MATHEMATICAL ANALYSIS OF ALGORITHMS

BY

DONALD E. KNUTH

STAN-CS-71-206

MARCH, 1971

COMPUTER SCIENCE DEPARTMENT

School of Humanities and Sciences

STANFORD UNIVERSITY

ACCESSION for

NTIS	White Section	[✓]
DDC	Buff Section	[ ]
UNANNOUNCED		[ ]

JUSTIFICATION ....

....

BY ....

DISTRIBUTION/AVAILABILITY CODES

DIST.	AVAIL. and/or SPECIAL	
A		

Unclassified
Security Classification

DOCUMENT CONTROL DATA - R & D

(Security classification of title, body of abstract and indexing annotation must be entered when the overall report is classified)

1. ORIGINATING ACTIVITY *(Corporate author)* Stanford University	2a. REPORT SECURITY CLASSIFICATION Unclassified
	2b. GROUP
3. REPORT TITLE MATHEMATICAL ANALYSIS OF ALGORITHMS	
4. DESCRIPTIVE NOTES *(Type of report and inclusive dates)*	
5. AUTHOR(S) *(First name, middle initial, last name)* Donald E. Knuth	

6. REPORT DATE March 1971	7a. TOTAL NO. OF PAGES 26	7b. NO. OF REFS 14
8a. CONTRACT OR GRANT NO. b. PROJECT NO. ONR - N-00014-67-A-0112-0057 NR 044-402 c. NSF GJ-992 d.	9a. ORIGINATOR'S REPORT NUMBER(S) STAN-CS-71-206 9b. OTHER REPORT NO(S) *(Any other numbers that may be assigned this report)* None	

10. DISTRIBUTION STATEMENT

Distribution of this document is unlimited.

11. SUPPLEMENTARY NOTES	12. SPONSORING MILITARY ACTIVITY Office of Naval Research, Pasadena Branch National Science Foundation, Computer Science Dept.

13. ABSTRACT

This report consists of the texts of lectures presented to the International Congress of Mathematicians in 1970 and to the IFIP Congress in 1971. The lectures are essentially sales pitches intended to popularize work in algorithmic analysis, a field of study which involves numerous applications of discrete mathematics to computer science. Both lectures attempt to indicate the flavor of the general field by considering particular applications in detail. The "mathematical" lecture deals with the problem of calculating greatest common divisors, and includes a presentation of a new algorithm which lowers the asymptotic running time for gcd of n-bit integers from n^2 to $n^{1+\varepsilon}$. The "information processing" lecture deals with the problems of in situ permutation and selection of the t-th largest element, emphasizing techniques for analyzing particular algorithms which have appeared in the literature.

1a

DD FORM 1 NOV 65 1473 (PAGE 1)

S/N 0101-807-6801

Unclassified
Security Classification

Unclassified
Security Classification

14. KEY WORDS	LINK A		LINK B		LINK C	
	ROLE	WT	ROLE	WT	ROLE	WT
Analysis of Algorithms, Computational Complexity, Euclid's Algorithm, Continued Fractions, Greatest Common Divisor, In Situ Permutation, Selection of t-th Largest, Recurrence Relations.						

Unclassified
Security Classification

Mathematical Analysis of Algorithms

by Donald E. Knuth

Abstract: This report consists of the texts of lectures presented to the International Congress of Mathematicians in 1970 and to the IFIP Congress in 1971. The lectures are essentially sales pitches intended to popularize work in algorithmic analysis, a field of study which involves numerous applications of discrete mathematics to computer science. Both lectures attempt to indicate the flavor of the general field by considering particular applications in detail. The "mathematical" lecture deals with the problem of calculating greatest common divisors, and includes a presentation of a new algorithm which lowers the asymptotic running time for gcd of n-bit integers from n^2 to $n^{1+\varepsilon}$. The "information processing" lecture deals with the problems of in situ permutation and selection of the t-th largest element, emphasizing techniques for analyzing particular algorithms which have appeared in the literature.

This research was supported by the National Science Foundation under grant number GJ-992, and the Office of Naval Research under grant number N-00014-67-A-0112-0057 NR 044-402. Reproduction in whole or in part is permitted for any purpose of the United States Government.

THE ANALYSIS OF ALGORITHMS

The advent of high-speed computing machines, which are capable of carrying out algorithms so faithfully, has led to intensive studies of the properties of algorithms, opening up a fertile field for mathematical investigations. Every reasonable algorithm suggests interesting questions of a "pure mathematical" nature; and the answers to these questions sometimes lead to useful applications, thereby adding a little vigor to the subject without spoiling its beauty. The theory of queues, which analyzes a very special class of algorithms, indicates the potential richness of the theories which can be obtained when algorithms of all types are analyzed in depth.

The purpose of this paper is to illustrate some general principles of algorithmic analysis by considering an example which is interesting for both historical and mathematical reasons, the calculation of the greatest common divisor (gcd) of two integers by means of Euclid's algorithm. Euclid's procedure [2], which is one of the oldest nontrivial algorithms known, may be formulated as follows, given integers $U > V \geq 0$:

E1. If $V = 0$, stop; U is the answer.

E2. Let R be the remainder of U divided by V , so that $U = AV + R$, $0 \leq R < V$. Replace U by V , then replace V by R , and return to E1.

1. <u>"Local" analysis</u>. Analyses of algorithms are generally of two kinds, "local" and "global". A local analysis consists of taking one particular algorithm (like Euclid's) and studying the amount of work it does as a function of the inputs; a global analysis, on the other hand, considers an entire family of algorithms and investigates the "best possible" procedures in that class, from some point of view. In both types of analysis we can consider either the "worst case" of the algorithms, namely the work involved under least favorable choice of inputs, or the "average case", the expected amount of work under a given input distribution. More generally, we may be able to obtain the distribution of work given the distribution of inputs. "Work" may be measured in terms of the number of times each step of the algorithm is performed, or the amount of things which need to be remembered, etc.

The first local analysis of Euclid's algorithm was published in 1844 by G. Lamé [10], who showed that step E2 will never be performed more than five times the number of digits in the decimal representation of V. His analysis was based on the fact that the method is least efficient when U and V are consecutive Fibonacci numbers.

The <u>average</u> behavior of Euclid's algorithm is much more difficult to determine than the worst case, and it has been established only in recent years. J. D. Dixon proved [1] that, for all ε and $C > 0$, the probability that $|T(U,V)-(12\pi^{-2} \ln 2) \ln U| \geq (\ln U)^{\frac{1}{2}+\varepsilon}$ is $O((\ln N)^{-C})$, given that $1 \leq V \leq U \leq N$. His proof is based on careful refinements on Kuz'min's study of continued fractions [9], showing that partial quotients which are far apart in the sequence are nearly independent.

