Outstanding Dissertations in the

COMPUTER SCIENCES

A Continuing Garland Research Series

Quicksort

Robert Sedgewick

Garland Publishing, Inc.
New York & London, 1980

Library of Congress Cataloging in Publication Data

Sedgewick, Robert, 1946–
Quicksort.

(Outstanding dissertations in the computer sciences)
"STAN-CS-75-492."
Reprint of the author's thesis, Stanford University, 1975.
Bibliography: p.
1. Sorting (Electronic computers) I. Title.
II. Series.
QA76.6.S44 1980 001.6'425 79-50821
ISBN 0-8240-4417-7

All volumes in this series are printed on acid-free, 250-year-life paper.

Printed in the United States of America

QUICKSORT

by Robert Sedgewick

Abstract

A complete study is presented of the best general purpose method for sorting by computer: C. A. R. Hoare's Quicksort algorithm. Special attention is paid to the methods of mathematical analysis which are used to demonstrate the practical utility of the algorithm. The most efficient known form of Quicksort is developed, and exact formulas are derived for the average, best case, and worst case running times. The merits of the many modifications which have been suggested to improve Quicksort are discussed, with an emphasis on their impact upon the analysis. Van Emden's method, samplesort, and the median-of-three modification are discussed in detail, and it is shown that the latter is the most effective improvement to Quicksort for practical sorting applications.

New results presented include: improvements to the algorithm based on a refined partitioning strategy and a new method of handling small subfiles, the best and worst case analysis, contrasting analyses of minor variants and the study of the effect of equal keys, new implementations of and new approaches to analyzing adaptive partitioning and samplesort, the complete general analysis of fixed sample size partitioning, and the application of "loop unwrapping" to Quicksort and the analysis of the optimized program.

The thesis is presented in an expository fashion so that it may be useful as a textbook in the field of "analysis of algorithms". It is self-contained, and it includes a complete treatment of a simpler sorting algorithm (insertion sorting) as well as three appendices which complement the material in the text.

This research was supported in part by The Fannie and John Hertz Foundation. The printing of this paper was supported in part by National Science Foundation grant GJ 36473X and by the Office of Naval Research contract NR 044-402.

Preface

The general field of mathematical analysis of algorithms has come to encompass two quite different kinds of analysis. One type, more commonly referred to as "computational complexity", involves the study of entire classes of algorithms to solve a particular problem, usually from within a fairly general framework. The other, called simply "analysis of algorithms", involves the study of the space and time requirements of particular algorithms and particular implementations of them. This kind of analysis has been popularized by D. E. Knuth in his series of books. The Art of Computer Programming. Both of these kinds of analysis are challenging, interesting, and useful, but they really are quite different. This thesis deals with the complete analysis of one algorithm, Quicksort. but the subject matter relates as much to the "analysis of algorithms" as to Quicksort itself.

Although much of what we know about the analysis of algorithms appears somewhere in Professor Knuth's books, there seems to be no elementary textbook which introduces this subject. For this reason, I have tried to write this thesis in an expository fashion, so that a newcomer to the field will be able to use it profitably for self-study. I have assumed some basic familiarity with both concrete mathematics and computer programming, but I hope that readers at various levels of sophistication in both of these fields will be able to benefit from this thesis. Analysis of algorithms really does require competence in both: The mathematical analysis often suggests how we might make our programs more efficient; and improvements to programs often make their analysis more interesting.

The algorithms in the text are all described in a programming language similar to Algol, but relatively few of the features of the language are used, so that they should be easily understood by anyone who has programmed in a high-level language. The programs use an exchange (:=:) operator, and the control constructs if ... then ... else ... endif and loop ... while ... repeat , which are like those described by Knuth in Computing Surveys 6 (December 1974). The assembly language programs in the Appendices are written in the MIX language defined in The Art of Computer Programming, and there is some comment in Appendix C about the implementation of the programs in some real programming languages for the IBM System 360 computer.

It is customary in a Ph.D. thesis to state explicitly which results are original and presented here for the first time. I have refrained from including such comments in the text for fear that they would detract from the expository nature of the thesis. To the best of my knowledge, the following major aspects of this dissertation are new:

(i) The technique of ignoring small subfiles during partitioning, then insertion sorting the entire array after partitioning (Program 2.4);

(ii) the analysis of the best case and the worst case of the algorithms (Chapter 4);

(iii) the contrasting analyses of the various different partitioning methods for the basic Quicksort algorithm (Chapter 5);

(iv) the study of the effect of equal keys (Chapter 5);

(v) "two-partition" Quicksort and its analysis (Chapter 5);

(vi) the careful implementation of van Emden's method and the proof that the subfiles are non-random (Chapter 6);

(vii) the analysis of the number of exchanges taken by samplesort, and the general method of analysis which can be applied to the other quantities (Chapter 7);

(viii) the idea of leaving the sample elements out of the partitioning process in the median-of-three method (Program 8.1);

(ix) the extension of the analysis of fixed-sample size partitioning to the general case, which leads to the proof that using the median is best (Chapter 8); and

(x) the application of "loop unwrapping" to Quicksort, and the analysis of the optimized program (Appendix A).

I have tried to tell the complete story of the analysis of Quicksort; indeed, I feel that this in itself is a major contribution of this thesis. This has necessitated including many "well-known" results. The origin and history of this research is discussed in Chapter 9. Many of the results appear in The Art of Computer Programming, and, where possible, I have tried to maintain consistency with Knuth's notation.

I hope that the example given here of the complete analysis of an important computer algorithm will help to stimulate similar intensive studies on the performance of other algorithms. This is not to suggest that we should submit *every* algorithm that we use to such a broad analysis, but I would hope that our most *important* algorithms could be analyzed as completely. I can envision a series of monographs similar to this on the analysis of, say, the ten most important computer algorithms. Of course, we must first decide which algorithms belong on this list; and we may not be able to analyze some of them very well. But Quicksort certainly belongs; and its analysis is exhibited here.

Acknowledgments

This thesis originated in Don Knuth's feeling that "there's still one good thesis left in Quicksort". Professor Knuth has contributed to the project in many ways: by encouraging my interest in the subject of analysis of algorithms through his courses and books, by patiently monitoring and directing my research with countless helpful suggestions, and by carefully reading and annotating early versions of the manuscript. Thanks are due also to Peter Wegner and Forest Baskett for reading the final copy of the thesis. The professional and consistent style of the manuscript is due entirely to Phyllis Winkler, who can add one more title to the impressive list of manuscripts that she has produced.

Financial assistance was provided by the Fannie and John Hertz Foundation. This generous fellowship provided support when none other was available to me, and allowed me to pursue this research on a full-time basis.

Finally, on a personal level, I must thank my Parents, whose example encouraged me to begin my work towards a Ph.D., and Don Knuth, whose example inspired me to complete it. Most of all, thanks are due to my wife Linda, who has shared these years at Stanford with me. We have made the "successful completion of the requirements for the Ph.D. degree" not an arduous task, but a great Western adventure.

Table of Contents

List of Programs

CHAPTER ONE

Quicksort is a sorting method suitable for use on computers which was introduced by C. A. R. Hoare in 1960. Like most good algorithms, it is based on an inherently simple idea, and it remains today the best general purpose sorting method for computers. However, this simplicity is deceiving, and the algorithm has many hidden subtleties. A variety of modifications have also been suggested for the purpose of improving the performance of the algorithm. The purpose of this thesis is to expose the subtleties and study the variations of Quicksort through mathematical analysis. The analysis leads to many important methods of general interest, so that in the process we will learn as much about the analysis of algorithms and concrete mathematics as about Quicksort itself

We will be analyzing computer programs written for the purpose of rearranging a given set of "keys" $A[1], A[2], \ldots, A[N]$ to make

$$A[1] \leq A[2] \leq A[3] \leq \cdots \leq A[N] \quad .$$

The order relation may be numeric order, alphabetic order, or any transitive relation whatever, which is defined on all the keys (so that exactly one of the possibilities $A[i] < A[j]$ or $A[i] > A[j]$ or $A[i] = A[j]$ holds for all i and j). Straightforward extensions will allow these programs to handle the more practical situation where more information is associated with each key.

Many sorting methods can be characterized by the fact that they only involve a few fundamental operations on the keys, such as "comparisons" or "exchanges". It is tempting to measure the efficiency of these methods by counting the frequency of such operations. However, this can often be misleading, because actual implementations of the algorithms involve overhead which can contribute significantly to the running time. The goal of our analyses will be the derivation of formulas (depending on N, the number of keys) for the total running times of the programs: in the best case, in the worst case, and on the average. Another temptation in working with such formulas is to discard all but the "leading" terms and to present only approximate formulas as a result. This can be very dangerous in a practical situation. (For sorting methods the leading term may be $\alpha N \lg N$ and the discarded term βN for some α, β. If, for example, $N < 10000$ and $\beta > 14\alpha$, then the "discarded" term would be larger than the "leading" term.) Although it is generally much more difficult to obtain, the derivation of an exact formula provides a far more accurate description of the operation of an algorithm.

The rest of this chapter will be devoted to a demonstration of this kind of analysis applied to a simpler sorting method, insertion sorting. This analysis not only will provide a full introduction to the analysis of Quicksort, but also it is very interesting in its own right. In addition, the results obtained will be of use to us later.

Insertion sorting is a natural sorting method which is commonly used, for example, by bridge players when putting their hands into order. The idea is to consider the elements one at a time, and insert each into position among the elements already considered. Program 1.1 is a slightly more formal description of this method.

Program 1.1

```
    i := 2;
    loop while i ≤ N:
        insert A[i] into position among A[1],A[2],...,A[i-1];
        i := i+1;
    repeat;
```

It is not difficult to convince ourselves that this program actually sorts the keys A[1],...,A[N] into nondecreasing order. It is easy to prove by induction that the loop preserves the condition

$$2 \le i \le N+1 \quad , \quad A[1] \le A[2] \le \dots \le A[i-1] \quad .$$

In general, we will not be concerned with formal proofs that the programs we deal with work, but rather with informal arguments which indicate that such proofs can be easily constructed.

In order to put the i-th key encountered, A[i] , into position, it is necessary to move all of the keys among A[1],...,A[i-1] which are greater than A[i] one position to the right. This leads to the following more explicit implementation of the algorithm.

Program 1.2

```
    i := 2;
    loop while i ≤ N:
        v := A[i]; j := i-1;
        loop while A[j] > v and j > 0:
            A[j+1] := A[j]; j := j-1;
        repeat;
        A[j+1] := v;
        i := i+1;
    repeat;
```

The operation of this program, on an arbitrarily chosen set of fifteen keys, is shown in Example 1.1. The arrangement of the keys is shown after each insertion: only those keys that were moved are written, so that the leftmost key on each line is the one just inserted. For example, the line marked with a * shows the situation just after the 08 key was inserted. The 26 , 44 , and 95 keys were moved to make room for it.

Since sorting programs are most clearly understood through the study of such examples, the operation of all the programs we will study will be demonstrated on the same fifteen keys as in Example 1.1. These keys come from the fractional part of the number lg e :

$$A[i] = \lfloor 100^i(\lg e - 1)\rfloor \bmod 100 \qquad \text{for } 1 \leq i \leq 15 .$$

Although Program 1.2 is a clear and correct implementation of insertion sorting, its efficiency can be improved by making two simple changes. First, the test " $j > 0$ " almost always fails: the only time it succeeds is when $A[i]$ is less than all of the keys $A[1],A[2],\ldots,A[i-1]$, and then only after they have all been scanned. A standard programming technique for eliminating such "almost always redundant" tests is to arrange things so that some other (necessary) part of the program catches the condition when it occurs. In Program 1.2, the test " $A[j] > v$ " will do the trick, if we simply make $A[0]$ smaller than all the other keys. (For this we write " $A[0] := -\infty$ ".) Then, if j becomes zero, the " $A[j] > v$ " test will stop the loop, and the " $j > 0$ " test is unnecessary. Although the test " $j > 0$ " doesn't involve a key and is usually not counted as a comparison, it does represent overhead which should be minimized. The second improvement to Program 1.2 follows from considering another special case: When $A[i]$ is larger than all of the keys $A[1],A[2],\ldots,A[i-1]$, the loop will never be executed, so there is no need to set up for it with the statements " $v := A[i]$ " and " $j := i-1$ ". These two improvements lead to a more efficient algorithm.

Example 1.1

sorting the file:

```
    44  26  95  04  08  88  96  34  07  35  99  24  68  10  01
    44
    26  44
            95
    04  26  44  95
  *     08  26  44  95
                    88  95
                            96
                34  44  88  95  96
        07  08  26  34  44  88  95  96
                        35  44  88  95  96
                                            99
                24  26  34  35  44  88  95  96  99
                                    68  88  95  96  99
                10  24  26  34  35  44  68  88  95  96  99
    01  04  07  08  10  24  26  34  35  44  68  88  95  96  99
```

Program 1.3

```
    i := 2; A[0] := -∞                                      1
    loop while i ≤ N:                                       N
         if A[i] < A[i-1] then                              N-1
              v := A[i]; j := i-1;                          D
              loop:                                         D+E
                   A[j+1] := A[j]; j := j-1;                E
              while A[j] > v;                               E
              repeat;                                       E
              A[j+1] := v;                                  D
         endif;                                             N-1
         i := i+1;                                          N-1
    repeat;                                                 N-1
```

This program sorts the keys exactly as in Example 1.1; it just does so more efficiently than Program 1.2. Our analysis will demonstrate this. To avoid confusion, we will normally try to develop the most efficient version of a program before analyzing it. The reader will then be able to look back over the various improvements and see the obvious justifications given by the analysis for them. (This is not how efficient algorithms are invented. An analysis must first be performed on some version of the algorithm to determine which improvements can do the most good.)

Our analysis of the time taken by this program begins by counting the number of times each statement is executed. These frequencies are given in the right hand column in Program 1.3. For some statements, i.e., " i := 2 ", we can deduce the frequency immediately. For others, the frequency analysis may be more complicated, although we may know that several statements have the same (unknown) frequency. These unknown quantities are generally related to characteristics of the algorithm, and it is usually helpful to think of them in such terms. Notice that the frequencies of the <u>loop</u> ... <u>repeat</u> statements can be deduced from the frequencies of the adjacent statements. In general we have

```
      :                          X
      :
    loop:                        X+Y
      :                          Y
      :
    repeat;                      Y
```

This is an example of the application of "Kirchhoff's law", which says that the number of times control flows into a statement must equal the number of times control flows out of it.

The running time of Program 1.3 depends on two unknown quantities:

I -- the number of insertions, and

E -- the number of keys moved during insertion.

The main objective is to investigate the "average" value of these quantities, though we will also look at other moments, as well as the maximum and minimum possible values.

Before we can begin this investigation, we must make some assumptions about the keys being sorted. In order to talk meaningfully about the "average" running time of the program, we need to have some concept of an "average" input file. If the N keys are all distinct, it is reasonable to assume that all of the N! permutations of the keys are equally likely to be the input file. This is the model that will be used in most of our analyses. If the input keys are not in this "random" order, or not necessarily all distinct, we will ensure that our programs run correctly, and we will try to make them run efficiently. However, our analyses will usually assume distinct, randomly order keys. (In Chapter 5 we do consider the effect of equal keys.) Now, Program 1.3 (and all of the other programs we will consider) makes its decisions based only on the relative order of the keys, never on their actual values. This means that we can further assume that the keys are the numbers $\{1,2,\ldots,N\}$.

Example 1.2 shows the values of D and E when the 4! permutations of $\{1,2,3,4\}$ are sorted by Program 1.3. Also given are the maximum and minimum values, as well as the average and variance, assuming that all 24 permutations are equally likely. The exhaustive treatment of a concrete example in this way is often a useful first step in analyzing such quantities. Not only does it provide us with a good intuition about the quantities, but also it provides a simple check on any more general answers we might derive. Examining Example 1.2, we notice that D and E both take on their minimum when the keys are already in order, and their maximum when the keys are in reverse order. In general, the more "out of order" the keys are, the higher the values of D and E .

To analyze these quantities, we need to be able to express more precisely the degree to which a permutation is "out of order". To this end, we will associate with each permutation $a_1 a_2 \ldots a_n$ of $\{1,2,\ldots,n\}$ an <u>inversion table</u> $B_1 B_2 \ldots B_n$, where B_i is defined to be the number of elements to the left of a_i which are greater than a_i . For example, the inversion tables for all permutations on four elements are given in Example 1.2, and the inversion table for

$$10\quad 7\quad 13\quad 2\quad 4\quad 12\quad 14\quad 8\quad 3\quad 9\quad 15\quad 6\quad 11\quad 5\quad 1$$

is

$$0\quad 1\quad 0\quad 3\quad 3\quad 1\quad 0\quad 4\quad 7\quad 4\quad 0\quad 8\quad 4\quad 10\quad 14 \quad .$$

Inversion tables have many useful elementary properties which follow directly from the definition. First, since there are only $i-1$ elements to the left of a_i for $1 \le i \le n$, we must have

Example 1.2

permutation	D	E	inversion table	permutation	D	E	inversion table
1 2 3 4	0	0	0 0 0 0	1 2 4 3	1	1	0 0 0 1
2 1 3 4	1	1	0 1 0 0	2 1 4 3	2	2	0 1 0 1
2 3 1 4	1	2	0 0 2 0	2 4 1 3	2	3	0 0 2 1
2 3 4 1	1	3	0 0 0 3	2 4 3 1	2	4	0 0 1 3
1 3 2 4	1	1	0 0 1 0	1 4 2 3	2	2	0 0 1 1
3 1 2 4	2	2	0 1 1 0	4 1 2 3	3	3	0 1 1 1
3 2 1 4	2	3	0 1 2 0	4 2 1 3	3	4	0 1 2 1
3 2 4 1	2	4	0 1 0 3	4 2 3 1	3	5	0 1 1 3
1 3 4 2	1	2	0 0 0 2	1 4 3 2	2	3	0 0 1 2
3 1 4 2	2	3	0 1 0 2	4 1 3 2	3	4	0 1 1 2
3 4 1 2	2	4	0 0 2 2	4 3 1 2	3	5	0 1 2 2
3 4 2 1	2	5	0 0 2 3	4 3 2 1	3	6	0 1 2 3

D:	max	3	E:	max	6
	min	0		min	0
	avg	23/12		avg	3
	var	95/144		var	13/6

$$B_1 = 0 \ , \ 0 \le B_2 \le 1 \ , \ 0 \le B_3 \le 2 \ , \ \dots \ , \ 0 \le B_n \le n-1 \quad .$$

There are exactly n! different n-tuples of numbers which satisfy these inequalities. Further, each of these is an inversion table, and corresponds to a unique permutation of $\{1,2,\dots,n\}$. To construct a permutation from a given inversion table, start from the right end, and for $i = n, n-1, \dots, 1$ write down the (B_i+1) -st largest of the numbers not yet used. For example, the permutation corresponding to the table

0 1 1 3 2

is

5 2 4 1 3

since 3 is the third largest of $\{1,2,3,4,5\}$; 1 is the fourth largest of $\{1,2,4,5\}$; 4 is the second largest of $\{2,4,5\}$; 2 is the second largest of $\{2,5\}$; and 5 is the largest of $\{5\}$. This is always possible since $B_i+1 \le i$ and there are i numbers not yet used. This one-to-one correspondence between permutations and inversion tables means that we can treat all inversion tables as equally likely and analyze the values of our quantities when the program is running on permutations as defined by inversion tables. This is particularly convenient for Program 1.3.

We notice immediately from our algorithm that D is the number of keys which have at least one greater key to the left. But this is exactly the number of non-zero elements in the inversion table. The maximum value of D is therefore N-1 (B_1 is always 0), and the minimum value is 0 . The probability that $B_i \ne 0$ is $1-1/i$ for all i , so the average value of D is

$$(1-1) + \left(1 - \frac{1}{2}\right) + \left(1 - \frac{1}{3}\right) + \dots + \left(1 - \frac{1}{N}\right) = N - H_N \quad ,$$

where H_N is the N-th harmonic number.

The quantity E in Program 1.3 is also related to the inversion table. Indeed, each element in the permutation must be moved past every element to the left of it which is greater than it. This is exactly the number counted by entries in the inversion table, so the value of E for a given inversion table is

$$E = B_1 + B_2 + B_3 + \dots + B_N \quad .$$

This sum is also called the number of <u>inversions</u> of the permutation. An inversion of a permutation $a_1 a_2 \dots a_n$ is a pair (a_i, a_j) for which $i < j$ and $a_i > a_j$. For example, the inversions of 3 4 1 2 are (3,1), (3,2), (4,1), and (4,2). The minimum value of E is therefore 0 ($B_i = 0$ for all i); and the maximum value is $\binom{N}{2}$ ($B_i = i-1$ for $1 \le i \le N$). Also, there is an easy way to determine the average value of E. This is to notice that for every permutation $a_1 a_2 \dots a_n$ with k inversions there corresponds a permutation $a_n a_{n-1} \dots a_2 a_1$ with $\binom{n}{2} - k$ inversions. Thus if e_{Nk} is the probability that a permutation of $\{1,2,\dots,N\}$ has exactly k inversions, and $k' = \binom{N}{2} - k$, then

$$e_{Nk} = e_{Nk'} \quad .$$

The average number of inversions is given by

$$\text{avg}(E_N) = \sum_k k e_{Nk} = \sum_k \left(\binom{N}{2} - k\right) e_{Nk'}$$

$$= \sum_k \left(\binom{N}{2} - k\right) e_{Nk} \quad ,$$

so that

$$\begin{aligned} 2\ \mathrm{avg}(E_N) &= \sum_k \left(k + \binom{N}{2} - k \right) e_{Nk} \\ &= \binom{N}{2} \sum_k e_{Nk} \\ &= \binom{N}{2} \quad , \end{aligned}$$

since $\sum_k e_{Nk} = 1$ is a sum of probabilities. Therefore

$$\mathrm{avg}(E_N) = \frac{N(N-1)}{4} \quad .$$

Now that we have the average values of the quantities D and E , we can find the average running time of Program 1.3. In general, the total running time will be

$$\alpha D + \beta E + \gamma N + \delta \quad ,$$

where the coefficients α , β , γ , and δ depend on the particular machine and compiler used to run Program 1.3. To provide a concrete basis for analysis, the programs that we analyze are coded in assembly language in Appendix A. This will give us representative values for the coefficients, although they could be higher or lower for particular machines. For Program 1.3, the running time is

$$3D + 8E + 7N - 6 \quad ,$$

so that we have shown the average running time to be

$$3(N - H_N) + 8\,\frac{N(N-1)}{4} + 7N - 6$$

or

$$2N^2 + 8N - 3H_N - 6 \quad .$$

This dependence on N^2 makes Program 1.3 undesirable for large N , but for small N it is very efficient.

Program 1.2 is also coded in assembly language in Appendix A, and the running time turns out to be

$$3D^* + 9E + 7N - 6 \quad ,$$

where E is the number of inversions, as before, and D^* is the number of keys which have at least one smaller key to the left. This is the number of entries in the inversion table such that $B_i \neq i-1$, and the average value is $N - H_N$, just as above. Our first improvement makes Program 1.3 $\frac{N(N-1)}{4}$ time units faster than Program 1.2, on the average. The second improvement changes the running time by $D - D^*$, which doesn't seem to be an improvement at all, since its average value is zero. However, this quantity tends to be negative if the file has a low number of inversions, so Program 1.3 gains an added advantage if the file is approximately in order.

The derivation of the variance of the quantities D and E will require more sophisticated methods than we have used up to this point. Let d_{Nk} be the probability that D takes on the value k when Program 1.2 is sorting a random permutation of N elements, and let $D_N(z) = \sum_{k \geq 0} d_{Nk} z^k$ be the generating function for $\{d_{Nk}\}$. Then $N! \, d_{Nk}$ is the number of inversion tables for N elements with exactly k non-zero entries. If $N = 1$, the inversion table is 0 , so that $D_1(z) = 1$. For $N > 1$, we will derive a recurrence relation for $D_N(z)$ by defining a correspondence between inversion tables for permutations of N elements and inversion tables for $N-1$ elements. With each inversion table $B_1 B_2 \cdots B_{N-1}$ for $N-1$ elements, we associate the N inversion tables

$$\begin{array}{ccccc} B_1 & B_2 & \cdots & B_{N-1} & 0 \\ B_1 & B_2 & \cdots & B_{N-1} & 1 \\ & \vdots & & & \\ B_1 & B_2 & \cdots & B_{N-1} & N \end{array}$$

The first of these has the same number of non-zero elements in the inversion table as the original; the other (N-1) all have exactly one more. This means that

$$N!\, d_{Nk} = (N-1)!\, d_{(N-1)k} + (N-1)(N-1)!\, d_{(N-1)(k-1)} \quad ,$$

or

$$d_{Nk} = \frac{1}{N}\, d_{(N-1)k} + \frac{N-1}{N}\, d_{(N-1)(k-1)} \quad .$$

Multiplying by z^k and summing over all k , we get

$$D_N(z) = \frac{1}{N}\, D_{(N-1)}(z) + \frac{N-1}{N}\, z\, D_{(N-1)}(z) \quad ,$$

which telescopes to

$$D_N(z) = D_1(z) \prod_{2 \le k \le N} \frac{(k-1)z+1}{k}$$

$$= \prod_{1 \le k \le N} \frac{(k-1)z+1}{k} \quad .$$

This is a product of the probability generating functions $d_k(z) = \frac{(k-1)z+1}{k}$, which have mean $d'_k(1) = 1 - \frac{1}{k}$, and variance $d''_k(1) + d'_k(1) - [d'_k(1)]^2 = \frac{1}{k} - \frac{1}{k^2}$. The mean of the product is the sum of the means, and the variance of the product is the sum of the variances (see Eqs. (48) and (49) in Appendix B), so that

$$\operatorname{avg}(D_N) = \sum_{1 \le k \le N} (1 - \frac{1}{k}) = N - H_N \quad ,$$

which agrees with our earlier result, and

$$\operatorname{var}(D_N) = \sum_{1 \le k \le N} (\frac{1}{k} - \frac{1}{k^2}) = H_N - H_N^{(2)} \quad .$$

The derivation of the variance for E follows exactly the same method. Let e_{Nk} be the probability that a permutation of $\{1,2,\ldots,N\}$ has exactly k inversions, and let $E_N(z) = \sum_{k \ge 0} e_{Nk} z^k$. Then, using the same correspondence between permutations on N elements and permutations on N-1 elements as above, we get

$$e_{Nk} = \frac{1}{N} \sum_{0 \le j \le k} e_{(N-1)(k-j)} \quad ,$$

so that

$$E_N(z) = \frac{1}{N} \sum_{k \geq 0} \sum_{0 \leq j \leq k} e_{(N-1)(k-j)} z^k$$

$$= \frac{1}{N} \sum_{0 \leq j \leq N-1} \sum_{k \geq j} e_{(N-1)(k-j)} z^k$$

$$= \frac{1}{N} E_{N-1}(z) \sum_{0 \leq j \leq N-1} z^j ,$$

which telescopes to

$$E_N(z) = \prod_{1 \leq k \leq N} \frac{1}{k} \sum_{0 \leq j \leq k-1} z^j .$$

This is a product of the probability generating functions $e_k(z) = \frac{1+z+\ldots+z^{k-1}}{k}$, which have mean $e'_k(1) = \frac{1+2+\ldots+(k-1)}{k} = \frac{k-1}{2}$ and variance $e''_k(1) + e'_k(1) - [e'_k(1)]^2 = \frac{k^2-1}{12}$, so that

$$\text{avg}(E_N) = \sum_{1 \leq k \leq N} \frac{k-1}{2} = \frac{N(N-1)}{4}$$

as before, and

$$\text{var}(E_N) = \sum_{1 \leq k \leq N} \frac{k^2-1}{12} = \frac{N(N-1)(2N+5)}{72} .$$

The fact that the generating functions for both D and E turned out to be products of probability generating functions suggests that we might have found a simpler derivation, using independent random variables. In general, if X and Y are random variables described by the generating functions $X(z) = \sum_{k \geq 0} \Pr\{X = k\} z^k$ and $Y(z) = \sum_{k \geq 0} \Pr\{Y = k\} z^k$, then

$$X(z)Y(z) = \sum_{k \geq 0} \left(\sum_{0 \leq j \leq k} \Pr\{X = j\} \Pr\{Y = k-j\} \right) z^k \quad .$$

If X and Y are independent, then $\Pr\{X = j\}\Pr\{Y = k-j\} = \Pr\{X = j \text{ and } Y = k-j\}$, or $\Pr\{X + Y = k\}$, so $X(z)Y(z)$ is the generating function describing $X + Y$. To apply this to our problem, we consider the inversion table entries $B_1 B_2 \ldots B_N$ as random variables. Our assumption that all inversion tables are equally likely is equivalent to the assumption that these random variables are all independent and that B_k takes on each of the values $0, 1, \ldots, k-1$ with probability $1/k$ for $1 \leq k \leq N$. The generating function for B_k is therefore

$$\frac{1}{k}\,(1 + z + \ldots + z^{k-1}) \qquad \text{for } 1 \leq k \leq N \quad .$$

Now, by definition, we know that the sum of the B_k's is the number of inversions

$$E = B_1 + B_2 + \ldots + B_N$$

and from the law above, the generating function for this sum is the product of the generating functions for the individual terms:

$$E(z) = \prod_{1 \leq k \leq N} \frac{1}{k} \sum_{0 \leq j \leq k-1} z^j \quad ,$$

which is what we found before. Similarly, if we define

$$X_k = \begin{cases} 0 & B_k = 0 \\ 1 & B_k \neq 0 \end{cases} \qquad 1 \leq k \leq N \quad ,$$

then

$$D = X_1 + X_2 + \ldots + X_N$$

and the generating function for X_k is $\frac{1}{k} + \frac{k-1}{k} z$, so that

$$D(z) = \prod_{1 \le k \le N} \frac{(k-1)z+1}{k} \quad ,$$

as before. From this function, we can proceed even further, to find an explicit expression for d_{Nk} . We have

$$D(z) = \frac{1}{N!} \prod_{1 \le k \le N} ((k-1)z+1)$$

$$= \frac{z^N}{N!} \prod_{0 \le k \le N-1} \left(\frac{1}{z} + k\right)$$

$$= \frac{z^N}{N!} \sum_k \begin{bmatrix} N \\ k \end{bmatrix} \left(\frac{1}{z}\right)^k \quad ,$$

where $\begin{bmatrix} N \\ k \end{bmatrix}$ are Stirling numbers of the first kind (see Eqs. (26)-(30) and (37) in Appendix B). Therefore $D(z) = \sum_k \begin{bmatrix} N \\ N-k \end{bmatrix} z^k$, so $d_{Nk} = \frac{1}{N!} \begin{bmatrix} N \\ N-k \end{bmatrix}$. We could now compute our average directly,

$$\operatorname{avg}(D_N) = \frac{1}{N!} \sum_k k \begin{bmatrix} N \\ N-k \end{bmatrix}$$

$$= \frac{1}{N!} \sum_k (N-k) \begin{bmatrix} N \\ k \end{bmatrix}$$

$$= \frac{1}{(N-1)!} \sum_k \begin{bmatrix} N \\ k \end{bmatrix} - \frac{1}{N!} \sum_k k \begin{bmatrix} N \\ k \end{bmatrix}$$

$$= N - \frac{1}{N!} \begin{bmatrix} N+1 \\ 2 \end{bmatrix}$$

$$= N - H_N \quad ,$$

but the method that we used before was much simpler. (See Appendix B for some identities relevant to this derivation: the sums of Stirling numbers are evaluated from the generating function (37) and its derivative.) For many generating functions, it is most convenient to compute moments directly from the probabilities; for others (including $D(z)$) it is easier to work solely with derivatives of the generating function; and for still others (including $E(z)$) the individual probabilities are not available, and it is <u>necessary</u> to use methods such as we used above.

Our analysis of the quantities D and E for Program 1.3 is now complete. We have found their maximum and minimum values, and the generating functions, from which we have determined the first and second moments. Higher moments could also be derived from the generating functions if desired. To summarize, we know that

D_N:	max	$N-1$	E_N:	max	$\frac{N(N-1)}{2}$
	min	0		min	0
	avg	$N-H_N$		avg	$\frac{N(N-1)}{4}$
	var	$H_N-H_N^{(2)}$		var	$\frac{N(N-1)(2N+5)}{72}$.

These expressions, for $N=4$, agree with those in Example 1.2.

From a practical standpoint, of course, we are interested in the performance of the program as a whole. We have already used the fact that the average running time of the program can be obtained by adding the average values of the contributing quantities (with appropriate coefficients). For the maximum, minimum, and standard derivation, however, this simple rule may not hold in general because of interactions among the quantities.

Fortunately, for Program 1.3, the minimum value of D occurs for the same permutation $(1\,2\,\ldots\,N)$ as the minimum value of E ; also the maximum values both occur for the permutation $N\ N{-}1\ \ldots\ 2\ 1$. This means, of course, that the minimum and maximum running times of the whole program must occur for these permutations: the values are $7N-6$ and $4N^2+6N-9$ respectively.

We can also derive the variance of the running time of Program 1.3 in exactly the same way that we found the variance of D and E . If t_{Nk} is the probability that Program 1.3 takes time k , namely the probability that $3D+8E+7N-6=k$, and if $T_N(z) = \sum_{k\geq 0} t_{Nk}\, z^k$ is the associated generating function, we proceed exactly as we did for the separate quantities to get the generating function

$$T_N(z) = z \prod_{2\leq k\leq N} \frac{1}{k}\Big(z^7 + \sum_{1\leq j\leq k-1} z^{8j+10}\Big)$$

where the probability generating functions $t_k(z) = \frac{1}{k}\Big(z^7 + \sum_{1\leq j\leq k-1} z^{8j+10}\Big)$ have mean $4k+6-\frac{3}{k}$ and variance $\frac{16}{3}k^2 + \frac{56}{3} - \frac{15}{k} - \frac{9}{k^2}$. Therefore

$$\mathrm{mean}(T_N) = 1 + \sum_{2 \le k \le N} (4k + 6 - \frac{3}{k}) = 2N^2 + 8N - 3H_N - 6 \quad ,$$

as before, and

$$\mathrm{var}(T_N) = \sum_{2 \le k \le N} (\frac{16}{3} k^2 + \frac{56}{3} - \frac{15}{k} - \frac{9}{k^2}) =$$

$$= \frac{16}{9} N^3 + \frac{8}{3} N^2 + \frac{176}{9} N - \frac{112}{3} - 15 H_N - 9 H_N^{(2)} \quad .$$

This solution for the variance of the total running time was possible only because we were able to set up such a simple recurrence on the generating function. It is ordinarily very difficult to get the variance of the total running time of a program, and we usually must be content with the variances of the various contributing quantities.

In summary, we have derived exact formulas describing the total running time of Program 1.3:

$$\begin{array}{lll} T_N: & \max & 4N^2 + 6N - 9 \\ & \min & 7N - 6 \\ & \mathrm{avg} & 2N^2 + 8N - 3H_N - 6 \\ & \mathrm{var} & \frac{1}{9}(16N^3 + 24N^2 + 176N - 336 - 45H_N - 81H_N^{(2)}) \quad . \end{array}$$

This analysis, in addition to providing a showcase for many of the methods used in analysis of algorithms, represents a goal towards which we work when studying programs. For many programs, it is not possible to compute exact formulas for the total running time. In fact, it is often the case that even approximate formulas cannot be derived. However, when analysis does yield exact answers, we have very complete information about the program, not only in the results, but also in the clear understanding of the algorithm required for analysis.

Program 1.3 is a simple sorting method whose performance we completely understand -- why do we look at other sorting methods? The answer, as we have already remarked, is that the program takes much too long if N is very large. If our unit of time is 1 microsecond, then Program 1.3 will take over five hours to sort 100,000 elements. We should expect to be able to do much better. (On the other hand, for small N, Program 1.3 is about the best sorting method known.)

Studies of computational complexity (which comprise another kind of "analysis of algorithms") show that the sorting problem should require at least $O(N \lg N)$ operations. The argument goes as follows. In order to sort every permutation of N keys properly, a sorting program must be able to "distinguish between" (i.e., operate differently for) all of the $N!$ possible inputs. A sorting program which uses only k comparisons in the worst case can only distinguish between 2^k input permutations. This means that k must be large enough so that

$$2^k \geq N!$$

or

$$k \geq \lg N!$$

or

$$k > N \lg N - N \lg e \quad , \quad \text{by Stirling's approximation (Eq. (55) in Appendix B).}$$

Therefore, every sorting method requires at least $N \lg \left(\frac{N}{e}\right)$ comparisons in the worst case. A similar but slightly more complex argument says that the average number of comparisons must also be about $N \lg N$. And this lower bound can be achieved -- there are several sorting methods which use $c N \lg N$ comparisons for some constant c.

Of course real programs consist of more than comparisons, and there are other factors which must be taken into account. For example, we might consider modifying our insertion algorithm as follows: To find the proper position for $v \equiv A[i]$ among the elements already sorted $(A[1],A[2],\ldots,A[i-1])$, first compare it with the middle element $A[(i-1)\div 2]$ to see if it belongs in the left half or the right half. If it belongs in the left half, then compare it with the middle element of the left half, etc., continuing in this manner until the proper position is found. This "binary insertion" algorithm requires about $N \lg N$ comparisons, but its running time is still dominated by $\frac{1}{4} N^2$ "moves during insertion". We might consider using linked list techniques to eliminate these, but then we would need an amount of extra storage proportional to N , which might be undesirable. Nevertheless. there are several sorting methods which overcome such difficulties and sort N elements in a total amount of time proportional to $N \lg N$. One of the most efficient of these, with some qualifications, is Quicksort.

Fortunately, the Quicksort algorithm and its best variants admit to complete analysis: this will be our concern for the rest of the thesis. The next chapter will be a careful development of a practical, elegant, and efficient program based on the Quicksort algorithm. Of course, the program will be much more involved than Program 1.3, and both the results and methods of analysis will be more complex and interesting. The average running time of the program will be derived in Chapter 3. The following chapter considers the performance of the program in the best case and in the worst case. Then, after a look at some minor modifications and practical considerations, the three major variants of Quicksort will be examined: van Emden's approach; samplesort; and the median-of-three modification. A complete analysis will be presented for the best of these.

Throughout this investigation, we will be interested in learning about the effectiveness of the various algorithms, as demonstrated by exact analysis. In addition, we will pay attention to the impact of the various modifications on the analysis itself. A variety of interesting general problems in concrete mathematics are suggested by the analysis of Quicksort. We will be studying not only an important general-purpose sorting method, but also a family of important general methods of analysis.

CHAPTER TWO

The Quicksort algorithm is an application to sorting of the general "divide and conquer" principle of solving a problem by dividing it into two subproblems, then solving them in the same way, repeating the process until the resulting problems are simple enough to solve in some other way. The algorithm can be expressed recursively as follows:

Program 2.1

```
procedure quicksort (integer value ℓ,r);
     if r > ℓ then
          partition on A[j];
          quicksort (ℓ,j-1);
          quicksort (j+1,r);
     endif;
```

Here the procedure call " quicksort(ℓ,r) " will cause the $r-\ell+1$ elements in the array $A[\ell] \ldots A[r]$ to be sorted. In particular " quicksort(1,N) " will sort the entire array.

The crux of the algorithm, of course, is the " partition on A[j] " process, which will now be defined. Partitioning means to rearrange the array so that two conditions are satisfied:

(i) some element, say the j-th smallest, is in its final position in the array (A[j]) ;

(ii) all elements to the left of A[j] are less than or equal to it and all elements to the right of A[j] are greater than or equal to it.

Thus, since A[j] is in position, the original problem of sorting the entire array is reduced to the problem of sorting the "left subfile" (the elements to the left of A[j]) and the "right subfile" (the elements to the right of A[j]). If a file is of size 0 or 1 , it is already sorted. If Program 2.1 works properly for all files of size < N , then it clearly sorts N elements. Thus, by induction, we see that Program 2.1 is a proper sorting procedure.

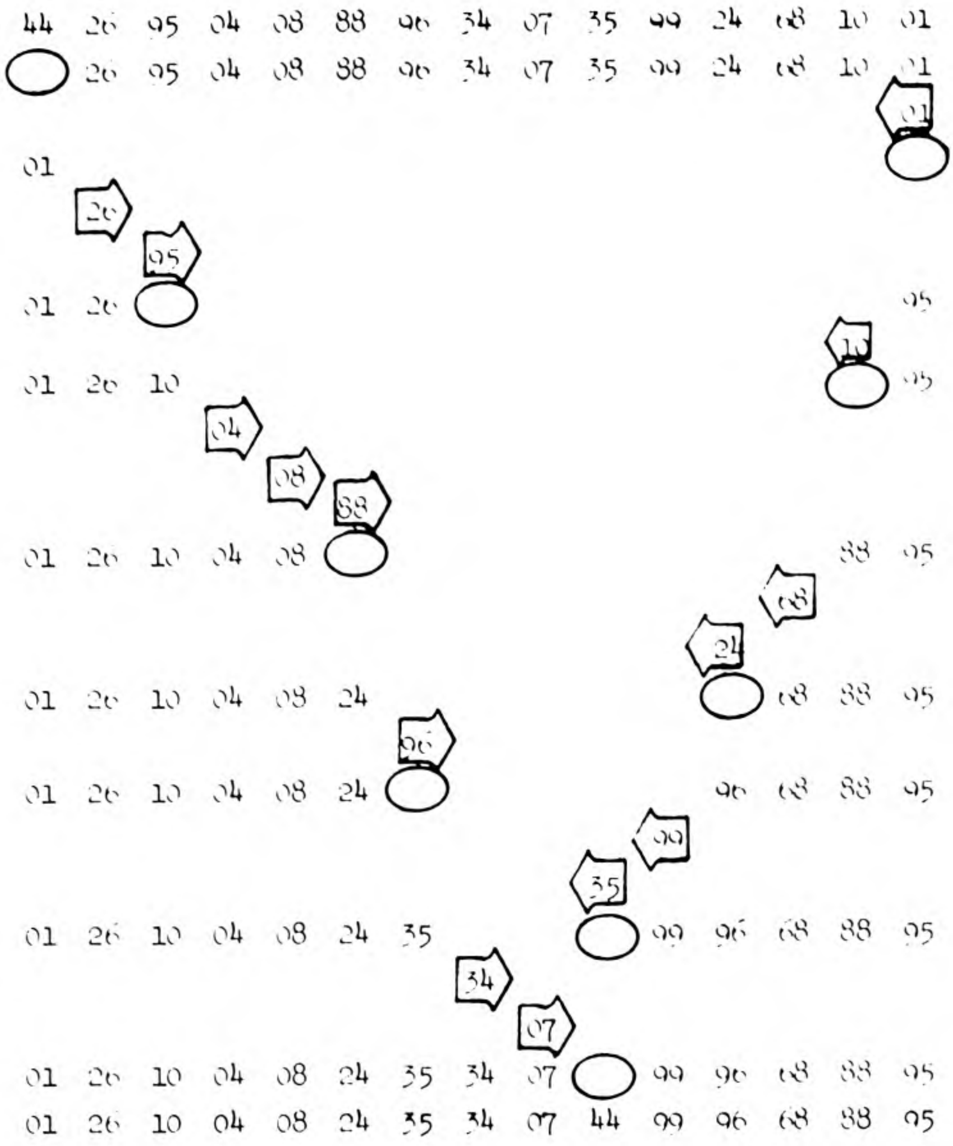
There are several methods which have been suggested to achieve partitioning, all with the same general structure. As we shall see, it is a somewhat delicate process. While the methods seem to differ only slightly, the performance of the methods may differ significantly.

One of the most natural partitioning methods is detailed in Example 2.1. Suppose that the fifteen keys shown on the top line are to be sorted. First the leftmost element of the array, the 44 , is arbitrarily selected as the partitioning element. It is removed from the array, leaving a hole on the left. An element is found to fill this hole by scanning from the right for the first element ≤ 44 . The 01 is found, and moved to the hole vacated by the 44 , but leaving a hole of its own on the right. Similarly, an element to fill this hole is found by scanning from the left to find the first element ≥ 44 , in this case the 95 . This leaves a hole on the left again, and the process continues until all the elements to the left of the hole are ≤ 44 and all the elements to the right of the hole are ≥ 44 . Partitioning is then completed by filling the hole with the 44 key.

If this partitioning algorithm is used in Program 2.1, then the entire file is sorted as shown in the second part of Example 2.1. Each line shows the result of partitioning the subfile defined by

Example 2.1

partitioning:



sorting the file:

															l	r
44	26	95	04	08	88	96	34	07	35	99	24	68	10	01		
01	26	10	04	08	24	35	34	07	$\textcircled{44}$	99	96	68	88	95	1	15
$\textcircled{01}$	26	10	04	08	24	35	34	07							1	9
	07	10	04	08	24	$\textcircled{26}$	34	35							2	9
	04	$\textcircled{07}$	10	08	24										2	6
			08	$\textcircled{10}$	24										4	6
							$\textcircled{34}$	35							8	9
										95	96	68	88	$\textcircled{99}$	11	15
										88	68	$\textcircled{95}$	96		11	14
										68	$\textcircled{88}$				11	12

the given values of ℓ and r . The array is completely sorted in 9 partitioning stages.

This loose description leaves some latitude in implementing this algorithm: one of the simplest implementations is the following program.

Partitioning Method 2.1

```
i := $\ell$; j := r+1; v := A[$\ell$];
loop until pointers have met:

    loop: j := j-1; while A[j] > v repeat;
    if i $\geq$ j then j := i; pointers have met endif;
    A[i] := A[j];

    loop: i := i+1; while A[i] < v repeat;
    if i $\geq$ j then pointers have met endif;
    A[j] := A[i];

repeat;
A[j] := v;
```

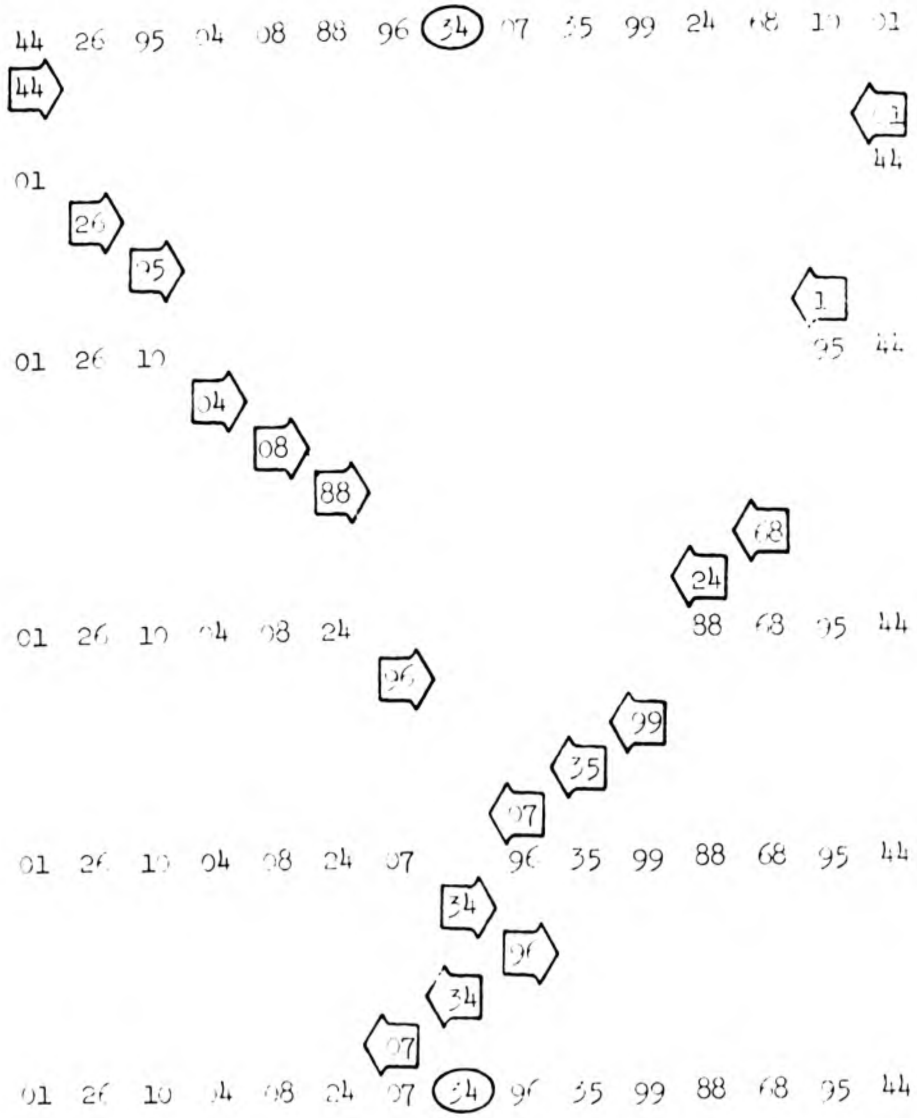
The operation of this program is straightforward except in the case that A[ℓ] is larger than all of the keys A[ℓ+1],...,A[r] . Then the condition A[i] < v will be true for all $i \neq \ell$, and the i := i+1 loop cannot be guaranteed to terminate. If the subfile being partitioned is within a larger file being sorted, as occurs in the last partitioning stage of Example 2.1, this may not be a problem, since there will usually be larger elements to the right. But it is a problem if it occurs when r = N , as in the 7th partitioning stage of Example 2.1. This case is handled with the same technique that was used in Program 1.3: the assumption that A[N+1] is larger than all of the other keys (written A[N+1] = ∞) is sufficient to guarantee that the loop always terminates and the program works correctly. Some partitioning methods have similar trouble at the left end, and we will assume in such cases that A[0] is less than all of the other keys (A[0] = -∞) .

The main difficulty with this partitioning method is that the termination condition can occur either after the i pointer has incremented and stopped or after the j pointer has decremented and stopped. The two tests required to see whether the "pointers have met" make this algorithm less efficient than is necessary. The second partitioning method that we will examine does not have this problem.

The partitioning procedure shown in Example 2.2.is the method used in the original Quicksort algorithm in 1960 by C. A. R. Hoare. His method is based on exchanging elements. Again, an arbitrary key is chosen as the partitioning element, this time the element in the middle of the array, the 34 . Now, we scan from the left for the first element > 34 and from the right for the first element < 34 . These two, the 44 and the 01 , are obviously out of place if the file is to be partitioned correctly, so they are simply exchanged. Continuing in exactly the same manner, we exchange the 95 and the 10 ; the 88 and the 24 ; and the 96 and the 07 . The next time our scans stop,

Example 2.2

partitioning:



sorting the file:

															ℓ	r
44	26	95	04	08	88	96	34	07	35	99	24	68	10	01		
01	26	10	04	08	24	07	34	96	35	99	88	68	95	44	1	15
01	04	10	26	08	24	07									1	7
		07	08	26	24	10									3	7
				10	24	26									5	7
								44	35	68	88	99	95	96	9	15
								35	44	68					9	11
									44	68					10	11
												95	99	96	13	15
													96	99	14	15

on the 07 and the 96 , the pointers have crossed, so that no exchange is necessary. The algorithm has ensured that the keys to the right of the 07 are all greater than 34 and the keys to the left of the 96 are all less than 34 . Partitioning is now complete. In general, we would have to put the partitioning element into position as shown in the algorithm below, but we were fortunate in this example: the partitioning element was already in place.

In this method, the test on whether the pointers have crossed occurs only after <u>both</u> the left and right pointers have stopped. This is approximately half as often as in the previous method, and the inner loop is much simpler.

<u>Partitioning Method 2.2</u>

```
    A[0] := -∞; A[N+1] := ∞;
    i := ℓ-1; j := r+1; p := (ℓ+r) ÷ 2; v := A[p];
    loop:
          loop:  i := i+1; while A[i] ≤ v repeat;
          loop:  j := j-1; while A[j] ≥ v repeat;
    while i < j:
          A[i] :=: A[j];
    repeat;
    if i < p then A[i] :=: A[p]; i := i+1 endif;
    if j > p then A[p] :=: A[j]; j := j-1 endif;
   ⎡quicksort (ℓ,j); ⎤
   ⎣quicksort (i,r); ⎦
```

It is instructive to note the care which must be exercised after the pointers have crossed in this method. The object is to get the partitioning element A[p] into its proper place within the array. The pointer scans don't move A[p] , and after the pointers have stopped, there are three cases: A[p] may be in the right subfile; the left subfile; or neither. If A[p] is in the right subfile, it is known to be no larger than any of the other elements there, and it can be put into place by exchanging it with the leftmost element of the right subfile. This occurs, for example, in the second partitioning stage of Example 2.2. The symmetric argument holds if A[p] is in the left subfile. Finally, if A[p] is in neither subfile, as occurred in our first partitioning stage, it is known to be already in place.

If all of the keys are distinct, as in our examples, then it will always be true that $i-1 = j+1 =$ (final position of partitioning element) at the end of Partitioning Method 2.2. If, however, there are equal keys present, then $i-j$ may be greater than two, and possibly more than one key may be known to be in its final position after partitioning. Despite this, the method actually tends to perform inefficiently when equal keys are present. As we shall see later, Quicksort performs best when the two subfiles are approximately the same size. It turns out that stopping the pointers on keys equal to the partitioning element tends to bring the partition closer to the center. Although this may result in equal keys being exchanged, this is more than compensated for by the more balanced partitions. Furthermore, this corrects an even more serious defect of Method 2.2: the recursive calls " quicksort(ℓ,j) " and " quicksort(i,r) " might access elements far outside their subscript ranges. For example, if a partitioning element equal to r is chosen when the left subfile is partitioned, then the left pointer will pass over all the keys in the partition just made and reach into the right subfile! We shall study the question of equal keys in more detail in Chapter 5.

Example 2.3 shows the result of modifying Hoare's original method to exchange on keys equal to the partitioning element. For variety, we arbitrarily choose the second element of the file as the partitioning element for this example. In the first stage, then, we scan from the left for the first element ≥ 26 and from the right for the first element ≤ 26 . These two, the 44 and the 01 , are exchanged. The next element ≥ 26 on the right is the 26 itself. It is exchanged with the 10 . Two more exchanges are made, and the pointers cross at the 96 and the 07 .

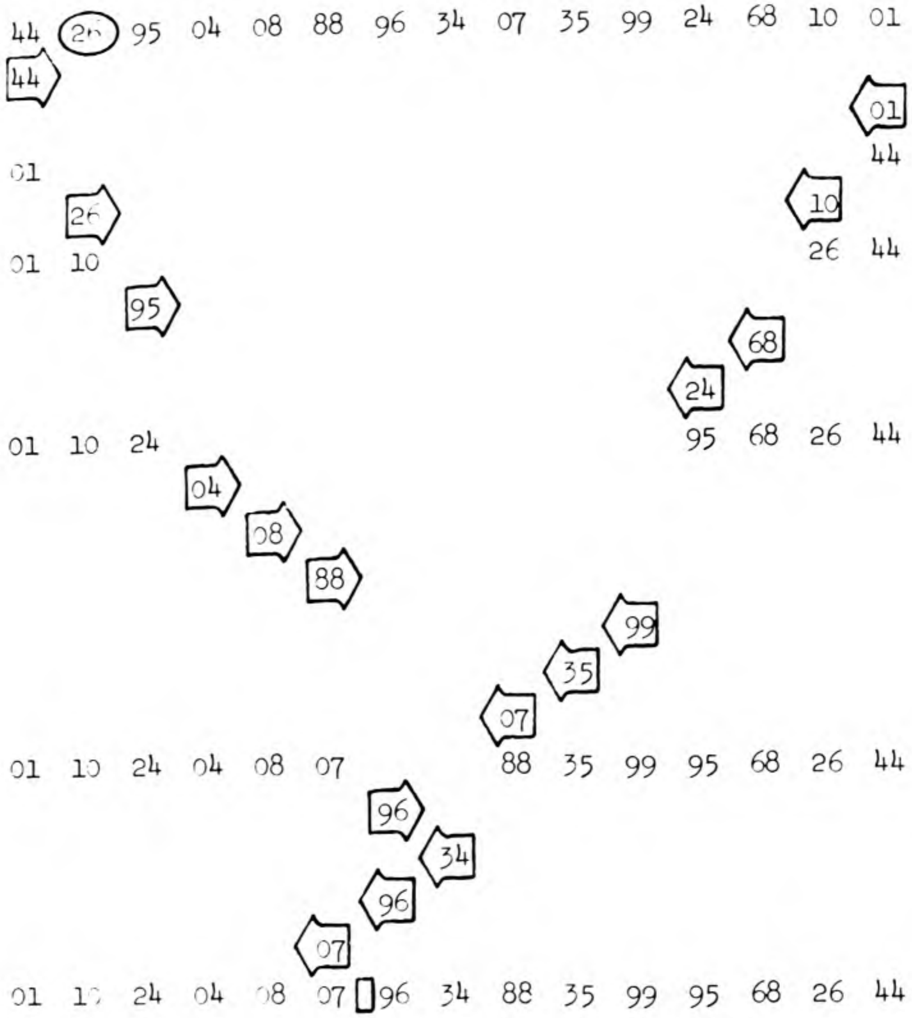
There is no efficient way to get the partitioning element into its proper place for this method. Unless it is already in place, it is "lost" on an exchange. To keep track of it would require extra overhead in the inner loop in the following algorithm. Despite the fact that the method violates condition (i) of partitioning in this way, it does sort the elements as shown in the second part of Example 3, and the algorithm is very elegant.

Partitioning Method 2.3

```
i := ℓ-1; j := r+1; p := ℓ+1; v := A[p];
loop:
      loop:  i := i+1; while A[i] < v repeat;
      loop:  j := j-1; while A[j] > v repeat;
while i < j:
      A[i] :=: A[j];
repeat;
[quicksort (ℓ,i-1); ]
[quicksort (j+1,r); ]
```

Example 2.3

partitioning:



```
sorting
the file:   44  26  95  04  08  88  96  34  07  35  99  24  68  10  01  |  l  r
            01  10  24  04  08  07▯96  34  88  35  99  95  68  26  44   |  1 15
            01  07  08  04▯24  10                                       |  1  6
            01  04▯08  07                                               |  1  4
            01 (04)                                                     |  1  2
                    07▯08                                               |  3  4
                            10▯24                                       |  5  6
                                    26 (34) 88  35  99  95  68  96  44  |  7 15
                                            35▯88  99  95  68  96  44   |  9 15
                                                88  44  95  68  96▯99   | 10 15
                                                44▯88  95  68  96       | 10 14
                                                    88  68▯95  96       | 11 14
                                                    68▯88               | 11 12
                                                            95 (96)     | 13 14
```

If some element equal to the partitioning element is already in position, this algorithm could end with $i = j$, so that the subfiles on the left and right of that element can be sorted. Otherwise, the algorithm will always end with $j+1 = i$ (there can be no k such that $v < A[k] < v$) and the subfiles to be sorted are $A[\ell],\ldots,A[j]$ and $A[i],\ldots,A[r]$. Notice that the dummy keys $A[0] = -\infty$ and $A[N+1] = \infty$ are not needed in this algorithm, because both pointers must at least stop at p . Also we cannot have $j = r$ or $i = \ell$, because this would imply $A[i] < v$ for all i or $A[j] > v$ for all j , which is impossible.

Unfortunately, this very elegant method cannot be recommended for partitioning because it introduces a bias into the subfiles. When partitioning a random permutation of distinct keys, all of the other methods we've seen will produce a random permutation of keys in both the left and right subfiles. (This fact is very important to the analysis of Quicksort, and it is proved carefully below.) Partitioning Method 2.3, however, only produces random subfiles if the partitioning element is already in place. If it falls in the left subfile it is the largest element there; if it falls in the right subfile it is the smallest element there. In either case, it does not fall into every position with equal probability. In Example 2.3, since the partitioning element is exchanged with the first or second key from the right that has a smaller value, it tends to fall near the right end of the right subfile. This bias not only makes analysis of the method virtually impossible, it also slows down the sorting process considerably. We will study this in more detail in Chapter 5.

We are beginning to face the rather discouraging prospect that we might not find a partitioning method which avoids all of the anomalies encountered above. Are there methods which put the partitioning element in its proper place, produce random subfiles, perform acceptably when equal keys are present, and have an efficient inner loop? Fortunately there are: for example, the following program satisfies all of these requirements.

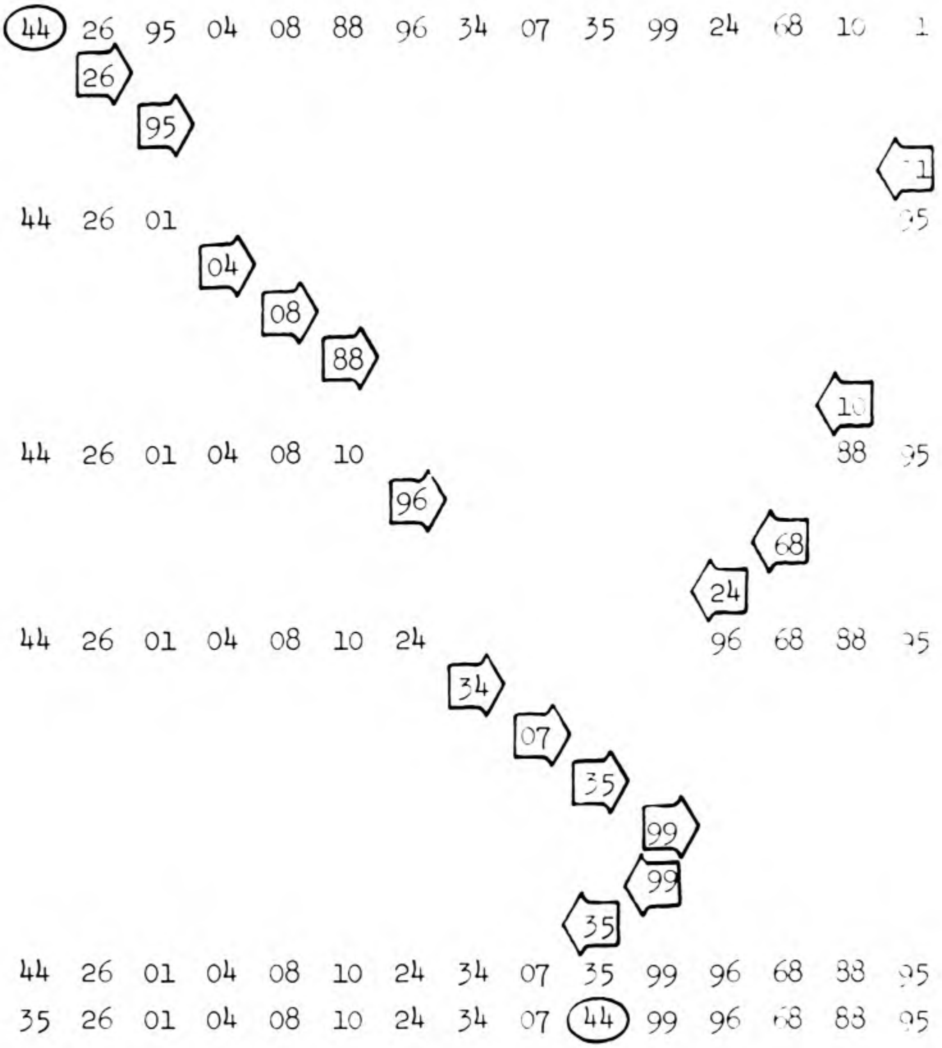
Program 2.2 (Partitioning Method 2.4)

```
procedure quicksort (integer value ℓ,r);
    if r > ℓ then
        i := ℓ; j := r+1; v := A[ℓ];
        loop:
             loop:  i := i+1; while A[i] < v repeat;
             loop:  j := j-1; while A[j] > v repeat;
        while i < j:
             A[i] :=: A[j];
        repeat;
        A[ℓ] :=: A[j];
        quicksort (ℓ,j-1);
        quicksort (j+1,r);
    endif;
```

The partitioning method used here is detailed in Example 2.4. Again the first element of the array is arbitrarily chosen to be the partitioning element. But now the idea is to leave that element where it is and partition the rest of the array on its value. The left pointer therefore starts at the second element of the array. The first element ≥ 44 is 95 , and the first element ≤ 44 from the right is 01 , so that these two are exchanged. Next the 88 and the 10 , then the 96 and the 24 are exchanged; and the pointers cross on the 35 and the 99 . The inner loops have now ensured that all of the elements to the left of the 99 are ≤ 44 ; and all those to the right of the 35

Example 2.4

partitioning:



```
44  26  95  04  08  88  96  34  07  35  99  24  68  10  01
    26
        95
                                                        01
44  26  01                                              95
            04
                08
                    88
                                                    10
44  26  01  04  08  10                              88  95
                        96
                                                68
                                            24
44  26  01  04  08  10  24                  96  68  88  95
                            34
                                07
                                    35
                                        99
                                        99
                                    35
44  26  01  04  08  10  24  34  07  35  99  96  68  88  95
35  26  01  04  08  10  24  34  07  44  99  96  68  88  95
```

sorting
the file:

```
44  26  95  04  08  88  96  34  07  35  99  24  68  10  01 |  l  r
35  26  01  04  08  10  24  34  07  44  99  96  68  88  95 |  1 15
07  26  01  04  08  10  24  34  35                         |  1  9
01  04  07  26  08  10  24  34                             |  1
01  04                                                     |  1  2
            24  08  10  26  34                             |  4
            10  08  24                                     |  4  6
            08  10                                         |  4  5
                                        95  96  68  88  99 | 11 15
                                        68  88  95  96     | 11 14
                                        68  88             | 11 12
```

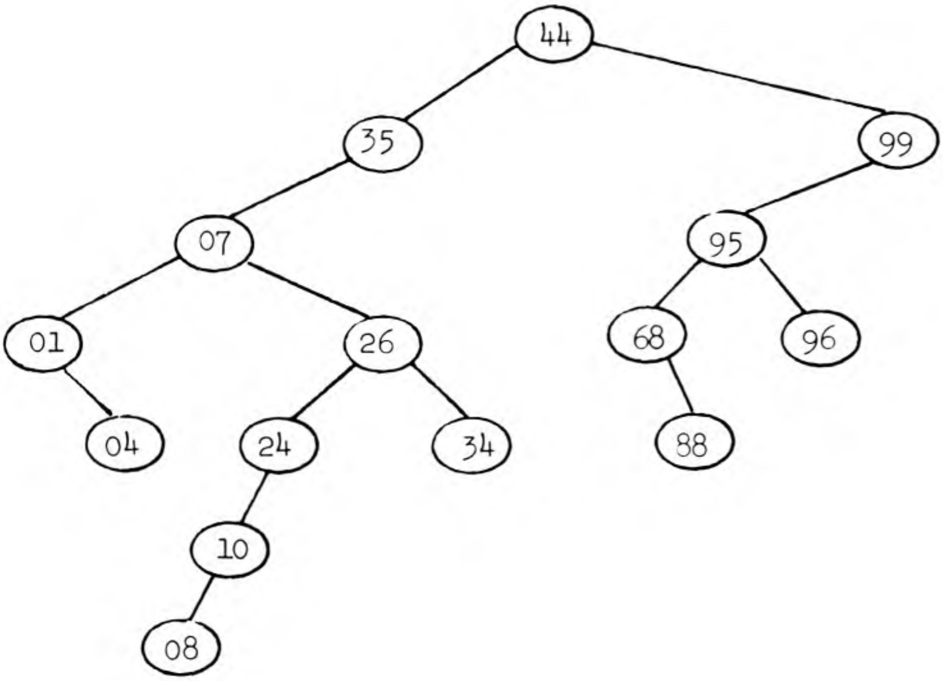
are ≥ 44 . Therefore, the 44 is put into its proper position simply by exchanging it with the 35 .

If all the keys are distinct, this method performs exactly as Hoare's original method (Partitioning Method 2.2) with $p := \ell$. However, the technique used here of leaving the partitioning element out of the partitioning process allows this method to perform properly and efficiently when equal keys are present.

The proof of the fact that Partitioning Method 2.4 produces random subfiles after partitioning is not difficult. Suppose that a permutation of $\{1,2,\ldots,N\}$ is being partitioned with all $N!$ such permutations equally likely, and suppose that the partitioning element is s , $1 \leq s \leq N$. Consider the $s-1$ elements which are less than s . To each of the $(s-1)!$ possible permutations of these in the original file there corresponds one and only one permutation in the left subfile. Since all of the original permutations are equally likely, all permutations of the left subfile must be equally likely. The analogous argument holds, independently, for the right subfile. This same argument holds for Partitioning Methods 2.1 and 2.2, but _not_ for Partitioning Method 2.3. The fact that partitioning produces random subfiles will be the basis for our analysis in Chapter 3.

Program 2.2 is a very elegant description of the Quicksort algorithm based on the most efficient partitioning scheme known. However, the program as a whole is not a practical sorting method, because of the recursion. Recursion makes Program 2.2 impractical not necessarily because of the time overhead involved, but rather because of the space overhead which could be involved. To illustrate this, we will first look at a useful way of describing the operation of Quicksort with binary tree structures.

The binary tree corresponding to the operation of Quicksort on any permutation of n elements can be constructed as follows: If n = 0 the tree is empty, and if n = 1 the tree consists of one node with that one element. Otherwise, the root node is the partitioning element, the left subtree is the tree corresponding to the left subfile after partitioning, and the right subtree is the tree corresponding to the right subfile after partitioning. For example, the tree corresponding to the operation of Program 2.2 on our fifteen keys is



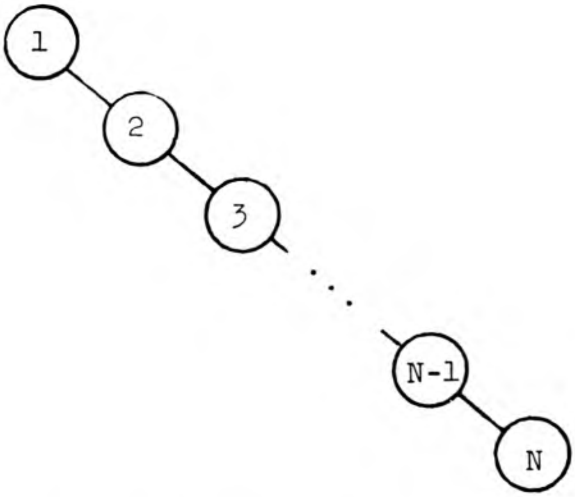
This tree structure is a succinct way of describing the operation of the Quicksort algorithm.

Clearly, there is such a tree for each permutation of the keys and there is at least one permutation for each tree, though many permutations may produce the same tree. We can always work backwards to reconstruct a permutation of the keys from a given tree. For Program 2.2 the method is to scan the tree in symmetric order, writing down the keys as they are encountered, except that the root node of a subtree is always exchanged with the first number corresponding to its left subtree. This procedure for the tree above is illustrated below:

```
    01
        04
    07  04  01
                08
                10  08
                24  08  10
                26  08  10  24
                                34
    35  04  01  26  08  10  24  34  07
    44  04  01  26  08  10  24  34  07  35
                                            68
                                                88
                                            95  88  68
                                                        96
                                            99  88  68  96  95
    44  04  01  26  08  10  24  34  07  35  99  88  68  96  95
```

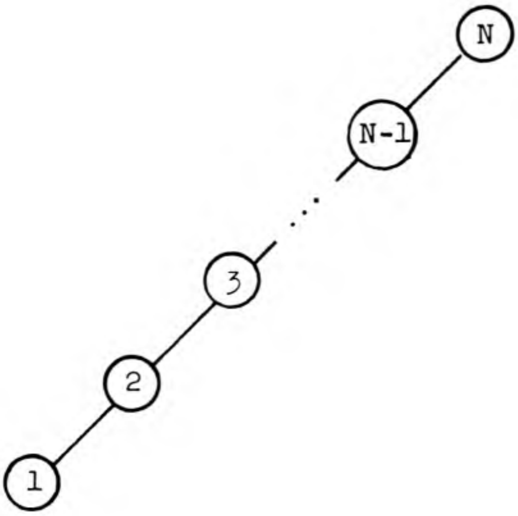
The procedure to generate <u>all</u> permutations corresponding to a given partitioning tree is more complicated; we have constructed the unique permutation, for each tree, such that no exchanges A[i] :=: A[j] are performed during the entire Quicksort operation.

The purpose of the recursion in Program 2.2 is to save subfiles for later consideration. The "depth" of the recursion is the number of subfiles which have been saved. From a practical standpoint, it is desirable to minimize the maximum depth that could occur. It is not difficult to see that the depth of recursion at any point during the execution of Program 2.2 is exactly the distance from the current partitioning element to the root in the corresponding binary tree. (Put another way, it is the number of nodes whose left or right subtree contains the partitioning element.) This means, of course, that the recursive stack must be as large as the maximum height of any binary tree. If the N keys to be sorted are the numbers 1 2 ... N in order, then the tree is

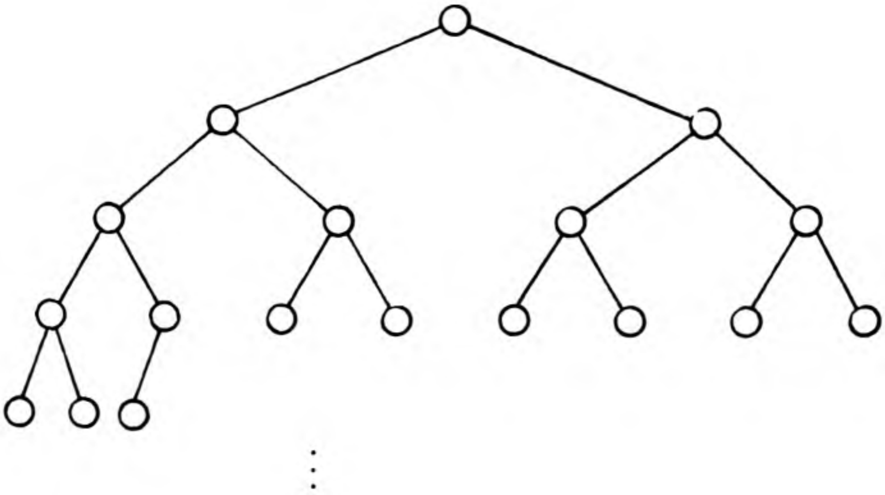


and an amount of storage proportional to N must be reserved for the recursive stack. This may be unacceptable, particularly if N is very large.

