Systematic Analysis of Algorithms

Lyle A. Anderson III
University of Rhode Island

SYSTEMATIC ANALYSIS OF ALGORITHMS

BY

LYLE A. ANDERSON, III

A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE OF MASTER OF SCIENCE IN COMPUTER SCIENCE

UNIVERSITY OF RHODE ISLAND

1981

MASTER OF SCIENCE THESIS

OF

LYLE ALLEN ANDERSON, III

Approved:

Thesis Committee

Major Professor Edmund A. Lamagna

[illegible]

[illegible]

[illegible]

Dean of the Graduate School

UNIVERSITY OF RHODE ISLAND

1981

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

Throughout the research and writing of this thesis, I have been indebted to my major professor, Dr. Edmund A. Lamagna, for his guidance, encouragement, and support.

The research reported in this thesis was supported by the U.S. Air Force Systems Command, Rome Air Development Center, under contract F30602-79-C-0124. Additional support was provided by Aquidneck Data Corporation.

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. Gödel'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. 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 complete formal system can be constructed which can analyze all algorithms. This fact is firmly established 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 branching probabilities. Wegbreit [2] developed a formal system 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 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-ofmagnitude 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 flow-chart 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 in-troduced 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.

In the examples which follow, this notation applies:

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

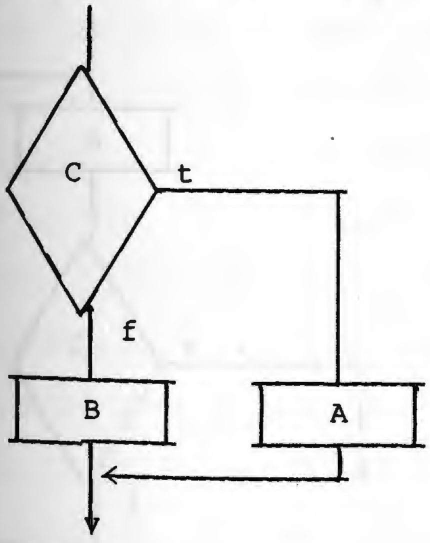
T is a prefix that indicates that the quantity is an execution time for a program block or element (Examples are TA, TC_f)

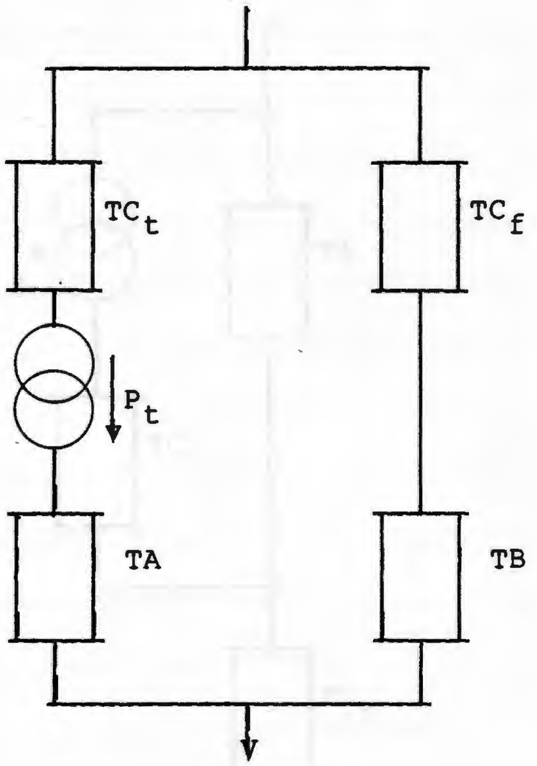
n is the number of executions of a loop body

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

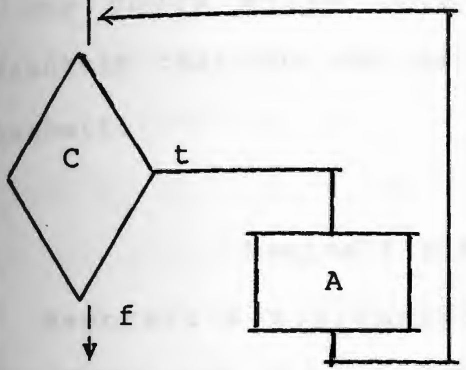
The structured programming constructs involving closed flowchart loops are translated as follows:

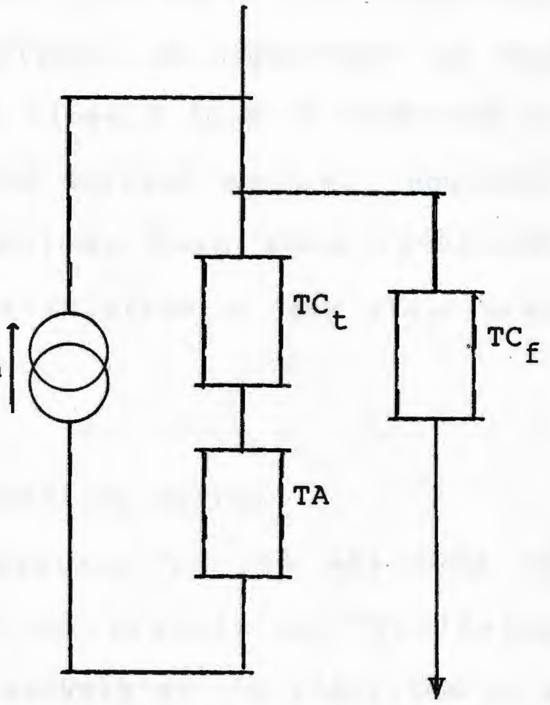
- if-then-else is equivalent to a single statement block with a value of $P_t(TC_t+TA) + (1-P_t)(TC_f+TB)$



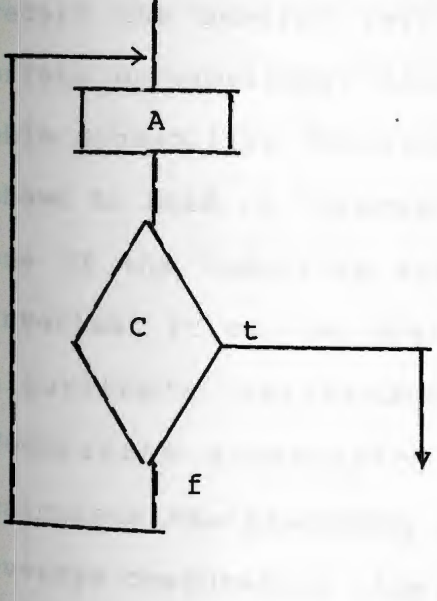


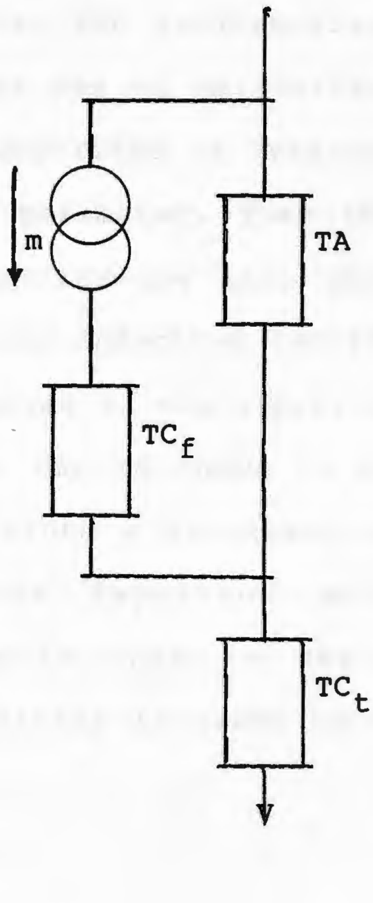
- do-while is equivalent to a single statement block with a value of $n(TC_t + TA) + TC_f$





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





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, $0 \leq e \leq 1$. 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.,

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

The output assertion which one would expect to get is:

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

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

$$\Pr([K=1] \vee [K=2]) = 1$$

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:

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

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

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≠0) = 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:

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

which is logically consistent, but

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

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

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

which is a valid assertion.

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

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

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:

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

This statement contains the logical contradiction:

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

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.

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 recurrence 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 n^{th} 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 coefficients 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.

First we consider the **FOR** loop.

Given the input syntax:

```
 <label>:     for <var> <-- <exp1> to <exp2> by <exp3> do
     <statements with live variables>
     repeat
```

we get the recursive syntax:

```
     start <-- <exp1>; stop <-- <exp2>; incr <-- <exp3>
     <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 $RANDOM_{ht}$ = T do
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 $RANDOM_{ht}$ = T then
        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:

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

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 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 n^{th} 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.

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 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 <u>probability set function</u>, $P_X(A)$, 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 <u>probability density function</u> (p.d.f.), $f_X(x)$:

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

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 <u>probability density function</u>, $f_X(x)$:

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

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.

<u>Theorem 1:</u>

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 Anderson 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 Anderson delta maps the Boolean space into the numbers 0 and 1.

Definition

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

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

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 Anderson 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 Anderson delta $\delta(R)$ we have the augmented algorithm:

```
            { $f_X(x)$ }
if R
  then      { $f_X(x) \cdot \delta(R)$ }
       nothing
  else      { $f_X(x) \cdot \delta(\neg R)$ }
       nothing
endif       { $g_X(x) = f_X(x)\delta(R) + f_X(x)\delta(\neg R)$ }
            { $g_X(x) = f_X(x) \cdot (\delta(R) + \delta(\neg R))$ }
            { $g_X(x) = f_X(x)$ }
```

Q.E.D.

So far, this discussion of the joint p.d.f. of the program variables is very close to Ramshaw's [5] frequentistic states. In fact, we can show that Ramshaw's frequentistic assertions can be derived from marginal or joint p.d.f.s. As we have said before, where 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.

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

```
if K=0 then    { $\delta(k=0)(\frac{1}{2}\delta(k=0)+\frac{1}{2}\delta(k=1))$ }
               { $\frac{1}{2}\delta(k=0)$ }
    K <- K+2   { $\frac{1}{2}\delta((k-2)=0)$ }
               { $\frac{1}{2}\delta(k=2)$ }
[ else ]       { $\delta(k\neq0)(\frac{1}{2}\delta(k=0)+\frac{1}{2}\delta(k=1))$ }
               { $\frac{1}{2}\delta(k=1)$ }
endif          { $\frac{1}{2}\delta(k=2) + \frac{1}{2}\delta(k=1)$ }
```

Which is exactly what we should get.

In handling the assignment statement, K <- 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 <- RANDOM$_{ht}$
   if X=Y then { S$_t$ } else { S$_f$ } endif.
```

<u>Theorem 3</u>

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
```

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 <- 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 <- 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:

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

$$\sum_{x_i} \left(f(x_i)\ \delta(x_i = x_j) \right) = f(x_j)$$

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

where $F(x_j) = \Pr(X \in A) \mid A = \{ X \le x_j \}$ 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.

<u>Theorem 5</u>

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:

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

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

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

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

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

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 :

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

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                 { $f_I(i) = \delta(i=0)$ }
    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        { $f_I(i) \cdot (\frac{1}{2}\delta(y=H) + \frac{1}{2}\delta(y=T))$ }
    if Y = T then         { $f_I(i) \cdot \frac{1}{2}\delta(y=T)$ }
        print 'OK, so far!'
        I <- I + 1        { $f_I(i-1) \cdot \frac{1}{2}\delta(y=T)$ }
        call TC(I)        { $g_I'(i)$ }
    end if
                          { $g_I'(i) + f_I(i) \cdot \frac{1}{2}\delta(y=H)$ }
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:

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

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:

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

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:

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

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 distirbution 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 $(\frac{1}{2})^j\delta(i=j)\frac{1}{2}\delta(y=H)$, this joins the rest at the end if to give the final answer:

$$g(i) = \tfrac{1}{2} \sum_j (\tfrac{1}{2})^j \delta(i=j),\ j \in \{\ 0,\ 1,\ 2, \ldots .\ \}$$

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

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

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

by distributing and regrouping each fraction we get:

$$= \tfrac{1}{2}(\ \tfrac{1}{2} + \tfrac{2}{4} + \tfrac{3}{8} + \tfrac{4}{16} + \ldots\ldots\ldots\ldots$$

$$= \tfrac{1}{2}(\ \tfrac{1}{2} + \tfrac{1}{4} + \ldots + \tfrac{1}{4} + \tfrac{1}{8} + \ldots + \tfrac{1}{8} + \tfrac{1}{16} + \ldots$$

$$= \tfrac{1}{2}(\ 1 + \tfrac{1}{2} + \tfrac{1}{4} + \ldots\ldots\ldots$$

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

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

$C \leftarrow 0;$

loop $X \leftarrow RANDOM_{ht}$; $C \leftarrow C + 1$; while X=T repeat

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

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

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

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

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              { $\delta(c=0)\ \delta(i=2)$ }
   M <- $RANDOM_f$             { $\delta(c=0)\ \delta(i=2)\ f(m)$ }
   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 $I \leq N$ then
                               { $h(n,m,c,i)\ \delta(i\leq n)$ }
      T <- $RANDOM_f$          { $h(n,m,c,i)\ \delta(i\leq n)\ f(t)$ }
      if T>M then              { $h(n,m,c,i)\ \delta(i\leq n)\ f(t)\ \delta(t>m)$ }
         C <- C + 1            { $h(n,m,c-1,i)\delta(i\leq n)f(t)\delta(t>m)$ }
         M <- T

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

```
    end if
                    { $\delta(m=t)\ (\sum_m h(n,m,c-1,i)\delta(t>m))\ \delta(i\le n)f(t)$
                              $+ h(n,m,c,i)\ \delta(i\le n)\ f(t)\ \delta(t\le m)$ }
    I <- I + 1
                 {$\delta(i-1\le n)(\delta(m=t)(\sum_m h(n,m,c-1,i-1)\delta(t>m))f(t)$
                              $+ h(n,m,c,i-1)\ f(t)\ \delta(t\le m)$ ) }
    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:

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

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

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

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, 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.

