Quicksort

By C. A. R. Hoare

A description is given of a new method of sorting in the random-access store of a computer. The method compares very favourably with other known methods in speed, in economy of storage, and in ease of programming. Certain refinements of the method, which may be useful in the optimization of inner loops, are described in the second part of the paper.

Part One: Theory

The sorting method described in this paper is based on the principle of resolving a problem into two simpler subproblems. Each of these subproblems may be resolved to produce yet simpler problems. The process is repeated until all the resulting problems are found to be trivial. These trivial problems may then be solved by known methods, thus obtaining a solution of the original more complex problem.

Partition

The problem of sorting a mass of items, occupying consecutive locations in the store of a computer, may be reduced to that of sorting two lesser segments of data, provided that it is known that the keys of each of the items held in locations lower than a certain dividing line are less than the keys of all the items held in locations above this dividing line. In this case the two segments may be sorted separately, and as a result the whole mass of data will be sorted.

In practice, the existence of such a dividing line will be rare, and even if it did exist its position would be unknown. It is, however, quite easy to rearrange the items in such a way that a dividing line is brought into existence, and its position is known. The method of doing this has been given the name *partition*. The description given below is adapted for a computer which has an *exchange* instruction; a method more suited for computers without such an instruction will be given in the second part of this paper.

The first step of the partition process is to choose a particular key value which is known to be within the range of the keys of the items in the segment which is to be sorted. A simple method of ensuring this is to choose the actual key value of one of the items in the segment. The chosen key value will be called the *bound*. The aim is now to produce a situation in which the keys of all items below a certain dividing line are equal to or less than the bound, while the keys of all items above the dividing line are equal to or greater than the bound. Fortunately, we do not need to know the position of the dividing line in advance; its position is determined only at the end of the partition process.

The items to be sorted are scanned by two pointers: one of them, the *lower pointer*, starts at the item with lowest address, and moves upward in the store, while the other, the *upper pointer*, starts at the item with the highest address and moves downward. The lower pointer starts first. If the item to which it refers has a key which is equal to or less than the bound, it moves up to point to the item in the next higher group of locations. It continues to move up until it finds an item with key value greater than the bound. In this case the lower pointer stops, and the upper pointer starts its scan. If the item to which it refers has a key which is equal to or greater than the bound, it moves down to point to the item in the next lower locations. It continues to move down until it finds an item with key value less than the bound. Now the two items to which the pointers refer are obviously in the wrong positions, and they must be exchanged. After the exchange, each pointer is stepped one item in its appropriate direction, and the lower pointer resumes its upward scan of the data. The process continues until the pointers cross each other, so that the lower pointer refers to an item in higher-addressed locations than the item referred to by the upper pointer. In this case the exchange of items is suppressed, the dividing line is drawn between the two pointers, and the partition process is at an end.

An awkward situation is liable to arise if the value of the bound is the greatest or the least of all the key values in the segment, or if all the key values are equal. The danger is that the dividing line, according to the rule given above, will have to be placed outside the segment which was supposed to be partitioned, and therefore the whole segment has to be partitioned again. An infinite cycle may result unless special measures are taken. This may be prevented by the use of a method which ensures that at least one item is placed in its correct position as a result of each application of the partitioning process. If the item from which the value of the bound has been taken turns out to be in the lower of the two resulting segments, it is known to have a key value which is equal to or greater than that of all the other items of this segment. It may therefore be exchanged with the item which occupies the highest-addressed locations in the segment, and the size of the lower resulting segment may be reduced by one. The same applies, *mutatis mutandis*, in the case where the item which gave the bound is in the upper segment. Thus the sum of the numbers of items in the two segments, resulting from the partitioning process, is always one less than the number of items in the original segment, so that it is

certain that the stage will be reached, by repeated partitioning, when each segment will contain one or no items. At this stage the process will be terminated.

Quicksort

After each application of the partitioning process there remain two segments to be sorted. If either of these segments is empty or consists of a single item, then it may be ignored, and the process will be continued on the other segment only. Furthermore, if a segment consists of less than three or four items (depending on the characteristics of the computer), then it will be advantageous to sort it by the use of a program specially written for sorting a particular small number of items. Finally, if both segments are fairly large, it will be necessary to postpone the processing of one of them until the other has been fully sorted. Meanwhile, the addresses of the first and last items of the postponed segment must be stored. It is very important to economize on storage of the segment details, since the number of segments altogether is proportional to the number of items being sorted. Fortunately, it is not necessary to store the details of all segments simultaneously, since the details of segments which have already been fully sorted are no longer required.