This can be alleviated somewhat after noticing that the second recursive call in Program 2.2 is not really recursive. Since it is the last action in a procedure, it could be replaced by a branch (after the parameters have been reset appropriately). If this is done (hopefully by a clever compiler), the depth of recursion is the number of nodes whose left subtree contains the current partitioning element. This is usually smaller than before, but it can still be as large as N , for example if the numbers N 1 2 3 ... N-2 N-1 are sorted:



These difficulties stem from the fact that Program 2.2 makes an implied decision which is arbitrary: it always sorts the left subfile first. The strategy of always sorting the smaller of the two subfiles first, as we will see, will reduce the maximum recursive depth to $\lfloor \lg(N+1) \rfloor$. We cannot expect to do better than this because such depth is required by the permutations corresponding to the complete tree on N nodes:



This tree has $\lfloor \lg(N+1) \rfloor$ "full" levels, and possibly one more partially filled level. No matter which subfile is partitioned first, the recursive depth when the first node on any level k $(1 \leq k \leq \lfloor \lg(N+1) \rfloor)$ is reached must be exactly k . This means in particular that the maximum recursive depth is $\lfloor \lg(N+1) \rfloor$.

Program 2.3 is the implementation of this idea of always sorting the smaller of the two subfiles first. Because of the effect mentioned above, it is convenient to use an explicit stack and to replace the recursion by an iteration. For clarity, the details of the stack manipulation and the selection of the smaller subfile have been omitted.

Program 2.3

```
procedure quicksort (integer value ℓ,r);
     loop until done:
          i := ℓ; j := r+1; v := A[ℓ];
          loop:
               loop:  i := i+1; while A[i] < v repeat;
               loop:  j := j-1; while A[j] > v repeat;
          while i < j:
               A[i] :=: A[j];
          repeat;
          A[ℓ] :=: A[j];
          if subfiles both empty
               then if stack empty then done;
                                   else (ℓ,r) := popstack;
                    endif;
               else if small subfile empty
                         then (ℓ,r) := large subfile;
                         else pushstack(large subfile);
                              (ℓ,r) := small subfile;
                    endif;
          endif;
     repeat;
```

When n elements are partitioned, the smaller of the two subfiles can be no larger than $\left\lfloor \frac{n-1}{2} \right\rfloor$; and it is always "empty" for $n \leq 2$. Furthermore, the upper bound $\left\lfloor \frac{n-1}{2} \right\rfloor$ is achieved when $s = \left\lfloor \frac{n-1}{2} \right\rfloor + 1$. Therefore, the maximum stack depth for Program 2.3 satisfies the recurrence

$$f(n) = \begin{cases} 1 + f\left(\left\lfloor \frac{n-1}{2} \right\rfloor \right) , & n > 2 \ ; \\ 0 \ , & n \leq 2 \ . \end{cases}$$

By telescoping this formula a few times,

$$\begin{aligned} f(n) &= 1 + f\left(\left\lfloor \frac{n-1}{2} \right\rfloor \right) \\ &= 1 + 1 + f\left(\frac{\left\lfloor \frac{n-1}{2} \right\rfloor - 1}{2} \right) = 2 + f\left(\left\lfloor \frac{n-3}{4} \right\rfloor \right) \\ &\qquad\qquad = 2 + 1 + f\left(\frac{\left\lfloor \frac{n-3}{4} \right\rfloor - 1}{2} \right) = 3 + f\left(\left\lfloor \frac{n-7}{8} \right\rfloor \right) , \end{aligned}$$

we are led to the formula, easily proven by induction,

$$f(n) = k + f\left(\left\lfloor \frac{n+1}{2^k} \right\rfloor - 1 \right) \quad \text{for} \quad \left\lfloor \frac{n+1}{2^{k-1}} \right\rfloor - 1 > 2 \quad .$$

Since $f(0) = f(1) = f(2) = 0$, this implies that $f(n) = k$, where k is the smallest integer for which $\left\lfloor \frac{n+1}{2^k} \right\rfloor - 1 \leq 2$, or

$$\frac{n+1}{2^k} < 4 \leq \frac{n+1}{2^{k-1}} \quad .$$

This inequality is easily solved for k :

$$2^{k-1} \leq \frac{n+1}{4} < 2^k$$

$$k-1 \leq \lg\left(\frac{n+1}{4}\right) < k$$

$$k = 1 + \left\lfloor \lg\left(\frac{n+1}{4}\right) \right\rfloor$$

$$= \lfloor \lg(n+1) \rfloor - 1 \quad .$$

Therefore the maximum number of subfiles which can be on the stack at any point during the execution of Program 2.3 is $\lfloor \lg(N+1) \rfloor - 1$.

Although Program 2.3 may be a useful implementation of Quicksort in a practical situation, there is one more modification which will improve its performance significantly. We notice that Program 2.3 is not especially efficient for very small files. This is a problem because, for any size file, the recursive nature of the algorithm guarantees that it will be used for many small subfiles. We know a method that is efficient for small files: insertion sorting. This suggests using insertion sorting within Program 2.3 by simply inserting the statements

```
if r-ℓ < M then insertionsort(ℓ,r) else
.
.
.
endif
```

into Program 2.3 where indicated by the asterisks. It is not immediately obvious what the precise value of the parameter M should be: in fact this will be one of the results of our analysis.

Our knowledge of the performance of insertion sorting indicates that there is an even better way to proceed. Suppose that small subfiles (of size $\leq M$) are simply ignored during partitioning. This can be

Example 2.5:

sorting the file (M = 4) :

$$\begin{array}{ccccccccccccccc}
44 & 26 & 95 & 04 & 08 & 88 & 96 & 34 & 07 & 35 & 99 & 24 & 68 & 10 & 01 \\
35 & 26 & 01 & 04 & 08 & 10 & 24 & 34 & 07 & \boxed{44} & 99 & 96 & 68 & 88 & 95 \\
 & & & & & & & & & & 95 & 96 & 68 & 88 & \boxed{99} \\
07 & 26 & 01 & 04 & 08 & 10 & 24 & 35 & \boxed{35} & & & & & & \\
01 & 04 & \boxed{07} & 26 & 08 & 10 & 24 & 34 & & & & & & & \\
 & & & 24 & 08 & 10 & \boxed{26} & 34 & & & & & & & \\
01 & 04 & \boxed{07} & 24 & 08 & 10 & \boxed{26} & 34 & \boxed{35} & \boxed{44} & 95 & 96 & 68 & 88 & \boxed{99} \\
01 & 04 & 07 & 08 & 10 & 24 & 26 & 34 & 35 & 44 & 68 & 88 & 95 & 96 & 99
\end{array}$$

implemented easily in Program 2.3 by replacing the conditions "subfiles both empty" and "small subfile empty" with "subfiles both small" and "small subfile small". Then the file is only partially sorted by the partitioning procedure; but a single insertion sort will complete the job quite efficiently.

Example 2.5 shows what happens with our set of keys when $M = 4$. After partitioning is complete the keys which were used as partitioning elements (07 , 36 , 35 , 44 , 99) are all in their correct positions; and in between any two of these are at most M other keys whose values also fall between the two partitioning elements. This means that the number of inversions in the whole file is the same as the sum of the number of inversions in all of these small subfiles. Therefore, it takes only slightly longer to insertion sort the entire file than to insertion sort all the subfiles. But small subfiles are never put on the stack during partitioning, and the overhead of calling the insertionsort procedure for every small subfile is eliminated, so this method is much more efficient. Program 2.4 is the explicit implementation of this method.

Since this program will be the subject of our analysis, the complete details of stack manipulation, etc., are included. It may be difficult to implement the decision structure following partitioning efficiently in programming languages which don't allow event variables such as "done" in Programs 2.3 and 2.4, and an alternate implementation (which uses go to statements) may be found in Appendix C. Also, the algorithm is "in-line" -- it can easily be made into a subroutine by incorporating the first five lines into a procedure in the manner of Program 2.3.

Program 2.4

```
integer ℓ,r,p,i,j;
integer array stack[0::2 x f(N)-1];
arbmode array A[0::N+1];
arbmode v;
A[0] := -∞; A[N+1] := ∞; ℓ := 1; r := N;
p := 0;
loop until done:
     i := ℓ; j := r+1; v := A[ℓ];                                                          A
     loop:
          loop:  i := i+1; while A[i] < v repeat;                                          C'
          loop:  j := j-1; while A[j] > v repeat;                                          C-C'
     while i < j:
          A[i] :=: A[j];                                                                   B
     repeat;
     A[ℓ] :=: A[j];
     if j-ℓ > r-j then if M ≥ j-ℓ then if p = 0 then done endif;
                                        p := p-2;
                                        ℓ := stack[p]; r := stack[p+1];
                                   else if r-j > M then stack[p] := ℓ; stack[p+1] := j-1;  S'
                                                        p := p+2; ℓ := j+1;
                                                   else r := j-1;
                                        endif;
                       endif;
                  else if M ≥ r-j then if p = 0 then done endif;
                                        p := p-2;
                                        ℓ := stack[p]; r := stack[p+1];
                                   else if j-ℓ > M then stack[p] := j+1; stack[p+1] := r;  S-S'
                                                        p := p+2; r := j-1;
                                                   else ℓ := j+1;
                                        endif;
                       endif;
     endif;
repeat;
i := 2;
loop while i ≤ N:
     if A[i] < A[i-1] then
          v := A[i]; j := i-1;                                                             D
          loop:  A[j+1] := A[j]; j := j-1; while A[j] > v repeat;                          E
          A[j+1] := r;
     endif;
     i := i+1;
repeat;
```

Before we can run Program 2.4 on an actual computer we will need to know the best value of the parameter M, and the amount of storage which should be allocated for the stack. The optimum value of M will be one of the results of the next chapter. The maximum stack depth $f(N)$ is determined exactly as for Program 2.3. For $N \le 2M+2$ there is no stack push, since one of the subfiles is guaranteed to be of size $\le M$. The solution for $N > 2M+2$ is $f(N) = k$, where k is the largest integer for which $\left\lfloor \frac{N+1}{2^k} \right\rfloor - 1 \le 2M+2$. Proceeding exactly as above, we get

$$f(N) = \begin{cases} \left\lfloor \lg\left(\frac{N+1}{M+2}\right) \right\rfloor & N > 2M+2 \\ 0 & N \le 2M+2 \end{cases} .$$

This function also depends on M; but this dependence is left implicit since we will later fix M at a value which minimizes the running time.

This completes our development of the Quicksort algorithm. Program 2.4 is a very efficient sorting program which can be useful in a wide variety of applications. This makes the study of the running time of this program, the subject of the next two chapters, of direct practical interest. Furthermore, the analysis is very interesting in its own right, so our attention will turn now from programming to mathematics.

CHAPTER THREE

The analysis of the average running time of Program 2.4 begins, as with Program 1.3, by counting the number of times each statement is executed and relating these frequencies to basic characteristics of the algorithm. This frequency analysis is most easily done at the assembly language level by repeated application of Kirchhoff's law. Other characteristics of the algorithm may reduce the number of independent variables still further: for example the number of stack "pushes" must equal the number of stack "pops". The running time of Program 2.4 turns out to depend on the six quantities

A -- the number of partitioning stages,

B -- the number of exchanges during partitioning,

C -- the number of comparisons during partitioning,

S -- the number of stack pushes (and pops),

D -- the number of insertions, and

E -- the number of keys moved during insertion.

Each instruction in the assembly language version of Program 2.4 (in Appendix A) is labeled with its frequency. Also Program 2.4 itself is labeled accordingly. Due to symmetries in the algorithm, some of the quantities, for example C' , cancel out when the total running time of the program is computed. This cancellation might not occur in some implementations, and there could therefore be some other quantities involved.

The last two frequency counts listed above, D and E, are the same quantities that we analyzed in Chapter 1. The analysis in Chapter 1 obviously does not apply because the keys are not in random order when the insertion sort program is invoked. However, we know how to describe the nonrandomness in the keys, and we will still be able to study the quantities D and E.

To calculate the average values of our quantities A, B, C, S, D, and E we adopt the same model as in Chapter 1: we assume that the keys to be sorted are the numbers $\{1,2,\ldots,N\}$ and that all permutations of these numbers are equally likely as input. The fact that the subfiles after partitioning also fit this model, a primary concern in our development of a partitioning method in Chapter 2, makes it possible to find these average values by setting up recurrence relations.

For example, let C_N be the average number of "comparisons during partitioning" required by Program 2.4 to sort a random permutation of $\{1,2,\ldots,N\}$. Then this is clearly the average number of comparisons required by the first partitioning stage plus the average number of comparisons required to sort the two subfiles. But it is obvious from Partitioning Method 2.4 and Example 2.4 that the first partitioning stage requires exactly $N+1$ comparisons: There is one comparison each time the i pointer is incremented or the j pointer is decremented. The pointers start with $i = 1$ and $j = N+1$, and when all the keys are distinct, the pointers stop with $i-1 = j = s$ where s is some number between 1 and N. Therefore, i is incremented s times; j is decremented $N-s+1$ times; and the total number of comparisons is $N+1$. (Some other methods require a different number of comparisons depending on the partitioning element. For example, Partitioning Method 2.2 requires N+3 comparisons if the partitioning element is in place, N+2 comparisons otherwise.) If s is the partitioning element, the left subfile has s-1 elements and the right subfile has N-s elements, so we have established that

$$C_N = \sum_{1 \le s \le N} \Pr\{\text{s is the partitioning element}\}(N+1 + C_{s-1} + C_{N-s})$$

$$\text{for } N > M \quad .$$

But we have assumed random input, so that any particular element s is the partitioning element with probability 1/N . This simplifies the equation to

$$C_N = N+1 + \frac{1}{N} \sum_{1 \le s \le N} (C_{s-1} + C_{N-s}) \qquad \text{for } N > M \quad ,$$

which further simplifies to

$$C_N = N+1 + \frac{2}{N} \sum_{1 \le s \le N} C_{s-1} \qquad \text{for } N > M \quad .$$

If a subfile is of size $\le M$, it is not partitioned further, so that we may define

$$C_N = 0 \qquad \text{for } N \le M \; .$$

These two equations define a recurrence relation which can be solved to yield an exact formula for C_N . In particular, we can immediately see that $C_{M+1} = M+2$. To solve for larger values of N , a successful strategy is to eliminate the summation by first multiplying both sides of the recurrence by N :

$$NC_N = N(N+1) + 2 \sum_{1 \le s \le N} C_{s-1} \quad , \qquad \text{for } N > M \quad ;$$

and then subtracting from this equation the same equation for $(N-1)$:

$$NC_N - (N-1)C_{N-1} = N(N+1) - (N-1)N + 2 \sum_{1 \le s \le N} C_{s-1} - 2 \sum_{1 \le s \le N-1} C_{s-1}$$

$$= 2N + 2C_{N-1} \quad , \qquad \text{for } N-1 > M \quad ;$$

or

$$NC_N = (N+1)C_{N-1} + 2N \quad , \qquad \text{for } N > M+1 \quad .$$

This equation can be divided on both sides by the "summation factor" $N(N+1)$ to yield

$$\frac{C_N}{N+1} = \frac{C_{N-1}}{N} + \frac{2}{N+1}$$

which telescopes to a summation

$$\frac{C_N}{N+1} = \frac{C_{M+1}}{M+2} + \sum_{M+2 \le k \le N} \frac{2}{k+1}$$

$$= 1 + 2H_{N+1} - 2H_{M+2} \qquad , \qquad \text{for } N > M \quad .$$

(This technique of reducing a recurrence to a summation by multiplying by a "summation factor" is analogous to solving a differential equation by using an "integrating factor" -- see Eqs. (9) and (10) in Appendix B.) Therefore, the exact formula for the average number of comparisons required by Quicksort is

$$C_N = (N+1)(2H_{N+1} - 2H_{M+2} + 1) \quad , \qquad \text{for } N > M \quad .$$

The average values of all of the other quantities are found in exactly the same manner, although the calculations required are occasionally more complex. For A_N , the average number of partitioning stages, the derivation is simpler. The recurrence

$$A_N = 1 + \frac{1}{N} \sum_{1 \le s \le N} (A_{s-1} + A_{N-s}) \quad , \qquad \text{for } N > M$$

reduces, after eliminating the summation, to

$$\frac{A_N}{N+1} = \frac{A_{N-1}}{N} + \frac{1}{N(N+1)}$$

or

$$\frac{A_N + 1}{N+1} = \frac{A_{N-1} + 1}{N}$$

which telescopes to

$$\frac{A_N + 1}{N+1} = \frac{A_{M+1} + 1}{M+2}$$

$$= \frac{2}{M+2}$$

since $A_N = 0$ for $N \leq M$ and $A_{M+1} = 1$. Therefore, the average number of partitioning stages is

$$A_N = 2\,\frac{N+1}{M+2} - 1 \qquad , \qquad \text{for } N > M \quad .$$

The derivation for the average number of exchanges is more complicated, because the number of exchanges required on the first partitioning stage is dependent not only on the partitioning element, but also on the arrangement of the keys. The recurrence relation is

$$B_N = \frac{1}{N} \sum_{1 \leq s \leq N} \left\{ \begin{matrix} \text{average number of exchanges when} \\ \text{s is the partitioning element} \end{matrix} \right\} + \frac{1}{N} \sum_{1 \leq s \leq N} (B_{s-1} + B_{N-s})$$

By studying Example 2.4 and Partitioning Method 2.4, we can formulate a more specific expression for the first term in this equation. We notice that Quicksort is as efficient as we could expect with respect to exchanges. The only keys exchanged are those which must be exchanged: If $s = A[1]$ is the partitioning element, then only those keys among $A[2],\ldots,A[s]$ which are greater than s are exchanged into the left subfile. The probability that there are exactly t such keys for a given value of s is

$$\frac{\binom{N-s}{t}\binom{s-1}{s-1-t}}{\binom{N-1}{s-1}}$$

so that the average number of exchanges when s is the partitioning element is

$$\begin{aligned}\sum_{0 \le t \le s-1} t \frac{\binom{N-s}{t}\binom{s-1}{s-1-t}}{\binom{N-1}{s-1}} &= \frac{N-s}{\binom{N-1}{s-1}} \sum_t \binom{N-s-1}{t-1}\binom{s-1}{s-1-t} \\ &= \frac{N-s}{\binom{N-1}{s-1}} \binom{N-2}{s-2} \\ &= \frac{(N-s)(s-1)}{N-1} \quad .\end{aligned}$$

(The identity $\sum_t \binom{N-s-1}{t-1}\binom{s-1}{s-1-t} = \binom{N-2}{s-2}$ is an instance of Vandermonde's convolution -- see Eq. (21) in Appendix B.) The recurrence for the total average number of exchanges is therefore

$$B_N = \frac{1}{N(N-1)} \sum_{1 \le s \le N} (N-s)(s-1) + \frac{2}{N} \sum_{1 \le s \le N} B_{s-1}$$

$$= \frac{1}{N(N-1)} \binom{N}{3} + \frac{2}{N} \sum_{1 \le s \le N} B_{s-1}$$

$$= \frac{N-2}{6} + \frac{2}{N} \sum_{1 \le s \le N} B_{s-1} \qquad .$$

This is a linear combination of the equations we have just solved, so that

$$B_N = \frac{1}{6} C_N - \frac{1}{2} A_N \quad , \qquad \text{or}$$

$$B_N = (N+1)\left(\frac{1}{3} H_{N+1} - \frac{1}{3} H_{M+2} + \frac{1}{6} - \frac{1}{M+2} \right) + \frac{1}{2} \quad , \quad \text{for } N > M \; .$$

Notice that since $B_N/C_N \approx 1/6$, the i and j pointers move in about three elements at a time on the average. This fact is of interest when we consider the problem of "tuning" the program to highest efficiency for particular computers (see Appendix A).

The fourth quantity describing the running time of the partitioning portion of Program 2.4 is S , the number of stack pushes. This differs from the other quantities that we have studied because its value is always 0 for $N \le 2M+2$ (rather than for $N \le M$). As we noted in Chapter 2, the algorithm is designed to save a subfile on the stack only when necessary. If $N \le 2M+2$, then one of the subfiles is guaranteed to be of size $\le M$ and will be left for the insertion sort, while the other is used for the next partitioning stage. For $N > 2M+2$, there will be no stack push if the partitioning element is $1,2,\ldots,M+1$ since the left subfile can then be left for the insertion sort; and there will be no stack push if the partitioning element is $N-M, n-M+1, \ldots, N-1, N$ since the right subfile can then be left for the insertion sort. If the partitioning element is in the range $M+2, M+3, \ldots, N-M-2, N-M-1$ then both subfiles are bigger than the threshold, and one must be saved on the stack. This argument means that the recurrence

$$S_N = \frac{1}{N} \sum_{1 \le s \le N} \left\{ \begin{matrix} \text{average number of stack pushes when} \\ \text{s is the partitioning element} \end{matrix} \right\}$$

$$+ \frac{1}{N} \sum_{1 \le s \le N} (S_{s-1} + S_{N-s})$$

can be simplified to

$$S_N = \frac{1}{N}(N - 2M+2) + \frac{2}{N} \sum_{1 \le s \le N} S_{s-1} \quad , \quad \text{for} \quad N \ge 2M+3 \quad .$$

In particular, since $S_N = 0$ for $0 \le N \le 2M+2$, then $S_{2M+3} = \frac{1}{2M+3}$. This reduces to the same equation that we had for A_N ,

$$\frac{S_N + 1}{N+1} = \frac{S_{N-1} + 1}{N} \quad ,$$

except that the recurrence does not telescope as far:

$$\frac{S_N + 1}{N+1} = \frac{S_{2M+3} + 1}{2M+4} = \frac{\frac{1}{2M+3} + 1}{2M+4} = \frac{1}{2M+3} \quad .$$

The solution is therefore

$$S_N = \frac{N+1}{2M+3} - 1 \quad , \quad \text{for} \quad N \ge 2M+3 \quad .$$

Finally we must look at D and E , the quantities describing the time taken by the insertion sort. We can use the same devices as in Chapter 1 to describe these quantities. The quantity D is the number of non-zero elements in the inversion table of the permutation remaining after partitioning, and the quantity E is the number of inversions in the permutation after partitioning.

The definition of partitioning makes the recurrence relations for these quantities especially simple. Since the partitioning element s is put into position, the inversion table entry B_s for the permutation left after partitioning is 0 . Further, if an inversion table entry for some element in either subfile is non-zero, it must be because there is a larger element to its left in the same subfile. The number of non-zero elements in the inversion table for the whole file is the sum of the number of non-zero elements in the inversion tables of the subfiles. Similarly the sum of the inversion table entries for the whole file (the number of inversions) is the sum of the sums of the inversion table entries for the two subfiles. Therefore, for $N > M$, we know that

$$D_N = \frac{1}{N} \sum_{1 \leq s \leq N} (D_{s-1} + D_{N-s}) = \frac{2}{N} \sum_{1 \leq s \leq N} D_{s-1}$$

and

$$E_N = \frac{1}{N} \sum_{1 \leq s \leq N} (E_{s-1} + E_{N-s}) = \frac{2}{N} \sum_{1 \leq s \leq N} E_{s-1} \quad .$$

These reduce quite simply to

$$\frac{D_N}{N+1} = \frac{D_{N-1}}{N} = \cdots = \frac{D_{M+1}}{M+2}$$

and

$$\frac{E_N}{N+1} = \frac{E_{N-1}}{N} = \cdots = \frac{E_{M+1}}{M+2} \quad .$$

The subfiles produced of size $\leq M$ are random by our assumptions, so that the results of Chapter 1 do apply for $N \leq M$:

$$D_N = N - H_N$$

and

$$E_N = \frac{N(N-1)}{4} \quad .$$

We now use these expressions to compute the average values for $N = M+1$. The telescoped recurrences above will then give the formulas for all N. First,

$$D_{M+1} = \frac{2}{M+1} \sum_{1 \le s \le M+1} D_{s-1}$$

$$= \frac{2}{M+1} \sum_{0 \le s \le M} (s - H_s)$$

$$= \frac{2}{M+1} \frac{M(M+1)}{2} - \frac{2}{M+1} ((M+1)H_M - M)$$

(see Eqs. (4) and (19) in Appendix B), which simplifies to

$$= M + 2 - 2H_{M+1} \quad ,$$

and leads to the solution

$$D_N = N+1 - 2 \frac{N+1}{M+2} H_{M+1} \quad , \quad \text{for} \quad N > M \quad .$$

Similarly,

$$E_{M+1} = \frac{2}{M+1} \sum_{1 \le s \le M+1} E_{s-1}$$

$$= \frac{2}{M+1} \sum_{0 \le s \le M} \frac{s(s-1)}{4}$$

$$= \frac{1}{M+1} \sum_{0 \le s \le M} \binom{s}{2}$$

$$= \frac{1}{M+1} \binom{M+1}{3}$$

$$= \frac{M(M-1)}{6}$$

and the average number of inversions left after partitioning is

$$E_N = \frac{(N+1)M(M-1)}{6(M+2)} \quad , \quad \text{for} \quad N > M \quad .$$

We have now found the average values of all quantities upon which the running time of Program 2.4 depends. To summarize, we know that the Quicksort program requires, on the average,

$$A_N = 2\,\frac{N+1}{M+2} - 1 \qquad \text{stages,}$$

$$B_N = (N+1)\left(\frac{1}{3}\,H_{N+1} - \frac{1}{3}\,H_{M+2} + \frac{1}{6} - \frac{1}{M+2}\right) + \frac{1}{2} \qquad \text{exchanges,}$$

$$C_N = (N+1)(2H_{N+1} - 2H_{M+2} + 1) \qquad \text{comparisons,}$$

$$D_N = (N+1)\left(1 - 2\,\frac{H_{M+1}}{M+2}\right) \qquad \text{insertions,}$$

$$E_N = (N+1)\,\frac{M(M-1)}{6(M+2)} \qquad \text{moves during insertion, and}$$

$$S_N = \frac{N+1}{2M+3} - 1 \qquad \text{stack pushes.}$$

These formulas all hold for $N > 2M+2$. For $M < N \leq 2M+2$, we know that $S_N = 0$, and all of the other formulas are still valid. For $N \leq M$, we have defined $A_N = B_N = C_N = S_N = 0$; $D_N = N - H_N$; and $E_N = \frac{N(N-1)}{4}$. These definitions made it convenient to solve our recurrence relations, but the analysis is not entirely accurate if the initial file to be sorted is of size smaller than M . Since we are interested primarily in the performance of the program for large values of N , we will pass over these details and work with the formulas for $N > 2M+2$.

We notice that the largest of these terms are B_N and C_N, the number of exchanges and comparisons. The harmonic numbers behave like logarithms, so that these are the " N lg N " terms which dominate the running time of Quicksort. The assembly language implementation of Program 2.4, which is given in Appendix A, requires a total time of

$$24A + 11B + 4C + 3D + 8E + 9S + 7N \quad .$$

Again, these coefficients are only representative, and they may vary from computer to computer. (Appendix C is a discussion of the implementation on some real computers.) The significant thing about them is that the coefficients of B and especially C are very small. This is important because we know that these quantities dominate the total average running time. It is important to note that this is what makes Quicksort "quick". Not only does its average running time depend on quantities that behave like N lg N , but also the coefficients of the quantities are small. The coefficient of the number of comparisons counts a compare, a pointer increment, and a conditional jump. It is hard to imagine a simpler inner loop (although the technique of "loop unwrapping" will reduce the overhead per comparison even further -- see Appendix A). This point may seem obvious, but it is often overlooked when programs are being analyzed. Insertion sorting is the best method for small files for the same reason: the coefficient of N^2 in the expression for its running time is small.

Substituting the formulas for the average values of the quantities A , B , C , D , E , and S into the expression for the total time of Program 2.4, we get:

$$\frac{35}{3}(N+1)H_{N+1} - \frac{69}{2} + \frac{1}{6}(N+1)\left(8M + 71 - 70H_{M+2} + \frac{270}{M+2} + \frac{54}{2M+3} - 36\,\frac{H_{M+1}}{M+2}\right)$$

for the expected running time of Quicksort. From this exact formula, we can compute the best value of the parameter M . The graph of the function

$$f(M) = 8M + 71 - 70H_{M+2} + \frac{270}{M+2} + \frac{54}{2M+3} - 36\frac{H_{M+1}}{M+2}$$

is shown in Figure 3.1 -- it takes on its minimum value when $M = 9$. Although the value of M does not affect the leading term of the expression for the total running time, the proper choice of this value does have a significant effect in practical situations, because when N is in a practical range $f(M)$ is the same order of magnitude as H_{N+1}. The graph of the total running time for $N = 10{,}000$ is shown in Figure 3.2, as a function of different values of M. The optimal value of M results in a 12 to 20 percent savings over the time taken by more naive implementations of the algorithm. (For example, Program 2.3, which is essentially Program 2.4 with $M = 1$, is about 18% slower when $N = 1000$ and 14% slower when $N = 10{,}000$.)

Even if the exact values of the coefficients of the quantities which affect the running time of a Quicksort program are not known, it is wise to avoid the partitioning of small subfiles, for this analysis shows that the precise choice of the parameter M is not highly critical. For Program 2.4 any value of M between 5 and 20 would do about as well as $M = 9$. We can expect in general that if our implementation involves a large amount of overhead, a larger value of M should be used, and Figure 3.2 shows that it will not hurt much to pick a value of M that is higher than the optimum.

We can cast our result for the average running time of Program 2.4 in more conventional terms by using Eq. (52) from Appendix B,

$$H_N = \ln N + \gamma + \frac{1}{2N} + O\left(\frac{1}{N^2}\right) \quad ,$$

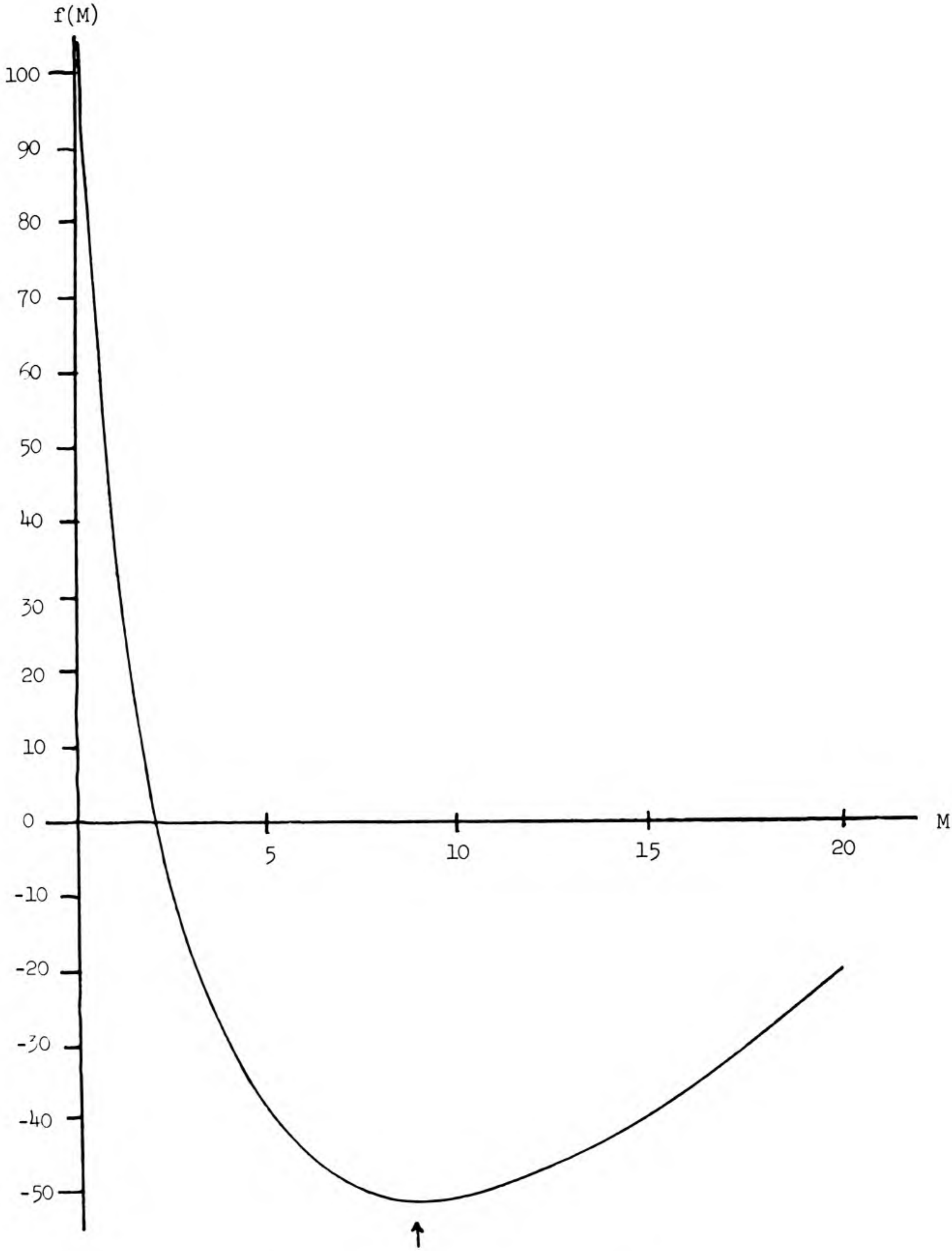


Figure 3.1. Contribution of M .

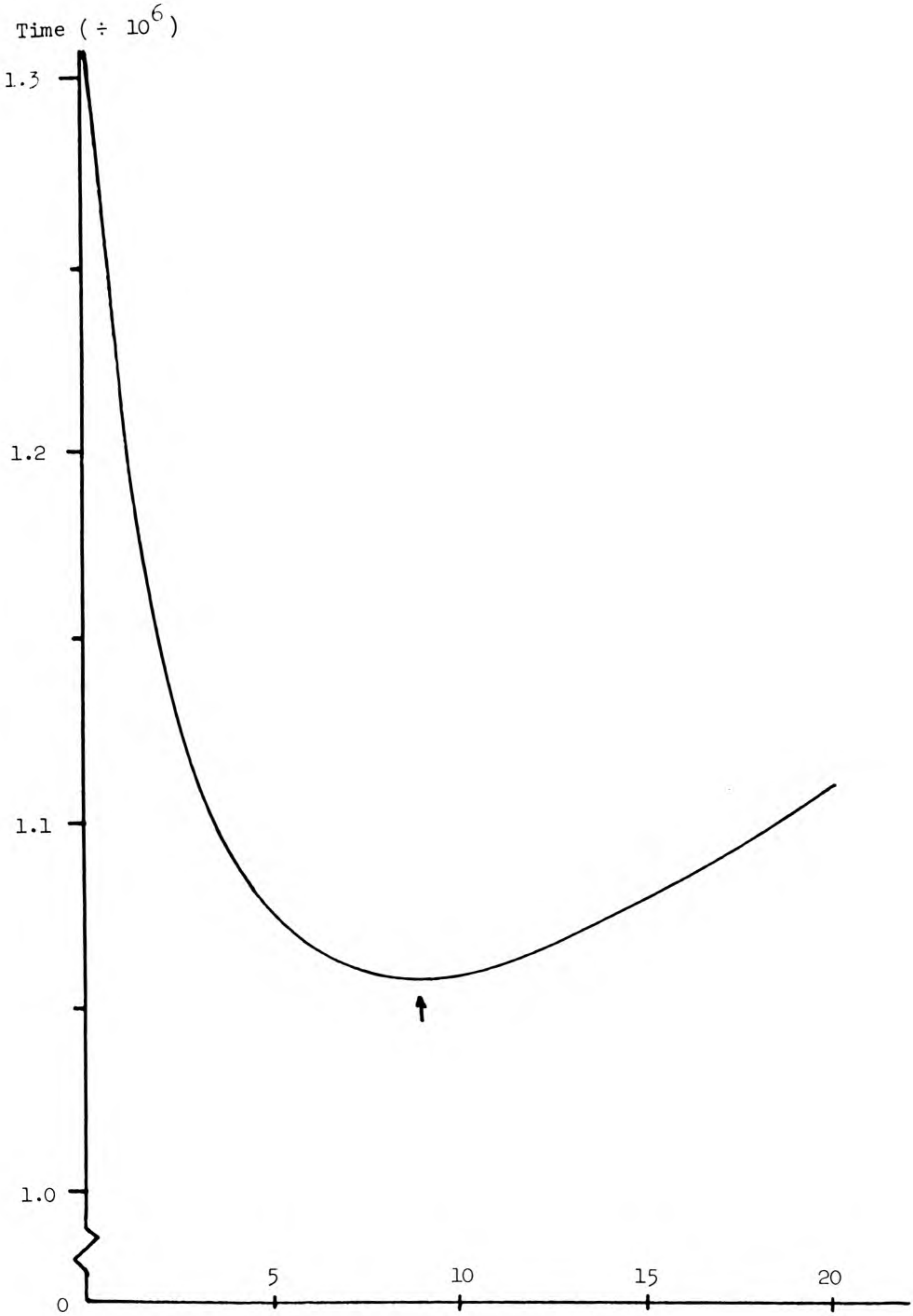


Figure 3.2. Total running time for N = 10,000 .

in the following asymptotic calculations on our exact formula:

$$\frac{35}{3}(N+1)H_{N+1} - \frac{69}{2} + \frac{1}{6}(N+1)f(M)$$

$$= \frac{35}{3}(N+1)\left(\ln N + \gamma + \frac{1}{2N} + O\left(\frac{1}{N^2}\right) + \frac{1}{N+1}\right) - \frac{69}{2} + \frac{1}{6}(N+1)f(M)$$

$$= \frac{35}{3}(N+1)\ln N + (N+1)\left(\frac{35}{3}\gamma + \frac{1}{6}f(M)\right) - 17 + O\left(\frac{1}{N}\right) \quad .$$

This, after we substitute $\gamma = .57721\ldots$ and the optimum value $f(9) = -50.860\ldots$, gives an approximate formula

$$11.67(N+1)\ \ln N - 1.74N - 18.74$$

or

$$8.09(N+1)\ \lg N - 1.74N - 18.74$$

for the average total running time of Quicksort. We can have confidence in this approximation: because we started with an exact formula, we can carry out the derivation to any asymptotic accuracy we desire.

The calculation of the variance of the total running time of Program 2.4 is far too long and involved to be presented here. Although it is very important to have an approximate idea of the magnitude of the standard deviation, the exact value of the variance is, from a practical standpoint, the least interesting of the quantities we study in the analysis of algorithms. Moreover, it is the most difficult to get -- the calculations involved in deriving variances are often very intricate. To illustrate this, we will study the variance of the number of comparisons required by Quicksort. The derivation of an exact formula for this involves a variety of sums involving harmonic numbers, most of which are worked out in Appendix B.

The most convenient way to set up the problem is in terms of generating functions. Let c_{Nk} be the probability that Program 2.4 uses exactly k comparisons to partition a random permutation of $\{1,2,\ldots,N\}$, and let $C_N(z) = \sum_{k\geq 0} c_{Nk}\, z^k$ be the generating function for $\{c_{Nk}\}$. Suppose that s is the partitioning element for the first stage, $1 \leq s \leq N$. Then $C_{s-1}(z)$ is the generating function describing the number of comparisons Program 2.4 uses for the left subfile, $C_{N-s}(z)$ is the generating function for the right subfile, and z^{N+1} describes the number of comparisons used in the first partitioning stage. Since these are all independent, we can multiply them to get

$$z^{N+1}C_{s-1}(z)C_{N-s}(z) = \sum_{k\geq 0} \Pr\left\{\begin{matrix}\text{Program 2.4 uses exactly k comparisons}\\ \text{when } s \text{ is the partitioning element}\end{matrix}\right\} z^k \quad .$$

Let us denote the conditional probability on the right hand side by c_{Nks}. Then $\sum_{1\leq s\leq N} \Pr\{\text{s is the partitioning element}\}c_{Nks} = c_{Nk}$, so that we can remove this condition by multiplying both sides of the above equation by $\frac{1}{N} = \Pr\{\text{s is the partitioning element}\}$ and summing over all s :

$$\begin{aligned}\frac{1}{N}\, z^{N+1} \sum_{1\leq s\leq N} C_{s-1}(z)C_{N-s}(z) &= \sum_{k\geq 0}\ \sum_{1\leq s\leq N} \Pr\left\{\begin{matrix}\text{s is the}\\ \text{partitioning element}\end{matrix}\right\} c_{Nks}\, z^k \\ &= \sum_{k\geq 0} c_{Nk}\, z^k \\ &= C_N(z) \quad .\end{aligned}$$

As before, this will hold for $N > M$; for $N \leq M$, $C_N(z) = 1$ since no "comparisons during partitioning" are used:

$$C_N(z) = \begin{cases} \frac{1}{N} z^{N+1} \sum_{1 \le s \le N} C_{s-1}(z) C_{N-s}(z) \quad , & N > M \quad ; \\ \\ 1 \quad , & N \le M \quad . \end{cases}$$

This is a probability generating function for all N, and the recurrence appears to be difficult to solve explicitly. However, it does provide enough information to compute the average and variance, since it leads directly to recurrences in the derivatives of the generating function, evaluated at $z = 1$. Since

$$C'_N(z) = \frac{1}{N} (N+1) z^N \sum_{1 \le s \le N} C_{s-1}(z) C_{N-s}(z)$$

$$+ \frac{1}{N} z^{N+1} \sum_{1 \le s \le N} (C'_{s-1}(z) C_{N-s}(z) + C_{s-1}(z) C'_{N-s}(z)) \quad ,$$

the average is given by

$$C'_N(1) = \begin{cases} N + 1 + \frac{1}{N} \sum_{1 \le s \le N} (C'_{s-1}(1) + C'_{N-s}(1)) & N > M \\ \\ 0 & N \le M \quad . \end{cases}$$

This is exactly the recurrence that we have solved already for the average number of comparisons required by Program 2.4. To simplify the calculations as much as possible, we will take $M = 0$, so that the solution is

$$C'_N(1) = 2(N+1)(H_{N+1} - 1) \quad ,$$

which holds for all N. To get the variance, we first proceed in the same manner to calculate $C''_N(1)$. The second derivative of the generating function satisfies

$$C_N''(z) = \frac{1}{N}\,(N+1)\,N\,z^{N-1} \sum_{1\le s\le N} C_{s-1}(z)C_{N-s}(z)$$

$$+ \frac{2}{N}\,(N+1)\;z^{N} \sum_{1\le s\le N} (C'_{s-1}(z)C_{N-s}(z) + C_{s-1}(z)C'_{N-s}(z))$$

$$+ \frac{1}{N}\,z^{N+1} \sum_{1\le s\le N} (C''_{s-1}(z)C_{N-s}(z) + 2C'_{s-1}(z)C'_{N-s}(z) + C''_{N-s}(z))$$

so that the recurrence for $C_N''(1)$ is

$$C_N''(1) = N(N+1) + \frac{4}{N}\,(N+1) \sum_{1\le s\le N} C'_{s-1}(1)$$

$$+ \frac{2}{N} \sum_{1\le s\le N} C'_{s-1}(1)C'_{N-s}(1) + \frac{2}{N} \sum_{1\le s\le N} C''_{s-1}(1) \quad .$$

This is essentially the same as, but much more complicated than, the recurrences that we solved to get our average values. The same method of solution is appropriate. First we multiply by N and subtract the same equation for $(N-1)$:

$$N\,C_N''(1) - (N-1)C''_{N-1}(1) = N^2(N+1) - (N-1)^2\,N$$

$$+ 4(N+1) \sum_{1\le s\le N} C'_{s-1}(1) - 4N \sum_{1\le s\le N-1} C'_{s-1}(1)$$

$$+ 2 \sum_{1\le s\le N} C'_{s-1}(1)C'_{N-s}(1)$$

$$- 2 \sum_{1\le s\le N-1} C'_{s-1}(1)C'_{N-1-s}(1)$$

$$+ 2\,C''_{N-1}(1) \quad .$$

After simplifying and rearranging some terms, we have

$$N C''_N(1) - (N+1)C''_{N-1}(1)$$

$$= N(3N-1) + 4(N+1)C'_{N-1}(1) + 2 \sum_{1 \le s \le N-1} C'_{s-1}(1)(C'_{N-s}(1) - C'_{N-s-1}(1) + 2) \ ,$$

into which we can substitute the known expression for $C'_N(1)$:

$$N C''_N(1) - (N+1)C''_{N-1}(1) = N(3N-1) + 8(N+1)N(H_{N-1}) + 2 \sum_{1 \le s \le N-1} 2s(H_s-1)(2H_{N-s} + 2)$$

$$= N(3N-1) + 8(N+1)N(H_{N-1}) + 8 \sum_{1 \le s \le N-1} s H_s H_{N-s}$$

$$+ 8 \sum_{1 \le s \le N-1} s H_s - 8 \sum_{1 \le s \le N-1} s H_{N-s} - 4(N-1)N \ .$$

These sums involving harmonic numbers are tricky to evaluate -- symmetry is used to reduce them to the basic sums given by Eqs. (8) and (23) in Appendix B.

$$2 \sum_{1 \le s \le N-1} s H_s H_{N-s} = \sum_{1 \le s \le N-1} s H_s H_{N-s} + \sum_{1 \le s \le N-1} (N-s)H_{N-s} H_s$$

$$= N \sum_{1 \le s \le N-1} H_s H_{N-s}$$

$$= N(N+1)(H_N^2 - H_N^{(2)}) - 2N^2 H_N + 2N^2 \quad ,$$

$$\sum_{1 \le s \le N-1} s\ H_s = \binom{N}{2}\left(H_N - \frac{1}{2}\right) \quad ,$$

and

$$\sum_{1 \le s \le N-1} s\, H_{N-s} = N \sum_{1 \le s \le N-1} H_s - \sum_{1 \le s \le N-1} s\, H_s$$

$$= N(N\, H_N - N) - \binom{N}{2}\left(H_N - \frac{1}{2} \right) \quad .$$

Substituting these expressions gives

$$N\, C''_N(1) - (N+1)C''_{N-1}(1) = N(3N-1) + 8(N+1)N(H_{N-1}) + 4N(N+1)(H_N^2 - H_N^{(2)})$$

$$- 8N^2 H_N + 8N^2 + 8N(N-1)H_N - 4N(N-1)$$

$$- 8N^2 H_N + 8N^2 - 4(N-1)N$$

$$= N(3N-1) + 4N(N+1)(H_N^2 - H_N^{(2)}) \quad .$$

If both sides of this equation are divided by $N(N+1)$, the result is the telescoping recurrence

$$\frac{C''_N(1)}{N+1} = \frac{C''_{N-1}(1)}{N} + \frac{3N-1}{N+1} + 4H_N^2 - 4H_N^{(2)}$$

which has the solution

$$\frac{C''_N(1)}{N+1} = C''_0(1) + \sum_{1 \le k \le N} \left(\frac{3k-1}{k+1} + 4H_k^2 - 4H_k^{(2)} \right)$$

$$= 3N - 4H_{N+1} + 4 \sum_{1 \le k \le N} H_k^2 - 4 \sum_{1 \le k \le N} H_k^{(2)} \quad .$$

Using Eqs. (5) and (6) from Appendix B, we have

$$\frac{C''_N(1)}{N+1} = 3N - 4H_{N+1} + 4(N+1)H_N^2 - 4(2N+1)H_N + 8N - 4(N+1)H_{N+1}^{(2)} - 4H_N \quad ,$$

or

$$C''_N(1) = 11N(N+1) - 4 - 4(N+1)(N+3)H_N + 4(N+1)^2(H_N^2 - H_{N+1}^{(2)}) \quad .$$

Finally, the variance is given by

$$\begin{aligned} C''_N(1) + C'_N(1) - C'_N(1)^2 = {} & 11N(N+1) - 4 - 4(N+1)(2N+3)H_N + 4(N+1)^2(H_N^2 - H_{N+1}^{(2)}) \\ & + 2(N+1)(H_{N+1} - 1) \\ & - 4(N+1)^2(H_{N+1} - 1)^2 \end{aligned}$$

$$\mathrm{var}(C_N) = 7N^2 + 9N - 4 - 10(N+1)H_N - 4(N+1)^2 H_{N+1}^{(2)} \quad .$$

Since $H_{N+1}^{(2)}$ is asymptotically $\frac{\pi^2}{6}$, (see Eq. (53) in Appendix B), this means that the asymptotic formula for the variance is

$$\mathrm{var}(C_N) = (7 - \tfrac{2}{3}\pi^2)N^2 + O(N \ln N) \quad .$$

In principle, we could proceed as in Chapter 1 to find the variance of the total running time, using exactly the same method. The exact solution is quite long and involved, but even for general M we can derive the asymptotic formula above. Also, none of the other quantities has a variance of the same order as C_N , and so we know that the standard deviation of the running time of Program 2.4 is approximately .68N . This is enough information to give us confidence in the stability of the formula that we have derived for the average running time of Quicksort.

CHAPTER FOUR

In this chapter we will look at the operation of Program 2.4 in the best case and in the worst case. Interactions between the quantities involved make this analysis more complex than was the discussion for the best and worst case of insertion sorting. As we will see, for Quicksort the input permutation which leads to the highest possible number of comparisons requires no exchanges, so that it is not at all clear what the worst case of the whole algorithm is. Such interference among the quantities makes it unwise to rely on intuition, and we will try to develop our results carefully.

Our strategy to solve for the average running time was to set up a recurrence relation by conditioning on the element used for the first partitioning stage. This approach is also attractive for the analysis of the worst case. Let T_N be the time taken by Program 2.4 in the worst case, out of all permuations of $\{1,2,\ldots,N\}$. Then we have the formula

$$T_N = \max_{1 \le s \le N} \{\text{time taken} \mid s \text{ is used as the partitioning element}\} .$$

As before, we will set up a recurrence relation for this quantity by looking at what happens during the first partitioning stage. Since we are dealing with the total time taken, it is convenient to subtract off the time which is independent of how partitioning is done, and to write the recurrence in terms of the quantity $T_N - 7N$:

$$T_N - 7N = \max_{1 \le s \le N} (\{\text{time taken for the first partition, using } s\} + T_{s-1} - 7(s-1) + T_{N-s} - 7(N-s)) \quad , \quad \text{for} \quad N > M \quad .$$

After inspecting the program, we are lead to the equation

$$T_N = \max_{1 \le s \le N} (24A + 11B + 4C + 9S + 7 + T_{s-1} + T_{N-s}) \qquad \text{for } N > M ,$$

where A , B , C , and S denote the contribution of the first partitioning stage to the quantities we defined in Chapter 3. We found during our derivation of the average running time that these contributions, if s is the partitioning element, are

$$A = 1 , \quad B = t , \quad C = N+1 , \text{ and } S = \Delta_{sNM} , \quad \text{for } 1 \le s \le N .$$

Here Δ_{sNM} is defined to be 1 if there is a stack push:

$$\Delta_{sNM} = \begin{cases} 1 & M+2 \le s \le N-M-1 \\ 0 & \text{otherwise} \end{cases} ,$$

and t is the number of keys among A[2],...,A[s] which are greater than s . If all of the keys A[2],...,A[s] are greater than s , or if all of the keys greater than s are among A[2],...,A[s] , then t assumes its maximum value, namely min(s-1 , N-s) . This occurs independently of the values of the other quantities, and independently of what happens in the subfiles.

Substituting all of these values into our recurrence, we have

$$T_N = \max_{1 \le s \le N} (24 + 11 \min(s-1 , N-s) + 4(N-1) + 9\Delta_{sNM} + T_{s-1} + T_{N-s} + 7)$$

$$= \max_{1 \le s \le N} (11 \min(s-1 , N-s) + T_{s-1} + T_{N-s} + 9\Delta_{sNM}) + 4N + 35 , \quad \text{for } N > M .$$

The contributions of the quantities D and E are accounted for in our equation for small N :

$$T_N = \max(3D + 8E + 7N) \qquad \text{for } N \le M .$$

Here the maximum is taken over all permutations of $\{1,2,\ldots,N\}$. Since Program 2.4 will actually make one partition on a file of initial size $\le M$, this equation is not entirely accurate; but defining it in this way makes the recurrence for $N > M$ correct. Now, we know from Chapter 1 that D and E both take on their maximum values when a permutation in reverse order is sorted, and that these maximum values are $D = N-1$ (0 if $N = 0$) and $E = \frac{1}{2} N(N-1)$. This leads to a complete recurrence for the worst case running time:

$$T_N = \begin{cases} \max\limits_{1 \le s \le N} (11 \min(s-1, N-s) + T_{s-1} + T_{N-s} + 9\Delta_{sNM}) + 4N + 35 , & N > M ; \\ 4N^2 + 6N - 3 , & 1 \le N \le M ; \\ 0 , & N = 0 . \end{cases}$$

The recurrence for the <u>best</u> case is derived in an entirely analogous manner. Let U_N be the time taken by Program 2.4 in the best case of all input permutations of $\{1,2,\ldots,N\}$. Then the same argument as above says that

$$U_N = \begin{cases} \min\limits_{1 \le s \le N} (24A + 11B + 4C + 9S + U_{s-1} + U_{N-s} + 7 , & N > M ; \\ \min(3D + 8E + 7N) , & N \le M . \end{cases}$$

We know that the best case for subfiles of length $\le M$ is to have them in order $(D = E = 0)$; and the best case for B is to have no exchanges at all (i.e., when all of the $s-1$ keys $A[2], A[3], \ldots, A[s]$ are less than s). This gives, as above,

$$U_N = \begin{cases} \min\limits_{1 \le s \le N} (U_{s-1} + U_{N-s} + 9\Delta_{sNM}) + 4N + 35 , & N > M \\ 7N , & N \le M . \end{cases}$$

Let us now attempt to solve these recurrences. We will begin with T_N, the easier of the two. The quantity to be maximized in the expression for T_N is symmetric about $s = N+1-s$, so we need only include the terms $1 \le s \le \frac{N+1}{2}$. For all of these terms, we know that $s-1 \le N-s$, so that

$$T_N = \begin{cases} \max\limits_{1 \le s \le \frac{N+1}{2}} (11(s-1) + T_{s-1} + T_{N-s} + 9\Delta_{sNM}) + 4N + 35 , & N > M ; \\ 4N^2 + 6N - 3 , & 1 \le N \le M ; \\ 0 , & N = 0 . \end{cases} \quad (*)$$

We expect that Quicksort should perform worst for $N > M$ when it chooses the smallest or the largest element as the partitioning element $(s = 1$ or $s = N)$. When the partitioning trees are degenerate in this way we can easily compute the values of the various quantities. Again, because of symmetry, we will treat the case where the maximum in the expression above occurs at $s = 1$: the smallest element is used as the partitioning element at every partitioning step. Then the recurrence becomes

$$T_N = \begin{cases} T_{N-1} + 4N + 35 , & N > M ; \\ 4N^2 + 6N - 3 , & 1 \le N \le M ; \end{cases}$$

which is easily solved:

$$T_N = T_M + \sum_{M+1 \leq k \leq N} (4k + 35)$$

$$= 4M^2 + 6M - 3 + 2N(N+1) - 2M(M+1) + 35N - 35M$$

$$T_N = 2N^2 + 37N + 2M^2 - 31M - 3 \quad , \qquad \text{for} \quad N \geq M \geq 1 \quad .$$

We can similarly solve for the various quantities, and we find that, when the smallest element is used as the partitioning element, Program 2.4 requires for partitioning

$$A = \sum_{M+1 \leq k \leq N} 1 = N - M \qquad \text{stages,}$$

$$B = 0 \qquad \text{exchanges,}$$

$$C = \sum_{M+1 \leq k \leq N} (k+1) = \binom{N+2}{2} - \binom{M+2}{2} \qquad \text{comparisons, and}$$

$$S = 0 \qquad \text{stack pushes.}$$

Neither B nor S are maximized in this case (in fact they are both minimized) so that it is necessary to prove this intuitive result. There might possibly be a permutation for which Program 2.4 requires slightly fewer stages or comparisons, but many more exchanges or stack pushes. Fortunately, however, we can prove the following:

Theorem 4.1. The worst way to partition is to always choose the smallest element as the partitioning element.

Proof. This is equivalent to showing that the maximum in the recurrence (*) always occurs at $s = 1$. The induction proof of this is more complicated than seems necessary, because the function T_N is not quite convex.

First, for $N = M+1$, we have

$$T_{M+1} = \max_{1 \leq s \leq \frac{M+2}{2}} (11(s-1) + T_{s-1} + T_{M+1-s}) + 4(M+1) + 35 .$$ If $M = 1$, then $s = 1$ is the only value in the range $1 \leq s \leq \frac{M+2}{2}$. Otherwise, if $M \geq 2$, then since $T_0 = 0$ this expression has the value $T_M + 4(M+1) + 35$ for $s = 1$, so we need only show that

$$T_M \geq 11(s-1) + T_{s-1} + T_{M+1-s} \qquad \text{for} \quad 2 \leq s \leq \frac{M+2}{2} .$$

We know the values of T_M, T_{s-1}, and T_{M+1-s} from (*). Substituting these expressions gives

$$4M^2 + 6M - 3 \geq 11(s-1) + 4(s-1)^2 + 6(s-1) - 3 + 4(M+1-s)^2 + 6(M+1-s) - 3$$

or

$$8s^2 - (8M+5)s + 8M - 6 \leq 0 .$$

This inequality clearly holds because the convex parabola $8s^2 - (8M+5)s + 8M - 6$ must take on its maximum in the interval $2 \leq s \leq \frac{M+2}{2}$ at one of the endpoints. For $s = 2$ the value is $-8M + 16$ and for $s = \frac{M+2}{2}$ it is $\frac{1}{2}(M-2)(3-4M)$. These are both ≤ 0, so the theorem holds for $N = M+1$.

If the maximum occurs at $s = 1$ for all $n < N$, then we know from the discussion above that

$$T_n = \begin{cases} 2n^2 + 37n + 2M^2 - 31M - 3 & n > M \\ 4n^2 + 6n - 3 & 1 \le n \le M \\ 0 & n = 0 \quad . \end{cases}$$

To prove the theorem, we want to show that

$$\max_{1 \le s \le \frac{N+1}{2}} (11(s-1) + T_{s-1} + T_{N-s} + 9\Delta_{sNM})$$ occurs at $s = 1$. In other words, it is sufficient to show that

$$T_{N-1} \ge 11(s-1) + T_{s-1} + T_{N-s} + 9 \qquad \text{for} \quad 2 \le s \le \frac{N+1}{2} \quad .$$

We can use the inductive hypothesis to get expressions for T_{s-1} and T_{N-s} here, but the proof breaks down into four cases, depending on whether $s-1$ and $N-s$ are greater than or $\le M$. The calculations involved are straightforward algebraic manipulations, so we do only one as an example. If $s-1 \le M$ and $N-s > M$, then we want to show that

$$2(N-1)^2 + 37(N-1) + 2M^2 - 31M - 3 \ge 11(s-1) + 4(s-1)^2 + 6(s-1) - 3$$
$$+ 2(N-s)^2 + 37(N-s) + 2M^2 - 31M - 3 + 9$$

which simplifies to

$$6(s-1)^2 - 20(s-1) - 4(N-1)(s-1) + 6 \le 0 \quad .$$

Again we have a convex parabola (in the variable $s-1$) which takes on its maximum in the interval $1 \le s-1 \le \frac{N-1}{2}$ at the endpoints, and a quick calculation shows that this maximum is negative in both cases. Similar arguments hold for the other three cases: $s-1 \le M$

and $N\text{-}s \leq M$; $s\text{-}1 > M$ and $N\text{-}s > M$; and $s\text{-}1 > M$ and $N\text{-}s \leq M$.

□

The worst case running time is therefore described by two parabolas: $4N^2 + 6N - 3$ for $1 \leq N \leq M$; and $2N^2 + 37N + 2M^2 - 31M - 3$. These curves intersect when $(N\text{-}M)(31 - 2(N + M)) = 0$; that is, when $N = M$ or $N = 15\frac{1}{2} - M$. This explains the complication in the proof of Theorem 4.1 -- for $M \geq 8$, the T_N function is not convex. The situation is shown in Figure 4.1. The heavy dashed line shows the form of T_N for $M \leq 7$ and the heavy solid line shows that T_N is not convex for $M \geq 8$.

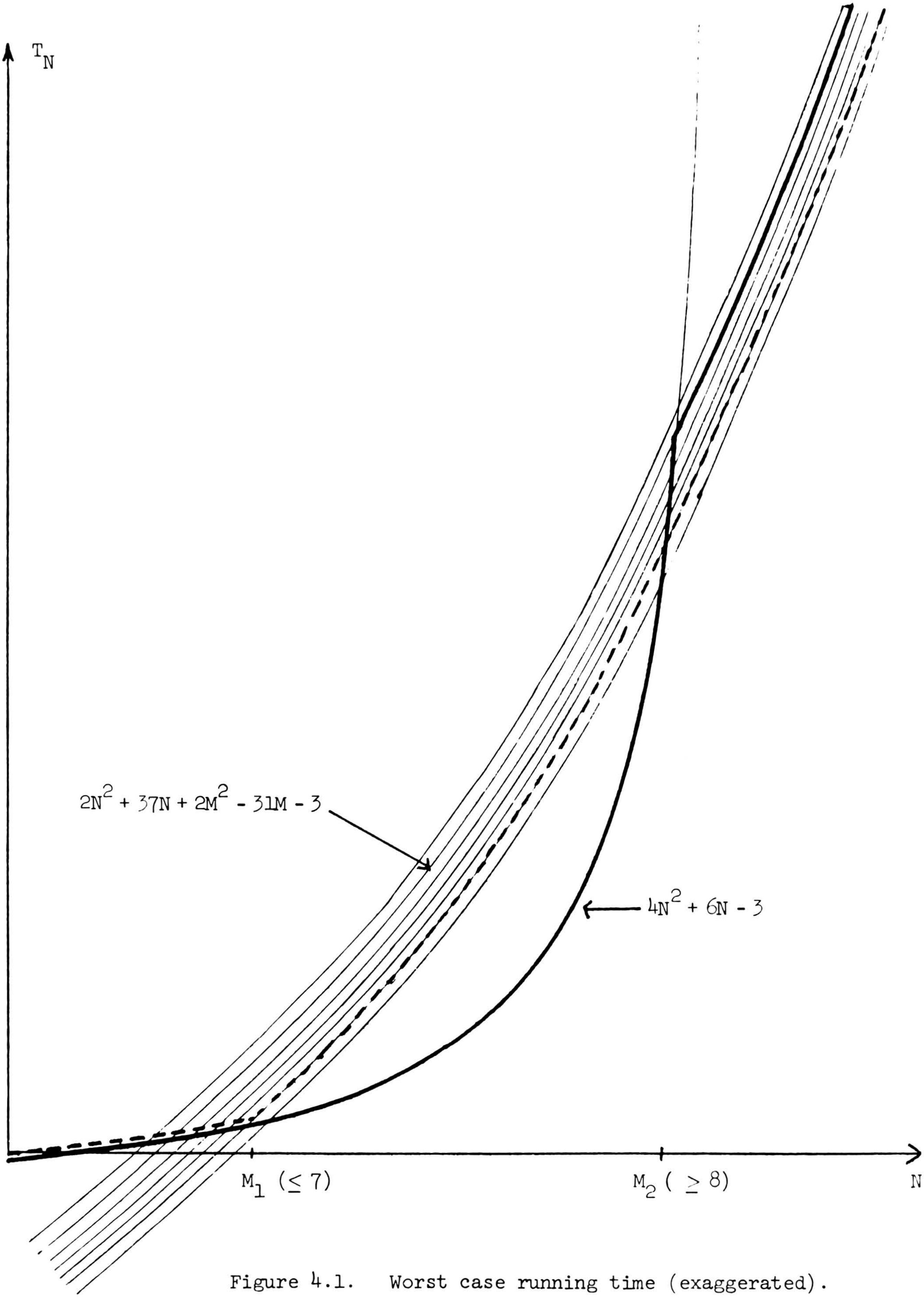
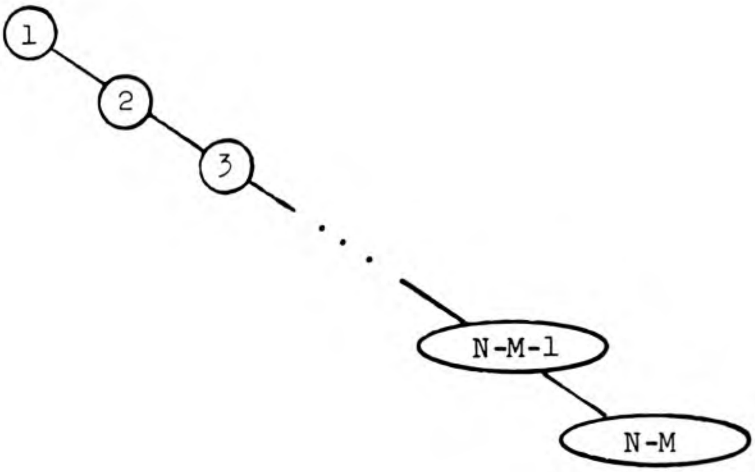


Figure 4.1. Worst case running time (exaggerated).

Theorem 4.1 tells us that Program 2.4 performs worst when, for all $N > M$, the smallest element is used as the partitioning element, and when, for $N \leq M$, the keys are in reverse order. Put another way, this means that the worst case partitioning tree is



We can use the procedure described in Chapter 2 to find a permutation that has this partitioning tree. (The procedure must be modified slightly to account for the subfile of size $\leq M$.) It is easy to see that a worst case permutation is therefore

$$1 \;\; 2 \;\; 3 \;\; \ldots \;\; N\text{-}M\text{-}1 \;\; N\text{-}M \;\; N \;\; N\text{-}1 \;\; \ldots \;\; N\text{-}M\text{+}2 \;\; N\text{-}M\text{+}1 \qquad .$$

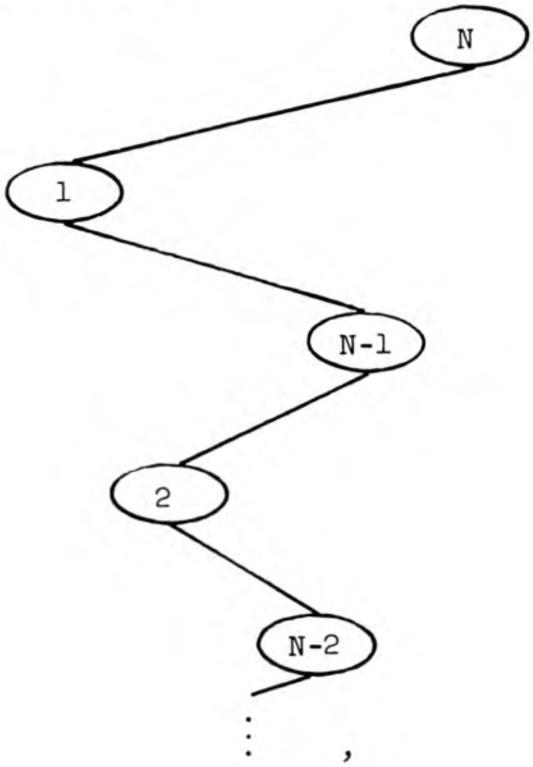
In fact, symmetry says that the worst case also occurs when the largest element is used as the partitioning element, so that

$$N \;\; M\text{-}1 \;\; M\text{-}2 \;\; \ldots \;\; 2 \;\; 1 \;\; M \;\; M\text{+}1 \;\; \ldots \;\; N\text{-}2 \;\; N\text{-}1$$

also leads to the worst case, as will any permutation whose partitioning tree has N-M levels (each node having only one non-empty subtree). Another surprising example of worst case performance of Program 2.4 is when the keys are in reverse order. The permutation

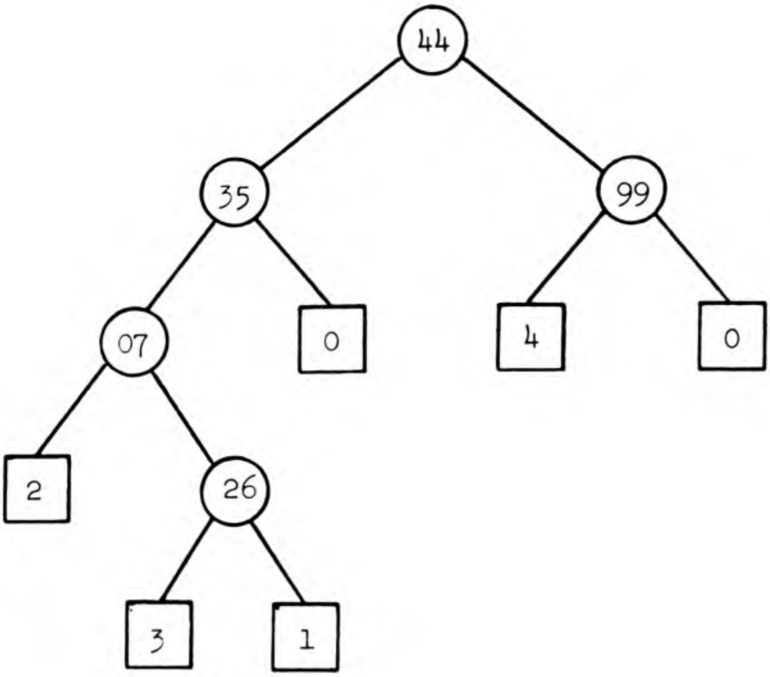
N N-1 N-2 ... 4 3 2 1

has a "zigzag" partitioning tree,

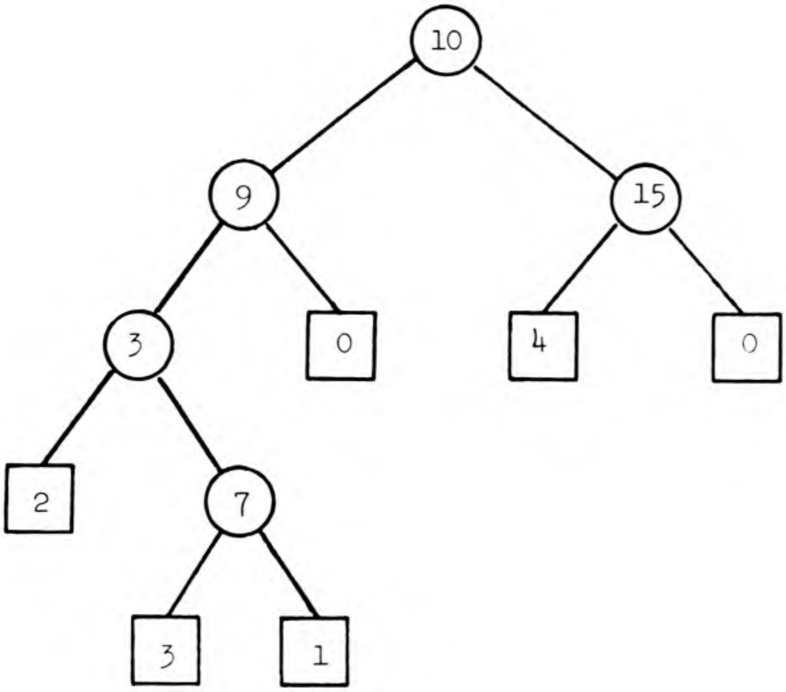


with a file of size M of keys finally occurring somewhere in the middle. If N-M is even, this file will be in reverse order, and we have a worst case permutation.

In order to study the operation of Program 2.4 in the best case, we need to refine the above techniques, because the function is substantially more erratic. It will be convenient to work with binary tree structures, although the correspondence that we will use will be slightly different than the one used in Chapter 2. First, we have to extend the correspondence to account for the fact that subfiles of length $\leq M$ are not partitioned. For example, the operation of Program 2.4 on our sample of fifteen keys when $M = 4$ (see Example 2.5) can be described by the tree



Here, external (□) nodes have been added to indicate the size of the subfiles not partitioned. If we replace each key by its relative rank in the file (this is the same as assuming that the numbers 1 to 15 are being partitioned), we get the tree

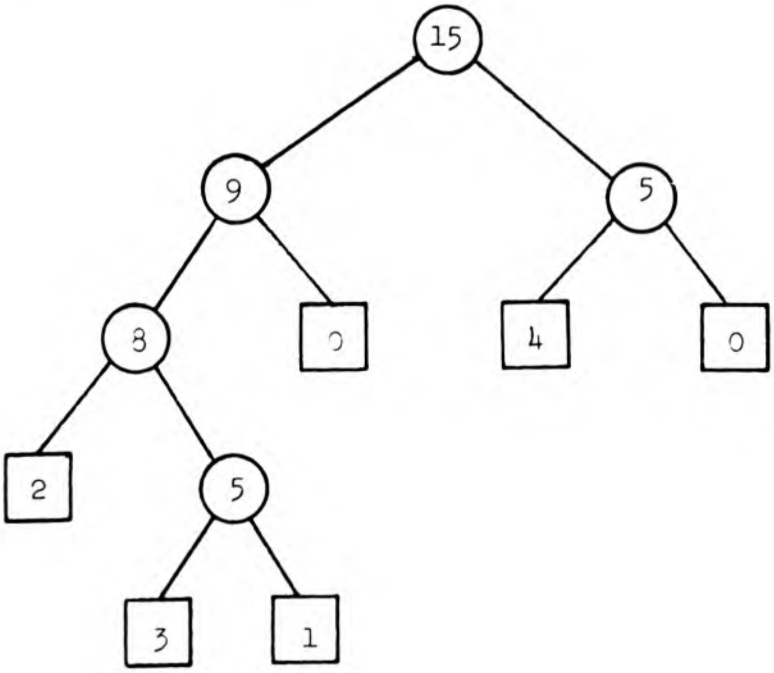


An important characteristic of this tree is that the numbers assigned to the internal nodes are redundant -- they can be reconstructed from the numbers in the external nodes. Recall that each internal node represents a partitioning phase, and the number assigned is the rank of the partitioning element used in that partitioning phase. But each of the elements less than this partitioning element must be represented somewhere in the left subtree. Each internal node represents one, and each external node $\boxed{x}$ represents x such elements. We have

rank of partitioning element = {number of internal nodes in left subtree} + {sum of weights of external nodes in left subtree} + 1 .

