Striving for Efficiency in Algorithms: Sorting

Inna Pivkina*

Sorting is the fundamental algorithmic problem in computer science. It is the first step in solving many other algorithmic problems. Donald Knuth, a world famous computer scientist and author of the book "The Art of Computer Programming, Volume 3: Sorting and Searching" ([6]), wrote: "I believe that virtually every important aspect of programming arises somewhere in the context of searching or sorting".

Quicksort is a comparison sorting algorithm that, on average, makes $O(n \log n)$ comparisons to sort n items. This is as efficient as a comparison sorting algorithm can be ([1]). Quicksort is often faster in practice than other $O(n \log n)$ sorting algorithms and it has another advantage - it sorts in place, that is, the items are rearranged within the array, so it does not require a lot of additional space ([1]).

Quicksort was invented by a British computer scientist, C.A.R. Hoare, in 1960. Sir Charles Antony Richard Hoare describes how he invented Quicksort in his interview published in [10]. After graduating from the University of Oxford in 1956, Hoare did his national service in the Royal Navy studying Russian. In 1958 he took a course in Mercury Autocode, which was the programming language used on a computer in Oxford University. Later, he was a visiting student at Moscow State University in the Soviet Union for a year. That is when he developed the Quicksort algorithm. The following is a quote from the interview with Hoare ([10]), where he describes his invention of Quicksort:

"The National Physical Laboratory was starting a project for the automatic translation of Russian into English, and they offered me a job. I met several of the people in Moscow who were working on machine translation, and I wrote my first published article, in Russian, in a journal called Machine Translation.

In those days the dictionary in which you had to look up in order to translate from Russian to English was stored on a long magnetic tape in alphabetical order. Therefore it paid to sort the words of the sentence into the same alphabetical order before consulting the dictionary, so that you could look up all the words in the sentence on a single pass of the magnetic tape.

I thought with my knowledge of Mercury Autocode, I'll be able to think up how I would conduct this preliminary sort. After a few moments I thought of the obvious algorithm, which is now called bubble sort, and rejected that because it was obviously rather slow. I thought of Quicksort as the second thing. It didn't occur to me that this was anything very difficult. It was all an interesting exercise in programming. I think Quicksort is the only really interesting algorithm that I ever developed."

Hoare described the algorithm in his papers in 1961 and 1962 ([2], [3], [4], [5]). After its invention by Hoare, Quicksort has undergone extensive analysis by Robert Sedgewick in 1975, 1977, 1978 ([7], [8], [9]). Sedgewick in his paper "Implementing Quicksort programs" ([9]) presented "a practical study of how to implement the Quicksort sorting algorithm and its best variants on real computers". The paper contains the original version of Quicksort and presents step-by-step modifications to the algorithm which, as Sedgewick says, make its implementation on real computers more efficient.

*Department of Computer Science; New Mexico State University; Las Cruces, NM 88003; `ipivkina@cs.nmsu.edu`.

The main idea of the project is to experimentally verify whether results of Sedgewick in [9] are still true nowadays by implementing the algorithms and comparing their running times. The paper is attached at the end of the project.

The project is divided into several parts. Each part contains a reading assignment and a list of tasks.

1 Project Part 1

Read the paper until section "Worst Case" on page 850 (read sections Introduction, The Algorithm, Improvements, Removing Recursion, Small Subfiles).

Exercise 1.1. Program 1 in the paper describes original version of Quicksort. The operators and the constructs which are used in Program 1 are different from those which are used in the textbook (Cormen et al. [1]). For instance, operator :=: and the control structures loop ... repeat. Rewrite Program 1 using pseudocode conventions from the textbook.

Exercise 1.2. Partitioning algorithm which is used in Program 1 is different from the version of PARTITION given in the textbook. Write a pseudocode for procedure Partition1 which performs partitioning using the partitioning algorithm from Program1. Use pseudocode conventions from the textbook.

Exercise 1.3. Demonstrate the operation of Partition1 from exercise 1.2 on the array A=$\langle 11, 12, 4, 8, 15, 9, 7, 1, 6, 16\rangle$. Show the values of the array and auxiliary values (values of i and j) after each step. (You may use Figure 1 on page 849 of the paper as an example of showing the values of the array.)

Exercise 1.4. Demonstrate the operation of Program 1 (quicksorting) on the array A from the above question. You may use Figure 2 on page 849 of the paper as an example.

Exercise 1.5. Demonstrate the operation of Partition1 from exercise 1.2 on the array consisting of equal numbers A=$\langle 3, 3, 3, 3, 3, 3\rangle$. Show the values of the array and auxiliary values (values of i and j) after each step. (You may use Figure 1 on page 849 of the paper as an example of showing the values of the array.) Notice that this example fits the description of the partitioning process from the last paragraph of the first column on page 848 of the paper which starts with words "If equal keys are present among A[1],...,A[N]...".

Exercise 1.6. Give an example of array A such that:

- A has 5 elements, and
- exactly 2 out of these 5 elements are equal, and
- using Partition1 on A fits the description of the partitioning process from the last paragraph of the first column on page 848 of the paper which starts with words "If equal keys are present among A[1],...,A[N]...".

Demonstrate the operation Partition 1 on this array A.

Exercise 1.7. On page 848 Sedgewick writes: "For example, rather than using the "sentinel" A[N+1] = ∞ we could use

loop: i:= i+1; while i$\leq$ N and A[i]$<$ ν repeat;

for the i pointer increment, but this would be far less efficient." Explain why this would be far less efficient.

Exercise 1.8. What strategy does Sedgewick adopt on the question of how keys equal to the partitioning element should be treated?

Exercise 1.9. On page 849 Sedgewick says: "For example, if the file A[1],...,A[N] is already in order, then the program will invoke itself to recursive depth N". Verify the statement by drawing tree of recursive calls for execution of Program 1 on already sorted array A[1],...,A[N] (assume that all the elements being sorted are distinct).

Exercise 1.10. Rewrite procedure *insertionsort* on pages 849-850 of the paper using pseudocode conventions from the textbook. Notice that it is different from the insertion sort implementation in the textbook.

Exercise 1.11. Demonstrate the procedure *insertionsort* from exercise 1.10 on the array A=$\langle 11, 12, 4, 8, 15, 9, 7, 1, 6, 16\rangle$. Show the contents of the array after each execution of the outer loop.

Exercise 1.12. On page 850 Sedgewick writes: "The obvious way to improve Program 1 is to change the first if statement to

if r-l$\leq$M then insertionsort(l,r) else ...".

Why this is an improvement?

Exercise 1.13. What is an even better way to modify Program 1 than the one in question 1.12?

Exercise 1.14. What is the best value of the length of unsorted subfiles?

2 Project Part 2

Read the paper until section "Assembly Language" on page 852 (read sections Worst Case, Median-of-three Modification, Implementation).

Exercise 2.1. What method did Hoare suggest to make the worst case unlikely to occur in practice? Explain in your own words why this method works.

Exercise 2.2. How did Sedgewick suggest to modify Program 1 in order to implement the method from exercise 2.1? When describing the modification please use pseudocode conventions from the textbook.

Exercise 2.3. Describe in your own words the idea of median-of-three modification. What is the purpose of this modification?

Exercise 2.4. What implementation of median-of-three modification does Sedgewick present in the paper? When describing the implementation please use pseudocode conventions from the textbook.

Exercise 2.5. Rewrite Program 2 using pseudocode conventions from the textbook.