At about the same time, H. Heilbronn introduced a new approach [6] to the study of continued fractions and Euclid's algorithm. Let $T(U,V)$ be the number of times step E2 is performed, and let

$$T(V) = \lim_{N \to \infty} \frac{1}{N} \sum_{U=V+1}^{V+N} T(U,V) = \frac{1}{V} \sum_{U=V+1}^{2V} T(U,V)$$

be the average number of times when V is fixed. Heilbronn showed in effect that

$$nT(n) = \lfloor 3n/2 \rfloor + 2 \sum \lceil (\frac{n}{y+t} - t') \frac{1}{y} \rceil$$

where $\lfloor x \rfloor$ is the greatest integer $\leq x$, $\lceil x \rceil$ is the least integer $\geq x$, and the sum is over all positive integers y, t, t' such that $\gcd(t,y) = 1$, $t \leq y$, $t' \leq y$, $tt' \equiv n$ (modulo y) . Evaluating this sum, he essentially found that $T(n) = (12\pi^{-2} \ln 2) \ln n + O(\sigma_{-1}(n)^2)$. Indeed, somewhat more seems to be true, although proof is still lacking; there is extensive empirical evidence [8, pp. 330-333] that $(\sum_{1 \leq k \leq V,\ \gcd(k,V)=1} T(V+k,V))/\varphi(V) = (12\pi^{-2} \ln 2) \ln V + 1.47 + O(1)$ as $V \to \infty$.

2. <u>"Global" analysis</u>. Is Euclid's algorithm the "best" way to calculate greatest common divisors? Analyses of other gcd algorithms (cf. [8]) show that, under certain conditions, Euclid's method is inferior; and the average behavior of an interesting new algorithm discovered by V. C. Harris [4] is still unknown.

In searching for a "best" method, one way to measure the work is to consider the amount of time taken to perform the algorithm with pencil and paper, or with a conventional computer. Various abstract automata have been proposed by which the latter notions can be made precise (cf. [5, 7]). When we apply such models to Euclid's algorithm, it is not difficult to see [8, p. 526] that the amount of work is essentially proportional to the square of the number of digits in U , for both the average case and the worst case, analogous to the familiar method of long division. On the other hand, extremely fast methods of multiplication and division have recently been discovered; A. Schönhage and V. Strassen have proved [13] that an m-digit number can be multiplied by an n-digit number in only $O(n(\log m)(\log \log m))$ units of time, when $n \geq m > 1$. It is therefore natural to ask whether the gcd of two n-digit numbers can be calculated in less than $O(n^2)$ steps. Section 3 of this paper shows that this is indeed possible, in $O(n^{1+\varepsilon})$ steps for all $\varepsilon > 0$, by suitably arranging the calculations of Euclid's algorithm. Obviously at least n steps are necessary in any event (we must look at the inputs), so this result provides some idea of the asymptotic complexity of gcd computation.

3. <u>High-speed gcd calculation with large numbers</u>. If step E2 is performed t times, let $A_1, \ldots, A_t$ be the partial quotients obtained. It is well known that $U = Q_t(A_1, \ldots, A_t)D$, $V = Q_{t-1}(A_2, \ldots, A_t)D$, where $D = \gcd(U,V)$ and Q_t is the continuant polynomial defined by $Q_{-1} = 0$, $Q_0 = 1$, $Q_{t+1}(x_0, x_1, \ldots, x_t) = x_0 Q_t(x_1, \ldots, x_t) + Q_{t-1}(x_2, \ldots, x_t)$. We shall call $[A_1, \ldots, A_t, D]$ the <u>Euclidean representation</u> of U and V . After k iterations of step E2 we have $U = U_k = Q_{t-k}(A_{k+1}, \ldots, A_t)D$, $V = V_k = Q_{t-k-1}(A_{k+2}, \ldots, A_t)D$. Euler [3] observed that $Q_t(x_1, \ldots, x_t)$ is the set of all terms obtainable by starting with $x_1 \ldots x_t$ and striking out pairs $x_i x_{i+1}$ zero or more times. From this remark, it follows immediately that

$$Q_{s+t}(x_1,\ldots,x_{s+t}) = Q_s(x_1,\ldots,x_s)Q_t(x_{s+1},\ldots,x_{s+t}) + Q_{s-1}(x_1,\ldots,x_{s-1})Q_{t-1}(x_{s+2},\ldots,x_{s+t}) \;, \qquad (*)$$

an identity which forms the basis of Heilbronn's work cited above; it was used on several occasions by Sylvester [14] and given in more general form by Perron [12, p. 14-15].

For convenience we shall write nonnegative integers N in binary notation, using $\ell N \equiv \lceil \log_2(N+1) \rceil$ binary digits. It is easy to prove that $\ell Q_t(A_1,\ldots,A_t) \leq \ell A_1 + \ldots + \ell A_t + 1$, and Lamé's theorem implies that $\ell A_1 + \ldots + \ell A_t \leq \ell Q_t(A_1,\ldots,A_t) + t = O(\log U)$ in Euclid's algorithm; hence (except for a constant factor) it takes essentially as much space to write down the Euclidean representation $[A_1,\ldots,A_t,D]$ as it does to write U and V themselves in binary form. We shall show that it is possible to convert rapidly between these two representations of U and V.

Theorem 1. Let $S(n) = n(\log n)(\log\log n)$ and $n = \ell A_1 + \ldots + \ell A_t$. There is an algorithm which, given the binary representations of $A_1,\ldots,A_t$, computes the binary representation of $Q_t(A_1,\ldots,A_t)$ in $O(S(n)(\log t))$ steps.

Proof. Consider four continuants associated with $(A_1,\ldots,A_t)$, namely $Q = Q_t(A_1,\ldots,A_t)$, $Q^{\cdot} = Q_{t-1}(A_1,\ldots,A_{t-1})$, ${}^{\cdot}Q = Q_{t-1}(A_2,\ldots,A_t)$, and ${}^{\cdot}Q^{\cdot} = Q_{t-2}(A_2,\ldots,A_{t-1})$. The four continuants associated with $(0,A_1,\ldots,A_t)$ are the same, in another order, so we add zeroes if necessary until t is a power of 2. Now let $L, L^{\cdot}, {}^{\cdot}L, {}^{\cdot}L^{\cdot}$ and $R, R^{\cdot}, {}^{\cdot}R, {}^{\cdot}R^{\cdot}$ be the continuants associated with $A_1,\ldots,A_t$ and $A_{t+1},\ldots,A_{2t}$ respectively. By (*), $Q = LR + L^{\cdot}\,{}^{\cdot}R$, $Q^{\cdot} = LR^{\cdot} + L^{\cdot}\,{}^{\cdot}R$, ${}^{\cdot}Q = {}^{\cdot}LR + {}^{\cdot}L^{\cdot}\,{}^{\cdot}R$, ${}^{\cdot}Q^{\cdot} = {}^{\cdot}LR^{\cdot} + {}^{\cdot}L^{\cdot}\,{}^{\cdot}R^{\cdot}$. Choosing C so that we can evaluate the L's in $CS(\ell A_1 + \ldots + \ell A_t)k$ steps, the R's in $CS(\ell A_1 + \ldots + \ell A_{2t})$ further steps by the Schönhage-Strassen algorithm, we can evaluate the Q's in at most $CS(\ell A_1 + \ldots + \ell A_{2t})(k+1)$ steps. ∎

Let $U = 2^m U' + U''$, $V = 2^m V' + V''$, where $0 \leq U'', V'' < 2^m$. D. H. Lehmer [11] has suggested that the partial quotients for (U,V) be found by first obtaining some of those for U' and V', stopping at A_s where s is maximal such that $(U'+1,V')$ and $(U',V'+1)$ have $A_1,\ldots,A_s$ in common. Then $A_1,\ldots,A_s$ are partial quotients for (U,V) also. We shall call $(A_1,\ldots,A_s$ the "Lehmer quotients" for (U',V'). The example $(U',V') = (2^m, 2^{m-1})$ shows that Lehmer quotients might not amount to anything, but we can prove that four additional Euclidean iterations will always give a useful reduction.