(It is customary to call the numbers assigned to nodes "weights".) Put another way, this equation means that an algorithm to assign weights to internal nodes is to scan the tree in symmetric order (or inorder), assigning weights from 1 to N to the internal nodes, but skipping x numbers when an external node $\boxed{x}$ is encountered in the scan.

For studying the best case, it will be useful to put a different weight on each internal node, namely the size of the subfile represented by that node (rather than the partitioning element used for that subfile). We are led to a consistent formal definition of this correspondence between the operation of Quicksort on a permutation of n elements and binary tree structures: If $n \leq M$, the tree is the single node $\boxed{n}$. Otherwise, the root node is $\textcircled{n}$ and the left and right subtrees are the trees for the left and right subfiles. Our example for $M = 4$ under this correspondence is



The structure of the tree and the weights of the external nodes are obviously the same as before, by definition. The weights of the internal nodes are again redundant, and can be reconstructed just as above, except by considering both subtrees. In this correspondence, the weight of every node is one more than the sum of the weights of its successors. Clearly, we can change from this correspondence to the other whenever convenient. Also, given any binary tree with root node Ⓝ and weights from 0 to M assigned to its external nodes, we can calculate weights for the internal nodes. If these are all $> M$, then we can also construct a corresponding permutation on N elements.

Using the weights and some other elementary properties, these trees are easily related to the analysis of the best case. Given a tree with root node Ⓝ , define the following four quantities:

a = the number of internal nodes;

s = the number of internal nodes whose successors are both internal nodes;

c = the sum of the weights of the internal nodes; and

c' = the sum of the weights of the external nodes.

Now, recall that we developed a recurrence for the best case running time:

$$U_N = \begin{cases} \min\limits_{1 \le s \le N} (U_{s-1} + U_{N-s} + 9\Delta_{sNM}) + 4N + 35 \ , & N > M \ ; \\ \\ 7N \ , & N \le M \ . \end{cases}$$

This expression can be recast in terms of the binary tree structures. Essentially we argued that the only "costs" incurred by Quicksort in the best case are for comparisons, partitioning stages, stack pushes, and the insertion sort scan. Everything else can be made to vanish independently of how partitioning is done. This means that for any given binary tree there is a corresponding permutation for which Program 2.4 will require

$$9s + 4c + 35a + 7c'$$

time units, and all other permutations corresponding to that tree will take as long or longer. For example, in the tree pictured above, we have $a = 5$, $s = 1$, $c = 42$, and $c' = 10$, so that the best case running time of all permutations corresponding to that tree is 422 time units. We can slightly simplify this expression by noticing that every element must either be used as a partitioning element or fall into a small subfile, so that $a + c' = N$ by definition. Our problem is now reduced to minimizing the quantity

$$28a + 4c + 9s + 7N$$

over all trees with root node Ⓝ . Let us refer to this quantity as the *cost* of a tree, and let us refer to the minimum cost trees as the *best case trees*, leaving implicit the dependence on N and M .

Our method will be to define a sequence of transformations on trees which leave the weight of the root node unchanged, but which lower the cost. Any tree to which none of the transformations apply will be a best case tree. Our transformations may modify the structure of the tree and change the weights of external nodes -- the weights of the internal nodes can always be computed.

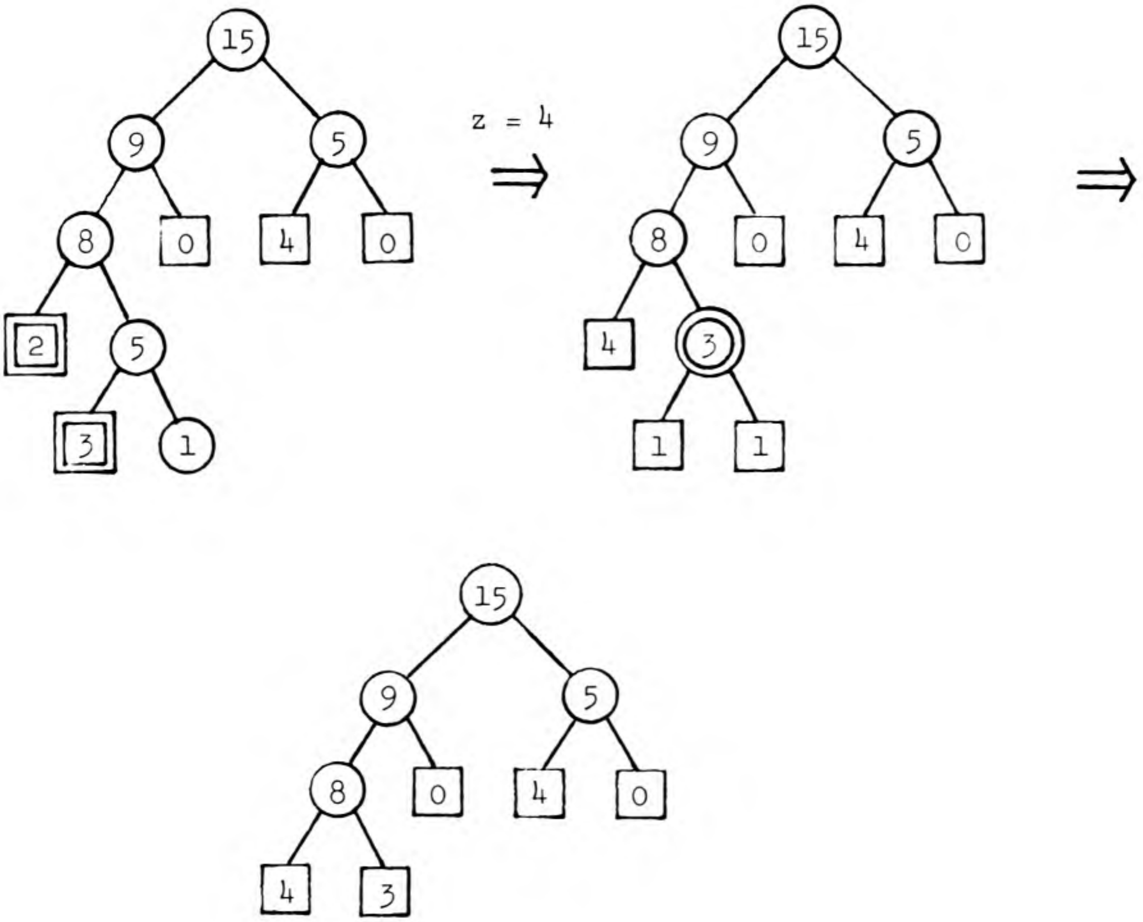
For example, two external nodes which are not $\boxed{M}$ can be merged so that only one of them is not $\boxed{M}$. A formal definition of a transformation to do this is as follows:

Transformation 1: Suppose that a tree with root node Ⓝ contains an external node $\boxed{x}$ at level p and an external node $\boxed{y}$ at level q with $p \leq q$ and $x, y \neq M$. Let $z \equiv \min(M, x+y+1)$.

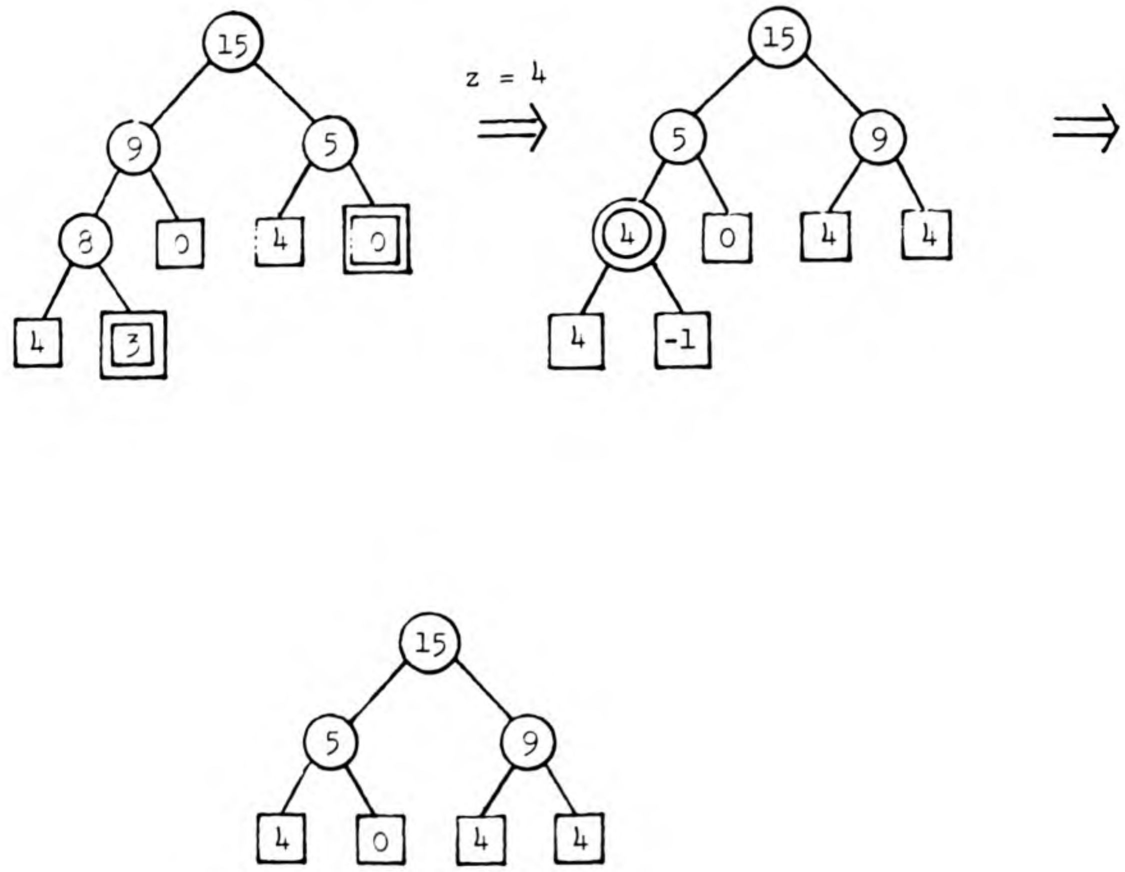
(i) Replace $\boxed{x}$ by $\boxed{z}$ and $\boxed{y}$ by $\boxed{x+y-z}$. (Note that this could be $\boxed{-1}$.)

(ii) As described above, assign to each node a weight equal to one more than the sum of the weights of its immediate successors. If any internal node has a weight $\leq M$ then make it external and delete its successors. Then if any node has $\boxed{-1}$ as an immediate successor, replace it with its other successor and delete the $\boxed{-1}$. □

For example, the $\boxed{2}$ and the $\boxed{3}$ in our sample tree are merged by Transformation 1 as follows:



The $\boxed{3}$ may then be merged with the rightmost $\boxed{0}$ to illustrate another case of Transformation 1:



Transformation 1 can no longer be applied, since there is only one external node which is not $\boxed{M}$.

It is not difficult to verify that Transformation 1 has the properties that we desire, if we notice that there is another way to compute the cost of a tree. Specifically, we can find an alternate expression for c , the sum of the weights of the internal nodes, because each external node contributes its weight to all of the internal nodes above it in the tree and each internal node contributes 1 to its own weight and the weights of all the internal nodes above it in the tree. The number of internal nodes which appear above a given node in the tree is just its level in the tree, so we have

$$c = \sum_{\substack{\text{external}\\ \text{nodes } \boxed{e}}} e\cdot\text{level}(\boxed{e}) + \sum_{\substack{\text{internal}\\ \text{nodes } \textcircled{i}}} \text{level}(\textcircled{i}) + a \quad .$$

(The second sum is customarily called the "internal path length" of the tree; and the first is called the "weighted external path length".)

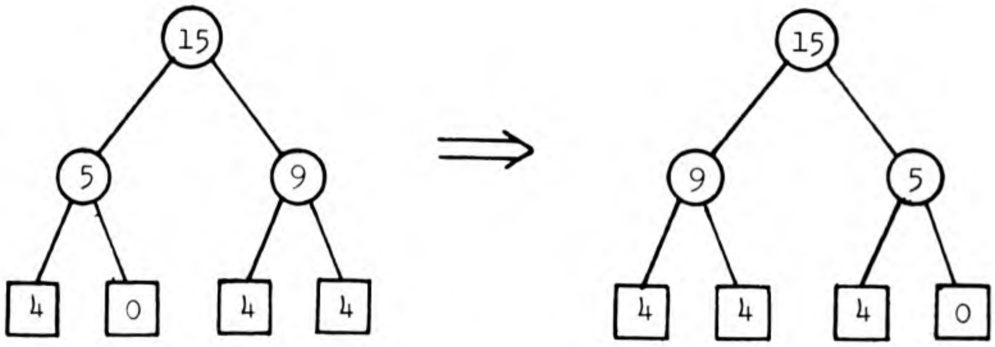
Now it is easy to see that Transformation 1 does not increase the cost of the tree. Step (i) simply replaces the terms $x\cdot p + y\cdot q$ by $z\cdot p + (x+y-z)\cdot q$ in the expression for c . The difference between these is $(z-x)p + (x-z)q = (z-x)(p-q)$ which must be ≤ 0 since $x \leq z$ and $p \leq q$. Step (ii) cannot increase the cost, since it can only delete nodes, which might cause a or c to decrease, and obviously cannot add to the cost of the tree. Therefore, the net cost $28a + 4c + 9s + 7N$ cannot be increased by the transformation.

Furthermore, the only internal nodes affected by Transformation 1 are those on the path to the root from $\boxed{x}$ and $\boxed{y}$ in the smallest subtree containing them both. The weight of this root and the weights of all other nodes in the tree are unaffected by the transformation. In particular, the weight of the root node of the whole tree $\textcircled{N}$ is never changed by Transformation 1.

Notice that the application of Transformation 1 always results in the number of external nodes which are not $\boxed{M}$ being decreased by 1 . This means that it can be applied again and again until there is at most one such node, and we are led to:

Theorem 4.2. There is always a best case tree in which every partitioning occurs on an element whose rank is a multiple of M+1 .

Proof. Given any partitioning tree with root node Ⓝ apply Transformation 1 to it until all of the external nodes are $\boxed{M}$, except possibly one node $\boxed{x}$. Then interchange the subtrees of any node having $\boxed{x}$ in its left subtree. This obviously has no effect on the cost. In our example, we have



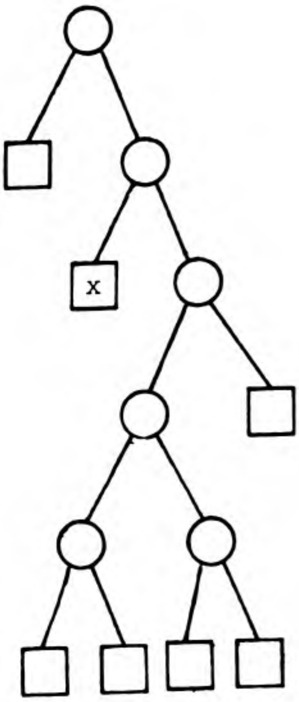
In general, it is a well-known fact that the number of internal nodes in every binary tree is exactly one greater than the number of external nodes. Therefore the left subtree of every internal node in our tree contains k internal nodes and $k+1$ $\boxed{M}$ nodes for some k . We have found from our definitions that

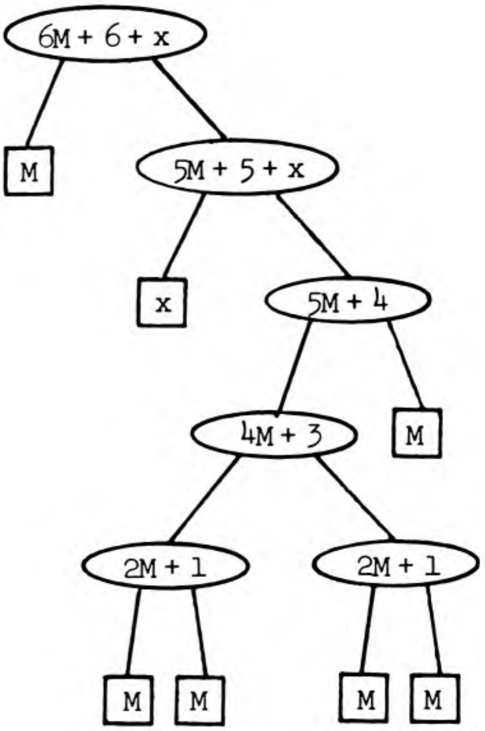
$$\begin{aligned}\text{rank of partitioning element} &= \{\text{number of internal nodes in left subtree}\}\\ &\quad + \{\text{sum of weights of external nodes in}\\ &\qquad \text{left subtree}\}\\ &\qquad + 1\\ &= k + (k+1)\,M + 1\\ &= (k+1)(M+1) \quad ,\end{aligned}$$

so that the rank of every partitioning element is a multiple of $M+1$. Since the above manipulations do not raise the cost of any tree, even one which purports to represent the best case, Theorem 4.2 is established.

$\square$

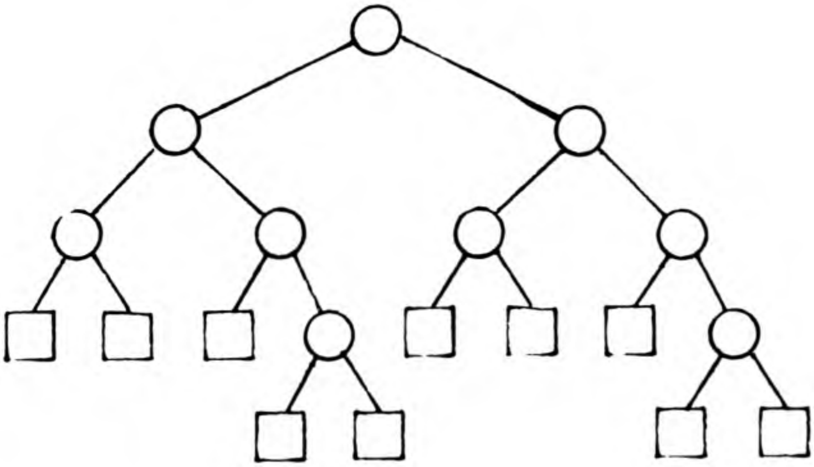
One of the consequences of Theorem 4.2 is that we no longer have to write down the weights of even the external nodes, so long as the value of M is understood. The weights of any tree which could represent the best case can be reconstructed from the weight of the one external node $\boxed{x}$ which is not $\boxed{M}$, so that, for example, the following two trees are equivalent:



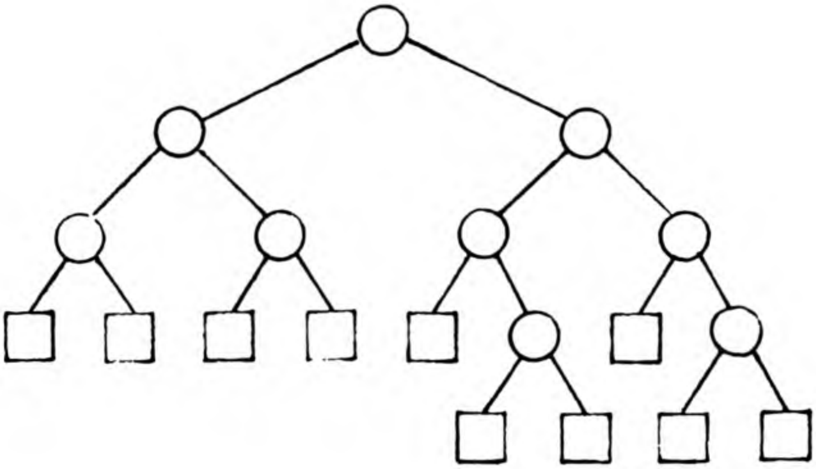


If a tree has no external node which is not $\boxed{M}$, it is convenient to pick one and call it $\boxed{x}$, so that all of our trees have one external node $\boxed{x}$ with $0 \leq x \leq M$. Notice that the internal nodes have weights of the form $k(M+1)+x$ for some integer k if they are on the path from $\boxed{x}$ to the root; $k(M+1)+M$ otherwise.

The last step in our derivation of the best case will be to come to the intuitive result that the best way to choose the partitioning element is as close to the middle of the file as permitted by Theorem 4.2. This choice will not in general be unique -- for example the two trees



and

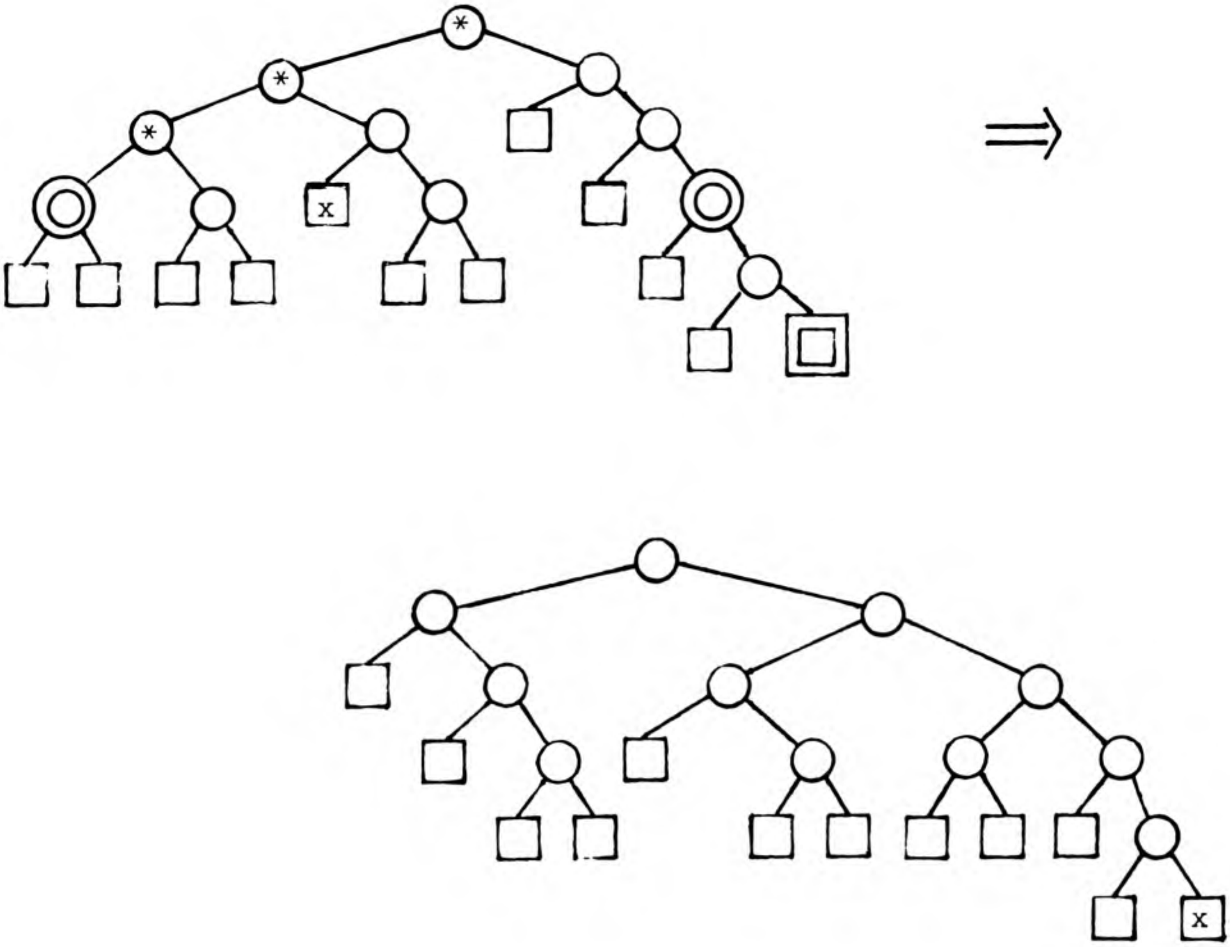


both have the same cost, but the partitioning element is $5M+5$ for the first and $4M+4$ for the second. We will work with a canonical form from which trees of equivalent cost can be generated. This is done by the following transformation on trees.

<u>Transformation 2</u>: Suppose that Transformation 1 can no longer be applied to some tree. That is, it has exactly one external node $\boxed{x}$ $0 \leq x \leq M$ which is not necessarily $\boxed{M}$. If $\boxed{x}$ is not at the deepest level of the tree, interchange $\boxed{x}$ with any $\boxed{M}$ node

which is at the deepest level. Then, if the subtree of the root containing $\boxed{x}$ has fewer external nodes than the subtree of the root not containing $\boxed{x}$, interchange any internal node at the deepest level of the subtree not containing $\boxed{x}$ with $\boxed{x}$'s ancestor at that level. Finally, interchange the subtrees of all nodes with $\boxed{x}$ in their left subtree. □

The effect of this transformation is to put $\boxed{x}$ in the rightmost and deepest position in the tree, and to put at least as many external nodes in the right subtree as in the left, all without increasing the cost. The application of Transformation 2 to a sample tree is shown below. First $\boxed{x}$ is interchanged with ⧈ ; then the two ◎ nodes are interchanged; and then the subtrees of the three ⊛ nodes are interchanged.

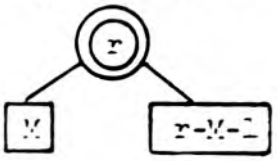


Interchanging subtrees or interchanging any two internal nodes at the same level clearly has no effect on the cost; and by the same argument as used for Transformation 1, the $\boxed{x}$ interchange cannot raise the cost since it results in the nonnegative quantity M-x being added to some internal weights, but subtracted from as many or more other internal weights.

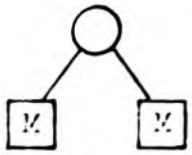
We are now ready to finally minimize the cost of the trees by "balancing" them. The next transformation takes two external nodes from the "heavy" part of the tree and adds them to the "light" part, in such a way as to leave the resulting tree in the canonical form.

(That is, all internal nodes will have at least as many external nodes on the right as on the left.)

Transformation 3: Suppose that Transformation 1 can no longer be applied to some tree; that Transformation 2 has been applied to it; and that the number of external nodes in the right subtree of the root exceeds the number of external nodes in the left subtree of the root by more than 1. Let p-1 be the level of the $\boxed{x}$ node, and let q be the level of the shallowest $\boxed{M}$ node in the left subtree of the root. Let $\textcircled{r}$, in the right subtree, denote the leftmost internal node on level p whose brother is also internal, or simply the leftmost internal node on level p if they all have external brothers; and let $\boxed{\boxed{M}}$, in the left subtree, denote the rightmost external node on level q whose brother is also external, or simply the rightmost external node on level q if they all have internal brothers. The transformation is to replace

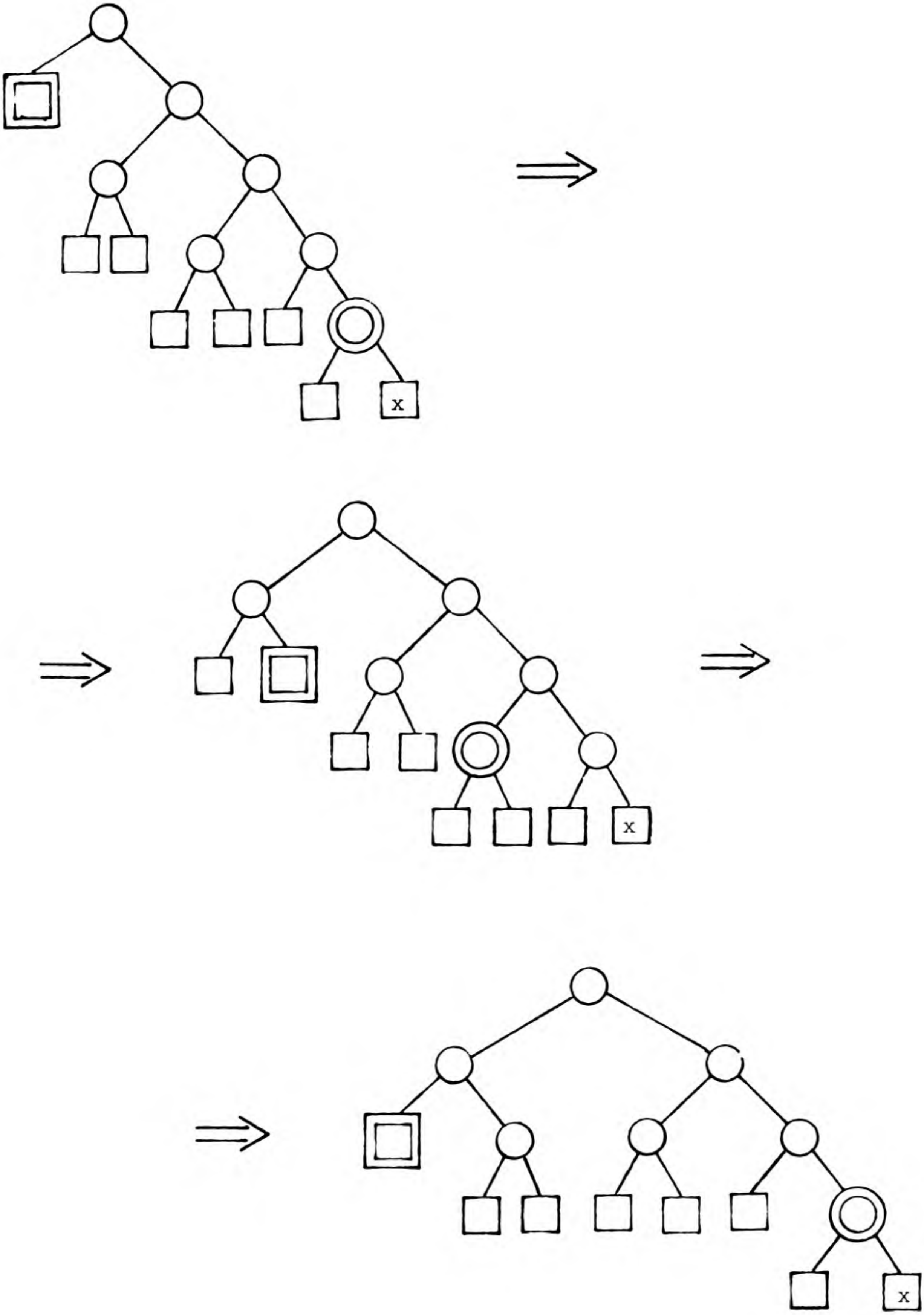


by $\boxed{r-M-1}$ and $\boxed{\boxed{M}}$ by

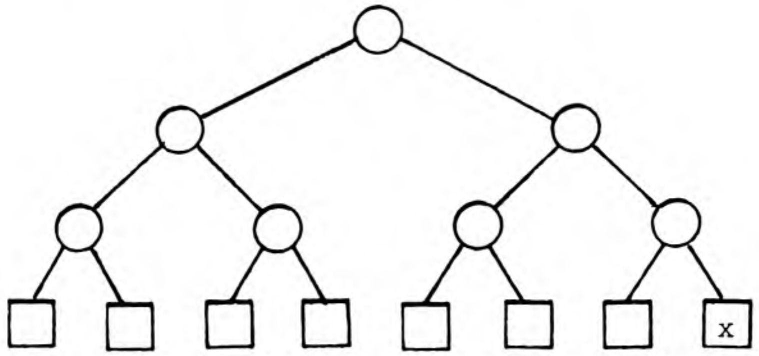


. Do this only if it does not increase the cost. (It almost always decreases the cost -- see the discussion below.) □

This transformation is not as complicated as it seems. Basically, it involves subtracting M+1 at a deep level in the tree and adding it back on at a shallower level, without changing the number of external or internal nodes. The same arguments that we have used before will show that the cost is decreased, except possibly in the case when a new stack push is introduced. An example of the repeated application of Transformation 3 follows:


x
x
x

Depending on the value of $\boxed{x}$, this tree might further be transformed to



,

but this transformation introduces a stack push and therefore could increase the cost.

We are finally ready to discover the best choice of the partitioning element. It is almost always possible to apply Transformation 3 until the number of external nodes in the right subtree is either the same or one greater than the number of external nodes in the left subtree. More precisely, there will be $\lceil \frac{k}{2} \rceil$ $\boxed{M}$ nodes in the left subtree and $\lfloor \frac{k}{2} \rfloor$ $\boxed{M}$ nodes and the $\boxed{x}$ node in the right subtree, where $N = k(M+1)+x$ $(0 \leq x \leq M)$. From the definition of our tree structures, this means that the best choice of the partitioning element is $\lceil \frac{k}{2} \rceil (M+1)$. With this knowledge, we can compute the exact value of U_N, the running time of Program 2.4 in the best case. First, however, we must find the cases where it is not possible to "perfectly" balance the tree in this way, and prove the following theorem.

Theorem 4.3. Let $M > 1$. The best case of Program 2.4 occurs when, at every partitioning stage, if N is the number of elements being partitioned then the element with rank $\left\lceil \frac{N+1}{2(M+1)} - \frac{1}{2} \right\rceil (M+1)$ is used as the partitioning element. There are two exceptions: if $N = 3M+3$ or $3M+4$, then the $(M+1)$-st largest element is used.

Proof. If N is written as $N = k(M+1)+x$ with $0 \leq x \leq M$, then

$$\left\lceil \frac{N+1}{2(M+1)} - \frac{1}{2} \right\rceil = \left\lceil \frac{k}{2} - \frac{M-x}{2(M+1)} \right\rceil$$

$$= \left\lceil \left\lceil \frac{k}{2} \right\rceil - \frac{1}{2}\left(k \bmod 2 + \frac{M-x}{M+1} \right) \right\rceil$$

$$= \left\lceil \frac{k}{2} \right\rceil \quad ,$$

so that it will be sufficient to show that any tree with root node $\textcircled{N}$ can be transformed into a lower (or equal) cost tree with root node $\textcircled{N}$ and with exactly $\left\lceil \frac{k}{2} \right\rceil$ $\boxed{M}$ nodes in its left subtree.

Following the discussion above, we will use Transformations 1, 2, and 3 to produce such a tree: It remains only to analyze the conditions under which Transformation 3 is applicable. If the change stated in Transformation 3 is made, then the total cost $28a + 4c + 9s + 7N$ is changed by

$$\Delta = 4(2M + 1 - r + (q-1)(M+1) - (p-1)(M+1)) + \delta \quad ,$$

$$\text{where } \delta = \begin{cases} 9 & \text{if a new stack push is introduced} \\ -9 & \text{if a stack push is eliminated} \\ 0 & \text{otherwise.} \end{cases}$$

Either $r = M+x+1$, where $0 \leq x \leq M$, or $r = 2M+1$, which is the same as the case $x = M$, so the change is

$$\Delta = 4((q-p+1)(M+1) - x - 1) + \delta \qquad 0 \leq x \leq M \quad .$$

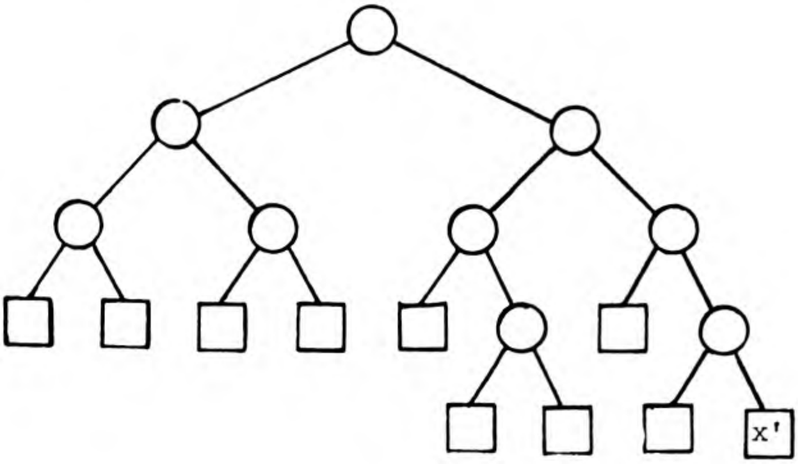
We are interested in identifying the cases where Δ is positive. These are the cases for which Transformation 3 does not apply. Clearly, since $M > 1$, this quantity will be negative for $p \geq q+2$ or $x \geq 2$ no matter what the value of δ is . This leaves two cases: $p = q$ and $p = q+1$.

Let N_L , N_R be the number of external nodes in the left and right subtrees. From the stipulations of Transformation 3, we have the three inequalities

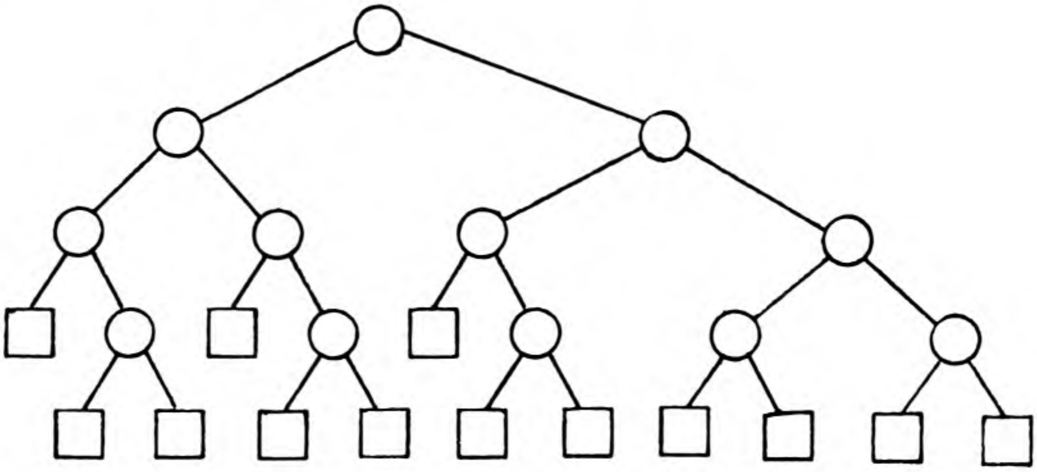
$$2^q \leq N_L \quad , \quad N_R \leq 2^{p+1} \quad , \quad \text{and} \quad N_L \leq N_R - 2 \quad .$$

These will be useful in the remainder of the proof.

A typical example where $p = q$ is the tree

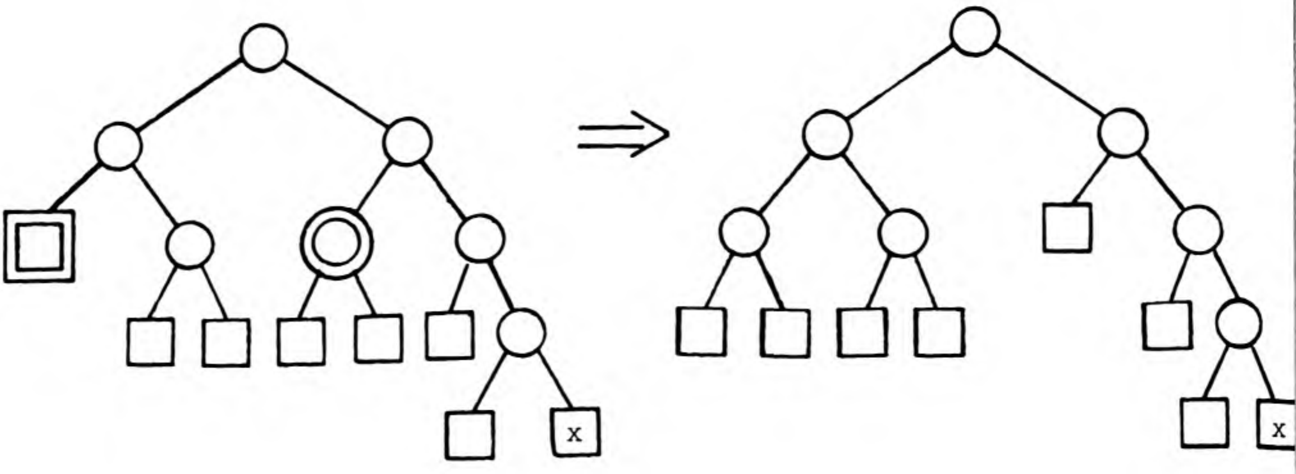


We know from the first and third inequalities above that $N_R \geq 2^p + 2$, so that there is another internal node besides $\left(x' + M + 1 \right)$ on level p, which implies that $\Delta = \delta$. For a stack push to be required on the left side, we must have $N_L \geq 2^q + 2^{q-1}$. A minimal such tree has alternate $\boxed{M}$ and $\left(2M+1 \right)$ nodes on level q, as the following example shows:

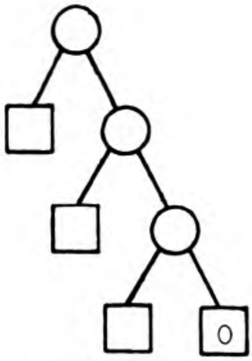


But the right subtree has the same number of levels and two more external nodes, so there must be at least one more internal node on level q, so a stack push is also required on the right. This implies that $\delta \neq 9$, so $\Delta \leq 0$.

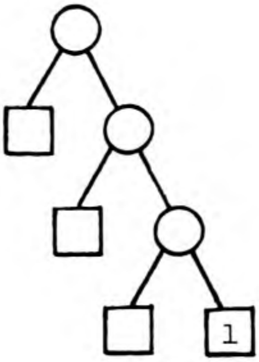
A similar argument holds for $p = q+1$. If $x = 0$ or 1, then by the way $\textcircled{r}$ is chosen in Transformation 3, $\left(x + M + 1 \right)$ must be the only internal node on level p. Using the same argument as above, there must be a node on level $p-1 = q$ in the right subtree which can be moved to the left at a cost $\Delta \leq 0$. An example of this transformation is:



This argument degenerates if it is the root which causes the stack push, so that the trees



and



are the best case trees for $N = 3M + 3$ and $N = 3M + 4$.

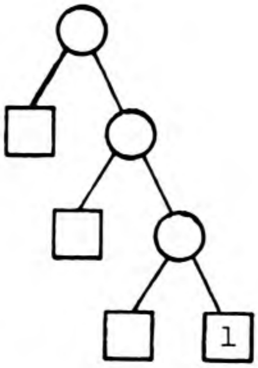
Strictly speaking, this final argument should be incorporated into our series of transformations.

Transformation 3a. Under the conditions of Transformation 3: if $p = q+1$; $N > 3M + 4$; and the modification stipulated in Transformation 3 would increase the cost, then let be the leftmost internal node on level $p-1$ in the right subtree and perform the modification as stated. □

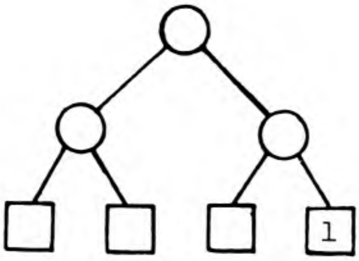
This completes the proof of Theorem 4.3. Our series of transformations is sufficient to "balance" any tree with root node Ⓝ .

□

It is important to be aware that this derivation depends completely on the choice of coefficients for the quantities contributing to the running time of Program 2.4 Most important are the relative values of the coefficients for comparisons and stack pushes. If the overhead for stack pushes is relatively high (which could happen in a practical situation, if a recursive implementation is used), then the best trees will tend to be more unbalanced, for it will pay to avoid stack pushes. This effect appears in our derivation as the exceptions for $N = 3M + 3$ or $3M + 4$. The tree



is less expensive than



because it requires one less stack push. The reader may find more examples of this effect by looking for counterexamples in the above derivation for $M = 0$ and 1.

We are now ready to derive an exact expression for the total running time of Program 2.4 in the best case. If all of the external nodes of the partitioning tree are $\boxed{M}$, then N is a number of the form $k(M+1)-1$, and Theorem 4.2 tells us that both subfiles will also have this property. Theorem 4.3 tells us exactly what the partitioning element is in the best case. Substituting this information into our original recurrence for the running time of Program 2.4 in the best case gives the equation

$$U_{k(M+1)-1} = \begin{cases} U_{\lceil\frac{k}{2}\rceil(M+1)-1} + U_{\lceil\frac{k}{2}\rceil(M+1)-1} + 9\Delta_{\lceil\frac{k}{2}\rceil(M+1)NM} + 4k(M+1)+31 & k(M+1)-1 > M \\ 7k(M+1)-7 & k(M+1)-1 \le M. \end{cases}$$

Now, if $k = 1$, we have $U_M = 7M$; if $k = 2$, then $U_{2M+1} = U_M + U_M + 8M + 39 = 22M + 39$; and if $k = 3$ then $U_{3M+2} = U_{2M+1} + U_M + 12M + 43 = 41M + 82$. For $k \geq 4$, there is always a stack push, so we have eliminated the $\Delta_{\lceil\frac{k}{2}\rceil(M+1)NM}$ term. To simplify notation we will define $f(k) \equiv U_{k(M+1)-1}$, so that we now have the recurrence

$$f(k) = \begin{cases} f\left(\left\lceil \frac{k}{2} \right\rceil\right) + f\left(\left\lfloor \frac{k}{2} \right\rfloor\right) + 4k(M+1) + 40 & k \geq 4 \\ 41M + 82 & k = 3 \\ 22M + 39 & k = 2 \end{cases} .$$

One way to solve this recurrence is to look at the differences between successive terms. For $k \geq 4$ we have

$$\begin{aligned} f(k+1) - f(k) &= f\left(\left\lceil \frac{k+1}{2} \right\rceil\right) + f\left(\left\lfloor \frac{k+1}{2} \right\rfloor\right) + 4(k+1)(M+1) + 40 \\ &\qquad - f\left(\left\lceil \frac{k}{2} \right\rceil\right) - f\left(\left\lfloor \frac{k}{2} \right\rfloor\right) - 4k(M+1) - 40 \\ &= f\left(\left\lfloor \frac{k}{2} \right\rfloor + 1\right) - f\left(\left\lfloor \frac{k}{2} \right\rfloor\right) + 4(M+1) \quad . \end{aligned}$$

This last formula can be verified by observing that $\left\lceil \frac{k+1}{2} \right\rceil = \left\lfloor \frac{k}{2} \right\rfloor + 1$ and $\left\lfloor \frac{k+1}{2} \right\rfloor = \left\lceil \frac{k}{2} \right\rceil$. Now, let $p = \lfloor \lg k \rfloor$, so that $2^p \leq k < 2^{p+1}$. Noting that $\left\lfloor \frac{1}{2} \left\lfloor \frac{k}{2^j} \right\rfloor \right\rfloor = \left\lfloor \frac{k}{2^{j+1}} \right\rfloor$, we may telescope the recurrence p times to give

$$f(k+1) - f(k) = f\left(\left\lfloor \frac{k}{2^{p-1}} \right\rfloor + 1\right) - f\left(\left\lfloor \frac{k}{2^{p-1}} \right\rfloor\right) + 4(p-1)(M+1) \quad .$$

If k is in the left half of the interval, $2^p \leq k < 3 \cdot 2^{p-1}$, then $\left\lfloor \frac{k}{2^{p-1}} \right\rfloor = 2$, and if it is in the range $3 \cdot 2^{p-1} \leq k < 2^{p+1}$, then $\left\lfloor \frac{k}{2^{p-1}} \right\rfloor = 3$, so that we have

$$f(k+1) - f(k) = \begin{cases} f(3) - f(2) + 4(p-1)(M+1) \quad , & \text{for} \quad 2^p \le k < 3\cdot 2^{p-1} \; ; \\ f(4) - f(3) + 4(p-1)(M+1) \quad , & \text{for} \quad 3\cdot 2^{p-1} \le k < 2^{p+1} \; . \end{cases}$$

After substituting the values for $f(2)$, $f(3)$, and $f(4)$ we can combine these into the single formula

$$f(k+1) - f(k) = 19M + 43 + 4(\lfloor \lg k \rfloor - 1)(M+1) + 9\nu_2(k) \qquad k \ge 2 \quad ,$$

where $\nu_2(k)$ denotes the second most significant bit in the binary representation of k :

$$\nu_2(k) = \begin{cases} 0 & \text{for} \quad 2^{\lfloor \lg k \rfloor} \le k < 3\cdot 2^{\lfloor \lg k \rfloor - 1} \\ 1 & \text{for} \quad 3\cdot 2^{\lfloor \lg k \rfloor - 1} \le k < 2^{\lfloor \lg k \rfloor + 1} \end{cases} \quad .$$

This recurrence now telescopes into a summation:

$$f(k) = f(2) + \sum_{3 \le j \le k} (19M + 43 + 4(\lfloor \lg(j-1) \rfloor - 1)(M+1) + 9\nu_2(j-1))$$

$$= 22M + 39 + (15M + 39)(k-2) + 4(M+1) \sum_{2 \le j \le k-1} \lfloor \lg j \rfloor + 9 \sum_{2 \le j \le k-1} \nu_2(j).$$

The first sum remaining is a well-known identity (see Appendix B):

$$\sum_{2 \le j \le k-1} \lfloor \lg j \rfloor = k \lfloor \lg k \rfloor - 2^{\lfloor \lg k \rfloor + 1} + 2 \quad ;$$

and the second sum is easily evaluated from the definition of $\nu_2(k)$:

$$\sum_{2 \le j \le k-1} \nu_2(j) = 2^{\lfloor \lg k \rfloor - 1} - 1 - \nu_2(k) + \left(k - 3\cdot 2^{\lfloor \lg k \rfloor - 1}\right)\nu_2(k) \; .$$

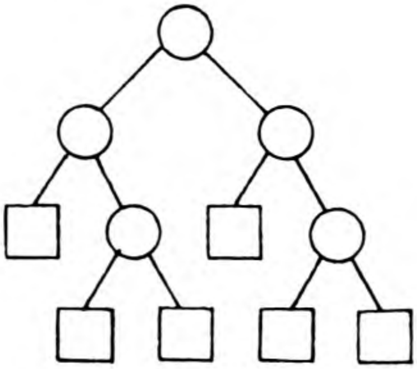
Substituting these gives the exact formula for the running time of Program 2.4 in the best case, when N is a number of the form $N = k(M+1)-1$:

$$U_N = (N+1)\left(4\left\lfloor \lg \frac{N+1}{M+1} \right\rfloor + 15\right) + 24\,\frac{N+1}{M+1} - (16M+7)2^{\left\lfloor \lg \frac{N+1}{M+1}\right\rfloor - 1} - 40$$

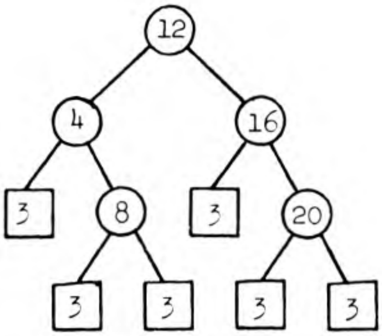
$$+\ 9\nu_2\left(\frac{N+1}{M+1}\right)\left(\frac{N+1}{M+1} - 3\cdot 2^{\left\lfloor \lg \frac{N+1}{M+1} - 1\right\rfloor}\right),$$

$$\text{where}\quad \nu_2(k) = \begin{cases} 0 & \text{if}\ \ 2^{\lfloor \lg k\rfloor} \le k < 3\cdot 2^{\lfloor \lg k\rfloor - 1} \\ 1 & \text{if}\ \ 3\cdot 2^{\lfloor \lg k\rfloor - 1} \le k < 2^{\lfloor \lg k\rfloor + 1} \end{cases}.$$

As we did with the worst case, we can work backwards from the partitioning trees to generate permutations which lead to the best case performance. For example, the best case tree for $N = 6(M+1)-1$ is



If we fill in partitioning elements on the internal nodes for $M = 3$, we get the tree



and then we can use the procedure described in Chapter 2 (slightly modified to handle $M > 1$) to get a permutation of $\{1,2,\ldots,23\}$ corresponding to this tree:

```
 1  2  3
 4  2  3  1
             5  6  7
             8  6  7  5
                        9  10  11
12  2  3  1  8  6  7  5  9  10  11  4
                                       13  14  15
                                       16  14  15  13
                                                       17  18  19
                                                       20  18  19  17
                                                                       21  22  23
12  2  3  1  8  6  7  5  9  10  11  4  16  14  15  13  20  18  19  17  21  22  23
```

The formula derived above says that Program 2.4 will require

$$24(4\cdot 2+15)+24\cdot 6 - 55\cdot 2 - 40 = 546$$

time units to sort this permutation, and no other permutation on 23 elements will require less time.

Finally, we can write down an exact formula which works for all integers $N > M$. To keep our notation consistent, it will be convenient to write N in the form $N = k(M+1)+x$ with $-1 \leq x \leq M-1$. Any number N can obviously be represented in this way. Again, by Theorems 4.2 and 4.3 we have, after separating out small terms and dealing with the special cases $N = 3M+3$ and $N = 3M+4$, the recurrence

$$U_{k(M+1)+x} = \begin{cases} U_{\left\lceil \frac{k}{2} \right\rceil (M+1)-1} + U_{\left\lfloor \frac{k}{2} \right\rfloor (M+1)+x} + 4(k(M+1)+x) + 35 & k \geq 4 \\ \\ U_{M+1+x} + 34M + 95 + 4x + (\delta_{0x} + \delta_{1x})(4x-5) - 9\delta_{-1x} & k = 3 \\ \\ U_{M+1+x} + 15M + 43 + 4x & k = 2\,. \end{cases}$$

(Here δ_{ij} represents the Kronecker delta function, which has the value 1 if $i = j$, 0 otherwise.) The easiest way to solve this is to subtract it from our original equation for $U_{k(M+1)-1}$, so that the $U_{\left\lceil \frac{k}{2} \right\rceil (M+1)-1}$ term cancels. This results in the equation

$$U_{k(M+1)+x} - U_{k(M+1)-1} = \begin{cases} U_{\left\lfloor \frac{k}{2} \right\rfloor (M+1)+x} - U_{\left\lfloor \frac{k}{2} \right\rfloor (M+1)-1} + 4(x+1) & k \geq 4 \\ \\ U_{M+1+x} - 7M + 4x + 13 + (\delta_{0x} + \delta_{1x})(4x-5) - 9\delta_{-1x} & k = 3 \\ \\ U_{M+1+x} - 7M + 4(x+1) & k = 2\,. \end{cases}$$

We can telescope this equation just as before to get

$$U_{k(M+1)+x} - U_{k(M+1)-1}$$

$$= U_{M+1+x} - 7M + 4(\lfloor \lg k \rfloor)(x+1) + \nu_2(k)(9(i-\delta_{-1x}) + (\delta_{x0}+\delta_{x1})(4x-5)) \ .$$

If $x = -1$, the right hand side is of course 0. If $x \neq -1$, then $U_{M+1+x} = U_M + U_x + 4(M+1+x) + 35 = 11(M+x) + 39$, and we can calculate the final solution

$$U_N = (N+1)\left(4\left\lfloor \lg \frac{N+1}{M+1}\right\rfloor + 15\right) + 24\,\frac{N+1}{M+1} - (16M+7)2^{\left\lfloor \lg \frac{N+1}{M+1}\right\rfloor - 1} - 40 - 4r$$

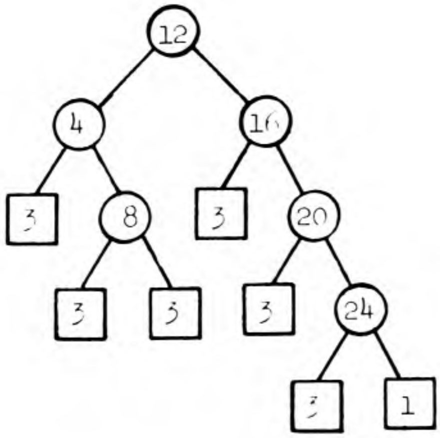
$$+\ 9\nu_2\left(\frac{N+1}{M+1}\right)\left(\frac{N+1}{M+1} - 3\cdot 2^{\left\lfloor \lg \frac{N+1}{M+1}\right\rfloor - 1}\right)$$

$$+\ (1-\delta_{r0})\left(4M+28+\nu_2\left(\frac{N+1}{M+1}\right)(9+(\delta_{r1}+\delta_{r2})(4r-9))\right)$$

where $r = (N+1) \bmod M+1$

$$\text{and}\quad \nu_2\left(\frac{N+1}{M+1}\right) = \begin{cases} 0 & 2^{\left\lfloor \lg \frac{N+1}{M+1}\right\rfloor} \le \left\lfloor \frac{N+1}{M+1}\right\rfloor < 3\cdot 2^{\left\lfloor \lg \frac{N+1}{M+1}\right\rfloor - 1} \\ 1 & 3\cdot 2^{\left\lfloor \lg \frac{N+1}{M+1}\right\rfloor - 1} \le \left\lfloor \frac{N+1}{M+1}\right\rfloor \le 2^{\left\lfloor \lg \frac{N+1}{M+1}\right\rfloor + 1} \end{cases}$$

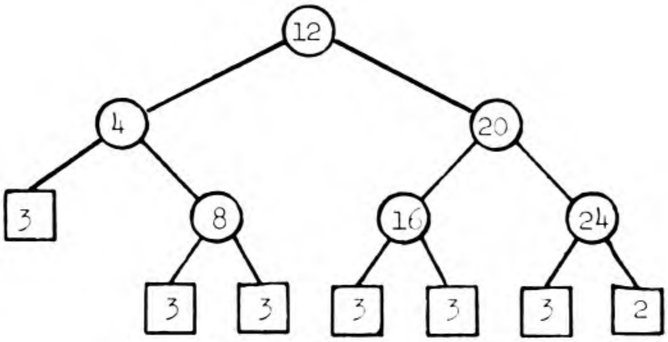
For example, for $M = 3$ and $N = 24$ the best case tree is



and the permutation

12 2 3 1 8 6 7 5 9 10 11 4 16 14 15 13 20 18 19 17 24 22 23 21

takes 632 time units. For 26 elements the tree is

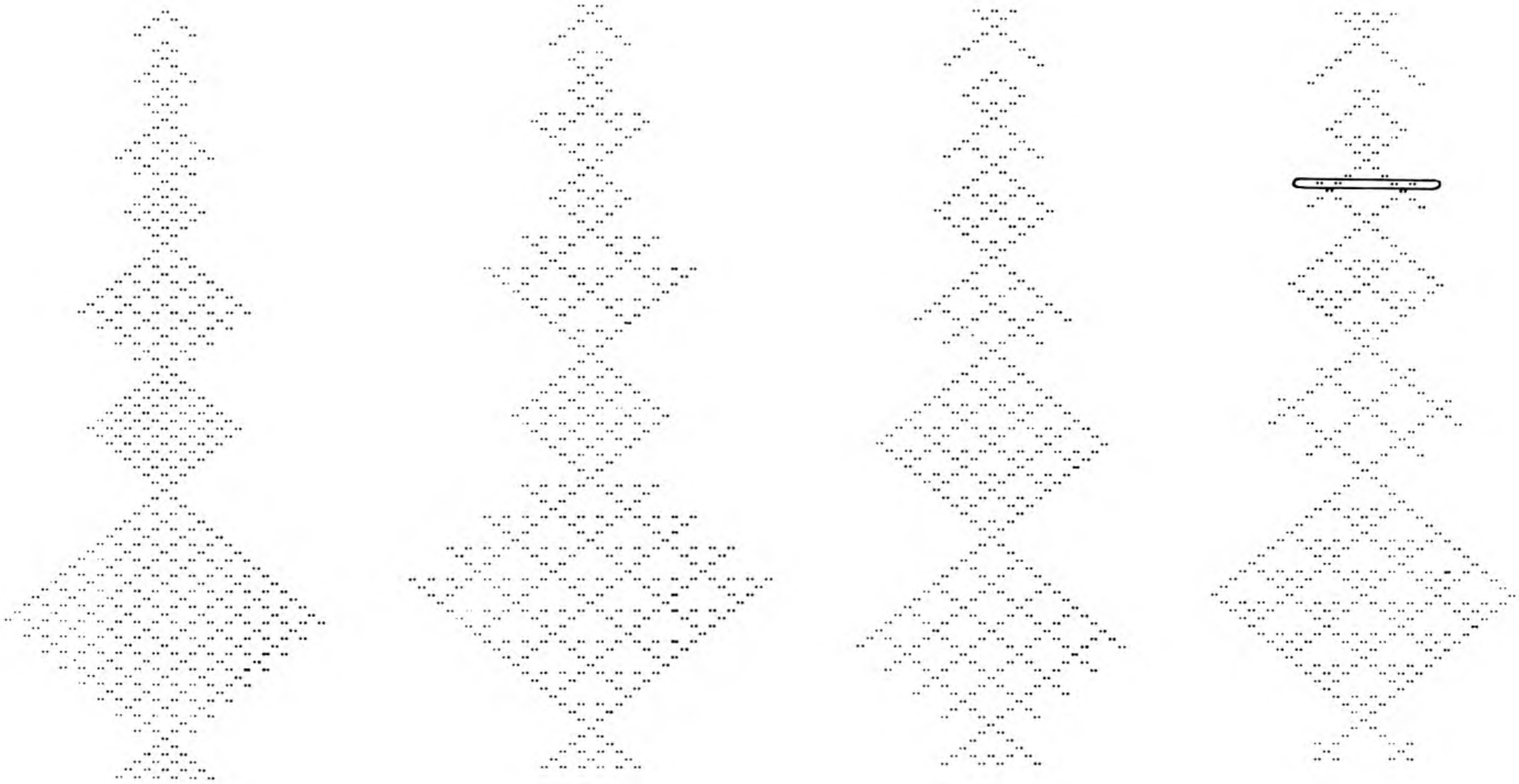


and the permutation

12 2 3 1 8 6 7 5 9 10 11 4 20 14 15 13 17 18 19 16 24 22 23 21 25 26

takes 652 time units when given as input to Program 2.4.

We have seen throughout our study of the best case that the best choice of the partitioning element is not in general unique. Our transformations on the trees often have no effect on the cost. Figures 4.2 show all choices of partitioning elements for M from 0 to 5 and for $M+1 \le N \le 100$. There is a ** corresponding to each choice of the partitioning element which leads to the best case. Each line is centered around the middle of the page. For example, the circled line in Figure 4.2a indicates that, if $M = 3$ and $N = 26$, then the elements with rank 11, 12, 15 and 16 are equally good choices for the partitioning element. Our "canonical form" and proofs restricted our attention to only one of these on each line: the element with rank $\left\lceil \frac{N+1}{2(M+1)} - \frac{1}{2} \right\rceil (M+1)$. These are circled, for $M = 4$, in Figure 4.2b. Notice that the patterns for $M = 0$ and 1 are different -- this is because, as mentioned above, the relative weight of the stack push unbalances the trees. We recognize many other aspects of these complex and intricate designs from the analysis given above. Indeed, the reason that the analysis itself was so long and difficult is that it is necessary to account for all of the features of these diagrams. It is fitting to end our study of the best and worst case with these interesting figures, for they tell us nearly as much about Quicksort as the involved formulas that we have labored so hard to derive.



M = 0

M = 1

M = 2

M = 3

Figure 4.2a. Best case partitioning elements $(M+1 \le N \le 100)$.

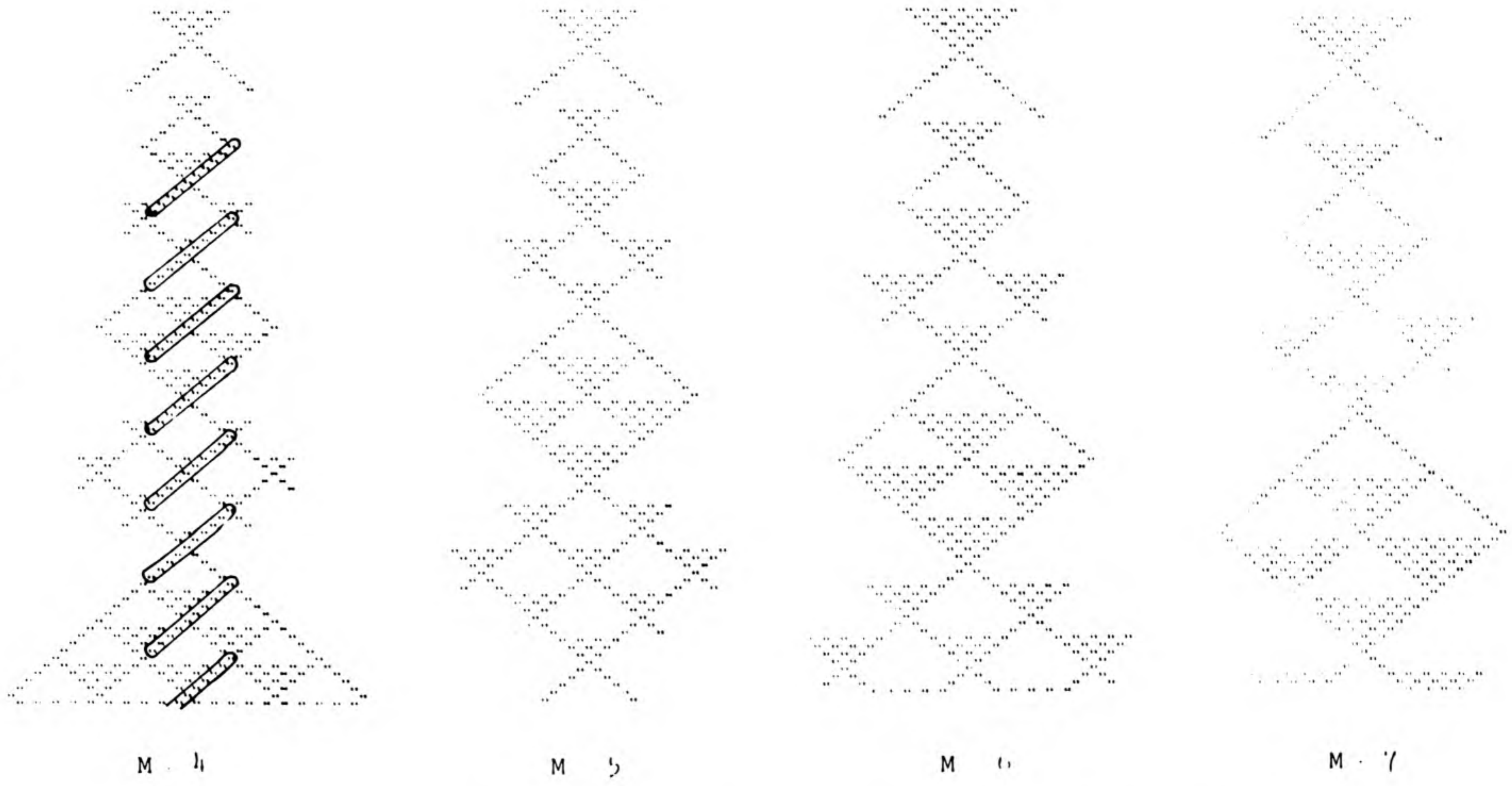


Figure 4.2b. Best case partitioning elements (cont.)

CHAPTER FIVE

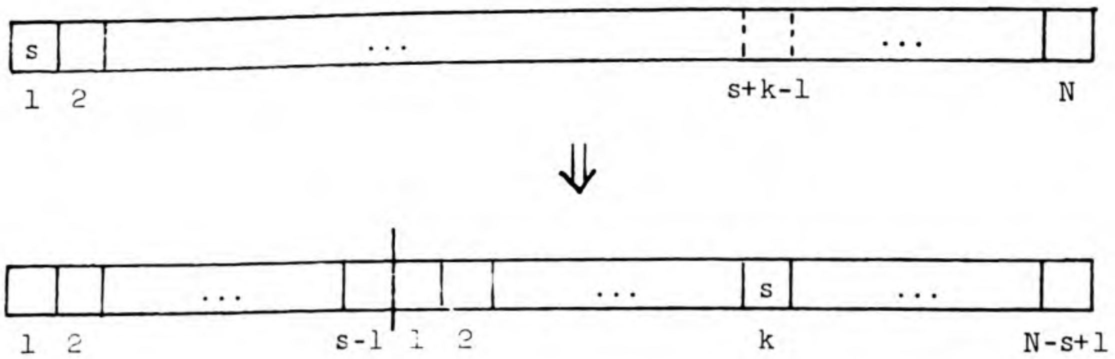
Our analysis during the last two chapters has told us a great deal about the Quicksort algorithm, but we have been so deep into the analysis that we have nearly lost sight of our original program. Therefore, it is appropriate at this time to use what we have learned during the analysis to study some more practical modifications of the program. In addition, now that we are competent and confident in the methods of analysis, we will be able to compare most of the variants of the algorithm intelligently. This will not be quite as easy as it might seem. We saw in Chapter 2 that minor changes in the partitioning strategy can have major effects on the performance of the algorithm -- we will see in this chapter that there may be drastic effects on the analysis as well.

First, we will look back at the various partitioning methods that we studied in Chapter 2 and justify some of the arguments made there in light of what we now know from the analysis. This will include an attempt to analyze Partitioning Method 2.3, so that we may have some indication of why it is desirable to have random subfiles. Next, we will look at some better methods of choosing the partitioning element, all designed to make the occurrence of the worst case less likely in a practical situation. These have little impact on the analysis except for a modification of Method 2.3 which can be analyzed (an analysis which is quite different from what we have seen so far). Finally, in order to compare Methods 2.2 and 2.4, we look at the situation when equal keys may be present in the file. This leads us to a "two-partition" Quicksort algorithm, and we will conclude the chapter by studying the very interesting aspects of the analysis of this algorithm.

The analysis of the average running time of Quicksort using Partitioning Method 2.1 is essentially the same as the analysis in Chapter 3. Although the program does two "half" exchanges (rather than one full exchange) in the inner loop, the same argument that we used to find the average value of the quantity B holds. However, as we noticed, the test " $i \geq j$ " is performed twice per exchange, so the coefficient of B is higher. This is the reason that we rejected this method. It is possible to improve this method by simply deleting one of the tests, then repairing the damage outside the inner loop. This results in a method which is less elegant than Method 2.4, but which could be more efficient on machines where it is inconvenient to implement full exchanges. Another difference in the analysis occurs because the algorithm is not quite symmetric between left and right -- a little more overhead is involved when the left pointer crosses the right pointer than when the right crosses the left. A new quantity X must be included in the analysis to account for this. As we might expect, it turns out that the average value of X is $\frac{1}{2}$ the average value of A .

If all of the keys are distinct, then Partitioning Method 2.2 performs exactly the same as Partitioning Method 2.4 when the leftmost element is used as the partitioning element. It is not difficult to see that if this change is made ($p := \ell$ rather than $p := (\ell+r) \div 2$) in Method 2.2, then Example 2.4 describes its performance as well as the performance of Method 2.4. This means that the average values of all the quantities we studied are the same and, on inspecting the program, we see that the coefficients of B and C are not changed and the coefficient of A is slightly higher for Method 2.2. (Depending on the implementation, some new quantities with average value $\approx \frac{1}{2} A$ may also be involved as in Method 2.1 -- see Appendix A.) The main difference between the methods, of course, occurs when equal keys are present. We shall defer discussion of this question until later in the chapter.

In the discussion of Partitioning Method 2.3, we discovered that the method produces nonrandom subfiles, and therefore it violates one of the basic assumptions of our analysis. We are now in a position to study more closely the difficulty of analyzing this method. We shall restrict our attention to what should be the simplest quantity to analyze: A_N , the average number of partitioning stages required to sort N elements, under the assumption that all $N!$ permutations of them are equally likely as input. To further simplify the calculations, we will assume that the leftmost element is used as the partitioning element at every stage ($p := \ell$ in Partitioning Method 2.3). To begin, we notice that the subfiles are almost random -- only the smallest element in the right subfile is in a nonrandom position after partitioning. Therefore it is appropriate to begin the analysis by conditioning on the first partitioning element and setting up a recurrence, in the same way as before. Again we assume that the numbers $1,2,\ldots,N$ are being sorted, and at the first stage each element s , $1 \le s \le N$, is equally likely to be used as the partitioning element (i.e., each number s is equally likely to be leftmost in the file). If $s = 1$, then it stays where it is and a random subfile of length $N-1$ is left after partitioning. Otherwise, if $s > 1$, then the first exchange will put it into the right subfile, and all positions are not equally likely. The left subfile, on the other hand, will be random. We have the following situation:



The left subfile consists of $(s-1)$ randomly ordered keys; and the right subfile consists of $(N-s+1)$ keys which are randomly ordered except for the smallest key s which does not fall into each position with equal probability. We need to find the probability that s falls into position k in the right subfile. This is the probability that, in the original file, all of the $N-s+1-k$ keys at the right end of the file (A[s+k], A[s+k+1], ..., A[N]) were $> s$ and that A[s+k-1] was $< s$. This leads to the expression

$$\frac{s-1}{N-1}\,\frac{\binom{N-s}{N-s+1-k}}{\binom{N-2}{N-s+1-k}} = \frac{(s-1)}{(N-1)}\,\frac{(N-s)!}{(k-1)!}\,\frac{(s+k-3)!}{(N-2)!}\cdot\frac{(s-2)!}{(s-2)!}$$

$$= \frac{\binom{s+k-3}{s-2}}{\binom{N-1}{s-1}} ,$$

for the probability that s falls into position k in the right subfile. Now, if we define A_{Nk} to be the average number of stages to sort a file of N elements which is randomly ordered except for the smallest element, which is in position k , then the above discussion leads to the recurrence

$$A_N = 1 + \frac{1}{N}\left(A_{N-1} + \sum_{2 \le s \le N}\left(A_{s-1} + \sum_{1 \le k \le N-s+1} \frac{\binom{s+k-3}{s-2}}{\binom{N-1}{s-1}} A_{(N-s+1)k}\right)\right).$$

Also, directly from the definitions, we know that

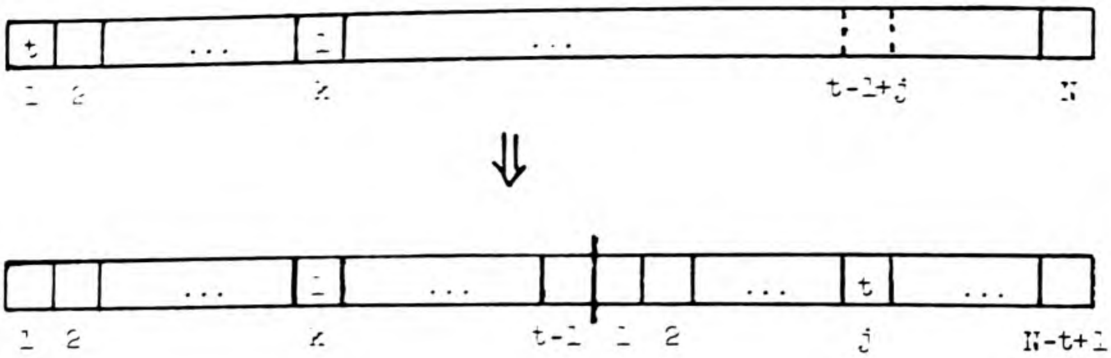
$$A_N = \frac{1}{N} \sum_{1 \le k \le N} A_{Nk} \quad .$$

These equations hold for $N > M$ and we will simplify things by taking $M = 0$ and $A_0 = A_{0k} = 0$. From these equations we see that in order to solve for A_N , we need to have a formula for A_{Nk} (or one of the sums involving A_{Nk}).

