1

Algorithms and Computers

1.1 Introduction

Discussing algorithms before computers emphasizes the point that algorithms are valuable mathematical constructs in themselves and exist even in the absence of computers. An *algorithm* is a list of instructions for the completion of a task. The best examples of algorithms in everyday life are cooking recipes, which specify how a list of ingredients are to be manipulated to produce a desired result. For example, consider the somewhat trivial recipe for cooking a three-minute egg. The recipe (or algorithm) for such a task exemplifies the need to take nothing for granted in the list of instructions.

Algorithm Three-Minute Egg
Put water in a pan.
Turn on the heat.
When the water boils, flip over the egg timer.
When the timer has run out, turn off the heat.
Pour some cold water in the pan to cool the water.
Remove egg.

Although this algorithm may appear trivial to most readers, detailed examination further emphasizes (or belabors) how clear and unambiguous an algorithm must be. First, the receiver of these instructions, call it the actor, must recognize all of the jargon of food preparation: water, pan, egg, egg timer, boil, and so forth. The actor must recognize constructions: "put ____ in ____"; "when ____, do ____." Some parts of these instructions are unnecessary: "to cool the water." If the actor is an adult who understands English, this may be a fine algorithm. To continue to belabor the point, the actor can only do what the instructions say. If the instructions failed to tell the actor to put an egg in the pan (as I did in writing this), then the actor may be stuck trying in vain to complete the last instruction.

The actor must also interpret some instructions differently in different environments. Even if the actor is working in a standard household, "Turn on the heat" has the same intent – but different implementations – for gas versus electric ranges. Moreover, in some environments, the heat may never be sufficient to boil the water, or be so hot that the water boils away in less than three minutes, and so the actor following this

algorithm will fail to complete the task. Lastly, if the actor is not an adult, then the instructions must be much more specific and must reflect foresight for contingencies that a parent can imagine a child to encounter. This precision and foresight become even more important when considering instructions to be given to a machine.

Knuth (1997) listed five properties that algorithms must possess:

(1) finiteness – execution can be done with finite resources;
(2) definiteness – instructions are completely defined and unambiguous;
(3) input;
(4) output;
(5) effective – the instructions can be executed (my words).

To gain a general view of algorithms, consider that every statistical procedure must also abide by these requirements. Of course, the input is the data and the output the decision, test result, or estimator. But the steps of a statistical procedure must be as clearly defined, finite, and effective as any algorithm. If the desired estimator is the sample median, then the definition of the estimator must clearly state that if the sample size is an even number then the median is defined to be the average of the two middle values. How do you define a maximum likelihood estimator when the algorithm to compute it fails with some positive probability? What are the properties of such an estimator, such as consistency, efficiency, or mean square error? A statistical procedure is often clearly defined only by the algorithm used to compute it.

Most effective algorithms are repetitive: do these instructions and when you're done go back and do them again. The route of the instructions executed inspired the name *loop,* and the list of instructions to be completed is the *range* of the loop.

Algorithm Scramble(n)
Do these statements n times:
  take an egg out of the refrigerator;
  crack the egg's shell on the edge of the counter;
  pull the egg apart above the bowl;
  let the contents of the egg fall into the bowl;
  throw the egg shell in the waste basket.
Stir egg contents in the bowl.
Pour contents of bowl into frying pan.
⋮

In this algorithm, the range of the loop consists of five instructions. Notice that the number of repetitions is specified by n as an argument to the algorithm.

Iterative algorithms may use a condition to control the number of times a loop is to be executed.

Algorithm Cup-of-Water
⋮
Do these statements until it's exact:

if the level is above the line then pour a little water out;
if the level is below the line then pour a little water in.
⋮

Iterative algorithms are common and effective, but they hold the potential for danger. In this case, an actor with great eyesight and poor hand coordination may never be able to get the water level exactly on the line. If the condition is impossible to satisfy (in this case, "exact" may not be possible) then the algorithm may repeat indefinitely, becoming stuck in the black hole of computing known as an *infinite loop.* Good programming practice requires that – unless it is computationally provable that the algorithm is finite – all iterative algorithms should have a stated limit on the number of repetitions.

