Acta Informatica 7, 327—355 (1977)
© by Springer-Verlag 1977

# The Analysis of Quicksort Programs*

Robert Sedgewick

Received January 19, 1976

*Summary.* The Quicksort sorting algorithm and its best variants are presented and analyzed. Results are derived which make it possible to obtain exact formulas describing the total expected running time of particular implementations on real computers of Quicksort and an improvement called the median-of-three modification. Detailed analysis of the effect of an implementation technique called loop unwrapping is presented. The paper is intended not only to present results of direct practical utility, but also to illustrate the intriguing mathematics which arises in the complete analysis of this important algorithm.

## 1. Introduction

In 1961–62 C.A.R. Hoare presented a new algorithm called Quicksort [7, 8] which is suitable for putting files into order by computer. This method combines elegance and efficiency, and it remains today the most useful general-purpose sorting method for computers. The practical utility of the algorithm has meant not only that it has been subjected to countless modifications (though few real improvements have been suggested beyond those described by Hoare), but also that it has been used in countless applications, often to sort very large files. Consequently, it is important to be able to estimate how long an implementation of Quicksort can be expected to run, in order to be able to compare variants or estimate expenses. Fortunately, as we shall see, this is an algorithm which can be analyzed. (Hoare recognized this, and gave some analytic results in [8].) It is possible to derive exact formulas describing the average performance of real implementations of the algorithm.

The history of Quicksort is quite complex, and a full survey of the many variants which have been proposed is given in [17]. In addition, [17] gives analytic results describing many of the improvements which have been suggested for the purpose of determining which are the most effective. There are many examples in [17] which illustrate that the simplicity of Quicksort is deceiving. The algorithm has hidden subtleties which can have significant effects on performance. Furthermore, as we shall see, simple changes to the algorithm or its implementation can radically change the analysis. In this paper, we shall consider in detail how practical implementations of the best versions of Quicksort may be analyzed.

In this paper, we will deal with the analysis of: (i) the basic Quicksort algorithm; (ii) an improvement called the "median-of-three" modification which reduces the average number of comparisons required; and (iii) an implementation technique called "loop unwrapping" which reduces the amount of overhead per comparison. These particular methods not only represent the most effective vari-

* This work was supported in part by the Fannie and John Hertz Foundation, and in part by the National Science Foundation Grants No. GJ-28074 and MCS75-23738.

ants of Quicksort, but also they illustrate the interrelationships between algorithm, implementation and analysis.

A purpose of this paper is to demonstrate that a program of practical utility and importance can be fully analyzed mathematically. This analysis will show us precisely how effective our improvements are, and it will allow us to predict exactly how long we may expect the programs to run. The analysis is sometimes difficult and complicated due to the many details which must be accounted for, but it is also sometimes elegant and fascinating. We shall make use of a variety of techniques and concepts from concrete mathematics, and this paper is as much about the analysis of algorithms as it is about Quicksort.

It is common in studying the performance of programs to carry out an approximate analysis by (i) characterizing the program in terms of a few basic operations (such as "comparisons" and "exchanges" for sorting programs); or (ii) dealing only with the "leading term" of the running time. While such approximate results can be useful in classifying algorithms, it can be dangerous to use them to compare specific programs, and many erroneous conclusions have been drawn in the literature. In this paper we shall derive *exact* formulas for the *total* average running time of the programs as implemented on a typical computer. These results allow us to properly choose parameters left unspecified during the implementation, to intelligently compare the programs, and to accurately predict their performance.

The two main topics of this paper, analysis of algorithms and Quicksort, have been treated extensively elsewhere in the literature. Hoare's original paper [8] and Sedgewick's thesis [17] cover most of what is known about the Quicksort algorithm and its analysis; and Knuth's series of books [9, 10, 11] describe most of the techniques that are known about algorithmic analysis in general. (In addition, [11] contains an excellent treatment of Quicksort.) A full treatment of practical issues involved in real implementations can be found in [19]. This paper complements these works in the following ways. First, the best of the algorithms are chosen, based on the results in [17], and those specific algorithms analyzed. Second, the analysis of loop unwrapping is much more complete than that given in [17]. In particular, a new analysis of the effect of multiple unwrapping is given. Third, the analysis of Quicksort and the median-of-three modification is organized in a manner less specific to the algorithms, so that it may be used directly in the analysis of future modifications. Also, the algorithms have been updated so that they work efficiently when equal keys are present (see [18]). Although this paper has been designed to stand alone, a reader who feels that details are lacking should consult [17]; someone wishing to implement a practical sorting program should read [19]; a reader interested in the analysis of algorithms should be familiar with [9–11]; and anyone interested in Quicksort should certainly read [8].

## 2.1. Basic Algorithm

Quicksort is a "divide and conquer" procedure which sorts a file $A[1], \ldots, A[N]$ by first rearranging it to make the condition $A[1], \ldots, A[j-1] \leqq A[j] \leqq A[j+1], \ldots, A[N]$ hold for some $j$, then recursively applying the same procedure to the subfiles $A[1], \ldots, A[j-1]$ and $A[j+1], \ldots, A[N]$. There are several ways to specify this rearrangement procedure, which is called "partitioning". (Due to the many details which must be attended to, nearly all published implementations of Quicksort differ.) The following method has been shown to have several desirable properties:

```
**procedure** quicksort (**integer value** $l, r$);
    **comment** The array $A$ is declared to be $A[1:N+1]$, with $A[N+1]=\infty$;
    **if** $r-l \geqq M$ **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**;
        $A[l] :=: A[j]$;
        **if** $j-l < r-i+1$
            **then** quicksort $(l, j-1)$; quicksort $(i, r)$;
            **else** quicksort $(i, r)$; quicksort $(l, j-1)$;
        **endif**;
    **endif**;
```

(This program uses an exchange operator $:=:$ and the control constructs **loop**... **repeat** and **if**...**endif**, which are like those described by D. E. Knuth in [12].)

If $M=0$ and all the keys are distinct, then the program operates exactly as described above. First the file is partitioned: the leftmost element is chosen as the partitioning element; then the rest of the array is divided 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 pointers cross. The loop terminates with $j+1=i$ at which point the exchange $A[l]:=:A[j]$ completes the job of partitioning $A[l], \ldots, A[r]$. (The notation $A[N+1]=\infty$ is meant to indicate that $A[N+1]$ must be $\geqq$ all of $A[1], \ldots, A[N]$: this condition is included to stop the $i$ pointer in the case that $v$ is the largest of the keys.) After partitioning, the smaller of the two subfiles is sorted first to limit the recursive depth required. The procedure call "quicksort $(1, N)$" will therefore sort $A[1], \ldots, A[N]$ (if $M=0$).

The parameter $M$ is included in response to the observation that the program is not particularly efficient for small files. A method which is known to be efficient for small files is insertion sorting: scanning through the file, inserting each element into place among those previously considered by moving smaller elements up to make room. It may be implemented as follows:

```
**procedure** insertionsort $(l, r)$:
    **loop for** $r-1 > i > l$:
        **if** $A[i] > A[i+1]$ **then**
            $v := A[i]; j := i+1$;
            **loop**: $A[j-1] := A[j]; j := j+1$ **while** $A[j] < v$ **repeat**;
            $A[j-1] := v$;
        **endif**;
    **repeat**;
```

| | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | 3 | 1 | 4 | 1 | 5 | 9 | 2 | 6 | 5 | 3 | 5 | 8 | 9 | 7 | 4 | 3 |
| Quicksort: | 2 | 1 | 3 | 1 | **3** | **3** | 9 | 6 | 5 | 5 | 5 | 8 | 9 | 7 | 9 | 4 |
| | | | | | | | 7 | 6 | 5 | 5 | 5 | 8 | 4 | **9** | **9** | **9** |
| | | | | | | | 4 | 6 | 5 | 5 | 5 | 7 | 8 | | | |
| Insertionsort: | 2 | 1 | 3 | 1 | 3 | 3 | 4 | 6 | 5 | 5 | 5 | 7 | 8 | 9 | 9 | 9 |
| | | | | | | | | 5 | 5 | 5 | 6 | | | | | |
| | | | 1 | 3 | | | | | | | | | | | | |
| | 1 | 1 | 2 | | | | | | | | | | | | | |
| | 1 | 1 | 2 | 3 | 3 | 3 | 4 | 5 | 5 | 5 | 6 | 7 | 8 | 9 | 9 | 9 |

Fig. 1. Quicksorting $\pi$

(The notation $r-1 \geqq i \geqq l$ means that $i$ takes on the values $r-1$, $r-2, \ldots$, $l+1$, $l$ in that order.) The obvious way to improve Quicksort is to replace the last "**endif**" with "**else** insertionsort $(l, r)$ **endif**". However, it turns out to be better to ignore small subfiles during partioning, then insertion sort the whole file afterwards. The procedure calls

```
if $N > M$ then quicksort $(1, N)$ endif;
insertionsort $(1, N)$;
```

will quite efficiently sort $A[1], \ldots, A[N]$. One aim of the analysis of these procedures below is to determine the best value for the parameter $M$.

When we refer to the "basic Quicksort algorithm" below, we will be considering these two procedures, invoked in this way.

If equal keys are present in the files to be sorted, the reader may verify that the programs above still operate properly and efficiently, though not exactly as described above. The subject of Quicksort with equal keys is treated in detail in [18]. In this paper, we shall assume throughout that the keys being sorted are distinct.

Figure 1 shows the operation of Quicksort, with $M=5$, on the first 16 digits of $\pi$. Each line in the Quicksort section is the result of one "partitioning stage", and boldface elements are those put into position by partitioning. Each line in the insertionsort section is the result of a non-trivial insertion, and the elements shown are those moved.

### 2.2. Analysis of the Basic Algorithm

The total running time of the sorting method described above can, for most implementations, be described in terms of the five variable quantities

$A$ — the number of partitioning stages,

$B$ — the number of exchanges during partitioning,

$C$ — the number of comparisons during partitioning,

$D$ — the number of insertions, and

$E$ — the number of keys moved during insertion.

In the Quicksort procedure above, $C$ is the number of times $i := i+1$ is executed plus the number of times $j := j-1$ executed within the scanning loops; $B$ is the number of times $A[i] :=: A[j]$ is executed in the partitioning loop; and $A$ is the number of times the main loop is iterated. In the insertionsort procedure, $D$ is the number of times $v$ is changed, and $E$ is the number of times $A[j-1] := A[j]$ is executed within the inner loop.