We can use the same method to set up a recurrence relation for A_{Nk} . We start with a permutation of $\{1,2,\ldots,N\}$, with the 1 in position k and the rest randomly arranged. Let us call the leftmost element in this permutation t , so that we may examine the result of partitioning this file on t . If $t = 1$, then by our definitions $k = 1$ also, so that a random file of size N-1 is left after partitioning. In other words,

$$A_{N1} = 1 + A_{N-1} \quad .$$

If $k < t$, then we have a slightly more complicated situation:



It is easy to see that the 1 is not moved, because the exchanges only move elements which are not initially in the proper subfile. The left subfile still has its smallest element (the 1) in position k , so $A_{(t-1)k}$ stages will be required, on the average, to sort it. The right subfile, by the same argument as used above, has its smallest element (the t) in position j with probability

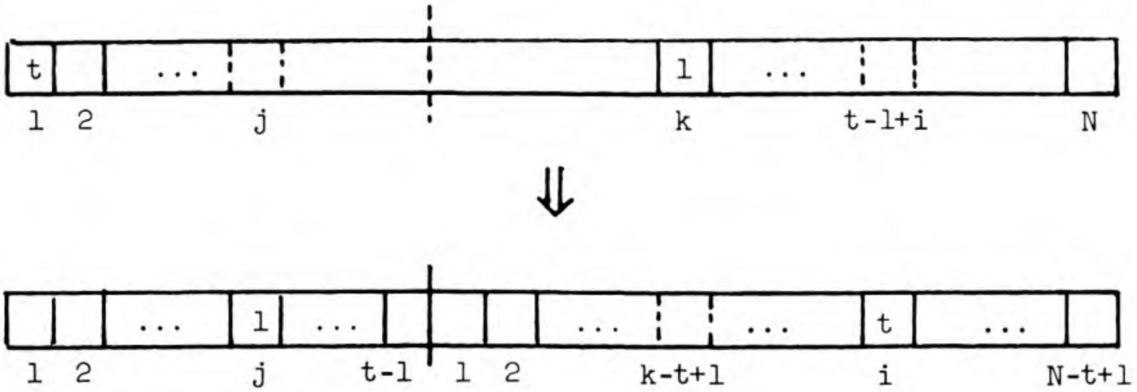
$$\frac{t-2}{N-2} \frac{\binom{N-t}{N-t+1-j}}{\binom{N-3}{N-t+1-j}} = \frac{\binom{t+j-4}{t-3}}{\binom{N-2}{t-2}} ,$$

so that an average of

$$\sum_{1 \le j \le N-t+1} \frac{\binom{t+j-4}{t-3}}{\binom{N-2}{t-2}} A_{(N-t+1)j}$$

stages will be required to sort the right subfile.

The remaining case, when $k \geq t$, is the most complicated, since both the 1 and the t are moved by partitioning:



Again we need to compute the probability distribution for the smallest element in both of these subfiles. For the right subfile, the probability that the t falls into position i is zero for $1 \leq i < k-t+1$, since we know that the t must at least switch with the 1. For $i = k-t+1$, we are dealing with the case when the t and the 1 are exchanged, which can occur only if all of the elements $A[k+1], A[k+2], \ldots, A[N]$ in the original file are $> t$. The probability that this occurs is

$$\frac{\binom{N-t}{N-k}}{\binom{N-2}{N-k}} \quad , \quad \text{or} \quad \frac{\binom{k-3}{t-3}}{\binom{N-2}{t-2}} \quad .$$

For $i > k-t+1$, we have the same situation as above, and the t will fall in position i with probability

$$\frac{\binom{t+i-4}{t-3}}{\binom{N-2}{t-2}} \quad .$$

Therefore an average of

$$\frac{\binom{N-t}{N-k}}{\binom{N-2}{N-k}} A_{(N-t+1)(k-t+1)} + \sum_{k-t+2 \leq i \leq N-t+1} \frac{\binom{t+i-4}{t-3}}{\binom{N-2}{t-2}} A_{(N-t+1)i}$$

partitioning stages are needed to sort the right subfile. All that remains to consider is the left subfile for the case $k \geq t$. Now, the 1 falls into position j if and only if: (i) the key $A[j]$ in the original file is $> t$; and (ii) the $k-j-1$ keys $A[j+1],\ldots,A[k-1]$ comprise exactly $t-j-1$ keys which are $< t$ (which will fall to the right of the 1 in the left subfile) and $k-t$ keys which are $> t$ (which will fall to the left of position $k-t+1$ in the right subfile). This occurs with probability

$$\frac{N-t}{N-2} \frac{\binom{t-2}{t-j-1}\binom{N-t}{k-t}}{\binom{N-3}{k-j-1}} \quad , \quad \text{or} \quad \frac{\binom{k-j-1}{t-j-1}\binom{N-k+j-2}{j-1}}{\binom{N-2}{t-2}} \quad .$$

Putting this all together, we have our recurrence for A_{Nk} when $k > 1$:

$$A_{Nk} = 1 + \frac{1}{N-1} \sum_{2 \le t \le N} \left\{ \begin{array}{c} \text{avg. no. of partitioning stages if} \\ \text{t is the partitioning element} \end{array} \right\}$$

$$= 1 + \frac{1}{N-1} \sum_{2 \le t \le k} \left(\sum_{1 \le j \le t-1} \frac{\binom{k-j-1}{t-j-1}\binom{N-k+j-2}{j-1}}{\binom{N-2}{t-2}} A_{(t-1)j} \right.$$

$$\left. + \frac{\binom{N-t}{N-k}}{\binom{N-2}{N-k}} A_{(N-t+1)(k-t+1)} + \sum_{k-t+2 \le j \le N-t+1} \frac{\binom{t+j-4}{t-3}}{\binom{N-2}{t-2}} A_{(N-t+1)j} \right)$$

$$+ \frac{1}{N-1} \sum_{k < t \le N} \left(A_{(t-1)k} + \sum_{1 \le j \le N-t+1} \frac{\binom{t+j-4}{t-3}}{\binom{N-2}{t-2}} A_{(N-t+1)j} \right)$$

$$A_{Nk} = 1 + \frac{1}{N-1} \sum_{2 \le t \le k} \sum_{1 \le j \le t-1} \frac{\binom{k-j-1}{t-j-1}\binom{N-k+j-2}{j-1}}{\binom{N-2}{t-2}} A_{(t-1)j}$$

$$+ \frac{1}{N-1} \sum_{2 \le t \le k} \frac{\binom{N-t}{N-k}}{\binom{N-2}{N-k}} A_{(N-t+1)(k-t+1)}$$

$$+ \frac{1}{N-1} \sum_{3 \le t \le N} \sum_{k-t+2 \le j \le N-t+1} \frac{\binom{t+j-4}{t-3}}{\binom{N-2}{t-2}} A_{(N-t+1)j}$$

$$+ \frac{1}{N-1} \sum_{k < t \le N} A_{(t-1)k}$$

$$+ \frac{1}{N-1} \sum_{k < t \le N} \sum_{1 \le j \le k-t+1} \frac{\binom{t+j-4}{t-3}}{\binom{N-2}{t-2}} A_{(N-t+1)j} .$$

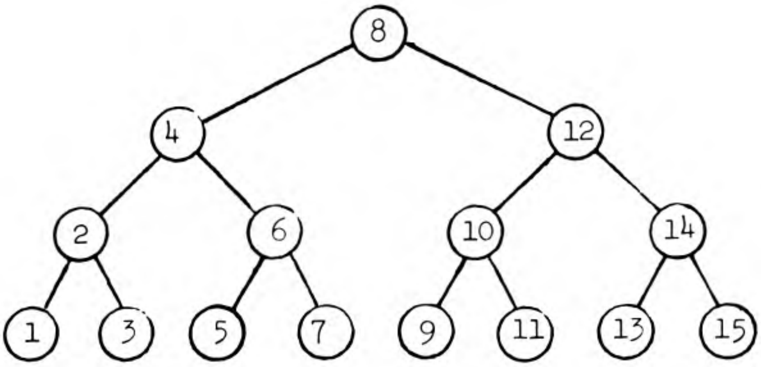
This recurrence appears impossible to solve, mainly because of the first term, which involves summing over the lower index of a binomial coefficient appearing in the denominator. We can, of course, use this equation to compute the values of A_{Nk} . Such a computation leads to values of A_N for this method which are significantly higher than for all the other methods that we have seen. This can also be verified by "simulation", i.e., running the methods on a variety of "random" files, and for all $N!$ files when N is small. Thus we have good reason to believe that the maintenance of random subfiles is as desirable from a practical standpoint as it is necessary for thorough analysis.

In Chapter 3 we found that Quicksort takes time proportional to $N \ln N$ on the average, and in Chapter 4 we saw that the algorithm takes time proportional to N^2 in the worst case. Furthermore, we saw that the standard deviation is low, so that in the probabilistic sense, this worst case is not very likely to occur. However, from a practical standpoint, the worst case (or close to it) is <u>very</u> likely to occur, for the algorithm performs very badly when the keys are already in order. Unfortunately, we are rarely faced with a truly "random" set of keys to sort. On the contrary, in practical applications it is common to find some natural order in the keys. Some sorting methods, such as insertion sorting, take advantage of this and run more efficiently when such order is present. Program 2.4, on the other hand, is handicapped by this: any long run of keys already in their proper position will eventually span a whole subfile, for which the program will run very slowly. This effect should clearly be avoided if possible -- it is somewhat embarrassing to have a sorting method which is slowest when sorting a file that is already in order! Fortunately, it is not difficult to determine the cause of this problem. The culprit is the "arbitrary" choice of the first element as the partitioning element, which was done as a matter of convenience in specifying the algorithm. We will now investigate some more intelligent ways to choose the partitioning element.

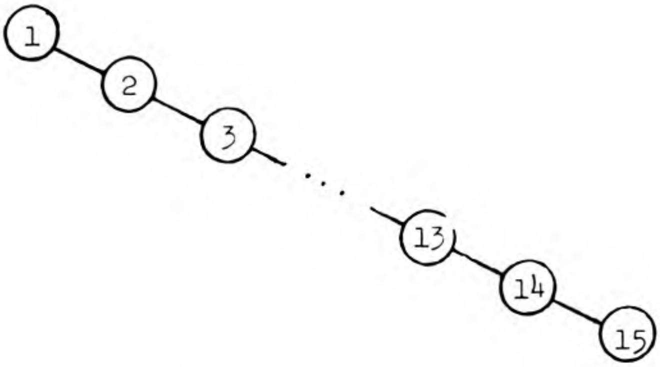
First, we might consider choosing some other fixed element from the subfile, as we did in some of the methods in Chapter 2. The last element would be as bad a choice as the first: a more logical candidate is the element in the middle. In order to avoid the various difficulties that we encountered in Chapter 2, we will implement this idea by simply interchanging the first element with the one we have chosen, then using Partitioning Method 2.4 as before. In other words, we insert the statement $A[\ell] :=: A[(\ell+r) \div 2]$ into Program 2.4 just before the statement $v := A[\ell]$. This will not affect our analysis, because if the file is randomly ordered, any fixed element is a random choice from the file. The average running time will be increased because of a slight increase in the coefficient of A (we have added one exchange for each partitioning stage), but it is a small price to pay since it results in much faster running times for many input files which might occur in practice. It is obvious that this algorithm performs very well for the "worst case" permutations that we saw for Program 2.4. For example, for $N = 15$, the partitioning tree for

1 2 3 4 5 6 7 8 9 10 11 12 13 14 15

is



as opposed to the worst case tree for Program 2.4,



In fact, files which are already in order lead to the best case for this choice of the partitioning element. However, partial order in the input can still cause this method to perform badly. If we think of a very large file, with the smallest or largest elements occurring in order near the middle, then we see that it is a mistake to partition on the middle element, for the first several partitions will be degenerate. If there is any order whatever in our original file, then it is not unlikely that this unfortunate situation will occur in one or more subfiles during the sorting process. Also, similar difficulties arise if any fixed element is used as the partitioning element.

A more attractive method, which was suggested by Hoare in his original paper, is to use a "random" element from the file as the partitioning element. As above, we can implement this by inserting the statement

```
        A[ℓ] :=: A[random(ℓ,r)]
```

just before the statement v := A[ℓ] in Program 2.4, where **random**(ℓ,r) is a procedure which returns a random integer between ℓ and r. This approach is attractive for two reasons: First, it will help make the worst case less likely, in the same way as above. Second, if a good random number generator is used, it makes our analysis of the average running time much more realistic. If our file is already random, this modification, like the last, has no effect. However, even if biases do appear in the file, this method ensures that every element is equally likely to be the partitioning element, so that the analysis in Chapter 2 remains valid. We will not concern ourselves with the details of implementing random number generators here, except to note that they are relatively expensive, and would result in the coefficient of the quantity A in Program 2.4 being increased by 100 to 200%. Again, this is a price worth paying if it is known that there will be extreme biases in the files to be sorted. Increasing M will reduce some of this cost.

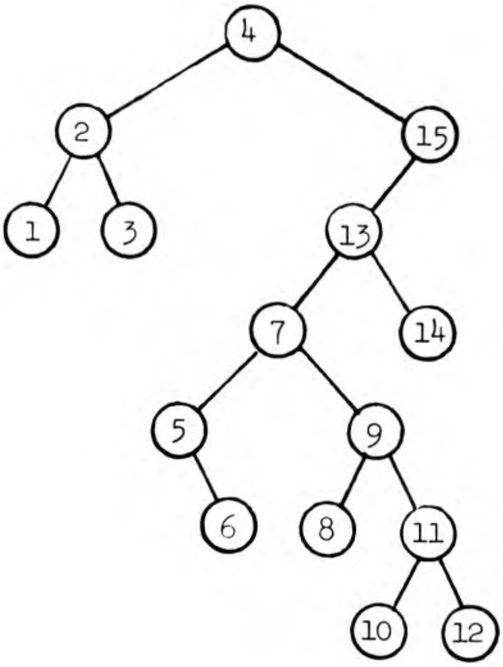
Another approach, which may be more efficient in some situations, stems from the observation that we don't really require numbers having all of the characteristics of truly random numbers. We certainly can expect to find some randomness in our files -- otherwise it would not be necessary to write a sorting program. It is therefore reasonable to consider, for example, a simpler method, where the same relative position is chosen for all subfiles of a given size, as in the following

```
procedure random (ℓ,r);
        $\ell + \lfloor (r-\ell+1)\{10^{r-\ell+1}\pi\}\rfloor$ ;
```

Here the braces ({ }) are meant to indicate the "fractional part": $\{x\} = x - \lfloor x \rfloor$ for $x \geq 0$. On a binary computer, $2^{r-\ell+1}$ should be used, so that the inner multiplication can be implemented as a "shift". The use of π as the multiplier is arbitrary -- any other irrational number would do as well. The computation of this kind of function is more efficient than the generation of random numbers, and it adequately serves our purposes for many applications. For this particular function, the small values are

$r-\ell+1$	2	3	4	5	6	7	8	9	10	11	12	13	14	15
$\lfloor (r-\ell+1)\{10^{r-\ell+1}\pi\}\rfloor$	0	1	3	1	3	3	2	5	8	10	8	12	4	3

From this table, we can find the partitioning tree when this modification of Program 2.4 is used on the keys 1 to 15 in order (with $M = 1$):



This is a typical partitioning tree, far from the worst case. Our analysis does not tell us exactly how this method performs on the average unless we assume that the input file is random (in which case it is unnecessary). However, this method requires less overhead than random number generation, and may, in some situations, represent an appropriate balance between practical and theoretical considerations in choosing the partitioning element.

It is interesting to notice that the same effect as choosing a random partitioning element can be achieved by randomly "scrambling" the entire array before sorting it, as in the following

```
procedure scramble;
    begin i := 1;
          loop:
                A[i] :=: A[random(i,N)];
          while i < N:
                i := i+1;
          repeat;
    end;
```

This method is usually less efficient, since it always requires N exchanges and random number calculations, while these operations are performed only $2\frac{N+1}{M+2} - 1$ times on the average in choosing random partitioning elements. As above, we might improve this by not scrambling the file randomly, but rather scrambling it systematically; but to compete with other methods, we would still need a procedure which runs about $\frac{M+2}{2}$ times as fast as the random number generator, which might be difficult to devise.

We have seen several simple methods to make the worst case of Program 2.4 less likely in practical situations, and we will not dwell further on the relative merits of these methods. Of course, none of the suggestions above will eliminate the worst case -- we can always find a permutation for which a method makes the worst choice for the partitioning element at every stage.

If this idea of choosing a random partitioning element is applied to Partitioning Method 2.3, then we get a method which, although it does not always put the partitioning element into position, does produce random subfiles, and so should submit to analysis as do our other methods. We expect this method to be slightly less efficient than our other methods because of the fact that the partitioning element usually falls into one of the subfiles, making it one larger than in the other methods. The analysis which proves this is of a quite different character than we have seen before, so we will now pursue it further.

We are mainly interested in methods of analysis, so we will restrict our attention to finding an expression for A_N , the average number of partitioning stages taken by the program

```
procedure quicksort (integer value $\ell$,r);
     if r-$\ell$ $\geq$ M then
        i := $\ell$-1; j := r+1; p := random($\ell$,r); v := A[p];
        loop:
              loop:  i := i+1; while A[i] < v repeat;
              loop:  j := j-1; while A[j] > v repeat;
        while i < j:
              A[i] :=: A[j];
        repeat;
        quicksort ($\ell$ , i-1);
        quicksort (j+1 , r);
     endif;
```

under the assumption that all of the $N!$ permutations of the integer $\{1,2,\ldots,N\}$ are equally likely as input. The same methods can be extended to find the number of exchanges, comparisons, etc., and to find the exact running time of a practical version of this program as in Chapter 2 and 3. In order to simplify our calculations somewhat, we will take $M = 0$ in our analysis, even though this magnifies differences between methods. For reference, we may compare our result with

$$A_N = N$$

the number of stages required by our other methods when $M = 0$.

As always, we begin by conditioning on the first partitioning stage, so that we have the recurrence

$$A_N = 1 + \frac{1}{N} \sum_{1 \leq s \leq N} \left\{ \begin{matrix} \text{average number of stages required for the} \\ \text{subfiles when } s \text{ is the partitioning element} \end{matrix} \right\} .$$

Complications arise because the sizes of the subfiles vary depending on the position, p, of the partitioning element. We noticed in Chapter 2 that if $p = s$ (the partitioning element is already in place) then the pointers will meet at $A[s]$ and the subfiles will have $s-1$ and $N-s$ elements, as in our other methods. If $p < s$, then the partitioning element will be encountered by the i pointer before the pointers have met, and therefore exchanged into the right subfile, so the subfiles after partitioning will have $s-1$ and $N-s+1$ elements. Similarly, if $p > s$, then the partitioning element will end up in the left subfile, so the two subfiles will have s and $N-s$ elements. Unfortunately, for fixed p and s, we cannot always say that the two subfiles produced will be random. For example, if $p = s-1$, then s will be the smallest element of the right subfile, and it will tend to be very near the left end of that file. However, the subfiles are random for fixed s, and we can develop a recurrence by considering separately the three cases $p < s$, $p = s$, and $p > s$. If $p = s$, the subfiles are random by the same argument that we used in Chapter 2, since we never know anything about the relative order of the other elements. If $p < s$, which occurs with probability $\frac{s-1}{N}$, then s is equally likely to be involved in any one of the exchanges which occur during partitioning. In other words, it is exchanged with a random element from the right subfile, so it is equally likely to fall in each of the $N-s+1$ positions in the right subfile. This is sufficient to show that the subfiles are random, since our previous argument clearly holds for the other elements in the subfiles. A similar argument holds for the symmetric case $p > s$, so we are left with the recurrence

$$A_N = 1 + \frac{1}{N} \sum_{1 \le s \le N} \left(\frac{s-1}{N} (A_{s-1} + A_{N-s+1}) + \frac{1}{N} (A_{s-1} + A_{N-s}) + \frac{N-s}{N} (A_s + A_{N-s}) \right),$$

which holds for $N \geq 1$, with $A_0 = 0$. This can be simplified through a series of straightforward algebraic manipulations,

$$A_N = 1 + \frac{1}{N^2} \sum_{1 \leq s \leq N} (s-1)(A_{s-1} + A_{N-s+1}) + \frac{1}{N^2} \sum_{1 \leq s \leq N} (A_{s-1} + A_{N-s})$$

$$+ \frac{1}{N^2} \sum_{1 \leq s \leq N} (N-s)(A_s + A_{N-s})$$

$$N^2 A_N = N^2 + \sum_{0 \leq s \leq N-1} s(A_s + A_{N-s}) + 2 \sum_{0 \leq s \leq N-1} A_s$$

$$+ \sum_{0 \leq s \leq N-1} (N-s)(A_s + A_{N-s}) - N(A_N + A_0)$$

$$= N^2 + N \sum_{0 \leq s \leq N-1} A_s + N \sum_{0 \leq s \leq N-1} A_{N-s} - NA_N + 2 \sum_{0 \leq s \leq N-1} A_s$$

$$= N^2 + 2(N+1) \sum_{0 \leq s \leq N-1} A_s \quad ,$$

to the formula

$$\frac{N^2}{N+1} (A_N - 1) = 2 \sum_{0 \leq s \leq N-1} A_s \quad .$$

As we have done before, we can eliminate the summation by subtracting, from this last equation, the same equation for $(N-1)$, leaving

$$\frac{N^2}{N+1} (A_N - 1) - \frac{(N-1)^2}{N} (A_{N-1} - 1) = 2 A_{N-1} \quad ,$$

which can be rewritten as

$$\frac{A_N - 1}{N+1} = \left(1 + \frac{1}{N^2}\right) \frac{A_{N-1} - 1}{N} + \frac{2}{N^2} \quad .$$

We are now very close to having a recurrence which telescopes into a sum. Multiplying both sides of this last equation by the "summation factor"

$\prod_{j \geq N+1} \left(1 + \frac{1}{j^2} \right)$, we get

$$\frac{A_N - 1}{N+1} \prod_{j \geq N+1} \left(1 + \frac{1}{j^2} \right) = \frac{A_{N-1} - 1}{N} \prod_{j \geq N} \left(1 + \frac{1}{j^2} \right) + \frac{2}{N^2} \prod_{j \geq N+1} \left(1 + \frac{1}{j^2} \right) ,$$

which telescopes to yield

$$\frac{A_N - 1}{N+1} \prod_{j \geq N+1} \left(1 + \frac{1}{j^2} \right) = \sum_{2 \leq k \leq N} \frac{2}{k^2} \prod_{j \geq k+1} \left(1 + \frac{1}{j^2} \right) .$$

It remains only to evaluate this sum and product. Unfortunately, there seems to be no simple "closed form" expression for $\prod_{j \geq k+1} \left(1 + \frac{1}{j^2} \right)$, and we must be content with an asymptotic result. The first step is to notice that this product is very close to 1 for large k , so it is convenient to separate out the "dirty" terms by defining

$$f(k) = \prod_{j \geq k+1} \left(1 + \frac{1}{j^2} \right) - 1 ,$$

which, when substituted into the recurrence, gives

$$\frac{A_N - 1}{N+1} (f(N) + 1) = \sum_{2 \leq k \leq N} \frac{2}{k^2} + \sum_{2 \leq k \leq N} \frac{2}{k^2} f(k)$$

$$= 2(H_N^{(2)} - 1) + \sum_{2 \leq k \leq N} \frac{2}{k^2} f(k) .$$

Next, we notice that this sum certainly converges to some constant

$$c \equiv \sum_{k \geq 2} \frac{2}{k^2} f(k) ,$$

so that we can further isolate the small terms in our formula by separating out the "tail" of the summation:

$$\frac{A_N - 1}{N+1}\,(f(N)+1) = 2(H_N^{(2)} - 1) + c - \sum_{k \geq N+1} \frac{2}{k^2}\, f(k) \quad .$$

At this point it is necessary to resort to asymptotic estimates of the small terms (see Eqs. (50) - (54) in Appendix B). First, from the three basic formulas

$$\ln(1+x) = x - \frac{x^2}{2} + \ldots + \frac{(-1)^{k+1}}{k}\, x^k + O(x^{k+1}) \quad ;$$

$$e^x = 1 + x + \frac{x^2}{2} + \ldots + \frac{x^k}{k!} + O(x^{k+1}) \quad ;$$

and

$$\sum_{1 \geq n} \frac{1}{i^a} = \frac{1}{(a-1)n^{a-1}} + \frac{1}{2n^a} + O \quad \frac{1}{n^{a+1}}$$

we can derive an asymptotic formula for our product:

$$\begin{aligned}
\prod_{j \geq k+1}\left(1 + \frac{1}{j^2}\right) &= \exp\left\{\ln \prod_{j \geq k+1}\left(1 + \frac{1}{j^2}\right)\right\} \\
&= \exp\left\{\sum_{j \geq k+1} \ln\left(1 + \frac{1}{j^2}\right)\right\} \\
&= \exp\left\{\sum_{j \geq k+1}\left(\frac{1}{j^2} + O\left(\frac{1}{j^4}\right)\right)\right\} \\
&= \exp\left\{\frac{1}{k} - \frac{1}{2k^2} + O\left(\frac{1}{k^3}\right)\right\} \\
&= 1 + \frac{1}{k} - \frac{1}{2k^2} + \frac{1}{2k^2} + O\left(\frac{1}{k^3}\right) \quad .
\end{aligned}$$

This implies that

$$f(k) = \frac{1}{k} + O\left(\frac{1}{k^3}\right) ,$$

so

$$\sum_{k \geq N+1} \frac{2}{k^2} f(k) = \sum_{k \geq N+1} \frac{2}{k^2} \left(\frac{1}{k} + O\left(\frac{1}{k^3}\right)\right)$$

$$= \frac{1}{N^2} + O\left(\frac{1}{N^3}\right) ,$$

and

$$\frac{1}{f(N)+1} = 1 - \frac{1}{N} + \frac{1}{N^2} + O\left(\frac{1}{N^3}\right) .$$

Substituting all of this into our equation for A_N gives the answer

$$\frac{A_N - 1}{N+1} = \left(1 - \frac{1}{N} + \frac{1}{N^2} + O\left(\frac{1}{N^3}\right)\right)\left(2H_N^{(2)} - 1 + c - \frac{1}{N^2} + O\left(\frac{1}{N^3}\right)\right) .$$

By multiplying this out and simplifying, we can get a formula for A_N to within $O\left(\frac{1}{N^2}\right)$. In fact, with the methods we have used, we could derive a formula to <u>any</u> asymptotic accuracy. For now, we will be content with $O\left(\frac{1}{N}\right)$, so, using the formula above, we get

$$A_N = 2N\left(H_N^{(2)} - 1 + \frac{c}{2}\right) + 1 + O\left(\frac{1}{N}\right) .$$

Finally, applying Eq. (53) in Appendix B, we get the final answer

$$A_N = N\left(\frac{\pi^2}{3} - 2 + c\right) - 1 + O\left(\frac{1}{N}\right) \quad .$$

The constant c evaluates to $.3862\ldots$, so $A_N \approx (1.677)N$, which is significantly higher than for our other methods. Of course, we expect this difference to be smaller for larger M , and indeed, we can use the same methods as above to show that

$$A_N = \frac{2N}{M+2} - 2 + O\left(\frac{N}{M^3}\right) + O\left(\frac{1}{N}\right) \quad ,$$

which is fairly close to our previous result of $A_N = \frac{2(N+1)}{M+2} - 1$, until N gets very much larger than M . Also, we can use these methods to get asymptotic formulas for all of the other quantities. For example, when $M = 0$, the average number of comparisons is

$$C_N = N(2H_N + c') - 1 + O\left(\frac{H_N}{N}\right) \quad ,$$

where

$$c' = \sum_{k \geq 1} \left(\frac{2}{k} f(k) - \frac{2}{k^2(k+1)} (1 + f(k))\right) \approx .923\ldots \quad .$$

Although this method is always slightly less efficient than the others, it is interesting to see the radical effect of such a minor perturbation on the analysis.

It has been convenient in our analyses to assume that all of the keys A[1], A[2], ..., A[N] are distinct. Of course, this may not always occur in practice, so we now will study the operation of Quicksort when equal keys are present. In some applications it is important that the sorting algorithm not change the relative order of equal keys. This property is called stability. Unfortunately, Quicksort is not a stable algorithm, no matter how we treat keys equal to the partitioning element. For example, suppose that a file consists only of 1's , 2's , and 3's ; 2 is the partitioning element; and the file contains the pattern

...	3	1	...	3	1	...

Then the relative order of both the 3's and the 1's must be disturbed by any of our normal partitioning methods. An easy way to make any sorting algorithm stable is to force all the keys to be distinct before sorting, by appending each key's index to itself:

```
i := 1;
loop:
      A[i] := A[i] * N + i - 1;
while $i \le N$:
      i := i+1
repeat;
```

This transformation, in addition to making all the keys distinct, preserves the relative order of the keys. We have $A[i] < A[j]$ before the transformation if and only if $A[i] < A[j]$ after the transformation, except for the equal keys: if $i < j$ and $A[i] = A[j]$ before, then $A[i] < A[j]$ after. We now achieve a stable method by sorting the file and then transforming back to our original keys:

```
i := 1;
loop:
      A[i] := $\lfloor A[i]/N \rfloor$
while $i \le N$:
      i := i+1
repeat;
```

Of course, this method is costly in terms of both time and space (each key must be a little bigger), so that it should not be used unless stability is important and even then it may not be practical.

The main reason that we assume distinct keys in our analyses is that it is difficult to model the situation simply when equal keys are present. However, we can feel somewhat justified in treating only distinct keys, because in most practical situations the "key space" (all possible key values) is very large, so that the probability that a significant number of equal keys will be present is very small. This means that the effect of equal keys, when dealing with the average performance, can generally be safely ignored. If it is known that only a small number of key values are possible, then a method which takes advantage of that fact should be used. (One example of this, when the range of keys is known to be small, is a method called "distribution counting", which involves making two passes through the file: one to count the number of occurrences of each key, and a second to move the keys into place according to the counts.)

However, equal keys do occur in many applications, and even though we have ignored them in our analysis, there is no reason to ignore them in our programs. We have seen that Partitioning Methods 2.2 and 2.4 perform almost exactly the same for all permutations when the keys are distinct, but when equal keys are present they perform quite differently. For example, given the input file

2 2 2 2 1 1 1 2 2 3 3 3 3 3 3 ,

Partitioning Method 2.2 will produce the partition

2 2 2 2 1 1 1 (2 2) 3 3 3 3 3 3

(and when the left subfile is partitioned, the left pointer will cross over into the 3's), while Partitioning Method 2.4 results in the less balanced partition

1 2 2 1 1 (2) 2 2 2 3 3 3 3 3 3 .

On the other hand, Method 2.2 performs worse for the input file

2 3 3 3 3 3 3 2 1 1 1 2 2 2 2 ,

since it produces the partition

2 1 1 1 (2) 3 3 3 3 3 3 2 2 2 2 ,

while Method 2.4 partitions the file perfectly:

$$1\ 2\ 2\ 2\ 2\ 1\ 1\ \textcircled{2}\ 2\ 3\ 3\ 3\ 3\ 3\ 3 \quad .$$

It is dangerous to attempt to draw conclusions from such anomalous cases, but they do help to illustrate differences between the algorithms. Fortunately we can prove that Partitioning Method 2.4 will always result in a partition closer to the center, on the average, if it is assumed that all of the $N!$ arrangements of the N (not necessarily distinct) input keys are equally likely. This is a direct consequence of the following:

Theorem 5.1. When Partitioning Method 2.2 (with $p := r$) operates on a file $A[1],\ldots,A[N]$, it produces a partition no closer to the center than Partitioning Method 2.4 operating on the reverse of that file.

Proof. Specifically, let j and i define the position of the partition after Method 2.2 (with $p := r$) is used on $A[1],\ldots,A[N]$, so that after partitioning we have $A[1],A[2],\ldots,A[j] \le A[j+1] = A[j+2] = \ldots = A[i-1] \le A[i],A[i+1],\ldots,A[N]$; and let j' define the position of the partition after Method 2.4 is used on $A[N],\ldots,A[1]$, so that after partitioning we have $A[1],A[2],\ldots,A[j'-1] \le A[j'] \le A[j'+1],\ldots,A[N]$. In both cases, the file is partitioned on the value of $A[N]$. Call that value v and let t be the number of keys in the file which are $< v$. Our goal will be to show that the inequality $\left|j' - \frac{N+1}{2}\right| \le \left|t+k - \frac{N+1}{2}\right|$ holds for $j-t < k < i-t$.

First we notice that since Method 2.2 does not move keys which are $= v$, we can have $j = t+k$ only if exactly k of the keys $A[1],\ldots,A[t+k]$ were originally $= v$. In fact, this must be true for all k in the range $j-t \le k < i-t-1$ since $A[j+1],\ldots,A[i-2]$ are all $= v$ and are not moved. (When $k = i-t-1$, we know that $k-1$ of the keys $A[1],\ldots,A[i-1]$ were $= v$, since the last exchange must have been $A[i-1] :=: A[N]$.) Similarly, since Method 2.4 always moves keys which are $= v$, then $j' = t+k'$ for some fixed k' only if there were exactly $k'-1$ keys $= v$ in the last $N-t-k'+1$ positions of the reverse file: $A[N-t-k'+1],\ldots,A[1]$.

To complete the proof it is necessary to consider three cases depending on the relative values of k and k'. If $k = k'$, then the inequality $\left|j' - \frac{N+1}{2}\right| \le \left|t+k - \frac{N+1}{2}\right|$ obviously holds. If $k > k'$, then the discussion above says that $A[1],\ldots,A[t+k]$ has more keys $= v$ than $A[1],\ldots,A[N-t-k'+1]$, which can only be true if $t+k > N-t-k'+1$, or $j' > N-t-k+1$. (Note that this argument holds even when $k = i-t-1$.) Now, if $j' - \frac{N+1}{2}$ is ≥ 0, then $k' < k$ implies that $0 \le j' - \frac{N+1}{2} < t+k - \frac{N+1}{2}$; and if $\frac{N+1}{2} - j'$ is ≥ 0 then $j' > N-t-k+1$ implies that $0 \le \frac{N+1}{2} - j' < t+k - \frac{N+1}{2}$. In either case, taking absolute values gives the desired result,

$$\left|j' - \frac{N+1}{2}\right| \le \left|t+k - \frac{N+1}{2}\right| .$$

The argument for the third case, $k' > k$, is slightly more complex. First, if $k = i-t-1$, then $k-1$ of the keys $A[1],\ldots,A[t+k]$ were $= v$ for Method 2.2, which is fewer than the $k'-1$ keys of $A[N-t-k'+1],\ldots,A[1]$ which must have been $= v$ for Method 2.4, so $t+k < N-t-k'+1$, or $j' < N-t-k+1$. For other values of k, we can only say that the number of keys $= v$ in $A[N-t-k'+1],\ldots,A[1]$ is greater than <u>or equal to</u> the number of keys $= v$ in $A[1],\ldots,A[t+k]$. This would not imply anything about the array bounds, were it not for the fact that $A[t+k] = v$ for $j-t < k < i-t-1$, so now we must have $N-t-k'+1 \geq t+k$, or $j' \leq N-t-k+1$. Finally, an argument symmetric to that in the above paragraph shows that $\left| j' - \frac{N+1}{2} \right| \leq \left| t+k - \frac{N+1}{2} \right|$ and we have shown that this holds for all k in the range $j-t < k < i-t$.

The theorem follows immediately from this inequality. If the first partition is to the left of center $\left(i-1 < \frac{N+1}{2} \right)$ then the second is at least as close $\left(\left| \frac{N+1}{2} - j' \right| < \left| \frac{N+1}{2} - i \right| \right)$; and the symmetric argument holds for the right. If the first partition straddles the center, or $j+1 \leq \frac{N+1}{2} \leq i-1$, then $\left| t+k - \frac{N+1}{2} \right| \leq \frac{1}{2}$ for some k , and therefore $\left| j' - \frac{N+1}{2} \right| \leq \frac{1}{2}$, or the second partition must also be at the center. □

Theorem 5.1 establishes a one-to-one correspondence between permutations of keys used as input to Method 2.2 and permutations of keys used as input to Method 2.4. The same result holds for general p : let $A[p],A[N],A[N-1],\ldots,A[p+1],A[p-1],\ldots,A[2],A[1]$ be used for Method 2.4 when $A[1],\ldots,A[N]$ is used for Method 2.2. (The examples given above fit into this construction for $p = (\ell+r) \div 2$.) Therefore, if we average over all such permutations, Method 2.4 will result in an average partition at least as close to the center. This does not necessarily mean that it will always run faster, but the intuition that we have gained from our study of the best case in Chapter 4 tells us that it is desirable to have the partition as close to the center as possible.

Of course, Method 2.2 could still yield smaller subfiles in some cases, since its partition might include more than one element. (In fact, if all the keys are equal, it requires only one stage to sort the entire file.) However, this sometimes turns out to be a liability, as we see when we examine Example 5.1. The effects are intentionally exaggerated in this example through the use of binary files. (The particular pattern used is from the binary representation of e.) Also, to make things easier to follow, it is assumed in Example 5.1 and in the arguments below that $p := \ell$ is used in Partitioning Method 2.2. The behavior for Method 2.2 as it stands is similar. Now, when 0 is the partitioning element, the left pointer stops at the first 1 and the right pointer scans the whole file (no element is < 0) in Method 2.2. In Method 2.4, the left pointer stops at every position and the right pointer stops at each 0. The pointers behave similarly when 1 is the partitioning element. The most glaring defect of Method 2.2 is that one of the pointers therefore goes all the way to the end of the file on every partitioning stage, even when an interior subfile is being partitioned. This can be corrected, in a less efficient algorithm, by testing for the conditions $i \geq \ell$ and $j \leq r$ during the pointer scans rather than using $-\infty$ and ∞ keys to stop the pointers, but even this does not reduce the number of comparisons sufficiently, as we shall now see.

Suppose that C_N is the average number of comparisons required by Method 2.2 to sort a binary file of N 0's and 1's under the assumption that all 2^N such files are equally likely, and that no comparisons are made outside the range $[\ell,r]$. Let $C_N^{(0)}$ and $C_N^{(1)}$ be the averages for files that begin with 0 and 1. First, we will find a recurrence for $C_N^{(0)}$ by noticing that the situation after the first stage of Partitioning Method 2.2 is as follows ("x" denotes keys which may be 0 or 1):

$$\overbrace{\underset{\substack{\uparrow\\ j}}{0}00\ldots0}^{k}\overbrace{\underset{\substack{\uparrow\\ i}}{1}xxx\ldots x}^{N-k}$$

Example 5.1

Method 2.2 $(p := \ell)$:

```
0 1 0 1 0 1 1 0 1 1 1 1 1 1 0 0 0 0 1 0 1 0 1 0
⓪ 1 0 1 0 1 1 0 1 1 1 1 1 1 0 0 0 0 1 0 1 0 1 0
  0 0 1 0 1 1 0 1 1 1 1 1 1 0 0 0 0 1 0 1 0 1 ①
  ⓪ ⓪ 1 0 1 1 0 1 1 1 1 1 1 0 0 0 0 1 0 1 0 1
      0 0 1 1 0 1 1 1 1 1 1 0 0 0 0 1 0 1 ① ①
      ⓪ ⓪ 1 1 0 1 1 1 1 1 1 0 0 0 0 1 0 1
          0 1 0 1 1 1 1 1 1 0 0 0 0 1 ① ①
          ⓪ 1 0 1 1 1 1 1 1 0 0 0 0 1
            0 0 1 1 1 1 1 1 0 0 0 ① ①
            ⓪ ⓪ 1 1 1 1 1 1 0 0 0
                0 1 1 1 1 1 0 0 ①
                ⓪ 1 1 1 1 1 0 0
                  0 1 1 1 1 0 ①
                  ⓪ 1 1 1 1 0
                    0 1 1 1 ①
                    ⓪ 1 1 1
                      ① ① ①
```

Method 2.4:

```
0 1 0 1 0 1 1 0 1 1 1 1 1 1 0 0 0 0 1 0 1 0 1 0
0 0 0 0 0 0 0 ⓪ 1 1 1 1 1 1 0 1 1 0 1 1 1 0 1 1
0 0 0 ⓪ 0 0 0
0 ⓪ 0
        0 ⓪ 0
                0 1 1 0 1 1 0 1 ① 1 1 1 1 1 1 1
                0 0 ⓪ 1 1 1 1 1
                0 ⓪
                      1 1 ① 1 1
                      1 ①
                            1 ①
                                  1 1 1 ① 1 1 1
                                  1 ① 1
                                          1 ① 1
```

Partitioning required N+k+1 comparisons, and all that is left to be sorted is a file of size N-k , random, except for its first key, which is 1 . If the file is all 0's (k = N) , this is not quite correct, since 2N comparisons are required and the file is sorted. This leads us to the recurrence

$$C_N^{(0)} = \frac{2N}{2^{N-1}} + \sum_{1 \le k \le N-1} \frac{1}{2^k} (N + k + 1 + C_{N-k}^{(1)}) \quad .$$

By a similar argument, we can show that

$$C_N^{(1)} = \frac{2N}{2^{N-1}} + \sum_{1 \le k \le N-1} \frac{1}{2^k} (N + k + C_{N-k}^{(0)}) \quad ,$$

and therefore $C_N = \frac{1}{2}\left(C_N^{(0)} + C_N^{(1)}\right)$ satisfies

$$C_N = \frac{2N}{2^{N-1}} + \sum_{1 \le k \le N-1} \frac{1}{2^k} (N + k + \frac{1}{2} + C_{N-k}) \quad .$$

We can use the same methods that we have used before to solve this equation. Multiplying by 2^N , we have

$$2^N C_N = 4N + \sum_{1 \le k \le N-1} 2^{N-k}(N + k + \frac{1}{2} + C_{N-k})$$

$$= 4N + \sum_{1 \le k \le N-1} 2^k(2N - k + \frac{1}{2} + C_k) \quad ,$$

and subtracting the same equation for N-1 gives

$$2^N C_N - 2^{N-1} C_{N-1} = 4 + 2^{N-1}(N + \frac{3}{2} + C_{N-1}) + \sum_{1 \le k \le N-2} 2^{k+1} \quad .$$

This simplifies to give the recurrence

$$C_N = C_{N-1} + \frac{N}{2} + \frac{7}{4} \quad , \quad \text{for } N \ge 2$$

with the initial condition $C_1 = 2$, which immediately telescopes to the solution

$$C_N = \frac{1}{4} (N^2 - 1) + 2N \quad .$$

We might have expected that this average number of comparisons would be proportional to N^2 if we had noticed that two successive stages simply exchange the leftmost 1 with the rightmost 0 .

On the other hand, we can show that the number of comparisons required by Partitioning Method 2.4 is proportional to $N \lg N$ in the worst case. This is suggested by an examination of Example 5.1: Notice that each partition results in one subfile with all keys equal and one "unsorted" subfile. The subfiles with all keys equal are clearly processed in a logarithmic number of stages, since they are always split in the middle. Now consider the unsorted subfile. After each partitioning stage, at least half of the keys equal to the partitioning element must be removed. Therefore, the unsorted part of the file cannot last through more than $\lg N$ partitions, and every element in the whole file is involved in at most $\lg N$ partitioning stages. We may count the total number of comparisons by noticing that each partitioning stage contributes one comparison to the total for each element involved plus one extra comparison when the pointers cross. By these arguments, the total number of comparisons taken by Method 2.4 on binary files can be no more than $N \lg N + N$. This is substantially better than the quadratic performance of Method 2.2.

Although it is interesting to consider the problem of sorting binary files, the results should not be taken too seriously, since the effects are so exaggerated. We have plenty of other evidence to convince us that Method 2.4 is preferable when equal keys are present. Theorem 5.1 shows that the partitions are no farther from the center on the average, and whatever advantage is gained by the fact that Method 2.2 may have more than one key in the partition is lost because the pointers may scan past the bounds of interior subfiles. Of course, this does not represent a complete argument that Method 2.4 is more efficient (since we have not considered exchanges and other overhead), but we will not dwell further on this subject.

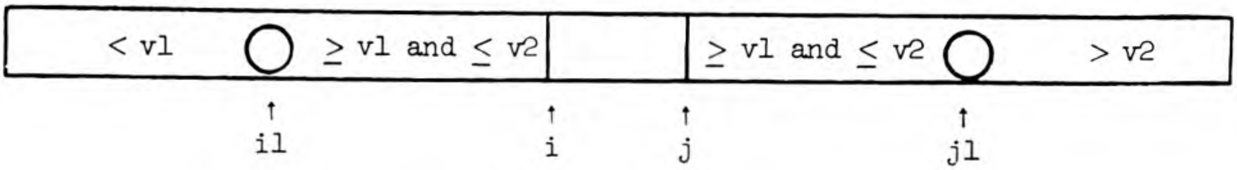
An important practical reason to stop on equal keys is that it makes easier the definition of the $-\infty$ and ∞ keys that we use to stop the pointers. For example, suppose that our numbers to be sorted can take on any value which we can represent in one word on our computer. By definition, we can't represent a key larger than all of these numbers in one word, but if we use Program 2.4, we only need a key larger than or equal to all of these numbers: i.e., the largest representable number can serve as the ∞ key, and the smallest representable number as the $-\infty$ key. If this is still inconvenient, the need for the $-\infty$ can be eliminated by switching the direction of the insertion sort (cf. Appendix A). The sentinels can be eliminated altogether, in a slightly less efficient partitioning method, by exchanging the first two keys if necessary before partitioning to put them out of order; then correcting the situation afterwards.

Ideally, when equal keys are present, we would like to have a partitioning method which puts all of the keys equal to the partitioning element into position. Such a method really results in the establishment of two partitions: all elements to the left of the first will be less than the partitioning element; all elements between the two will be equal to the partitioning element; and all elements to the right of the second will be greater than the partitioning element. But if we are going to have three subfiles, why not try to make them all about the same size? A more general, "two-partition" Quicksort is suggested, which puts two partitioning elements into position:

Program 5.1

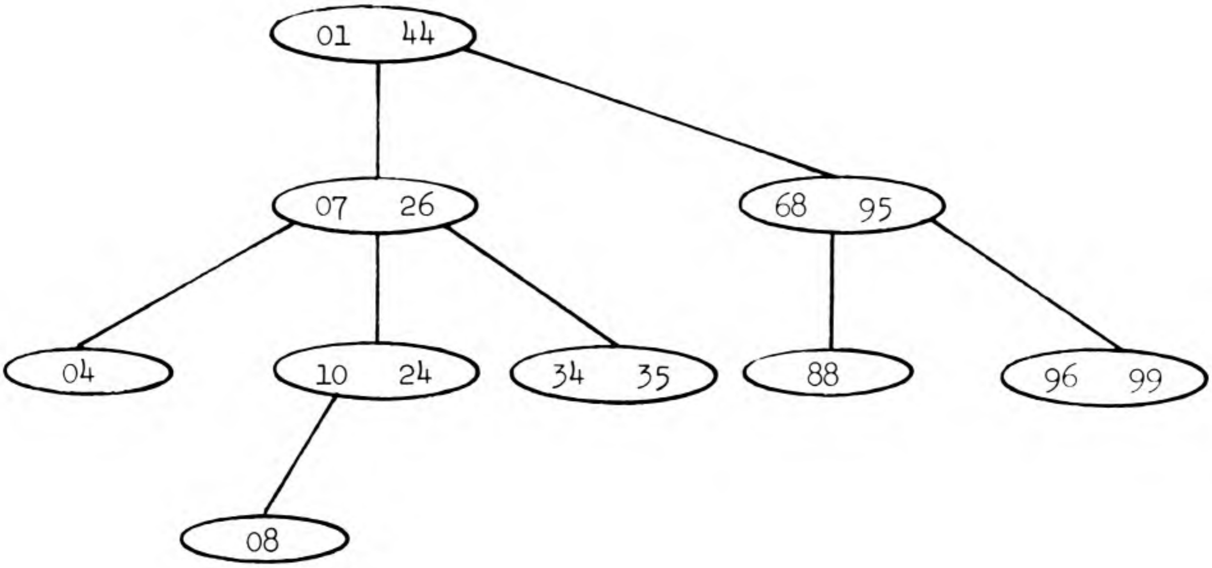
```
procedure quicksort (integer value ℓ,r);
    if r-ℓ ≥ M then                                                  A
        if A[ℓ] > A[r] then A[ℓ] :=: A[r] endif;
        i := il := ℓ; vl := A[ℓ];
        j := jl := r; v2 := A[r];
        loop until pointers have met:
            loop: i := i+1;
            while A[i] ≤ v2:                                         C'
                if A[i] < vl then                                    $C'_1$
                    A[il] := A[i];                                   B'
                    il := il+1;
                    A[i] := A[il];
                    if i ≥ j then pointers have met endif;
                endif;
            repeat;
            loop: j := j-1;
            while A[j] ≥ vl:                                         C-C'
                if A[j] > v2 then                                    $C_1-C'_1$
                    A[jl] := A[j];                                   B''
                    jl := jl-1;
                    A[j] := A[jl];
                    if i ≥ j then pointers have met endif;
                endif;
            repeat;
            A[il] := A[j];                                           B'''
            A[jl] := A[i];
            il := il+1;
            jl := jl-1;
            A[i] := A[il];
            A[j] := A[jl];
        repeat;
        A[il] := vl;
        A[jl] := v2;
        quicksort (ℓ , il-1);
        quicksort (il+1 , jl-1);
        quicksort (jl+1 , r);
    endif;
```

This program is much more complicated than the others we have studied, but it is based on many of the same ideas. Example 5.2 shows the operation of the method on our set of fifteen keys -- it can be described in a manner similar to Partitioning Method 2.1, except that there are two "holes" to be filled, initially at the left and right ends of the file. The partitioning elements v1 and v2 are picked from the first and last elements, such that v1 < v2 . The pointers i1 and j1 keep the current positions of the holes. The "invariant" at the main loop is the following situation:

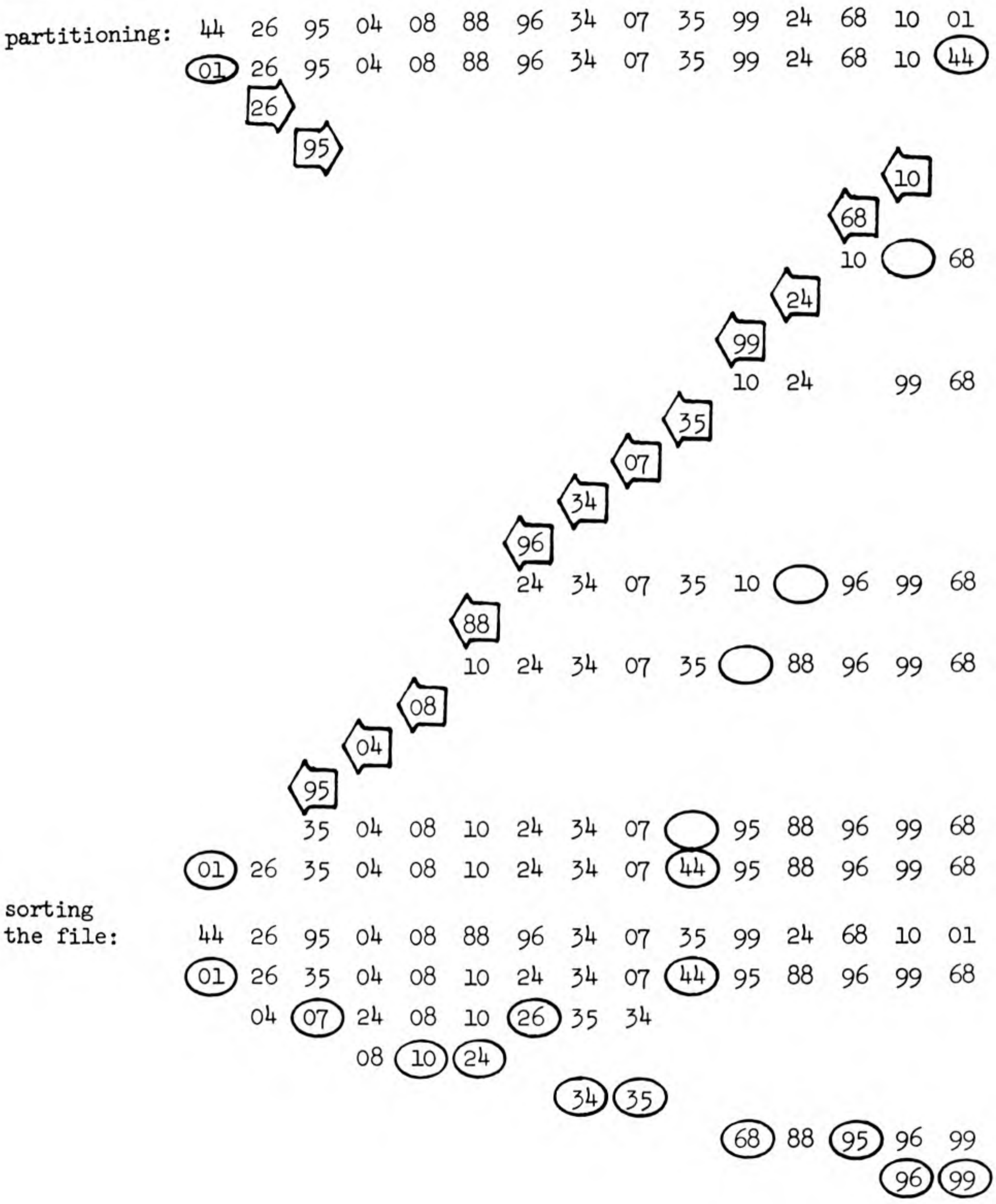


Both of the pointers i and j scan over keys between v1 and v2 . When the left pointer, i , encounters a key < v1 , it is put into the left hole, and the hole is moved one position to the right (by incrementing i1 and putting that element into A[i]). Similarly, when the right pointer, j , encounters a key > v2 , it is put into the right hole, and the hole is moved one position to the left. (In Example 5.2 the 68 , the 99 , the 96 , and the 95 are moved in this way.) Finally, when the i pointer has stopped on a key > v2 and the j pointer on a key < v1 , then each is put into the hole on the other side, and the holes moved appropriately. (This does not occur in the first stage in Example 5.2, but it does occur at the second partitioning stage.)

Clearly, these manipulations maintain the invariant conditions sketched above, and when the i and j pointers meet, then v1 belongs in the i1 hole; v2 in the j1 hole; and three subfiles are clearly defined. The partitioning tree is therefore a ternary tree: for Example 5.1 the tree is



Example 5.2


partitioning:
44 26 95 04 08 88 96 34 07 35 99 24 68 10 01
01 26 95 04 08 88 96 34 07 35 99 24 68 10 44
26
95
10
68
10 68
24
99
10 24 99 68
35
07
34
96
24 34 07 35 10 96 99 68
88
10 24 34 07 35 88 96 99 68
08
04
95
35 04 08 10 24 34 07 95 88 96 99 68
01 26 35 04 08 10 24 34 07 44 95 88 96 99 68
sorting the file:
44 26 95 04 08 88 96 34 07 35 99 24 68 10 01
01 26 35 04 08 10 24 34 07 44 95 88 96 99 68
04 07 24 08 10 26 35 34
08 10 24
34 35
68 88 95 96 99
96 99

Unfortunately, as we will soon see, this method cannot compete with our other partitioning methods. However, the method is based on an intriguing idea, the analysis is interesting, and it will serve as a good introduction to the more significant variants of Quicksort to come in the following chapters. For brevity, we will ignore such practical improvements as sorting the shortest subfile first (actually for this method it suffices to avoid sorting the longest subfile first), and concentrate on the "leading term" in the running time: the number of comparisons and exchanges (by "exchange" in this context we mean "move a hole" which requires the same amount of work). As we have done before, instructions whose frequencies represent these quantities are labelled in Program 2.4. Upon inspecting the program, we find that the number of exchanges is given by

$$B = 2B''' + B'' + B' \quad .$$

If the partitioning elements are s and t, with $1 \leq s < t \leq N$, then by counting the number of times the il pointer is incremented and the jl pointer is decremented we find that the quantity $B' + B'''$ contributes $(s-1)$ to the first partitioning stage, and $B'' + B'''$ contributes $(N-t)$. Averaging over all partitioning elements s and t, assuming all pairs equally likely, we see that the average contribution of $B' + B'''$ to the first partitioning stage is

$$\frac{2}{N(N-1)} \sum_{1 \le s \le N-1} \sum_{s < t \le N} (s-1) = \frac{2}{N(N-1)} \sum_{1 \le s \le N-1} (s-1)(N-s)$$

$$= \frac{2}{N(N-1)} \binom{N}{3} \quad \text{(see Eq. (22) in Appendix B)}$$

$$= \frac{N-2}{3} \quad .$$

Similarly, it turns out that the contribution of $B'' + B'''$ is also $\frac{N-2}{3}$. The number of exchanges is obtained by simply summing these: since $B = B'' + B''' + B' + B'''$, the average contribution of the first partitioning stage to B is $\frac{2}{3}(N-2)$. It is convenient to modify this slightly by adding the two "exchanges" needed to get the partitioning elements into place, giving $\frac{2}{3}(N+1)$ as the average contribution of the first stage to the number of exchanges.

Now we can, as we have before, set up a recurrence for the total average number of exchanges taken by Program 5.1:

$$B_N = \frac{2}{3}(N+1) + \frac{2}{N(N-1)} \sum_{1 \le s \le N-1} \sum_{s < t \le N} (B_{s-1} + B_{t-s-1} + B_{N-t}) , \quad N > M ;$$

with $B_N = 0$ for $N \le M$, so that in particular $B_{M+1} = \frac{2}{3}(M+2)$ and $B_{M+2} = \frac{2}{3}(M+3)$. Not surprisingly, after manipulating indices, these three sums all turn out to be the same (this is because the partitioning elements are random), and we have

$$B_N = \frac{2}{3}(N+1) + \frac{6}{N(N-1)} \sum_{0 \le s \le N-2} (N-s-1) B_s \quad ,$$

or

$$\binom{N}{2}B_N = 2\binom{N+1}{3} + 3\sum_{0 \le s \le N-2} (N-s-1)B_s \quad .$$

As before, we will try to eliminate the summation by subtracting this formula from the same formula for $(N+1)$. This can be succinctly expressed with the <u>difference</u> <u>operator</u> Δ , defined by $\Delta x_n = x_{n+1} - x_n$. Notice that for binomial coefficients $\Delta\binom{n}{k} = \binom{n+1}{k} - \binom{n}{k} = \binom{n}{k-1}$ for any k . Applying Δ to our recurrence, we get

$$\Delta\binom{N}{2}B_N = 2\binom{N+1}{2} + 3\sum_{0 \le s \le N-1} (N-s)B_s - 3\sum_{0 \le s \le N-2} (N-s-1)B_s$$

$$= 2\binom{N+1}{2} + 3\sum_{0 \le s \le N-1} B_s \quad .$$

In the same way, the remaining sum can be eliminated:

$$\Delta^2\binom{N}{2}B_N = 2(N+1) + 3B_N \quad .$$

Expanding the Δ , we have

$$\binom{N+2}{2}B_{N+2} - 2\binom{N+1}{2}B_{N+1} + \binom{N}{2}B_N = 2(N+1) + 3B_N$$

$$(N+2)(N+1)B_{N+2} - 2(N+1)\,N\,B_{N+1} + (N(N-1)-6)B_N = 4(N+1)$$

$$(N+2)(N+1)B_{N+2} - 2(N+1)\,N\,B_{N+1} + (N+2)(N-3)B_N = 4(N+1) \quad .$$

The next step is a "magic" rearrangement of the terms which will lead to a solution. (Indeed, much of this derivation may seem "magic", though we could easily get this far with the use of generating functions.) We will see a more important example, and some theoretical justification for this, in Chapter 8. The idea is to break up the middle term as follows:

$$(N+1)((N+2)B_{N+2} - (N-2)B_{N+1}) - (N+2)((N+1)B_{N+1} - (N-3)B_N) = 4(N+1) \quad .$$

After dividing by $(N+1)(N+2)$, we now get a recurrence which telescopes:

$$\frac{(N+2)B_{N+2} - (N-2)B_{N+1}}{N+2} = \frac{(N+1)B_{N+1} - (N-3)B_N}{N+1} + \frac{4}{N+2}$$

$$= \frac{(M+2)B_{M+2} - (M-2)B_{M+1}}{M+2} + 4 \sum_{M+1 \le k \le N} \frac{1}{k+2}$$

$$= \frac{10}{3} + 4(H_{N+2} - H_{M+2}) \quad .$$

This equation is the same as

$$NB_N = (N-4)B_{N-1} + 4\,N\,H_N + N\left(\frac{10}{3} - 4H_{M+2}\right) \quad ,$$

and we can make <u>this</u> recurrence telescope by multiplying through by $\frac{(N-1)(N-2)(N-3)}{24}$, giving

$$\binom{N}{4}B_N = \binom{N-1}{4}B_{N-1} + 4\binom{N}{4}H_N + \binom{N}{4}\left(\frac{10}{3} - 4H_{M+2}\right)$$

$$= \binom{M+1}{4}B_{M+1} + 4\sum_{M+2 \le k \le N}\binom{k}{4}H_k + \left(\frac{10}{3} - 4H_{M+2}\right)\sum_{M+2 \le k \le N}\binom{k}{4} \quad .$$

Both of these sums are easily evaluated (see Eqs. (19) and (23) in Appendix B), giving the result

$$\binom{N}{4}B_N = \frac{10}{3}\binom{M+2}{5} + 4\left(\binom{N+1}{5}\left(H_{N+1} - \frac{1}{5}\right) - \binom{M+2}{5}\left(H_{M+2} - \frac{1}{5}\right)\right)$$

$$+ \left(\frac{10}{3} - 4H_{M+2}\right)\left(\binom{N+1}{5} - \binom{M+2}{5}\right)$$

$$= 4\binom{N+1}{5}(H_{N+1} - H_{M+2}) + \frac{38}{15}\binom{N+1}{5} + \frac{4}{5}\binom{M+2}{5} \quad ,$$

so that our final answer is

$$B_N = \frac{4}{5}(N+1)(H_{N+1} - H_{M+2}) - \frac{38}{75}(N+1) + \frac{4}{5}\frac{\binom{M+2}{5}}{\binom{N}{4}} \quad .$$

This is asymptotically $\frac{4}{5}(N+1)\ln\left(\frac{N+1}{M+2}\right)$, nearly $2\frac{1}{2}$ times worse than the $\frac{1}{3}(N+1)\ln\left(\frac{N+1}{M+2}\right)$ exchanges required by our other algorithms. We need go no further with the analysis, for we can never hope to recoup this loss. (It does turn out that C_N is asymptotically $\frac{6}{5}(N+1)\ln\left(\frac{N+1}{M+2}\right)$, and that the total overhead due to comparisons is slightly lower than for any other method that we have seen so far.)

Our analysis up to this point has been concerned mainly with time and not space, since we showed in Chapter 2 that the strategy of sorting the small subfile first limits the space requirements to $\left\lfloor \lg \frac{N+1}{M+2} \right\rfloor$ stack entries. However, as we have noted, it is somewhat inconvenient to implement this idea in a high-level language, unless recursion is handled properly by the compiler. Since Program 2.2 is much shorter and simpler than Program 2.4, we might legitimately ask just how much space it wastes. As always, we begin by conditioning on the first partitioning stage, so that the average maximum stack depth required by Program 2.2 on a random permutation of $\{1,2,\ldots,N\}$ is described by the recurrence

$$D_N = 1 + \frac{1}{N} \sum_{1 \le s \le N} \max(D_{s-1}, D_{N-s}) \quad , \qquad N \ge 1$$

(with $D_0 = 0$) since the maximum stack depth needed is 1 plus the maximum stack depth needed for the subfiles. If we assume that $D_i \le D_j$ for $i \le j$ then this simplifies to

$$N D_N = N + 2 \sum_{\lceil \frac{N+1}{2} \rceil \le s \le N-1} D_s + \begin{cases} D_{\lceil \frac{N-1}{2} \rceil} & , \quad N \text{ odd} \\ 0 & , \quad N \text{ even} \end{cases} .$$

Subtracting the same formula for $N-1$ gives

$$N D_N = (N+1) D_{N-1} - D_{\lfloor \frac{N-1}{2} \rfloor} + 1 \quad , \qquad\qquad N > 1 \ .$$

This recurrence appears very difficult to solve explicitly, but an easy computer calculation shows that it grows slowly with respect to N : D_{10000} is only about 28 . Therefore Program 2.2 will require less than 28 stack entries to sort less than 10000 elements on the average, and even fewer if it is extended to insertion sort small subfiles. Of course, the worst case may still occur -- if Program 2.2 as it stands is used to sort the file 1 2 3 ... 10000 , it will require 10000 stack entries. But this is the worst case, and we have looked at several techniques to make unbalanced partitioning trees very unlikely. If the worst case does occur, then space overflow is a convenient "alarm" that tells us that the program might take much too long to sort the given file. On balance, Program 2.2 with " A[ℓ] :=: A[random(ℓ,r)] " (or even " A[ℓ] :=: A[(ℓ+r)÷2] ") inserted just before " v := A[ℓ] " will perform very well as a "quick and dirty" sorting program when space is not a pressing constraint. The extension to insertion sort small subfiles is probably worthwhile, and of course Program 2.4 is always preferable if the program is to be used often.

We have seen a variety of simple modifications to Quicksort in this chapter, and we should suspect that there may be more sophisticated variants which may significantly improve the performance of the algorithm. But we also should expect that care should be exercised, since improvements in one part of the program may be offset by extra overhead in another. In the next three chapters we shall examine three strategies designed to improve Quicksort, and we shall analyze their effects on its performance.

CHAPTER SIX

Everything that we have learned about the Quicksort algorithm tells us that the algorithm performs best when the partitioning element is close to the middle of the file being partitioned at every stage. Yet the methods we have seen for choosing the partitioning element simply involve picking one element and relying on the randomness of the files to ensure that, on the average, the partitions will be near the center. We should expect to be able to do better than that, but we must be careful to balance out the costs involved. Indeed, there are algorithms which will find the median of n elements in time proportional to n : if we were to use such an algorithm to find our partitioning elements, we would have a Quicksort with a worst case running time of $O(n \lg n)$. However, such a method is not practical because of the comparatively high overhead required to find the median at each partitioning stage. We would like to examine methods somewhere between these two extremes: methods which tend to partition close to the center, but at relatively low costs. Such methods may compete, on a practical basis, with the best sorting algorithms that we have seen.

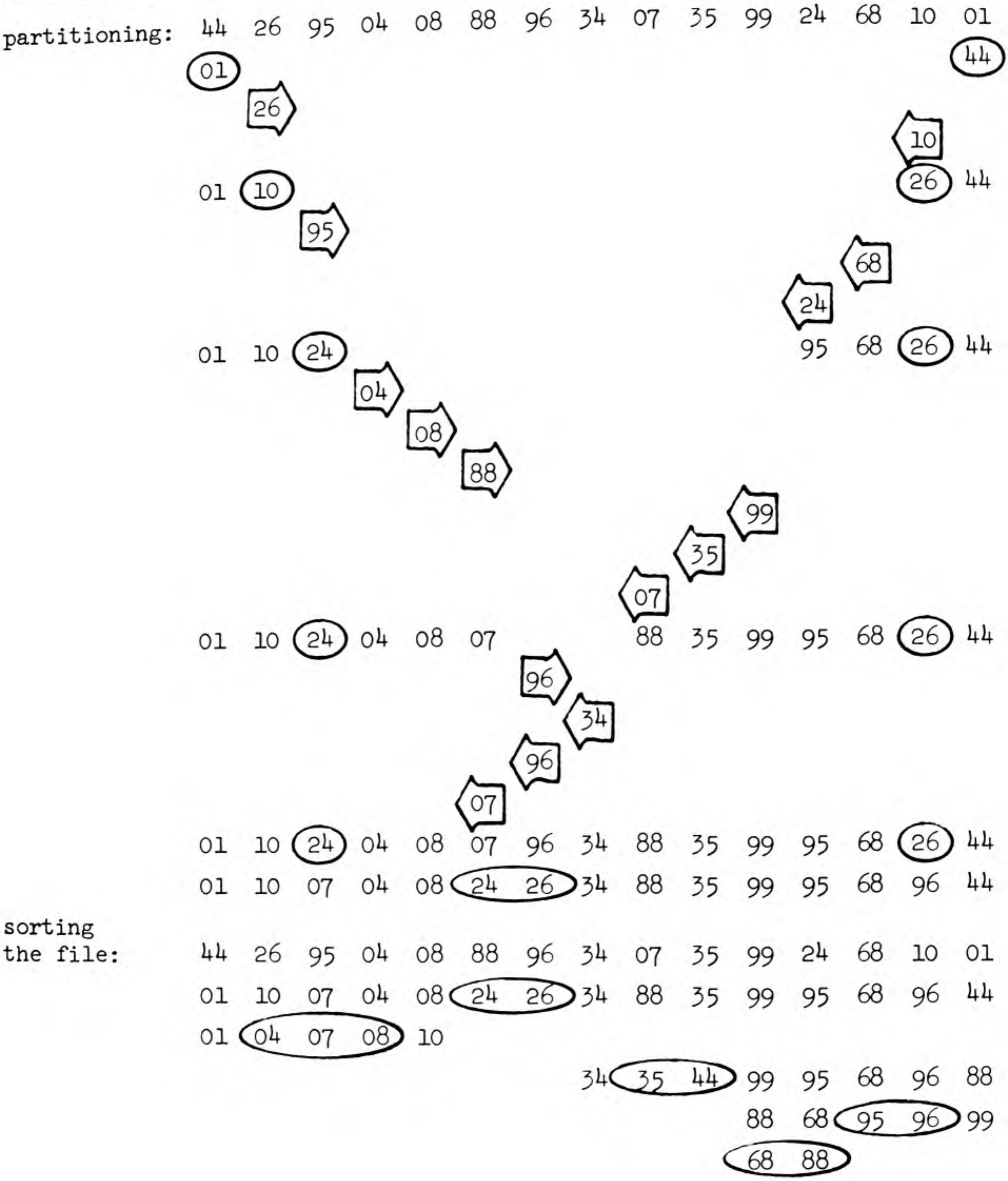
The method that we will study in this chapter is based on a technique introduced by M. H. van Emden in 1970. The idea is to delay, as long as possible, the decision on what should be the partitioning element. As the left and right pointers move in, we keep track of the largest element found so far in the left subfile and the smallest element found so far in the right subfile. The partitioning element might turn out to be any element between these bounds. Elements outside these limits are scanned over or exchanged just as in our normal methods, and the bounds are adjusted when new elements within them are encountered. The following program is a more complete definition of this "adaptive" partitioning scheme:

Program 6.1

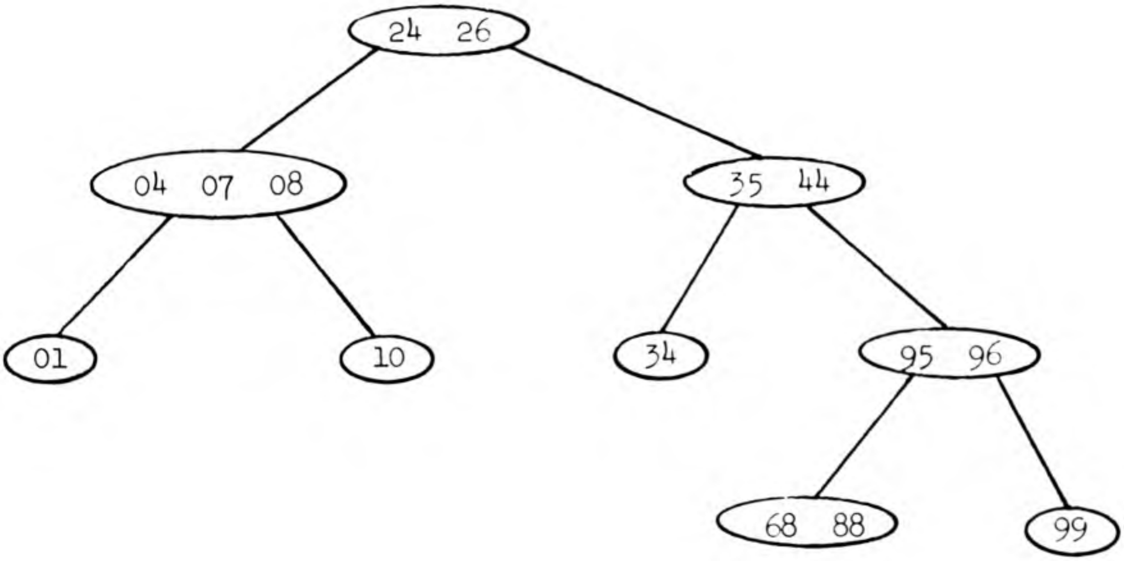
```
procedure quicksort (integer value ℓ,r);
    if r-ℓ ≥ M then
        i := ℓ; j := r; vmax := -∞; vmin := ∞;
        loop:
            if A[i] > A[j] then A[i] :=: A[j]; endif;                  B
            if A[i] > vmax then maxi := i; vmax := A[maxi]; endif;
            if A[j] < vmin then minj := j; vmin := A[mini]; endif;
            loop: i := i+1; while A[i] < vmax repeat;                  C'
            loop: j := j-1; while A[j] > vmin repeat;                  C-C'
        while i < j:
        repeat;
        if i = j then i := i+1; j := j-1; endif;
        A[j] :=: A[maxi];
        A[i] :=: A[minj];
        quicksort (ℓ , j-1);
        quicksort (i+1 , r);
    endif;
```

During the inner loop of this program, we always have $A[k] \leq vmax$ for $k < i$ and $vmin \leq A[k]$ for $k > j$, with the interval $[vmax, vmin]$ containing the partition. The operation of the program on our set of fifteen keys is shown in Example 6.1. The interval containing the partition shrinks from [01,44] to [10,26] to [24,26]. After the pointers have met, at 07 and 96, it is known that all of the elements to the right of the 07 are ≥ 26 and all of the elements to the left of the 96 are ≤ 24. This means that the final position of both the

Example 6.1


partitioning:
44 26 95 04 08 88 96 34 07 35 99 24 68 10 01
01
44
26
10
01 10
26 44
95
68
24
01 10 24
95 68 26 44
04
08
88
99
35
07
01 10 24 04 08 07
88 35 99 95 68 26 44
96
34
96
07
01 10 24 04 08 07 96 34 88 35 99 95 68 26 44
01 10 07 04 08 24 26 34 88 35 99 95 68 96 44
sorting the file:
44 26 95 04 08 88 96 34 07 35 99 24 68 10 01
01 10 07 04 08 24 26 34 88 35 99 95 68 96 44
01 04 07 08 10
34 35 44 99 95 68 96 88
88 68 95 96 99
68 88

24 and the 26 are known, and they both are exchanged into position. If some element of the array is already in position, the pointers might meet at that element, and three elements are then put into position by partitioning. This occurs, for example, in the second partitioning stage in Example 6.1. The nodes of the partitioning tree now may contain more than one partitioning element -- for Example 6.1 we have



As we have done before, we will restrict our attention to this recursive program, while recognizing that all of the improvements that we saw between Program 2.2 and Program 2.4 can be applied to make Program 6.1 much more efficient.