One personal experience may emphasize the need for care in constructing iterative algorithms. In constructing a fast algorithm for the Hodges–Lehmann location estimator (see Section 14.4), I used the average of the maximum and minimum of a set of real numbers to partition the set. Even if the set was weird and if this average separated only one number from the rest, the partitioning would eventually succeed at cutting the size of the set by one each time and eventually down to one element and finish. To my surprise, for one set of data the algorithm never finished because the algorithm was stuck in an infinite loop. Working on a computer with base-16 arithmetic, it is possible (see Exercise 2.9) that the computed average of max and min will be smaller than every number in the set. The set was never made smaller and so continued until I intervened to stop the program.

Some algorithms use a version of themselves as a subtask to complete their work and are called *recursive* algorithms. Consider the following algorithm for computing factorials.

Algorithm Factorial(n)
If $n = 1$ then Factorial $= 1$
else Factorial $= n *$ Factorial$(n - 1)$

Recursive algorithms are extraordinarily effective in many problems, and some of the great breakthroughs (FFT, sorting; see Chapter 14) in computing rely on recursion. Although some software and computer languages do not permit explicit recursion, clever coding can often implement a simply stated recursive algorithm.

1.2 Computers

Many of the ideas in statistical computing originated in an era where a computer was a person, but the world of scientific computing now looks quite different. The visible equipment, or hardware, usually consists of a processor box, video screen, keyboard, mouse, and connections to other devices (e.g., a printer or modem). In spite of tremendous changes in computing equipment over the last thirty years, the underlying model for computing remains unchanged. The model for computing for this book consists of a central processing unit (CPU), memory, mass storage, and input and output devices. The CPU is a collection of semiconductor devices that controls the unit. Memory,

usually referred to as RAM (random access memory), is so named because the time to access (write or read) any part of it does not depend on any order – a random order takes as long as a consistent sequence. Mass storage refers to any source of slower memory, which may be an internal disk drive, a floppy disk, magnetic tape, or a disk drive on the other side of the world connected by the Internet. These memory devices are slower to access and are designed to be accessed sequentially, at least in the small scale. Input and output devices may refer to the tangible ones – such as the keyboard, mouse, or printer – but could also include mass storage devices.

This general-purpose computer can only earn its name through the software designed to run it. At its furthest abstraction, software is a list of instructions to operate the machinery of a computer. The most fundamental software for a computer is its operating system. Other software – such as word processors, spreadsheets, games, compilers, and, yes, statistical software – are written in a very general framework and are designed to work within the framework that the operating system provides. Whereas a user may give instructions to a spreadsheet to add one number to another and put the sum in a third place, the software will give instructions that may look like the following:

> get the first number from memory location x and put it here;
> get the second number from memory location y and put it there;
> add the two numbers stored in here and there, leaving the sum here;
> store the number here to the memory location z.

The software's instructions are, in their executable form, machine-level instructions such as fetch, store, and add, but the storage locations are only relative. The operating system specifies the physical locations for x, y, and z, manages the list of instructions, and coordinates the input and output devices.

Computers operate most efficiently with both the data and instructions residing in memory. But some software, including the operating system, is very complicated and takes an enormous amount of space to store it. Some problems, such as simulating the weather system of the earth, have so much data that no computer has enough memory to hold all of it. These problems are managed by the operating system through *paging*. While the fundamental tools of the operating system reside permanently in memory, other pieces of the operating system, other pieces of the software, and other pieces of data reside in mass storage, usually an internal disk. As these pieces are needed, the operating system swaps space with other pieces, writing pieces residing in memory in mass storage, making room for the needed pieces in memory. Imagine a person doing calculations on a desk that doesn't have enough room for all of the pieces of paper. When this Computor needs more space, he takes some sheets of paper off the desk to make room and puts them in a file cabinet. When a sheet in the file cabinet is needed, he takes something else off the desk and exchanges sheets of paper from the file cabinet to the desk. This method allows incredibly large problems to be done with finite memory, at the cost of slower computation due to the swapping. More memory can allow some problems to run much faster by reducing the swapping, as if the Computor's desk were made much bigger. Poorly written software can aggravate the situation. Problems with enormous amounts of data improperly stored may require a swap for every computation; "spaghetti code" software can continually shift code around instead of

keeping some core routines constantly resident in memory and using compartmentalized auxiliary routines.

