QUICKSORT WITH EQUAL KEYS*

ROBERT SEDGEWICK†

Abstract. This paper considers the problem of implementing and analyzing a Quicksort program when equal keys are likely to be present in the file to be sorted. Upper and lower bounds are derived on the average number of comparisons needed by any Quicksort program when equal keys are present. It is shown that, of the three strategies which have been suggested for dealing with equal keys, the method of always stopping the scanning pointers on keys equal to the partitioning element performs best.

Key words. analysis of algorithms, equal keys, Quicksort, sorting

Introduction. The Quicksort algorithm, which was introduced by C. A. R. Hoare in 1960 [6], [7], has gained wide acceptance as the most efficient general-purpose sorting method suitable for use on computers. The algorithm has a rich history: many modifications have been suggested to improve its performance, and exact formulas have been derived describing the time and space requirements of the most important variants [7], [9], [14].

Although most files to be sorted contain at least some equal keys and sorting programs must always deal with them properly, it is generally considered reasonable to assume in the analysis that the keys are distinct. This assumption is fundamental to the analysis of nearly all sorting programs, and it is very often realistic. In any situation where the number of possible key values far exceeds the number of keys to be sorted, the probability that equal keys are present will be very small. However, if the number of possible key values is not large, or if there is some other information about the file which indicates that equal keys are likely to be present, then the performance of many sorting programs, including Quicksort, has not been carefully studied.

The purpose of this paper, then, is to investigate the performance of Quicksort when equal keys are present. The following section describes the algorithm and its analysis for distinct keys. Next, lower and upper bounds are derived for the average number of comparisons taken when equal keys are present. Following that, we shall consider, from a practical standpoint, the problem of implementing a version of Quicksort to handle equal keys. Finally we shall compare the various methods and discover which is the most useful in practical sorting applications.

1. Distinct keys. Suppose that an array of keys $A[1], \cdots, A[N]$ is to be rearranged to make

$$A[1] < A[2] < \cdots < A[N],$$

where the order relation $<$ is any transitive relation whatever defined on all the keys.

Quicksort is a "divide and conquer" approach to this problem. For some key with value v, the file is rearranged so that $A[j] = v$ for some j, $1 \leq j \leq N$, all of the keys to the left of $A[j]$ are $<v$ and all of the keys to the right of $A[j]$ are $>v$. This process is called *partitioning*, and it turns out that it can be performed efficiently. After partitioning, the key $A[j]$ is in its final position in the sorted file and need not be considered further. If the same procedure is applied recursively to the subfiles $A[1], \cdots, A[j-1]$ and $A[j+1], \cdots, A[N]$, then the whole file becomes sorted. The following program is an implementation of the method, and the partitioning process is spelled out explicitly.

PROGRAM 1.

procedure quicksort (**integer value** l, r);

    comment The array A is declared to be $A[1:N+1]$; with $A[N+1] = \infty$;

    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**;

        until $j < i$:

            $A[i] :=: A[j]$;

        repeat;

        $A[l] :=: A[j]$;

        quicksort($l, j-1$);

        quicksort(i, r);

    endif;

(This program uses an exchange operator $:=:$, and the control constructs **loop** $\cdots$ **repeat** and **if** $\cdots$ **then** $\cdots$ **endif**, which are like those described by D. E. Knuth in [10].) The leftmost element is chosen as the partitioning element, and then the rest of the array is partitioned on that value. This is done by scanning from the left to find an element $>v$, scanning from the right to find an element $<v$, exchanging them, and continuing the process until the scanning pointers cross. The loop always terminates with $j+1 = i$, and it is known at that point that $A[l+1], \cdots, A[j]$ are $<v$ and $A[j+1], \cdots, A[r]$ are $>v$, so that the exchange $A[l] :=: A[j]$ completes the job of partitioning $A[l], \cdots, A[r]$. The procedure call "quicksort$(1, N)$" will therefore sort $A[1], \cdots, A[N]$. Figure 1 shows the operation of the program on the first 9 distinct digits of π.

```
3   1   4   5   9   2   6   8   7
2   1  ③   5   9   4   6   8   7
1  ②
                4  ⑤   9   6   8   7
                        7   6   8  ⑨
                        6  ⑦   8
1   2   3   4   5   6   7   8   9
```
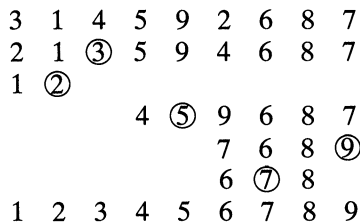
FIG. 1

A number of different partitioning methods have been suggested for the implementation of Quicksort, and the particular method described above is

motivated fully in [14]. There are, however, some facets of the implementation which should be noted here.

The loops which implement the pointer scans are the "inner loops" of the program—most of the execution time is spent there. (This fact comes from the analysis, which is discussed more fully below.) Some efficiency is achieved in the inner loops by introducing two redundant comparisons to avoid the necessity for checking if the pointers have crossed each time a pointer is changed. The last comparison in each of the loops is redundant: the last $i := i+1$ makes $i = j$ and it is known that $A[j] > v$ at that point (provided that $A[N+1]$ is greater than all the other keys—this is the meaning of the notation $A[N+1] = \infty$); the last $j := j-1$ makes $j = i-1$ and it is known that $A[i-1] \leqq v$ at that point. Program 1 uses $N+1$ comparisons on the first partitioning stage when only $N-1$ are absolutely necessary, but its inner loop is much more efficient as a result.

Although the above program gives a very efficient implementation of partitioning, there are a number of ways that the program as a whole can be made more efficient. It turns out that efficiency can be gained by choosing the partitioning element based on a small sample from the file; by removing the recursion and always sorting the smaller of the two subfiles first; and especially by handling small subfiles differently. All of these improvements are documented in [9] and [14], and they apply uniformly to all of the programs that we will consider.

We can derive exact formulas for the total average running time of Quicksort by solving recurrence relations which describe the average number of times the various statements in the program are executed. In this paper, we shall be concerned chiefly with the average number of comparisons: the average number of times the tests "$A[i] < v$" and "$A[j] > v$" are performed during the execution of Program 1 on a randomly ordered input file. If we denote this quantity by C_N, we find that it is described by the recurrence

$$C_N = N+1+\frac{1}{N} \sum_{1 \leqq k \leqq N} (C_{k-1}+C_{N-k}), \qquad N \geqq 2$$

$$= N+1+\frac{2}{N} \sum_{1 \leqq k \leqq N} C_{k-1}$$

with $C_0 = C_1 = 0$. The $N+1$ term represents the number of comparisons used on the first partitioning stage, and the other term represents the average number of comparisons used for the subfiles. By writing this recurrence, we have made the important assumption that the partitioning process preserves randomness in the subfiles: if the original file is a random permutation of its elements, then the left and right subfiles will also have this property. It is easy to prove that the partitioning method in Program 1 preserves randomness, but there are partitioning methods which do not (see [10], [14]).

To solve the recurrence, we first multiply by N and then eliminate the summation by differencing (subtracting the same equation for $N-1$). After rearranging terms, we get

$$NC_N = (N+1)C_{N-1}+2N, \qquad N \geqq 3,$$

which, after we divide by $N(N+1)$, telescopes to the solution

$$\frac{C_N}{N+1} = \frac{C_2}{3} + \sum_{3 \leq k \leq N} \frac{2}{k+1}$$

or

$$C_N = 2(N+1)(H_{N+1} - \tfrac{4}{3}), \qquad N \geqq 2.$$

This shows that Quicksort achieves the theoretical minimum of $O(N \log N)$ comparisons on the average. In a similar fashion, the average number of times each of the other statements in Program 1 is executed can be calculated exactly. If the time taken to execute each statement is also known, then we can get an exact formula for the total expected running time of Program 1 (see [9] and [14]).

In studying the performance of Quicksort with equal keys, we will deal chiefly with the average number of comparisons, not with the total running time. Analysis shows that the comparisons do dominate (though there is about one exchange for every three comparisons) and the calculations become tedious when dealing with the total running time. We will be comparing the relative performance of a number of very similar variants of Quicksort, and we can be fairly certain that conclusions that we draw based on the number of comparisons will carry through to the total running time. It is nearly always the case that if one version of Quicksort uses a lower number of comparisons than another, then the frequencies of execution of all of the other instructions are also lower.

It is intuitive that Quicksort performs best when the partition happens to be near the middle of the file at each stage, and worst when the partition falls near the ends. (The exact analysis of the best and worst case for practical versions of the program is interesting and complex, but not particularly relevant to the study of Quicksort with equal keys.) For Program 1, it turns out that the worst case for the number of comparisons (and for the whole algorithm) occurs when the file is already in order, and partitioning does nothing but take one key from the left end of the file at each stage. The total number of comparisons in this case is $\sum_{2 \leq k \leq N} (k+1) = \frac{1}{2}(N+4)(N-1)$. This $O(N^2)$ worst case is often viewed as Quicksort's weakness, but there are many ways to avoid it in practical situations. It will be a concern when we begin to consider equal keys. On the other hand, the best case occurs when the file is split exactly in half at each stage, and the total number of comparisons taken turns out to be less than $N \lg N$ ($\lg \equiv \log_2$). A complete discussion and derivations of exact upper and lower bounds for Quicksort may be found in [14].

In summary, the operation of Quicksort on files of distinct keys is very completely understood. Unfortunately, few of the results carry over to the case when equal keys are present. Before examining the question of actually implementing a program to handle equal keys properly, let us look more carefully into the analysis, so that we may get some idea of how well we might expect to do.

2. Basic assumptions. The first problem that we face in trying to analyze any sorting method with equal keys is the formulation of an appropriate model describing the input file. Suppose that the N keys to be sorted have n distinct values. Since we only use the relative values of the keys in sorting, we may as well

assume that the values are $1, 2, \cdots, n$. If we also know that there are x_1 ones, x_2 twos, etc., then we might consider each of the $N!$ permutations of the multiset $\{x_1 \cdot 1, x_2 \cdot 2, \cdots, x_n \cdot n\}$ (where $x_1 + \cdots + x_n = N$) to be equally likely as input files. If this additional information is not available, a second possibility would be to assume that each of the n^N ways of making an input file of length N from n distinct values is equally likely. (Notice that many of the possible input files have less than n distinct values in this model.) While one or the other of these models might be appropriate for some particular sorting applications, neither is entirely satisfactory as a general model. We shall work to some degree with both. When we speak of sorting a *random permutation from a multiset*, we will be referring to the first model; when we refer to a *random n-ary file*, we will be working with the second.