In order to be able to analyze the performance of Program 6.1 we first need to determine the exact probability distribution for the position of the partition in the file at each stage. As always, we assume that the numbers being sorted are the integers $\{1,2,\ldots,N\}$. For example, the position of the partition for all of the permutations of four elements is shown in Example 6.2. By simply counting in this table, we can determine the frequencies of all of the "two-partitions" and "three-partitions". In general, let

$$a_{Ns} = \left\{\begin{matrix}\text{number of permutations of } \{1,2,\ldots,N\} \text{ for which the} \\ \text{partition falls on } s \text{ and } s+1\end{matrix}\right\}$$

and

$$b_{Ns} = \left\{\begin{matrix}\text{number of permutations of } \{1,2,\ldots,N\} \text{ for which the} \\ \text{partition falls on } s \text{ , } s+1 \text{ , and } s+2\end{matrix}\right\} \quad .$$

From our algorithm, we can begin to determine some basic properties of these numbers, and then we can try to derive formulas for them. First, since we have one of these kinds of partition for every permutation of $\{1,2,\ldots,N\}$ we obviously have

$$\sum_s a_{Ns} + \sum_s b_{Ns} = N! \quad .$$

Also, there is a symmetry property: if the partition falls on s and $s+1$ for a permutation $c_1 c_2 \ldots c_N$ of $\{1,2,\ldots,N\}$ then it falls on $N-s$ and $N-s+1$ for the permutation $d_1 d_2 \ldots d_N$, where $d_i = N+1 - c_{N+1-i}$. For example, the permutation

$$10\ \ 7\ \ 13\ \ 2\ \ 4\ \ 12\ \ 14\ \ 8\ \ 3\ \ 9\ \ 15\ \ 6\ \ 11\ \ 5\ \ 1$$

is partitioned to

$$1\ \ 5\ \ 3\ \ 2\ \ 4\ \boxed{6\ \ 7}\ 8\ \ 12\ \ 9\ \ 15\ \ 13\ \ 11\ \ 14\ \ 10$$

(this corresponds to our fifteen sample keys and Example 6.1), while the "symmetrically corresponding" permutation

$$15\quad 11\quad 5\quad 10\quad 1\quad 7\quad 13\quad 8\quad 2\quad 4\quad 12\quad 14\quad 3\quad 9\quad 6$$

is partitioned to

$$6\quad 2\quad 5\quad 3\quad 1\quad 7\quad 4\quad 8\quad \boxed{9\quad 10}\quad 12\quad 14\quad 13\quad 11\quad 15$$

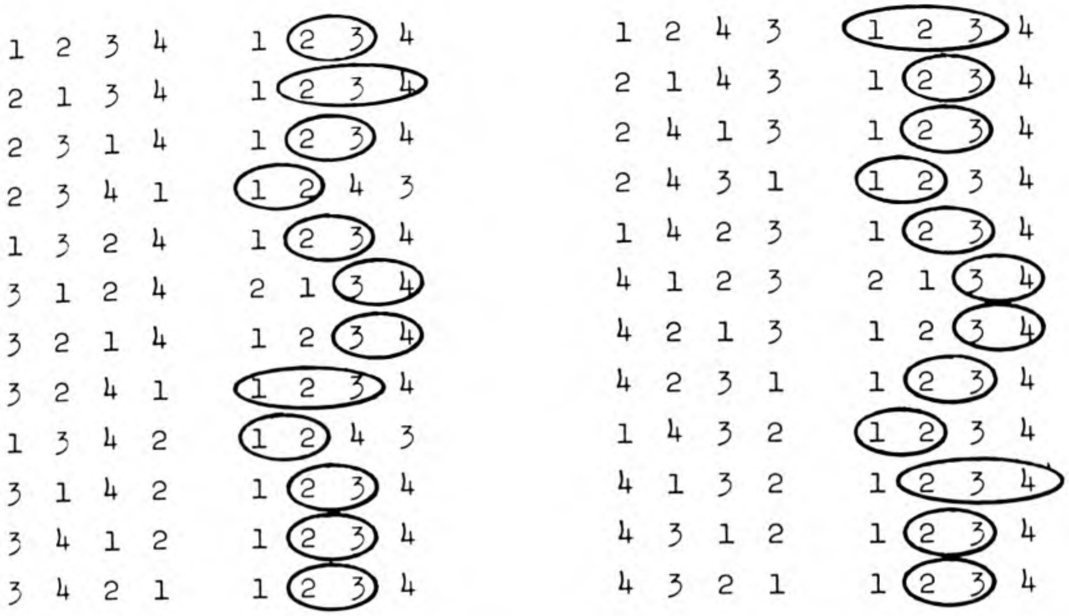
which "symmetrically corresponds" to the result of partitioning the original. This symmetry means that

$$a_{Ns} = a_{N(N-s)} \quad ,$$

and, similarly, we have

$$b_{Ns} = b_{N(N-1-s)} \quad .$$

Example 6.2



	s		
	1	2	3
a_{4s}	4	12	4
b_{4s}	2	2	

The tables below, obtained empirically, show the values of a_{Ns} and b_{Ns} for small values of N and s :

a_{Ns}

N \ s	1	2	3	4	5	6 ...
2	2					
3	2	2				
4	4	12	4			
5	12	38	38	12		
6	48	150	224	150	48	
7	240	732	1238	1238	732	240
⋮						

b_{Ns}

N \ s	1	2	3	4	5 ...
3	2				
4	2	2			
5	4	12	4		
6	12	38	38	12	
7	48	150	224	150	48
⋮					

As we expect, these tables are symmetric, and they indicate that the partition tends to be near the center. However, the most obvious feature of these tables is the relation between the two frequencies:

$$a_{Ns} = b_{(N+1)s} \quad .$$

It is not too difficult to prove that this relation holds, now that we have guessed it. Suppose that we form all permutations of $\{1,2,\ldots,N,N+1\}$ from permutations of $\{1,2,\ldots,N\}$ using the same kind of correspondence that we used in Chapter 1: with each permutation of N elements we associate the N+1 permutations of N+1 elements defined by:
(i) incrementing each item $\geq s+1$ by 1 ; (ii) inserting $(s+1)$ into each of the N+1 possible positions between elements. Then the partition will fall on s , s+1 , and s+2 in one of these permutations of N+1 elements, if and only if the partition falls on s and s+1 in the corresponding permutation of N elements.

We can therefore concentrate on finding an expression for a_{Ns} , and because of symmetry we can assume $s \leq N-s$. To begin, we might notice that the only way that the partition can fall on the elements 1 and 2 is if they are initially at the left and right ends of the array. This is easy to verify by examining the algorithm. First, the variable vmax cannot decrease during the execution of the program, so that if it is not initially 1 , it can never be. If vmax is initially 1 and vmin is not initially 2 , then the pointers must stop at least once (since there are elements between vmax and vmin within the array) with A[i] and A[j] both > 1 , which must result in vmax being increased. Therefore, the initial values of vmax and vmin must be 1 and 2 if the partition is to fall on 1 and 2 . There are exactly $2(N-2)!$ permutations of $\{1,2,\ldots,N\}$ with 1 and 2 at the ends, so we have shown that

$$a_{N1} = 2(N-2)! \quad .$$

In fact, $a_{Ns} \geq 2(N-2)!$ for all s , since the $2(N-2)!$ permutations with s and $s+1$ at the ends will result in the partition being at s and $s+1$. However this type of argument does not generalize nicely to yield an exact expression for a_{Ns} for all s , so that we will have to resort to a more complicated argument.

First we consider the left subfile after partitioning but before the s key has been exchanged into position. This is a permutation of the elements $\{1,2,\ldots,s\}$. Some of these elements were brought in by exchanges during the partitioning process. By knowing these and examining the left subfile after partitioning, we can determine how many times the pointers must have stopped during partitioning. For example, if we have the left subfile

$$\textcircled{3}\quad \boxed{1}\quad \textcircled{5}\quad \boxed{\textcircled{8}}\quad 2\quad \boxed{6}\quad 7\quad \textcircled{9}\quad 4\ ,$$

where the ○'s indicate left-to-right maxima and □'s mark elements brought in by exchange, then we can tell that the pointers must have stopped 6 times during partitioning: 3 for the exchanges and 3 for the left-to-right maxima not brought in by exchanges. Now, the number of left-to-right maxima in a permutation of $\{1,2,\ldots,N\}$ is $N-D$, where D is the quantity we studied in Chapter 1. From the analysis in Chapter 1, we know that the number of permutations of $\{1,2,\ldots,s\}$ with exactly $k-x+j$ left-to-right maxima is $\left[{s \atop k-x+j}\right]$. We can therefore generalize our observations to say that the number of possible left subfiles after a partition during which the pointers stopped k times and x exchanges were made (not including the first element) is

$$\sum_j \left[{s \atop k-x+j}\right]\binom{k-x+j-1}{j}\binom{s-k+x-j}{x-j} \qquad .$$

Here, the index of summation j is intended to count the number of left-to-right maxima brought in by exchanges, i.e., the number of elements both ○'d and □'d in the example above. This expression can be verified by considering the number of ways of distributing the x □'s among the s elements in the left subfile. The number of left-to-right maxima (the number of ○'s) is $k-x+j$: j □'s can be distributed among the $k-x+j-1$ ○'s (not counting the first) in $\binom{k-x+j-1}{j}$ ways, and the remaining $x-j$ □'s can be distributed among the remaining $s-k+x-j$ elements in $\binom{s-k+x-j}{x-j}$ ways. We do not count exchanges involving the first position because it can only be exchanged with the last element in the right subfile -- we will double our final answer to account for this.

Similarly, for the right subfile we have the expression

$$\sum_i \begin{bmatrix} n-s \\ k-x+i \end{bmatrix} \binom{k-x+i-1}{j} \binom{n-s-k+x-i}{x-i}$$

for the number of possible right subfiles after a partition during which the right pointer stopped k times and x exchanges were made. Now, an obvious feature of the algorithm is that not only must the number of exchanges counted be the same for the left and right subfiles, but also the number of times the pointers stop must be the same. If this were the only requirement, then we could make some progress by multiplying the above expressions together and summing over x and k. Unfortunately, however, we have the additional (obvious) requirement that if the one pointer stops for an exchange, the other must also. We have only succeeded in showing that

$$a_{Ns} \le 2 \sum_x \sum_k \Biggl(\sum_j \begin{bmatrix} s \\ k-x+j \end{bmatrix} \binom{k-x+j-1}{j} \binom{s-k+x-j}{x-j} \Biggr) \Biggl(\sum_i \begin{bmatrix} n-s \\ k-x+i \end{bmatrix} \binom{k-x+i-1}{i} \binom{N-s-k+x-i}{k-i}$$

$$= 2 \sum_x \sum_k \sum_i \sum_j \begin{bmatrix} s \\ k-x+j \end{bmatrix} \binom{k-x+j-1}{j} \binom{s-k+x-j}{x-j} \begin{bmatrix} N-s \\ k-x+i \end{bmatrix} \binom{k-x+i-1}{i} \binom{N-s-k+x-i}{x-i}$$

$$= 2 \sum_i \sum_j \sum_k \sum_x \begin{bmatrix} s \\ j \end{bmatrix} \binom{j-1}{j-k+x} \binom{s-j}{k-j} \begin{bmatrix} N-s \\ i \end{bmatrix} \binom{i-1}{i-k+x} \binom{N-s-i}{k-i} ,$$

where the last transformation was to interchange the order of summation and to replace j by $j-k+x$ and i by $i-k+x$. Now the sums on k and x both reduce to Vandermonde's convolution (see Eq. (21) in Appendix B):

$$\sum_x \binom{j-1}{j-k+x} \binom{i-1}{i-k+x} = \sum_x \binom{j-1}{j+x} \binom{i-1}{i+x} = \binom{i+j-2}{i-1} ,$$

and

$$\sum_k \binom{s-j}{k-j}\binom{N-s-i}{k-i} = \binom{N-j-i}{s-i} \quad .$$

Substituting these, we get

$$a_{Ns} \le 2 \sum_i \sum_j \begin{bmatrix} s \\ j \end{bmatrix} \begin{bmatrix} N-s \\ i \end{bmatrix} \binom{i+j-2}{i-1} \binom{N-j-i}{s-i} \quad .$$

This expression does not seem to simplify further, and it really doesn't offer much information for general N and s . Fortunately, however, equality holds for $s = 1$ and $s = 2$ (the pointers, after the initial stop at $i = 1$ and $j = N$, stop again only for possibly one exchange), and this will be sufficient for our purposes. For $s = 1$, we have a check on the result we already have derived:

$$a_{N1} = 2 \sum_i \sum_j \begin{bmatrix} 1 \\ j \end{bmatrix} \begin{bmatrix} N-1 \\ i \end{bmatrix} \binom{i+j-2}{i-1} \binom{N-j-i}{1-i}$$

$$= 2 \begin{bmatrix} N-1 \\ 1 \end{bmatrix} \quad ,$$

since all terms vanish except those for $i = 1$ and $j = 1$. But $\begin{bmatrix} N-1 \\ 1 \end{bmatrix} = (N-2)!$ by Eq. (29) in Appendix B, so this agrees with our earlier result. Similarly, for $s = 2$, the only non-zero terms are those for $i = 1, 2$ and $j = 1, 2$:

$$a_{N2} = 2\begin{bmatrix} 2 \\ 1 \end{bmatrix}\begin{bmatrix} N-2 \\ 1 \end{bmatrix}(N-2) + 2\begin{bmatrix} 2 \\ 1 \end{bmatrix}\begin{bmatrix} N-2 \\ 2 \end{bmatrix} + 2\begin{bmatrix} 2 \\ 2 \end{bmatrix}\begin{bmatrix} N-2 \\ 1 \end{bmatrix}(N-3) + 2\begin{bmatrix} 2 \\ 2 \end{bmatrix}\begin{bmatrix} N-2 \\ 2 \end{bmatrix} 2$$

$$= 2\left(\begin{bmatrix} N-2 \\ 1 \end{bmatrix}(N-2) + \begin{bmatrix} N-2 \\ 1 \end{bmatrix}(N-3) \right) + 6\begin{bmatrix} N-2 \\ 2 \end{bmatrix}$$

$$= 2((N-2)! + (N-3)(N-3)!) + 6(N-3)!\, H_{N-3} \quad ,$$

since $\binom{N-2}{2} = (N-3)!H_{N-3}$ by Eq. (30) in Appendix B. The complications involved in this expression and this derivation indicate that, if we were to get a formula for a_{Ns} for general s, it would involve a sum of Stirling numbers.

However, we need go no further, because the expressions for a_{N1} and a_{N2} indicate that this method cannot produce random subfiles after partitioning. Consider the number of permutations of $\{1,2,\ldots,N\}$ for which the right subfile turns out to be of size $N-3$. From our definitions, this is

$$a_{N2} + b_{N1}$$

or, substituting the expressions we have derived, the right subfile is of size $N-3$ with frequency

$$2((N-2)! + (N-3)(N-3)!) + 6(N-3)!\,H_{N-3} + 2(N-3)!$$

or

$$(N-3)!\,(4(N-2) + 6H_{N-3}) \quad .$$

Now, in order for the right subfile to be "random", each of the $(N-3)!$ permutations of its elements must be equally likely -- that is, each must appear with equal frequency when all permutations of $\{1,2,\ldots,N\}$ are considered. But this can only occur if the above expression is an integral multiple of $(N-3)!$, which is certainly false (for $N = 7$, we get $20 + 6H_4 = 65\ 1/2$). Therefore, the subfiles after partitioning for Program 6.1 cannot be random, and we cannot hope to complete an exact analysis of this program. On hindsight, we should not be too surprised by this result. Our standard proof for random subfiles says that the partitioning process is independent of the relative order of the elements which turn out to fall in the left (and right) subfiles.

For this method, the relative order of all of the elements affects the final position of the partition, so that the proof breaks down entirely.

It is unfortunate that we are unable to analyze this algorithm, but it would be disastrous indeed (from an analyst's point of view) if Program 6.1 outperformed all of our other methods. Fortunately, this is not the case -- simulation studies consistently show that Program 6.1 is less efficient than our other methods (such as Program 2.4) for essentially the same reason that Program 5.1 was inefficient. The partitioning element does tend to be closer to the center, and for this reason the number of "comparisons" (the average value of the quantity C) tends to be lower. However, we now know better than to count "comparisons" only. In fact, Program 6.1 has other comparisons within its inner loop -- those necessary to maintain the partitioning bounds vmax and vmin . Not only are the pointers stopped more often, but also there is more overhead (three extra comparisons) involved each time the pointers stop. Again we have a case of adding overhead to one part of the inner loop (the quantity B) to optimize other part of the inner loop (the quantity C), and again this strategy does not succeed.

Moreover, these extra comparisons succeed much more often than they fail -- the variables vmax and vmin are not changed very many times on the average. Also, each successive change is of less importance than the preceding. The idea is to get a partition which tends to be close to the center, and this is mostly accomplished the first few times the partitioning bounds are improved. Most of the extra work involved in Program 6.1 is required to get the two partitioning bounds to meet exactly, and this is not really necessary. An improvement to Program 6.1, therefore, might involve using the partitioning method given until vmax and vmin have been changed once or twice, then arbitrarily using one of the values as the partitioning value for a method like Partitioning Method 2.4. This would give, for large enough N , an improved choice of the partitioning element without affecting the "inner loop" of the program.

The idea of an adaptive partitioning method is attractive, and many serious attempts at improving the Quicksort algorithm result in the definition of such a method. As another example, we might consider a modification to Partitioning Method 2.1 which initially exchanges $A[\ell]$ and $A[r]$ if necessary to make $A[\ell] \leq A[r]$; then scans from the left to find an element $> A[\ell]$ and from the right to find an element $< A[r]$. If the left scan is shorter than the right, then $A[\ell]$ is used as the partitioning element, with Partitioning Method 2.1 used for the rest of the partitioning. If the right scan was shorter, then $A[r]$ is used as the partitioning element. The motivation behind this idea is that the method tends to use as the partitioning element the closer of $A[\ell]$ and $A[r]$ to the center, and the decision costs very little.

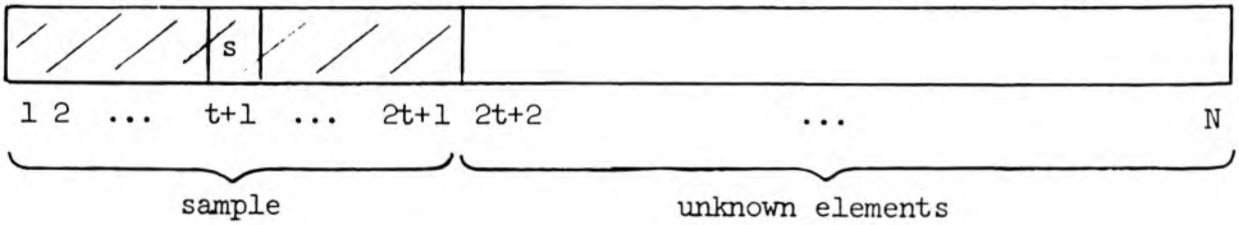
While these methods do perform much better than Program 6.1 and almost compete with Program 2.4, we will not consider them in any further detail for two reasons. First, they still do not produce random subfiles. It seems that, by their very nature, any of these adaptive methods cannot result in random subfiles after partitioning. As we have already remarked, they are not necessarily inefficient for this reason, but they certainly are difficult if not impossible to analyze. The second reason that we will not look further at these methods is that they involve deciding which should be the partitioning element on the basis of examining a relatively small number of elements in the file. This relates these to the method discussed in Chapter 8, where it is shown that, if the partitioning element is chosen on the basis of examining a fixed number of elements at every stage, then the best strategy is to choose the median of those elements. This indicates that the methods described above will not perform as well as the method that we will see in Chapter 8, and this is verified by simulation.

Unfortunately, then, it seems that we can always do better than these adaptive partitioning methods. We will look next at an idea which does eventually lead us to a genuine improvement in the Quicksort algorithm. This idea, which can be implemented in the two quite different ways discussed in Chapter 7 and Chapter 8, does lead to a choice of the partitioning element which tends to be close to the center of the file. Unlike Program 6.1, the extra overhead required to achieve this does not fall in the "inner loop", so that the savings achieved can be significant.

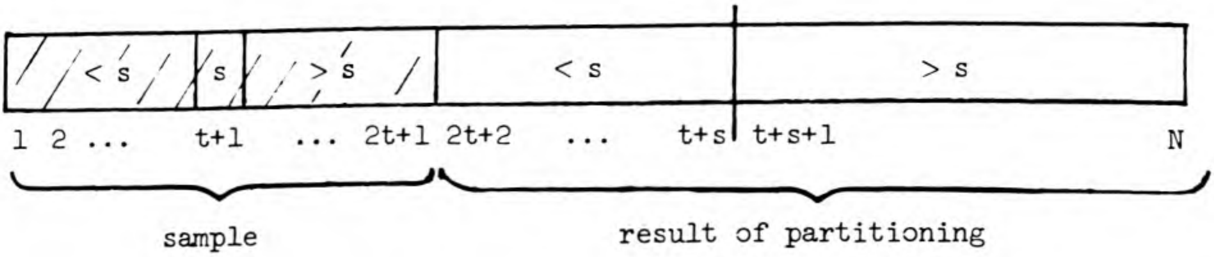
CHAPTER SEVEN

Our efforts towards finding a better strategy for choosing the partitioning element have been directed towards getting the partition as close as possible to the center of the file being partitioned. In trying to make a good choice of the partitioning element, then, we are simply trying to estimate the median element of the file. Essentially, this estimate is based on "sampling", or examining some of the elements in the array. Now, it is a well known principle from statistics that the best estimate of the median of an array, based on a sample from the array, is the median element of the sample. This leads us to a partitioning method called samplesort, which was introduced by W. D. Frazer and A. C. McKellar in 1970.

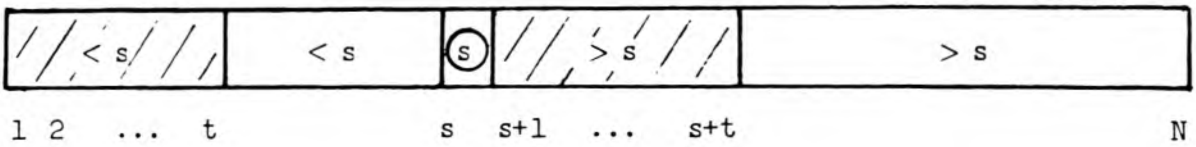
Samplesort is more complicated than most of the other methods that we have seen, because of the mechanics of choosing the sample and finding the median of the sample. One way of implementing the idea is to choose a small sample at each partitioning stage and then to use the median element of that sample for the partitioning element. This is the method that we will discuss in Chapter 8. Of course, the larger the sample, the more accurate will be our estimate of the median -- but it is not practical to take a large sample at every partitioning stage. Samplesort overcomes this difficulty by taking one very large sample at the first partitioning stage, sorting it, and then using elements from the sample as partitioning elements on succeeding partitioning stages. First, the median element is used. Then the element bounding the first and second quartiles is used for the left subfile, and the element bounding the third and fourth quartiles is used for the right subfile. The process is continued in this manner while there are sample elements to be used, and normal Quicksort is used after the sample is exhausted. Jt is most convenient to use an initial sample of size 2^k-1 for some integer k. We will see later that the effect is the same as taking a sample of size 2^k-1 at the first partitioning stage; then taking samples of size $2^{k-1}-1$ for partitioning at the second level of the partitioning tree; then samples of size $2^{k-2}-1$ at the third level, etc. To mechanize the partitioning process we will choose the first 2^k-1 elements of the array as the sample and ensure that the sample does not participate in the partitioning process. Specifically, after the sample has been chosen and sorted, the array has the following format before partitioning:



(For notational convenience we define $t \equiv 2^{k-1}-1$.) In other words, the (sorted) sample occupies array positions $A[1],A[2],\ldots,A[2t+1]$, and, by definition, the median element s is in position $A[t+1]$. If we now apply our normal partitioning procedure (Partitioning Method 2.4) to the end of the array $(A[2t+2], A[2t+3], \ldots, A[N])$ using s as the partitioning value, we get:



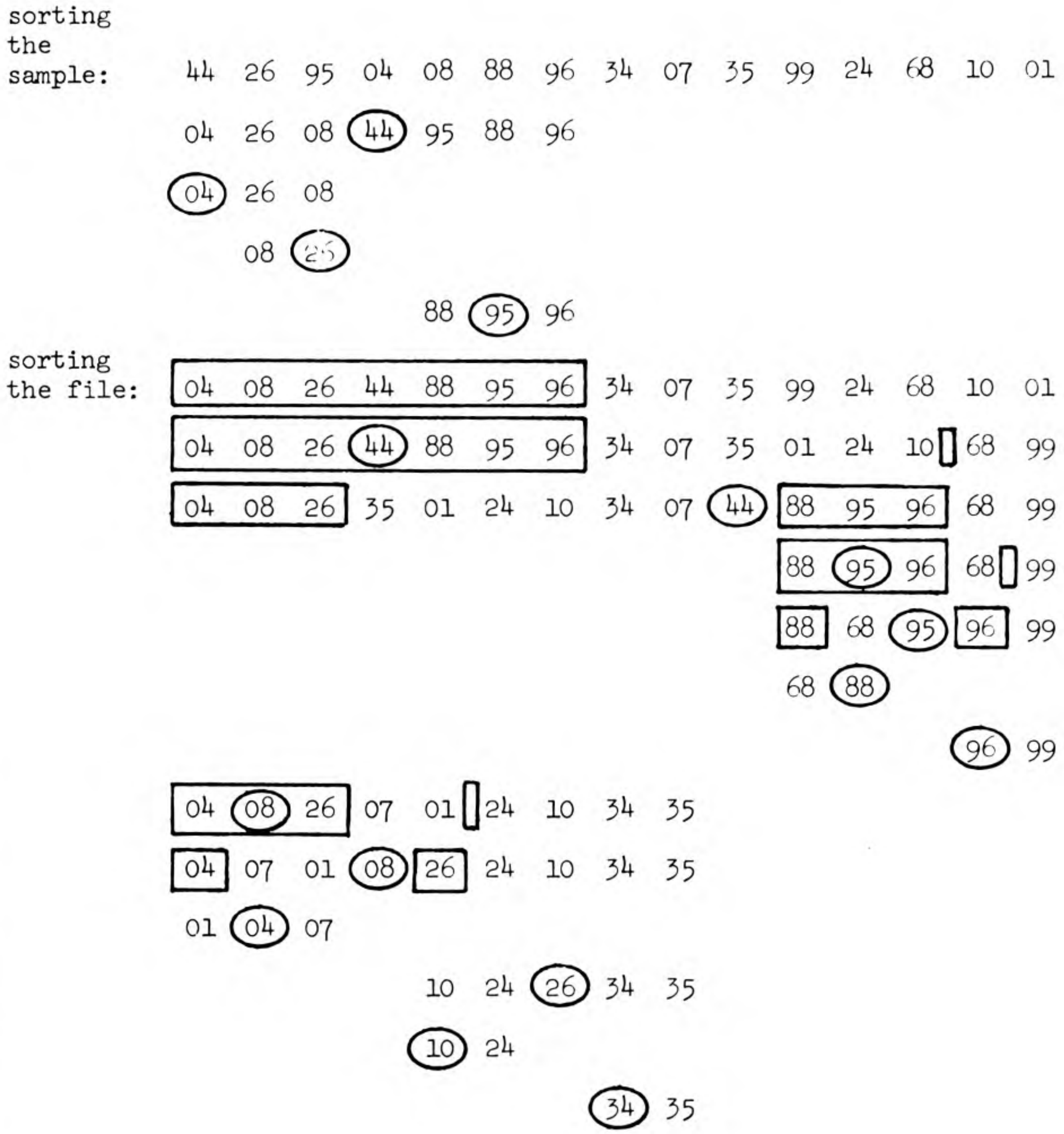
Now we can perform t+1 exchanges to get s and the sample elements > s into position, leaving two subfiles, each with t elements of the sample at the left end:



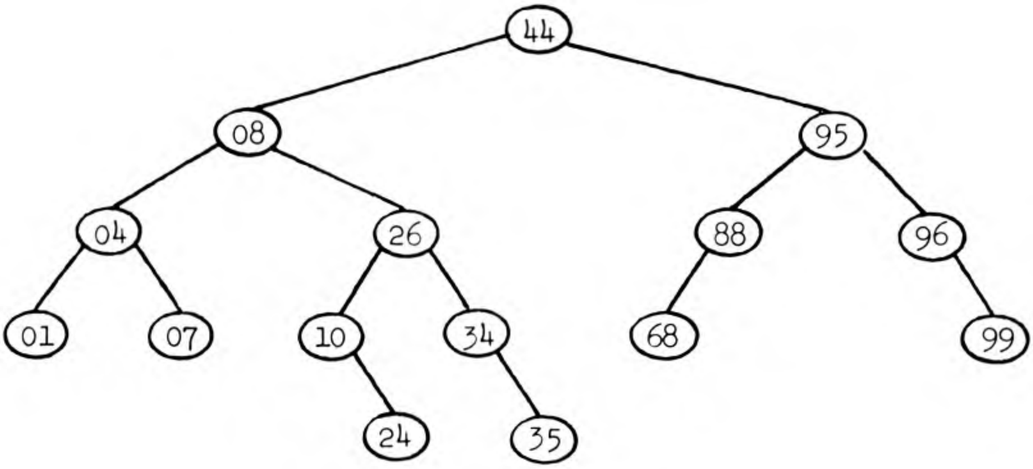
If the right half of the sample was kept sorted when it was moved, then the two subfiles now have the same format as the original file, and the procedure can be applied recursively. When the process has degenerated to "samples" of size 1 (t = 0) , then this is Quicksort as usual.

Example 7.1 shows how our fifteen keys are sorted using this method, with a sample of size 7 (k = 3) . The first seven keys of the file are sorted, using Quicksort (for example, Program 2.4). Then the file is partitioned on the median element of the sample, the 44 . After partitioning, the 44 and the right half of the sample are moved to the right subfile, leaving the 04 , the 08 , and the 26 in the left subfile; and the 88 , the 95 , and the 96 in the right subfile. The

<u>Example 7.1</u> (k = 3)



median of these three, the 95 , is used in partitioning the right subfile, and the 08 is used in partitioning the left subfile. Continuing in this manner, the sample elements are eventually exhausted as partitioning elements, and the procedure reverts to normal Quicksort, partitioning on the first element in each subfile. The partitioning tree for Example 7.1 is



which is nearly optimal. From this example, we can clearly see the motivation for the samplesort idea. The first k levels of the partitioning tree are forced to be complete -- they contain all of the elements of the sample. The most costly kind of imbalance in the partitioning tree, imbalance at the highest levels, is avoided.

The implementation of this idea turns out to be surprisingly simple -- it is little more than a generalization of Program 2.4, with only a few hidden subtleties:

Program 7.1

```
    procedure samplesort (integer value ℓ,r,t);
        if r-ℓ > M then
            i := ℓ+2×t; j := r+1; v := A[ℓ+t];
            loop:
                loop: i := i+1; while A[i] < v repeat;
                loop: j := j-1; while A[j] > v repeat;
            while i < j:
                A[i] :=: A[j];
            repeat;
            i := t;
          * loop: A[ℓ+t+i] :=: A[j-t+i]; while i > 0:  i := i-1; repeat;
            samplesort (ℓ , j-1 , t ÷ 2);
            samplesort (j+1 , r , t ÷ 2);
        endif;
```

In order to make this program perform as described above, it is necessary to first choose and sort the sample, so that the calling sequence

samplesort (1 , 2^k-1 , 0);

samplesort (1 , N , 2^{k-1}-1);

insertionsort (1,N);

should be used to sort A[1],A[2],...,A[N] with a sample of size 2^k-1 . (In a practical implementation, it may be desirable to attempt to avoid the worst case by scrambling the file as discussed in Chapter 5.) Again, this program can be improved by removing the recursion and sorting the smaller subfile first.

There are two subtle features to be noted about Program 7.1. First, the variable t always carries the size of the sample to be used in the subfiles, and it is always a number of the form 2^k-1 for some integer k . At each stage, it is changed to $t \div 2$ which is the the same as $\frac{t-1}{2}$ or $2^{k-1}-1$ for $t > 0$, but which remains 0 for $t = 0$. Program 7.1 with $t = 0$ is exactly the same as Program 2.2. A second point to notice in Program 7.1 is the operation of the loop which splits the sample (which is marked with an *) when there are less than $2t$ elements $< s$ in the file. This occurs, for example, when the first right subfile in Example 7.1 is sorted (at the seventh partitioning stage). It might seem more natural to have the variable i in this loop run from 0 up to t rather than from t down to 0 . If we were to do so in this case when there are more elements $< s$ in the sample than in the rest of the file, then the sample would not remain in order since some of its elements would be involved in two exchanges. By going in the other direction, we have some of the elements not in the sample involved in two exchanges. (In the seventh partitioning stage of Example 7.1, the 68 is exchanged with both the 95 and the 96 .) This doesn't otherwise affect the operation of the program, since these elements are randomly ordered anyway. (Program 7.1 could be made slightly more efficient by eliminating these extra exchanges, but this case occurs so infrequently that it is not worthwhile to complicate the program in this way.)

We now have the freedom of choosing two parameters to make this program run efficiently: the sample size, 2^k-1 ; and the cutoff for small subfiles, M . The optimal sample size is of most interest, since, as we will see, it affects the leading term of the average running time. To simplify the analysis, we will take $M = 0$, as we have done before. We will now attack the problem of finding the best value for k , starting with the derivation of an exact expression for the average number of comparisons used to sort N elements (as a function of k). It is often convenient to work with the sample size as $t = 2^{k-1}-1$ as above, so that the initial sample is of size $2t+1$. We will use the t and k notations interchangeably, without explicitly denoting their dependence.

To begin, we assume as always that the file being sorted is a random permutation of $\{1,2,\ldots,N\}$, and we see from Chapter 3 that the average number of comparisons needed to sort the sample is

$$(4t+4)(H_{2t+2}-1) \quad .$$

Next we set up a recurrence for the number of comparisons needed by samplesort proper by conditioning on the first partitioning stage, as usual. Now, however, there are two variables in our recurrence. Let us define $C_{N(2t+1)}$ to be the average number of comparisons required for N elements, given that the first $(2t+1)$ are sorted and used as a sample in Program 7.1. Again, from Chapter 3, we have

$$C_{N0} = 2(N+1)(H_{N+1}-1) \quad .$$

Now, for general t, the first partitioning stage takes $N-2t+1$ comparisons (by an elementary argument as in Chapter 2), and the subfiles each use samples of size t (by definition), so that we have the recurrence

$$C_{N(2t+1)} = N-2t+1 + \sum_{t+1 \le s \le N-t} \Pr\{s \text{ is the partitioning element}\}(C_{(s-1)t}+C_{(N-s)t}) \quad .$$

Since there are t elements $< s$ and t elements $> s$ in the sample, clearly the partitioning element s cannot be $< t+1$ or $> N-t$. In fact, the probability that any particular value s is used as the partitioning element is the probability that t elements $< s$ and t elements $> s$ were included in the sample together with s . Specifically,

$$\Pr\{s \text{ is the partitioning element}\} = \frac{\binom{s-1}{t}\binom{N-s}{t}}{\binom{N}{2t+1}} \quad .$$

Substituting this into our recurrence, we get

$$C_{N(2t+1)} = (N-2t+1) + \sum_{t+1 \le s \le N-t} \frac{\binom{s-1}{t}\binom{N-s}{t}}{\binom{N}{2t+1}} (C_{(s-1)t} + C_{(N-s)t}) \; .$$

By writing this recurrence, we are implicitly assuming that it makes no difference that we use the same sample elements for the subfiles, as opposed to choosing new samples of size t for each of the subfiles. But we are justified in doing so because the probability that any particular group of $2t+1$ elements used is $\frac{1}{\binom{N}{2t+1}}$ for both methods. This is obvious for the strategy that we're using. For the other method the argument goes as follows: Let s be the median of the $2t+1$ elements we are interested in. Then s is used as the partitioning element for the first stage with probability

$\frac{\binom{s-1}{t}\binom{N-s}{t}}{\binom{N}{2t+1}}$; the t elements $< s$ are chosen for the sample for the left subfile with probability $\frac{1}{\binom{s-1}{t}}$; and the t elements $> s$ are chosen for the sample for the right subfile with probability $\frac{1}{\binom{N-s}{t}}$. Therefore, the particular group of $2t+1$ elements is used with probability

$$\frac{\binom{s-1}{t}\binom{N-s}{t}}{\binom{N}{2t+1}} \frac{1}{\binom{s-1}{t}} \frac{1}{\binom{N-s}{t}} = \frac{1}{\binom{N}{2t+1}} .$$

Therefore we may properly write $C_{(s-1)t}$ and $C_{(N-s)t}$ for the average number of comparisons used for the subfiles.

As usual, since the probability distribution for s is symmetric, the two sums in our recurrence are the same (change s to $N-s+1$ in the second). After multiplying by $\binom{N}{2t+1}$ we are left with

$$\binom{N}{2t+1} C_{N(2t+1)} = (N-2t+1)\binom{N}{2t+1} + 2 \sum_{t+1 \le s \le N-t} \binom{s-1}{t}\binom{N-s}{t} C_{(s-1)t} .$$

It is convenient to write $N-2t+1 = (N-2t-1)+2$ since $(N-2t-1)\binom{N}{2t+1} = (2t+2)\binom{N}{2t+2}$ by Eq. (18) in Appendix B. From this point on, we are actually solving two recurrences in parallel, one for $N-2t-1$, one for the constant 2 . We will examine the implications of this below. Our recurrence now is

$$C_{2t+1}(z) = (2t+2)\frac{z^{2t+2}}{(1-z)^{2t+3}} + 2\frac{z^{2t+1}}{(1-z)^{2t+2}}$$

$$+ \sum_{s\geq t+1}\ \sum_{N\geq s+t} \binom{N-s}{t}\binom{s-1}{t} C_{(s-1)t}\, z^N$$

$$= (2t+2)\frac{z^{2t+2}}{(1-z)^{2t+3}} + 2\frac{z^{2t+1}}{(1-z)^{2t+2}}$$

$$+ \sum_{s\geq t}\ \sum_{N\geq 0} \binom{N+t}{t}\binom{s}{t} C_{st}\, z^{N+s+t+1}$$

$$= (2t+2)\frac{z^{2t+2}}{(1-z)^{2t+3}} + 2\frac{z^{2t+1}}{(1-z)^{2t+2}} + 2\,C_t(z)\frac{z^{t+1}}{(1-z)^{t+1}} \quad .$$

We now have a recurrence on t which we can solve to get an equation for the generating function $C_t(z)$. First, we multiply both sides of this equation by the "summation factor" $\frac{(1-z)^{2t+1}}{z^{2t+1}}$, which gives

$$\frac{(1-z)^{2t+1}}{z^{2t+1}} C_{2t+1}(z) = (2t+2)\frac{z}{(1-z)^2} + \frac{2}{1-z} + 2\frac{(1-z)^t}{z^t} C_t(z) \quad .$$

This recurrence is almost, but not quite, ready to telescope to a sum. To make it do so, we change notation back to $k = \lg(2t+2)$ (since $t = 2^{k-1}-1$), and define

$$F_k(z) = \frac{1}{2^k}\frac{(1-z)^{2^k-1}}{z^{2^k-1}} C_{2^k-1}(z) \qquad ,$$

which transforms the recurrence to

$$2^k F_k(z) = 2^k \frac{z}{(1-z)^2} + \frac{2}{1-z} + 2^k F_{k-1}(z) \quad .$$

This telescopes, after dividing by 2^k, to the sum

$$F_k(z) = F_0(z) + \sum_{1 \le j \le k} \left(\frac{z}{(1-z)^2} + \frac{2}{2^j(1-z)} \right)$$

$$= F_0(z) + k \frac{z}{(1-z)^2} + \frac{2}{1-z} \left(1 - \frac{1}{2^k} \right) \quad .$$

By changing notation once again, back to $t = 2^{k-1}-1$ and

$F_k(z) = \frac{1}{2t+2} \frac{(1-z)^{2t+1}}{z^{2t+1}} C_{2t+1}(z)$, we can convert this equation to

$$\frac{1}{2t+2} \frac{(1-z)^{2t+1}}{z^{2t+1}} C_{2t+1}(z) = C_0(z) + \lg(2t+2) \frac{z}{(1-z)^2} + \frac{2}{1-z}\left(1 - \frac{1}{2t+2}\right),$$

or

$$C_{2t+1}(z) = \frac{z^{2t+1}}{(1-z)^{2t+1}} \left((2t+2)C_0(z) + (2t+2)\lg(2t+2) \frac{z}{(1-z)^2} + \frac{2}{1-z}(2t+1) \right) .$$

Now we convert back to power series:

$$C_{2t+1}(z) = z \sum_{i \ge 0} \binom{i}{2t} z^i \sum_{j \ge 0} \Big((2t+2)c_{j0} + (2t+2)\lg(2t+2)j + 2(2t+1) \Big) z^j .$$

This equation follows from the fact that $C_0(z) = \sum_{j \geq 0} C_{j0} z^j$,

$\frac{z}{(1-z)^2} = \sum_{j \geq 0} j\, z^j$, and $\frac{1}{1-z} = \sum_{j \geq 0} z^j$ (see Eqs. (35) and (36) in Appendix B). This is now a convolution of two sums which is easily transformed to

$$C_{2t+1}(z) = z \sum_{N \geq 0} z^N \sum_{0 \leq j \leq N} \Big((2t+2)C_{j0} + (2t+2)\lg(2t+2)j + 2(2t+1) \Big) \binom{N-j}{2t}.$$

Since $C_{2t+1}(z) = \sum_{N \geq 0} \binom{N}{2t+1} C_{N(2t+1)}\, z^N$ by definition, we can set coefficients of z^N equal in this equation to get the solution:

$$\binom{N}{2t+1} C_{N(2t+1)} = \sum_{0 \leq j \leq N-1} \Big((2t+2)C_{j0} + (2t+2)\lg(2t+2)j + 2(2t+1) \Big) \binom{N-j-1}{2t}$$

$$= (2t+2) \sum_{0 \leq j \leq N-1} 2(j+1)(H_{j+1} - 1) \binom{N-j-1}{2t}$$

$$+ (2t+2)\lg(2t+2) \sum_{0 \leq j \leq N-1} j \binom{N-j-1}{2t}$$

$$+ 2(2t+1) \sum_{0 \leq j \leq N-1} \binom{N-j-1}{2t} .$$

The last two sums are easily evaluated as convolutions of binomial coefficients, but the first is more difficult, and is left for Appendix B. It follows from Eq. (45) in Appendix B that

$$\sum_{0 \le j \le N-1} (j+1)(H_{j+1} - 1)\binom{N-j-1}{2t} = \binom{N+1}{2t+2}(H_{N+1} - H_{2t+2}) ,$$

which gives us our answer

$$\binom{N}{2t+1} C_{N(2t+1)} = (4t+4)\binom{N+1}{2t+2}(H_{N+1} - H_{2t+2})$$

$$+ (2t+2)\lg(2t+2)\binom{N}{2t+2} + 2(2t+1)\binom{N}{2t+1} ,$$

or

$$C_{N(2t+1)} = 2(N+1)(H_{N+1} - H_{2t+2}) + (N-2t-1)\lg(2t+2) + 2(2t+1) .$$

To this must be added the $(4t+4)(H_{2t+2} - 1)$ comparisons that were required to sort the sample. Finally, then, let us change notation back to $k = \lg(2t+2)$ and define C^T_{Nk} to be the total number of comparisons used to sort a permutation of $\{1,2,\ldots,N\}$ using samplesort, when an initial sample of size 2^k-1 is used. Then, substituting into the equations we have derived, we get, after some trivial algebraic simplifications:

$$C^T_{Nk} = 2(N+1)H_N + (N+1-2^k)(k-2H_{2^k}) .$$

It is somewhat reassuring to find such a simple expression for this quantity.

As we noted during the derivation, we actually solved two recurrences in developing this solution. We have

$$C_{N(2t+1)} = C'_{N(2t+1)} + 2\, A_{N(2t+1)} ,$$

where $C'_{N(2t+1)}$ is defined by the recurrence

$$\binom{N}{2t+1}C'_{N(2t+1)} = (N-2t-1)\binom{N}{2t+1} + 2\sum_{t+1\le s\le N-t}\binom{s-1}{t}\binom{N-s}{t}C'_{(s-1)t} \ ,$$

and $A_{N(2t+1)}$ is defined by the recurrence

$$\binom{N}{2t+1}A_{N(2t+1)} = \binom{N}{2t+1} + 2\sum_{t+1\le s\le N-t}\binom{s-1}{t}\binom{N-s}{t}A_{(s-1)t} \ ,$$

which has the solution

$$A_{N(2t+1)} = N \quad .$$

There is an immediate interpretation of this decomposition from our algorithm. The quantity $A_{N(2t+1)}$ is simply the number of partitioning stages, and the $N-2t+1$ comparisons used during each partitioning stage consist of $N-2t-1$ absolutely necessary comparisons plus 2 redundant comparisons which we included to eliminate the overhead of testing for when the pointers cross. It is interesting that the redundant comparisons manifest themselves in the solution in this way.

If we wish to find the value of k for which C^T_{Nk} is minimized, we can take the difference with respect to k, ΔC^T_{Nk}, and find where it changes sign. This is analogous to finding the minimum of a continuous function by setting the derivative to zero. To calculate ΔC^T_{Nk}, we notice that $\Delta f(k)g(k) = f(k+1)\Delta g(k) + g(k)\Delta f(k)$ for any functions f and g; and that $\Delta 2^k = 2^k$. We then get

$$\Delta C^T_{Nk} = (N+1-2^{k+1})(1 - 2H_{2^{k+1}} + 2H_{2^k}) - 2^k(k - 2H_{2^k}) \quad .$$

The desired "best" value of k is the smallest k for which $\Delta C^{T}_{Nk} > 0$; by setting the above equation to zero, we find that this is the smallest k for which

$$N+1 \;<\; 2^{k+1} + \frac{2^{k}(k - 2H_{2^k})}{1 - 2H_{2^{k+1}} + 2H_{2^k}} \qquad .$$

From this formula, it is easy to compute the value of k which minimizes the number of comparisons for practical values of N . These are given in the table below: for example, if $332 \le N < 717$, then $k = 6$, or a sample size of 63 , minimizes the number of comparisons.

N	332		717		1550		3341		7180		15367		32765
k		6		7		8		9		10		11	

We can also use this formula to derive an asymptotic expression describing the value of k which minimizes the number of comparisons. Starting with the asymptotic formula

$$H_{2^k} = k \ln 2 + \gamma + \frac{1}{2^{k-1}} + O\left(\frac{1}{(2^k)^2}\right)$$

we find that

$$2^{k}(k - 2H_{2^k}) = (1 - 2 \ln 2)\, k\, 2^{k} - 2^{k+1}\gamma - 1 + O\left(\frac{1}{2^k}\right)$$

and

$$\frac{1}{1 - 2H_{2^{k+1}} + 2H_{2^k}} = \frac{1}{1 - 2 \ln 2} - \frac{1}{(1 - 2 \ln 2)^2}\;\frac{1}{2^{k+1}} + O\left(\frac{1}{(2^k)^2}\right)$$

so that we are looking for the smallest k for which

$$N+1 < 2^k\left(k + \frac{2\gamma}{2\ln 2 - 1} + 2\right) + \frac{1}{2\ln 2 - 1}\left(\frac{k}{2} + 1 + \frac{\gamma}{2\ln 2 - 1}\right) + O\left(\frac{k}{2^k}\right) .$$

Now we wish to solve this equation for k . First, we note that the sample is taken from the file, so $2^k \le N$, which implies that $2^k = O(N)$ and $k = O(\lg N)$. Therefore, we have

$$N = 2^k\left(k + \frac{2\gamma}{2\ln 2 - 1} + 2 + O\left(\frac{\lg N}{N}\right)\right) ,$$

or, taking logarithms,

$$k = \lg N - \lg\left(k + \frac{2\gamma}{2\ln 2 - 1} + 2 + O\left(\frac{\lg N}{N}\right)\right) .$$

Now, we can iterate this formula once to give

$$\begin{aligned} k &= \lg N - \lg\left(\lg N - \lg\left(k + \frac{2\gamma}{2\ln 2 - 1} + 2 + O\left(\frac{\lg N}{N}\right)\right)\right) \\ &= \lg N - \lg\lg N + O\left(\frac{\lg\lg N}{\lg N}\right) \\ &= \lg\left(\frac{N}{\lg N}\right) + O\left(\frac{\lg\lg N}{\lg N}\right) . \end{aligned}$$

Therefore, k should be chosen so that 2^k is about $\frac{N}{\lg N}$. It takes some faith in asymptotic methods to have confidence in the utility of this result, but it does reasonably match the exact values computed above. Also, this derivation leads to a simple proof of one of the most significant features of samplesort.

We will find an asymptotic formula for the number of comparisons used to sort $\{1,2,\ldots,N\}$, given that the best sample size was used -- we will concentrate on finding the "leading term" (the term which dominates when N is very large). Now, since $2^k = O(N)$ and $k = O(\ln N)$ our exact formula for the total number of comparisons can be approximated by the asymptotic formula

$$C_{Nk}^{T} = 2(N+1)\ \ln N + (N+1-2^{k})k(1-2\ \ln 2) + O(N)$$

by substituting the asymptotic formula for the harmonic series that we used above. This expression can be simplified further to give

$$C_{Nk}^{T} = 2\ N\ \ln N + (1-2\ \ln 2)\,N\,k - (1-2\ \ln 2)\,k\,2^{k} + O(N) \quad .$$

But we know that $k\,2^{k} = O(N)$ and $k = \lg N + O(\lg \lg N)$, and substituting these, we get the result

$$C_{N}^{T} = 2\ N\ \ln N + (1-2\ \ln 2)\ N\ \lg N + O(N\ \lg \lg N)$$

$$= N\ \lg N + O(N\ \lg \lg N)$$

$$C_{N}^{T} = N\ \lg N + O(N\ \lg \lg N) \quad .$$

This is the theoretic minimum number of comparisons required by every sorting method. In other words, samplesort is asymptotically optimal with respect to comparisons. As N gets very large, the total average number of comparisons used by Program 7.1 approaches N lg N . This is an interesting theoretical result, but the converegence is very slow, and the overhead required to implement this algorithm limits its attractiveness in most practical situations.

From a practical standpoint, our interest is to pick a sample size which will minimize the total running time, not just the number of comparisons. We know that the other main contributor to the running time is the number of exchanges, so we shall now consider the analysis of this quantity. This analysis will be somewhat more complex than our analysis of the number of comparisons, for the same reasons that we found in Chapter 3. First, we need to find the average number of exchanges needed on the first partitioning stage, or

$$\sum_{t+1 \le s \le N-t} \Pr\left\{\begin{matrix}\text{s is the}\\ \text{partitioning element}\end{matrix}\right\}\left\{\begin{matrix}\text{average number of exchanges when}\\ \text{s is the partitioning element}\end{matrix}\right\} .$$

The probability that s is the partitioning element is $\binom{s-1}{t}\binom{N-s}{t}\Big/\binom{N}{2t+1}$, as above. To find the average number of exchanges, we notice that there are two kinds of exchanges. First, just as in Chapter 3, all of the keys among A[2t+2], A[2t+3], ..., A[t+s] which are $> s$ are involved in exchanges. The average number of such keys is

$$\sum_{0 \le j \le s-t-1} j \frac{\binom{N-t-s}{j}\binom{s-t-1}{s-t-1-j}}{\binom{N-2t-1}{s-t-1}}$$

$$= \frac{N-t-s}{\binom{N-2t-1}{s-t-1}} \sum_{0 \le j \le s-t-1} \binom{N-t-s-1}{j-1}\binom{s-t-1}{s-t-1-j}$$

$$= \frac{N-t-s}{\binom{N-2t-1}{s-t-1}} \binom{N-2t-2}{s-t-2}$$

$$= \frac{(s-t-1)(N-t-s)}{N-2t-1} \quad .$$

In addition, after partitioning is done, the right half of the sample is moved into place at the beginning of the right subfile -- this requires t exchanges. Therefore, the average number of exchanges used on the first partitioning stage is

$$\sum_{t+1 \le s \le N-t} \frac{\binom{s-1}{t}\binom{N-s}{t}}{\binom{N}{2t+1}} \left(t + \frac{(s-t-1)(N-t-s)}{(N-2t-1)} \right)$$

$$= t + \sum_{t+1 \le s \le N-t} \frac{(s-t-1)\binom{s-1}{t}(N-t-s)\binom{N-s}{t}}{(N-2t-1)\binom{N}{2t+1}} \quad .$$

We can expand in factorials or apply the identity $(r-k)\binom{r}{k} = (k+1)\binom{r}{k+1}$ three times to find that

$$t + \sum_{t+1 \le s \le N-t} \frac{(t+1)\binom{s-1}{t+1}(t+1)\binom{N-s}{t+1}}{(2t+2)\binom{N}{2t+2}} = t + \frac{t+1}{2} \frac{\binom{N}{2t+3}}{\binom{N}{2t+2}}$$

$$= t + \frac{t+1}{2} \frac{N-2t-2}{2t-3}$$

exchanges are made, on the average, during the first partitioning stage. Notice that this is asymptotically $1/4$ the number of comparisons. Therefore the recurrence describing the number of exchanges taken by Program 7.1 is

$$\binom{N}{2t+1} B_{N(2t+1)} = \left(t + \frac{t+1}{2} \frac{N-2t+2}{2t-3} \right) \binom{N}{2t+1}$$

$$+ 2 \sum_{t+1 \le s \le N-t} \binom{s-1}{t}\binom{N-s}{t} B_{(s-1)t} \quad .$$

This can be solved in exactly the same way as the equation above for comparisons, although the sums involved are more complicated. The solution is

$$
\begin{aligned}
B_{N(2t+1)} = {} & \tfrac{1}{2}\,(t+1)\ \lg(2t+2) - 2t - 1 \\
& + \tfrac{1}{3}\,(N+1)H_{N+1} - \tfrac{1}{3}\,(N+1)H_{2t+2} - \tfrac{1}{2}\,(N-1-2t) \\
& + \tfrac{1}{4}\,(N+1)\ \lg(2t+2) - \tfrac{1}{4}\,\epsilon_{\lg(2t+2)}(N+1) \quad ,
\end{aligned}
$$

where $\epsilon_k = \sum_{1 \le j \le k} \frac{1}{2^j+1}$, which converges to 1.26499 ... as k gets large. The first three terms $\left(\frac{1}{2}\,(t+1)\lg(2t+2) - 2t - 1\right)$ in this expression represent the contribution of the exchanges which move the sample sections, and the rest of the terms count the total exchanges used in partitioning. Adding in the $\frac{1}{3}\,(2t+2)H_{2t+2} - \frac{5}{6}\,(2t+2) + \frac{1}{2}$ exchanges required to sort the sample, and changing notation back to $k = \lg(2t+2)$, we get a final expression for the total average number of exchanges required to samplesort a random permutation of $\{1,2,\ldots,N\}$ using a sample of size 2^k-1 :

$$
B^T_{Nk} = \tfrac{1}{3}\,(N+1)H_N - \tfrac{1}{3}\,(N+1)H_{2^k} + \tfrac{1}{2}\,(N+1+2^k)\left(\tfrac{k}{2} - 1\right) + \tfrac{4}{3} - \tfrac{1}{4}\,\epsilon_k(N+1) \quad .
$$

Note that this agrees with our result for Chapter 3 when $k = 0$.

In principle, we could develop similar expressions for the average values of all of the quantities involved in the running time of Program 7.1, multiply them by the appropriate coefficients, and sum them to get an expression for the total running time, just as we did in Chapter 3. We will refrain from doing so here because we will be doing such a complete analysis in Chapter 8 for a similar algorithm, which is simpler and more efficient than samplesort. Also, the samplesort analysis is much more complex than indicated above if we allow for arbitrary M. (We chose $M = 0$.) The results include sums of M terms which cannot be reduced further.

However, we can get some indication of what the sample size should be from the formulas that we have derived so far. For convenience we assume that we have a slightly inefficient version of Program 7.1 which does a stack push at every partitioning stage, so that the total running time is

$$34A + 11B + 4C \quad ,$$

where A is the (total) number of partitioning stages, B is the number of exchanges, and C is the number of comparisons. From the analysis above, we know that the average value of these quantities, when a random permutation of $\{1,2,\ldots,N\}$ is sorted with a sample of size 2^k-1 is, for $M = 0$:

$$\begin{aligned} A_{Nk}^{T} &= N + 2^k - 1 \quad ; \\ B_{Nk}^{T} &= \frac{1}{3}(N+1)H_N - \frac{1}{3}(N+1)H_{2^k} + \frac{1}{2}(N+1+2^k)\left(\frac{k}{2} - 1\right) \\ &\qquad + \frac{4}{3} - \frac{1}{4}\epsilon_k(N+1) \quad ; \\ C_{Nk}^{T} &= 2(N+1)H_N + (N+1-2^k)(k - 2H_{2^k}) \quad . \end{aligned}$$

Carrying out the calculations exactly as above, we find that the value of k which minimizes the total average running time $34A_{Nk}^{T} + 11B_{Nk}^{T} + 4C_{Nk}^{T}$ is the smallest k for which

$$N+1 \; < \; \frac{2^{k+3}\left(\frac{52}{16} - \frac{5}{32}\,k + 2\,H_{2^{k+1}} - 2\,H_{2^k}\right)}{\frac{35}{3}\left(H_{2^{k+1}} - H_{2^k}\right) - \frac{27}{4} + \frac{11}{2^{k+3}+4}} \quad .$$

This leads to much smaller sample sizes than those which minimized the number of comparisons, as can be seen in the table below:

N	1569		3290		6935		14634		30851		64927
k		6		7		8		9		10	

Now a sample of size 63 minimizes the total running time for much larger files than before: in the range $1569 \leq N \leq 3290$. This does not affect the asymptotic behavior of the average number of comparisons -- it is still N lg N if samples are chosen in this way. These results are all distorted somewhat because we have taken $M = 0$, but, because each choice of the sample size spans such a broad range of values of N , they do provide a fairly good indication of how the sample size should be chosen when Program 7.1 is used in a practical situation, with M about equal to 10 .

There are variations in the implementation of samplesort which may be more efficient than Program 7.1 in some situations. The most important of these stems from the observation that the exchanges required to move the sample elements about during partitioning may not really be necessary. We might consider a program which chooses the partitioning elements from the sample in exactly the same way as Program 7.1, but which leaves the sample untouched and partitions the rest of the array on the sample values only. The result is two sorted arrays: the sample

$(A[1],A[2],\ldots,A[2t+1])$, which was sorted initially, and the rest of the array $(A[2t+2],A[2t+3],\ldots,A[N])$, which is sorted by the partitioning process. In some situations, this result may be sufficient. For example, if the file is to be output immediately then the two subarrays may be merged during the output process, at the cost of only $N-2t-1$ comparisons. Of course, it may require some extra overhead to implement the partitioning method in this way -- but in many situations it can be made nearly as efficient as Program 7.1. A related modification can be made if the array is input just before it is sorted. In such a case, the sample may be spread out over the file (for example, every $\left\lfloor \frac{N}{2t+1} \right\rfloor$-th element may be taken) at no extra cost. However, none of these modifications improve samplesort to the point where it can compete with the median-of-sample method discussed in Chapter 8 in most practical situations. (An exception to this is when the file is known to be strongly biased. Taking a large sample is one way to measure and react to such a bias.)

Even though samplesort is asymptotically optimal with respect to the number of comparisons, it is interesting to consider ways of making it more efficient for very, very, <u>very</u> large values of N . When N is extremely large, the sample is also very large, and it may be inefficient to simply quicksort it. Indeed, why not samplesort it, since samplesort is known to be asymptotically optimal for large N ? But the sample used to sort the sample may also be very large, so it might be more efficient to samplesort <u>it</u>! Continuing in this way, we are led to a sequence of sample sizes $2^{k_1}-1 , 2^{k_2}-1 , \ldots , 2^{k_n}-1$, and our file can be sorted by the sequence

```
samplesort $(1, 2^{k_1}-1, 0)$ ;
samplesort $(1, 2^{k_2}-1, 2^{k_1-1}-1)$ ;
samplesort $(1, 2^{k_3}-1, 2^{k_2-1}-1)$ ;
    ⋮
samplesort $(1, 2^{k_n}-1, 2^{k_{n-1}-1}-1)$ ;
samplesort $(1, N, 2^{k_n-1}-1)$ .
```

This is an intriguing idea, but even for $n = 2$ or 3 this method will achieve significant savings only for truly astronomical values of N.

In fact, we find that Program 7.1 itself achieves significant savings only for very large N. Even if we examine only the average number of comparisons, we find that it converges very slowly to its asymptotically optimal level. The table below shows, for some values of N in the range $1000 \le N \le 100{,}000$, the value of $C_N^T / N \ln N$, where C_N^T is the average total number of comparisons used by Program 7.1 when the optimal sample size is used.

N	1000	2000	5000	10000	20000	30000	40000	50000	100000	∞
$\dfrac{C_N^T}{N \ln N}$	1.681	1.665	1.648	1.636	1.627	1.624	1.619	1.615	1.608	$\frac{1}{\ln 2} = 1.442695\ldots$

For normal quicksort, this ratio is about 2, and as $N \to \infty$, the ratio approaches $\frac{N \lg N}{N \ln N} = \frac{1}{\ln 2}$. But for practical values of N, the ratio is not very close to this optimal value. The method discussed in the next chapter is much simpler than samplesort, and, as we will see, the ratio of the number of comparisons taken to N ln N is about $\frac{12}{7} \approx 1.714$, which makes it better than samplesort for practical size files, since it incurs significantly less overhead in other parts of the program. The idea of estimating the distribution of the keys to be sorted by taking a very large sample, then partitioning according to this sample, is very appealing in theory -- the soundness of the idea is borne out by the asymptotic analysis. However, for files in a practical size range, it seems that the technique is too powerful, and is less efficient than the much simpler implementation of the notion of using the median of a sample that we will examine next.

CHAPTER EIGHT

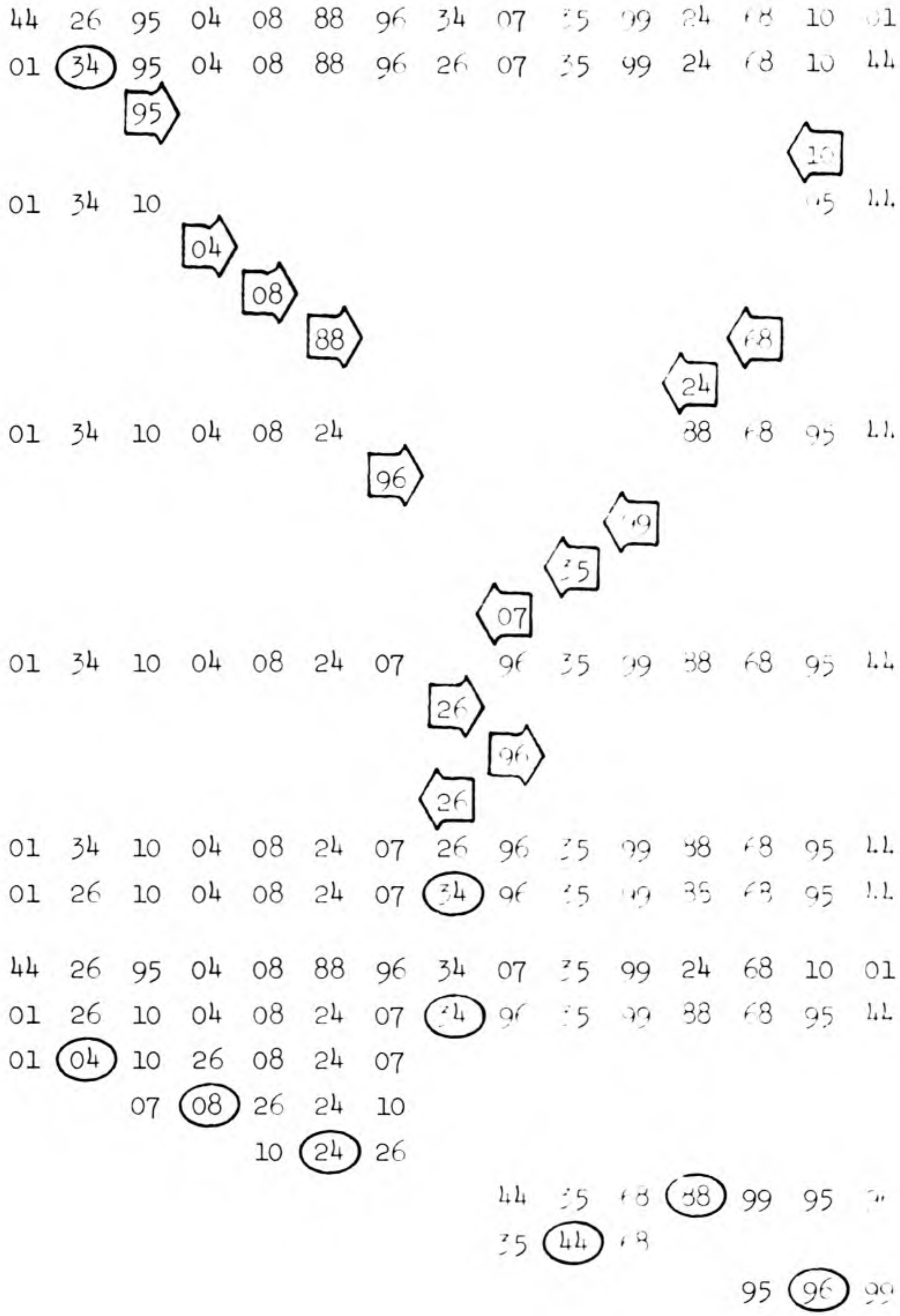
Samplesort achieves "global" balancing of the partitioning tree by ensuring that the first k levels are complete, where the sample size 2^k-1 is chosen large enough to make the balancing worthwhile, but small enough so that the balancing is not prohibitively expensive. An alternative approach, which turns out to be very fruitful, is to perform "local" balancing of the partitioning tree at every stage, by ensuring that the left and right subfiles are nonempty. This is done by choosing a small sample, then partitioning on the median of that sample. In fact, it turns out that very small samples are sufficient, and most of the savings inherent in this idea can be achieved simply by partitioning on the median of three elements at each partitioning stage. To make the worst case unlikely, we will choose the first, middle, and last elements as the sample. We still must be careful to ensure that the subfiles remain random, as in the following implementation of this median-of-three quicksort:

Program 8.1

```
procedure quicksort (integer value ℓ,r);
  if r-ℓ ≥ M then
    if A[ℓ] > A[r] then A[ℓ] :=: A[r] endif;
    if A[(ℓ+r) ÷ 2] > A[r] then A[(ℓ+r) ÷ 2] :=: A[r] endif;
    if A[ℓ] > A[(ℓ+r) ÷ 2] then A[ℓ] :=: A[(ℓ+r) ÷ 2] endif;
    A[ℓ+1] :=: A[(ℓ+r) ÷ 2];
    i := ℓ+1; j := r; v := A[ℓ+1];
    loop:
        loop: i := i+1; while A[i] < r repeat;
        loop: j := j-1; while A[j] > r repeat;
    while i < j:
        A[i] :=: A[j];
    repeat;
    A[ℓ+1] :=: A[j]
    quicksort (ℓ,j-1);
    quicksort (j+1,r);
  endif;
```

The three consecutive if statements at the beginning of the partitioning process simply put the elements A[ℓ] , A[(ℓ+r) ÷ 2] , and A[r] into order.

Example 8.1



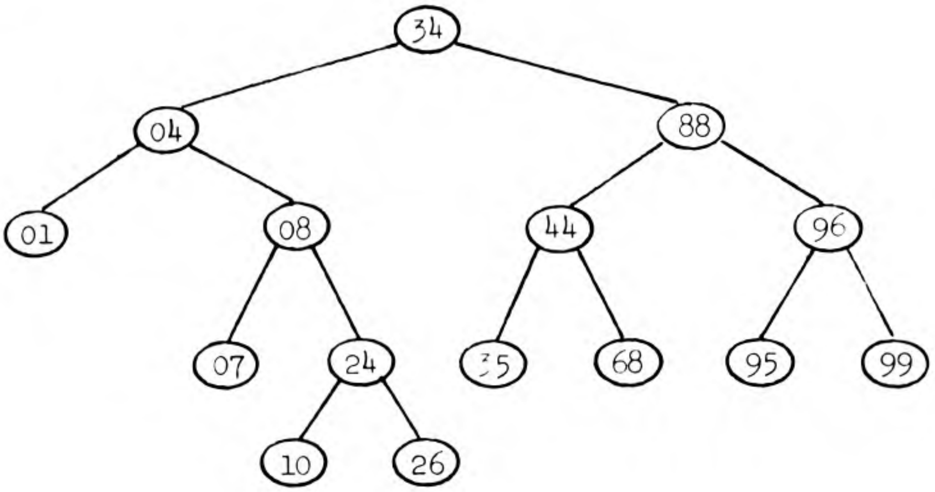
partitioning:

```
44   26   95   04   08   88   96   34   07   35   99   24   68   10   01
01  (34)  95   04   08   88   96   26   07   35   99   24   68   10   44
          95>
                                                                <10
01   34   10                                                    95   44
               04>
                    08>
                         88>
                                                           <68
                                                      <24
01   34   10   04   08   24                             88   68   95   44
                              96>
                                                 <99
                                            <35
                                       <07
01   34   10   04   08   24   07        96   35   99   88   68   95   44
                                   26>
                                        96>
                                  <26
01   34   10   04   08   24   07   26   96   35   99   88   68   95   44
01   26   10   04   08   24   07  (34)  96   35   99   88   68   95   44
```

sorting
the file:

```
44   26   95   04   08   88   96   34   07   35   99   24   68   10   01
01   26   10   04   08   24   07  (34)  96   35   99   88   68   95   44
01  (04)  10   26   08   24   07
          07  (08)  26   24   10
                    10  (24)  26
                                        44   35   68  (88)  99   95   96
                                        35  (44)  68
                                                            95  (96)  99
```

Our sixteen sample keys are sorted by this method as shown in Example 8.1. The partitioning tree is



which is nearly perfectly balanced, even though the partitioning of the left subfile represents the worst case. The local balancing which is effected by the partitioning procedure ensures that every node which has at least two descendents has non-empty left and right subtrees. In addition, as we will see, the partitioning element tends to be close to the middle, so that each subtree tends to be relatively balanced.

It is possible to replace the three <u>if</u> statements in Program 8.1 which put $A[\ell]$, $A[(\ell+r) \div 2]$, and $A[r]$ into order by slightly more efficient code in some situations, since exchanges can be unnecessarily expensive when so few elements are sorted. However, it is important to be sure that the smallest and largest elements are not included in the partitioning process, at the peril of producing non-random subfiles. One method which produces random subfiles and does seem to avoid many of the extra exchanges involved in Program 7.1 is to determine which of $A[\ell]$, $A[(\ell+r) \div 2]$, and $A[r]$ is the median of the three; then exchange that with $A[\ell]$: and partition as usual. This method would seem to involve fewer exchanges than Program 8.1, but it is not really more efficient, since if extra exchanges are necessary in Program 8.1, then corresponding exchanges will always be required to achieve partitioning. However, the real reason that this method is not attractive is that it fails to avoid the worst case, in a subtle and surprising way. Suppose the file to be sorted is

$$N \quad 1 \quad 2 \quad 3 \quad \ldots \quad N-3 \quad N-2 \quad N-1 \ .$$

Then the median of N , $\left\lfloor \frac{N-1}{2} \right\rfloor$, and $N-1$ is $N-1$, so after the median is exchanged with N (the element in $A[\ell]$) we have

$$N-1 \quad 1 \quad 2 \quad 3 \quad \ldots \quad N-3 \quad N-2 \quad N \ ,$$

and this array is partitioned (by, for example, Partitioning Method 2.4), to yield

$$N-2 \quad 1 \quad 2 \quad 3 \quad \ldots \quad N-4 \quad N-3 \quad N-1 \quad N \ .$$

Not only is this a degenerate partition, but also the left subfile has the same form as the original, which means that the next, and, by induction, all succeeding partitions will be degenerate. What is worse, this type of file appears as the right subfile when the input file is

$$N \quad N-1 \quad N-2 \quad \ldots \quad 4 \quad 3 \quad 2 \quad 1 \ .$$

From this example, we can be reminded of the delicacy of the partitioning process that we found when we first examined partitioning methods in Chapter 2. We have to beware of non-random subfiles, and of simple anomalies which might lead to the worst case in a practical situation. The careful implementation of Program 8.1 seems to avoid such anomalies, to produce random subfiles, and still to run very efficiently.