1.3 Software and Computer Languages

Most computer software is written for completing a specific task. For example, as complicated as the U.S. income tax laws may be, tax preparation software can satisfy the needs of the vast majority of citizens every April. To satisfy the statistical needs of all scientists, however, the range of tools available extends from very specific software to general computer languages. Many standard office products, such as word processors and accounting software, include tools for simple statistical analysis, such as t-tests or simple linear regression. These tools are embedded in the main product and are limited in their scope. A step more general are spreadsheets – themselves more general than, say, accounting software in that they are designed to do a variety of mathematical tasks – with which the user has complete control over the arrangement of the calculations. At the far end of the spectrum are computer languages. Although most computer languages are designed to solve a great variety of problems, some languages are better at some tasks than others.

To assess the appropriate software needs in statistics, consider first the mathematical tools of statistics. Statistical theory may rely on calculus for maximizing likelihoods or computing posterior probabilities, but most statistical methods can be well explained using linear algebra. Clearly, the main computational needs in statistics are implementing the tools of linear algebra. On the simple side, these are sums and sums of squares for computing t-tests and doing simple linear regression. The more complex needs then extend to solving systems of linear equations in positive definite matrices. Derivatives are occasionally needed, but numerical approximations will suffice for most applications. Similarly, integration is sometimes required, but often numerical approximations are the only route. Linear algebra encompasses most of the mathematical needs for statistical applications. Even though some mathematical software can do calculus through symbolic manipulation, most calculus needs in statistics can be implemented numerically.

In short, any software that can do sums and inner products can do a certain level of statistical computing. Even sophisticated statistical techniques can be implemented in spreadsheets. Some statistical software lies at this level of sophistication for the user: the data take the form of vectors or rows or columns, and the more sophisticated tools operate on these. Indeed, if most of the mathematics of statistics can be written in linear algebra, then most statistical computing can be done with software designed to manipulate vectors and matrices. Of statistical software, Minitab and SAS's IML operate with vectors and matrices, and have special ("canned") routines to do more sophisticated mathematical and statistical analysis beyond the view (and control) of the user. An early computer language designed to do linear algebra is APL, which was once quite popular in spite of its use of special symbols; its legacy can be seen in IML and R. SAS's `DATA` step also can perform many of the required linear algebraic manipulations and then tie into canned routines for more sophisticated analyses. Both R and S-Plus are designed to do most linear algebraic manipulations using native operations, with a few

special canned routines and a structure (objects) for doing increasingly sophisticated manipulations. At all points along the way, there is a compromise between generality and control in one direction and a canned, special-purpose routine in the other.

Computer languages are needed to gain the full advantage of the "general" in general-purpose computing. These languages resemble human languages and consist of a set of instructions with a functional syntax – perhaps not a complete one, but then few human languages have one either. The grammar of subject-predicate-object works due to the recognizability of the structure through a complete categorizing or *typing* of the pieces. In deciphering spoken or written language we can recognize the predicate as the action word in a sequence of words, and (in English) the actor precedes the action as the subject and the object succeeds the action. Recognizing the parts of speech (nouns, adjectives, adverbs, etc.) limits their function in a sentence. The order of the words describes their function; for example, the adjectives "quick" and "brown" describe the noun and subject "fox" in the following sentence:

The quick brown fox jumped over the lazy dog.