Now, if we wish to use Quicksort to sort a file containing equal keys, we must decide how to treat keys equal to the partitioning element during the partitioning process. Ideally, we would like to get all of them into position in the file, with all the keys with a smaller value to their left, and all the keys with a larger value to their right. Unfortunately, no efficient method for doing so has yet been devised, so we shall have some keys equal to the partitioning element in the left subfile and some in the right. We shall use the term *Quicksort program* to describe any program which sorts by recursively subdividing files of more than one element into three subfiles: a (nonempty) middle subfile whose elements are all equal to some value j; a left subfile with no elements $>j$; and a right subfile with no elements $<j$. The only further restrictions are that the value s must be chosen by examining one element from the file, and that if the input file is randomly ordered, so must be the subfiles. With these assumptions, we can write down a recurrence for the average number of comparisons to sort a random permutation from the multiset $\{x_1 \cdot 1, \cdots, x_n \cdot n\}$ (with $x_1 + \cdots + x_n = N$) for any Quicksort program:

$$C(x_1, \cdots, x_n) = N + 1 + \frac{1}{N!} \sum_{\substack{\text{all permutations} \\ \text{of} \{x_1 \cdot 1, \cdots, x_n \cdot n\}}} (C(x_1, \cdots, x_{j-1}, \alpha)$$

$$+ C(\beta, x_{j+1}, \cdots, x_n)).$$

(The notation $C(\beta, x_{j+1}, \cdots, x_n)$ is defined to mean $C(\beta)$ when $j = n$.) This formula assumes that $N + 1$ comparisons are used in the first partitioning stage. The keys equal to the partitioning element are distributed among the subfiles in some way, as described by the parameters α and β, which are functions of the partitioning method and the particular permutation being sorted. (By assuming that at least one element is put in position, we are assuming that $\alpha + \beta < x_j$.) We will use various initial conditions in the derivations below to complete this recurrence.

3. Lower bounds. From this formula we can begin to derive a lower bound on the number of comparisons, for, as we have already noted, the best that we can do with any partitioning method is to get all of the keys equal to the partitioning element into position at each partitioning stage. If the partitioning element is chosen randomly, then each of the $x_j(N-1)!$ permutations for which j is the partitioning element will be divided into a left subfile which is a random permutation of $\{x_1 \cdot 1, \cdots, x_{j-1} \cdot (j-1)\}$ and a right subfile which is a random permutation

of $\{x_{j+1} \cdot (j+1), \cdots, x_n \cdot n\}$. This means that a lower bound on the average number of comparisons used is certainly described by the recurrence

$$C(x_1, \cdots, x_n) = N - 1 + \frac{1}{N} \sum_{1 \leq j \leq n} x_j(C(x_1, \cdots, x_{j-1}) + C(x_{j+1}, \cdots, x_n)),$$

for N and $n \geq 1$, with $C(0) = C(1) = 0$. (As remarked above, only $N-1$ comparisons are absolutely necessary for the first stage.) To solve this recurrence, we will try to eliminate the summation by differencing, as we did above. If we multiply both sides by N, and then subtract the same equation for $\{x_2 \cdot 1, \cdots, x_n \cdot (n-1)\}$, we get

$$\left(\sum_{1 \leq j \leq n} x_j\right) C(x_1, \cdots, x_n) - \left(\sum_{2 \leq j \leq n} x_j\right) C(x_2, \cdots, x_n)$$

$$= x_1^2 - x_1 + 2x_1 \sum_{2 \leq j \leq n} x_j + x_1 C(x_2, \cdots, x_n)$$

$$+ \sum_{2 \leq j \leq n} x_j(C(x_1, \cdots, x_{j-1}) - C(x_2, \cdots, x_{j-1})) \quad \text{for } n \geq 1.$$

After rearranging terms and defining $G(x_1, \cdots, x_n) = C(x_1, \cdots, x_n) - C(x_2, \cdots, x_n)$, this equation becomes

$$\left(\sum_{1 \leq j \leq n} x_j\right) G(x_1, \cdots, x_n) = x_1^2 - x_1 + 2x_1 \sum_{2 \leq j \leq n} x_j$$

$$+ \sum_{2 \leq j \leq n} x_j G(x_1, \cdots, x_{j-1}) \quad \text{for } n \geq 1.$$

Now we difference again, except this time we subtract the same equation for $\{x_1 \cdot 1, x_2 \cdot 2, \cdots, x_{n-1} \cdot (n-1)\}$ to yield

$$\left(\sum_{1 \leq j \leq n} x_j\right) G(x_1, \cdots, x_n) - \left(\sum_{1 \leq j \leq n-1} x_j\right) G(x_1, \cdots, x_{n-1})$$

$$= 2x_1 x_n + x_n G(x_1, \cdots, x_{n-1}) \quad \text{for } n \geq 2,$$

or

$$G(x_1, \cdots, x_n) = G(x_1, \cdots, x_{n-1}) + \frac{2x_1 x_n}{x_1 + \cdots + x_n}.$$

This equation telescopes to

$$G(x_1, \cdots, x_n) = G(x_1) + \sum_{2 \leq j \leq n} \frac{2x_1 x_j}{x_1 + \cdots + x_j}.$$

(This formula assumes that $x_1 x_j/(x_1 + \cdots + x_j) = 0$ if $x_1 = x_j = 0$, even if $x_2, \cdots, x_{j-1}$ are also 0. We shall adopt this convention throughout this paper.)

After substituting for G, we get another telescoping recurrence,

$$C(x_1, \cdots, x_n) = C(x_2, \cdots, x_n) + C(x_1) + 2 \sum_{2 \leq j \leq n} \frac{x_1 x_j}{x_1 + \cdots + x_j},$$

which leads to the result

$$C(x_1, \cdots, x_n) = \sum_{1 \leq j \leq n} C(x_j) + 2 \sum_{1 \leq k < j \leq n} \frac{x_k x_j}{x_k + \cdots + x_j}.$$

This derivation was suggested by the analysis given by Burge [3] for a similar problem which we will discuss below. The formula is surprisingly simple, and it can tell us exactly how well we can expect to do in a variety of situations. For example, if $x_j = x$ for $1 \leq j \leq n$, then we have

$$C(x, \cdots, x) = N - n + 2 \sum_{1 \leq k < j \leq n} \frac{x}{j - k + 1}$$

$$= N - n + 2 \frac{N}{n} \sum_{1 < j \leq n} \sum_{1 < k \leq j} \frac{1}{k}$$

$$= N - n + 2 \frac{N}{n} \sum_{1 < k \leq n} \frac{n - k + 1}{k}$$

$$= 2(1 + 1/n) N H_n - 3N - n.$$

If we take $x = 1$ (and therefore $n = N$), then we have analyzed Program 1 with distinct keys, and this result differs from the answer in the previous section only because we used the lower bound of $N - 1$ comparisons for the first partitioning stage.

We can proceed further, and use the general result for a random permutation of a multiset to derive a lower bound for a random n-ary file. If C_{Nn} is defined to be the average number of comparisons taken by a Quicksort program on random n-ary files of length N, then we have

$$C_{Nn} = \frac{1}{n^N} \sum_{x_1 + \cdots + x_n = N} \binom{N}{x_1, \cdots, x_n} C(x_1, \cdots, x_n).$$

This is true because the probability that a given input is a permutation of a particular multiset $\{x_1 \cdot 1, \cdots, x_n \cdot n\}$ is

$$\frac{1}{n^N} \binom{N}{x_1, \cdots, x_n}.$$

Therefore, our lower bound is given by

$$\frac{1}{n^N} \sum_{x_1 + \cdots + x_n = N} \binom{N}{x_1, \cdots, x_n} \left(\sum_{1 \leq k \leq n} C(x_k) + 2 \sum_{1 \leq k < j \leq n} \frac{x_k x_j}{x_k + \cdots + x_j} \right).$$

The first term is easy to evaluate, since $C(x_k) = x_k - 1$ for $x_k > 0$ and $C(0) = 0$. We

have

$$\frac{1}{n^N} \sum_{x_1+\cdots+x_n=N} \binom{N}{x_1,\cdots,x_n} \sum_{1\leq k\leq n} C(x_k)$$

$$=\frac{1}{n^N} \sum_{x_1+\cdots+x_n=N} \binom{N}{x_1,\cdots,x_n} \sum_{1\leq k\leq n} (x_k-1+\delta_{x_k0})$$

$$=N-n+\frac{1}{n^N} \sum_{1\leq k\leq n} \sum_{\substack{x_1+\cdots+x_n=N\\x_k=0}} \binom{N}{x_1,\cdots,x_n}$$

$$=N-n+(n-1)^N/n^{N-1}$$

$$=N-n+n(1-1/n)^N.$$

The second term is more difficult, but it can also be simplified through the use of the multinomial theorem. After interchanging the order of summation, we have

$$\frac{2}{n^N} \sum_{1\leq k<j\leq n} \sum_{x_1+\cdots+x_n=N} \binom{N}{x_1,\cdots,x_n} \frac{x_k x_j}{x_k+\cdots+x_j}.$$

The first step is to split the sum and the multinomial coefficient in two parts:

$$\frac{2}{n^N} \sum_{1\leq k<j\leq n} \sum_{x_1+\cdots+x_{k-1}+i+x_{j+1}+\cdots+x_n=N} \binom{N}{x_1,\cdots,x_{k-1},i,x_{j+1},\cdots,x_n}$$

$$\sum_{\substack{x_k+\cdots+x_j=i\\x_k,x_j\geq 1}} \binom{i}{x_k,\cdots,x_j} \frac{x_k x_j}{i}.$$