Exercise 2.6. Explain why condition A[N+1] = ∞ is needed in Program 2. What will happen if the condition is not there?

Exercise 2.7. Demonstrate the operation of Program 2 upon the digits of constant e, that is, A=$\langle 2, 7, 1, 8, 2, 8, 1, 8, 2, 8, 4, 5, 9, 0, 4, 5\rangle$. Use M=4. You may use Figure 6 on page 852 of the paper as an example. (This exercise is closely related to the next exercise 2.8.)

Exercise 2.8. What are the values of A_N , B_N, C_N, S_N, D_N, E_N in the execution of Program 2 on array A from exercise 2.7? (Give exact numbers.)

3 Project Part 3

Exercise 3.1. Implement Program 1 from the paper. Assume that the elements to be sorted are positive integers. Given a filename, the program should read integers to be sorted from the file, sort them, and write sorted integers to an output file. Your program should also compute how much time was spent on sorting. The computed time should not include time spent on reading and writing data to/from files.

Exercise 3.2. Implement Program 2 from the paper. Assume that the elements to be sorted are positive integers. Given a filename, the program should read integers to be sorted from the file, sort them, and write sorted integers to an output file. Your program should also compute how much time was spent on sorting. The computed time should not include time spent on reading and writing data to/from files.

Exercise 3.3. Write a separate program (let us call it Generator) which will create a file containing a specified number of random integers within certain range. The inputs to this program should include the number of integers you want to be in the file and the lower and upper bounds for the integers. For example, it should be able to create a file with 1000 integers between 100 (inclusive) and 999 (inclusive).

Exercise 3.4. Generate 10 different input files using your Generator program. Each file should have ten thousand integers in the range from 1 to 10000.

Exercise 3.5. Sort each of the input files generated in exercise 3.4 using program 1. Record sorting times. What are the average, minimum and maximum sorting times?

Exercise 3.6. Sort each of the input files generated in exercise 3.4 using Program 2 with values of M equal to 3, 6, 9, 10, 14, 20. Record sorting time in each run. What are the average, minimum and maximum sorting times for each of these values of M?

Exercise 3.7. Summarize your results in a table like the following:

Sorting time	Program 1	Program 2					
		M=3	M=6	M=9	M=10	M=14	M=20
Average							
Minimum							
Maximum							

Answer the following questions.

(a) Is sorting time for Program 2 always better than sorting time for Program 1 in your experiments?

(b) What value of M in your experiments gave the best sorting time? Is that value of M the same as the best value of M in Sedgewick's paper?

(c) Is there a big difference between average, minimum and maximum sorting times in your experiments?

Notes to the instructor

The project is designed for a junior level Data Structures and Algorithms course. It is based on the paper by R. Sedgewick "Implementing Quicksort Programs". In the paper Sedgewick presents "a practical study of how to implement the Quicksort sorting algorithm and its best variants on real computers". The paper contains the original version of Quicksort and its modification, which as Sedgewick says combines the most effective improvements to Quicksort. The main idea of the project is to experimentally verify results of Sedgewick by implementing the algorithms and comparing their running times.

The project allows students to learn and practice Quicksort, insertion sort, recursive thinking, using implicit stack data structure to remove recursion, computing running times of algorithms, etc. The project is divided into three parts. Each part contains a reading assignment and a list of tasks.

References

[1] T. Cormen, C. Leiserson, R. Rivest, and C. Stein. *Introduction to Algorithms*. MIT Press, 2 edition, 2002.

[2] C. A. R. Hoare. Algorithm 63: Partition. *Communications of the ACM*, 4(7):321, 1961.

[3] C. A. R. Hoare. Algorithm 64: Quicksort. *Communications of The ACM*, 4(7):321, 1961.

[4] C. A. R. Hoare. Algorithm 65: Find. *Communications of The ACM*, 4(7):321–322, 1961.

[5] C. A. R. Hoare. Quicksort. *Computer Journal*, 5(1):10–15, 1962.

[6] D. Knuth. *The Art of Computer Programming: Sorting and Searching*, volume 3. Addison-Wesley, 2 edition, 1998.

[7] R. Sedgewick. *Quicksort*. PhD thesis, Stanford University, Stanford, CA, May 1975. Stanford Computer Science Report STAN-CS-75-492.

[8] R. Sedgewick. The analysis of quicksort programs. *Acta Informatica*, 7:327–355, 1977.

[9] R. Sedgewick. Implementing quicksort programs. *Communications of The ACM*, 21:847–857, 1978.

[10] L. Shustek. Interview: An interview with C.A.R. Hoare. *Communications of The ACM*, 52(3):38–41, March 2009.

Programming Techniques

S. L. Graham, R. L. Rivest Editors

Implementing Quicksort Programs

Robert Sedgewick
Brown University

This paper is a practical study of how to implement the Quicksort sorting algorithm and its best variants on real computers, including how to apply various code optimization techniques. A detailed implementation combining the most effective improvements to Quicksort is given, along with a discussion of how to implement it in assembly language. Analytic results describing the performance of the programs are summarized. A variety of special situations are considered from a practical standpoint to illustrate Quicksort's wide applicability as an internal sorting method which requires negligible extra storage.

Key Words and Phrases: Quicksort, analysis of algorithms, code optimization, sorting

CR Categories: 4.0, 4.6, 5.25, 5.31, 5.5

Introduction

One of the most widely studied practical problems in computer science is sorting: the use of a computer to put files in order. A person wishing to use a computer to sort is faced with the problem of determining which of the many available algorithms is best suited for his purpose. This task is becoming less difficult than it once was for three reasons. First, sorting is an area in which the mathematical analysis of algorithms has been particularly successful: we can predict the performance of many sorting methods and compare them intelligently. Second, we have a great deal of experience using sorting algorithms, and we can learn from that experience to separate good algorithms from bad ones. Third, if the file fits into the memory of the computer, there is one algorithm, called Quicksort, which has been shown to perform well in a variety of situations. Not only is this algorithm simpler than many other sorting algorithms, but empirical [2, 11, 13, 21] and analytic [9] studies show that Quicksort can be expected to be up to twice as fast as its nearest competitors. The method is simple enough to be learned by programmers who have no previous experience with sorting, and those who do know other sorting methods should also find it profitable to learn about Quicksort.

Because of its prominence, it is appropriate to study how Quicksort might be improved. This subject has received considerable attention (see, for example, [1, 4, 11, 13, 14, 18, 20]), but few real improvements have been suggested beyond those described by C.A.R. Hoare, the inventor of Quicksort, in his original papers [5, 6]. Hoare also showed how to analyze Quicksort and predict its running time. The analysis has since been extended to the improvements that he suggested, and used to indicate how they may best be implemented [9, 15, 17]. The subject of the careful implementation of Quicksort has not been studied as widely as global improvements to the algorithm, but the savings to be realized are as significant. The history of Quicksort is quite complex, and [15] contains a full survey of the many variants which have been proposed.

The purpose of this paper is to describe in detail how Quicksort can best be implemented to handle actual applications on real computers. A general description of the algorithm is followed by descriptions of the most effective improvements that have been proposed (as demonstrated in [15]). Next, an implementation of Quicksort in a typical high level language is presented, and assembly language implementation issues are considered. This discussion should easily translate to real languages on real machines. Finally, a number of special issues are considered which may be of importance in particular sorting applications.