Lemma 1. Let $U = 2^m U' + U'' \geq V = 2^m V' + V''$, where $0 \leq U'', V'' < 2^m$. Let $[A_1,\ldots,A_t,D]$ be the Euclidean representation of (U,V), and let $(A_1,\ldots,A_s)$ be the Lehmer quotients for (U',V'), where $t \geq s+4$. Then $U_{s+4} < U/\sqrt{U'}$.

Proof. Let $P_k = Q_{k-1}(A_2,\ldots,A_k)$, $Q_k = Q_k(A_1,\ldots,A_k)$, and let $\theta = V/U$. The well-known pattern of convergence of P_k/Q_k to θ, schematically

$$\frac{P_k}{Q_k} < \frac{P_k + P_{k+1}}{Q_k + Q_{k+1}} < \frac{P_k + 2P_{k+1}}{Q_k + 2Q_{k+1}} < \ldots < \frac{P_{k+2}}{Q_{k+2}} < \theta < \frac{P_{k+2} + P_{k+1}}{Q_{k+2} + Q_{k+1}} < \frac{P_{k+1}}{Q_{k+1}}$$

when k is even, shows that if θ and θ' are two real numbers whose continued fractions differ first at $A_{s+1} \neq A'_{s+1}$, either P_{s+1}/Q_{s+1} or P_{s+2}/Q_{s+2} lies between θ and θ'. Hence $\frac{1}{2} Q^2_{s+4} \geq Q_{s+2}(Q_{s+3} + Q_{s+2}) > 1/|\theta - P_{s+2}/Q_{s+2}| \geq 1/|(V'+1)/U' - V'/(U'+1)| > \frac{1}{2} U'$, using the well-known relation $|\theta - P_k/Q_k| > 1/Q_k(Q_{k+1} + Q_k)$. And by (*), $Q_{t-s-4}(A_{s+5},\ldots,A_t) > U/Q_{s+4}$. ∎

Lemma 2. *There is an algorithm which, given* $U \geq V \geq 0$ *with* $\ell U = n$, *finds all the Lehmer quotients for* (U,V) *in at most* $O(S(n)(\log n)^3)$ *steps.*

Proof. For large n the algorithm first applies itself recursively to the leading $\frac{1}{2} n$ binary digits of U and V, finding r partial quotients; then it computes $U_r = (-1)^r(Q_{r-2}(A_2,\ldots,A_{r-1})U - Q_{r-1}(A_1,\ldots,A_{r-1})V)$, $V_r = (-1)^r(Q_r(A_1,\ldots,A_r)V - Q_{r-1}(A_2,\ldots,A_r)U)$ in $O(S(n) \log(n))$ steps by the method of Theorem 1. We can find A_{r+1} in $O(S(n) \log(n))$ further steps (see [8, p. 275]), so by Lemma 1 the algorithm performs a bounded number of Euclidean iterations until reaching U_{r+k} with at most $\frac{3}{4} n$ digits. Now the same process is repeated on the $\frac{1}{2} n$ leading digits of U_{r+k}, V_{r+k}; after a bounded number of further Euclidean iterations, we have reduced U to less than $\frac{1}{2} n$ digits, and we have found quotients $A_1,\ldots,A_p$, where $p \geq s$ (since the proof of Lemma 1 can be readily modified to show that $Q_s < \sqrt{U'}$). Finally the value of s is located in approximately $\log_2 p = O(\log n)$ iterations, using the well known "binary search" bisection technique; each iteration tests some k to see whether or not $k < s$ or $k > s$. Such a test can rely on the fact that P_k/Q_k and P_{k+1}/Q_{k+1} are both "good" when $k \leq s$, while they are not both "good" when $k \geq s+2$, where P_k/Q_k is called good when it is $< V_k/(U_k+1)$, for k even, or $> (V_k+1)/U_k$, for k odd. The running time $L(n)$ of this algorithm as a whole now satisfies $L(n) \leq 2L(\frac{1}{2} n) + O(S(n)(\log n)^2)$. ∎

Theorem 2. *There is an algorithm which, given* $2^n > U > V \geq 0$, *determines the Euclidean representation* $[A_1,\ldots,A_t,D]$ *in* $O(n(\log n)^5(\log \log n))$ *steps as* $n \to \infty$.

Proof. Begin as in Lemma 2 to reduce n to $\frac{3}{4} n$ in $L(\frac{1}{2} n) + O(S(n) \log n)$ steps, then apply the same method until $V_t = 0$. The running time $G(n)$ of this algorithm satisfies the recurrence

$$G(n) = G(\tfrac{3}{4} n) + O(S(n)(\log n)^3) = G(\tfrac{9}{16} n) + O(S(n)(\log n)^3) + O(S(\tfrac{3}{4} n)(\log n)^3) = \ldots = O(S(n)(\log n)^4).$$

∎

In particular, we can find the gcd of n-digit numbers in $n^{1+\varepsilon}$ steps, as $n \to \infty$, for all $\varepsilon > 0$. The method we have used is rather complicated, but no simpler one is apparent to the author.[*] In general, the idea of reducing n to αn for $\alpha < 1$ often leads to asymptotically efficient algorithms.

[*] Note added in proof: A similar, somewhat simpler construction was found by A. Schönhage shortly after he received a preliminary copy of this paper; his improved construction takes only $O(n(\log n)^2(\log \log n))$ steps.

References

[1] John Douglas Dixon, AMS Notices 16, (October, 1969), 958.

[2] Euclid, Elements (c. 300 B.C.), Book 7, Propositions 1 and 2.

[3] Leonhard Euler, Introductio in Analysin Infinitorum (Lausanne: 1748), Section 359.

[4] Vincent Crockett Harris, Fibonacci Quarterly 8 (1970), 102-103.

[5] Juris Hartmanis and Richard Edwin Stearns, Transactions of the Amer. Math. Soc. 117 (1965), 285-306.

[6] Hans Heilbronn, Abhandlungen aus Zahlentheorie und Analysis zur Erinnerung an Edmund Landau (Berlin: VEB Deutcher Verlag der Wissenschaften, 1968), 89-96.

[7] Donald Ervin Knuth, Fundamental Algorithms, The Art of Computer Programming 1 (Addison-Wesley, 1968), pp. 462-463.

[8] Donald Ervin Knuth, Seminumerical Algorithms, The Art of Computer Programming 2 (Addison-Wesley, 1969).

[9] Rodion Osievich Kuz′min, Atti del Congresso internazionale dei matematici 6 (Bologna, 1928), 83-89.

[10] Gabriel Lamé, Comptes Rendus, Acad. Sci. Paris 19 (1844), 867-870.

