Striving for Efficiency in Algorithms: Sorting

Inna Pivkina[*]

Introduction

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" ([7]), 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 [14]. 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 ([14]), 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."

In his Turing Award Lecture, "The Emperor's Old Clothes," ([6]) Hoare mentions that around Easter 1961 he attended a course on Algol 60 in Brighton, England. He says:

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

"It was there that I first learned about recursive procedures and saw how to program the sorting method which I had earlier found such difficulty in explaining. It was there that I wrote the procedure, immodestly named QUICKSORT, on which my career as a computer scientist is founded."

Even though Hoare had the algorithm in mind in the late 1950's, it was not until 1961, when attending a course in the ALGOL 60 programming language, that he discovered that Quicksort could be more easily explained as a recursive algorithm. This is because no mainstream high-level language prior to ALGOL 60 had even allowed recursive procedures. 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 ([10], [11], [13]). Sedgewick in his paper "Implementing Quicksort programs" ([13]) 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.

Programming in the seventies was very different from programming now. Today, if you need to sort items your best choice may be to use the system sort function. Compilers today are often doing a better job than people at optimizing the code. This was not the case in the seventies. Back in the seventies, sorting was extremely important and good system sort functions were not developed yet. Creativity was necessary for programmers of the era. This creativity included techniques to eliminate recursion, to switch to a more elementary algorithm when the subarray under consideration was small enough, and other things described in Sedgewick's paper. When reading the paper, you will find that optimizations discussed in the paper make the code harder to understand but it was needed to complicate the code this way to make it more efficient.

The main idea of the project is to experimentally verify whether modifications to the Quicksort algorithm proposed by Sedgewick in [13] are still good today or not. Would they make any difference in sorting time today?

The project is divided into two parts. The first part contains excerpts of Sedgewick's paper ([13]) interleaved with exercises. The purpose of these exercises is to engage the reader with the algorithm and its optimizations discussed in the paper. The second part contains exercises guiding the reader to experimentally compare the original and modified versions of Quicksort.

Part 1: Sedgewick's paper

The Algorithm

Quicksort is a recursive method for sorting an array $A[1]$, $A[2]$, ..., $A[N]$ 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 $A[1]$, $A[2]$, ..., $A[j-1]$ are called the "left subfile.")

(iii) All elements to the right of $A[j]$ are greater than or equal to it. (These elements $A[j+1]$, ..., $A[N]$ 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 $A[r+1] \geq A[l], \ldots, A[r]$;
    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 [8]. 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 $A[1]$, $A[2]$, . . . , $A[N]$ 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 $A[l + 1]$, . . . , $A[j]$ are $< v$ and $A[j + 1]$, . . . , $A[r]$ are $> v$, so that the exchange $A[i]$:=:$A[j]$ 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 $A[l]$,. . . , $A[r]$ 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 [10]. By using this particular method,

Fig. 1. Partitioning π (Program 1).

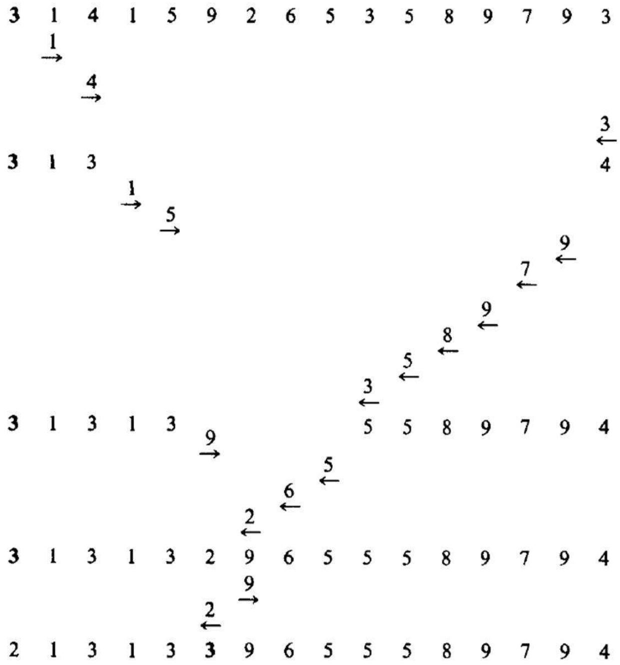
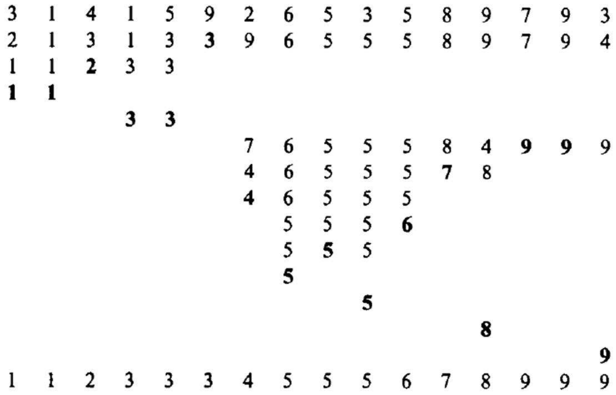