This paper is intended to be a self-contained overview of the properties of Quicksort for use by those who need to actually implement and use the algorithm. A companion paper [17] provides the analytical results which support much of the discussion presented here.

The Algorithm

Quicksort is a recursive method for sorting an array $A[1], A[2], \ldots, A[N]$ by first "partitioning" it so that the following conditions hold:

(i) Some key v is in its final position in the array. (If it is the jth smallest, it is in position $A[j]$.)

(ii) All elements to the left of $A[j]$ are less than or equal to it. (These elements $A[1], A[2], \ldots, A[j-1]$ are called the "left subfile.")

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the ACM copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Association for Computing Machinery. To copy otherwise, or to republish, requires a fee and/or specific permission.

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

Author's address: Division of Applied Mathematics and Computer Science Program, Brown University, Providence, RI 02912.

© 1978 ACM 0001-0782/78/1000-0847 $00.75

(iii) All elements to the right of $A[j]$ are greater than or equal to it. (These elements $A[j + 1], \ldots, A[N]$ are called the "right subfile.")

After partitioning, the original problem of sorting the entire array is reduced to the problem of sorting the left and right subfiles independently. The following program is a recursive implementation of this method, with the partitioning process spelled out explicitly.

Program 1

```
procedure quicksort (integer value l, r);
  comment Sort A[l : r] where A[r + 1] ≥ A[l], ..., A[r];
  if r > l then
    i := l; j := r + 1; v := A[l];
    loop:
      loop: i := i + 1; while A[i] < v repeat;
      loop: j := j − 1; while A[j] > v repeat;
    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 (or swap) operator :=:, and the control constructs **loop** ... **repeat** and **if** ... **endif**, which are like those described by D.E. Knuth in [10]. Statements between **loop** and **repeat** are iterated: when the **while** condition fails (or the **until** condition is satisfied) the loop is exited immediately. The keyword **repeat** may be thought of as meaning "execute the code starting at **loop** again," and, for example, "**until** $j < i$" may be read as "**if** $j < i$ **then** leave the loop".)

The partitioning process may be most easily understood by first assuming that the keys $A[1], \ldots, A[N]$ are distinct. The program starts by taking the leftmost element 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 it is known that $A[l + 1], \ldots, A[j]$ are $< v$ and $A[j + 1], \ldots, A[r]$ are $> v$, so that the exchange $A[l]$:=: $A[j]$ completes the job of partitioning $A[l], \ldots, A[r]$. The condition that $A[r + 1]$ must be greater than or equal to all of the keys $A[l], \ldots, A[r]$ is included to stop the i pointer in the case that v is the largest of the keys. The procedure call quicksort (1, N) will therefore sort $A[1], \ldots, A[N]$ if $A[N + 1]$ is initialized to some value at least as large as the other keys. (This is normally specified by the notation $A[N + 1] := \infty$.)

If equal keys are present among $A[1], \ldots, A[N]$, then Program 1 still operates properly and efficiently, but not exactly as described above. If some key equal to v is already in position in the file, then the pointer scans could both stop with $i = j$, so that, after one more time through the loop, it terminates with $j + 2 = i$. But at this point it is known not only that $A[l + 1], \ldots, A[j]$ are $\leq v$ and $A[j + 2], \ldots, A[r]$ are $\geq v$ but also that $A[j + 1] = v$. After the exchange $A[l]$:=: $A[j]$, we have *two* elements in their final place in the array ($A[j]$ and $A[j + 1]$), and the subfiles are recursively sorted.

Figures 1 and 2 show the operation of Program 1 on the first 16 digits of π. In Figure 1, elements marked by arrows are those pointed to by i and j, and each line is the result of a pointer increment or an exchange. In Figure 2, each line is the result of one "partitioning stage," and boldface elements are those put into position by partitioning.

The differences between the implementation of partitioning given in Program 1 and the many other partitioning methods which have been proposed are subtle, but they can have a significant effect on the performance of Quicksort. The issues involved are treated fully in [15]. By using this particular method, we have already begun to "optimize" Quicksort, for it has three main advantages over alternative methods.

First, as we shall see in much more detail later, the inner loops are efficiently coded. Most of the running time of the program is spent executing the statements

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

each of which can be implemented in machine language with a pointer increment, a compare, and a conditional branch. More naive implementations of partitioning include other tests, for the pointers crossing or exceeding the array bounds, within these loops. For example, rather than using the "sentinel" $A[N + 1] = \infty$ we could use

```
loop: i := i + 1; while i ≤ N and A[i] < v repeat;
```

for the i pointer increment, but this would be far less efficient.

Second, when equal keys are present, there is the question of how keys equal to the partitioning element should be treated. It might seem better to scan over such keys (by using the conditions $A[i] \leq v$ and $A[j] \geq v$ in the scanning loops), but careful analysis shows that it is always better to stop the scanning pointers on keys equal to the partitioning element, as in Program 1. (This idea was suggested in 1969 by R.C. Singleton [18].) In this paper, we will adopt this strategy for all of our programs, but in the analysis we will assume that all of the keys being sorted are distinct. Justification for doing so may be found in [16], where the subject of Quicksort with equal keys is studied in considerable detail.

Third, the partitioning method used in Program 1 does not impose a bias upon the subfiles. That is, if we start with a random arrangement of $A[1], \ldots, A[N]$, then, after partitioning, the left subfile is a random arrangement of its elements and the right subfile is a random permutation of its elements. This fact is crucial to the analysis of the program, and it also seems to be a requirement for efficient operation. It is conceivable that a method could be devised which imparts a favorable bias to the subfiles, but the creation of nonrandom subfiles is usually done inadvertently. No method which

Fig. 1. Partitioning π (Program 1).

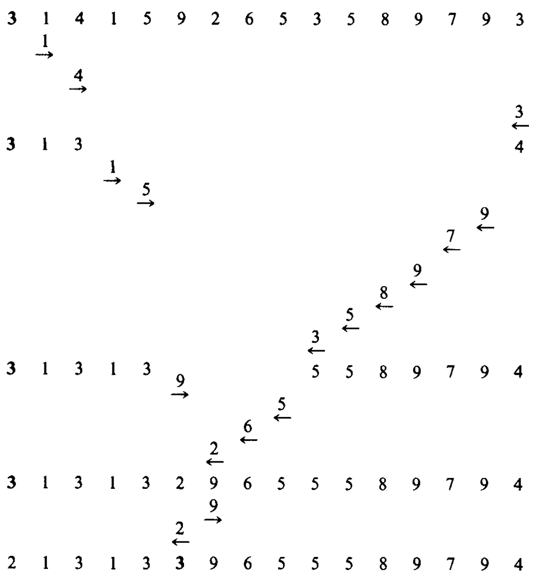


Fig. 2. Quicksorting π (Program 1).

3	1	4	1	5	9	2	6	5	3	5	8	9	7	9	3
2	1	3	1	3	**3**	9	6	5	5	5	8	9	7	9	4
1	1	**2**	3	3											
1	**1**														
			3	**3**											
						7	6	5	5	5	8	4	**9**	**9**	9
						4	6	5	5	5	**7**	8			
						4	6	5	5	5					
							5	5	5	**6**					
							5	**5**	5						
							5								
									5						
												8			
															9
1	1	2	3	3	3	4	5	5	5	6	7	8	9	9	9

produces nonrandom subfiles has yet been successfully analyzed, but empirical tests show that such methods slow down Quicksort by up to 20 percent (see [10, 15]).

Improvements

Program 1 is, then, an easily understandable description of an efficient sorting algorithm. It can be a perfectly acceptable sorting program in many practical situations. However, if efficiency is a primary concern, there are a number of ways in which the program can be improved. This will be the subject of the remainder of this paper. Each improvement requires some effort to implement, which it rewards with a corresponding increase in efficiency. To illustrate the effectiveness of the various modifications, we shall make use of the analytic results given in [17], where exact formulas are derived for the total average running time of realistic implementations of different versions of Quicksort on a typical computer.

Removing Recursion

The most serious problem with Program 1 is that it can consume unacceptable amounts of space in the implicit stack needed for the recursion. For example, if the file $A[1], \dots, A[N]$ is already in order, then the program will invoke itself to recursive depth N, and it will thus implicitly require extra storage proportional to N. Hoare pointed out that this is easily corrected by changing the recursive calls so that the shorter of the two subfiles is sorted first. The recursive depth is then limited to $\log_2 N$ [6]. Care must be exercised in implementing this change, since many compilers will not recognize that the second recursive call is not really recursive. Whenever a procedure ends with a call on another procedure, the stack space used for the first call may be reclaimed before the second call is made (see [10]). Rather than expose ourselves to the whims of compilers we will remove the recursion and use an explicit stack. This will also eliminate some overhead, and it is a straightforward transformation on Program 1.

When implemented in assembly language with recursion removed in this way, the expected running time of Program 1 is shown in [17] to be about $11.6667N \ln N + 12.312N$ time units. The "time unit" used is the time required for one memory reference (i.e. count one for each instruction, plus one more if the instruction references data in memory). The model is similar to Knuth's MIX [7]—we shall see it in more detail below when we examine assembly language implementation. The formulas derived in [17] are exact, but rather complicated: the simple formula above is accurate to within 0.1 percent for $N > 1000$, 1 percent for $N > 100$, and 2 percent for $N > 20$. Similar formulas with this accuracy are derived in [17] for all the improvements described below, and these are quite sufficient for comparing the methods and predicting their performance.

Small Subfiles

Another major difficulty with Program 1 is that it simply is not very efficient for small subfiles. This is especially unfortunate because the recursive nature of the program guarantees that it will always be used for many small subfiles. Therefore Hoare suggested that a more efficient method be used when $r - l$ is small [6]. A method which is known to be very efficient for small files is insertion sorting. This is the method of scanning through the file and inserting each element into place among those previously considered, by successively moving smaller elements up to make room. It may be implemented as follows:

```
procedure insertionsort(l, r);
  comment Sort A[l : r] where A[r + 1] ≥ A[l], ... ,   A[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;
```

(Just as there are many different implementations of Quicksort, so there are a variety of ways to implement Insertionsort. This subject is treated in detail in [9] and [15].) Now, the obvious way to improve Program 1 is to change the first **if** statement to

if $r - l \leq M$ **then** *insertionsort(l, r)* **else** ...

where M is some threshold value above which Quicksort is faster than Insertionsort.

It is shown in [15] that there is an even better way to proceed. Suppose that small subfiles are simply ignored during partitioning, e.g. by changing the first **if** statement in Program 1 to "**if** $r - l > M$ **then** ... ." Then, after the entire file has been partitioned, it has all the elements which were used as partitioning elements in place, with unsorted subfiles of length M or less between them. A single Insertionsort of the entire file will quite efficiently complete the job of sorting the file.

Analysis shows that it takes Insertionsort only slightly longer to sort the whole file than it would to sort all of the subfiles, but all of the overhead of invoking Insertionsort during partitioning is eliminated. For example, subfiles with M or fewer elements never need be put on the stack, since they are ignored during partitioning. It turns out that this eliminates $\frac{3}{4}$ of the stack pushes used, on the average. This makes the method preferable to the scheme of sorting the small subfiles during partitioning (even in an "optimal" manner).

For most implementations, the best value of M is about 9 or 10, though the exact value is not highly critical: Any value between 6 and 15 would do about as well. Figure 3 shows the total running time on the machine in [17] for $N = 10{,}000$ for various values of M. The best value is $M = 9$, and the total running time for this value is about $11.6667N \ln N - 1.743N$ time units. Figure 4 is a graph of the function $14.055N/(11.6667N \ln N + 12.312N)$, which shows the percentage improvement for this optimum choice $M = 9$ over the naive choice $M = 1$ (Program 1).

Fig. 3. Total running time of Quick sort for $N = 10{,}000$.

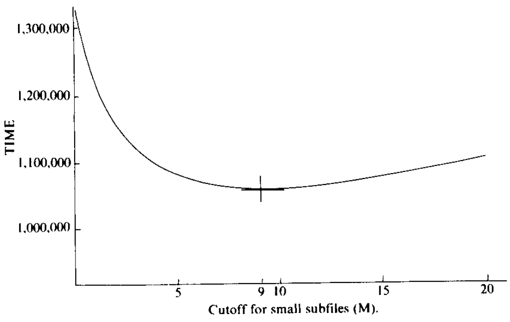
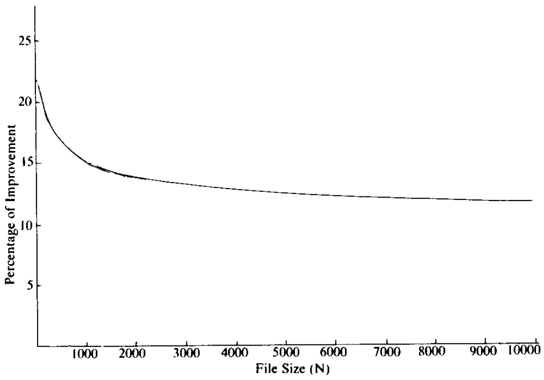


Fig. 4. Improvement due to sorting small subfiles on a separate pass.



Worst Case

A third main flaw of Program 1 is that there are some files which are likely to occur in practice for which it will perform badly. For example, suppose that the numbers $A[1]$, $A[2]$, ... , $A[N]$ are in order already when Program 2 is invoked. Then $A[1]$ will be the first partitioning element, and the first partition will produce an empty left subfile and a right subfile consisting of $A[2]$, ... , $A[N]$. Then the same thing will happen to that subfile, and so on. The program has to deal with files of size N, N-1, N-2, ... and its total running time is obviously proportional to N^2. The same problem arises with a file in reverse order. This $O(N^2)$ worst case is inherent in Quicksort: it is especially unfortunate if it occurs on files so likely to occur in practice.

There are many ways to make such anomalies very unlikely in practical situations. Rather than using the first element in the file as the partitioning element, we might try to use some other fixed element, like the middle element. This helps some, but simple anomalies still can occur. Hoare suggested a method which does work: choose a random element as the partitioning element [6]. As remarked above, care must be taken when implementing these simple changes to ensure that the partitioning method still produces random subfiles. The safest method, if $A[p]$ is to be used as the partitioning element (where, for example, p is computed to be a pseudorandom number between l and r), is to simply precede the statement $v := A[l]$ by the statement $A[p] :=: A[l]$ in Program 1.

Using a random partitioning element will virtually ensure that anomalous cases for Program 2 will not occur in practical sorting situations, but it has the disadvantage that random number generation can be relatively expensive. We are probably being overcautious to slow down the program for all files, just to avoid a few anomalies. The next method that we will examine actually improves the average performance of the program while at the same time making the worst case unlikely to occur in practice.

Median-of-Three Modification

The method is based on the observation that Quicksort performs best when the partitioning element turns out to be near the center of the file. Therefore choosing a good partitioning element is akin to estimating the median of the file. The statistically sound method for doing this is to choose a sample from the file, find the median, and use that value as the estimate for the median of the whole file. This idea was suggested by Hoare in his original paper, but he didn't pursue it because he found it "very difficult to estimate the saving." It turns out that most of the savings to be had from this idea come when samples of size three are used at each partitioning stage. Larger sample sizes give better estimates of the median, of course, but they do not improve the running time significantly. Primarily, sampling provides insurance that the partitioning elements don't consistently fall near the ends of the subfiles, and three elements are sufficient for this purpose. (See [15] and [17] for analytic results confirming these conclusions.) The average performance would be improved if we used any three elements for the sample, but to make the worst case unlikely we shall use the first, middle, and last elements as the sample, and the median of those three as the partitioning element. The use of these three particular elements was suggested by Singleton in 1969 [18]. Again, care must be taken not to disturb the partitioning process. The method can be implemented by inserting the statements