[11] Derrick Henry Lehmer, American Math. Monthly 45 (1938), 227-233.

[12] Oskar Perron, Die Lehre von den Kettenbrüchen (Leipzig: 1913).

[13] Arnold Schönhage and Volker Strassen, "Schnelle multiplication grosser Zahlen", Universität Konstanz (1970), submitted for publication.

[14] James Joseph Sylvester, Philosophical Magazine 6 (1853), 297-299.

Mathematical Analysis of Algorithms

In this paper I shall try to illustrate the flavor of some current work in algorithmic analysis, by making rather detailed analyses of two algorithms. Since I was asked to be "mathematical", I have chosen some examples which are interesting primarily from a theoretical standpoint. The procedures I shall discuss (namely, in-situ permutation and selecting the k-th largest of n elements) are not among the ten most important algorithms in the world, but they are not completely useless and their analysis does involve several important concepts. Furthermore they are sufficiently unimportant that comparatively few people have studied them closely, hence I am able to say a few new things about them at this time.

The general field of algorithmic analysis is an interesting and potentially important area of mathematics and computer science that is undergoing rapid development. The central goal in such studies is to make quantitative assessments of the "goodness" of various algorithms. Two general kinds of problems are usually treated:

Type A. Analysis of a particular algorithm. We investigate important characteristics of some algorithm, usually a frequency analysis (how many times each part of the algorithm is likely to be executed), or a storage analysis (how much memory it is likely to need). For example, it is possible to predict the execution time of various algorithms for sorting numbers into order.

Type B. Analysis of a class of algorithms. We investigate the entire family of algorithms for solving a particular problem, and attempt to identify one that is "best possible". Or we place bounds on the computational complexity of the algorithms in the class. For example, it is possible to estimate the minimum number $S(n)$ of comparisons necessary to sort n numbers by repeated comparison.

Type A analyses have been used since the earliest days of computer programming; each program in Goldstine and von Neumann's classic memoir [7] on "Planning and Coding Problems for an Electronic Computing Instrument" is accompanied by a careful estimate of the "durations" of each step and of the total program duration. Such analyses make it possible to compare different algorithms for the same problem.

Type B analyses were not undertaken until somewhat later, although certain of the problems had been studied for many years as parts of "recreational mathematics". Hugo Steinhaus analyzed the sorting function $S(n)$, in connection with a weighing problem [14]; and the question of computing x^n with fewest multiplications was first raised by Arnold Scholz in 1937 [13]. Perhaps the first true study of computational complexity was the 1956 thesis of H. B. Demuth [3], who defined three simple classes of automata and studied how rapidly such automata are able to sort n numbers, using any conceivable algorithm.

It may seem that Type B analyses are far superior to Type A, since they handle infinitely many algorithms at once; instead of analyzing each algorithm that is invented, it is obviously better to prove once and for all that a particular algorithm is the "best possible". But this is only true to a limited extent, since Type B analyses are extremely technology-dependent; very slight changes in the definition of "best possible" can significantly affect which algorithm is best. For example, x^{31} cannot be calculated in fewer than 9 multiplications, but it can be done with only 6 arithmetic operations if division is allowed.

In fact the first result in Demuth's pioneering work on computational complexity was that "bubble sorting" was the optimum sorting method for a certain class of automata. Unfortunately, Type A analyses show that bubble sorting is almost the _worst_ possible way to sort, of all known methods, in spite of the fact that it is optimum from one particular standpoint.

There are two main reasons that Type B analyses do not supersede Type A analyses. First, it is generally necessary to formulate a rather simple model of the complexity, abstracting what seem to be the most relevant aspects of the class of algorithms considered, in order to make any progress at all on a Type B analysis. These simplified models are often sufficiently unrealistic that they lead to impractical algorithms. Secondly, even with simple models of complexity, the Type B analyses usually are considerably difficult, and comparatively few problems have been solved. Even the problem of computing x^n with fewest multiplications is far from solved (see [10, vol. 2, Section 4.6.3]), and the exact value of $S(n)$ is known only for $n \leq 12$ and $n = 20, 21$ (see [10, vol. 3, Section 5.3.1]). The sorting method of Ford and Johnson [6] uses fewer comparisons than any other known sorting technique, yet it is hardly ever useful in practice since it requires a rather unwieldy program. Comparison counting is not a good enough way to rate a sorting algorithm.

Thus I believe that computer scientists might well look on research in computational complexity as mathematicians traditionally view number theory: it is an interesting way to sharpen our tools, for the more routine problems we face from day to day. Although Type B analyses are extremely interesting, they do not deserve all the glory; Type A analyses are probably even more important in practice, since they can be designed to measure all of the relevant factors about the performance of an algorithm, and they are not quite as sensitive to changes in technology.

Fortunately, Type A analyses are stimulating intellectual challenges in their own right; nearly every algorithm that isn't extremely complicated leads to interesting mathematical questions. But of course we don't need to analyze every algorithm that is invented, and we can't hope to have a precise theoretical analysis of any really big programs.

In situ permutation. As our first example, let us consider the problem of replacing $(x_1,x_2,\ldots,x_n)$ by $(x_{p(1)},x_{p(2)},\ldots,x_{p(n)})$ where p is a permutation of $\{1,2,\ldots,n\}$. The algorithm is supposed to permute the x's in place, using only a finite amount of auxiliary memory. The function p is one of the inputs to the algorithm, we can compute $p(k)$ for any k but we cannot assign a new value to $p(k)$ as the algorithm proceeds. For example, p might be the function corresponding to transposition of a rectangular matrix, or to the unscrambling of a finite Fourier transform.

If $(p(1),p(2),\ldots,p(n))$ were stored in a read/write memory, or if we were allowed to manipulate n extra "tag" bits specifying how much of the permutation has been carried out at any time, there would be simple ways to design such an algorithm whose running time is essentially proportional to n. But we are not allowed to change p dynamically, nor are we allowed n bits of extra memory. Thus there seem to be comparatively few solutions to the problem.

The desired rearrangement of $(x_1, x_2, \ldots, x_n)$ is most naturally done by following the cycle structure of p (cf. [10, vol. 1, p. 161]). Let us say that j is a "cycle leader" if $j \leq p(j)$, $j \leq p(p(j))$, $j \leq p(p(p(j)))$, etc.; each cycle of the permutation has a unique leader, and so the following procedure (cf. Boothroyd [2], MacLeod [12]) carries out the desired permutation by doing each cycle when its leader is detected:

```
for $j$:=1 step 1 until $n$ do                                         1
   begin comment the permutation has been carried out                  n
      over all cycles whose leader is less than $j$;                   n
   $k$:=$p(j)$;                                                        n
   while $k > j$ do                                                    n+a
      $k$:=$p(k)$;                                                     a
   if $k = j$ then                                                     n
      begin comment $j$ is a cycle leader;                             b
      $y$:=$x[j]$;  $\ell$:=$p(k)$;                                    b
      while $\ell \neq j$ do                                           b+c
         begin $x[k]$:=$x[\ell]$; $k$:=$\ell$; $\ell$:=$p(k)$ end;     c
      $x[k]$:=$y$;                                                     b
      end permutation on cycle;                                        b
   end loop on $j$.                                                    n
```

The first and most basic part of the analysis of any algorithm is of course to prove that the algorithm works. The comments in this program essentially provide the key inductive assertions which will lead to such a proof. On the other hand, the program seems to be beyond the present range of "automatic program verification" techniques, and to go a step further to "automatic frequency analysis" is almost unthinkable.

