UNCLASSIFIED

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER RADC-TR-82-152, Vol II (of two)	2. GOVT ACCESSION NO. AD-A118814	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) ALGORITHMIC COMPLEXITY		5. TYPE OF REPORT & PERIOD COVERED Final Technical Report June 1979 - August 1981
		6. PERFORMING ORG. REPORT NUMBER 81-161
7. AUTHOR(s) Edmund A. Lamagna Ralph E. Bunker Leonard J. Bass Philip J. Janus Lyle A. Anderson		8. CONTRACT OR GRANT NUMBER(s) F30602-79-C-0124
9. PERFORMING ORGANIZATION NAME AND ADDRESS University of Rhode Island Dept of Computer Science and Statistics Kingston RI 02881		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS 61101F LD9202C1
11. CONTROLLING OFFICE NAME AND ADDRESS Rome Air Development Center (COEE) Griffiss AFB NY 13441		12. REPORT DATE June 1982
		13. NUMBER OF PAGES
14. MONITORING AGENCY NAME & ADDRESS(If different from Controlling Office) Same		15. SECURITY CLASS. (of this report) UNCLASSIFIED
		15a. DECLASSIFICATION/DOWNGRADING SCHEDULE N/A

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release; distribution unlimited

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, If different from Report)

Same

18. SUPPLEMENTARY NOTES

RADC Project Engineer: Joseph P. Cavano (COEE)

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

Analysis of Algorithms
Computational Complexity
Software Quality
Efficiency

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

The objective of this study was to conduct applied research directed toward understanding the relationship between the complexity or efficiency of algorithms and the overall quality of computer software. The final report is presented in a two volume series consisting of a total of eight parts. This volume, containing Parts 3 through 8 describes the results of several technical investigations which were conducted.

Part 3 is a tutorial on computational algebra, illustrating the nature of

DD FORM 1473   EDITION OF 1 NOV 65 IS OBSOLETE

UNCLASSIFIED

Con+d

research in the area of algorithm analysis and computational complexity. The specific problems examined include raising a number to a power, evaluating a polynomial at one or several points, and multiplying polynomials and matrices.

Part 4 develops a systematic approach to the analysis of algorithms. The method consists of translating program loops into recursive subroutines, and the semantic manipulation of expressions representing the joint probability distribution function of the program variables. This technique is applied to several simple algorithms, sorting and searching algorithms, and a tree insertion/deletion algorithm.

Part 5 is an experimental analysis of a fast, new sorting method called DPS (distributive partitioning sorting). It develops a framework for conducting such experiments, and proposes several improvements to DPS for dealing with data from unknown or skewed distribution.

Part 6 applies order statistics to investigate the expected quality of several approximation algorithms for the Euclidean traveling salesman problem, known to be NP-complete.

Part 7 presents a survey of data base access methods for both univariate and multivariate range queries. The techniques discussed include B-trees and extensible hashing for the univariate case, and radix bit mapping and K-D-B-trees for the multivariate case.

Part 8 describes an experimental evaluation of the frame memory model of a data base structure. Frame memory is an analytic model which enables the prediction of access performance measures in terms of user behavior parameters.

Accession For

NTIS GRA&I	☒
DTIC TAB	☐
Unannounced	☐
Justification	

By
Distribution/
Availability Codes

| Dist | Avail and/or Special |
| A | |

PREFACE

This is the second of two volumes constituting the final technical report for a study entitled "Algorithmic Complexity". The work was performed in support of the Information Sciences Division, Rome Air Development Center, under U.S. Air Force Systems Command contract F30602-79-C-0124. The duration of the project was from June 1979 through August 1981.

The research described herein was performed by members of the Department of Computer Science and Experimental Statistics at the University of Rhode Island. Dr. Edmund A. Lamagna served as Principal Investigator for this effort. Dr. Leonard J. Bass was Co-Principal Investigator. Three graduate assistants -- Messrs. Lyle A. Anderson, Ralph E. Bunker, and Philip J. Janus -- also worked on the project. Technical guidance was provided by Mr. Joseph P. Cavano, RADC Project Engineer.

The study consists of eight parts, whose titles are:

1. Measures of Algorithmic Efficiency: An Overview (Lamagna)

2. The Performance of Algorithms: A Research Plan (Lamagna, Bass, and Anderson)

3. Fast Computer Algebra (Lamagna)

4. Systematic Analysis of Algorithms (Anderson)

5. Adaptive Methods for Unknown Distributions in Distributive Partitioning Sorting (Janus)

6. Expected Behavior of Approximation Algorithms for the Euclidean Traveling Salesman Problem (Lamagna with E. J. Carney and P. V. Kamat)

7. Data Base Access Methods (Bass)

8. An Experimental Evaluation of the Frame Memory Model of a Data Base Structure (Bunker and Bass)

Volume I contains Parts 1 and 2, comprising a general introduction to the entire series and a research plan. Volume II contains the remaining six parts, describing the results of several technical investigations which were conducted.

ALGORITHMIC COMPLEXITY
Part 3

by

Edmund A. Lamagna

FAST COMPUTER ALGEBRA

Abstract

New algorithms for solving familiar algebraic problems on computers have recently been devised. These methods are more efficient than classical ones for large problem sizes, and some can be shown to be optimal. This tutorial illustrates these ideas by examining the problems of raising a number to a power, evaluating a polynomial at one or several points, and multiplying polynomials and matrices.

FAST COMPUTER ALGEBRA

The astounding speed of modern digital computers has made it possible to perform computations of a size that would be completely infeasible without their use. For example, the fastest computers of today can solve a system of one hundred simultaneous linear equations in a hundred unknowns in a matter of seconds. If a person could perform one arithmetic operation, such as an addition or multiplication, per minute and worked on the problem non-stop using the classical method of Gaussian elimination, it would take almost one year to obtain the same result. In fact, under the same assumptions, it would take a person just over a day to solve a system of only a dozen equations. The calculations of our tireless human computer are, of course, far more susceptible to error.

Before the advent of digital computers, the sizes of most algebraic and numeric problems which could be solved was severely limited although it was known how to solve large problems in principle. Because the sizes of the problems tackled by hand were small, applying a simple formula usually sufficed to produce the desired results. Little attention was generally paid to finding computationally efficient, but perhaps longer, formulas or methods of calculation

During the past decade, a new branch of mathematical computer science known as analysis of algorithms and computational complexity has blossomed. The goal of this field is to compare the relative efficiency of alternative techniques for solving a problem and, whenever possible, to prove that some method is the best one could hope to find. As a result of this work, a number of surprising new algorithms, or computational procedures, have been developed. These techniques sometimes seem counterintuitive at first and often do not outperform the classical methods for small problem sizes. However, as the problem size increases, the improvement in execution speed on a computer can be quite dramatic.

As a simple example of such a result, we consider the problem of multiplying two complex numbers. A single complex number is generally denoted by a pair of values representing its real and imaginary parts. The product of two such numbers $a+bi$ and $c+di$ is given by the formula $(ac-bd)+(ad+bc)i$, where i is $\sqrt{-1}$. Suppose we are given a and b, representing the real and imaginary part of the first complex number, and c and d, representing the corresponding parts of the second number, and are asked to calculate the real and imaginary parts of their product. Computers represent and operate on complex numbers in just this manner. Applying the formula for complex product, four multiplications (viz., ac, bd, ad, and bc), one subtraction (viz., $ac-bd$), and one addition (viz., $ad+bc$) are used.

An alternative method for computing the result is as follows. First add a to b and c to d, multiplying these two sums: $(a+b)\cdot(c+d)=ac+ad+bc+bd=m_1$. Next form the two products $m_2=a\cdot c$ and $m_3=b\cdot d$. The real part of the complex product can be formed as m_2-m_3 and the imaginary part as $m_1-m_2-m_3$. This method uses three multiplications, two additions, and three subtractions. Although this new method performs eight operations to the classical method's six, it does use one fewer multiplication. But a multiplication operation executes far more slowly than either an addition or subtraction on a computer. (Additions and subtractions execute at comparable speeds.) If a multiplication takes a not uncommon factor of ten times longer, the new procedure for complex product will run about 20% faster than the classical one. Due to the speed of digital computers, this improvement will go unnoticed if only a few complex products are to be taken, however it can become increasingly important as the amount of work to be done grows.

In order to compare the efficiencies of alternative algorithms for some algebraic problem, we will have to make more precise just what types of computations will be allowed. Researchers who study the complexity of algebraic problems use a model of computation called a "straight-line algorithm". Within this framework we are given a set of input data plus any constants we choose to work with. Algorithms consist of a sequence of steps in which arithmetic operations are applied to the input data, constants, or the results obtained in previous computation steps. Figure 2 shows two straight-line algorithms which evaluate the polynomial $p(x)=x^3+4x^2+5x+2$ given the value of the variable x as data.

To assess the efficiency of an algorithm, we will count either the total number of arithmetic operations performed or the number of some specific type (e.g., multiplications). This model of computation ignores many practical considerations which will affect the running time of an algorithm if it is actually programmed to be executed on a computer. In a programmed realization, the operations to be performed are systematically specified using loops and tests so that the program will work for all input sizes. The straight-line algorithm paradigm neglects the cost of the overhead associated with loop control and testing operations, as well as the time required to fetch and store information inside a computer's memory. These costs can vary greatly from computer to computer, and will not even be the same for two programming language compilers implemented on the same machine. Fortunately the overall times of the algorithms studied are driven primarily by the underlying structure of the arithmetic operations performed, rather than such overhead considerations, so the results obtained are generally accurate to within a small constant factor for actual implementations.

Evaluation of Powers

Suppose we are given a real number x and a positive integer n and are asked to find the value of x^n. The obvious way to solve this problem is to start with x and multiply by x a total of $n-1$ times. For example, to find x^{32} we compute each of the partial results $x^2, x^3, x^4, \ldots, x^{31}, x^{32}$ and arrive at the desired answer after 31 multiplication steps. We will call this algorithm the "brute force" method of computing x^n.

A more efficient way to arrive at x^{32} is by repeated squaring of each partial result. For example, at the first step we square x to obtain x^2. At the second step x^2 is squared to yield x^4, and so on. Using this technique we arrive at x^{32} in 5 multiplications via the following sequence of partial results: $x^2, x^4, x^8, x^{16}, x^{32}$.

The method of repeated squaring can be used to compute x^n with $\log_2 n$ multiplications when n is a power of two. (If $y=\log_2 x$, the base-2 logarithm of x, then $2^y=x$.) This achieves an exponential improvement over the brute force method. The difficulty is that the algorithm can only be used directly when n is a power of two.

An algorithm called the "binary method" generalizes the principle of repeated squaring to work for all values of n. To apply this technique, we begin by writing down the binary representation of the number n with any leading zeros deleted. For example, if $n=21$ we write $21=(10101)_2$. Ignoring the first bit in the binary representation (which must be 1), we next replace each remaining 1 by the letters SX and each 0 by the letter S. When $n=21$ we obtain the sequence of letters $SSXSSX$. This sequence yields a rule for evaluating x^n if we interpret each S to mean "square the result of the previous step" and each X to mean "multiply the result of the previous step by x".

3-4

In our example, we begin by squaring x to obtain x^2 since the first letter in the sequence is S. We next square this partial result to obtain x^4 since the second letter is again S. Because the third letter is X, we multiply the result of the second step by x to yield x^5 at the third step, and so on. Hence we arrive at x^{21} by the following sequence of partial results: $x^2, x^4, x^5, x^{10}, x^{20}, x^{21}$.

We now wish to investigate the number of multiplication steps the binary method uses to compute x^n. There are $\lfloor \log_2 n \rfloor + 1$ bits in the binary representation of n. ($\lfloor x \rfloor$ denotes the "floor" of x, or the largest integer less than or equal to x.) Let $\nu(n)$ denote the number of these bits which are 1. Since one S occurs in the evaluation sequence for each bit in the binary representation of n other than the first, the number of squaring operations used is $\lfloor \log_2 n \rfloor$. Furthermore, the number of X's in the sequence is just one less than $\nu(n)$. Hence $\lfloor \log_2 n \rfloor + \nu(n) - 1$ multiplications are used overall. Since $\nu(n) \le \lfloor \log_2 n \rfloor + 1$, with equality holding when all the bits in n's representation are 1, the number of steps in the binary method does not exceed $2\lfloor \log_2 n \rfloor$.

The smallest value of n for which the binary method is not optimal is 15. The binary method uses 6 multiplications to evaluate x^{15}, with the sequence $SXSXSX$ giving rise to the partial results $x^2, x^3, x^6, x^7, x^{14}, x^{15}$. However, x^{15} can be calculated in 5 steps by first finding $y = x^3$, and raising y to the fifth power with three more multiplications since $y^5 = (x^3)^5 = x^{15}$.

This method for computing x^{15} is based on the realization that 15 can be factored as 3 times 5. In general if the number n can be factored as $n = p \cdot q$, then x^n can be evaluated by first computing $y = x^p$ and then calculating $y^q = (x^p)^q = x^{p \cdot q} = x^n$. We now describe an algorithm called the "factor method" which is based upon this principle.

Algorithm F: factor method. (Note that a prime number is one having no integer factors other than 1 and itself.)

F1. If $n=1$, we have x^n with no calculation.

F2. If n is prime, calculate x^n by first finding x^{n-1} using the factor method; then multiply this quantity by x.

F3. Otherwise, write n as $p \cdot q$, where p is the smallest prime factor of n and $q>1$. Calculate x^n by first finding x^p via the factor method; then raise this quantity to the qth power, again via the factor method.

We illustrate this technique by showing how x^{21} is evaluated. First, 21 is factored as $3 \cdot 7$, and x^3 is calculated by repeated use of the algorithm with 2 multiplications. Our problem reduces to calculating y^7 where $y=x^3$. Since 7 is a prime, y^7 will be computed by first finding y^6 and then multiplying by y. Repeated use of the algorithm reveals that y^6 will be found by taking $(y^2)^3$. Letting $z=y^2$, the steps used to evaluate x^{21} are: (1) $x \cdot x = x^2$, (2) $x^2 \cdot x = x^3 = y$, (3) $x^3 \cdot x^3 = x^6 = y^2 = z$, (4) $x^6 \cdot x^6 = x^{12} = z^2$, (5) $x^{12} \cdot x^6 = x^{18} = z^3 = (y^2)^3$, and (6) $x^{18} \cdot x^3 = x^{21} = y^6 \cdot y$.

Although the performance of the factor method is better than that of the binary method on the average, there are instances when the binary method is superior. The smallest such case is $n=33$, where the factor method uses 7 multiplications and the binary method only 6. In fact there are infinitely many values of n for which the factor method is better than the binary method, and vice versa. Moreover, neither the factor method nor the binary method need be optimal. The smallest such case is $n=23$, where both the factor and binary methods use 7 multiplications. However, x^{23} can be calculated with 6 multiplications as follows: (1) $x \cdot x = x^2$, (2) $x^2 \cdot x = x^3$, (3) $x^3 \cdot x^2 = x^5$, (4) $x^5 \cdot x^5 = x^{10}$, (5) $x^{10} \cdot x^{10} = x^{20}$, and (6) $x^{20} \cdot x^3 = x^{23}$.

Since neither the factor method nor the binary method is always optimal, it would seem worthwhile to investigate just how good these two methods are. Let $P(n)$ denote the minimum number of multiplications required to calculate x^n regardless of the method employed. Observe that the greatest power of x which can be obtained using k multiplications is x^{2^k}, and this is obtained by successively squaring x at each step of the computation. Thus in order to compute a power of x as large as x^n, we must use at least k multiplications where $2^k \geq n$, or equivalently, $k \geq \lceil \log_2 n \rceil$. ($\lceil x \rceil$ denotes the "ceiling" of x or the smallest integer greater than or equal to x.) Therefore $P(n) \geq \lceil \log_2 n \rceil$.

The above result provides a lower bound on the number of multiplications required to compute x^n. It states that at least $\lceil \log_2 n \rceil$ multiplications are necessary, but gives no indication at all of whether this number is sufficient. Our earlier analysis of the binary method further reveals that $P(n) \leq 2\lfloor \log_2 n \rfloor$, and hence the binary method is guaranteed to be efficient in the sense that it never uses more than twice the minimal number of multiplications. The factor method demonstrates that if $n=p \cdot q$, then $P(n) \leq P(p)+P(q)$.

The tree in Figure 4 gives a minimal, or optimal, multiplication sequence when n is 100 or less. To calculate x^n we locate n in the tree. The path from the root (bottom) of the tree to n indicates the sequence of exponents which occurs in one optimal evaluation of x^n. The value of $P(n)$, for $1 \leq n \leq 100$, is simply the length of the path in the tree from the root to n. For example, to compute x^{31} we find that we should calculate the following powers of x: 2, 3, 5, 10, 11, 21, 31. Hence an optimal chain is given by: (1) $x \cdot x = x^2$, (2) $x^2 \cdot x = x^3$, (3) $x^3 \cdot x^2 = x^5$,

(4) $x^5 \cdot x^5 = x^{10}$,   (5) $x^{10} \cdot x = x^{11}$,   (6) $x^{11} \cdot x^{10} = x^{21}$,   (7) $x^{21} \cdot x^{10} = x^{31}$.   The idea behind the method is that any number in the tree can be written as the sum of two numbers, or twice a single number on the path between itself and the root.   The sums formed to reach n in the tree correspond to the intermediate powers formed in calculating x^n.   Of course we knew all along that $x^a \cdot x^b = x^{a+b}$.

In our discussion of the power evaluation problem we have concentrated on the operation of multiplication.   Neither addition nor subtraction are of any help in evaluating powers.   But what about division?   We have just seen that 7 multiplications are minimal to compute x^{31}.   However if division is allowed, x^{31} can be found in 6 operations by calculating x^{32} with 5 multiplications via repeated squaring, and then dividing this quantity by x. Unfortunately, the availability of division does not alter our lower bound on $P(n)$, and hence cannot substantially improve on the algorithms we have discussed.

The problem of finding an optimal computation sequence for x^n has a long and interesting history.   Although Arnold Scholz formally raised the question in 1937, before the appearance of digital computers, algorithms for computing x^n had been studied for some time earlier.   A version of the binary method was expounded by the famous French mathematician Adrien M. Legendre in 1798, and it is closely related to a multiplication procedure used by Egyptian mathematicians as early as 1800 B.C.   Several authors have published statements of the optimality of the method but, as we have seen, these claims are false. We note in closing that the algorithms studied work not only for single number, but carry over to the problems of raising polynomials and matrices to a power.

Evaluation of Polynomials

The next problem we consider is that of evaluating a general polynomial of degree n. Such a polynomial may be written as $p(x) = c_n x^n + c_{n-1} x^{n-1} + \ldots + c_1 x + c_0$. We want to devise an efficient scheme to evaluate any such polynomial when given the values of the coefficients $c_n, c_{n-1}, \ldots, c_1, c_0$ and the variable x as data.

The usual method for evaluating a polynomial is to apply directly the general formula given above. In so doing we first calculate each of the powers of x using $n-1$ multiplications. Next we take the product of each coefficient and its corresponding power. This requires another n multiplications. Finally the $n+1$ terms in the canonical expansion are summed with n additions. Thus a total of $2n-1$ multiplications and n additions are used by this method.

When the term-by-term method described above is applied to the degree two polynomial $c_2 x^2 + c_1 x + c_0$, three multiplications are performed: $x \cdot x$, $c_2 \cdot x^2$, $c_1 \cdot x$. This number can be reduced to two by observing that x can be factored out of the first two terms, yielding the formula $(c_2 x + c_1) x + c_0$. The number of additions remains two.

This insight suggests that we rearrange degree n polynomials as $p(x) = (\ldots (c_n x + c_{n-1}) x + c_{n-2}) x + \ldots + c_1) x + c_0$. To evaluate the polynomial we start with c_n, multiply by x, add c_{n-1}, multiply by x, add c_{n-2}, multiply by $x, \ldots$, add c_0. This method was expounded in 1819 by William G. Horner in conjunction with an efficient technique for finding the coefficients of the polynomial $p(x+a)$. Today the method is usually referred to as "Horner's rule" although it was actually devised by Isaac Newton over 100 years earlier in 1711.

Horner's rule employs n multiplication and n addition steps to evaluate polynomials of degree n. We might ask whether it is possible to do better. The answer is no for general polynomials in which all of the coefficients and the variable x are left unspecified. Of course particular polynomials, like the one examined in Figure 2, can be evaluated with fewer operations.

We can readily see that n addition/subtraction steps are required because any scheme for evaluating $p(x)$ clearly works when $x=1$ and $p(1)=c_n+c_{n-1}+\ldots+c_0$. This implies that Horner's rule can be adapted to find the sum of any $n+1$ numbers by letting these numbers play the roles of coefficients and setting x to 1. Since n add/sub steps are required to sum $n+1$ numbers and our adaptation of Horner's rule uses exactly this many, the method is optimal with respect to the number of add/sub steps. A demonstration that n mult/div steps are also required is more complex. Since such a proof is based upon more advanced concepts of linear algebra, we shall omit the details here.

Éduard G. Belaga, a Russian mathematician, first demonstrated the necessity of n add/sub steps in 1958. Another Russian, Viktor Pan, showed in 1966 that n mult/div steps are also required. In 1971 Allan Borodin of the University of Toronto further proved that Horner's method is uniquely optimal in the sense that it is the only way to evaluate a general nth degree polynomial with $2n$ arithmetic operations.

Next we consider the problem of evaluating an nth degree polynomial at several points. Applying either the classical algorithm or Horner's method at, say, each of n points requires a number of operations proportional to n^2. Using the concept of a "modular transform", we shall obtain an algorithm whose performance is only slightly worse than linear in n, resulting in a considerable speed-up for increasingly larger values of n.

Evaluating a polynomial at the single point $x=a$ is equivalent to finding the remainder when $p(x)$ is divided by $x-a$. This follows from the Remainder Theorem of algebra, since we can write $p(x)=(x-a)q(x)+r(x)$ where $q(x)$ and $r(x)$ are the quotient and remainder polynomials, respectively, when the division is performed. Note that the degree of $q(x)$ is one less than that of $p(x)$ and $r(x)$ is a constant. Setting $x=a$ we obtain the desired result, that $p(a)$ is equal to the constant r.

This technique can be generalized to the situation in which we wish to evaluate the polynomial $p(x)$ at k points, $a_1, a_2, \ldots, a_k$, where $k \leq$ degree of p. We first form the product $m(x) = \prod_{i=1}^{k} (x-a_i)$. Again from the Remainder Theorem, we find that $p(x) = m(x)q(x) + r(x)$ where $q(x)$ and $r(x)$ are the quotient and remainder polynomials when $p(x)$ is divided by $m(x)$. At each of the points $x = a_i$ the value of $m(a_i) = 0$, so we have $p(a_i) = r(a_i)$. Since the degree of r is less than that of p, we have reduced our problem to the simpler one of evaluating $r(x)$ at the k points.

A common property of many fast algorithms is that they reduce a problem to a simpler one by dividing it into two subproblems, each of which is at most half as difficult as the original problem. Applying this principle, a fast algorithm for the problem of evaluating an nth degree polynomial at $n+1$ points suggests itself. First, divide the $n+1$ points in half and form the polynomials $m_1(x) = \prod_{i=1}^{n/2} (x-a_i)$ and $m_2(x) = \prod_{i=n/2+1}^{n+1} (x-a_i)$. Analogous to what we did above, we next divide $p(x)$ by $m_1(x)$ to get $r_1(x)$ and $m_2(x)$ to get $r_2(x)$. We have now reduced our original problem to that of evaluating the two $n/2$-th degree polynomials $r_1(x)$ and $r_2(x)$ at $n/2+1$ points. To do this we apply the method repeatedly.

For example, suppose we wish to evaluate the polynomial $p(x) = x^3 - 2x^2 + 3x + 1$ at the points $x = -1, 0, 1, 2$. We first form $m_1(x) = (x+1)x = x^2 + x$ and $m_2(x) = (x-1)(x-2) = x^2 - 3x + 2$. Dividing $p(x)$ by $m_1(x)$ and $m_2(x)$, we obtain $r_1(x) = 6x + 1$ and $r_2(x) = 4x - 1$. Dividing $r_1(x)$ by $x+1$ and x, we find from the remainders that $p(-1) = -5$ and $p(0) = 1$. Similarly dividing $r_2(x)$ by $x-1$ and $x-2$, we get that $p(1) = 3$ and $p(2) = 7$.

The tree in Figure 6 illustrates the manner in which the products $m_i(x)$, or "moduli", are built up in general. The divisions of $p(x)$ and the subsequent remainders are computed in the reverse order. If the products are formed moving from the top of the tree downward, and then the divisions are performed going from the bottom of the tree upward, only one polynomial multiplication and one polynomial division need be performed for each node in the tree.

3-11

It turns out that the number of scalar arithmetic operations required to either multiply or divide two polynomials of like degree are the same to within a multiplicative constant. Because of this, the running time of our algorithm for evaluating an nth degree polynomial at $n+1$ points is driven by the time required to symbolically multiply two polynomials given their coefficients. In fact the overall running time of the modular transform algorithm described above is just a factor of $\log n$ greater than the time required to multiply two polynomials of degree n. With this motivation we now turn our attention to the polynomial multiplication problem.

Polynomial Multiplication

In the polynomial multiplication problem, we are given two polynomials represented by their coefficients as data and are asked to compute the coefficient representation of their product. Let the two input polynomials be $p(x) = \sum_{i=0}^{n-1} a_i x^i$ and $q(x) = \sum_{i=0}^{n-1} b_i x^i$. (It will be easier to work with polynomials having n coefficients and degree $n-1$, rather than those of degree n.) Their product is a polynomial of degree $2n-2$, $p(x) \cdot q(x) = \sum_{k=0}^{2n-2} c_k x^k$, whose coefficients expressed in terms of the inputs are $c_k = \sum_{i=0}^{k} a_{k-i} b_i$. Note that c_k is the sum of all products of the form $a_i b_j$ in which $i+j=k$. Thus $c_0 = a_0 b_0$, $c_1 = a_1 b_0 + a_0 b_1$, $c_2 = a_2 b_0 + a_1 b_1 + a_0 b_2$, etc.

The classical algorithm for polynomial multiplication is to apply the formula given above to compute each coefficient in the result directly. In so doing n^2 scalar multiplications are used since every product of the form $a_i b_j$, involving one of the n coefficients from each input polynomial, is formed exactly once. The total number of additions made is equal to the number of $a_i b_j$ pairs less the number of coefficients formed, or $n^2 - (2n-1)$. For example, if $p(x) = a_1 x + a_0$ and $q(x) = b_1 x + b_0$, the product is $p(x) \cdot q(x) = (a_1 b_1) x^2 + (a_1 b_0 + a_0 b_1) x + a_0 b_0$. Here $n=2$ and we see that $2^2 = 4$ multiplications (viz., $a_1 b_1$, $a_1 b_0$, $a_0 b_1$, $a_0 b_0$) and $2^2 - (2 \cdot 2 - 1) = 1$ addition (viz., $a_1 b_0 + a_0 b_1$) are performed.

3-12

It turns out that the product of two 2-coefficient polynomials can be found with 3 multiplications, instead of the usual 4. This product can be expressed in terms of the three multiplications $m_1 = a_0 \cdot b_0$, $m_2 = a_1 \cdot b_1$, and $m_3 = (a_1 + a_0) \cdot (b_1 + b_0)$ as $p(x) \cdot q(x) = m_2 x^2 + (m_3 - m_2 - m_1)x + m_1$. Although this scheme uses 4 add/sub steps, it can serve as the basis for an algorithm to multiply two n-coefficient polynomials with a substantially smaller total number of operations than the classical algorithm for increasingly larger values of n.

To see how such a reduction in work is possible, consider the case when $n=4$. The classical algorithm uses $4^2=16$ multiplications to find the product of the two polynomials $p(x) = a_3 x^3 + a_2 x^2 + a_1 x + a_0$ and $q(x) = b_3 x^3 + b_2 x^2 + b_1 x + b_0$. Observe that we can split these polynomials into upper and lower halves, expressing them as $p(x) = s(x)x^2 + t(x)$ and $q(x) = u(x)x^2 + v(x)$ where $s(x) = a_3 x + a_2$, $t(x) = a_1 x + a_0$, $u(x) = b_3 x + b_2$, and $v(x) = b_1 x + b_0$. In this form the product $p(x) \cdot q(x) = sux^4 + (sv + tu)x^2 + tv$.

If we take each of these 4 subproducts of 2-coefficient polynomials in the classical way we use $4 \cdot 4 = 16$ multiplications. But instead we can take advantage of our 3 multiplication scheme by forming the products $m_1 = t \cdot v$, $m_2 = s \cdot u$, and $m_3 = (s + t) \cdot (u + v)$. Since each of these 3 products of 2-coefficient polynomials can be found by a repeated application of the 3 multiplication scheme, only $3 \cdot 3 = 9$ scalar multiplications are used. The desired result is $p(x) \cdot q(x) = m_2 x^2 + (m_3 - m_2 - m_1)x + m_1$. The details of this scheme are illustrated in Figure 11.

Now let us generalize to the case of arbitrary size polynomials. For simplicity we will assume that the number of coefficients, n, is a power of two although a similar result can be derived for any value of n. We begin by dividing the coefficients of the inputs into upper and lower halves, expressing these polynomials as $p(x) = s(x)x^{n/2} + t(x)$ and $q(x) = u(x)x^{n/2} + v(x)$. We next form

3-13

the following products of $n/2$-coefficient polynomials: $m_1 = t \cdot v$, $m_2 = s \cdot u$, and $m_3 = (s+t) \cdot (u+v)$. In taking each of these products, we will apply the algorithm recursively (i.e., repeatedly) until $n=1$. The desired product $p(x) \cdot q(x)$ is ultimately formed as $m_2 x^n + (m_3 - m_2 - m_1) x^{n/2} + m_1$.

The approach to the polynomial multiplication problem described above is an example of what is known as a "divide-and-conquer" algorithm. The idea behind this common algorithm design technique is to split a problem into a number of subproblems of the same kind involving disjoint subsets of the inputs. The subproblems are solved separately by reapplying the divide-and-conquer strategy, and then a method is found to combine the solutions of these subproblems into a solution of the whole problem. Frequently, as in the case of polynomial multiplication, a divide-and-conquer approach can lead to a more efficient algorithm than a direct attack on the original problem. We shall see another application of this paradigm later.

We now investigate the efficiency of the divide-and-conquer polynomial multiplication algorithm. Let $M(n)$ denote the number of scalar multiplications performed in taking the product of two n-coefficient polynomials. Since the scalar multiplications made in forming the product of the two original polynomials are exactly those used to compute the resulting 3 products of $n/2$-coefficient polynomials, we have that $M(n)=3M(n/2)$. This equation, called a recurrence relation, can be solved by back-substitution as follows: $M(n)=3M(n/2)=3^2 M(n/4)=\ldots=3^k M(n/2^k)$. The process stops when $2^k = n$ or $k = \log_2 n$, at which point we use the initial condition $M(1)=1$ to obtain $M(n)=3^{\log_2 n} = n^{\log_2 3}$. ($M(1)=1$ since one multiplication is used to find the product of two 1-coefficient polynomials, $a_0 b_0$.) Because $\log_2 3 \approx 1.59$, the divide-and-conquer algorithm uses $n^{1.59}$ scalar multiplications to the classical algorithm's n^2.

We next obtain a recurrence relation for the number of scalar add/sub steps, $A(n)$. The additions performed when multiplying two n-coefficient polynomials arise from three sources: (1) recursively applying the algorithm to find 3 products of $n/2$-coefficient polynomials, (2) performing 2 additions of $n/2$-coefficient polynomials (viz., $s+t$, $u+v$) and 2 subtractions of $(n-1)$-coefficient polynomials (viz., $m_3-m_2-m_1$), and (3) forming the coefficients of the original product from the results of recursively applying the procedure (e.g., the additions $d_{2,0}+d_{1,1}$ and $d_{1,0}+d_{0,2}$ in the third step of Figure 11). By definition, there are $3A(n/2)$ additions from the first source. The sum of two polynomials is found by merely adding their corresponding coefficients, so the second source contributes $2(n/2)+2(n-1)=3n-2$ additions. The third source generates $n-2$ additions. Hence the desired recurrence relation is $A(n)=3A(n/2)+4n-4$, whose solution with initial condition $A(1)=0$ is $A(n)=6n^{\log_2 3}-8n+2$.

We have just shown that both the number of multiplications and add/sub steps in the divide-and-conquer algorithm grow proportionally to $n^{1.59}$. This can represent a substantial improvement over the classical algorithm, where the number of operations grows as n^2. When $n=8$ the total number of arithmetic operations performed by both algorithms are comparable: 127 for divide-and-conquer to 113 for the classical method. For larger values of n the divide-and-conquer method is superior.

Is this the fastest that two polynomials can be multiplied? The method just described is based on the fact that the product of 2-coefficient polynomials can be found with 3 multiplications. It yields a general algorithm for n-coefficient polynomials in which the number of scalar arithmetic operations performed grows as $n^{\log_2 3}$. Using a divide-and-conquer approach, it is possible to convert any scheme for computing the product of two polynomials of some specific size m with p multiplications into a method for multiplying

arbitrarily large n-coefficient polynomials using $O(n^{\log_m p})$ operations. (The O-notation denotes the order of magnitude of a function's growth rate, ignoring any constant factors of proportionality.) If m and p are such that $\log_m p < 1.59$, the resulting algorithm will be asymptotically better than the one we have discussed. To date no one has been able to produce a faster algorithm using such a strategy.

Algebraic Transforms

One way to specify a polynomial is to give its coefficients. This is the only representation we have been working with up to now. A well-known result in algebra states that there is a unique polynomial of degree less than n which will fit through any n points. Thus an alternate representation for a polynomial is to give its values at n points.

The product of two n-coefficient polynomials is a polynomial with $2n-1$ coefficients. Such a polynomial can be uniquely represented by its value at $2n-1$ points. This suggests a new method for multiplying the n-coefficient polynomials $p(x)$ and $q(x)$. We begin by evaluating both $p(x)$ and $q(x)$ at $2n-1$ selected points, $x = a_1, a_2, \ldots, a_{2n-1}$. We next multiply together the corresponding values of the polynomials at these points, forming $2n-1$ products $p(a_i)\, q(a_i)$. The polynomial which uniquely fits these $2n-1$ values is the desired product $p(x) \cdot q(x)$.

This approach to multiplying two polynomials is called an algebraic transform. Instead of dealing directly with the coefficients of $p(x)$ and $q(x)$, as we have done previously, we first transform the coefficient representation of p and q into another form, one in which the polynomials are represented by their values at a collection of points. We perform the actual multiplication on this second representation by taking the pairwise products of the values of

p and q at the sample points. We now have a representation of the product polynomial in the form of its value at a number of points, and must perform an inverse transformation to obtain the coefficient representation of the result. This final step is called interpolation. The entire process is illustrated in Figure 12.

In a previous section we considered the problem of evaluating a polynomial at several arbitrarily selected points. The performance of the algorithm we described is asymptotically the best of any method known to date. This algorithm uses a number of basic arithmetic operations proportional to $n(\log n)^2$ when evaluating an n-coefficient polynomial at $2n-1$ points. Similarly, the best known algorithm for interpolating a polynomial through $2n-1$ arbitrary points also performs a number of basic arithmetic operations proportional to $n(\log n)^2$. (The more familiar classical interpolation algorithms of Isaac Newton and the noted French mathematician Count Joseph Louis Lagrange employ a number of operations growing as n^2.) Thus the number of operations used in the transform method for multiplying two polynomials is dominated by the evaluation and interpolation steps, rather than the $2n-1$ pairwise multiplications, and is $O(n(\log n)^2)$.

In the transform just described, the values of x where the evaluation and interpolation take place are arbitrarily chosen. It turns out that a judicious choice of points can lead to a slightly improved algorithm. Observe that the polynomial $p(x) = a_0 + a_1 x + \ldots + a_{n-1} x^{n-1}$ can be broken into a sum of odd and even powered terms: $p(x) = (a_0 + a_2 x^2 + \ldots + a_{n-2} x^{n-2}) + (a_1 x + a_3 x^3 + \ldots + a_{n-1} x^{n-1})$. Substituting $y = x^2$, we have $p(x) = (a_0 + a_2 y + \ldots + a_{n-2} y^{n/2-1}) + x(a_1 + a_3 y + \ldots + a_{n-1} y^{n/2-1}) = s(y) + x t(y)$. Thus the problem of evaluating the n-coefficient polynomial $p(x)$ reduces to the problem of evaluating two polynomials $s(y)$ and $t(y)$, of half that size, plus three additional operations: $y = x^2$, $x \cdot t(y)$, $s(y) + x t(y)$. However, we are still faced with the task of evaluating both s and t at the same number of points, and no reduction in the number of operations has occurred yet.

When the points where the evaluation and interpolation take place are chosen to be the primitive n-th roots of the equation $x^n=1$, the process can be speeded. If ω is one of these primitive n-th roots of unity, then $\omega^n=1$ and $\omega^k\neq1$ for all $k<n$. Moreover, if n is even then ω^2 is a primitive $n/2$-th root of 1 since $\omega^n=(\omega^2)^{n/2}=1$. Furthermore, $\omega^{n/2}=-1$ which is easily verified by $(-1)^2=(\omega^{n/2})^2=\omega^n=1$.

We now return to the problem of evaluating our odd and even sum polynomial $p(x)=s(y)+xt(y)$, where $y=x^2$, but at the n distinct points $x=\omega^j$ for $0\leq j\leq n-1$. Then $p(\omega^j)=s(\omega^{2j})+\omega^j t(\omega^{2j})$ and $p(\omega^{j+n/2})=s(\omega^{2j})-\omega^j t(\omega^{2j})$, since $\omega^{j+n/2}=\omega^j\omega^{n/2}=-\omega^j$ and $(\omega^{j+n/2})^2=(-\omega^j)^2=\omega^{2j}$. These formulas reveal how the problem of evaluating the polynomial $p(x)$ at n points can be divided into two sub-problems which involve evaluating polynomials of half the original size at half as many points. The subproblems are the evaluation of s and t, both having $n/2$ coefficients, at the points $\omega^{2j}=(\omega^j)^2$ for $0\leq j\leq n/2-1$, the primitive $n/2$-th roots of unity.

This strategy for splitting the problem can be applied repeatedly until we eventually arrive at the trivial problem of evaluating a constant polynomial. The total number of scalar arithmetic operations performed is governed by the recurrence relation $T(n)=2T(n/2)+cn$, where the last term represents one addition and one multiplication for each point x_i. (The number of multiplications can be cut in half by realizing that $x_{j+n/2}=-x_j$.) Since the roots of unity are in general complex numbers, several scalar operations will be needed for each arithmetic operation as written. The solution to the recurrence, with boundary condition $T(1)=0$, is given by $T(n)=cn\log_2 n$ for n a power of two.

The algorithm described yields an $O(n\log n)$ algorithm for polynomial multiplication, the asymptotically best method known. It is still an open

question whether the technique is optimal since the best lower bounds to date are of order n. Such lower bounds are not surprising since solving the problem involves processing $2n$ inputs, each of which must be used at least once.

The algebraic transform serving as the basis of the algorithm is the well-known fast Fourier transform, or FFT. The FFT traces its origins to the German mathematicians Carl Runge and H. König in the 1920's. G. C. Danielson and Cornelius Lanczos (1942) and Irving J. Good (1958) were other early contributors. A fundamental paper by James W. Cooley and John W. Tukey in 1965 clarified the technique and led to its widespread use. The recursive formulation of the algorithm described here is due to Allan Borodin and Ian Munro. Several other researchers, including Charles M. Fiduccia, Ellis Horowitz, John D. Lipson and Robert Moenck, have also made substantial contributions in the area to provide an interesting and coherent view of the relationship between evaluation, interpolation, and modular arithmetic. The FFT, itself, is utilized in many fields of science and engineering, perhaps most notably in signal processing applications such as communications, and speech and image processing.

Polynomial multiplication finds a useful analog in the problem of forming the product of two n-digit numbers. In fact the divide-and-conquer polynomial multiplication algorithm using $O(n^{1.59})$ operations, which we considered earlier, is based on a technique described by the Russian mathematicians A. Karatsuba and Yu. Ofman in 1962 for the digit product problem. In 1971 the German mathematicians Arnold Schönhage and Volker Strassen applied the FFT to produce an algorithm using $O(n \log n \log \log n)$ digitwise operations to multiply two n-digit numbers.

Matrix Multiplication

One final problem we will investigate is that of multiplying two square matrices. This problem and several related operations arise frequently in scientific applications of computers. An efficient algorithm for matrix multiplication can be used, for example, to obtain fast algorithms for inverting a matrix, finding its determinant, and solving systems of simultaneous linear equations.

Suppose we are given two $n \times n$ matrices A and B. We will denote the n^2 elements of each of these matrices by a_{ij} and b_{ij}, where i and j range between 1 and n. The result of multiplying these two matrices together is another $n \times n$ matrix $C = A \cdot B$, whose entries are given by the formula $c_{ij} = \sum_{k=1}^{n} a_{ik} b_{kj}$. Note that i and j remain constant in the sum, while k ranges over all n values.

The standard method for multiplying two matrices is to apply directly the above formula n^2 times. Since the product $a_{ik} b_{kj}$ is used in the computation of exactly one entry, c_{ij}, no overlapping of operations is possible. Observe that n multiplications and $n-1$ additions are used to calculate each entry, and thus a total of $n^2 \cdot n = n^3$ multiplications and $n^2(n-1) = n^3 - n^2$ additions are used overall.

Thus the standard algorithm uses 8 multiplications and 4 additions to compute the product of two 2 x 2 matrices. In 1969 Volker Strassen of the University of Zurich showed, surprisingly, that only 7 multiplications were required. Strassen's scheme, which is given in Figure 14, trades one multiplication at the expense of 14 extra add/sub steps. The key point, however, is that the method does not make use of the commutativity of multiplication, and hence can be used as the basis of a divide-and-conquer algorithm for multiplying larger size matrices.

For example, suppose we want to find the product of two 4 x 4 matrices A and B. Each of these matrices can be partitioned into four 2 x 2 submatrices, as illustrated in Figure 15. Thus we can regard both A and B as 2 x 2 matrices whose entries are themselves 2 x 2 matrices. Applying Strassen's algorithm recursively to obtain the product $C=A \cdot B$, we first form 7 products of 2 x 2 matrices. Since each of these products can be calculated with 7 scalar multiplications, $7 \cdot 7 = 49$ multiplications are used overall. This represents a considerable improvement over the $4^3 = 64$ multiplications employed in the standard algorithm!

Let us now examine how Strassen's scheme can be applied to obtain a fast method for multiplying two square matrices of any size n. For simplicity we will assume that n is a power of two, although this restriction is not essential. To multiply two n x n matrices, we first partition both of the matrices into four $n/2$ x $n/2$ submatrices. The product of the original n x n matrices can be formed using Strassen's scheme by computing the product of 7 square matrices of size $n/2$. To find these products we can apply the technique once again.

We now examine the efficiency of Strassen's algorithm. Let $M(n)$ denote the number of scalar multiplications used in computing the product of two n x n matrices. Since this product can be reduced to the problem of forming 7 products of $n/2$ x $n/2$ matrices, we have $M(n) = 7M(n/2)$. The solution to this recurrence relation, with initial condition $M(1) = 1$, is $M(n) = 7^{\log_2 n} = n^{\log_2 7}$. This result is easily obtained by back-substitution: $M(n) = 7M(n/2) = 7^2 M(n/4) = \ldots = 7^k M(n/2^k) = \ldots = 7^{\log_2 n} M(1)$. Since $\log_2 7 \approx 2.81$, we have that Strassen's algorithm uses $n^{2.81}$ multiplications, instead of the usual n^3, to multiply two n x n matrices.

What about the number of add/sub steps? For the 2 x 2 case, the standard method uses only 4 additions, while Strassen's scheme employs 18. Robert L. Probert of the University of Saskatchewan showed in 1973 how to reduce this

number to 15 and, moreover, proved that 15 additions were necessary in any scheme using only 7 multiplications. It appears that although Strassen's method may save on multiplications, this savings will be more than offset by the extra cost of the seemingly large number of additions. Surprisingly, we shall see that for sufficiently large matrices the number of additions is actually reduced!

Let $A(n)$ denote the number of scalar additions performed when Strassen's method is used to multiply two n x n matrices. Examination of the algorithm reveals that this quantity is equal to the number of additions performed in multiplying 7 matrices of size $n/2$ plus the scalar additions used in forming a (18 or 15) sums of $n/2$ x $n/2$ matrices. When adding two matrices we merely add the corresponding pairs of elements using scalar additions. Hence the recurrence relation describing the number of additions is $A(n)=7A(n/2)+a(n/2)^2$. The solution to this equation, with initial condition $A(1)=0$, is $A(n)=a/3 \cdot (n^{2 \cdot 81} - n^2)$.

We have just seen that both the number of multiplications and the number of additions performed by Strassen's algorithm are proportional to $n^{2 \cdot 81}$. For sufficiently large values of n, the value of any function proportional to $n^{2 \cdot 81}$ will be less than one growing as n^3. Hence Strassen's algorithm is asymptotically faster than the standard one. But when does it begin to pay to use Strassen's algorithm? Jacques Cohen and Martin Roth at Brandeis University have shown that the crossover point is at about $n=40$. Their results are based on timing experiments on an actual computer which take into account the added overhead incurred by more complex accessing of the data as well as the number of arithmetic operations performed.

The divide-and-conquer approach underlying Strassen's algorithm might be used to generate even faster matrix multiplication algorithms. Any non-

commutative scheme for finding the product of two m x m matrices with p
multiplications can serve as the basis of a method for multiplying matrices
of arbitrary size n with $O(n^{\log_m p})$ operations. Thus if we could show how
to multiply 2 x 2 matrices with 6 scalar multiplications, we could take the
product of arbitrary size matrices with $O(n^{\log_2 6}) \approx O(n^{2.58})$ operations.
Unfortunately, John E. Hopcroft and Leslie R. Kerr at Cornell University
and Shmuel Winograd of IBM showed independently in 1971 that this is impossible.
To date the best method for the 3 x 3 case uses 23 multiplications, while 21
would be needed to better Strassen's result.

Viktor Pan, working at the IBM Thomas J. Watson Research Center, has taken
an entirely different approach to the matrix multiplication problem. In 1979
he exhibited an algorithm using only $O(n^{2.61})$ operations, but with such a
serious increase in the constant of proportionality that the method would be
impractical to implement. To date all that is known is that at least on the
order of n^2 operations are needed. This is not surprising in view of the fact
that the input consists of $2n^2$ matrix elements, and all of the data must be
used at least once. Researchers are actively trying to bridge the gap between
the best upper and lower bounds for this problem.

Bibliography

The Art of Computer Programming (Vol. 2, Seminumerical Algorithms). Donald E.
   Knuth. Addison-Wesley Publishing Co., 1969.

The Design and Analysis of Computer Algorithms. Alfred V. Aho, John E. Hopcroft
   and Jeffrey D. Ullman. Addison-Wesley Publishing Co., 1974.

The Computational Complexity of Algebraic and Numeric Problems. Allan Borodin
   and Ian Munro. American Elsevier Publishing Co., 1975.

Fundamentals of Computer Algorithms. Ellis Horowitz and Sartaj Sahni. Computer
   Science Press, 1978.

List of Figures

Input data

D1. a D2. b D3. c D4. d

Computation steps

S1. D1 x D3 = ac S1. D1 + D2 = $a+b$

S2. D2 x D4 = bd S2. D3 + D4 = $c+d$

S3. D1 x D4 = ad S3. S1 x S2 = $ac+ad+bc+bd$

S4. D2 x D3 = bc S4. D1 x D3 = ac

S5. S1 - S2 = $ac-bd$ (real)   S5. D2 x D4 = bd

S6. S3 + S4 = $ad+bc$ (imaginary) S6. S4 - S5 = $ac-bd$ (real)

      Classical method   S7. S3 - S4 = $ad+bc+bd$

                                  S8. S7 - S5 = $ad+bc$ (imaginary)

                                        Three multiplication method

Figure 1. Two methods for forming the product of the complex numbers
$(a+bi)(c+di)=(ac-bd)+(ad+bc)i$. The classical method uses 4 multiplications
and 2 additions/subtractions, while the problem can also be solved with 3
multiplications and 5 additions. If M and A denote the time required to
perform a single multiplication and addition, respectively, the second method
is faster if $3M+5A < 4M+2A$, or $M/A > 3$.

Input data

    D1. x

Constants

    C1. 2

    C2. 4

    C3. 5

Computation steps

    S1. D1 x D1 = x^2

    S2. D1 x S1 = x^3

    S3. C2 x S1 = $4x^2$

    S4. C3 x D1 = $5x$

    S5. S2 + S3 = x^3+4x^2

    S6. S5 + S4 = x^3+4x^2+5x

    S7. S6 + C1 = x^3+4x^2+5x+2

Input data

    D1. x

Constants

    C1. 1

    C2. 2

Computation steps

    S1. D1 + C1 = $x+1$

    S2. D1 + C2 = $x+2$

    S3. S1 x S1 = x^2+2x+1

    S4. S2 x S3 = x^3+4x^2+5x+2

Figure 2. A straight-line algorithm consists of a series of computation steps in which an arithmetic operation is applied to either the input data, constants, or the results of prior computation steps. Two algorithms for computing $p(x)=x^3+4x^2+5x+2$ are shown. The first applies the formula directly using 4 multiplications and 3 additions. The second, which takes advantage of the fact that $p(x)$ can be factored as $p(x)=(x+1)^2(x+2)$, uses only 2 multiplications and 2 additions.

Input data

D1. x

Computation steps

Brute force method	Binary method	Factor method	Power tree method
S1. D1 x D1 = x^2	S1. D1 x D1 = x^2 S	S1. D1 x D1 = x^2 y	S1. D1 x D1 = x^2
S2. S1 x D1 = x^3	S2. S1 x S1 = x^4 S	S2. S1 x S1 = x^4 $y^2=z$	S2. S1 x D1 = x^3
S3. S2 x D1 = x^4	S3. S2 x D1 = x^5 X	S3. S2 x S2 = x^8 z^2	S3. S2 x S1 = x^5
.	S4. S3 x S3 = x^{10} S	S4. S3 x S3 = x^{16} $(z^2)^2$	S4. S3 x S3 = x^{10}
.			
S21. S20 x D1 = x^{22}	S5. S4 x D1 x^{11} X	S5. S4 x S2 = x^{20} $z^5=y^{10}$	S5. S4 x S4 = x^{20}
S22. S21 x D1 = x^{23}	S6. S5 x S5 = x^{22} S	S6. S5 x S1 = x^{22} $y^{10} \cdot y$	S6. S5 x S2 = x^{23}
	S7. S6 x D1 = x^{23} X	S7. S6 x D1 = x^{23} $y^{11} \cdot x$	

Figure 3. Comparison of several algorithms for x^{23}. The brute force method uses 22 multiplications, the binary and factor methods 7, and the power tree only 6. In the binary method, $23=(10111)_2$ gives rise to the computation sequence $SSXSXSX$ where S means "square the result of the previous step" and X means "multiply the result of the previous step by X". In the factor method, $x^{23}=x^{22} \cdot x=(x^2)^{11} \cdot x$; letting $y=x^2$, $y^{11}=y^{10} \cdot y=(y^2)^5 \cdot y$; letting $z=y^2$, $z^5=(z^2)^2 \cdot z$.

Figure 4. Optimal power tree giving the minimal number of multiplications required to compute x^n for $n \le 100$. To compute x^n, first locate n in the tree. The numbers on the path from the root of the tree to n represent the powers to be taken in an optimal computation of x^n. For example, the minimal multiplication sequence for computing x^{31} involves successively computing $x^2, x^3, x^5, x^{10}, x^{11}, x^{21}, x^{31}$.

Input data

D1. x D2. c_0 D3. c_1 D4. c_2 D5. c_3

Computation steps

$\left.\begin{array}{l} \text{S1.} \quad \text{D1} \times \text{D1} = x^2 \\ \text{S2.} \quad \text{S1} \times \text{D1} = x^3 \end{array}\right\}$ compute powers

$\left.\begin{array}{l} \text{S3.} \quad \text{D3} \times \text{D1} = c_1 x \\ \text{S4.} \quad \text{D4} \times \text{S1} = c_2 x^2 \\ \text{S5.} \quad \text{D5} \times \text{S2} = c_3 x^3 \end{array}\right\}$ multiply by coefficients

$\left.\begin{array}{l} \text{S6.} \quad \text{S5} + \text{S4} = c_3 x^3 + c_2 x^2 \\ \text{S7.} \quad \text{S6} + \text{S3} = c_3 x^3 + c_2 x^2 + c_1 x \\ \text{S8.} \quad \text{S7} + \text{D2} = c_3 x^3 + c_2 x^2 + c_1 x + c_0 \end{array}\right\}$ sum terms

Usual method

S1. $\text{D5} \times \text{D1} = c_3 x$

S2. $\text{S1} + \text{D4} = c_3 x + c_2$

S3. $\text{S2} \times \text{D1} = c_3 x^2 + c_2 x$

S4. $\text{S3} + \text{D3} = c_3 x^2 + c_2 x + c_1$

S5. $\text{S4} \times \text{D1} = c_3 x^3 + c_2 x^2 + c_1 x$

S6. $\text{S5} + \text{D2} = c_3 x^3 + c_2 x^2 + c_1 x + c_0$

Horner's method

Figure 5. Two methods for evaluating a general third degree polynomial $p(x) = c_3 x^3 + c_2 x^2 + c_1 x + c_0$. The usual method is to first compute the powers of x: x^2, x^3; then multiply the powers by the appropriate coefficients: $c_1 x$, $c_2 x^2$, $c_3 x^3$; and finally to sum the terms. Horner's method uses repeated factoring to evaluate $p(x)$ as $((c_3 x + c_2)x + c_1)x + c_0$. When evaluating an nth degree polynomial, the usual method performs $2n-1$ multiplications and n additions, while Horner's method employs only n operations of each type.
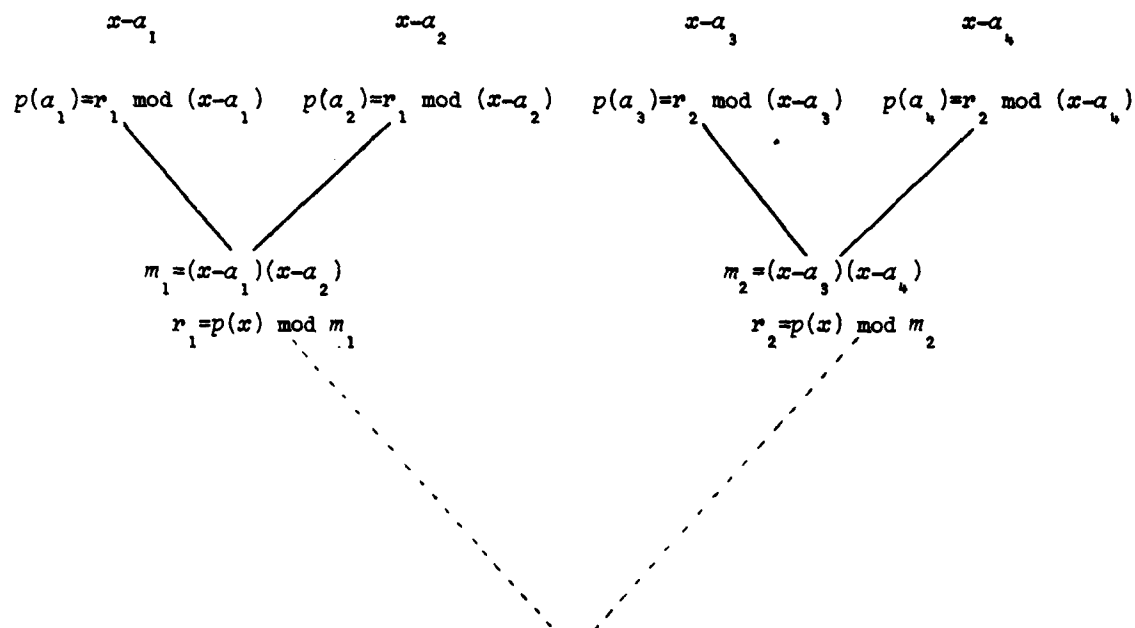
$$x-a_1 \qquad x-a_2 \qquad x-a_3 \qquad x-a_4$$

$$p(a_1)=r_1 \bmod (x-a_1) \quad p(a_2)=r_1 \bmod (x-a_2) \qquad p(a_3)=r_2 \bmod (x-a_3) \quad p(a_4)=r_2 \bmod (x-a_4)$$

$$m_1=(x-a_1)(x-a_2) \qquad\qquad m_2=(x-a_3)(x-a_4)$$

$$r_1=p(x) \bmod m_1 \qquad\qquad r_2=p(x) \bmod m_2$$

Figure 6. Tree showing the products and remainders to be computed when evaluating a polynomial $p(x)$ of degree ≥ 3 at the 4 points $x=a_1, a_2, a_3, a_4$ using the modular transform method. First the products m_i called "moduli", are built up in the manner shown moving from the top of the tree downward. Then the divisions of $p(x)$ and the subsequent remainders indicated are computed in the reverse order, going up the tree. ($r = s \bmod m$ means that r is the remainder when s is divided by m.) The overall running time of the algorithm is driven by the time to symbolically multiply and divide polynomials.

$$\begin{array}{r} 5x^3+2x^2\quad\ +3 \\ +\ 2x^3\qquad -6x\ -1 \\ \hline 7x^3+2x^2-6x\ +2 \end{array} \qquad\qquad \begin{array}{r} 5x^3+2x^2\quad\ +3 \\ -\ 2x^3\qquad -6x\ -1 \\ \hline 3x^3+2x^2+6x\ +4 \end{array}$$

Figure 7. Polynomial addition and subtraction. Two polynomials are added (or subtracted) by adding (or subtracting) the coefficients of the corresponding powers of the variable x. The sum and difference of $5x^3+2x^2+3$ and $2x^3-6x-1$ are shown.

$$
\begin{array}{lr}
5x^3 + 2x^2 \quad\quad +3 & \text{multiplicand} \\
\text{X} \quad 2x^3 \quad\quad -6x \ -1 & \text{multiplier} \\
\hline
-5x^3 - 2x^2 \quad\quad -3 & \text{(-1 times } 5x^3 + 2x^2 + 3) \\
-30x^4 - 12x^3 \quad\quad -18x & \text{(-6x times } 5x^3 + 2x^2 + 3) \\
10x^6 + 4x^5 \quad\quad 6x^3 & \text{(2x}^3 \text{ times } 5x^3 + 2x^2 + 3) \\
\hline
10x^6 + 4x^5 - 30x^4 - 11x^3 - 2x^2 - 18x - 3 & \text{product}
\end{array}
$$

Figure 8. Polynomial multiplication. The classical way to take the product
of two polynomials is to multiply each term in the multiplier by the multiplicand
and then sum the results. The product of $5x^3 + 2x^2 + 3$ and $2x^3 - 6x - 1$ is illustrated.

$$
\begin{array}{r}
2x\ -1 \\
3x^2-x+2\ \overline{\smash{)}\ 6x^3-5x^2+9x+3} \\
6x^3-2x^2+4x \\
\hline
-3x^2+5x+3 \\
-3x^2+\ x-2 \\
\hline
\ \ 4x+5
\end{array}
$$

divisor $\quad 3x^2-x+2$

remainder $\quad 4x+5$

quotient
dividend
($2x$ times $3x^2-x+2$)

($6x^3-5x^2+9x+3$ less $6x^3-2x^2+4x$)
(-1 times $3x^2-x+2$)

($-3x^2+5x+3$ less $-3x^2+x-2$)

Figure 9. Synthetic division of polynomials is similar to long division of two integers. First the high order terms of the dividend and divisor are divided, this result is multiplied by the entire divisor, and the resultant product is subtracted from the entire dividend to yield a trial remainder. (In the example shown, $3x^2$ into $6x^3$ is $2x$, $2x$ times $3x^2-x+2$ is $6x^3-2x^2+4x$, and $6x^3-5x^2+9x+3$ less $6x^3-2x^2+4x$ yields a trial remainder of $-3x^2+5x+3$.) The entire process is then repeated with the trial remainder in the role that the dividend played initially. ($3x^2$ into $-3x^2$ is -1, -1 times $3x^2-x+2$ is $-3x^2+x-2$, $-3x^2+5x+3$ less $-3x^2+x-2$ is $4x+5$.) The procedure continues until a trial remainder of degree less than the divisor is obtained.

$$p(x) = a_1 x + a_0$$
$$q(x) = b_1 x + b_0$$

$$p(x) \cdot q(x) = a_1 b_1 x^2 + (a_1 b_0 + a_0 b_1) x + a_0 b_0$$

Input data

   D1. a_1 D2. a_0 D3. b_1 D4. b_0

Computation steps

   S1. D2 x D4 $= a_0 b_0$ (constant term)   S1. D2 x D4 $= a_0 b_0$ (constant term)

   S2. D1 x D4 $= a_1 b_0$ S2. D1 x D3 $= a_1 b_1$ (coeff. of x^2)

   S3. D2 x D3 $= a_0 b_1$ S3. D1 + D2 $= a_1 + a_0$

   S4. S2 + S3 $= a_1 b_0 + a_0 b_1$ (coeff. of x)   S4. D3 + D4 $= b_1 + b_0$

   S5. D1 x D3 $= a_1 b_1$ (coeff. of x^2)   S5. S3 x S4 $= a_1 b_1 + a_1 b_0 + a_0 b_1 + a_0 b_0$

          Usual method   S6. S5 - S2 $= a_1 b_0 + a_0 b_1 + a_0 b_0$

                                      S7. S6 - S1 $= a_1 b_0 + a_0 b_1$ (coeff. of x)

                                      Three multiplication method

Figure 10. Two algorithms for multiplying first degree (i.e., 2-coefficient) polynomials. The usual method uses 4 scalar multiplications, while the product can be formed with only 3 scalar multiplications by using extra additions and subtractions.