```
A[(l + r) ÷ 2] :=: A[l + 1];
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;
```

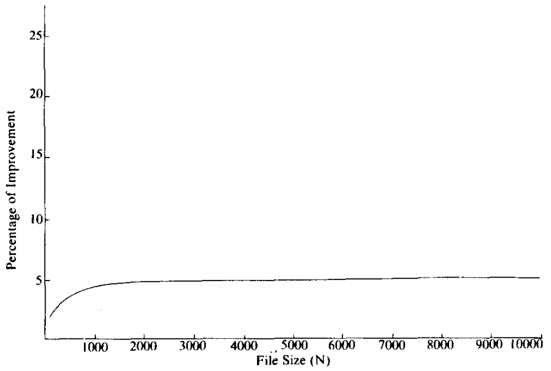
before partitioning (after "**if** $r > l$ **then**" in Program 1. This change makes $A[l]$ the median of the three elements originally at $A[l]$, $A[(l + r) \div 2]$, and $A[r]$ before partitioning. Furthermore, it makes $A[l + 1] \leq A[l]$ and $A[r] \geq A[l]$, so the pointer initializations can be changed to "$i := l + 1; j := r$". This method preserves randomness in the subfiles.

Median-of-three partitioning reduces the number of comparisons by about 14 percent, but it increases the number of exchanges slightly and requires the added overhead of finding the median at each stage. The total expected running time for the machine in [17] (with the optimum value $M = 9$) is about $10.6286N \ln N + 2.116N$ time units, and Figure 5 shows the percentage savings.

Implementation

Combining all of the improvements described above, we have Program 2, which has no recursion, which ignores small subfiles during partitioning, and which partitions according to the median-of-three modification. For clarity, the details of stack manipulation and selecting the smaller of the two subfiles are omitted. Also, since recursion is no longer involved, we will deal with an in-line program to sort $A[1], \ldots, A[N]$.

Fig. 5. Improvement due to median-of-three partitioning.



Program 2

```
integer l, r, i, j;
integer array stack[1 : 2 × f(N)];
boolean done;
arbmode array A[1 : N + 1];
arbmode v;
l := 1; r := N; done := N ≤ M;
loop until done:
  A[(l + r) ÷ 2] :=: A[l + 1];
  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;
  i := l + 1; j := r; 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 max(j − l, r − i + 1) ≤ M
then if stack empty
       then done := true
       else (l, r) := popstack
       endif;
else if min (j − l, r − i + 1) ≤ M
       then (l, r) := large subfile;
       else pushstack (large subfile);
            (l, r) := small subfile
       endif;
endif;
repeat;
A[N + 1] := ∞;
loop for N − 1 ≥ i ≥ 1:
  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;
```

In the logic for manipulating the stack after partitioning, $(l, j - 1)$ is the "large subfile" and (i, r) is the "small subfile" if $\max(j - l, r - i + 1) = j - l$, and vice versa if $r - i + 1 > j - l$. This may be implemented simply and efficiently by making one copy of the code for each of the two outcomes of comparing $j - l$ with $r - i + 1$.

Note that the condition $A[N + 1] = \infty$ is now only needed for the insertionsort. This could be eliminated, if desired, at only slight loss by changing the conditional in the inner loop of Insertionsort to "**while** $A[j] < v$ **and** $j \leq N$".

Left unspecified in Program 2 are the values of M, the threshold for small subfiles, and $f(N)$, the maximum stack depth. These are implementation parameters which should be specified as constants at compile time. As mentioned above, the best value of M for most implementations is 9 or 10, although any value from 6 to 15 will do nearly as well. (Of course, we must have $M \geq 2$, since the partitioning method needs at least three elements to find the median of.) The maximum stack depth turns out to be always less than $\log_2 (N + 1)/(M + 2)$ so (for $M = 9$) a stack with $f(N) = 20$ will handle files of up to about ten million elements. (See the analysis in [11, 15, 17].)

Figure 6 diagrams the operation of Program 2 upon the digits of π. Note that after partitioning all that is left for the insertionsort is the subfile 5 5 5 4, and the insertion sort simply scans over the other keys.

The total average running time of a program is determined by first finding analytically the average frequency of execution of each of the instructions, then multiplying by the time per instruction and summing over all instructions. It turns out that the total expected running time of Program 2 can be determined from the six quantities:

A_N the number of partitioning stages,
B_N the number of exchanges during partitioning,
C_N the number of comparisons during partitioning,
S_N the number of stack pushes (and pops),
D_N the number of insertions, and
E_N the number of keys moved during insertion.

In Program 2, C_N is the number of times $i := i + 1$ is executed plus the number of times $j := j + 1$ is executed within the scanning loops; B_N is the number of times $A[i] := A[j]$ is executed in the partitioning loop; A_N is the number of times the main loop is iterated; D_N is the number of times v is changed in the insertionsort; and E_N is the number of times $A[j - 1] := A[j]$ is executed in the insertionsort. Each instruction in an assembly language implementation can be labeled with its frequency in terms of these quantities and N. (There may be a few other quantities involved: if they do not relate simply to the main quantities or cancel out when the total running time is computed, then they generally can be analyzed in the same way as the other quantities [17].) The analysis in [17] yields exact values for these quantities, from which the total running time can be computed and the best value of M chosen. For $M = 9$ it turns out that

$$
\begin{aligned}
C_N &\simeq 1.714N \ln N - 3.74N, \\
B_N &\simeq .343N \ln N - .84N \\
E_N &\simeq 1.14N, \quad D_N \simeq .60N, \\
A_N &\simeq .16N, \quad S_N \simeq .05N.
\end{aligned}
$$

From these equations, the total running time of any particular implementation of Program 2 (with $M = 9$) can easily be estimated. For the model in [9, 15, 17], the total expected running time is $53\frac{1}{3}A_N + 11B_N + 4C_N + 3D_N + 8E_N + 9S_N + 7N$, which leads to the equation $10.6286N \ln N + 2.116N$ given above.

Fig. 6. Quicksorting π—improved method (Program 2, $M = 4$).

Quicksort:	3	1	4	1	5	9	2	6	5	3	5	8	9	7	9	3
	2	3	3	1	1	**3**	9	5	5	4	5	8	9	7	9	6
	1	1	**2**	3	3											
							5	5	5	4	**6**	8	9	7	9	9
												7	**8**	9	9	9
Insertion-	1	1	**2**	3	3	**3**	5	5	5	4	**6**	7	**8**	9	9	9
sort:										4	6	7	8	9	9	9
									4	5						
	1	1	2	3	3	3	4	5	5							
	1	1	2	3	3	3	4	5	5	5	6	7	8	9	9	9

Assembly Language

Program 2 is an extremely efficient sorting method, but it will not run efficiently on any particular computer unless it is translated into an efficient program in that computer's machine language. If large files are to be sorted or if the program is to be used often, this task should not be entrusted to any compiler. We shall now turn from methods of improving the algorithm to methods of coding the program for a machine.

Of most interest is the "inner loop" of the program, those statements whose execution frequencies are proportional to $N \ln N$. We shall therefore concern ourselves with the translation of the statements