Other quantities may be involved in some implementations. For example, the two alternatives of the statement "**if** $j-l<r-i$..." might require the execution of different amounts of code on some computers, because of lack of symmetry in the instruction set or too few general registers. In this case, it would be necessary to know the number of times the left subfile is smaller than the right. Such quantities can be studied in exactly the same way as the standard ones, using the methods described below.

As another example, consider what happens if the recursion in the Quicksort program above is transformed into an iteration based on an explicit stack. The standard way to effect this transformation involves moving the test for small subfiles to the point of invocation of the recursion, so that subfiles of $M$ or fewer elements never are put on the stack. In this case, the total running time of the program will depend on an additional quantity

$S$ — the number of stack pushes.

This is the number of times both subfiles have more than $M$ elements. It will arise in determining the running time of careful implementations of Quicksort.

The goal of our analysis in this paper will be to find the average values of the various quantities under the assumption that the keys $A[1], \ldots, A[N]$ are distinct and randomly ordered. From this information it is easy to calculate the total expected running time for any particular implementation. (It is also possible to find the maximum and minimum possible values of the running time, and to estimate the variance [17]. Furthermore, some results have been obtained for the case when equal keys are present [18].)

To calculate the total running time, it is necessary to determine the overhead associated with each quantity. For the model in [11] and [17], where instructions which do not reference memory cost one time unit and instructions which do reference memory cost two time units, the total running time is

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

(with $3D+8E+7N-6$ contributed by the insertionsort). The relative values of these coefficients are typical, and similar expressions can easily be derived for any particular implementation on any particular real machine.

The basis for the analysis is to take advantage of the recursive structure of the program to set up recurrence relations describing the average value of the various quantities. It is not difficult to verify that the subfiles produced by partitioning in the Quicksort program above are random (although some partitioning methods do not preserve randomness [12, 17]). This means that by calculating the average values of the quantities for the first partitioning stage, we can set up equations describing their average values for the whole program.

When a random file of $N$ elements is partitioned, the $k$th smallest is used as the partitioning element with probability $1/N$, leaving random subfiles with $k-1$ and $N-k$ elements. Therefore, all of the quantities satisfy recurrences of the form

$$F_N = f_N + \frac{1}{N} \sum_{1\leq k\leq N} (F_{k-1} + F_{N-k}) \qquad \text{for } N > M, \tag{1}$$

where $F_N$ denotes the average value of some quantity when a random arrangement of $N$ items is sorted, and $f_N$ denotes the average value of that quantity for the first partitioning stage. (For the quantity $S$, the recurrence holds only for $N > 2M+1$; in the solution for $S$ we will substitute $2M+1$ for $M$ everywhere.)

The validity of (1) for the quantities $D$ and $E$ depends on some particular properties of insertionsort. Full details may be found in [17]. Both quantities count combinatorial properties of the permutation being sorted: $D$ is the number of keys which have at least one smaller element to their right (or $N$ minus the number of "right to left minima"), and $E$ is the number "inversions" (the number of pairs $i$, $j$ such that $A[i]$ is smaller than and to the right of $A[j]$). For random files, it turns out that $D_N = N - H_N$ and $E_N = N(N-1)/4$. These are our answers for $N \leq M$. The total expected running time of insertionsort (and of Quicksort for $N \leq M$, except that the cost of the test "**if** $N > M$" must be added) is $2N^2 + 8N - 3H_N - 6$ time units. For $N > M$ we notice that, after partitioning, all keys smaller than and to the right of any key must be in the same subfile as that key. Therefore, the values of $D$ and $E$ for the whole file after partitioning are simply the sums of the values for the subfiles, and (1) holds with $f_N = 0$. It is this property which allows us to insertionsort the whole file after partitioning rather than having to insertionsort the small subfiles during partitioning.

Using standard manipulations, we can explicitly solve the recurrence (1) to get a formula for $F_N$ in terms of $f_N$. First, change $k$ to $N+1-k$ in the second part of the sum:

$$F_N = f_N + \frac{2}{N} \sum_{1\leq k\leq N} F_{k-1} \qquad \text{for } N > M.$$

Next multiply by $N$ and subtract the same formula for $N-1$:

$$NF_N - (N-1)F_{N-1} = \nabla N f_N + 2F_{N-1} \qquad \text{for } N-1 > M.$$

We have used the "backward difference operator" notation $\nabla$ in this equation. (For convenience, we adopt the convention that $\nabla N f_N \equiv \nabla(Nf_N) = Nf_N - (N-1) f_{N-1}$.) Simplifying and dividing by $N(N+1)$, we have

$$\frac{F_N}{N+1} = \frac{F_{N-1}}{N} + \frac{\nabla N f_N}{N(N+1)} \qquad \text{for } N > M+1.$$

This immediately telescopes to the solution

$$F_N = (N+1)\left(\frac{F_{M+1}}{M+2} + \sum_{M+2\leq k\leq N} \frac{\nabla k f_k}{k(k+1)}\right) \qquad \text{for } N > M. \tag{2}$$

Now, depending on the form of the function $f_k$, the evaluation of this sum could involve some tedious calculations. We can simplify it by first rearranging the terms,

$$\sum_{M+2\leq k\leq N} \frac{\nabla k f_k}{k(k+1)} = \sum_{M+2\leq k\leq N} \frac{f_k - f_{k-1}}{k+1} + \sum_{M+2\leq k\leq N} \frac{f_{k-1}}{k(k+1)},$$

and then applying "summation by parts" to the second sum, as follows:

$$= \sum_{M+2\leq k\leq N} \frac{\nabla f_k}{k+1} + \sum_{M+2\leq k\leq N} \frac{f_{k-1}}{k} - \sum_{M+2\leq k\leq} \frac{f_{k-1}}{k+1},$$

$$= 2 \sum_{M+2\leq k\leq N} \frac{\nabla f_k}{k+1} + \frac{f_{M+1}}{M+2} - \frac{f_N}{N+1}.$$

Substituting, we have an alternate form of the solution to the recurrence (1):

$$F_N = 2(N+1) \sum_{M+2\leq k\leq N} \frac{\nabla f_k}{k+1} + \frac{N+1}{M+2}(f_{M+1}+F_{M+1}) - f_N \quad \text{for } N > M. \quad (3)$$

Summation by parts is a useful technique for manipulating differences within summations that we shall encounter again (see [9]).

From these general formulas we can easily derive the solutions given in [11] and [17] for the average values of the various quantities. For example, for the number of comparisons, $f_N = N+1$, $\nabla f_k = 1$, and $f_{M+1} = F_{M+1} = M+2$, so the solution $C_N = (N+1)(2H_{N+1} - 2H_{M+2} + 1)$ follows immediately. For the number of partitioning stages, we have $f_N = f_{M+1} = F_{M+1} = 1$ and $\nabla f_k = 0$, so $A_N = 2\frac{N+1}{M+2} - 1$.

For the insertionsort quantities, we have $f_N = 0$ from the discussion above, so $D_N = \frac{N+1}{M+2} D_{M+1}$ and $E_N = \frac{N+1}{M+2} E_{M+1}$. The values of $D_{M+1}$ and $E_{M+1}$ follow from (1) and the values for random files:

$$D_{M+1} = \frac{2}{M+1} \sum_{1\leq k\leq M+1} (k-1-H_{k-1}) = M+2-2H_{M+1};$$

and

$$E_{M+1} = \frac{2}{M+1} \sum_{1\leq k\leq M+1} \frac{(k-1)(k-2)}{4} = \frac{M(M-1)}{6}.$$

The function $f_N$ for the number of stack pushes is slightly more complicated. If $N \leq 2M+2$, then there will be no stack push, since one of the subfiles must have $\leq M$ elements. In particular, $F_{2M+2} = 0$. For $N > 2M+2$ there is a stack push if and only if the rank of the partitioning element is between $M+2$ and $N-M-1$, so the average number of stack pushes i $\frac{1}{N}(N-2M-2)$. Therefore $\nabla k f_k = 1$ for $k > 2M+2$, 0 otherwise, and from the solution (2) we find that $S_N = \frac{N+1}{2M+3} - 1$.

To find $B_N$, we have to first calculate the average number of exchanges used during the first partitioning stage. If $A[1]$ is the $k$th smallest element in the file, this is the number of keys among $A[2], \ldots, A[k]$ which are $> A[1]$. There are exactly $t$ such keys with probability $\binom{N-k}{t}\binom{k-1}{k-1-t}\Big/\binom{N-1}{k-1}$. Averaging, and removing the condition on $k$, we have an expression for the average number of exchanges used during the first partitioning stage:

$$\frac{1}{N}\sum_{1\le k\le N}\sum_{0\le t\le k-1} t\frac{\binom{N-k}{t}\binom{k-1}{k-1-t}}{\binom{N-1}{k-1}}$$

$$=\frac{1}{N}\sum_{1\le k\le N}\frac{N-k}{\binom{N-1}{k-1}}\sum_{0\le t\le k-1}\binom{N-k-1}{t-1}\binom{k-1}{k-1-t}.$$

Two applications of Vandermonde's convolution lead to the result $\frac{N-2}{6}$. Therefore by the linearity of the recurrences, $B_N=\frac{1}{6}C_N-\frac{1}{2}A_N$.

To summarize, we know that Quicksort requires, on the average,

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

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

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

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

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

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

(The formula for $S_N$ is valid for $N>2M+1$; the others are valid for $N>M$.) By assigning appropriate coefficients to these expressions, we can compute the total average running time. These coefficients are dependent on the amount of time required by each of the instructions of the program. This is of course dependent on the particular compiler and machine used. For the typical machine cited above and described in [11] and [17], the total running time is $24A+11B+4C+3D+8E+9S+7N$ time units. Notice that the quantities which have high average values, $B$ and $C$, have low coefficients. This is what makes Quicksort quick, and no matter what compiler or machine it is implemented on, these coefficients can and should be kept low. (Any algorithm can be improved either by reducing the average value of the various quantities *or* by lowering the coefficients, so a proper implementation is very important.) Using these coefficients, we find that the total average running time of Quicksort is

$$\frac{35}{3}(N+1)H_{N+1}-\frac{69}{2}$$
$$+\frac{1}{6}(N+1)\left(8M+71-70H_{M+2}+\frac{270}{M+2}+\frac{54}{2M+3}-36\frac{H_{M+1}}{M+2}\right) \quad (4)$$

time units. This formula is accurate for $N>2M+1$ only; for $M<N\leqq 2M+1$ there are no stack pushes so $9S_N$ must be substracted; and for $1\leqq N\leqq M$ the time is $2N^2+8N-3H_N-3$, contributed entirely by insertionsort.