Computer languages work in much the same way: (a) words are typed; (b) the type of a word limits its possible functions; and (c) the structure of the sentence determines the meaning. In comparison with human languages, computer languages must be more rigid in typing and structure, since a computer does not have the intelligence of a human. In using computer languages, an important skill is to recognize how the computer "thinks." Using certain rules, the computer interprets the input by *parsing* the text (or *code*) to determine the "parts of speech" for the language. The structure of the text then further clarifies the list of commands. Many languages parse text into four types: operators (such as $+, *$), constants (such as 2.7, 64), reserved words, and variables. Constructs such as "put ____ in ____" would have "put" and "in" as *reserved words* (i.e., text with special meaning). Often everything that is not an operator, a constant, or a reserved word is considered to be the name of a variable.

Once the text has been parsed, computer languages follow two routes for execution: interpreted or compiled. Interpreted code is set up for immediate execution. If the operator is (say) "norm" and the operand is a vector, then the command is to compute the norm of the accompanying vector. In further detail, the vector is stored as data, and the interpreted code passes the vector's pointer to a list of instructions that computes the norm. However, in some computer languages the code is first *compiled,* or translated into a single body of instructions. The difference between the two methods is a trade-off of effort. Interpreted code is intended to be executed just once for the particular arrangement of operator and operand. For example, we may compute the norm of a particular vector just once. Compiled code is designed to be used many times. The language itself dictates which route will be taken. In general, languages with more sophisticated operators will be interpreted whereas more general-purpose languages, which operate with lower-level commands, will be compiled.

The hierarchy of languages puts the array of software into perspective. At the base level of all computing is machine-language instructions. Only the basic steps of computing are written in machine language: usually just the basic commands to start the computer. At the next level are assembler commands. These are usually in a language with commands that directly translate to machine commands. At the next step

are the low-level general-purpose computer languages, such as Pascal, C, Fortran, Lisp, and Cobol. Writing at this level gains access to certain machine-level operations, but the code can be written in a style that avoids most of the repetitive steps that assembler-language level would require. The gains with using these languages are (a) the great generality that is available and (b) the ability to rewrite tasks in terms of subtasks whose code can be accessed in a wide variety of other tasks. For example, a routine can be written to compute the norm of a vector that can be used by a variety of other routines. Once compiled, the code need never be written again and is simply accessed by its name. The strength of these languages is this foundation of subtasks, each one solidly written and tested, so that extensive structures can rest on these building blocks. These languages are quite general, and the same task can be written in any of them. The reason for the variety of languages is a variety of purpose; each language has its own features or constructs that make certain tasks easier.

At the next level of languages, the operands available are much more sophisticated and the details are hidden from the user. Notice the clear trade-off between power and control: more powerful constructs will limit the control of the user. If the user wants the usual Euclidean norm of a vector, then using a higher-level language saves time and worry about coding it correctly. However, if the user wants the $p = 4$ norm of a vector then the lower-level language will be needed. The higher-level languages are often interpreted, with the specific operands actually accessing the compiled code of a routine written in a lower-level language. For example, R and most of SAS are written in C.

Higher-level interpreted languages (e.g., R, SAS's IML, and GAUSS) have some strong advantages. Primarily, the languages are designed to operate with the same basic mathematical tools common in statistics: vectors and matrices. As a result, the code resembles familiar mathematical analysis and is easily read and understood without extensive documentation. Moreover, the user is seldom bothered by the details of the more sophisticated operations: solving linear equations, computing eigenvalues, and so on. This removes a strong distraction if these tools are soundly implemented, but it creates severe problems if they are not or if they are pushed to their limits. The disadvantages follow from their one-time interpretive nature. These languages often do not have a natural looping structure, and if looping is possible then the structures are clumsy. The one-time operation leads to the use of extensive dynamic storage allocation – that is, in the middle of computation, more storage is needed on a temporary basis. The problem of creating and using this temporary storage has been solved, yet often this temporary storage cannot be fully recovered and reused. This problem is known as "garbage collection," and the failure to solve it adequately leads to a continual demand for memory. As a result, these languages eat up memory and sometimes grind to a halt while trying to find more space.