```
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;
```

Assembly-language implementations of the rest of the programs may be found in [9] or [15]. Rather than use any particular assembly-language or deal with any particular machine, we shall use a mythical set of instructions similar to those in Knuth's MIX [7]. Only simple machine-language capabilities are used, and the programs and results that we shall derive may easily be translated to apply to most real machines.

To begin, a direct translation of the inner loop of Programs 1 and 2 is given below. The comments on each line explain what the instructions are intended to do. The mnemonics I, V, J, X, and Y are symbolic register names, and the notation $A(I)$ means the contents of the memory location whose address is A plus the contents of index register I, or $A[i]$. Readers unfamiliar with assembly language programming should consult [7].

```
LOOP 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].
     DEC  J, 1     Decrement register J by 1.
     CMP  V, A(J)  Compare v with A[j].
     JL   * - 2    Go back two instructions if v < A[j].
     CMP  J, I     Compare J with I.
     JL   OUT      Leave loop if j < i.
     LD   X, A(I)  Load A[i] into register X.
     LD   Y, A(J)  Load A[j] into register Y.
     ST   X, A(J)  Store register X into A[j].
     ST   Y, A(I)  Store register Y into A[i].
     JMP  LOOP     Unconditional jump to LOOP.
OUT
```

This direct translation of the inner loop of Programs 1 and 2 is much more efficient than the code that most compilers would produce, and there is still room for improvement.

First, no inner loop should ever end with an unconditional jump. Any such loop must contain a conditional jump somewhere, and it can always be "rotated" to end with the conditional jump, as follows:

```
     JMP  INTO
LOOP LD   X, A(I)
     LD   Y, A(J)
     ST   X, A(J)
     ST   Y, A(I)
INTO INC  I, 1
     CMP  V, A(I)
     JG   * - 2
     DEC  J, 1
     CMP  V, A(J)
     JL   * - 2
     CMP  J, I
     JGE  LOOP
OUT
```

This sequence contains exactly the same number of instructions as the above, and they are identical when executed; but the unconditional jump has been moved out of the inner loop. (If the initialization of I were changed, a further savings could be achieved by moving INTO down one instruction.) This simple change reduces the running time of the program by about 3 percent.

The coefficients 11 and 4 for B_N and C_N in the expression given above for the total running time can be verified by counting two time units for instructions which reference memory and one time unit for those which do not. It is this low amount of overhead that makes Quicksort stand out among sorting algorithms. In fact, the true "inner loop" is even tighter, because we have two loops within the inner loop here: the pointer scanning instructions

```
INC I, 1          DEC J, 1
CMP V, A(I)       CMP V, A(J)
JG  * - 2         JL  * - 2
```

are executed, on the average, three times more often than the others for Program 1. (The factor is $2\frac{1}{2}$ for Program 2.) It is hard to imagine a simpler sequence on which to base an algorithm: pointer increment, compare, and conditional jump. The fact that these loops are so small makes the proper implementation and translation of Quicksort critical. If we had a translation of **loop**: $i := i + 1$; **while** $A[i] < v$ **repeat** which used only three superfluous instructions, or if we had checked for the pointers crossing or going outside the array bounds within these loops, then the running time of the whole program could be doubled!

Loop Unwrapping

On the other hand, with our attention focused on these two pairs of three instructions, we can further improve the efficiency of the programs. The only real overhead within these inner loops is the pointer arithmetic, INC I, 1 and DEC J, 1. We shall use a technique called "loop unwrapping" (or "loop unrolling"—see [3]) which uses the addressing hardware to reduce this overhead. The idea is to make two copies of the loop, one for $A[i]$ and one for $A[i + 1]$, then increment the pointer once by 2 each time through. Of course, the code coming into and going out of the loop has to be appropriately modified.

Loop unwrapping is a well-known technique, but it is not well understood, and it will be instructive to examine its application to Quicksort in detail. The straightforward way to proceed would be to replace the instructions