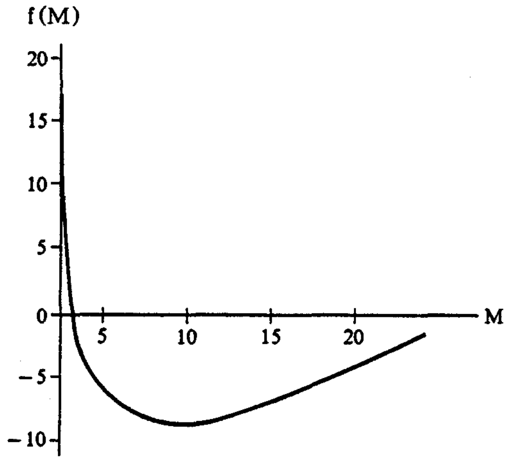

Fig. 2a. Contribution of $M$ in Quicksort

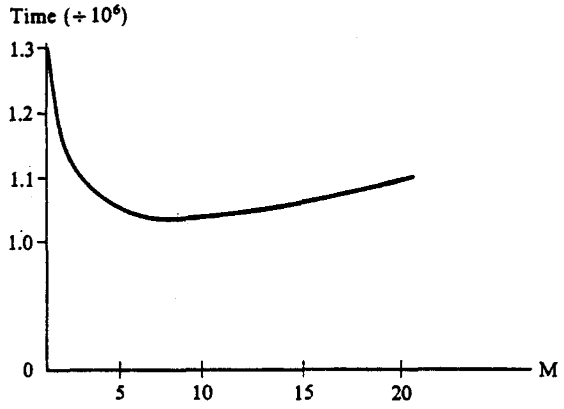

Fig. 2b. Total running time of Quicksort for $N = 10{,}000$

From this exact formula, we can evaluate the effect of using insertion sort for small files, and also choose the best value of the parameter $M$. Figure 2a shows the values of the function

$$f(M) = \frac{1}{6}\left(8M + 71 - 70H_{M+2} + \frac{270}{M+2} + \frac{54}{2M+3} - 36\frac{H_{M+1}}{M+2}\right)$$

and Figure 2b shows the total running time for $N = 10{,}000$, for small values of $M$. The best value is $M = 9$. Although $M$ does not affect the "leading term" of the total running time, the proper choice of this value does have a significant effect, because in practical situations $f(M)$ is about as large as $H_{N+1}$. For example, if we take $M = 1$, which occurs in naive implementations, we have a program which is 18 % slower when $N = 1{,}000$, and 14 % slower when $N = 10{,}000$. The optimum value $M = 9$ of course depends on the particular coefficients that we have used, but Figures 2a and 2b show that the precise choice is not highly critical and any value between 5 and 20 would do about as well. Notice that although all of the quantities participate in $f(M)$ the main tradeoff is between the inner loop of insertion sort (represented by $8M$, which comes from $E_N$) and the inner loop of Quicksort (represented by $70H_{M+2}$, which comes from $B_N$ and $C_N$).

Using the apprioximation $H_N = \ln N + \gamma + \frac{1}{2N} + 0(1/N^2)$ (where $\gamma = 0.57721\ldots$ is Euler's constant), our exact formula reduces to

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

Substituting for $\gamma$ and the optimum value $f(9) = -8.476\ldots$, we get the approximate formula

$$11.6667(N+1)\ln N - 1.743N - 19 \tag{5}$$

for the average total running time of Quicksort. (We started with an exact formula, so we could of course carry out this derivation to any asymptotic accuracy desired.) The total running time is reasonably stable about this average, because it has been shown (see [17]) that the standard deviation is about $N\sqrt{7-2\pi^2/3}$, or approximately $0.648N$.

### 3.1. Median-of-Three Modification

One way to improve Quicksort is to use the median of a sample of three elements from the file as the partitioning element at each stage. This tends to produce better partitioning splits, and so reduces the average running time. Care must be taken not to disturb the partitioning process when implementing this modification. In the procedure above, if we assume that $M \geqq 3$, we can insert the statements

```
if $A[l+1] > A[r]$ then $A[l+1] :=: A[r]$ endif;
if $A[l] > A[r]$ then $A[l] :=: A[r]$ endif;
if $A[l+1] > A[l]$ then $A[l+1] :=: A[l]$ endif;
```

at the beginning (after "**if** $r-l>M$ **then**"). This makes $A[l+1] \leqq A[l] \leqq A[r]$ before partitioning, so $A[l]$ is the median of these three. Furthermore, the outcome of the first comparison in each inner loop is determined, so a slight savings could be achieved by changing the pointer initializations to "$i := l+1$; $j := r$". Otherwise partitioning proceeds as before. In practical situations it might be desirable to make the worst case unlikely by, for example, inserting the statement

$$A[(l+r) \div 2] :=: A[l+1]$$

before the **if** statements above. (This idea was suggested by Singleton [20]. To see its value, consider what happens when the programs are used to sort a file which is already in increasing or decreasing order. See [17] for a more complete discussion of the worst case of Quicksort.)

### 3.2. Analysis of the Median-of-Three Modification

The running time of the median-of-three Quicksort depends on the same six abstract quantities as did the running time of the basic algorithm, but their average values are different. The coefficients of all the quantities in the expression for the total running time are the same, except that the coefficient for $A$ increases to reflect the cost of finding the median. (For the model in [11] and [17] it rises from 24 to $53\frac{1}{3}$, on the average.) Finding the median of three elements is a relatively expensive operation to be performing on each partitioning stage, but, as we shall see, it is well worthwhile.

For the median-of-three Quicksort, the average values of the quantities all satisfy recurrences of the form

$$F_N = f_N + \sum_{1\leqq k\leqq N} \frac{(N-k)(k-1)}{\binom{N}{3}} (F_{k-1} + F_{N-k}) \qquad \text{for } N > M \geqq 3$$

since the probability that the $k$th smallest element is the partitioning element is now $(N-k)(k-1)/\binom{N}{3}$. For the insertionsort quantities we have $f_N = 0$ as before, and for the Quicksort quantities, we take $F_N = 0$ for $N \leqq M$. It will be convenient to work with a slightly different form of this recurrence:

$$\binom{N}{3} F_N = \binom{N}{3} f_N + 2 \sum_{M<k<N} (N-k-1)kF_k \qquad \text{for } N > M \geqq 3. \tag{6}$$

As before, $F_N$ denotes the average value of some quantity when a random file of $N$ elements is sorted. For the Quicksort quantities, $f_N$ represents the average contribution of the first partitioning stage, as before. For the insertionsort quantities, we define $f_N$ by the equation $\binom{N}{3} f_N = 2 \sum_{0\leqq k\leqq M} (N-k-1)kF_k$. This allows us to treat all of the quantities uniformly.

To find the solution to the recurrence (6), it is convenient to use generating functions. Multiplying both sides by $z^{N-3}$ and summing on $N$ leads to the differential equation

$$F'''(z) = f'''(z) + 12 \frac{F'(z)}{(1-z)^2}$$

where $F(z) = \sum_{N>M} F_N z^N$ and $f(z) = \sum_{N>M} f_N z^N$. This third order differential equation might appear difficult to solve, but it is actually quite manageable. Multiplying both sides by $(1-z)^3$ gives an equation where the degree equals the order of each term. Differential equations of this type can be decomposed and solved by introducing an operator which both multiplies and differentiates. In this case, the appropriate operator is

$$\theta F(z) \equiv -(1-z)F'(z).$$

In this notation our differential equation becomes

$$-\theta(\theta-1)(\theta-2)F(z) = -12\theta F(z) + (1-z)^3 f'''(z)$$

which factors to yield

$$-\theta(-2-\theta)(5-\theta)F(z) = (1-z)^3 f'''(z).$$

Therefore, by successively solving three first-order differential equations:

$$-\theta U(z) = (1-z)^3 f'''(z),$$

$$(-2-\theta)T(z) = U(z),$$

and

$$(5-\theta)F(z) = T(z),$$

we could get explicit formulas for $F(z)$ and $F_N$. However, the parameter $M$ complicates $f(z)$, and it is best to translate back to power series. If we define $U(z) = \sum_{N>M} U_N z^N$ and $T(z) = \sum_{N>M} T_N z^N$, the differential equations correspond exactly to the difference equations

$$(N+1)U_{N+1} = N U_N + 6\nabla^3 f_{N+3}\binom{N+3}{3},$$

$$(N+1)T_{N+1} = (N+2)T_N + U_N,$$

and

$$(N+1)F_{N+1} = (N-5)F_N + T_N, \qquad N > M.$$

Knuth [11] shows that we can successively solve these difference equations (for the specific values of $f_N$ which arise) to get the average values of the median-of-three Quicksort quantities (also see [17]). However, we can proceed further, and get an explicit solution for $F_N$.