As mentioned previously, many general-purpose computer languages have been written, often designed for specific purposes. Over the years, Fortran has been the dominant language for scientific computation. The strength of general-purpose languages lies in the formation of building blocks of subtasks, and Fortran does this particularly well. Fortran has often been considered an inferior language with byzantine syntax, but recent (1990, 1995) changes in the standard have cast off its worse parts and added a few really useful structures. The revised language now permits strong typing of variables; that is,

all variables must have their type (integer, real, character) declared. This avoids silly errors such as misspellings and requires more disciplined coding. The use of local and global scoping, subprogram interfaces, and variable dimension declarations obviates any need for poor programming practices long associated with the language, such as spaghetti coding and circumlocutions for passing arguments to subprograms.

One of the more recent advances in software design is object-oriented programming. Instead of the traditional paradigm of flow charts, the focus is on *objects,* which are fundamentally containers for data with attendant software (for operating on the data) and communication tools (see e.g. Priestly 1997). In statistical applications, objects are primarily collections of data. For example, in a regression scenario an object may consist of the response vector $\mathbf{y}$ and design matrix $\mathbf{X}$, as well as least-squares estimates $\hat{\boldsymbol{\beta}}$ and residuals $\hat{\mathbf{e}}$.

1.4 Data Structures

Although most data in statistics are stored as vectors or matrices, a brief introduction to some more sophisticated structures – together with some details on the more common ones – provides some perspectives on the potential tools for problem solving. The fundamental data structure is a *linear list,* which is natural for storing a vector. Internally, a vector $\mathbf{x}$ is merely referred to by a pointer to the storage location of its first element. Any element of the vector, say x_j, is found by adding $(j-1)$ to the pointer for $\mathbf{x}$. Matrices (and any higher-dimensional arrays) are still stored as a linear list. In some computer languages (e.g. Fortran), if the matrix $\mathbf{A}$ has dimensions m and n, then its pointer is the location of A_{11} and the element A_{ij} is stored $(j-1)*n+(i-1)$ locations away. As a result, A_{11} is next to A_{21}, which is next to $A_{31}, \ldots, A_{m1}$; then A_{12} through to A_{m2} and then A_{13}. As you traverse the list of elements, the leftmost index varies fastest. Higher-dimensional arrays follow the same rule. In other languages (e.g. C), the reverse convention is followed, with the order $A_{11}, A_{12}, \ldots, A_{1n}, A_{21}$, etc.

One type of linear list permits changes, additions, or deletions of elements only at the end of the list or at the top of a "stack." The most common scientific application of stacks is for counting in nested situations. For example, if we have j nested within i (say, days within months), then j or days are at the top of the stack and only *after* the cycling through j is completed do we drop to the lower level (i or months) and make a change there. Stacks are commonly used in computer systems to handle similar situations: nested loops or nested calls to subprograms.

Most data take the form of vectors, matrices, and other arrays; however, two other structures are occasionally useful in scientific programming: the linked list and the binary tree. Each element of a linked list consists of the body and pointers. The elements of a linked list are not stored in consecutive locations. The body of each element holds the data and, in the simplest case of a singly linked list, the pointer holds the address of the next element of the list. As a result, traversing a linked list requires going through the list from the front to the back – there is no way of knowing where the jth element is stored without traversing the first, second, third, to the $(j-1)$th element. So what can be gained? The biggest gain is the ability to add or delete an element from the list without moving the body of information stored in that element of the list; all that needs to be done is to change the pointer. For example, in order to delete

the third element, replace the pointer to "next" in the second element with the pointer to the fourth element (stored in "next" of the third element). To add a new element between the second and third, have "next" in the second point to the new one with the new one's "next" pointing to the third element. To overcome some of the problems in traversing the list, other pointers can be added. A doubly linked list has pointers both to the next and previous elements.

Linked lists are more commonly used in commercial applications for databases, sorting, and information retrieval. Moreover, they also permit simple dynamic storage allocation and garbage collection. Creating space means deleting elements from a linked list called FREESPACE, and garbage collection means adding elements to FREESPACE.