The recommended method of storage makes use of a *nest*, i.e. a block of consecutive locations associated with a pointer. This pointer always refers to the lowest-addressed location of the block whose contents may be overwritten. Initially the pointer refers to the first location of the block. When information is to be stored in the nest, it is stored in the location referred to by the pointer, and the pointer is stepped on to refer to the next higher location. When information is taken from the list, the pointer is stepped back, and the information will be found in the location referred to by the pointer. The important properties of a nest are that information is read out in the reverse order to that in which it is written, and that the reading of information automatically frees the locations in which it has been held, for the storage of further information.

When the processing of a segment has to be postponed, the necessary details are placed in the nest. When a segment is found to consist of one or no items, or when it has been sorted by some other method which is used on small segments, then it is possible to turn to the processing of one of the postponed segments; the segment chosen should always be the one most recently postponed, and its details may therefore be read from the nest. During the processing of this segment, it may be necessary to make further postponements, but now the segment details may overwrite the locations used during the processing of the previous segment. This is, in fact, achieved automatically by the use of a nest.

It is important to know in advance the maximum number of locations used by the nest; in order to ensure that the number of segments postponed at any one time never exceeds the logarithm (base 2) of the number of items to be sorted, it is sufficient to adopt the rule of always postponing the processing of the larger of the two segments.*

Estimate of Time Taken

The number of key comparisons necessary to partition a segment of N items will depend on the details of the method used to choose the bound, or to test for the completion of the partition process. In any case the number of comparisons is of the form $N - k$, where k may be $-1, 0, 1, 2$.

The number of exchanges will vary from occasion to occasion, and therefore only the expected number can be given. An assumption has to be made that the value of the bound is a random sample from the population of key values of the items in the segment. If this assumption is not justified by the nature of the data being sorted, it will be advisable to choose the item which yields the bound value *at random*, so that in any case the assumption of randomness will be valid.

In the calculations which follow, use is made of the principle of conditional expectation. We consider separately the case where the bound is the rth in order of magnitude of all the key values in the segment; the value of the conditional expectation of the quantity which interests us may now be expressed quite simply as a function of r. The rule of conditional expectation states that if each conditional expectation is multiplied by the probability of occurrence of the condition, and they are summed over the whole range of conditions, the result gives the unconditional or absolute expectation. According to the assumption of randomness, all the values of r between 1 and N inclusive are equally likely, so that they each have a probability of $\frac{1}{N}$. If, therefore, the expression which gives the conditional expectation on assumption of a given r is summed with respect to r and divided by N, we obtain the value of the absolute expectation of the quantity concerned.

Consider the situation at the end of the partition process, when the bound was the rth key value in order of magnitude. As a result of the final exchange, the item which yielded this key value will occupy the rth position of the segment, and the $r - 1$ items with lesser key value will occupy the $r - 1$ positions below it in the store. The number of exchanges made in the course of the partition process is equal to the number of items which originally occupied the $r - 1$ positions of the lower resulting segment, but which were removed because they were found to have key values greater than the bound. The probability of any key value being greater than the bound is $\frac{N - r - 1}{N}$, and therefore the expected number of such items among the $r - 1$ items

* A description of Quicksort in ALGOL (Hoare, 1961) is rather deceptively simple, since the use of recursion means that the administration of the nest does not have to be explicitly described. The claim to a negative sorting time in the reference is, of course, due to a misprint.

which originally occupied what was to be the lower resulting segment is:

$$\frac{(N-r-1)(r-1)}{N}.$$

Summing with respect to r, dividing by N, and adding one for the final exchange of the item which yielded the bound, we get the absolute expectation of the number of exchanges:

$$\frac{N}{6}+\frac{5}{6N}$$

This figure may be reduced by $\frac{1}{N}$ if the final exchange is always omitted in the case when the item which provided the bound is already in its correct position. In general it will not be worth while to test for this case.

Given the expected theoretical number of comparisons and exchanges, it should be quite easy to calculate the expected time taken by a given program on a given computer. The formula for the time taken to partition a segment of N items will take the form

$$aN+b+\frac{c}{N},$$

where the coefficients a, b and c are determined by the loop times of the program. The expected time taken to sort N items will be denoted T_N. We shall suppose that a different method of sorting is used on segments of size less than M. The values of T_r for $r<M$ are taken as given. We shall find a recursive relationship to give the values of T_r for $r \geqslant M$.