1. Split each input polynomial having 4 coefficients into an upper and lower half with 2 coefficients.

$$p(x) = a_3 x^3 + a_2 x^2 + a_1 x + a_0 \qquad q(x) = b_3 x^3 + b_2 x^2 + b_1 x + b_0$$

$$= s(x) \cdot x^2 + t(x) \qquad\qquad = u(x) \cdot x^2 + v(x)$$

where $s(x) = a_3 x + a_2$ $\qquad$ where $u(x) = b_3 x + b_2$

$\qquad\quad t(x) = a_1 x + a_0$ $\qquad\qquad\quad v(x) = b_1 x + b_0$

2. Multiply the polynomials $p(x)$ and $q(x)$ by forming 3 products of 2-coefficient polynomials. Each of these products can be computed with 3 multiplications, for a total of $3 \cdot 3 = 9$ scalar multiplications.

$$m_1 = t \cdot v \qquad\qquad D_0 = m_1 = tv = d_{0,2} x^2 + d_{0,1} x + d_{0,0}$$

$$m_2 = s \cdot u \qquad\qquad D_1 = m_3 - m_2 - m_1 = sv + tu = d_{1,2} x^2 + d_{1,1} x + d_{1,0}$$

$$m_3 = (s+t) \cdot (u+v) \qquad D_2 = m_2 = su = d_{2,2} x^2 + d_{2,1} x + d_{2,0}$$

where $d_{0,0} = a_0 b_0$ $\qquad\quad$ $d_{1,0} = a_2 b_0 + a_0 b_2$ $\qquad\quad$ $d_{2,0} = a_2 b_2$

$\qquad\quad d_{0,1} = a_1 b_0 + a_0 b_1$ $\quad$ $d_{1,1} = a_3 b_0 + a_2 b_1 + a_1 b_2 + a_0 b_3$ $\quad$ $d_{2,1} = a_3 b_2 + a_2 b_3$

$\qquad\quad d_{0,2} = a_1 b_1$ $\qquad\qquad$ $d_{1,2} = a_3 b_1 + a_1 b_3$ $\qquad\qquad$ $d_{2,2} = a_3 b_3$

3. The desired product $p(x) \cdot q(x)$ may be expressed as follows.

$$p(x) \cdot q(x) = D_2 x^4 + D_1 x^2 + D_0$$

$$= (d_{2,2} x^2 + d_{2,1} x + d_{2,0}) x^4 + (d_{1,2} x^2 + d_{1,1} x + d_{1,0}) x^2 + (d_{0,2} x^2 + d_{0,1} x + d_{0,0})$$

$$= d_{2,2} x^6 + d_{2,1} x^5 + (d_{2,0} + d_{1,2}) x^4 + d_{1,1} x^3 + (d_{1,0} + d_{0,2}) x^2 + d_{0,1} x + d_{0,0}$$

$$= c_6 x^6 + c_5 x^5 + c_4 x^4 + c_3 x^3 + c_2 x^2 + c_1 x + c_0$$

where $c_0 = d_{0,0} = a_0 b_0$

$\qquad\quad c_1 = d_{0,1} = a_1 b_0 + a_0 b_1$

$\qquad\quad c_2 = d_{1,0} + d_{0,2} = a_2 b_0 + a_1 b_1 + a_0 b_2$

$\qquad\quad c_3 = d_{1,1} = a_3 b_0 + a_2 b_1 + a_1 b_2 + a_0 b_3$

$\qquad\quad c_4 = d_{2,0} + d_{1,2} = a_3 b_1 + a_2 b_2 + a_1 b_3$

$\qquad\quad c_5 = d_{2,1} = a_3 b_2 + a_2 b_3$

$\qquad\quad c_6 = d_{2,2} = a_3 b_3$

Figure 11. Divide-and-conquer polynomial multiplication algorithm applied to two general 4-coefficient(degree 3) polynomials $p(x)$ and $q(x)$. Because two 2-coefficient polynomials can be multiplied taking 3 products instead of the usual 4, $p(x)$ and $q(x)$ can be split into two 2-coefficient polynomials and multiplied with $3 \cdot 3 = 9$ scalar products. The classical method would have employed 16 products. In general the divide-and-conquer approach leads to an algorithm using $O(n^{1.58})$ arithmetic operations overall, while the usual method is $O(n^2)$.
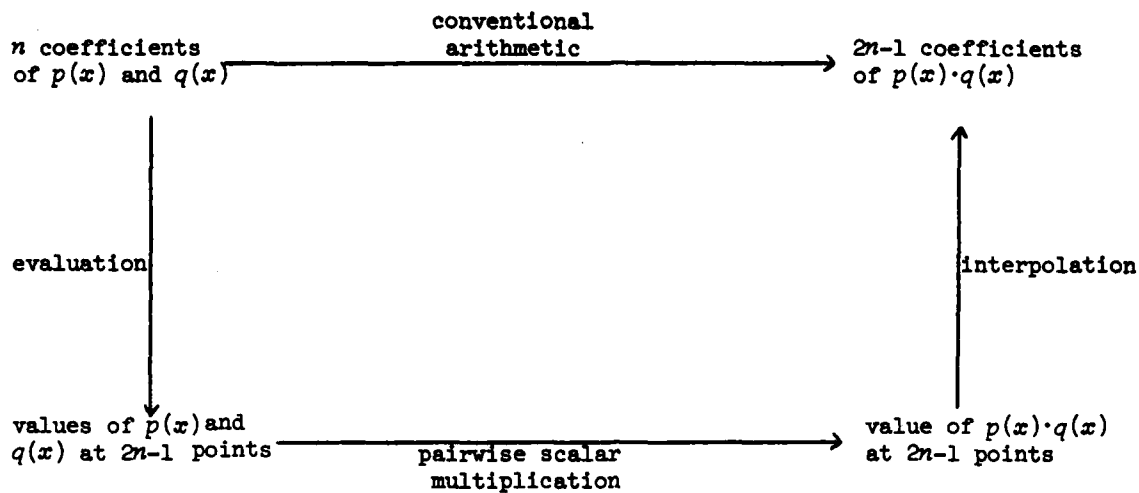
```
n coefficients          conventional              2n-1 coefficients
of p(x) and q(x)         arithmetic                of p(x)·q(x)



evaluation                                         interpolation



values of p(x) and                                 value of p(x)·q(x)
q(x) at 2n-1 points      pairwise scalar           at 2n-1 points
                         multiplication
```

Figure 12. Algebraic transform for the product of two n-coefficient polynomials

$p(x) \cdot q(x)$. The classical algorithm obtains the coefficients of the product

polynomial directly using conventional arithmetic. In the transform method

$p(x)$ and $q(x)$ are both evaluated at $2n-1$ points, and their values at corresponding

points are multiplied together. This yields the value of $p(x) \cdot q(x)$ at $2n-1$ points.

The coefficients of the product polynomial are obtained via interpolation since

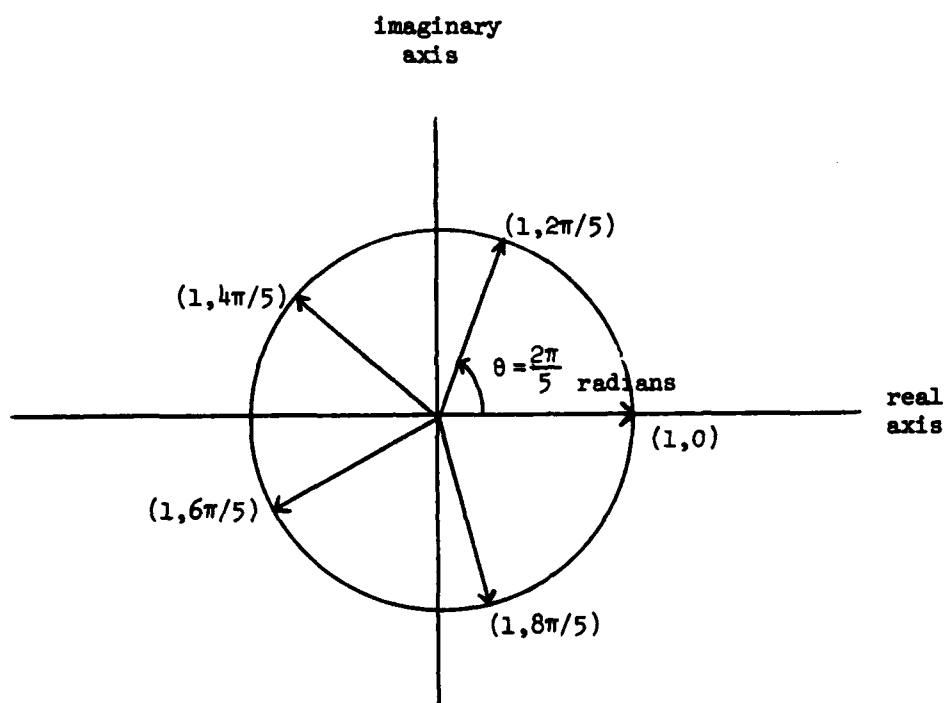there is a unique polynomial with $2n-1$ coefficients which fits the points.

Figure 13. A complex number $a+bi$ can be represented by a vector in a plane using the real and imaginary parts of the number for Cartesian coordinates. Alternatively, a number can be represented on the same grid by giving its polar coordinates (r,θ) where $r=\sqrt{a^2+b^2}$ and $\theta=\tan^{-1} b/a$. The polynomial x^n-1 has n roots, called the n-th principal roots of unity. Geometrically, the vectors representing these numbers slice the unit circle into n equal pie-shaped pieces. The polar coordinates of the fifth roots of unity are shown.

2 x 2 matrix-matrix product: $A \cdot B = C$

$$\begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix} \cdot \begin{bmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{bmatrix} = \begin{bmatrix} c_{11} & c_{12} \\ c_{21} & c_{22} \end{bmatrix}$$

Usual method: 8 mult, 4 add/sub

$$c_{11} = a_{11}b_{11} + a_{12}b_{21} \qquad\qquad c_{12} = a_{11}b_{12} + a_{12}b_{22}$$

$$c_{21} = a_{21}b_{11} + a_{22}b_{21} \qquad\qquad c_{22} = a_{21}b_{12} + a_{22}b_{22}$$

Strassen's method: 7 mult, 18 add/sub

$$m_1 = (a_{12} - a_{22})(b_{21} + b_{22}) \qquad\qquad c_{11} = m_1 + m_2 - m_4 + m_6$$

$$m_2 = (a_{11} + a_{22})(b_{11} + b_{22}) \qquad\qquad c_{12} = m_4 + m_5$$

$$m_3 = (a_{11} - a_{21})(b_{11} + b_{12}) \qquad\qquad c_{21} = m_6 + m_7$$

$$m_4 = (a_{11} + a_{12})b_{22} \qquad\qquad c_{22} = m_2 - m_3 + m_5 - m_7$$

$$m_5 = a_{11}(b_{12} - b_{22})$$

$$m_6 = a_{22}(b_{21} - b_{11})$$

$$m_7 = (a_{21} + a_{22})b_{11}$$

Figure 14. The usual method for multiplying two 2 x 2 matrices involves 8 multiplications and 4 additions. In 1969 Volker Strassen showed how the number of multiplications could be reduced to 7 by using 18 additions/subtractions.

4 x 4 matrix-matrix product: $A \cdot B = C$

$$\left[\begin{array}{cc|cc} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \\ \hline a_{31} & a_{32} & a_{33} & a_{34} \\ a_{41} & a_{42} & a_{43} & a_{44} \end{array}\right] \cdot \left[\begin{array}{cc|cc} b_{11} & b_{12} & b_{13} & b_{14} \\ b_{21} & b_{22} & b_{23} & b_{24} \\ \hline b_{31} & b_{32} & b_{33} & b_{34} \\ b_{41} & b_{42} & b_{43} & b_{44} \end{array}\right] = \left[\begin{array}{cc|cc} c_{11} & c_{12} & c_{13} & c_{14} \\ c_{21} & c_{22} & c_{23} & c_{24} \\ \hline c_{31} & c_{32} & c_{33} & c_{34} \\ c_{41} & c_{42} & c_{43} & c_{44} \end{array}\right]$$

1. Partition A, B, C into four 2 x 2 submatrices.

$$\left[\begin{array}{c|c} A_{11} & A_{12} \\ \hline A_{21} & A_{22} \end{array}\right] \cdot \left[\begin{array}{c|c} B_{11} & B_{12} \\ \hline B_{21} & B_{22} \end{array}\right] = \left[\begin{array}{c|c} C_{11} & C_{12} \\ \hline C_{21} & C_{22} \end{array}\right]$$

2. Apply Strassen's scheme to the submatrices.

$M_1 = (A_{12} - A_{22})(B_{21} + B_{22})$ $C_{11} = M_1 + M_2 - M_4 + M_6$

$M_2 = (A_{11} + A_{22})(B_{11} + B_{22})$ $C_{12} = M_4 + M_5$

$M_3 = (A_{11} - A_{21})(B_{11} + B_{12})$ $C_{21} = M_6 + M_7$

$M_4 = (A_{11} + A_{12})B_{22}$ $C_{22} = M_2 - M_3 + M_5 - M_7$

$M_5 = A_{11}(B_{12} - B_{22})$

$M_6 = A_{22}(B_{21} - B_{11})$

$M_7 = (A_{21} + A_{22})B_{11}$

Figure 15. Strassen's method can be extended to larger matrices using a divide-and-conquer strategy. In multiplying two 4 x 4 matrices, the 7 products of 2 x 2 matrices indicated are taken. Since each 2 x 2 matrix product can be formed with 7 scalar multiplications, only $7 \cdot 7 = 49$ scalar products (instead of the usual 64) are needed. In general, two n x n matrices can be multiplied using $O(n^{2 \cdot 81})$ arithmetic operations (both multiplications and additions/subtractions), instead of the usual $O(n^3)$, via this technique.

ALGORITHMIC COMPLEXITY
Part 4

by
Lyle A. Anderson

SYSTEMATIC ANALYSIS OF ALGORITHMS

ABSTRACT

The limits and methods involved in the systematic analysis
of algorithms are explored. A review of the existing work
in this field is presented. A specific method of systematic
analysis is developed. The method consists of (1) the
translation of algorithm loop structures into recursive
subroutines and recursive subroutine references, and (2) the
semantic manipulation of expressions representing the joint
probability distribution function of the program variables.
A new delta function is introduced to describe the effects
of conditional statements on the joint probability density
function of the program variables. The method is applied to
several simple algorithms, sorting and searching algorithms,
and a tree insertion/deletion algorithm.

ACKNOWLEDGEMENT

TABLE OF CONTENTS

TABLE OF CONTENTS
(Continued)

CHAPTER 1

INTRODUCTION

This chapter is divided into two parts. In the first part we will state and discuss the problem in computer science that will be addressed in the rest of the thesis. In the second part we will give an overview of the remaining chapters of the thesis.

Statement of the Problem

This thesis is concerned with the systematic analysis of algorithms. In order to understand what it is about, we must answer these three questions:

1. What are algorithms?

2. What is the analysis of algorithms?

3. What is the systematic analysis of algorithms?

We will also be discussing a fourth question:

4. What are the limits of systematic analysis?

This will involve a short discussion of:

a. Godel's Theorem

b. The Halting Problem

c. Characteristics of the Completeness Problem

What are Algorithms?

Horowitz and Sahni [7] give this definition of an algorithm: "Algorithm has come to refer to a precise method useable by a computer for the solution of a problem." In order to be considered an algorithm the method must have the following characteristics:

1. A finite number of steps of one or more operations

2. Each operation must be definite, i.e. unambigously defined as to what must be done

3. Each operation must be effective, i.e. a person with pencil and paper or a Turing Machine must be able to perform each operation in a finite amount of time

4. Produce at least one output

5. Accept zero or more inputs

6. Terminate after a finite number of operations

What is the Analysis of Algorithms?

Webster's New Collegiate Dictionary defines analysis as "an examination of a complex, its elements, and their relations". In the analysis of an algorithm we are interested in the relationship between characteristics of the inputs and the performance characteristics of the algorithm. Foremost among these characteristics is the execution time of the algorithm; that is, the relationship between some sizing parameter of the input data and the amount of time it takes

for the algorithm to get an answer. Other performance parameters of interest include:

1. Number of comparisons in sorting/searching algorithms

2. Number of scalar multiplications/divisions in algebraic algorithms, such as matrix-matrix product

3. Number of input/output operations required for problems dealing with database access

4. Size of the computer memory required to solve a problem

All of these performance parameters have one thing in common. They all can be transformed into the cost of computing the answer. This is the reason that the analysis of algorithms is so important. Aside from its intellectual and recreational aspects, the economic aspects of the analysis of algorithms are important to the users of computer systems. Especially in the computer-based industries, time is money. An algorithm which takes twice as long to run may not only cost twice as much to run, but may not even get done in time to be useful. In other applications, accurate predictions of probable running times are needed before a system is actually built. These predictions can help make overall cost and feasibility estimates for a proposed system more accurate. In these kinds of applications the analysis of algorithms is a software engineering tool. Other potential uses are in automatic program synthesizers or in

compiler systems for very high-level languages. [1]

In most cases the analysis of an algorithm consists of determining the time behavior of the algorithm. This is not the only measure of a program for which an analysis can be performed. An algorithm can be analyzed by "instrumenting" it, meaning that the values of the parameter of interest are recorded in a counter variable which is added to the algorithm. We often do this when analyzing for the time behavior of an algorithm. For this reason the analysis of different measures have a great deal in common with the analysis of time behavior. When we talk about the analysis of an algorithm, we will only be concerned with its time behavior unless otherwise stated.

What is the Systematic Analysis of Algorithms?

There are two basic ways to approach the analysis of algorithms. The first way is to approach each alogrithm as a separate new problem and to find the solution by appealing to previous experience with similar problems. The second way is to make up general rules which apply to "all" algorithms and to apply these rules step by step to the algorithm being studied.

The first way is very suitable to humans who come equipped with a great deal of problem-solving and pattern-recognition ability. It is not so well suited to the digital computers of today because they are not so equipped.

4-4

The more systematic approach of the second way to analyze
algorithms is better suited to implementation by digital
computers. We shall say that the human approach involves ad
hoc procedures, and the computer approach involves
systematic procedures.

What are the limits of Systematic Analysis?

The gross limits of systematic or automatic algorithm
analysis are known.

1. We know that systems can be built which will analyze
   simple programs. [1,3,4]

2. We know that no completely automatic system or com-
   plete formal system can be constructed which can
   analyze all algorithms. This fact is firmly estab-
   lished by computability theory. [15]

In between the simple programs and all possible programs
there is a lot of ground which can be covered.

What We Can Do

Wegbreit [1] has built a system which can analyze
simple LISP programs automatically. Cohen and Zuckerman [3]
have built a system which greatly aids in the analysis of
algorithms written in an ALGOL-like programming language.
Their system helps the analyst with the details of the
analysis while requiring the analyst to provide the branch-
ing probabilities. Wegbreit [2] developed a formal system

4-5

for the verification of program performance. His technique
can also be used to provide the branching probabilities
which are needed. Recently, Ramshaw [5] has shown that
there are problems with Wegbreit's probabilistic approach
and has developed a formal system which he calls the
Frequency System. There are problems with the Frequency
System, which Ramshaw points out in his thesis [5]. We will
show that some of the problems in the Frequency System can
be overcome.

What We Cannot Do

Douglas R. Hofstadter [15] gives a beautiful exposition
of the nature of the whole question of computability and
decidability and the wide-ranging and unexpected topics upon
which it touches. The formal study of this subject springs
from Gödel's Theorem which Hofstadter paraphrases:

"All consistent axiomatic formulations of number
theory include undecidable propositions."

The undecidability of the Halting Problem is an example
of one such "undecidable proposition." Stated in terms of a
Turing Machine, the Halting Problem is this:

Can one construct a Turing Machine which can decide
whether any other Turing Machine will halt for any
input, when given an input tape containing a
description of the other Turing Machine and its
input?

A negative answer to this question was given in 1937 by
Alan Turing. The argument which he used is called a diagonal
method. This method was discovered by Georg Cantor, the

4-6

founder of set theory. It involves feeding a hypothetical Turing Machine, which could decide whether any other Turing Machine would halt for any input, a description of itself which has been modified in a particularly diabolical manner. Hofstadter's book [15] devotes much of its 740 pages to the variety of topics to which this method may be applied.

It appears to us that undecidability and incompleteness creep into formal systems when statements which can be interpreted as being about the system itself are allowed. In our discussions we will try to avoid these kinds of questions, and thereby the completeness problem.

Overview of the Thesis

We have chosen to organize this thesis along the lines which were taken in the development of the research upon which it is based. We feel that the road taken is interesting in and of itself. For this reason we will point out the "dead-ends" which periodically blocked our path.

The first step which we took was a survey of the work which had been done in this field. In Chapter 2, we will discuss the current state of the art of algorithm analysis. We will point out the areas where results are firmly established and the benefits of particular procedures that are known. We will examine some of the recent advances both to see how they work and to discover the kinds of problems which they cannot solve.

When this survey was completed we formulated a plan. The approach which we used was to start from the program statements themselves. We attempted to determine just how much could be learned from manipulations of the programs using various translation schema. We restricted ourselves to programs written in a "structured" language. SPARKS, developed by Horowitz and Sahni [7,9], was chosen as the language for representing algorithms for the same reasons they used it in their books.

Our initial work revealed a transformation which proved to be effective in analyzing several deterministic algorithms in a straight-forward manner. Chapter 3 describes this technique which involves the transformation of all looping structures of a program into a series of recursive subroutines and recursive subroutine calls. Because this process is designed to follow the syntax of the algorithm, we refer to this as a "syntax-directed translation." The program characteristic to be analyzed is selected, and the recursive program statements are transformed into recurrence equations. The analysis is done by solving the recurrence equations. This is not always easy [8]. For this reason we concerned ourselves with solving as well as setting up the recursions.

In Chapter 3, we will examine some very simple, deterministic algorithms (i.e. ones for which we know the inputs exactly), then some very simple probabilistic algorithms

(i.e. ones where we only know some characteristics of the inputs). While looking at these examples we will discover the "problem of the conditional statement." We started with the FINDMAX algorithm which was analyzed both by Knuth [6] and by Ramshaw [5]. We soon discovered that when the statistical behavior of algorithms is being analyzed, the distribution from which the input data is drawn is an important factor in the running time. While we could solve the problems relating to distributions in algorithms such as FINDMAX, we often found ourselves using information from "outside the system".

Chapter 4 presents our formal approach for handling the conditional statement. This approach is to use statements about the distributions of program variables directly in the analysis of the algorithms. We found that we had to study the propagation of the distributions of the program variables through the program. As a result, we developed a "calculus" for the behavior of the distributions themselves. We will use this method to analyze the probabilistic algorithms from Chapter 3.

We will then move on and apply the techniques to some sorting and searching algorithms in Chapter 5, and to a miscellaneous problem in Chapter 6. Chapter 7 is a summary of the work and an outline of possible future efforts.

Appendix A contains some details of the work discussed in Chapter 5.

CHAPTER 2

CURRENT STATE OF THE ART

In this chapter, we will discuss what is currently known about the analysis of algorithms. The chapter is divided into two sections. The first discusses what we call ad hoc procedures, and the second discusses current systematic approaches.

Ad Hoc Procedures

We are going to characterize an analysis technique as "ad hoc" if we cannot see a way to easily remove the "intuition" required to get the answers. The analysis procedures which are so categorized are more suited for use by humans than for the programming of a computer. They take advantage of the rich background of experience which forms the context of a human's ability to perform such analysis. We will present the techniques of three sets of researchers in order of increasing mathematical elegance of the techniques. A method with a high degree of elegance is very hard for the uninitiated to understand, but facilitates quick and meaningful communication between the initiated.

de Freitas and Lavelle

The most straight-forward, and hence the least elegant, way to analyze an algorithm is to write down how long each statement takes and to add up the result. S. L. de Freitas and P. J. Lavelle describe "A Method for the Time Analysis of Programs" [4] which does the first part of this procedure. Their method consists of superimposing timing data about the assembly/machine code produced by a FORTRAN program on the program source listing. The programmer may then use the timing information to identify inefficient portions of the program. The method does not calculate the repetition counts for loops, but presents the time required to perform one iteration of a loop. It therefore requires the application of all the ad hoc analysis techniques we will describe, but allows the analyst to come up with exact answers to time performance questions. Even though it uses a computer program, it can still be considered an ad hoc technique.

Aho, Hopcroft and Ullman
Horowitz and Sahni

Aho, Hopcroft and Ullman [10] and Horowitz and Sahni [7] describe a level of analysis which is one step removed from the machine dependent technique described above. This level deals with the statements of the algorithm as primitive entities and largely ignores the variation in execution

time between them. This type of analysis seeks order-of-magnitude or "Big O" performance data. In their excellent introductory text [7], Horowitz and Sahni are primarily interested in this kind of analysis. They introduce a methodology which is very close to the high level "code" of the algorithm to be analyzed. Aho, Hopcroft and Ullman [10] give an excellent presentation of the various computer and computability models which have been used.

Knuth's Analysis Techniques

It would be unfair to imply that Knuth's techniques are all ad hoc. Nothing can be further from the truth. Donald E. Knuth, perhaps more than anyone else, has established the definitions and directions of algorithmic analysis [6]. Jonassen and Knuth present an ad hoc tour de force in "A Trivial Algorithm Whose Analysis Isn't" [8]. In the beginning of his book [6], Knuth sets down the tools and techniques which may be brought to bear during the analysis of an algorithm. It is this grouping of techniques which we refer to as "ad hoc":

1. Mathematical Induction
2. Sums and Products
3. Elementary Number Theory and Integer Functions
4. Permutations and Factorials
5. Binomial Coefficients
6. Harmonic Numbers
7. Generating Functions
8. Euler's Summation Formula
9. Combinatorics

The application of these techniques requires a considerable amount of intuition and experience in the analysis of algorithms. The analyses which result are characterized by a high degree of abstraction.

Systematic Approaches

We now begin a discussion of systematic approaches to the analysis of algorithms. These methods are characterized by the exposition of a "theory" which is applied consistently in the analysis of algorithms. We will discuss three manual approaches in order of increasing effectiveness, and then discuss two automatic analyzers. The manual approaches which we will discuss are:

1. Electrical Network Analysis

2. Wegbreit's Probability System

3. Ramshaw's Frequentistic System

For each one we will cover the theoretical basis of the system, describe how it works, give an example, and discuss the inherent weaknesses and their causes.

Electrical Network Analysis

Knuth mentions the applicability of Kirchhoff's Current Law to the analysis of algorithms and applies it quite often [6]. He also mentions that Kirchhoff's Voltage Law is not applicable to the analysis of algorithms. An attempt to introduce Kirchhoff's Voltage Law into the analysis of

algorithms was proposed by Kodres [13] and extended by Davies. The following section closely follows Davies [14]. A generalization of Kirchhoff's Voltage and Current Laws is applied to the analysis of program or algorithm flowcharts in the following way:

1. the number of executions of a statement corresponds to the current in an electrical circuit

2. the execution time of a statement corresponds to the resistance of a circuit element

3. the total time spent executing the statement corresponds to the voltage across an electrical circuit element

Kirchhoff's Current Law states the the sum of all currents at any circuit node is zero. By assigning a "sign" to the direction of flow in the flowchart, it is easy to show that this is true for the number of executions in a flowchart. The number of times into any node in the flowchart is equal to the number of times out of that node. Kirchhoff's voltage law states that the sum of all voltage drops and emf's around any circuit loop is zero. The analogy for the voltage law breaks down in the case of parallel connected sections in a flowchart. Here Kodres introduced the idea of placing "current" sources in each closed loop in the flowchart. The value of the current source is equivalent to the number of times the loop is executed.

4-14

In the examples which follow, this notation applies:

P_t is the fractional execution count for the true (t) branch of an if statement

T is a prefix that indicates that the quantity is an execution time for a program block or element (Examples are TA, TC_f)

n is the number of executions of a loop body

The expressions which are given with each program construct represent the equivalent "voltage" or total execution time of the block in question.

The structured programming constructs involving closed flowchart loops are translated as follows:

- if-then-else is equivalent to a single statement block with a value of
$P_t(TC_t + TA) + (1 - P_t)(TC_f + TB)$
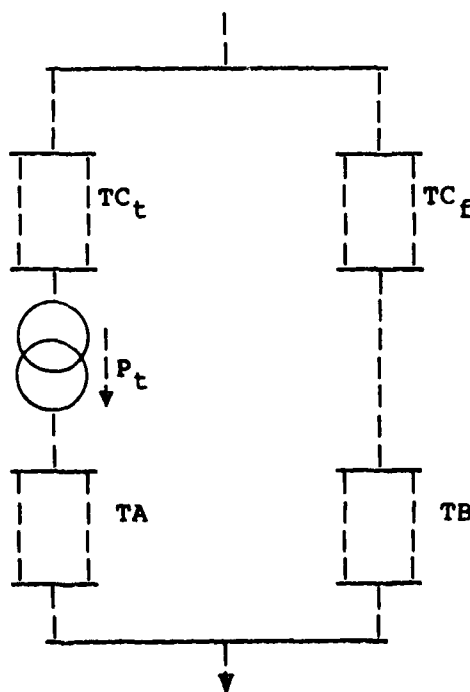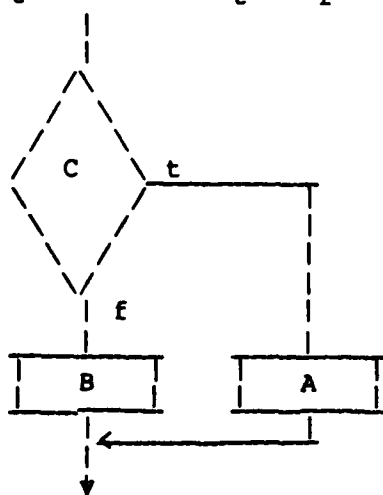
- **do-while** is equivalent to a single statement block with a value of
$$n(TC_t + TA) + TC_f$$

- **do-until** is equivalent to a single statement block with a value of
$$m(TC_f + TA) + TA + TC_t$$

4-16
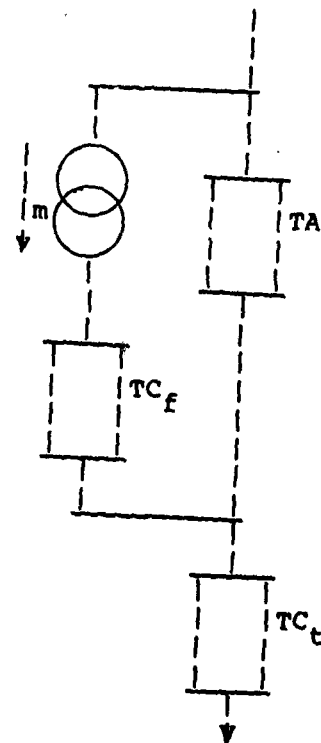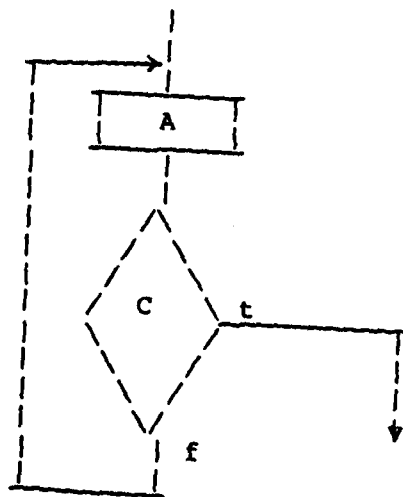
The limit of this approach is clear and has been pointed out by all who have written about the technique. The difficult part of the analysis of algorithms is the determination of the number of times a loop is executed or in this analog, the value of the current source. However, if one could solve this problem, then this technique guarantees that one can get the solution to any structured flowchart.
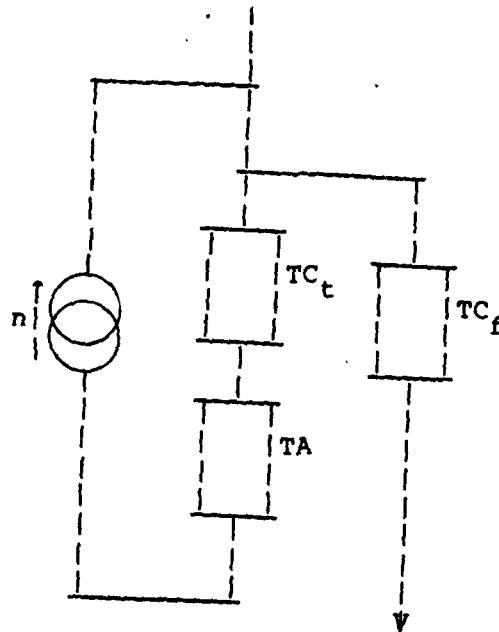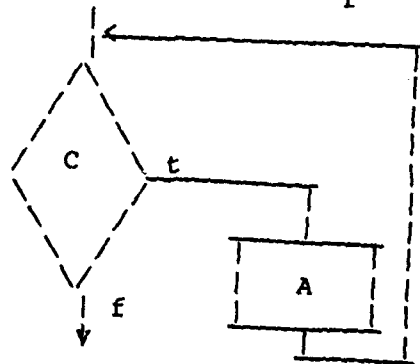
Wegbreit's Probability System

Wegbreit's systematic approach to the analysis of algorithms was introduced in an article on "Verifying Program Performance" [2]. The analysis of the algorithm is a natural by-product of proving that the program/algorithm is correct, and a refinement of the use of well-ordered sets, first suggested by Floyd. The algorithm is instrumented to record the desired performance parameter. Then the appropriate probabilistic input assertions are made about variable probability distributions and inductive assertions are shown to hold at intermediate stages in the algorithm. When one of the inductive assertions can be shown to be a loop invariant it can be manipulated into a statement about the algorithm's performance. The important advance of Wegbreit's probability system is that it sets out to calculate the branching probabilities in order to determine average computation time.

Ramshaw [5] states that this method is based on the ideas of Floyd and Hoare. It uses formal reasoning about predicates of the form $Pr(P) = e$, $0 \le e \le 1$. Which means that the probability that the predicate P is true is equal to the real-valued expression e. Ramshaw has shown [5] that systems of this form have problems with a very simple program which he calls the Leapfrog Problem:

Leapfrog: if K = 0 then K <- K + 2 endif

We assume that K can take on the values of 1 and 0 with equal probability, i.e.,

$$[Pr(K=0)=\tfrac{1}{2}] \; \bigwedge \; [Pr(K=1)=\tfrac{1}{2}]$$

The output assertion which one would expect to get is:

$$[Pr(K=1)=\tfrac{1}{2}] \; \bigwedge \; [Pr(K=2)=\tfrac{1}{2}]$$

However, all that can be asserted using a Floyd-Hoare system is:

$$Pr([K=1] \; \bigvee \; [K=2]) = 1$$

This is not particularly informative or of much use in subsequent portions of the program since all of the information about the distribution of the input has been lost.

Ramshaw's Frequentistic System

In his Ph.D. dissertation, Ramshaw [5] reformulates the ideas about probabilistic assertions into what he calls "frequentistic" assertions. In this way he "avoids the rescalings that are associated with taking conditional

probabilities." Ramshaw's frequency "is like probability in every way except that it doesn't always have to add up to one." He defines a frequentistic state as a collection of deterministic states with their associated frequencies. Atomic assertions are statements of the form $Fr(P)=e$, where P is a predicate and e is a real-valued expression.

Ramshaw applies his frequency system successfully to the Leapfrog problem.

Leapfrog: if K = 0 then K ←- K + 2 endif

His input assertion is:

$$[Fr(K=0)=\tfrac{1}{2}] \;\; \bigwedge \;\; [Fr(K=1)=\tfrac{1}{2}]$$

This means that the frequency associated with the state K=0 is $\tfrac{1}{2}$ and the frequency associated with the state K=1 is also $\tfrac{1}{2}$. The total frequency associated with the variable K is $\tfrac{1}{2}+\tfrac{1}{2} = 1$.

So far we have followed Ramshaw's thesis closely. The following is a slightly different interpretation of the application of his method which arrives at the same answer. We present it here in this way because it seems a little more formal than his presentation.

The if-test on the predicate { K=0 } conjoins the branch atomic assertion [$Fr(K\neq0)$ = 0] to the TRUE out-branch. This is derived by setting the frequency of the negation of the if-test predicate equal to zero. For the FALSE out-branch, the branch atomic assertion is [$Fr(K=0)$ = 0]. This simply states that the frequency with which the

if-test predicate is true in the FALSE out-branch is zero!

Each atomic assertion in the input assertion is individually resolved with the branch atomic assertion, in the manner of theorem proving systems. If there is a contradiction, then that conjunct of the input assertion is dropped. In the TRUE branch we have:

$$[Fr(K=0)=\tfrac{1}{2}] \wedge [Fr(K\neq0)=0]$$

which is logically consistent, but

$$[Fr(K=1)=\tfrac{1}{2}] \wedge [Fr(K\neq0)=0]$$

is a contradiction and is dropped. In the FALSE branch we have:

$$[Fr(K=0)=\tfrac{1}{2}] \wedge [Fr(K=0)=0]$$

which is a contradiction, and

$$[Fr(K=1)=\tfrac{1}{2}] \wedge [Fr(K=0)=0] = [Fr(K=1)=\tfrac{1}{2}] \wedge [Fr(K\neq1)=0]$$

which is a valid assertion.

In the TRUE branch, the assignment statement changes the deterministic states of K to have the value K+2.

$$[Fr(K=2)=\tfrac{1}{2}] \wedge [Fr(K\neq2)=0]$$

The assignment statement maps all of the frequencies of the states of K in this branch into the frequency of the state K+2.

At the final join, the output assertion is the conjunction of the two branch assertions, namely:

$$[Fr(K=2)=\tfrac{1}{2}] \wedge [Fr(K\neq2)=0] \wedge [Fr(K=1)=\tfrac{1}{2}] \wedge [Fr(K\neq1)=0]$$

This statement contains the logical contradiction:

$$[Fr(K\neq1)=0] \wedge [Fr(K\neq2)=0]$$

4-20

Unlike the case with the restriction at the if-test, a contradiction at the join (which must·be between atomic assertions from separate out-branches) is resolved by conjoining each branch's contribution to a given frequentistic state within a single predicate. In this case:

$$[Fr(K \neq 1)=0] \;\wedge\; [Fr(K \neq 2)=0] \implies [Fr(K \neq 1 \;\wedge\; K \neq 2)=0].$$

We arrive at Ramshaw's output assertion:

$$[Fr(K=1)=\tfrac{1}{2}] \;\wedge\; [Fr(K=2)=\tfrac{1}{2}] \;\wedge\; [Fr(K \neq 1 \;\wedge\; K \neq 2)=0].$$

This result is a little more useful! It says that K is either 1 or 2 and that it takes on either value with equal probability.

Now, one would think that all this would lead to a very powerful method. It does. Ramshaw shows how to apply this straight forward approach to the COINFLIP algorithm in Chapter 5 of his thesis [5]. His analysis is very similar to the one that we will give in Chapter 4. But, instead of continuing to use the more straight-forward approach, Ramshaw follows Kozen's semantics for probabilisitic programs, applies measure theory, and shifts to a "theorem-proving" approach. He uses the following rule of consequence to prove theorems about the conditional statement:

$$\frac{\vdash [A|P]S[B], \;\; \vdash [A|\neg P]T[C]}{\vdash [A]\text{if } P \text{ then } S \text{ else } T \text{ fi}[B+C]}$$

This rule of consequence means that, if the truth of predicate A given that P is true implies that B is true

after the execution of program section S, and if the truth of predicate A given that P is false implies the truth of predicate C after the execution of program section T, then if A is true before the if statement involving P, S, and T, then it follows that either B or C is true afterward.

Ramshaw's frequency system can handle some of the programs which Wegbreit's can't, because Ramshaw avoids problems of renormalizing probabilities. But because Ramshaw chose to use this rule of consequence for the if statement, his system still can't handle the "useless test":

if R then nothing else nothing endif.

Ramshaw must include a special rule of consequence for the "useless test" (one that says that nothing happens). This seems to be symptomatic of those formal systems of algorithm analysis which have grown from the work in program verification based on theorem proving.

We have just given a taste of Ramshaw's frequency system. Readers who are interested in learning more about it should see Ramshaw's dissertation [5].

Automatic Analyzers

We now turn our attention to the current state of automatic analysis. We will look at two systems which have been reported in the literature.

Wegbreit's METRIC

METRIC [1] is a system, written in Interlisp, which is able to analyze simple LISP programs and produce closed-form expressions for the parameter of interest in terms of the size (in some sense) of the input. The analysis of a program takes place in three distinct phases:

1. Assign a cost to each primitive operation. This process continues as long as the procedure is not recursive. Blocks of primitive operations are assigned the cost of the sum of their individual costs.

2. Analyze the recursive procedures. This phase analyzes how the recursion variables change from one iteration to the next. A series of difference equations is generated by projecting this recursive structure onto the set of integers.

3. Solve the difference equations. This phase finds a closed-form expression for the difference equations. Wegbreit has implemented solutions to these equations based on: direct summation, pattern matching, elimination of variables, best-case/worst-case analysis, and differentiation of generating functions.

In Wegbreit's processing of conditional statements, he assumes that all tests are independent. This is perhaps the most serious flaw in the approach. Again the problem stems from the difficulty in handling conditional probabilities.

4-23

Cohen and Zuckerman's EL/PL

Evaluation Language/Programming Language [3] is a system that consists of an ALGOL-like language for expressing algorithms (PL) and a language for analyzing the resulting algorithms (EL). The PL statements are compiled by the PL compiler into a symbolic formula representing the time for executing the program. This "object deck" is present to the EL processor. The EL processor, in turn, provides a human operator with the means to manipulate the symbolic formula into answers. EL runs in an interactive mode. It allows the operator to bind formal or numerical values to the execution counts of loops and to assign formal or numerical values to the probabilities of boolean expressions.

Here, as with METRIC, the operator has to provide the critical data on the branching probabilities. The branching probabilities of different conditional statements are assumed to be independent of each other. This seems to be the most serious defect in the automatic analyzers to date.

CHAPTER 3

SYNTAX DIRECTED TRANSLATION APPROACH

In this chapter, we will discuss our approach to the systematic analysis of algorithms. The presentation follows the order in which the work actually progressed. Our research was sparked by the arrival of Ramshaw's thesis [5]. It seemed to us, at the time, that the theorem-proving approach was overly mathematical. There must be, we said, a way to look at this which is more closely related to the code and more understandable by programmers. Wegbreit's article on METRIC [1] got us thinking about the utility of translating program loops into recursive subroutines.

Loops make the analysis of algorithms interesting. Without loops it's once through and done. Straight line code is easy to analyze. When you add some branching statements it gets a little harder; but it's the loops which make an analysis really interesting. The first observation is that there has been a lot of work done on solving recurrence relations. If we can convert all of the different loop structures to recursive subroutine calls, then we can apply the same techniques to attempt to analyze all kinds of

loops.   In fact, one can do exactly that, as Wegbreit [1]

points out.   He also points out that if there are no

conditional branches in the loops, then there is an exact

solution to the recurrence relations.   Our procedure is

basically quite simple:

1. Convert all loops into recursive subroutine calls

2. Convert the recursive subroutine calls into
   recurrence relations

3. Solve the recurrence relations

Solving Recurrence Relations

There are three basic methods for solving recurrence

relations:

1. Inspect the relation to see if you have seen it
   before in another problem, or recognize a general
   form

2. Try a few iterations to get the feel of the recur-
   rence relationships and the way the relations
   behave, then guess a closed-form answer, and prove
   its correctness by induction

3. Apply one of the standard techniques to solve the
   recurrence relation

Within these simple steps are contained a lot of art

and experience.   G. S. Lueker in a recent tutorial "Some

Techniques for Solving Recurrences" [16] gives an excellent

introduction to these methods.   Advanced techniques can be

found in Knuth [6], and especially Jonassen and Knuth [8].

We shall list some of the techniques mentioned by Lueker [16].

1. Summing factors -- where one tries to manipulate the recurrence relations by addition of expressions for adjacent terms in the hope that the sum will "telescope" into a few terms, one of which is the nth term.

2. Characteristic equations -- where the problem is mapped into that of finding the roots of a characteristic system of polynomial equations. This approach works for linear recurrences with constant coefficients.

3. Range transformation -- where the unknown coeffic-ents in the recurrence relations are transformed by some function which turns an unknown problem into a known problem, or one that can be solved by another technique.

4. Domain transformation -- where the index value is transformed to make the progression of values additive instead of some other function. Once this is done, summing factors can often be used.

5. Generating functions -- where the problem is transformed into another domain in a way similar to the transformation of a time-domain function into a frequency-domain function by a Fourier transform. This method is particularly powerful for handling probabilistic aspects of solutions.

Our work in this thesis, involved some very familiar recurrences for which the answers were easily guessed.

Translating Loops into Recursive Subroutines

We will limit our discussion to algorithms expressed using structured programming constructs only. This is not a particularly restrictive limitation since the structured programming constructs are all that is theoretically needed to describe any alogrithm. For this reason and the fact that such programs are easier to maintain, most new programming is being done using structured programming methods.

We will adopt SPARKS as the language for expressing algorithms. SPARKS was developed by Horowitz and Sahni in 1976 [9] and sightly modified in 1978 [7].

We have developed a formal syntax-directed translation schema for converting structured loop constructs into recursive subroutines.

Given the input syntax of the FOR loop:

```
<label>:     for <var> <- <exp_1> to <exp_2> by <exp_3> do
    <statements with live variables>
    repeat
```

we get the recursive syntax:

```
    start <- <exp_1>; stop <- <exp_2>; incr <- <exp_3>
    <var> <- start
    call <label>(<var>,incr,stop,{ live variables } )
procedure <label>(var,incr,stop,{ live variables })
    if SGN(incr) * ( stop - var ) > 0  then
        <statements with live variables>
        var <- var + inc
        call <label>(var,incr,stop,{ live variables } )
    endif
end <label>
```

The live variables from <statements> are those variables which are used or created in <statements> and have a scope that extends outside of <statements>.

The procedure for converting DO WHILE loops to recursive subroutine calls is quite similar.

```
<label>: while < relational expression > do
     < statements with live variables >
     repeat
```
The recursive syntax is:
```
     call <label>( {live variables, relational variables} )
  procedure <label> ({live variables, relational variables})
     if < relational expression > then
         < statements with live variables >
         call <label> ( { live variables,
                          relational variables } )
     end if
  end <label>
```

Simple Examples

do while example (Algorithm for n^n)

The following algorithm is a modification of one by Horowitz and Sahni [10].

```
procedure N_to_the_N
     read R1
     R2 <- 1; R3 <- R1
T1: while R3 > 0 do
         R2 <- R2 * R1; R3 <- R3 - 1
     repeat
     print R2
end N_to_the_N
```

This procedure contains a single while loop which we wish to analyze. The time behavior of this algorithm is dominated by the number of times that the body of the while loop is executed. We first translate the while loop into a recursive subroutine. The algorithm becomes:

```
procedure N_to_the_N
    read R1
    R2 <- 1; R3 <- R1
    call T1( R1, R2, R3 )
    print R2
end N_to_the_N
procedure T1 ( R1, R2, R3 )
    if R3 > 0 then
        R2 <- R2 * R1; R3 <- R3 - 1
        call T1( R1, R2, R3 )
    end if
end T1
```

Only program variable R3 has any effect on the course of the recursion. Let i be the mathematical variable which corresponds to R3, and T be the number of calls on the subroutine. Then:

$$
T(i) = \begin{cases} 1, & \text{if } i \leq 0 \\ 1 + T(i-1), & \text{if } i > 0 \end{cases}
$$

The subroutine T1 is called from the main program with $i = R1$. Therefore, the recursion is solved by:

$$
T1(R1) = \sum_{j=R1}^{0} 1 = R1 + 1
$$

The subroutine T1 is called one time more than the value of R1, which we expected.

ODD/EVEN Print Example

This example is a little more difficult. It involves an if statement, but one which is completely determined by the starting number. ODD(I) is a built-in function which returns True if its argument is odd, and False if the argument is even.

```
procedure ODD_EVEN ( N )
    I <- N
    while I > 1 do
Ta:     print 'AAA'
        if ODD( I ) then
            I <- I - 3
        else
            I <- I + 1
        end if
    repeat
end ODD_EVEN
```

The recursive form of the program is:

```
procedure ODD_EVEN ( N )
    I <- N
    call Ta(I)
end ODD_EVEN
procedure Ta ( I )
    if I > 1 then
        print 'AAA'
        if ODD( I ) then
            I <- I - 3
        else
            I <- I + 1
        end if
        call Ta(I)
    end if
end Ta
```

Wegbreit [1] points out the idea for the next step and goes into it in greater detail than we shall here. He states, "The essential idea is to map a recursive procedure P into a new recursive procedure whose value is the cost of P." We are interested in the number of times that AAA is printed. The recurrence relation for it is given by:

$$T_a(i) = \begin{cases} 0, & \text{if } i < 1 \\ 1 + T_a(i-3), & \text{if } i \text{ is odd} \\ 1 + T_a(i+1), & \text{if } i \text{ is even} \end{cases}$$

Starting with the case where i is odd, we have:

$$T_a(i_o) = 1 + T_a(i_o - 3)$$

Now, $i_o - 3$ is even so we have (assuming $i_o - 3 \geq 1$)

$$T_a(i_o) = 1 + 1 + T_a(i_o - 3 + 1) = 2 + T_a(i_o - 2)$$

Note that $i_o - 2$ is also odd.

We now examine the case when i_o is even:

$$T_a(i_e) = 1 + T_a(i_e + 1)$$

Now, $i_e + 1$ is odd, so we have

$$T_a(i_e) = 1 + 1 + T_a(i_e + 1 - 3) = 2 + T_a(i_e - 2)$$

Since the recursions for the odd and even cases have been transformed to eliminate the dependence on parity, we have the new recurrence relations:

$$T_a(i) = 2 + T_a(i-2), \text{ if } i \geq 2$$

$$T_a(1) = 1$$

$$T_a(0) = 0$$

Whose solution is easily shown to be $T_a(i) = i$.

COINFLIP

COINFLIP is an algorithm which Ramshaw [5] uses. Here we translate it into SPARKS. The built-in function $RANDOM_{ht}$ returns a value of Heads or Tails with equal probability.

```
procedure COINFLIP
    I <- 0
    while RANDOM   = T do
              ht
Tc:       print 'ok, so far!'; I <- I + 1
    repeat
    print I, ' times!!'
end COINFLIP
```

The recursive version is:

```
procedure COINFLIP
    I <- 0
    call Tc(I)
    print I,' times!!'
end COINFLIP
procedure Tc( I )
    if RANDOM   = T then
            ht
        print 'ok, so far!'; I <- I + 1
        call Tc( I )
    end if
 end Tc
```

The question "how many times will tails turn up in succession?" is equivalent to asking how many times will 'ok, so far!' be printed out. We see that:

$$T_c(i) = \begin{cases} 0, & \text{if } RANDOM_{ht} = H \\ 1 + T_c(i+1), & \text{if } RANDOM_{ht} = T \end{cases}$$

where T_c is the number of times that the statement labeled

Tc in the original program is executed. If RANDOM$_{ht}$ returns
H the first time that it is called, then the statement is
never executed. If RANDOM$_{ht}$ always returns T, then the
program does not terminate. The in-between cases are the
interesting ones. What is the expected value of i, i.e. the
expected number of times that 'ok, so far' is printed? To
answer this question requires an investigation of the part
that probability plays in the conditional statement. We
will come back to this question later.

FINDMAX

This algorithm has been used as an example by several
authors [5, 6, and 7]. It is the usual algorithm for
finding the maximum value of a set of numbers. This is the
first example which we have given in which the recursive
form of the algorithm is not obvious. For this reason we
will give the translation explicitly.


```
procedure FINDMAX( A, N, XMAX )
/* set XMAX to the maximum value in A(1:N), N>0. */
    XMAX <- A(1)
L1: for I <- 2 to N do
        if A(I) > XMAX then  XMAX <- A(I); end if
    repeat
end FINDMAX
```

The recursive version of this program is:

```
procedure FINDMAX( A, N, XMAX )
/* set XMAX to the maximum value in A(1:N), N>0. */
    XMAX <- A(1); I <- 2
    call L1( A, N, I, XMAX )
end FINDMAX
procedure L1( A, N, I, XMAX)
    if I <= N then
        if A(I) > XMAX then
T1:         XMAX <- A(I); end if
        I <- I + 1
        call L1( A, N, I, XMAX)
    end if
end L1
```

The next step is to convert the recursive algorithm
into a recurrence relation for the number of times that
control passes T1. In this case we are interested in the
number of times that a new maximum is found.

$$T(A, n, i, xmax) = \begin{cases} 1 + T(A,n,i+1,A(i)) & \text{if } A(i) > xmax \\ 0 + T(A,n,i+1,xmax) & \text{if } A(i) \leq xmax \end{cases}$$

with the boundary condition $T(A, n, k, xmax) = 0$ for $k>n$.

Given a known input array, $A(1:n)$, this recurrence
relation completely determines the value of T. If this were
all that could be learned, then it would not be very useful.
The answer could just as well be determined by instrumenting
the original algorithm with a test counter in the true
branch. In this case we observe that the true branch is
taken if the i-th element is the largest of the first i
elements. If p_i is the probability that A(i) is the largest

4-35

of i elements we have:

$$T(A,i) = p_i + T(A,i+1)$$

as a description of the average behavior of the algorithm. At this point we have dropped the arguments of T which return the "answer" so that we can concentrate on the time behavior.

If the elements $A(i)$ are drawn from a uniform distribution, then $p_i = \frac{1}{i}$ and

$$T(A,i) = \frac{1}{i} + T(A,i+1)$$

$$T(A,i) = 0, \quad \text{for } i > n$$

Since the initial value of i is 2, the solution to this recursion is easily shown to be $T(A,2) = H_n - 1$, where H_n is the nth harmonic number:

$$H_n = \frac{1}{1} + \frac{1}{2} + \frac{1}{3} + \ldots + \frac{1}{n}$$

While we were able to get the correct solution, this way of analyzing the algorithm is not suited for automation. The insight into the distribution of the data and its effect on the probability that the branch would be taken requires human-like understanding.

The Problem of the Conditional Statement

At this point, our approach has the same problem that plagues the Electrical Network approach--it works fine if one knows the branching probabilities. It was at this point in our research that we went back and studied the work of Wegbreit and Ramshaw more closely. We noted the strengths

and weaknesses which we described in Chapter 2. Knuth [5] provides an analysis of FINDMAX which relies on some subtle reasoning about left-to-right maxima among random permutations. Since we plan to teach a computer how to do this analysis, we wanted to keep any real "thinking" out of it until absolutely necessary. In Wegbreit's and Ramshaw's approaches, the fact that the program variables of interest are random variables and have distributions is recognized. However, most of their analyses are performed by making assertions about the frequencies or probabilities of these distributions, and then proving theorems about the assertions. The problem of the "useless test" led us to think that it might be useful to see what happened when one followed the distributions themselves around the program.

At this point we had been concentrating so much on understanding the true meaning of "differentially disjoint vanilla assertions", the measure theory, and theorem proving aspects of Ramshaw's frequency system [5], we had forgotten that his treatment of COINFLIP dealt with the distributions themselves. It was only after we had devised a major portion of our approach that we realized the great similarity between our's and Ramshaw's frequency system (as it stood in Chapter 5 of his thesis [5]). We then recognized that we had continued down the path of following the distributions, while Ramshaw had turned to follow the path of proving theorems about frequentistic assertions.

4-37

CHAPTER 4

DEALING WITH CONDITIONAL STATEMENTS

In this chapter we introduce the central idea which, we feel, is either a new idea or one which has been inadequately expressed in the past. The problem with the conditional statement stems from the normalizations required when taking probabilities, so why not, we reasoned, put off taking the probabilities as long as possible? Ramshaw's thesis [5] was a key to this. We observed his abandoning of his raw frequencies in favor of asserting predicates about frequencies. Another key factor in our choosing this direction was Jonassen and Knuth's paper on "A Trivial Algorithm Whose Analysis Isn't" [8]. Here were these nice joint probability distribution functions (p.d.f.) which appeared from "directly translating the algorithm into mathematical formalism." We set out to find the rules that had to have been used to get to these simple recurrence relations. Because we took so many wrong turns on our way to our final ideas, we will abandon our historical presentation in favor of a more expository one. We also have to abandon our initial assessment that Ramshaw's approach was "too mathematical". There

seems to be no way to avoid mathematics if one desires more than the analysis of the simplest algorithms.

Algorithms and Probability Distributions

Each execution of an algorithm can be thought of as a random experimental sample from the universe of possible input data. We will be concerned with the behavior of the probability distributions associated with the program variables during execution of the algorithm. These probability distributions can be thought of as the repository of all the information about possible execution histories for an algorithm. We perform the analysis of an algorithm's behavior by manipulating these distributions to find probabilites for various conditions. We can then use this information in any of the analysis techniques (e.g., those given in Chapters 2 and 3), which work for known branching probabilities.

We begin by associating a _random variable_ with each _algorithm or program variable_. We will follow Ramshaw [5] and differentiate between the two by continuing to represent algorithm variables by upper-case character strings and representing the corresponding random variable by the same characters in lower-case letters. For example, the random variable xmax is associated with the program variable XMAX. The value of the random variable x at any time in the execution of the algorithm is the value of the corresponding algorithm variable at that time. Unlike Ramshaw, we have no

4-39

prohibition about mixing program and mathematical variables in the same expression. In fact this will be how we get some of our answers.

We define the _probability set function_, $P_X(A)$, to be the probability that the program variable X is contained in the set of possible values A, i.e., $P_X(A) = Pr(X \in A)$. If the set A is countable, we obtain the _discrete probability density function_ (p.d.f.), $f_X(x)$:

$$f_X(x) = Pr(X \in A) \mid A = \{ \text{ some finite set of x's } \} \quad (4\text{-}1a)$$

If we let the set A be the set of all values of $\{X \mid x \leq X \leq x+dx\}$ we have the _continuous probability density function_, $f_X(x)$:

$$f_X(x) = Pr(X \in A) \mid A = \{ x \leq X \leq x+dx \} \quad (4\text{-}1b)$$

We will deal with the discrete type of random variable in our formalism because of the fact that all values within a computer can be mapped onto a finite set of integers. By staying with discrete representations, we avoid the need for the concept of "differential equality" which Ramshaw [5] introduced to bridge the gap between continuous variables and program equality expressions. We will develop a notation which is very close to the calculus of finite differences. Some of the rules which we will use will be derived from analogous rules in continuous probability theory and the calculus of continuous variables.

Equations (4-1) can be generalized to any finite number of program variables by thinking of the X as a vector of the n ordered program variables and x as an n dimensional random

vector. The random variables form a vector space in $\mathbb{R}^n$ and $f_X(x)$ is a functional over that space. ·

The joint p.d.f. of the program variables describes the state of the program up to a point in the execution of the program. If we have a loop translated into a recursive subroutine call, and if we can describe the joint p.d.f. before the next recursive call in terms of the joint p.d.f. entering the body of the subroutine, then we have a recurrence relation that we may be able to solve to get the joint p.d.f. as a function of the number of calls on the subroutine. This knowledge will allow us to calculate the branching probabilities at any step in the process and hence complete the analysis of the algorithms begun in Chapter 3.

Let us now examine the behavior of the joint p.d.f. with various programming constructs. We begin with the conditional statement.

Theorem 1:

If R is a deterministic logical relation of the program variables then, the conditional statement

> if R then { S_t } else { S_f } endif

a. Divides the joint p.d.f. entering the if statement into two parts by:

1. setting to zero all terms of the joint p.d.f. entering the then clause { S_t } for which R is FALSE, and

2. setting to zero all terms of the joint p.d.f. entering the else clause { S_f } for which R is TRUE.

b. Forms the joint p.d.f. leaving the **endif** from the algebraic sum of the joint p.d.f.s leaving the two clauses.

We will not present a formal proof, but will use Theorem 1 as a rule and see how it handles situations for which we have answers by other means.

The effect of the conditional statement on the joint p.d.f. entering each clause can be represented in a compact manner using a new type of delta function which we will refer to as the Boolean delta. This new delta function is closely related to the Kronecker and Dirac delta functions, except that its domain is a Boolean space with possible values True and False. The Boolean delta maps the Boolean space into the numbers 0 and 1.

Definition

Let R be a deterministic logical relation of program variables, then the Boolean delta function

$$\delta(R) = \begin{cases} 1 \text{ if R is TRUE} \\ 0 \text{ if R is FALSE.} \end{cases}$$

It is easy to see that the following properties hold:

$$\delta(R) \cdot \delta(\neg R) = 0$$
$$\delta(R) + \delta(\neg R) = 1$$
$$\delta(R) = 1 - \delta(\neg R)$$
$$\delta(R \wedge S) = \delta(R) \cdot \delta(S)$$
$$\delta(R \vee S) = \delta(R) + \delta(S) - \delta(R) \cdot \delta(S)$$

With these properties one can find the Boolean delta of any Boolean expression. We can now state a theorem about the effects of the "useless test" on the joint p.d.f.

Theorem 2

Let $f_X(x)$ be the joint p.d.f. of the n program variables $x_1, x_2, \ldots, x_n$ at a point in an algorithm just prior to the "useless test",

    if R then nothing else nothing endif

where R is a deterministic logical relation on the program variables X, and let $g_X(x)$ be the joint p.d.f. of the program variables after the join at the endif, then $g_X(x) = f_X(x)$.

Proof:

Using Theorem 1 and the Boolean delta $\delta(R)$ we have the augmented algorithm:

$$\{ \; f_X(x) \; \}$$

```
if R
    then      { f_X(x) · δ(R) }
          nothing
    else      { f_X(x) · δ(¬R) }
          nothing
endif         { g_X(x) = f_X(x) δ(R) + f_X(x) δ(¬R) }
              { g_X(x) = f_X(x) · ( δ(R) + δ(¬R) ) }
              { g_X(x) = f_X(x) }
```

<div align="right">Q.E.D.</div>

This discussion of the joint p.d.f. of the program variables is very close to Ramshaw's [5] frequentistic states. We can show that Ramshaw's frequentistic assertions can be derived from marginal or joint p.d.f.s. We depart from Ramshaw is that we will stay with the rules for the transformation of the joint p.d.f. by the algorithms instead of moving to the next higher level of abstraction, i.e. rules for the transformation of assertions about the marginal or joint p.d.f.s. It was this abstraction which destroyed the ability of Ramshaw's system to handle the "useless test".

