1}^{N} n(k)$ where $n(k)$ is the value of n during the k^{th} procedure call. But $n(k) = N/2^k$ and hence this sum is $O(N)$. Thus the recursive formula for the run-time is $T(N) = 2T(N/2) + cN$, where c is a constant. Suppose $N = 2^m$. Then

$$\begin{aligned} T(N) &= 2T(2^{m-1}) + c2^m \\ &= 2[2T(2^{m-2}) + c2^{m-1}] + c2^m \\ &= \cdots \\ &= mc2^m + T(1)2^m. \end{aligned}$$

Since $T(1)$ is also a constant, we have with $N = 2^m$,

$$\begin{aligned} T(N) &= cN \log_2 N + T(1)N \\ &= O(N \log_2 N). \end{aligned}$$

Remarks. 1. For a more complete introduction to the method of divide-and-conquer, see [**20**, Chap. 3].

2. The previous analysis ignores to a large extent the considerable amount of overhead involved in a computer's keeping track of recursive calls of procedures. Many algorithms were first developed recursively. They were then rewritten to replace the recursion with explicit iteration. Computers keep track of the procedures that are being called in a special are which is usually called the system stack. Many small computers have small system stacks and so recursive programs which work well on large computers fail on small personal computers even though the personal computer has sufficient memory.

Also, a computer must do a considerable amount of housekeeping while keeping track of a program calling itself with other parameters.

In the procedure given by this program for example, the initial parameters are N, $a(x)$, w, and A. If $N = 4$, the program continues execution until it reaches line 9. The procedure is then called again with parameters $4/2 = 2$, $b(x)$, w^2, and B. Execution of the program statements continues again until we reach line 9 in which we call the procedure with parameter $2/2 = 1$, etc. Since the value of the first parameter is now 1, line 3 takes effect and appropriate values of other variables are computed. Keeping track of these values requires a large amount of storage space and a reasonably large amount of time. These are perhaps the major reasons that recursion is not used more in the non-mathematical programming community. Recursion certainly allows the writing of elegant, easy to understand programs. With the rapid decrease in computer prices and the increase in power and speed, programming time becomes a major consideration.

3. The program can be modified to work on inputs with any value of N, not just a power of 2. A simple way to do this is by adding new points a_k which are 0. This is not the most efficient way to proceed; however, the running time is still $O(N \log N)$.

4. This completes our discussion of arithmetic complexity where we are concerned with minimizing the number of arithmetic operations. We began with the simple problem of evaluation of a polynomial and progressed to the Fast Fourier Transform and the divide-and-conquer technique. A large amount of work is currently being done on arithmetic complexity; see **[42]** for a representative sample. See also **[22]** and **[38]** for an example of a simple problem for which the commonly taught algorithm can be improved—the problem of multiplying two m—digit numbers.

3. Sorting.

3.1. *Some sorting algorithms.* In the introduction we found the median of a set of numbers by sorting the set and then counting. In spite of the incredible calculations being performed on some modern computers, much of the time computers spend on programs is on sorting data. Estimates of the amount of time vary from 35 to 80%. Thus many sorting methods have been developed.

Among the more common methods are: insertion sort, Shell sort (also called diminishing increment sort), bubble sort, Hoare's quick sort, straight selection sort, quadratic selection sort, heap sort, binary tree sort, merge sort, radix sort. We will analyze two of these methods. For a more complete discussion of sorting, the reader is advised to read volume 3 of Knuth's book **[27]**. Our measure of complexity will now be the number of comparisons.

A von-Neumann machine performs a comparison between objects in locations A and B in roughly the following way. Move the contents of A to the accumulator of the cpu and compare the contents of the accumulator bit by bit with the contents of location B. The time for the comparison, including the movement of the contents of A to the accumulator, is greater than the times for most other operations used in sorting. Using the number of comparisons as the measure of complexity provides fairly good agreement between theoretical estimates and actual run-time performance in many cases.

Note that the length of the objects of "records" to be sorted need not be bounded by the maximum number of bits that can be stored in a memory location. In this case, we often simply change the addresses of records rather than moving the records from one memory location to another. More complications arise if the file of records to be sorted is too large to fit into memory at one time but must be stored in part in secondary storage such as a disk or tape. In any event, the number of comparisons provides a very good first approximation to running time in many situations; it is the only measure we consider in this paper.