The first recurrence telescopes immediately, and we find that

$$N U_N = 6\nabla^2 f_{N+2}\binom{N+2}{3}.$$

The recurrence for $T_N$ must be divided by the summation factor $(N+1)(N+2)$ before it will telescope:

$$\frac{T_N}{N+1} = \frac{T_{M+1}}{M+2} + \sum_{M+3\le k\le N+1} \frac{\nabla^2 f_k\binom{k}{3}}{\binom{k}{3}}.$$

Finally, if $N > 5$, the recurrence for $F_N$ telescopes when it is multiplied by $\frac{1}{6}\binom{N}{5}$:

$$\binom{N}{6}F_N = \binom{M+1}{6}F_{M+1} + \frac{T_{M+1}}{M+2}\sum_{M+2\le j\le N}\binom{j}{6}$$

$$+ \sum_{M+2\le j\le N}\binom{j}{6}\sum_{M+3\le k\le j}\frac{\nabla^2 f_k\binom{k}{3}}{\binom{k}{3}}.$$

The initial values $F_{M+1}$ and $T_{M+1}$ can be obtained from (6). The sum in the second term is a sum of binomial coefficients on the upper index, which evaluates simply to $\binom{N+1}{7} - \binom{M+2}{7}$. After interchanging the order of summation in the double sum, we get the same thing for the inner sum. Performing all of the these calculations leads to the solution

$$\binom{N}{6}F_N=\binom{M+1}{6}f_{M+1}+\left(\nabla f_{M+2}+\frac{6}{M+2}f_{M+1}\right)\left(\binom{N+1}{7}-\binom{M+2}{7}\right)$$
$$+\sum_{M+3\le k\le N}\frac{\nabla^2 f_k\binom{k}{3}}{\binom{k}{3}}\left(\binom{N+1}{7}-\binom{k}{7}\right)\qquad N>\max(M,5). \tag{7}$$

As with Equation (2), the calculations involved in evaluating this function can become exceedingly tedious if $f_k$ is at all complicated, and the summations can be simplified somewhat. Two applications of summation by parts lead to the remarkably simple expression

$$F_N=f_N+\frac{1}{\binom{N}{6}}\sum_{M<k<N}\frac{12f_k}{(k+2)(k+1)}\left(\binom{N+1}{7}-\binom{k+2}{7}\right)\qquad N>\max(M,5) \tag{8}$$

for the solution to the recurrence (6). The dominant term is $\frac{12}{7}(N+1)\sum_{M<k<N} f_k/(k+2)(k+1)$. This is the same sum that we encountered in the analysis of normal Quicksort, where we further simplified it with one more application of summation by parts. The result is

$$F_N=\frac{12}{7}\,\frac{N+1}{M+2}f_{M+1}-\frac{5}{7}f_N+\frac{12}{7}(N+1)\sum_{M+2\le k\le N}\frac{\nabla f_k}{k+1}$$
$$-\frac{2}{7}\frac{1}{\binom{N}{6}}\sum_{M+1\le k\le N-1}\binom{k}{5}f_k. \tag{9}$$

From these solutions to the recurrence (6) we can easily calculate the results given in [11] and [17] for the average number of partitioning stages, exchanges, comparisons, stack pushes, insertions, and moves during insertion for median-of-three Quicksort.

The average number of partitioning stages is found by taking $f_{M+1}=f_N=f_k=1$ and $\nabla f_k=0$ in (9), with the result $A_N=\frac{12}{7}\frac{N+1}{M+2}-1+O(N^{-6})$. For the average number of comparisons, we take $f_N=N+1$ and $\nabla f_k=1$, with the result $C_N=\frac{12}{7}(N+1)(H_{N+1}-H_{M+2})+\frac{37}{49}(N+1)-\frac{24}{7}\frac{N+1}{M+2}+2+O(N^{-6})$. A calculation similar to the one for normal Quicksort shows that the average number of exchanges on the first partitioning stage is $\frac{N-4}{5}$, so $B_N=\frac{1}{5}C_N-\frac{3}{5}A_N$.

The calculation for $S$ is similar. In this case we have

$$f_N=\sum_{M+2\le k\le N-M-1}(k-1)(N-k)\Big/\binom{N}{3}.$$

As before, the recurrence (6) holds for $N>2M+1$, so we replace $M$ by $2M+1$ everywhere in the solution. Since $\nabla^2\binom{k}{3}f_k=k-2$, it is most convenient to use the form (7) of the solution. Substituting the values $f_{2M+2}=0$ and $f_{2M+3}=$

$(M+1)^2 \Big/ \binom{2M+3}{3}$, the solution

$$S_N = \frac{3}{7}(N+1)\frac{5M+3}{(2M+3)(2M+1)} - 1 + O(N^{-6})$$

follows immediately.

The results for the insertionsort quantities $D$ and $E$ are also most easily calculated using the form (7) of the solution, since in this case $\nabla^2 f_k \binom{k}{3} = 0$. We therefore have

$$D_N = \frac{N+1}{7}\left(\nabla d_{M+2} + \frac{6}{M+2}\, d_{M+1}\right) + O(N^{-6})$$

and a similar formula holds for $E$. The initial values follow directly from the definition of $f_N$ for the insertionsort quantities. For example,

$$d_{M+1} = \frac{2}{\binom{M+1}{3}} \sum_{0 \le k \le M} (M-k)\, k\, (k - H_k) = M + 1 - 2D_{M+1} + \frac{2}{3}$$

and similarly $d_{M+2} = M + 2 - 2H_{M+2} + \frac{2}{3}$, so $\nabla d_{M+2} = 1 - \frac{2}{M+2}$ and

$$D_N = N + 1 - \frac{4}{7}\,\frac{N+1}{M+2}\,(3H_{M+1} - 1).$$

The calculation for $E$ follows precisely the same steps.

In summary, these calculations tell us that Program 3 requires

$A_N = \frac{12}{7}\,\frac{N+1}{M+2} - 1$ stages,

$B_N = \frac{12}{35}(N+1)(H_{N+1} - H_{M+2}) + \frac{37}{245}(N+1) - \frac{12}{7}\,\frac{N+1}{M+2} + 1$ exchanges,

$C_N = \frac{12}{7}(N+1)(H_{N+1} - H_{M+2}) + \frac{37}{49}(N+1) - \frac{24}{7}\,\frac{N+1}{M+2} + 2$ comparisons,

$D_N = (N+1) - \frac{4}{7}\,\frac{N+1}{M+2}\,(3H_{M+1} + 1)$ insertions,

$E_N = \frac{1}{35}(N+1)(6M - 17) + \frac{6}{7}\,\frac{N+1}{M+2}$ moves during insertion,

and

$S_N = \frac{3}{7}(N+1)\frac{5M+3}{(2M+3)(2M+1)} - 1$ stack pushes.

These formulas are all accurate to within $O(N^{-6})$ and they are valid for $N > \max(M, 5)$ except $S_N$, which is valid for $N > \max(2M+1, 5)$. Notice that, although the number of partitioning stages and comparisons is significantly lower than for Program 2, the other quantities, particularly the number of exchanges, are all slightly higher.

The total average running time on the typical machine that we have been considering is $53\frac{1}{2}A_N + 11B + 4C_N + 3D_N + 8E_N + 9S_N + 7N$, and substituting the expressions above we get the exact formula

$$\frac{372}{35}(N+1)H_{N+1}-\frac{111}{2}+\frac{1}{7}(N+1)\Big(\frac{48}{5}M+\frac{529}{7}-\frac{372}{5}H_{M+2}+\frac{450}{M+2}+\frac{27(5M+3)}{(2M+3)(2M+1)}-\frac{36H_{M+1}}{M+2}\Big). \tag{10}$$

Again, this holds for $N>2M+1$; for $M<N\leqq 2M+1$, we substract $9S_N$, and for $N\leqq M$ the insertionsort time holds. The best value of $M$ is again 9, and for this choice of $M$ the total average running time of Program 3 is about

$$10.6286(N+1)\ln N+2.116N-71 \tag{11}$$

as compared with $11.6667(N+1)\ln N+1.743N-19$ for the basic algorithm. For $N=1{,}000$ this is a 4 % improvement; for $N=10{,}000$ it is about 5 %; as $N$ gets very large it approaches 8.8 %, but for practical values of $N$ the improvement is limited to about 6 %.

## 3.3. Larger Samples

A further improvement in the performance might be expected if we were to use the median of a larger sample as the partitioning element. To analyze this case where the method is extended to employ samples of $2t+1$ elements, it is necessary to solve recurrences of the form

$$F_N=f_N+\sum_{1\leqq k\leqq N}\frac{\binom{N-k}{t}\binom{k-1}{t}}{\binom{N}{2t+1}}(F_{k-1}+F_{N-k}) \qquad \text{for } N>M\geqq 2t+1.$$

If $f_N=N+O(1)$ then methods similar to those above can be used to show that the number of comparisons is given by

$$F_N=\frac{1}{H_{2t+2}-H_{t+1}}(N+1)H_{N+1}+O(N).$$

(This result was first obtained by van Emden [21]; also see [11] and [17]. Exact formulas, even for $f_N=1$, have not been derived for $t>1$.) As $t$ gets large, this approaches the theoretic minimum $N\log_2 N+O(N)$ (see [14]), but very slowly. The average number of exchanges for the first partitioning stage comes out to be $\frac{t+1}{4t+6}N+O(1)$. Therefore, for the machine model that we have been using, the total running time of a median-of-$(2t+1)$ Quicksort (when $t=O(1)$) will be

$$\frac{4+11\left(\frac{t+1}{4t+6}\right)}{H_{2t+2}-H_{t+1}}N\ln N+O(N).$$

The coefficient evaluates to about 11.67, 10.66, 10.31, 10.12, for $t=0, 1, 2, 3$ and approaches (very slowly) the value $27/4(\ln 2)\simeq 9.73$. Most of the improvement occurs for $t=1$, and larger samples will probably not be wortwhile. The cost of finding the median, which is hidden in the $O(N)$ terms above, will cancel out the small improvement in the leading term.

### 4.1. Loop Unwrapping

The performance of assembly language implementations of Quicksort can sometimes be improved with a technique called loop unwrapping. Unlike the median-of-three modification, which should be used in all implementations of Quicksort, this improvement is useful only when the sorting routine is to be used often or for a very large file, i.e. when it is worthwhile to implement it in assembly language. We consider it here for three reasons. First, it illustrates how a minor change to the implementation of an algorithm can have a major impact on the analysis (and it demonstrates the utility of the general formulas (3) and (9) derived above). Second, Quicksort implemented in this way might be characterized as the fastest known sorting algorithm, so it merits careful study. Third, the general characteristics of loop unwrapping are not well understood, and this particular example illustrates the complexities that it can introduce and the limits of its usefulness.