Let us now do a frequency analysis of the above program, counting how often each statement is executed and each condition is tested. There are 9 statements, and 3 conditions, but we don't have to solve 12 separate problems because there are obvious relations between the frequencies. "Kirchhoff's law", which says that the number of times we get to a place in the program is the number of times we leave it, makes it possible to reduce the 12 individual frequencies to only 4, namely n, a, b, and c, as shown in the column to the right of the program. Kirchhoff's law is especially easy to apply in this case, since there are no go to statements; for example, we must test the condition " $k > j$ " in line 5 exactly $n+a$ times, if we execute line 4 n times and line 6 a times.

The next step in a frequency analysis is to interpret the remaining unknowns in terms of characteristics of the data. Obviously n, the number of times we do line 4, is the number of elements in the vector x. And b is the number of cycles in the permutation p. Furthermore we can see that each element of x is assigned a new value exactly once, either on line 11 or line 12, hence $c+b = n$ (a relation which cannot be deduced solely from Kirchhoff's law). This leaves only one variable, a, to be interpreted; it is somewhat more complicated, the sum of "distances" from j to the first element of $p(j), p(p(j))$, etc. that is $\leq j$.

To complete the analysis we should explore the behavior of these quantities a and b. It is customary to start by making a "worst case" analysis, which leads to an upper bound on the program's running time.

If $(p(1),p(2),\ldots,p(n)) = (2,\ldots,n,1)$, we have $a = (n-1)+(n-2)+\ldots+0 = \frac{1}{2}(n^2-n)$, which is surely the worst case for a .
The same choice of p makes $b = 1$, which is the best case for b . If $(p(1),p(2),\ldots,p(n)) = (1,2,\ldots,n)$, we get the worst case for b (and the best for a).

A more interesting problem arises when we try to consider the average case. First we must decide what is meant by the average case; this is often the chief stumbling block in making a Type A analysis, since it is not always easy to specify "typical" input distributions. For the problem at hand we may say that each of the $n!$ permutations p is equally likely.

A special technique is often useful when the cycle properties of permuations are being considered (cf. Foata [5], Knuth [10, vol. 1, Sec. 1.3.3; vol. 3, Sec. 5.1.2]), since it changes cycle properties into ordering properties. Consider for example the permutation $(p(1),\ldots,p(9)) = (8,2,7,1,6,9,3,4,5)$; in cycle form this is $(184)(569)(2)(73)$. The cycle form can be written in exactly one way such that

a) the leader comes first in each cycle;

b) the leaders of different cycles are in decreasing order from left to right.

In our example this canonical representation is $(569)(37)(2)(184)$. In canonical form the parentheses are redundant, since ")(" occurs just before each number which is smaller than all of its predecessors. Thus we obtain a one-to-one mapping of permutations onto permutations, such that cycle properties are mapped into ordering properties. In our example, $(8,2,7,1,6,9,3,4,5)$ maps into $(5,6,9,3,7,2,1,8,4)$.

Let $(p(1),p(2),\ldots,p(n))$ map into the permutation $(q(1),q(2),\ldots,q(n))$. It is easy to reinterpret the quantity b in terms of this transformation; it is the number of cycles in p, so it is the number of "left-to-right minima" in q, namely the number of indices j such that $q(j) = \min\{q(i) \mid 1 \le i \le j\}$. This quantity has been analyzed in detail in [10, vol. 1, section 1.2.10], where it is shown that the number of permutations with k left-to-right minima is $\left[{n \atop k}\right]$, a Stirling number of the first kind. The average value of b is shown there to be H_n, and the variance $H_n - H_n^{(2)}$, where

$$H_n = 1 + \frac{1}{2} + \ldots + \frac{1}{n}$$

$$H_n^{(2)} = 1 + \frac{1}{4} + \ldots + \frac{1}{n^2}$$

are harmonic numbers of degrees 1 and 2.

We can also analyze the quantity a, although the problem is somewhat deeper. When the loop variable j in the algorithm takes on the value $q(i)$, note that k will take on the successive values $q(i+1), q(i+2), \ldots$ because of the way we obtained q from p; we continue until reaching a value $q(i+r) < q(i)$. There is an exception to this rule, namely if k is set equal to the leader of the cycle: then either $i+r > n$ or $q(i+r)$ is the leader of the <u>next</u> cycle; in the latter case, again $q(i+r) < q(i)$.

Consequently we can represent a in the following way. Let y_{ij} be functions of q defined for all $1 \le i < j \le n$, where

$$y_{ij} = \begin{cases} 1\ , & \text{if } q(i) < q(k) \text{ for } i < k \le j\ ; \\ 0\ , & \text{otherwise.} \end{cases}$$

Then

$$a = \sum_{1 \le i < j \le n} y_{ij} \quad ;$$

indeed, for fixed i , $\sum_{i < j \le n} y_{ij}$ is the number of times line 6 of the program is performed when the loop variable $j = q(i)$.

For example, if $(p(1),\ldots,p(9)) = (8,2,7,1,6,9,3,4,5)$, we have $(q(1),\ldots,q(9)) = (5,6,9,3,7,2,1,8,4)$; hence $y_{12} = y_{13} = y_{23} = y_{45} = y_{78} = y_{79} = 1$, and all other y's are zero. Line 6 is performed $(2,1,0,1,0,0,2,0,0)$ times when $j = (5,6,9,3,7,2,1,8,4)$ respectively.

Let $\bar{y}_{ij}$ be the average value of y_{ij} , as $(q(1),\ldots,q(n))$ ranges over all permutations. This is simply the number of permutations with $y_{ij} = 1$, divided by $n!$, so it is the probability that $q(i) = \min\{q(k) \mid i \le k \le j\}$, namely $1/(j-i+1)$. It follows that $\bar{a}$, the average value of a , is given by

$$\bar{a} = \sum_{1 \le i < j \le n} \overline{y_{ij}} = \sum_{1 \le i < j \le n} \frac{1}{j-i+1} = \sum_{2 \le r \le n} \frac{n+1-r}{r} \quad ,$$

where we have replaced $j-i+1$ by a new variable r which occurs $n+1-r$ times in the original sum. Hence

$$\bar{a} = (n+1) \sum_{2 \le r \le n} \frac{1}{r} - \sum_{2 \le r \le n} 1 = (n+1)(H_n - 1) - (n-1) = (n+1)H_n - 2n \ .$$