LEAPFROG Revisited

In order to get some understanding of the effects of simple assignment statements, let us look again at LEAPFROG.

$$\text{Leapfrog: if } K=0 \text{ then } K \leftarrow K+2 \text{ endif}$$

The input joint p.d.f. to Leapfrog is

$$f_K(k) = \frac{1}{2} \delta(k=0) + \frac{1}{2} \delta(k=1)$$

which simply means that $Pr(k=0) = \frac{1}{2}$, and $Pr(k=1) = \frac{1}{2}$.

The augmented program would be:

$$\text{if } K=0 \text{ then} \quad \{ \delta(k=0)(\tfrac{1}{2}\delta(k=0)+\tfrac{1}{2}\delta(k=1)) \}$$

$$\{ \tfrac{1}{2}\delta(k=0) \}$$

$$K \leftarrow K+2 \quad \{ \tfrac{1}{2}\delta((k-2)=0) \}$$

$$\{ \tfrac{1}{2}\delta(k=2) \}$$

$$[\text{ else }] \quad \{ \delta(k\neq 0)(\tfrac{1}{2}\delta(k=0)+\tfrac{1}{2}\delta(k=1)) \}$$

$$\{ \tfrac{1}{2}\delta(k=1) \}$$

$$\text{endif} \quad \{ \tfrac{1}{2}\delta(k=2) + \tfrac{1}{2}\delta(k=1) \}$$

Which is exactly what we should get.

In handling the assignment statement, $K \leftarrow K+2$, we observed that it maps k as follows:

k before	k after
.	.
.	.
-2	0
-1	1
0	2
1	3
2	4
.	.
.	.

In general, if we wish to keep the equations in terms of the original variables, we have:

$[x_i \leftarrow x_i + c]$:

$$\langle x_1, x_2, \ldots, x_i, \ldots, x_n \rangle \rightarrow \langle x_1, x_2, \ldots, x_i - c, \ldots, x_n \rangle.$$

Next we will look again at the COINFLIP algorithm. To do that we need some rules about the effects of a conditional statement which contains a non-deterministic part. We can easily transform a non-deterministic relation into a non-deterministic assignment followed by a deterministic conditional statement. For example:

if X = RANDOM$_{ht}$ then { S_t } else { S_f } endif

becomes

$Y \leftarrow$ RANDOM$_{ht}$
if $X=Y$ then { S_t } else { S_f } endif.

Theorem 3

Let $f_X(x)$ be the joint p.d.f. of the n program variables $X_1, X_2, \ldots, X_n$ in the algorithm just prior to the conditional statement

if R then { S_t } else { S_f} endif

where R is a logical relation containing a finite number, m, of random (possibly pseudo-random) functions RANDOM$_{fj}$. Let R' be derived from R by replacing each instance of RANDOM$_{fj}$ with a reference to a new program variable Y_j, then the following sequence of statements are equivalent to the original statement:

Y_1 = RANDOM$_{f1}$
Y_2 = RANDOM$_{f2}$
. . . .
. . . .
Y_m = RANDOM$_{fm}$
if R' then { S_t } else { S_f } endif

4-45

Theorem 4

Let $f_X(x)$ be the joint p.d.f. of program variables $X_1, X_2, \ldots, X_n$ which have been defined, and let Y be a "new" variable defined by the statement $Y \leftarrow \text{RANDOM}_g$, where RANDOM_g generates a statistically independent random number from distribution $g(y)$, then the joint p.d.f. after this statement, $h_Z(z)$, is

$$h_Z(z) = f_X(x) \cdot g(y)$$

where,

$$z = \langle x_1, x_2, \ldots, x_n, y \rangle$$
$$Z = \langle X_1, X_2, \ldots, X_n, Y \rangle .$$

It is now time to examine the general assignment statement between two program variables. We will use a memory-to-register, register-to-memory model for the assignment statement. This will allow us to have the statement $X \leftarrow X$ be a NOOP in the formalism without any special rules. We introduce the notation

$$\sum_{x_i}$$

to mean the summation over all values of random variable x_i. This is the discrete equivalent of the definite integral. When it is applied to a function of x_i, the result does not depend on x_i. If this summation is done symbolically, all occurences of x_i are removed from the equation of the result. Here are some properties of this summation which we shall use later:

$$\sum_{x_i} f(x_i) = 1 , \quad \text{when } f(x_i) \text{ is a p.d.f.}$$
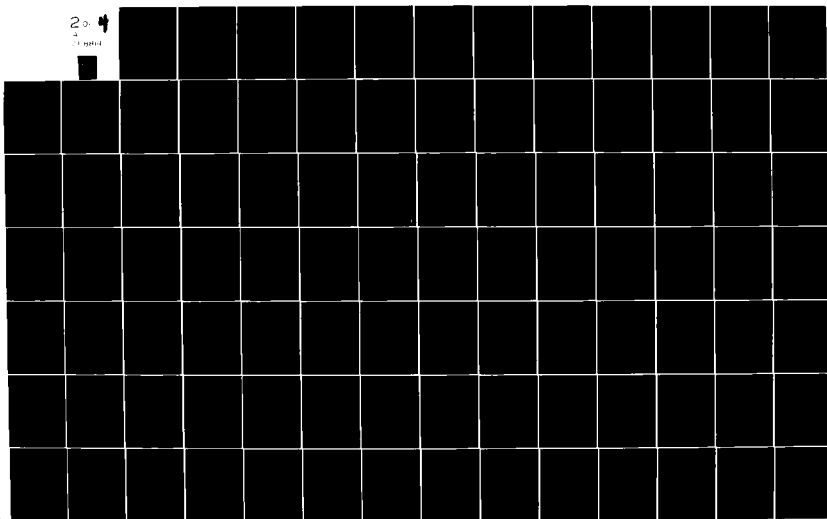
$$\sum_{x_i} (f(x_i) \, \delta(x_i = x_j)) = f(x_j)$$

$$\sum_{x_i} (f(x_i) \, \delta(x_i \leq x_j)) = F(x_j)$$

where $F(x_j) = Pr(X \in A) \mid A = \{ X \leq x_j \}$ is the cumulative probability density function (c.p.d.f.) for f. Note that in the case of discrete random variables we usually have to worry about whether or not the c.p.d.f. is defined to include x_j or whether it is just "up to" x_j. In the continuous representation we would not have to worry about this because the two are equivalent.

Theorem 5

Let $f_X(x)$ be the joint p.d.f. of the n program variables $X_1, X_2, \ldots, X_n$ just before the program statement

$$X_i \leftarrow X_j$$

Then the joint p.d.f. after this assignment statement is

$$g_X(x) = (\sum_{x_i} f_X(x) \, \delta(x_i = r)) \, \delta(r = x_j)$$

The application of $\delta(x_i = r)$ within the summation takes care of the case when x_i is the same variable as x_j. In the cases where x_i and x_j are different variables, the rule reduces to:

$$g_X(x) = (\sum_{x_i} f_X(x)) \, \delta(x_i = x_j)$$

For an example we will look at a simple program which interchanges the contents of two variables X_1 and X_2 using a third variable X_3 as temporary storage. The augmented program goes like this:

$$\{ \ f_X(x_1,x_2,x_3) \ = \ g_X(x_1,x_2) \delta(x_3=0) \ \}$$

$$X_3 \leftarrow X_1 \qquad \{ \ f_X(x_1,x_2,x_3) \ = \ g_X(x_1,x_2) \delta(x_3=x_1) \ \}$$

$$X_1 \leftarrow X_2 \qquad \{ \ f_X(x_1,x_2,x_3) \ = \ g_X(x_3,x_2) \delta(x_1=x_2) \ \}$$

$$X_2 \leftarrow X_3 \qquad \{ \ f_X(x_1,x_2,x_3) \ = \ g_X(x_3,x_1) \delta(x_2=x_3) \ \}$$

$$\{ \ f_X(x_1,x_2,x_3) \ = \ g_X(x_2,x_1) \delta(x_3=x_2) \ \}$$

Note that we need not have assumed that X_3 initially contained 0. We could have started with the general joint p.d.f.:

$$f_X(x_1,x_2,x_3) \ = \ g_X'(x_1,x_2,x_3)$$

Then the first assignment would have resulted in :

$$X_3 \leftarrow X_1 \ \{ \ f_X(x_1,x_2,x_3) \ = \ (\sum_{x_3} g_X'(x_1,x_2,x_3)) \ \delta(x_1=x_3) \ \}$$
$$= \ g_X(x_1,x_2) \ \delta(x_1=x_3)$$

$$\text{where } g_X(x_1,x_2) \ = \ \sum_{x_3} g_X'(x_1,x_2,x_3)$$

The remainder of the example would be as before.

COINFLIP Revisited

We now have all the tools to handle COINFLIP and get the real answer in a systematic way. The annotated main program is:

```
procedure COINFLIP
    I <- 0              { f_I(i) = δ(i=0) }
    call TC(I)          { f_I(i) = g(i) }
    print i,' times.'   { f_I(i) = g(i) }
```

The problem is to determine what the function $g(i)$ looks like. This is, of course, determined by the subroutine TC. We now proceed to the analysis of TC. Assume

that the p.d.f. entering TC is $f_I(i)$.

```
procedure TC(I)
    Y ← RANDOM_ht       { f_I(i) · ( ½δ(y=H) + ½δ(y=T)) }
    if Y = T then       { f_I(i) · ½δ(y=T) }
        print 'OK, so far!'
        I ← I + 1       { f_I(i-1) · ½δ(y=T) }
        call TC(I)      { g'_I(i) }
    end if
                        { g'_I(i) + f_I(i) · ½δ(y=H) }
end TC
```

Where $g'_I(i)$ represents the value of I returned by the recursive call to TC. Now, the distribution $\{\ f_I(i-1)\ \frac{1}{2}\delta(y=T)\ \}$ is presented to the next call of TC(I), so we must have in general:

$$f_I(i) = f_I(i-1) \cdot \tfrac{1}{2}\delta(y=T)$$

Since the variable Y is local to TC(I), it must be eliminated from the joint p.d.f. that is returned. We will refer to this process as "killing" a variable. This is done by finding the marginal p.d.f. of I with respect to y:

$$f_I(i) = \sum_y f_I(i-1)\ \tfrac{1}{2}\delta(y=T) = \tfrac{1}{2}f_I(i-1)$$

Note that if Y were to be treated as a global variable, this step would take place as part of the RANDOM_ht assignment statement. The initial condition from the main program is $f_I(i) = \delta(i=0)$, so the distribution for the first recursive call is:

$$f_I(i) = \tfrac{1}{2}\delta(i-1 = 0) = \tfrac{1}{2}\delta(i=1)$$

and in general we see that

$$f_I(i) = (\tfrac{1}{2})^j \, \delta(i=j)$$

where j is the number of times that 'OK, so far!' has been printed out. This distribution represents the part of the distribution which is "caught in the loop". Each time some of the distribution "escapes". This corresponds to the chance that Heads will turn up at any time. For each value of j, the joint p.d.f. that "escapes" is $(\tfrac{1}{2})^j \delta(i=j)\tfrac{1}{2}\delta(y=H)$, this joins the rest at the end if to give the final answer:

$$g(i) = \frac{1}{2} \sum_j (\tfrac{1}{2})^j \delta(i=j), \quad j \in \{ 0, 1, 2, \dots \}$$

We note that this is in fact a normalized p.d.f. What is the expected value of I?

$$E(I) = \sum_i \frac{1}{2} i \sum_j (\tfrac{1}{2})^j \delta(i=j) \qquad i,j \in \{ 0, 1, 2, \dots \}$$

$$= \frac{1}{2}(0\cdot 1 + 1\cdot\frac{1}{2} + 2\cdot(\tfrac{1}{2})^2 + \dots\dots\dots$$

by distributing and regrouping each fraction we get:

$$= \frac{1}{2}(\frac{1}{2} + \frac{2}{4} + \frac{3}{8} + \frac{4}{16} + \dots\dots\dots$$

$$= \frac{1}{2}(\frac{1}{2} + \frac{1}{4} + \dots + \frac{1}{4} + \frac{1}{8} + \dots + \frac{1}{8} + \frac{1}{16} + \dots$$

$$= \frac{1}{2}(1 + \frac{1}{2} + \frac{1}{4} + \dots\dots\dots$$

$$= \frac{1}{2}(2) = 1$$

If we had performed this analysis on Ramshaw's [5] version of COINFLIP,

    C ← 0;

    loop X ← RANDOM$_{ht}$; C ← C + 1; while X=T repeat

we would have gotten the final joint p.d.f.:

$$\delta(x=H) \sum_j (\tfrac{1}{2})^j \delta(c=j) , \quad j \in \{ 1,2,3,\dots \}$$

This contains all of the information that is in Ramshaw's output assertion for the same problem [5, p.78]

$$[\text{Fr}(C<1)=0] \wedge [\text{Fr}(X=T)=0] \wedge \bigwedge_{c \geq 1} [\text{Fr}(C=c,X=H) = .2^{-C}]$$

FINDMAX Revisited

We will again follow Ramshaw [5, p.81] and use a slightly different form of the FINDMAX program than was presented in Chapter 3. We will replace the input array A(I) of random variables by repeated calls to a random number generator. This simplifies the notation somewhat without sacrificing generality. We will return to the array notation when we deal with the sorting algorithms. The program is instrumented to record the number of times a new maximum is selected. The modified and annotated program in recursive form is:

```
procedure FINDMAX( N,M )
    C <- 0; I <- 2              { δ(c=0) δ(i=2) }
    M <- RANDOM_f              { δ(c=0) δ(i=2) f(m) }
    call LOOP1 ( N,M,C,I )     { g(n,m,c,i) }
end FINDMAX
procedure LOOP1 (N,M,C,I)      { h(n,m,c,i) }
    if I ≤ N then
                               { h(n,m,c,i) δ(i≤n) }
        T <- RANDOM_f          { h(n,m,c,i) δ(i≤n) f(t) }
        if T>M then            { h(n,m,c,i) δ(i≤n) f(t) δ(t>m) }
            C <- C + 1         { h(n,m,c-1,i) δ(i≤n) f(t) δ(t>m) }
            M <- T

                { δ(m=t) (∑_m h(n,m,c-1,i) δ(t>m)) δ(i≤n) f(t) }
```

[else]   { h(n,m,c,i) $\delta(i\leq n)$ f(t) $\delta(t\leq m)$ }
end if

$$\{ \delta(m=t) (\sum_m h(n,m,c-1,i)\delta(t>m))\ \delta(i\leq n) f(t)$$

$$+ h(n,m,c,i)\ \delta(i\leq n)\ f(t)\ \delta(t\leq m) \}$$

I $\leftarrow$ I + 1

$$\{\delta(i-1\leq n)(\delta(m=t)(\sum_m h(n,m,c-1,i-1)\delta(t>m))f(t)$$

$$+ h(n,m,c,i-1)\ f(t)\ \delta(t\leq m)\ ) \}$$

call LOOP1 (N,M,C,I)

$$\{ g(m,n,c,i) \}$$

end if

$$\{ h(n,m,c,i)\delta(i>n) + g(m,n,c,i) \}$$

Note that all of the joint p.d.f. is caught in the loop or recursive calls until I is incremented past N. The recursion which we must solve is:

$$h(n,m,c,i) = \{\delta(i-1\leq n)(\delta(m=t)\ \sum_m h(n,m,c-1,i-1)\delta(t>m))f(t)$$

$$+ h(n,m,c,i-1)\ f(t)\ \delta(t\leq m)\ ) \}$$

T is a local variable to LOOP1 and not sent outside that subroutine so we must "kill" it.

$$h(n,m,c,i) = \sum_t \{\delta(i-1\leq n)(\delta(m=t)(\sum_m h(n,m,c-1,i-1)\delta(t>m))f(t)$$

$$+ h(n,m,c,i-1)\ f(t)\ \delta(t\leq m)\ ) \}$$

At first glance, this recursion doesn't look very useful. To get a handle on what is going on, we will follow the first few iterations of the program. In doing so we will drop the termination delta function. The initial call is made with

$$h(n,m,c,i) = \delta(c=0)\ \cdot f(m)\ \cdot\delta(i=2)$$

Applying the rules we find that

$$h(n,m,c-1,i-1) = \delta(c=1) \cdot f(m) \cdot \delta(i=3)$$

and

$$h(n,m,c,i-1) = \delta(c=0) \cdot f(m) \cdot \delta(i=3)$$

so we have

$$h(n,m,c,i) =$$

$$\delta(i=3) \sum_t \{ \delta(c=1) \cdot \delta(m=t) \cdot (\sum_m f(m) \cdot \delta(t>m)) \cdot f(t)$$
$$+ \delta(c=0) \cdot f(m) \cdot f(t) \cdot \delta(t \leq m) \}$$

$$h(n,m,c,i) = \delta(i=3) \sum_t \{ \delta(c=1) \cdot \delta(m=t) \cdot (F(t)) \cdot f(t)$$
$$+ \delta(c=0) \cdot f(m).f(t).\delta(t \leq m) \}$$

$$h(n,m,c,i) = \delta(i=3) \{ \delta(c=1) \cdot F(m) \cdot f(m) + \delta(c=0) \cdot f(m) \cdot F(m) \}$$

We can rewrite this into an equivalent form

$$h(n,m,c,i) = \delta(i=3) \{ 2 \cdot F(m) \cdot f(m) (\tfrac{1}{2}\delta(c=1) + \tfrac{1}{2}\delta(c=0)) \}$$

If we crank through another iteration we get:

$$h(n,m,c,i) =$$

$$\delta(i=4) \{ 3 \cdot F^2(m) \cdot f(m) \cdot (\tfrac{1}{6}\delta(c=2) + \tfrac{1}{2}\delta(c=1) + \tfrac{1}{3}\delta(c=0)) \}$$

The third time around we get:

$$h(n,m,c,i) =$$

$$\delta(i=5) \{ 4F^3(m) f(m) (\tfrac{1}{24}\delta(c=3) + \tfrac{1}{4}\delta(c=2) + \tfrac{11}{24}\delta(c=1) + \tfrac{1}{4}\delta(c=0))$$

Each time that we cycle through the equations we find that
the joint p.d.f. is a product of the marginal p.d.f.s of the
individual variables. We have factored the coefficients to
normalize the marginal p.d.f.s with respect to m and c.
When the joint p.d.f. of a set of random variables can be
written as the product of their respective marginal p.d.f.s,

4-53

then the variables are said to be stochastically independent. This is a very important thing for us to confirm in this case. It tells us that we have not affected the distribution of the maximum value by instrumenting the program. The stochastic independence also simplifies the solution of the recurrence relations. Because of it we can set up a recursion for each variable separately by following the marginal p.d.f. for each variable. We change the induction variable from i to $j = i - 1$ so that the formulas will look more familiar.

$$f_M(m)_j = \frac{j}{j-1} F(m) \ f_M(m)_{j-1}$$

and

$$f_C(c)_j = \frac{1}{j} f_C(c-1)_{j-1} + \frac{j-1}{j} f_C(c)_{j-1}$$

The recursion for $f_M(m)$ gives the final distribution of

$$f_M(m)_n = n \cdot F^{n-1}(m) \cdot f(m)$$

which is the answer given by Hogg [12]. The recursion for $f_C(c)$ is the same as Knuth's [6] and Ramshaw's [5].

CHAPTER 5

APPLICATION TO SORTING AND SEARCHING

We now turn our attention to the further application of
our approach to sorting and searching algorithms. We will
look at three such algorithms: The "oblivious" Insertion
(Bubble) Sort, the "improved" Insertion Sort, and Binary
Search.

"Oblivious" Insertion Sort

Insertion Sort was used by Wegbreit [2] as the example
for verifying program performance. He used the "improved"
version which has an exit in the inner loop after each
candidate element is properly positioned. The "oblivious"
version of this program does not have this exit. It con-
tinues to compare the element being inserted to all of the
elements in the sorted sublist. While it is an inefficient
software algorithm, this version of the algorithm is of
interest because it can be realized using a network of com-
parators (i.e. using hardware logic circuits).

```
1     procedure INSERTION SORT ( B , N )
2         real B(1:N)
3     OUTER:
          for J <- 1 to N-1 do
4     INNER:
              for I <- J to 1 by -1 do
5                 if B(I) > B(I+1) then
6                     EXCHANGE ( B(I), B(I+1) )
7                 endif
8             repeat
9         repeat
10    end INSERTION SORT
```

The first step is to convert the loops to recursive subroutine calls. We will number the statements so that they may be related back to the original program. We will also insert a counter variable, Y, to keep track of the number of times an EXCHANGE takes place.

```
1     procedure INSERTION SORT ( B , N )
2         real B(1:N)
          global integer Y
3a        J <- 1; Y <- 0
3b        call OUTER( J, N-1, B )
10    end INSERTION SORT

3c    procedure OUTER( J, LIM, B )
3d        if LIM - J >= 0 then
4a            I <- J
4b            call INNER( I, B )
9a            J <- J + 1
9b            call  OUTER( J, LIM, B )
9c        endif
9d    end OUTER
```

```
4c    procedure INNER( I, B )
4d        if I ≥ 1 then
5             if B(I) > B(I+1) then
6                     EXCHANGE ( B(I), B(I+1) )
6a                    Y ←- Y + 1
7             endif
8a            I ←- I - 1
8b            call INNER ( I, B )
8c        endif
8d    end INNER
```

Appendix A contains a detailed, line-by-line tracing of the joint p.d.f. which is used in an "average case" analysis. From it we can develop the form which the distribution of a "sorted" list takes. Specifically, we have:

$$\delta(b_N \geq b_{N-1}) \cdots \delta(b_2 \geq b_1) \cdot f'(b_1, b_2, \cdots, b_N),$$

where $f'(b_1, b_2, \ldots, b_N)$ is some transformation of the initial joint p.d.f. The leading product of Anderson deltas contains the information that the list is sorted. This may seem like a simple thing, but remember that having started with an algorithm and the assertion that it "sorts a list", we have arrived at a form of joint p.d.f. which means "the list is sorted". If we were to give an automatic analyzer an algorithm, and if it came up with a final joint p.d.f. that had this form, the automatic analyzer could say, "this algorithm sorts a list." Conversely, if the analysis does not result in a joint p.d.f. of this form then the analyzer can say, "this algorithm does not sort a list."

When analyzing sorting algorithms, three different

types of input distributions are usually used. These represent the initally sorted list, the initially reverse sorted list, and the initially "random" list. These three sometimes cover the best, worst, and average case execution times, although not necessarily in that order. In some more exotic algorithms, there is a more complicated input distribution which leads to the best or worst case behavior. Our approach can be used to determine the best and worst case distributions, although we will not dwell on this. The best case performance for Insertion Sort comes when the EXCHANGE never takes place, and the worst case performance comes when the exchange always takes place.

The work shown in Appendix A, for the average case analysis, suggests the induction hypothesis that if you give INNER, at its call from OUTER, the distribution

$$\delta(i=j) \cdot \delta(j<n) \cdot \delta(j=k) \cdot$$

$$k! \cdot \delta(b_k \geq b_{k-1}) \cdots \delta(b_2 \geq b_1) \cdot f(b_1) \cdot f(b_2) \cdots f(b_N),$$

INNER returns the distribution

$$\delta(i=0) \cdot \delta(j<n) \cdot \delta(j=k) \cdot$$

$$(k+1)! \cdot \delta(b_{k+1} \geq b_k) \cdots \delta(b_2 \geq b_1) \cdot f(b_1) \cdot f(b_2) \cdots f(b_N).$$

In other words, INNER inserts the $k+1^{th}$ element into the sorted list of the first k elements. We are therefore justified in picking as the general form for a joint p.d.f. going into INNER

$$\delta(i=m) \cdot \delta(m \leq j) \cdot \delta(j<n) \cdot$$

$$\delta(b_j \geq b_{j-1}) \cdots \delta(b_2 \geq b_1) \cdot f'(y,b_1,b_2,\ldots,b_j,\ldots,b_N).$$

4-58

Rather than doing that, let us start with a completely general joint p.d.f. $g(j,i,n,y,b_1,b_2,\ldots,b_N)$ after 4c.

After 4d, in the true branch:

$$\delta(i \geq 1) \cdot g(j,i,n,y,b_1,b_2,\ldots,b_N)$$

Sent to 8c, is the false branch:

$$\delta(i=0) \cdot g(j,i,n,y,b_1,b_2,\ldots,b_N)$$

After 5, in the true branch:

$$\delta(i \geq 1) \cdot \delta(b_i > b_{i+1}) \cdot g(j,i,n,y,b_1,b_2,\ldots,b_N)$$

Sent to 7, in the false branch is:

$$\delta(i \geq 1) \cdot \delta(b_{i+1} \geq b_i) \cdot g(j,i,n,y,b_1,b_2,\ldots,b_N)$$

After 6,

$$\delta(i \geq 1) \cdot \delta(b_{i+1} > b_i) \cdot g(j,i,n,y,b_1,b_2,\ldots,b_{i+1},b_i,\ldots,b_N)$$

After 6a,

$$\delta(i \geq 1) \cdot \delta(b_{i+1} > b_i) \cdot g(j,i,n,y-1,b_1,b_2,\ldots,b_{i+1},b_i,\ldots,b_N)$$

After 7,

$$\delta(i \geq 1) \cdot \delta(b_{i+1} \geq b_i) \cdot (\ g(j,i,n,y-1,b_1,b_2,\ldots,b_{i+1},b_i,\ldots,b_N)$$
$$+ \ g(j,i,n,y,b_1,b_2,\ldots,b_i,b_{i+1},\ldots,b_N) \ )$$

After 8a,

$$\delta(i+1 \geq 1) \cdot \delta(b_{i+2} \geq b_{i+1}) \cdot$$
$$(\ g(j,i+1,n,y-1,b_1,b_2,\ldots,b_{i+2},b_{i+1},\ldots,b_N)$$
$$+ \ g(j,i+1,n,y,b_1,b_2,\ldots,b_{i+1},b_{i+2},\ldots,b_N) \ )$$

We have arrived at the recursive calling of INNER, so we must have:

$$g(j,i,n,y,b_1,b_2,\ldots,b_N) =$$
$$\delta(i+1 \geq 1) \cdot \delta(b_{i+2} \geq b_{i+1}) \cdot$$
$$(\ g(j,i+1,n,y-1,b_1,b_2,\ldots,b_{i+2},b_{i+1},\ldots,b_N)$$
$$+ \ g(j,i+1,n,y,b_1,b_2,\ldots,b_{i+1},b_{i+2},\ldots,b_N) \ )$$

From the other parts of the algorithm, we get the boundary conditions

$$\delta(j<N) \cdot \delta(i \leq j)$$

and the initial condition

$$g(j,i,n,y,b_1,b_2,\ldots,b_N) =$$
$$\delta(i=j) \cdot \delta(n=N) \cdot h(y) \cdot \delta(b_j \geq b_{j-1}) \cdots \delta(b_2 \geq b_1) \cdot f(b_1,b_2,\ldots,b_N),$$

assuming that f is symmetric with respect to interchange of variables.

Note that this is a "backward" recursion, i.e. we start with i=j and move backward to the desired answer for i=0. Once we have solved the recursive relationship for INNER (based on i), we can use that to solve the recursive relation for OUTER (based on j), which gives the final answer for the joint p.d.f. Doing this in the general case cannot result in a closed form answer in the usual sense. It is possible to "write down" the general solution for any given N, but the equation would be equivalent to the one that we would get if we were to "unwind" the loops into straight line code. In order to get really useful results, we need to select the form of the joint p.d.f. for the unsorted list.

Once one has selected an initial joint p.d.f., and solved the recursion relations, one has a joint p.d.f. which represents the distributions of the variables at the termination of the algorithm. The distribution of the counter variable is then isolated by summation (integration) over

all the other variables. This marginal p.d.f. is then used
to find the expected value, variance, and other statistics
in the usual manner.

"Improved" Insertion Sort

The relative performance of the "oblivious" insertion
sort can be improved, by noting that the portion of the
joint p.d.f. that fails the test at statement 5, is already
in sorted order.   We can exit from the INNER loop at this
point without affecting the algorithm's ability to sort.
Such "obvious" improvements often have hidden side effects,
but our method will let us prove that the modified algorithm
still sorts.   It also turns out that the distribution of I
will give a direct indication of the algorithm's
performance. For this reason, we will delete the counter
variable Y.

```
1    procedure INSERTION SORT ( B , N )
2        real B(1:N)
3    OUTER:
         for J <- 1 to N-1 do
4    INNER:
             for I <- J to 1 by -1 do
5                if B(I) > B(I+1) then
6                    EXCHANGE ( B(I), B(I+1) )
6a               else exit /* This is the change */
7                endif
8            repeat
9        repeat
10   end INSERTION SORT
```

The recursive equivalent is:

```
1     procedure INSERTION SORT ( B , N )
2         real B(1:N)
3a        J <- 1
3b        call OUTER( J, N-1, B )
10    end INSERTION SORT
3c    procedure OUTER( J, LIM, B )
3d        if LIM - J > 0 then
4a            I <- J
4b            call INNER( I, B )
9a            J <- J + 1
9b            call  OUTER( J, LIM, B )
9c        endif
9d    end OUTER
4c    procedure INNER( I, B )
4d        if I > 1 then
5             if B(I) > B(I+1) then
6                     EXCHANGE ( B(I), B(I+1) )
6a            else return
7             endif
8a            I <- I - 1
8b            call INNER ( I, B )
8c        endif
8d    end INNER
```

The return in the recursive program is equivalent to
the exit in the loop version. Everything works the same as
before up to statement 6a. At this point, the joint p.d.f.
from the false branch "escapes" from INNER. We will pick up
the analysis at that point on the J=1 iteration.

5   This is the first test involving the data itself. This
     statement splits the joint p.d.f. on the basis of the
     values of B(I) and B(I+1).

4-62

In the true branch:

$$\delta(i \geq 1) \cdot \delta(i=j) \cdot \delta(j<n) \cdot \delta(j=1) \cdot \delta(b_1 > b_2) \cdot$$
$$f(b_1) \cdot f(b_2) \cdots f(b_N)$$

In the false branch:

$$\delta(i \geq 1) \cdot \delta(i=j) \cdot \delta(j<n) \cdot \delta(j=1) \cdot \delta(b_2 \geq b_1) \cdot$$
$$f(b_1) \cdot f(b_2) \cdots f(b_N)$$

6   This EXCHANGEs the values of b_2 and b_1

$$\delta(i \geq 1) \cdot \delta(i=j) \cdot \delta(j<n) \cdot \delta(j=1) \cdot \delta(b_2 > b_1) \cdot$$
$$f(b_2) \cdot f(b_1) \cdots f(b_N)$$

6a   This sends the false branch joint p.d.f. back to OUTER.

$$\delta(i \geq 1) \cdot \delta(i=j) \cdot \delta(j<n) \cdot \delta(j=1) \cdot \delta(b_2 \geq b_1) \cdot$$
$$f(b_1) \cdot f(b_2) \cdots f(b_N)$$

It is accumulated there as we shall see.

7   At the join for the if statement we have only the true branch left

$$\delta(i \geq 1) \cdot \delta(i=j) \cdot \delta(j<n) \cdot \delta(j=1) \cdot \delta(b_2 > b_1) \cdot$$
$$f(b_1) \cdot f(b_2) \cdots f(b_N)$$

8a   This adjusts I for the next iteration

$$\delta(i+1 \geq 1) \cdot \delta(i+1=j) \cdot \delta(j<n) \cdot \delta(j=1) \cdot \delta(b_2 > b_1)$$
$$\cdot f(b_1) \cdot f(b_2) \cdots f(b_N)$$

8b   We know from step 4d above, that this joint p.d.f. will be returned with the additional (superfluous) restriction $\delta(i<1)$. Simplifying we have

$$\delta(i=0) \cdot \delta(j<n) \cdot \delta(j=1) \cdot \delta(b_2 > b_1) \cdot f(b_1) \cdot f(b_2) \cdots f(b_N)$$

This joint p.d.f. is returned at 4b. It joins with joint p.d.f. that "escaped".

The result is:

$$\{\delta(i=1)+\delta(i=0)\} \cdot \delta(j<n) \cdot \delta(j=1) \cdot \delta(b_2 \geq b_1) \cdot f(b_1) \cdot f(b_2) \cdots f(b_N)$$

9a   This statement adjusts J for the next iteration, and

$$\{\delta(i=1)+\delta(i=0)\} \cdot \delta(j-1<n) \cdot \delta(j-1=1) \cdot \delta(b_2 \geq b_1)$$
$$\cdot f(b_1) \cdot f(b_2) \cdots f(b_N)$$

is again passed to OUTER.

3d   We see now that this test "traps" all of the joint
p.d.f. in the loop until J exceeds LIM (N-1 in our
case). So we won't mention the false branch until the
end.   In the true branch:

$$\{\delta(i=1)+\delta(i=0)\} \cdot \delta(j<n) \cdot \delta(j=2) \cdot \delta(b_2 \geq b_1) \cdot$$
$$f(b_1) \cdot f(b_2) \cdots f(b_N)$$

4a   This collapses the old joint p.d.f. on i and results in

$$\delta(i=j) \cdot \delta(j<n) \cdot \delta(j=2) \cdot 2 \cdot \delta(b_2 \geq b_1) \cdot f(b_1) \cdot f(b_2) \cdots f(b_N)$$

In the oblivious version, this was a trivial operation.
Here it destroys information about the distribution of
the I in the last iteration.

4d   This joint p.d.f. arrives at INNER, where this
statement controls the exit of the last of the joint
p.d.f.

5   In the true branch:

$$\delta(i=j) \cdot \delta(j<n) \cdot \delta(j=2) \cdot 2 \cdot \delta(b_2 \geq b_1) \cdot \delta(b_2 > b_3) \cdot$$
$$f(b_1) \cdot f(b_2) \cdots f(b_N)$$

In the false branch:

$$\delta(i=j) \cdot \delta(j<n) \cdot \delta(j=2) \cdot 2 \cdot \delta(b_2 \geq b_1) \cdot \delta(b_3 \geq b_2) \cdot$$
$$f(b_1) \cdot f(b_2) \cdots f(b_N)$$

6   The exchange yields:

$$\delta(i=j) \cdot \delta(j<n) \cdot \delta(j=2) \cdot 2 \cdot \delta(b_3 \geq b_1) \cdot \delta(b_3 > b_2) \cdot$$
$$f(b_1) \cdot f(b_2) \cdots f(b_N)$$

6a   Here the false branch again escapes in the form of

$$\delta(i=2) \cdot \delta(j<n) \cdot \delta(j=2) \cdot 2 \cdot \delta(b_2 \geq b_1) \cdot \delta(b_3 \geq b_2) \cdot$$
$$f(b_1) \cdot f(b_2) \cdots f(b_N)$$

7   At the join we have only the true branch joint p.d.f. left:

$$\delta(i=j) \cdot \delta(j<n) \cdot \delta(j=2) \cdot 2 \cdot \delta(b_3 \geq b_1) \cdot \delta(b_3 > b_2) \cdot$$
$$f(b_1) \cdot f(b_2) \cdots f(b_N)$$

8a   Prepares for the next call of INNER

$$\delta(i=j-1) \cdot \delta(j<n) \cdot \delta(j=2) \cdot 2 \cdot \delta(b_3 \geq b_1) \cdot \delta(b_3 > b_2) \cdot$$
$$f(b_1) \cdot f(b_2) \cdots f(b_N)$$

This gets through to statement 5 in INNER.

5   In the true branch (multiply by $\delta(b_1 > b_2)$ and simplify):

$$\delta(i=j-1) \cdot \delta(j<n) \cdot \delta(j=2) \cdot$$
$$2 \cdot \{\delta(b_1 > b_2) \cdot \delta(b_3 \geq b_1) \cdot \delta(b_3 > b_2)\}$$
$$\cdot f(b_1) \cdot f(b_2) \cdots f(b_N)$$

In the false branch (multiply by $\delta(b_2 \geq b_1)$, simplify):

$$\delta(i=j-1) \cdot \delta(j<n) \cdot \delta(j=2)$$
$$\cdot 2 \cdot \{\delta(b_3 \geq b_2) \cdot \delta(b_2 \geq b_1)\}$$
$$\cdot f(b_1) \cdot f(b_2) \cdots f(b_N)$$

6   The EXCHANGE in the true branch yields:

$$\delta(i=j-1) \cdot \delta(j<n) \cdot \delta(j=2) \cdot$$
$$2 \cdot \{\delta(b_3 \geq b_2) \cdot \delta(b_2 \geq b_1)\} \cdot f(b_1) \cdot f(b_2) \cdots f(b_N)$$

4-65

6a   Again the false branch joint p.d.f. escapes

$$\delta(i=1)\cdot\delta(j<n)\cdot\delta(j=2)$$
$$\cdot 2\cdot\{\delta(b_3\geq b_2)\cdot\delta(b_2\geq b_1)\}$$
$$\cdot f(b_1)\cdot f(b_2)\cdots f(b_N)$$

7   At the join we have only the true branch joint p.d.f. left:

$$\delta(i=j-1)\cdot\delta(j<n)\cdot\delta(j=2)\cdot$$
$$2\cdot\{\delta(b_3\geq b_2)\cdot\delta(b_2\geq b_1)\}\cdot f(b_1)\cdot f(b_2)\cdots f(b_N)$$

8a   Sets I to zero in this case, and the next call of INNER returns this joint p.d.f.

$$\delta(i=0)\cdot\delta(j<n)\cdot\delta(j=2)\cdot$$
$$2\cdot\{\delta(b_3\geq b_2)\cdot\delta(b_2\geq b_1)\}\cdot f(b_1)\cdot f(b_2)\cdots f(b_N)$$

to OUTER at statement 9a.

4b   The three sets of joint p.d.f.s meet and are added here. We have:

$$\{\delta(i=0)+\delta(i=1)+\delta(i=2)\}\cdot\delta(j<n)\cdot\delta(j=2)\cdot$$
$$2\cdot\{\delta(b_3\geq b_2)\cdot\delta(b_2\geq b_1)\}\cdot f(b_1)\cdot f(b_2)\cdots f(b_N)$$

9a   Increments J and we get, going back into OUTER at 9b:

$$\{\delta(i=0)+\delta(i=1)+\delta(i=2)\}\cdot\delta(j<n+1)\cdot\delta(j=3)\cdot$$
$$2\cdot\{\delta(b_3\geq b_2)\cdot\delta(b_2\geq b_1)\}\cdot f(b_1)\cdot f(b_2)\cdots f(b_N)$$

By now the pattern is clear. It is even easier to show that the result at the end will be:

$$\{\delta(i=0)+\delta(i=1)+\ldots+\delta(i=N-1)\}\cdot\delta(j=N)\cdot$$
$$(N-1)!\cdot\{\delta(b_N\geq b_{N-1})\cdots\delta(b_2\geq b_1)\}\cdot f(b_1)\cdot f(b_2)\cdots f(b_N)$$

If we collapse this on i, then we get the same result as before. Therefore, the change in the program has not

4-66

changed its ability to sort. This form tells us some other things. Specifically, the value of I that is returned by INNER represents the number of elements that were found to be smaller than the $J+1^{th}$ element. It is easy to see that I can take on exactly $J+1$ values from 0 to J, and that each of those values is equally likely. This is something that one would have expected, but we have proved it without recourse to any elaborate combinatorial or probabilistic arguments. The result just "fell out" of the analysis. It is easier to write a program that can recognize that the probability density function of a discrete variable has the same value at each point, than to have that program say "Each I is equally likely!"

The other thing that the values and p.d.f. for I tells us is the number of exchanges that take place. From the observation above, we get that $P(i=j) = \frac{1}{j+1}$ so that the expected number of exchanges for any value of i is

$$\sum_{i=0}^{j} \frac{i}{j+1} = \frac{j}{2}$$

for the entire N elements, this is

$$\sum_{j=1}^{N-1} \frac{j}{2} = \frac{(N^2-N)}{4}$$

which is the correct answer. This turns out to be the expected number of comparisons, also. We can see that the running time performance of the sort has been improved by a factor of two.

Binary Search

We now turn our attention to the analysis of an algorithm for a Binary Search. This particular version closely follows one given by Horowitz and Sahni [9]. We introduce it here for two reasons: (1) it gives us a chance to present the case statement, and (2) it is the first "divide and conquer" algorithm that we have considered. The function INT returns the INTeger part of the argument (i.e. the floor function).

```
1      procedure BINARY_SEARCH ( N, I, X )
           global real K(1:N)
2          LOW <- 1; UP <- N
3          I <- 0
4      SPLIT:while LOW < UP do
5              MID <- INT ( ( LOW + UP ) / 2 )
6              case
7                  : X > K(MID) : LOW <- MID + 1
8                  : X = K(MID) : I <- MID; return
9                  : X < K(MID) : UP <- MID - 1
10             end
11         end
12     end BINARY_SEARCH
```

The recursive equivalent is:

```
1      procedure BINARY_SEARCH ( N, I, X )
           global real K(1:N)
2          LOW <- 1; UP <- N
3          I <- 0
4a         call SPLIT ( LOW, UP, X, I )
12     end BINARY_SEARCH
4b     procedure SPLIT( LOW, UP, X, I )
4c         if LOW < UP then
5              MID <- INT ( ( LOW + UP ) / 2 )
6              case
7                  : X > K(MID) : LOW <- MID + 1
8                  : X = K(MID) : I <- MID; return
9                  : X < K(MID) : UP <- MID - 1
10             end
11a            call SPLIT ( LOW, UP, X, I )
11b        endif
11c        return
11d    end SPLIT
```

Since it is very straight forward, we will just sketch the analysis. We start with the array K(1:N) ordered, so we have the initial joint p.d.f.

$$\delta(k_1 < k_2) \cdot \delta(k_2 < k_3) \cdots \delta(k_{n-1} < k_n) \cdot f(k_1) \cdot f(k_2) \cdots f(k_n)$$

The search key X is drawn from a p.d.f. g(x), and the assignment statements 2 and 3 have their usual effect. As a result we have SPLIT called with the joint p.d.f.

$$\delta(low=1) \cdot \delta(up=N) \cdot \delta(i=0) \cdot g(x) \cdot$$
$$\delta(k_1 < k_2) \cdot \delta(k_2 < k_3) \cdots \delta(k_{n-1} < k_n) \cdot f(k_1) \cdot f(k_2) \cdots f(k_n)$$

After 4c

$$\delta(low \leq up) \cdot \delta(low=1) \cdot \delta(up=N) \cdot \delta(i=0) \cdot g(x) \cdot$$
$$\delta(k_1 < k_2) \cdot \delta(k_2 < k_3) \cdots \delta(k_{n-1} < k_n) \cdot f(k_1) \cdot f(k_2) \cdots f(k_n)$$

After 5

$$\delta(mid = \lfloor (1+N)/2 \rfloor) \cdot \delta(low \leq up) \cdot$$
$$\delta(low=1) \cdot \delta(up=N) \cdot \delta(i=0) \cdot g(x) \cdot$$
$$\delta(k_1 < k_2) \cdot \delta(k_2 < k_3) \cdots \delta(k_{n-1} < k_n) \cdot f(k_1) \cdot f(k_2) \cdots f(k_n)$$

At 6 the joint p.d.f. splits into three parts with the arms of the case statement. The middle leg allows a portion of the joint p.d.f. to escape back to the calling program.

After 7

$$\delta(x > k_{mid}) \cdot \delta(mid = \lfloor (1+N)/2 \rfloor) \cdot$$
$$\delta(low=mid+1) \cdot \delta(up=N) \cdot \delta(i=0) \cdot g(x) \cdot$$
$$\delta(k_1 < k_2) \cdot \delta(k_2 < k_3) \cdots \delta(k_{n-1} < k_n) \cdot f(k_1) \cdot f(k_2) \cdots f(k_n)$$

After 8

$$\delta(x = k_{mid}) \cdot \delta(mid = \lfloor (1+N)/2 \rfloor) \cdot \delta(low=1) \cdot \delta(up=N) \cdot \delta(i=mid) \cdot g(x) \cdot$$
$$\delta(k_1 < k_2) \cdot \delta(k_2 < k_3) \cdots \delta(k_{n-1} < k_n) \cdot f(k_1) \cdot f(k_2) \cdots f(k_n)$$

After 9

$$\delta(x < k_{mid}) \cdot \delta(mid = \lfloor (1+N)/2 \rfloor) \cdot$$

$$\delta(low=1) \cdot \delta(up=mid-1) \cdot \delta(i=0) \cdot g(x) \cdot$$

$$\delta(k_1 < k_2) \cdot \delta(k_2 < k_3) \cdots \delta(k_{n-1} < k_n) \cdot f(k_1) \cdot f(k_2) \cdots f(k_n)$$

The sum of the joint p.d.f. after 7 and after 9 is presented to the next call on SPLIT. Each time SPLIT is called, some of the joint p.d.f. escapes and is returned, until the final return for no find. It is relatively easy to see that the final joint p.d.f. will be

$$[\ \delta(i=0) \ \{ \ \delta(x < k_1) + \delta(x > k_1)\delta(x < k_2) + \ldots + \delta(x > k_n) \ \} \ +$$

$$\sum_{mid=1}^{n} (\ \delta(i=mid) \ \delta(x = k_{mid})) \ ]$$

$$\cdot g(x) \cdot \delta(k_1 < k_2) \cdot \delta(k_2 < k_3) \cdots \delta(k_{n-1} < k_n) \cdot f(k_1) \cdot f(k_2) \cdots f(k_n)$$

The behavior of this joint p.d.f. is dependent on the form of $g(x)$. If this p.d.f. restricts the value of x to those of the K(M) with equal probability, then we see that any of the values is equally likely. The behavior of the number of comparisons can be derived by instrumenting the algorithm. Doing so results in the usual log n behavior.

CHAPTER 6

APPLICATION TO A MISCELLANEOUS PROBLEM

We will now look at Jonassen's and Knuth's celebrated "Trivial Algorithm Whose Analysis Isn't" [8]. Ramshaw, a student of Knuth's, applies his Frequentistic System to this algorithm in his thesis [5]. Jonassen and Knuth did not give the derivation of the initial recursion relationships, but derived them "by reasoning almost directly from the code of the program" [5]. We now believe that our work has formalized this "reasoning almost directly from the code", because, when applied to this algorithm, it proceeds directly to their equations 2.1, 2.2, and 2.3 [8].

Basically the algorithm involves the insertion and deletion of keys in a binary tree structure. The insertion is done with the standard binary insertion algorithm and the deletion is done using Hibbard's algorithm[18]. The two possible trees with two keys are called F and G. The five possible binary trees with three keys are called A, B, C, D, and E. With x < y < z, we have the following pictures for these binary trees:

```
A(x,y,z)      B(x,y,z)      C(x,y,z)      D(x,y,z)      E(x,y,z)
    z             z             y             x             x
   /             /             / \            \             \
  y             x             x   z            z             y
 /               \               /              \
x                 y             y                z

        F(x,y)                        G(x,y)
          y                             x
         /                               \
        x                                 y
```
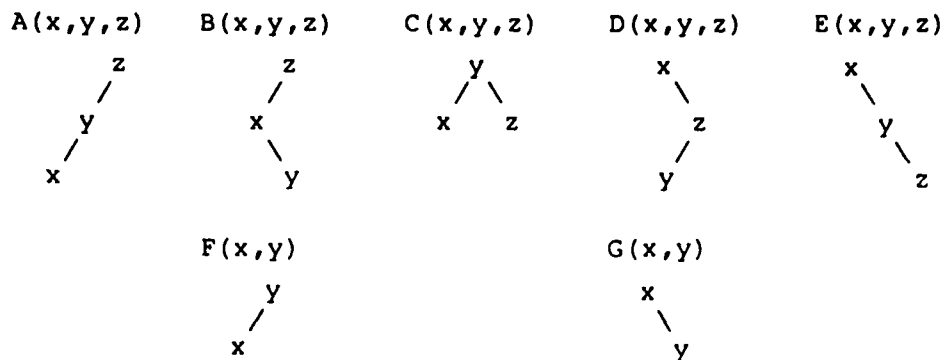
The insertion algorithm is the standard one for binary insertion, the new element is appended to the tree in the appropriate place. Hibbard's deletion algorithm proceeds in a straight-forward manner except that the deletion of x from D(x,y,z) results in G(y,z) instead of F(y,z), as one might expect. The insertion and deletion algorithm is given in detail in the program which follows. We will not go further into the background of the algorithm. Anyone interested should see the Jonassen and Knuth article [8], which does that quite nicely.

While the others [5,8] have always assumed that the keys are selected from a uniform distribution, it turns out that this restriction is unnecessary in our approach. It is only necessary to have the keys drawn from the same, stationary distribution $f(x)$.

Jonassen and Knuth [8] give the graphical and word procedure representation of the algorithm, we will only present the algorithm as a SPARKS program. We will use Ramshaw's [5] notation for the tuples representing the

condition of the tree. Furthermore, we will adopt the convention that *after* assignment the "from" variables are set to zero ("killed"). This is not really necessary, but it does simplify the notation, since after the variables are "killed" we no longer have to carry them in the joint p.d.f. equations.

```
1     procedure TRIVIAL ( N )
      /*  Load the initial tree */
2         X <- random_f; Y <- random_f
3         if ( X < Y ) then
4              <S;V,W> <- <G;X,Y>
5         else
6              <S;V,W> <- <F;Y,X>
7         endif
      /*  The main algorithm loop */
8         for K <- 1 to N
      /*  Insert a key */
9              R <- random_f
10             case
11                 : S = F and R < V      : <T;X,Y,Z> <- <A;R,V,W>
12                 : S = F and V < R < W  : <T;X,Y,Z> <- <B;V,R,W>
13                 : S = F and W < R      : <T;X,Y,Z> <- <C;V,W,R>
14                 : S = G and R < V      : <T;X,Y,Z> <- <C;R,V,W>
15                 : S = G and V < R < W  : <T;X,Y,Z> <- <D;V,R,W>
16                 : S = G and W < R      : <T;X,Y,Z> <- <E;V,W,R>
17             end
      /*  Now do the deletion */
18             L <- random_XYZ
19             case
20                 : T = A and L = X :  <S;V,W> <- <F;Y,Z>
21                 : T = A and L = Y :  <S;V,W> <- <F;X,Z>
22                 : T = A and L = Z :  <S;V,W> <- <F;X,Y>
23                 : T = B and L = X :  <S;V,W> <- <F;Y,Z>
```

```
24              : T = B and L = Y :   <S;V,W> <- <F;X,Z>
25              : T = B and L = Z :   <S;V,W> <- <G;X,Y>
26              : T = C and L = X :   <S;V,W> <- <G;Y,Z>
27              : T = C and L = Y :   <S;V,W> <- <F;X,Z>
28              : T = C and L = Z :   <S;V,W> <- <F;X,Y>
29              : T = D and L = X :   <S;V,W> <- <G;Y,Z>
30              : T = D and L = Y :   <S;V,W> <- <G;X,Z>
31              : T = D and L = Z :   <S;V,W> <- <G;X,Y>
32              : T = E and L = X :   <S;V,W> <- <G;Y,Z>
33              : T = E and L = Y :   <S;V,W> <- <G;X,Z>
34              : T = E and L = Z :   <S;V,W> <- <G;X,Y>
35              end
36         repeat
37     end TRIVIAL
```

The recursive version of this program is then,

```
1      procedure TRIVIAL ( N )
       /*  Load the initial tree */
2          X <- random_f;  Y <- random_f
3          if ( X < Y ) then
4                 <S;V,W> <- <G;X,Y>
5          else
6                 <S;V,W> <- <F;Y,X>
7          endif
       /*  The main algorithm loop */
8a         K <- 1
8b         call MAIN ( K , N )
37     end TRIVIAL


8c     procedure MAIN ( K, N )
8d         if ( K <= N ) then
       /*  Insert a key */
9              R <- random_f
10             case
11             : S = F and R < V       : <T;X,Y,Z> <- <A;R,V,W>
12             : S = F and V < R < W : <T;X,Y,Z> <- <B;V,R,W>
```

4-74

```
13              : S = F and W < R      : <T;X,Y,Z> <- <C;V,W,R>
14              : S = G and R < V      : <T;X,Y,Z> <- <C;R,V,W>
15              : S = G and V < R < W : <T;X,Y,Z> <- <D;V,R,W>
16              : S = G and W < R      : <T;X,Y,Z> <- <E;V,W,R>
17          end
    /*  Now do the deletion */
18          L <- random_XYZ
19          case
20              : T = A and L = X : <S;V,W> <- <F;Y,Z>
21              : T = A and L = Y : <S;V,W> <- <F;X,Z>
22              : T = A and L = Z : <S;V,W> <- <F;X,Y>
23              : T = B and L = X : <S;V,W> <- <F;Y,Z>
24              : T = B and L = Y : <S;V,W> <- <F;X,Z>
25              : T = B and L = Z : <S;V,W> <- <G;X,Y>
26              : T = C and L = X : <S;V,W> <- <G;Y,Z>
27              : T = C and L = Y : <S;V,W> <- <F;X,Z>
28              : T = C and L = Z : <S;V,W> <- <F;X,Y>
29              : T = D and L = X : <S;V,W> <- <G;Y,Z>
30              : T = D and L = Y : <S;V,W> <- <G;X,Z>
31              : T = D and L = Z : <S;V,W> <- <G;X,Y>
32              : T = E and L = X : <S;V,W> <- <G;Y,Z>
33              : T = E and L = Y : <S;V,W> <- <G;X,Z>
34              : T = E and L = Z : <S;V,W> <- <G;X,Y>
35          end
36a         K = K + 1
36b         call MAIN ( K, N )
36c     endif
36d  end MAIN
```

The analysis is as follows:

After 2

$$f(x) \cdot f(y)$$

After 3

$$\delta(x<y) \cdot f(x) \cdot f(y)$$

After 4

$$\delta(s=G) \cdot \delta(v<w) \cdot f(v) \cdot f(w)$$

After 5

$$\delta(x>y) \cdot f(x) \cdot f(y)$$

After 6

$$\delta(s=F) \cdot \delta(v<w) \cdot f(v) \cdot f(w)$$

After 7

$$\{\delta(s=F) + \delta(s=G)\} \cdot \delta(v<w) \cdot f(v) \cdot f(w)$$

After 8a

$$\delta(k=1) \cdot \{\delta(s=F) + \delta(s=G)\} \cdot \delta(v<w) \cdot f(v) \cdot f(w)$$

Which is what we expected, either tree is equally likely, and the joint p.d.f. is that of a sorted list of two variables. Rather than continue to follow an explicit example through the algorithm, as we have done in the past, we will define unknown functions to represent the various tree forms. Following these through the algorithm will result in the recursive equations. Let:

$$\delta(k=K) \cdot \delta(v<w) \cdot \{\delta(s=F) \cdot f_k(v,w) + \delta(s=G) \cdot g_k(v,w)\}$$

represent the joint p.d.f. that is presented to each call of the recursive subroutine MAIN. This form comes from looking ahead and recognizing that no joint p.d.f. "leaks out" until the end of the loop.

After 8d

$$\delta(k\leq N) \cdot \delta(k=K) \cdot \delta(v<w) \cdot \{\delta(s=F) \cdot f_k(v,w) + \delta(s=G) \cdot g_k(v,w)\}$$

After 9

$$\delta(k\leq N) \cdot \delta(k=K) \cdot \delta(v<w) \cdot \{\delta(s=F) \cdot f_k(v,w) + \delta(s=G) \cdot g_k(v,w)\} \cdot f(r)$$

In order to simplify the expressions, we will drop the loop-counting-and-stopping factor $\delta(k\leq N) \cdot \delta(k=K)$. We will

also note that $\delta(s=F) \cdot \delta(s=G) = 0$, and use this in each arm of the case statement.

After 11
$$\delta(s=F) \cdot f_k(v,w) \cdot f(r) \cdot \delta(v<w) \cdot \delta(r<v) \cdot$$
$$\delta(t=A) \cdot \delta(x=r) \cdot \delta(y=v) \cdot \delta(z=w)$$

using the convention of "killing" the old variables,
$$\delta(t=A) \cdot f_k(y,z) \cdot f(x) \cdot \delta(x<y<z)$$

Note that this convention simplifies the assignments to $\langle t;x,y,z \rangle$ because the distributions of these variables is always $\delta(s=0) \cdot \delta(v=0) \cdot \delta(w=0)$ at this point.

After 12
$$\delta(t=B) \cdot f_k(x,z) \cdot f(y) \cdot \delta(x<y<z)$$

After 13
$$\delta(t=C) \cdot f_k(x,y) \cdot f(z) \cdot \delta(x<y<z)$$

After 14
$$\delta(t=C) \cdot g_k(y,z) \cdot f(x) \cdot \delta(x<y<z)$$

After 15
$$\delta(t=D) \cdot g_k(x,z) \cdot f(y) \cdot \delta(x<y<z)$$

After 16
$$\delta(t=E) \cdot g_k(x,y) \cdot f(z) \cdot \delta(x<y<z)$$

After 17

We have the sum of the six arms of the case statement. It is at this point that, by looking ahead, we see that the next general functions should be defined as:

$$a_k(x,y,z) = f_k(y,z) \cdot f(x)$$
$$b_k(x,y,z) = f_k(x,z) \cdot f(y)$$
$$c_k(x,y,z) = f_k(x,y) \cdot f(z) + g_k(y,z) \cdot f(x)$$
$$d_k(x,y,z) = g_k(x,z) \cdot f(y)$$
$$e_k(x,y,z) = g_k(x,y) \cdot f(z)$$

With $f(x)=\delta(0<x<1)$ for a unitary distribution, these are equations 2.1 in Jonassen and Knuth [8].

The whole joint p.d.f. after 17 is then:

$$\{\delta(t=A)\cdot a_k(x,y,z) + \delta(t=B)\cdot b_k(x,y,z) + \delta(t=C)\cdot c_k(x,y,z)$$
$$+ \delta(t=D)\cdot d_k(x,y,z) + \delta(t=E)\cdot e_k(x,y,z)\ \}\ \cdot\ \delta(x<y<z)$$

After 18

$$\{\delta(t=A)\cdot a_k(x,y,z) + \delta(t=B)\cdot b_k(x,y,z) + \delta(t=C)\cdot c_k(x,y,z)$$
$$+ \delta(t=D)\cdot d_k(x,y,z) + \delta(t=E)\cdot e_k(x,y,z)\ \}\ \cdot$$
$$\delta(x<y<z)\ \cdot\ \{\ \tfrac{1}{3}\delta(1=X) + \tfrac{1}{3}\delta(1=Y) + \tfrac{1}{3}\delta(1=Z)\ \}$$

where the last term expresses the fact that any of the keys may be deleted with equal probability.

After 20

$$\delta(t=A)\cdot a_k(x,y,z)\cdot\tfrac{1}{3}\delta(1=X)\cdot\delta(s=F)\cdot\delta(v=y)\cdot\delta(w=z)\cdot\delta(x<y<z)$$

We now apply the convention of setting t,x,y, and z to zero. This is done by "integration" over these variables using Theorem 5. We will use our summation notation, which is defined to work the same as integration if the functions are taken to be continuous. Remember that if a variable of integration appears in an Boolean delta function and is equal to a free variable, then the effect is the same as a change of variable. In this case y and z appear this way, while x appears only with respect to other variables of integration.

$$\sum_{1,t,x,y,z}\{\delta(t=A).a_k(x,y,z).\tfrac{1}{3}\delta(1=X)$$
$$\cdot\delta(s=F)\cdot\delta(v=y)\cdot\delta(w=z)\cdot\delta(x<y<z)\}\ =$$

$$\tfrac{1}{3}\delta(s{=}F).\delta(v{<}w).\sum_{x} a_k(x,v,w).\delta(x{<}v)$$

Do the same thing with the 14 other arms of the case statement.

.
.
.

After 35

$$\delta(v{<}w).[\ \tfrac{1}{3}\delta(s{=}F).\{\ \sum_{x}\ (a_k(x,v,w)\ +\ b_k(x,v,w)\ ).\delta(x{<}v)$$

$$+\sum_{y}\ (a_k(v,y,w)\ +\ b_k(v,y,w)\ +\ c_k(v,y,w)\ ).\delta(v{<}y{<}w)$$

$$+\sum_{z}\ (a_k(v,w,z)\ +\ c_k(v,w,z)\ ).\delta(w{<}z)\ \}$$

$$+\tfrac{1}{3}\delta(s{=}G).\{\ \sum_{x}\ (c_k(x,v,w)\ +\ d_k(x,v,w)\ +\ e_k(x,v,w)).\delta(x{<}v)$$

$$+\sum_{y}\ (d_k(v,y,w)\ +\ e_k(v,y,w)).\delta(v{<}y{<}w)$$

$$+\sum_{z}\ (b_k(v,w,z)\ +\ d_k(v,w,z)\ +\ e_k(v,w,z)).\delta(w{<}z)\ \}\ ]$$

After 36a

The value of k is incremented, and we can identify the terms of the joint p.d.f. after 36a as equal to $f_{k+1}(v,w)$ and $g_{k+1}(v,w)$ respectively. We now have arrived at Jonassen's and Knuth's recursive equations 2.2 [8].

CHAPTER 7

SUMMARY AND CONCLUSIONS

What have we accomplished? We have sketched the foundation for a systematic approach to algorithm analysis that is based on two ideas:

1. Convert all loop constructs within a program to recursive subroutine calls
2. Develop a representation of the initial joint p.d.f. of the program variables, and follow the effects that the program has on that joint p.d.f.

These two ideas yield recurrence relations for the joint p.d.f. which can be solved to get the joint p.d.f. at any point in the execution of the algorithm. The branching probabilities can be calculated directly from the joint p.d.f. at each conditional statement. It is this detailing of the branching probabilities that was missing from the automatic analyzers METRIC and EL/PL. Therefore, the logical next step would be to add this method to the existing analyzers.