One many computers, the inner loops

```
**loop** $i := i + 1$ **while** $A[i] < v$ **repeat**;
**loop** $j := j - 1$ **while** $A[j] > v$ **repeat**;
```

can be implemented with only three instructions each. For example, the first loop above might consist of the code:

```
INC I, 1        Increment register i by 1.
CMP V, A(I)     Compare v with A[i].
JG  *-2         Go back two instructions if v > A[i].
```

(Here the mnemonics $I$, $V$ are symbolic register names, and the notation $A(I)$ denotes the memory location whose address is at $A$ plus the contents of $I$, or $A[i]$.) One iteration of these instructions involves four memory references; hence the coefficient of $C$ is 4 in our expressions for the total running time of the programs. Loop unwrapping is a technique for reducing the pointer arithmetic overhead by making two copies of the loop, one for $A[i]$ and one for $A[i+1]$. The code

```
        CMP  V, A+1(I)
        JLE  OUT1
LOOP    INC  I, 2
        CMP  V, A(I)
        JLE  OUT
        CMP  V, A+1(I)
        JG   LOOP
OUT1    INC  I, 1
OUT     ⋮
```

is exactly equivalent to the code above, but the $I$ pointer is incremented only about half as often. More precisely, there is a savings of $\left\lfloor \frac{s}{2} \right\rfloor$ time units each time the loop is iterated $s$ times. The $j$ loop can obviously be unwrapped in the same way.

Loop unwrapping must be carefully implemented to be effective (see [5, 11, 19]). Although the method above always reduces the number of instructions required no matter how many times the loop is iterated, loop unwrapping may not be desirable for some computers and applications more sophisticated than the simple model considered here. For example, a few computers have "compare and skip" and "increment and jump" instructions which allow the inner loops to be implemented in two instructions and obviate the need for loop unwrapping. Other computers have "instruction buffers" which make it undesirable to enlarge the inner loops. If multiword items are being sorted, the improvement due to loop unwrapping will be less significant. Nevertheless, the technique is of practical interest for many sorting applications and for most computers, and we shall now study its effectiveness.

## 4.2. Analysis of Loop Unwrapping

This improvement affects only a small section of the programs, so we will analyze the total average savings due to loop unwrapping and then simply subtract the result from the formulas that we have already derived for the total average running time of the programs. This analysis is more complicated than that we have seen so far, because the savings on the first partitioning stage depends heavily on the distribution of keys in the subfiles. For example, suppose that the file

$$7\quad 1\quad 8\quad 2\quad 9\quad 3\quad 10\quad 4\quad 5\quad 6$$

is to be partitioned in the normal way. The result, after $8:=:6$, $9:=:5$, $10:=:4$, and then $7:=:4$, is

$$4\quad 1\quad 6\quad 2\quad 5\quad 3\quad 7\quad 10\quad 9\quad 8.$$

Now suppose that the inner loops are unwrapped. Each loop is executed four times: the number of iterations is 2, 2, 2, 1 for the $i$ loop and 1, 1, 1, 1 for the $j$ loop. Therefore the total savings is 3. But now consider the file

$$7\quad 1\quad 6\quad 2\quad 8\quad 9\quad 10\quad 4\quad 3\quad 5$$

which is also partitioned, after $8:=:5$, $9:=:3$, $10:=:4$, and then $7:=:4$, into

$$4\quad 1\quad 6\quad 2\quad 5\quad 3\quad 7\quad 10\quad 9\quad 8.$$

Again, each loop is executed four times, iterating 4, 1, 1, 1 times for $i$ and 1, 1, 1, 1 times for $j$, and the savings is only 2. The numbers of exchanges, comparisons, partitioning stages, etc. for these two files are identical, since none of these quantities depend on the distribution of the keys in the subfiles, but the savings due to loop unwrapping differ. This effect makes the average saving on the first partitioning stage difficult to derive and it complicates the calculation of the total savings.

First, we shall deal with the analysis of the application of loop unwrapping to the Quicksort algorithm. In order to capture in the analysis the complications described above, we must determine exactly what factors contribute to the savings. Suppose that the partitioning element is the $k$th smallest in the file, and consider the left subfile. The savings in $i$ movements is determined by how the keys among $A[2], \ldots, A[k]$ which are less than the partitioning element are organized. There are $t$ such keys with probability $\binom{k-1}{t}\binom{N-k}{k-1-t}\Big/\binom{N-1}{k-1}$, as we saw when we calculated $B_N$. And any time a "run" of $s$ such keys occurs consecutively, we have a savings of $\left\lfloor \frac{s+1}{2} \right\rfloor = \left\lceil \frac{s}{2} \right\rceil$. Removing the condition on $k$, $t$ and $s$, we find that the average total savings in the left subfile on the first partitioning stage is

$$\frac{1}{N} \sum_{1 \le k \le N} \sum_{1 \le t < k} \sum_{1 \le s \le t} \frac{\binom{k-1}{t}\binom{N-k}{k-1-t}}{\binom{N-1}{k-1}} p_{kts} \left\lceil \frac{s}{2} \right\rceil$$

where $p_{kts}$ is the probability that, when $A[1]$ is the $k$th smallest element, a "run" of length $s$ occurs in the $t$ keys among $A[2], \ldots, A[k]$ which are $< A[1]$. It is slightly more convenient to work with the quantity $\binom{k-1}{t} p_{kts}$, which we shall denote by $P_{kts}$. By our randomness assumptions we can make the more abstract interpretation that $P_{kts}$ is the number of runs of $s$ consecutive zeros which occur in all $k-1$ digit binary numbers with $t$ zeros and $k-1-t$ ones (there are $\binom{k-1}{t}$ such numbers). Furthermore, an argument completely symmetric to the above holds for the right subfile, with exactly the same result. Also, since the program can be characterized in this way even when the loops are unwrapped more than once, we shall make this derivation more general by writing $g(s)$ rather than $\left\lceil \frac{s}{2} \right\rceil$ for the savings when the inner loops are iterated $(s+1)$ times. Although implementation of multiple unwrapping is difficult, it is clear by extending the technique shown above that the savings is no more than $g(s) = (p-1) \lfloor s/p \rfloor + s \bmod p$ when the loop is unwrapped $p$ times, or

$$g(s) = \frac{1}{p} \left((p-1)\, s + s \bmod p\right).$$

The total average savings due to loop unwrapping in Program 2 may therefore be described by the expression

$$\frac{2}{N} \sum_{1 \le k \le N} \sum_{1 \le t < k} \sum_{1 \le s \le t} \frac{\binom{N-k}{k-1-t}}{\binom{N-1}{k-1}} P_{kts}\, g(s). \tag{12}$$

To calculate $P_{kts}$, we can derive a recurrence relation by setting up a correspondence between $k-1$ digit binary strings with $t$ zeros and certain $k-2$ digit binary strings. (The reader might find it useful to refer to Figure 3, which shows the strings and non-zero values of $P_{kts}$ for $2 \le k \le 5$. Notice that $P_{kts} = P_{(k-s+1)\,(t-s+1)\,1}$.) Specifically, all $k-1$ digit binary numbers with $t$ zeros can be formed by (i) appending a 1 at the right of all $k-2$ digit numbers with $t$ zeros and (ii) appending a 0 at the right of all $k-2$ digit numbers with $t-1$ zeros. Now, exactly $\binom{k-s-3}{t-s-1}$ of the latter $k-2$ digit strings ended with exactly $s$ zeros, and $\binom{k-s-2}{t-s}$ of them ended with exactly $s-1$ zeros. All other runs of consecutive zeros are unaffected.

|  | $t$ |  |  |  |  |  |  |  |  |  |
|---|---|---|---|---|---|---|---|---|---|---|
|  | 0 | 1 | 2 | 3 | 4 | 5 |  |  |  |  |
| $k=2$ | 1 | 0 |  |  |  |  | $P_{211}=1$ |  |  |  |
| $k=3$ | 11 | 01 | 00 |  |  |  | $P_{311}=2$ |  |  |  |
|  |  | 10 |  |  |  |  |  | $P_{322}=1$ |  |  |
| $k=4$ | 111 | 110 | 001 | 000 |  |  | $P_{411}=3$ |  |  |  |
|  |  | 101 | 010 |  |  |  | $P_{421}=2$ | $P_{422}=2$ |  |  |
|  |  | 011 | 100 |  |  |  |  |  | $P_{433}=1$ |  |
| $k=5$ | 1111 | 1110 | 1100 | 0001 | 0000 |  | $P_{511}=4$ |  |  |  |
|  |  | 1101 | 1010 | 0010 |  |  | $P_{521}=6$ | $P_{522}=3$ |  |  |
|  |  | 1011 | 0110 | 0100 |  |  | $P_{531}=2$ | $P_{532}=2$ | $P_{533}=2$ |  |
|  |  | 0111 | 1001 | 1000 |  |  |  |  |  | $P_{544}=1$ |
|  |  |  | 0101 |  |  |  |  |  |  |  |
|  |  |  | 0011 |  |  |  |  |  |  |  |

Fig. 3. Counting runs in binary strings

Therefore, we have the recurrence

$$P_{kts}=P_{(k-1)ts}+P_{(k-1)(t-1)s}-\binom{k-s-3}{t-s-1}+\binom{k-s-2}{t-s}$$

or

$$P_{kts}=P_{(k-1)ts}+P_{(k-1)(t-1)s}+\binom{k-s-3}{t-s}.$$

This recurrence may be solved by transforming it into a recurrence on the generating function $P_{ks}(z)=\sum_t P_{kts}z^t$, which telescopes on $k$. The solution is

$$P_{kts}=(k-t)\binom{k-s-2}{t-s} \tag{13}$$

which holds for all $1\leqq s\leqq t<k$.