The variance of a can be calculated too; the derivation is instructive but quite complicated, so the details will only be summarized here. We need the average value of

$$\Bigl(\sum_{1\le i<j\le n} y_{ij}\Bigr)^2 = \sum_{1\le i<j\le n} y_{ij}^2 + \sum_{\substack{1\le i<j\le n\\ 1\le k<\ell\le n\\ (i,j)\ne(k,\ell)}} y_{ij}y_{k\ell}$$

$$= \sum_{1\le i<j\le n} y_{ij} + 2\sum_{1\le i<j<k<\ell\le n} (y_{ij}y_{k\ell}+y_{ik}y_{j\ell}+y_{i\ell}y_{jk})$$

$$+ 2\sum_{1\le i<j<k\le n} (y_{ij}y_{jk}+y_{ik}y_{jk}+y_{ij}y_{ik})$$

$$= \bar{a} + 2(A+B+C+D+E+F)$$

where $A,\ldots,F$ represent the sums $\sum y_{ij}y_{k\ell},\ldots,\sum y_{ij}y_{ik}$. When $i<j<k<\ell$ are fixed, it is not difficult to prove that the average value of $y_{ij}y_{k\ell}$ is $1/(j-i+1)(\ell-k+1)$, of $y_{ik}y_{j\ell}$ is $1/(\ell-i+1)(\ell-j+1)$, of $y_{i\ell}y_{jk}$ is $1/(\ell-i+1)(k-j+1)$, of $y_{ij}y_{jk}=y_{ij}y_{jk}$ is $1/(k-i+1)(k-j+1)$, and of $y_{ij}y_{ik}=y_{ik}$ is $1/(k-i+1)$. This leaves us with several triple and quadruple summations to perform; it is not difficult to carry out a few of the sums, reducing them to

$$B = \tbinom{n}{2} - 2Z\ ,\quad C = Y - Z - 2\tbinom{n}{2} + 3X\ ,$$

$$D = E = Z - X\ ,\quad F = \tbinom{n}{2} - 2X\ ,$$

where

$$X = \sum_{1\le i<j\le n} \frac{1}{j-i+1}\ ,\quad Y = \sum_{1\le i<j\le n} H_{j-i}\ ,\quad Z = \sum_{1\le i<j\le n} \frac{1}{j-i+1}H_{j-i}\,.$$

We have already summed X by replacing $j-i+1$ by r, and the same device works for Y and Z; after applying well-known formulas for dealing with harmonic numbers (cf. [10, vol. 1, section 1.2.7]), we therefore obtain

$$X = (n+1)H_n - 2n \ , \quad Y = \tfrac{1}{2}(n^2+n)H_n - \tfrac{3}{4}n^2 - \tfrac{1}{4}n \ , \quad Z = \tfrac{1}{2}(n+1)(H_n^2 - H_n^{(2)}) - nH_n + n \ .$$

This determines B , C , D , E and F . The quantity A is harder to calculate; we have

$$A = \sum_{1 \le i < j < k < \ell \le n} \frac{1}{(j-i+1)(\ell-k+1)} = \sum_{\substack{r \ge 2 \\ s \ge 2 \\ r+s \le n}} \frac{1}{rs} \binom{n-r-s+2}{2}$$

$$= \sum_{\substack{2 \le r \le t-2 \\ 4 \le t \le n}} \frac{1}{t}\left(\frac{1}{r} + \frac{1}{t-r}\right)\binom{n-t+2}{2} = 2 \sum_{\substack{2 \le r \le t-2 \\ 4 \le t \le n}} \frac{1}{rt}\binom{n-t+2}{2}$$

$$= \sum_{\substack{2 \le r \le t-2 \\ 4 \le t \le n}} \frac{1}{rt}\left((n+2)(n+1) - t(2n+3) + t^2\right)$$

$$= (n+2)(n+1)U - (2n+3)V + W$$

by letting $r = j-i-1$, $s = \ell-k+1$, $t = r+s$. Then

$$U = \frac{1}{2}(H_n - 1)^2 - \frac{1}{2}H_n^{(2)} + \frac{1}{n} \ ,$$

$$V = (n-1)H_{n-2} - 2n + 4 \ ,$$

$$W = \frac{1}{2}\left((n^2+n-2)(H_{n-2}-1) - \tfrac{1}{2}(n-1)(n-2) + 1 - 3(n-3)\right) \ .$$

Putting the whole mess together, and subtracting $\bar{a}^2$ from the average of a^2, gives the exact value of the variance,

$$\sigma^2 = 2n^2 - (n+1)^2 H_n^{(2)} - (n+1)H_n + 4n \quad .$$

(This is a calculation that should have been done on a computer.)

Taking asymptotic values, we can summarize the statistics as follows:

$$a = (\text{min } 0,\ \text{ave } n \ln n + \underline{O}(n),\ \text{max } \tfrac{1}{2}(n^2-n),\ \text{dev } \sqrt{2-\pi^2/6}\ n + \underline{O}(\log n)) \ ;$$

$$b = (\text{min } 1,\ \text{ave } \ln n + \underline{O}(1),\ \text{max } n,\ \text{dev } \sqrt{\ln n} + \underline{O}(1)) \quad .$$

The total running time of the algorithm is therefore of order $n \log n$ on the average, although the worst case is order n^2; the comparatively small standard deviation indicates that worst-case behavior is very rare.

This completes our frequency analysis of the above algorithm. What information have we learned? We have found that the algorithm almost always takes about $n \log n$ steps, and that lines 5 and 6 are the "inner loop" which consumes most of the computation time.

The analysis of one algorithm often applies to another algorithm as well. The quantity a in the above analysis appears also in the analysis of an algorithm to compute the inverse of a permutation [10, vol. 1, Algorithm 1.3.35].

The above analysis can also be extended to measure the advisability of incorporating various refinements into the algorithm. For example, suppose that we introduce a new counter variable called <u>tally</u>, which is initially set to n. In lines 11, and also in line 12, we will insert the statement "<u>tally</u> := <u>tally</u>-1 ", and at the end of line 12 we insert a new test "<u>if</u> tally = 0 <u>then</u> <u>go</u> <u>to</u> exit " where exit ends the program. This modification (cf. MacLeod [12]) will terminate the algorithm earlier, since it stops the j loop when the largest leader has been found. At the cost of 1 more variable, 1 execution of the statement " tally := n ", n executions of " tally := tally - 1 " and $b \approx \log n$ tests " <u>if</u> tally = 0 ", we save some of the work in lines 2-7 and make it unnecessary to test " if $j > n$ " to see whether the j loop is exhausted. How much is saved? It is not difficult to see that the number of iterations of lines 4 and 7 is reduced by $y_{12} + y_{13} + \cdots + y_{1n}$, so the average saving is $\frac{1}{2} + \frac{1}{3} + \ldots + \frac{1}{n} = H_n - 1 \approx \ln n$. The number of iterations of line 6, the main loop, is reduced by

$\sum_{2 \le i < j \le n} y_{1j} y_{ij}$, an average saving of

$$\sum_{2 \le i < j \le n} \frac{1}{j(j-i+1)} = \sum_{2 \le r < j \le n} \frac{1}{rj} = \frac{1}{2}\left(\left(\sum_{2 \le r \le n} \frac{1}{r}\right)^2 - \sum_{2 \le r \le n} \frac{1}{r^2}\right)$$

$$= \frac{1}{2}(H_n^2 - H_n^{(2)}) - H_n + 1 \approx \frac{1}{2}(\ell n\ n)^2 .$$

So the net improvement caused by this change appears to be rather small. This conclusion would not be evident <u>a priori</u>, if we had not made the analysis, since the average length of a cycle which starts at a <u>random</u> place in a random permutation is well known to be $\frac{1}{2}(n+1)$.