The central addition we have made to the understanding of the behavior of joint p.d.f.s in a program is the introduction of the Boolean delta function. This function, by

4-80.

connecting the boolean world of the algorithmic conditional statement to the real numbers, makes it possible to keep track of the effects of conditional statements on the joint p.d.f.s. Its form, essentially a list of arguments, makes it very easy to represent and operate upon in a computer program, especially since LISP seems to be the language most used in this type of work.

Our approach, by capturing the behavior of the program variables in detail, also includes a means for verifying the performance of algorithms. All of the information that can be obtained from previous methods of program verification seems to be present in our method.

Regardless of the underlying simplicity of the ideas, the method is very tedious to apply to any significant algorithm. The examples given in this thesis were made possible by the string manipulation features of a DIGITAL WS/78 Word Processor. The next thing that must be done before more useful work can be done in this area is to automate the technique. This automated processor should be an interactive one in the EL/PL style.

With an automatic processor, investigations can begin into some of the simple program constructs which we have not addressed. Multiplication, division, addition and subtraction of variables have not been considered. Since these are very important parts of many algorithms, this work must be extended to cover them before it becomes really useful.

References

1. Wegbreit, B., "Mechanical Program Analysis", Comm. ACM Vol.18, No.9 (Sept.1975), 528-539.

2. Wegbreit, B., "Verifying Program Performance", J. ACM, Vol.23, No.4 (October 1976), 691-699.

3. Cohen, J. and Zuckerman, C. "Two Languages for Estimating Program Efficiency", Comm. ACM, Vol.17, No.6 (June 1974), 301-308.

4. deFreitas, S.L. and Lavelle, P.J., "A Method for the Time Analysis of Programs", IBM Syst J, Vol.17, No.1 (1978), 26-38.

5. Ramshaw, L.H., "Formalizing the Analysis of Algorithms", Ph.D. Dissertation, Stanford University, 1979.

6. Knuth, D.E., The Art of Computer Programming (Vol.1 and Vol.3). Addison-Wesley, Menlo Park, California, 1968.

7. Horowitz, E. and Sahni, S., Fundamentals of Computer Algorithms. Computer Science Press, Potomac, Maryland, 1978.

8. Jonassen, A.T. and Knuth, D.E., "A Trivial Algorithm Whose Analysis Isn't", Journal of Computer and System Sciences, Vol.16 (1978), 301-322.

9. Horowitz, E. and Sahni, S., Fundamentals of Data Structures. Computer Science Press, Potomac, Maryland, 1976.

10. Aho,A., Hopcroft,J. and Ullman,J., The Design and Analysis of Computer Algorithms. Addison-Wesley, Reading, Massachusetts, 1976.

11. Cohen, J. and Roth, M., "On the Implementation of
    Strassen's Fast Multiplication Algorithm", _Acta
    Informatica_, Vol.6 (1976), 341–355.

12. Hogg, R.V. and Craig, A.T., _Introduction to
    Mathematical Statistics_ (Second Edition).
    Macmillian, New York, 1959.

13. Kodres, U.R., "Discrete Systems and Flowcharts", _IEEE
    Trans. Software Eng._, Vol. SE-4, No. 6 (November
    1978), 521–525.

14. Davies, A.C., "The Analogy Between Electrical Networks
    and Flowcharts", _IEEE Trans. Software Eng._, Vol.
    SE-6, No. 4 (July 1980), 391–394.

15. Hofstadter, D.R., _Gödel, Escher, Bach: an Eternal
    Golden Braid_. Basic Books, New York, 1979.

16. Lueker, G.S., "Some Techniques for Solving
    Recurrences", _Computing Surveys_, Vol.12, No.4
    (December 1980), 419–436.

17. Anderson, R.B., _Proving Programs Correct_. John Wiley &
    Sons, New York, 1979.

18. Hibbard, T.N., "Some Combinatorial Properties of
    Certain Trees with Applications to Searching and
    Sorting", _J. ACM_, Vol. 9 (1962), 13–28.

APPENDIX A

LINE-BY-LINE ANALYSIS

of

"OBLIVIOUS" INSERTION SORT

We must do the analysis for a specific class of initial distributions for the problem to be tractable. Specifically, we will assume that each element of B(1:N) is drawn independently from a well defined, stationary p.d.f. $f(b_i)$. Therefore the initial joint p.d.f. is simply

$$f_B(b_1,b_2,b_3,\ldots,b_N) = f(b_1) \cdot f(b_2) \cdots f(b_N).$$

The converted program is:

```
1    procedure INSERTION SORT ( B , N )
2        real B(1:N)
3a       J <- 1
3b       call OUTER( J, N-1, B )
10   end INSERTION SORT


3c   procedure OUTER( J, LIM, B )
3d       if LIM - J >= 0 then
4a           I <- J
4b           call INNER( I, B )
9a           J <- J + 1
9b           call  OUTER( J, LIM, B )
9c       endif
9d   end OUTER
```

```
4c     procedure INNER( I, B )
4d        if I > 1 then
5             if B(I) > B(I+1) then
6                     EXCHANGE ( B(I), B(I+1) )
7             endif
8a            I <- I - 1
8b            call INNER ( I, B )
8c        endif
8d     end INNER
```

The numbers will refer to the statement numbers of the recursive version of the algorithm.

1   Initial joint p.d.f.

$$f_B(b_1, b_2, b_3, \ldots, b_N) = f(b_1) \cdot f(b_2) \cdot \cdot \cdot f(b_N).$$

3a Adds a new variable

$$\delta(j=1) \cdot f(b_1) \cdot f(b_2) \cdot \cdot \cdot f(b_N).$$

3d Splits the distribution based on the values of J and LIM.

In the **true** branch:

$$\delta(j<n) \cdot \delta(j=1) \cdot f(b_1) \cdot f(b_2) \cdot \cdot \cdot f(b_N).$$

In the **false** branch:

$$\delta(j \geq n) \cdot \delta(j=1) \cdot f(b_1) \cdot f(b_2) \cdot \cdot \cdot f(b_N).$$

We have made the substitutions of the instances of the dummy variables in the routine. Now, if $N = 1$, then the true branch is zero, the false branch reduces to $\delta(j=1) \cdot f(b_1)$, and we are done.

4a Adds a new variable in the true branch

$$\delta(i=j) \cdot \delta(j<n) \cdot \delta(j=1) \cdot f(b_1) \cdot f(b_2) \cdot \cdot \cdot f(b_N).$$

This joint p.d.f. is transfered with the call at 4b.

4d   Splits the distribution based on the value of I.

In the **true** branch:

$$\delta(i\geq1)\cdot\delta(i=j)\cdot\delta(j<n)\cdot\delta(j=1)\cdot f(b_1)\cdot f(b_2)\cdots f(b_N).$$

In the **false** branch:

$$\delta(i<1)\cdot\delta(i=j)\cdot\delta(j<n)\cdot\delta(j=1)\cdot f(b_1)\cdot f(b_2)\cdots f(b_N).$$

5   Finally things get interesting! This is the first test involving the data itself. This statement splits the joint p.d.f. on the basis of the values of B(I) and B(I+1).

In the **true** branch:

$$\delta(i\geq1)\cdot\delta(i=j)\cdot\delta(j<n)\cdot\delta(j=1)\cdot\delta(b_1>b_2)\cdot f(b_1)\cdot f(b_2)\cdots f(b_N).$$

In the **false** branch:

$$\delta(i\geq1)\cdot\delta(i=j)\cdot\delta(j<n)\cdot\delta(j=1)\cdot\delta(b_2\geq b_1)\cdot f(b_1)\cdot f(b_2)\cdots f(b_N).$$

6   This EXCHANGEs the values of b_2 and b_1

$$\delta(i\geq1)\cdot\delta(i=j)\cdot\delta(j<n)\cdot\delta(j=1)\cdot\delta(b_2>b_1)\cdot f(b_2)\cdot f(b_1)\cdots f(b_N).$$

7   At the join for the if statement we have

$$\delta(i\geq1)\cdot\delta(i=j)\cdot\delta(j<n)\cdot\delta(j=1)\cdot$$

$$\{\delta(b_2>b_1)+\delta(b_2\geq b_1)\}\cdot f(b_1)\cdot f(b_2)\cdots f(b_N).$$

It is now that we can see the significance of our choice of initial joint p.d.f. which is symmetric with respect to the exchange of variable indicies.

At this point we must decide whether the probability that $b_i=b_j$ is going to be significant, or not. If we choose to deal with continuous distributions, then this probability is zero. Likewise, if we say that the discrete elements are distinct we have the same thing. We will do this so that we

can write the joined joint p.d.f. as

$$\delta(i \geq 1) \cdot \delta(i=j) \cdot \delta(j<n) \cdot \delta(j=1) \cdot 2 \cdot \delta(b_2 \geq b_1) \cdot$$
$$f(b_1) \cdot f(b_2) \cdots f(b_N)$$

8a This adjusts I for the next iteration

$$\delta(i+1 \geq 1) \cdot \delta(i+1=j) \cdot \delta(j<n) \cdot \delta(j=1) \cdot$$
$$2 \cdot \delta(b_2 \geq b_1) \cdot f(b_1) \cdot f(b_2) \cdots f(b_N)$$

8b We know from step 4d above, that this joint p.d.f. will
be returned with the additional (superfluous)
restriction $\delta(i<1)$. Simplifying we have

$$\delta(i=0) \cdot \delta(j<n) \cdot \delta(j=1) \cdot 2 \cdot \delta(b_2 \geq b_1) \cdot f(b_1) \cdot f(b_2) \cdots f(b_N)$$

This joint p.d.f. is returned at 4b.

9a This statement adjusts J for the next iteration, and

$$\delta(i=0) \cdot \delta(j-1<n) \cdot \delta(j-1=1) \cdot 2 \cdot \delta(b_2 \geq b_1) \cdot f(b_1) \cdot f(b_2) \cdots f(b_N)$$

is again passed to OUTER.

3d We see now that this test "traps" all of the joint
p.d.f. in the loop until J exceeds LIM (N-1 in our
case). So we won't mention the **false** branch until the
end.

In the true branch:

$$\delta(j<n) \cdot \delta(i=0) \cdot \delta(j-1<n) \cdot \delta(j-1=1) \cdot$$
$$2 \cdot \delta(b_2 \geq b_1) \cdot f(b_1) \cdot f(b_2) \cdots f(b_N)$$

4a This collapses the old joint p.d.f. on i and results in

$$\delta(i=j) \cdot \delta(j<n) \cdot \delta(j=2) \cdot 2 \cdot \delta(b_2 \geq b_1) \cdot f(b_1) \cdot f(b_2) \cdots f(b_N)$$

We have simplified the expression with respect to j.

4d This joint p.d.f. arrives at INNER, where this
statement traps the joint p.d.f. until I<1.

4-87

5   In the true branch:

$$\delta(i=j)\cdot\delta(j<n)\cdot\delta(j=2)\cdot 2\cdot\delta(b_2\geq b_1)\cdot\delta(b_2>b_3)\cdot$$

$$f(b_1)\cdot f(b_2)\cdots f(b_N)$$

In the false branch:

$$\delta(i=j)\cdot\delta(j<n)\cdot\delta(j=2)\cdot 2\cdot\delta(b_2\geq b_1)\cdot\delta(b_3\geq b_2)\cdot$$

$$f(b_1)\cdot f(b_2)\cdots f(b_N)$$

6   The exchange yields:

$$\delta(i=j)\cdot\delta(j<n)\cdot\delta(j=2)\cdot 2\cdot\delta(b_3\geq b_1)\cdot\delta(b_3>b_2)\cdot$$

$$f(b_1)\cdot f(b_2)\cdots f(b_N)$$

7   At the join we have:

$$\delta(i=j)\cdot\delta(j<n)\cdot\delta(j=2)\cdot$$

$$2\cdot\{\delta(b_2\geq b_1)\cdot\delta(b_3\geq b_2)+\delta(b_3\geq b_1)\cdot\delta(b_3>b_2)\}\cdot$$

$$f(b_1)\cdot f(b_2)\cdots f(b_N)$$

8a   Prepares for the next call of INNER

$$\delta(i=j-1)\cdot\delta(j<n)\cdot\delta(j=2)\cdot$$

$$2\cdot\{\delta(b_2\geq b_1)\cdot\delta(b_3\geq b_2)+\delta(b_3\geq b_1)\cdot\delta(b_3>b_2)\}\cdot$$

$$f(b_1)\cdot f(b_2)\cdots f(b_N)$$

This gets through to statement 5 in INNER.

5   In the true branch(multiply by $\delta(b_1>b_2)$ and simplify):

$$\delta(i=j-1)\cdot\delta(j<n)\cdot\delta(j=2)\cdot$$

$$2\cdot\{\delta(b_1>b_2)\cdot\delta(b_3\geq b_1)\cdot\delta(b_3>b_2)\}\cdot f(b_1)\cdot f(b_2)\cdots f(b_N)$$

In the false branch(multiply by $\delta(b_2\geq b_1)$ and simplify):

$$\delta(i=j-1)\cdot\delta(j<n)\cdot\delta(j=2)\cdot$$

$$2\cdot\{\delta(b_2\geq b_1)\cdot\delta(b_3\geq b_2)+\delta(b_2\geq b_1)\cdot\delta(b_3\geq b_1)\cdot\delta(b_3>b_2)\}\cdot$$

$$f(b_1)\cdot f(b_2)\cdots f(b_N)\ =$$

$$\delta_{(i=j-1)} \cdot \delta_{(j<n)} \cdot \delta_{(j=2)} \cdot$$
$$2 \cdot \{2 \cdot \delta_{(b_3 \geq b_2)} \cdot \delta_{(b_2 \geq b_1)}\} \cdot f(b_1) \cdot f(b_2) \cdots f(b_N)$$

6   The EXCHANGE in the true branch yields:
$$\delta_{(i=j-1)} \cdot \delta_{(j<n)} \cdot \delta_{(j=2)} \cdot$$
$$2 \cdot \{\delta_{(b_2 > b_1)} \cdot \delta_{(b_3 \geq b_2)} \cdot \delta_{(b_3 > b_1)}\} \cdot f(b_1) \cdot f(b_2) \cdots f(b_N)$$

$$\delta_{(i=j-1)} \cdot \delta_{(j<n)} \cdot \delta_{(j=2)} \cdot$$
$$2 \cdot \{\delta_{(b_3 \geq b_2)} \cdot \delta_{(b_2 \geq b_1)}\} \cdot f(b_1) \cdot f(b_2) \cdots f(b_N)$$

7   At the join we have:
$$\delta_{(i=j-1)} \cdot \delta_{(j<n)} \cdot \delta_{(j=2)} \cdot$$
$$2 \cdot \{3 \cdot \delta_{(b_3 \geq b_2)} \cdot \delta_{(b_2 \geq b_1)}\} \cdot f(b_1) \cdot f(b_2) \cdots f(b_N)$$

8a   Sets I to zero in this case, and the next call of INNER
      returns this joint p.d.f.
$$\delta_{(i=0)} \cdot \delta_{(j<n)} \cdot \delta_{(j=2)} \cdot$$
$$2 \cdot \{3 \cdot \delta_{(b_3 \geq b_2)} \cdot \delta_{(b_2 \geq b_1)}\} \cdot f(b_1) \cdot f(b_2) \cdots f(b_N)$$

to OUTER at statement 9a.

This suggests the induction hypothesis that if you give INNER, at its call from OUTER, the distribution
$$\delta_{(i=j)} \cdot \delta_{(j<n)} \cdot \delta_{(j=k)} \cdot$$
$$k! \cdot \delta_{(b_k \geq b_{k-1})} \cdots \delta_{(b_2 \geq b_1)} \cdot f(b_1) \cdot f(b_2) \cdots f(b_N)$$

it returns the distribution
$$\delta_{(i=0)} \cdot \delta_{(j<n)} \cdot \delta_{(j=k)} \cdot$$
$$(k+1)! \cdot \delta_{(b_{k+1} \geq b_k)} \cdots \delta_{(b_2 \geq b_1)} \cdot f(b_1) \cdot f(b_2) \cdots f(b_N)$$

This can be shown to be true in a straight-forward, if somewhat tedious, manner.

OUTER's "loop-stopper" releases this joint p.d.f. when

J=N and we have the result:

$$\delta(i=0) \cdot \delta(j=N) \cdot N! \cdot \delta(b_N \geq b_{N-1}) \cdots \delta(b_2 \geq b_1) \cdot$$

$$f(b_1) \cdot f(b_2) \cdots f(b_N)$$

This is precisely the proper answer which is usually derived using combinatorial arguments [12]. It may be easier to implement this method of analysis, even though it requires an induction proof solver, than to automate the rules of combinatorial arguments and proofs. It should also be noted that at every step of the way we had a precise expression for the performance of the program. The marginal p.d.f. for any program variable gives the probability that the variable will take on a particular value.

Once the analysis of the bare algorithm is complete, an analysis for any particular aspect can be done by instrumenting the algorithm. It is easy to show that this algorithm requires exactly $\frac{(N^2-N)}{2}$ comparisons between the elements, which is twice as many as the "improved" version of the algorithm.

ALGORITHMIC COMPLEXITY
Part 5

by

Philip J. Janus

ADAPTIVE METHODS FOR UNKNOWN DISTRIBUTIONS
IN DISTRIBUTIVE PARTITIONING SORTING

ABSTRACT

Distributive Partitioning Sorting (DPS) is a new, innovative, practical method to sort a set of items on a computer. This method has been shown to be biased toward uniformly distributed data, performing poorly on skewed distributions. The purpose of this work is to find adaptive methods of DPS which will sort any unknown distribution equally well and remain competitive with DPS.

Two adaptive methods were developed and thoroughly tested, the Ranking Method and the Cumulative Distribution Function (CDF) Method. These methods transform unknown distributions into uniform distributions, and then perform the sorting. After an implementation of DPS was benchmarked against Quicksort, experiments were run on four distributions (Uniform, Normal, Poisson, and Exponential) using four algorithms (two

versions of DPS, Ranking, and CDF). Statistics were taken to measure the efficiencies and run times of the algorithms. The results were analyzed against theoretical and intuitive expectations so that conclusions could be reached regarding the performance of the methods.

It was found that if it is known in advance that the data distribution will typically be uniform, normal, or slightly skewed, then it is advisable to use DPS. However, if it is possible the data distribution might be very skewed, or extremely large or small data values exist relative to the rest of the data, then there is little to lose and much to gain by using the CDF adaptive method. CDFDPS contained only a 2% to 4% overhead to DPS in the uniform case, and ran up to 12% better for 30,000 items than DPS on exponentially distributed data. The ranking method was found to contain too much overhead to be competitive with DPS. Suggestions to further improve CDF are made, and future implications of this thesis work are discussed.

ACKNOWLEDGEMENTS

I would first of all like to express my deep appreciation to my thesis advisor, Dr. Edmund Lamagna, whose comments and criticisms influenced the development and implementation of this thesis work. Also, I would like to thank the other members of my thesis committee, Dr. Edward Carney, Dr. Leonard Bass, and Dr. Richard Bates, whose critiques and suggestions motivated and clarified many of the ideas in this work. I also extend my thanks to the staff of the University of Rhode Island Academic Computer Center for putting up with "that guy with PL/I problems". I am deeply grateful to Eleanor Gray for her hard work and patience typing and editing the manuscript. Research materials were referenced at the University of Rhode Island Library and the M.I.T. Engineering Library. The work was supported by Air Force Systems Command, Rome Air Development Center, under Contract No. F30602-79-C-0124.

Lastly, I extend my love and thanks to my Mother, Father, Da'kid, and Gail for their encouragement when things weren't going well, their support when things did, and their understanding when I took my troubles out on them instead of the computer! I thank you all!

<div align="right">

Pj

</div>

"Think of all difficulties as opportunities for creating something new."

<div align="center">

5-iii

</div>

PREFACE

A large percentage of data processing applications is spent sorting data. For that reason, it is not surprising that sorting is the most widely studied problem in computer science. The faster data can be sorted, the more computer time and money can be saved.

The history of the sorting problem is long and interesting. As expected in any field of study, once an algorithm has been developed, somebody tries to find a better one. This is true with a sorting method called Quicksort, developed in 1962. Quicksort had been shown to perform fastest on most machines once some modifications were made to it.

This was the case until 1978. In January of that year, a Polish computer scientist named Wlodzimierz Dobosiewicz published a paper in Information Processing Letters detailing a sorting algorithm called Distributive Partitioning Sorting, or DPS. This new sorting method was shown to perform much better than Quicksort. Very soon afterward, debate began as to its true practicality and significance.

And as could be expected, people started looking for ways to improve it. This thesis conducts an in depth look at DPS, and the various problems associated with it. The main focus of this work is to improve the overall performance of the algorithm. The author pleads guilty to first degree improvement.

5-iv

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF TABLES (continued)

CHAPTER I

DISCUSSION OF PRIOR WORK

I.1 Introducing DPS

In 1978, a new sorting method called Distributive Partitioning Sorting, or DPS, was presented to the world. The algorithm was published in the European periodical <u>Information Processing Letters</u> by a Polish computer science student named Wlodzimierz Dobosiewicz (pronounced Vod-jim'-yits Do-bo'-shev-its) [DOBO78a]. The article detailed a fast, practical, $O(n)$ sorting algorithm that could outperform current "fastest" methods. The results of the paper were so astounding that <u>Datamation</u> proclaimed it "the first real innovation in (sorting) in about 15 years!" [DATA78]. Experimental results on a CDC computer found DPS to be 30 times faster for 5000 items than its nearest competitor, Quicksort. It has also been shown that this factor increases as the number of items increases. The potential for saving computer time and money using DPS is great.

In Distributive Partitioning Sorting it can be shown that if the data is uniformly distributed, the expected complexity of the algorithm is $O(n)$; that is, the expected running time of the program is cn, where c is a constant that is multiplied by n, the number of items to be sorted. The drawback with DPS is that it is much slower in the worst case. Although DPS has an $O(n)$ expected case complexity, it is $O(n \log n)$ in the worst

case. DPS approaches the worst case as the data distribution
becomes more and more skewed, such as with a series of
factorials. The purpose of this work is to show that the
performance of DPS can be improved for unknown distributions.
The algorithm would then be guaranteed to outperform its
competitors for any input distribution.

I.2 Definition of Sorting

First it is necessary to define sorting, and the
limitations involved with it. Sorting, as the word implies, is
the arranging of a set of data into some prescribed order. In
his famous book, <u>Searching and Sorting</u> [KNUT73], Knuth
rigorously defines sorting in the following manner:

Suppose n items are given

$$R_1, R_2, \ldots, R_n$$

called records, to be sorted in either ascending or descending
order. The records collectively are called a file. Each
record, R_j, has a piece of information called a key field,
K_j, on which the record is to be sorted.

A linear ordering is defined on the keys with two
relational laws. Given three keys a, b, and c:

   i) Law of Trichotomy --

      One of either

$$a < b \quad a = b \quad a > b$$

      must be true.

ii) Law of Transitivity --

If a < b and b < c

then a < c.

Governed by this linear ordering, the goal of sorting is to rearrange the keys into a permutation

$$p(1), p(2), \ldots, p(n)$$

such that

$$K_{p(1)} \leq K_{p(2)} \leq \cdots \leq K_{p(n)}$$

The analysis of the various sorting algorithms in this paper will be concerned with a number of criteria on which a method's performance may be judged. An algorithm should be shown to work correctly for all types of expected inputs. The amount of work done and the amount of storage used should also be considered. Equally important is whether the resulting program is simple and lends itself to being easily understood, modified, and debugged. Lastly it should be seen if the method is optimal; that is, if another method exists which does less work or uses less space.

For example, consider the optimality of the sorting problem. We would like to know how many comparisons are necessary to sort a set of n items in a comparison-based sorting method. This means establishing a lower bound for such methods. For any comparison method, a comparison tree can be constructed.

Figure I.1 shows a comparison tree for three items. Each internal node represents a comparison and the lowest level

Figure I.1

Comparison Tree

Given Keys:

A_1, A_2, A_3
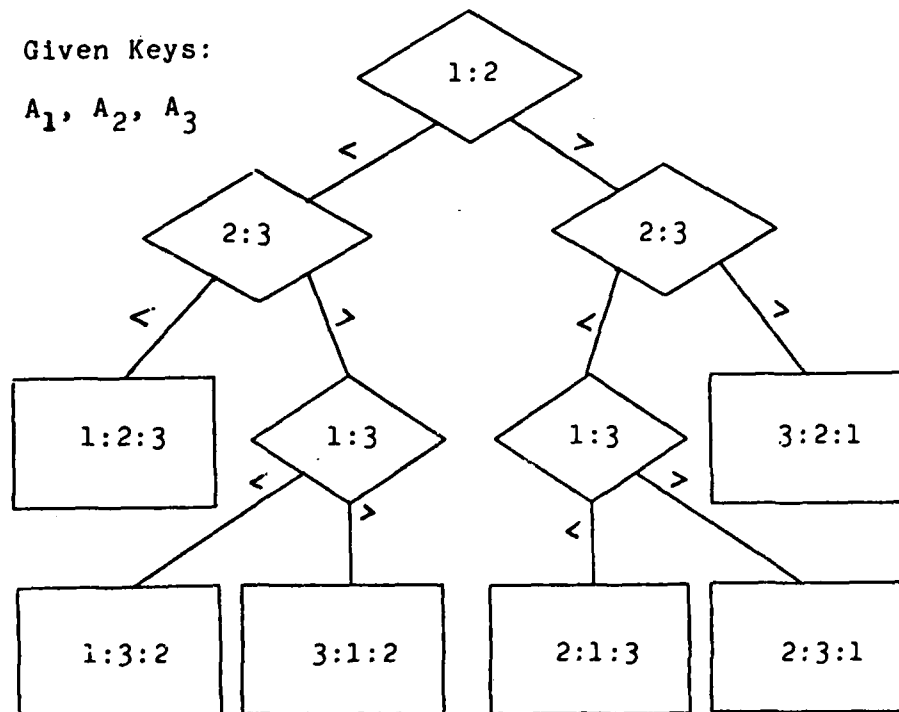
nodes (leaves) show the possible outcomes. Note that if we are given n items to sort, there are n factorial (n!) possible outcomes. Hence 3!, or 6, leaves are in the example tree.

Also note that in any binary tree of height k levels, there are at most 2^k leaves. In comparison trees, the leaves are the possible outcomes of the sort. So:

$$2^k \geq n! \qquad (1)$$

Let C(n) be the minimum number of comparisons necessary in the worst case. This corresponds to a path that is followed down to an outcome, which is just the height of the tree. So:

$$k = C(n) \qquad (2)$$

and then

$$2^{C(n)} \geq n! \qquad (3)$$

Taking the $\log_2$ of both sides

$$C(n) \geq \log_2 n! \qquad (4)$$

Approximating this using Stirling's formula gives

$$\log n! = n \log n - n/\ln 2 + 1/2 \log n + a \qquad (5)$$

This shows a lower bound of (n log n) on C(n).

This means that no comparison based method will work in less than (n log n) comparisons, and that any comparison based method attaining (n log n) comparisons is considered optimal.

Unless otherwise stated, all logs will imply $\log_2$.

Distributive Partitioning Sorting is a union of two classes of sorts; partition-exchange sorts and distributive bucket sorts. These classes of sorts will now be discussed, and then DPS will be presented.

I.3 Partition-Exchange Sorting

A number of sorting algorithms use an approach known as Exchange Sorting. This class of methods uses the idea that if two keys are found to be out of order, then the records are exchanged. The position in the file of the elements being exchanged can also be thought of as being swapped or interchanged. The exchanging continues until no more pairs are found to be out of order and the entire file is sorted.

One such exchange sort is called Quicksort. This algorithm was first presented by C.A.R. Hoare in 1962 in a very detailed paper, and has been the subject of very close study [HOAR62]. Briefly Quicksort works as follows:

Given an array A_1, A_2, ... , A_n to be sorted

Partition: Position some key, A_j, in its final

position, so that the file is divided into two parts. A_j is the partitioning element. All the items in the left subfile A_1, A_2, ... , A_{j-1} are less than A_j, and all the items in the right subfile A_{j+1}, ... , A_n are greater than A_j.

Recurse: Now the problem reduces to Quicksorting the two resulting subfiles until the subfiles have one element left.

Algorithm Quicksort is presented in Figure I.2.

It is fitting here to walk through an example of QUICKSORT. Suppose the elements to be sorted are those frequently used by Knuth.

503 87 512 61 908 170 897 275 653 426 154 509 612 677 765 703

    left = 1   right = 16

On the first pass through QUICKSORT

        part = 503

```
503 87 512 61 908 170 897 275 653 426 154 509 612 677 765 703
       i---------exchange-------------j
```

```
503 87 154 61 908 170 897 275 653 426 512 509 612 677 765 703
             i-----exchange------j
```

```
503 87 154 61 426 170 897 275 653 908 512 509 612 677 765 703
                      i---j
```

```
503 87 154 61 426 170 275 897 653 908 512 509 612 677 765 703
left------------------j   i
```

At this point the final position of the partition element, 503, has been found.

```
275 87 154 61 426 170 503 897 653 908 512 509 612 677 765 703
-----QUICKSORT-------      --------------QUICKSORT------------
```

Now QUICKSORT is performed recursively on the two resulting subfiles. One more pass on the smaller subfile will be illustrated.

part = 275

```
275 87 154 61 426 170
             i---j
```

```
275 87 154 61 170 426
left----------j   i
```

```
170 87 154 61 275 426
--QUICKSORT--     -Q-
```

Note that 275,426, and 503 are now sorted.

Figure I.2

Algorithm Quicksort

The swap operator :=: is used, where a :=: b means swap the values of a and b.

```
1)   procedure QUICKSORT (left,right)
        //------------------------------------------------
        //    Sort Array A bounded by A(left) to A(right)
        //------------------------------------------------
2)   integer left, right
3)   real array A
        //--------------
        //   Partition
        //--------------
4)   if right > left
5)     then do
6)        i = left
7)        j = right+1
8)        part = A(left)          //partition element
        //----------------------------------------------------
        //    Burn the candle at both ends until
        //    the position of the partition element is found
        //----------------------------------------------------
9)        do until j < i
10)          do i = i+1 while A(i) < part & i ≤ right
11)          do j = j-1 while A(j) > part & j ≥ left
12)          if j > i then A(i) :=: A(j)
13)        end
14)        A(left) :=: A(j)
        //----------------------------------
        //   Recurse on the two subfiles
        //----------------------------------
15)        QUICKSORT(left,j-1)
16)        QUICKSORT(i,right)
17)     end if
18)  end QUICKSORT
```

This divide and conquer approach to sorting achieves the desired result fairly rapidly. In fact, Quicksort is by far the fastest sorting method available for implementation on most computers [SEDG78]. This suggests that an analysis of the running time of Quicksort is appropriate to explain why this is so.

The worst case for Quicksort is when the file is already sorted. This is because the partitioning element being chosen each time results with that element in one subfile and the rest of the elements in the other subfile. But suppose a good choice is made of a partitioning element so that half go to one subfile and half go to the other. In each successive pass, the algorithm has two subfiles of $n/2$ elements to sort. Since the amount of time spent to find the position of the partition element is $O(n)$, the number of comparisons to Quicksort is then

$$C(n) \le n + 2C(n/2) \qquad (1)$$

Solving the recurrence relation

$$C(n) \le n + 2(n/2 + 2C(n/4)) \quad \text{at } n/2$$

$$\le 2n + 4C(n/4) \qquad (2)$$

$$\le 2n + 4(n/4 + 2C(n/8)) \text{ at } n/4$$

$$\le 3n + 8C(n/8) \qquad (3)$$

$$\vdots$$

$$C(n) \le kn + 2^k C(n/2k) \qquad (4)$$

The time to sort one element is zero

$$C(1) = 0 \qquad\qquad (5)$$

so we want to stop at $n/2^k = 1$ or $2^k = n$ or

$$k = \log n \qquad\qquad (6)$$

Substituting (5) and (6) back into (4)

$$C(n) \leq n \log n + 2^{\log n} C(1)$$

$$= O(n \log n)$$

So in a good case Quicksort is $O(n \log n)$. A rigorous analysis of the expected case shows that Quicksort is also $O(n \log n)$ [BAAS78]. As it turns out, Quicksort performs faster than any other sort on most machines. This is due to a low constant of proportionality in the $O(n \log n)$. But it is also due to some modifications that can be made to the original algorithm. With these changes Quicksort has evolved to its present day accepted implementation [SEDG78].

The first modification pertains to Quicksort's worst case problems. Recall that the worst case is when the file is already sorted. This paradox arises because on each pass the partitioning element divides the file into one subfile with one element and a second subfile with n-1 elements. This yields the number of comparisons

$$C(n) = \sum_{k=2}^{n} (k-1) = n(n-1)/2$$

which is $O(n^2)$.

The way to improve this worst case is by choosing a better partitioning element on each pass. As it has been seen, it would be nice if the partition divided the file in half, that is, if it is the median element. Finding the exact median is time consuming, although it can be done using 1.5n comparisons in the expected case [FLOY75]. A quick way to get an acceptable partitioning element is to choose the "median of three" elements:

A(left)   A((left + right)/2)   A(right)

where left and right are the upper and lower bounds of the array A. It has been found that this modification not only improves the worst case significantly, but also improves the average case by 5%.

Another problem with Quicksort is in sorting small subfiles. The algorithm spends much time partitioning, comparing, and exchanging elements. It seems worthwhile to use some other sorting method which is more efficient on these smaller files. The idea is to stop Quicksorting any subfile with less than some number of elements. The resulting file then contains a series of unsorted subfiles in their relative correct order. An Insertionsort (a simple sorting method) then completes the sorting by ordering these small individual groups as it scans through the file. It has been shown that there is at least a 15% savings when Insertionsort is used on subfiles with at most 9 items in them.

The last major problem with Quicksort is that it is recursive. Recursive procedures run very slow on most computers. Recursion can be removed by pushing and popping the left and right endpoints of the subfiles to be sorted on a Last-In-First-Out stack. If the smaller subfile is sorted first, it can be shown that the maximum stack depth never exceeds

log (n+1)/(M+2)   where M is the small file cutoff.

If M=9 and a maximum depth of 20 is assumed, then Quicksort can handle up to 10,000,000 elements! And the sorting is done with very little extra storage.

With these three modifications, Quicksort's run time can be improved by a factor of 20% over Hoare's original algorithm. This version has been implemented by R. Sedgewick and verified. The Sedgewick implementation of Quicksort will be used in this thesis work as a benchmark for DPS.

Quicksort still is not free of its problems however. The algorithm has an unstable property, that is, records with equal keys do not maintain their relative order. In most applications this is not a crucial factor, but it is an important consideration when implementing Quicksort. Although the median-of-three modification is made to improve the worst case, it is nonetheless unfortunate that the worst case is still $O(n^2)$.

Quicksort's disadvantages are far outweighed by its advantages. Aside from being practical and extremely fast, it uses relatively little extra space. With a day or two of effort, a working version of Quicksort can be implemented. All things considered, it is hard to beat. The reader is referred to [AHO74, BAAS78, GOOD77, HORO76, HORO78, KNUT73, FRAZ70, GRIF70, HOAR62, LOES74, SEDG78, SING69] for histories and discussions of Quicksort.

I.4 Bucket Sorting

Depending on the reference, this class of sorts is called Bucket, Distributive, Radix, or List Sorting [AHO74, BAAS78, GOOD77, HORO78, KNUT73], but the idea is all the same. Elements are distributed according to the value of their keys into buckets. Each bucket will be assigned those elements in a predefined range. Each bucket is then ordered by bucket sorting recursively or by some other sorting method. Then the items in the buckets are linked together to produce the final sorted sequence.

For example, here are Knuth's numbers again.
503 87 512 61 908 170 897 275 653 426 154 509 612 677 765 703
Suppose 4 buckets are created where the range 0-999 is evenly divided such that:

Bucket #	Values
1	0-249
2	250-499
3	500-749
4	750-999

Then on the first pass

Bucket Heads   Links

1 ------ 87 --- 61 --- 170 --- 154 --- nil

2 ------ 275 --- 426 --- nil

3 ------ 503 -- 512 -- 653 -- 509 --

             612 -- 677 -- 703 -- nil

4 ------ 908 --- 897 --- 765 --- nil

Recursively sorting bucket #1. 4 new buckets are created

Bucket #	Values
1.1	0-62.5
1.2	62.5-125
1.3	125-187.5
1.4	187.5-250

Upon distributing

Bucket Heads   Links

1.1 ------ 61 --- nil

1.2 ------ 87 --- nil

1.3 ------ 170 --- 154 --- nil

1.4 ------ nil

Notice that 61 and 87 are now sorted. Continuing to sort these
smaller buckets and appending them to one another will yield a
sorted set of items. Knuth calls this method Multiple List
Insertion (MLI).

Another variation of this method is to create ten buckets
and scan the key's digits from the least significant digit to
most significant digit (right to left) dropping them into the
appropriate buckets as we go along. This is highly known as
the LSD Radix Sort method. For example, here are Knuth's
numbers:

503 87 512 61 908 170 897 275 653 '26 154 509 612 677 765 703

For each successive pass after the first, buckets 0 through 9
are LSD sorted until all three significant decimal places have
been scanned, as in Figure I.3.

The time to perform this sort is just the number of
significant places, d, times the number of items. So this
algorithm takes dn passes and is $O(n)$.

Karp has shown that Knuth's Multiple List Insertion is
also $O(n)$ if the distribution function of the data is
well-behaved [KNUT73,p.105]. Without going into the details,
Knuth also shows that the best case for MLI is when the number
of buckets, M, is equal to the number of items, n, where the
items to be sorted are uniformly distributed. These concerns
will reappear for Distributive Partitioning Sorting.

Figure I.3
LSD Radix Sorting

<u>Bucket Head</u>   <u>Pass 1</u>   (ones place)

0 ---------- 170 --- nil
1 ---------- 61 --- nil
2 ---------- 512 --- 612 --- nil
3 ---------- 503 --- 653 --- 703 --- nil
4 ---------- 154 --- nil
5 ---------- 275 ---765 --- nil
6 ---------- 426 --- nil
7 ---------- 87 --- 897 --- 677 --- nil
8 ---------- 908 --- nil
9 ---------- 509 --- nil

                         <u>Pass 2</u>   (tens place)

0 ---------- 503 --- 703 --- 908 --- 509 --- nil
1 ---------- 512 --- 612 --- nil
2 ---------- 426 --- nil
3 ---------- nil
4 ---------- nil
5 ---------- 653 --- 154 --- nil
6 ---------- 61 --- 765 --- nil
7 ---------- 170 --- 275 --- 677 --- nil
8 ---------- 87 --- nil
9 ---------- 897 --- nil

                         <u>Pass 3</u>   (hundredths place)

0 ---------- 61 --- 87 --- nil
1 ---------- 154 --- 170 --- nil
2 ---------- 275 --- nil
3 ---------- nil
4 ---------- 426 --- nil
5 ---------- 503 --- 509 --- 512 --- nil
6 ---------- 612 --- 653 --- 677 --- nil
7 ---------- 703 --- 765 --- nil
8 ---------- 897 --- nil
9 ---------- 908 --- nil

It should be noted that although this method is both theoretically and practically faster than Quicksort, it uses much more storage. Recall that Quicksort only needed a fractional amount of extra space to maintain a stack to eliminate recursion. Bucket sorts need lots of extra space to maintain the buckets. The amount of extra space will be proportional to the number of buckets plus the number of items. Most implementations use extra space for bucket heads to show which item is first in each bucket, and for item links to show which items are in each bucket. The worst case for bucket sorts is when all but one item goes into one bucket on each pass. This leads to an amount of work:

$$\sum_{i=1}^{n} i = \frac{n(n+1)}{2}$$

which is $O(n^2)$.

In discussing bucket sorts, Baase suggests [BAAS78]:

"Thus in the worst case a bucket sort would be very inefficient. If the distribution of the keys is known in advance, the range of keys to go into each bucket can be adjusted so that all buckets receive an approximately equal number of keys."

Before introducing LSD sorting, she says:

"The reader might wonder why we don't use a bucket sort algorithm recursively to create smaller and smaller buckets. There are several reasons. The bookkeeping would quickly get out of hand; pointers indicating where various buckets begin and information needed to recombine the keys into one file would have to be stacked and unstacked often. Due to the amount of bookkeeping necessary for each recursive call, the algorithm should not count on ultimately having only one key per bucket, so another sorting algorithm will be used anyway to sort small buckets. Thus if a fairly large number of buckets is used in the first place, there is little to gain and a lot to lose by bucket sorting recursively."

These are precisely the issues this thesis will address with regard to Distributive Partitioning Sorting. (It is interesting to note these comments were published in the same year as DPS.)

I.5 Distributive Partitioning Sorting

The previous sections were intended to give the reader enough background to fully understand and appreciate the advantages and disadvantages of the sorting algorithm about to be presented.

On the one hand, DPS is an extension of Quicksort. Given n items, instead of two partitions being created on each pass, n partitions are created. But the similarity to Quicksort stops there, because DPS is not a comparison-based sort. Rather, it is a distributive based sort more resembling Knuth's Multiple List Insertion, where the number of buckets created equals the number of items being sorted.

Basically what is done is as follows:

Given n items to be sorted,

Select: Find the maximum, minimum, and median (middle) elements, all of which can be found in O(n) time.

Partition: Using these values, divide the range of the data between the maximum and minimum into n (buckets) with n/2 equal intervals on one side of the median, and another n/2 equal intervals on the other side.

Distribute: For each item, determine which of the intervals it falls into.

Recurse: For each interval with more than one element in it,

    sort the bucket using DPS.

The algorithm as originally published by Dobosiewicz in a pseudo-ALGOL language now follows in Figure I.4.

In short, here is what procedure SORT does. The items to be sorted are stored in array A. Line 11 creates a linked list S with the i-th item pointing to the $(i+1)$st in the list. Tne list end is designated by $S(m)=0$. The first half of the array is sorted by calling LSORT(1,0,m). The array is reordered by procedure MACLAREN which looks at the pointers and swaps the elements around to produce a sorted array A [KNUT73,p.596]. Lines 15 to 18 then complete the sorting on the second half of the array.

Procedure LSORT does the actual sorting. The array L of list heads is initialized. This array points to the elements at the top of the buckets being created. Each of these buckets can be thought of as being a Last-In-First-Out (LIFO) list. As an element is found to belong to a particular bucket, it is considered to be the new list head. Its pointer value is put into the L array and the S array is consequently updated.

Specifically, LSORT works as follows. The bucket of size n headed by pointer 'link' will be sorted. Step 1 is the initialization. Line 26 sets up the list heads for n buckets by initializing them to zero. Next the maximum, minimum, and median elements are determined. Lines 28 to 31 check for the case where all items are equal and considers them sorted.

5-19

Figure I.4

Algorithm SORT

```
1)   procedure SORT(A,n)
2)   integer n
3)   array A
4)   begin
         real min, max, median
5)       integer m
         //   the declaration of LSORT should be here   //

preparatorypass:

6)       FINDMINMAXMED(A,n)

         //   FINDMINMAXMED finds the smallest, largest, and
              median elements of an array A of length n.  These
              elements are stored in min, max, median respec-
              tively.  The array A is partitioned by the median
              selection algorithm in 2 halves:  A(1:⌈n/2⌉ ) and
              A( ⌈n/2⌉ +1:n).  The Rivest-Tarjan selection algorithm
              is suitable for use here   //

7)       m := (n+1)/2
8)       begin
9)           integer array S(1:m)
10)          integer i

initialize 1st:

11)          for i := 1 step 1 until m-1 do S(i) := i+1
12)          S(m) := 0

sorting:

13)          LSORT(1,0,m)
             //   LSORT does the sorting.  The array S will
                  contain a list of pointers showing the correct
                  order of elements   //
14)          MACLAREN(1,n)
             //   to complete sorting, it is necessary to reorder
                  the input vector A.  An algorithm due to M.D.
                  Maclaren is doing it in linear time.  Any other
                  method could be used   //

now sorting of the 2nd half:
initialize 2nd:

15)          for i :=1 step 1 until n-m-1 do S(i) := i+m+1
16)          S(n) :=0
17)          LSORT(1,m,n-m)
18)          MACLAREN(m+1,n)
19)      end
20)  end SORT
```

5-20

```
21)    procedure LSORT(link,incr,n)
22)    integer link,incr,n

//    LSORT performs a list sort on the elements of array A
      pointed at by a list stored in array S.  The value of
      link gives the head of the list and n is the number
      of elements of the list.  The parameter incr is a bit
      tricky:  it is used to distinguish between first and
      second halves of the array A.  Why is it used?
      Because, in order to save storage, there is a one to
      one correspondance between S and the current half of
      A:  if S(i) is in the list, it means that A(i+incr)
      is to be sorted in this pass  //

23)    begin
24)        integer array L(1:n)
25)        integer length,i,j,p

//    L is used to store pointers to nonempty lists (kept
      in S)  //

26)        for i := 1 step 1 until n do L(i) := 0

step 1:

27)        LFINDMINMAXMED(link,n)

//    selects the smallest, largest, and median of medians
      elements of list starting with S(link)  //

28)        if min = max then
29)        begin
30)            for i := link, S(i) while i > 0 do APPEND (i,incr)
31)        end  //  this was in case of identical items  //
32)        else

step 2:

33)            for i := link, p while i > 0 do
34)            begin
35)                p := S(i)
36)                j := if A(i+incr) < median then ...else ...

//    a complex expression finding to which group does the
      item A(i+incr) belong  //

37)                S(i) := L(j)
38)                L(j) := i

// item A(i+incr) is put on top of the LIFO list  //

39)            end
```

step 3:

```
40)        for j := 1 step 1 until n do
41)        if L(j) > 0 then
42)        begin
43)            length := -1
44)            for i := L(j), S(i) while i > 0 do
               length := length + 1
```

//   compute the length of j-th list.
     If more than 1 element, call LSORT again, otherwise
     append at the end of sorted part of the vector (a
     list kept in S). In actual program the array L
     should be compacted first: empty groups should be
     deleted. This ensures that a total number of
     pointers will never exceed the number of items //

```
45)            if length > 1 then LSORT(L(j),incr,length)
46)                else APPEND(L(j),incr)
47)        end
48)    end LSORT
```

Step 2 is the heart of the sorting. Line 33 is a loop which will scan down the elements of the bucket headed by the pointer 'link'. Line 35 saves the value of the pointer to the next element. The bucket value for that item is then calculated using "a complex expression" to be discussed in detail later. Line 37 shifts the LIFO list for that bucket, and line 38 puts that item at the head of the list.

Step 3 is the recursive step. All n buckets are scanned to see which ones need to be sorted still further. Line 44 determines the size of the i-th bucket. Lines 45 and 46 will call LSORT if there are still items to be sorted, otherwise the sorted item will be appended to the output array.

To avoid any further confusion, an example follows:

Given Knuth's numbers

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
503	87	512	61	908	170	897	275	653	426	154	509	612	677	765	703

and that Dobosiewicz's suggested partition formula is used:

$$\text{if } x \leq \text{median} \text{ then } j = \left\lfloor \frac{x - \min}{\text{median} - \min} \cdot \frac{n-2}{2} + 1 \right\rfloor$$

$$\text{if } x > \text{median} \text{ then } j = \left\lceil \frac{x - \text{median}}{\max - \text{median}} \cdot \frac{n-1}{2} + \frac{n+1}{2} \right\rceil$$

For reasons to be explained, this example will not sort one half first and then the other half, but rather will sort everything all together.

For the first call of LSORT:

min = 61

max = 908

med: $\lceil (n+1)/2 \rceil$ = 9th , so med=512

5-23

Array S is a set of pointers showing which element is next in any given bucket. L is an array of pointers showing which element is at the head of the i-th bucket. Figure I.5 demonstrates the sorting procedure.

Items in a bucket are found by the algorithm

```
for i := 1 to n    // For each bucket
    p := L(i)        // List Head
    do until p = 0
        print A(p)
        p :=S(p)     // Next Pointer
    end
end
```

The size of a bucket can be found in a similar manner. By recursively sorting each bucket of size greater than one, the sorting process will be complete.

It is easy to see that in the best case this algorithm is O(n). For perfectly equally spaced data, each bucket would contain one item after the first pass, and everything is sorted. Although it seems intuitive that this algorithm is O(n) in the average case, rigorously showing so is difficult. In his original paper, Dobosiewicz shows that the algorithm is O(n) in the expected case for uniform distributions. The reader is referred to this paper for details of the derivation [DOBO78a].

Figure I.5

Example of Distributive Partitioning Sorting

i	i	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
A(i)	A	503	087	512	061	908	170	897	275	653	426	154	509	612	677	765	703
j	S	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	0
S(i)=L(j)	L	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
L(j)=i																	

i=1	S	0	3	4	5	6	7	8	9	10	11	12	13	14	15	16	0
A(1)=503																	
j=7	L	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0
S(1)=0																	
L(7)=1	Bkt							503									

i=2	S	0	0	4	5	6	7	8	9	10	11	12	13	14	15	16	0
A(2)=87																	
j=1	L	2	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0
S(2)=0																	
L(1)=2	Bkt	87						503									

i=3	S	0	0	0	5	6	7	8	9	10	11	12	13	14	15	16	0
A(3)=512																	
j=8	L	2	0	0	0	0	0	1	3	0	0	0	0	0	0	0	0
S(3)=0																	
L(8)=3	Bkt	87						503	512								

i=4	S	0	0	0	2	6	7	8	9	10	11	12	13	14	15	16	0
A(4)=61																	
j=1	L	4	0	0	0	0	0	1	3	0	0	0	0	0	0	0	0
S(4)=2																	
L(1)=4	Bkt	61 87						503	512								

And so on until all the items have been scanned and sorted.

i=16	i	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
A(16)=703	A	503	087	512	061	908	170	897	275	653	426	154	509	612	677	765	703
j=13	S	0	0	0	2	0	0	5	0	0	0	6	1	0	9	0	0
S(16)=0	L	4	11	0	8	0	10	12	3	0	0	13	14	16	15	0	7
L(13)=16	Bkt	61 87	154 170		275		426 503	509	512			612	677 653	703	765		897 908
	Size	2	2	0	1	0	1	2	1	0	0	1	2	1	1	0	2

Now consider the worst case. This occurs when for each half of the buckets divided by the median, $n/2 - 1$ elements go into one bucket and one element goes into some other bucket. Therefore out of n buckets only four get used. The time, $T(n)$, in the worst case is given by the recurrence formula:

$$T(n) = cn + 2T(n/2), \qquad T(1) = c_0$$

Solving this in a similar fashion as with Quicksort's best case:

$$T(n) = cn \log n + O(n)$$

which is $O(n \log n)$.

It has been shown that DPS has a great advantage over classical comparison-based sorts because it is $O(n)$ in the expected case. DPS is also faster because there is no swapping of elements as in exchange sorts like Quicksort. Rather, linked lists are kept, where link values are replaced to reflect the changing bucket statuses. The data values themselves never move, but instead link values change. In this way, DPS is much like Knuth's Multiple Insertion which has many of the same qualities and characteristics.

I.6 Problems with DPS

There are many problems with DPS in both a theoretical and practical sense. They vary from fundamental problems with the algorithm, to large time and space overheads, to theoretically bad worst case running times. These will be discussed here with suggested solutions that form the basis for the implementations developed in this research.

5-26

There are several flaws in the algorithm as originally published. First of all, it is not necessary to find the maximum, minimum, and median in the preparatory pass (line 6).

Then there is a logic problem with the divide and conquer approach used in procedure SORT. The logic divides the array in half. The first call of LSORT sorts the first half of the array and the whole array is then reordered by procedure MACLAREN. For example:

    SORT(7 8 10 3 6 9 4 11)

    LSORT(7 8 10 3)

    MACLAREN yields (3 7 8 10 6 9 4 11)

Then LSORT is called for the second half of the array:
(6 9 4 11) . *This will yield two sorted vectors.*

    (3 7 8 10   4 6 9 11)

These now need to be merged to produce the correct ordering. To correct this problem, LSORT needs to be called only once to sort the entire array, and the code for 'incr' is eliminated. Otherwise the two sorted halves need to be merged.

Another problem is the partition formula, restated here:

if x $\leq$ median

$$\text{then } j := \left\lfloor \frac{x-min}{median-min} \cdot \frac{n-2}{2} + 1 \right\rfloor$$

$$\text{else } j := \left\lceil \frac{x-median}{max-median} \cdot \frac{n-1}{2} + \frac{n+1}{2} \right\rceil$$

In the else clause, the range of

$\frac{x-median}{max-median}$ is $(0,1]$

which will yield values in the _else_ expression of

$$(\quad \left\lceil \frac{n+1}{2} \right\rceil \quad , \; n \quad]$$

For example, with 16 buckets this formula will give values from 9 to 16, which is the desired result. It can be concluded that the _else_ clause is a valid expression.

Now consider the _then_ clause, where the range of

$$\frac{x-min}{median-min} \quad \text{is } [0,1]$$

This is inclusive because the median is included in the domain of the formula. The entire formula will yield values

$$[1 \; , \quad \left\lfloor n/2 \right\rfloor \quad]$$

For example, with 16 buckets, this will give values from 1 to 8. However, in only one case will these formulas ever yield a value of 8. That case occurs when the bucket value for the median is being calculated. A better choice for the $\left\lfloor (n+1)/2 \right\rfloor$ th bucket should be made.

This can be done by choosing the _then_ expression

$$\left\lfloor \frac{x-min}{median-min2} \cdot \frac{n-2}{2} + 1 \right\rfloor \quad \text{for } x \leq median$$

where min2 = min-0.0000001.   So   now   the   range   for $(x-min)/(median-min2)$ is $[0,1)$, and the range for the _then_ clause is $[1, \left\lfloor n/2 \right\rfloor)$. For 16 buckets, this will yield values from 1 to 7 which is closer but not quite there. Now 8 is missing. It will soon be seen how this problem is solved.

For now, consider the case when n=2 in the _then_ clause. For example, a bucket contains values

(61   87)

Here, FINDMAXMINMED will yield

    MAX = 87   MIN = 61   and   MED = 61

The median is chosen to be the $\lfloor (n+1)/2 \rfloor$ th (or 1st) element.
Using these values in the _then_ expression will give a zero or
some other small value in the denominator. This can easily be
corrected by choosing the median to be the $\lceil (n+1)/2 \rceil$ th
element. So now:

    MAX = 87   MIN = 61   and   MED = 87

But now upon evaluating the _then_ clause, both elements yield
bucket values of 1. This is because the $(n-2)/2$ term of the
expression will be zero since $n=2$. A suggested correction for
this is to evaluate the _then_ clause

$$\left\lfloor \frac{x-min}{median-min2} \cdot \frac{n}{2} + 1 \right\rfloor$$

which yields integer values in the range $[1, \lfloor (n/2)+1 \rfloor)$.
So for 16 buckets, the values 1 to 8 will be generated, where
now 8 is included as desired. For odd n, the extra bucket
value will be generated in the _then_ clause.

    There is yet a further problem with the _then_ clause.
Consider the case where the median is equal to or nearly equal
to the minimum. Previously, the concern was that a zero in the
denominator was likely. This might still be the case for an
input vector such as:

    (61 61 61 61 87 92)

Here: MAX = 92   MED = 61   and   MIN = 61

Again there is the same problem as before. One way to get
around this is to add the code:

if min = median then median := max

This way everything will be evaluated in the then clause. Note
that the case where the median is equal to the maximum is of no
concern. This is because the else clause would never get
evaluated in such a case. But the median
still causes some headaches, as it will soon be seen.

DPS proved to be very slow when implemented as previously
presented. In fact, the running time was 75% slower than
Quicksort on the average. It was obvious that to become
compatible with results published in Information Processing
Letters, the algorithm needed to be optimized as much as
possible. Later publications indicated that a certain amount
of code optimization was being done on the original DPS
algorithm.

As pointed out by Lamagna, Bass, and Anderson [LAMA80],
the consideration of constant factors in algorithms is
important, as is the case here. Sloppy code and inefficient
algorithms can be the source of large bottlenecks. DPS
requires the selection of the maximum, minimum, and median
elements. The published version of DPS used a 15n crude median
selection algorithm [KNUT73, p. 216] and it is possible to use
an inefficient 2n method to find the maximum and minimum.

The implementation of DPS in this thesis uses
Floyd-Rivest's 1.5n exact median selection algorithm [FLOY75],
and a 1.5n maximum-minimum selection algorithm; a significant

savings. This median selection method chooses the exact
median, as opposed to the crude estimate method used in the
prior implementation. Although there is still a high overhead
for these algorithms, it is by no means as great as for the
methods originally suggested by Dobosiewicz. The Floyd-Rivest
method uses n extra storage as opposed to $n/2$ extra for the
suggested method, but the time savings is well worth the extra
space. The reader is referred to [BLUM73, FUSS79, SCHO76] for
more details of median selection.

There are two bottlenecks that arise in DPS as they did in
Quicksort. One of these is the problem brought about by
recursion. As Lamagna, Bass, and Anderson [LAMA80] point out,
much time and space is used in most implementations of
recursion. In most cases the overhead can be eliminated by
creating a stack to store crucial values and efficiently coding
some type of outer loop to simulate the recursion. This is
true with DPS also. It has been found by this author that the
recursion can be removed without creating extra stacks or using
extra space. This is done by taking advantage of a suggestion
Dobosiewicz makes in Step 3. He states that "the array L
should be compacted", and this can be done quickly in $O(n)$.
The extra available space created by this compression allows
room for any needed bucket heads in subsequent levels of
recursion. Everything else being performed with pointers is
done in place, so no extra space is required.

Additionally, there is the matter of what to do with small
buckets, which is similar to the problem of Quicksort's small

5-31

subfiles. At some point it becomes advantageous to use an efficient sorting routine on these small buckets rather than recursively using DPS until the bucket size is 1. It was found that for DPS, it is more efficient to use an Insertionsort on bucket sizes less than or equal to 9 or 10.

Further improvements can be made in the algorithm's run time by optimizing the code. It is possible to combine certain loops in what can be considered to be the heart of the algorithm in Steps 2 and 3. Step 2 performs many common arithmetic operations repeatedly. These may be removed from the loop to save time. It is also possible to optimize Step 3 and the code resulting from removing recursion such that a very tight efficient loop can be implemented.

Unfortunately, there are certain characteristics of the algorithm that cannot be dealt with. It turns out that due to the nature of the Last-In-First-Out (LIFO) lists used as pointers for the buckets, DPS is unstable. Records with equal keys may not necessarily be output in the same relative order in which they were input. In fact, if an odd number of passes is made over the equal keys, they will be sorted in reverse relative order. If an even number of passes is made, the items will be stably sorted. Quicksort also has a stability problem, although the reasons for this are entirely different. It is an interesting phenomenon.

As Baase pointed out, there is a large storage overhead associated with this type of algorithm [BAAS78]. For the pointers alone, the overhead is 2n, and for the median

5-32

selection it is an additional n. However, with today's virtual memory environments, the impact of this consideration is minimized. Only for extremely large n would problems in storage occur.

The reader is referred to [AKER78, BURT78, DATA78, DOBO78b, DOBO79, HUTT79, JACK79] for additional arguments concerning the practical significance of Distributive Partitioning Sorting which are of little concern to this work. The DPS algorithm used in this work is presented in Figure I.6. Summary of suggested improvements to DPS:

. Delete preparatory pass

. Remove divide-and-conquer approach

. Adjust the partitioning expression

. Handle the minimum = median case

. Remove recursion

. Optimize loops

. Multiply by 0.5 instead of dividing by 2.0

. Eliminate mixed mode arithmetic

. Take common expressions out of loops

. Choose the median = (min+max)/2

. Use Insertionsort on small buckets

Figure I.6

Algorithm DPS

```
procedure DPS(n,A)
integer array L(n),S(n)
integer n,length,i,j,p,link
array A

for i:= 1 step 1 until n do L(i):=0
L(n):=1
for i:= 1 step 1 until n-1 do S(i):=i+1
S(n):=0
top:= length:=n

do while(top < n)              //    Recurse  //
   FINDMAXMIN(L(top))

   //    Find max and min of list pointed at by L(top)   //
   //    Note:  Adaptive methods are placed here  //

   link:=L(top)
   L(top):=0
   if min = max then APPEND(link)
   else
      for i:= link,p while i>0 do
         begin
            p:=S(i)
            j:= partitioning formula that distributes A(i)
            S(i):=L(j)
            L(j):=i
         end
   COMPRESS(L,top,n,length)

   //  List heads are pushed to the back of array L such
       that the front of the array is all zeroed.  top:=
       the first non-zero pointer  //

   length:=0
   for i:=L(top),S(i) while i>0 do length:=length+1
   do while (length < insertionsort cutoff)
      APPEND(L(top))
      L(top):=0
      top:=top+1
      if top < n then  // Find length of next bucket //
         begin
            length:=0
            for i:=L(top),S(i) while i>0 do
               length:=length+1
         end
      else  exit DPS
   end
end DPS
```

5-34

Having addressed most of the issues raised by Baase
earlier, the adaptive methods of DPS can now be
discussed. As it has been shown, the worst case for DPS
is $O(n \log n)$, which is no better than Quicksort on the
average. The question is: Can anything be gained by
knowing something about the distribution of the data in
advance? And, if so, is it worth it?

CHAPTER II

ADAPTIVE METHODS FOR UNKNOWN DISTRIBUTIONS

Recently, work was done by Meijer and Akl [MEIJ80] to try to "Hybrid" Distributive Partitioning Sorting according to a known distribution. Though this work is promising, it is by no means general enough to handle empirical or general distributions. The authors suggest,

> "...when the distribution of the input sequence is not known, another topic for future research would be to study the problems associated with estimating this distribution."

When that paper was published, the proposal for this thesis was independently being formulated and exactly those ideas were suggested as a course of thesis work. (In fact, this author did not receive the above publication until seven months after it was issued, and the thesis work was well into the experimental stage.)

The purpose of exploring adaptive methods for DPS is to improve its worst case performance. It is readily seen that as the data distribution becomes more and more skewed, the worst case is approached. Dobosiewicz shows that the worst case is a set of factorials. It is desired, then, that these methods be 'adaptive' in the sense that they adjust to whatever data distribution is given.