Substituting, we have a still formidable-looking triple sum describing the total average savings due to loop unwrapping in Program 2:

$$\frac{2}{N}\sum_{1\leqq s\leqq t<k\leqq N}\frac{(k-t)\binom{N-k}{k-1-t}\binom{k-s-2}{t-s}g(s)}{\binom{N-1}{k-1}}.$$

This sum can be evaluated by summing first on $t$, then on $k$, and leaving the sum on $s$ until last. The sums turn out to be simple convolutions, but we have to rearrange the binominal coefficients, and pay careful attention to the limits of summation. First,

$$\begin{aligned}(k-t)\binom{N-k}{k-1-t}&=(k-t-1)\binom{N-k}{k-1-t}+\binom{N-k}{k-1-t}\\&=(N-k)\binom{N-k-1}{k-2-t}+\binom{N-k}{k-1-t},\end{aligned}$$

so we have

$$\begin{aligned}\frac{2}{N}\sum_{1\leqq s<N}g(s)\sum_{s<k\leqq N}\frac{1}{\binom{N-1}{k-1}}\Bigg((N-k)&\sum_{s\leqq t<k}\binom{N-k-1}{k-2-t}\binom{k-s-2}{t-s}\\&+\sum_{s\leqq t<k}\binom{N-k}{k-1-t}\binom{k-s-2}{t-s}\Bigg).\end{aligned}$$

The inner sums are both instances of Vandermonde's convolution. The first evaluates to $\binom{N-s-3}{k-s-2}$, and the second to $\binom{N-s-2}{k-s-1}$, which leaves

$$\frac{2}{N}\sum_{1\le s<N} g(s) \sum_{s<k\le N} \frac{(N-k)\binom{N-s-3}{k-s-3}}{\binom{N-1}{k-1}} + \frac{2}{N}\sum_{1\le s<N} g(s) \sum_{s<k\le N} \frac{\binom{N-s-2}{k-s-1}}{\binom{N-1}{k-1}}.$$

Eliminating zero terms and separating out the last term (for $s=N-1$) in the second sum, this is

$$\frac{2}{N}\sum_{1\le s\le N-3} g(s) \sum_{s+2\le k\le N-1} \frac{(N-k)\binom{N-s-3}{k-s-2}}{\binom{N-1}{k-1}}$$
$$+\frac{2}{N}\sum_{1\le s\le N-2} g(s) \sum_{s+1\le k\le N-1} \frac{\binom{N-s-2}{k-s-1}}{\binom{N-1}{k-1}} + 2\frac{g(N-1)}{N}.$$

Now, by manipulating the binominal coefficients, we find that

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

and, similarly,

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

Therefore, the total savings is

$$\frac{2}{3}(N+1)\sum_{1\le s\le N-3}\frac{g(s)}{\binom{s+4}{3}} - \sum_{1\le s\le N-3}\frac{g(s)}{\binom{s+3}{2}} + \sum_{1\le s\le N-2}\frac{g(s)}{\binom{s+2}{2}} + 2\frac{g(N-1)}{N},$$

or, changing the index of summation from $s$ to $s-1$ in the first two sums and noting that $g(0)=0$, we have the result

$$\frac{2}{3}(N+1)\sum_{1<s\le N-2}\frac{g(s-1)}{\binom{s+3}{3}} + \sum_{1\le s\le N-2}\frac{\nabla g(s)}{\binom{s+2}{2}} + 2\frac{g(N-1)}{N}.$$

This formula can be simplified further with summation by parts. In general, for any function $a(s)$ and any integer $t\ge 2$, we have the identity

$$\sum_{m\le s\le n}\frac{a(s)}{\binom{s+t}{t}} = \sum_{m\le s\le n}\frac{t(t-1)}{(s+t)\,(s+1)}\,\frac{a(s)}{\binom{s+t-1}{t-2}}$$
$$= \frac{t}{t-1}\left(\sum_{m\le s\le n+1}\frac{\nabla a(s)}{\binom{s+t-1}{t-1}} + \frac{a(m-1)}{\binom{m+t-1}{t-1}} - \frac{a(n+1)}{\binom{n+t}{t-1}}\right). \tag{14}$$

Therefore, in particular,

$$\sum_{1\le s\le N-2} \frac{\nabla g(s)}{\binom{s+2}{2}} = 2 \sum_{1\le s\le N-1} \frac{\nabla^2 g(s)}{s+1} - 2\frac{\nabla g(N-1)}{N}.$$

$\Bigl($Notice that $g(-1) = -\frac{1}{p}(-p+1+(-1 \bmod p)) = 0$, so $\nabla g(0) = 0.\Bigr)$ Also, applying (14) twice,

$$\sum_{1\le s\le N-2} \frac{g(s-1)}{\binom{s+3}{3}} = 3 \sum_{2\le s\le N} \frac{\nabla^2 g(s-1)}{s+1} - 3\frac{\nabla g(N-1)}{N+1} - \frac{3}{2}\frac{g(N-2)}{\binom{N+1}{2}}.$$

Substituting these last sums into our result and collecting terms, we find that the average savings on the first partitioning stage due to the application of loop unwrapping to Quicksort is

$$2(N+1)\sum_{2\le s\le N} \frac{\nabla^2 g(s-1)}{s+1} + 2\sum_{1\le s\le N} \frac{\nabla^2 g(s)}{s+1} - 2\nabla g(N-1). \tag{15}$$

In the case that the loop is unrolled only once, this formula assumes a particularly simple form. For this case, we know that $g(s) = \left\lceil \frac{s}{2} \right\rceil = \frac{1}{2}(s + s \bmod 2)$, so $\nabla g(s) = \frac{1}{2}(1 + s \bmod 2 - (s-1) \bmod 2) = \frac{1}{2}(1 - (-1)^s)$, and $\nabla^2 g(s) = (-1)^{s+1}$. Therefore, from (15) the average savings on the first partitioning stage due to unwrapping the loop once is

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

This can be more concisely expressed in terms of the sum of the alternating harmonic series, $\sum_{1\le s\le N} (-1)^s/s$, which we shall denote by $\bar{H}_N$. In this notation, we have

$$-2N\bar{H}_N - N + (-1)^N \tag{16}$$

for the average savings on the first partitioning stage due to unwrapping the loop once.

We shall be encountering a number of other variations on the harmonic numbers in the analysis below. Before continuing with the analysis, it will be convenient to summarize the various notations. The four sequences that arise in the study of loop unwrapping are

$$H_n = \sum_{1\le k\le n} \frac{1}{k} = \ln n + \gamma + \frac{1}{2n} - \frac{1}{12n^2} + O\left(\frac{1}{n^4}\right),$$

$$\bar{H}_n = \sum_{1\le k\le n} \frac{(-1)^k}{k} = -\ln 2 + O\left(\frac{1}{n}\right),$$

$$H_n^{(2)} = \sum_{1\le k\le n} \frac{1}{k^2} = \frac{\pi^2}{6} + O\left(\frac{1}{n}\right),$$

$$H_n^* = \sum_{1\le k\le n} \frac{(-1)^k H_k}{k} = \frac{1}{2}(\ln 2)^2 - \frac{\pi^2}{12} + O\left(\frac{\log n}{n}\right).$$

The asymptotic estimates for $H_n$ and $H_n^{(2)}$ are well-known (see [1, 11]); the estimate for $\bar{H}_n$ is elementary; and the reader may find the estimate for $H_n^*$ an interesting exercise.

Returning to the analysis, we have yet to determine the total average savings due to unwrapping the loop once in Quicksort—we have only dealt with the first partitioning stage. But finding the total savings is simply a matter of substituting our formula for the average savings on the first partitioning stage, $-2N\bar{H}_N - N + (-1)^N$, into the general formula (3) that we derived for the average values of all the quantities upon which the running time of Quicksort depends. In (3), we have $f_N = -2N\bar{H}_N - N + (-1)^N$, $f_{M+1} = F_{M+1} = -2(M+1)\bar{H}_{M+1} - (M+1) - (-1)^M$, and $\nabla f_k = -2\bar{H}_{k-1} - 1$, so we find that the average total savings is

$$-2(N+1)\sum_{M+2\leq k\leq N}\frac{2\bar{H}_{k-1}+1}{k+1} - 2\frac{N+1}{M+2}\left(2(M+1)\bar{H}_{M+1}+(M+1)+(-1)^M\right) + 2N\bar{H}_N + N - (-1)^N.$$

To evaluate this sum, the main step is to note that

$$\sum_{1\leq k\leq N}\frac{\bar{H}_{k-1}}{k} = H_N\bar{H}_N - H_N^*.$$

After evaluating the sum and clearing up the other terms, we end up with an exact formula for the total average savings due to unwrapping the inner loops of Quicksort once:

$$\begin{aligned}\Big(&-4\bar{H}_{N+1} - 2)(N+1)H_{N+1} + (N+1)(4H_{N+1}^* + 10\bar{H}_{N+1} - 1 + 4\bar{H}_{M+2}\bar{H}_{M+2}\\ &+ 2H_{M+2} - 4H_{M+2}^* - 12\bar{H}_{M+2} + \frac{4\bar{H}_{M+1}+2+6(-1)^M}{M+2}\Big) - 2\bar{H}_N - 1 + 5(-1)^N.\end{aligned}\tag{17}$$

This formula is admittedly a bit unwieldy, but it is an exact result. We can subtract this formula from our formula (4) for the total running time to get a formula for the total running time of Quicksort with the inner loops unwrapped once. From that expression, it turns out that the best choice of $M$ is still $M = 9$. Collecting the terms involving $M$ into a function $g(M)$ and applying the asymptotic expressions given above for the variants on the harmonic series, we find that the asymptotic expression for our total savings is

$$(4\ln 2 - 2)N\ln N + \left(4\gamma\ln 2 - 2\gamma + 2(\ln 2)^2 - \frac{\pi^2}{3} - 10\ln 2 - 1 + g(M)\right)N + O(\log N)$$

or, since $g(9) \simeq 8.243$, about

$$0.7726N\ln N - 1.571N$$

and the total average running time of the basic Quicksort program with the loops unwrapped is approximately

$$10.8941\,N\ln N - 0.172N. \tag{18}$$

Therefore, this programming technique achieves a 5 % improvement in the total running time of the program for all but small values of $N$.

### 4.3. Multiple Unwrapping

Having completed this analysis, the next question that we must answer is whether or not it is worthwhile to unwrap the loop further. As remarked above, it is much more difficult to unwrap the loop more than once, so in most situations we probably wouldn't be interested in doing so unless the savings were significant. In the limiting situation (if we unwrap the loop enough times), the pointer increments can theoretically be moved out of the "comparison" loops (which are iterated at total of $2N\ln N+O(N)$ times) and into the "exchange" loop (which is iterated $\frac{1}{3}N\ln N+O(N)$ times). The savings is $\frac{4}{3}N\ln N+O(N)$, nearly twice the savings that we get by unwrapping once. We cannot hope to achieve this limit because (for example) space might be a constraint, but we must investigate how close we can get to the limit by unwrapping the loop two, three, four, ... times.