The application to rectangular matrix transposition suggests that we might also design a similar algorithm in which the function p^{-1} is input as well as p. Then we could look for a cycle leader by first testing $p(j)$, then $p^{-1}(j)$, $p(p(j))$, $p^{-1}(p^{-1}(j))$, etc., until finding a value $\le j$. It turns out that the average number of operations performed is the same as in the above algorithm, but the worst case is reduced to $O(n \log n)$. (This solves a problem stated by MacLeod [12].)

It is amusing to derive the latter $n \log n$ bound by obtaining the exact maximum number $f(n)$ of steps needed to rule out each of the non-leaders, while processing an n-cycle, if we assume that both $p^k(j)$ and $p^{-k}(j)$ are fetched simultaneously in one step: Consider first placing the element 1 , then 2 , 3 , etc. into an initially empty cycle so as to obtain the worst case; we obtain the recurrence

$$f(1) = 0\ , \quad f(n) = \max_{1 \leq k < n} (\min(k,n-k) + f(k) + f(n-k)) \ .$$

The solution to this recurrence is rather interesting, it turns out to be $f(n) = \sum_{0 \leq k < n} \nu(k)$, where $\nu(k)$ is the number of 1's in the binary representation of k . If $a_1 > a_2 > \ldots > a_r$,

$$f(2^{a_1} + 2^{a_2} + \ldots + 2^{a_r}) = \tfrac{1}{2}(a_1 2^{a_1} + (a_2+1)2^{a_2} + \ldots + (a_r+r-1)2^{a_r}) \ .$$

The fact that this function satisfies the recurrence can be proved by letting $g(m,n) = f(m+n)-m-f(m)+f(n)$, and showing that $g(2m,2n) = 2g(m,n)$, $g(2m+1,2n) = g(m,n)+g(m+1,n)$, $g(2m,2n+1) = g(m,n)+g(m,n+1)$, $g(2m+1,2n+1) = 1+g(m+1,n) + g(m,n+1)$; hence by induction $g(m,n) \geq 0$ when $m \leq n$, with equality when $m = n$ or $n-1$. (Asymptotic properties of $f(n)$ have been studied by Bellman and Shapiro [1].)

So much for Type A analysis of <u>in situ</u> permutation; what can be said about the computational complexity of this problem? I don't really know; it seems reasonable to conjecture that every algorithm for <u>in situ</u> permutation will require at least $n \log n$ steps on the average, but I don't know how to prove it.

In the first place there is a difficulty in defining the idea of a "step"; the above frequency analysis assumes that $p(k)$ can be calculated in one step, and that $x[k]$ can be fetched or stored in one step, for arbitrary k between 1 and n. A complexity analysis must, however, consider the limit as $n \to \infty$; an algorithm which works optimally for all $n \leq n_0$ could be set up with something like $n_0!$ branches, and this would be uninteresting. But as $n \to \infty$, it takes at least $\log n$ steps just to look at the number k when we are dealing with $p(k)$ or $x[k]$, so the above program really takes $n(\log n)^2$ steps instead of $n \log n$, as $n \to \infty$. On the other hand, no programmer really believes that the above algorithm really takes $\log n$ steps each time $x[k]$ is fetched or stored, since the time is bounded for any reasonable n to which he wants to apply the algorithm. In other words, we want a complexity measure that models the situation for practical ranges of n, even though the model is unrealistic as $n \to \infty$, and in spite of the fact that we require the algorithm to be valid for arbitrarily large n.

A second difficulty is how to phrase the "no auxiliary memory" idea. If we assume that the x's are integers, and if we allow arithmetic operations to be carried out, we could replace each $x[k]$ by $2x[k]$ and use the units digits as n extra tag bits. If operations on the x's are forbidden, yet auxiliary integer variables like j, k, ℓ in the above algorithm are allowed, we could still get the effect of n extra bits of memory by doing arithmetic on an integer variable whose value ranges from 0 to 2^n-1; on the other hand, it isn't obvious that any $\underline{O}(n)$ algorithm could be designed even when such a trick is used.

These considerations suggest a possible model of the problem. Consider a device with n^c states, for some constant c. Each state deterministically specifies the a "step" of the computation by (a) specifying numbers i and j, $1 \leq i \leq j \leq n$, such that $x[i]$ is to be interchanged with $x[j]$; and (b) specifying a number k and n states $q_1, \ldots, q_n$ such that the next step is $q_{p(k)}$. Can such a device do the rearrangement in $O(n)$ steps, or are $n \log n$ steps needed?

Selecting the t-th largest. Now let us turn to another problem, this time somewhat less academic. C. A. R. Hoare [8] has given a method for finding the t-th largest of n elements, by making repeated comparisons, and F. E. J. Kruseman Aretz has shown (see [15] that Hoare's method makes approximately $(2+2 \ln 2)n$ comparisons when finding the median of the elements, i.e., when $t = (n+1)/2$. Our goal is to do a partial frequency analysis of the algorithm, determining the exact average number of comparisons which are made, as a function of t and n.

I shall state the algorithm informally, since I am not attempting to make a frequency analysis of each step. Let $x[1], \ldots, x[n]$ be the given elements, and assume that they are distinct. We start by selecting an arbitrary element y, and compare it to each of the $n-1$ others, rearranging the other elements (as in "quicksort") so that all elements $> y$ appear in positions $x[1], \ldots, x[k-1]$, while all elements $< y$ appear in positions $x[k+1], \ldots, x[n]$. Thus, y is the k-th largest. If $k = t$, we are done; if $k > t$ we use the same method to find the t-th largest of $x[1],\ldots,x[k-1]$; and if $k < t$ we find the $(t-k)$-th largest of $x[k+1],\ldots,x[n]$. A very interesting formalization and proof of this procedure has recently been given by Hoare [9].

Let $C_{n,t}$ be the average number of comparisons made by the above process; we have

$$C_{1,1} = 0 \; ; \quad C_{n,t} = n-1+\frac{1}{n}(A_{n,t}+B_{n,t}) \; , \quad \text{for} \quad 1 \le t \le n \; , \quad n \ge 2 \; ,$$

where

$$A_{n,t} = C_{n-1,t-1} + C_{n-2,t-2} + \cdots + C_{n-t+1,1} \; ,$$

$$B_{n,t} = C_{t,t} + C_{t+1,t} + \cdots + C_{n-1,t} \; .$$

This is not the kind of recurrence that we would ordinarily expect to solve, but let us make the attempt anyway. The first step in problems of this kind is to get rid of the sums, by noting that

$$A_{n+1,t+1} = A_{n,t} + C_{n,t} \; , \quad B_{n+1,t} = B_{n,t} + C_{n,t} \; ;$$

then we can eliminate the A's and B's to get a "pure" recurrence in the C's:

$$\begin{aligned}(n+1)C_{n+1,t+1} - nC_{n,t+1} - nC_{n,t} + (n-1)C_{n-1,t} \\ = (n+1)n - n(n-1) - n(n-1) + (n-1)(n-2) + A_{n+1,t+1} - A_{n,t+1} - A_{n,t} + A_{n-1,t} \\ + B_{n+1,t+1} - B_{n,t+1} - B_{n,t} + B_{n-1,t} \\ = 2 + C_{n,t} - C_{n-1,t} + C_{n,t+1} - C_{n-1,t}\end{aligned}$$

or in other words

$$C_{n+1,t+1} - C_{n,t+1} - C_{n,t} + C_{n-1,t} = 2/(n+1) \; . \qquad (*)$$

What an extraordinary coincidence that $n+1$ was a common factor on each of the C's ! This phenomenon suggests that we may actually be able to solve the recurrence after all.