Recall that DPS divides the range of the data into n partitions based on the maximum, minimum, and median (or mean) values. Half of these partitions are all of one fixed length,

and the other half of another length. For skewed distributions, the resulting bucket sizes could vary greatly. It is the goal of the adaptive methods to examine the data distribution and somehow transform these potentially large bucket sizes into buckets with as close to one item per bucket as possible (DPS's best case). Figure II.1 graphically shows these ideas and concerns.

The search for various adaptive methods has crossed many different fields of mathematics, including: linear algebra, numerical analysis, statistics, probability, combinatorics, and plain old "horse sense" math. These approaches will be discussed in this section along with their advantages and disadvantages.

A number of questions arise as these adaptive methods are being looked at:

. What information about the distribution will be
    useful?

. How can the information be used to obtain the goal?

. Is the information and goal obtained easily and
    quickly at relatively little cost?

In discussing DPS, the term bucket size refers to the number of items per bucket. Partition length refers to the length of a partition within the range of the data.

Figure II.1

Concerns of Adaptive Methods
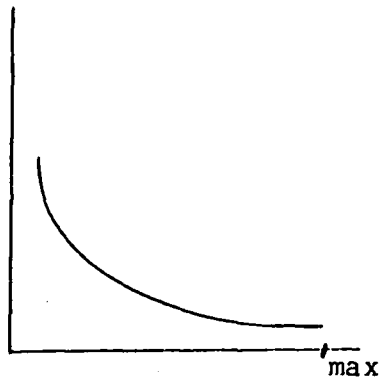
Find Data Distribution   DPS Partitions

.DPS Bucket Sizes   Desired Bucket Sizes

II.1 Frequency Distribution Curves

If a small sample of the data is taken and distributed into buckets, then the bucket sizes can be thought of as being a set of frequency occurrences. Often this set of fr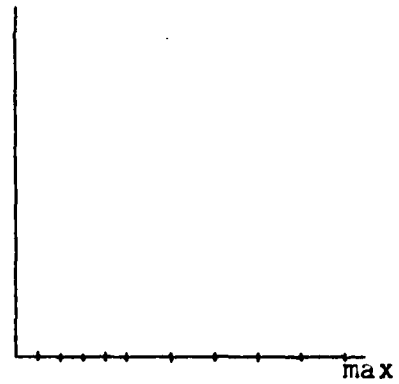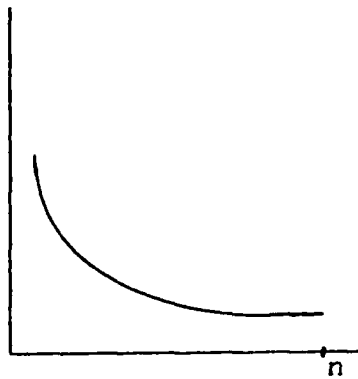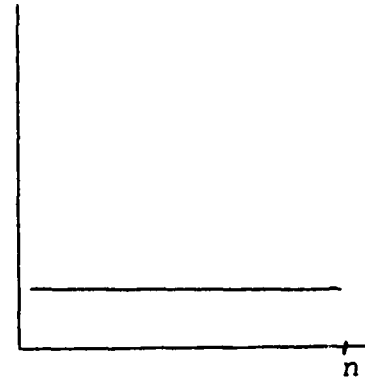equencies fits a theoretical distribution such as Uniform, Normal, Poisson, or Exponential, as shown in Figure II.2. Many times a curve can be fit to these frequencies. Usually, the probabilities of the frequencies are found and a curve is fit to them. This is known as a Probability Density Curve. By finding such a curve, it might be possible to find a transformation to appropriately adjust the partition lengths so the bucket sizes are more uniform. Some methods of finding Probability Density Curves will now be discussed.

II.1.1 Method of Moments

Various statistics concerning distributions can be gathered such as the mean, skewness, kurtosis, and others. These are called moments. The moments about the origin for the elements x_1, x_2, ..., x_n can be calculated by

$$m_r = \frac{1}{n} \sum_{i=1}^{n} x_i^r$$

According to Elderton and Johnson [ELDE69], if n is the number of points and m_r is the r-th moment, then a frequency distribution curve can be fit to:

Figure II.2

Frequency Distributions
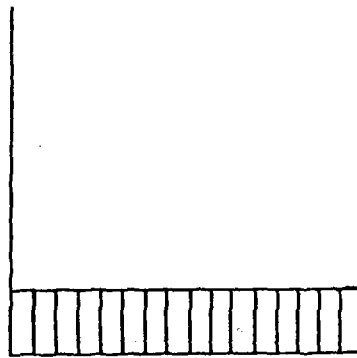
Uniform   Normal
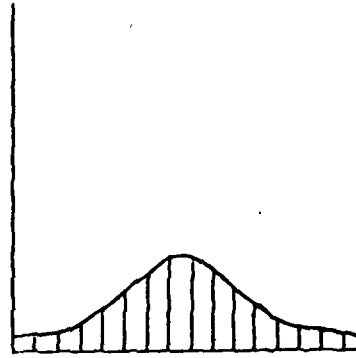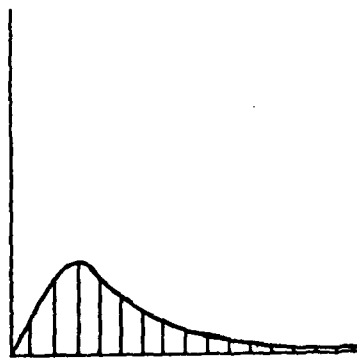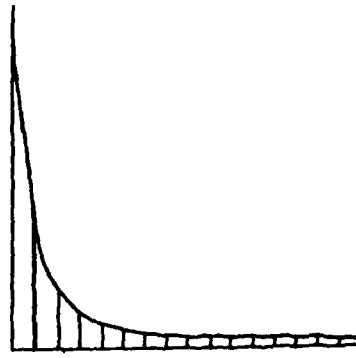
Poisson   Exponential

1) $y = a + bx$, where

$$a = \frac{1}{2n} \, m_0$$

$$b = \frac{3}{n} \cdot \frac{1}{2n} \cdot \frac{m_1}{n}$$

2) $y = a + bx + cx^2$, where

$$a = \frac{3}{4} \left(\frac{3}{2n} \, m_0 - \frac{5}{2n} \cdot \frac{m_2}{n^2} \right)$$

$$b = \frac{3}{n} \cdot \frac{1}{2n} \cdot \frac{m_1}{n}$$

$$c = \frac{15}{4n^2} \left(-\frac{1}{2n} \cdot m_0 + \frac{3}{2n} \cdot \frac{m_2}{n^2} \right)$$

3) $y = a + bx + cx^2 + dx^3$

$$a = \frac{3}{4} \left(\frac{3}{2n} \, m_0 - \frac{5}{2n} \cdot \frac{m_2}{n^2} \right)$$

$$b = \frac{15}{4n} \left(\frac{5}{2n} \cdot \frac{m_1}{n} - \frac{7}{2n} \cdot \frac{m_3}{n^3} \right)$$

$$c = \frac{15}{4n^2} \left(-\frac{1}{2n} \cdot m_0 + \frac{3}{2n} \cdot \frac{m_2}{n^2} \right)$$

$$d = \frac{35}{4n^3} \left(-\frac{3}{2n} \cdot \frac{m_1}{n} + \frac{5}{2n} \cdot \frac{m_3}{n^3} \right)$$

A system of equations fitting various commonly occurring distributions were developed by Karl Pearson. These equations are for curves of the form

$$b_0 + b_1 x + b_2 x^2 = 0$$

Type I) If the roots are real and of different sign

$$Y = y_0 \left(1 + \frac{x}{a_1} \right)^{m_1} \cdot \left(1 - \frac{x}{a_2} \right)^{m_2}$$

where a_1 = $root_1$ - (distance from origin to mode)

a_2 = $root_2$ - (distance from origin to mode)

and $m_1/a_1 = m_2/a_2$

Type VI) If the roots are of the same sign.

$$Y = y_0 \left(x - a \right)^{m_1} \cdot x^{-m_2}$$

And so on for other types.

There is also a set of normal curves known as Gram-Chalier curves which use moments in fitting a curve to a distribution [GRAM45]. Given a normal frequency function $f(x)$, and $g(x)$ is the standard normal function where mean=0, and variance=1, such that

$$g(x) = \frac{1}{\sqrt{2\pi}} e^{x^2/2}$$

then

$$f(x) = g(x) + \frac{c_3}{3!} g^{(3)}(x) + \frac{c_4}{4!} g^{(4)}(x) + \dots$$

where c_3 = $-m_3$ = - skewness coefficient

c_4 = m_4 - 3 = excess coefficient

c_5 = $-m_5$ + $10m_3$

c_6 = m_6 - $15m_4$ + 30

The problem with these moment methods is that an excessive amount of time is used in determining the moments and coefficients of the equations. The number of arithmetic

5-42.

operations being performed would rapidly become very large. Although moment methods would provide a good guess to the distribution, they lack the efficiency that is desired for a modification to DPS.

II.1.2 Curve Fitting

Another way to 'discover' the distribution is to try to fit a curve [DANI80] to the probability density function based on the sampled probabilities. The first thought that might come to mind is to try a high degree least squares fit [STRA76], such that a third or fourth degree polynomial fit. Although a large number of calculations are needed, it would not be as great as with the mom᷊᷋ calculations, especially if the number of sampling cells is kept relatively small.

The least squares fit would work nicely if the distribution were smooth. In practice, though, many distributions do not fit smooth, monotonic, or well behaved curves (i.e., dictionary data, last names, social security numbers, etc.). Leasts squares methods might yield a badly fitting curve [GERA78].

Suppose a straight line fit is used between cell probabilities as shown in Figure II.3. For each line, the slope and y-intercept can be saved and used later to determine what bucket to adjust an item to. But can this practically be done? Given an item and this sample probability density curve, the item's relative position in the range needs to be found.

Figure II.3

Line Fit to Frequency Probabilities
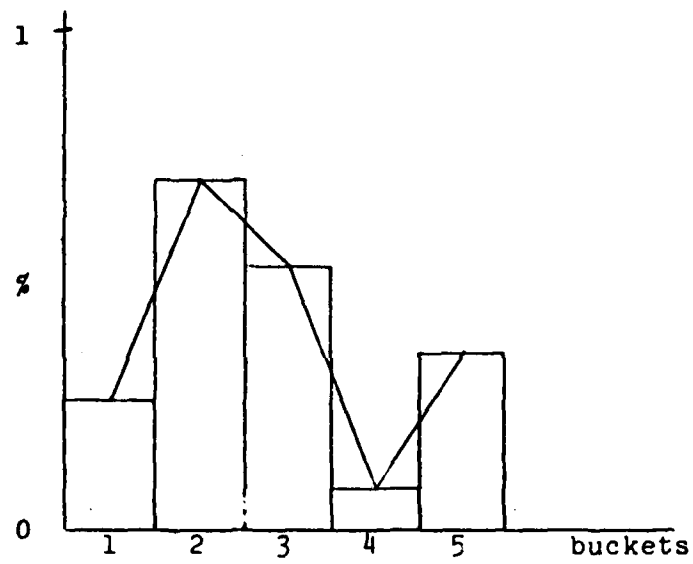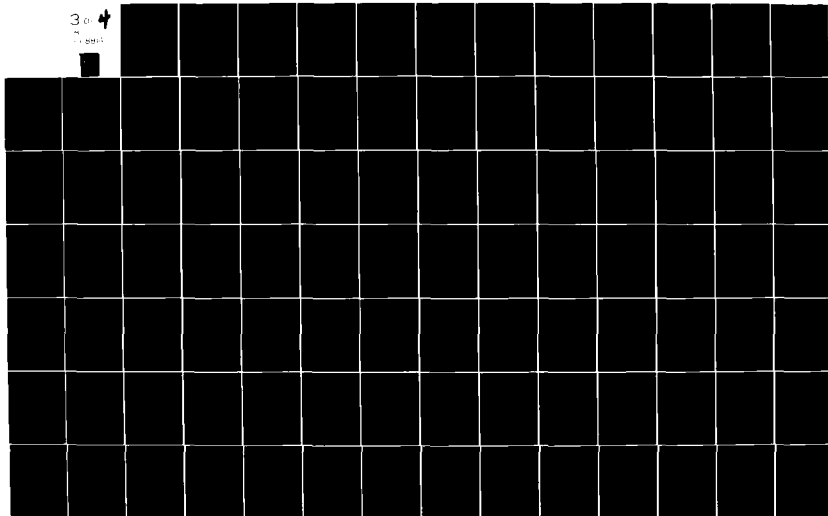
The only sensible way to accomplish this is to determine the probability that an item will fall into the i-th bucket with respect to the rest of the data. This suggests we need to find the Cumulative Distribution Function (CDF) as opposed to the Probability Density Curve (PDC).

The CDF can be found by integrating the PDC. The preprocessing necessary to find the PDC by these prior techniques would be quite time consuming. A method will now be suggested to find the CDF quickly and efficiently.

II.2 Cumulative Distribution Function (CDF) Method

A more useful tool for adaptive methods is the Cumulative Distribution Function. This was used independently in work done by Meijer and Akl [MEIJ80] for known distributions. If f(t) is the Probability Density Curve, then the Cumulative Distribution Function is

$$F(x) = \int_{-\infty}^{X} f(t) \, dt$$

If the probability, p_i, for an item falling into the i-th sampling cell is given, then the cumulative probability distribution for the i-th cell is

$$p_i = \sum_{k=0}^{i} p_k \, , \quad p_0 = 0 \, , \quad p_n = 1 \quad 0 \le p_i \le 1$$

A useful property of this curve is that it is continuous monotonic nondecreasing. Fitting a line between each successive pair of sampling cells should give a good estimate of the Cumulative Distribution Function.

It will now be shown that if the cumulative distribution function is known or can be approximated, then the resulting transformation of items by this function is uniform.

Given:

X is the underlying random variable of the data.

$$G_X(x) = P(X \leq x) \tag{1}$$

is the Cumulative Distribution Function. It is continuous monotonic increasing so the inverse, $G_X^{-1}(x)$, also exists.

Y is a random variable where $Y = G_X(X)$ is the transform data. To find the distribution of y, we observe that the distribution of Y is uniform on [0,1] because

$$
\begin{aligned}
F_Y(y) &= P(Y \leq y) && \text{by def of CDF} \\
&= P(G_X(X) \leq y) && \text{substitution} \\
&= P(X \leq G_X^{-1}(y)) && \text{inverse of both sides} \\
&= G_X(G_X^{-1}(y)) && \text{by def (1)} \\
&= y
\end{aligned}
$$

Therefore the transformation is uniform.

Figure II.4 shows how the CDF takes any distribution along the x-axis and transforms it onto a uniform distribution on the y-axis. This implies that if a sample CDF can be found, then the resulting items can be spread uniformly among the buckets. This is essentially what is done "in reverse" when a uniform random distribution is transformed into another distribution in simulation systems [GRAY80].

5-46

Figure II.4

Cumulative Distribution Function

Uniform Transformation
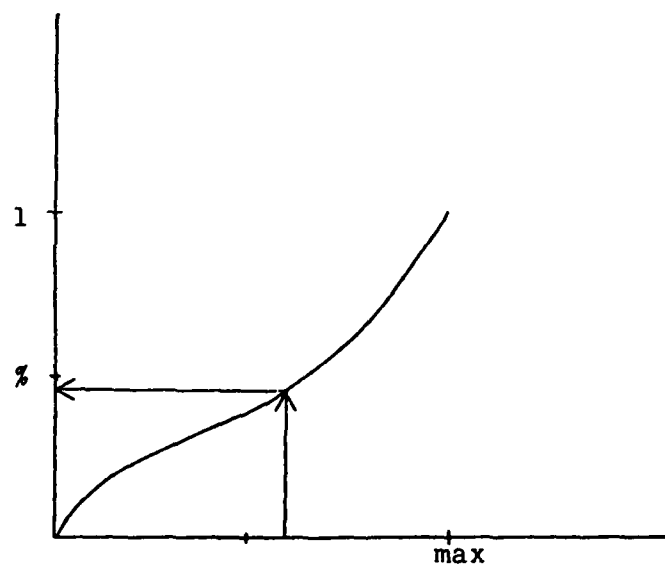
Figure II.5 shows a probability density curve and its corresponding cumulative distribution function.

A very simple, practical algorithm can be written using the idea of cumulative distribution functions to create an adaptive DPS method.

Step 1) Sample the data and distribute it into cells by

$$k = \left\lfloor \frac{x-min}{mx2-min} \right\rfloor \cdot M + 1$$

where $mx2 = max + .0000001$ and M is some arbitrary number of sampling cells. This formula yields integers in the range $[1, M]$.

Step 2) Find the cumulative probabilities of the M cells.

Step 3) Fit a line between each pair of cumulative probabilities using $P_0 = 0.0$ and $P_M = 0.9999999$. Save the slope and y-intercept of each line. This yields a sample Cumulative Distribution Function.

Step 4) Distribute all of the items by first determining which sample cell it belongs to, say k, and then use the k-th line equation to find what bucket the item really falls into. Note that to insure the CDF is monotonic increasing, as opposed to nondecreasing, each sampling cell is initialized to have one item. This is to guarantee the inverse CDF function exists.

Figure II.6 shows these steps pictorially, and Figure II.7 describes the algorithm in detail.

Figure II.5

Sampled Probability Density Curve

and Its Cumulative Distribution Function

f(x)

F(x)
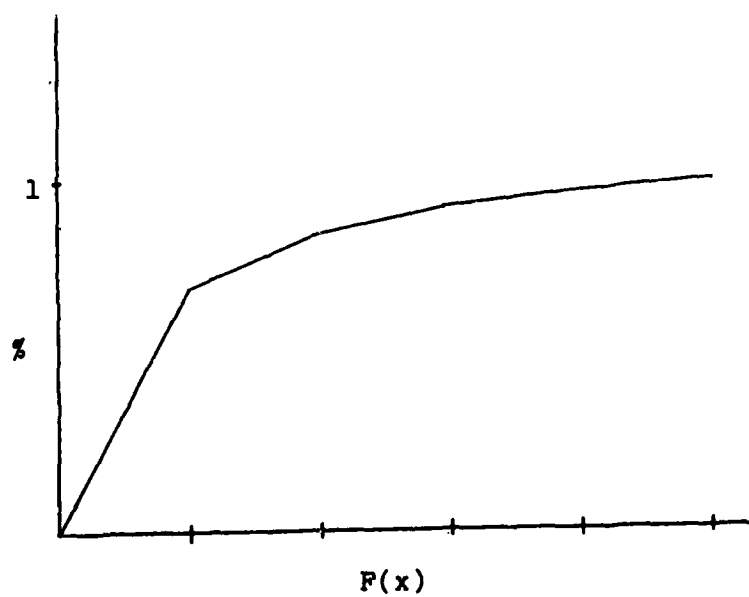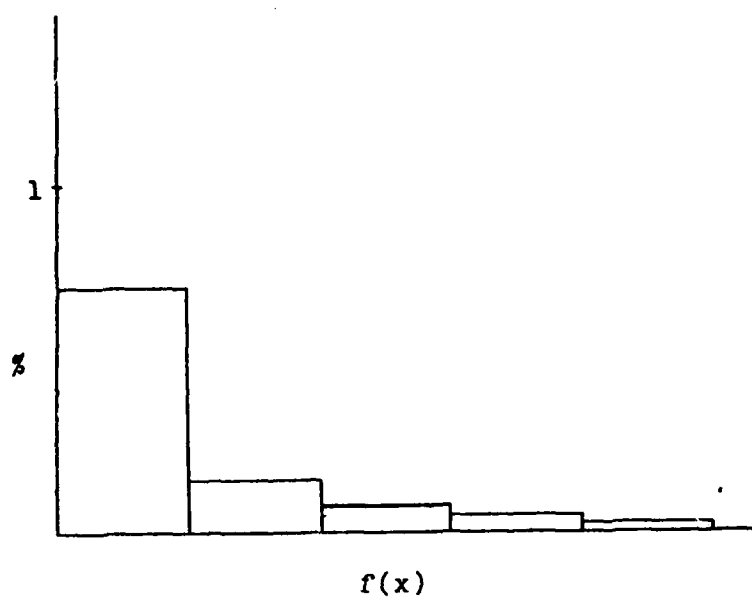
Figure II.6

The Steps of Algorithm CDF

Data Distribution

Sample Items into Cells

Step 1

Find Cumulative Probabilities

Step 2

Determine Sample CDF

Step 3

Distribute Items

Step 4

Resulting Distribution
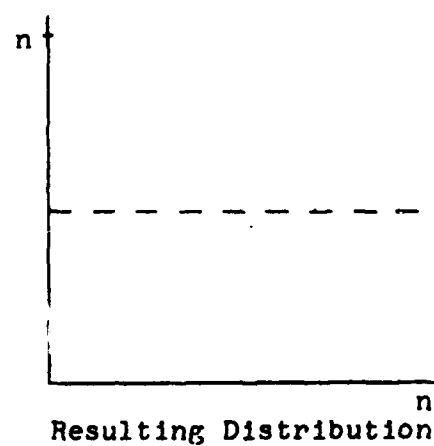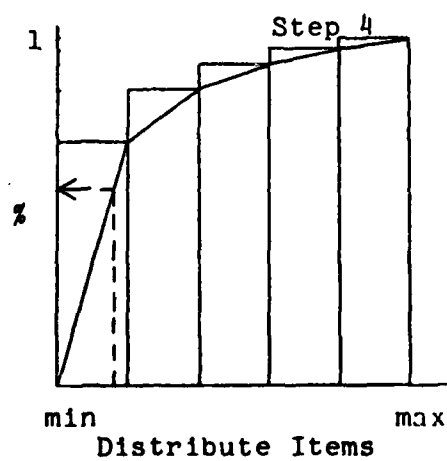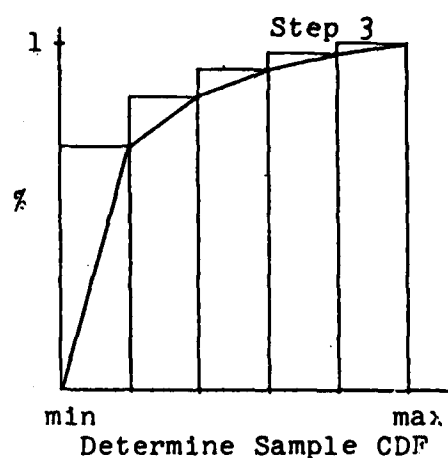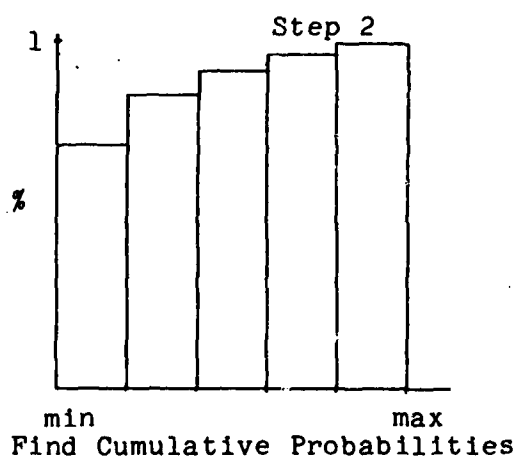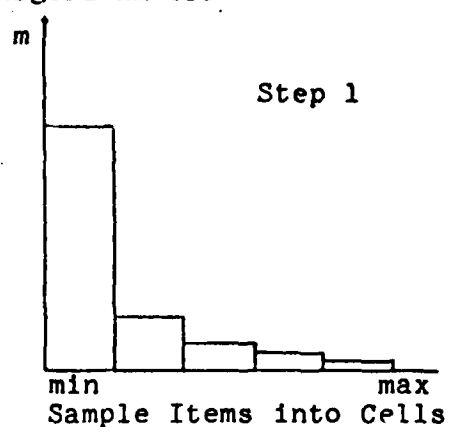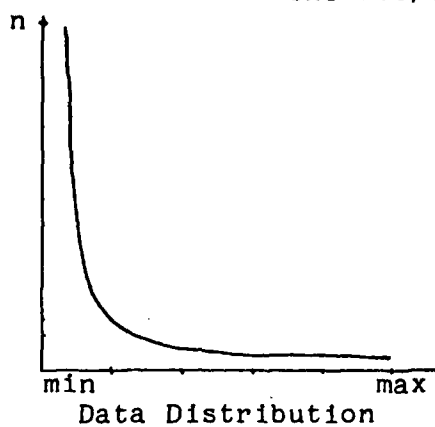
Figure II.7

Algorithm CDF

The following algorithm is placed after FINDMAXMIN in DPS:

```
integer sample
if length > sample then
    begin                           //  CDF  //
        integer k
        array CELL(0:m),M(m),B(m)

        //CELL -- sampling cells
            m   -- number of cells
            M   -- slope of lines
            B   -- y-intercept of lines  //

        FREQUENCY(L(top))

        //Take a frequency count of items assuming a
            uniform distribution into m cells ranging
            from min to max.  Each frequency is initially 1.//

        CUMPROB(CELL,m)

        //Finds the cumulative probabilities of frequency
            cells  //

        CELL(0):=0.0
        CELL(m):=0.9999999
        LINEFIT(CELL,m)

        //Fit m lines to m+1 points.  Put slopes into M
            and y-intercepts into B                      //

        for i:=L(top),p while i>0 do
            begin
                p:=S(i)
```
$$k := \left\lfloor \frac{x-min}{max-min2} \cdot m \right\rfloor + 1 \qquad // \text{ find cell } //$$

$$j := \left\lfloor \left(M(k) \cdot \frac{x-min}{max-min2} \cdot m + B(k)\right) \cdot length \right\rfloor + 1$$

```
                          //  Transforms data into uniform distri-
                                  bution  //

                S(i):=L(j)
                L(j):=i
            end
    end
else  DPS(L(top))  //  as usual  //
COMPRESS(L,top,n,length)
        .
        .
        .
```

5-51

As an example, suppose there is a skewed distribution like that in Figure II.6. If there are 100 items being distributed into 100 buckets, then note what happens to the items that would normally have gone in cell #1. The cumulative probability for this cell ranges from 0.0 to 0.7. Therefore 70% of the data falls into cell #1. Plugging the values of the items belonging to cell number 1 into the first line equation and multiplying by 100 will now yield bucket values from 1 to 70, instead of 1 to 20 as would normally have occurred in DPS. This is the result that is desired from adaptive methods.

II.3 Ranking Method

Now the question arises: Is it really necessary to find out anything at all about the distribution? In fact, there is a simple method by which to adapt the partition lengths to a particular data distribution without gathering information about the distribution itself. This can be achieved by the following algorithm:

Step 1) Sort a sample of the data by some fast method.

Step 2) Divide the number of items sampled by the number of cells (m/M), and then choose Partition Endpoints by selecting every (m/M)th item. Note that M should be of the form $2^k - 1$ to facilitate the binary search in step 3. $E_0 = \text{min} - 0.0000001$, and $E_M = \text{max}$.

Step 3) For each item in the file, perform a binary search to find what cell it belongs in.

Step 4) Find the bucket it falls in by the expression:

$$j = \left\lceil \left(\frac{x - E_{k-1}}{E_k - E_{k-1}} + k-1 \right) \cdot \frac{n}{M} \right\rceil$$

where k is the cell found

E_k -- k-th right side endpoint

E_{k-1} -- k-th left side endpoint

n -- number of data points

M -- number of cells

If the number of cells and the sample size are kept small and fixed, then the overhead associated with the binary searching and sorting will be a fixed constant factor. Figure II.8 outlines the method.

It is hoped that an experimental analysis of these methods will show that DPS can be improved to handle unknown distributions.

Figure II.8

Algorithm RANKING

```
    ⋮
FINDMAXMIN(L(top))
integer sample
if length > sample then
    begin
        integer k
        array E(m)
        QUICKSORT(sample,RA)

                // Order the sample array RA //

        for i:= 1 step 1 until m do E(i):= RA(i . sample/m)

                // Get every (sample/m)th item //

        E(0):=min2
        E(m):=max
        link:=L(top)
        L(top):=0
        for i:=link,p while i>0 do
            begin
                p:=S(i)
                k:=BINSEARCH(x,E)

                // Find cell x belongs to by binary search //
```

$$j:= \left\lceil \left(\frac{x-E(k-1)}{E(k)-E(k-1)} + k-1 \right) . \frac{length}{m} \right\rceil$$

```
                S(i):=L(j)
                L(j):=i
            end
    end
    else  DPS(L(top))    // as usual //
COMPRESS(L,top,n,length)
    ⋮
```

CHAPTER III

EXPERIMENTAL DESIGN AND ISSUES

This chapter is intended to describe the issues and problems in designing appropriate experiments for DPS and the adaptive methods. It could also serve as a guide for other researchers conducting work in a virtual machine environment where algorithm timings are needed. The last part of this chapter describes the reasons for choosing various parameters used in the experiments.

III.1 Experimental Problems and Issues

There are a number of considerations in designing experiments to test algorithms. Lamagna, Bass, and Anderson [LAMA80] discuss many of these in developing a research plan to study the performance of algorithms. They note that the programming language chosen has a large effect on how well a program will perform. Various compilers will generate widely different machine code as would be the case for COBOL, FORTRAN, and PL/I compilers. The University of Rhode Island Academic Computer Center houses a National Advanced System/5 Model 7031 which is an IBM 3031 equivalent. Due to the advanced features of the PL/I Optimizing Compiler, PL/I was chosen to code the aforementioned algorithms. As will be seen, PL/I also contains some useful compiler options.

The machine chosen to execute on plays a large role in how fast a given program will run. For example, the floating point

operations on a CDC machine are many times faster than on a comparable IBM due to the hardware configuration of the machines. Thus it should be noted that while one algorithm may outperform another by a large factor on one machine, this may not necessarily be true on another. This has been seen in previous experiments conducted with DPS [DOBO79].

There now comes a problem common to many fields of endeavor. And that is, the extent to which one considers the work of the theorist when putting a concept into practice. In computer science, this problem is exemplified by the conflict between theoretical order of magnitudes, and practical considerations for loop control, testing, bookkeeping, and memory accesses. These latter factors can contribute a high constant of proportionality to the theoretic order of magnitude. A $(2n \log n + 3n)$ algorithm will, for example, generally perform better than a $(3n \log n + 5n)$ algorithm, even though they are both theoretically $O(n \log n)$.

This consideration lends itself to the issue of crossover points, that is, the point where one algorithm begins outperforming another. For example, a $2n^3$ algorithm is better than a $50n^2$ algorithm for $n < 25$, although for $n > 25$ the reverse it true. It will be seen where the crossover is for DPS and Quicksort.

Lamagna, Bass, and Anderson [LAMA80] also point out that in addition to these issues, an algorithm can be greatly improved by various modifications, although the order of magnitude stays the same. By utilizing clever data structures,

loop control, and insights, an algorithm can greatly improve its performance as was the case with Quicksort. Another example of this is the remarkable insight of Dobosiewicz in transforming $O(n^2)$ Bubblesort into an algorithm which outperforms Quicksort on input sizes less than 2000 [DOB080]! But while one constant factor may decrease due to a change, another could increase. So the question is: At what point is cleverness and extra overhead not worth it? There comes a point where simplicity may outweigh efficiency.

III.2 Experimental Design

As mentioned, work for this thesis was done on a National Semiconductor plug compatible IBM computer in a virtual memory environment. Due to paging, cycle stealing, swapping, and load on the computer, the run time of two identical experiments could vary by up to 25%. Experiments were designed to eliminate this undesirable "noise" from the run times. Lamagna, Bass, and Anderson suggested determining weights for straight line code in an algorithm and then counting how many times each section was executed.

These estimates could prove to be inaccurate in practice if they are not chosen carefully. A method of obtaining fairly accurate run times was used in this work, and will now be described. The PL/I compiler used contains a COUNT option which produces a printout of how many times each statement is executed. This can easily be simulated in languages not containing this feature. There is also a LIST option which

5-57

generates listings of assembly code similar to the machine code produced by the compiler. Using instruction timings available in the IBM System/370 Model 158 Functional Characteristics [IBM78], it is possible to calculate the timing for each instruction. Although this is tedious, it does yield accurate run times. And with some amount of work, this process can be completely automated. By multiplying the count of each instruction by its timing, and then summing over the entire instruction set, a good estimate of the algorithm's run time can be achieved. In addition, the problems associated with job loads, and virtual paging environments are non-existent.

There now remain a number of variables to be identified for the experimental design [MYER79].

Irrelevant Variables: System load, virtual memory paging,
    swapping, cycle stealing

Independent Variables:

    Quantitative: Input size

    Qualitative: Distribution type, algorithm used

    Fixed: Sample size, cell number, Insertionsort cutoff

    Random: Values in the random data file

Dependent variables: Time

Benchmark Model (Controlled Experiment):

    Quicksort vs. DPS

Practical Requirements: No array size may exceed 32767 in
    PL/I, Time is money.

5-58

Experimental Design:

Benchmark Distributive Partitioning Sorting algorithm against Quicksort, and compare the results to those published in [DOBO79]. The version of Quicksort used is the Sedgewick implementation, and DPS uses an Insertionsort cutoff at 9 and a median chosen to be the mean of the max and min, or midrange, (max+min)/2.

Determine run times using

Algorithms	Distributions	Input Sizes
DPS (mdrg)	Uniform	500
median = midrange		1000
DPS (median)	Normal	5000
exact median selection		10000
Ranking	Poisson	20000
CDF	Exponential	30000

Analyze

. The effect of a distribution with an algorithm.

. The effect of the input size on an algorithm.

. An algorithm's performance against another

algorithm within a distribution.

Each experiment consists of five runs. The run times and various percentages are taken from the average of these five runs. Each of the five runs contains different random values as data.

It should be noted that the Poisson distribution is continuous as opposed to discrete Poisson.

Due to the nature of the experimental design, there are certain constraints on what can be said about the conclusions to be reached. Essentially the experiments are simulating the run time as if the program were being given stand alone time on an IBM 370/158. In reality, operating system dependent factors are difficult to measure, and would contribute to the actual run time. But these have been eliminated in the hope of producing good relative execution time results.

Since single precision real numbers were used in previous DPS experiments, they were used here also. Due to the large amount of arithmetic operations in DPS, changing the input stream to integer or double precision could radically change the timings. Alphanumeric keys would have to be adapted in some way so DPS could work with them. These are problems which do not occur in comparison-based sorts.

III.3 Discussion of Fixed Variables

There are three fixed variables that need values assigned to them. One of these is the Insertionsort cutoff point for DPS. The optimum cutoff for DPS(mdrg) in the uniform case was determined by starting with a value of 6 and incrementing by 1 until it was found. A cutoff of 9 or 10 was found to work best. Since 9 was known to be the cutoff for Quicksort, 9 was also chosen as the cutoff for DPS. It should be apparent that a different cutoff might be possible for each DPS algorithm, input size, and distribution. To avoid a lot of extra work to determine cutoffs for the two DPS methods, and out of fairness

5-60

to DPS(mdrg) in the uniform case, 9 was used as the Insertionsort cutoff in all experiments. This value should also be optimum for the adaptive methods if they do indeed transform the distributions to a uniform spread.

Another fixed variable is the number of sampling cells used in the adaptive methods. The value chosen for this variable is closely related to the sample size. The number of cells and the sample size should be the same for Ranking and CDF out of fairness to each. Ranking has a binary search that requires that the number of cells be one less than a power of two. Good values to choose might be 7, 15, 31, 63, and 127. The higher the number, the more work the O(log n) binary search will have to do.

The idea of the CDF method is to sample the Cumulative Distribution Function. It would be desirable if the cells could sample statistically good proportions of the range of the data. If each cell samples a 1% or 3% proportion, then an appropriate number of cells can be chosen which divides the range into 1% or 3% intervals. Figure III.1 lists the possible cell proportions.

Figure III.1

Cell Proportions

k	# Cells ($2^k - 1$)	Proportions 100/# Cells
3	7	14.29
4	15	6.66
5	31	3.23
6	63	1.59
7	127	.79

5-61

Values of 7 and 15 cells would not divide the range into small enough proportions to be of much accuracy. 127 cells would have too large a k value for the binary search. 63 cells is not close to either 1% or 2% and it would be ambiguous to choose one or the other.

If 31 cells are chosen, the range is roughly divided into 3% proportions. This is small enough to have a good amount of accuracy, and efficient enough for use in a binary search.

The sample size now needs to be determined. Using proportional sample statistics and standard normal distribution tables, a good sample size can be found if 3% proportions of the data are desired.

Given 3% proportions with .013 error and 90% confidence, then a good sample size is 469. Since 31*15 is 465, a size of 465 was chosen. This also allows for easy adaptability to 15 cells if there is a large overhead with 31 cells.

CHAPTER IV

RESULTS AND CONCLUSIONS

IV.1 Expectations, Results, and Conclusions

Before the experiments were conducted, one might hypothesize certain results to occur. As already discussed, the adaptive methods should transform an unknown distribution into a uniform distribution. It is to be expected that in many ways, the performance of these methods for skewed distributions will resemble the DPS methods in the uniform case. The only exception here would be the run time differences due to overheads in the adaptive methods.

In the uniform case these methods will be distributing items into buckets, and on the first pass, one might expect a certain percentage of the buckets to be used. Certainly, all of the buckets will not get used, and using combinatorial analysis, the expected percentage of buckets used can be found.

Given one item and n buckets, the probability that the i-th bucket is empty is

$$(\frac{n-1}{n}) \tag{1}$$

For all n items, the probability the i-th bucket is empty is

$$(\frac{n-1}{n})^n \tag{2}$$

So the percentage of buckets being used is

$$100 \cdot (1 - (\frac{n-1}{n})^n) \tag{3}$$

Since

$$\lim_{n \to \infty} (\frac{n-1}{n})^n = \frac{1}{e} \tag{4}$$

The expected percentage is

$$100 \cdot (1 - \frac{1}{e}) = 63.21\%$$

Recall that one of the concerns of analyzing algorithms is the constant of proportionality of the theoretic order of magnitude. Determining these constants based on the observed run times should help determine where crossovers might occur, that is, at what input size one algorithm begins outperforming another.

Table 1.1 illustrates the Benchmarking results. The Time Quicksort/Time DPS gives a percentage of how much better DPS is performing than Quicksort. Comparing results observed with those previously published in [DOBO79], (which appear in the Expected column) it can be seen that this benchmark of DPS outperforms previous results except for small sample sizes. Figure IV.1 illustrates these results graphically. Notice there is a crossover where DPS begins to outperform Quicksort somewhere below 2000 items. Sedgewick [SEDG78] noted that the expected run time of his implementation of Quicksort would be proportional to

$$10.6286N \log N + 2.116N$$

Table 1. Benchmark

| Input Size | Run Time Millisecs | | Time Quicksort/Time DPS | | |
	Quicksort	DPS	Expected	Observed	Δ%
2000	464.25	414.86	1.16	1.12	-.04
5000	1302.88	1036.80	1.24	1.26	.02
10000	2818.94	2051.62	1.27	1.37	.10
15000	4420.06	3108.83	1.28	1.42	.14
30000	9413.27	6222.61	1.35 *	1.51	.16
50000	---		1.46	1.63 *	.17

* estimate

Figure IV.1

Benchmark
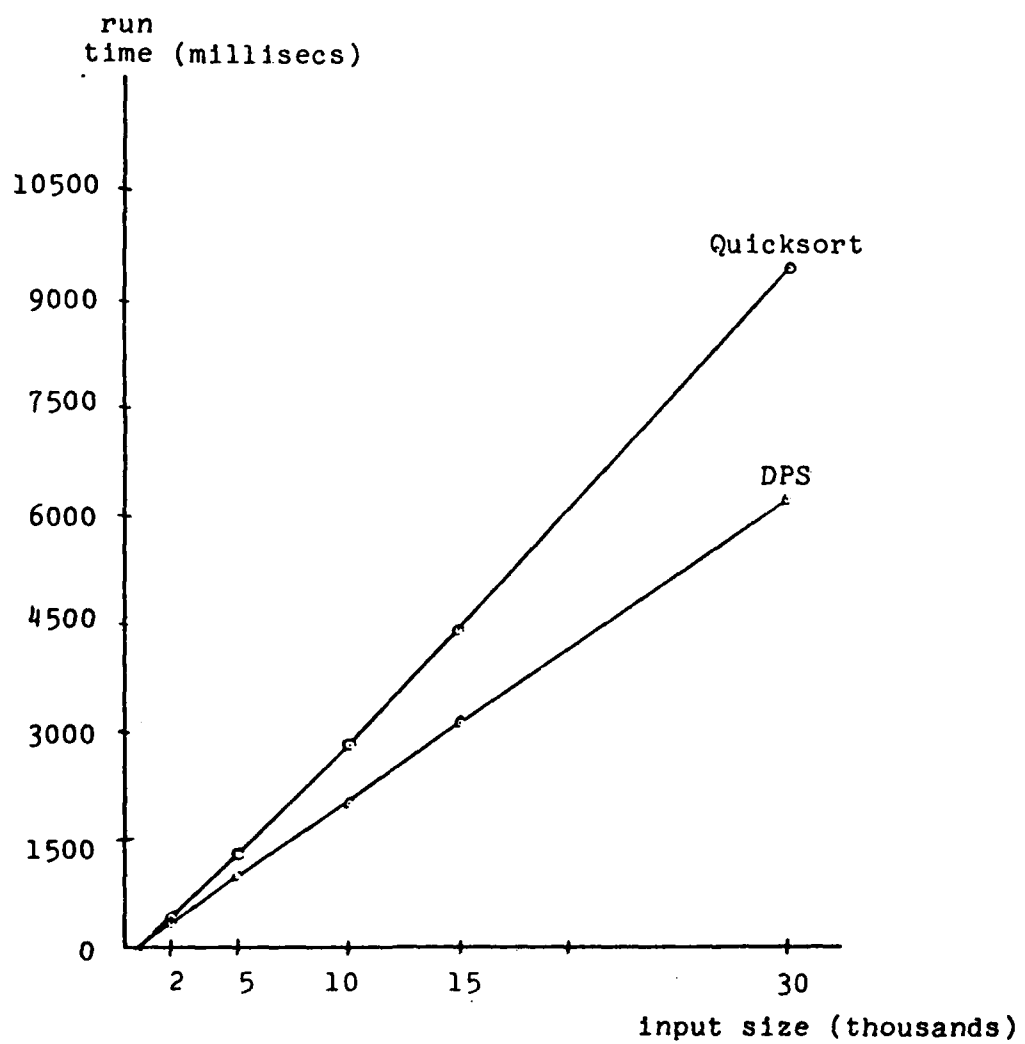
In fact, the run times (in microseconds) observed are approximately double this formula.

Since DPS is $O(n)$, one should expect to fit a linear expression to its run times. For these experiments, the expression

$$207N$$

works very well. To find the crossover point, the equations are set equal to one another and solved for N.

$$2 (10.6286N \log N + 2.116N) = 207N$$

$$\log N = 9.5388$$

$$N \approx 744$$

which conforms well to Figure IV.1.

Although Quicksort accesses items directly, and DPS accesses items indirectly through a pointer list, DPS is still faster. In reality, one must consider that as the algorithms begin recursing, Quicksort will demonstrate a higher degree of locality than DPS in searching for items, and therefore generate fewer page faults. As pointed out earlier, this operating system factor does not play a role in these experiments.

The first four sets of tables to be presented list results of how well each algorithm performed on each of the distributions. It is expected that the DPS methods perform worse as the distribution becomes skewed, and the adaptive methods will behave approximately constant.

Tables 2.1 - 2.3 show results for DPS where the median is chosen to be (max+min)/2.

Table 2.1 is a table of the largest bucket sizes created on the first pass. The average of the largest buckets from the five runs and the maximum bucket size out of the five runs are listed. As expected, the sizes get larger as the distributions become more skewed.

Table 2.2 shows what percentage of buckets are used in the first pass through the data. This reflects how efficiently the algorithm is distributing the data into buckets. A lower percentage might indicate that the algorithm is doing a certain amount of recursion to handle the larger bucket sizes being created. The first column is the percentage of buckets with sizes greater than or equal to one, and the second column is for those with sizes greater than or equal to two. As was expected, DPS used fewer buckets as the distribution became skewed. In the uniform case the percentage of buckets used is around 63.1% - 63.5%. This collaborates well with the expected 63.2% derived earlier. It is slightly higher here due to the uneven partitioning of bucket intervals as a result of the median selection and distribution expressions. Interestingly, the percentage of buckets with at least 2 items did not vary greatly, whereas the percentage of buckets with at least one item varied between 16% to 64% throughout the entire series of experiments.

Table 2.3 lists the run times observed for DPS(mdrg). The greatest difference was observed for the skewed exponential case, as would be expected. The other distributions were fairly consistent.

Table 2. DPS (mdrg) Experiments

Table 2.1

Largest Bucket Sizes

	Uniform		Normal		Poisson		Exponential	
	Avg.	Max.	Avg.	Max.	Avg.	Max.	Avg.	Max.
500	4.4	5	7.2	8	6.8	8	9.4	12
1000	5.4	6	8.2	11	7.8	9	12.6	14
5000	6.2	7	9.8	11	9.4	10	17.4	19
10000	6.4	8	9.8	11	10.2	13	18.6	21
20000	6.4	7	10.0	11	10.6	12	21.4	23
30000	7.4	8	10.2	11	10.8	11	23.0	24

Table 2.2

% of Filled Buckets (First Pass)

	Uniform		Normal		Poisson		Exponential	
	%>=1	%>=2	%>=1	%>=2	%>=1	%>=2	%>=1	%>=2
500	63.84	26.32	49.68	26.80	49.48	28.08	40.20	22.76
1000	63.70	26.38	48.82	27.00	50.00	27.74	33.74	20.68
5000	63.48	26.33	46.24	27.10	45.16	27.72	30.05	19.42
10000	63.18	26.53	45.37	26.95	42.54	27.33	27.85	18.34
20000	63.28	26.33	45.10	26.98	41.68	27.21	25.99	17.44
30000	63.43	26.38	44.58	26.92	41.63	27.11	24.30	16.64

Table 2.3

Run Times (millisecs)

	Uniform	Normal	Poisson	Exponential
500	103.65	104.44	104.33	108.22
1000	201.54	208.90	208.55	223.38
5000	1036.80	1043.37	1045.45	1160.49
10000	2051.62	2085.85	2095.75	2369.29
20000	4144.05	4170.01	4195.50	4825.32
30000	6222.61	6247.89	6362.22	7159.07

Tables 3.1-3.3 illustrate observations for DPS which employs the Floyd-Rivest expected time 1.5n exact median selection algorithm. It would be expected that while the overall efficiency might improve, there would be a certain amount of overhead in run times associated with the median selection.

Overall these tables demonstrate the same characteristics Tables 2.1-2.3 did. There were two major differences to be noted. Table 3.2 shows that while there was a tendency for the algorithm to distribute items less efficiently for skewed distributions, the Poisson data was slightly more efficient than the normal data. This is because Poisson generated more buckets with exactly size 1, and fewer with at least 2, than the normal case.

The other observation to be made is found in Table 3.3. For small input sizes, DPS(median) performed better for the skewed distributions than for the uniform cases. This is mostly due to fewer buckets that need to be handled for skewed data. As a result, the algorithm runs slightly better.

Better run times for the normal distribution over the uniform distribution were not observed, as they were in [DOB078a].

In Tables 4.1-4.3, data for the Ranking Method is presented. A concern for this method is that it performs consistently through the various distributions.

Table 3. DPS(median) Experiments

Table 3.1

Largest Bucket Sizes

	Uniform		Normal		Poisson		Exponential	
	Avg.	Max.	Avg.	Max.	Avg.	Max.	Avg.	Max.
500	4.6	5	7.7	9	7.2	9	8.8	11
1000	5.6	6	8.0	9	7.8	10	11.6	14
5000	6.2	7	10.0	11	10.0	12	15.0	17
10000	6.6	7	10.0	12	12.0	14	17.6	21
20000	6.8	7	10.4	12	13.8	15	19.8	22
30000	7.2	9	10.6	11	12.2	14	21.8	27

Table 3.2

% of Filled Buckets (First Pass)

	Uniform		Normal		Poisson		Exponential	
	% >= 1	% >= 2	% >= 1	% >= 2	% >= 1	% >= 2	% >= 1	% >= 2
500	63.24	26.52	50.56	26.96	53.76	25.96	52.88	24.92
1000	63.54	26.48	48.82	27.24	54.32	26.66	49.38	24.14
5000	63.30	26.33	46.25	27.01	51.87	25.68	46.46	23.05
10000	63.31	26.31	45.40	26.96	50.27	25.43	45.76	22.74
20000	63.42	26.37	45.17	26.85	49.80	25.30	44.88	22.14
30000	63.34	26.44	44.58	26.90	49.74	25.10	44.06	21.64

Table 3.3

Run Times (millisecs)

	Uniform	Normal	Poisson	Exponential
500	145.16	134.40	133.66	139.74
1000	257.47	257.58	254.11	270.08
5000	1232.72	1241.65	1240.33	1296.61
10000	2439.84	2454.41	2494.27	2652.15
20000	4823.61	4847.84	4927.75	5368.40
30000	7210.15	7246.58	7338.22	8171.92

Table 4.1 conforms well to this expectation. The bucket sizes did not vary greatly in the Exponential cases as compared to the DPS programs. Table 4.2 more vividly shows that the algorithm is behaving consistently. For each distribution, the percentages remained fairly constant. The percentage of buckets with at least one item came very close to the predicted 63.2%. Table 4.3 illustrates that the run times also behave very consistently. There is very little run time variance through distributions. Overall it can be concluded that the Ranking Method is a valid Adaptive Method for DPS and deserves further consideration.

Last in this series are Tables 5.1-5.3 for the Cumulative Distribution Function Method. It can easily be seen in these tables how well the algorithm performs across distributions. Each run performs equally well regardless of skewness. This strongly supports the theory that the sample Cumulative Distribution Function effectively transforms an unknown distribution into a uniform distribution.

The next three tables, 6.1-6.3, show the number of second level passes using recursion that were needed for each experiment. Uniform cases are not listed because none of the experiments ever recursed to the second level. It should also be noted that none of the experiments ever recursed to the third level. Two passes on a bucket always sufficed to do the sorting.

Table 4. Ranking Experiments

Table 4.1

Largest Bucket Sizes

	Uniform		Normal		Poisson		Exponential	
	Avg.	Max.	Avg.	Max.	Avg.	Max.	Avg.	Max.
500	4.2	5	5.0	6	5.4	6	4.6	5
1000	6.0	7	6.2	7	5.8	7	7.0	10
5000	6.8	8	8.6	12	9.8	13	9.4	13
10000	6.6	7	9.4	13	12.6	15	10.6	17
20000	7.0	8	10.2	11	14.6	17	13.6	21
30000	7.2	8	10.2	11	15.4	17	14.2	23

Table 4.2

% of Filled Buckets (First Pass)

	Uniform		Normal		Poisson		Exponential	
	$\% >= 1$	$\% >= 2$	$\% >= 1$	$\% >= 2$	$\% >= 1$	$\% >= 2$	$\% >= 1$	$\% >= 2$
500	63.64	26.68	64.12	25.20	61.88	26.04	63.16	26.88
1000	63.78	25.52	62.26	25.96	62.02	26.86	62.48	25.76
5000	62.57	26.34	61.70	26.20	61.55	26.28	62.02	26.27
10000	62.36	26.48	61.28	26.15	61.20	26.17	61.53	26.22
20000	62.49	26.43	61.16	26.29	61.08	26.18	61.61	26.17
30000	62.49	26.40	60.94	26.40	61.22	26.05	61.51	26.18

Table 4.3

Run Times (millisecs)

	Uniform	Normal	Poisson	Exponential
500	257.96	257.94	256.20	258.33
1000	428.78	427.58	425.91	428.30
5000	1789.90	1785.12	1784.10	1788.02
10000	3485.41	3479.50	3483.73	3487.98
20000	6857.82	6869.38	6879.54	6889.47
30000	10277.25	10251.06	10273.79	10294.71

Table 5. CDF Experiments

Table 5.1

Largest Bucket Sizes

	Uniform		Normal		Poisson		Exponential	
	Avg.	Max.	Avg.	Max.	Avg.	Max.	Avg.	Max.
500	4.6	6	4.6	5	5.6	7	4.8	5
1000	5.6	7	5.6	7	5.2	6	5.2	6
5000	6.8	7	6.4	7	6.6	7	6.4	8
10000	6.8	8	6.8	7	6.4	7	6.6	7
20000	7.0	7	7.8	9	7.4	8	7.4	8
30000	7.4	8	7.4	9	8.0	10	8.0	9

Table 5.2

% of Filled Buckets (First Pass)

	Uniform		Normal		Poisson		Exponential	
	% >=1	% >=2	% >=1	% >=2	% >=1	% >=2	% >=1	% >=2
500	63.92	26.60	64.36	25.92	62.60	26.00	63.80	26.24
1000	63.32	26.66	63.08	26.08	62.08	27.00	63.02	26.30
5000	62.61	26.20	62.43	26.42	62.21	26.54	62.23	26.34
10000	62.43	26.17	62.30	26.46	62.25	26.49	62.08	26.59
20000	62.35	26.30	62.29	26.38	62.13	26.62	62.02	26.76
30000	62.50	26.27	62.09	26.60	62.08	26.62	61.89	26.74

Table 5.3

Run Times (millisecs)

	Uniform	Normal	Poisson	Exponential
500	127.37	127.53	127.50	127.46
1000	234.27	233.86	233.70	233.93
5000	1085.52	1083.71	1084.46	1084.25
10000	2148.87	2145.85	2148.11	2147.49
20000	4274.07	4265.47	4274.10	4272.52
30000	6398.65	6378.67	6399.72	6398.81

A couple of observations can be made about the nature of the data presented in these tables. The number of second level passes is a good indication of how much work an algorithm is doing. The fewer passes, the less work being performed. The number of second level passes is a direct result of how well the data was distributed in the first pass. The results of these tables compare well with the run times observed in Tables 7.0-10.0.

As expected, the number of second level passes increased within an algorithm as the data became more skewed. Another observation is that Ranking needed only a small number of passes, and CDF did not use a second level of recursion except in one experiment! In this respect, CDF far outperformed the other algorithms. Again, this further supports the theory of the Cumulative Distribution Function acting as a uniform transformation. This is true to a lesser extent for the Ranking algorithm.

The next four series of tables present how the algorithms performed in any one distribution. These are especially helpful on showing how the algorithms are competing against one another.

The first three tables, 7.1-7.3, show the Uniform case. As can be seen in 7.1 and 7.2, all algorithms appear to be performing equally well with respect to the uniform case. However, Table 7.3 shows the first large discrimination between the methods. The second column of figures in these run times

Table 6. Number of Second Level Recursions

Table 6.1

Normal

	DPS mdrg		DPS median		Ranking		CDF	
	Avg.	Max.	Avg.	Max.	Avg.	Max.	Avg.	Max.
500	0	0	0	0	0	0	0	0
1000	.2	1	0	0	0	0	0	0
5000	1.2	2	1.0	2	.2	1	0	0
10000	1.4	4	1.0	3	.4	1	0	0
20000	2.2	5	2.0	3	1.8	5	0	0
30000	2.0	3	3.4	5	1.4	3	0	0

Table 6.2

Poisson

	DPS mdrg		DPS median		Ranking		CDF	
	Avg.	Max.	Avg.	Max.	Avg.	Max.	Avg.	Max.
500	0	0	0	0	0	0	0	0
1000	0	0	.2	1	0	0	0	0
5000	.6	1	1.2	3	.6	1	0	0
10000	1.8	4	7.6	13	1.8	3	0	0
20000	3.0	7	15.8	25	4.6	7	0	0
30000	4.6	7	24.4	36	6.0	13	.2	1

Table 6.3

Exponential

	DPS mdrg		DPS median		Ranking		CDF	
	Avg.	Max.	Avg.	Max.	Avg.	Max.	Avg.	Max.
500	1.2	3	1.2	3	0	0	0	0
1000	7.2	14	2.4	5	.2	1	0	0
5000	69.6	92	27.0	43	.8	3	0	0
10000	171.2	258	74.6	113	2.6	9	0	0
20000	431.4	519	195.2	237	6.6	23	0	0
30000	763.0	786	351.6	363	11.8	31	0	0

represent the percentage improvement of DPS(mdrg) over the given algorithm. For example, a 1.19 means that DPS(mdrg) runs 19% faster than the given algorithm in that experiment. The conclusion to be reached from Table 7.3 is that as the sample gets larger, DPS(mdrg) is about 16% faster than DPS(median), 65% faster than Ranking, and 3% faster than CDF. The Uniform experiment times are represented graphically in Figure IV.2. (Since the Normal and Poisson experiments have relatively the same proportions as Uniform, as seen in Tables 8 and 9, this graph would be similar in those distributions as well.)

The reason for these time differences can be explained in the overhead associated with each method as compared to DPS(mdrg). Formulas can be fit to the run times to approximate what the constants of proportionality are. These expressions yield times in microseconds, and fit better as the sample size increases.

	Time (microseconds)	Space
DPS(mdrg)	207N	2N
CDF	207N + 5.6N + 22600	2N + 3M
DPS(median)	207N + 31.6N + 52000	2N + N
Ranking	207N + 132N + 96255	2N + M + m

N = # items, M = # cells, m = sample size

CDF Overhead: Sample frequency, line fits, and larger partitioning expression

DPS Overhead: Median selection

Ranking Overhead: Quicksort, Binary search, and complex partitioning expression

Table 7. Uniform Experiments

Table 7.1

Largest Bucket Sizes

	DPS mdrg		DPS median		Ranking		CDF	
	Avg.	Max.	Avg.	Max.	Avg.	Max.	Avg.	Max.
500	4.4	5	4.6	5	4.2	5	4.6	6
1000	5.4	6	5.6	6	6.0	7	5.6	7
5000	6.2	7	6.2	7	6.8	8	6.8	7
10000	6.4	8	6.6	7	6.6	7	6.8	8
20000	6.4	7	6.8	7	7.0	8	7.0	7
30000	7.4	8	7.2	9	7.2	8	7.4	8

Table 7.2

% of Filled Buckets (First Pass)

	DPS mdrg		DPS median		Ranking		CDF	
	%>=1	%>=2	%>=1	%>=2	%>=1	%>=2	%>=1	%>=2
500	63.84	26.32	63.24	26.52	63.64	26.68	63.92	26.60
1000	63.70	26.38	63.54	26.48	63.78	25.52	63.32	26.66
5000	63.48	26.33	63.30	26.33	62.57	26.34	62.61	26.20
10000	63.18	26.53	63.31	26.31	62.36	26.48	62.43	26.17
20000	63.?8	26.33	63.42	26.37	62.49	26.43	62.35	26.30
30000	63.43	26.38	63.34	26.44	62.49	26.40	62.50	26.27

Table 7.3

Run Times (millisecs)

	DPS mdrg	DPS median	$\frac{Dmed}{Dmdrg}$	Ranking	$\frac{Ranking}{Dmdrg}$	CDF	$\frac{CDF}{Dmdrg}$
500	103.65	145.16	1.40	257.96	2.49	127.37	1.23
1000	201.54	257.47	1.28	428.78	2.18	234.27	1.16
5000	1036.80	1232.72	1.19	1789.90	1.73	1085.52	1.05
10000	2051.62	2439.84	1.16	3485.41	1.70	2148.87	1.05
20000	4144.05	4823.61	1.16	6857.82	1.66	4274.07	1.03
30000	6222.61	7210.15	1.16	10277.25	1.65	6398.65	1.03

Figure IV.2

Uniform Experiments
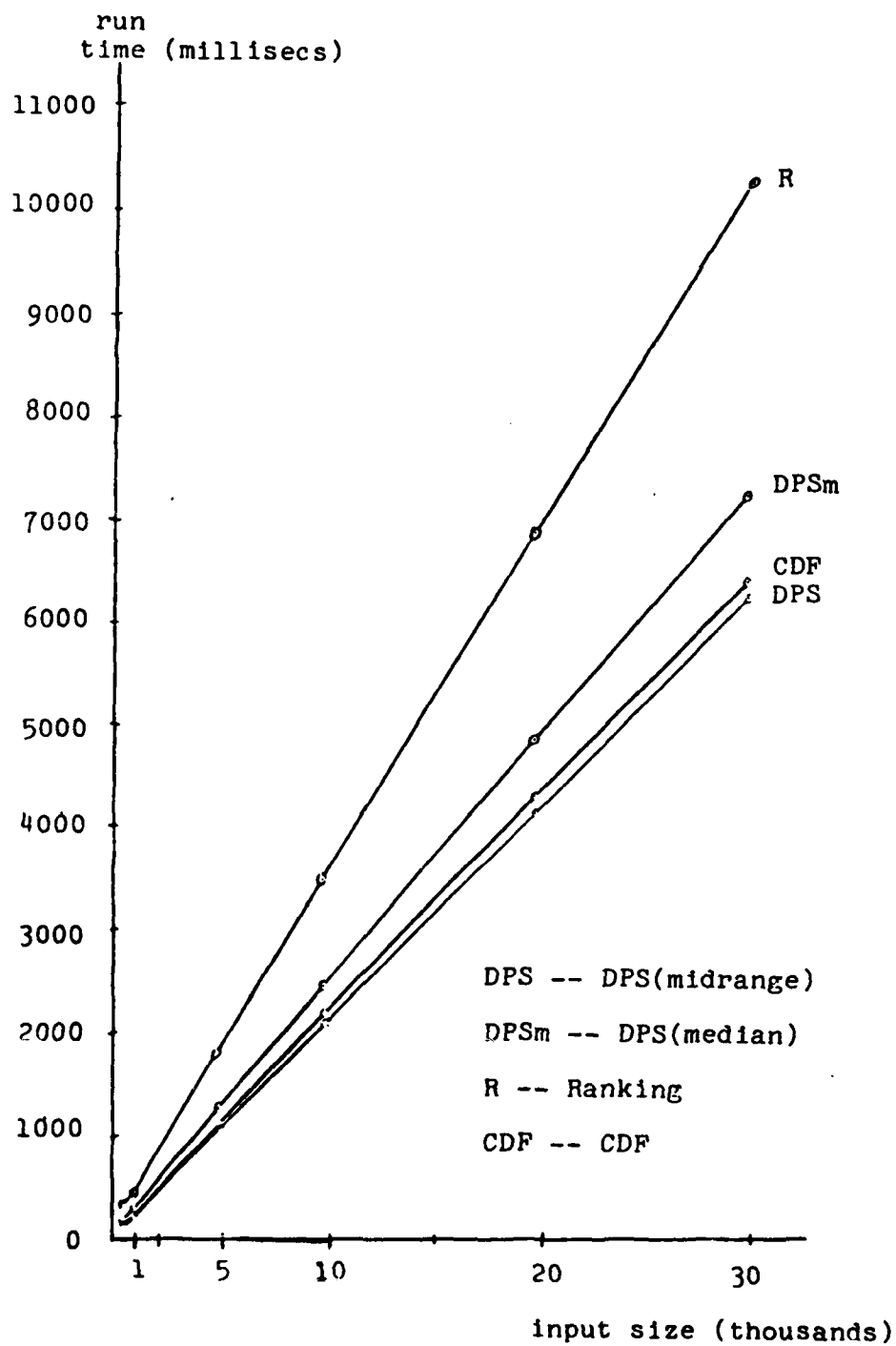
It is here where the importance of constants of proportionality in orders of magnitude is truly appreciated. Although the overhead for Ranking is very high due to the binary searching and initial sorting, only about an estimated 20% can be saved on the run time if a smaller sample size and 15 sampling cells are used.