Most of the analysis is not at all different from what we have already done, and as indicated above some of the results that we have derived were kept in a general form so that we could more easily study the effects of multiple unwrapping. Specifically, we can use Equation (15) to find the average savings on the first partitioning stage, with $g(s)=\frac{1}{p+1}(ps+s\bmod(p+1))$, if the loop is unwrapped $p$ times. To use (15), we need to calculate the first and second differences of the function $g(s)$. It is easily verified that, for $p>1$,

$$\nabla g(s)=\begin{cases}0 & s \bmod p=0\\ 1 & \text{otherwise}\end{cases} \quad\text{and}\quad \nabla^2 g(s)=\begin{cases}-1 & s\bmod p=0\\ 1 & s \bmod p=1\\ 0 & \text{otherwise.}\end{cases}$$

Now, substituting this into (15), we find that the total savings on the first partitioning stage, when the inner loop is unwrapped $(p-1)$ times, is

$$2N\Bigg(\sum_{\substack{2\le s\le N-1\\(s-1)\bmod p=1}}\frac{1}{s+1}-\sum_{\substack{2\le s\le N-1\\(s-1)\bmod p=0}}\frac{1}{s+1}\Bigg)+O(1).$$

Here we are dealing only with the "leading term". We could proceed as above and get complete exact answers, but it will not be necessary to do so. Now, we can also include the "tails" of the sums in the $O(1)$ term, and we can rewrite the sums to get the expression

$$2N\left(\frac{1}{3}+\sum_{s\ge 1}\left(\frac{1}{ps+3}-\frac{1}{ps+2}\right)\right)+O(1).$$

These remaining sums are yet another variant of the harmonic numbers which were discovered by Gauss and have been studied by Knuth in conjunction with the analysis of Euclid's algorithm [9, 10]. Knuth defines an extension of the harmonic numbers to cover non-integral indices:

$$H_x=\sum_{n\ge 1}\left(\frac{1}{n}-\frac{1}{n+x}\right)\qquad\text{for any } x.$$

If $x$ is an integer, this agress with the standard definition. But now take $x=3/p$. Then we have

$$H_{3/p}=\sum_{n\ge 1}\left(\frac{1}{n}-\frac{1}{n+\dfrac{3}{p}}\right)=p\sum_{n\ge 1}\left(\frac{1}{pn}-\frac{1}{pn+3}\right).$$

Similarly, $H_{2/p}=p\sum_{n\geq 1}\left(\frac{1}{pn}-\frac{1}{pn+2}\right)$, so, in this notation, our expression for the savings on the first partitioning stage becomes

$$\frac{2N}{p}\left(\frac{p}{3}+H_{2/p}-H_{3/p}\right)+O(1).$$

Therefore, from (3) the total average savings due to unwrapping the inner loops of Quicksort $(p-1)$ times is

$$\frac{4}{p}\left(\frac{p}{3}+H_{2/p}-H_{3/p}\right)N\ln N+O(N). \tag{19}$$

This surprisingly simple formula allows us to accurately estimate the benefits of multiple unwrapping. Knuth shows that when $x$ is a rational number less than one, $H_x$ can be explicitly evaluated (see [9]):

$$H_{p/q}=\frac{q}{p}-\frac{1}{2}\pi\cot\frac{p}{q}\pi-\ln 2q+2\sum_{1\leq n<\frac{q}{2}}\cos\frac{2\pi np}{q}\ln\sin\frac{n}{q}\pi,$$

and he tabulates some small values. Table 1 shows the formulas for $H_{2/p}$ and $H_{3/p}$ for $2\leq p\leq 6$, along with formulas and values for the coefficient of $N\ln N$ in the total savings due to loop unwrapping. In the line for $p=5$, $\phi$ denotes the "golden ratio", $\frac{1}{2}(\sqrt{5}+1)=1.61803\ldots$, and $x$ denotes the quantity $-\frac{1}{2}(2+\phi)\ln 5+(\phi-\frac{1}{2})\ln(2+\phi)$, which cancels between the two terms. Now, the coefficient of $N\ln N$ in the original program is $11.66667\ldots$, and as we have seen, the improvement in this coefficient due to unwrapping the loop once is about 6.6%. Unwrapping the loop a second time will yield an additional 2.2% improvement in the leading term; a third will yield an additional 1.0%; a fourth an additional 0.5%; and a fifth an additional 0.4%. The savings approaches the limit of 4/3 very slowly, and most of the improvement is gained by unwrapping the loop just once.

Table 1. Savings for multiple unwrapping

| $p$ | $H_{2/p}$ | $H_{3/p}$ | $\frac{4}{p}\left(\frac{p}{3}+H_{2/p}-H_{3/p}\right)$ | $\simeq$ |
|---|---|---|---|---|
| 2 | 1 | $\frac{8}{3}-2\ln 2$ | $4\ln 2-2$ | 0.77259 |
| 3 | $\frac{3}{2}+\frac{\pi}{2\sqrt{3}}-\frac{3}{2}\ln 3$ | 1 | $2-2\ln 3+\frac{2\pi}{3\sqrt{3}}$ | 1.01197 |
| 4 | $2-2\ln 2$ | $\frac{4}{3}+\frac{\pi}{2}-3\ln 2$ | $2-\frac{\pi}{2}+\ln 2$ | 1.12235 |
| 5 | $\frac{5}{2}-\frac{1}{2}\frac{\pi}{\phi\sqrt{2+\phi}}+x$ | $\frac{5}{3}+\frac{1}{2}\frac{\pi}{\phi\sqrt{2+\phi}}+x$ | $2-\frac{4}{5}\frac{\pi}{\phi\sqrt{2+\phi}}$ | 1.18339 |
| 6 | $3-\frac{\pi}{2\sqrt{3}}-\frac{3}{2}\ln 3$ | $2-2\ln 2$ | $2-\frac{\pi}{3\sqrt{3}}+\frac{4}{3}\ln 2-\ln 3$ | 1.22098 |
| $\infty$ | 0 | 0 | $\frac{4}{3}$ | 1.33333 |

## 5. Median-of-Three with Loops Unwrapped

These results lead us to expect that the most effective sorting program that we can devise will be the result of applying *both* sampling and loop unwrapping once, that is, unwrapping the inner loops of median-of-three Quicksort once. But we have yet to analyze precisely the effect of this. Sampling complicates the analysis of the savings due to loop unwrapping just as it complicated the analysis of all the other quantities upon which the running time of the program depends. However, the analysis is very similar to many of the derivations that we have already done, so we shall often refer to the analysis above for details.

First, we must determine the average savings on the first partitioning stage. Using the notation of (12) above, and arguing exactly as we did in Section 4.2, we find that the savings is given by the expression

$$2 \sum_{2\leq k\leq N-1} \sum_{1\leq t<k} \sum_{1\leq s\leq t} \frac{(k-1)(N-k)}{\binom{N}{3}} \frac{\binom{N-k-1}{k-2-t}}{\binom{N-3}{k-2}} P_{(k-1)ts}\, g(s). \qquad (20)$$

The difference between this expression and (12) is that the probability that the *kth* smallest element is used for partitioning is $(k-1)(N-k)\Big/\binom{N}{3}$, and then only $N-3$ elements ($k-2$ of which are smaller than the partitioning element) participate in the partitioning process. Now, $(k-1)(N-k)\Big/\binom{N}{3}\binom{N-3}{k-2}$ simplifies immediately to $6/N\binom{N-1}{k-1}$, and we know that $P_{(k-1)ts}=(k-t-1)\binom{k-s-3}{t-s}$ from (13), so we have to evaluate

$$\frac{12}{N} \sum_{1\leq s\leq t<k\leq N-1} \frac{(k-t-1)\binom{N-k-1}{k-2-t}\binom{k-s-3}{t-s} g(s)}{\binom{N-1}{k-1}}$$

in order to find the average savings on the first partitioning stage due to unwrapping the inner loops of the median-of-three Quicksort.

To evaluate this triple sum, we use precisely the same procedure as above. First, we write $(k-t-1)\binom{N-k-1}{k-2-t}=(N-k-1)\binom{N-k-2}{k-3-t}+\binom{N-k-1}{k-2-t}$ and apply Vandermonde's convolution to the sums on $t$. Then we manipulate the binomial coefficients to move those involving $k$ to the numerator, so that the sums on $k$ can be easily evaluated. (As before, this must be done carefully—zero terms must be eliminated in the sums to aviod binomial coefficients with negative upper indices.) Performing these manipulations, we have a result analogous to (15), before we applied summation by parts:

$$3(N+1) \sum_{3\leq s\leq N-3} \frac{g(s-2)}{\binom{s+4}{4}} - 8 \sum_{3\leq s\leq N-3} \frac{g(s-2)}{\binom{s+3}{3}} + 4 \sum_{2\leq s\leq N-3} \frac{g(s-1)}{\binom{s+3}{3}} + \frac{12g(N-3)}{N(N-1)}.$$

And, as we did to get (15), we can further simplify this formula by applying summation by parts, formula (14). Applying the formula three times to the first sum and twice to the next two, we get the final expression

$$12(N+1)\sum_{3\le s\le N}\frac{\nabla^3 g(s-2)}{s+1}-24\sum_{3\le s\le N-1}\frac{\nabla^2 g(s-2)}{s+1}+12\sum_{2\le s\le N-1}\frac{\nabla^2 g(s-1)}{s+1}-(N+1)\left(\frac{12\nabla^2 g(N-2)}{N}+3\right) \tag{21}$$

for the average savings on the first partitioning stage.

We are most interested in the case where the inner loops are unwrapped once. For this case, we have seen that $\nabla^2 g(s)=(-1)^{s+1}$, and so $\nabla^3 g(s)=2(-1)^{s+1}$. Using these values in (21), and expressing the result in terms of $\bar{H}_N$, we get the answer

$$24N\bar{H}_N+17N-12\bar{H}_{N-1}-12(-1)^N-9 \tag{22}$$

for the average savings on the first stage when the inner loops of the median-of-three Quicksort are unwrapped once. Notice that for large $N$ this is about $(24\bar{H}_N+17)N\simeq 0.3644N$, as compared with the $(-1-2\bar{H}_N)N\simeq 0.3863N$ savings that we achieved with Quicksort.