The first method we consider is the bubble sort which is the one most commonly found in textbooks (see for example **[20]**, **[3]**, **[11]**). The object is to sort a set of numbers $R_1, \ldots, R_n$ in increasing order. We introduce a new number called BOUND. At each step of the algorithm, BOUND will represent an upper bound on the number of elements still to be sorted. The algorithm is given below.

```
1. BOUND = $N$
2. $T = 0$
3. FOR $j = 1$ TO BOUND-1 DO
          if $R_j > R_{j+1}$, interchange $R_j$ and $R_{j+1}$, set $T = j$
4. If $T = 0$, we are done
   ELSE
          BOUND = $T$
          GOTO Step 2.
```

We pass through the loop at step 3 (BOUND-1) times; at each step we make a comparison. On average, step 4 sends us back to step 2 $N/2$ times for an average and maximum run-time of $O(N^2)$. The minimum time is $O(N)$.

As an example of bubble sort consider the action of a bubble sort on {All mathematicians should learn about computers}. We wish to have the set sorted in alphabetical order. In order to do this we must place the words in an array $R(1)$ = All, $R(2)$ = mathematicians, . . . , $R(6)$ = computers. The array entries must then be translated into a binary representation. This translation is usually done automatically. The programmer indicates the number of characters in the longest expression expected (this is necessary in standard Pascal and FORTRAN, not in BASIC) and the translation is done automatically. The most common scheme is called ASCII. ASCII (pronounced Askey) stands for the American Standard Code for Information Interchange. For example, the ASCII codes for space, A, Z, a, and z are 32, 65, 90, 97 and 122, respectively. Thus a statement such as

All < mathematicians

follows from the fact that ASCII (A) < ASCII (m). If the first letters agree, then the computer will look at the next letter, etc.

Let us examine bubble sort on the set. Initially we have

1	2	3	4	5	6
All	mathematicians	should	learn	about	computers

Here BOUND = 6 and $T = 0$. Is "All" < "mathematician"? Yes. Is "mathematician" < "should"? Yes. Is "should" < "learn"? No. Interchange "learn" and "should", setting $T = 3$. Is "should" < "about"? No. Interchange "should" and "about", setting $T = 4$. Is "should" < "computers"? No. Interchange "should" and "computers", setting $T = 5$. Now set BOUND = 5 and GO TO step 3. At this point we have

1	2	3	4	5	6
All	mathematicians	learn	about	computers	should.

We continue the process. After the second "pass" through the loop we have

1	2	3	4	5	6
All	learn	about	computers	mathematicians	should

with BOUND = 4 and $T = 4$. Successive passes give

1	2	3	4	5	6
All	about	computers	learn	mathematicians	should

with BOUND $= 3$ and $T = 3$,

1	2	3	4	5	6
All	about	computers	learn	mathematicians	should

with BOUND $= 3$ and $T = 0$,

with the program terminating because no interchanges were made.The sort is called bubble sort because elements "bubble up" to their correct positions one at a time. Note also that "All" is the first word in the sorted list. This is because upper case letters have ASCII numbers between 65 and 90 while lower case letters have ASCII numbers between 97 and 122. It is reassuring to know that even nonsense sentences will begin with upper case letters.

A more elegant sorting method is the merge sort. The idea is very simple. Suppose we had two sets A and B of n numbers which were already sorted in increasing order. We could merge them together by comparing (say) the first element a of A with the elements of B until we find its proper place and then insert it. Continue this with $A - \{a\}$, etc. If we get to the end of B before exhausting A, the remaining elements of A simply wind up at the end of the now sorted set $A \cup B$. If the number of elements in each is n, the time needed for the sort is $O(n)$ since there are between n and $2n$ comparisons.

How do we relate the merging of two sorted sets to the sorting of a large unsorted set? Answer-keep breaking up the set until we get to subsets of one element and then merge these (already sorted) subsets. If we call the procedure for merging already sorted subsets MERGE and the general procedure MERGESORT, the algorithm for MERGESORT can be written recursively as