```
INC  I, 1
CMP  V, A(I)
JG   * - 2
```

by one of the equivalent code sequences

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

We can measure the relative efficiency of these alterantives by considering how many memory reference they involve, assuming that the loop iterates s times. The original code uses $4s$ memory references (three for instructions, one for data). For the unwrapped program on the left above, the number of memory references taken for $s = 1, 2, 3, 4, 5, \ldots$ is 5, 8, 12, 15, 19, ... , and a general formula for the number of references saved is $\lfloor (s - 2)/2 \rfloor$. For the program on the right, the values are 4, 8, 11, 15, 18, ... and the savings are $\lfloor \frac{1}{2}(s - 1) \rfloor$. In both cases about $\frac{1}{2}s$ increments are saved, but the program on the right is slightly better.

However, both sequences contain unnecessary unconditional jumps, and both can be removed, although with quite different techniques. In the second program, the code at OUT could be duplicated and a copy substituted for JMP OUT. This technique is cumbersome if this code contains branches, and for Quicksort it even contains another loop to be unwrapped. Despite such complications, this will increase the savings to $\lfloor s/2 \rfloor$

when the loop is iterated s times. Fortunately, this same efficiency can be achieved by repairing the jump into the loop in the program on the left. The code is exactly equivalent to

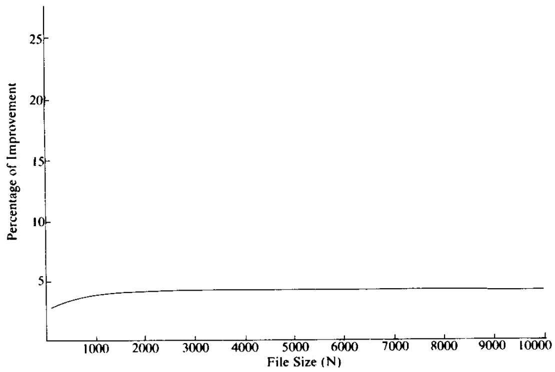
```
      CMP V, A + 1(I)
      JLE  OUT1
LOOP INC  I, 1
      CMP V, A(I)
      JLE  OUT
      CMP V, A + 1(I)
      JG   LOOP
OUT1 INC  I, 1
OUT   :
```

and this code saves $\lfloor s/2 \rfloor$ memory references over the original when the loop is iterated s times. The j loop can obviously be unwrapped in the same way, and these transformations give us a more efficient program in which the I and J pointers are altered much less often.

Note that since the inner loops of Quicksort are iterated only a few times on the average, it is very important that loop unwrapping be carefully implemented. The first implementation above is slower than the original loop if it is iterated just once, and actually increases the total running time of the program.

The analysis of the effect of loop unwrapping turns out to be much more difficult than the other variants that we have seen. The results in [17] show that unwrapping the loops of Program 2 once reduces its running time to about $10.0038N \ln N + 3.530N$, time units, and that it is not worthwhile to unwrap further. Figure 7 shows the percentage improvement when this technique is applied to Program 2.

Fig. 7. Improvement due to loop unwrapping.



Perspective

By describing algorithms to sort randomly ordered and distinct single-word keys in a high level language, and using performance statistics from low level implementations on a mythical machine, we have avoided a number of complicated practical issues. For example, a real application might involve writing a program in a high level language to sort a large file of multiword keys on a virtual memory system. While other sorting methods may be appropriate for some particular applications, Quicksort is a very flexible algorithm, and the programs described above can be adapted to run efficiently in many special situations that arise in practice. We shall examine, in turn, ramifications of the analysis, special characteristics of applications, software considerations, and hardware considerations.

Analysis

In a practical situation, we might not expect to have randomly ordered files of distinct keys, so the relevance of the analytic results might be questioned. Fortunately, we know that the standard deviation is low (for Program 1 the standard deviation has been shown to be about $0.648N$ [11, 17]), so we can expect the average running time to be reasonably close to the formulas given (for example, we can be 99 percent sure that the formula for Program 1 is accurate to within $2N$). It is shown in [16] that the assumption that the keys are distinct is justified and that Program 2 performs well when equal keys are present. Furthermore the technique of partitioning on the median of the first, middle, and last elements of the file ensures that Program 2 will work well on files that are almost in order, which do occur in practice. If other biases are suspected, the use of a random element for partitioning will lead to acceptable performance.

All of the Quicksort programs do have an $O(N^2)$ worst case. One can always "work backwards" to find a file which will require time proportional to N^2 to sort. This fact often dissuades people from using Quicksort, but it should not. The low standard deviation says that the worst case is extremely unlikely to occur in a probabilistic sense. This provides little consolation if it does occur in a practical file, and this is possible for Program 1 since files already in order and other simple files will lead to the worst case. This does not seem to be the case for Program 2. Hoare's technique of using a random partitioning element makes it extremely unlikely that the running time will be far from the predicted averages. (The analysis is entirely valid in this case, no matter what the input is.) However, this is more expensive than the method of Program 2, which appears to offer sufficient protection against the worst case.

Applications

We have implicitly assumed throughout that all of the records to be sorted fit into memory—Quicksort is an "internal" sorting method. The issues involved in sorting very, very large files in external storage are *very* different. Most "external" sorting methods for doing so are based on sorting small subfiles on one pass through the data, then merging these on subsequent passes. The time taken by such methods is dependent on physical device characteristics and hardware configurations. Such methods have been studied extensively, but they are not comparable to internal methods like Quicksort because they are solving a different problem.

It is common in practical situations to have multiword keys and even larger records in the fields to be sorted. If records are more than a few words long, it is best to keep a table of pointers and refer to the records indirectly, so only one-word pointers need be exchanged, not long records. The records can be rearranged after the pointers have been "sorted." This is called a "pointer" or "address table" sort (see [11]). The main effect of multiword keys to be considered is that there is more overhead associated with each comparison and exchange. The results given above and in [17] make it possible to compare various alternatives and determine the total expected running time for particular applications. For large records, the improvement due to loop unwrapping becomes unimportant. If the keys are very long, it may pay to save extra information on the stack indicating how many words in the keys are known to be equal (see [6]). Our conclusions comparing methods are still valid, because the extra overhead associated with large keys and records is present in all the sorting methods.

When we say that Quicksort is a good "general purpose" method, one implication is that not much information is available on the keys to be sorted or their distribution. If such information is available, then more efficient methods can be devised. For example, if the keys are the numbers 1, 2, ... , N, and an extra table of size N is available for output, they can be sorted by scanning through the file sequentially, putting the key with value i into the ith position in the table. (This kind of sorting, called "address calculation," can be extended to handle more general situations.) As another example, suppose that the N elements to be sorted have only $2^t + 1$ distinct values, all of which are known. Then we can partition the array on the median value, and apply the same procedure to the subfiles, in total time proportional to $(t + 1)(N + 1)$. It is shown in [16] that Program 1 will take on the order of $(2 \ln 2)tN$ comparisons on such files, so Quicksort does not perform badly. Other special-purpose methods can be adapted to other special situations, but Program 2 can be recommended as a general purpose sorting method because it handles many of these situations adequately.

Software

Modern compilers have not progressed to the point where they can produce the best possible (or even very good) assembly-language translations of high level programs, so we have dealt with "ideal" assembly-language implementations. Standard compilers produce code for Quicksort that is 300–400 percent slower than the assembly-language implementation (see [15]). It is not unreasonable to expect that compilers may someday produce programs close to the ideal, since some of the improvements that we made could be done mechanically and are used in so-called "optimizing" compilers. Quicksort's partitioning loop, because of its structure, is actually a good test case for optimizing compilers—one well-known compiler actually makes the inner loop longer when its optimizing feature is invoked [15].

If a sorting program *must* run efficiently, it should be implemented in assembly language, and we have shown a good way to do so. It is interesting to note that on many computers an implementation of Quicksort in Fortran (for example) will require about as many source statements as an assembly-language implementation (see [15], but it will of course produce a much less efficient program.