Finally, just as we did for the basic algorithm, we can substitute our formula for the average savings on the first stage, (22), into the general formula (9) that we found for the totals of our quantities. To make this substitution we first find from (22) that $\nabla f_k=24\bar{H}_{k-1}+17-\frac{12(-1)^{k-1}}{k-1}$. After the substitution, the first sum in (9) is easily evaluated as above. The second sum introduces a number of new terms, but can be evaluated exactly with the use of the identities $\sum_{0\le k\le n}\binom{k}{t}=\binom{n+1}{t+1}$ and $\sum_{0\le k\le n}(-1)^k\binom{k}{t}=-\frac{1}{2}\sum_{0\le k\le n}(-1)^k\binom{k}{t-1}+\frac{1}{2}(-1)^n\binom{n+1}{t}$. For brevity, we shall not write down the very small terms and be content to derive the answer correct to within $O(1)$. The entire second sum is then simply $-\frac{12}{49}(N+1)(24\bar{H}_N+17)+O(1)$. After evaluating the other sum and collecting terms, we have the result

$$\begin{aligned}\frac{12}{7}(24\bar{H}_{N+1}+17)(N+1)H_{N+1}&+\frac{12}{7}(N+1)(-24H^*_{N+1}-\frac{430}{7}\bar{H}_{N+1}\\&+\frac{629}{84}-24H_{M+2}\bar{H}_{M+2}-17H_{M+2}+24H^*_{M+2}+72\bar{H}_{M+2}\\&+\frac{1}{M+2}\left(-36\bar{H}_{M+1}-36(-1)^M-6\frac{(-1)^M}{M+1}-26)\right)+O(1).\end{aligned} \tag{23}$$

Subtracting this "exact" formula from (10), we get the total average running time of Program 3 with the loops unwrapped once. From that expression, it again turns out that the best choice of $M$ is still $M=9$. Collecting terms involving $M$ into a function $h(M)$, we get the asymptotic expression

$$\begin{aligned}&\tfrac{12}{7}(17-24\ln 2)N\ln N\\&\quad+\tfrac{12}{7}(17\gamma-24\gamma\ln 2-12(\ln 2)^2+2\pi^2+\tfrac{430}{7}\ln 2+\tfrac{629}{84}+h(M))N+O(\log N)\end{aligned}$$

for the total savings due to loop unwrapping in the median-of-three Quicksort, or, since $h(9) \simeq -65.08$, about

$$0.6248 N \ln N - 1.414 N$$

and the total average running time of the median-of-three Quicksort with the loops unwrapped once is approximately

$$10.0038 \ln N - 3.530 N. \tag{24}$$

The time gained by loop unwrapping is about 4 % of the total running time of the program for $N = 10{,}000$.

We wouldn't expect multiple unwrapping to be any more effective for the median-of-three method than for normal Quicksort. This suspicion can be confirmed by an analysis similar to that preceding (19). From (21), we find that the savings due to unwrapping the inner loops of the median-of-three method $(p-1)$ times is (for $p \geqq 3$)

$$\frac{144}{7p}\left(\frac{p}{20} + H_{3/p} - 2H_{4/p} + H_{5/p}\right) N \ln N + O(N).$$

For $p = 3$, 4, 5 the coefficient evaluates to 0.8133, 0.8971, 0.9415 as compared with the coefficient 0.6248 just derived for single unwrapping. The savings approaches the limit of $\frac{36}{35} N \ln N + O(N)$ as $p$ gets large.

## 6. Conclusion

The total improvement from a good implementation of Quicksort to an "optimized" implementation, the median-of-three method with inner loops unwrapped once, can be found by comparing (5),

$$11.6667(N+1)\ln N - 1.743 N - 19$$

with (24),

$$10.0038 N \ln N + 3.530 N,$$

the equations for the total running time. The improvement is about 10 % for files in the practical size range. If a sorting program is to be used extensively, or for very large files, as Quicksort often is, the slight extra effort required to implement the median-of-three modification and unwrap the inner loop will be well worthwhile.

We have dealt with efficient versions of the programs which are based upon several ideas introduced in [17]. Complete justification for the particular methods used in the context of the countless variations which have been proposed may be found in [17]. The median-of-three modification and loop unwrapping illustrate how the practical utility of improvements to the algorithm and its implementation can be demonstrated through exact analysis. Normally, one analyzes a program before attempting to improve it, for the analysis tells where the improvements can do the most good. The programs presented here are the product of many iterations of implementation and analysis.

Naturally, the implementation of a sorting algorithm for a particular application can involve many issues not considered here. For example, we have ignored the problem of translating the algorithm from the high-level language descriptions given to the efficient machine-language implementations assumed in the analysis. This and other practical issues are treated in detail in [17] and [19]. However, the analyses given in this paper make it possible to derive exact formulas for the average running time of particular implementations of Quicksort on real computers. Many implementations differ only in the coefficients to be used for the number of partitioning stages, comparisons, exchanges, stack pushes, insertions, and moves during insertion, and the results given here apply directly. Other implementations may involve new quantities, but these can be analyzed using the general solutions (3) or (9) to the Quicksort recurrences. In addition, the general solutions may prove useful studying the effect of future modifications to Quicksort.

Many readers may find the analysis extremely complicated, and question whether it is worthwhile to so relentlessly pursue the exact answers to our problems. Part of the reason for this is that readers are unfamiliar with the finite difference calculus, techniques such as summation by parts, etc., which are the basis for many of the manipulations. This kind of mathematics arises naturally in the analysis of algorithms and it must be mastered if we are to understand how programs perform. The analysis of the basic Quicksort algorithm is actually quite simple. For this program it is easier to derive the exact answer than it is to estimate it. (See, for example, [2] for a typical "approximate" derivation of the number of comparisons used by Quicksort, and compare with the derivation of (2).) The more advanced programs require more advanced analysis, and the mathematics involved becomes interesting in itself.

A prime attraction of the study of Quicksort is that it teaches us so much. Proper implementation of the algorithm involves a variety of techniques in program optimization such as special handling of small cases and loop unwrapping. Proper analysis of the algorithm involves a variety of techniques in algorithmic analysis such as summation by parts and generating functions. This combination of algorithm and analysis makes the study of Quicksort very fruitful. It is important not only as a useful algorithm in practical sorting applications, but also as a showcase for the analysis of algorithms.

*Acknowledgements.* Many of the derivations in this paper require painstaking attention to detail, and the referees must be thanked for their careful checking of the various formulas. One of the referees wishes to acknowledge the help of MACSYMA, a computer system whose development was supported by the Advanced Research Projects Agency of the U. S. Government.

## References

1. Abramowitz, M., Stegun, I. A.: Handbook of mathematical functions. New York: Dover Publications 1970
2. Aho, A. V., Hopcroft, J. E., Ullman, J. D.: The design and analysis of computer algorithms. Reading (Mass.): Addison-Wesley 1974
3. Boothroyd, J.: Sort of a section of the elements of an array by determining the rank of each element (Algorithm 25); Ordering the subscripts of an array section according to the magnitudes of the elements (Algorithm 26). Computer J. **10**, 308–310. [See also notes by R. S. Scowen in Computer J. **12**, 408–409 (1969) and by A. D. Woodall in Computer J. 13 (1970)]
4. Brawn, B. S., Gustavson, F. G., Mankin, E.: Sorting in a paging environment. Comm. ACM **13**, 483–494 (1970)

5. Cocke, J., Schwartz, J. T.: Programming languages and their compilers. Preliminary notes. Courant Inst. of Math. Sciences, New York University (1970)
6. Frazer, W. D., McKellar, A. C.: Samplesort: a sampling approach to minimal storage tree sorting. J. ACM **17**, 496–507 (1970)
7. Hoare, C. A. R.: Partition (Algorithm 63); Quicksort (Algorithm 64); Find (Algorithm 65). Comm. ACM **4**, 321–322 (1961). [See also certification by J. S. Hillmore in Comm. ACM **5**, 439 (1962) and by B. Randell and L. J. Russell in Comm. ACM **6**, 446 (1963)]
8. Hoare, C. A. R.: Quicksort. Computer J. **5**, 10–15 (1962)
9. Knuth, D. E.: The art of computer programming 1: Fundamental algorithms. Reading (Mass.): Addison-Wesley 1968
10. Knuth, D. E.: The art of computer programming 2: Seminumerical algorithms. Reading (Mass.): Addison-Wesley 1969
11. Knuth, D. E.: The art of computer programming 3: Sorting and searching. Reading (Mass.): Addison-Wesley 1972
12. Knuth, D. E.: Structured programming with go to statements. Computing Surveys **6**, 261–301 (1974)
13. Loeser, R.: Some performance tests of "quicksort" and descendants. Comm. ACM **17**, 143–152 (1974)
14. Morris, R.: Some theorems on sorting. SIAM J. Appl. Math. **17**, 1–6 (1969)
15. Rich, R. P.: Internal sorting methods illustrated with PL/I programs. Englewood Cliffs (N.J.): Prentice-Hall 1972
16. Scowen, R. S.: Quickersort (Algorithm 271). Comm. ACM **8**, 669–670 (1965). (See also certification by C. R. Blair in Comm. ACM **9**, 354 (1966))
17. Sedgewick, R.: Quicksort. Stanford University, Stanford Computer Science Report STAN-CS-75-492, Ph. D. Thesis, May 1975
18. Sedgewick, R.: Quicksort with equal keys. SIAM J. Computing (to appear)
19. Sedgewick, R.: Implementing Quicksort programs (to appear)
20. Singleton, R. C.: An efficient algorithm for sorting with minimal storage (Algorithm 347). Comm. ACM **12**, 185–187 (1969). [See also remarks by R. Griffin and K. A. Redish in Comm. ACM **13**, 54 (1970) and by R. Peto in Comm. ACM **13**, 624 (1970)]
21. van Emden, M. N.: Increasing the efficiency of quicksort (Algorithm 402). Comm. ACM **13**, 693–694 (1970). [See also the article by the same name in Comm. ACM **13**, 563–567 (1970)]
22. Wirth, N.: Algorithms + Data Structures = Programs. Englewood Cliffs (N. J.): Prentice-Hall 1976

Prof. Robert Sedgewick
Program in Computer Science
Division of Applied Mathematics
Brown University
Providence, Rhode Island 02912, USA