Fig. 2. Quicksorting π (Program 1).



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 $i \leq N$ 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 [15].) 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 [12], 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 $A[l], \ldots, A[N]$, 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 produces nonrandom subfiles has yet been successfully analyzed, but empirical tests show that such methods slow down Quicksort by up to 20 percent (see [8], [10]).

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 likely different from those which you use when writing pseudocode. For instance, operator :=: and the control structures loop ... repeat. Rewrite Program 1 using your favorite pseudocode conventions.

Exercise 1.2. Partitioning algorithm which is used in Program 1 is different from versions of partitioning given in some modern textbooks, e.g. [1]. Write a pseudocode for procedure Partition1 which would perform partitioning using the partitioning algorithm from Program 1. Use your favorite pseudocode conventions.

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 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 exercise 1.3. You may use Figure 2 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 as an example of showing the values of the array.) Notice that this example fits the description of the partitioning process from the paragraph 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 paragraph of the paper which starts with words "If equal keys are present among $A[1]$,…,$A[N]$ …".

Demonstrate the operation Partition1 on this array A.

Exercise 1.7. Sedgewick writes: "For example, rather than using the "sentinel" $A[N+1] = \infty$ we could use

> loop: $i := i + 1$; while $i \leq N$ and $A[i] < \nu$ 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?

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 [11], 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 $A[1], \ldots A[N]$ 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 $\log_2 N$ [5]. 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 [8]). 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 [11] 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 formulas derived in [11] 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 [11] for all the improvements described below, and these are quite sufficient for comparing the methods and predicting their performance.

Exercise 1.9. Sedgewick says: "For example, if the file $A[1], \ldots, A[N]$ 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 $A[1], \ldots, A[N]$ (assume that all the elements being sorted are distinct).

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 [5]. 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 $A[r+1] \geq A[l], \ldots, A[r]$;
    loop for $r-1 \geq i \geq l$:
        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 [7] and [10].) 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 [10] 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 [11] 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.

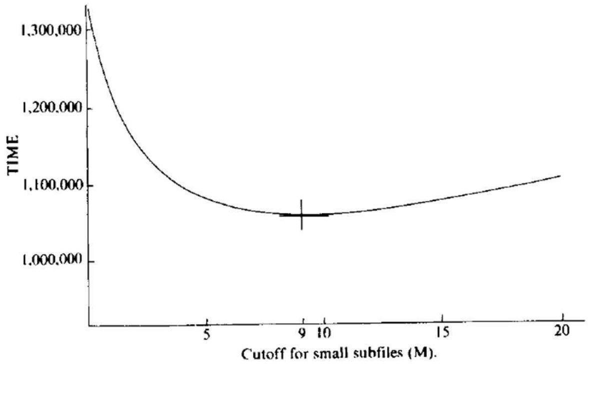
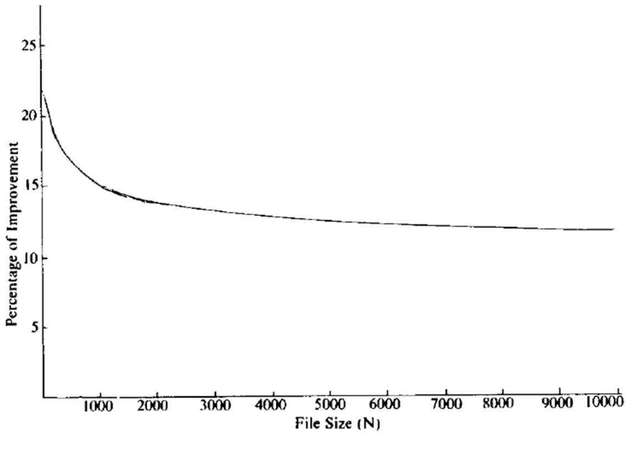


Fig. 4. Improvement due to sorting small subfiles on a separate pass.