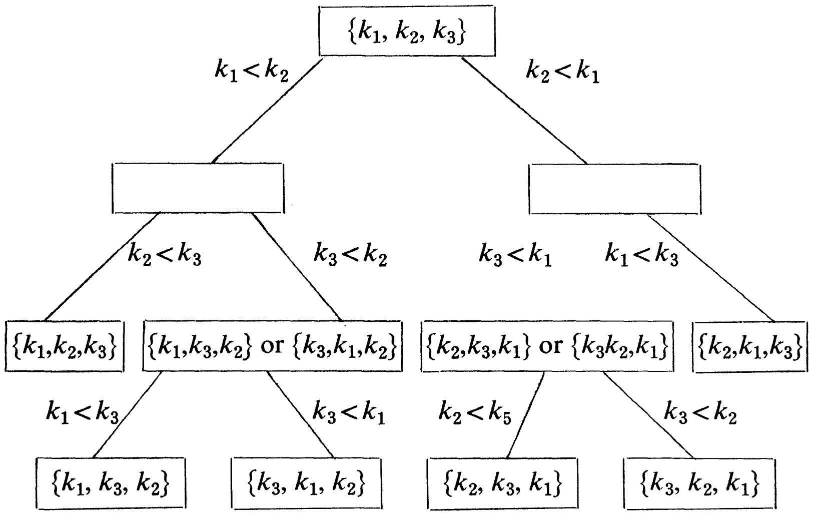
$$\text{MERGESORT}\,(X_1, \ldots, X_N) = \text{MERGE}(\text{MERGESORT}(X_1, \ldots, X_{[N/2]}), \text{MERGESORT}(X_{[N/2]+1}, \ldots, X_N)).$$

Each MERGE requires $O(N)$ and there are $\log_2 N$ recursive calls of this procedure, each one dividing N by 2 until we get to single element sets which are already sorted. This sorting method requires a running time of $O(N \log N)$, a considerable improvement over the $O(N^2)$ required for the bubble sort. The author has been told of a sort of approximately 5000 items which was speeded up from 5 hours to 30 seconds by replacing the $O(N^2)$ bubble sort with a $O(N \log N)$ merge sort.

3.2. *A lower bound on the number of comparisons.* It is natural to ask if the lower limit on the number of comparisons is $O(N \log N)$. The following well-known argument shows that this is the case, at least for any method based on comparisons.

A bit of terminology is in order. The items to be sorted by some sorting method are called keys. A key is generally associated with a unique record; think of Social Security numbers as keys and employment histories as records.

Suppose we wish to sort a set $X_1, \ldots, X_n$. We set up a tree to indicate the comparisons. The internal nodes of the tree are comparisons between keys. The leaves of the tree are permutations of the indices and represent the order we have if we choose a particular path. For $n = 3$, we have



There are three elements to be sorted using this comparison tree. The height of this tree is 3; there are 11 nodes and 6 leaves.

If this comparison tree is to represent a sort which works in all possible cases, then all $n!$ permutations of the keys must be present in the leaves. Hence the number of leaves is $\geqq n!$. Let $S(n)$ = minimum number of comparisons to sort all possible sets of n element using comparisons. The minimum number of comparisons $S(n)$ that will work on the arbitrary ordered set is the longest path from the root to a leaf; that is, $S(n)$ is the height of the tree.

Recall that a binary tree of height h has at most one node (the root) at level 0, at most 2 nodes at level 1, at most 4 nodes at level 2, etc. Hence a binary tree of height h (level at most h) has at most $2^{h+1} - 1$ nodes. Similarly, a binary tree of height h has at most 2^h leaves. Thus a tree of minimum height $S(n)$ which contains all $n!$ permutations of the keys as leaves has

$$2^{s(n)} \geqq n!$$

or

$$\begin{aligned} S(n) &\geqq \log_2 n! \\ &\sim \log_2(\sqrt{2\pi n}\,(n/e)^n) \\ &\geqq C(n \log n) \end{aligned}$$

by Stirling's formula.

REMARKS 1. There is a lot of recent work on parallel algorithms for searching and sorting. The reference [7] is a good starting point; however, the reader should keep in mind that new algorithms are being discovered almost monthly. To see the difficulty in designing a parallel sorting algorithm, write the numbers 1, 7, 13, 9, 24, 17, 18, 26, 11, 3, 55 on cards and try to find an optimal method of sorting if you and a friend are each given a subset of the cards.

2. While the order of magnitude of the number of comparisons is known, the choice of algorithm and its implementation on a particular machine with a particular storage device for a particular file is as much an art as it is a science.