It now becomes interesting to see what happens as the data becomes more skewed. Tables 8.1-8.3 describe the Normal case. Table 8.1 shows that the adaptive methods have smaller bucket sizes. Table 8.2 illustrates how efficiently the adaptive methods distribute the items for buckets with at least 1 item. The efficiency is better by roughly 15%. Table 8.3 shows that the run times are in the same proportion as they were for the uniform case.

Tables 9.1-9.3 illustrate the Poisson experiments. The results here are much like those of the Normal experiments and the same conclusions can be reached.

Tables 10.1-10.3 list the results of the experiments with an Exponential distribution. Table 10.1 shows that the adaptive methods outperform the DPS methods, and Table 10.2 shows that the adaptive methods distribute items much more efficiently. However, the major conclusion to be reached is in Table 10.3. For input sizes greater than about 2500, CDF outperforms the DPS(mdrg) algorithm. For 20000 to 30000 items, it runs about 12% better. Figure IV.3 illustrates the data in Table 10.3.

Table 8. Normal Experiments

Table 8.1

Largest Bucket Sizes

	DPS mdrg		DPS median		Ranking		CDF	
	Avg.	Max.	Avg.	Max.	Avg.	Max.	Avg.	Max.
500	7.2	8	7.2	9	5.0	6	4.6	5
1000	8.2	11	8.0	9	6.2	7	5.6	7
5000	9.8	11	10.0	11	8.6	12	6.4	7
10000	9.8	11	10.0	12	9.4	13	6.8	7
20000	10.0	11	10.4	12	10.2	11	7.8	9
30000	10.2	11	10.6	11	10.2	11	7.4	9

Table 8.2

% of Filled Buckets (First Pass)

	DPS mdrg		DPS median		Ranking		CDF	
	$\% >= 1$	$\% >= 2$	$\% >= 1$	$\% >= 2$	$\% >= 1$	$\% >= 2$	$\% >= 1$	$\% >= 2$
500	49.68	26.80	50.56	26.96	64.12	25.20	64.36	25.92
1000	48.82	27.00	48.82	27.24	62.26	25.96	63.08	26.08
5000	46.24	27.10	46.25	27.01	61.70	26.20	62.43	26.42
10000	45.37	26.95	45.40	26.96	61.28	26.15	62.30	26.46
20000	45.10	26.98	45.17	26.85	61.16	26.29	62.30	26.46
30000	44.58	26.92	44.58	26.90	60.94	26.40	62.09	26.60

Table 8.3

Run Times (millisecs)

	DPS mdrg	DPS median	$\dfrac{Dmed}{Dmdrg}$	Ranking	$\dfrac{Ranking}{Dmdrg}$	CDF	$\dfrac{CDF}{Dmdrg}$
500	104.44	134.40	1.29	257.94	2.47	127.53	1.22
1000	208.90	257.58	1.23	427.58	2.05	233.86	1.12
5000	1043.37	1241.65	1.19	1785.12	1.71	1083.71	1.04
10000	2085.85	2454.41	1.18	3479.50	1.67	2145.85	1.03
20000	4170.01	4847.84	1.16	6869.38	1.65	4265.47	1.02
30000	6247.89	7246.58	1.16	10251.06	1.64	6378.67	1.02

Table 9. Poisson Experiments

Table 9.1

	DPS mdrg		DPS median		Ranking		CDF	
	Avg.	Max.	Avg.	Max.	Avg.	Max.	Avg.	Max.
500	6.8	8	7.2	9	5.4	6	5.6	7
1000	7.8	9	7.8	10	5.8	7	5.2	6
5000	9.4	10	10.0	12	9.8	13	6.6	7
10000	10.2	13	12.0	14	12.6	15	6.4	7
20000	10.6	12	13.8	15	14.6	17	7.4	8
30000	10.8	11	12.2	14	15.4	·17	8.0	10

Table 9.2

% of Filled Buckets (First Pass)

	DPS mdrg		DPS median		Ranking		CDF	
	%>=1	%>=2	%>=1	%>=2	%>=1	%>=2	%>=1	%>=2
500	49.48	28.08	53.76	25.96	61.88	26.04	62.60	26.00
1000	50.00	27.74	54.32	26.66	62.02	26.86	62.08	27.00
5000	45.16	27.72	51.87	25.68	61.55	26.28	62.21	26.54
10000	42.54	27.33	50.27	25.43	61.20	26.17	62.25	26.49
20000	41.68	27.21	49.80	25.30	61.08	26.18	62.13	26.62
30000	41.63	27.11	49.74	25.10	61.22	26.05	62.08	26.62

Table 9.3

Run Times (millisecs)

	DPS mdrg	DPS median	Dmed/Dmdrg	Ranking	Ranking/Dmdrg	CDF	CDF/Dmdrg
500	104.33	133.66	1.28	256.20	2.46	127.50	1.22
1000	208.55	254.11	1.22	425.91	2.04	233.70	1.12
5000	1045.45	1240.33	1.19	1784.10	1.71	1084.46	1.04
10000	2095.75	2494.27	1.19	3483.73	1.66	2148.11	1.02
20000	4195.50	4927.75	1.17	6879.54	1.64	4274.10	1.02
30000	6362.22	7338.22	1.15	10273.89	1.61	6399.72	1.006

Table 10. Exponential Experiments

Table 10.1

Largest Bucket Sizes

	DPS mdrg		DPS median		Ranking		CDF	
	Avg.	Max.	Avg.	Max.	Avg.	Max.	Avg.	Max.
500	9.4	12	8.8	11	4.6	5	4.3	5
1000	12.6	14	11.6	14	7.0	10	5.2	6
5000	17.4	19	15.0	17	9.4	13	6.4	8
10000	18.6	21	17.6	21	10.6	17	6.6	7
20000	21.4	23	19.8	22	13.6	21	7.4	8
30000	23.0	24	21.8	27	14.2	23	8.0	9

Table 10.2

% of Filled Buckets (First Pass)

	DPS mdrg		DPS median		Ranking		CDF	
	%>=1	%>=2	%>=1	%>=2	%>=1	%>=2	%>=1	%>=2
500	40.20	22.76	52.88	24.92	63.16	26.88	63.80	26.24
1000	33.74	20.68	49.38	24.14	62.48	25.76	63.02	26.30
5000	30.05	19.42	46.46	23.05	62.02	26.27	62.23	26.34
10000	27.85	18.34	45.76	22.74	61.53	26.22	62.08	26.59
20000	25.99	17.44	44.88	22.14	61.61	26.17	62.02	26.76
30000	24.30	16.64	44.06	21.64	61.51	26.18	61.89	26.74

Table 10.3

	DPS mdrg	DPS median	Dmed/Dmdrg	Ranking	Ranking/Dmdrg	CDF	CDF/Dmdrg	Dmdrg/CDF
500	108.22	139.74	1.29	258.33	2.39	127.46	1.18	
1000	223.38	270.08	1.21	428.30	1.92	233.93	1.05	
5000	1160.49	1296.61	1.12	1788.02	1.54	1084.25	.93	1.07
10000	2369.29	2652.15	1.12	3487.98	1.47	2147.49	.91	1.10
20000	4825.32	5368.40	1.11	6889.47	1.43	4272.52	.89	1.13
30000	7159.07	8171.92	1.14	10294.71	1.44	6398.81	.89	1.12

Figure IV.3

Exponential Experiments

run time (millisecs) vs input size (thousands)

DPS -- DPS(midrange)

DPSm -- DPS(median)

R -- Ranking

CDF -- CDF
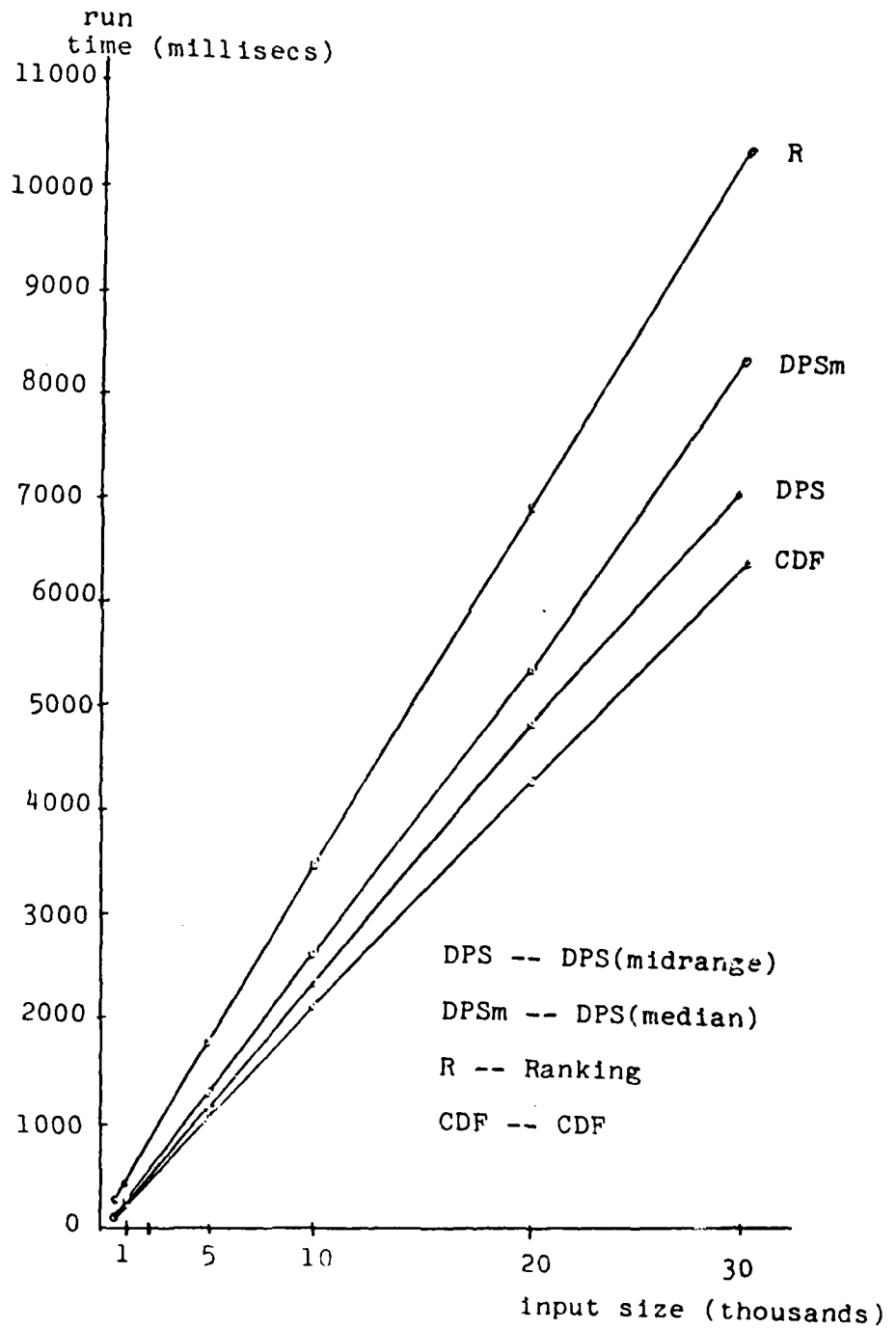
Tables 11.1-11.3 demonstrate how well the algorithm run against DPS(mdrg) across the distributions. It is interesting to note the consistency of the percentages across distributions. Tables 11.1-11.3 are illustrated graphically in Figure IV.4.

IV.2 Summary of Conclusions

The reader may have noticed that up to now experiments have dealt with various types of distributions, and little has been done with the worst case. An expunential distribution exemplifies a typical bad case of data for DPS. Where the worst case for Quicksort is a realistic sorted set of items, the worst case for DPS is an impractical set of factorials [DOBO79]. This is by no means a typical case. For these reasons this author feels that worst case experimentation is justified only as a curiosity factor, rather than of any practical importance. The adaptive methods have more than proved themselves on the skewed distributions given to them as input.

It was pointed out at the beginning of this paper that an algorithm should be measured on a number of criteria. Thus far, the algorithms have been thoroughly analyzed for theoretical and practical time and space considerations. Additionally, they should be easily understood, implemented, and maintained. DPS(median) has a very complex median selection algorithm, and Ranking has a lengthy initial Quicksort and a cumbersome binary search to execute. The CDF algorithm, on the other hand, is quite simple minded in its

Table 11. Run Time Percentages ($\frac{\text{Time Alg.}}{\text{Time Dmdrg}}$)

These tables indicate how much longer it takes an algorithm to run for the given distribution as compared to DPS mdrg on that distribution.

Table 11.1

DPS median

	DPSmdrg Uniform	DPSmdrg Normal	Normal	DPSmdrg Poisson	Poisson	DPSmdrg Exp.	Exp.
500	103.65	104.44	1.29	104.33	1.28	108.22	1.29
1000	201.54	208.90	1.23	208.55	1.22	223.38	1.21
5000	1036.80	1043.37	1.19	1045.45	1.19	1160.49	1.12
10000	2051.62	2085.85	1.18	2095.75	1.19	2369.29	1.12
20000	4144.05	4170.01	1.16	4195.50	1.17	4825.32	1.11
30000	6222.61	6247.89	1.16	6362.22	1.15	7159.07	1.14

Table 11.2

Ranking

	Normal	Poisson	Exponential
500	2.47	2.46	2.39
1000	2.05	2.04	1.92
5000	1.71	1.71	1.54
10000	1.67	1.66	1.47
20000	1.65	1.64	1.43
30000	1.64	1.61	1.44

Table 11.3

CDF

	Normal	Poisson	Exponential
500	1.22	1.22	1.18
1000	1.12	1.12	1.05
5000	1.04	1.04	.93 (1.07)
10000	1.03	1.02	.91 (1.10)
20000	1.02	1.02	.89 (1.13)
30000	1.02	1.006	.89 (1.12)

Figure IV.4

Run Time Percentages
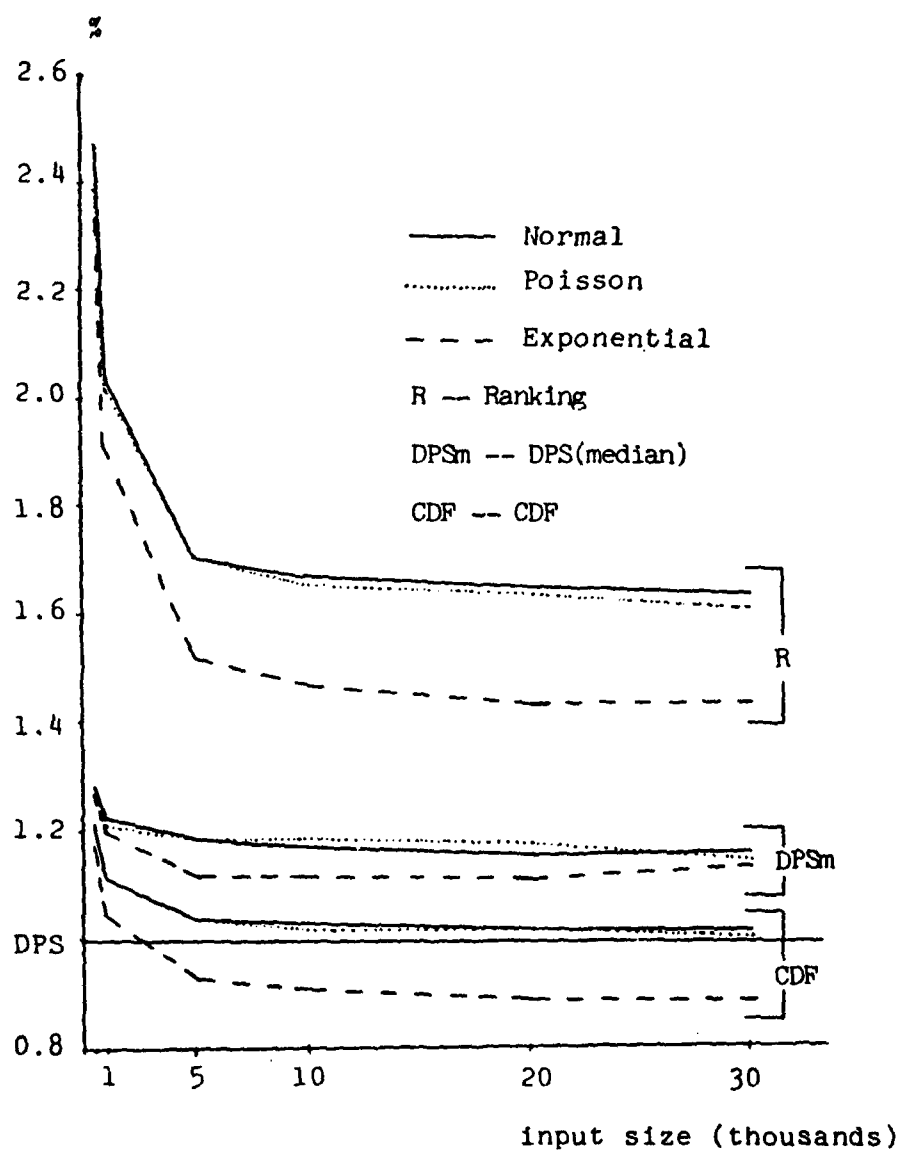
approach, which makes it readily comprehensible. This simplicity lends itself to competitive run times with DPS(mdrg).

For larger input sizes, the CDF algorithm runs to within 4% of DPS(mdrg), and actually outperforms it by 12% in exponential and more skewed cases. Smaller inputs take only a fraction of a second to sort, so overhead is not an important consideration here. When using CDF, we are guaranteed that any unknown distribution will be sorted as quickly and efficiently as though it were a uniform case, and the sorting can be done about as cheaply as the fastest available DPS method. Therefore, there is little to lose, and possibly something to gain, by implementing the Cumulative Distribution Function Adaptive Method for Distributive Partitioning Sorting. It is well worth using.

CHAPTER V

CONSIDERATIONS FOR THE FUTURE

With the Cumulative Distribution Function adaptation, Distributive Partitioning Sorting is an extremely efficient and valuable sorting technique. It easily outperforms Quicksort and other "fast" sorting algorithms. But there remain a number of aspects in which DPS may be even further improved, and a number of areas in which it has future implications.

V.1 Modifications

Some modifications can be suggested to improve the efficiency of DPS. If DPS(mdrg) or DPS(median) is used knowing that the data will typically be symmetrically distributed, then it is not necessary to select a median. All that is needed is to partition the range into n buckets and distribute the items. The median is so close to the mean for these distributions that it is not worth finding or using. If one insists on choosing a median quickly, it would be sufficient to choose the median of a small sample of the data, rather than the entire data set.

CDFDPS does not choose a median. But like the other methods, it has an Insertionsort cutoff. For these experiments Insertionsort was used for bucket sizes of 9 or less. Since about 63.2% of the buckets are used, it would be practical to use some fraction of the partitions. This is because many

buckets with one item can be combined, and still come under the cutoff. The same basic idea was suggested in [KNUT73, DOBO79]. It would be worthwhile to see if there is an optimum number of buckets to use given the cutoff. This could result in a substantial space savings.

V.2 Implications

As is the case with other sorting algorithms, there is some question as to whether DPS is practical for machines other than large mainframes. On microcomputers, if large inputs are used, the answer is, of course, no, due to memory size limitations and slow processing speeds. But with the recent advances in mass storage and CPU speeds on micros, it might not be long before large scale programs become reality on small computers.

There are practical space problems on minicomputers as well, which are largely a function of the amount of available space and the load on the machine. Theoretically there is no reason why DPS could not be implemented on a mini. In reality, in addition to DPS's space overhead, there would be many system and user dependent factors affecting its performance. At some point it may become advantageous to resort to an external sort should system resources become too limited.

It would be very worthwhile to examine adapting DPS to handle alphanumeric keys. This would be of great practical concern for the data processing community, since most sorting in reality is done on name fields of one type or another. The

main concern would be to keep DPS fast, simple, and competitive with other algorithms.

It is fitting here to cite ⟨previous work in the applications of the idea of distributive partitioning. Just as the basic idea of partitioning in Quicksort was used by Floyd for selection, so Allison and Noga have suggested using distributive partitioning in selection [ALLI80]. Van der Nat has suggested adapting distributive partitioning in binary merging and merge sorting applications [VAN79, VAN80]. And, as mentioned earlier, Meijer and Akl have developed a Hybrid of DPS which uses a CDF for known distributions [MEIJ80].

CDFDPS could be generalized to sort n-dimensioned arrays. A CDF in n-dimensions is defined to be:

$$G_X(x_1, x_2, \ldots, x_n) = P(X_1 \leq x_1, X_2 \leq x_2, \ldots, X_n \leq x_n)$$

Finding cumulative frequency probabilities is easily expanded to n-dimensions. Since this function can be considered monotonic increasing in n-dimensions, the resulting transformation from one n-dimensional space to another will be uniform. It should be relatively easy to implement CDFDPS for multi-dimensioned arrays.

Perhaps another way to use the basic idea of CDF distributive partitioning is in hashing applications. An item can be hashed using distributive partitioning for fast lookup and retrieval in databases. Collisions could be handled in any number of ways described in database theory. The hope is that a very fast and simple mechanism can be developed for information

storage and retrieval systems. The hashing process would of course be O(1)!

V.3 In Conclusion...

Distributive Partitioning Sorting has only recently begun to receive the attention it deserves. With the Cumulative Distribution Function adaptation, it can be made to handle all types of unknown distributions equally well. The space considerations can also be minimized as can the run times. Since DPS is practical, fast, and easy to implement, serious consideration should be given to it by the programming community as a viable and cost effective sorting method.

BIBLIOGRAPHY

Articles

ALLI80   Allison, D. C. S., and Noga, M. T., "Selection by
          Distributive Partitioning", Information Processing
          Letters, Vol. 11, No. 1, (August 29, 1980), pp. 7-8.

AKER78   Akers, S. B., "Review #33,338", Computing Reviews, Vol.
          19, No. 8, (August 1978), p. 326.

BLUM78   Blum, M., Floyd, R. E., Pratt, V., Rivest, R. L., and
          Tarjan, R. E., "Time Bounds for Selection", Journal of
          Computer Systems Science, Vol. 7, No. 4, (August
          1973), pp. 448-461.

BURT78   Burton, Warren, "Comments on Sorting by Distributive
          Partitioning", Information Processing Letters, Vol. 7,
          No. 4, (June 1978), p. 205.

DATA78   ----------, "Sorting 30 Times Faster with DPS",
          Datamation, (February 1978), pp. 200-203.

DOBO78a   Dobosiewicz, Wlodzimierz, "Sorting by Distributive
          Partitioning", Information Processing Letters, Vol.
          7, No. 1, (January 12, 1978), pp. 1-6.

DOBO78b   Dobosiewicz, Wlodzimierz, "Author's Reply to Warren
          Burton's   Comments   on   Distributive   Partitioning
          Sorting", Information Processing Letters, Vol. 7, No.
          4, (June 1978), p. 206.

DOBO79   Dobosiewicz, Wlodzimierz, "The Practical Significance
          of D P Sort Revisited", Information Processing
          Letters, Vol. 8, No. 4, (April 30, 1979), pp. 170-172.

DOBO80   Dobosiewicz, Wlodzimierz, "An Efficient Variation of
          Bubblesort", Information Processing Letters, Vol. 11,
          No. 1, (August 29, 1980), pp. 5-6.

FLOY75   Floyd, Robert W., and Rivest, Ronald L., "Expected Time
          Bounds for Selection" and "Algorithm 489: SELECT",
          Communications of the ACM, Vol. 18, No. 3, (March
          1975), pp. 165-173.

FRAZ70   Frazer, W. D., and McKellar, A. C., "Samplesort: A
          Sampling Approach to Minimal Storage Tree Sorting",
          Journal of the ACM, Vol. 17, No. 3, (July 1970), pp.
          496-507.

FUSS79   Fussenegger, Frank and Gabow, Harold N., "A Counting
          Approach to Lower Bounds for Selection Problems",
          Journal of the ACM, Vol. 26, No. 2, (April 1979), pp.
          227-238.

GRIF70   Griffin, Robin and Redish, K. A., "Remark on Algorithm
          347", Communications of the ACM, Vol. 13, No. 1,
          (January 1970), p. 54.

HOAR62   Hoare, C.A.R., "Quicksort", The Computer Journal,
          Vol. 5, No. 1, (April 1962), pp. 10-15.

HUIT79   Huits, Martin, and Kumar, Vipin, "The Practical
          Significance of Distributive Partitioning Sort",
          Information Processing Letters, Vol. 8, No. 4, (April
          1979), pp. 168-169.

IBM78   ----------, "IBM System/370 Model 158 Functional
          Characteristics", IBM Systems Manual GA22-7011-5,
          (September 1978), Appendix B.

JACK79   Jackowski, Boguslaw L., Kubiak, Ryszard, and
          Sokolowski, Stefan, "Complexity of Sorting by
          Distributive Partitioning", Information
          Processing Letters, Vol. 9, No. 2, (August
          1979), p. 100.

LAMA80   Lamagna, Edmund A., Bass, Leonard J., and Anderson,
          Lyle A., "The Performance of Algorithms: A Research
          Plan", URI Dept. of Computer Science, Technical
          Report No. 80-148, (July 1980).

LOES74   Loeser, Rudolf, "Some Performance Tests of 'Quicksort'
          and Descendents", Communications of the ACM, Vol. 17,
          No. 3, (March 1974), pp. 143-152.

MEIJ80   Meijer, Henk, and Akl, Selim G., "The Design and
          Analysis of a New Hybrid Sorting Algorithm",
          Information Processing Letters, Vol. 10, No. 4, 5,
          (July 1980), pp. 213-218.

SCHO76   Schonhage, A., Paterson, M., Pippenger, N., "Finding
          the Median", Journal of Computer and System Sciences,
          Vol. 13, (1976), pp. 184-199.

SEDG78   Sedgewick, Robert, "Implementing Quicksort Programs",
          Communications of the ACM, Vol. 21, No. 10, (October
          1978), pp. 847-856.

SING78   Singleton, Richard C., "Algorithm 347: An Efficient
          Algorithm for Sorting with Minimal Storage",
          Communications of the ACM, Vol. 12, No. 3, (March
          1969), pp. 185-187.

VAN79   Van der Nat, M., "Binary Merging by Partitioning",
         Information Processing Letters, Vol. 8, No. 2,
         (February 1979), pp. 72-75.

VAN80   Van der Nat, M., "A Fast Sorting Algorithm, A Hybrid of
         Distributive and Merge Sorting", Information
         Processing Letters, Vol. 10, No. 3, (April 18, 1980),
         pp. 163-167.

                              Books

AHO74   Aho, Alfred V., Hopcroft, John E., and Ullman,
         Jeffrey D., The Design and Analysis of Computer
         Algorithms, Addison-Wesley, Reading MA, 1974.

BAAS78   Baase, Sara, Computer Algorithms: Introduction to
         Design and Analysis, Addison-Wesley, Reading, MA,
         1978.

CRAM45   Cramer, Harold, Mathematical Methods of Statistics,
         Princeton University Press, Princeton, NJ, 1945.

DANI80   Daniel, Cuthbert, and Wood, Fred S., Fitting Equations
         to Data, John Wiley a Sons, New York, NY, 1980.

ELDE69   Elderton, William P., and Johnson, Norman L.; Systems
         of Frequency Curves, Cambridge University Press,
         London, 1969.

GERA78   Gerald, Curtis F., Applied Numerical Analysis, Academic
         Press, New York, NY, 1976.

GOOD77   Goodman, S. E., and Hedetniemi, S. T., Introduction to
         the Design and Analysis of Algorithms, McGraw-Hill,
         New York, NY, 1977.

GRAY80   Graybeal, Wayne J., and Pooch, Udo W., Simulation:
         Principles amd Methods, Winthrop Publishers, Inc.,
         Cambridge, MA, 1980.

HORO76   Horowitz, Ellis, and Sahni, Sartaj, Fundamentals of
         Data Structures, Computer Science Press, Potomac, MD,
         1976.

HORO78   Horowitz, Ellis, and Sahni, Sartaj, Fundamentals of
         Computer Algorithms, Computer Science Press, Potomac,
         MD, 1978.

KNUT73 Knuth, D. E., The Art of Computer Programming: Sorting
        and Searching, Vol. 3, Addison-Wesley, Reading, MA,
        1973.

MYER79 Myers, Jerome L., Fundamentals of Experimental Design,
        Allyn and Bacon, Inc., Boston, MA, 1979.

STRA76 Strang, Gilbert, Linear Algebra and Its Applications,
        Academic Press, New York, NY, 1976.

ALGORITHMIC COMPLEXITY
Part 6

by

Edmund A. Lamagna+
Edward J. Carney
Purushottam V. Kamat*

EXPECTED BEHAVIOR OF APPROXIMATION ALGORITHMS
FOR THE EUCLIDEAN TRAVELING SALESMAN PROBLEM

Abstract

The behavior of several approximation algorithms for the traveling salesman problem is considered when the points are randomly allocated in the Euclidean plane according to some known distribution. The expected length of the tour constructed by an algorithm is estimated from the order statistics of the distribution of the distance between points. The approximation methods considered include nearest neighbor, arbitrary insert, nearest and cheapest insert, and two methods based on finding the minimal spanning tree (including Christofides' algorithm). For the distribution examined, all of the approximations are shown to produce a tour whose expected length is $O(\sqrt{n})$, where n is the number of points, and at most a small constant factor (ranging from 25.7% to 87.5%) from optimal.

EXPECTED BEHAVIOR OF APPROXIMATION ALGORITHMS
FOR THE EUCLIDEAN TRAVELING SALESMAN PROBLEM

In this paper, several simple polynomial time approximation algorithms for the Traveling Salesman Problem (TSP) are analyzed for their expected performance when the points are distributed in two-dimensional Euclidean space. This version of the TSP may be briefly stated as follows.

> Given a set of points in a plane, find the minimum
> length tour going through each point exactly once.

This problem has a long and interesting history, and many attempts at its solution are surveyed in Bellmore and Nemhauser [2]. Recently, Garey, Graham, and Johnson [7] and Papadimitriou [15] have independently shown that the Euclidean TSP is NP-complete, and thus it appears that an exact solution to the problem for more than several points is computationally infeasible. As a result, much recent interest has centered around the behavior of approximation algorithms, or heuristics, for this problem.

Rosenkrantz, Stearns, and Lewis [17] have investigated the worst case performance of a number of approximation methods for the TSP. The tenor of their work is to examine a specific algorithm and bound the ratio of the length of the approximate tour it produces to that of the optimal tour. They also attempt to construct graphs for which the algorithm performs nearly as badly as this ratio might imply. The best known guaranteed approximation algorithm for the TSP is due to Christofides [3], and always finds a tour whose length is within a factor of $1\frac{1}{2}$ times the optimal solution.

Worst case performance analysis provides a warning to users of an algorithm how far from the optimum the method might deviate. Unfortunately, results of this nature provide little or no insight as to the typical behavior of the method. The algorithm with the best worst case ratio does not necessarily have the best expected one. The expected performance of an algorithm is usually more difficult to ascertain. One has to make assumptions about the distributions of inputs, and realistic assumptions are often mathematically intractable. Even the introduction of slightly complex heuristics can lead to probabilistic dependencies that can be extremely difficult to analyze.

In this paper, we investigate the expected length of the solution to an n-point TSP when the points are randomly allocated in the plane according to some given probability distribution. Using techniques from order statistics, we examine the following approximation algorithms:

- nearest neighbor method
- arbitrary insert method
- nearest and cheapest insert methods
- minimal spanning tree (MST) based method
- Chrisofides' method

All of these methods are found to produce a tour whose expected length is $O(\sqrt{n})$. We also bound the expected tour length from below to show that the algorithms are optimal to within at most a small constant factor. These results tend to confirm experimental work in actually using the algorithms [9]. Further-

more, the results are significant in that the worst case performances of some of the algorithms studied can vary greatly, as shown by Rosenkrantz, et al [17]. The nearest and cheapest insert and MST-based methods always produce a tour whose length is at most twice that of the optimum, but the best known upper bounds on the worst case ratio for the nearest neighbor and arbitrary insert methods grow as log n. In fact, it has been further shown that this logarithmic divergence is unavoidable for the nearest neighbor algorithm.

Some prior related work has been done on the problem studied in this paper. Employing techniques quite different from those used here, Morozinskii [14] has shown that the expected length of a tour constructed by the arbitrary insert method is $O(\sqrt{n})$ and within a factor of 4 of a lower bound on the expected tour length. His result is quite general in the sense that it does not assume any specific distribution of points, but only some weak conditions about the way they are generated. Although our results apply only to the specific distribution considered, our bounds yield more concrete information about the actual tour length. Furthermore, the techniques used in our derivations are general and could be applied to other distributions.

In two frequently cited papers, Karp [10,11] describes an algorithm based on dividing the points into a number of small regions, constructing an optimum tour within each region, and then joining the subtours. Although this algorithm is not guaranteed to find a tour within any specified range of the optimum, Karp states that the method solves the problem to within

1+ε "almost everywhere" for every ε>0. The success of Karp's algorithm depends on a theorem (with a long and difficult proof) by Beardwood, Halton, and Hammersley [1]. This result states that the length of the optimal tour through n points in a bounded plane region of area A is "almost always" proportional to $\sqrt{nA}$ for sufficiently large n. Weide [18] has recently pointed out that some confusion exists when interpreting and comparing such results due to differences in (1) the probabilistic models under which they are derived, and (2) the measures of convergence used. The results of Karp and Beardwood, et al are proved within what Weide calls the "incremental model" -- i.e., the n-th instance of the problem differs only incrementally from the previous one. Our results are proved for the "independent model", in which the n-th problem in the sequence is totally independent of previous ones. Weide has shown that results for the independent model are stronger in the sense that they subsume results for the incremental model, while the reverse does not always hold. Another difficulty with the results of Karp and Beardwood, et al is that they hold only in the limit as the number of points tends to infinity, and hence the results do not speak about moderate (and the usually interesting) values of n. Our results are derived in a framework that is not plagued by this difficulty.

1. Distribution of Points

Our objective is to derive theoretical bounds on the expected lengths of tours constructed by various approximation algorithms for the TSP when the points are distributed randomly in two-dimensional Euclidean space. In this paper, we shall assume that the Cartesian coordinates (x,y) of each point are generated from a normal distribution with mean 0 and variance σ^2, denoted $N(0,\sigma^2)$. This distribution obeys the statistical assumptions made in previous work. It was selected to obtain concrete quantitative results, and because it was quite tractable to analyze. Although we deal here with the normal distribution, the analytic techniques themselves are applicable to any distribution of points that depends only on the origin and decreases monotonically outside a circle of sufficiently large radius.

One of the important statistical techniques that we shall use in our analysis comes from the distribution of order statistics [5,6,8]. Let $x_1,\ldots,x_m$ be a random sample of size m from some probability density function $f(x)$. We can find the distribution functions of the order statistics $y_1,\ldots,y_m$, where the y_i's are the x_i's arranged in order of magnitude so that $y_1 \leq y_2 \leq \ldots \leq y_m$. From the joint distribution of the y_i's, other interesting and useful distributions -- including those of the maximum, the minimum, and the range -- may be derived. Specifically, the probability function g_i of the i-th smallest element of $\{y_j\}$ is given by

$$g_i(y)\, dy = m \binom{m-1}{i-1} [F(y)]^{i-1} [1-F(y)]^{m-i} f(y)\, dy$$

where F gives the cumulative density function of the y_i's. In order to apply this technique to the TSP, it is necessary that (1) the points should be generated independently from the distribution, and (2) the distribution of the length of the edge connecting any two random points should be known. We now derive this distribution.

Lemma 1: The distribution function of the distance between two points selected at random from $N(0,\sigma^2)$ is

$$F(t) = 1 - e^{-t^2/4\sigma^2}$$

and the expected distance between two points is $E(t) = \sqrt{\pi}\sigma$.

Proof: Let (x_1,y_1) and (x_2,y_2) be the coordinates of two randomly selected points. Then, the distance z between them is

$$z = \sqrt{(x_1-x_2)^2 + (y_1-y_2)^2}$$

Cramér [5] proves that if

$$w = \sqrt{\sum_{i=1}^{n} \xi_i^2}$$

where the ξ_i are generated from $N(0,\sigma^2)$, the density of w is given by

$$g(w) = \frac{2w^{n-1} e^{-w^2/2\sigma^2}}{2^{n/2} \sigma^n \Gamma(n/2)}$$

where Γ denotes the gamma function. Since (x_1-x_2) and (y_1-y_2) are differences of normal distributions, they are generated by $N(0,2\sigma^2)$. Substituting n=2, we obtain

$$f(z) = \frac{1}{2\sigma^2} z \ e^{-z^2/4\sigma^2}$$

and hence

$$F(t) = \text{Prob}(z \leq t) = \int_0^t f(z) \ dz$$

$$= \int_0^t \frac{1}{2\sigma^2} z \ e^{-z^2/4\sigma^2} dz = \int_0^t -d(e^{-z^2/4\sigma^2})$$

$$= 1 - e^{-t^2/4\sigma^2}$$

Furthermore, the expected distance between two random points is

$$E(t) = \int_0^\infty t \ f(t) \ dt$$

$$= \int_0^\infty \frac{1}{2\sigma^2} t^2 \ e^{-t^2/4\sigma^2} dt = \sqrt{\pi}\sigma \qquad \underline{q.e.d.}$$

2. Nearest Neighbor Method

The nearest neighbor algorithm for the TSP may be briefly described as follows.

> One of the nodes is arbitrarily selected as the starting point. Among all the nodes not yet visited, the one that is closest to the current node is selected as the next to be visited. After all the nodes have been visited, return to the starting point.

Rosenkrantz, et al [17] have shown that this algorithm always constructs an n-point tour those length is at most $\frac{1}{2} \log_2 n$ of the optimal, and that there exist graphs for which its tour is $\frac{1}{3} \log_2 n$ times the optimal. We now derive the expected path length when the coordinates of the points are selected from $N(0, \sigma^2)$.

Theorem 2: The expected length of the tour for n points constructed by the nearest neighbor method, $T_{NN}(n)$, is bounded from above by

$$T_{NN}(n) \leq 2\sqrt{\pi} \ \sigma \ \sqrt{n-1} + O(\sqrt{\log n})$$

Proof: Suppose we start at an arbitrary node A. The expected length of the first edge in the tour is the minimum of the n-1 edges joining A to all other points (see Figure 1).
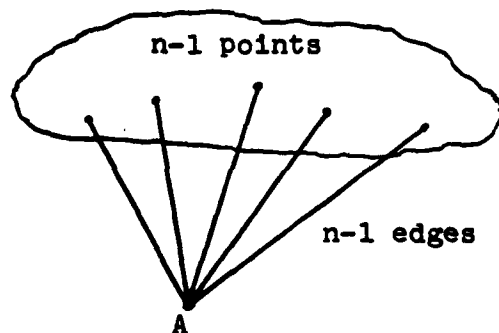
Figure 1. n-1 edges emanate from an arbitrary point A.

The lengths of these edges follow the distribution F given by Lemma 1 and are independent. By order statistics, the distribution of g_1, the length of the shortest of these $n-1$ edges, is given by

$$g_1(t)\,dt = (n-1)\,[1-F(t)]^{n-2}\,f(t)\,dt$$

Thus, the expected length L_1 of the first edge is

$$E(L_1) = \int_0^\infty t\,g_1(t)\,dt$$
$$= \int_0^\infty t(n-1)\,[1-F(t)]^{n-2}\,f(t)\,dt$$
$$= \int_0^\infty t(-d[1-F(t)]^{n-1})$$

Integrating by parts, we get

$$E(L_1) = \int_0^\infty [1-F(t)]^{n-1}\,dt$$
$$= \int_0^\infty e^{-(n-1)t^2/4\sigma^2}\,dt$$
$$= \sqrt{\pi}\sigma/\sqrt{n-1}$$

Similarly, the expected length of the i-th edge L_i added to the tour is the minimum of the $n-i$ edges from the current node to the remaining unvisited points. Hence, by the properties of order statistics and Lemma 1,

$$E(L_i) = \int_0^\infty [1-F(t)]^{n-i}\,dt = \sqrt{\pi}\sigma/\sqrt{n-i}$$

The closing edge of the tour joins the last node added to the starting point. Denoting its length by L_{ce}, we obtain for the total tour length

$$T_{NN}(n) = \sum_{i=1}^{n-1} E(L_i) + E(L_{ce})$$
$$= \sum_{i=1}^{n-1} \frac{\sqrt{\pi}\sigma}{\sqrt{n-i}} + E(L_{ce})$$
$$= \sum_{i=1}^{n-1} \frac{\sqrt{\pi}\sigma}{\sqrt{i}} + E(L_{ce})$$

$$\leq \int_{0+}^{n-1} \frac{\sqrt{\pi}\sigma}{\sqrt{x}} \, dx + E(L_{ce})$$

$$= 2\sqrt{\pi}\sigma \sqrt{n-1} + E(L_{ce})$$

Observe that $E(L_{ce})$ is at most equal to the expected value of the longest edge joining the starting point A to any of the other n-1 points (see Figure 1). By order statistics, the distribution g_{n-1} of the longest of n-1 edges is given by

$$g_{n-1}(t) \, dt = (n-1) \, F(t)^{n-1} \, f(t) \, dt$$

and its expected value by

$$E(g_{n-1}) = \int_0^\infty t \, g_{n-1}(t) \, dt \leq E(L_{ce}).$$

Gumbel [8, Sect. 6.3.8] shows that this quantity asymptotically becomes

$$E(g_{n-1}) = 2\sigma \sqrt{\ln(n-1)} + \gamma\sigma/\sqrt{\ln(n-1)}$$

where γ is Euler's constant, $\gamma = .577^+$.   q.e.d.

To find the expected length of L_{ce}, some authors [18] have made the simplifying assumption that all points except the one closest to the starting point are equally likely to be selected as the last point added to the tour. To our knowledge, the validity of this assumption has never been formally proven, and there exists experimental evidence to the contrary [9]. If the assumption holds, the expected length of the closing edge can be estimated by the average distance from the starting point to all points other than its nearest neighbor. As the number of points increases, this quantity approaches the expected distance between any two points, which is $\sqrt{\pi}\sigma$ by Lemma 1. This distance, although independent of the number of points n, does not significantly alter the length of tour bound we have derived.

3. Arbitrary Insert Method

The arbitrary insert algorithm for the TSP operates as follows.

1. Choose any node A as the starting point, and another arbitrary point B as the second node to be visited. Construct the tour going from A to B and back.

2. Randomly choose one of the nodes P not yet visited as the next point to be inserted. Find the node Q already in the tour which is closest to P. From the two nodes adjacent to Q in the tour, select the one R such that

$$d_{PQ} + d_{PR} - d_{QR}$$

    is minimal, where d_{ij} denotes the distance between points i and j. Add edges $\overline{PQ}$ and $\overline{PR}$ to, and delete $\overline{QR}$ from, the tour. Repeat this step until all points are included.

Rosenkrantz, et al [17] have shown that this algorithm always produces a tour whose length is within a factor of $\log_2 n$ of the optimal, but it is an open question whether this logarithmic growth can actually be realized. Using a complicated proof, Morozinskii [14] has shown that this algorithm constructs a tour whose expected length is $O(\sqrt{n})$ and is within a factor of 4 from the optimal for a general class of probability distributions which includes the normal.

Theorem 3a: The expected length of the tour for n points constructed by the arbitrary insert method, $T_{AI}(n)$, is bounded from above by

$$T_{AI}(n) < 4\sqrt{\pi} \ \sigma \ \sqrt{n-1}$$

Proof: Suppose we have i points in the tour, where $i \geq 2$. The (i+1)-st point P is chosen at random from the remaining set of n-1 points (see Figure 2).
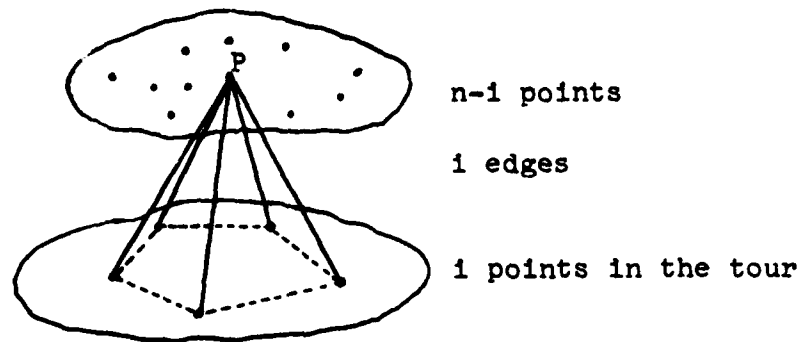
Figure 2. P is the i-th point to be inserted in the tour.

The expected value of the minimum distance D_i from P to the i points in the tour is

$$E(D_i) = \int_0^\infty [1-F(t)]^i \, dt = \sqrt{\pi} \, \sigma / \sqrt{i}$$

We must now compute the cost of adding point P to the tour. Consider the situation illustrated in Figure 3.
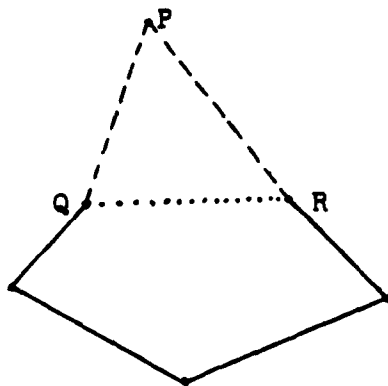
By the Triangle Inequality,

$$d_{PQ} + d_{QR} > d_{PR}$$
$$2d_{PQ} + d_{QR} > d_{PQ} + d_{PR}$$
$$2d_{PQ} > d_{PQ} + d_{PR} - d_{QR}$$
$$= \text{cost of inserting P}$$

Figure 3. To insert P, delete edge $\overline{QR}$ and add edges $\overline{PQ}$ and $\overline{PR}$.

Hence, the expected cost of inserting P is at most twice $E(D_i)$, and the total tour length can be bounded from above by

$$T_{AI}(n) < \sum_{i=2}^{n-1} 2E(D_i) + 2E(L_{ie})$$

where $E(L_{ie}) = \sqrt{\pi}\sigma$ is the expected length of the random starting edge in step 1 of the algorithm (see Lemma 1). Therefore,

$$T_{AI}(n) < 2 \sum_{i=2}^{n-1} \frac{\sqrt{\pi}\sigma}{\sqrt{i}} + 2\sqrt{\pi}\sigma$$

$$= 2 \sum_{i=1}^{n-1} \frac{\sqrt{\pi}\sigma}{\sqrt{i}}$$

$$\leq 4 \sqrt{\pi}\, \sigma \, \sqrt{n-1} \qquad\qquad \underline{q.e.d.}$$

The bound of Theorem 3a is quite conservative since it uses the Triangle Inequality as the basis for estimating the cost of inserting the new point P. The Triangle Inequality actually describes the worst case cost of inserting P. Let us examine each of the three lengths involved in the computation of this cost, $d_{PQ} + d_{PR} - d_{QR}$, more closely.

By applying order statistics, we determined the expected value of d_{PQ} to be $E(D_i)$, the expected minimum distance from P to the i points already in the tour. Since point Q appears at some random spot in the i-point tour being modified, we would expect d_{QR} to be an average length edge in this partial tour. Thus, if we let $E(L_i)$ denote the expected length of the i-point tour during the construction, then the expected value of d_{QR} is $E(L_i)/i$.

Finally, we consider d_{PR}. By the operation of the algorithm, P is known to be closer to Q than R and so $d_{PR} > d_{PQ}$. By the Triangle Inequality, $d_{PR} < d_{PQ} + d_{QR}$. Just where d_{PR} falls in this range is unknown, but a reasonable assumption might be that the distance d_{PR} is distributed uniformly

between the two limits. This would imply that the expected value of d_{PR} is $d_{PQ} + \frac{1}{2} d_{QR}$, and that the expected cost of inserting P is

$$E(d_{PQ} + d_{PR} - d_{QR}) = E(D_i) + \left[E(D_i) + \frac{E(L_i)}{2i} \right] - \frac{E(L_i)}{i} = 2E(D_i) - \frac{E(L_i)}{2i}$$

Since we have no formal basis for the validity of this assumption, we will refer to it as the "reasonable insertion hypothesis".

Theorem 3b: Under the reasonable insertion hypothesis, $T_{AI}(n)$ is bounded from above by

$$T_{AI}(n) \leq 2 \sqrt{\pi} \, \sigma \, \sqrt{n-1} + O(1)$$

Proof: From the above discussion, the expected length $E(L_{i+1})$ of the (i+1)-point tour is equal to the expected length of the i-point tour plus the expected cost of inserting the (i+1)-st point P. A recurrence relation describing this fact is

$$E(L_{i+1}) = E(L_i) + E(d_{PQ} + d_{PR} - d_{QR})$$

$$= \frac{2i-1}{2i} E(L_i) + 2E(D_i)$$

We are interested in solving this recurrence for $T_{AI}(n) = E(L_n)$. This relation can be solved using the method of summing factors, described in Lueker [12]. To do so, we need an appropriate boundary condition which, from Step 1 of the algorithm, is $E(L_2) = 2\sqrt{\pi} \, \sigma$.

A general recurrence relation of the form

$$x_{i+1} = f_i x_i + g_i$$

has solution

$$x_n = \sum_{i=a}^{n-2} \left(\prod_{j=i+1}^{n-1} f_j \right) g_i + g_{n-1} + \left(\prod_{i=a}^{n-1} f_i \right) x_a$$

where x_a denotes the value of x at the boundary condition. Hence, the solution to our recurrence is

$$E(L_n) = \sum_{i=2}^{n-2} \left(\prod_{j=i+1}^{n-1} \frac{2j-1}{2j} \right) 2E(D_i) + 2E(D_{n-1}) + \left(\prod_{i=2}^{n-1} \frac{2i-1}{2i} \right) E(L_2)$$

$$= 2\sqrt{\pi}\sigma \sum_{i=2}^{n-2} \left(\prod_{j=i+1}^{n-1} \frac{2j-1}{2j} \right) \frac{1}{\sqrt{i}} + \frac{2\sqrt{\pi}\sigma}{\sqrt{n-1}} + \left(\prod_{i=2}^{n-1} \frac{2i-1}{2i} \right) \sqrt{\pi}\sigma$$

Applying the inequality

$$\prod_{i=k}^{m} \frac{2i-1}{2i} \leq \sqrt{\frac{k}{m}}$$

we obtain

$$E(L_n) \leq \frac{2\sqrt{\pi}\sigma}{\sqrt{n-1}} \sum_{i=2}^{n-2} \sqrt{\frac{i+1}{i}} + \frac{(2+\sqrt{2})\sqrt{\pi}\sigma}{\sqrt{n-1}}$$

The desired result follows from the following estimate of the sum.

$$\sum_{i=2}^{n-2} \sqrt{\frac{i+1}{i}} \leq \int_{1}^{n-2} \sqrt{\frac{x+1}{x}} \, dx$$

$$< \int_{1}^{n-2} \left[1 + \frac{1}{\sqrt{x}} \right] dx$$

$$= (n-1) + 2\sqrt{n-2} - 4 \qquad \underline{\text{q.e.d.}}$$

4. Nearest and Cheapest Insert Methods

The nearest insert algorithm operates similiarly to the arbitrary insert method except that the next point P to be inserted into the tour, instead of being chosen arbitrarily, is selected to be the point not yet in the tour which is closest to any node already in the tour. (The second point selected is the starting point's nearest neighbor.) Rosenkrantz, et al [17] have shown that this method always produces a tour whose length is at most twice the optimal, and that there exist graphs for which it performs virtually this badly.

<u>Theorem 4a</u>: The expected length of the tour for n points constructed by the nearest insert method, $T_{NI}(n)$, is bounded from above by

$$T_{NI}(n) < 4(2 - \sqrt{2}) \sqrt{\pi} \ \sigma \sqrt{n-1}$$

<u>Proof</u>: Consider the case when we have i points in the tour. To get the (i+1)-st point, we take the minimum of n-i edges connecting each of the i nodes in the tour to all of the remaining n-i nodes (see Figure 4).

remaining n-i points

i sets of n-i edges
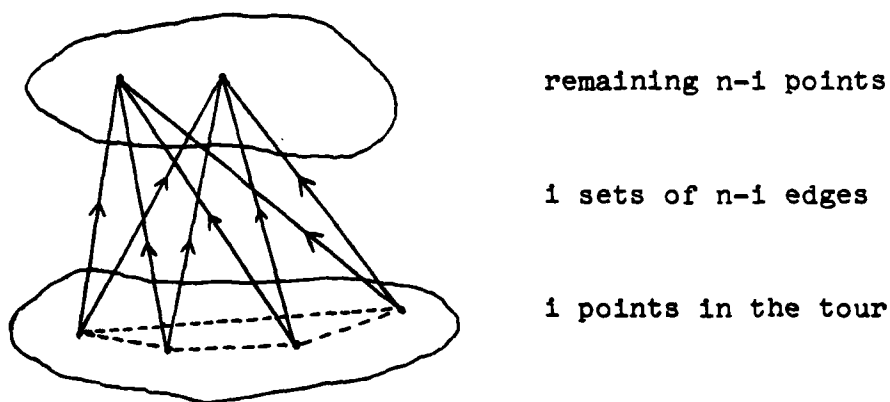
i points in the tour

Figure 4. i sets of n-i edges join the i points in the tour with the n-i points not yet inserted.

Observe that the expected minimum in i sets of n-i edges is less than or equal to the expected minimum of the n-i edges emanating from one of the i points already in the tour. Hence, by Lemma 1 and order statistics, the expected distance from the point P to be inserted to any of the i points already in the tour is upper bounded by $\sqrt{\pi}\sigma/\sqrt{n-i}$

$$E(\text{min. of i sets of n-i edges}) \leq E(\text{min. of n-i random edges})$$
$$= \sqrt{\pi}\sigma/\sqrt{n-i}$$

Let L' denote the length of the tour constructed through the first $\left\lceil \frac{n}{2} \right\rceil$ points, and L" be the addition to the length by the remaining $\left\lfloor \frac{n}{2} \right\rfloor$ points. Then,

$$T_{NI}(n) = E(L') + E(L")$$

Given the expected length of the i-th edge added to the tour, our analysis proceeds as in the proof of Theorem 3a. We find

$$E(L') < \sum_{i=1}^{\left\lceil \frac{n}{2} \right\rceil - 1} 2(D_i) \leq 2\sqrt{\pi}\sigma \sum_{i=1}^{\left\lceil \frac{n}{2} \right\rceil - 1} \frac{1}{\sqrt{n-i}} = 2\sqrt{\pi}\sigma \sum_{i=\left\lceil \frac{n}{2} \right\rceil + 1}^{n-1} \frac{1}{\sqrt{i}}$$

Now let the tour contain more than $\left\lceil \frac{n}{2} \right\rceil$ points. Then finding the minimum of i sets of n-i edges drawn from the nodes in the tour to the remaining nodes is equivalent to finding the minimum of n-i sets of i edges drawn from the nodes not in the tour to the nodes already in the tour (see Figure 4). As before,

$$E(\text{min. of n-i sets of i edges}) \leq E(\text{min. of i random edges})$$
$$= \sqrt{\pi}\sigma/\sqrt{i}$$

and

$$E(L") < \sum_{i=\left\lceil \frac{n}{2} \right\rceil}^{n-1} 2E(D_i) \leq 2\sqrt{\pi}\sigma \sum_{i=\left\lceil \frac{n}{2} \right\rceil}^{n-1} \frac{1}{\sqrt{i}}$$

Hence,

$$T_{NI}(n) = E(L') + E(L'')$$

$$< 2\sqrt{\pi}\sigma \left[\sum_{i=\lfloor\frac{n}{2}\rfloor+1}^{n-1} \frac{1}{\sqrt{i}} + \sum_{i=\lceil\frac{n}{2}\rceil}^{n-1} \frac{1}{\sqrt{i}} \right]$$

$$\leq 4\sqrt{\pi}\sigma \sum_{i=\lceil\frac{n}{2}\rceil}^{n-1} \frac{1}{\sqrt{i}} \leq 4\sqrt{\pi}\sigma \left[2\sqrt{n-1} - 2\sqrt{\lceil\frac{n}{2}\rceil-1} \right]$$

$$\approx 4(2-\sqrt{2})\sqrt{\pi}\sigma\sqrt{n-1} \qquad\qquad \underline{q.e.d.}$$

Again, the bound of Theorem 3a is very conservative since it is based upon using the Triangle Inequality for bounding from above the cost of inserting each new point. We now wish to investigate the improvement in tour length under the reasonable insertion hypothesis introduced in the previous section.

__Theorem 4b:__   Under the reasonable insertion hypothesis, $T_{NI}(n)$ is bounded from above by

$$T_{NI}(n) \leq \frac{\pi}{2} \sqrt{\pi}\ \sigma\ \sqrt{n-1} + O(1)$$

__Proof:__   As in the proof of Theorem 3b, the reasonable insertion hypothesis gives rise to the recurrence relation

$$E(L_{i+1}) = \frac{2i-1}{2i} E(L_i) + 2E(D_i)$$

This time, the recurrence must be solved twice.

The expected length of the tour through the first $\lceil\frac{n}{2}\rceil$ points is

$$E(L') = \sum_{i=2}^{\lceil\frac{n}{2}\rceil-2} \left(\prod_{j=i+1}^{\lceil\frac{n}{2}\rceil-1} \frac{2j-1}{2j} \right) 2E(D_i) + 2E(D_{\lceil\frac{n}{2}\rceil-1}) + \left(\prod_{i=2}^{\lceil\frac{n}{2}\rceil-1} \frac{2i-1}{2i} \right) E(L_2)$$

where $E(D_i)$ and the boundary condition $E(L_2)$ are given by

$$E(D_i) \leq \frac{\sqrt{\pi}\ \sigma}{\sqrt{n-i}} \qquad \text{and} \qquad E(L_2) = \frac{2\sqrt{\pi}\sigma}{\sqrt{n-1}}$$

6-18

Using the inequality

$$\prod_{i=k}^{m} \frac{2i-1}{2i} \leq \sqrt{\frac{k}{m}}$$

We find that

$$E(L') \leq \frac{2\sqrt{2}\ \sqrt{\pi\sigma}}{\sqrt{n-2}} \sum_{i=2}^{\lceil \frac{n}{2}\rceil -2} \sqrt{\frac{i+1}{n-1}} \ + \ \frac{2\sqrt{2}\ \sqrt{\pi\sigma}}{\sqrt{n+1}} \ + \ \frac{4\sqrt{\pi\sigma}}{\sqrt{(n-1)(n-2)}}$$

The first of the three terms will dominate, and a good estimate of this term may be obtained as follows.

$$\sum_{i=2}^{\lceil \frac{n}{2}\rceil -2} \sqrt{\frac{i+1}{n-1}} \ \leq \ \int_{2}^{\frac{n}{2}-1} \sqrt{\frac{x+1}{n-x}} \ \ dx$$

$$= (n+1)\left[\tan^{-1}\sqrt{\frac{n}{n+2}} + \frac{\sqrt{3(n-2)}}{n+1} - \tan^{-1}\sqrt{\frac{3}{n-2}} - \frac{\sqrt{n(n+2)}}{2(n+1)} \right]$$

$$\leq (n+1)\left[\frac{\pi}{4} - 0^{+} - \frac{1}{2}^{-} \right] \ \approx \ \frac{\pi-2}{4}(n+1)$$

since $\frac{\sqrt{3(n-2)}}{n+1}$ and $\tan^{-1}\sqrt{\frac{3}{n-2}}$ both approach $\sqrt{\frac{3}{n}}$ rapidly. Hence,

$$E(L') \leq \frac{\pi-2}{2}\sqrt{\pi\sigma}\frac{n+1}{\sqrt{n-2}} + 0(\frac{1}{\sqrt{n}})$$

For the second half of the tour, we must again solve the recurrence.

$$T_{NN}(n) = E(L_n) = \sum_{i=\lceil\frac{n}{2}\rceil}^{n-2} (\prod_{j=i+1}^{n-1} \frac{2j-1}{2j}) 2E(D_i) + 2E(D_{n-1}) + (\prod_{i=\lceil\frac{m}{2}\rceil}^{n-1} \frac{2i-1}{2i}) E(L_{\lceil\frac{n}{2}\rceil})$$

where $E(L_{\lceil\frac{m}{2}\rceil}) = E(L')$, which we just estimated, serves as the boundary condition. Applying our usual inequality for $\prod (2j-1)/2j$ and remembering that $E(D_i) \leq \sqrt{\pi\sigma}/\sqrt{i}$ in the second half of the tour, we obtain

$$E(L_n) \leq \frac{2\sqrt{\pi}\sigma}{\sqrt{n-1}} \sum_{i=\lceil \frac{n}{2} \rceil}^{n-2} \sqrt{\frac{i+1}{i}} + \frac{2\sqrt{\pi}\sigma}{\sqrt{n-1}} + \frac{\pi-2}{2}\sqrt{\pi}\sigma \sqrt{n+1}^{+}$$

Next, we bound the summation in the first term from above by an integral.

$$\sum_{i=\lceil \frac{n}{2} \rceil}^{n-2} \sqrt{\frac{i+1}{i}} \leq \int_{\frac{n}{2}-1}^{n-2} \sqrt{\frac{x+1}{x}}\, dx$$

$$< \int_{\frac{n}{2}-1}^{n-2} \left[1 + \frac{1}{\sqrt{x}}\right] dx$$

$$= \frac{1}{2}(n-1) + (2-\sqrt{2})\sqrt{n-2}$$

The theorem immediately follows by adding together the terms growing as $\sqrt{n}$ in the sum for $E(L_n)$.   q.e.d.