Exercise 1.10. Rewrite procedure *insertionsort* of the paper using your favorite pseudocode conventions. (The procedure is different from the insertion sort implementation in some textbooks, e.g. [1].)

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. 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 is this an improvement?

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

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

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 $A[1], A[2], \ldots, A[N]$ are in order already when Program 1 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 $A[2], \ldots, A[N]$. Then the same thing will happen to that subfile, and so on. The program has to deal with files of size N, $N-1$, $N-2, \ldots$ 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 [5]. 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.

Exercise 1.15. 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 1.16. How did Sedgewick suggest to modify Program 1 in order to implement the method from exercise 1.15? When describing the modification please use your favorite pseudocode conventions.

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 sub files, and three elements are sufficient for this purpose. (See [10] and [11] 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 [15]. Again, care must be taken not to disturb the partitioning process. The method can be implemented by inserting the statements

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

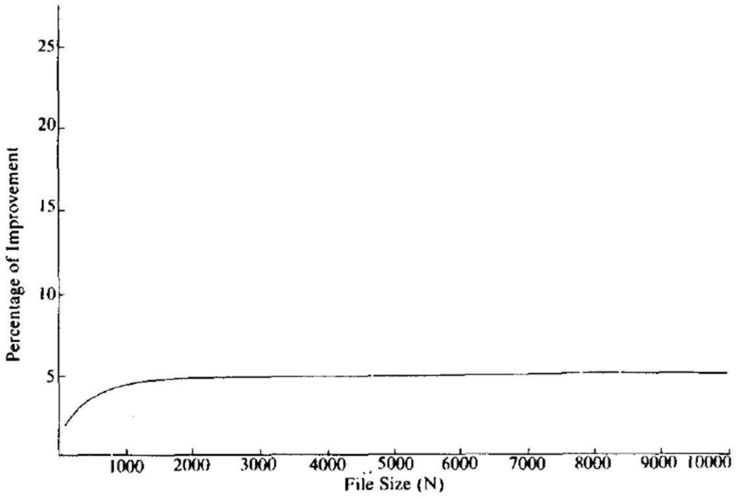
before partitioning (after “if $r > l$ then” in Program 1. This change makes $A[l]$ the median of the three elements originally at $A[l]$, $A[(l+r) \div 2]$, and $A[r]$ before partitioning. Furthermore, it makes $A[l+1] \leq A[l]$ and $A[r] \geq A[l]$, so the pointer initializations can be changed to “$i := l+1$; $j := r$”. 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 [11] (with the optimum value $M = 9$) is about $10.6286N \ln N + 2.116N$ time units, and Figure 5 shows the percentage savings.

Exercise 1.17. Describe in your own words the idea of median-of-three modification. What is the purpose of this modification?

Exercise 1.18. What implementation of median-of-three modification does Sedgewick present in the paper? When describing the implementation please use your favorite pseudocode conventions.

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



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 $A[1], \ldots, A[N]$.

```
Program 2

integer $l, r, i, j$;
integer array *stack*$[1 : 2 \times f(N)]$;
boolean *done*;
arbmode array $A[1 : N + 1]$;
arbmode $v$;
$l := 1$; $r := N$; *done*$:= N \leq M$;
loop until *done*:
    $A[(l + r) \div 2] :=: A[l + 1]$;
    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;
    $i := l + l$; $j := r$; $v := A[l]$;
    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$(j - l, r - i + 1) \leq M$
    then if *stack empty*
        then *done* := true
        else $(l, r)$ := *popstack*
        endif;
    else if $min(j-l, r-i+l) \le M$
        then $(l,r)$ :=large subfile;
        else pushstack(large subfile);
            $(l,r)$ := small subfile
        endif;
    endif;
repeat;
$A[N+1] := \infty$;
loop for $N-1 \ge i \ge 1$ :
    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;
```