4. Concluding remarks.

4.1. *Complexity in a model of computation.* We have used two measures of run-time complexity in this paper: the number of "essential m/d steps" for arithmetic algorithms and the number of comparisons for sorting algorithms. A more detailed study of complexity theory would begin with the work of Turning [39] on machine models. Many other models such as finite automata and various types of pushdown automata have been developed since then for special purposes (see [17], [19], [23], [28]). For examples of such ideas and their use in the design of compilers, see [2], [12], [13], [26].

Empirical evidence shows that most algorithms fall into one of two categories: run-time is bounded by a polynomial of low degree (eg [12], [29], [30], [36], [37]) or run-time that is not bounded by any polynomial (see [16] for a large collection). This has lead to a large amount of work on the topic of NP-completeness and the $P \neq NP$ conjecture. (See [6], [8], [16] and [23] for an introduction to this subject.)

4.2. *Other measures of complexity.* In this note we have been primarily concerned with the idea of run-time complexity, with particular emphasis on arithmetic complexity. We have only considered computers and models which performed computations sequentially. Parallel algorithms which assume that many comparisons and arithmetic operations can be performed at the same time were not considered. The survey article by S. Cook [7] has a good set of references for such algorithms.

Another topic not considered here is that of probabilistic algorithms. Perhaps the first discussion of such algorithms is the paper [**4**] where the goal was to factor polynomials over the field $GF(p)$. The algorithm has a probability $\geqq 1/2$ that it will find a correct factorization. The use of the algorithm is as follows. If the algorithm finds a factorization, then we are done. If not, repeat the algorithm. Suppose that we apply the algorithm n times without finding a factorization. Since the trials are independent, we eventually obtain the result that the polynomial cannot be factored, with a high probability of success. The article [**35**] describes a probabilistic algorithm for determining if an integer is prime.

Finally we mention the notion of the complexity of the program itself. Much of the emphasis in programming is on structured style. Languages such as Pascal encourage the programmer to write programs in "modules" which can be easily understood. These modules are much easier to test for correctness than a large unstructured program. This is a great advantage since the time when all programs were written in machine languages, driving programmers crazy bit by bit.

Acknowledgements. I would like to thank John Cherniavsky and John Sadowsky for introducing me to this subject. Many thanks also to Sam Lomonaco and Brooke Stephens for criticizing an earlier version of this paper. Special thanks to Mary Leach for many helpful suggestions on improving the organization of this manuscript.

REFERENCES

1. A. V. Aho, J. E. Hopcroft and J. D. Ullman, *The design and analysis of computer algorithms*, Addison Wesley, Reading, Mass., 1974.

2. A. V. Aho and J. D. Ullman, *Principles of Compiler design*, Addison-Wesley, Reading, Mass., 1977.

3. R. Bent and G. Sethares, FORTRAN *with problem solving: a structured approach*, Brooks/Cole, Monterrey, California, 1981.

4. E. Berlenkamp, *Factoring polynomials over large finite fields*, Math. Comp. **24** (1970), 713–735.

5. A. Cobham, *The intrinsic comptational difficulty of functions*, in Proc. International Congress for Logic Methodology and Philosophy of Science, Y. Bar-Hillel, ed., North Holland, Amsterdam, 1965.

6. S. Cook, *The complexity of theorem proving procedures*, Proc. Third ACM Symposium on Theory of Computing, 1971, 151–158.

7. ———, *Towards a complexity theory of synchronous parallel computation*, L'Enseignement Mathematique, XXVII (1981), 99–124.

8. ———, *An overview of computational complexity*, Comm. ACM **26** (1983), 401–408.

9. J. W. Cooley, P. A. Lewis and P. D. Welch, *History of the fast Fourier transform*, Proc. IEEE **55** (1967), 1675–1679.

10. J. M. Cooley and J. W. Tukey, *An algorithm for the machine calculation of complex Fourier series*, Math. Comp. **19** (1965), 297–301.

11. D. Cooper and M. Clancy, *Oh Pascal*, W. W. Norton, New York, 1982.

12. F. DeRemer and T. Pennello, *Efficient Computation of* LARL(1) *Look Ahead Sets*, ACM Trans. on Prog. Lang. and System **4** (1982), 615–649.

13. J. Earley, *An efficient context-free parsing algorithm*, Comm. ACM **13** (1970), 94–102.

14. J. Edmonds, *Paths, trees and flowers*, Can. J. Math. **3** (1965), 449–467.

15. R. Fourer, Technical Correspondence, Comm. ACM **26** (1983), 382–84.

16. M. Garey and D. Johnson, *Computers and Intractability: A guide to the theory of NP-completeness*, Freemen and Co., San Francisco, 1979.