(Here we have also taken note of the fact that all of the terms with x_k or $x_j=0$ vanish.) Now,

$$\binom{i}{x_k,\cdots,x_j} \frac{x_k x_j}{i}=(i-1)\binom{i-2}{x_k-1,x_{k+1},\cdots,x_{j-1},x_j-1},$$

so we can apply the multinomial theorem to the innermost sum, which leaves us with

$$\frac{2}{n^N} \sum_{1\leq k<j\leq n} \sum_{\substack{x_1+\cdots+x_{k-1}+i+x_{j+1}+\cdots+x_n=N\\i\geq 2}} (i-1)\binom{N}{x_1,\cdots,x_{k-1},i,x_{j+1},\cdots,x_n}$$

$$\cdot(j-k+1)^{i-2}.$$

The inner sum now reduces to three terms, one for the case $i=0$ and two more resulting from splitting the first factor, all of which can be evaluated with the

multinomial theorem. We have

$$\sum_{\substack{x_1+\cdots+x_{k-1}+i+x_{j+1}+\cdots+x_n=N \\ i \geqq 1}} i\binom{N}{x_1, \cdots, x_{k-1}, i, x_{j+1}, \cdots, x_n}(j-k+1)^{i-2}$$

$$= N\sum\binom{N-1}{x_1, \cdots, i-1, x_{j+1}, \cdots, x_n}(j-k+1)^{i-2}$$

$$= \frac{N}{j-k+1}n^{N-1}$$

and

$$\sum_{x_1+\cdots+x_{k-1}+i+x_{j+1}+\cdots+x_n=N}\binom{N}{x_1, \cdots, x_{k-1}, i, x_{j+1}, \cdots, x_n}(j-k+1)^{i-2}$$

$$= \frac{1}{(j-k+1)^2}n^N$$

and finally

$$\frac{1}{(j-k+1)^2}\sum_{x_1+\cdots+x_{k-1}+x_{j+1}+\cdots+x_n=N}\binom{N}{x_1, \cdots, x_{k-1}, x_{j+1}, \cdots, x_n}$$

$$= \frac{1}{(j-k+1)^2}(n-j+k-1)^N.$$

Substituting all of these into our expression for the lower bound, we have simplified it to

$$N-n+n\left(1-\frac{1}{n}\right)^N+2\sum_{1\leqq k<j\leqq n}\left(\frac{N}{n}\frac{1}{j-k+1}-\frac{1}{(j-k+1)^2}\right.$$

$$\left.+\frac{1}{(j-k+1)^2}\left(1-\frac{j-k+1}{n}\right)^N\right).$$

As we saw when we evaluated $C(x, \cdots, x)$, we know that

$$\sum_{1\leqq k<j\leqq n} f(j-k+1) = \sum_{1<j\leqq n}\sum_{1<k\leqq j} f(k) = \sum_{1<k\leqq n} (n-k+1)f(k),$$

so we now have

$$N-n+n\left(1-\frac{1}{n}\right)^N+2\sum_{1<k\leqq n}(n-k+1)\left(\frac{N}{n}\frac{1}{k}-\frac{1}{k^2}+\frac{1}{k^2}\left(1-\frac{k}{n}\right)^N\right).$$

This sum appears difficult to evaluate explicitly, mainly because of the last term. However, we may use this expression to prove:

THEOREM 1. *Any Quicksort program must require, on the average, at least*

$$N-n+2\sum_{1\leqq k<j\leqq n}\frac{x_k x_j}{x_k+\cdots+x_j}$$

comparisons to sort a random permutation of the multiset $\{x_1 \cdot 1, \cdots, x_n \cdot n\}$ *(where*

$x_1 + \cdots + x_n = N$) and at least

$$2N(1 + 1/n)H_n - 3(N + n)$$

or, for large n, at least

$$2(N + 1)H_N - 4N + 2(N/n)(H_N - 1) + O(N^3/n^2)$$

comparisons to sort a random n-ary file of length N.

Proof. The result for multisets is proved in the discussion above, except that the theorem avoids some complications by using the fact that $\sum_{1 \le k \le n} C(x_i) = \sum_{1 \le k \le n} (x_k + 1 + \delta_{x_k 0}) > N - n$.

To prove the results for n-ary files, we follow the discussion above and start with the expression

$$N - n + n\left(1 - \frac{1}{n}\right)^N + 2 \sum_{1 < k \le n} (n - k + 1)\left(\frac{N}{n}\frac{1}{k} + \frac{1}{k^2} + \frac{1}{k^2}\left(1 - \frac{k}{n}\right)^N\right).$$

The first sum obviously evaluates to $2N(1 + 1/n)(H_n - 1) - 2N(1 - 1/n)$, as above; the second sum can be bounded by noticing that

$$\sum_{1 < k \le n} \frac{n - k + 1}{k^2} = (n + 1) \sum_{1 \le k \le n} \frac{1}{k^2} - n - H_n < n,$$

since $\sum_{1 \le k \le n} (1/k^2) < \sum_{k \ge 1} (1/k^2) = \pi^2/6$; and the third sum is even smaller in absolute value than the second, so it won't weaken our bound much to ignore it. If these expressions are all substituted in, and the $n(1 - 1/n)^N$ term is also ignored, we get the expression $2N(1 + 1/N)H_n - 3(N + n)$, as desired.

For large values of n, we can get a somewhat better bound, if we are content with an asymptotic answer. For example, the binomial theorem tells us that

$$N - n + n\left(1 - \frac{1}{n}\right)^N = N - n + n \sum_{0 \le i \le N} \binom{N}{i}\left(-\frac{1}{n}\right)^i = \sum_{1 < i \le N} \binom{N}{i}\frac{(-1)^i}{n^{i-1}}$$

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

The notation $O(N^3/n^2)$ can be assigned a precise meaning, but here it will suffice to say that the terms represented by this notation can be ignored for $n \gg N$. Now, to evaluate the sum, we begin in the same way, applying the binomial theorem to get

$$-\frac{1}{k^2} + \frac{N}{n}\frac{1}{k} + \frac{1}{k^2}\left(1 - \frac{k}{n}\right)^N = \sum_{1 < i \le N} \binom{N}{i}\left(-\frac{1}{n}\right)^i k^{i-2}.$$

After rearranging terms slightly, our lower bound becomes

$$2 \sum_{1 \le k \le n} (n - k + 1) \sum_{1 < i \le N} \binom{N}{i}\frac{k^{i-2}(-1)^i}{n^i} - \frac{1}{n}\binom{N}{2} + O\left(\frac{N^3}{n^2}\right).$$

Now, we know from Euler's summation formula that

$$\sum_{1 \le k \le n} k^{i-2} = \frac{n^{i-1}}{i-1} + \sum_{1 \le j \le i-2} \frac{B_j}{j}\binom{i-2}{j-1}n^{i-j-1},$$

where B_j are the Bernoulli numbers. Therefore,

$$2(n+1) \sum_{1 \leq k \leq n} \sum_{1 < i \leq N} \binom{N}{i} \frac{k^{i-2}(-1)^i}{n^i}$$

$$= 2(n+1) \sum_{1 \leq j \leq N-2} \frac{B_j}{jn^{j+1}} \sum_{j+2 \leq i \leq N} \binom{N}{i}\binom{i-2}{j-1}(-1)^i$$

$$+ 2\left(1+\frac{1}{n}\right) \sum_{1 < i \leq N} \binom{N}{i}\frac{(-1)^i}{i-1}$$

$$= 2(n+1) \sum_{1 \leq j \leq N-2} \frac{B_j}{jn^{j+1}}(-1)^{j+1}\left(N-j-\binom{N}{j+1}\right)$$

$$+ 2\left(1+\frac{1}{n}\right) \sum_{1 < i \leq N} \binom{N}{i}\frac{(-1)^i}{i-1}$$

$$= 2\left(1+\frac{1}{n}\right) \sum_{1 < i \leq N} \binom{N}{i}\frac{(-1)^i}{i-1} - \frac{1}{n}\left(N-1-\binom{N}{2}\right) + O\left(\frac{N^3}{n^2}\right).$$

(The inner sum evaluated on the second line of this derivation is tricky, but involves only elementary identities from Knuth [8, § 1.2.6].) Similarly, we find that

$$2 \sum_{1 \leq k \leq n} \sum_{1 < i \leq N} \binom{N}{i}\frac{k^{i-1}(-1)^i}{n^i} = 2 \sum_{1 < i \leq N} \binom{N}{i}\frac{(-1)^i}{i} - \frac{N-1}{n} + O\left(\frac{N^2}{n^2}\right).$$

Putting these results together gives a rather simple expression for our lower bound:

$$2\left(1+\frac{1}{n}\right) \sum_{1 < i \leq N} \binom{N}{i}\frac{(-1)^i}{i-1} + 2 \sum_{1 < i \leq N} \binom{N}{i}\frac{(-1)^i}{i} + O\left(\frac{N^3}{n^2}\right).$$

Finally, we can evaluate these sums by applying an identity given by Knuth [8, §1.2.7, Ex. 13].

$$\sum_{1 \leq i \leq n} \frac{x^i}{i} = H_n + \sum_{1 \leq k \leq n} \binom{n}{k}\frac{(x-1)^k}{k}.$$

If we take $x = 0$ in this formula, we get an identity for evaluating our first sum; if we integrate the equation from 0 to t, we get

$$\sum_{1 \leq i \leq n} \frac{t^{i+1}}{i(i+1)} = H_n t + \sum_{1 \leq k \leq n} \binom{n}{k}\frac{(t-1)^{k+1}}{k(k+1)} - \sum_{1 \leq k \leq n} \binom{n}{k}\frac{(-1)^{k+1}}{k(k+1)},$$

which, evaluated at $t = 1$, gives an identity for evaluating our second sum. The stated result follows immediately. $\square$

The lower bounds given in Theorem 1 are particularly weak for small values of n. For example, when $n = 2$, they grow linearly with N. As we will see, many practical implementations of Quicksort do not do so well for binary files. In fact, many implementations use $O(N^2)$ comparisons for binary files, and we can raise our lower bound for such programs.