Checking the derivation shows that formula (*) is valid for $1 < t < n$; we need to look at the boundary conditions next, when $t = 1$ or $t = n$:

$$C_{n,1} = n - 1 + \frac{1}{n}(C_{1,1} + C_{2,1} + \ldots + C_{n-1,1}) \ ;$$

$$(n+1)C_{n+1,1} - nC_{n,1} = (n+1)n - n(n-1) + C_{n-1} \ ;$$

$$C_{n+1,1} - C_{n,1} = 2n/(n+1) = 2 - 2/(n+1) \ .$$

This is a recurrence that is easily solved, $C_{n,1} = 2n-2H_n$. By symmetry, $C_{n,n} = 2n-2H_n$ also. Now the recurrence

$$(C_{n+1,t+1} - C_{n,t}) - (C_{n,t+1} - C_{n-1,t}) = 2/(n+1)$$

implies that

$$C_{n+1,t+1} - C_{n,t} = \frac{2}{n+1} + \frac{2}{n} + \ldots + \frac{2}{t+2} + C_{t+1,t+1} - C_{t,t}$$

$$= 2(H_{n+1} - H_{t+1}) + 2 - 2/(t+1) \ ,$$

and this relation likewise can be iterated:

$$C_{n,t} = 2\sum_{2 \le k \le t} (H_{n-t+k} - H_k + 1 - 1/k) + C_{n+1-t,1} \ .$$

Thus, finally, we have the solution

$$C_{n,t} = 2((n+1)H_n - (n+3-t)H_{n+1-t} - (t+2)H_t + n + 3) \ , \quad \text{for } 1 \le t \le n \ .$$

For example, when calculating the median of $n = 2t-1$ elements, the average number of comparisons comes to

$$4t(H_{2t-1}-H_t) + 4t - 8H_t + 4 = (4+4 \ln 2)t - 8 \ln t + 1 - 8\gamma + \underset{\sim}{O}(t^{-1}) .$$

A Type B analysis of this problem is essentially the question "What is the smallest number of comparisons needed to select the t-th largest of n elements?" There are really two questions, depending on whether we want to minimize comparisons in the worst case or in the average case.

When $t = 1$, the questions are easily answered; we always need at least $n-1$ comparisons to determine the largest elements. For if we consider each comparison as a match in a knockout tournament, every player except the champion must lose at least one game. This argument can be extended also to the case $t = 2$, to show that an algorithm to determine the second best player must use at least $n - 2 + \lceil \log_2 n \rceil$ comparisons, a result first stated by J. Schreier in 1932 and first proved rigorously by S. S. Kislitsin in 1964. (See [, vol. 3, section 5.3.3] for further details and references.)

When $t = 3$ the minimum number of comparisons in the worst case is still not known; and the minimum <u>average</u> number of comparisons is not even known when $t = 2$.

The random-finding problem is especially fascinating. No algorithm for computing medians is known which requires less than $n \log n$ comparisons in its worst case. And no proof that $n \log n$ comparisons are necessary has been found. However, R. W. Floyd has recently discovered efficient ways to compute medians with an average of only $\frac{3}{2} n + \underset{\sim}{O}(n^{2/3} \log n)$ comparisons; and he has proved that at least $\frac{5}{4} n + \underset{\sim}{o}(n)$ comparisons are necessary on the average, no matter what algorithm is used [4].

Summary. I have tried to indicate the nature of algorithmic analysis by describing two nontrivial problems in detail. Perhaps the complexities of these examples have obscured the main points I wanted to make, so I will attempt to summarize what I think is most important.

1. Analysis of algorithms is an interesting activity which contributes to our fundamental understanding of computer science. In this case, mathematics is being applied to computer problems, instead of applying computers to mathematical problems.

2. Analysis of algorithms relies heavily on techniques of discrete mathematics, such as the manipulation of harmonic numbers, the solution of difference equations, and combinatorial enumeration theory. Most of these topics are not presently being taught in colleges and universities, but they should form a part of many computer scientists' education.

3. Analysis of algorithms is beginning to take shape as a coherent discipline. Instead of using a different trick for each problem, there are some reasonably systematic techniques which are applied repeatedly. (Numerous examples of these unifying principles may be found by consulting the entries under "Analysis of algorithms" in the index to [10].) Furthermore, the analysis of one algorithm often applies to other algorithms.

4. Many fascinating problems in this area are still waiting to be solved.

Bibliography

[1] R. Bellman and H. N. Shapiro, "On a Problem in Additive Number Theory," Annals of Mathematics **49** (1948), 333-340.

[2] J. Boothroyd, "Algorithm 302, Transpose vector stored array," Comm. ACM **10** (May, 1967), 292-293.

[3] Howard B. Demuth, Electronic Data Sorting (Ph.D. thesis, Stanford University, 1956), 92 pp.

[4] R. W. Floyd, "Notes on computing medians, percentiles, etc." In preparation; perhaps to appear in IFIP Congress Proceedings.

[5] D. Foata, "Problèmes d'Analyse Combinatoire," Publ. Inst. Statistique, Univ. Paris, **14** (1965), 81-241.

[6] Lester R. Ford, Jr. and Selmer Johnson, "A tournament problem," American Mathematical Monthly **66** (1959), 387-389.

[7] Herman H. Goldstine and John von Neumann, "Planning and Coding Problems for an Electronic Computing Instrument," in John von Neumann's Collected Works, A. H. Taub, ed., **5** (Pergamon Press, 1963), 80-235.

[8] C. A. R. Hoare, "Algorithm 65, Find" Comm. ACM (July 1961), 321-322.

[9] C. A. R. Hoare, "Proof of a program: Find," Comm. ACM **14** (January 1971), 39-45.

[10] Donald E. Knuth, The Art of Computer Programming (Addison-Wesley Publishing Corporation: Volume 1, 1968; volume 2, 1969; volume 3, 1972).

[11] Donald E. Knuth, "The Analysis of Algorithms," Proc. International Congress of Mathematics, Nice, September 1970.

[12] I. D. G. MacLeod, "An algorithm for in-situ permutation," Australian Computer Journal **2** (1970), 16-19.

[13] Arnold Scholz, "Aufgabe 253," Jahresbericht der deutschen Mathematiker-Vereinigung, class II, **47** (1937), 41-42.

[14] Hugo Steinhaus, Mathematical Snapshots. (Oxford University Press, 1950), 38-39.

[15] M. H. van Emden, "Increasing the efficiency of Quicksort," Comm. ACM **13** (September, 1970), 563-567.