The analysis of Program 8.1 generalizes very nicely, so before we attack the specific analysis of the average running time of Program 8.1, we will consider a more general situation. Suppose that we have any partitioning method which bases its decision on what should be the partitioning element at each stage by examining $2t+1$ elements, and that it chooses the k-th largest of these elements with probability p_k, for $k = 1,2,\ldots,2t+1$. We shall derive an expression for the leading term in the total number of comparisons taken by Quicksort when such a partitioning method is used. (If we take $t = 1$ and $p_2 = 1$, $p_1 = p_3 = 0$, then we are discussing Program 8.1.) For simplicity in the analysis, we will assume that the number of comparisons used to partition a random permutation of $\{1,2,\ldots,N\}$ is $N+1$. This is the "natural" number which comes up in the analysis -- since we are deriving only the leading term, we are really assuming that the number of comparisons for the first partitioning stage is $N+1+O(1)$, which it certainly will be. Finally, we must of course assume that the partitioning method always produces random subfiles. Now, if a random permutation of $\{1,2,\ldots,N\}$ is being sorted, the probability that an element s is the partitioning element is

$$\sum_{1\le k\le 2t+1} p_k \frac{\binom{N-s}{2t+1-k}\binom{s-1}{k-1}}{\binom{N}{2t+1}} ,$$

or the sum over all k of the probability that s is the k-th largest of the $2t+1$ elements being examined times the probability that it is selected. This means that our standard recurrence describing the number of comparisons becomes

$$C_N = N+1+ \sum_{1 \le s \le N} \sum_{1 \le k \le 2t+1} p_k \frac{\binom{N-s}{2t+1-k}\binom{s-1}{k-1}}{\binom{N}{2t+1}} (C_{s-1}+C_{N-s}) \ .$$

As always, the sum involving C_{s-1} and the sum involving C_{N-s} are about the same (substitute $s = N+1-s$ and $k = 2t+2-k$ in the second), and, after multiplying both sides by $\binom{N}{2t+1}$, we get

$$\binom{N}{2t+1} C_N = (N+1)\binom{N}{2t+1} + \sum_{1 \le s \le N} \sum_{1 \le k \le 2t+1} q_k \binom{N-s}{2t+1-k}\binom{s-1}{k-1} C_{s-1}$$

$$= (2t+2)\binom{N+1}{2t+2} + \sum_{1 \le k \le 2t+1} \sum_{1 \le s \le N} q_k \binom{N-s}{2t+1-k}\binom{s-1}{k-1} C_{s-1} \ ,$$

where $q_k \equiv p_k + p_{2t+2-k}$. With the view that we should apply generating functions to solve this equation, let us multiply both sides by z^N and sum over all N :

$$\sum_{N \ge 0} \binom{N}{2t+1} C_N z^N = \sum_{N \ge 0} (2t+2)\binom{N+1}{2t+2} z^N$$

$$+ \sum_{N \ge 0} \sum_{1 \le k \le 2t+1} \sum_{1 \le s \le N} q_k \binom{N-s}{2t+1-k}\binom{s-1}{k-1} C_{s-1} z^N \ .$$

The first summation on the right hand side is the generating function for binomial coefficients (see Eq. (34) in Appendix B), and we can see that the sums on N and s in the second term comprise a convolution. Replacing s by $s+1$, N by $N+1$ and then interchanging the order of summation, we have

$$\sum_{N\geq 0}\binom{N}{2t+1}C_N z^N = (2t+2)\,\frac{z^{2t+1}}{(1-z)^{2t+3}}$$

$$+\; z \sum_{1\leq k\leq 2t+1}\;\sum_{s\geq 0}\;\sum_{N\geq s} q_k\binom{N-s}{2t+1-k}\binom{s}{k-1}C_s\, z^N$$

$$= (2t+2)\,\frac{z^{2t+1}}{(1-z)^{2t+3}}$$

$$+\; z \sum_{1\leq k\leq 2t+1} q_k\Bigl(\sum_{s\geq 0}\binom{s}{k-1}C_s\, z^s\Bigr)\Bigl(\sum_{N\geq 0}\binom{N}{2t+1-k}z^N\Bigr)$$

$$= (2t+2)\,\frac{z^{2t+1}}{(1-z)^{2t+3}}$$

$$+\; z \sum_{1\leq k\leq 2t+1} q_k\Bigl(\sum_{s\geq 0}\binom{s}{k-1}C_s\, z^s\Bigr)\frac{z^{2t+1-k}}{(1-z)^{2t+2-k}} \quad .$$

Now, if $C(z) = \sum_{N\geq 0} C_N z^N$ is the generating function for $\{C_N\}$, then by differentiating j times, we get the equation

$$C^{(j)}(z) = \sum_{N\geq j} N(N-1)\ldots(N-j+1)C_N\, z^{N-j} \quad ,$$

or

$$\sum_{N\geq 0}\binom{N}{j}C_N\, z^N = \frac{C^{(j)}(z)\; z^j}{j!} \quad .$$

(See Eq. (41) in Appendix B.) This equation allows us to express our recurrence in terms of the generating function $C(z)$. We have

$$\frac{C^{(2t+1)}(z)\ z^{2t+1}}{(2t+1)!} = (2t+2)\ \frac{z^{2t+1}}{(1-z)^{2t+3}}$$

$$+ \sum_{1 \le k \le 2t+1} q_k\ \frac{C^{(k-1)}(z)\ z^{k-1}}{(k-1)!}\ \frac{z^{2t+1-k}}{(1-z)^{2t+2-k}} \quad .$$

This equation is simplified somewhat by multiplying both sides by $\frac{(1-z)^{2t+1}}{z^{2t+1}}$, which leaves

$$\frac{(1-z)^{2t+1}\ C^{(2t+1)}(z)}{(2t+1)!} = \frac{(2t+2)}{(1-z)^2} + \sum_{1 \le k \le 2t+1} q_k\ \frac{(1-z)^{k-1}\ C^{(k-1)}(z)}{(k-1)!} \quad .$$

Although we could continue to work with the equation in this form, the next few manipulations that we do will be clearer if we change variables to $x = 1-z$, and then define $f(x) = C(1-x)$. Then $f^{(j)}(x) = (-1)^j\ C^{(j)}(1-x)$, and our equation is

$$\frac{x^{2t+1}(-1)^{2t+1}\ f^{(2t+1)}(x)}{(2t+1)!} - \sum_{1 \le k \le 2t+1} q_k\ \frac{x^{k-1}(-1)^{k-1}\ f^{(k-1)}(x)}{(k-1)!} = \frac{(2t+2)}{x^2} \quad .$$

This is a differential equation of order $2t+1$ in $f(x)$. But it is also of degree $2t+1$, and it is in fact a special kind of differential equation since every term of order j is also of degree j for all j . This kind of differential equation can be solved by introducing the operator Θ defined by

$$\Theta f(x) = x\ f'(x) \quad .$$

Now, we can use this operator to produce terms of order j and degree j for any j :

$$\begin{aligned}
\theta(\theta-1)f(x) &= \theta(xf'(x)-f(x)) &&= x^2f''(x) \quad ; \\
\theta(\theta-1)(\theta-2)f(x) &= \theta(\theta-1)(xf'(x)-2f(x)) &&= x^3f'''(x) \quad : \\
\theta(\theta-1)(\theta-2)(\theta-3)f(x) & &&= x^4f^{(4)}(x) \quad ; \\
&\vdots \\
\theta(\theta-1)(\theta-2)\ldots(\theta-j+1)f(x) & &&= x^jf^{(j)}(x) \quad .
\end{aligned}$$

Dividing both sides by $j!$, we have

$$\binom{\theta}{j}f(x) = \frac{x^jf^{(j)}(x)}{j!} \quad ,$$

where it is understood that the left hand side is to evaluated by expanding the binomial coefficient formally from the definition $\binom{\theta}{j} \equiv \frac{\theta(\theta-1)\ldots(\theta-j+1)}{j!}$, then applying the operator θ as indicated. This equation is easily proved by induction. Substituting this into our recurrence gives

$$(-1)^{2t+1}\binom{\theta}{2t+1}f(x) - \sum_{1\le k\le 2t+1} q_k(-1)^{k-1}\binom{\theta}{k-1}f(x) = \frac{2t+2}{x^2} \quad .$$

Denoting by $P(\theta)$ the operator polynomial which is applied to $f(x)$, this equation is

$$P(\theta)f(x) = \frac{2t+2}{x^2}$$

where

$$P(\Theta) = (-1)^{2t+1}\binom{\Theta}{2t+1} - \sum_{1 \le k \le 2t+1} q_k(-1)^{k-1}\binom{\Theta}{k-1} \quad .$$

To proceed further, we need to be able to factor this polynomial. First, we notice that, since $\binom{-2}{j} = (-1)^j(j+1)$, one root of the polynomial is always -2 :

$$P(-2) = (-1)^{2t+1}\binom{-2}{2t+1} - \sum_{1 \le k \le 2t+1} q_k(-1)^{k-1}\binom{-2}{k-1}$$

$$= (2t+2) - \sum_{1 \le k \le 2t+1} k\ q_k$$

$$= (2t+2) - \sum_{1 \le k \le 2t+1} k\ p_k \quad \sum_{1 \le k \le 2t+1} k\ p_{2t+2-k}$$

$$= (2t+2) - \sum_{1 \le k \le 2t+1} (k + (2t+2-k))p_k = 0 \quad .$$

Therefore we can write $P(\Theta) \equiv Q(\Theta)(\Theta+2)$, and

$$Q(\Theta)(\Theta+2)f(x) = \frac{2t+2}{x^2} \quad .$$

Now, suppose that the roots of $Q(\Theta)$ are $r_1, r_2, \ldots, r_{2t}$, so that we have

$$Q(\Theta) = (\Theta-r_1)(\Theta-r_2)\ldots(\Theta-r_{2t}) \quad ,$$

and

$$(\Theta-r_1)(\Theta-r_2)\ldots(\Theta-r_{2t})(\Theta+2)f(x) = \frac{2t+2}{x^2} \quad .$$

We can solve this equation, because we know from integral calculus that the solution to the differential equation

$$(\Theta-\alpha)g(x) = x^{\beta}$$

is

$$g(x) = \begin{cases} \dfrac{x^{\beta}}{\beta-\alpha} + c\,x^{\alpha} \quad , & \text{if } \alpha \neq \beta \ ; \\ x^{\beta} \ln x + c\,x^{\alpha} \quad , & \text{if } \alpha = \beta \ . \end{cases}$$

Here c represents a constant of integration whose value depends on the initial conditions. Applying this general solution 2t+1 times, we can derive an expression for $f(x)$:

$$(\Theta-r_1)(\Theta-r_2)(\Theta-r_3)\ldots(\Theta-r_{2t})(\Theta+2)f(x) = \frac{2t+2}{x^2} \quad ;$$

$$(\Theta-r_2)(\Theta-r_3)\ldots(\Theta-r_{2t})(\Theta+2)f(x) = \frac{2t+2}{(-2-r_1)x^2} + c_1\,x^{r_1} \quad ;$$

$$(\Theta-r_3)\ldots(\Theta-r_{2t})(\Theta+2)f(x) = \frac{2t+2}{(-2-r_1)(-2-r_2x^2} + c_1\,x^{r_1} + c_2\,x^{r_2} \quad ;$$

$$\vdots$$

$$(\Theta+2)f(x) = \frac{2t+2}{(-2-r_1)(-2-r_2)\ldots)-2-r_{2t})x^2} + \sum_{1 \le k \le 2t} c_k\,x^{r_k} \quad ;$$

$$f(x) = \frac{2t+2}{(-2-r_1)(-2-r_2)\ldots(-2-r_{2t})} \frac{\ln x}{x^2} + \sum_{1 \le k \le 2t+1} c_k\,x^{r_k} \quad (r_{2t+1} \equiv -2) \ .$$

The constants c_k change at each step in this derivation, but these changes are not indicated because we are not interested in their values. Also, it is implicitly assumed that $r_1, r_2, \ldots, r_{2t}$ are distinct and not equal to -2. (If they are not, the "remainder" term $\sum_{1 \le k \le 2t+1} c_k x^{r_k}$ becomes slightly more complicated -- we will ignore such effects for now.) We notice that $(-2-r_1)(-2-r_2)\ldots(-2-r_{2t}) = Q(-2)$, and, postponing for the moment the problem of evaluating this, we can change back to our original notation,

$$C(z) = \frac{2t+2}{Q(-2)} \frac{\ln(1-z)}{(1-z)^2} + \sum_{1 \le k \le 2t+1} c_k (1-z)^{r_k} ,$$

and then expand back to power series. By the Binomial Theorem, $(1-z)^{r_k} = \sum_{N \ge 0} (-1)^N \binom{r_k}{N} z^N$. The series for $\frac{\ln(1-z)}{(1-z)^2}$ is a special case of Eq. (42) in Appendix B and is the generating function for the sums of the harmonic numbers. Specifically, Eqs. (39), (38) and (4) in Appendix B imply that

$$\frac{\ln(1-z)}{(1-z)^2} = - \sum_{N \ge 0} ((N+1)H_N - N) z^N .$$

Therefore, substituting these in, we have

$$C(z) = -\frac{2t+2}{Q(-2)} \sum_{N\geq 0} ((N+1)H_N - N)z^N + \sum_{N\geq 0} \sum_{1\leq k\leq 2t+1} c_k(-1)^N \binom{r_k}{N} z^N ,$$

so that

$$C_N = -\frac{2t+2}{Q(-2)} (N+1)H_{N+1} + O(N) \qquad ,$$

as long as the r_k behave properly.

It remains to evaluate $Q(-2)$. First, we observe that

$$P(\Theta) = (\Theta+2)Q(\Theta) \quad ,$$

so that

$$P'(\Theta) = (\Theta+2)Q'(\Theta) + Q(\Theta)$$

and

$$P'(-2) = Q(-2) \quad .$$

To evaluate $P'(-2)$ we first notice that in general formal differentiation gives

$$\binom{\Theta}{k}' = \sum_{0\leq j\leq k-1} \frac{1}{\Theta-j} \binom{\Theta}{k}$$

so that

$$P'(\Theta) = (-1)^{2t+1} \sum_{0\leq j\leq 2t} \frac{1}{\Theta-j}\binom{\Theta}{2t+1}$$

$$- \sum_{1\leq k\leq 2t+1} q_k(-1)^{k-1} \sum_{0\leq j\leq k-2} \frac{1}{\Theta-j}\binom{\Theta}{k-1}$$

and

$$P'(-2) = \binom{-2}{2t+1} \sum_{0 \le j \le 2t} \frac{1}{j+2} + \sum_{1 \le k \le 2t+1} q_k (-1)^{k-1} \binom{-2}{k-1} \sum_{0 \le j \le k-2} \frac{1}{j+2}$$

$$= -(2t+2)(H_{2t+2} - 1) + \sum_{1 \le k \le 2t+1} k\, q_k (H_k - 1) \quad ,$$

or

$$Q(-2) = -(2t+2)H_{2t+2} + \sum_{1 \le k \le 2t+1} k\, q_k H_k \quad ,$$

where again we have used the fact that $\sum_{1 \le k \le 2t+1} k\, q_k = 2t+2$. (It is not difficult to see that this expression for $P'(-2)$ is always non-zero, so that -2 is therefore not a multiple root of $P(\theta)$.)

This now leads to a final expression for the average number of comparisons:

$$C_N = \frac{1}{H_{2t+2} - \frac{1}{2t+2} \sum_{1 \le k \le 2t+1} k\, H_k\, q_k} (N+1)H_{N+1} + O(N) \quad .$$

This amazingly simple general formula includes our normal quicksort as a special case. If we partition on the basis of examining only one element, as we did in Chapter 2, then we can find the asymptotic behavior of the average number of comparisons by taking $t = 0$ and $p_1 = 1$ $(q_1 = 2)$ in the formula, which yields

$$C_N = \frac{1}{H_2 - H_1} (N+1)H_{N+1} + O(N)$$

$$= 2(N+1)H_{N+1} + O(N) \quad .$$

This, of course, agrees with the exact formula that we derived in Chapter 3. For $t > 0$, we need to consider reasonable probability distributions for the choice of the partitioning element. One common distribution we might consider is the binomial distribution:

$$p_k = \binom{2t}{k-1}\frac{1}{2^{2t}} \qquad (q_k = 2p_k) \quad .$$

We can substitute this into our general equation to show that (see Appendix B)

$$C_N = \frac{1}{\ln 2 + \frac{1}{2t+1} - \frac{2}{2t+2} + \epsilon}(N+1)H_{N+1} + O(N) \quad ,$$

$$\text{where } |\epsilon| < \frac{1}{(2t+2)2^{2t-1}} \quad .$$

This looks very good, because it is very close to $\frac{1}{\ln 2}(N+1)H_{N+1} + O(N)$ for even moderately large t, and we know that this is the best that we can hope to do, since $\frac{1}{\ln 2}(N+1)H_{N+1} + O(N) = \frac{1}{\ln 2} N \ln N + O(N) = N \lg N + O(N)$. We could examine other distributions and see how they compare, using the general formula, but it turns out to be unnecessary to do so. We can show, as we have already mentioned in this and the previous chapter, that the best way to proceed is always to choose the median of the $2t+1$ elements ($p_{t+1} = 1$, $p_k = 0$ for $k \neq t+1$.) The formula says that the average number of comparisons taken by this method is

$$C_N = \frac{1}{H_{2t+2} - H_{t+1}}(N+1)H_{N+1} + O(N) \quad ,$$

and this is proved optimal in the following:

Theorem 8.1.

Suppose that a quicksort program is based on a partitioning strategy which examines $2t+1$ elements and chooses from among them according to fixed probabilities p_k , $1 \le k \le 2t+1$. Then the coefficient of the leading term of the average number of comparisons required by the program to sort a random permutation of $\{1,2,\ldots,N\}$ is minimized by the distribution $p_{t+1} = 1$; $p_k = 0$, $k \neq t+1$. That is, it is best to always choose the median element of the sample as the partitioning element.

Proof. From the discussion above, our objective is to show that

$$\sum_{1 \le k \le 2t+1} k H_k (p_k + p_{2t+2-k}) \ge (2t+2) H_{t+1}$$

for all probability distributions $\{p_k\}$. First, we change k to $2t+2-k$ in the seccnd part of the summation to get

$$\sum_{1 \le k \le 2t+1} k H_k (p_k + p_{2t+2-k}) = \sum_{1 \le k \le 2t+1} p_k (k H_k + (2t+2-k) H_{2t+2-k}) \;.$$

Since $\Delta k H_k = (k+1)H_{k+1} - kH_k = H_{k-1}+1$, we know that

$$\begin{aligned} k H_k + (2t+2-k)H_{2t+2-k} &= (k+1)H_{k+1} - H_k - 1 + (2t+2-k-1)H_{2t+2-k-1} \\ &\qquad + H_{2t+2-k} + 1 \\ &> (k+1)H_{k+1} + (2t+2-k-1)H_{2t+2-k-1} \end{aligned}$$

for $1 \le k \le t$, since $H_{2t+2-k} > H_k$ for k in this range. Therefore, telescoping the inequality, we see that it holds for all k :

$$kH_k + (2t+2-k)H_{2t+2-k} > (2t+2)H_{t+1} \qquad \text{for } 1 \le k \le t \ ,$$

and by symmetry

$$kH_k + (2t+2-k)H_{2t+2-k} \ge (2t+2)H_{t+1} \qquad \text{for } 1 \le k \le 2t+1 \ ,$$

with equality holding only for $k = t+1$. Substituting this into our summation gives

$$\sum_{1\le k\le 2t+1} kH_k(p_k + p_{2t+2-k}) \ge \sum_{1\le k\le 2t+1} p_k(2t+2)H_{t+1}$$

which is the desired result since $\sum_{1\le k\le 2t+1} p_k = 1$. Equality holds only if $p_{t+1} = 1$ and $p_k = 0$ for $k \ne t+1$.

It remains to show that the roots r_k of the operator polynomial $P(\Theta)$ "behave properly" when the median element is used as the partitioning element, so that the $(N+1)H_{N+1}$ term is indeed the leading term. The polynomial

$$P(\Theta) = (-1)^{2t+1}\binom{\Theta}{2t+1} - (-1)^t\binom{\Theta}{t}$$

can be factored into the form

$$P(\Theta) = \frac{(-1)^t\binom{\Theta}{t}}{\binom{2t+2}{t}}\left(\binom{2t-\Theta}{t+1} - \binom{2t+2}{t+1}\right) \ ,$$

from which we can infer that $0,1,\ldots,t-1$ are all roots as well as -2. The other roots do not seem so easy to find explicitly (except that when t is odd, $\binom{2t-\Theta}{t+1} = \binom{\Theta-t}{t+1}$, so $\Theta = 3t+2$ is also a root). We could proceed from this point and use complex variable theory to show that if $\binom{2t-r_k}{t+1} = \binom{2t+2}{t+1}$ and $r_k \neq -2$ then we must have $\binom{r_k}{N} = O(N)$. (We would also have to consider the slight extra complication introduced by multiple roots.) However, we shall use a more elementary indirect argument to show that the number of comparisons is $O(N \ln N)$.

Specifically, we can show from the recurrence for the number of comparisons when partitioning is done on the median of $2t+1$ elements,

$$C_N = N+1+2 \sum_{1 \leq s \leq N} \frac{\binom{N-s}{t}\binom{s-1}{t}}{\binom{N}{2t+1}} C_{s-1} \quad ,$$

that $C_N < 2(N+1)H_{N+1}$ regardless of the value of $t \geq 0$.

The proof, of course, is by induction. The assertion is certainly true for $N < 2t+1$ since $C_N = N+1 < 2(N+1)H_{N+1}$ for these values of N . For $N \geq 2t+1$, we have by the inductive hypothesis

$$C_N < N+1+\frac{2}{\binom{N}{2t+1}} \sum_{1 \leq s \leq N} \binom{N-s}{t}\binom{s-1}{t} 2s H_s$$

$$= N+1+\frac{4(t+1)}{\binom{N}{2t+1}} \sum_{1 \leq s \leq N} \binom{N-s}{t}\binom{s}{t+1} H_s \quad .$$

This seemingly formidable sum involving binomial coefficients and harmonic numbers is related to a sum we encountered during our study of samplesort. It follows from Eq. (45) in Appendix B that

$$\sum_{1 \leq s \leq N} \binom{N-s}{t}\binom{s}{t+1} H_s = \binom{N+1}{2t+2}(H_N - H_{2t+2} + H_{t+1}) \quad ,$$

so that we now have

$$C_N < 2(N+1)\left(H_N - H_{2t+2} + H_{t+1} + \frac{1}{2}\right) \quad ,$$

or

$$C_N < 2(N+1)H_{N+1} \quad ,$$

since $-H_{2t+2} + H_{t+1} = -1 + \frac{1}{2} - \frac{1}{3} + \frac{1}{4} - \ldots + \frac{1}{2t+2} < -\frac{1}{2}$. This completes the proof of Theorem 8.1 -- we know that the r_k contribute terms of $O(N^\alpha)$ for some α , and this inequality implies that α must be ≤ 1 . □

This general result should give us a great deal of confidence in Program 8.1, and we shall now return to considering that program from a practical standpoint. First, we have shown above that if we partition by choosing the median of $2t+1$ elements, then the average number of comparisons is

$$C_N = \frac{1}{H_{2t+2} - H_{t+1}} (N+1)H_{N+1} + O(N) \quad ,$$

and that this is the best way to proceed. In Program 8.1, we only used three elements $(t = 1)$, and we might ask if it would be practical to use a larger value of t . Intuitively, since t measures work which is done on every partitioning stage, we expect that it should be relatively small -- in fact, the analysis above implicitly assumes that it is $O(1)$. The table below shows how the leading term is affected by making t larger: the exact and approximate values of the coefficient $\frac{1}{H_{2t+2} - H_{t+1}}$ are given for various values of t ; as well as the percentage improvement over the coefficient for $t = 0$ and for the previous value of t .

t		0	1	2	3	4	...	∞
$\frac{1}{H_{2t+2} - H_{t+1}}$	=	2	$\frac{12}{7}$	$\frac{60}{37}$	$\frac{840}{535}$	$\frac{2520}{1627}$		$\frac{1}{\ln 2}$
	≈	2	1.714	1.622	1.576	1.549		1.443
percentage improvement:	over t = 0	0	14.3	18.9	21.2	22.6		32.8
	over t = t-1	-	14.3	5.4	2.8	1.7		0

This table shows that the average number of comparisons required by Program 8.1 is

$$\frac{12}{7}(N+1)H_{N+1} + O(N) \quad ,$$

which is (asymptotically) a 14.3% improvement over Program 2.4. The table also indicates that it probably will not be worthwhile to go to a median-of-five (or higher) quicksort, because the percentage improvement in the leading term is relatively small, and would be washed out by the extra work required to find the median of a larger number of elements, for practical values of N .

Of course, we have considered so far only the number of comparisons, and we know that exchanges will also contribute to the leading term. In Chapter 3 we found that the average number of exchanges required on the first stage given that s is the partitioning element is the number of keys among $A[2],\ldots,A[s]$ which are $> s$, or $\frac{(N-s)(s-1)}{N-1}$. (This may count some exchanges which are used in the process of picking s as the median of $2t+1$ elements, but again such effects will not affect the leading term.) But now the probability that s is the partitioning element is $\dfrac{\binom{s-1}{t}\binom{N-s}{t}}{\binom{N}{2t+1}}$, so the average number of exchanges for the first partitioning stage is

$$\sum_s \frac{\binom{s-1}{t}\binom{N-s}{t}}{\binom{N}{2t+1}} \; \frac{(N-s)(s-1)}{N-1}$$

$$= \frac{\sum_s \binom{s-1}{t}\binom{N-s}{t}(N-s+1)s}{\binom{N}{2t+1}(N-1)} + \frac{N \sum_s \binom{s-1}{t}\binom{N-s}{t}}{\binom{N}{2t+1}(N-1)}$$

$$= \frac{(t+1)^2\binom{N+2}{2t+3}}{\binom{N}{2t+1}(N-1)} + \frac{(t+1)^2 N\binom{N}{2t+1}}{\binom{N}{2t+1}(N-1)}$$

$$= \frac{t+1}{4t+6}(N+1) + O(1) \qquad .$$

As with samplesort, the number of exchanges is asymptotically $1/4$ the number of comparisons. Since slightly more exchanges are involved, the improvement in the total running time is slightly less than the improvement in the number of comparisons as the table below shows.

t		0	1	2	3	...	∞
$\dfrac{4+11\,\frac{t+1}{4t+6}}{H_{2t+2}-H_{t+1}}$	=	$\frac{35}{3}$	$\frac{372}{35}$	$\frac{2670}{259}$	$\frac{48720}{4815}$		$\frac{27}{4\ \ln\ 2}$
	$\approx$	11.67	10.66	10.31	10.12		9.73
percentage improvement:	over t = 0	0	8.5	11.7	13.0		15.8
	over t = t-1	-	8.5	3.3	1.8		0

(This table assumes that the total running time is $4C + 11B + \ldots$, as in the assembly language implementations of Appendix A.) Again, most of the improvement occurs for $t = 1$, and larger samples will probably not be worthwhile.

(We have considered only odd sample sizes that we may have a unique median. The reader might be interested to discover how the preceding analysis applied to even sample sizes shows that it is best to use odd sized samples.)

Our study of the leading term in the total running time has indicated that Program 8.1 may indeed be a very good way to sort. Fortunately we can verify this conclusion, since we can use the analysis above to help us derive a complete exact formula for the total average running time of Program 8.1. To begin this derivation, we follow the analysis above to the point where we began finding roots of the operator polynomial. We had

$$P(\Theta)f(x) = \frac{2t+2}{x^2} \quad ,$$

where

$$P(\Theta) = (-1)^{2t+1}\binom{\Theta}{2t+1} - \sum_{1 \le k \le 2t+1} (p_k + p_{2t+2-k})(-1)^{k-1}\binom{\Theta}{k-1} ,$$

$$x = 1-z \quad ;$$

$$f(x) = C(1-x) \quad ,$$

$$C(z) = \sum_{N \ge 0} C_N \, z^N \quad ,$$

and C_N is the average number of comparisons to sort a random permutation of $\{1,2,\ldots,N\}$. For the median-of-three method that we have implemented in Program 8.1, we substitute $t = 1$, $p_2 = 1$, and $p_1 = p_3 = 0$, so $P(\Theta) = -\binom{\Theta}{3} + 2\Theta = -\frac{1}{6}\,\Theta(\Theta-1)(\Theta-2) + 2\Theta$, which factors to $P(\Theta) = \frac{1}{6}\,(-\Theta)(-2-\Theta)(5-\Theta)$. Our operator equation now becomes

$$(-\Theta)(-2-\Theta)(5-\Theta)f(x) = \frac{24}{x^2} \quad .$$

We could proceed to solve this as we did in the more general case, by successively solving three differential equations. However, we will not do so for an important reason. We wish to obtain an exact expression for the average number of comparisons which holds for general M , just as we did in Chapter 3. By pushing the solution through as above, we are implicitly assuming $M = 0$. It becomes highly inconvenient to work with the generating function $\sum_{N > M} C_N z^N$, which is the actual function that we need. In Chapter 3, we got around this difficulty by manipulating the recurrence and making substitutions in such a way as to reduce the problem to telescoping recurrences that we could solve. Fortunately, we can also do this in this case, and the method for doing so is suggested by the factorization in terms of Θ given above. Consider the application of the innermost factor $(5-\Theta)$ to $f(x)$. By definition, we have $f(x) = C(1-x) = \sum_{N\geq 0} C_N(1-x)^N$, so

$$(5-\Theta)f(x) = 5\sum_{N\geq 0} C_N(1-x)^N - \Theta \sum_{N\geq 0} C_N(1-x)^N$$

$$= 5\sum_{N\geq 0} C_N(1-x)^N + x \sum_{N\geq 0} C_N\, N(1-x)^{N-1}$$

Switching variables on the right hand side to $z = 1-x$, we get

$$(5-\Theta)f(x) = 5\sum_{N\geq 0} C_N z^N + (1-z) \sum_{N\geq 0} C_N\, N\, z^{N-1} \quad , \quad \text{or}$$

$$(5-\Theta)C(z) = \sum_{N\geq 0} ((N+1)C_{N+1} - (N-5)C_N)z^N \quad .$$

This means that if we define $T(z) = (5-\Theta)C(z)$, and the expansion into power series is $T(z) = \sum_{N\geq 0} T_N\, z^N$, then we must have

$$T_N = (N+1)C_{N+1} - (N-5)C_N \quad .$$

We can continue in this manner for each factor of the operator polynomial. Next we define $U(z) = (-2-\Theta)T(z)$, which means that if $U(z) = \sum_{N\geq 0} U_N z^N$ then

$$U_N = (N+1)T_{N+1} - (N+2)T_N \quad ,$$

and, similarly if $V(z) = (-\Theta)U(z) = \sum_{N \geq 0} V_N z^N$, then

$$V_N = (N+1)U_{N+1} - N\,U_N \quad .$$

But then by definition, we have

$$V(z) = (-\Theta)(-2-\Theta)(5-\Theta)C(z) \quad ,$$

and the original equation that we derived then says that

$$V(z) = \frac{24}{(1-z)^2} \quad ,$$

or, in other words, $V_N = 24(N+1)$. These equations now give us a method to solve for C_N by substitution, in terms of telescoping recurrences that we can stop at the value M . First we solve the equation

$$(N+1)U_{N+1} = N\,U_N + 24(N+1) \quad ,$$

then we use the result of that to solve

$$(N+1)T_{N+1} = (N+2)T_N + U_N \quad ,$$

and then we use the result of that to solve

$$(N+1)C_{N+1} = (N-5)C_N + T_N \quad .$$

We get the initial values for all of these equations by again working backwards. This last equation, when evaluated at $N = M+1$ and $N = M+2$ says that

$$T_{M+1} = (M+2)C_{M+2} - (M-4)C_{M+1} \qquad \text{and}$$

$$T_{M+2} = (M+3)C_{M+3} - (M-3)C_{M+2} \quad .$$

Similarly, we have

$$U_{M+1} = (M+2)T_{M+2} - (M+3)T_{M+1} \quad .$$

Now, the recurrence that we are solving is

$$C_N = \begin{cases} N+1+2 \displaystyle\sum_{1 \leq s \leq N} \frac{(s-1)(N-s)}{\binom{N}{3}} C_{s-1} & N > M \\ \\ 0 & N \leq M \quad , \end{cases}$$

which says in particular that

$$C_{M+1} = M+2 \quad ,$$

$$C_{M+2} = M+3 \quad , \quad \text{and}$$

$$C_{M+3} = M+4 + \frac{12}{M+3} \quad .$$

Substituting these gives the initial values that we need for T and U :

$$T_{M+1} = 7(M+2) \quad ,$$

$$T_{M+2} = 7(M+3) + 12 \quad , \qquad \text{and}$$

$$U_{M+1} = 12(M+2) \quad .$$

We can now solve our three recurrences. First, the recurrence

$$(N+1)U_{N+1} = N\ U_N + 24(N+1)$$

telescopes immediately to

$$(N+1)U_{N+1} = (M+1)U_{M+1} + 24 \sum_{M+1 \le k \le N} (k+1)$$

$$= 12(M+1)(M+2) + 12(N+1)(N+2) - 12(M+1)(M+2) \quad ,$$

or

$$U_N = 12(N+1) \quad .$$

Next, the recurrence

$$(N+1)T_{N+1} = (N+2)T_N + U_N$$

$$= (N+2)T_N + 12(N+1)$$

can be transformed by multiplying both sides by the summation factor $\frac{1}{(N+1)(N+2)}$ to a recurrence which immediately telescopes

$$\frac{T_{N+1}}{N+2} = \frac{T_N}{N+1} + \frac{12}{N+2}$$

$$= \frac{T_{M+1}}{M+2} + 12 \sum_{M+1 \le k \le N} \frac{1}{k+2}$$

to the solution

$$T_N = 7(N+1) + 12(N+1)(H_{N+1} - H_{M+2}) \quad .$$

Finally, the third recurrence

$$(N+1)C_{N+1} = (N-5)C_N + T_N$$

$$= (N-5)C_N + 7(N+1) + 12(N+1)H_{N+1} - H_{M+2})$$

requires a slightly more complicated summation factor (assuming $N \geq 6$): $\frac{1}{6!} N(N-1)(N-2)(N-3)(N-4)$. When both sides of the recurrence are multiplied by this expression, we get

$$\binom{N+1}{6} C_{N+1} = \binom{N}{6} C_N + 7\binom{N+1}{6} + 12\binom{N+1}{6}(H_{N+1} - H_{M+2}) \quad ,$$

which telescopes to

$$\binom{N+1}{6} C_{N+1} = \binom{M+1}{6} C_{M+1} + 7 \sum_{M+1 \leq k \leq N} \binom{k+1}{6}$$

$$+ 12 \sum_{M+1 \leq k \leq N} \binom{k+1}{6}(H_{k+1} - H_{M+2}) \quad .$$

We know that $C_{M+1} = M+2$, and the sums are easily evaluated (see Eqs. (19) and (23) in Appendix B), which leaves

$$\binom{N+1}{6} C_{N+1} = \binom{N+2}{7}\left(12H_{N+2} - 12H_{M+2} + \frac{37}{7}\right) + \frac{12}{7}\binom{M+2}{7} \quad .$$

Since $\binom{N+2}{7} = \frac{N+2}{7}\binom{N+1}{6}$, this leads us to our result: an exact equation for the average number of comparisons taken by Program 8.1:

$$C_N = \frac{12}{7}(N+1)(H_{N+1} - H_{M+2}) + \frac{37}{49}(N+1) + \frac{12}{7} \frac{\binom{M+2}{7}}{\binom{N}{6}} \quad .$$

It is very encouraging to end up with such a simple formula, despite the complex methods that were needed to derive it -- this simplicity can give us some confidence that our algorithm is indeed a "natural" way to sort.

Of course, the methods that we have used above have much more general applicability than the particular problem that we have just solved. In general, suppose that we are trying to find a formula for a sequence G_N which is represented by the generating function $G(z) = \sum_{N \geq 0} G_N z^N$. Suppose further that we succeed in showing through manipulations on the generating function that

$$(r_t-\Theta)(r_{t-1}-\Theta) \ldots (r_2-\Theta)(r_1-\Theta)G(z) = R(z) \quad ,$$

where Θ is the operator $x\frac{d}{dx}$; x is defined to be $1-z$; and $R(z)$ is some function describing a sequence R_N , $R(z) = \sum_{N \geq 0} R_N z^N$.

Then we can get an exact formula for G_N by defining

$$\begin{aligned} F_1(z) &= (r_1-\Theta)G(z) \\ F_2(z) &= (r_2-\Theta)F_1(z) = (r_2-\Theta)(r_1-\Theta)G(z) \\ &\vdots \\ F_t(z) &= (r_t-\Theta)F_{t-1}(z) = (r_t-\Theta)(r_{t-1}-\Theta)\ldots(r_1-\Theta)G(z) = R(z) \quad , \end{aligned}$$

with $F_j(z) \equiv \sum_{N \geq 0} F_{jN} z^N$ for $1 \leq j \leq t$; and then solving the t recurrences

$$
\begin{aligned}
(N+1)F_{(t-1)(N+1)} &= (N-r_t)F_{(t-1)N} + R_N \\
(N+1)F_{(t-2)(N+1)} &= (N-r_{t-1})F_{(t-2)N} + F_{(t-1)N} \\
&\vdots \\
(N+1)F_{1(N+1)} &= (N-r_2)F_{1N} + F_{2N} \\
(N+1)G_{N+1} &= (N-r_1)G_N + F_{1N} \quad .
\end{aligned}
$$

When the roots r_j $(1 \leq j \leq t)$ of the operator equation are integers, these recurrences are easy to solve. If $r_j = 0$ then the recurrence

$$(N+1)F_{(j-1)(N+1)} = (N-r_j)F_{(j-1)N} + F_{jN}$$

telescopes immediately; if r_j is a positive integer, the "summation factor" $\frac{1}{(r_j+1)!} N(N-1) \ldots (N-r_j+1)$ will reduce the recurrence to the telescoping recurrence

$$\binom{N+1}{r_j+1} F_{(j-1)(N+1)} = \binom{N}{r_j+1} F_{(j-1)N} + \frac{1}{r_j+1}\binom{N}{r_j} F_{jN} \quad ;$$

and if r_j is a negative integer, the "summation factor" $\frac{(-r_j-1)!}{(N-r_j)(N-r_j-1)\ldots(N+1)}$ will reduce it to the telescoping recurrence

$$\frac{F_{(j-1)(N+1)}}{\binom{N-r_j}{-r_j-1}} = \frac{F_{(j-1)N}}{\binom{N-r_j-1}{-r_j-1}} + \frac{F_{jN}}{(N+1)\binom{N-r_j}{-r_j-1}} \quad .$$

If the r_k are not integers, then the problem becomes much more difficult. We can always define a "summation factor" as above and reduce the recurrence to a sum, but we might not be able to explicitly evaluate the sum. For example, we might use the Gamma function

$$\Gamma(x) = \lim_{m\to\infty} \frac{m^x m!}{x(x+1)(x+2)\dots(x+m)} ,$$

an extension of the factorial function to real numbers, which has the property $\Gamma(x+1) = x\,\Gamma(x)$. (The function is not defined if x is a nonpositive integer.) Multiplying both sides of the recurrence by the "summation factor" $\frac{N!}{\Gamma(N-r_j+1)}$ we have

$$\frac{(N+1)!}{\Gamma(N-r_j+1)} F_{(j-1)(N+1)} = \frac{N!(N-r_j)}{\Gamma(N-r_j+1)} F_{(j-1)N} + \frac{N!}{\Gamma(N-r_j+1)} F_{jN}$$

$$= \frac{N!}{\Gamma(N-r_j)} F_{(j-1)N} + \frac{N!}{\Gamma(N-r_j+1)} F_{jN} ,$$

which telescopes to give the "solution"

$$F_{(j-1)N} = \frac{\Gamma(N-r_j)}{N!} \left(\frac{F_{(j-1)0}}{\Gamma(-r_j)} + \sum_{0\le k\le N-1} \frac{k!}{\Gamma(k-r_j+1)} F_{jk} \right) .$$

The sum involved in this solution might be very difficult to evaluate, depending on the value of r_j and the form of F_{jk} . This situation occurs, for example, if we try to find the average number of comparisons taken by a median-of-five quicksort.

We have already seen at least two examples of problems to which the above techniques could be applied. First, the very first recurrence that we solved in Chapter 3,

$$C_N = (N+1) + \frac{2}{N} \sum_{1 \le s \le N} C_{s-1} \quad ,$$

reduces, if we use the generating function $C(z) = \sum_{N \ge 0} C_N z^N$, to the differential equation

$$(1-z)C'(z) - 2C(z) = \frac{2}{(1-z)^2} \quad .$$

In terms of the operator Θ, this can be transformed as above to

$$(-2-\Theta)C(z) = \frac{2}{(1-z)^2} \quad ,$$

and the discussion in the last paragraph says that we should solve the recurrence

$$(N+1)C_{N+1} = (N+2)C_N + 2(N+1) \quad ,$$

which is exactly what we did in Chapter 3. We had a more complicated example in Chapter 5, when we were finding the average number of exchanges taken by the "two partition" Quicksort, Program 5.1. There we solved the equation

$$\binom{N}{2}B_N = 2\binom{N+1}{3} + 3 \sum_{0 \leq s \leq N-2} (N-s-1)B_s$$

by manipulating it (without motivation) into the two recurrences

$$(N+1)F_{N+1} = (N+2)F_N + 4(N+1)$$

$$(N+1)B_{N+1} = (N-3)B_N + F_N \quad ,$$

and solving these. Again, we can now see why this worked, because the application of generating functions to the original recurrence yields

$$(-2-\theta)(3-\theta)B(z) = \frac{4}{(1-z)^2}$$

and again the preceding paragraph leads us to the two recurrences that we solved in Chapter 5.

Returning to the question of completing the analysis of the average running time of Program 8.1, we find that we have to extend the above results slightly to allow us to work with the other quantities involved in the running time. We know from all the other analyses that we have done that we will need to solve a series of recurrences similar to the one that we have just solved for the average number of comparisons. Specifically, we need to be able to solve the recurrence

$$Y_N = y_N + 2 \sum_{1 \leq s \leq N} \frac{(N-s)(s-1)}{\binom{N}{3}} Y_{s-1} \qquad N > M \quad ,$$

where Y_N represents the average value of one of our quantities when Program 8.1 is run on a random permutation of $\{1,2,\ldots,N\}$, and y_N represents the average contribution of the first partitioning stage to the quantity. By tracing through the analysis above, using the generating function $Y(z) = \sum_{N \geq 0} Y_N z^N$, we find the recurrence transformed to

$$(-\Theta)(-2-\Theta)(5-\Theta)Y(z) = 6 \frac{(1-z)^3}{z^3} \sum_{N \geq 0} y_N \binom{N}{3} z^N$$

$$= 6 \sum_{N \geq 3} \Delta^3\left(y_N \binom{N}{3} \right) z^N \quad .$$

Continuing to follow the analysis above, we see that, to solve for $Y(z)$, we need only solve the three recurrences

$$(N+1)U_{N+1} = N U_N + 6\Delta^3\left(y_N \binom{N}{3} \right) \quad ,$$

$$(N+1)T_{N+1} = (N+2)T_N + U_N \quad , \quad \text{and}$$

$$(N+1)Y_{N+1} = (N-5)Y_N + T_N \qquad N > M \quad ,$$

where

$$T_{M+1} = (M+2)Y_{M+2} - (M-4)Y_{M+1} \quad ,$$

$$T_{M+2} = (M+3)Y_{M+3} - (M-3)Y_{M+2} \quad , \quad \text{and}$$

$$U_{M+1} = (M+2)T_{M+2} - (M+3)T_{M+1} \quad .$$

Above, for the number of comparisons, we had $y_N = N+1$, and $6\Delta^3\left((N+1)\binom{N}{3} \right) = 24\Delta^3\binom{N+1}{4} = 24(N+1)$ which leads directly to the recurrences that we solved. As another example, we might consider the number of partitioning stages. The solution to

$$A_N = 1 + \sum_{1 \le s \le N} \frac{(N-s)(s-1)}{\binom{N}{3}} A_{s-1} \qquad N > M \ , \quad A_N = 0 \quad N \le M \ ,$$

is found by solving the three recurrences

$$(N+1)U_{N+1} = N\,U_N + 6 \ ,$$

$$(N+1)T_{N+1} = (N+2)T_N + U_N \ , \quad \text{and}$$

$$(N+1)A_{N+1} = (N-5)A_N + T_N \qquad N > M \ ,$$

using the initial conditions $A_{M+1} = A_{M+2} = 1$ and $A_{M+3} = 1 + \frac{12}{(M+3)(M+2)}$, which implies that $T_{M+1} = 6$, $T_{M+2} = 6 + \frac{12}{M+2}$, and $U_{M+1} = 0$. Telescoping the recurrences in exactly the same way as before, we find that

$$U_N = 6$$

$$T_N = 12\,\frac{N+1}{M+2} - 6 \ , \quad \text{and}$$

$$A_N = \frac{12}{7}\,\frac{N+1}{M+2} - 1 + \frac{2}{7}\,\frac{\binom{M+1}{6}}{\binom{N}{6}} \ .$$

Before continuing with our analysis of the median-of-three Quicksort, we will consider an efficiently programmed version of the algorithm. Although this program is an obvious combination of Program 2.4 and Program 8.1, it deserves a place of its own as the most efficient general-purpose sorting method that is known.

Program 8.2

```
integer ℓ,r,p,i,j;
integer array stack[0::2 x f(N)-1];
arbmode array A[0::N+1];
arbmode v;
A[0] := -∞; A[N+1] := ∞; ℓ := 1; r := N;
p := 0:
loop until done:
     if A[ℓ] > A[r] then A[ℓ] :=: A[r] endif;                                                    A
     if A[(ℓ+r) ÷ 2] > A[r] then A[(ℓ+r) ÷ 2] :=: A[r] endif;
     if A[ℓ] > A[(ℓ+r) ÷ 2] then A[ℓ] :=: A[(ℓ+r) ÷ 2] endif;
     A[ℓ+1] :=: A[(ℓ+r) ÷ 2];
     i := ℓ+1; j := r; v := A[ℓ+1];
     loop:
          loop: i := i+1; while A[i] < v repeat;                                                 C'
          loop: j := j-1; while A[j] > v repeat;                                                C*-C'
     while i < j:
          A[i] :=: A[j];
     repeat;                                                                                     B
     A[ℓ] :=: A[j];
     if j-ℓ > r-j then if M ≥ j-ℓ then if p = 0 then done endif;
                                       p := p-2;
                                       ℓ := stack[p]; r := stack[p+1];
                                  else if r-j > M then stack[p] := ℓ; stack[p+1] := j-1;         S'
                                                 p := p+2; ℓ := j+1;
                                            else r := j-1;
                                       endif;
                         endif;
                    else if M ≥ r-j then if p = 0 then done endif;
                                       p := p-2;
                                       ℓ := stack[p]; r := stack[p+1];
                                  else if j-ℓ > M then stack[p] := j+1; stack[p+1] := r;        S-S'
                                                 p := p+2; r := j-1;
                                            else ℓ := j+1;
                                       endif;
                         endif;
     endif;
repeat;
i := 2;
loop while i ≤ N:
     if A[i] < A[i-1] then
          v := A[i]; j := i-1;                                                                   D
          loop:  A[j+1] := A[j]; j := j-1; while A[j] > v repeat;                                E
     endif;
     i := i+1;
repeat;
```

As with Program 2.4, we know that the running time of Program 8.2 depends on the six quantities

A -- the number of partitioning stages,

B -- the number of exchanges during partitioning,

C^* -- the number of comparisons during partitioning,

S -- the number of stack pushes,

D -- the number of insertions, and

E -- the number of keys moved during insertion.

The assembly language implementation of Program 2.4, which is given in Appendix A, requires a total average running time of

$$53\frac{1}{2}A_N + 11B_N + 4C_N^* + 3D_N + 8E_N + 9S_N + 7N \quad .$$

The running time depends on six quantities all of which have an average value of $\frac{1}{6}A_N$ -- each counts the number of times one of the six possible outcomes occurs for the tests that we added to find the median. This explains the non-integer coefficient for A_N in the expression above. It is important to notice that only the coefficient of A_N has changed from Program 2.4. This method does not add overhead to the inner loop.

We have already found that the average number of partitioning stages required by Program 8.2 in sorting a random permutation of $\{1,2,\ldots,N\}$ is

$$A_N = \frac{12}{7}\frac{N+1}{M+1} - 1 + \frac{2}{7}\frac{\binom{M+1}{6}}{\binom{N}{6}} \quad .$$

We have also found the average number of comparisons, but we must modify the answer slightly since Program 8.2 takes exactly N-1 comparisons for the first partitioning stage, not the N+1 that were assumed in the derivations above. (As always, by "comparisons", we mean "comparisons during partitioning" -- the comparisons used in determining the partitioning element are counted in the coefficient of the quantity A .) This means, by the linearity of our recurrences, that $C_N^* = C_N - 2A_N$, or

$$C_N^* = \frac{12}{7}(N+1)(H_{N+1} - H_{M+2}) + \frac{37}{49}(N+1) - \frac{24}{7}\;\frac{N+1}{M+1} + 2$$
$$+ \frac{4}{7}\;\frac{3\binom{M+2}{7} - \binom{M+1}{6}}{\binom{N}{6}} \quad .$$

To find the average number of exchanges taken by Program 8.2, we must first go through our usual manipulations to find the average number of exchanges used on the first partitioning stage. If s is the partitioning element, the number of exchanges is exactly the number of keys among $A[3], A[4], \ldots, A[s]$ which are $> s$. Averaging over all permutations of $\{1,2,\ldots,N\}$ we find that the average number of exchanges when s is the partitioning element is

$$\sum_{0 \le t \le s-2} t \frac{\binom{N-s-1}{t}\binom{s-2}{s-2-t}}{\binom{N-3}{s-2}} = \frac{(N-s-1)(s-2)}{N-3} \quad .$$

Averaging over all partitioning elements s , we find that the average number of exchanges used on the first partitioning stage is

$$\sum_{1 \le s \le N} \frac{(s-1)(N-s)}{\binom{N}{3}} \frac{(s-2)(N-s-1)}{N-3} = \frac{1}{\binom{N}{4}} \sum_{1 \le s \le N} \binom{s-1}{2}\binom{N-s}{2}$$

$$= \frac{\binom{N}{5}}{\binom{N}{4}}$$

$$= \frac{N-4}{5} \quad .$$

This means that the average number of exchanges required by Program 8.2 is described by the recurrence

$$B_N = \frac{N-4}{5} + 2 \sum_{1 \le s \le N} \frac{(s-1)(N-s)}{\binom{N}{3}} B_{s-1} \quad ,$$

and, by linearity, we now know that

$$B_N = \frac{1}{5} C_N - A_N \quad ,$$

or

$$B_N = \frac{12}{35}(N+1)(H_{N+1} - H_{M+2}) + \frac{37}{245}(N+1) - \frac{12}{7}\frac{N+1}{M+2} + 1$$

$$+ \frac{2}{7} \frac{\frac{6}{5}\binom{M+2}{7} - \binom{M+1}{6}}{\binom{N}{6}} \quad .$$

Rather than carry through the terms with $\binom{N}{6}$ in the denominator, we will ignore them from now on, and therefore derive our answer to within $O(N^{-6})$. This is so very small for the value of N in the range of interest that we can still think of our answers as "exact".

The solution for the average number of stack pushes S_N , requires similar calculations, and we will omit the details. The average number of stack pushes on the first partitioning stage is

$$\frac{1}{\binom{N}{3}} \sum_{M+2 \leq s \leq N-M-1} (s-1)(N-s) = 1 - 6\frac{\binom{M+1}{2}}{\binom{N}{2}} + 4\frac{\binom{M+1}{3}}{\binom{N}{3}}$$

and the recurrence

$$S_N = 1 - 6\frac{\binom{M+1}{2}}{\binom{N}{2}} + 4\frac{\binom{M+1}{3}}{\binom{N}{3}} + 2\sum_{1 \leq s \leq N} \frac{(s-1)(N-s)}{\binom{N}{3}} S_{s-1} ,$$

$$N > 2M+2 \quad ; \quad S_N = 0 \quad , \quad N \leq 2M+2 \quad ;$$

has three components, each of which can be easily solved using the methods developed above. It turns out that

$$S_N = \frac{3}{7}(N+1)\frac{5M+3}{(2M+3)(2M+1)} - 1 + O(N^{-6}) \qquad .$$

It remains only to determine the contribution of the insertion sorting to the total running time of Program 8.2. As in Chapter 5, we find that partitioning does not contribute to these quantities, so we need to solve the recurrences

$$D_N = \begin{cases} 2\displaystyle\sum_{1 \leq s \leq N} \frac{(N-s)(s-1)}{\binom{N}{3}} D_{s-1} & N > M \\ \\ N - H_N & N \leq M \end{cases}$$

and

$$E_N = \begin{cases} 2 \sum\limits_{1 \le s \le N} \dfrac{(N-s)(s-1)}{\binom{N}{3}} E_{s-1} & N > M \\ \\ \dfrac{N(N-1)}{4} & N \le M \quad . \end{cases}$$

Again, we will omit the calculations. The methods described above can be applied directly to yield the answers

$$D_N = (N+1) - \frac{4}{7} \frac{N+1}{M+2} (3H_{M+1} - 1) + O(N^{-6}) \qquad \text{and}$$

$$E_N = \frac{1}{35} (N+1)(6M-17) + \frac{6}{7} \frac{N+1}{M+2} + O(N^{-6}) \quad .$$

We have now found the average values of all of the quantities upon which the running time of Program 8.2 depends, all to within $O(N^{-6})$. To summarize, we know that the median-of-three program requires, on the average,

$$A_N = \frac{12}{7} \frac{N+1}{M+2} - 1 \qquad \text{stages,}$$

$$B_N = \frac{12}{35} (N+1)(H_{N+1} - H_{M+2}) + \frac{37}{245} (N+1) - \frac{12}{7} \frac{N+1}{M+2} + 1 \qquad \text{exchanges,}$$

$$C^*_N = \frac{12}{7} (N+1)(H_{N+1} - H_{M+2}) + \frac{37}{49} (N+1) - \frac{24}{7} \frac{N+1}{M+2} + 2 \qquad \text{comparisons,}$$

$$D_N = (N+1) - \frac{4}{7} \frac{N+1}{M+2} (3H_{M+1} - 1) \qquad \text{insertions,}$$

$$E_N = \frac{1}{35} (N+1)(6M-17) + \frac{6}{7} \frac{N+1}{M+2} \qquad \text{moves during insertion, and}$$

$$S_N = \frac{3}{7} (N+1) \frac{5M+3}{(2M+3)(2M+1)} - 1 \qquad \text{stack pushes.}$$

These formulas are all accurate to within $O(N^{-6})$. As in Chapter 3 we will avoid the details introduced by small N by assuming $N > 2M+2$. Substituting these into the expression that we have for the total running time gives the formula

$$\frac{372}{35}(N+1)H_{N+1} - \frac{111}{2}$$

$$+ (N+1)\left(\frac{529}{49} + \frac{450}{7(M+2)} - \frac{372}{35} H_{M+2} - \frac{36}{7}\frac{H_{M+1}}{M+2} + \frac{48}{35} M + \frac{27}{7}\frac{5M+3}{(2M+3)(2M+1)} \right)$$

for the expected running time of the median-of-three Quicksort.

Continuing as in Chapter 3, we can find the best value of the parameter M by considering the function

$$f(M) = \frac{529}{49} + \frac{450}{7(M+2)} - \frac{372}{35} H_{M+2} - \frac{36}{7}\frac{H_{M+1}}{M+2} + \frac{48}{35} M + \frac{27}{7}\frac{5M+3}{(2M+3)(2M+1)} \quad .$$

The graph of this function is shown in Figure 8.1 -- it is flatter than the corresponding graph for normal quicksort, and it takes on its minimum at $M = 9$ (although $M = 10$ is very close). The flatness means that the precise choice of M is not as critical for this method, as we would expect. With the choice of $M = 9$, the approximate formula for the total running time is

$$10.63(N+1) \ln N + 2.11N - 70.68$$

or

$$7.37(N+1) \lg N + 2.11N - 70.68 \quad ,$$

which shows that the median-of-three Quicksort (Program 8.2) is significantly faster than normal Quicksort (Program 2.4) which has an approximate running time of

$$11.67(N+1) \ln N - 1.74N - 18.74$$

or

$$8.09(N+1) \lg N - 1.74N - 18.74$$

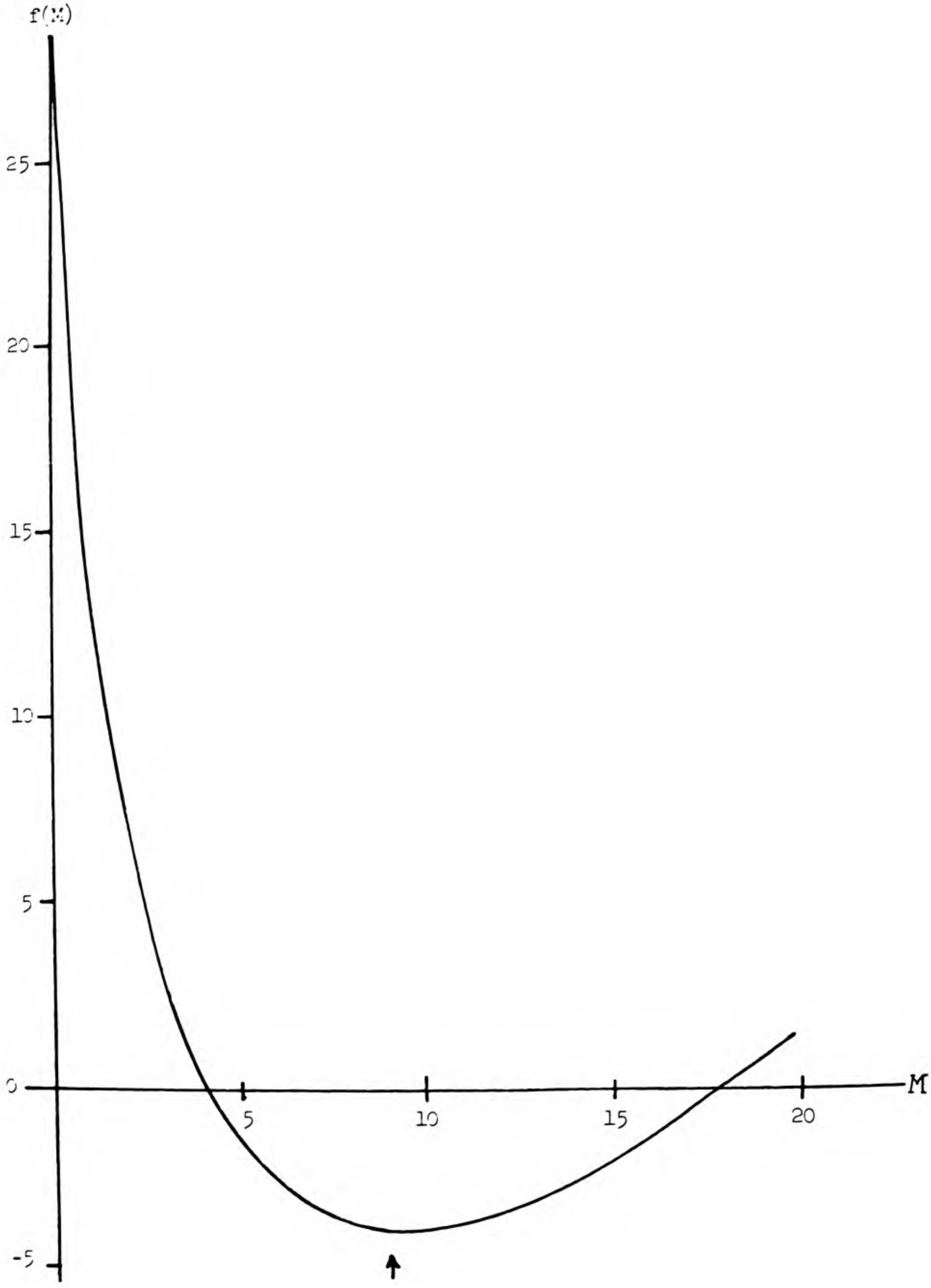
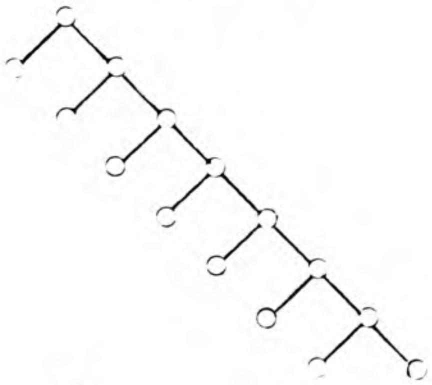


Figure 8.1. Contribution of M .

As we have already commented, the median-of-three method results in about a 9% improvement in total average running time for large N .

In fact, for most files in a practical size range, Program 8.2 has a lower total average running time than any other sorting method known which does not use extra space proportional to the size of the file. We have shown it to be the fastest that we have considered for the particular machine used for the assembly language programs in Appendix A. Such comparative studies will not vary too much from machine to machine -- the analysis for another machine simply consists of attaching different coefficients to the average values that we have derived for the quantities upon which the running time depends. Notwithstanding, we know that any sorting method (based on comparisons) must take at least N lg N comparisons, and that the coefficient of N lg N in the total running time of Program 8.2 is $\frac{372}{35 \lg e} \approx 7.37$; if another method is to be more efficient, it must incur an overhead of less than 8 memory accesses per comparison for data and instructions in the inner loop. It would be very surprising indeed if there were a faster sorting method than Program 8.2, because of the very efficient inner loop inherent in Quicksort. If the technique of "unwrapping the inner loop" is used in the implementation, Program 8.2 can be made even more efficient (see Appendix A), and the coefficient of N lg N in the total running time can be reduced to about 6.63 .

The median-of-three method does still have a worst case running time of $O(N^2)$, because it is still possible for a degenerate partitioning tree such as



to occur. (The above tree occurs, for example, when Program 8.2 (with $M = 0$) is used to sort the permutation

1 11 3 9 5 13 7 2 4 6 8 10 12 14 15 .)

However, such worst case performance is extremely unlikely to occur because of the way the middle element of the file is used in making the partitioning decision. If it is known that there are extreme biases in the file, one of the techniques discussed in Chapter 5 could be used to choose some element other than the middle element, so that the worst case would be even less likely. This is probably being overcautious, however, since the fact that three elements are used in the partitioning decision makes the worst case very much less likely here than in Chapter 5. In any case, the $O(N^2)$ worst case should not be a deterrent to using the median-of-three Quicksort in a practical situation. We use algorithms all the time (such as hashing, for example) which have a horrible worst case that is very unlikely to occur. It takes only a slight amount of confidence in the laws of probability to use Program 8.2 as the most efficient sorting method available in almost any practical situation.

The median-of-three method is also significant from the standpoint of analysis. Although we had to resort to a variety of interesting mathematical techniques, we were able to derive exact formulas describing the average running time of the algorithm. This combination of a program of practical importance giving rise to interesting mathematical problems which can be completely solved is rare in the analysis of algorithms. We could not hope for a more satisfying conclusion to our study of the Quicksort algorithm and the methods of analyzing it.

CHAPTER NINE

The history of the published literature on Quicksort comprises a curious story, which illustrates the development of our understanding of the difference between an algorithm and its implementation as a program. Hoare understood this difference very clearly: In 1961 he published the first description of Quicksort in the Algorithms section of the Communications of the Association for Computing Machinery [12] with a caution that "suitable refinements of the method will be desirable for its implementation on any actual computer". This was a recursive description of the algorithm in ALGOL 60, with the partitioning value chosen at random from within the array. He then published a comprehensive article dealing with the implementation of the algorithm in The Computer Journal in Britain in 1962 [13]. This article contains an extraordinary number of ideas, and is a model of clarity and conciseness. In just over five pages Hoare anticipates nearly all of the future "developments" related to Quicksort. He dicusses the use of an explicit stack for the purpose of sorting short subfiles first; the idea of treating small subfiles separately; other methods for choosing the partitioning element such as the median of a sample; and the use of "sentinels" (∞ keys) to remove the pointer test from the inner loop. Also, he presents a nearly complete analysis of the total running time of the algorithm. (Many of these ideas were apparently discovered independently in an influential paper written by T. N. Hibbard in 1962 [10].) In a most unfortunate development, this article went virtually unnoticed and authors began publishing their "suitable refinements" to the implementation of Hoare's Comm. ACM algorithm as "new" algorithms similar to Quicksort [2,10,32]. Not until 1969 did all of Hoare's ideas appear in a published implementation [34]. We will discuss this in more detail below during a chapter by chapter account of the literature related to the topics we have discussed in the text. We often find ourselves referring back to "Hoare's original article" in The Computer Journal, and it should be read by anyone interested in the development of the Quicksort algorithm.

Equally important as a source for the material that we have covered is D. E. Knuth's The Art of Computer Programming [16,17,19]. The vast wealth of topics covered in these books includes the study of Quicksort [19, Section 5.2.2] and the techniques which have been used in the analysis of computer algorithms (see the headings under "Analysis of Algorithms" in the indices of the books). The style and methods of analysis that we have used have been inspired by these books.

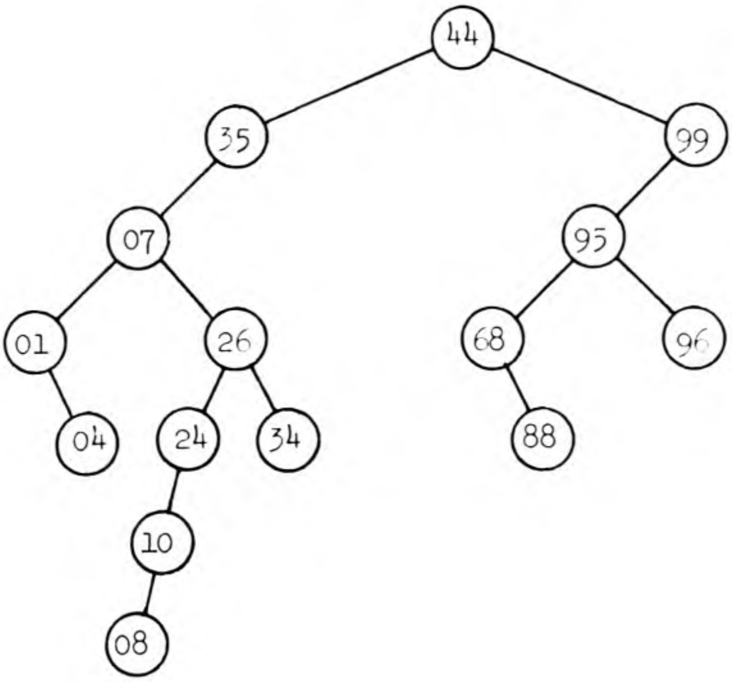
The insertion sorting method in Chapter 1 is based on an extremely simple and natural idea, and its origins are difficult to trace. In fact, the improvement mentioned at the end of Chapter 1, binary insertion, appeared in the very first published account of computer sorting in 1946 [25]. Insertion sorting has also been called a "sifting" or "sinking" sort, because large keys are "sifted" out and "sink" to their proper position in the file. It is important in other sorting methods besides Quicksort, most notably Shell's diminishing increment method [33]. The analysis of the method is largely elementary and it has always been known to require $O(N^2)$ comparisons, but the specific approach taken in Chapter 1 in developing exact formulas for the moments of the total running time is adapted from Knuth ([16], Section 1.2.10 and [19], Sections 5.1.1, 5.2.1).

The Quicksort algorithm itself has been published often as an example in texts and papers on sorting and computer programming, and it would be futile to attempt to catalog all such references here.

Unfortunately, nearly every published implementation seems to use a different partitioning method, both because it is inconvenient to express the algorithm in many programming languages and because authors have differing concepts of the tradeoff between elegance and efficiency in computer programs (see [21]). The methods that have been published can generally be classed as being similar to one of the Partitioning Methods 2.1, 2.2, or 2.3. The method of "filling the hole" (Partitioning Method 2.1) was mentioned by Hoare (as a possibility for machines on which exchanges are inconvenient) and was adapted by Dijkstra [4] and Knuth [19,21]. This method lends itself to a clear English description of the algorithm, and these authors use Quicksort as an example in the "art" of computer programming. As mentioned in Chapter 2, Partitioning Method 2.2 was the very first method to be published, in Hoare's original paper. The idea to stop the scans on keys equal to the partitioning element, as in Partitioning Method 2.3, is due to Singleton [34]. These methods lend themselves to a concise formal description, and they are often used as examples in work on proving programs correct (for example, [14]). The fact that exchanging equal keys can lead to non-random subfiles (and inefficient operation) was not noticed until 1974, by Knuth [21]. Other published methods are generally less carefully implemented than these, or involve concessions to some particular programming language. None of the methods are as simply expressed or as efficient as Partitioning Method 2.4, which was inspired from the treatment of Quicksort in an early version of [21]. A similar method is used by Floyd and Rivest [5] in their SELECT algorithm.

Much of the analysis of the total running time in Chapter 3 is found in Hoare's original paper, though he stops short of finding the coefficients for a particular implementation and studying the best choice of the cutoff for small subfiles. The approach taken in Chapter 3 is from Knuth [19, Section 5.2.2]. The problem was also studied by P. Windley in 1960 [37], and he also presented (but did not solve) the recurrence for the variance of the distribution describing the number of comparisons. This recurrence was solved by Knuth [19, Ex. 6.2.2-8] and the solution was rediscovered by E. Palmer, M. Rahimi and R. Robinson [27]. Of course, all of these analyses begin by showing that the subfiles after partitioning are random. A precise formulation of the approach used to prove this, which has been called the "Principle of Conservation of Ignorance" by L. Guibas, is given by T. Porter and I. Simon [28].

A careful reader might wonder how Windley came to discover the recurrences for the number of comparisons in 1960, before Quicksort was even invented! The answer is that there is an interesting duality between the analysis of Quicksort and the study of binary tree searching, an important search technique discovered in the 1950's (see [19], Section 6.2.2). As an example, consider the binary tree (from Chapter 2) corresponding to the operation of Program 2.2 on one sample set of fifteen keys:



Suppose that we need to know whether or not a given key is in this tree. We can "search" for it by comparing it with the root; then moving to the left subtree if it is smaller and to the right if it is greater; then continuing in the same manner until we either encounter the desired key or come to a null subtree. To put a new key into such a tree, we simply insert it in place of the null subtree which terminates the unsuccessful search for the key. For example, the tree above might have been constructed by starting with a null tree and inserting the keys in the order

44 35 07 99 01 26 24 95 10 68 08 88 34 96 04 ,

and many other orders lead to the same tree. A natural question which arises is the derivation of the average number of comparisons used when a random permutation of $\{1,2,\ldots,N\}$ is used to build a tree in this way. (This is N times the average number of comparisons needed to find a key which is known to be in the tree.) Now, after the first key is inserted as the root, each of the other N-1 keys must be compared with it, and the two subtrees are constructed independently. Therefore, if g_{Nk} is the probability that exactly k comparisons are used, and $G_N(z) = \sum_{k \geq 0} g_{Nk} z^k$ is the associated generating function, we have

$$G_N(z) = \begin{cases} \frac{1}{N} z^{N-1} \sum_{1 \leq s \leq N} G_{s-1}(z) G_{N-s}(z) & N > 0 \\ 1 & N = 0 \end{cases}$$

which is nearly the same as the recurrence that we derived for $C_N(z)$ (with $M = 0$) at the end of Chapter 2. In fact $G_N(z) = z^2 C_N(z)$, so the mean is 2 greater and the variance is the same. This correspondence between these two algorithms is at once both obvious and mystifying. The mathematics appears in the literature in both disguises. In addition, binary tree searching gives rise to some problems which do not arise in the study of Quicksort. For example, W. C. Lynch [23] studies the moments of the distribution for the average number of comparisons required to insert a new key into a binary search tree. A general treatment of this topic is given by Knuth ([19], Section 6.2.2, especially Exercises 6-8).

The usual practice in studying the best case and worst case of Quicksort is to consider only the best case and worst case for the number of comparisons (with $M = 0$) . This makes the correspondence to binary tree structures immediate and the result for the worst case obvious. Also, the best case analysis then corresponds to the establishment of a lower bound for the average number of comparisons required by <u>any</u> sorting method (see the work by Morris [26]). The results given in Chapter 4 are of course very specific to the Quicksort problem, but similar recurrences arise frequently. (For example, Knuth solves the recurrence $f(1) = 0$, $f(n) = (\max_{1 \le k < n} (\min(k , n-k) + f(k) + f(n-k))$ in connection with an in-situ permutation algorithm in [18].) A general treatment of similar recurrence relations can be found in Fredman and Knuth [9].