The cheapest insert algorithm operates somewhat like the nearest insert method. Again, the next point P to be inserted is chosen to be the node not yet in the tour which is closest to any node already in the tour. Point P is inserted by finding the edge $\overline{QR}$ already in the tour such that $d_{PQ} + d_{PR} - d_{QR}$ is minimized, and deleting this edge while adding $\overline{PQ}$ and $\overline{PR}$. Hence, this algorithm inserts P at the least costly place. Rosenkrantz, et al [17] have shown that the worst case behavior of this method is the same as that of the nearest insert algorithm. Since the tour constructed by the cheapest insert method cannot be longer than that constructed by the nearest insert method, the upper bounds of Theorems 4a and 4b also apply to cheapest insert.

5. <u>MST – Based Method</u>

The minimal spanning tree of a set of n points consists of the n-1 edges connecting all the points in such a way that the total length of the edges is minimized. Lewis and Papadimitriou [13] and others have shown how to convert the MST into a TSP tour whose length is at most twice that of the optimal tour. Christofides [3] has further refined this method to produce a TSP tour whose length is at most $1\frac{1}{2}$ times the optimum, to be described in the next section.

We now proceed to explore the relationship between the length of the optimal TSP tour, denoted $|OPT|$, and the length of the MST, denoted $|MST|$. Since the optimal TSP tour can be converted into a spanning tree (not necessarily the minimal one) by removing one edge, we have

$$|MST| < |OPT|$$

Furthermore,

$$|OPT| \leq 2 \; |MST|$$

and this occurs when all the points are collinear. The validity of this latter claim will be clarified in what follows.

The tour building technique described in Lewis and Papadimitriou is based on the observation that the MST can be converted into a tour visiting all the points by traversing each edge twice and returning to the origin, as illustrated in Figure 5a. This twice-around-the-tree tour is then converted into a legitimate TSP tour by shortcutting any previously visited points and proceeding directly to the next unvisited point, as

shown in Figure 5b. It is easy to see that the length of the TSP tour produced is. bounded above by twice the length of the MST.
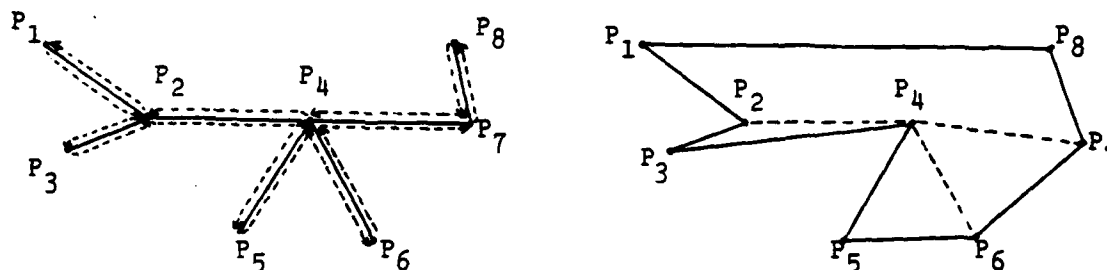
a) MST and tour with length = $2|MST|$.   b) TSP tour based on the MST.

Figure 5.

We now proceed to bound the expected length of the MST from above using Prim's algorithm [16]. This method for constructing the exact MST may be briefly described as follows.

> Arbitrarily choose any node as the starting point, and include it in the tree. From among all the nodes not yet in the tree, select the one that is closest to any tree node, and add this node and the corresponding edge to the tree. Continue this procedure until all nodes are included.

Theorem 5: The expected length of the MST for n points, $L_{MST}(n)$, is bounded from above by

$$L_{MST}(n) \leq 2(2-\sqrt{2}) \sqrt{\pi} \, \sigma \, \sqrt{n-1}.$$

Proof: The situation when the i-th edge is added is identical to that for the nearest insert algorithm. To get the length L_i of the i-th edge added, we take the minimum of the n-i edges connecting each of the i nodes already in the tree to the remaining n-i points (see Figure 6). As in the proof of
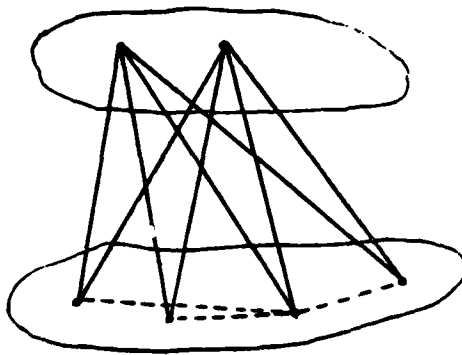
remaining n-i points

i sets of n-i edges
or
n-i sets of i edges

i points in MST

Figure 6. i(n-i) edges join the i points in the
MST with the n-i points still to be added.

Theorem 4a, for $1 \leq i \leq \left\lfloor \frac{n}{2} \right\rfloor$, we have

$$E(L_i) = E(\text{min. of } i \text{ sets of } n-i \text{ edges})$$

$$\leq E(\text{min. of } n-i \text{ random edges})$$

$$= \frac{\sqrt{\pi}\, \sigma}{\sqrt{n-i}}$$

We may obtain a better bound for the remaining nodes added by considering the directions of the edges to be reversed. Again, as in Theorem 4a,

$$E(L_i) = E(\text{min. of } n-i \text{ sets of } i \text{ edges})$$

$$\leq E(\text{min. of } i \text{ random edges})$$

$$= \frac{\sqrt{\pi}\sigma}{\sqrt{i}}$$

for $\left\lfloor \frac{n}{2} \right\rfloor + 1 \leq i \leq n-1$. Hence,

$$L_{MST}(n) = \sum_{i=1}^{n-1} E(L_i) \leq \sum_{i=1}^{\left\lfloor \frac{n}{2} \right\rfloor} \frac{\sqrt{\pi}\sigma}{\sqrt{n-i}} + \sum_{i=\left\lfloor \frac{n}{2} \right\rfloor +1}^{n-1} \frac{\sqrt{\pi}\sigma}{\sqrt{i}}$$

$$\leq 2 \sum_{i=\left\lfloor \frac{n}{2} \right\rfloor}^{n-1} \frac{\sqrt{\pi}\sigma}{\sqrt{i}} \leq 2(2-\sqrt{2})\sqrt{\pi}\sigma\sqrt{n-1}$$

q.e.d.

We note that the expected length of the MST can be bounded from below by

$$L_{MST}(n) \geq \sqrt{\pi} \, \sigma \, \sqrt{n-1}$$

using the technique to be described in the proof of Theorem 7a. This result follows from the observation that the MST contains n-1 edges, each of whose expected length is at least as great as the expected distance from a point to its nearest neighbor. Hence, the bound of Theorem 5 is quite tight since $2(2-\sqrt{2}) \approx$ 1.17.

The MST-based algorithm described above produces a TSP tour whose expected length, $T_{MSTB}(n)$, can be bounded from above by

$$T_{MSTB}(n) \leq 2L_{MST}(n) \leq 4(2-\sqrt{2})\sqrt{\pi} \, \sigma \, \sqrt{n-1}$$

In fact, we should expect the length of the TSP tour constructed to be significantly less due to the shortcutting procedure. Unfortunately, the geometric and statistical techniques necessary to obtain a good estimate of this improvement have not yet been identified.

6. Christofides' Method

The MST-based algorithm described in the previous section can be regarded as consisting of four basic steps.

1. Construct the minimal spanning tree.

2. Convert the MST into a multigraph consisting of two edges for each edge in the MST (e.g., the dotted edges in Figure 5a).

3. Construct an Eulerian tour of the multigraph produced in Step 2. An Eulerian tour traverses each of the edges in a graph exactly once, returning to the origin.

4. Convert the Eulerian tour of Step 3 into a legitimate TSP tour by shortcutting edges to previously visited points.

It is well-known that a connected multigraph contains an Eulerian tour if and only if the degree of each of its vertices is even. Such a graph is called an Eulerian multigraph. Clearly, the procedure of Step 2 ensures that this condition will be met.

Christofides [3] has discovered another way of converting the original MST into a TSP tour yielding an even better performance guarantee on the length of the path constructed. His method is the same as above except that Step 2 is changed to the following.

2'. Construct the Eulerian multigraph consisting of all the MST edges plus the edges in the minimal weight matching on the vertices of odd degree in the MST.

A matching on a set of 2m vertices V is a partition of V into m disjoint 2-element sets. Associated with the matching is the set of edges $\overline{PQ}$ for each 2-element set {P,Q}. The minimal weight matching on V is the one in which the total sum of its associated edge lengths is smallest.

6-25

The operation of Step 2' is illustrated in Figure 7. Since
the matching adds one new edge incident with each vertex of odd
degree in the MST, all of the vertices in the multigraph are of
even degree and the existence of an Eulerian tour is guaranteed.
Furthermore, the perfect matching must exist since the number of
vertices of odd degree in the MST is even, according to another
well-known result from graph theory.
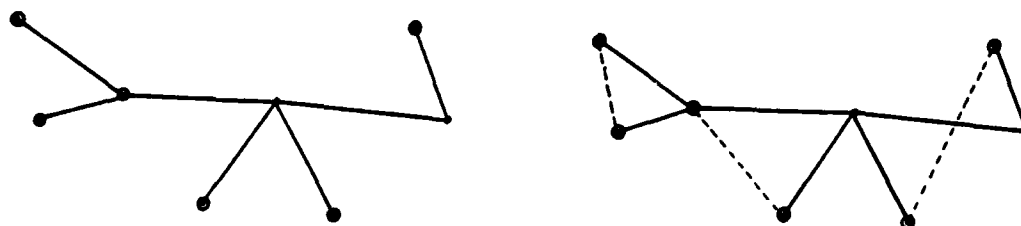
a) MST, odd degree vertices circled.   b) Minimal odd vertex matching added.

Figure 7.

We now explore the relationship between the lengths of
Christofides' tour (denoted $|CM|$), the minimal odd matching
(denoted $|MOM|$), the minimal spanning tree ($|MST|$), and the
optimal TSP tour ($|OPT|$). Since the length of the Euclidean
tour constructed in Step 3 equals $|MST| + |MOM|$, we have

$$|CM| \leq |MST| + |MOM|$$

We observed in Section 5 that $|MST| < |OPT|$. It also turns out
that $|MOM| \leq \frac{1}{2} |OPT|$. This occurs because the optimal TSP can be
converted into a tour T through the vertices of odd degree in the
MST by shortcutting any edges passing through the even vertices.

6-26

Clearly, $|T| \leq |OPT|$. Furthermore, T contains two matchings on the odd vertices, formed by taking every other edge, and the length of the shorter of these cannot exceed $\frac{1}{2}|OPT|$. (See Figure 8.) We conclude that

$$|CM| < \frac{3}{2} |OPT|$$

Cornuejols and Nemhauser [4] have further shown that this bound is tight by exhibiting instances of the problem for which the algorithm performs this badly.
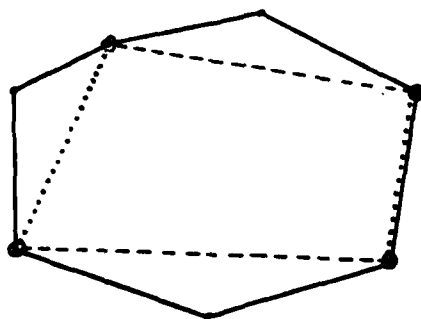
Figure 8. Optimal TSP tour with odd degree vertices in MST circled. Shortcut tour through these vertices contains two matchings.

The expected length of the tour produced by Christofides' algorithm can be bounded above by the sum of the expected lengths of the MST and MOM. Since we already considered the length of the MST in Section 5, we turn our attention to the problem of determining the expected length of the matching. The number of points participating in the matching varies from one problem instance to another. All n points participate in the worst case, although we would anticipate this situation to arise

only rarely. Unfortunately, we do not know of any techniques
for determining the expected number of points in the matching,
and this remains an interesting open question.

Another difficulty arises in estimating the expected length
of the minimal matching. All of the algorithms studied so far
adhere to the "greedy" design paradigm. That is, they make a
series of decisions on how to proceed based on finding the
smallest edge with a certain property. Order statistics lends
itself nicely to examining the expected behavior of such
procedures. However, we know of no such greedy algorithm for
the optimal matching. Instead, we shall bound its expected
length from above by analyzing a greedy matching heuristic which
does not, in general, produce the best match. This method
operates as follows.

> Randomly select a point and pair it with its nearest
> neighbor. From the remaining n-2 points, randomly select
> one and pair it with the nearest unmatched point. Repeat
> the procedure n/2 times, when all points will be paired.

Theorem 6: The expected length of the matching for n points
constructed by the greedy matching heuristic, $L_{GM}(n)$, is
bounded from above by

$$L_{GM}(n) \leq \sqrt{\pi} \, \sigma \sqrt{n-1}$$

Proof: At the i-th iteration of the algorithm, the point P
selected randomly is paired with one of the n-2i+1 remaining
points. From order statistics and Lemma 1, the shortest of the
n-2i+1 edges joining P to the unpaired points has expected length

$$E(L_i) = \frac{\sqrt{\pi} \, \sigma}{\sqrt{n-2i+1}}$$

6-28

Hence, the expected length of the entire matching is given by

$$L_{GM}(n) = \sum_{i=1}^{n/2} E(L_i) = \sum_{i=1}^{n/2} \frac{\sqrt{\pi}\ \sigma}{\sqrt{n-2i+1}} = \sqrt{\pi}\ \sigma \left[\sum_{i=1}^{\frac{n}{2}-1} \frac{1}{\sqrt{n-2i+1}} + 1 \right]$$

$$\leq \sqrt{\pi}\ \sigma \left[\int_1^{n/2} \frac{dx}{\sqrt{n-2i+1}} + 1 \right]$$

$$= \sqrt{\pi}\ \sigma\ \sqrt{n-1} \qquad\qquad \underline{q.e.d.}$$

An obvious improvement on the greedy method is to pick the shortest edge among any of the remaining points at each step. Because of statistical dependencies between the edges, we cannot say anything significant about this technique. However, we can content ourselves with the following interesting fact. Although the length of the matching produced by our simple greedy heuristic can be quite bad, its expected value is within a factor of two of the expected length of the minimal matching. To see this, observe that the expected length of the minimal matching on n points, $L_{MM}(n)$, can be bounded below by

$$L_{MM}(n) \geq \frac{n}{2}\ \frac{\sqrt{\pi}\ \sigma}{\sqrt{n-1}} \geq \frac{1}{2}\ \sqrt{\pi}\ \sigma\sqrt{n}$$

since the matching contains $\frac{n}{2}$ edges, each of whose expected length is as great as the expected distance from a point to its nearest neighbor. A general discussion of such lower bounding techniques for the TSP follows in Section 7.

Suppose cn points participate in the matching, where the fraction c is such that $0 \leq c \leq 1$. Then, as a corollary to Theorem 6, the expected length of the TSP tour constructed by

Christofides' algorithm, $T_{CM}(n)$, can be bounded above by

$$T_{CM}(n) \leq L_{MST}(n) + L_{GM}(cn)$$

$$\leq 2(2-\sqrt{2})\sqrt{\pi}\ \sigma\sqrt{n-1} + \sqrt{c}\ \sqrt{\pi}\ \sigma\sqrt{n-\frac{1}{c}}$$

In the worst case c=1, yielding a bound of

$$T_{CM}(n) \leq (5-2\sqrt{2})\sqrt{\pi}\ \sigma\sqrt{n-1}$$

Under the reasonable assumption that half of the points are of odd degree in the MST, $c=\frac{1}{2}$ and

$$T_{CM}(n) \leq (4-\frac{3}{2}\sqrt{2})\ \sqrt{\pi}\ \sigma\ \sqrt{n-1}$$

As in Section 5, a better estimate of the savings resulting from the shortcutting procedure would enable us to sharpen these bounds.

7. Lower Bound on Optimal Tour Length

Theorem 7a: The expected length of the optimal tour through n points, $T_{OPT}(n)$, is bounded below by

$$T_{OPT}(n) \geq \sqrt{\pi} \; \sigma \; \sqrt{n}$$

Proof: Consider an arbitrary node in the graph. The expected distance to its nearest neighbor is given by the expected length of the minimum of the n-1 edges connecting the node to all the other nodes in the graph. Using order statistics, we have already seen

$$E(\text{distance to nearest neighbor}) = \frac{\sqrt{\pi} \; \sigma}{\sqrt{n-1}}$$

Since the expected length of each of the n edges in the optimal tour is at least the expected distance from a vertex to its nearest neighbor, we have

$$T_{OPT}(n) \geq n \frac{\sqrt{\pi} \; \sigma}{\sqrt{n-1}} \geq \sqrt{\pi} \; \sigma \; \sqrt{n} \qquad\qquad \underline{\text{q.e.d.}}$$

A better lower bound can be derived by noting that exactly two edges are incident with each point in any tour. In the proof of Theorem 7a, we observed only that the expected length of the shorter of these edges is at least as great as the expected distance from a point to its nearest neighbor. But the longer of the two edges emanating from a point has expected length at least equal to the expected distance from a point to its second nearest neighbor. Using this observation, we now derive a better lower bound.

Theorem 7b: The expected length of the optimal tour through n points, $T_{OPT}(n)$, is bounded below by

$$T_{OPT}(n) \geq \frac{5}{4} \sqrt{\pi} \; \sigma \; \sqrt{n}$$

Proof: Let D_1 and D_2 denote the distance from a point P to its nearest and second nearest neighbors, respectively. Attributing half of the length of any tour edge to each of its endpoints, we have

$$T_{OPT}(n) \geq n \left[\tfrac{1}{2} E(D_1) + \tfrac{1}{2} E(D_2)\right]$$

As in Theorem 7a, $E(D_1) = \sqrt{\pi}\, \sigma / \sqrt{n-1}$. The distribution g_2 of the second shortest of the $n-1$ edges incident with a point P is given by order statistics to be

$$g_2(t)dt = (n-1)(n-2)F(t)[1-F(t)]^{n-3} f(t)dt$$

and its expected value is

$$E(D_2) = \int_0^\infty t\, g_2(t)dt$$

$$= \int_0^\infty t(n-1)(n-2)\left[1-e^{-t^2/4\sigma^2}\right]\left[e^{-t^2/4\sigma^2}\right]^{n-3} \frac{1}{2\sigma^2}\, te^{-t^2/4\sigma^2}dt$$

$$= \frac{(n-1)(n-2)}{2\sigma^2}\left[\int_0^\infty t^2 e^{-(n-2)t^2/4\sigma^2}dt - \int_0^\infty t^2 e^{-(n-1)t^2/4\sigma^2}dt\right]$$

$$= \frac{(n-1)(n-2)}{2\sigma^2}\frac{\sqrt{\pi}}{4}\left[\left(\frac{4\sigma^2}{n-2}\right)^{3/2} - \left(\frac{4\sigma^2}{n-1}\right)^{3/2}\right]$$

$$= \sqrt{\pi}\,\sigma\left[\frac{n-1}{\sqrt{n-2}} - \frac{n-2}{\sqrt{n-1}}\right]$$

$$= \sqrt{\pi}\,\sigma\left[\frac{1}{\sqrt{n-2}} + \frac{1}{\sqrt{n-1}} - \frac{1}{\sqrt{n+2}} + \frac{1}{\sqrt{n+1}}\right]$$

$$\geq \frac{3}{2}\frac{\sqrt{\pi}\,\sigma}{\sqrt{n-1}}$$

Hence,

$$T_{OPT}(n) \geq \tfrac{1}{2} n\left[E(D_1) + E(D_2)\right]$$

$$\geq \tfrac{5}{4}\sqrt{\pi}\,\sigma\,\sqrt{n} \qquad\qquad \underline{q.e.d.}$$

This states that the optimal tour, no matter how it is constructed, will have an expected length of at least $\frac{5}{4}\sqrt{\pi}\,\sigma\sqrt{n}$. This is significant in that all of the algorithms discussed above produce a tour whose length is within a small constant factor of this lower bound. This factor ranges from a low of 25.7% for the nearest insert algorithm under the reasonable insertion hypothesis to a high of 87.5% for the MST-based method. As mentioned in Section 4, the cheapest insert method performs at least as well as nearest insert.

Using different techniques, Morozenskii [14] has shown that the asymptotic expected length of an optimal tour is proportional to $\sqrt{n}$ for any probability density which depends solely on the distance from the origin and is monotonic outside some circle of sufficiently large radius, and not merely for a normal distribution. Morozenskii's derivation is based upon the expected distance from a point to its nearest neighbor, rather than both this distance and that of a point's second nearest neighbor. A related result by Beardwood, et al [1] states that the length of the shortest closed path through n points in a bounded plane region of area A is "almost always" proportional to $\sqrt{nA}$ for sufficiently large n.

8. Summary and Conclusions

The famous traveling salesman problem of operations research is NP-complete, even when the points are restricted to the Euclidean plane [7,15]. Because of this apparent computational intractability, one must resort to the use of approximation algorithms which, in general, produce suboptimal tours. Previous research has focused on the worst case behavior of such approximations [17,3,4]. Such results tend to be overly pessimistic since worst case data seldom, if ever, is encountered in practice. Furthermore, one may still expect most reasonable approximation methods to perform about equally well on random inputs, even though the worst case performances of the algorithms may vary greatly. Experience in working with several approximations tends to confirm this hypothesis [9]. The primary motivation for this work is to provide a theoretical basis for explaining this intuition and experience.

In this paper, we applied the methods of order statistics to estimate the expected lengths of the tours produced by several approximation schemes for the Euclidean TSP. To do so, we selected one specific distribution of points for extensive study. A primary reason for choosing the two-dimensional normal distribution was that it proved to be mathematically tractable. Furthermore, this distribution conforms to all of the statistical assumptions made in prior investigations, and the $O(\sqrt{n})$ tours produced are also in line with previous work [1,14]. Hopefully, the distribution is typical of this class so that one might expect somewhat similar results to hold had a different choice been made.

Our principal conclusion is that for the distribution chosen, all of the approximation algorithms studied produce a tour whose expected length is within a small constant factor of optimal. One line of possible future research would be to investigate the variance in path length associated with the algorithms, again using order statistics. A low variance would tend to enforce our belief in the algorithm's ability to produce generally good tours, whereas a high variance would make us more skeptical of the method. Another possible line of investigation would be to extend the results to other specific distributions or, better yet, to general classes of distributions obeying certain statistical assumptions.

Perhaps the most important contribution of this work is to show *how order statistics* can be applied to say significant things about the expected behavior of heuristics for the Euclidean TSP. There is no reason why these techniques could not be applied to other computational problems, as well. One way of coping with the apparent intractability of NP-complete problems is to devise fast procedures which approximate the optimal solution. To date, most research has focused on deriving worst case performance guarantees for these methods, while very little is known about their expected performance. Since many of these approximations can be characterized as "greedy" algorithms (i.e., they minimize or maximize some criterion at each step), they would be good candidates for the application of order statistics provided it is possible to characterize reasonably the distribution of inputs. Further explorations of this type could be most useful and interesting.

References

[1] J. Beardwood, J. H. Halton, and J. M. Hammersley, "The Shortest Path Through Many Points", _Proc. Cambridge Phil. Soc._, Vol. 55, No. 4 (Oct. 1959), pp. 299-327.

[2] M. Bellmore and G. L. Nemhauser, "The Traveling Salesman Problem: A Survey", _Oper. Res._, Vol. 16, No. 3 (May 1968), pp. 538-558.

[3] N. Christofides, "Worst Case Analysis of a New Heuristic for the Traveling Salesman Problem", Technical Report, Graduate School of Industrial Administration, Carnegie-Mellon Univ. (1976).

[4] G. Cornuejols and G. L. Nemhauser, "Tight Bounds for Christofides' Traveling Salesman Heuristic", _Math. Programming_, Vol. 14, No. 1 (Jan. 1978), pp. 116-121.

[5] H. Cramér, _Mathematical Methods of Statistics._ Princeton Univ. Press, Princeton, NJ, 1958.

[6] H. A. David, _Order Statistics._ John Wiley and Sons, New York, 1970.

[7] M. R. Garey, R. L. Graham, and D. S. Johnson, "Some NP-Complete Geometric Problems", _Proc. 8th Annual ACM Symp. on Theory of Computing_ (1976), pp. 10-22.

[8] E. J. Gumbel, _Statistics of Extremes._ Columbia Univ. Press, New York, 1958.

[9] P. V. Kamat, "Expected Behavior of Approximation Algorithms for the Euclidean Traveling Salesman Problem", M.S. Thesis, Univ. of Rhode Island (Aug. 1978).

[10] R. M. Karp, "The Probabilistic Analysis of Some Combinatorial Search Algorithms", in _Algorithms and Complexity: New Directions and Recent Results_, J. F. Traub (ed.), Academic Press, New York, 1976, pp. 1-19.

[11] R. M. Karp, "Probabilistic Analysis of Partitioning Algorithms for the Traveling Salesman Problem in the Plane", _Math. Oper. Res._, Vol. 2, No. 3 (Aug. 1977), pp. 209-224.

[12] G. S. Leuker, "Some Techniques for Solving Recurrences", _Comput. Surv._, Vol 12, No. 4 (Dec. 1980), pp. 419-436.

[13] H. R. Lewis and C. H. Papadimitriou, "The Efficiency of Algorithms", _Sci. Amer._, Vol. 238, No. 1 (Jan. 1968), pp. 96-109.

[14] L. Yu. Morozenskii, "On The Asymptotic Length of a Commercial Traveler's Path When Towns are Randomly Allocated", Theory Prob. Applications, Vol. 19, No. 4 (Dec. 1974), pp. 798-801.

[15] C. H. Papadimitriou, "The Euclidean Traveling Salesman Problem is NP-Complete", Theoret. Comput. Sci., Vol. 4, No. 3 (1977), pp. 237-244.

[16] R. C. Prim, "Shortest Connection Networks and Some Generalizations", Bell Sys. Tech. J., Vol. 36, No. 6 (Nov. 1957), pp. 1389-1401.

[17] D. J. Rosenkrantz, R. E. Stearns, and P. M. Lewis, "An Analysis of Several Heuristics for the Traveling Salesman Problem", SIAM J. Comput., Vol. 6, No. 3 (Sept. 1977), pp. 563-581.

[18] B. W. Weide, "Statistical Methods in Algorithm Design and Analysis", Ph.D. Thesis, Carnegie-Mellon Univ. (Aug. 1978).

ALGORITHMIC COMPLEXITY
Part 7

by

Leonard J. Bass

DATA BASE ACCESS METHODS

ABSTRACT

A survey is made of several different access methods for both univariate and multivariate range queries. These techniques include B-tree and extendible hashing as univariate techniques and radix bit mapping and K-D-B trees as multivariate techniques.

All techniques discussed are currently suitable for practical use.

DATA BASE ACCESS METHODS

As the requirements for accessing large data bases have
grown, the techniques used in managing these accesses have
become more sophisticated. In this paper, several of these
techniques are surveyed. First we review K-ary and radix trees
which are utilized by the access methods discussed. Next we
discuss two different univariate access techniques, B-trees and
extendible hashing. Finally we present two multivariate access
methods: radix bit mapping and K-D-B trees. All of the
techniques discussed are currently suitable for practical use.

The problem we are discussing is the accessing of a data
base by the values of one or more of its variables. That is, a
large data file exists which contains records for many
variables and it is desired to retrieve the record(s) with the
specified values of certain variables. The forms of this
problem depend on the number of variables used to define the
records desired and whether these variables are defined
specifically (with a single value) or by a range of values.

More formally, if each record of the data base has
variables X_1, X_2, ..., X_s then there are four degrees of
generality for this problem.

1) For a fixed i and value v, locate all records with

   $X_1 = v$ (univariate match)

2) For a fixed i and $u \leq v$ locate all records with
   $u \leq X_i \leq v$ (univariate range)

3) For a set of variables $X_1, \ldots, X_r$ $r \leq s$ and a corresponding set of values $v_1, \ldots, v_r$ locate all records with $(X_1, \ldots, X_r) = (v_1, \ldots, v_r)$ (multivariate match on r variables)

4) For a set of variables $X_1, \ldots, X_r$ $r \leq s$ and two corresponding sets of values $u_i \leq v_i$ $i = 1, \ldots, r$ locate all records with $u_i \leq x_i \leq v_i$ $i = 1, \ldots, r$ (multivariate range on r variables)

On a typical computer system the central processor operates about 1000 times faster than the associated input/output processor. Since a data base resides on an external device (generally a disk drive) the most important measure of a data base accessing algorithm is the number of I/O requests that must be satisfied to execute the algorithm.

Furthermore, data bases change over time, and these changes are reflected in modification of data items. A modification is a deletion followed by an insertion and another important measure of an accessing algorithm is how well it adapts to changes in the underlying data base.

Our focus, then, will be on the amount of I/O necessary to access and modify a collection of records in a data base.

Notation

We are dealing with a data base of N distinct records, each with its own physical record address. Within each record we have s special variables which are to be used to access the data base.

7-2

For each query, we are given $r \leq s$ pairs of values, u_1, ..., u_r and v_1, ..., v_r and we are looking for those records $(X_1, ..., X_r)$ with $u_i \leq x_i \leq v_i$ for $1 \leq i \leq r$. (Note that we are assuming that the variables in the data base have been numbered in a certain order and any query must be couched in terms of the first r of these variables. This is certainly not true in practice but it simplified our notation and, for the purposes of analysis any two variables are interchangable).

Within the data base, for each pair of values u_i, v_i we have M_i records which satisfy condition $u_i \leq X_i \leq v_i$ and, for r pairs of values (u_i, v_i), $i = 1, ... r$ we have M records which satisfy condition $u_i \leq X_i \leq v_i$ for $i = 1, ..., r$.

By choosing $u_i = v_i$ we have the exact match problem and by making $u_i < v_i$ we have the range problem.

Trees

A node of a k-ary tree based on variable X (called the key) consists of k values of x together with k pointers to other nodes.

If node n_1 contains a pointer to node n_2
then   a) n_1 is called a parent of n_2

          b) n_2 is called a child of n_1

Descendent is defined in the obvious manner from child. Any node which has no children is called a leaf node.

A k-ary tree is a finite collection of nodes such that:

1) there is exactly one node (named the root) which is the child of no other node.

2) Every node other than the root is the child of exactly one node.

3) No node is a descendent of itself.

An ordered k-ary tree is a k-ary tree in which, for any node, the k values of X are ordered $x_1 \leq x_2 \leq \cdots \leq x_k$ and for non-leaf nodes, the k pointers $P_1, \ldots, P_k$ are such that $x_i \leq x'$ for every value x' in P_i or any of its descendents. I.E. x_i provides a lower bound for any value in any descendent.

We will only be dealing with ordered trees and will assume the ordering without specific reference.

A k-ary tree provides a mechanism for searching the list of values $x_1, \ldots, x_n$ to locate a particular value y. The algorithm proceeds as follows

0) Set P to be root node

1) Search node P with values $x_1, \ldots, x_k$ of tree to find i such $x_i \leq y < x_{i+1}$, if no such i set i = k.

2) Retrieve node p_i.

    a) If node is not leaf then set $P = p_i$ and repeat step 1.

    b) If node is leaf then if y is in $x_1, \ldots, x_n$ it will be in node p_i.

Figure 1 gives an example of a k-ary tree and a retrieval from a k-ary tree.

7-4

Since we are concerned with I/O requests, with appropriate choice of k, retrieving a node requires exactly one read.

Searching for a value requires traversing the tree from the root to a particular node and thus the worse case measure of the number of read requests is the length of the longest path from the root to a leaf. (This is the height of the tree.)

The height of an ordered k-ary tree is minimized if the tree is kept balanced as points are inserted into or deleted from the tree.

In the applications we will make of trees several points can be made.

1) We are assuming the existence of a data base and the various types of trees will provide an access path to the data base based on the values of variables. It does no good to find a value efficiently using a tree structure unless we can subsequently locate the associated data record. Thus, we will assume that the values in the leaf nodes have associated with them the appropriate record number.

2) Our definitions allow the values at the higher level nodes to either appear again at lower level nodes or to have the retrieval algorithm terminate when it successfully finds a value at a higher level. In our applications all values from the data base will occur at the leaves. An implication of requiring all of the data values to occur at the leaves is that the values at the non-leaf nodes need not be values from the data

7-5
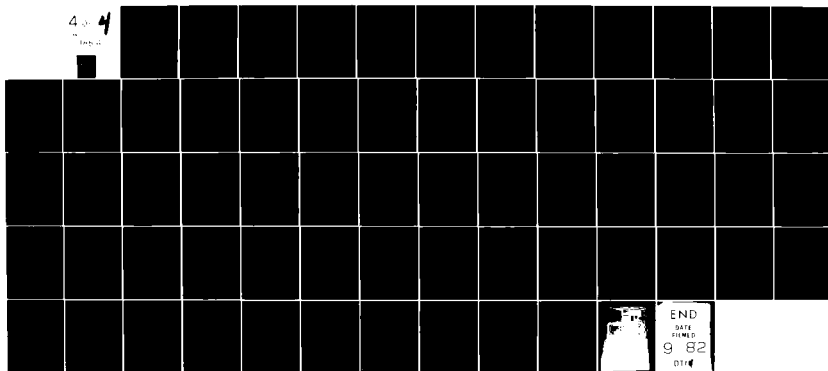
base. The values at the non-leaf nodes serve only to discriminate between the values at the leaves. A type of tree where only the first portion of a value is used to discriminate is called a radix tree.

B-tree

The first access method we shall examine is the B-tree of Bayer and McCreight (2). In this section we present the univariate version of this structure. In subsequent sections we present two different applications of B-trees to solve the multivariate range searching problem.

A B^+-tree is an ordered k-ary tree where k chosen to be the maximum number of items that can be read with one read (kept in a single page in a virtual memory environment). In a B^+-tree all of the x_i, i = 1, k appear in the leaf nodes, regardless of whether they also appear in a non-leaf node.

The searching algorithm for a B^+ tree we have already given. We now give the algorithm for insertions and deletions and then discuss these algorithms.

Insertion

To insert a new value y into an existing B^+ tree, use the following algorithm.

1) Search for y in tree and locate the leaf node which would contain y if it were already in list.

2) Insert y into node.

3) If now are less than or equal to k values in node then exit.

4) There are now k+1 entries in node. Split node into two nodes, both with same number of elements (both k/2 if k is odd or one with (k+1)/2 and one with $\frac{k-1}{2}$ if k is even). The two new nodes are such that if x is in node A and y is in node B then x < y.

5) The new node must now be reflected in the parent of the split. Retrieve parent, insert smallest value of node B and pointer to node B in parent. Modify discriminant within parent for A (if necessary).

6) Repeat steps 3-5 for new node.

Figure 2 gives an example of the splitting process.

Deletions

To delete a value y from an existing B^+ tree use following algorithm:

1) Locate value y in tree in leaf node P.

2) Delete y from node P.

3) If greater than or equal to $\frac{k}{2}$ values in P ($\frac{k-1}{2}$ if k is odd) or P is a root then exit.

4) There are now k/2-1 entries in node. Choose sibling (Q) of node with same parent. If node Q has more than k/2 entries move one (either smallest or largest) entry from Q to P. Reflect new discriminant values in parent of Q and P and exit.

If node Q has k/2 entries then merge nodes Q and P into node P and delete node Q.

5) Retrieve parent of Q and call it P. Delete reference to Q from R.

Repeat steps 3-5.

It should be obvious from the insertion/deletion algorithm that no node (excluding root) will ever have fewer than k/2 items in it. Thus the maximum possible height of the tree with N items is $\log_{k/2}$ N. Thus a retrieval will take at most $\log_{k/2}$ N reads. It should also be obvious from the insertion algorithm that at most one split can occur at each level of the tree. Retrieval and splitting are the only I/O operations required by insertions. Thus at most 2 $\log_{k/2}$ N I/O operations are required for an insertion.

Deletions also require at most one merger per level. Thus, deletions also can be done in $O(\log_{k/2}$ N) I/O operations.

The type of B tree we have presented maintains all of the data items in the leaf. Thus to solve the range query in one dimension it is only necessary to search for the lower bound of the range and then traverse the tree until the upper bound is reached. This takes at most $\log_{k/2}$ N + $\frac{2M}{k}$ reads where M is the number of data items in the range.

Note that the solution to the range query retrieves the values in increasing order of the key.

Extendible Hashing

B-trees operate in logarithmic number of I/O requests and give the ability to retrieve records in order on the key being searched. If it is not desired to retrieve records in key order another univariate technique is available with a better expected retrieval behavior (although not necessarily a better insertion/deletion behavior).

This technique (extendible hashing (3)) is a combinatio of radix trees and hashing - a well established technique for randomized but repeatable access into a table. We assume a general familiarity with hashing. A general introduction to hashing is provided by Standish (5).

Hashing into a fixed size table (say of size n) consists of two parts.

1) A randomizing function f such that if x is an arbitrary key value and $y \leq n$ the probability that $f(x)=y$ is $1/n$. (f distributes the keys uniformly from 1 to n).

2) If $x \neq y$ and $f(x)=f(y)$ and x is already in the table then a method exists which will find a free cell to hold y. This is called collision resolution.

Two problems exist with the standard hashing techniques. These are

1) The table size, n, must be chosen a priori. Hashing works well when no collisions occur. If n was chosen too small for the particular set of data inserted into n then no good remedy exists.

2) It is not easily possible to access the values from the table in a specified order. Since f was chosen to be a randomizing function, it cannot simultaneously maintain a particular order of the keys. This is only a problem if retrieving in key order is a requirement of the particular application.

The algorithm we now present removes the first of these problems and allows the table size to grow dynamically. The basic idea behind the algorithm is to build a radix tree using the hashing function as the search mechanism for the tree.

The algorithm assumes the existence of a randomizing function f such that $1 \leq f(x) \leq 2^n$ where n is chosen so that 2^n is the largest possible table size. n=32 is a typical type of value.

At any point in time, there is a value d which reflects essentially the table size. The first d bits of $f(x)$ are used as the radix with which to index into the hash table. Thus the root of the radix tree is 2^d entries long. The tree has only one level, aside from the root.

The retrieval algorithm works as follows for a key x.

1) Calculate $f(x)$.

2) Retrieve current depth, d, of the root. Use the first d bits of $f(x)$ to index into the root of the tree. The value retrieved is a pointer to a leaf node which contains x (if it is in the table).

3) Hash x into leaf using standard hashing and collision resolution techniques.

7-10.

Figure 3 gives an example of this type of tree with d=3.

This algorithm takes exactly two read requests to retrieve a value. The first request reads in the correct portion of the root (no requirement exists that the entire root be retrievable with one read). The second read retrieves the leaf node with the desired value. The correct portion of the leaf can be retrieved directly from the first d bits of $f(x)$ and thus no searching need be done to find the pointer to the leaf.

Observe in Figure 3 that several locations in the root point to the same leaf. This allows, for example, the doubling in size of the root node without affecting any of the leaf nodes. Thus if $d_1 d_2 d_3$ is a 3 bit binary number with pointer P, by setting $d_1 d_2 d_3 0$ and $d_1 d_2 d_3 1$ to both have pointer P we have increased d from 3 to 4, doubled the size of the root and not affected the leaf nodes.

The insertion algorithm for the extendible hashing structure will now be presented.

To insert a value x into the extendible hashing structure:

1) Locate the leaf node for x by retrieval algorithm.

2) If node is not full insert x by hashing and collision resolution and exit.

3) If node is full and node is pointed to by several places in root (this can be detected efficiently) then split node into two nodes according to division in parent node. Leave d unchanged.

4) If node is full and is not pointed to by several places in root then the size of the root is doubled, by incrementing d, each non-affected pointer in root is replicated and then the node containing X is split into two as in 3).

The deletion algorithm is similar and will be omitted.

As can be seen from the insertion algorithm the behavior of the extendible hashing algorithm depends heavily upon the uniformity of the function f. In the worst case the behavior of the algorithm is linear in n but both analytic and simulation results (3) indicate that the expected behavior of the algorithm is somewhat better than that of B-trees.

Discussion

Both algorithms discussed provide efficient access to a set of keys. B-trees are logarithmic in both the expected and the worst cases. Extendible hashing is the order of a constant in the expected case for retrievals and apparently logarithmic in insertions (based on timing charts and not analytic results).

Both provide for dynamic modification of the underlying data base and respond well to modifications.

B-trees require at most $\frac{2n}{k} \log_{k/2} n$ pages of disk storage and allow for retrieval in the order of a key once the lower bound has been found.

Extendible hashing takes 2 page accesses for a retrieval

and requires at least $n/k + \log_2 n/k$ pages on the disk.

Extendible hashing also will not allow retrieval on key order

but only in f(key) order.

Multivariate Retrieval

We now turn to the more general case of finding those

records such that if (u_j, v_j) $j=1, \ldots, r$ are r ranges then

retrieve all records with $u_j \leq x_j \leq v_j$ for $j = 1, \ldots, r$.

If r=2 then this may be visualized geometrically by viewing

records in the data base on points in two space and

(u_1, v_1), and (u_2, v_2) as defining a rectangle in two

space. In this case we wish to retrieve all points that lie in

the rectangle. If r=3 we are in 3 space and are defining a

rectangular solid and in general we are defining a region in r

space.

The geometric interpretation will become useful in the

second algorithm presented which solves the problem in

r-space. The initial algorithm we present will solve this

problem by iterating on the range problem for each of the keys

and thus essentially solves the problem by projecting the

rectangle onto each of the coordinates in turn. This algorithm

has been implemented in a statistical data management system

available on mini computers (1).

Assume all the records of the data base are numbered
1, ..., n and assume that given $1 \leq i \leq n$ we can retrieve the
record easily. The algorithm we present maintains B-trees for
each of the r desired keys. Associated with each key is the
number of the record wich contains it.

The B-trees are maintained permanently on the disk. When
responding to a particular query a radix tree is created. This
radix tree contains the current set of records that satisfy the
query. The philosophy behind the construction of the radix
tree is to view the leaves of the tree as being N consecutive
bits. If record i satisfies the current query then bit i will
be on, otherwise it will be off.

Viewing this bit map as a radix tree both reduces the
memory required (under reasonable assumptions) and simplifies
the retrieval from the tree. Suppose each leaf can hold k
bits. Then to locate record i in the radix tree use i/k as
the radix and interrogate the bit numbered i mod k in the
appropriate leaf.

E.G. If k=1024 and we wish to indicate the presence of
record 18360 then turn on bit 952 in the leaf pointed to by the
17th pointer in the root.

Using a radix tree rather than a standard bit map
introduces one additional node (the root) and allows the
omission of any leaf not referenced by a particular query.

If 256 pointers can be kept in the root and each node contains 256 x 16 bits then 10^6 records can be represented with a tree of height two. Once the individual trees have been constructed for each variable then they can be merged into a tree which contains the desired subset. Figure 4 demonstrates this process for one variable.

The algorithm for retrieving the desired subset for a multivariate range query is

1) for each of the r keys (say j) construct the radix tree that reflects those records with $u_j \leq x_j \leq v_j$.

2) Merge the r constructed radix trees by ANDing the leaves together.

3) The resulting radix tree contains exactly the records desired.

The difficulty of constructing the radix trees for a single key depends upon the number of distinct pages referenced by the range of values for that key, if the first d bits of the record number are the same (where d is the length of $^i/k$ in bits) only one page is referenced, etc. If P_j pages are referenced by the jth key and M_j are the number of distinct data records in the jth range then the construction of the jth prefix tree takes

$$\log_{k/2} n + \frac{2M_j}{k} + P_j$$ page references and the determination

of the appropriate subset takes

$$r \log_{k/2} n + \sum_{j=1}^{r} \left(\frac{2M_j}{k} + P_j \right)$$

Thus this algorithm works well if the projection on each axis contains points that are clustered together in terms of data base record number. It also works well if sufficient real memory is available to hold the constructed prefix trees since then no page faults would be generated.

Notice, however, that if $r=1$ and $u_1 = v_1$ and the keys are unique that the construction of the prefix tree requires one additional page reference beyond the B-tree retrieval. If this page is permanently allocated in real memory then for the case of a single unique identification variable this method costs no additional input/output

Since this algorithm depends upon univariate B-trees, if the data base is updated the individual B-trees respond to the changes as already discussed.

Also, once a desired subset is defined we can retrieve the records in the subset in the order of a particular key by retrieving from the B-tree for that key and using the radix tree to determine whether each record was in the desired subset.

Finally, since the data structures permanently maintained, the B-trees, are univariate the only dependence upon more than one variable is in response to a specific request. Thus, the B-trees that are maintained are suitable for univariate requests on any key or multivariate requests on any combination of keys.

K-D-B Trees

The final algorithm and data structure discussed provides promise of a more efficient access for requests couched in specified a priori terms of set of keys. This structure is a generalization of B-trees themselves to multiple dimensions. For simplicity we present the 2-dimensional case and the higher dimensional structures are similar.

One dimensional B-trees can be viewed geometrically as providing a partitioning of an axis with (roughly) an equal number of points in each interval. The k adjacent partitions are grouped into one partition at the next higher level to provide the access path.

The K-D-B tree (4) is a generalization of this geometric view to higher dimensions. In the 2-dimensional case instead of partitioning a single axis into intervals as in one dimension, we partition the plane into rectangles. At the lowest level each rectangle has (roughly) the same number of points. At higher levels, the access paths are provided by grouping rectangles from lower levels into larger rectangles. See figure 5 for a graphical representation of a 2-D-B tree.

Thus, a 2-dimensional range query defines a rectangle in the plane and all rectangles in the 2-dimensional K-D-B tree that overlap the desired region would be searched to retrieve all the records that satisfy the query.

If, in fact, all of the lowest level rectangles had roughly the same number of points, this structure would guarantee a logarithmic worst case behavior. The problem is that no insertion and deletion strategies currently exist which guarantee a minimum number of points in a rectangle.

This is most easily seen when dealing with deletions, although a similar problem exists with insertions. Recall that the deletion algorithm for one dimensional B-trees provided for merging two adjacent intervals when both had less than $k/2$ points. This works because two adjacent intervals also define an interval. When dealing with rectangles, however, this does not work. If A and B are two adjacent rectangles then one edge of A must be a portion of an edge of B (or vice versa). If the overlapping edges are not identical then the merger of the two rectangles is not a rectangle. Thus, when deleting points, either the definition of the regions in terms of rectangles must be abandoned or the merger of two sparsely filled rectangles must he abandoned.

Robinson (4) advocates eliminating the merger step when deleting which in the worst case could result in empty rectangles. A similar problem results when doing insertions.

The mechanism for using K-D-B trees then is to build the underlying data base first. Then build the K-D-B trees and use ad hoc techniques to allow for such insertions and deletions as may occur. Simulations show that the expected behavior in such circumstances is very good.

Comparison

Figure 6 gives a table which compares the four algorithms we have surveyed. They are compared from the point of view of time to retrieve, time to update, and suitability for various types of queries.

If only univariate exact match queries are expected then either B-trees or extendible hashing provide the best responses. The choice between those two should be based on the dynamic nature of the data base. If the data base is relatively static (few insertions or deletions) then extendible hashing is recommended. If the data base is highly dynamic then B-trees are recommended. If univariate range queries are expected as well as exact match, then B-trees are recommended.

If multivariate exact match queries are expected with little a priori knowledge of which variables are involved then either B-trees or extendible hashing may be used to create the radix bit map. Again the choice is based on the dynamism of the data base.

If multivariate exact match queries are expected for a specific set of variables then a K-D-B tree could be constructed for those variables if the data base is not very dynamic.

If multivariate range queries are expected then the choice is between K-D-B trees and B-trees with radix bit mapping. If the underlying data base is not highly dynamic and the queries are always in terms of the same variables then use a K-D-B tree otherwise use the univariate B-trees with radix bit mapping.

7-19

Data Base

Record Number	Key Value
1	100
2	10
3	28
4	36
5	5
6	83
7	75
8	62
9	93
10	68
11	50
12	41
13	12
14	21
15	32

Node 1: 5 ② 62 ③

Node 2: 5 ④ 21 ⑤ 36 ⑥

Node 3: 62 ⑦ 83 ⑧

Node 4: 5 ⑮ 10 ② 12 ⑬

Node 5: 21 ⑭ 28 ③ 32 ⑮

Node 6: 36 ④ 41 ⑫ 50 ⑪

Node 7: 62 ⑧ 68 ⑩ 75 ⑦

Node 8: 83 ⑥ 93 ⑨ 100 ①

To retrieve 41
visit nodes 1, 2, 6
and find it is in
record 12
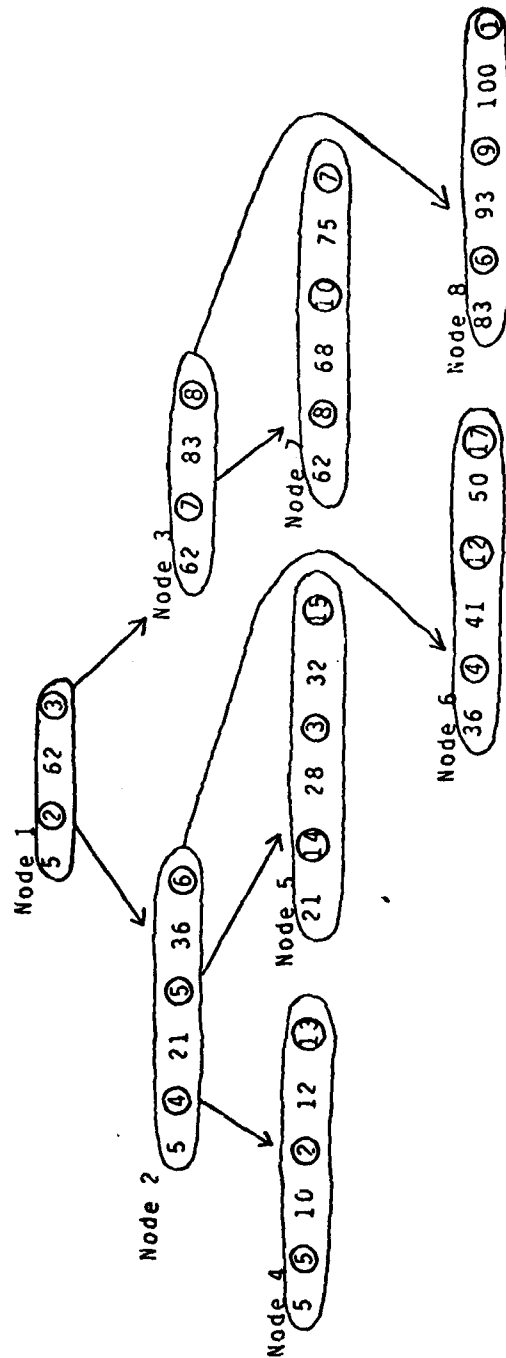
Pointers designated by O

- in non leaf nodes pointers
  point to other nodes

- in leaf nodes pointers
  point to data base
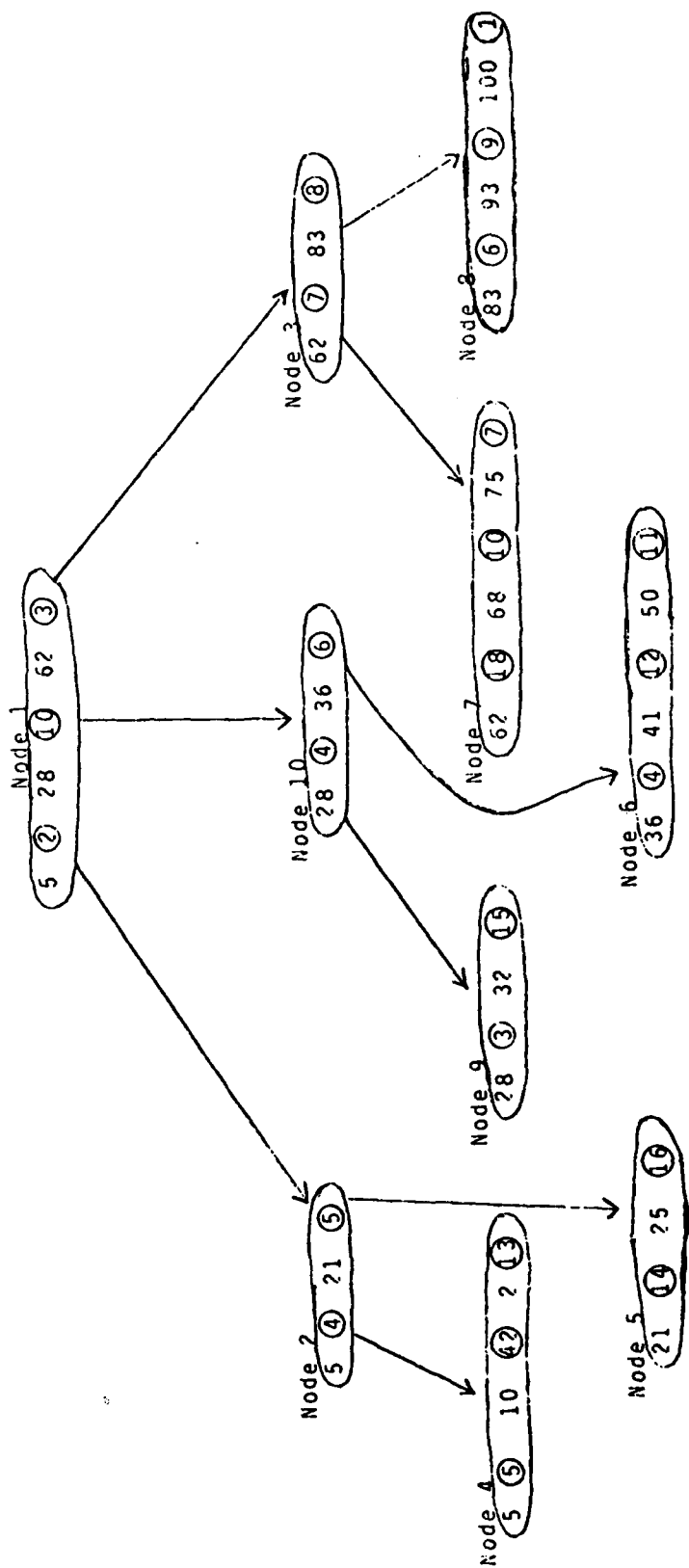
Figure 1
3-ary tree

Figure 2

After adding record 16 with key value 25 to Figure 1

Root

First 3 bits of f(x)

Depth=3

000
001
010
011
100
101
110
111

f(x)=00...

f(x)=010....

f(x)=011....

f(x)=100....
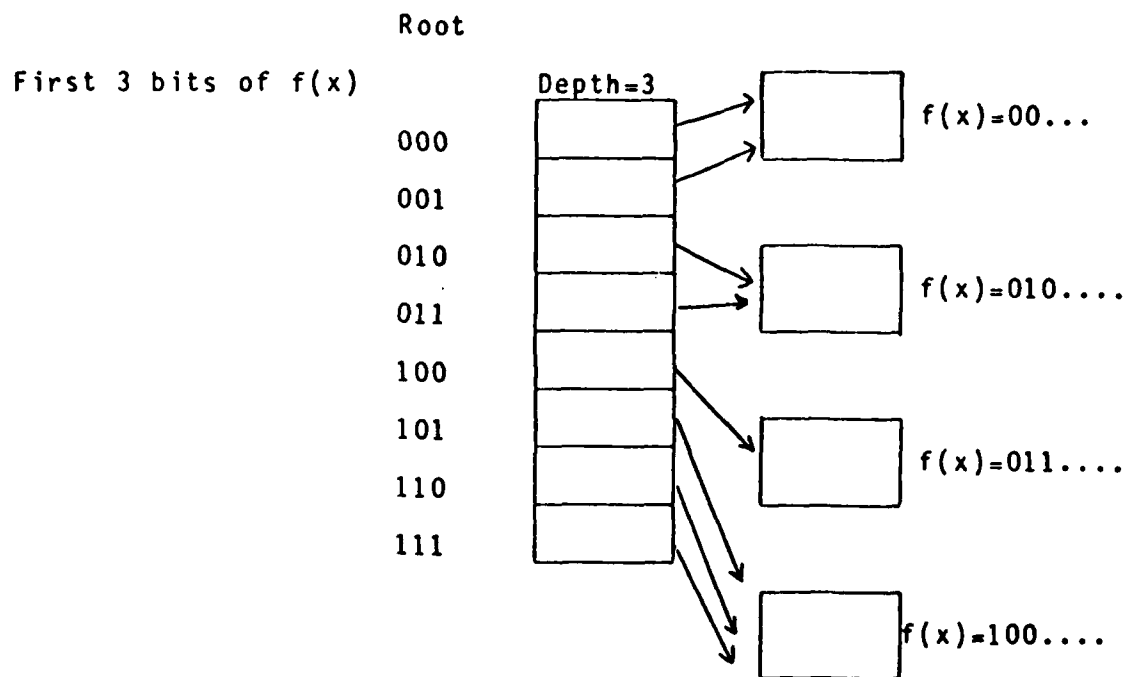
Figure 3
Extendible Hashing Tree

```
 ┌───┬───┬─┬─┬─┬─┬─┬─┬─┬─┐
 │   │   │0│0│0│0│0│0│0│ │
 └─┬─┴─┬─┴─┴─┴─┴─┴─┴─┴─┘
   │    └──────────┐
   ▼               ▼
 ┌─────┐         ┌─────┐
 │  3  │         │ 11  │
 │     │         │     │
 │  4  │         │ 12  │
 └─────┘         │     │
                 │ 15  │
                 │     │
                 │ 18  │
                 └─────┘
```
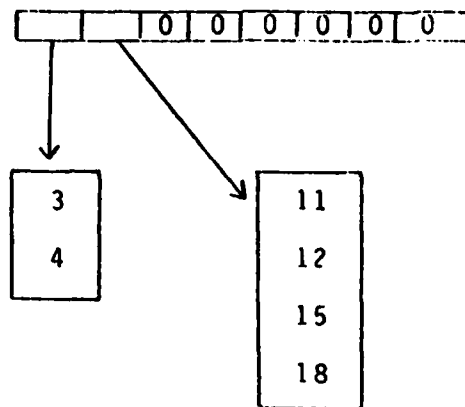
Values in root node are implied by having pointers.

Values in leaf nodes are maintained by turning on appropriate
bit in node.

Figure 4
Radix bit map after searching Figure ?
for $28 \leq x \leq 62$
(assume 8 entries per node)

Node 1

Not Shown

Not Shown

2-0-8 Tree
Figure 5

7-24
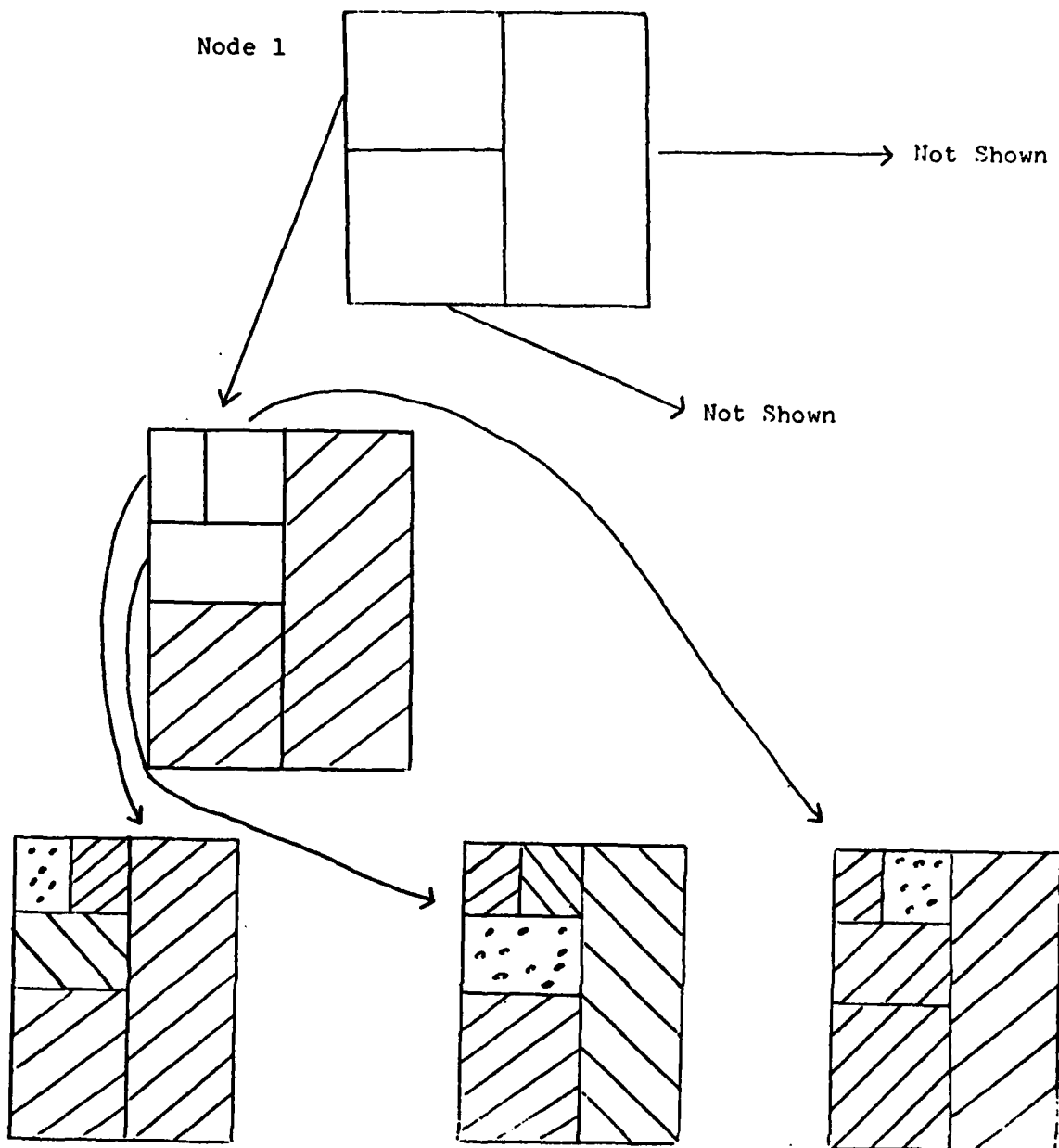
	Number of Variables	I/O Behavior for Retrieval	I/O Behavior for Modification
B-tree	1	log (worst case)	log (worst case)
Extendible Hashing	1	? (worst case)	log (expected)
Radix Bit Mapping Using B-trees	r (chosen as result of query)	r log (if clustered in records referenced)	r log (worst case)
K-D-B	r (chosen a priori)	log (expected)	log (if not too dynamic)

Figure 6
Comparison of Methods

References

1) Bass, L. J., "DATMAN FORTRAN User's Guide", URI Computer
     Science Department, TR80-1456.