COROLLARY. *If a Quicksort program requires, on the average, more than* $\alpha\binom{N}{2}$ *comparisons for unary or binary files of length N, then it will require at least*

$$2N\left(n+\frac{1}{n}\right)H_n - 4N - 3n + \left(1-\frac{1}{n}\right)\frac{\alpha}{n}\binom{N}{2}$$

comparisons on the average, for n-ary files of length N.

Proof. The result for unary files follows directly by not evaluating $C(x_k)$ immediately in the derivation of Theorem 1: the bound is just

$$C_{Nn} > \frac{1}{n^N} \sum_{1 \leq k \leq n} \sum_{x_1 + \cdots + x_n = N} \binom{N}{x_1, \cdots, x_n} C(x_k) + 2N\left(1+\frac{1}{n}\right)H_n - 4N - 2n.$$

To bound this term, we shall use the general identity

$$\frac{1}{n^N} \sum_{x_1 + \cdots + x_n = N} \binom{N}{x_1, \cdots, x_n}\binom{x_k, \cdots, x_{k+m-1}}{t}$$

$$= \frac{1}{n^N} \sum_{x_1 + \cdots + x_{k+1} + i + x_{k+m} + \cdots + x_n = N} \binom{i}{t}\binom{N}{x_1, \cdots, x_{k-1}, i, x_{k+m}, \cdots, x_n}$$

$$\sum_{x_k + \cdots + x_{k+m-1} = i} \binom{i}{x_k, \cdots, x_{k+m-1}}$$

$$= \frac{1}{n^N}\binom{N}{t} \sum_{x_1 + \cdots + x_{k-1} + i + x_{k+m} + \cdots + x_n = N} \binom{N-t}{x_1, \cdots, x_{k-1}, i-t, x_{k+m}, \cdots, x_n} m^i$$

$$= \frac{m^t}{n^N}\binom{N}{t} n^{N-t} = \frac{m^t}{n^t}\binom{N}{t}.$$

If $C(x_k) > \alpha\binom{x_k}{2}$, then we may take $m = 1$, $t = 2$ to get the result

$$C_{Nn} > 2N\left(1+\frac{1}{n}\right)H_n - 4N - 2n + \frac{\alpha}{n}\binom{N}{2},$$

which implies the stated bound.

For binary files, we follow exactly the derivation of Theorem 1 except that the telescoping recurrences for C and G can each be stopped one step sooner to yield

$$C(x_1, \cdots, x_n) = \sum_{1 \leq k \leq n-1} C(x_k, x_{k+1}) - \sum_{1 \leq k \leq n-2} C(x_{k+1})$$

$$+ 2 \sum_{3 \leq k+2 \leq j \leq n} \frac{x_k x_j}{x_k + \cdots + x_j}.$$

When we average over all multisets on N elements, the calculations are similar to those in the proof of Theorem 1. If $C(x_{k+1}) > \alpha\binom{x_{k+1}}{2}$, then the proof for unary files proves the corollary. Therefore we may assume that $C(x_{k+1}) \leq \alpha\binom{x_{k+1}}{2}$ and

$C(x_k, x_{k+1}) > \alpha\left(\dfrac{x_k + x_{k+1}}{2}\right)$. The identity above then tells us that

$$\frac{1}{n^N}\sum_{1 \le k \le n-1}\sum_{x_1+\cdots+x_n=N}\binom{N}{x_1,\cdots,x_n}C(x_k, x_{k+1}) > \left(1-\frac{1}{n}\right)\frac{4\alpha}{n}\binom{N}{2} \qquad (m=2, t=2)$$

and

$$\frac{1}{n^N}\sum_{1 \le k \le n-2}\sum_{x_1+\cdots+x_n=N}\binom{N}{x_1,\cdots,x_n}C(x_{k+1}) \le \left(1-\frac{1}{n}\right)\frac{\alpha}{n}\binom{N}{2} \qquad (m=1, t=2).$$

Evaluating the third term exactly as for Theorem 1, we find that

$$C_{Nn} > 2N\left(1+\frac{1}{n}\right)H_n - \frac{17}{6}n + \frac{5}{6}\frac{N}{n} + \left(1-\frac{1}{n}\right)\frac{3\alpha}{n}\binom{N}{2},$$

which implies the stated bound.

From the corollary, we conclude that if a Quicksort program is quadratic for binary files, then it is quadratic for all n-ary files when n is small. This type of effect arises often in the study of Quicksort, since all files are eventually partitioned to yield degenerate ones. It will be even more prominent in the next section, when we deal with upper bounds.

4. Upper bounds. The derivation of a general upper bound on the number of comparisons needed by any Quicksort program proceeds in much the same manner as for the lower bound. However, some extra care will be necessary for two reasons. First, a bound is needed for the number of comparisons used to partition N elements. We will be content to use $N+1$, since we shall later see some programs that use exactly that many. Other programs might use more, but if the number of comparisons that they use grows linearly with N, we can still apply our results by multiplying through by a constant. Another problem arises because, as we have seen, some partitioning methods can perform badly with files that only have a few distinct values. For the present, it will be convenient to restrict these problems to a single term by defining

$$I(x_1, \cdots, x_n) = \sum_{1 \le k \le n-2} C(x_k, x_{k+1}, x_{k+2}) - \sum_{1 \le k \le n-3} C(x_{k+1}, x_{k+2})$$

where $C(x_1, \cdots, x_n)$ is the maximum number of comparisons needed, on the average, to sort a permutation of the multiset $\{x_1 \cdot 1, \cdots, x_n \cdot n\}$. We will look at assumptions about our programs to help bound $I(x_1, \cdots, x_n)$ after we have proved

THEOREM 2. *Any Quicksort program which partitions N elements with $N+1$ comparisons will require, on the average, no more than*

$$2\sum_{4 \le k+3 \le j \le n}\frac{x_k x_j}{1 + x_{k+1} + \cdots + x_{j-1}} + I(x_1, \cdots, x_n)$$

comparisons to sort a random permutation of the multiset $\{x_1, \cdots, x_n\}$ and no more

than

$$2N\left(1 - \frac{1}{n}\right)H_n - 3N + 2\frac{N}{n} - 9\left(\frac{N}{n}\right)^2 - 7\frac{N}{n^2}$$

or, for large values of n, no more than

$$2N(H_N + 1) - 2 + O(N^2/n)$$

comparisons to sort a random n-ary file of length N.

   Proof. Arguing the same way as in the derivation for the lower bound, we start by noticing that an upper bound is certainly described by the recurrence

$$C(x_1, \cdots, x_n) = N + 1 + \frac{1}{N} \sum_{1 \le j \le n} x_j (C(x_1, \cdots, x_j - 1) + C(x_j, \cdots, x_n)), \qquad n > 1.$$

Proceeding exactly as before, we difference twice: first subtract the same equation for $\{x_2 \cdot 1, \cdots, x_n \cdot (n-1)\}$; then define $G(x_1, \cdots, x_n) = C(x_1, \cdots, x_n) - C(x_2, \cdots, x_n)$ and subtract the same equation for $\{x_1 \cdot 1, \cdots, x_{n-1} \cdot (n-1)\}$. This leaves

$$\left(\sum_{2 \le j \le n} x_j\right) G(x_1, \cdots, x_n) - \left(\sum_{2 \le j \le n-1} x_j\right) G(x_1, \cdots, x_{n-1})$$

$$= 2x_1 x_n + x_n G(x_1, \cdots, x_{n-1}, x_n - 1), \qquad n > 3$$

(Notice that this equation does not hold for $n = 2$ or 3 as was the case for the lower bound.) In order to get this equation to telescope, we multiply both sides by

$$\frac{(x_2 + \cdots + x_n - 1)!}{x_n!(x_2 + \cdots + x_{n-1})!},$$

which gives

$$\binom{x_2 + \cdots + x_n}{x_n} G(x_1, \cdots, x_n) = \binom{x_2 + \cdots + x_n - 1}{x_n - 1}(G(x_1, \cdots, x_n - 1) + 2x_1)$$

$$+ \binom{x_2 + \cdots + x_n - 1}{x_n} G(x_1, \cdots, x_{n-1}).$$

Now every place that x_n appears on the left side, $x_n - 1$ appears in the first term on the right, so this equation telescopes to yield

$$\binom{x_2 + \cdots + x_n}{x_n} G(x_1, \cdots, x_n) = \binom{x_2 + \cdots + x_n}{x_n} G(x_1, \cdots, x_{n-1})$$

$$+ 2x_1 \binom{x_2 + \cdots + x_n}{x_n - 1}$$

or

$$G(x_1, \cdots, x_n) = G(x_1, \cdots, x_{n-1}) + \frac{2x_1 x_n}{1 + x_2 + \cdots + x_{n-1}}, \qquad n > 3.$$

Substituting $G(x_1, \cdots, x_n) = C(x_1, \cdots, x_n) - C(x_2, \cdots, x_n)$ and telescoping once more leads to the desired result for permutations of multisets.