Although we saw in Chapter 5 that the suggestion in Hoare's original algorithm to use a random element as a partitioning element is sound, the idea was not well received in "practical" implementations of the algorithm because of the expense involved. This is doubtless due to the fact that most implementations (exceptions are Hibbard [10], Singleton [34], and Knuth [19]) in the literature use $M = 0$, 1 or 2 , and the expense of calculating a random number at each partitioning stage is intolerable under these circumstances. Both Scowen [32] and Boothroyd [2] use the middle element instead. As mentioned in Chapter 5, the effect of such minor perturbations on the analysis has received scant attention in the literature. The interesting asymptotic analysis of Partitioning Method 2.3 with a random partitioning element given in Chapter 5 is modeled on a derivation given by Knuth [20], when a similar recurrence (which arose in connection with a paging algorithm) appeared on a Stanford Ph.D. Qualifying Examination in Analysis of Algorithms.

The general assumption in the analysis that the keys being sorted are distinct is not limited to Quicksort. It is difficult to analyze the effect of equal keys on nearly all sorting methods with the possible exception of radix methods. Knuth ([19], Section 5.1.2) derives exact formulas for the average and standard deviation of the number of inversions of a random permutation of a "multiset", and this could be used to extend the analysis of insertion sorting to the case when equal keys are present. But the derivation is difficult, and analyses of more complex sorting methods always assume distinct keys. From a practical standpoint, the author of a Quicksort program is of course forced to worry about equal keys. Most follow Hoare's original method, and scan over equal keys. Rich [29] achieves some saving by testing for the case that all keys are equal, and many authors (for example, Aho, Hopcroft, and Ullman [1]) choose the asymmetric implementation of putting all keys equal to the partitioning element into one of the subfiles. None give any qualitative indication of the effects of their strategy, and the analysis in Chapter 5 indicates that these methods are all less desirable than the technique of stopping both scans on keys equal to the partitioning element, which is due to Singleton [34]. A stable version of Quicksort, which is asymptotically efficient but not really practical, has been suggested by Rivest [30].

The history of van Emden's modification adds another curious twist to the story of the literature on Quicksort. In his original articles in 1970 [35,36], van Emden worked with continuous approximations to the situation to show that the average number of comparisons in his method is about $1.14\ N \lg N \approx 1.65\ N \ln N$. This derivation incorrectly assumed that the subfiles after partitioning are random, and so may not be very accurate. Even if it were, the method would not compete (as we saw in Chapter 6) because the analysis doesn't count the comparisons and other overhead needed to maintain the partitioning bounds. Van Emden observed a 15% saving in computing time, but he didn't publish both of the programs he compared, and later empirical studies by Rich [29] and Loeser [22] among others showed the method to be as much as 20% slower than even normal Quicksort. But it is easy to believe that it should run faster, and van Emden's was for a time the method of choice (see for example [24]) to be used to improve Quicksort, even though it results in longer and slower programs, produced at extra effort.

The result that we derived in Chapter 7 for the average number of comparisons used by samplesort matches that found in Frazer and McKellar's original articles [7 , 8], but the optimal sample sizes differ because Frazer and McKellar went to a continuous approximation to the function before minimizing. The asymptotic derivation that we followed also led to a much simpler proof of the asymptotic efficiency of samplesort. Frazer and McKellar considered only comparisons and they recognized that convergence to the asymptotic minimum is slow. They did not attempt to compare their method with fixed sample size partitioning.

The most effective modification, the idea of using the median of a small sample as the partitioning element, also was suggested in Hoare's original paper. Hoare did not pursue the idea, since he found it "very difficult to estimate the saving which would be achieved by this", and he went on to consider efficient methods of implementing the algorithm. (As well he should: an implementation which tests for the pointers crossing in the inner loop (as many do) will be up to twice as slow as the methods we have seen; sampling only saves about 8%.) The idea was not rediscovered until 1969, when Singleton [34] published a careful implementation of Quicksort which involved partitioning on the median of the first, middle, and last keys. (Another, less efficient, implementation of Singleton's method is given by R. B. Sander-Cederlof [31].) This implementation also introduced the idea of stopping the scans on keys equal to the partitioning element, and proposed using insertion sort for subfiles of 11 or fewer elements. The only problem with this algorithm is that the median element is left in the middle of the file, and so the subfiles after partitioning are not random. (The effects of this are not nearly as bad as for Partitioning Method 2.3 since three elements participate in the choice of partitioning element.) A very important feature of Singleton's algorithm from a practical standpoint is that it is the only one of the published algorithms [2,10,12,29,31,36] that doesn't test for the pointers crossing in the inner loop. This is a perfect example of misdirected effort in optimizing a program. After presenting logical changes to Quicksort designed to reduce the average number of comparisons by 10 or 20%, or changes to the implementation (such as removing recursion) designed to improve the "efficiency", other articles then include programs which can be made to run up to twice as fast by simple changes to the inner loop.

Singleton presented only empirical evidence in support of his implementation, and didn't attempt to derive the average number of comparisons or the average running time. The coefficient $\frac{1}{H_{2t+2} - H_{t+1}}$ of $N \ln N$ for the average number of comparisons when partitioning on the median of $2t+1$ elements was in fact derived by van Emden [35] as an application of his continuous approximation to the analysis of partitioning. A similar asymptotic derivation is given by Hurwitz [15], who also indicates that a larger value of t may give rise to a smaller variance, as we might expect. The method of finding the exact formula for the total running time in Chapter 8 is based on the solution given by Knuth [19], but Knuth was unable to obtain a complete exact formula for the total running time because of complications introduced by the partitioning method that he used. Knuth's partitioning method is the one described near the beginning of Chapter 8 (exchange the median element with $A[\ell]$); the anomaly described there was discovered by D. B. Coldrick (see [19], Ex. 5.2.2-55).

There are a few issues relating to Quicksort in the literature which we haven't yet treated, mainly related to applications which don't fit our basic ground rules. If the records to be sorted are more than one word then the coefficients of B and C in the total running time will be larger, and improvements that we have seen which rely on coding efficiency in the inner loop may assume relatively less importance than substantive improvements to the algorithm. A typical situation is to have multiword records sorted according to the values of single word keys within the records. In this case exchanges become relatively more expensive than comparisons. If the keys themselves are more than one word, P. Shackleton (see [13]) suggests making comparisons more efficient essentially by storing a counter in the stack with each subfile which tells how many leading words of the keys in the subfile are known to be identical.

If there are so many records that they don't fit into memory all at once, then Quicksort may not be appropriate, since this was an implicit assumption in our analysis: Quicksort is an "internal" sorting method. There are many "external" methods which are designed to sort large files which can't possibly fit into memory. Many of these have "internal" phases for which Quicksort may be used. Knuth ([19], Section 5.4.8) describes an external method based on the idea of partitioning, but there are many better methods, mostly based on merging. However, if the sorting program is run in a paging environment, then B. Brawn, F. Gustavson, and E. Mankin [3] have shown that Quicksort is perfectly acceptable. This was actually anticipated by Hoare, and it is to be expected, since the program has only two slowly changing "localities", those containing the scanning pointers.

In his original presentation of the algorithm [12] Hoare showed that partitioning can be used for other applications besides sorting, by including with Quicksort a program to find the k-th smallest of a set of N elements. The idea is to partition the array $A[1],\ldots,A[N]$ so that $A[k]$ is in position; all the keys to the left of $A[k]$ are $\leq A[k]$ and all the keys to the right of $A[k]$ are $\geq A[k]$. First we put $A[s]$ into position using our normal partitioning method. If $s = k$, then we are done, otherwise if $k < s$ we need to work on $A[1],\ldots,A[s-1]$ and if $k > s$ we need to work on $A[s+1],\ldots,A[N]$. This leads us to

Program 9.1

```
procedure find (integer value ℓ,r,k);
     loop
          i := ℓ; j := r+1; v := A[ℓ];
          loop:
               loop: i := i+1; while A[i] < v repeat;
               loop: j := j-1; while A[j] > v repeat;
          while i < j:
               A[i] :=: A[j];
          repeat;
          A[ℓ] :=: A[j];
     while k ≠ j:
          if k < j then r := j-1; else ℓ := j+1; endif;
     repeat;
```

Hoare originally stated the procedure recursively (e.g., " if k < j then find(ℓ , j-1) else find(j+1 , r) endif; ") but this is unnecessary, and he did not do so in a later implementation [14]. If the program is to be used more than once on the same file, then it is best to use a random partitioning element.

It might seem that this program would be easier to analyze than Quicksort, but it is in fact much more difficult. If $k < s$, then we are finding the k-th smallest of $A[1],\ldots,A[s-1]$, and if $k > s$ we are finding the $(k-s)$-th smallest of $A[s+1],\ldots,A[N]$, so that we have the recurrence

$$C_{Nk} = N+1+\frac{1}{N}\sum_{1\le s<k} C_{(N-s)(k-s)} + \frac{1}{N}\sum_{k<s\le N} C_{(s-1)k} \quad \text{for} \quad 1 \le k \le N ,$$

describing the average number of comparisons required to find the k-th smallest of N randomly ordered distinct elements. We can now multiply by $N z^N w^k$ and sum over N and k to get (eventually)

$$\sum_{k\geq 1} \sum_{N\geq k} N C_{Nk} z^N w^k = \frac{w}{1-w}\left(\frac{2z}{(1-z)^3} - \frac{2wz}{(1-wz)^3}\right)$$

$$+ \frac{wz}{1-wz} \sum_{k\geq 1} \sum_{N\geq k} C_{Nk} z^N w^k$$

$$+ \frac{z}{1-z} \sum_{k\geq 1} \sum_{N\geq k} C_{Nk} z^N w^k \quad .$$

We can rewrite this in terms of the generating function $C(z,w) = \sum_{k\geq 1} \sum_{N\geq k} C_{Nk} z^N w^k$ and rearrange terms slightly to get the differential equation

$$\frac{d}{dz} C(z,w) - \left(\frac{1}{1-z} + \frac{w}{1-wz}\right) C(z,w) = \frac{2w}{1-w}\left(\frac{1}{(1-z)^3} - \frac{w}{(1-wz)^3}\right)$$

which has the solution

$$(1-z)(1-zw)C(z,w) = 2\left(\frac{w}{1-z} + \frac{1}{1-zw} - \frac{w^2}{1-w}\ln(1-z) + \frac{1}{1-w}\ln(1-zw) - w - 1\right).$$

We can now carefully expand each of these terms into power series, then multiply by $\frac{1}{1-z}$ (which corresponds to a sum on one index) and by $\frac{1}{1-zw}$ (which corresponds to a sum on both indices) to eventually get the final answer

$$C_{Nk} = 2((N+1)H_N - (k+1)H_k - (N-k+2)H_{N-k+1} + N + 2) \quad .$$

(Knuth [18] gives a solution to this problem which involves substitution and reduction to telescoping recurrences.) In particular, if $k = \left\lfloor \frac{N+1}{2} \right\rfloor$, then we find that the average number of comparisons used to find the median of N elements is asymptotically $2(\ln 2 + 1)N + O(1) \approx 3.386\,N$. We might also be interested in knowing how much this could be improved by a more intelligent choice of the partitioning element, for example the median-of-three method. On the theory that it is better to end with an interesting question than to end with a series of answers, this problem is left for the energetic reader. Floyd and Rivest have shown [5] that the idea of sampling can lead to dramatic improvements in this process, but their method is analogous to samplesort; and a method based on fixed size samples would probably do better.

Most of the topics included in this brief survey of the literature on Quicksort are covered in the previous eight chapters, but we have progressed to the point where we can express the algorithms more succintly and analyze them more completely. The algorithms presented are more efficient than those referred to above, and this has been justified (and even suggested) by complete and exact analysis of the algorithms. Such analysis has received little attention so far in the literature outside of Knuth's books [16,17,19], and this is indeed unfortunate. The Quicksort algorithm is better understood through analysis, and the analysis is very interesting in its own right. The many variations of the algorithm lead to much more spectacular variations in the analysis, and it is this combination of algorithm and analysis that makes the study of Quicksort so fascinating.

References

[1] Aho, A. V.; Hopcroft, J. E.; and Ullman, J. D. The Design and Analysis of Computer Algorithms, Addison-Wesley, 1974.

[2] Boothroyd, J. "Sort of a section of the elements of an array by determining the rank of each element (Algorithm 25); and ordering the subscripts of an array section according to the magnitudes of the elements (Algorithm 26)." Computer J. 10 (Nov. 1967), 308-310. (See notes by R. S. Scowen in Computer J. 12 (Nov. 1969), 408-409, and by A. D. Woodall in Computer J. 13 (Aug. 1970).

[3] Brawn, B. S.; Gustavson, F. G.; and Mankin, E. "Sorting in a paging environment." Comm. ACM 13, 8 (Aug. 1970), 483-494.

[4] Dijkstra, E. W. "EWD316: A short introduction to the art of programming." Technical University Eindhoven (Aug. 1971), 97 pp.

[5] Floyd, R. W.; and Rivest, R. L. "Expected time bounds for selection." Comm. ACM 18, 3 (Mar. 1975), 165-173.

[6] Foley, M.; and Hoare, C. A. R. "Proof of a recursive program: Quicksort." Computer J. 14, 4 (Nov. 1971), 391-395.

[7] Frazer, W. D.; and McKellar, A. C. "Samplesort: A sampling approach to minimal storage tree sorting." Proc. Third Annual Princeton Conf. on Information Sciences and Systems (1969), 276-280.

[8] Frazer, W. D.; and McKellar, A. C. "Samplesort: A sampling approach to minimal storage tree sorting." J. ACM 17 (1970), 496-507.

[9] Fredman, M.; and Knuth, D. E. "Recurrence relations based on minimization." J. Math. Analysis and Applications 48, 2 (Nov. 1974), 534-559.

[10] Hibbard, T. N. "Some combinatorial properties of certain trees with applications to searching and sorting." J. ACM 9, 1 (Jan. 1962), 13-1

[11] Hibbard, T. N. "An empirical study of minimal storage sorting." Comm. ACM 6, 5 (May 1963), 206-213.

[12] Hoare, C. A. R. "Partition (Algorithm 63), Quicksort (Algorithm 64), and Find (Algorithm 65)." Comm. ACM 4, 7 (July 1961), 321-322. (See also certification by J. S. Hillmore in Comm. ACM 5, 8 (Aug. 1962), 439, and by B. Randell and L. J. Russell in Comm. ACM 6, 8 (Aug. 1963), 446.)

[13] Hoare, C. A. R. "Quicksort." *Computer J.* 5, 4 (April 1962), 10-15.

[14] Hoare, C. A. R. "Proof of a Program: FIND." *Comm. ACM* 14, 1 (Jan. 1971), 39-45.

[15] Hurwitz, H. "On the probability distribution of the values of binary trees." *Comm. ACM* 14, 2 (Feb. 1971), 99-102.

[16] Knuth, D. E. Fundamental Algorithms, *The Art of Computer Programming* 1. Addison-Wesley, 1968.

[17] Knuth, D. E. Seminumerical Algorithms, *The Art of Computer Programming* 2. Addison-Wesley, 1969.

[18] Knuth, D. E. "Mathematical analysis of algorithms." Proc. IFIP Cong. 1971, North-Holland Publishing Co., Amsterdam, 1972, 19-27.

[19] Knuth, D. E. Sorting and Searching, *The Art of Computer Programming* 3. Addison-Wesley, 1972.

[20] Knuth, D. E. "Analysis of algorithms qualifying examination, 1974." Stanford University Computer Science Department, May 1974.

[21] Knuth, D. E. "Structured programming with go to statements." *Computing Surveys* 6, 4 (Dec. 1974), 261-301.

[22] Loeser, R. "Some performance tests of "quicksort" and descendants." *Comm. ACM* 17, 3 (Mar. 1974), 143-152.

[23] Lynch, W. C. "More combinatorial properties of certain trees." *Computer J.* 7 (Oct. 1964), 299-302.

[24] Martin, W. A. "Sorting." *Computing Surveys* 3, 4 (Dec. 1971), 147-174.

[25] Mauchly, J. W. "Sorting and collating." (Lecture delivered July 25, 1946.) Theory and techniques for the design of electronic digital computers III. Moore School Report 48-9, University of Pennsylvania (June 1948), Lecture 22.

[26] Morris, R. "Some theorems on sorting." *SIAM Journal of Appl. Math.* 17, 1 (Jan. 1969), 1-6.

[27] Palmer, E. M.; Rahimi, M. A.; and Robinson, R. W. "Efficiency of a binary comparison storage technique." *J. ACM* 21, 3 (1974), 376-384.

[28] Porter, T.; and Simon, I. "Random insertion into a priority queue structure." Computer Science Report STAN-CS-74-460, Stanford University (Oct. 1974), 25 pp.

[29] Rich, R. P. Internal Sorting Methods Illustrated with PL/I Programs, Prentice-Hall, 1972.

[30] Rivest, R. L. "A fast stable minimum-storage sorting algorithm." Institut de Recherche d'Informatique et d'Automatique Rapport 43, Dec. 1973.

[31] Sander-Cederlof, R. B. "Supersort: high speed sort routine for the Control Data 3300." Proceedings of Focus-3 Conference, St. Paul (May 1970), Control Data Company, Palo Alto, California.

[32] Scowen, R. S. "Quickersort (Algorithm 271)." Comm. ACM 8, 11 (Nov. 1965), 669-670. (See also certification by C. R. Blair in Comm. ACM 9, 5 (May 1966), 354.

[33] Shell, D. L. "A high-speed sorting procedure." Comm. ACM 2, 7 (July 1959), 30-32.

[34] Singleton, Richard C. "An efficient algorithm for sorting with minimal storage: Algorithm 347." Comm. ACM 12, 3 (Mar. 1969), 185-187. (See also remarks by R. Griffin and K. A. Redish in Comm. ACM 13, 1 (Jan. 1970), 54 and by R. Peto in Comm. ACM 13, 10 (Oct. 1970), 624.)

[35] Van Emden, M. N. "Increasing the efficiency of quicksort." Comm. ACM 13, 9 (Sept. 1970), 563-567.

[36] Van Emden, M. N. "Increasing the efficiency of quicksort: Algorithm 402." Comm. ACM 13, 11 (Nov. 1970), 693-694.

[37] Windley, P. F. "Trees, forests, and rearranging." Computer J. 3 (1960), 84-88.

APPENDIX A

The programs in the text are presented in a "high-level" language in which they are simply expressed. However, in order to analyze the running times of the programs, it is necessary to know the characteristics of the particular computer on which they are run, and their implementation in the machine language of that computer. The purpose of this Appendix is to illustrate how this might be done by studying the machine language implementation of some of the programs for the MIX computer described by D. E. Knuth in The Art of Computer Programming, Volume 1. The language is completely described in this book, but we use only a few of its features, and with the aid of the cursory description given below, an experienced assembly language programmer should have no trouble translating the programs to run on his particular machine. Our aim is to produce programs that are as efficient as possible, and to this end we shall use some standard programming techniques to improve the efficiency of the programs. Most of these have little effect on the analysis in the text, since they only affect the coefficients of the various quantities studied. However, perhaps the most effective technique does change the analysis, and we shall examine that briefly. Some further comment on the implementation of Quicksort, on real computers, is given in Appendix C.

The MIX processor has two registers, A and X, which are large enough to hold memory words; and six index registers, I1, I2, I3, I4, I5, and I6 which can hold addresses. These registers can be loaded from memory using one of the instructions LDA, LDX, or LDi $i = 1,\dots,6$. Every instruction can be "indexed". For example the instruction " LDA A+N,4 " loads into register A the word whose address is computed by adding the contents of index register 4 to the value A+ N . Similarly we can store into memory by using one of the instructions STA , STX or STi i = 1,...,6 . Addresses can be loaded using ENTA , ENTX , or ENTi i = 1,...,6 : for example " ENT4 2-N " loads the number 2-N into register 4 ; if this were followed by " ENT5 N-1,4 " then register 5 would get the number 1 . Further address arithmetic can be done using INCA , INCX , or INCi i = 1,...,6 and DECA , DECX , or DECi i = 1,...,6 to "increment" or "decrement" registers. The instruction " INC4 1 " adds 1 to register 4 (as would " ENT4 1,4 ") and " DECX 0,5 " subtracts the contents of register 5 from the X register. Finally, there are comparison instructions CMPA , CMPX and CMPi i = 1,...,6 and conditional jumps either depending on the outcome of a comparison (JL , JE , JG , JLE , JNE , JGE for jump if less, equal, greater, less than or equal, etc.) or on the value of a register (JXN , JXZ , JXP , JXNN , JXNZ , JXNP for jump if the X register is negative, zero, positive, nonnegative, etc.). The unconditional transfer is JMP . This includes nearly all of the instructions that are used in the programs below -- we will discuss more exotic instructions as we encounter them.

We shall assume in implementing the programs that the keys to be sorted are in memory locations A+1 , A+2 , ... , A+N , where the values of A and N are defined elsewhere. Further, we shall assume that memory location A contains a key smaller than all of the keys to be sorted $(-\infty)$ and memory location A+N+1 contains a key larger than all of the keys to be sorted (∞) . To minimize confusion, we shall always try to keep " i " in index register 4 , " j " in index register 5 , and " v " in the A register. For example, the instruction " STA A+1,5 " will correspond to the operation " A[j+1] := v; ".

Rather than dwelling further on the format and conventions, let us examine the assembly language implementation of the first full program that we encountered, Program 1.2.

Program 1.2A

```
        ENT4  2-N        1          i := 2;
NEXT    LDA   A+N,4      N-1        loop while i ≤ N:  v := A[i];
        FNT5  N-1,4      N-1                           j := i-1;
MOVE    CMPA  A,5        D*+E                          loop while A[j] > v and j > 0:
        JGE   INSERT     D*+E
        LDX   A,5        E
        STX   A+1,5      E                                 A[j+1] := A[j];
        DEC5  1          E                                 j := j-1;
        J5P   MOVE       E                             repeat;
INSERT  STA   A+1,5      N-1                           A[j+1] := v;
        INC4  1          N-1                           i := i+1;
        J4NP  NFXT       N-1        repeat;
```

The version of Program 1.2 that we studied in Chapter 1 is listed at right above, with each statement placed as near as possible to the corresponding assembly language instruction. The only coding "trick" used here is that register 4 contains the value of i-N rather than the value of i , which makes it possible to do the test " i ≤ N " with one instruction (" J4NP NEXT "). The frequency counts for each instruction are given in the middle column, in terms of the unknown quantities defined in Chapter 1. To find the running time of the program from these we need only know that address modification and jump instructions (which don't reference memory) take 1 time unit; and loads, stores and compares (which do reference memory) take 2 time units. Adding the total time spent executing each instruction, we find that the total running time of this program is $3D^{*} + 9E + 7N - 6$ time units.

We found in Chapter 1 that this could be improved by using the assumption that $A[0] = -\infty$ to remove the test " $j > 0$ ". We can now see this savings more clearly.

Program 1.7

```
          ENT4  -N           1          i := 2;
NEXT      LDA   A+N,4        N-1        loop while i ≤ N:
          CMPA  A+N+1,4      N-1            if A[i] < A[i-1] then
          JGE   NOINSERT     N-1                v := A[i];
          ENT5  N-1,4        D                  j := i-1;
MOVE      LDA   A,5          E                  loop:
          STA   A+1,5        E                      A[j+1] := A[j];
          DEC5  1            E                      j := j-1;
          CMPA  A,5          E                  while A[j] > v:
          JG    MOVE         E                  repeat;
          STA   A+1,5        D                  A[j+1] := v;           endif;
NOINSERT  INC4  1            N-1            i := i+1;
          J4NP  NEXT         N-1        repeat;
```

There is one less instruction executed E times, so that the total running of this program is $3D + 8E + 7N - 6$.

It turns out that we can improve this program even further, by using the powerful MIX "MOVE " instruction. This is an instruction of the "block transfer" type, which allows a block of up to 63 words to be moved from one location to another in memory. Index register 1 is used to specify the destination. For example, if register 1 contains 1000 , then the instruction " MOVE 2000(50) " would result in the 50 words in locations 2000 , ... , 2049 being moved to locations 1000 , ... , 1049 . The move proceeds from left to right, and register 1 is incremented by the number of words moved. The instruction takes one time unit plus two time units for each word moved. The above programs take four time units per word for moving in the inner loop, and they can be improved as follows.

```
          ENT4  N-1         1          i := N-1;
NEXT      LDA   A,4         N-1        loop while i ≥ 1:
          CMPA  A+1,4       N-1            if A[i] > A[i+1] then
          JLE   NOINSERT    N-1                v := A[i];
          ENT5  1,4         D^R                j := i+1;
          INC5  1           E                  loop: j := j+1;
          CMPA  A,5         E                  while A[j] < v:
          JG    *-2         E                  repeat;
          DEC5  1,4         D^R                t := j-i-1;
          ST5   *+2(4:4)    D^R
          ENT1  A,4         D^R                k := i;
          MOVE  1,1         D^R (+2E)          loop while t > 0: A[k] := A[k+1]; t := t-1; repeat;
          STA   0,1         D^R                A[k] := v;                    endif;
NOINSERT  DEC4  1           N-1            i := i-1;
          J4P   NEXT        N-1        repeat;
```

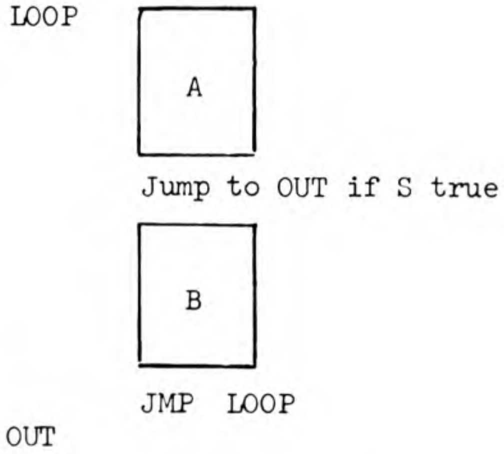
Here the insertion process is reversed: we work from right to left, (i = N-1,...,1) and put A[i] into position among A[i+1], A[i+2], ... , A[N] by moving all of the keys which are less than it to the left one position. The instruction " ST5 *+2(4:4) " stores the value of register 5 into the length field at the " MOVE 1,1 " instruction, which then performs the moves necessary to allow the insertion of A[i] . The running time of this program is $8D^R + 6E + 7N - 6$. Since D^R obviously has the same average value as D , we can see from our derivation in Chapter 1 that the average improvement over Program 1.2 is $2E_N - 5D_N = \frac{1}{2}(N^2 - 11N + 10H_N)$ which is positive for $N \geq 8$ and significant for larger values of N (although the program doesn't work, and shouldn't be used anyway, for $N \geq 64$). This improvement is not mentioned in the text because it is not an improvement for Quicksort, where we use insertion sorting only for small or well-ordered files.

The assembly language coding for the various partitioning methods in Chapter 3 is very simple and straightforward. For example, Partitioning Method 2.1 might be implemented as follows:

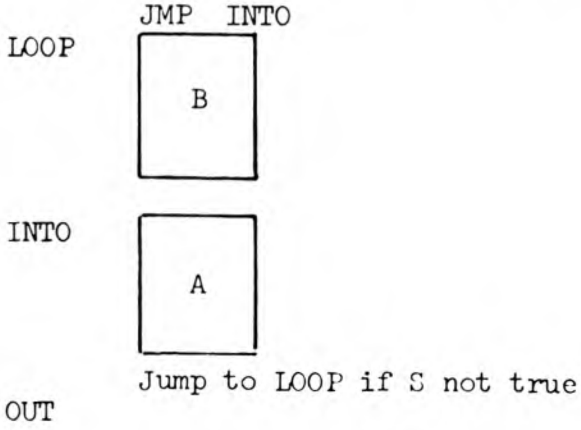
```
PARTITION ENT4  0,2       A       i := l;
          ENT5  1,3       A       j := r+1;
          LDA   A,2       A       v := A[l];
LOOP      DEC5  1         C'      loop until pointers have met:  loop: j := j-1;
          CMPA  A,5       C'                                     while A[j] > v
          JL    *-2       C'                                     repeat;
          ENTX  0,4       B+A
          DECX  0,5       B+A                                    if i ≥ j
          JXNN  OUTA      B+A                                    then j := i; pointers have met endif;
          LDX   A,5       B+X
          STX   A,4       B+X                                    A[i] := A[j];
          INC4  1         C-C'                                   loop: i := i+1;
          CMPA  A,4       C-C'                                   while A[i] < v
          JG    *-2       C-C'                                   repeat;
          ENTX  0,4       B+X
          DECX  0,5       B+X                                    if i ≥ j
          JXNN  OUT       B+X                                    then pointers have met endif;
          LDX   A,4       B
          STX   A,5       B                                      A[j] := A[i];
          JMP   LOOP      B       repeat;
OUTA      ENT5  0,4       A-X     j := i;
OUT       STA   A,5       A       A[j] := v;
[illegible]
```

Since the methods in the text are expressed in a goto -less language, it will not always be convenient to maintain a direct correspondence between the programs in the text and the assembly language, although the operation of the programs should be clear. In this program the loop termination condition "pointers have met" is implemented simply by a jump to " OUT ". Then " j := i; pointers have met " is conveniently implemented by a jump to " OUTA " with an " ENT5 0,4 " instruction at that location. The total running time of this program segment is $10A + 15B + 4C + 6X$. (The quantities A , B and C are the same as those studied in Chapter 3; the average value of X turns out to be 1/2 A .)

In the text we dealt with coding techniques to improve program efficiency which could easily be expressed in a high-level language; in this Appendix we will see some standard programming techniques relevant to assembly language programming. The most important of these, the elimination of unconditional jumps in inner loops, will improve the performance of the above program. The idea is that it is always wasteful to end an inner loop with an unconditional jump, since the loop must contain a conditional jump,



and we can always "rotate" it to get the equivalent code



We have removed one instruction from the inner loop. Applying this to the above program, we can save a little more by carefully placing INTO , and we have

Partitioning Method 2.1A

```
PARTITION ENT4  0,2    A      i := ℓ;
          ENT5  0,3    A      j := r;
          LDA   A,2    A      v := A[ℓ];
          JMP   INTO   A      goto into;
LOOP      LDX   A,5    B+X    loop until pointers have met:
          STX   A,4    B+X                                      A[i] := A[j];
          INC4  1      C-C'                                     loop:  i := i+1;
          CMPA  A,4    C-C'                                     while A[i] < v
          JG    *-2    C-C'                                     repeat;
          ENTX  0,4    B+X
          DECX  0,5    B+X                                      if i ≥ j
          JXNN  OUT    B+X                                      then pointers have met:  endif;
          LDX   A,4    B
          STX   A,5    B                                        A[j] := A[i];
          DEC5  1      C'-A                                     loop:  j := j-1;
INTO      CMPA  A,5    C'     into:                             while A[j] > v
          JL    *-2    C'                                       repeat;
          ENTX  0,4    B+A
          DECX  0,5    B+A                                      if i ≥ j then j := i; pointers have met endif;
          JXN   LOOP   B+A    repeat:
          ENT5  0,4    A-X    (j := i;)
OUT       STA   A,5    A      A[j] := v;
```

which takes $10A + 14B + 4C + 6X$ time units, a savings of B time units over the previous program. This technique applies to all of our partitioning methods, and we shall use it in all of the programs below.

The implementation of Partitioning Method 2.2 leads to a longer program, but a more efficient inner loop:

Partitioning Method 2.2A

```
PARTITION  ENT4   0,2            A          i := l;
           ENT5   1,2            A          j := r+1;
           ENTA   0,2            A
           INCA   0,3            A
           SRB    1              A
           STA    *+1(0:2)       A
           ENT6   *              A          p := (l+r) ÷ 2;
           LDA    A,6            A          v := A[p]
           JMP    INTO           A          goto into;
LOOP       LDX    A,4            B          loop:
           ENT1   A,4            B
           MOVE   A,5            B
           STX    A,5            B                   A[i] :=: A[j];
           INC4   1              C-C'-A              loop:  i := i+1;
INTO       CMPA   A,4            C-C'       into:    while A[i] ≤ v
           JGE    *-2            C-C'                repeat;
           DEC5   1              C'                  loop:  j := j-1
           CMPA   A,5            C'                  while A[j] ≥ v
           JLE    *-2            C'                  repeat;
           ENTX   0,4            B+A
           DECX   0,5            B+A        while i < j
           JXN    LOOP           B+A        repeat;
IF1        ENTX   0,5            A          if i < p
           DECX   0,6            A
           JG     IF2            A
           LDX    A,4            F          then
           STA    A,4            F                   A[i] :=: A[p];
           INC4   1              F                   i := i+1;
           JMP    PARTDONEA      F          endif;
IF2        ENTX   0,6            A-F        if j > p
           DECX   0,4            A-F
           JL     PARTDONE       A-F
           LDX    A,5            G          then
           STA    A,5            G                   A[p] :=: A[j];
           DEC5   1              G                   j := j-1;
PARTDONEA  STX    A,6            F+G        endif;
PARTDONE
```

This program involves a few new MIX instructions. First, the " SRB 1 " instruction is "shift right binary one" which takes two time units, and the instruction following stores register A into the address field of the ENT6 instruction; so that these three instructions result in register 6 being loaded with 1/2 the value in the A register. Second, the default length for the " MOVE " instruction is 1, so that the four instructions starting at LOOP perform the desired exchange. The running time of this program is $21A + 11B + 4C + 5F + 7G$. The average values of both F and G are slightly less than $\frac{1}{2}A$. The inner loop has been improved, at moderate expense outside the inner loop.

The implementation of Partitioning Method 2.5 is very similar to that of Method 2.4, so we will move next to the complete implementation of Program 2.4:

Program 2.4A (Partitioning Method 2.4A)

```
QUICKSORT  ENT2 1                1
           ENT3 N                1
           ENT6 0                1
PARTITION  ENT4 1,2              A               loop until done:  i := l+1;
           ENT5 1,3              A                                 j := r+1;
           LDA  A,2              A                                 v := A[l];
           JMP  INTO             A                                 goto into;
LOOP       LDX  A,4              B                                 loop:
           ENT1 A,4              B
           MOVE A,5              B
           STX  A,5              B                                           A[i] :=: A[j];
           INC4 1                C'-A                                        loop:  i := i+1;
INTO       CMPA A,4              C'                                into:     while A[i] < v
           JG   *-2              C'                                          repeat;
           DEC5 1                C-C'                                        loop:  j := j-1;
           CMPA A,5              C-C'                                        while A[j] > v
           JL   *-2              C-C'                                        repeat;
           ENTX 0,4              B+A
           DECX 0,5              B+A                               while i < j
           JXN  LOOP             B+A                               repeat;
           LDX  A,5              A
           STA  A,5              A
           STX  A,2              A                                 A[l] :=: A[j];
PARTDONE   ENT4 0,3              A
           DEC4 M,5              A
           ENT1 0,5              A
           DEC1 M,2              A
           ENTA 0,4              A
           DECA 0,1              A
           JANN RBIG             A
LBIG       J1NP POP              A'
           J4NP LEFT             A"-S+2S'                          if j-l > M > r-j then r := j-1; else
           INC6 2                S'                                if j-l > r-j > M then p := p+2;
           ST2  STACK,6(2:3)     S'                                                      stack[p] := l;
           ENTA 1,5              A-A"-S-1
           STA  STACK,6(4:5)     S'                                                      stack[p+1] := j-1;
RIGHT      ENT2 1,5              A-A"-S-1                                                l := j+1; else
           JMP  PARTITION        A-A"-S-1
RBIG       J4NP POP              A-A'
           J1NP RIGHT            A-A"-2S'-1                        if r-j > M ≥ j-l then l := j+1; else
           INC6 2                S-S'                              if r-j ≥ j-l > M then p := p+2;
           ST3  STACK,6(4:5)     S-S'                                                    stack[p] := j+1;
           ENTA 1,5              S-S'
           STA  STACK,6(2:3)     S-S'                                                    stack[p+1] := r;
LEFT       ENT3 -1,5             A"                                                      r := j-1; else
           JMP  PARTITION        A"
POP        LD2  STACK,6(2:3)     S+1                               if M ≥ j-l > r-j or
           LD3  STACK,6(4:5)     S+1                                  M ≥ r-j ≥ j-l then l := stack[p];
           DEC6 2                S+1                                                     r := stack[p+1];
           J6NN PARTITION        S+1             repeat;                                 if p = 0 then
INSERTION  ENT4 2-N              1               i := 2;                                done endallifs;
NEXT       LDA  A+N,4            N-1             loop while i ≤ N
           CMPA A+N+1,4          N-1                   if A[i] < A[i-1] then
           JGE  NOINSERT         N-1                         v := A[i];
           ENT5 N-1,4            D                           j := i-1;
MOVE       LDX  A,5              E                           loop:
           STX  A+1,5            E                                 A[j+1] := A[j];
           DEC5 1                E                                 j := j-1;
           CMPA A,5              E                           while A[j] > v
           JL   MOVE             E                           repeat;
           STA  A+1,5            D                           A[j+1] := v;          endif;
NOINSERT   INC4 1                N-1                   i := i+1;
           J4NP NEXT             N-1             repeat;
```

This implementation follows directly from the other examples that we have seen. The most complicated aspect of this program is the assignment of the instruction frequencies to the two instructions at each of the labels LBIG , RIGHT , RBIG and LEFT . In general, there are many different ways to assign frequencies. The method used here was to asssign A" to the instructions at LEFT ; then use Kirchhoff's Law at LEFT and RIGHT to deduce the frequencies of the instructions at LBIG+1 and RBIG+1 . Fortunately, all of the quantities A' , A" , C' , and S' cancel out in computing the total running time of the program, which is $24A + 11B + 4C + 3D + 8E + 9S + 7N$.

The assembly language implementations of the variants of Quicksort studied in Chapters 5, 6, and 7 can all be easily constructed using Program 2.4A as a model, and we will not examine them here. The median-of-three modification in Chapter 8 is also a simple extension of the above program, but we will conclude by examining its implementation because of its importance.

Partitioning Method 8.1A

```
PARTITION  ENTA  0,2        A
           INCA  0,3        A
           SRB   1          A
           STA   *+1(0:2)   A
           ENT4  *          A       p := (ℓ+r) ÷ 2;
           LDA   A,2        A
           CMPA  A,3        A
           JLE   IF2        A
           LDX   A,3        A1      if A[ℓ] > A[r] then A[ℓ] :=: A[r] endif;
           STX   A,2        A1
           STA   A,3        A1
IF2        LDA   A,4        A
           CMPA  A,3        A
           JLE   IF3        A
           LDX   A,3        A2      if A[(ℓ+r) ÷ 2] > A[r] then A[(ℓ+r) ÷ 2] :=: A[r] endif;
           STX   A,4        A2
           STA   A,3        A2
           LDA   A,4        A2
IF3        CMPA  A,2        A
           JG    CHOSEN     A
           LDX   A,2        A3      if A[ℓ] > A[(ℓ+r) ÷ 2] then A[ℓ] :=: A[(ℓ+r) ÷ 2] endif;
           STA   A,4        A3
           STA   A,2        A3
           LDA   A,4        A3
CHOSEN     LDX   A+1,2      A
           STX   A,4        A       A[ℓ+1] :=: A[(ℓ+r) ÷ 2]; v := A[ℓ+1];
           ENT4  2,2        A       i := ℓ+2;
           ENT5  0,3        A       j := r;
           JMP   INTO       A       go to into;
LOOP       LDX   A,4        B       loop:
           ENT1  A,4        B
           MOVE  A,5        B
           STX   A,5        B              A[i] :=: A[j];
           INC4  1          C'-A           loop:  i := i+1;
INTO       CMPA  A,4        C'      into:  while A[i] < v
           JG    *-2        C'             repeat;
           DEC5  1          C-C'           loop:  j := j-1;
           CMPA  A,5        C-C'           while A[j] > v
           JL    *-2        C-C'           repeat;
           ENTX  0,4        B+A
           DECX  0,5        B+A     while i < j
           JXN   LOOP       B+A     repeat;
           LDX   A,5        A
           STA   A,5        A
           STX   A+1,2      A       A[ℓ+1] :=: A[j];
PARTDONE
```

This code may be used to make a complete Quicksort program by inserting it into Program 2.4A in place of all of the instructions between PARTITION and PARTDONE-1 . After the initial manipulations, the partitioning method is of course the same, except that at the end we have " STX A+1,2 " (for $A[\ell+1] :=: A[j]$) rather than " STX A,2 " (for $A[\ell] :=: A[j]$). The total running time of Partitioning Method 8.2A is

$$35A + 6A_1 + 8A_2 + 8A_3 + 11B + 4C$$

and the quantities A_1 , A_2 , A_3 are $\frac{1}{2}A$, $\frac{1}{3}A$, $\frac{1}{3}A$ on the average, so the total average running time of the method is $43\frac{1}{3}A + 11B + 4C$, as compared with $13A + 11B + 4C$ for the corresponding code in Program 2.4A, so the additional cost is $30\frac{1}{3}A$, on the average.

An alternate method of implementing the choosing of the partitioning element in the median-of-three method is to avoid all extraneous data movements by treating separately each of the six possible cases determined by the relative order of $A[\ell]$, $A[(\ell+r)\div 2]$, and $A[r]$. This results in a longer program, which is very slightly more efficient in MIX:

Partitioning Method 8.1A (Alternate implementation: Part 1)

```
PARTITION  ENTA  0,2        A
           INCA  0,3        A
           SRB   1          A
           STA   *+1(0:2)   A
           ENT4  *          A                        p := (ℓ+r) ÷ 2;
           LDA   A,2        A
           LDX   A,3        A
           CMPA  A,3        A
           JL    LR         A
           CMPA  A,4        $A_1 + A_2 + A_3$
           JLE   RLP        $A_1 + A_2 + A_3$
           CMPX  A,4        $A_2 + A_3$
           JG    PRL        $A_2 + A_3$
RPL        STA   A,3        $A_3$                    if A[r] < A[p] < A[ℓ] then A[ℓ] :=: A[r]; v := A[p]; else
           STX   A,2        $A_3$
LPR        LDA   A,4        $A_3 + A_5$
           JMP   CHOSEN     $A_3 + A_5$
BRL        LDA   A,3        $A_2$                    if A[p] < A[r] < A[ℓ] then v := A[r]; A[r] := A[ℓ];
           LDX   A,2        $A_2$                                                       A[ℓ] := A[p]; else
           STX   A,3        $A_2$
           JMP   PLR        $A_2$
RLP        STX   A,2        $A_1$                    if A[r] < A[ℓ] < A[p] then v := A[ℓ]; A[ℓ] := A[r];
           LDX   A,4        $A_1$                                                       A[r] := A[p]; else
           STX   A,3        $A_1$
           JMP   CHOSEN     $A_1$
LR         CMPA  A,4        $A_4 + A_5 + A_6$
           JGE   PLR        $A_4 + A_5 + A_6$
           CMPX  A,4        $A_5 + A_6$
           JG    LPR        $A_5 + A_6$              if A[ℓ] < A[p] < A[r] then v := A[p]; else
LRP        LDA   A,3        $A_6$                    if A[ℓ] < A[r] < A[p] then v := A[r]; A[r] := A[p]; else
           LDX   A,4        $A_6$
           STX   A,3        $A_6$
           JMP   CHOSEN     $A_6$
PLR        LDX   A,4        $A_2 + A_4$              if A[p] < A[ℓ] < A[r] then v := A[ℓ]; A[ℓ] := A[p]; endallifs;
           STX   A,2        $A_2 + A_4$
CHOSEN     LDX   A+1,2      A
           STX   A,4        A
           STA   A+1,2      A                        A[ℓ+1] :=: A[p];
             .
             .
             .
```

The quantities $A_1, A_2, \ldots, A_6$ are all $\frac{1}{6} A_N$ on the average, so it turns out that this implementation takes $\frac{5}{6} A_N$ time units less than the last, a very slight improvement.

Finally, we shall consider a coding technique which often leads to significant improvements in programs which involve simple pointer arithmetic in their inner loops. The technique is to "unwrap" the inner loop, in response to the observation that 1/4 of our overhead for comparisons is taken up by pointer increments and decrements. The instruction sequence

```
INC4  1
CMPA  A,4
JG    *-2
```

is exactly equivalent to

```
CMPA  A+1,4
JLE   *+5
INC4  2
CMPA  A,4
JG    *-4
JMP   *+2
INC4  1
```

and the pointer in index register 4 is incremented only about half as often in the inner loop. This important coding technique should be used to improve the efficiency of Quicksort in a production situation, so we shall now examine it more closely.

It turns out that we can save a little more by introducing some duplicate code. If we apply the example above to our programs, then the overhead associated with comparisons is reduced, but the overhead associated with exchanges is increased slightly because of the " JMP *+2 " instruction. This can be eliminated by including two copies of the code following the loop, one for the case where register 4 has to be incremented and the other for the case where it does not. Applying this idea to *both* of our comparison loops, we have the second half of our final implementation of the algorithm.

Partitioning Method 8.1A (Alternate implementation: Part 2)

```
           ENT4  1,2           A            i := l+1;
           ENT5  0,3           A            j := r;
           JMP   LSCAN         A            goto into;
LOOP       LDX   A,4           B            loop:
           ENT1  A,4           B
           MOVE  A,5           B
           STX   A,5           B                  A[i] :=: A[j];
LSCAN      CMPA  A+1,4         C1                 loop until done odd or done even:
           JLE   COPYRSCAN     C1           into:       if A[i+1] ≥ v then done odd endif;
           INC4  2             C2                       i := i+2;
           CMPA  A,4           C2                       if A[i] ≥ v then done even endif;
           JG    LSCAN         C2                 repeat;
RSCAN      CMPA  A-1,5         C3'                then done even ⇒ loop until done odd or done even
           JGE   COPYTEST      C3'                                 if A[j-1] ≤ v then done odd endif;
           DEC5  2             C4'                                 j := j-2;
           CMPA  A,5           C4'                                 if A[j] ≤ v then done even endif;
           JL    RSCAN         C4'                                 repeat;
TEST       ENTX  0,4           B'                                  then done odd ⇒ j := j-1;
           DECX  0,5           B'                                  endloop;
           JXN   LOOP          B'
           JMP   SCANDONE      X'
COPYRSCAN  INC4  1             C1-C2                    done odd  ⇒ i := i+1;
           CMPA  A-1,5         C3-C3'                              loop until done odd or done even
           JGE   COPYTEST      C3-C3'                              if A[j-1] ≤ v then done odd endif;
           DEC5  2             C4-C4'                              j := j-2;
           CMPA  A,5           C4-C4'                              if A[j] ≤ v then done even endif;
           JL    COPYRSCAN+1   C4-C4'                              repeat;
           ENTX  0,4           B"
           DECX  0,5           B"
           JXN   LOOP          B"
           JMP   SCANDONE      X-X'
COPYTEST   DEC5  1             C3-C1                               then done odd ⇒ j := j-1;
           ENTX  0,5           B-A-B'-B"          endloop;         endloop;
           DECX  0,5           B-A-B'-B"    while i < j
           JMP   LOOP          B-A-B'-B"    repeat;
SCANDONE   LDX   A,5           A
           STA   A,5           A
           STX   A+1,2         A            A[l+1] :=: A[j];
PARTDONE
```

This involves quite a bit more code than our previous implementations of this inner loop (for example the last 20 lines of Partitioning Method 8.1A) but it is much faster. It requires $12A + 11B + 4C_1 + 3C_2 + 4C_3 + 3C_4 + X$ time units as opposed to $11A + 11B + 4C$ for the previous implementations, where $C_1 + C_2 + C_3 + C_4 = C$, so that the savings is $C_2 + C_4 - A - X$. It is shown below that on the average this savings is asymptotically $\left(\frac{1}{3} + \frac{3}{2} - 2 \ln 2\right) C_N \approx .447\, C_N$, which is quite substantial.

There are many examples in the text of a simple change to the algorithm having a major impact on the analysis, and the study of the effects of unwrapping the inner loop is yet another example of this. The analysis is not at all trivial, and is only sketched here. Also, to simplify the description, we will first consider the application of this technique to Program 2.4A (using the same code, with the first two and last instructions modified in the obvious way).

We begin, as usual, by finding the savings achieved on the first partitioning stage. Consider the keys $A[2], \ldots, A[s]$, and suppose that exactly t of them are $< s$. As we saw in Chapter 3, this occurs with probability $\dfrac{\binom{s-1}{t}\binom{N-s}{s-1-t}}{\binom{N-1}{s-1}}$. The savings resulting from unwrapping the loop is dependent on the distribution of the keys $< s$. For example: if $t = 0$, there is no savings (the instructions counted by C_2 are not executed); and if $t = s-1$, then the savings is $\left\lceil \frac{s-1}{2} \right\rceil$ (every other comparison is counted by C_2); but if $t = 2$ there may be a savings of one or two depending on whether or not the keys $< s$ are adjacent. In general, for each run of adjacent keys $< s$ of length u, the savings is $\left\lceil \frac{u}{2} \right\rceil$, so the total number of comparisons counted by C_2 is

$$\frac{t}{2} + \frac{1}{2}\ (\text{\# odd length runs of adjacent keys } < s) \quad .$$

An entirely symmetric argument holds for the right hand side. Similarly, the contribution of the first stage to X is 1 if $A[s+1],\ldots,A[N]$ begins with an even number of keys $< s$, so we are led to the following expression for the total average savings achieved on the first partitioning stage:

$$\frac{1}{N} \sum_{1 \le s \le N} \left(\sum_{t} \frac{\binom{s-1}{t}\binom{N-s}{s-1-t}}{\binom{N-1}{s-1}} \left(\frac{t}{2} + \frac{Q_{(s-1)t}}{2\binom{s-1}{t}} \right) \right.$$

$$\left. + \sum_{t} \frac{\binom{N-s}{t}\binom{s-1}{N-s-t}}{\binom{N-1}{s-1}} \left(\frac{t}{2} + \frac{Q_{(N-s)t}}{2\binom{N-s}{t}} - \frac{E_{(N-s)t}}{\binom{N-s}{t}} \right) \right) - 1 \quad .$$

Here Q_{st} is defined to be the total number of odd length runs of 0's when t 0's and $s-t$ 1's are randomly arranged in all $\binom{s}{t}$ ways, and E_{st} is defined to be the number of times such a random arrangement of 0's and 1's begins with an even length run of 0's. This equation simplifies to

$$\frac{1}{3}(N+1) - \frac{3}{2} + \frac{1}{N} \sum_{1 \le s \le N} \sum_{t} (Q_{(s-1)t} - E_{(s-1)t}) \frac{\binom{N-s}{s-1-t}}{\binom{N-1}{s-1}} \quad .$$

To evaluate E_{st} and Q_{st}, it is not difficult to derive the recurrences

$$E_{st} = \binom{s-1}{t} + \binom{s-1}{t-1} - E_{(s-1)(t-1)}$$

and

$$Q_{st} = Q_{(s-1)t} + Q_{(s-1)(t-1)} + E_{(s-1)(t-1)} - \left(\binom{s-1}{t-1} + E_{(s-1)(t-1)}\right)$$

which have the solution

$$Q_{st} = (1+s-t) \sum_{0 \le k \le s} (-1)^k \binom{s-k}{t-k} = (1+s-t) E_{st} \quad .$$

Substituting this into our formula, summing first on t and then on s, we get the expression

$$\frac{1}{3}(N+1) - \frac{3}{2} + \sum_{0 \le k \le N-2} (-1)^k \frac{N-k-2}{(k+3)(k+2)}$$

for the total savings on the first partitioning stage. Unfortunately there is no simple expression for this last sum, though we can estimate it closely to get the approximation

$$(N+1)\left(\frac{1}{3} + \frac{3}{2} - 2 \ln 2\right) + \ln 2 - 1 - \frac{3}{2} + \epsilon \qquad ,$$

where $|\epsilon| < \frac{2}{N}$, for the savings on the first stage due to loop unwrapping. Now, we can solve recurrences as we did in Chapter 3 to find the total savings achieved, and it comes out to be

$$(N+1)\left(\frac{11}{6} - 2\ln 2\right)(2H_{N+1} - 2H_{M+2} + 1) + \left(\ln 2 - \frac{5}{2}\right)\left(2\frac{N+1}{M+2} - 1\right) +$$

where $|\epsilon| < 2\,\frac{N+1}{(M+1)(M+2)}$. For $M = 9$, this is approximately

$$.89(N+1)\ln N - 2.12N + \epsilon + O(1) \quad , \qquad \text{where} \quad |\epsilon| < .02\,N \ ,$$

a very substantial savings.

The analysis for the alternate implementation of Program 8.2A as it stands is very similar to the above, and it turns out that the savings on the first partitioning stage is

$$(N+1)\left(12 \ln 2 - 8 + \frac{2}{10}\right) + 2 - 6 \ln 2 + \epsilon \quad , \qquad \text{where} \quad |\epsilon| < \frac{12}{N}$$

and the total savings is about

$$1.06(N+1)\ln N - 5.03N + \epsilon + O(1) \quad , \qquad \text{where} \quad |\epsilon| < .1\,N \ .$$

We shall refer to the complete Quicksort program obtained by insertings parts 1 and 2 of this alternate implementation of Partitioning Method 8.1A into Program 2.4A (in place of all of the instructions between PARTITION and PARTDONE - 1) as <u>Program 8.2A</u>. The analysis above (and in Chapter 8) shows that its total running time is about

$$9.57N \ln N + 7.14N \quad ,$$

and it is an extremely efficient implementation of Quicksort which can be adapted to most computers for use as a production sorting algorithm.

APPENDIX B

The emphasis in the mathematical analysis of the various algorithms in the text is on obtaining exact answers to the problems which arise. We normally deal with functions whose domain is the integers, and manipulate them using the "finite difference calculus" (which parallels the more familiar "differential calculus" which is appropriate for dealing with functions defined on the reals). In fact, most of the problems arising in the analysis of Quicksort reduce to the evaluation of finite summations involving only a few kinds of functions: harmonic numbers and binomial coefficients. This Appendix will be largely devoted to such problems, although it turns out that a variety of techniques are needed for their solution. We will also consider: the solution of recurrence relations; the use of generating functions, especially probability generating functions; and some simple asymptotic representations. Recognizing that many readers may not be familiar with this kind of mathematics, we shall derive most of the identities that are used in the text rather than simply state them. (However, since we shall only deal with Quicksort-related problems, a reader completely unfamiliar with such topics should study the more general treatment given by Knuth [16].) An index to the notations used in this thesis appears at the end of this Appendix.

Central to the study of Quicksort are the <u>harmonic numbers</u>, which are defined by the formula

$$H_n = \sum_{1 \le j \le n} \frac{1}{j} = 1 + \frac{1}{2} + \frac{1}{3} + \ldots + \frac{1}{n} \quad . \tag{1}$$

This sum does not converge as n gets large. In fact, we shall see that the harmonic numbers are closely related to logarithms, and H_n is very close to $\ln n + \gamma$ (where $\gamma = .5772156649\ldots$ is called Euler's constant) for large n. The values of H_n for small values of n are given in the table below.

n	H_n	$\approx H_n$
0	0	0
1	1	1
2	$\frac{3}{2}$	1.5
3	$\frac{11}{6}$	1.833333333
4	$\frac{25}{12}$	2.083333333
5	$\frac{137}{60}$	2.283333333
6	$\frac{49}{20}$	2.450000000
7	$\frac{363}{140}$	2.5928571428
8	$\frac{761}{280}$	2.7178571428
9	$\frac{7129}{2520}$	2.8289682539
10	$\frac{7381}{2520}$	2.9289682539

There is an entire family of related functions, and we also encounter

$$H_n^{(2)} = \sum_{1 \le i \le n} \frac{1}{i^2} = 1 + \frac{1}{4} + \frac{1}{9} + \ldots + \frac{1}{n^2} . \qquad 2.$$

This sum does converge, and we will have use for its limit,

$$\sum_{j \geq 1} \frac{1}{j^2} = \frac{\pi^2}{6} \quad , \tag{3}$$

which is a value of Riemann's "zeta-function". (This is a well-known result from classical mathematics which we shall not prove here.)

Often in our analysis we are faced with summations involving harmonic numbers. There is a seemingly endless variety of these, but they do provide a good exercise in working with the $\sum$ operator. For example, the simplest sum to consider might be

$$\sum_{1 \leq k \leq n} H_k = \sum_{1 \leq k \leq n} \sum_{1 \leq j \leq k} \frac{1}{j} \quad .$$

To evaluate this, we interchange the order of summation to get a trivial inner sum:

$$\begin{aligned}
\sum_{1 \leq k \leq n} H_k &= \sum_{1 \leq j \leq n} \sum_{j \leq k \leq n} \frac{1}{j} \\
&= \sum_{1 \leq j \leq n} \frac{1}{j} \sum_{j \leq k \leq n} 1 \\
&= \sum_{1 \leq j \leq n} \frac{n-j+1}{j} \\
&= (n+1) \sum_{1 \leq j \leq n} \frac{1}{j} - \sum_{1 \leq j \leq n} 1
\end{aligned}$$

$$\sum_{1 \leq k \leq n} H_k = (n+1)H_n - n = (n+1)(H_{n+1} - 1) \quad . \tag{4}$$

For a slightly harder problem, we might consider the sum of the squares of the harmonic numbers:

$$\sum_{1\le k\le n} H_k^2 = \sum_{1\le k\le n} \sum_{1\le j\le k} \frac{1}{j} H_k$$

$$= \sum_{1\le j\le n} \frac{1}{j} \sum_{j\le k\le n} H_k$$

$$= \sum_{1\le j\le n} \frac{1}{j} \left(\sum_{1\le k\le n} H_n - \sum_{1\le k<j} H_k \right)$$

$$= \sum_{1\le j\le n} \frac{1}{j} ((n+1)H_n - n - j(H_j - 1))$$

$$= (n+1)H_n^2 - nH_n - \sum_{1\le j\le n} (H_j - 1)$$

$$\sum_{1\le k\le n} H_k^2 = (n+1)H_n^2 - (2n+1)H_n + 2n = (n+1)(H_{n+1} - 1)^2 - H_{n+1} + n + 1 \quad . \qquad (5)$$

This, of course, is not to be confused with

$$\sum_{1\le k\le n} H_k^{(2)} = \sum_{1\le k\le n} \sum_{1\le j\le k} \frac{1}{j^2}$$

$$= \sum_{1\le j\le n} \frac{n-j+1}{j^2}$$

$$= (n+1)H_n^{(2)} - H_n = (n+1)H_{n+1}^{(2)} - H_{n+1} \quad . \qquad (6)$$

Similarly, the sum of the "convolution" of the series is an interesting exercise, though it is a bit more difficult. We start in the same way as above

$$\sum_{1\le k\le n} H_k H_{n+1-k} = \sum_{1\le j\le n} \frac{1}{j} \sum_{j\le k\le n} H_{n+1-k}$$

$$= \sum_{1\le j\le n} \frac{1}{j} \sum_{1\le k\le n+1-j} H_k$$

$$= \sum_{1\le j\le n} \frac{1}{j} \left((n+2-j)H_{n+1-j} - (n+1-j)\right)$$

$$= (n+2) \sum_{1\le j\le n} \frac{H_{n+1-j}}{j} - \sum_{1\le j\le n} H_{n+1-j} - (n+1)H_n + n$$

$$= (n+2) \sum_{1\le j\le n} \frac{H_{n+1-j}}{j} - 2(n+1)(H_{n+1}-1) \quad ,$$

but we have only succeeded in reducing it to another sum, which involves one harmonic number. The best that we can do with this is to reduce it to yet <u>another</u> sum as follows:

$$\sum_{1\le j\le n} \frac{H_{n+1-j}}{j} = \sum_{1\le j\le n} \frac{H_{n-j}}{j} + \sum_{1\le j\le n} m \frac{1}{j(n+1-j)}$$

$$= \sum_{1\le j\le n-1} \frac{H_{n-j}}{j} + \frac{1}{n+1} \sum_{1\le j\le n} \left(\frac{1}{j} + \frac{1}{n+1-j} \right)$$

$$= \sum_{1\le j\le n-1} \frac{H_{n-j}}{j} + 2\,\frac{H_n}{n+1} \quad .$$

Now the first term on the left is the same as the expression on the right, except with n reduced by one. We can therefore iterate this equation again and again (n times) to get the identity

$$\sum_{1\le j\le n} \frac{H_{n+1-j}}{j} = 2 \sum_{1\le k\le n} \frac{H_k}{k+1} \quad .$$

(This technique of evaluating a sum by identifying and re-expressing the last term and then "telescoping" back to a different sum is important and should be studied carefully.) Finally, the evaluation of this summation provides a good example of manipulating indices of summation:

$$
\begin{aligned}
2 \sum_{1 \le k \le n} \frac{H_k}{k+1} &= 2 \sum_{1 \le k \le n+1} \frac{H_k}{k} - 2 \sum_{1 \le k \le n+1} \frac{1}{k^2} \\
&= 2 \sum_{1 \le k \le n+1} \sum_{1 \le j \le k} \frac{1}{jk} - 2H_{n+1}^{(2)} \\
&= 2 \sum_{1 \le j \le n+1} \sum_{j \le k \le n+1} \frac{1}{jk} - 2H_{n+1}^{(2)} \\
&= 2 \sum_{1 \le k \le n+1} \sum_{k \le j \le n+1} \frac{1}{kj} - 2H_{n+1}^{(2)} \quad ,
\end{aligned}
$$

where the last two steps were to interchange the order of summation, then rename k as j and j as k. But now consider the second and fourth lines above. In one the inner sum is for $j \le k$, in the other it is for $k \le j$, and the summand is the same. This means that we can get the same answer by simply summing over all j, except we must add another term for $k = j$, which must be counted twice.

$$
\begin{aligned}
2 \sum_{1 \le k \le n} \frac{H_k}{k+1} &= \sum_{1 \le k \le n+1} \left(\sum_{1 \le j \le n+1} \frac{1}{kj} + \sum_{j = k} \frac{1}{kj} \right) - 2H_{n+1}^{(2)} \\
&= H_{n+1}^{2} + H_{n+1}^{(2)} - 2H_{n+1}^{(2)} \\
&= H_{n+1}^{2} - H_{n+1}^{(2)} \quad .
\end{aligned}
$$

Substituting this into the equations derived above, we have our answers

$$\sum_{1\le j\le n} \frac{H_{n+1-j}}{j} = 2 \sum_{1\le k\le n} \frac{H_k}{k+1} = H_{n+1}^2 - H_{n+1}^{(2)} \qquad (7)$$

and

$$\sum_{1\le k\le n} H_k H_{n+1-k} = (n+2)(H_{n+1}^2 - H_{n+1}^{(2)}) - 2(n+1)(H_{n+1}-1) \quad . \qquad (8)$$

Implicit in the evaluation of sums using the "telescoping" technique illustrated above are recurrence relations, which arise frequently in our analysis. Often, when we are trying to find an explicit formula for a sequence $\{x_n\}$ $n = 0,1,\ldots$ we are able to express x_n in terms of x_{n-1}, as follows:

$$x_n = a_n x_{n-1} + b_n \qquad n > 0 \qquad (9)$$

where a_n and b_n are some known quantities, and the "initial value" x_0 is known. This linear, first order recurrence can be solved explicitly. (Recurrences of higher order contain terms involving $x_{n-2}, x_{n-3}, \ldots$, and non-linear recurrences contain terms like x_n^2 or $x_n x_{n-1}$. It is not always possible to solve such recurrences, but they are best attacked with the use of generating functions, as discussed below.) For example if $a_n = 1$, then the recurrence telescopes:

$$\begin{aligned} x_n &= x_{n-1} + b_n \\ &= x_{n-2} + b_{n-1} + b_n \\ &= x_{n-3} + b_{n-2} + b_{n-1} + b_n \\ &\vdots \\ x_n &= x_0 + \sum_{1\le k\le n} b_k \quad . \end{aligned}$$

To solve for general a_n, we transform the recurrence into one that does telescope by dividing both sides by the "summation factor" $\prod_{1 \le j \le n} a_j$:

$$\frac{x_n}{\prod_{1 \le j \le n} a_j} = \frac{x_{n-1}}{\prod_{1 \le j \le n-1} a_j} + \frac{b_n}{\prod_{1 \le j \le n} a_j}$$

$$= x_0 + \sum_{1 \le k \le n} \frac{b_k}{\prod_{1 \le j \le k} a_j}$$

so that the solution to the recurrence (9) is

$$x_n = \prod_{1 \le j \le n} a_j \left(x_0 + \sum_{1 \le k \le n} b_k \prod_{1 \le j \le k} \frac{1}{a_j} \right) . \qquad (10)$$

Of course, this is valid only if $\prod_{1 \le j \le k} \frac{1}{a_j}$ is defined. Other summation factors may sometimes be convenient: another that is often useful is $\prod_{j \ge n+1} a_j$. If $\{a_n\}$ and $\{b_n\}$ are simple sequences, as they often are in our analysis, then this formula is not nearly so formidable as it seems. For example, if $a_n = \frac{n+1}{n}$, then $\prod_{1 \le j \le n} a_j$ is just $n+1$. Such simple summation factors can often be discovered by simply examining the recurrence.

Equally as important as the harmonic numbers in our analysis are the <u>binomial</u> <u>coefficients</u>, which are usually defined by the formula

$$\binom{n}{k} = \frac{n!}{k!(n-k)!} \qquad \text{integers } n \ge k \ge 0 . \qquad (11)$$

This is the number of different ways to choose k objects out of a set of n distinct objects. The table below, the well known Pascal's triangle, shows the value of $\binom{n}{k}$ for small n. The first number on row n is $\binom{n}{0}$ and the last is $\binom{n}{n}$, so for example $\binom{6}{2} = 15$.

$$
\begin{array}{lccccccccccccc}
n \\
0 & & & & & & & 1 \\
1 & & & & & & 1 & & 1 \\
2 & & & & & 1 & & 2 & & 1 \\
3 & & & & 1 & & 3 & & 3 & & 1 \\
4 & & & 1 & & 4 & & 6 & & 4 & & 1 \\
5 & & 1 & & 5 & & 10 & & 10 & & 5 & & 1 \\
6 & 1 & & 6 & & 15 & & 20 & & 15 & & 6 & & 1
\end{array}
$$

The list of interesting identities involving these numbers is endless, and they arise in many, many aspects of mathematical analysis. We shall try to restrict ourselves to the properties of binomial coefficients that are useful in the study of Quicksort.

First, it is usually convenient to work with a less restrictive, more general definition:

$$
\binom{r}{k} = \begin{cases} \dfrac{1}{k!} \displaystyle\prod_{1 \le j \le k} (r-k+j) & \text{integer } k \ge 0 \\ \\ 0 & \text{integer } k < 0 \; . \end{cases} \tag{12}
$$

This is equivalent to the above definition when r is an integer $\ge k$, but it extends the definition to allow any real number as an upper index.

For example $\binom{1/2}{3} = \frac{-1/2}{1} \cdot \frac{-3/2}{2} \cdot \frac{1/2}{3} = \frac{1}{16}$, and $\binom{-1}{k} = (-1)^k$.

Integers are still of most interest, and the above definition now gives values for binomial coefficients for all integers r and k.

The values of $\binom{r}{k}$ that are not zero for $k \le 10$ and $-5 \le r \le 10$ are shown in the following table:

							k					
	$\binom{r}{k}$	0	1	2	3	4	5	6	7	8	9	10
	-5	1	-5	15	-35	70	-126	210	-330	495	-715	1001
	-4	1	-4	10	-20	35	-56	84	-120	165	-220	286
	-3	1	-3	6	-10	15	-21	28	-36	45	-55	66
	-2	1	-2	3	-4	5	-6	7	-8	9	-10	11
	-1	1	-1	1	-1	1	-1	1	-1	1	-1	1
	0	1										
	1	1	1									
r	2	1	2	1								
	3	1	3	3	1							
	4	1	4	6	4	1						
	5	1	5	10	10	5	1					
	6	1	6	15	20	15	6	1				
	7	1	7	21	35	35	21	7	1			
	8	1	8	28	56	70	56	28	8	1		
	9	1	9	36	84	126	126	84	36	9	1	
	10	1	10	45	120	210	252	210	120	45	10	1

Simply by examining this table and the definitions we can deduce many simple properties of the binomial coefficients. For example, we have the special values

$$\binom{r}{0} = 1 \text{ and } \binom{r}{1} = r \quad \text{for all real } r , \tag{13}$$

and for positive integers we have the symmetry property

$$\binom{n}{k} = \binom{n}{n-k} \qquad \text{integer } n \geq 0 \quad . \tag{14}$$

Also for positive integers, the table is nearly half zeros:

$$\binom{n}{k} = 0 \qquad \text{integers } k > n \geq 0 \quad . \tag{15}$$

These are all obvious from the combinatorial interpretation of binomial coefficients (for example, there are no ways to choose 8 objects out of a set of 7 objects) and they also follow immediately from the definition (for example $\prod_{1 \leq j \leq k} (n-k+j)$ will include a 0 factor if $k > n \geq 0$). The last two identities don't hold for negative upper indices, but it is apparent from the table that there is a simple relation between binomial coefficients with negative and positive upper indices. This is

$$\binom{-r}{k} = (-1)^k \binom{r+k-1}{k} \quad , \tag{16}$$

which is easily proved from the definition, by negating each term in the product.

If k is positive and $r \geq 1$, then the upper index will always be greater than the lower index on the right hand side of (16), which says that there are no zeros in the upper half of our table. Another obvious characteristic of the table is that if we add any two adjacent numbers, we get a number on the row below. This "addition formula",

$$\binom{r}{k} + \binom{r}{k+1} = \binom{r+1}{k+1} \qquad (17)$$

holds for all real r, and all integers k, and is easily proved from our definition:

$$\binom{r+1}{k+1} - \binom{r}{k+1} = \frac{1}{(k+1)!}\left(\prod_{1 \le j \le k+1} (r-k+j) - \prod_{1 \le j \le k+1} (r-k-1+j)\right)$$

$$= \frac{1}{(k+1)!}\left((r+1) \prod_{1 \le j \le k} (r-k+j) - (r-k) \prod_{1 \le j \le k} (r-k+j)\right)$$

$$= \frac{1}{k!} \prod_{1 \le j \le k} (r-k+j)$$

$$= \binom{r}{k} \quad .$$

Similarly, we can prove from the definition that

$$\binom{r}{k} = \frac{r}{k}\binom{r-1}{k-1} = \frac{r}{r-k}\binom{r-1}{k} \qquad (18)$$

when no division by zero is involved. It is often convenient to separate out factors in binomial coefficients in this way.

Most of our manipulations with binomial coefficients involve finite summations of some kind. For example, the formula we use most often involves adding numbers in the same column in the table, which produces a number in the next column:

$$\sum_{0 \le k \le n} \binom{k}{m} = \binom{n+1}{m+1} \qquad \text{integers } n,m \ge 0 \quad . \qquad (19)$$

This identity can easily be proved by induction, using the "addition formula" (17). Unfortunately, there is no analogous simple formula for partial sums of a row of the table, although the sum of the entire row is simply evaluated:

$$\sum_k \binom{n}{k} = 2^n \quad . \tag{20}$$

(This is a direct consequence of the Binomial Theorem discussed below.) When a summation involves a product of two binomial coefficients, it can often be reduced to Vandermonde's convolution:

$$\sum_k \binom{r}{k}\binom{s}{m-k} = \binom{r+s}{m} \qquad \text{integer } m \ . \tag{21}$$

This important formula is most easily proved using generating functions and it is left as an example for the discussion of the Binomial Theorem below. It has a simple combinatorial interpretation when r and s are integers: both sides count the number of ways of choosing m cards out of a deck containing r black cards and s red cards. The identity takes on several forms since it is valid for all real r and s. For example, if r is an integer, then we know from equations (14) and (1[illegible]) that $\binom{r}{k} = \binom{r}{r-k} = (-1)^{r-k}\binom{-k-1}{r-k}$. If s is also an integer then we can apply this same equation to all three binomial coefficients in Vandermonde's convolution to get the identity

$$\sum_k \ (-1)^{r-k}\binom{-k-1}{r-k}(-1)^{s-m+k}\binom{-m+k-1}{s-m+k} = (-1)^{r+s-m}\binom{-m-1}{r+s-m} \quad .$$

After cancelling the (-1) factors and replacing k by $k-n-1$, where n is an integer, we have

$$\sum_k \binom{n-k}{r+n+1-k}\binom{-m-n-2+k}{s-m-n-1+k} = \binom{-m-1}{r+s-m} \quad .$$

Then, if we change variables to $m = -m-n-2$, $r = -r-1$, and $s = -s-1$, the equation becomes

$$\sum_k \binom{n-k}{n-r-k}\binom{m+k}{m-s+k} = \binom{m+n+1}{r+s+1} \quad .$$

Now, if $0 \le n-r-k \le n-k$ and $0 \le m-s+k \le m+k$, then equation (14) applies, and we have Vandermonde's convolution on the upper index:

$$\sum_{0 \le k \le n} \binom{n-k}{r}\binom{m+k}{s} = \binom{m+n+1}{r+s+1} \quad \text{integers } n \ge s \ge 0 \text{ ; } m,r \ge 0 \text{ .} \qquad (22)$$

We have seen how to evaluate sums involving binomial coefficients and sums involving harmonic numbers: we also must consider sums involving <u>both</u> binomial coefficients and harmonic numbers. The simplest is

$$\sum_{1 \le k \le n} \binom{k}{m} H_k \quad ,$$

where m is fixed. (Notice that all of the terms for $1 \le k < m$ are 0 : it is often convenient in such sums to include these terms rather than bothering with maintaining the exact lower bound.) We can easily evaluate this sum using the same techniques as above:

$$\begin{aligned}
\sum_{1 \le k \le n} \binom{k}{m} H_k &= \sum_{1 \le k \le n} \binom{k}{m} \sum_{1 \le j \le k} \frac{1}{j} \\
&= \sum_{1 \le j \le n} \frac{1}{j} \sum_{j \le k \le n} \binom{k}{m} \\
&= \sum_{1 \le j \le n} \frac{1}{j} \sum_{0 \le k \le n} \binom{k}{m} - \sum_{1 \le j \le n} \frac{1}{j} \sum_{0 \le k \le j-1} \binom{k}{m} \\
&= \sum_{1 \le j \le n} \frac{1}{j} \binom{n+1}{m+1} - \sum_{1 \le j \le n} \frac{1}{j} \binom{j}{m+1}
\end{aligned}$$

$$= \binom{n+1}{m+1}H_n - \frac{1}{m+1} \sum_{1 \le j \le n} \binom{j-1}{m}$$

$$= \binom{n+1}{m+1}H_n - \frac{1}{m+1}\binom{m}{m+1}$$

$$= \binom{n+1}{m+1}\left(H_n - \frac{n-m}{(m+1)(n+1)} \right)$$

$$= \binom{n+1}{m+1}\left(H_n + \frac{1}{n+1} - \frac{1}{m+1} \right)$$

$$\sum_{1 \le k \le n} \binom{k}{m}H_k = \binom{n+1}{m+1}\left(H_{n+1} - \frac{1}{m+1} \right) \quad . \tag{23}$$

Notice that this checks with Eq. (4) above when $m = 0$.