In a binary tree, elements are linked to pairs in a top-down fashion that resembles an upside-down tree. The first element, usually named the *root,* is at the top of the tree and is linked to two other elements or *children* (or the right and left child). These elements are then parents to the next generation of children. A special form of binary tree known as a "heap" is used in Chapter 14 for sorting. Other forms of a binary tree arise from time to time. A binary tree can be implemented in a fashion similar to a linked list, as long as the tree is traversed from top to bottom. Certain types of balanced trees can be implemented as linear lists.

1.5 Programming Practice

For whatever purpose the reader will be using this book – whether devising new statistical procedures or delving deeper into how codes for certain algorithms work – the author has considerable advice to offer. After over thirty years of programming, in many languages at different levels and throughout many changes in the computing environment, it is clear that the pitfalls at the interface between human and computer have not changed.

The appropriate attitude toward programming should be a healthy skepticism. Confidence in the results of a new routine can grow only from its proving itself in small steps. The envelope of trust opens as more and more difficult problems are attempted and successfully completed. Only after considerable successful testing does confidence become an appropriate attitude.

All big programming problems must be broken down into small component tasks. Then each subtask is resolutely tested until the user is confident in having established the envelope of proven reliability for that subtask. The great advantage of higher-level languages is that many of these subtasks have been taken care of, and only the bigger picture remains. Nevertheless, when starting out the user should be skeptical even of these fundamental subtasks and test them also. Testing involves creating a battery of test problems that are appropriate for the range of usage. The first test problems should be simple ones: there's no need to waste time on subtle problems when the easy ones don't work. Subsequent test problems should include some troublesome cases to ensure that the program does the right thing when the problem is beyond its capabilities. In other words, the program should unmistakably fail on an impossible task. For example, can a routine for solving a system of linear equations recognize and properly react when the system is singular?

Another lesson learned over many years is KISS, or "keep it simple, stupid." The great value of simple programs with simple, unambiguous steps is that if a mistake is made then it will be a big one and easy to recognize. As a program becomes more complicated, it becomes more difficult to find the "bugs" or mistakes in the code. Unnecessary features that increase complexity and potential for error are known as "bells and whistles"; they should be avoided until the main part of the program has been thoroughly tested and debugged.

The hardest lesson for this author to learn has been *documenting* the program – and I am still learning. All programs should be thoroughly documented with comments. The need for the comments follows from breaking down the large task into subtasks. As each subtask is completed, it must be fully documented as well as tested. This allows the programmer to then forget the details of one subtask when working on another. In commercial settings, this permits a programming team to divide up the effort and work in parallel. But when an error is encountered in a poorly documented program, only the person who wrote that part of the code will know how it works and what went wrong. In the case of a single person writing the whole thing, this means that the person who wrote the code a month ago can't remember how it works. Or in my case, there's no way I could remember what I was thinking ten or twenty years ago when writing some poorly documented code. The only solution may be to start all over, with considerable waste of time and energy.

The final lesson learned after many years of experience is that if you are not certain of your results then you have absolutely nothing. Hence, the only way to be sure of the whole task is to be sure of each piece. In working on some joint research several years ago, I observed some simulation results that were contrary to a theorem I thought we had proved. My colleague kept telling me that the program was wrong. Thinking he was right and that I was at fault, I started examining more than a thousand lines of Fortran code, looking for errors. After two weeks of checking each piece, I confidently told my colleague that no, the program is right; I am sure of each and every piece; the theorem must be wrong. The point is that it is possible to confidently write large programs that are entirely correct. It does take a lot of careful, disciplined checking, but the reader should be undaunted.

1.6 Some Comments on R

In the last ten years, the software system R has spread throughout the statistics community. Due to its flexibility and the powerful nature of its functions, a great deal of recent statistical research has been done in R. To those for whom R is not a native language, some of its peculiarities may not be obvious at first glance.

R and its predecessors were not designed with efficient computation as a primary goal. The author has often heard the rationale that computers will always be getting faster and faster with more and more storage. As a result, often multiple copies of data are stored at one time, and certain operations have considerable overhead that would be unthinkable with other languages or software systems. Vector/matrix operations are natural and efficient in R; looping is not so natural, although recent versions of R show considerable improvement.