To complete the proof, for random n-ary files of length N, we will first evaluate

$$\frac{2}{n^N} \sum_{x_1+\cdots+x_n=N} \binom{N}{x_1,\cdots,x_n} \sum_{4\leq k+3\leq j\leq n} \frac{x_k x_j}{1+x_{k+1}+\cdots+x_{j-1}}.$$

Continuing as before, we interchange the order of summation and then split the inner sum and the multinomial coefficient to get

$$\frac{2}{n^N} \sum_{4\leq k+3\leq j\leq n} \sum_{x_1+\cdots+x_k+i+x_j+\cdots+x_n=N} \binom{N}{x_1,\cdots,x_k,i,x_j,\cdots,x_n}$$
$$\frac{x_k x_j}{i+1} \sum_{x_{k+1}+\cdots+x_{j-1}=i} \binom{i}{x_{k+1},\cdots,x_{j-1}}.$$

The multinomial theorem applies to the inner sum, and the remaining multinomial coefficient simplifies, leaving

$$\frac{2N}{n^N} \sum_{4\leq k+3\leq j\leq n} \sum_{x_1+\cdots+x_k+i+x_j+\cdots+x_n=N} \binom{N-1}{x_1,\cdots,x_k-1,i+1,x_j-1,\cdots,x_n}$$
$$(j-k-1)^i.$$

After replacing i by $i-1$, including a term for $i=0$, and applying the multinomial theorem twice, we are left with

$$2\frac{N}{n} \sum_{4\leq k+3\leq j\leq n} \left(\frac{1}{j-k-1} - \frac{1}{j-k-1}\left(1-\frac{j-k-1}{n}\right)^{N-1}\right),$$

or

$$2\frac{N}{n} \sum_{2\leq k\leq n-2} (n-k-1)\left(\frac{1}{k} - \frac{1}{k}\left(1-\frac{k}{n}\right)^{N-1}\right).$$

If n is not large we will not weaken our bound much by ignoring the second term, so the result

$$2N(1-1/n)H_{n-1} - 4N + 6N/n$$

follows immediately. If n is large, then we expand $(1-k/n)^{N-1}$ by the binomial theorem to get

$$-2\frac{N}{n} \sum_{2\leq k\leq n-2} \frac{n-k-1}{k} \sum_{1\leq i\leq N-1} \binom{N-1}{i}\left(-\frac{k}{n}\right)^i,$$

which reduces, after evaluating the sum on k with Euler's summation formula just as above, to

$$-2N \sum_{1\leq i\leq N-1} \binom{N-1}{i}\frac{(-1)^i}{i} - 2N \sum_{1\leq i\leq N-1} \binom{N-1}{i}\frac{(-1)^i}{i+1} + O(N^2/n).$$

The second term turns out to be $2(N-1)$, and the first is just $2NH_N$, so we have

$$2N(H_N+1) - 2 + O(N^2/n).$$

Finally, we must average $I(x_1,\cdots,x_n)$ over all multisets of length N. First, since a Quicksort program uses $N+1$ comparisons and gets at least one element in

place on each partitioning stage, a trivial upper bound is

$$C(x_1, \cdots, x_n) \leq \sum_{2 \leq k \leq N} (k+1) = \tfrac{1}{2}(N+4)(N-1) < \binom{N}{2} + 2N.$$

Therefore, we must certainly have

$$I(x_1, \cdots, x_n) < \sum_{1 \leq k \leq n-2} \left(\binom{x_k + x_{k+1} + x_{k+2}}{2} + 2(x_k + x_{k+1} + x_{k+2}) \right)$$

and, applying the identity given in the proof to Corollary 1 of Theorem 1, we then find that

$$\frac{1}{n^N} \sum_{x_1 + \cdots + x_n = N} \binom{N}{x_1, \cdots, x_n} I(x_1, \cdots, x_n) < \left(1 - \frac{2}{n}\right)\left(\frac{9}{n}\binom{N}{2} + N\right),$$

and, in particular, for large n, the right-hand side is $O(N^2/n)$.

The theorem now follows immediately from the results in the preceding two paragraphs. □

Notice that if n is $O(1)$, then the bound becomes $O(N^2)$. Again, this is a result of the recursive structure of Quicksort, and it is due to the fact that some Quicksort programs are inefficient for files with a small number of key values. On the other hand, not all Quicksort programs have this problem, and if the method works well for binary files, we can eliminate the quadratic term.

COROLLARY. *If a Quicksort program can sort a random binary file of N elements with less than $2NH_N$ comparisons, on the average, then it will require no more than*

$$2N\left(1 - \frac{1}{n}\right)H_{n-1} - 4N + 6\frac{N}{n}H_N + \frac{39}{2}\frac{N}{n},$$

comparisons to sort a random n-ary file of N elements.

Proof. The expression given is a crude approximation intended only to show that the bound is not quadratic, so our estimates will be somewhat rough. First, we can remove the term describing ternary files by noticing that, from our most general recurrence, we certainly must have

$$(x_1 + x_2 + x_3)C(x_1, x_2, x_3) \leq 2\binom{x_1 + x_2 + x_3 + 1}{2} + x_1 C(x_1, x_2, x_3)$$
$$+ x_2 C(x_1, x_2) + x_2 C(x_2, x_3) + x_3 C(x_1, x_2, x_3).$$

From this it follows that

$$C(x_1, x_2, x_3) < \frac{3}{x_2 + 1}\binom{x_1 + x_2 + x_3 + 1}{2} + C(x_1, x_2) + C(x_2, x_3),$$

and, therefore,

$$I(x_1, \cdots, x_n) < 3 \sum_{1 \le k \le n-2} \frac{1}{x_{k+1}+1} \binom{x_k + x_{k+1} + x_{k+2} + 1}{2} + \sum_{1 \le k \le n-1} C(x_k, x_{k+1}).$$

By hypothesis, we certainly have $C(x_k, x_{k+1}) < 2(x_k + x_{k+1})H_N$. The calculations involved in averaging this over all multisets of N elements are similar to those we have seen many times before. It turns out that the first term is less than $\frac{27}{2}(N/n)$, and the second is equal to $6(N/n)H_n$, so the desired result follows directly. $\square$

The upper and lower bounds that we have derived for the number of comparisons give some indication of how well we can expect to do when implementing a Quicksort for files with equal keys. If n is very, very large, then the bounds differ by only $6N - 2H_N - 2$, so we have verified the traditional argument that equal keys are unlikely to occur in this case and their effect can be ignored. If $n = O(N)$, then the upper and lower bounds differ only slightly, and we should not expect one method for dealing with equal keys to differ substantially from another. And if n is small, then the bounds tell us that we should take care to ensure that our method operates efficiently for binary files.

5. Implementations. Although it is tempting to contemplate sophisticated algorithms for dealing with equal keys during the partitioning process, we shall be content to study three methods which require virtually no overhead for their implementation. We shall see that one of these performs very well, and it is unlikely that it would be worthwhile to incur any extra overhead in Quicksort to deal with equal keys.

The first method that we shall consider is of course Program 1 as it stands, which sorts properly and efficiently when equal keys are present. If we replace "$<$" by "$\le$" and "$>$" by "$\ge$" in the discussion following the program, we find that it applies as well when equal keys are present, except for one subtle point. It is possible for the condition $i = j$ to occur outside the inner loops, so that the pointer scans ultimately terminate with $i = j + 2$, and the two keys $A[j]$ and $A[j+1]$ are put in place by partitioning. (Although we do an extraneous exchange $A[i] := A[j]$ when $j = i$, it is much less efficient to exit the loop when $j = i$ because not only is the chance to get two elements in place missed, but also when the left subfile is later partitioned, the partitioning element chosen will be the largest in that subfile.) It is important to notice such anomalies if the analysis is to be correct. In any case, although Program 1 clearly works, it is reasonable to ask if there is a more efficient method of distributing the keys equal to the partitioning element into the subfiles.

Another possibility is to change the $<$ and $>$ signs in the inner loops of Program 1 to $\le$ and $\ge$. If a key smaller than all the others is chosen as the partitioning element, the j pointer will scan past the left end of the file, so we need to protect against this case by setting $A[0] := -\infty$. Even worse, it might be possible for the pointers to access elements far outside the array bounds $A[l], \cdots, A[r]$ during intermediate partitioning stages. This situation could be avoided by putting tests in the inner loops to check the pointers, but it is more efficient to put $-\infty$ and ∞ in $A[l-1]$ and $A[r+1]$ before partitioning and restore

them afterwards. This leaves us with
    PROGRAM 2.
procedure quicksort (integer value l, r);
        comment The array A is declared to be $A[0:N+1]$ with $A[0] = -\infty$ and
            $A[N+1] = \infty$;
      if $r > l$ **then**
          $A[l-1]:=:A[0]; A[r+1]:=:A[N+1]$
          $i := l; j := r+1; v := A[l]$;
          loop:
              loop: $i := i+1$; **while** $A[i] \leqq v$ **repeat**;
              loop: $j := j-1$; **while** $A[j] \geqq v$ **repeat**;
          until $j < i$:
              $A[i]:=:A[j]$;
          repeat;
          if $j > l$ **then** $A[l]:=:A[j]; j := j-1$; **endif**;
          $A[l-1]:=:A[0]; A[r+1]:=:A[N+1]$;
          quicksort (l, j);
          quicksort (i, r);
        endif;
Notice that after partitioning we have $A[l], \cdots, A[j-1] \leqq A[j] = A[j+1] = \cdots = A[i-1] \leqq A[i], \cdots, A[r]$, with $1 \leqq i, j \leqq N+1$, so that a number of keys can be put into position on one partitioning stage.

Finally, we might consider allowing equality in only one of the inner loops of Progam 1, and leave the other inequality strict. The two possibilities are basically symmetric, and we will consider
    PROGRAM 3.
procedure quicksort (**integer value** l, r):
        comment The array A is declared to be $A[1:N+1]$ with $A[N+1] = \infty$;
      if $r > l$ **then**
          $A[r+1]:=:A[N+1]$;
          $i := l; j := r+1; v := A[l]$;
          loop:
               loop: $i := i+1$; **while** $A[i] \leqq v$ **repeat**;
              loop: $j := j-1$; **while** $A[j] > v$ **repeat**;
          until $j < i$:
              $A[i]:=:A[j]$;
          repeat;
          $A[l]:=:A[j]$;
          $A[r+1]:=:A[N+1]$;
          quicksort $(l, j-1)$;
          quicksort $(j+1, r)$;
        endif;
This program always puts exactly one partitioning element into position.

In summary, Program 1 stops the pointers on keys equal to the partitioning element, Program 2 scans over equal keys, and Program 3 puts them into the left subfile. Clearly there is a version symmetric to Program 3 which puts them into the right subfile. Versions of all of these approaches have appeared at one time or

another in the literature. Hoare's original program scanned over equal keys [6], [7], and several authors then adopted that approach [2], [5], [11], [13]. (However, the later authors "improved" Hoare's program to test if the pointers cross each time they are changed. The reader will soon appreciate how unfortunate this strategy is when, for example, all the keys are equal.) R. C. Singleton was the first to suggest stopping the pointers on keys equal to the partitioning element [15], and the idea was accepted by others [4], [9], though no analytic justification was given. The idea of putting all the keys equal to the partitioning element in one subfile or the other appears in some versions of Quicksort [1], [3], though no one has given any particular reason for doing so.