The analogous sum over the lower index is more difficult to evaluate:

$$\sum_{1 \le k \le n} \binom{n}{k}H_k = \sum_{1 \le k \le n} \binom{n-1}{k}H_k + \sum_{1 \le k \le n} \binom{n-1}{k-1}H_k$$

$$= \sum_{1 \le k \le n-1} \binom{n-1}{k}H_k + \sum_{0 \le k \le n-1} \binom{n-1}{k}H_{k+1}$$

$$= 2 \sum_{1 \le k \le n-1} \binom{n-1}{k}H_k + \sum_{0 \le k \le n-1} \binom{n-1}{k} \frac{1}{k+1}$$

$$= 2 \sum_{1 \le k \le n-1} \binom{n-1}{k}H_k + \frac{1}{n} \sum_{0 \le k \le n-1} \binom{n}{k+1}$$

$$= 2 \sum_{1 \le k \le n-1} \binom{n-1}{k}H_k + \frac{1}{n} (2^n-1) \quad .$$

The summation factor for this first order linear recurrence is 2^n (see Eq. (10)), which leads to the solution:

$$\frac{1}{2^n} \sum_{1 \le k \le n} \binom{n}{k} H_k = H_n - \sum_{1 \le k \le n} \frac{1}{k2^k}$$

Unfortunately, there is no simple exact formula for the remaining sum. But it does converge, since we know from the Taylor expansion of $\ln \frac{1}{1-z}$ (see (34) below) that

$$\sum_{k \ge 1} \frac{1}{k2^k} = \ln 2 \quad ,$$

so

$$\sum_{1 \le k \le n} \frac{1}{k2^k} = \ln 2 + \sum_{k > n} \frac{1}{k2^k} \quad .$$

This remainder term is very small: $0 < \sum_{k>n} \frac{1}{k2^k} = \sum_{k>0} \frac{1}{(k+n)2^{k+n}}$ $< \frac{1}{(n+1)2^n} \sum_{k>0} \frac{1}{2^k} = \frac{1}{2^{n-m}(n+1)}$, and therefore

$$\sum_{1 \le k \le n} \binom{n}{k} H_k = 2^n(H_n - \ln 2) + \epsilon \quad , \quad 0 < \epsilon < \frac{1}{n+1} \quad . \tag{24}$$

Many of the finite summation problems that we have encountered above become much easier to understand if we recognize the correspondence between the discrete quantities with which we are dealing and more familiar concepts in differential calculus. For example, just as we have defined

$$H_n = \sum_{1 \le k \le n} \frac{1}{k} \quad ,$$

the natural logarithms can be defined by

$$\ln x = \int_1^x \frac{1}{t}\, dt \quad .$$

This duality between summation and integration is present in many of the formulas above. If we rewrite Eq. (4) in the form

$$\sum_{0 \le k < n} H_k = n(H_n - 1)$$

then we should not be surprised by the result since we know that

$$\int_0^x \ln t\, dt = x(\ln x - 1) \quad .$$

Similarly, Eq. (5) is the discrete analog of

$$\int_0^x (\ln t)^2\, dt = x(\ln x)^2 - 2x \ln x + 2x \quad ,$$

though this formula shows that we can't always rely upon an exact analog. The reason for this in this case is that powers do not carry the correspondence. This is where binomial coefficients fit in. The discrete analog of t^m is

$$k^{\underline{m}} = m!\binom{k}{m}$$

the "falling factorial" powers. Now Eq. (19) can be rewritten as

$$\sum_{0 \le k < n} m!\binom{k}{m} = m!\binom{n+1}{m+1} = \frac{1}{m+1}\,(m+1)!\binom{n+1}{m+1}$$

or

$$\sum_{0 \le k < n} k^{\underline{m}} = \frac{n^{\underline{m+1}}}{m+1} \quad ,$$

which is analogous to

$$\int_0^x t^m\, dt = \frac{x^{m+1}}{m+1} \quad .$$

In fact, if we extend the definition of factorial power to negative integers by preserving the property $x^{\underline{n}} = x(x-1)^{\underline{n-1}}$, then we find that

$$x^{\underline{-n}} = \frac{1}{n!\binom{x+n}{n}}$$

and the equation

$$\sum_{1\le k<n} k^{\underline{m}} = \frac{n^{\underline{m+1}}}{m+1}$$

holds for all m except $m = -1$, when we get a harmonic number, just as we get a logarithm in the continuous case. Finally, the discrete analog to differentiation is the difference operator Δ , defined by:

$$\Delta f(n) = f(n+1) - f(n) \quad . \qquad (25)$$

As we might expect, it turns out that,

$$\Delta n^{\underline{k}} = kn^{\underline{k-1}} \quad \text{or} \quad \Delta\binom{n}{k} = \binom{n}{k-1} \quad ,$$

$$\Delta H_n = n^{\underline{-1}} \quad ,$$

etc.

It is fascinating to pursue this correspondence between finite difference calculus and differential calculus: it covers a surprisingly wide range. For example, the discrete analog to e must be 2 , since $\Delta 2^k = 2^k$ and, by 20 , $\sum_{k\ge 0} \frac{n^{\underline{k}}}{k!} = 2^n$. These formal manipulations can even be used to suggest solutions to problems that we might not otherwise find. In any case, we need not understand it completely for it to reinforce our intuition. We should be able to evaluate a sum which corresponds to an elementary integral; we may have difficulty evaluating one that is not.

Because factorial powers occur "naturally" in the finite difference calculus, it is often convenient to be able to convert from them to regular powers. In other words, if we treat a binomial coefficient as a polynomial in the variable in the upper index (of degree given by the lower index) we want to know the coefficients of that polynomial. These coefficients are called <u>Stirling numbers of the first kind</u> and are defined by the formula

$$k!\binom{n}{k} = \sum_j (-1)^{k-j}\begin{bmatrix} k \\ j \end{bmatrix} n^j \quad . \tag{26}$$

(The coefficient $(-1)^{k-j}$ is included to make the Stirling numbers all positive.) There are Stirling numbers of the second kind to convert from powers to binomial coefficients. Since the polynomial $n(n-1)\ldots(n-k+1)$ has a constant term if and only if $k = 0$, we know that

$$\begin{bmatrix} k \\ 0 \end{bmatrix} = \begin{bmatrix} 0 \\ k \end{bmatrix} = \delta_{0k} \quad . \tag{27}$$

Also, we can use the fact that $k!\binom{n}{k} = (k-1)!\binom{n}{k-1}(n-k+1)$ to develop an "addition formula" analogous to Eq. (17):

$$\begin{aligned}
\sum_j (-1)^{k-j}\begin{bmatrix} k \\ j \end{bmatrix} n^j &= (n-k+1)\sum_j (-1)^{k-1-j}\begin{bmatrix} k-1 \\ j \end{bmatrix} n^j \\
&= \sum_j (-1)^{k-1-j}\begin{bmatrix} k-1 \\ j \end{bmatrix} n^{j+1} - \sum_j (k-1)(-1)^{k-1-j}\begin{bmatrix} k-1 \\ j \end{bmatrix} n^j \\
&= \sum_j (-1)^{k-j}\begin{bmatrix} k-1 \\ j-1 \end{bmatrix} n^j + \sum_j (-1)^{k-j}(k-1)\begin{bmatrix} k-j \\ j \end{bmatrix} n^j \quad ,
\end{aligned}$$

and setting coefficients of n^j equal we get

$$\left[{k \atop j}\right] = \left[{k-1 \atop j-1}\right] + (k-1)\left[{k-1 \atop j}\right] \quad . \tag{28}$$

From these formulas, we can build up a table of the Stirling numbers:

					j			
	$\left[{k \atop j}\right]$	0	1	2	3	4	5	6
	0	1	0	0	0	0	0	0
	1	0	1	0	0	0	0	0
	2	0	1	1	0	0	0	0
k	3	0	2	3	1	0	0	0
	4	0	6	11	6	1	0	0
	5	0	24	50	35	10	1	0
	6	0	120	274	225	85	15	1

and from this table, we can guess some more special values which are easily proved from the definition:

$$\left[{k \atop k}\right] = 1 \quad ; \quad \left[{k \atop k-1}\right] = \binom{k}{2} \quad ; \text{ and } \quad \binom{k}{1} = (k-1)! \quad \text{for } k > 0 \ . \tag{29}$$

Also, we have the non-trivial identity

$$\left[{k \atop 2}\right] = (k-1)!H_{k-1} \quad , \tag{30}$$

This follows from (28), since $\left[{k \atop 2}\right] = \left[{k-1 \atop 1}\right] + (k-1)\left[{k-1 \atop 2}\right] =$ $(k-2)! + (k-1)\left[{k-1 \atop 2}\right]$. After dividing by the summation factor $(k-1)!$ we get the recurrence

$$\frac{\left[{k \atop 2}\right]}{(k-1)!} = \frac{\left[{k-1 \atop 2}\right]}{(k-2)!} + \frac{1}{k+1} \quad ,$$

and (30) follows immediately.

We often encounter more complex recurrence relations and summations than we have discussed so far, and we need more powerful tools for dealing with them. Probably the most important such tool is the generating function. The idea is to represent a sequence $\langle a_k \rangle$ by a single function, the generating function, defined by the formula

$$A(z) = \sum_{k \geq 0} a_k z^k \quad .$$

For example, the infinite series expansions that we know from Taylor's Theorem lead to the generating functions for many sequences. The generating function for $\left\langle \frac{1}{k!} \right\rangle$ is e^z since

$$e^z = \sum_{k \geq 0} \frac{z^k}{k!} \quad , \tag{31}$$

and the generating function for $\left\langle \frac{1}{k} \right\rangle$ is $\ln\left(\frac{1}{1-z}\right)$ since

$$\ln\left(\frac{1}{1-z}\right) = \sum_{k \geq 1} \frac{z^k}{k} \quad . \tag{32}$$

The generating function for the binomial coefficients is given by the well-known Binomial Theorem:

$$(1+z)^n = \sum_{k \geq 0} \binom{n}{k} z^k \quad . \tag{33}$$

This important equation is easily proved by induction, using (17), or by a simple combinatorial argument. If we apply (16) to this we get

$$(1+z)^n = \sum_{k \geq 0} (-1)^k \binom{-n+k-1}{k} z^k$$

or

$$\frac{1}{(1-z)^{n+1}} = \sum_{k \geq 0} \binom{n+k}{k} z^k$$

$$= \sum_{k \geq n} \binom{k}{n} z^{k-n}$$

which gives

$$\frac{z^n}{(1-z)^{n+1}} = \sum_{k \geq 0} \binom{k}{n} z^k \quad , \tag{34}$$

the generating function for the binomial coefficients with the lower index fixed.

Important special cases of this last formula are

$$\frac{1}{1-z} = \sum_{k \geq 0} z^k \tag{35}$$

and

$$\frac{z}{(1-z)^2} = \sum_{k \geq 0} k\, z^k \quad . \tag{36}$$

Similarly the generating function for the Stirling numbers of the first kind follows directly from the definition (26):

$$\prod_{1 \leq j \leq n} (z+j-1) = \sum_{k \geq 0} \left[{n \atop k} \right] z^k \quad . \tag{37}$$

Generating functions are useful because they provide such a compact representation of sequences. Also, simple algebraic manipulations with generating functions often correspond to nontrivial transformations on the sequences that they represent. For example, the Binomial Theorem (33) leads to an easy proof of Vandermonde's convolution (21):

$$\begin{aligned}(1+z)^r(1+z)^s &= \sum_{k\geq 0}\binom{r}{k}z^k \sum_{m\geq 0}\binom{s}{m}z^m \\ &= \sum_{k\geq 0}\binom{r}{k} \sum_{m\geq k}\binom{s}{m-k}z^m \\ &= \sum_{m\geq 0} \quad \sum_{0\leq k\leq m}\binom{r}{k}\binom{s}{m-k}z^m\end{aligned}$$

and also

$$(1+z)^r(1+z)^s = (1+z)^{r+s} = \sum_{m\geq 0}\binom{r+s}{m}z^m \quad .$$

For these two infinite series to be identical, all of the coefficients must be equal, and setting coefficients of z^m equal leads to (21). It is true in general that convoluting sequences corresponds to multiplying generating functions: if $A(z)$ is the generating function for $\langle a_k \rangle$ and $B(z)$ is the generating function for $\langle b_k \rangle$, then by a manipulation just like the one above $A(z)B(z)$ is the generating function for $\left\langle \sum_{0\leq j\leq k} a_j b_{k-j} \right\rangle$:

$$A(z)B(z) = \sum_{k\geq 0}\left(\sum_{0\leq j\leq k} a_j b_{k-j}\right)z^k \quad \text{when} \quad A(z) = \sum_{k\geq 0} a_k z^k$$

$$\text{and} \quad B(z) = \sum_{k\geq 0} b_k z^k \quad . \qquad ([illegible])$$

A very important special case of this occurs when $B(z) = \frac{1}{1-z}$. Then we see that $\frac{1}{1-z} A(z)$ is the generating function for $\left\langle \sum_{0 \le j \le k} a_j \right\rangle$.

For example, we could have used this to derive (36) from (35). As another example, we get the generating function for the harmonic numbers from (36):

$$\frac{1}{1-z} \ln \frac{1}{1-z} = \sum_{k \ge 1} H_k z^k \quad . \tag{39}$$

Conversely, if $a_0 = 0$ it is easy to see that $\frac{1-z}{z} A(z)$ is the generating function for $\{\Delta a_k\}$:

$$\frac{1-z}{z} A(z) = \sum_{k \ge 0} \Delta a_k z^k \quad \text{when} \quad A(z) = \sum_{k \ge 1} a_k z^k \quad . \tag{40}$$

It is also useful to manipulate generating functions by differentiation and integration. Differentiating both sides of (35) leads to (36); and integrating both sides of (35) gives (39). In general, if $A(z)$ is the generating function for $\langle a_k \rangle$, then $zA'(z)$ is the generating function for $\langle ka_k \rangle$. Furthermore, we can take multiple derivatives to see that $\frac{z^m A^{(m)}(z)}{m!}$ is the generating function for $\left\langle \binom{k}{m} a_k \right\rangle$:

$$\frac{z^m A^{(m)}(z)}{m!} = \sum_{k \ge 0} \binom{k}{m} a_k z^k \quad \text{when} \quad A(z) = \sum_{k \ge 0} a_k z^k \quad . \tag{41}$$

For example, it is interesting to apply this to (39):

$$\frac{z^m}{m!} \left(\frac{1}{1-z} \ln \frac{1}{1-z} \right)^{(m)} = \sum_{n \ge 0} \binom{n}{m} H_n z^n \quad .$$

The evaluation of this derivative is a little tricky, but the pattern develops readily:

$$\left(\frac{1}{1-z}\ln\frac{1}{1-z}\right)' = \frac{1}{(1-z)^2}\left(\ln\frac{1}{1-z}+1\right)$$

$$\frac{1}{2}\left(\frac{1}{1-z}\ln\frac{1}{1-z}\right)'' = \frac{1}{(1-z)^3}\left(\ln\frac{1}{1-z}+\frac{3}{2}\right)$$

$$\frac{1}{6}\left(\frac{1}{1-z}\ln\frac{1}{1-z}\right)''' = \frac{1}{(1-z)^4}\left(\ln\frac{1}{1-z}+\frac{11}{6}\right)$$

$$\vdots$$

$$\frac{1}{m!}\left(\frac{1}{1-z}\ln\frac{1}{1-z}\right)^{(m)} = \frac{1}{(1-z)^{m+1}}\left(\ln\frac{1}{1-z}+H_m\right) \quad .$$

This leaves us with the identity

$$\frac{z^m}{(1-z)^{m+1}}\ln\frac{1}{1-z}+\frac{z^m}{(1-z)^{m+1}}H_m = \sum_{n\geq 0}\binom{n}{m}H_n z^n \quad . \tag{42}$$

Now, the second term on the left is just (34), the generating function for the binomial coefficients, and the first term is the convolution of (32) and (34), so we have

$$\sum_{n\geq 1}\ \sum_{0\leq k<n}\binom{k}{m}\frac{1}{n-k}z^k + \sum_{n\geq 0}\binom{n}{m}H_m z^n = \sum_{n\geq 0}\binom{n}{m}H_n z^n \quad ,$$

and setting coefficients of z^n equal tells us that

$$\sum_{0\leq k<n}\binom{k}{m}\frac{1}{n-k} = \binom{n}{m}(H_n - H_m) \quad . \tag{43}$$

There is still more information in this function since we can factor the term $\frac{z^m}{(1-z)^{m+1}}\ln\frac{1}{1-z}$ in Eq. [illegible] differently to get a different convolution:

$$\frac{z^m}{(1-z)^{m+1}}\ln\frac{1}{1-z} = \frac{z^{m-1}}{(1-z)^m}\left(\frac{z}{1-z}\ln\frac{1}{1-z}\right) = \sum_{n\ge1}\sum_{1\le k\le n}\binom{k}{m-1}H_{n-k}$$

and we therefore also know the sum for the convolution of the harmonic numbers with the binomial coefficients:

$$\sum_{1\le k\le n}\binom{k}{m-1}H_{n-k} = \binom{n}{m}(H_n-H_m) \quad .$$

Proceeding still further, if we multiply both sides of [illegible] by $\frac{z^{p-1}}{(1-z)^{p+1}}$, we can get a convolution on the right hand side:

$$\frac{z^{m+p-1}}{(1-z)^{m+p+2}}\ln\frac{1}{1-z} - \frac{z^{m+p-1}}{(1-z)^{m+p+2}}H_m = \sum_{n\ge1}\sum_{1\le k\le n}\binom{k}{m}\binom{n-k}{p}H_k z^{n-1} \quad .$$

But from [illegible] and the preceding equation, we know that

$$\frac{z^{m+p-1}}{(1-z)^{m+p+2}}\ln\frac{1}{1-z} = \sum_{n\ge1}\binom{n}{m+p+1}(H_n-H_{m+p+1})z^n$$

so that after applying [illegible] to $\frac{z^{m+p-1}}{(1-z)^{m+p+2}}H_m$ and setting coefficients of z^n equal, we have the identity:

$$\sum_{1\le k\le n}\binom{k}{m}\binom{n-k}{p}H_k = \binom{n+1}{m+p+1}(H_{n+1}-H_{m+p+1}+H_m) \quad .$$

This family of useful identities would be very difficult to derive without the use of generating functions, and they illustrate the power of generating functions as a tool for the evaluation of summations.

Generating functions are also useful for solving recurrence relations. As an example, suppose we have a sequence $\langle a_n \rangle$ defined by

$$a_0 = 0 \quad ; \quad a_1 = 2 \quad ; \quad a_n = 4a_{n-1} - 4a_{n-2} \qquad n > 1 \quad .$$

To solve this problem using generating functions, we first assume that $a_n = 0$ for $n < 0$, and then write down a single equation that holds for all n :

$$a_n = 4a_{n-1} - 4a_{n-2} + 2\delta_{n1} \quad .$$

Now, multiply both sides of the equation by z^n and sum over all n :

$$\sum_n a_n z^n = 4 \sum_n a_{n-1} z^n - 4 \sum_n a_{n-2} z^n + 2 \sum_n \delta_{n1} z^n$$

$$= 4z \sum_n a_n z^n - 4z^2 \sum_n a_n z^n + 2z \quad .$$

This is now a simple equation which can be algebraically solved for the generating function

$$A(z) = \frac{2z}{1 - 4z + 4z^2}$$

$$= \frac{2z}{(1-2z)^2}$$

$$= \sum_{n \geq 0} n\,(2z)^n \quad .$$

Therefore, setting coefficients of z^n equal, we have shown that $a_n = n2^n$.

This same technique can be extended to derive a general solution for linear recurrence relations with constant coefficients. It is also appropriate to use generating functions to attack more difficult problems such as non-linear recurrences or recurrences with non-constant coefficients, although more complex functional equations will be involved. There are many examples of this in the text.

When generating functions are used to represent probability distributions, they have several other useful properties. Suppose that $\langle p_k \rangle$ represents the probability that some random variable X defined in nonnegative integers takes on the value k. In particular, the sequence has the properties

$$p_k \geq 0 \qquad\qquad k = 0,1,2,\ldots$$

and

$$\sum_{k \geq 0} p_k = 1 \quad .$$

Then the "probability" generating function

$$P(z) = \sum_{k \geq 0} p_k z^k$$

is a very useful way of describing the distribution of values of X. First, notice that $P(1) = 1$ by definition. Next, the average value of X is simply

$$\sum_{k \geq 0} k\, p_k = P'(1) \tag{46}$$

and the variance is

$$\sum_{k \geq 0} (k - P'(1))^2 p_k = P''(1) + P'(1) - (P'(1))^2 \quad . \tag{47}$$

These formulas are especially significant because, if we are interested only in knowing the average and standard deviation, then we need not know the specific values of $\langle p_k \rangle$ if we can find the generating function. Higher moments of the distribution can be obtained in a similar manner.

Suppose that we have two probability generating functions $P(z) = \sum_{k \geq 0} p_k z^k$ and $Q(z) = \sum_{k \geq 0} q_k z^k$. Consider the probability generating function formed by taking the product of these two:

$$R(z) = P(z)Q(z) = \sum_{n \geq 0} \sum_{0 \leq k \leq n} p_k q_{n-k} z^n \quad .$$

Then the mean of the distribution described by $R(z)$ is

$$\begin{aligned} R'(1) &= \sum_{n \geq 0} \sum_{0 \leq k \leq n} n\, p_k q_{n-k} \\ &= \sum_{k \geq 0} \sum_{n \geq k} n\, p_k q_{n-k} \\ &= \sum_{k \geq 0} \sum_{n \geq 0} (n+k) p_k q_n \\ &= \sum_{k \geq 0} p_k \sum_{n \geq 0} n\, q_n + \sum_{k \geq 0} k\, p_k \sum_{n \geq 0} q_n \end{aligned}$$

$$R'(1) = Q'(1) + P'(1) \quad , \tag{48}$$

or the mean of the product is the sum of the means. Similarly the variance of the product is calculated by first finding the second derivative evaluated at 1 :

$$R''(1) = \sum_{n \ge 0} \sum_{0 \le k \le n} n(n-1) p_k q_{n-k}$$

$$= \sum_{k \ge 0} \sum_{n \ge 0} (n+k)(n+k-1)\, p_k q_n$$

$$= \sum_{k \ge 0} \sum_{n \ge 0} (n(n-1) + k(k-1) + 2nk)\, p_k q_n$$

$$= {}_{k \ge 0}\, p_k \sum_{n \ge 0} n(n-1) q_n + \sum_{k \ge 0} k(k-1) p_k \sum_{n \ge 0} q_n + 2 \sum_{k \ge 0} k\, p_k \sum_{n \ge 0} n\, q_n$$

$$= Q''(1) + P''(1) + 2P'(1)Q'(1)$$

so that the variance of the product is

$$R''(1) + R'(1) - (R'(1))^2$$

$$= Q''(1) + P''(1) + 2P'(1)Q'(1) + P'(1) + Q'(1) - (P'(1) + Q'(1))^2$$

$$= Q''(1) + Q'(1) - (Q'(1))^2 + P''(1) + P'(1) - (P(1))^2 \quad , \qquad (49)$$

the sum of the variances. This property of probability generating functions (actually we have not even used the fact that the coefficients are nonnegative) is very important in a practical sense because it says that we can calculate the mean and variance of distributions described by complex generating functions by expressing the generating function as a product of simpler generating functions, then summing their means and variances.

In all of the methods involving generating functions, we have stressed formal manipulations of power series, without regard to the fact that the manipulations are valid only if the series converges for some value of the argument z . We will ignore questions of convergence because we are using generating functions only as a problem solving tool.

It is not worthwhile to concern ourselves with convergence because we can nearly always use induction to prove that our answers are correct. In fact, it is occasionally the case, as we have seen in our brief look at the finite difference calculus, that what seem to be invalid formal manipulations lead us to a correct answer. This does not occur in the study of Quicksort, however, and all of the power series dealt with in the text do behave properly.

Most of the analysis in the text is concerned with finding the values of some function X_N, where N is the size of the file being sorted. In most of the problems, we are able to get simple exact expressions for X_N, but in a few cases the exact answers are too complex and it is necessary to resort to asymptotic representations, where we derive an answer that is very close to X_N for large values of N. (The methods that we use have the property that we can theoretically extend the accuracy as much as we might desire.) Because such cases are rare in the analysis of Quicksort, we shall only lightly treat this topic. A much fuller explanation may be found in Knuth ([16], Section 1.2.11).

The basis for asymptotic analysis is the "O" notation, which enables us to conveniently suppress details of the approximation. When the notation $O(f(N))$ appears on the right hand side of an equation, it means that the quantity X_N represented by $O(f(N))$ satisfies the inequality $|X_N| < M|f(N)|$ for some constant M, when N is larger than some constant N_0. For example, the tail end of any absolutely convergent power series can be bounded by a constant so that, for example, we have

$$\ln\left(1 + \frac{1}{N}\right) = \frac{1}{N} - \frac{1}{2N^2} + \dots + \frac{(-1)^{k+1}}{k\,N^k} + O\left(\frac{1}{N^{k+1}}\right) \qquad (50)$$

and

$$e^{1/N} = 1 + \frac{1}{N} + \frac{1}{2N^2} + \ldots + \frac{1}{k!N^k} + O\left(\frac{1}{N^{k+1}}\right) \quad . \tag{51}$$

The notation is convenient because it allows us to express the error involved in our approximations in one term, omitting all the details. There are several elementary properties which are easily proven from the definition: for example we can replace $O\left(\frac{1}{N}\right) + \frac{1}{N}$ by $O\left(\frac{1}{N}\right)$; $NO\left(\frac{1}{N^2}\right)$ by $O\left(\frac{1}{N}\right)$; $O\left(\frac{1}{N}\right) + O\left(\frac{1}{N^2}\right)$ by $O\left(\frac{1}{N}\right)$; etc. We are generally content, from a practical standpoint, to get answers accurate to within $O\left(\frac{1}{N}\right)$, since this means that if N is bigger than 1000 or so, the answer will probably be accurate to within a few decimal places.

In addition to the power series described above, we use only some very simple asymptotic formulas. We have already referred to one such formula,

$$H_N = \ln N + \gamma + \frac{1}{2N} + O\left(\frac{1}{N^2}\right) \quad , \tag{52}$$

and a few others are used in the text:

$$H_N^{(2)} = \frac{\pi^2}{6} - \frac{1}{N} + O\left(\frac{1}{N^2}\right) \quad , \tag{53}$$

$$\sum_{i \geq N} \frac{1}{i^a} = \frac{1}{(a-1)N^{a-1}} + \frac{1}{2N^a} + O\left(\frac{1}{N^{a+1}}\right) \quad , \tag{54}$$

and

$$N! = \sqrt{2\pi N}\left(\frac{N}{e}\right)^N\left(1 + O\left(\frac{1}{N}\right)\right) \quad . \tag{55}$$

The formulas (52), (54), and (55) follow directly from the Euler-McLaurin summation formula which is an important equation that provides a quantitative correspondence between finite sums and definite integrals -- see Knuth ([16], Section 1.2.11). Equation (33) then follows directly from (3) and (54).

INDEX TO NOTATIONS

$\sum_{1 \le k \le n} a_k$	$a_1 + a_2 + \dots + a_n$	$\sum_{R(k)} a_k$ is the sum of a_k for all integers k for which $R(k)$ is true. If none, it is defined to be 0.
$\prod_{1 \le k \le n} a_k$	$a_1 a_2 \dots a_n$	$\prod_{R(k)} a_k$ is the product of a_k for all integers k for which $R(k)$ is true. If none, it is defined to be 1.
H_n	$\sum_{1 \le k \le n} \frac{1}{k}$	the n-th Harmonic number.
$H_n^{(2)}$	$\sum_{1 \le k \le n} \frac{1}{k^2}$	
$\langle a_n \rangle$	$a_0, a_1, a_2, \dots$	the sequence a_n.
$n!$	$\prod_{1 \le k \le n} k$	n factorial.
$\binom{n}{k}$	See Eq. 12	binomial coefficient: "n things taken k at a time".
$n^{\underline{k}}$	$k!\binom{n}{k} \quad k \ge 0$	
$n^{\underline{-k}}$	$\frac{1}{k!\binom{n+k}{k}} \quad k \ge 0$	falling factorial powers.
$\Delta f(n)$	$f(n+1) - f(n)$	finite difference operator.

$\left[{n \atop k} \right]$	See Eq. (26)	Stirling number of the first kind.
e	$\sum_{k \geq 0} \frac{1}{k!}$	base of natural logarithms.
$\ln x$	$\log_e x$	natural logarithm.
$\lg x$	$\log_2 x$	binary logarithm (to the base 2).
γ	$\lim_{n \to \infty} (H_n - \ln n)$	Euler's constant.
$\lvert x \rvert$	$-x$ if $x < 0$; x otherwise	absolute value.
$\lfloor x \rfloor$	greatest integer $\leq x$	floor function.
$\lceil x \rceil$	least integer $\geq x$	ceiling function.
$\{x\}$	$x - \lfloor x \rfloor$	fractional part.
$x \bmod y$	$x - y \left\lfloor \frac{x}{y} \right\rfloor$	the remainder when x is divided by y .
δ_{ij}	1 if $i = j$, 0 otherwise.	Kronecker delta function.
$O(f(n))$	See comments preceding Eq. (52)	notation for asymptotic approximation.

$$\pi = 3.1415926535$$

$$\pi^2 = 9.8696044010$$

$$\ln 2 = \frac{1}{\lg e} = 0.6931471805$$

$$\lg e = \frac{1}{\ln 2} = 1.442695040888963407359924681001$$

$$\gamma = 0.5772156649$$

APPENDIX C

In this Appendix we shall examine some of the issues involved when it comes to programming Quicksort in real programming languages and running it on a real computer. Of course we cannot treat this topic exhaustively, but the few examples studied will bring out a number of important issues. We shall study implementations of Program 2.4 in ALGOL W and FORTRAN H for the IBM S/360, and we shall look at some compiled code for these programs and compare it with an efficient assembly language implementation of the inner loop. Some final comments are included on the conditions under which the programs that we have studied are appropriate.

To begin, let us consider the implementation of Program 2.4 in a *real* "Algol-like" language which is like the mythical Algol-like language used in the text: namely ALGOL W. (This language is described by R. L. Sites, "ALGOL W Reference Manual", Stanford University Computer Science Report STAN-CS-71-230.) There are two main problems which arise. First, there is no "exchange" statement, and we must replace exchanges such as " A[i] :=: A[j] " with three assignments: " t := A[i]; A[i] := A[j]; A[j] := t ". Second, there is no control construct as general as *loop* ... *while* ... *repeat* , and our various loops must be implemented using the simple *for* or *while* statements, or *go to* statements.

There are four loops to consider in the partitioning phase: the outermost loop, which terminates when the stack is empty; the partitioning loop, which terminates when the i and j pointers cross; and the two innermost scanning loops, which involve the actual key comparisons. The scanning loops are simple repeat constructs and can be programmed using a standard transformation:

```
          loop  i := i+1 while A[i] < v repeat;
```

is exactly equivalent to

```
          i := i+1; while A[i] < v do i := i+1;
```

in ALGOL W. The partitioning loop is an example of a loop which is performed " n and a half " times (see [ ]): when it is implemented with a while statement in ALGOL W, it results in an extraneous test on entry to the loop and an extra exchange on exit. We could eliminate the former by jumping into the loop; and the following implementation of Partitioning Method 2.4 shows that the exchange is not too costly.

```
     i := ℓ; j := r+1; r := A[ℓ];
     while i < j do begin
                    i := i+1; while A[i] < v do i := i+1;
                    j := j-1; while A[j] > v do j := j-1;
                    t := A[j]; A[j] := A[i]; A[i] := t;
                    end;
     A[i] := A[j]; A[j] := A[ℓ]; A[ℓ] := t;
```

(The last three assignments implement " A[i] :=: A[j] ", to undo the extra exchange, followed by " A[j] :=: A[ℓ] ".) The fourth loop, the outermost, is best implemented using a go to structure in the manner of the assembly language implementation given in Appendix B. Otherwise, since ALGOL W doesn't have "event variables" we would find ourselves testing to see if the stack is empty even on occasions when we haven't touched it.

The following four pages are the listing of a run of an ALGOL W program embodying these ideas. The first page is a listing of the program, which includes code to initialize the array A to a sequence of 400 pseudo-random numbers and to print them out. (The value $M = 9$ is used as the cutoff for small subfiles: the best value must be determined from the compiled code.) The second and third pages are the unsorted and sorted sequences as printed out by the program, and the fourth page shows the instruction frequency counts for this run of the program. These can be checked with the formulas for the expected values of the various quantities derived in Chapter 3. For example, the program used 2859 comparisons in the scanning loops and Chapter 3 tells us to expect about $401(H_{401} - H_{11} + 1) \approx 3249$ comparisons. The various other quantities can be similarly checked, though we haven't yet looked at the coefficients of any of the quantities.

```
STANFORD ALGOL W (12DEC74)                                              08 MAY 1975 @ 13:56          PAGE   1

0000 1-      BEGIN INTEGER N,M;
0002 --      N:=400;  M:=9;  INTOVFL:=NULL;
0005 --
0005 2-      BEGIN INTEGER ARRAY A(0::N+1);
0007 --            INTEGER ARRAY STACK(0::2*(ENTIER(LN((N+1)/(M+2))))+1);
0008 --            INTEGER P,L,R,I,J,V,T;
0009 --      A(0):=0;
0010 --      FOR I:=1 UNTIL N DO
0010 3-          BEGIN
0011 --          A(I):=A(I-1)*314159269+453806245;
0012 --          IF A(I)<0 THEN A(I):=(A(I)+2147483647)+1;
0013 -3          END;
0014 --      A(N+1):=2147483647;
0015 --      FOR I:=1 UNTIL N DO WRITEON(A(I));  IOCONTROL(3);
0017 --
0017 --                   P:=0; L:=1; R:=N;
0020 --      PARTITION:   I:=L; J:=R+1; V:=A(L);
0023 3-                  WHILE I<J DO BEGIN
0024 --                               I:=I+1; WHILE A(I)<V DO I:=I+1;
0026 --                               J:=J-1; WHILE A(J)>V DO J:=J-1;
0028 --                               T:=A(J); A(J):=A(I); A(I):=T;
0031 -3                              END;
0032 --                  A(I):=A(J); A(J):=A(L); A(L):=T;
0035 --                  IF R-J>J-L THEN GO TO RBIG;
0036 --                  IF J-L<=M THEN GO TO POP;
0037 --                  IF R-J<=M THEN GO TO LEFT;
0038 --                  P:=P+2;
0039 --                  STACK(P):=L;
0040 --                  STACK(P+1):=J-1;
0041 --      RIGHT:       L:=J+1;
0042 --                  GO TO PARTITION;
0043 --      RBIG:        IF R-J<=M THEN GO TO POP;
0044 --                  IF J-L<=M THEN GO TO RIGHT;
0045 --                  P:=P+2;
0046 --                  STACK(P):=J+1;
0047 --                  STACK(P+1):=R;
0048 --      LEFT:        R:=J-1;
0049 --                  GO TO PARTITION;
0050 --       POP:        L:=STACK(P);
0051 --                  R:=STACK(P+1);
0052 --                  P:=P-2;
0053 --                  IF P>=0 THEN GO TO PARTITION;
0054 --       INSERTION:  FOR I:=2 UNTIL N DO
0054 3-                          BEGIN
0055 --                          V:=A(I); J:=I-1;
0057 44                          WHILE A(J)>V DO BEGIN A(J+1):=A(J); J:=J-1; END;
0061 --                          A(J+1):=V;
0062 -3                          END;
0063 --
0063 --       FOR I:=1 UNTIL N DO WRITEON(A(I));
0064 -2       END;
0065 -1       END.

EXECUTION OPTIONS: DEBUG,2  NOCHECK TIME=10 SECONDS PAGES=30

000.55 SECONDS IN COMPILATION, (01880, 02232) BYTES OF CODE GENERATED
```

```
 453806245   21957374   404438619  1430079820  2111637153  1265204842   193848055  2089940504
 552212125  177192726  1610076627  1100420004  1858403161  1377876226  1606887151   837021872
1301246101 1822804078   773484327  1221017340   987848977  1481951386   518370279   817319048
1673187021  445642310   198222275   972678484  1752897481  1694371122  1761569247   278664544
1354256261  699366302   668854523   700328108   987692929   313099018  1015949527     2075448
 547644349 1200459942   653482419  1810789556   769521721   687191394  1710215375   566455824
1664973045 2011229838  1671817771   621610588    21815793   102291706  2129019335   964476392
2007658285 1777902886   232425123   534929588  1417977001  1482318706  2115460031   708115136
1694683411  368424894  1426888219  1927206524  1373767777   664248106  1452080823  1128891544
 107970205 1537777302   621572033  1974062180  1782973721   784335298   657787567  1577827184
 493443333 1966568942  1119036539   111058364   546665169    89667530  2053201831  1262550856
1450873357  299539706  1524405123   541895700  1698247049   418948594   357031327   279097376
 874384709 1281852958   229153275  1056397164    27140417  1993189258  1468163223  1985883640
1111457277  876670838  2092112243   370540484   330637817  1696046626   714003599  1687476432
 307971765   55316974   537576643   366216476   426972081  1286269370    71958023   282857640
 499041517 1739154790  1403454307  1083230068   206624361  1919917106  1737027455  2075997568
 440977269  555205502  1660693979  1348445900   214649377    14876650   560355959  1043703640
 893772893  256747670  1013403987   747111204   501436171  1637124738  1526017647   657901104
 735358357  940922358   582503883   788872316  1889387665  1722346010  1739017063   690352648
 783222733  341737414  1977911779   973517524  1065595481  1146650290   299012447   391940832
1525388069 1409290974   176116155   493864492   975274753  1517267018  2050023511   866644152
1086402365  354547766   290003251  1280824964     8948665  1305396962   123738191   197755792
1903252597 1706387470   245381547  1653228508   785220977  1911175802  1700927815   653697896
 790363365  151495590  1644729123  2047241780  2022264873    38429458   354206527    56143936
 446833125 1411759022  1571423643  1350546828  1415105505   738959530   198833719  1670187544
 926714397  754613590  1842376979  1021147620   928831641  1734214546  1657246255  1814368496
1745197669  944197730  1349236107  1958419260  1622824529   554097370   376928039   287712456
1082484461  987473030  1553921347   568850836   890905865   870160242  1449100575  1401529760
  99233085   19521438  1505181819  1206351084   707339457   844032266  1683594263  1371005944
 602929381  101490107  1752520947  2037443908  1108882809  2145071010    69442575   371289680
 843031093  803308750  2006106475  1575994012   387725105   982960954  2061460743  1653372456
1673015405  718515942  1307599587   117658868  1995311593  1957305298   512939775   264727296
 859029989   22295806   550901131   369034316   239766945  1037140330  1973932535  1779412184
1444114397 1395692054  1001378003   829042852   611497441  2094881794  1718367727  1768161456
1079501077  547097390   932011339  1249517436  1722492945   422710170  1989272295   167876488
 931705677 1578240326  1117954755   850427988  1136455465  1950871602   447658207   870973536
1988150917 1420094550  2009563451    18768812  1467287169   858483146   364681175   509425208
1625225917 1404769358  1173870771  1210139652  1713643321  1476188258  1754881999  1113364752
1193199221  841224654  1741916459   449085788  1746872561   728800250  1367797959  1784514792
 924665389  565097382  1797157539  1648851892   624886697  1984894098  1537944255   801727936
1703877989 1283254974  1212183835    60658444  1616940897   183804458  2123361719  1216188312
 467067101 1753041878   163934355  1424893796   316976153   517055682  1302534575   608921200
1023577813 1641154542    64832779  1662294204   942684625  1944747098   934710951  1517832776
1538124045  381791750   467351107   119555860   690755721  1381468402  1451240607  1732090656
1518810661  354884135   532716059  1272222316   130361409  1371875578  1310315415   961067256
 558776445 1176631734   764450931   589880004   980175097  1928577314  1015112591   255525840
1610939829 2122153550   723685611  1808302108  2006342321   154643642   957922439   623500200
1651679189  945346918  1297269347  1184031348  1631337833  1945718098  1680195199   840867968
1539995429  383175230   799608027  1008427468   911258145   674562794   106056055   899068504
1365375817  473463702  2077315155   841571876  1886582233   167456130  1076641135   352547120
```

```
     2075448     8948665    14876650    18768812    19521438    21815793    21957374    21958023
    26295806    27140417    38425458    55216974    56143936    60658444    62157203    64832779
    69442575    77349427    88230085    89667930   102291706   106056055   107970205   108248461
   111058364   117658868   119555860   120045942   123738191   130361409   154643642   163934355
   167456130   167876488   176116155   177192726   183804458   193848055   197755792   198833719
   206624361   214649377   229153275   239766945   245381547   252425123   254884126   255525840
   256747670   264727296   278664544   279097376   282857640   287712456   290003251   299012447
   299539206   301490102   307971765   313059018   316976153   330637817   341737414   352547120
   353495590   354206527   354347766   357031327   364681175   366216476   368424894   369034316
   370540484   371289680   376928039   381791750   383375230   387725105   391940832   398222275
   404438615   418948594   422710170   426972081   440927269   445642210   446833125   447658207
   449085788   453806245   462067101   467391107   473463702   493443029   493864492   499041517
   501436121   509425208   512939775   517055682   518370279   532216059   534929588   541895700
   546665169   547097390   547644349   550903131   553212125   554097370   557576683   558776445
   560355959   565097382   566455824   568850836   575358357   582503883   589880004   602928381
   608921200   611492441   621610588   623500200   624886697   653482419   653697896   655205502
   657787567   657901104   664248106   665854523   674562794   687191394   690352648   690755721
   698366302   700328108   707339457   708115136   714003599   718515942   723685611   728800250
   738959530   747111204   754613590   764450931   769521721   780360365   783222733   784335298
   785220977   788872316   799608027   801727936   803308750   811258145   817319048   829042852
   837021872   840867968   841571876   843031093   844032266   845346918   848224654   850427988
   858029989   858483146   866644152   870160242   870973536   874384709   879670838   890905865
   893772893   899068504   924665389   926714397   928831641   931705677   932011339   934710951
   940822958   942684625   944197230   957922439   961067256   964476392   972678484   973517524
   975274753   980175097   982960954   987473030   987692929   987848977  1001378003  1008427468
  1013403987  1015112591  1015949527  1021147620  1023577813  1037140330  1043703640  1056397164
  1065585481  1076641135  1079501077  1083230068  1086402365  1108882809  1113364752  1117954755
  1119036939  1126631734  1128891544  1136455465  1146650290  1173870771  1183199221  1184031348
  1201646101  1206351084  1210139652  1212183835  1216188312  1221017340  1249917436  1253041878
  1262550856  1265204842  1268161456  1272222316  1280824964  1281852958  1283254974  1286269370
  1297269347  1300420004  1301457277  1302534575  1305396962  1307599587  1310315415  1321005944
  1348445900  1349236107  1350546828  1354256261  1365375837  1367797959  1371875978  1373767777
  1377876226  1381468402  1395692054  1401525760  1403454307  1404769398  1409290974  1415105505
  1417577001  1420094558  1424893796  1426888219  1430079820  1431753022  1444114397  1449100575
  1450873357  1451240607  1452080823  1467287169  1468163223  1476188258  1481951386  1482338706
  1509181819  1517267018  1517832776  1518830661  1524405123  1526017647  1537944255  1538124045
  1552921347  1571423643  1575994012  1577827184  1578240326  1606887151  1610076627  1610939829
  1616940897  1622824529  1625225917  1631337833  1637124738  1641154547  1644729123  1648851892
  1651678189  1653228508  1653372456  1657246255  1660693979  1662294204  1664973045  1670187544
  1671817771  1673015405  1679187021  1680195199  1683594263  1687476432  1694371122  1695288069
  1696046626  1698247049  1700927815  1703877989  1706387470  1710215375  1713643321  1718367727
  1722346010  1722492945  1732090656  1734234946  1737027455  1739017063  1739154790  1741915459
  1745187669  1746872561  1752520947  1752897481  1754881999  1761569247  1775902886  1779417184
  1782573721  1784514792  1797157539  1808302108  1810589956  1814368496  1829804078  1842376979
  1858403161  1866568942  1884694117  1886582233  1889387665  1902777302  1903252597  1911175802
  1919937106  1927206924  1928577314  1939995429  1944747098  1945718098  1950871602  1957305298
  1958419260  1973932535  1974062180  1977811779  1984894098  1985883640  1988150917  1989272295
  1993189258  1995311593  2006106475  2006342321  2007658285  2009563451  2011229838  2022264873
  2037443908  2047241780  2050023511  2053201831  2061460743  2075997568  2077315155  2080940504
  2092112243  2094881794  2111637153  2115460031  2122153550  2123361719  2129019335  2145071010

000.35 SECONDS IN EXECUTION
```

```
=> EXECUTION FLOW SUMMARY

0000        1.--|       BEGIN
0001            |       INTEGER N, M;
0002            |       N := 400;  M := 9;   INTOVFL := NULL;
0005            |       BEGIN
0006            |       INTEGER ARRAY A(0 :: N + 1);
0007            |       INTEGER ARRAY STACK(0 :: 2*(ENTIER(LN((N + 1)/(M + 2)))) + 1);
0008            |       INTEGER P, L, R, I, J, V, T;
0009            |       A(0) := 0;
0010            |       FOR  I := 1  UNTIL  N  DO
0010      400.--|          BEGIN  A(I) := A(I - 1)*314159269 + 453806245;
0012               |       IF  A(I) < 0  THEN
0012         89.--|           A(I) := (A(I) + 2147483647) + 1;
0013               |       END;
0014            |       A(N + 1) := 2147483647;
0015            |       FOR  I := 1  UNTIL  N  DO
0015      400.--|          WRITEON(A(I));
0016            |       IOCONTROL(3);   P := 0;   L := 1;   R := N;
0020       69.--|       PARTITION:  I := L;   J := R + 1;   V := A(L);
0023            |       WHILE  I < J  DO
0023      585.--|          BEGIN  I := I + 1;
0025               |       WHILE  A(I) < V  DO
0025        906.--|           I := I + 1;
0026               |       J := J - 1;
0027               |       WHILE  A(J) > V  DO
0027        883.--|           J := J - 1;
0028               |       T := A(J);   A(J) := A(I);   A(I) := T;
0031               |       END;
0032            |       A(I) := A(J);   A(J) := A(L);   A(L) := T;
0035            |       IF  R - J > J - L  THEN
0035       27.--|          GOTO RRIG;
0036       42.--|       IF  J - L <= M  THEN
0036       13.--|          GOTO POP;
0037       29.--|       IF  R - J <= M  THEN
0037       18.--|          GOTO LEFT;
0038       11.--|       P := P + 2;   STACK(P) := L;   STACK(P + 1) := J - 1;
0041       23.--|       RIGHT:  L := J + 1;   GOTO PARTITION;
0043       27.--|       RRIG:
0043            |       IF  R - J <= M  THEN
0043        7.--|          GOTO POP;
0044       20.--|       IF  J - L <= M  THEN
0044       12.--|          GOTO RIGHT;
0045        8.--|       P := P + 2;   STACK(P) := J + 1;   STACK(P + 1) := R;
0048       26.--|       LEFT:  R := J - 1;   GOTO PARTITION;
0050       20.--|       POP:  L := STACK(P);   R := STACK(P + 1);   P := P - 2;
0053            |       IF  P >= 0  THEN
0053       19.--|          GOTO PARTITION;
0054        1.--|       INSERTION:
0054            |       FOR  I := 2  UNTIL  N  DO
0054      399.--|          BEGIN  V := A(I);   J := I - 1;
0057               |       WHILE  A(J) > V  DO
0057        421.--|           BEGIN  A(J + 1) := A(J);   J := J - 1;
0060                |          END;
0061               |       A(J + 1) := V;
0062               |       END;
0063            |       FOR  I := 1  UNTIL  N  DO
0063      400.--|          WRITEON(A(I));
0064            |       END;
0065            |       END
```

Our second example is an implementation of Program 2.4 in FORTRAN H. Because of the rather limited control constructs allowed by FORTRAN, this implementation is a direct adaptation of the assembly language implementation given in Appendix A. The following four pages are the listing of a run of this program, under the same conditions as the ALGOL W program just presented.

```
LEVEL 21.6 ( MAY 72 )                          OS/360  FORTRAN H                                        DATE  75.128/13.56.23

          COMPILER OPTIONS - NAME=  MAIN,OPT=00,LINECNT=58,SIZE=0000K,
                                    SOURCE,EBCDIC,NOLIST,NODECK,LOAD,NOMAP,NOEDIT,ID,NOXREF
      ISN 0002              IMPLICIT INTEGER (A-Z)
      ISN 0003              DIMENSION A(402),STACK(12)
      ISN 0004              M=400
      ISN 0005              M=9
      ISN 0006              A(1)=0
      ISN 0007              DO 1 I=1,N
      ISN 0008              A(I+1)=A(I)*314159269+453806245
      ISN 0009              IF (A(I+1)) 2,1,1
      ISN 0010        2     A(I+1)=(A(I+1)+2147483647)+1
      ISN 0011        1     CONTINUE
      ISN 0012              A(N+2)=2147483647
      ISN 0013              WRITE(6,1000) (A(I+1),I=1,N)
      ISN 0014     1000     FORMAT (8I16)
      ISN 0015              L=2
      ISN 0016              R=N+1
      ISN 0017              P=1
      ISN 0018       10     I=L+1
      ISN 0019              J=R+1
      ISN 0020              V=A(L)
      ISN 0021              GO TO 31
      ISN 0022       20     T=A(I)
      ISN 0023              A(I)=A(J)
      ISN 0024              A(J)=T
      ISN 0025       30     I=I+1
      ISN 0026       31     IF (A(I).LT.V) GO TO 30
      ISN 0028       40     J=J-1
      ISN 0029              IF (A(J).GT.V) GO TO 40
      ISN 0031              IF (I.LT.J) GO TO 20
      ISN 0033              T=A(L)
      ISN 0034              A(L)=A(J)
      ISN 0035              A(J)=T
      ISN 0036              IF ((R-J).GT.(J-L)) GO TO 60
      ISN 0038              IF ((J-L).LE.M) GO TO 80
      ISN 0040              IF ((R-J).LE.M) GO TO 70
      ISN 0042              P=P+2
      ISN 0043              STACK(P)=L
      ISN 0044              STACK(P+1)=J-1
      ISN 0045       50     L=J+1
      ISN 0046              GO TO 10
      ISN 0047       60     IF ((R-J).LE.M) GO TO 80
      ISN 0049              IF ((J-L).LE.M) GO TO 50
      ISN 0051              P=P+2
      ISN 0052              STACK(P)=J+1
      ISN 0053              STACK(P+1)=R
      ISN 0054       70     R=J-1
      ISN 0055              GO TO 10
      ISN 0056       80     L=STACK(P)
      ISN 0057              R=STACK(P+1)
      ISN 0058              P=P-2
      ISN 0059              IF (P) 100,10,10
      ISN 0060      100     DO 120 I=2,N
      ISN 0061              IF (A(I+1).GE.A(I)) GO TO 120
      ISN 0063              V=A(I+1)
      ISN 0064              J=I
```

```
    ISN 0065      110   A(J+1)=A(J)
    ISN 0066            J=J-1
    ISN 0067            IF (A(J).GT.V) GO TO 110
    ISN 0069            A(J+1)=V
    ISN 0070      120   CONTINUE
    ISN 0071            WRITE(6,1001)
    ISN 0072     1001   FORMAT(1H1)
    ISN 0073            WRITE(6,1000) (A(I+1),I=1,N)
    ISN 0074            RETURN
    ISN 0075            END

*OPTIONS IN EFFECT*      NAME=  MAIN,OPT=00,LINECNT=58,SIZE=0000K,

*OPTIONS IN EFFECT*      SOURCE,EBCDIC,NOLIST,NODECK,LOAD,NOMAP,NOEDIT,ID,NOXREF

*STATISTICS*     SOURCE STATEMENTS =      74 ,PROGRAM SIZE =     3116

*STATISTICS*  NO  DIAGNOSTICS GENERATED

****** END OF COMPILATION ******                               146K BYTES OF CORE NOT USED




F128-LEVEL LINKAGE EDITOR OPTIONS SPECIFIED NOMAP
         DEFAULT OPTION(S) USED -  SIZE=(190464,30720)
****MAIN      DOES NOT EXIST BUT HAS BEEN ADDED TO DATA SET
```

```
 453806245    21957374   404438619  1430079820  2111637153  1265204842   193848055  2080940504
 553212125   177192726  1610076627  1300470004  1858403161  1377876226  1606887151   837021872
1201546101  1829804078    77348427  1221017340   987848977  1481951386   518370279   817319048
1679197021   445642310   398222275   972678484  1752897481  1694371122  1761569247   278664544
1354256261   658166302   665854523   700328132   987692925   313059018  1015949527     2075448
 547644349   172045942   653482419  1810589956   769521721   687191394  1710215375   566455824
1664977045  2011229839  1671817771   621610586    21815793   102291706  2129019335   964476397
2107658285  1775902886   252425123   534929586  1417977001  1482138706  2115460031   708115136
1884694117   368424894  1426888219  1927206924  1373767777   664248106  1452080823  1128891544
 107970205  1902777302    62157203  1974062180  1782973721   784135298   657787567  1577827184
 493443029  1866568942  1119036939   111058364   546665169    89667930  2053201831  1262550856
1450877357   259539206  1524405123   541895700  1698247045   418948594   357031327   279097376
 874384709  1281852558   229153275  1056357164    27140417  1993189258  1468163223  1985883640
1301457277   979670838  2092112243   370540444   330637817  1696046626   714003599  1687476437
 307071765    55216974   557576683   366216476   426972081  1286269370    21958023   282857640
 499041517  1739154790  1403454307  1083230068   206624361  1919937106  1737027455  2075997568
 440927264   655205502  1660693979  1348445900   214649377    14876650   560355959  1043703640
 993772893   256747670  1013403987   747111204   501436121  1637124738  1526017647   657901104
 575358357   940822958   582503883   788872316  1889387665  1722346010  1739017063   690352648
 783227733   341737414  1977811779   973517524  1065585481  1146650290   299012447   391940832
1695289069  1409290974   176116155   493864492   975274753  1517267018  2050023511   866644152
1086402365   354347766   290003251  1280824964     8048665  1305396962   123738191   197755792
1903252597  1706387470   245381547  1653228508   785220977  1911175802  1700927815   653697896
 780360365   353495590  1644729123  2047241780  2022264873    38429458   354206527    56143936
 446833125  1431753022  1571423643  1350546828  1415105505   738959530   198833719  1670187544
 926714397   754613590  1842376979  1021147620   928831641  1734234946  1657246255  1814368496
1745187669   944197230  1349236107  1958419260  1622824525   554097370   376928039   287712456
 108248461   987473030  1552921347   568850836   890905865   870160242  1449100575  1401529760
  88230085    19521438  1509181819  1206351084   707339457   844037266  1683594263  1371005944
 602928381   301490102  1752520947  2037443908  1108882809  2145071010    69442575   371289680
 843031093   803308750  2006106475  1575994012   387725105   982960954  2061460743  1653372456
1673015405   718515942  1307599587   117658868  1995311593  1957305298   512939775   264727296
 858029989    26295806   550903131   369034316   239766945  1037140330  1973932535  1779412184
1444114397  1395692054  1001378003   829042852   611492441  2094881794  1718367727  1268161456
1079501077   547097390   932011339  1249917436  1722492945   427710170  1989272295   167076488
 931705677  1578240326  1117954755   850427988  1136459465  1950871602   447658207   870973536
1988150917  1420074558  2009563451    18768812  1467287169   858483146   364681175   509425208
1625225917  1404769398  1173870771  1210139652  1713643321  1476188258  1754881999  1113364752
1183199221   848224654  1741916459   449085788  1746872561    72880250  1367797959  1784514792
 924665389   565097382  1797157539  1648851492   624886697  1984894098  1537944255   801727936
1733877980  1283254974  1212183835    60658444  1616940897   183804458  2123361719  1216188317
 462067101  1253041878   163934355  1424853776   316976153   517055682  1302534575   608921200
1023577913  1641154542    64832779  1662294204   942684625  1944747098   934710951  1517832776
1538124045   381791750   467391107   119555860   690755721  1381468402  1451240607  1732090656
1518930661   254884126   532216059  1272222316   130361405  1371875978  1310315415   961067256
 558776445  1126631734   764450931   585880004   980175097  1928577314  1015112591   255525840
1610939329  2122153550   723685611  1808302108  2006342321   154643642   957922439   623500200
1451679190   845346918  1297269347  1184031348  1631337833  1945718098  1680195199   840867968
1939995429   383275230   799608027  1008427464   811258145   674567794   106056055   899068504
1365375837   473463702  2077315155   841571974  1886582213   167456130  1076641135   352547120
```

```
   2075448     8948665    14876650    18768812    19521438    21815793    21957374    21958023
  26295806    27140417    38429458    55216974    56143936    60658444    62157203    64832779
  69442575    77348427    88230085    89667930   102291706   106056055   107970205   108248461
 111058364   117658868   119555860   120045942   123738191   130361409   154643642   163934355
 167456130   167876488   176116155   177192726   183804458   193848055   197755792   198833719
 206624361   214649377   229153275   239766945   245381547   252425123   254884126   255525840
 256747670   264727296   278664544   279097376   282857640   287712456   290003251   299012447
 299539206   301490102   307971765   313059018   316976153   330637817   341737414   352547120
 353495590   354206527   354347766   357031327   364681175   366216476   368424894   369034316
 370540484   371289680   376928039   381791750   383375230   387725105   391940832   398222275
 404438619   418948594   422710170   426972081   440927269   445642310   446833125   447658207
 449085788   453806245   462067101   467391107   473463702   493442029   493864492   499041517
 501436121   509425208   512939775   517055682   518370279   532216059   534929588   541895700
 546665169   547097390   547644349   550903131   553212125   554097370   557576683   558776445
 560359959   565097382   566455824   568850836   575358357   582503883   589880004   602928381
 608921200   611492441   621610588   623500200   624886697   653482419   653697896   655205502
 657787567   657901104   664248106   665854523   674562794   687191394   690352648   690755721
 698366302   700328108   707339457   708115136   714003599   718515942   723685611   728800250
 738959530   747111204   754613590   764450931   769521721   780360365   783222733   784335298
 785220977   788872316   799608027   801727936   803308750   811258145   817319048   829042852
 837021872   840867968   841571876   843031093   844032266   845346918   848224654   850427988
 858029989   858483146   866644152   870160242   870973536   874384709   879670838   890905865
 893772893   899068504   924665389   926714397   928831641   931705677   932011339   934710951
 940822959   942684625   944197230   957922439   961067256   964476392   972678484   973517524
 975274753   980175097   982960954   987473030   987692929   987848977  1001378003  1008427468
1013403987  1015112591  1015949527  1021147620  1023577813  1037140330  1043703640  1056397164
1065585481  1076641135  1079501077  1083230068  1086402365  1108882809  1113364752  1117954755
1119036939  1126631734  1128891544  1136459465  1146650290  1173870771  1183199221  1184031348
1201646101  1206351084  1210139652  1212183835  1216188312  1221017340  1249917436  1253041878
1262550856  1265204842  1268161456  1272222316  1280824964  1281852958  1283254974  1286269370
1297269347  1300420004  1301457277  1302534575  1305396962  1307599587  1310315415  1321005944
1348445900  1349236107  1350546828  1354256261  1365375837  1367797959  1371875978  1373767777
1377876226  1381468402  1395692054  1401529760  1403454307  1404769398  1409290974  1415105505
1417977001  1420094558  1424893796  1426888219  1430079820  1431753022  1444114397  1449100575
1450873357  1451240607  1452080823  1467287169  1468163223  1476188258  1481951386  1482338706
1509181819  1517267018  1517832776  1518830661  1524405123  1526017647  1537944255  1538124045
1552921347  1571423643  1575994012  1577827184  1578240326  1606887151  1610076627  1610939829
1616940897  1622824529  1625225917  1631337833  1637124738  1641154542  1644729123  1648851892
1651678189  1653228508  1653372456  1657246255  1660693975  1662294204  1664973045  1670187544
1671817771  1673015405  1679187021  1680195199  1683594263  1687476432  1694371122  1695288069
1696046626  1698247049  1700927815  1703877989  1706387470  1710215375  1713643321  1718367727
1722346010  1722492945  1732090656  1734234946  1737027455  1739017063  1739154790  1741916459
1745187669  1746872561  1752520947  1752897481  1754881995  1761569247  1775902886  1779412184
1782973721  1784514792  1797157539  1808302108  1810589956  1814368496  1829804078  1842376979
1858403161  1866568942  1884694117  1886582233  1889387665  1902777302  1903252597  1911175802
1919937106  1927206924  1928577314  1939995429  1944747098  1945718098  1950871602  1957305298
1958419260  1973932535  1974062180  1977811779  1984894098  1985883640  1988150917  1989272295
1993189258  1995311593  2006106475  2006342321  2007658285  2009563451  2011229838  2022264873
2037443908  2047241780  2050023511  2053201831  2061460743  2075997568  2077315155  2080940504
2092112243  2094881794  2111637153  2115460031  2122153550  2123361719  2129019335  2145071010
```

In order to study the efficiency of these programs we have to look at the compiled code to get some idea of what the coefficients of the various quantities are. This is of course highly dependent on the kind of compiler used and the design goals of the compiler writer. To focus our attention on the issues involved let us examine the compiled code for just one statement in the inner loop of the programs above: the statement which implements the right pointer scan,

<u>loop</u>: j := j-1; <u>while</u> A[j] > v <u>repeat</u>;

in Program 2.4. The table below shows the code produced for the corresponding statements in the programs above by the ALGOL W and FORTRAN H compilers in operation on the IBM S/360-67 at Stanford University on March 2, 1975. The third column is the result of the "optimizing" phase available for the FORTRAN H compiler, and the fourth is the code that a good assembly language programmer would produce (corresponding to Program 2.4A).

```
ALGOL W               FORTRAN H              FORTRAN H             hand-coded
                                             (optimized)

     L   2,J          SCAN L   0,J           SCAN SR 7,11          SCAN SR 7,11
     S   2,=F'1'           S   0,=F'1'            A  5,=F'-4'           C  9,A(7)
     ST  2,J               ST  0,J                LR 10,5               BH SCAN
SCAN L   2,J               LR  6,0                L  9,A(5)
     SLL 2,2               SLL 6,2                C  9,V           (Reg. 11
     AL  2,A               L   0,A(6)             BH SCAN           contains 4)
     L   2,0(2)            C   0,V
     C   2,V               L   5,=A'SCAN'    (Reg. 11
     BNH OUT               BHR 5              contains 1)
     L   2,J
     S   2,=F'1'
     ST  2,J
     B   SCAN
OUT
```

(This code has been "dressed up" a little with labels to make it more readable.)

We can find simple reasons for nearly all of the discrepancies in this table. First, the ALGOL W code is slightly longer than the FORTRAN H code only because we had to use the <u>while</u> ... <u>do</u> statement. If ALGOL W allowed **us** to write <u>repeat</u> j := j-1 <u>until</u> A[j] $\leq$ v; then we might imagine that the compiler would produce the same code, except with the last four instructions deleted, the label SCAN moved to the first instruction, and " BNH OUT " changed to " BH SCAN ". The first two columns would then be comparable (and very inefficient). Both use three memory reference instructions to decrement the pointer, then load A(J) and compare it with V in memory. The ALGOL W program has an unnecessary " L 2,J " instruction at SCAN , and it takes two instructions to load A(J) when one (" L 2,A(2) ") would suffice; but the FORTRAN H program has an unnecessary " LR 6,0 " instruction (since 0 can't be used as an index register) and it takes two instructions for the branch at the end when one (BH SCAN) would suffice. Since the S/360 has byte addressing, both programs need to compute the displacement to A(J) by multiplying J by four (" SLL 2 "). Any assembly language programmer will avoid this by simply representing J by 4*J throughout the program, but the compilers have difficulty doing this. In fact, nearly 10% of the instructions in the ALGOL W and FORTRAN H compilations of the complete Quicksort programs given above are " SLL 2 " instructions. Even in the optimized code, this leads to inefficiency: registers containing <u>both</u> J and 4*J are maintained.

A more interesting inefficiency in the FORTRAN H optimized program is that it did not decide to keep V in a register, but rather to load A(J) into a register and then compare it with V in memory. To see a reason for this, let us examine the full inner loop:

```
        FORTRAN H
        (optimized)                          hand-coded

                B    INTO                                B    INTO
        LOOP    ST   9,A(4)                  LOOP        L    9,A(7)
                ST   8,A(10)                             L    8,A(6)
        LSCAN   AR   6,11                                ST   9,A(6)
                A    4,=F'4'                             ST   8,A(7)
        INTO    L    8,A(4)                  LSCAN       AR   6,11
                C    8,V                     INTO        C    5,A(6)
                BL   LSCAN                               BH   LSCAN
        RSCAN   SR   7,11                    RSCAN       SR   7,11
                A    5,=F'-4'                            C    5,A(7)
                LR   10,5                                BL   RSCAN
                L    9,A(5)                              CR   6,7
                C    9,V                                 BH   LOOP
                BH   RSCAN
                CR   6,7                     (Reg. 11 contains 4)
                BH   LOOP

        (Reg. 11 contains 1)
```

We now see that the reason that A(I) and A(J) are loaded into registers is to make the exchange efficient: it requires only two store instructions. This is of course less efficient than the hand-coded version, since the two load instructions saved are effectively pushed into the inner loop.

However, we should not be too harsh on the FORTRAN compiler, because the optimized code is actually quite good. If we look at the code produced by another widely used compiler, we can get a much better idea of the "pitfalls of compilation". It is easy to produce a PL/I program from the ALGOL W program just given, almost by direct translation. The following code was produced by the PL/I Optimizing Compiler (version 1 R 2.1 PTF56) on the IBM S/360-67 at Stanford University on April 10, 1975 for the statement

```
J = J-1; DO WHILE (A(J) > V); J = J-1; END;
```

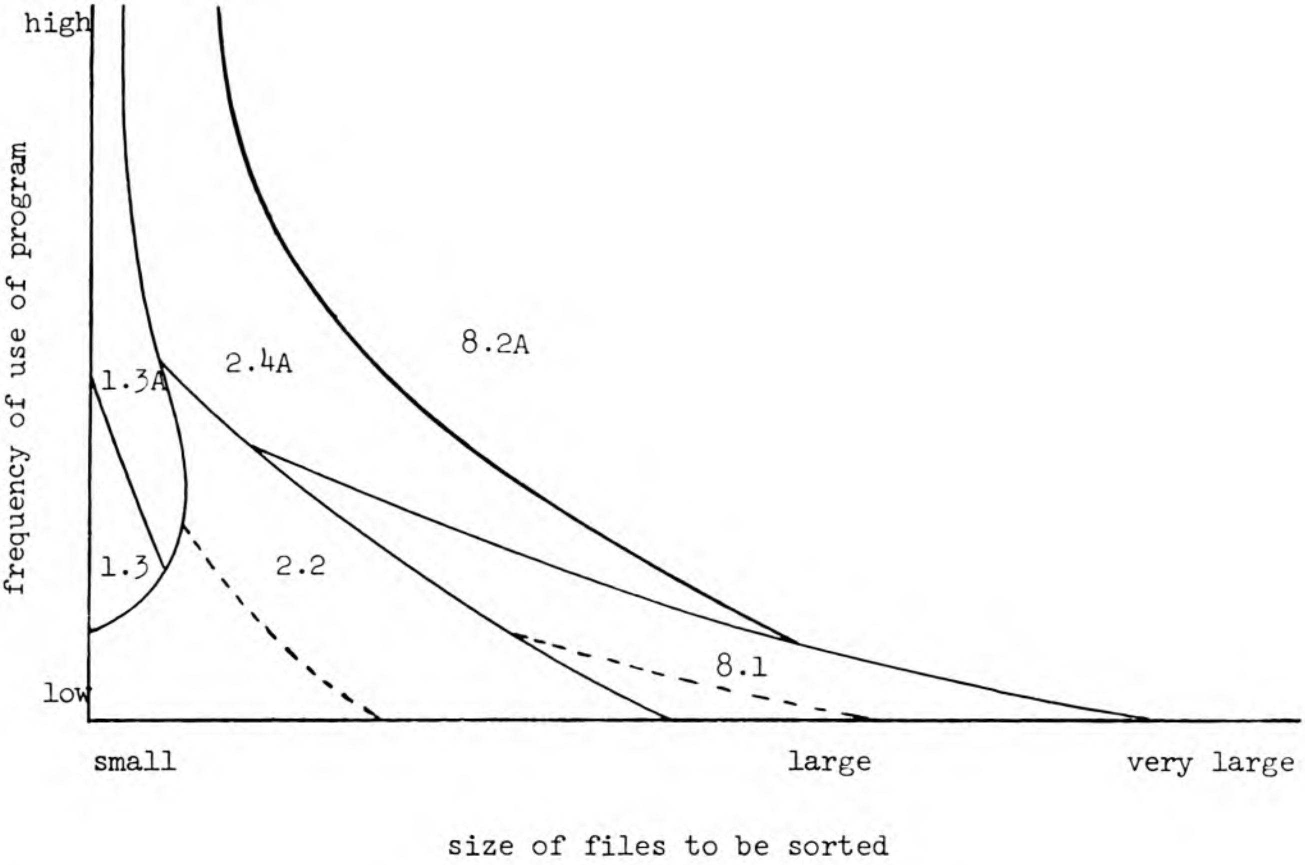
which corresponds to those we studied above.

```
          unoptimized                optimized
                                OPT=TIME; (REORDER)

          L    15,J                 L      15,J
          S    15,=F'4'             S      15,=F'4'
          ST   15,J                 ST     15,J
     SCAN L    9,J                  LH     8,=H'4'
          SLA  9,2                  MR     14,8
          L    6,A(9)               SLDA   14,32
          C    6,V                  ST     14,FOURJ
          BNH  OUT                  LR     4,14
          L    15,J            SCAN L      7,FOURJ
          S    15,"F'4'             L      6,A(7)
          ST   15,J                 C      6,V
          B    SCAN                 BNH    OUT
     OUT                            AH     7,=H'-4'
                                    ST     7,FOURJ
                                    L      15,J
                                    S      15,=F'4'
                                    ST     15,J
                                    B      SCAN
                               OUT
```

The compiler recognized that the " SLA 9,2 " is undesirable and could be removed by keeping a variable with the value 4*J . But this does not lead to a savings because the obvious optimizations of keeping J , FOURJ , and F'4' in registers are missed entirely. The optimized code is far less efficient than the unoptimized.

Although it is fascinating to study the comparative efficiencies of compiler produced code in this way, we have begun to stray some from our topic. We are interested in the implementation of Quicksort, not of optimizing compilers. It is no surprise to a student of compiler construction that a good optimizing compiler will produce good code (though Quicksort will give it a workout), but which is not quite as efficient as a hand-coded version. Fortunately, the Quicksort algorithm is not very difficult to program in assembly language and if the program is to be run often, or on a very large file, it is worthwhile to do so. Of course, the median-of-three method described in Chapter 8 should be used to reduce the average values of the various quantiti es involved in the running time of the program, and the "loop unwrapping" technique described at the end of Appendix A should be used to reduce the coefficient of the overhead associated with each comparison. On the other hand, if we are only going to use the program a few times, and space is not a problem, Program 2.2 may well be quite acceptable. Some of the tradeoffs involved in choosing a proper implementation are summarized in the following diagram:



(The numbers in this diagram represent the various programs that we have studied. The dotted lines in the areas for Programs 2.2 and 8.1 are meant to indicate when it might be worthwhile to modify those programs to ignore small subfiles and insertion sort after partitioning.) Near the bottom of this diagram, the methods are ordered by their relative ease of implementation; near the top, they are ordered by the average running times that we have calculated. For example, the crossover point between Program 2.4A and Program 8.2A is approximately where $11.67(N+1)\ln N - 1.74N - 18.74 = 9.57\,N\ln N + 7.14N$, or at about $N = 67$. Program 2.4A will always be faster for files smaller than this, and Program 1.3A will be even faster for very small files.

It is entertaining to construct such diagrams, and we could include more based upon other parameters. However, the final choice of implementation obviously depends on the circumstances under which the program will be used, and the reader that has persisted this far should have little difficulty making the proper choices for his application. One final caution: all of our deliberations have been based on the assumption that the programs run on a conventional computer, and sort records on well-distributed keys which all fit into memory. There are a variety of situations where Quicksort might be not at all appropriate: for example if people's lives depend on the speed of the sorting program, then the $O(N^2)$ worst case may be unacceptable. If a large amount of auxiliary memory is available, then a distribution type sort (address calculation) may be faster. In addition, exotic hardware features may make the tradeoffs involved very different. For example Quicksort may not be very good for computers which allow parallelism, and there are better methods for processors with highly optimized arithmetic units. Loop unwrapping may be disastrous on computers which have instruction stacks, and it might be best to insertion sort small subfiles when they are encountered in a paging environment. However, in a large variety of situations, Quicksort is the method of choice, and the programs that we have studied are extremely useful in practical applications.