<u>"Oblivious" Insertion Sort</u>

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 continues 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 comparators (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 $I \geq 1$ 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

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

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,

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

After 6a,

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

After 7,

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

After 8a,

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

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

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

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

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

and the initial condition

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

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

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

There is an easy way to improve the relative performance of the "oblivious" insertion sort, although the order of its running time remains the same. We note from the analysis that the portion of the joint p.d.f. that fails the test at statement 5, is already in sorted order. This suggests that we could exit from the INNER loop at this point without affecting the algorithm's ability to sort. Even such "obvious" improvements often have hidden side effects. Luckily our method will let us not only calculate the improvement in performance from this change, but also 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).

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:

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

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 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+1th 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

$$[\ \delta(i=0)\ \{\ \delta(x<k_1) + \delta(x>k_1)\delta(x<k_2) + \ldots\ldots + \delta(x>k_n)\ \} +$$
$$\sum_{mid=1}^{n} (\ \delta(i=mid)\ \delta(x=k_{mid}))\ ]$$
$$\cdot g(x)\cdot\delta(k_1<k_2)\cdot\delta(k_2<k_3)\cdots\delta(k_{n-1}<k_n)\cdot f(k_1)\cdot f(k_2)\cdots f(k_n)$$

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.

```
procedure TRIVIAL ( N )
/*  Load the initial tree */
        X ← $random_f$; Y ← $random_f$
        if ( X < Y ) then
                <S;V,W> ← <G;X,Y>
        else
                <S;V,W> ← <F;Y,X>
        endif
/*  The main algorithm loop */
        for K ← 1 to N
/*  Insert a key */
                R ← $random_f$
                case
                : S = F and R < V     : <T;X,Y,Z> ← <A;R,V,W>
                : S = F and V < R < W : <T;X,Y,Z> ← <B;V,R,W>
                : S = F and W < R     : <T;X,Y,Z> ← <C;V,W,R>
                : S = G and R < V     : <T;X,Y,Z> ← <C;R,V,W>
                : S = G and V < R < W : <T;X,Y,Z> ← <D;V,R,W>
                : S = G and W < R     : <T;X,Y,Z> ← <E;V,W,R>
                end
/*  Now do the deletion */
                L ← $random_{XYZ}$
                case
                : T = A and L = X :  <S;V,W> ← <F;Y,Z>
                : T = A and L = Y :  <S;V,W> ← <F;X,Z>
                : T = A and L = Z :  <S;V,W> ← <F;X,Y>
                : T = B and L = X :  <S;V,W> ← <F;Y,Z>
                : T = B and L = Y :  <S;V,W> ← <F;X,Z>
                : 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 $\leq$ 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>
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

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

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

After 20

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

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 Anderson 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.

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

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

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 Anderson delta function. This function, by 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.

Armed with an automatic processor, work can go forward to handle 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.

Bibliography

1. Aho,A., Hopcroft,J. and Ullman,J., The Design and Analysis of Computer Algorithms. Addison-Wesley, Reading, Massachusetts, 1976.
2. Anderson, R.B., Proving Programs Correct. John Wiley & Sons, New York, 1979.
3. Cohen, J. and Roth, M., "On the Implementation of Strassen's Fast Multiplication Algorithm", Acta Informatica, Vol.6 (1976), 341-355.
4. Cohen, J. and Zuckerman, C. "Two Languages for Estimating Program Efficiency", Comm. ACM, Vol.17, No.6 (June 1974), 301-308.
5. Davies, A.C., "The Analogy Between Electrical Networks and Flowcharts", IEEE Trans. Software Eng., Vol. SE-6, No. 4 (July 1980), 391-394.
6. 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.
7. Hibbard, T.N., "Some Combinatorial Properties of Certain Trees with Applications to Searching and Sorting", J. ACM, Vol. 9 (1962), 13-28.
8. Hofstadter, D.R., Gödel, Escher, Bach: an Eternal Golden Braid. Basic Books, New York, 1979.
9. Hogg, R.V. and Craig, A.T., Introduction to Mathematical Statistics (Second Edition). Macmillian, New York, 1959.
10. Horowitz, E. and Sahni, S., Fundamentals of Computer Algorithms. Computer Science Press, Potomac, Maryland, 1978.

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

12. 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.

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

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

15. Lueker, G.S., "Some Techniques for Solving Recurrences", Computing Surveys, Vol.12, No.4 (December 1980), 419-436.

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

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

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

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

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

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 $\geq$ 1 then
5             if B(I) > B(I+1) then
6                         EXCHANGE ( B(I), B(I+1) )
7             endif
8a            I $\leftarrow$ 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.

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

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:

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

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:

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

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

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

7 At the join for the if statement we have

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

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.

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.