It is not at all clear *a priori* which of the programs should be recommended, for there are situations in which each performs better than the others. Figure 2 illustrates this by showing the operation of the programs on three different files, along with the number of comparisons used. (The differences between the progams are most apparent in the second example, which shows all three "sorting" seven equal keys.) Program 1 expends a few extra exchanges to get balanced partitions, Program 2 can get more than one key into place on one partitioning stage, and Program 3, due to its asymmetrical nature, can produce unbalanced partitions. In the following sections, we shall attempt to quantify these remarks by looking at the analysis of the programs. We shall see exactly how many comparisons they use, on the average, for unary and binary files, and then we shall
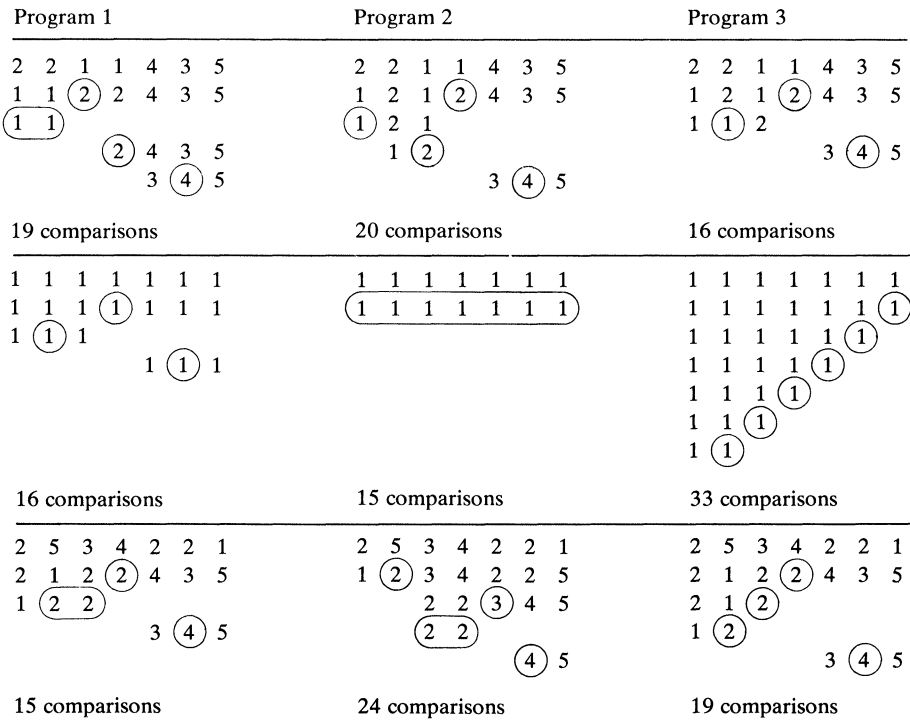

FIG. 2

prove that Program 1 must be preferred because it tends to produce partitions closer to the center.

There are a few more issues relating to the practical implementation of Quicksort on equal keys which we shall treat before moving on to the analysis. The first is a property called *stability* which is often of concern in practical sorting programs. A sorting program is *stable* if it preserves the relative order among equal keys. Unfortunately, our programs are not stable, because no matter how we treat keys equal to the partitioning element, the relative order of other keys might be disturbed. The easiest way to provide stability, if there is extra space available, is to append each key's index to itself before sorting. For example, if we are sorting small integers, this can be done by the statement

$$\textbf{loop for } 1 \leqq i \leqq N : A[i] := A[i] * N + i - 1 \textbf{ repeat};$$

This transformation makes all the keys distinct and preserves their relative order. We have $A[i] < A[j]$ before the transformation only if $A[i] < A[j]$ after the transformation; and if $i < j$ and $A[i] = A[j]$ before, then $A[i] < A[j]$ after. We can now achieve a stable method by sorting the file and then transforming back to our original keys:

$$\textbf{loop for } 1 \leqq i \leqq N : A[i] := A[i] / N \textbf{ repeat};$$

Of course, since this method is costly in terms of both time and space (each key must be a little bigger), it should not be used unless stability is important. If the extra space is not available, then Rivest has shown that a stable Quicksort involving $O(N(\log N)^2)$ comparisons can be devised [12]. This method is of limited practical utility, but it is an important theoretical result.

The programs we have defined gain efficiency by using sentinel keys, $-\infty$ and ∞, to stop the scanning pointers from going outside the array bounds. For Program 3 it is necessary for ∞ to be strictly greater than all of the other keys, and for Program 2 it is also necessary for $-\infty$ to be strictly less than the others. It may be difficult to define such keys in some practical situations. For example, if the keys to be sorted can take on any value which can be represented in one word in a computer, then by definition we cannot represent a key larger than all possible values in one word. This is not a problem for Program 1, since it only requires that ∞ be greater than or equal to all the other keys.

6. Unary files. Our derivation of the upper and lower bound suggests that we should know how our programs perform in the degenerate case when only a few distinct key values are present, so let us examine first what happens when all the keys are equal.

Program 1 comes as close as possible to dividing the file exactly in half at each stage. The number of comparisons used is described by the recurrence

$$C_N = N + 1 + 2C_{(N-1)/2}, \qquad N > 1,$$

with $C_1 = 0$. If N is of the form $2^k - 1$, then this reduces to

$$C_{2^k-1} = 2^k + 2C_{2^{k-1}-1}, \qquad k > 1.$$

Dividing both sides by 2^k, this immediately telescopes to the solution

$$C_N = (N+1) \lg ((N+1)/2), \quad \text{when } N = 2^k - 1, \quad k > 1.$$

The solution for general N is somewhat complicated because it depends on the binary representation of N, but it is easily shown by induction that $C_N \leq (N+1) \lg ((N+1)/2)$ for all N, so that Program 1 performs acceptably on unary files.

Program 2 is much easier to analyze, for it "sorts" all unary files in only one partitioning stage. Each pointer scans all the way across the file, the left and right subfiles are both empty, and a total of only $2N+1$ comparisons are used.

On the other hand, unary files represent the worst case for Program 3. Each partitioning stage only removes one element from the right end of the file to be sorted, so

$$\sum_{2 \leq j \leq N} (j+1) = \tfrac{1}{2}(N-1)(N+4)$$

comparisons are used.

It is interesting that these three programs, which seem to be so similar, perform so differently when the keys are all equal. One uses $O(N \log N)$ comparisons, the second is linear, and the third is quadratic!

7. Binary files. Now let us consider the less degenerate case when binary files are to be sorted. The analysis is more complex, but it does give us some more insight into the relative performance of the programs.

The easiest of the three to analyze is Program 2. We wish to find C_N, the average number of comparisons to sort a binary file of length N, given that all 2^N such files are equally likely. Suppose that the two values are 0 and 1, and define $C_N^{(0)}$ and $C_N^{(1)}$ to be the averages for files that start with 0 and 1, respectively, so that $C_N = \tfrac{1}{2}(C_N^{(0)} + C_N^{(1)})$. First, we will find a recurrence for $C_N^{(0)}$ by noticing that the situation after the first partitioning stage is as follows ("x" denotes keys which may be 0 or 1):

$$-\infty \quad \overbrace{0 \quad 0 \quad 0 \cdots 0}^{k} \quad \overbrace{1 \quad x \quad x \quad x \cdots x}^{N-k} \quad \infty.$$
$$ j i$$

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. This leads us to the recurrence

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

By a similar argument, we can show that

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

and therefore C_N satisfies

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

Multiplying by 2^N and replacing k by $N-k$ in the sum, we get

$$2^N C_N = 4N + 2 + \sum_{1 \le k \le N-1} 2^k(2N-k+\tfrac{1}{2}+C_k).$$

Subtracting the same equation for $N-1$, we get the recurrence

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

with the initial condition $C_1 = 3$, which telescopes to the solution

$$\tfrac{1}{4}(N^2 + 8N + 3).$$

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

Program 3 may be analyzed in a similar fashion, but the calculations are somewhat more complex. Alternatively, we can analyze Program 3 in much the same way as we developed our upper and lower bounds. (Unfortunately, the other programs don't lend themselves as easily to this kind of analysis.) The number of comparisons required by Program 3 to sort to random permutation of the multiset $\{x_1 \cdot 1, \cdots, x_n \cdot n\}$ is described by the recurrence

$$C(x_1, \cdots, x_n) = N + 1 + \frac{1}{N} \sum_{1 \le j \le n} x_j(C(x_1, \cdots, x_j - 1) + C(x_{j+1}, \cdots, x_n))$$

where $N = \sum_{1 \le j \le n} x_j$, and the notational conventions are the same as above. Proceeding in exactly the same manner as for the derivation of the upper bound, we find that

$$C(x_1, \cdots, x_n) = \sum_{1 \le k \le n} \binom{x_k + 2}{2} - n + 2 \sum_{1 \le k < j \le n} \frac{x_k x_j}{1 + x_k + \cdots + x_{j-1}}.$$

This formula is due to Burge [3], although he develops a slightly different version. For binary files, we get

$$C(x, N-x) = \binom{x+2}{2} + \binom{N-x+2}{2} - 2 + 2\frac{x(N-x)}{1+x}.$$

The average is

$$\sum_{0 \le x \le N} \frac{\binom{N}{x}}{2^N} C(x, N-x) = \frac{1}{2^{N-1}} \sum_{0 \le x \le N} \binom{x+2}{2}\binom{N}{x}$$

$$-2 + \frac{1}{2^{N-1}} \sum_{0 \le x \le N} \binom{N}{x}\frac{x(N-x)}{1+x}.$$

After application of the identities

$$\binom{x+2}{2}\binom{N}{x} = \binom{N}{2}\binom{N-2}{x-2} + 2N\binom{N-1}{x-1} + \binom{N}{x}$$

and

$$\binom{N}{x}\frac{x(N-x)}{1+x} = N\binom{N-1}{x} - \binom{N}{x+1},$$

this reduces to the solution

$$C_N = \tfrac{1}{4}(N^2 + 11N - 8) + \frac{1}{2^{N-1}},$$

so Program 3 is also quadratic for binary files.