If one is willing to pay for the extra overhead of implementing his sorting program in a high level language, then Quicksort should still be used because it will incur relatively less overhead than other methods. Program 2 can be used as it stands, although any effort spent trying to "optimize" it (such as choosing the very best value of M) would be better spent simply implementing it in assembly language. If a sorting program is to be used only a few times on files which are not large, then Program 1 (possibly with "$A[l] :=: A[(l + r) \div 2]$" inserted before partitioning to make the worst case unlikely) will do quite nicely. The only danger is that the stack for recursion might consume excessive space, but this is very unlikely (it will require less than 30 entries, on the average, for files of 10,000 elements [15]) and it provides a convenient "alarm" that the worst case is happening. Program 1 is a simple program whose average running time is low—it will sort thousands of elements in only a few seconds on most modern computer systems.

Hardware

Particular characteristics of particular real computers might allow for further improvements to Quicksort. For example, some computers have "compare and skip" and "increment and test" instructions which allow the inner loops to be implemented in two instructions, thus eliminating the need for loop unwrapping. Similar "local" improvements may be possible in other parts of the programs.

The hardware feature on modern computers that has the most drastic effect on the performance of algorithms is paging. Quicksort actually does not perform badly in a virtual memory situation (see [2]) because it has two slowly changing "localities" around the scanning pointers. In some situations, it will be wise to minimize page faults by performing the extra processing necessary to split the array into many partitions (instead of only two) on the first partitioning stage. Of course, the programs should be changed so that small subfiles are "insertionsorted" as they are encountered, because otherwise the last scan over the whole file will involve unnecessary page faults. Many internal sorting methods do not work well at all under paging, but Quicksort can be adapted to run quite efficiently.

Another hardware feature of interest is parallelism. Quicksort does not take good advantage of the parallelism in large scientific computers, and there are methods which should do better if parallel computations are involved. However, Quicksort has been shown to perform quite well on one such computer [19]. Of course, if true parallelism is available then subfiles can be sorted independently by different processors.

Many modern computers have hardware features such as instruction stacks, pipelined execution, caches, and interleaved storage which can improve performance greatly. Knuth [9] concludes that radix sorting might be preferred on "number-crunching" computers with pipelining. Loop unwrapping could be disastrous on computers with small instructions stacks, and the other features mentioned above will very often hide the time used for pointer arithmetic behind the time used for other instructions. The analysis of the effect of such hardware features can be very difficult, but again Quicksort makes a good test case for such studies because its inner loop is so small and its analysis is so well understood (see the analysis of loop unwrapping in [17]). However, there will probably always remain a role for empirical testing of alternatives in superoptimized implementations on advanced machines.

It is often the case that advanced hardware features allow the implementation of very fast routines for sorting small files. Using such a routine instead of Insertionsorts can lead to substantial improvements for Quicksort on some computers. To develop a good implementation of Quicksort on a new computer, one should first pay careful attention to the partitioning loops, then deal with the problem of sorting small subfiles efficiently.

Conclusion

Our goal in this paper has been to illustrate methods by which a typical computer can be made to sort a file as quickly and conveniently as possible. The algorithm, improvements, and implementation techniques described here should make it possible for readers to implement useful, efficient programs to solve specific sorting problems.

Economic issues surrounding modern computer systems are very complex, and it is necessary always to be sure that it will be worthwhile to implement projected improvements to programs. Many simple applications can be handled perfectly adequately with simple programs such as Program 1. However, sorting is a task which is performed frequently enough that most computer installations have "utility" programs for the purpose. Such programs should use the best techniques available, so something on the order of an assembly-language implementation of Program 2 is called for.

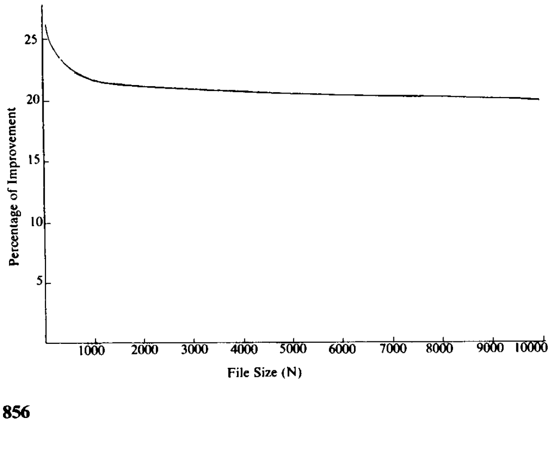
Sorting small subfiles on a separate pass, partitioning on the median of three elements, and unwrapping the inner loops reduces the expected running time on a typical computer from about $11.6667N \ln N + 12.312N$ to about $10.0038N \ln N + 3.530N$ time units. Figure 8 shows the total percentage improvement for these improvements together.

Many of the issues raised above relating to other sorting programs are treated fully in [9], and the issues specific to Quicksort are also dealt with in [15]. We have not described here the countless other variants of Quicksort which have been proposed to improve the algorithm or to deal with the various problems outlines above [1, 4, 13, 14, 20]. Many of these turn out not to be improvements at all: see [15] for complete descriptions. For example, nearly every published implementation of Quicksort uses a different partitioning method. The various methods seem to differ only slightly, but actually their performance characteristics can differ greatly. Caution should be exercised before a partitioning method which differs from those above is used.

Program 2 is the method of choice in many practical sorting situations and will be very quick if properly implemented. Quicksort is an interesting algorithm which combines utility, elegance, and efficiency.

Received May 1976; revised February 1978.

Fig. 8. Cumulative improvement due to sorting small subfiles on a separate pass, median-of-three partitioning, and loop unwrapping.



References

1. Boothroyd, J. Sort of a section of the elements of an array by determining the rank of each element: Algorithm 25; and Ordering the subscripts of an array section according to the magnitudes of the elements: Algorithm 26. *Comptr. J. 10* (Nov. 1967), 308–310. (See notes by R.S. Scowen in *Comptr. J. 12* (Nov. 1969), 408–409, and by A.D. Woodall in *Comptr. J. 13* (Aug. 1970.)
2. Brawn, B.S., Gustavson, F.G., and Mankin, E. Sorting in a paging environment. *Comm. ACM 13,* 8 (Aug. 1970), 483–494.
3. Cocke, J., and Schwartz, J.T. Programming languages and their compilers. Preliminary Notes. Courant Inst. of Math. Sciences, New York U., New York, 1970.
4. Frazer, W.D., and McKellar, A.C. Samplesort: A sampling approach to minimal storage tree sorting. *J. ACM 17,* 3 (July 1970), 496–507.
5. Hoare, C.A.R. Partition: Algorithm 63; Quicksort: Algorithm 64; and Find: Algorithm 65. *Comm. ACM 4,* 7 (July 1961), 321–322. (See also certification by J.S. Hillmore in *Comm. ACM 5,* 8 (Aug. 1962), 439, and B. Randell and L.J. Russell in *Comm. ACM 6,* 8 (Aug. 1963), 446.)
6. Hoare, C.A.R. Quicksort. *Computer J. 5,* 4 (April 1962), 10–15.
7. Knuth, D.E. *The Art of Computer Programming, Vol. 1:*

Fundamental Algorithms. Addison-Wesley, Mass., 1968.
8. Knuth, D.E. *The Art of Computer Programming, Vol. 2: Seminumerical Algorithms.* Addison-Wesley, Mass., 1969.
9. Knuth, D.E. *The Art of Computer Programming, Vol. 3: Sorting and Searching.* Addison-Wesley, Mass., 1972.
10. Knuth, D.E. Structured programming with go to statements. *Computing Surveys 6,* 4 (Dec. 1974), 261–301.
11. Loeser, R. Some performance tests of "quicksort" and descendants. *Comm. ACM 17,* 3 (March 1974), 143–152.
12. Morris, R. Some theorems on sorting. *SIAM J. Appl. Math. 17,* 1 (Jan. 1969), 1–6.
13. Rich, R.P. *Internal Sorting Methods Illustrated with PL/I Progams.* Prentice-Hall, Englewood Cliffs, N.J., 1972.
14. Scowen, R.S. Quickersort: Algorithm 271. *Comm. ACM 8,* 11 (Nov. 1965), 669–670. (See also certification by C.R. Blair in *Comm. ACM 9,* 5 (May 1966), 354.)
15. Sedgewick, R. Quicksort. Ph.D. Th. Stanford Comptr. Sci. Rep. STAN-CS-75-492, Stanford U., Stanford, Calif., May 1975.
16. Sedgewick, R. Quicksort with equal keys. Siam J. Comput. *6,* 2 (June 1977), 240–287.
17. Sedgewick, R. The analysis of Quicksort programs. *Acta Informatica 7* (1977), 327–355.
18. Singleton, R.C. An efficient algorithm for sorting with minimal storage: Algorithm 347. *Comm. ACM 12,* 3 (March 1969), 185–187. (See also remarks by R. Griffin and K.A. Redish in *Comm. ACM 13,* 1 (Jan. 1970), 54 and by R. Peto, *Comm. ACM 13,* 10 (Oct. 1970), 624.)
19. Stone, H.S. Sorting on STAR. IEEE Trans. Software Eng. *SE-4,* 2 (Mar. 1978), 138–146.
20. van Emden, M.N. Increasing the efficiency of quicksort: Algorithm 402. *Comm. ACM 13,* 11 (Nov. 1970), 693–694. (See also the article by the same name in *Comm. ACM 13,* 9 (Sept. 1970), 563–567.)
21. Wirth, N. *Algorithms + Data Structures = Programs.* Prentice-Hall, Englewood Cliffs, N.J., 1976.