2) Comer, D., "The Ubiquitous B-tree", Computing Surveys,
     Vol.2, No.3, September 1980.

3) Fagin, R, Nieverge, H. J., Pippenger, N. and Strong, H. R.,
     "Extendible Hashing - A Fast Access Method for Dynamic
     Files", ACM Transactions on Data Base System, Vol. 4,
     No. 3, September 1979.

4) Robinson, J., "The K-D-B Tree: A Search Structure for
     Large Multi-Dimensional Dynamic Indexes", Proceedings
     1981 ACM-SIGMOD Conference, May 1981.

5) Standish, T. "Data Structure Techniques", Addison-Wesley
     1980.

ALGORITHMIC COMPLEXITY
Part 8

by

Ralph E. Bunker
and
Leonard J. Bass

AN EXPERIMENTAL EVALUATION OF THE FRAME MEMORY
MODEL OF A DATA BASE STRUCTURE

ABSTRACT

Frame memory is an analytic model of a data base access method. This model enables the prediction of access performance measures in terms of user behavior parameters. This is an important aspect of the automatic generation of data structures.

In this study, a version of frame memory was implemented and then a simulation study was performed to validate the predictions of the analytic model against the implementation.

The model yielded good predictions (less than 10% error) for most of the cases tested. The assumptions under which the analytic results were derived were violated during a portion of the simulation to test the robustness of the model and again, the analytic model yielded good predictions.

AN EXPERIMENTAL EVALUATION OF THE FRAME MEMORY
MODEL OF A DATA BASE STRUCTURE

A desirable goal of data base research is the automatic generation of data base structures. A designer would specify some limited number of characteristics of the data and would have automatically returned the data structures, the access items, and the access paths. A step in the direction of that goal would be for the designer to furnish usage information and a proposed storage structure, and to have returned the expected response parameters. The frame memory model of storage structure has been proposed as a mechanism for predicting system response as a function of usage and structural information. In this study, we report on an experimental validation effort for frame memory.

Most attempts at automatic design involve the following steps:

1. Determine how the users of the file system are planning to use the system. This provides the necessary input for the automatic design system. Usage is defined by the different types of records in the system, their lengths and fields, plus the expected frequencies of additions, deletions, modifications, and retrievals to records and subsets of records in the file.

2. Select a set of storage structures for the records based on usage patterns defined in step 1.

3. Evaluate how this set of storage structures perform in the anticipated environment. This evaluation must take into account the change that the storage structures will undergo due to maintenance.

4. Assign a rating to the set of storage structures based on this evaluation. This rating will determine whether or not the set of structures will be considered further as a possible design choice.

5. Inform the designer as to the set of structures which have received the best evaluations.

Frame Memory

We are interested here in what is involved in step 3 of the design process. This step is complex partially because the amount of time needed to retrieve data from a storage structure rarely remains constant throughout the life of the storage structure.

March (MAR78) has proposed that step 3 of the design process be divided into two steps as follows:

3a. Compute the average time to perform fundamental operations on the storage structure, taking into account the effects of updates to the storage structure. Fundamental operations include reading a logical block, scanning a logical block of records for a particular record, directly accessing a record, and writing a logical block.

8-2

3b. Use information from step 3a to calculate the average time to perform an operation of interest, which may involve a number of fundamental operations. For example, the operation of adding a record to a data structure can involve first the operation of reading in the logical block which will contain the record and then writing the updated logical block.

March proposed a model of secondary memory which he called frame memory. He also analyzed the cost of using this model to implement retrievals and modifications to a data base. The designer would specify data structure and retrieval requirements in terms of the frame memory. The cost of satisfying these requirements would be calculated and reported to the designer. The designer could then choose the best data structures.

This makes sense only if the equations used to predict the performance are correct and there is an implementation of frame memory so that the designer can then use this implementation to actually access the data structures created.

This provides the motivation for the study reported here. An implementation of frame memory was done and then this implementation was tested to see if the analysis yielded correct predictions. Some of the assumptions within which the analysis was done were violated to test the dependency of the analysis on those assumptions.

The results indicate that the predictions were close to experimental results for almost all cases.

8-3

From a user's perspective, frame memory partitions secondary memory into contiguous and directly accessible areas of storage called frames. A frame is the logical unit of data transferred between main and secondary memory. Information is maintained in contiguous areas within frames called frame records. Figure 1 gives a user perspective of a frame memory. Frames have four essential functional characteristics:

1. Directly accessible records. As each new, possibly variable length, record is stored in a frame it is assigned a local identifier called a (frame relative record) token. The association between the record and this token is unaffected by subsequent frame storage and maintenance operations.

2. Sequentially accessible records - Once a frame is transferred to main memory its records may be sequentially referenced in either their physical order or in a user constructed logical order termed the (frame referencing) stream (Figure 2).

3. Frame elasticity - A frame is capable of stretching to accommodate arbitrary growth. This is the way in which maintenance operations which change the number or size of records are handled in this model. Frame growth (or shrinkage) has no direct affect on other frame functions but is reflected indirectly in frame performance characteristics (Figure 3).

4. Record stream maneuverability - The inter-record
   structure of the frame reference stream is maintained
   by the frame memory and may be dynamically modified by
   a frame user.

Three implementation questions arise:

1. What type of structure will support frame expansion
   (function 3)?

2. What type of structure will be used to maintain the
   internal order of frame records within a frame
   (functions 2,4)?

3. What types of structure will be used to maintain
   tokens for records within a frame (function 1)?

The implementation was subject to the constraint that it
must conform to the basic assumptions that March used in
analyzing his model of frame memory. The fundamental data
structure that was used to support the frame expansion function
was a chained overflow structure. This allows the user to
perceive the frame as expandable while the frame memory
implementations actually decide how to handle the expansion.

A fixed amount of space is initially allocated in secondary
storage for each frame. This initial allocation is called a
prime frame and the area of secondary storage in which prime
frames are allocated is called the primary data area. The
primary data area spans a number of cylinders of a disk.
Within each of these cylinders a certain percentage of space is
set aside for the primary data area. The remaining space is
used for frame expansion and is called the local overflow

area. Records are stored in a prime frame until the space allocated to the prime frame is exhausted at which time an additional allocation of secondary storage space called a frame extent is made. There are two types of frame extents - local extents and global extents. A local extent exists in the same cylinder as the prime frame it is assigned to and is allocated from the local overflow area. A global extent exists in a global overflow area which is separate from the primary data area. Figure 4 depicts the relation betwen these frame components and cylinders. Since the global overflow area is separate from the prime data area, the head of the disk must be moved and hence access to it is more expensive. A local extent will always be used if there is space available within the local overflow area.

Frame extent allocations (either local or global) can be either fixed or variable. A fixed extent is generally capable of holding several records whereas a variable extent is only large enough to store the record which caused the extent to be allocated.

Once an extent has been allocated it is necessary to associate it with the prime frame being extended. This is done by maintaining an extent index in the prime frame which points to each extent associated with the prime frame. Figure 4 illustrates this method.

The expansion structure which was used has fixed length extents and an extent index. Although an extent index uses some space in a prime frame, the amount of space is usually

small compared to the size of records stored in the frame and an index has the advantage of locating any extent in only one access. Fixed length extents may waste space if variable length records are used. They do reduce the number of extents needed, however, which is desirable if an extent index is used.

Next, the maintenance of the logical frame stream will be discussed. This is basically a determination of how the concept of "next record" will be implemented. The next record is the one which would be physically contiguous to the current record if all of the records of the frame were contiguous. We used an indexed mechanism for maintaining tokens. That is, a pointer is maintained for each frame record; a token is a relative pointer count from the beginning of the index.

In review, the implementation of a frame memory which is used in this research has the following characteristics:

1. Extents are fixed length and maintained by an index stored on the prime frame.

2. The logical frame stream is maintained by address sequential connections (i.e., the physical order of the records correspond to the logical order).

3. Frame record tokens are maintained by a token index stored in the prime frame.

This implementation has been chosen for the test system for two reasons:

1. The prediction of its performance measures involves a complex analysis. The purpose of this implementation is to verify the correctness of March's analysis of

frame memory performance, hence, it is appropriate to choose an implementation which is difficult to analyze.

2. Other researchers have used a similar implementation for experimental relational data base management systems (STO76). This is a situation, therefore, in which automatic data base design research may find a valuable application in the future.

March analyzed frame memory in terms of two types of parameters: usage and device. The usage parameters specify the characteristics of how the frame is to be used. March proposed the following usage parameters for his frame memory:

NR - the number of records initially loaded into the memory.

LR - the average length (characters) of a stored record. Records may be of variable length.

NADD - the number of additions per time period.

NDEL - the number of deletions per time period.

RINT -

the reorganization interval. This is the time period at the end of which the frame memory will be taken off line and all extents will be incorporated into prime frames. This usually happens when performance of the frame memory has deteriorated significantly due to update generated overflow chains. All performance measures are averages over this interval. This measurement standard is inspired by the idea that the best file organization is the one with the maximum average performance over its lifetime.

Some of his device parameters are:

TLOC - average disk latency time.

TRAN - the average disk seek time.

TFRTE - the data transfer rate between main memory and
disk storage.

MLMB - the size of a track (bytes) on the disk used to
store the frame memory.

LSZE - the amount of data storable in a cylinder.

Other usage parameters (those connected to the device usage):

PALF - the primary area load factor. This is the percentage of each cylinder that is used for prime frames. The remaining space on the cylinder is used for the local overflow area.

FMLF - the frame memory load factor; the proportion of the space allocated to prime frames that is required to hold initially loaded data. If this is less than one, then each prime frame has some free space to use before allocating an extent.

FRAE - the length of a frame extent.

Figure 5 illustrates the mapping of an extended frame into physical storage and illustrates parameters FMLF and FRAE.

March analyzed ten different performance measures. We focus on:

1) FFPHY - the average time to retrieve a full frame in physical stream order. A full frame consists of a prime frame and whatever extents have been generated for it.

2) FRTOK - the average time to retrieve a frame record by its token.

The experiments we performed were intended for two purposes. First we validated March's analysis by using the assumptions he made in doing the analysis. Secondly, we violated his assumptions to explore the limitations of the analysis. In general, the results of the experiments showed the robustness of his equations without regard for whether the assumptions were maintained.

Six fundamental assumptions regarding the characteristics and use of the frame memory were incorporated by March into his equations. These assumptions are:

1. March assumes only the most primitive type of buffering. Only one prime frame or one extent can be in main memory at a time and the current contents of the buffer aren't checked before doing I/O. For instance, in the evaluation of the time it takes to read a frame in stream order, it is assumed that the frame has to be fetched from the disk. In a more realistic buffering scheme, there is the possibility that the frame is already in core and hence no I/O would be necessary to fetch it.

This assumption is equivalent to the one that no frame in the memory sustains consecutive additions or deletions of records. That is, each addition or deletion operation involves a frame different from the one used in the preceding operation. We followed this assumption in the experiments.

2. The amount of disk storage space needed for extent and token indices is small (10 percent) compared to the storage used for frame records. March ignores the effect of overhead in several of his equations. There are many situations in which this assumption is questionable. For instance, if frame records are only ten characters long then overhead may consume twenty to twenty five percent of the storage used for the table: hence, the percentage of overhead is a function of the record length of a frame record and the length of a system pointer. Both of these are implementation parameters. The length of a system pointer is increased (thereby increasing the overhead percentage) in the experiment which tests the affect of altering this assumption.

3. Maintenance operations (addition, deletion, and modification) are uniformly distributed over the set of prime frames. This is a key assumption which has many exceptions. For instance, it has been observed that in many data base systems twenty percent of the records are involved in eighty percent of the transactions on the data base. We tested the effect of this assumption in one of the experiments.

4. The storage device containing the frame memory is
dedicated to only one user. In other words, there is no
concurrent use of the frame memory by two or more users and the
frame memory storage device experiences no activity other than
that initiated by the frame memory user. It is expected that
future models of a frame memory will allow more than one user
since concurrent use of a data base is one of the prime
motivations for the development of data base management
systems. A multiuser environment is approximated by randomly
changing the position of the disk head during the processing of
the frame memory updates and retrievals. There are three ways
in which the head can move corresponding to three different
usage situations. First, it has not moved since the last time
that a particular user fetched something from the frame
memory. Secondly, it has moved but has stayed within the frame
memory data set. This happens when two or more users are
concurrently using the frame memory. Third, it has moved
outside the frame memory data set. This movement would be
caused by a user accessing a data set other than the frame
memory data set. In the experiment testing the impact of many
users it is assumed that 50 percent of the time the head
doesn't move and 50 percent of the time it moves within the
frame memory data set. The frame memory is assumed to occupy
the entire disk so there is no movement outside the frame
memory.

5. The rate of maintenance and retrieval activity is linear. In other words, there are no flurries of maintenance activity followed by lulls of no activity. Also, the update operations are dispersed uniformly. That is, there is not a batch of additions followed by a batch of deletions. The experiments that were performed adhere to this assumption.

6. The degradation of the frame memory is a function of the difference between the number of additions and the number of deletions and does not depend on the actual number of additions or deletions. That is, two hundred additions and no deletions cause the same degradation as four hundred additions and two hundred deletions. This assumption is implicit in the experiments performed since in all experiments only additions are made to the frame memory.

Measurement Techniques

For each of his performance measures, March calculates a value at "steady state". He defines steady state as the time at which the number of records initially loaded into the frame has been doubled. He assumes his performance measures degrade linearly from an initial value to the value at steady state. He calculates the performance measures at steady state as a function of physical and usage parameters such as average access time and number of global extents generated.

As an experiment proceeded the physical and usage parameters needed by the predictive equations were gathered by the implementation. These were the values used in calculating

8-13

the predicted values. The performance measures were also measured during the course of an experiment. This provides measurement versus predicted performance from the start of an experiment to the end of the experiment.

The performance measures taken during the course of the experiment (average time to read all records in token order and average time to retrieve a token) were assumed to have been modified in an interval only by the records actually added during that interval. Thus, the calculations for the performance measures were done with each addition by adding an incremented value (the time to access the record just added either in logical order or alone) to a running total. This enabled the calculation of average values without actually having to read all of the records.

Since the predicted values were based on actual physical and usage parameters, any dependency upon the method of estimating these values were eliminated from the experiments.

This methodology provided a means for testing the prediction equations while still enabling variations in some of the fundamental assumptions upon which the predictions were based.

Experiments

Six experiments were performed. Table 1 lists the values of the parameters that were varied for each experiment. A brief rationale is now given for the choice of experiments.

TABLE 1 - PARAMETERS OF EXPERIMENTS

E#	DIST	SIZE	EPF	NADD	NDEL	PALF	USERS	LP
1	U	160	5	200	0	0.85	one	2
2	U	80	10	200	0	1.00	one	2
3	U	400	2	200	0	1.00	one	2
4	U	160	5	200	0	0.85	one	2
5	U	160	5	200	0	0.85	one	10
6	U	160	5	200	0	0.85	many	2

The meaning of the parameter mnemonics are:

E#   - experiment number

DIST - the distribution used for selecting frames for updates.
Here U means uniform distribution and N normal
distribution.

SIZE - the size of an extent allocation

EPF - the number of extent sized blocks in a prime frame

NADD - the number of additions per unit time interval

NDEL - the number of deletions per unit time interval

PALF - percentage of a cylinder used for prime frames

USERS - the number of users competing for access to the frame
memory

LP   - the length of a system pointer (in bytes)

Experiments 1-3 adhere to all of March's assumptions and are a test for fundamental errors in his equations. Experiment 1 is run with a frame memory containing one local extent for each frame. Extents can contain two records and at load time a prime frame contains nine records. Experiment 2 uses a frame memory with no local extents. Each extent can hold only one record and as before the prime frame is big enough to hold nine records. Experiment 3 has the same conditions as Experiment 2 except that an extent can contain four records.

The remaining experiments test the effect of altering the assumptions which March used for his analysis.

Experiment 4 uses a normal distribution to determine which frames get updates (assumption 3). The frame memory used has local extents (one per prime frame) and each extent can contain two records.

Experiment 5 uses a large value for the length of a system pointer in order to make the overhead needed for each record approximately 10 percent of the record length (assumption 2). The frame memory used has local extents (one per prime frame) and each extent can contain two records.

Experiment 6 tests the effect of other users competing for use of the frame memory (assumption 4). The frame memory used has no local extents and each extent can contain only one record.

The following parameters were held constant for all of the experiments:

1) (LOGOVHD) The track size of the logical frame memory
              device was 3170 bytes.

2) (LOGOVHD) No overhead is needed for a block on the
              logical frame device.

3) (LOGTPC)   There are 12 tracks per logical cylinder.

4) (TCYLS)   The frame memory has 200 cylinders.

5) (PHYMLMB) A physical track can contain 19254 bytes.

6) (PHYOVHD) The overhead for a physical block is 135
              bytes.

7) (FMLF)   Each frame was initially completely filled
              with frame records.

8) (TLOC)   The disk latency time for the logical frame
              memory device was 36.3 milliseconds.

9) (ACCFUNC) Figure 6 illustrates the seek time function
              used for the logical frame memory device.

10) (TEST)   The frame memory operated in test mode.

11) (NUMFRMS) The number of frames is 1800.

12) (NR)   The number of records initially loaded is
              16200.

13) (RINT)   steady state is defined to occur after 81 unit
              time intervals. This is the time when the
              size of the file has doubled. This is the
              criteria for steady state that was used by
              March.

14. (NUMTRKS) The physical data set supporting the frame
              memory uses 42 tracks.

Results

The results of the experiments are summarized in Table 2 and graphically depicted in Figures 7 - 8.

TABLE 2 - PERCENT-OF-ERROR STATISTICS

FFPHY

EXP	MEAN	STD	MAX	MIN
1	12.83	21.51	20.76	0.68
2	6.62	13.37	10.88	0.15
3	24.13	59.17	35.39	8.93
4	6.65	2.61	9.06	0.44
5	5.78	4.68	7.45	0.38
6	3.28	2.35	5.04	0.05

FRTOK

EXP	MEAN	STD	MAX	MIN
1	4.44	1.34	5.34	0.67
2	10.48	0.64	12.97	9.55
3	6.04	3.66	7.50	0.26
4	8.18	3.93	9.87	0.62
5	7.59	3.32	8.52	0.26
6	6.10	0.55	6.85	2.93

A brief overview of the results will be presented first followed by a more detailed analysis. A predicted value within ten percent of the observed one is considered a good prediction. Of course, more precise predictions are desirable but for the current state of the art in automatic data base design, the ten percent error range will probably be accurate enough. Each performance measure will be discussed separately.

1. Average time to retrieve a full frame in physical order (FFPHY).

Experiments 1, 4, and 5 are all performed on equivalent frame memories (i.e., local extents are available ard two records can fit on an extent). Experiment 1 uses all of March's assumptions and the observed value of FFPHY is, on the average, within 12.83 percent of the predicted value. Changing the assumptions of small overhead per record (experiment 4) and a uniform distribution of updates (experiment 5) reduces the average error by about 50 percent in both cases. Experiments 2 and 6 are also run on equivalent frame memories (no local extents and one record per extent). For both the predicted value is well within 10 percent of the observed one. Experiment 3 produced a large discrepancy between the predicted and observed values (i.e., 24.13 percent).

2. Average time to retrieve frame by token (FRTOK)

All experiments but one produced observed values within 10 percent of the predicted values of FRTOK. The exception was experiment 2 for which the average percent of error was 10.48

percent. In all cases the observed value was higher than the predicted one.

In his analysis, March first computes averages for performance degradation at what he calls steady state time. This is merely the number of unit time intervals required to double the number of records initially loaded in the frame memory. The degradation at steady state for a performance measure is denoted by DS(*measure).

DS(*FFPHY) = ANEXS * ATRES

DS(*FRTOK) = FVTKS * ATRES

where ANEXS is the average number of extents per frame at steady state time. ATRES is the average time to read an extent at steady state. FVTKS is the probability, at steady state time, that if a frame has extents then the desired record is in an extent.

Table 3 contains the analytical and observed values of ANEXS and FVTKS. In general, there is a good agreement between the observed and predicted values. An exception is experiment 5 (large system pointer overhead) for which there is not good agreement for either . Table 3 also contains a breakdown of the average time to read an extent into two different kinds of averages. The first average (AVGE) is the average time to read an extent given that the head is positioned at the cylinder containing the extent (or prime frame) which immediately precedes it in the chain of extents attached to the prime

frame. This is the average read time that is expected when a frame is read or scanned. The second average (AVGD) is the average time to read an extent given that the head is positioned at the cylinder containing the prime frame to which the extent belongs. This is the average read time for extents when records are directly accessed. In general, AVGE is less than AVGD since global extents may occupy the same cylinder or cylinders which are close to one another. Therefore, it takes less time to fetch an extent in the extent chain once the head is positioned in the global extent area than it does to fetch the same extent from the prime frame. March does not distinguish between AVGE and AVGD but calculates one average, ATRES, which is a function of the time to access a local extent and the time to access a global extent. He assumes that the time to access a global extent is TRAN (also listed in Table 3). For these experiments TRAN is the observed value for all random accesses (including prime frames) over the steady state time interval.

TABLE 3

PREDICTED AND OBSERVED PERFORMANCE MEASURE VARIABLES

E#	OBS ANEXS	PRE ANEXS	TRAN	AVGE	AVGD	PRE ATRES
1	4.80	4.76	86.7	71.3	88.5	76.7
2	9.54	9.03	79.7	67.3	102.0	80.3
3	2.20	2.20	94.8	94.9	103.0	95.5
4	4.89	4.76	80.8	67.7	91.6	72.1
5	5.58	4.76	86.3	71.4	91.7	77.4
6	9.54	9.03	87.5	81.8	97.9	88.2

Column headers are: (all averages are state averages at steady state)

E#   - experiment number

PRE   - abbreviation for predicted

OBS   - abbreviation for observed

ANEXS - the average number of extents per frame

TRAN   - the average time to do a random access

AVGE - the average time to read an extent from an extent chain

AVGD - the average time to read an extent from a prime frame

ATRES - overall average time to read an extent

VTK   - the probability that a record is in an extent

In most cases the discrepancies can be explained by observing the behavior of the variables which March used in the calculation of the performance measure under discussion. For this purpose, graphs have been provided which map the behavior FVTK (FVTKS is the value of FVTK at steady state) (Figure 9) AVGE and AVGD in Figure 10 and ANEX (ANEXS is the value of ANEX at steady state) in Figure 11.

FFPHY fared so poorly in experiment 3 primarily because of the behavior of the variable ANEX as shown in Figure 11. The rapid acquisition of extents at the beginning of the experiment introduced much more degradation than at that stage than predicted by March's analysis. In our experiments, the first update to a frame always causes an extent to be allocated. Hence, at the beginning of experiment 3 the chances of an extent being allocated is very great. Once a frame gets an extent, however, it does not need another until four additions have been made to it since each extent can contain four records. The same phenomenon can be observed in the ANEX curve for experiment 1. It is not as pronounced since each extent can contain only two records.

The variable ATRES also affects the behavior of the performance measure FFPHY. As mentioned earlier, AVGE is the average time to read an extent when extents are fetched sequentially in chains. Therefore, the use of AVGE in the equation for FFPHY is more accurate than the use of ATRES. The AVGE curve (Figure 10) for experiment 3 indicates that the value of AVGE remains fairly constant and is close to AVGD (the

8-23

average time to fetch an extent from a prime frame). This is because extent chains tend to be short (since each extent can hold four records) and hence most of the time to fetch a chain is represented in the fetch of the first extent which is a direct access of an extent (AVGD). Under these circumstances the theoretical value of ATRES will be close to AVGE and will be a good estimation of it (compare AVGE and AVGD in Table 3). Since ATRES is accurate it must be ANEXS which causes the inaccuracy of the predictions of FFPHY in experiment 3.

The AVGE curve for experiment 1 is more interesting. Here there is a sharp increase in the average time to read an extent at the beginning of the experiment. This increase is due to the fact that local extents are allocated at the beginning and these can be accessed quickly. As the local extent areas become full, global extents are allocated and the average time to read an extent increases. March's ATRES variable does not capture this behavior and tends to be higher than AVGE (much higher at the beginning of the experiment).

The AVGE curve for experiment 2 shows a decrease in the average time to read an extent as the experiment progresses. This is due to the fact that no local extents are available and as the extent chains grow longer (and they will since extents can contain only one record) the time to move from one global extent to another in the chain begins to have a greater affect on the average time to read an extent. In spite of this decrease in the value of AVGE, the value of FFPHY is predicted closely in experiment 2. This is because initially ATRES is

close to AVGE but becomes a poorer estimate as time passes. But the degradation at the beginning of the experiment has a greater effect on an interval average than degradation that occurs later. Hence, March's analysis predicts the early state average degradation well for experiment 2 and this tends to compensate for later poorer predictions when the interval average is calculated. In contrast, for experiment 1 the predicted early degradation is high and this makes all predicted interval averages high.

The AVGE curves for experiments 4 and 5 exhibit the same behavior as the one for experiment 1. However, the average error in the predicted values for FFPHY for experiments 4 and 5 is lower than the error in experiment 1. This can be explained as follows. For experiments 4 and 5 the local extents are dissipated much more rapidly than in experiment 1. Experiment 4 concentrates most updates on only half the prime frame cylinders, filling the local extent areas for these cylinders quickly. In experiment 5, early overhead overflow causes the allocation of extra extents. Hence the period of low degradation at the beginning of the experiment is shorter and doesn't have as much of an effect on the interval average.

For experiments 1, 4, and 5 the behavior of the AVGE variable is most responsible for the inaccurate predictions of FFPHY. This is particularly evident in experiment 5 where the addition of extents to handle overhead overflow actually improve the prediction of FFPHY (discussed above). Compare this to the effect that uneven allocation had in experiment 3

where ANEX was close to the analytic value throughout the experiment.

The following insights can be gleaned from the preceding discussion:

1. if AVGE is close to AVGD then March's ATRES will be a good estimate of the average time to read an extent and the behavior of the ANEX variable will determine the average prediction error. This occurs when many records can fit in an extent;

2. if AVGE and AVGD differ greatly then ATRES won't be a good estimate of the average time to read an extent and the effect of uneven and unexpected extent allocation will be reduced. This occurs when extents can contain only a few records;

3. if degradation is predicted accurately at the beginning of the experiment then the average percent of error will be less than if early degradation is poorly predicted.

The analysis of the behavior of the FRTOK performance measure is much simpler. Most of the discrepancy between the predicted and observed values appears to be caused by the difference between AVGD and ATRES. Experiment 2, which predicted FRTOK the worst (10.48 percent average error), was the one for which there was the largest percent of error between AVGD and ATRES at steady state (see Table 3). Experiment 4 also had a large error between AVGD and ATRES but this was compensated for somewhat by the fact that the predicted

8-26

value of VTK was slightly lower than the observed value.

The fact that AVGD is always greater than ATRES is compatible with the fact that the observed value of FRTOK is always greater than the predicted value.

VTK, the proportion of records in extents, is a secondary cause of error in the value of FRTOK. In the absence of any overhead induced extents, it is equal to the function -

$$f(nadd) = nadd/(16200 + nadd)$$

where nadd is the number of additions and 16200 is the number of records initially loaded. This is not linear but is nearly so. Records that get pushed out of the prime frame by expansion will raise the proportion of records in extents but not enought to seriously effect the calculation of FRTOK. For instance, the value of FRTOK in experiment 5 (overhead induced overflow) has a 7.59 percent average error whereas experiment 1 (very little overhead induced overflow) has a 4.44 percent average error.

Finally, a comment is made on the ability of March's analysis to predict the performance of a frame memory which is used concurrently by two or more users. The frame memory used to test the effect of more than one user had no local extents since the advantage of local extents is lost when the head may move before the local extent is accessed. In our experiment, additional users did not affect the number and distribution of extents, they only affected the time to fetch an extent. Since only global extents were used in the experiment, the average time to access an extent (AVGD or AVGE) was close to the average

8-27

time to do a random access (see Table 4.3). Since March has TRAN as one of the parameters to his equations and assumes the ATRES = TRAN if there are no local extents, his predictions were not adversely affected by a multiuser environment in experiment 6.

Conclusion

Given the assumptions made, March's analysis appears to predict the performance of a frame memory satisfactorily (i.e., within 10 percent) for the use for which it was designed - as a tool to aid in the development of automatic data base design systems. Furthermore, his predictors are robust since we altered several of the assumptions and still observed satisfactory results from his analysis. This robustness is achieved mainly by the judicious choice of one of the parameters of the analysis. This parameter, TRAN, is the average time to do a random access in the frame memory. Unlike the other parameters used, TRAN is far from obvious and its determination involves an insight into the performance characteristcs of the frame memory whose performance is being analyzed.

The frame memories which failed to perform as predicted by March's analysis were those which represented less than optimal designs. One of them used small extents but had no local extent areas thereby forcing any addition to the frame memory to be stored in the global extent area. Another one used large

extents resulting in a lot of unavailable free space and also had no local extent areas. The frame memory which performed best had medium sized extents and a local extent area.

March's analysis falters in the following situations:

1. If large extents are used, non-linear extent allocation results, leading to non-linear degradation. March assumes linear degradation.

2. When extents are in the global area, March's overall estimate of the average time to read an extent is incorrect. This occurs since during a frame read or frame scan many extents in the global extent area are close together thereby reducing the average time to fetch the next extent.

3. The expansion of indices can push records out of the prime frame into extents. This phenomenon produces a large number of unexpected extents if the overhead per record is greater than 10 percent. However, March's analysis produced satisfactory results when the overhead per record was approximately 10 percent.

In summary, the more extents a frame has the more inaccurate is March's estimate of the average time to read an extent. The number of extents can be reduced only by making extents larger, thereby causing a non-linear pattern of extent allocations. This has been shown to invalidate March's assumption of linear degradation. This situation is not as hopeless as it might sound. A reasonably simple set of heuristics could be developed to assure that the decisions

affecting these areas result in good data base designs which will be predicted accurately by March's frame memory analysis.

Directions for Further Research

March has demonstrated the usefulness of viewing the update induced change of performance of a data base over time as the sum of the measure of performance of the data base at load time and a time related measure of performance degradation. His model of frame memory fits this approach very well. Prime frames represent initial or non-degraded performance and extents represent degradation of performance. His assumption of linear degradation was shown to cause problems. This assumption would not be necessary if a non-linear degradation function were developed for each performance measure. These functions could be based on time and the number of records per extent. Even greater precision could be achieved if the effects of overhead expansion were considered and the assumption that the size of records was large compared to the overhead they require could be dropped. A more rigorous analysis than the one we performed would be necessary for the devlopment of the non-linear degradation functions.

March's analysis could be made more complete if it didn't have to rely so heavily on the average time to do a random access. Ideally, what would be supplied as a parameter is the function which describes the time to move the disk head a given number of cylinders. The average time to do a random access could then be calculated as part of the analysis.

A question which arises is whether more sophisticated data structures (e.g., B-trees, differential files, etc.) are amenable to analysis in terms of initial performance and some time related degradation of performance. If so, the structures could be classified according to their degradation functions.

Finally, an investigation could be made into transporting these ideas to fields other than data base management. In particular, the field of software quality measurement is interested in the degradation of performance (and quality) of programs caused by code changes. The work presented here has offered a model of change related degradation. Can this model be adapted so that is provides a useful way of analyzing program quality?

Secondary Memory

FIGURE 1 A FRAME MEMORY

Frame Reference Stream   - - - - - ➤

Frame Record Tokens   T1
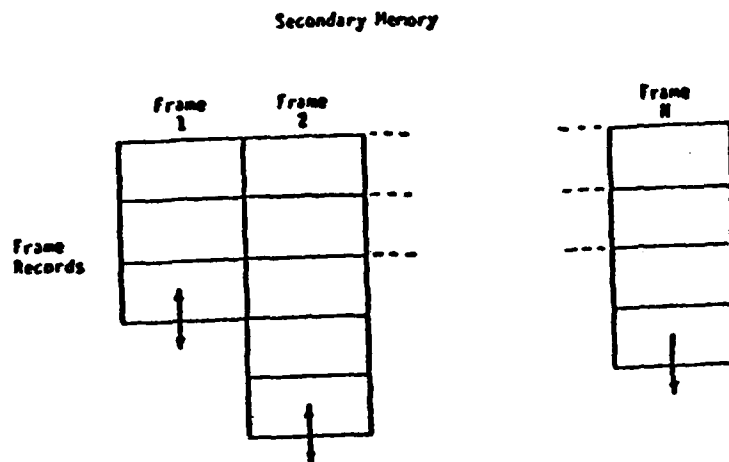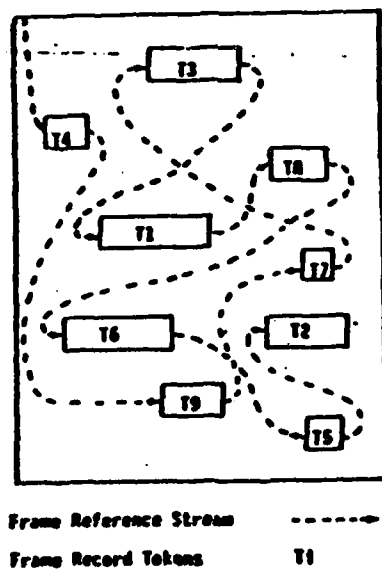
FIGURE 2   FRAME RECORDS WITHIN A FRAME

Frame
Records

Frame 1
"Front"
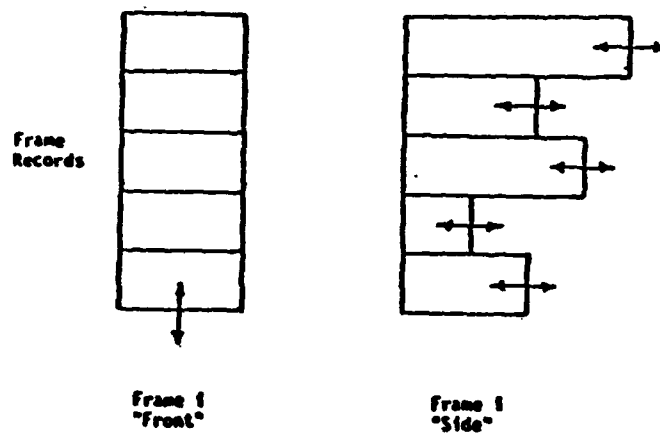
Frame 1
"Side"

FIGURE   3   FRAME RECORDS
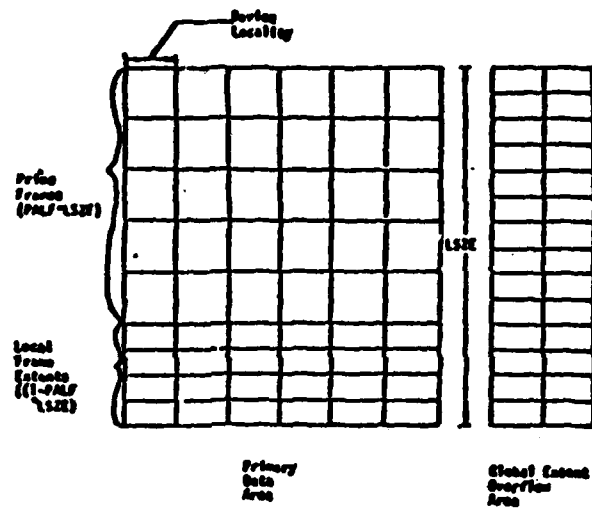
FIGURE   4   RELATION BETWEEN FRAME COMPONENTS AND STORAGE
DEVICE LOCALITIES
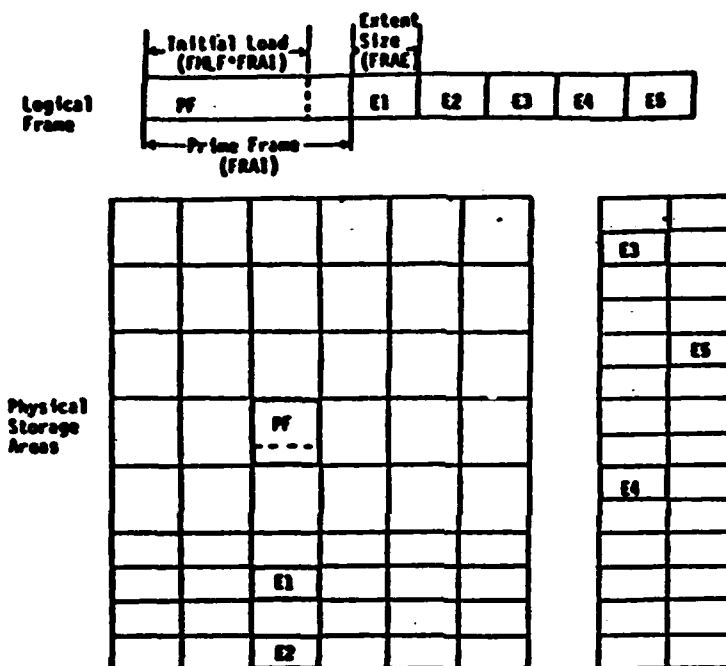
FIGURE 5   FRAME MAPPING TO PHYSICAL STORAGE AREAS

RANDOM ACCESS FUNCTION
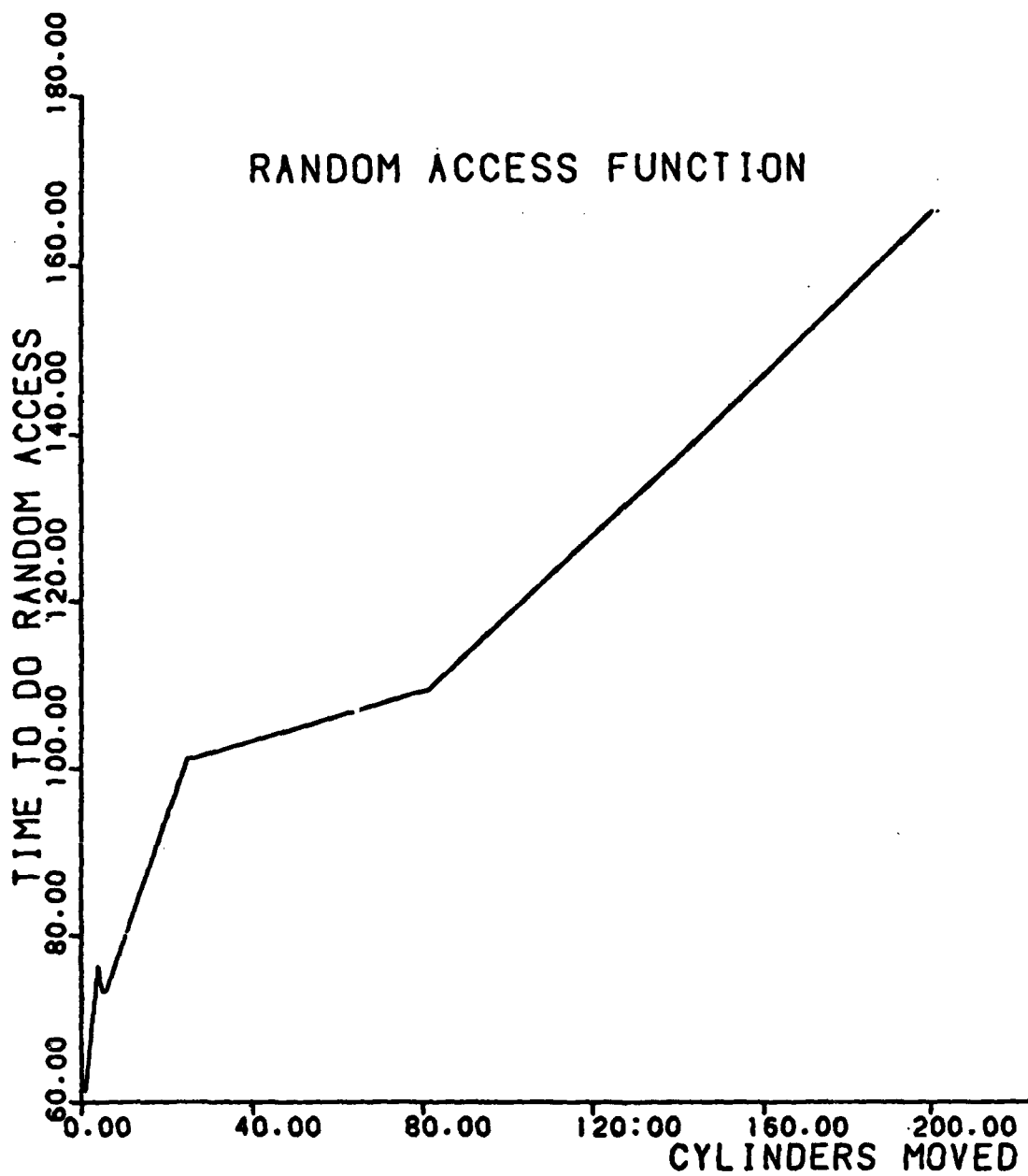
FIGURE 6

I ENDS OBSERVED CURVE.

X AXIS IS INTERVAL TIME. Y AXIS IS VALUE OF FFPHY
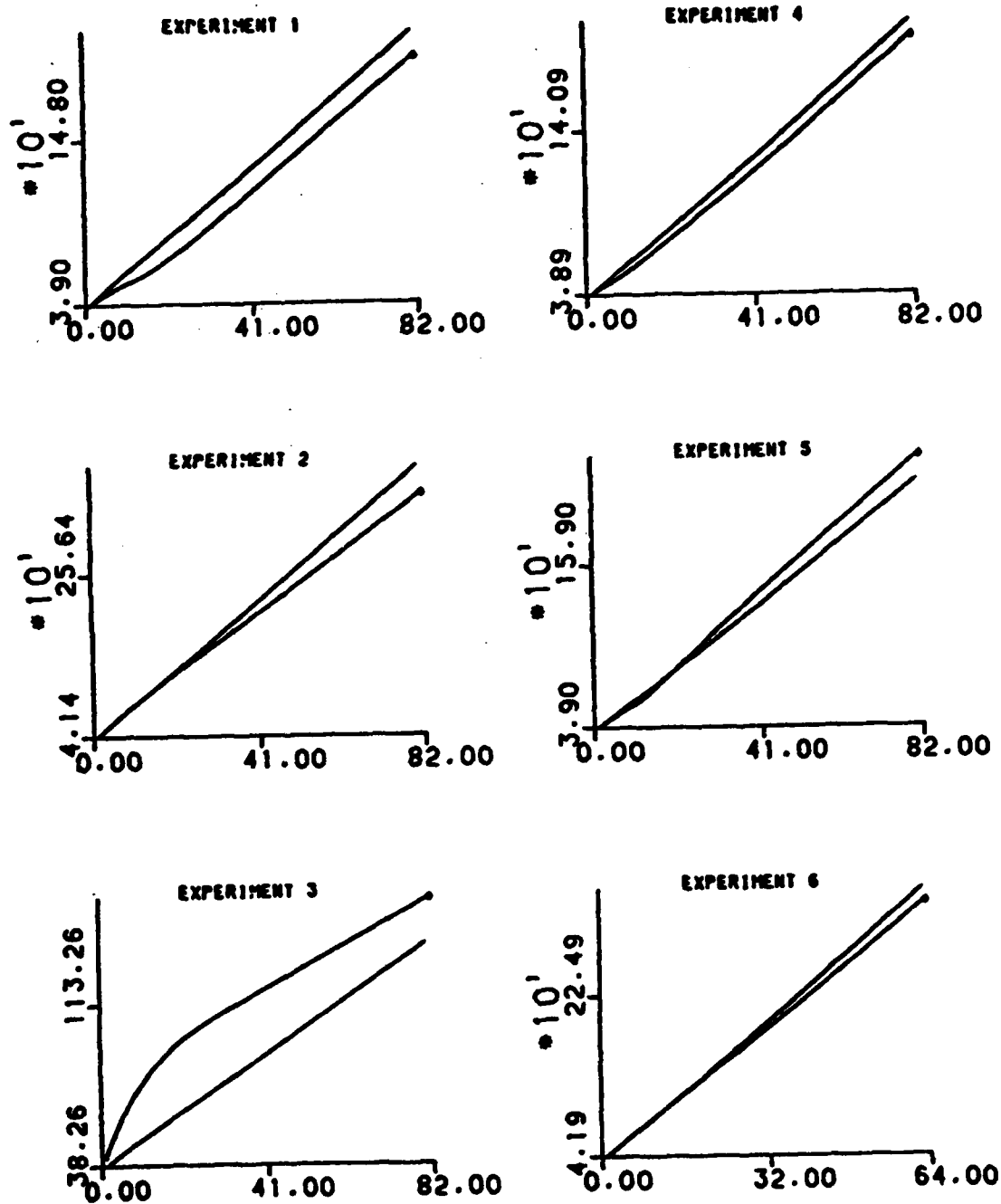
FIGURE 7 OBSERVED VS ANALYTIC INTERVAL AVERAGES FOR FFPHY

- ENDS OBSERVED CURVE.
- X AXIS IS INTERVAL TIME. Y AXIS IS VALUE OF FRYOK
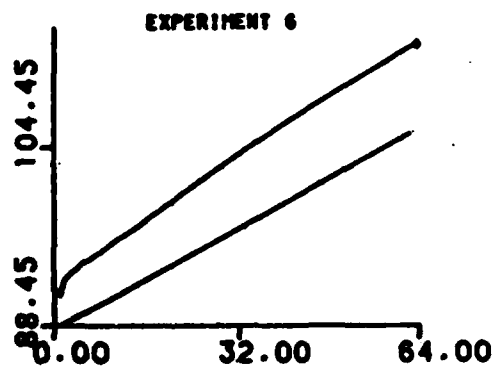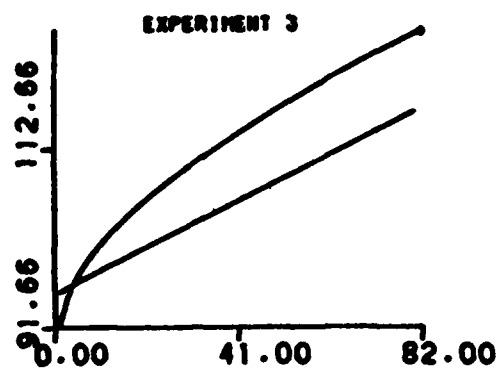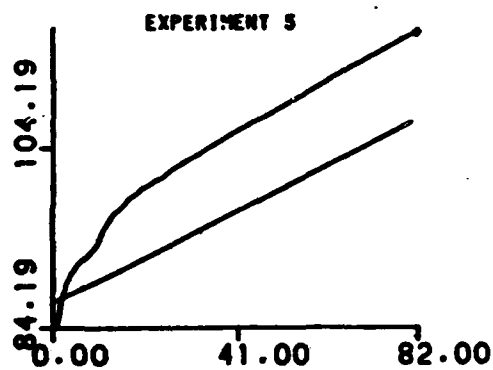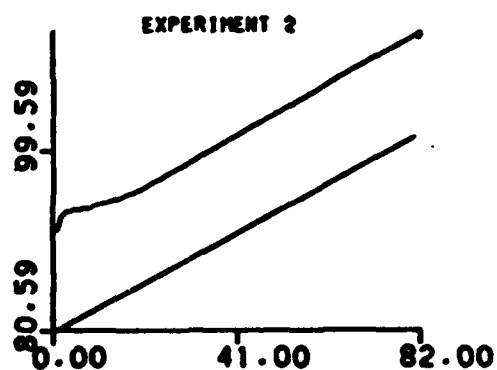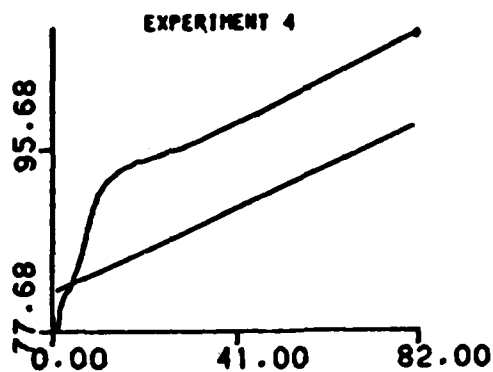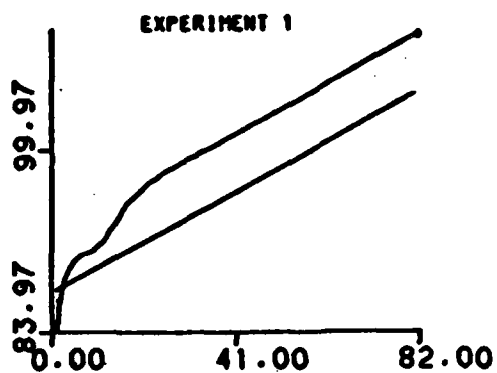
FIGURE 8 OBSERVED VS ANALYTIC INTERVAL AVERAGES FOR FRYOK
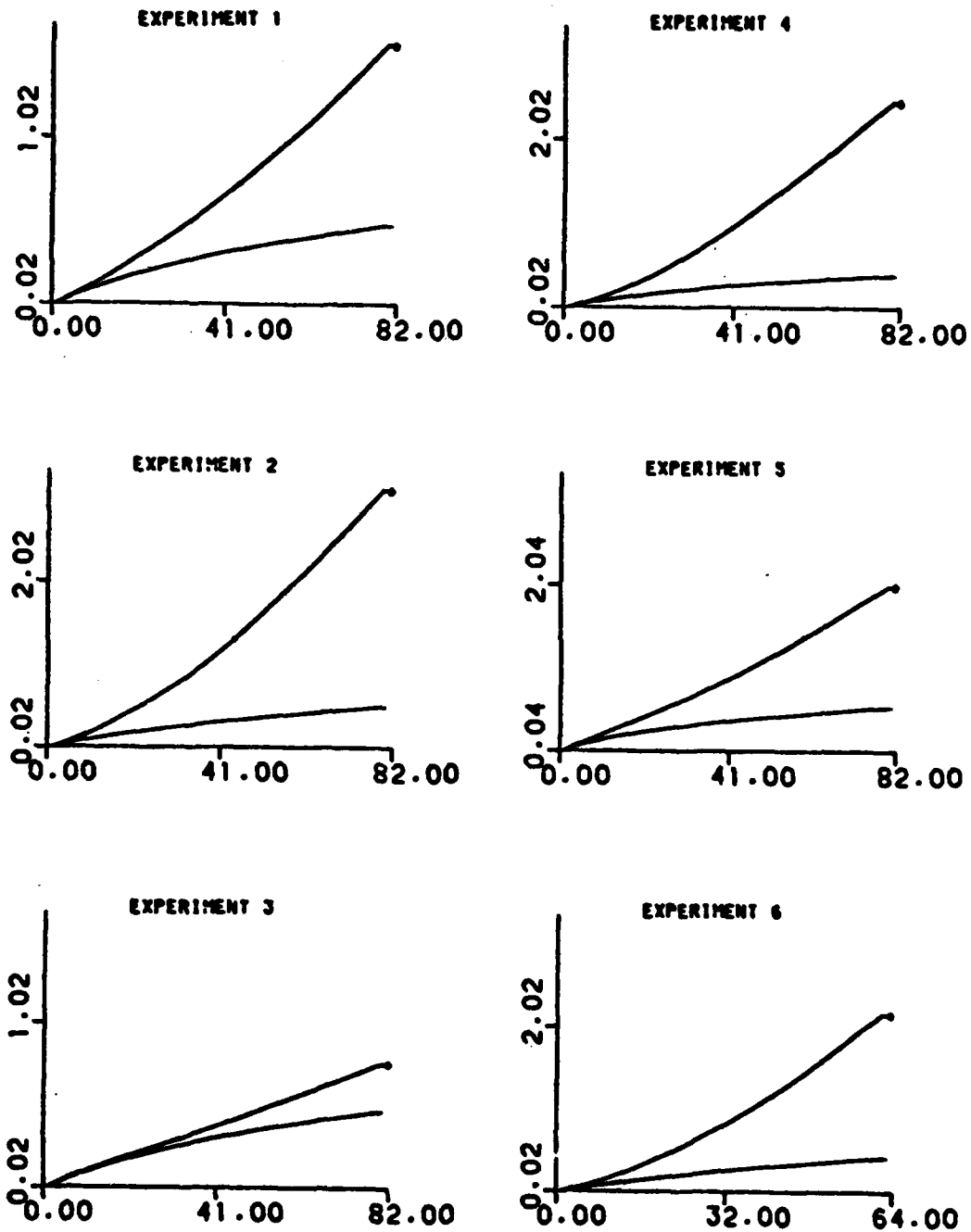
FIGURE 9   OBSERVED VALUES FOR VSCN AND FVTK

• ENDS AVGD CURVE.

X AXIS IS INTERVAL TIME. Y AXIS IS AVERAGE TIME TO READ AN EXTENT

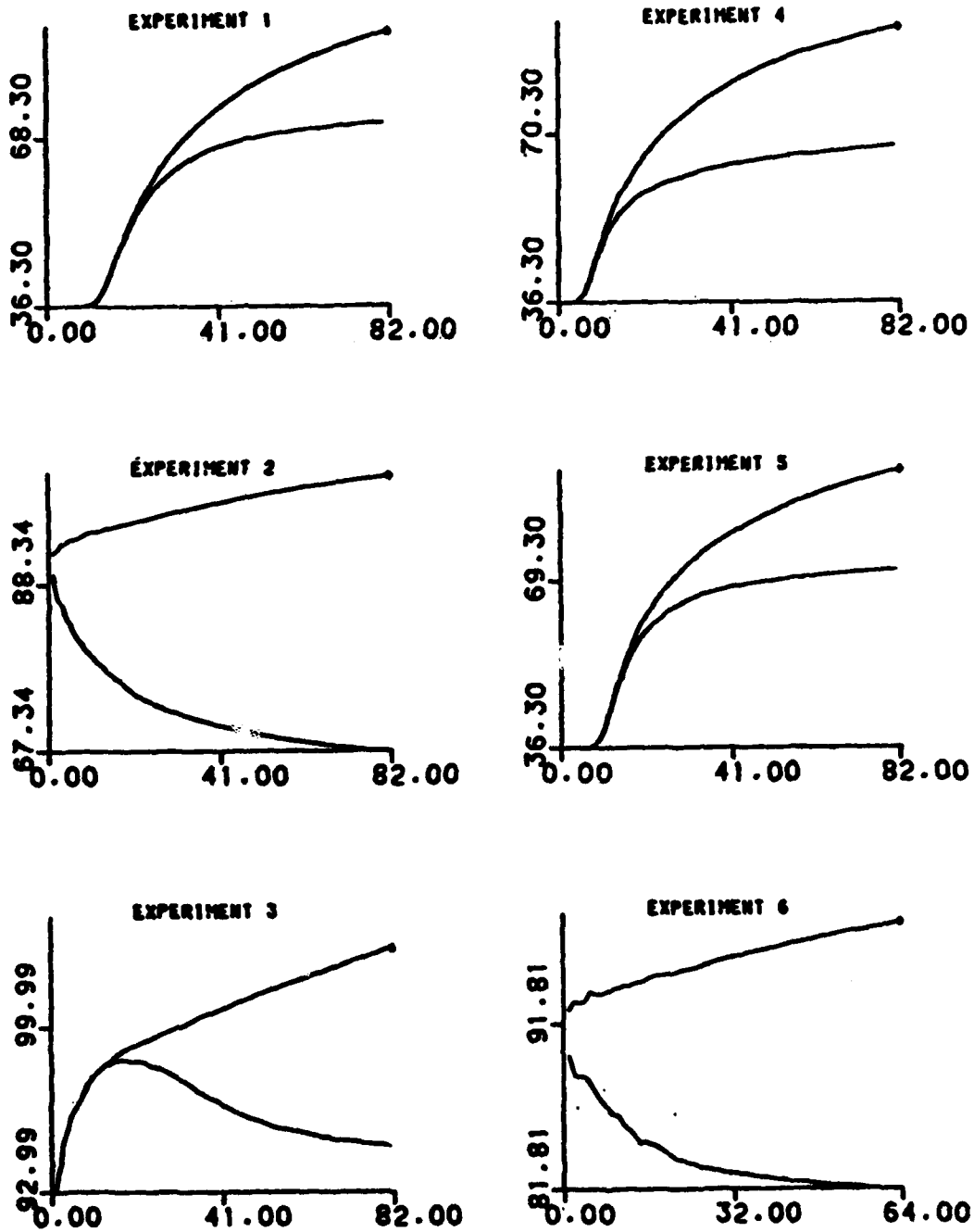FIGURE 10 OBSERVED VALUES FOR AVGD AND AVGE
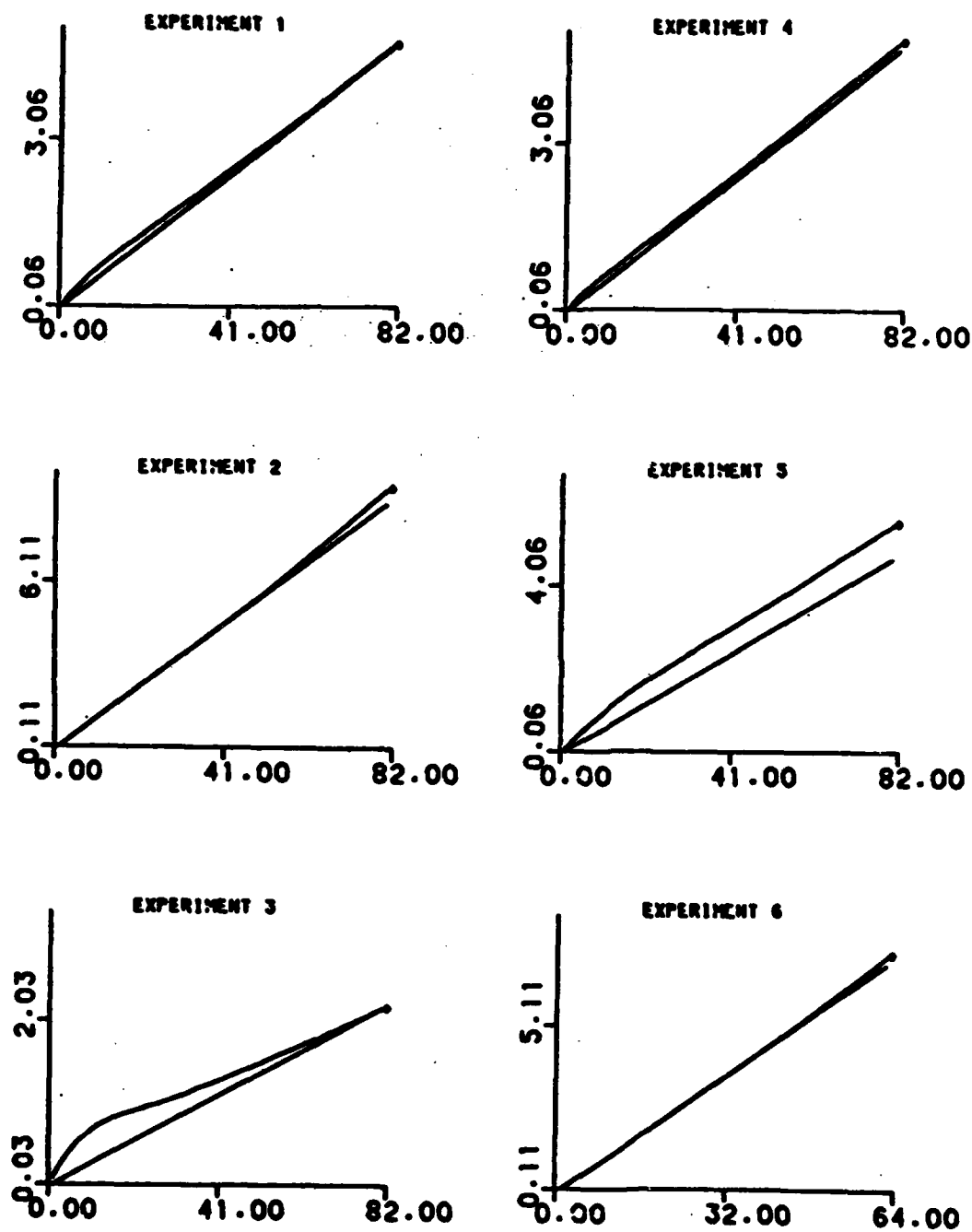
8-40.

FIGURE 11 OBSERVED VS ANALYTIC VALUES OF ANEX

REFERENCES

MAR77   March, S. T., Severance, D. G., "The Determination of
        Efficient Segmentation and Blocking Factors for
        Shared Data Files. ACM Transactions on Database
        Systems 2, 3 (September 1977), 279-296.

MAR78   March, S. T., "Models of Storage Structures and the
        Design of Database Records Based Upon a User
        Characterization", Ph.D. Dissertation, University
        of Minnesota, 1978.

STO76   Stonebraker, M., Wong, E., Kreps, P., "The Design and
        Implementation of INGRES", ACM Transactions on
        Database Systems 1, 3 (September 1976), 189-222.