Fortunately, we can show that Program 1 does not perform so badly on binary files, even though an exact formula for the average appears difficult to derive. What we can do is derive an upper bound on the number of comparisons taken by Program 1 on any binary file. This will of course also be a bound on the average. The proof is based on a different method of counting comparisons than we have been using. We know that each partitioning stage contributes one comparison to the total for each element involved plus one extra comparison when the pointers cross. But we can also count comparisons by counting how many partitioning stages each element is involved in, then adding N for the pointer crossing overhead (there can be no more than N partitioning stages). Notice that each partition in Program 1 results in one subfile with all keys equal and another "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 $2 \lg N$ partitions, and every element in the whole file is involved in at most $2 \lg N$ partitioning stages. Therefore the total number of comparisons must be less than $2N \lg N + N$ (and this is not a particularly tight bound). This is substantially better than the quadratic performance of Programs 2 and 3.

These results for binary files, coupled with the upper and lower bounds developed above, represent strong evidence that Program 1 is the method of choice when n is small. The corollary to Theorem 1 says that Programs 2 and 3 will be quadratic; and the corollary to Theorem 2 says that Program 1 will still require only $O(N \log N)$ comparisons, on the average, for small n. Of course, it must be noted that if it is known that n will always be small, a special-purpose sorting program written to take that fact into account might be more appropriate than the general-purpose programs that we have been studying. For example, the best way to sort a binary file is to effectively "partition" the file on the value $\tfrac{1}{2}$: scan from the left to find a 1, scan from the right to find a 0, exchange them, and continue until the pointers cross. The whole file can be sorted with $N + 1$ comparisons. Similarly, if a file is known to be ternary (consisting of 0's, 1's and 2's), it can be sorted with $2(N + 1)$ comparisons by first partitioning on the value 1, then treating the binary

subfiles as above. In the same manner, a file with $2^t + 1$ distinct values can always be sorted with $(t+1)(N+1)$ comparisons, if the values are all known. Another example, which is most useful when keys to be sorted fall into a small range, is the idea of *distribution counting* (see [9, § 5.2]), where the file is sorted in two passes: one to count the number of occurrences of each key, and a second to move the keys into place according to the counts. Such special-purpose programs may be made to outperform Program 1 under some conditions for small n; but we have shown that Program 1 does perform acceptably, and it can be expected to perform better than other general-purpose sorting programs when many equal keys are present.

8. The general case. In the general case, the exact analysis of Programs 1 and 2 appears to become intractable, so we shall adopt a more indirect approach to compare the programs. The idea is to notice that Quicksort performs best when the partitions at each stage tend to be near the center. Consequently, we would like to discover which of our algorithms produces partitions closest to the center, on the average.

When Singleton first proposed stopping the pointers on keys equal to the partitioning element [15], he claimed that it produces a "better split" than Hoare's original method of scanning over equal keys. However, he gave only empirical justification, and it is not at all obvious that this is so. For example, given the input file

$$2 \quad 2 \quad 2 \quad 2 \quad 1 \quad 1 \quad 1 \quad 2 \quad 2 \quad 3 \quad 3 \quad 3 \quad 3 \quad 3 \quad 3,$$

the first stage of Program 2 will produce the partition

$$1 \quad 2 \quad 2 \quad 2 \quad 1 \quad 1 \quad \boxed{2 \quad 2 \quad 2} \quad 3 \quad 3 \quad 3 \quad 3 \quad 3 \quad 3,$$

while Program 1 results in the less balanced partition

$$1 \quad 2 \quad 2 \quad 1 \quad 1 \quad ② \quad 2 \quad 2 \quad 2 \quad 3 \quad 3 \quad 3 \quad 3 \quad 3 \quad 3.$$

On the other hand, Program 2 performs worse for the input file

$$2 \quad 3 \quad 3 \quad 3 \quad 3 \quad 3 \quad 3 \quad 2 \quad 2 \quad 1 \quad 1 \quad 1 \quad 2 \quad 2 \quad 2,$$

since it produces the partition

$$1 \quad 1 \quad 1 \quad ② \quad 3 \quad 3 \quad 3 \quad 2 \quad 2 \quad 3 \quad 3 \quad 3 \quad 2 \quad 2 \quad 2,$$

while Program 1 partitions the file perfectly:

$$2 \quad 2 \quad 2 \quad 2 \quad 1 \quad 1 \quad 1 \quad ② \quad 2 \quad 3 \quad 3 \quad 3 \quad 3 \quad 3 \quad 3.$$

Although examples like these would seem to make comparing the algorithms difficult, it turns out that no matter how well Program 2 performs on an input file, there is another file for which Program 1 does at least as well.

THEOREM 3. *When Program 2 operates on a file $A[1], \cdots, A[N]$, it produces a partition no closer to the center than Program 1 operating on the file $A[1], A[N], A[N-1], \cdots, A[2]$.*

Proof. Specifically, let j and i define the position of the partition after Program 2 is used on $A[1], \cdots, A[N]$, so that after partitioning we have
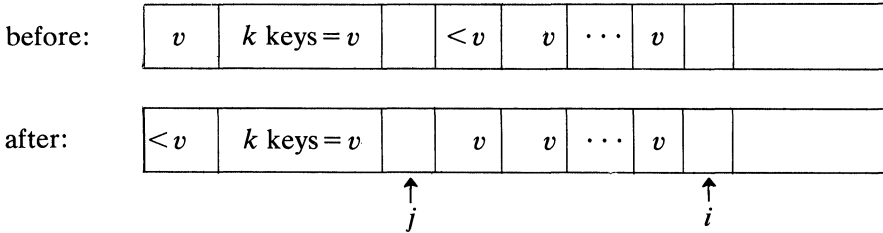
$A[1], A[2], \cdots, A[j] \leqq A[j+1] = A[j+2] = \cdots = A[i-1] \leqq A[i], \cdots, A[N]$; and let $j' + \delta$ define the position of the partition after Program 1 is used on $A[1], A[N], A[N-1], \cdots, A[2]$, so that after partitioning we have $A[1], A[2], \cdots, A[j'-1] \leqq A[j'] = A[j'+\delta] \leqq A[j'+\delta+1], \cdots, A[N]$, where δ is either 0 or 1 depending on whether the condition $i = j$ occurs outside the inner loops of Program 1. In both programs, the file is partitioned on the value of $A[1]$. Call that value v and let s be the number of keys in the file which are $<v$. Our goal will be to show that the inequality
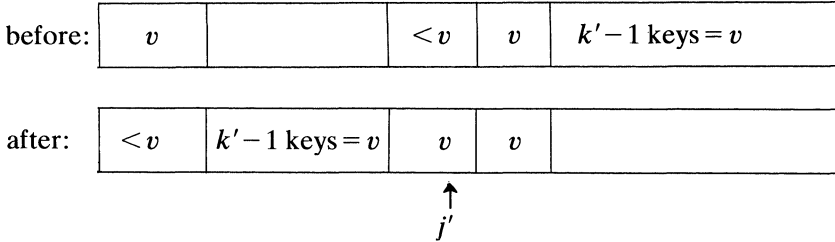
$$|j' + \delta - (N-1)/2| \leqq |s + k - (N+1)/2|,$$

holds for $j - s < k < i - s$.

First we notice that since Program 2 does not move keys which are $= v$, we can have $j = s + k$ only if exactly k of the keys $A[2], \cdots, A[s+k]$ were originally $= v$. But we also know that $A[j+1]$ was originally $<v$, and that $A[j+2], \cdots, A[i-1]$ were originally $= v$, since they were not moved by partitioning. In short, we can deduce that partitioning must have had the following effect:

before:

| v | k keys $= v$ | | $<v$ | v | $\cdots$ | v | |

after:

| $<v$ | k keys $= v$ | | v | v | $\cdots$ | v | |

            $\uparrow$ $\uparrow$
            j i

In the original file, exactly $k - 1$ of the keys $A[2], \cdots, A[s+k]$ were originally $= v$ for all k in the range $j - s < k < i - s$. Similarly, since Program 1 always moves keys which are $= v$, then $j' = s + k'$ for some fixed k' only if there were exactly $k' - 1 + \delta$ keys $= v$ in the last $N - j' + 1$ positions of the reverse file: $A[N - s - k' + 2], \cdots, A[2]$. The effect of partitioning when $\delta = 1$ is

before:

| v | | | $<v$ | v | $k' - 1$ keys $= v$ |

after:

| $<v$ | $k' - 1$ keys $= v$ | | v | v | |

            $\uparrow$
           j'

and the diagram for $\delta = 0$ is similar.

To complete the proof it is necessary to consider three cases depending on the relative values of k and $k' + \delta$. If $k = k' + \delta$, then the inequality $|j' + \delta - (N+1)/2| \leqq |s + k - (N+1)/2|$ obviously holds. If $k > k' + \delta$, then the discussion above says that $A[2], \cdots, A[s+k]$ has more keys $= v$ than $A[N - s - k' + 2], \cdots, A[2]$. This can only be true if $s + k > N - s - k' + 2$, or $j' + \delta \geqq N - s - k + 1$. Now, if $j' + \delta - (N+1)/2$ is $\geqq 0$, then $k' + \delta < k$ implies that $0 \leqq j' + \delta - (N+1)/2 \leqq s + k - (N+1)/2$; and if $(N+1)/2 - j' - \delta$ is $\geqq 0$, then $j' + \delta \geqq N - s - k + 1$ implies that $0 \leqq (N+1)/2 - j' - \delta \leqq s + k - (N+1)/2$. In either case, taking absolute values gives the desired result, $|j' + \delta - (N+1)/2| \leqq$

$|s+k-(N+1)/2|$. If $k<k'+\delta$, then $A[N-s-k'+2], \cdots, A[2]$ has more keys $=v$ than $A[2], \cdots, A[s+k]$. But also we know that $A[N-s-k'+2]<v$, so we must have $N-s-k'+1>s+k$, or $j'+\delta \leq N-s-k+1$. An argument symmetric to the above shows that $|j'+\delta-(N+1)/2| \leq |s+k-(N+1)/2|$, and we have shown that this inequality holds for all k in the range $j-s<k<i-s$.