17. M. Harrison, *Introduction to Formal Language Theory*, Addison-Wesley, Reading, Mass., 1978.

18. S. Hartmanis and R. Stearns, *On the computational complexity of algorithms*, Trans. AMS **117** (1965), 285–306.

19. J. E. Hopcroft and J. D. Ullman, *Introduction to automata theory, languages and computation*, Addison-Wesley, Reading, Mass., 1979.

20. E. Horowitz and S. Sanhi, *Fundamentals of computer algorithms*, Computer Science Press, Rockville, Md., 1978.

21. D. Johnson, Journal of Algorithms.

22. A. Karatsuba and Yo. Offman, *Multiplication of multidigit numbers on automata*, Doklady Akad. Nauk, SSSR. **145**, 2 (1962), 293–294, Translated in Soviet Physics Doklady. **77** (1963), 595–596.

23. R. M. Karp, *Reducibility among combinatorial problems*, in Complexity of computer computations, R. E. Miller and J. W. Thatcher, eds., Plenum Press, New York, 1972, 85–104.

24. L. G. Khachian, *A polynomial time algorithm for linear programming*, Doklady Akad Nauk SSSR **244** (1979), 1093–96. Translation: Soviet Math. Doklady 20, 191–194.

25. S. Kleene, *Representation of events by nerve nets*, in Automata studies, C. E. Shannon and J. McCarthy, eds., Princeton University Press, Princeton, New Jersey, 1956, 3–42.

26. D. Knuth, *On the translation of languages from left to right*, Information and Control **8** (1965), 605–639.

27. ———, *The art of computer programming*, Addison-Wesley, Reading, Mass., 2nd ed., 1981.

28. H. R. Lewis and C. H. Papadimitrious, *Elements of the theory of computation*, Prentice-Hall, Englewood Cliffs, New Jersey, 1981.

29. E. H. McCall, *Performance results of the simplex algorithm for a set of real-world linear programming problems*, Comm. ACM **25** (1982), 207–212.

30. ———, Technical Correspondence, Comm. ACM **26** (1983), 384–85.

31. C. Papadimitrious and K. Steiglitz, *Combinatorial optimization: algorithms and complexity*, Prentice Hall, Englewood Cliffs, New Jersey, 1982.

32. M. O. Rabin, *Complexity of computations*, Comm. ACM **20** (1977), 625–633.

33. A. Schonhage and V. Strassen, *V. Schnelle Multiplication grosser Zahlen*, Comput ing **7** (1971), 281–292.

34. C. Shannon, *The synthesis of two terminal switching circuits*, Bell Systems Technical Journal **28** (1949), 59–98.

35. R. Solovay and V. Strassen, *A fast Monte-Carlo test for primarlity*, SIAM J. Comp. **6** (1977), 84–85.

36. J. Schwartz and M. Sharir, *On the piano movers problem* II-*General Techniques of computing topological properties of real algebraic manifolds*, Adv. Appl. Math **4** (1983), 298–351.

37. ——— and ———, *On the piano movers problem* I. The case of a two dimensional body moving amidst polygonal barriers, Comm. Pure Appl. Math. **36** (1983), 345–398.

38. A. L. Toom, *The complexity of a scheme of functional elements realizing the multiplication of integers*, Doklady Akad. Nauk SSSR **150**, 3 (1963), 496–498. Translated in Soviet Math. Doklady **3** (1963), 714–716.

39. A. M. Turing, *On computable numbers with an introduction to the Entscheidungs problem*, Proc. London Math. Soc. (2) **42** (1936–7), 230–265. (Correction ibid **43** (1937), 544–546.)

40. M. Wand, *What is LISP?*, Amer. Math. Monthly **91** (1984), 32–42.

41. H. S. Wilf, *The disk with the college education*, Amer. Math. Monthly, **89** (1982).

42. S. Winograd, *Arithmetic Complexity of Computations*, CBMS-NSF Regional Conference Series **33**, SLAM, Philadelphia, 1980.

DEPARTMENT OF SYSTEMS AND COMPUTER SCIENCE, HOWARD UNIVERSITY, WASHINGTON, D.C. 20059