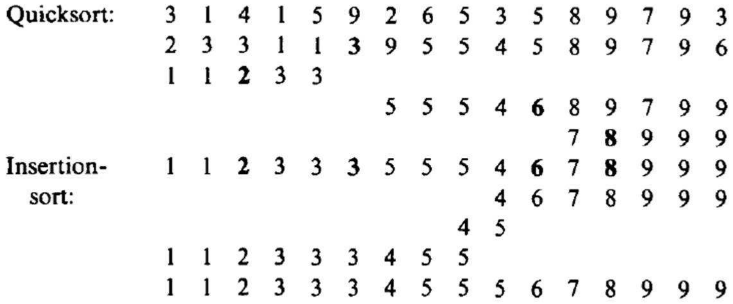
In the logic for manipulating the stack after partitioning, $(l, j-1)$ is the "large subfile" and (i, r) is the "small subfile" if $max(j-l, r-i+1) = j-l$, and vice versa if $r-i+1 > j-l$. This may be implemented simply and efficiently by making one copy of the code for each of the two outcomes of comparing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

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 $M \ge 2$, 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 [9], [10], [11].)

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.

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

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 [11].) The analysis in [11] 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
$C_N \simeq 1.714N \ln N - 3.74N$,
$B_N \simeq .343N \ln N - .84N$,
$E_N \simeq 1.14N$, $D_N \simeq .60N$,
$A_N \simeq .16N$, $S_N \simeq .05N$.
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 [7], [10], [11], the total expected running time is $53\frac{1}{2}A_N + 11B_N + 4C_N + 3D_N + 8E_N + 9S_N + 7N$, which leads to the equation $10.6286N \ln N + 2.116N$ given above.

Exercise 1.19. Rewrite Program 2 using your favorite pseudocode conventions.

Exercise 1.20. Explain why condition $A[N + 1] = \infty$ is needed in Program 2. What will happen if the condition is not there?

Exercise 1.21. 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 as an example. (This exercise is closely related to the next exercise 1.22.)

Exercise 1.22. 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 1.21? (Give exact numbers.)

Part 2: Implementation and testing

Exercise 2.1. 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 measure how much time was spent on sorting. The measured time should not include time spent on reading and writing data to/from files.

Exercise 2.2. 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 measure how much time was spent on sorting. The measured time should not include time spent on reading and writing data to/from files.

Exercise 2.3. 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 2.4. Generate 10 different input files using your Generator program. Each file should have ten thousand integers in the range from 1 to 10000.

Exercise 2.5. Sort each of the input files generated in exercise 2.4 using Program 1. Record sorting times. What are the average, minimum and maximum sorting times?

Exercise 2.6. Sort each of the input files generated in exercise 2.4 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 2.7. Summarize your results from exercises 2.5 and 2.6 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" ([13]). 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 whether modifications to the Quicksort algorithm proposed by Sedgewick would make any difference in sorting time today.

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. It could also be used to illustrate how developments in computer architecture and computer technology have affected programming since the seventies. When Sedgewick wrote the paper, compilers were unable to produce high-quality code and hand-optimization was important. Today the situation is reversed: compilers can often produce code that better exploits a specific architecture than people can.

The project is divided in two parts. The first part contains excerpts of Sedgewick's paper ([13]) interleaved with exercises. The purpose of these exercises is to engage the reader with the algorithm and its optimizations discussed in the paper. The second part contains exercises guiding the reader to experimentally compare the original and modified versions of Quicksort. The first part (Sedgewick's paper) can be split further in two segments which would contain approximately the same amount of reading from the Sedgewick's paper. The first segment could include sections "The Algorithm", "Improvements", "Removing Recursion", and "Small Subfiles", together with the related exercises. The second segment could include sections "Worst case", "Median-of-three Modification", and "Implementation", together with the related exercises.

The project can be given as a two- or three-part homework assignment. It can be done individually or in small groups in about 2 weeks.

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] C. A. R. Hoare. The emperor's old clothes. *Commun. ACM*, 24(2):75–83, 1981.

[7] D.E. Knuth. *The Art of Computer Programming, Vol. 3: Sorting and Searching*. Addison-Wesley, Mass., 1972.

[8] D.E. Knuth. Structured programming with go to statements. *Computing Surveys*, 6(4):261–301, Dec. 1974.

[9] R. Loeser. Some performance tests of "quicksort" and descendants. *Communications of the ACM*, 17(3):143–152, March 1974.

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

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

[12] R. Sedgewick. Quicksort with equal keys. *Siam J. Comput.*, 6(2):240–287, June 1977.

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

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

[15] R.C. Singleton. An efficient algorithm for sorting with minimal storage: Algorithm 347. *Communications of The ACM*, 12(3):185–187, March 1969.