The theorem follows immediately from this inequality. If the first partition is to the left of center ($i-1<(N+1)/2$), then the second is at least as close ($|(N+1)/2-j'-\delta|<|(N+1)/2-i|$); and the symmetric argument holds for the right. If the first partition straddles the center, or $j+1 \leq (N+1)/2 \leq i-1$, then $|s+k-(N+1)/2| \leq \frac{1}{2}$ for some k, and therefore $|j'-\delta-(N+1)/2| \leq \frac{1}{2}$, or the second partition must also be at the center.   □

A direct consequence of Theorem 3 is that if the files $A[1], \cdots, A[N]$ and $A[1], A[N], \cdots, A[2]$ always appear with equal probability as input files (for example, if a random permutation of a multiset or a random n-ary file is being sorted), then Program 2 will produce a partition no closer to the center, on the average, than Program 1.

We cannot make quite the same statement when comparing Program 3 with Program 1. For example, when sorting a random permutation of the multiset $\{5 \cdot 1, 1 \cdot 2, 1 \cdot 3, 1 \cdot 4, 1 \cdot 5\}$, the programs will produce the same first partition when 2, 3, 4 or 5 is the partitioning element, but Program 3 will always partition in the center when 1 is the partitioning element, while Program 1 will not. Of course, any advantage gained in this case will be lost because Program 3 will be left with a large unary file for which it requires $O(N^2)$ comparisons. In addition, we can prove the following analogue to Theorem 3.

THEOREM 4. *Consider two files* $A[1], \cdots, A[N]$ *and* $A'[1], \cdots, A'[N]$ *satisfying* $1 \leq A[i], A'[i] \leq n$ *and* $A'[i]=n+1-A[i]$ *for* $1 \leq i \leq N$. *The average position of the partition when Program* 3 *operates on these files is no closer to the center than the average position of the partition when Program* 1 *operates on them.*

Proof. Suppose that in $A[1], \cdots, A[N]$ there are s keys $<A[1]$, t keys $=A[1]$, and u keys $>A[1]$, so $s+t+u=N$. Then we also know that in $A'[i], \cdots, A'[N]$ there are u keys $<A'[1]$, t keys $=A'[1]$, and s keys $>A'[1]$. Since Program 3 puts keys equal to the partitioning element in the left subfile, it partitions $A[1], \cdots, A[N]$ at $s+t$ and it partitions $A'[1], \cdots, A'[N]$ at $u+t$. On the other hand, Program 1 puts $A[s+j_1]$ into place when partitioning $A[1], \cdots, A[N]$ and $A'[u+j_2]$ into place when partitioning $A'[1], \cdots, A'[N]$, where j_1 and j_2 are fixed between 1 and t.

We wish to show that

$$\left| s+t-\frac{N+1}{2} \right| + \left| u+t-\frac{N+1}{2} \right| \geq \left| s+j_1-\frac{N+1}{2} \right| + \left| u+j_2-\frac{N+1}{2} \right|.$$

If $s+t<(N+1)/2$ and $u+t<(N+1)/2$, then, since $s+t+u=N$, we must have $t<1$ which is impossible. If $s+t$ and $u+t$ are both $\geq(N+1)/2$, then we can remove the absolute value signs to get $2t \geq j_1+j_2$, which clearly holds. If $s+t \geq (N+1)/2$ and $u+t<(N+1)/2$, then the proof is more complex (and the case $s+t<(N+1)/2$ and $u+t \geq (N+1)/2$ is clearly symmetric). If we also have $s+j_1<(N+1)/2$, then (since $u+t<(N+1)/2$ implies that $u+j_2<(N+1)/2$) we

can remove absolute value signs in the inequality to get

$$s + t - \frac{N+1}{2} + \frac{N+1}{2} - u - t \geqq \frac{N+1}{2} - s - j_1 + \frac{N+1}{2} - u - j_2$$

or

$$s - u > t - j_1 - j_2.$$

But this inequality holds because $u + t < (N+1)/2$ and $u + t + s = N$ implies that $s + 1 > (N+1)/2$, so we have $u + t < (N+1)/2 < s + 1$, or $s - u \geqq t$. Finally, we must consider the case where $s + t \geqq (N+1)/2$, $u + t < (N+1)/2$, and $s + j_1 \geqq (N+1)/2$. Removing absolute values, our inequality reduces to

$$s - u \geqq s - u + j_1 - j_2,$$

which holds unless $j_1 > j_2$. Following the logic in the proof of Theorem 3, if we were to have $j_1 > j_2$, this would imply that the number of keys equal to $A[1]$ in $A[s + j_1], \cdots, A[N]$ must exceed the number of keys equal to $A'[1]$ in $A'[u + j_2], \cdots, A'[N]$. Since our transformation between A and A' preserves equality among keys, this can only be true if $s + j_1 < u + j_2$. But we know that $s > u$, since we have $s + t \geqq (N+1)/2 > u + t$, so this implies that $j_1 < j_2$, a contradiction. $\square$

As above, we know from this theorem that if the files $A[1], \cdots, A[N]$ and $A'[1], \cdots, A'[N]$ appear with equal probability as input files (for example, if a random n-ary file or a random symmetrically distributed multiset is being sorted) then Program 3 will produce a partition no closer to the center, on the average, than Program 1.

These theorems, of course, do not represent complete evidence that the total average running time of Program 1 will always be lower than the total average running time of Programs 2 and 3. We have already seen anomalous cases where Program 1 may be slightly slower. Also, we should note that although Program 1 may use extra exchanges to get the partition close to the center, this is more than compensated for by the effects of having the partition more balanced. When the partition is closer to the center, all aspects of total running time are improved, and Theorems 3 and 4 are strong general results.

9. Conclusion. The evidence in favor of stopping the scanning pointers on keys equal to the partitioning element in Quicksort is overwhelming. Theorems 1 and 2 and our analyses of the operation of the programs on unary and binary files show that this method will always require $O(N \log N)$ comparisons on the average, when other methods can be quadratic. Theorems 3 and 4 indicate that it will produce more balanced partitions, on the average, than other methods for most reasonable input distributions. Furthermore, it is easier to implement.

Before it can be recommended for use in a practical situation, three major improvements (fully described in [9] and [14]) must be applied to Program 1. First, the recursion should be removed, and the smaller subfile sorted first. This removes some overhead, and ensures that only limited extra space will be necessary to implement the recursive stack. It applies to all our programs, and has little effect on our results. Second, the partitioning element should be chosen by taking the median of the first, middle, and last element of the file. This not only tends to

balance the partitions and so reduce the running time, but also it makes the worst case less likely to occur in a real file, an important practical consideration. This is another advantage of Program 1 over Programs 2 and 3 because even with equal keys present, it is unlikely that a file for which Program 1 will perform really badly will arise in practice, while the others run badly for *any* file with a small number of distinct values. Third, small subfiles should be ignored during partitioning and a single insertion sort applied to the entire file afterwards. This eliminates a considerable amount of overhead, since Quicksort is inefficient on files of about ten elements or less. Its effect on our analyses is to reduce the significance of the differences between the programs, since the anomalies created by small files are removed.

The utility of a general-purpose sorting program may be measured by the range of input files over which it performs efficiently. Quicksort has been shown to be more efficient than most other sorting algorithms for files with distinct keys, but few sorting algorithms have been studied in the case when equal keys are present. The results of this paper demonstrate that the range of files over which a properly implemented Quicksort can run efficiently may be extended to include files with equal keys.

REFERENCES

[1] A. AHO, J. HOPCROFT AND J. ULLMAN, *The Design and Analysis of Computing Algorithms*, Addison-Wesley, Reading, MA, 1974.

[2] J. BOOTHROYD, *Sort of a section of the elements of an array by determining the rank of each element (Algorithm 25)*, Computer J., 10 (1967), pp. 308–310. (See notes by R. Scowen, Computer J., 12 (1969), pp. 408–409, and by A. Woodall, Computer J., 13 (1970), pp. 295–296.)

[3] J. DONNER, *Tree acceptors and some of their applications*, J. Comput. System Sci., 4 (1970), J. Assoc. Comput. Mach., 23(1976), pp. 451–454.

[4] E. DIJKSTRA, *EWD316: A short introduction to the art of programming*, Technical University Eindhoven, The Netherlands, 1971.

[5] T. HIBBARD, *Some combinatorial properties of certain trees with applications to searching and sorting*, J. Assoc. Comput. Mach., 9 (1962), pp. 13–18.

[6] C. A. R. HOARE, *Partition (Algorithm 63), Quicksort (Algorithm 64), and Find (Algorithm 65)*, Comm. ACM, 4 (1961), pp. 321–322. (See also certification by J. Hillmore, Comm. ACM, 5 (1962), p. 439, and by B. Randell and L. Russell, Comm. ACM, 6 (1963), p. 446.)

[7] ———, *Quicksort*, Computer J., 5 (1962), pp. 10–15.

[8] D. KNUTH, *Fundamental Algorithms*, The Art of Computer Programming 1, Addison-Wesley, Reading, MA, 1968.

[9] ———, *Sorting and Searching*, The Art of Computer Programming 3, Addison-Wesley, Reading, MA, 1972.

[10] ———, *Structured programming with go to statements*, Comput. Surveys, 6 (1974), pp. 261–301.

[11] R. RICH, *Internal Sorting Methods Illustrated with PL/I Programs*, Prentice-Hall, Englewood Cliffs, NJ, 1972.

[12] R. RIVEST, *A fast stable minimum-storage sorting algorithm*, Institut de Recherche d'Informatique et d'Automatique Rapport 43, 1973.

[13] R. SCOWEN, *Quickersort (Algorithm 271)*, Comm. ACM, 8 (1965), pp. 669–670. (See also certification by C. Blair, Comm. ACM, 9 (1966), p. 354.)

[14] R. SEDGEWICK, *Quicksort*, Ph.D. thesis, Stanford Univ., Stanford, CA, 1975. (Also Stanford Computer Science Rep. STAN-CS-75-492.)

[15] R. SINGLETON, *An efficient algorithm for sorting with minimal storage (Algorithm 347)*, Comm. ACM, 12 (1969), pp. 185–187. (See also remarks by R. Griffin and K. Redish, Comm. ACM, 13 (1970), p. 54, and by R. Peto, Comm. ACM, 13 (1970), p. 624.)