Suppose that the value of the bound chosen for the first partition is the rth in order of magnitude. Then the time taken to sort the whole segment of N items is equal to the time taken to partition the N items, plus the time taken to sort the $r-1$ items of the lower resulting segment, plus the time taken to sort the $N-r-1$ items of the upper resulting segment. This assertion must also be true of the expected times

$$T_N = T_r + T_{N-r} + aN + b + \frac{c}{N},$$

on condition that the first bound was the rth. Summing with respect to r and dividing by N we get the unconditional expectation

$$T_N = \frac{2}{N}\sum_{1}^{N-1} T_r + aN + b - \frac{c}{N}, \qquad N \geqslant M.$$

The exact solution of this recurrence equation is*

$$T_N = \frac{2(N-1)}{M(M-1)}\sum_{1}^{M-1} T_r + \frac{(N+1)c}{M(M-1)} - \left[\frac{2(N-1)}{M-1} - 1\right]b - \left[2(N-1)\sum_{M+1}^{N}\frac{1}{r} - \frac{4(N-1)}{M-1} + N + 4\right]a.$$

* We adopt the convention that a sum is zero if its upper bound is less than its lower bound.

The validity of the solution may be proved by substituting in the original equation, and showing that the result is an algebraic identity. For simplicity, the coefficients of $\sum_{1}^{M-1} T_r$, c, b, and a should be considered separately. The correctness of the first three coefficients is easily established. In verifying the coefficient of a, the following identities are used. Writing W_N for $\sum_{M+1}^{N}\frac{1}{r} + \frac{2}{N+1} - \frac{2}{M+1}$ and V_N for the coefficient of a in T_N, we get

$$V_N = (N+1)(N-2)W_{N-1} - N(N+1)W_N \qquad (1)$$

$$= \frac{2}{N}N(N+1)W_N + N$$

$$= \frac{2}{N}\sum_{M}^{N-1} V_r - N. \qquad \text{from (1)}$$

It is interesting to compare the average number of comparisons required to sort N items, where N is very large, with the theoretical minimum number of comparisons. We consider the case $M=2$, and find the expected number of comparisons by putting $a=1$, $b=c=T_1=0$ in the formulae of the last paragraph. When N is very large, all terms except the largest may be ignored. The figure obtained for the expected number of comparisons is

$$2N\sum_{1}^{N}\frac{1}{r} \sim 2N\log_e N.$$

The theoretical minimum average number of comparisons required to sort N unequal randomly-ordered items may be estimated on information-theoretic considerations. As a result of a single binary comparison, the maximum entropy which may be destroyed is $-\log 2$, while the original entropy of the randomly ordered data is $-\log N!$; the final entropy of the sorted data is zero. The minimum number of comparisons required to achieve this reduction in entropy is

$$\frac{-\log N!}{-\log 2} = \log_2 N! \sim N\log_2 N.$$

The average number of comparisons required by Quicksort is greater than the theoretical minimum by a factor of $2\log_e 2 \sim 1{\cdot}4$. This factor could be reduced by the expedient of choosing as the bound for each partition the median of a small random sample of the items in the segment. It is very difficult to estimate the saving which would be achieved by this, and it is possible that the extra complication of the program would not be justified. Probably more worthwhile is the attempt to reduce as far as possible the actual time taken by the innermost comparison cycle, and a number of simple programming devices to achieve this will be described in Part Two of this paper.

A Comparison of Quicksort with Merge Sorting

The National-Elliott 405 computer has a delay-line working store of 512 locations, and a magnetic-disc backing store of 16,384 words. The average access time for the working store is 0·8 msec and the average access time for a block of 64 words in the backing store is 32 msec. There are 19 words of immediate-access storage, which are used to contain instructions and working space of the inner loops; the time taken by such loops is about 0·15 msec per instruction.

Table 1 gives a comparison of times taken by Quicksort and a merge sorting method, both programmed by Mr. P. Shackleton for the 405. The times were measured automatically by the computer in tests on random data conducted by Mr. D. J. Pentecost. The figures relate to six-word items with a single-word key.

Table 1

NUMBER OF ITEMS	MERGE SORT	QUICKSORT
500	2 min 8 sec	1 min 21 sec
1,000	4 min 48 sec	3 min 8 sec
1,500	8 min 15 sec*	5 min 6 sec
2,000	11 min 0 sec*	6 min 47 sec

* These figures were computed by formula, since they cannot be achieved on the 405 owing to limited store size.

Part Two: Implementation

In the implementation of a sorting method on a given computer, it is often possible to make adaptations which will ensure optimization of the innermost loops. Quicksort turns out to be exceptionally flexible; a number of possible variations are described below. The choice of which variation is adopted on any given computer will, of course, depend on the characteristics of the computer. In making the decision, the theoretical estimate of time taken for various values of a, b, c, and M should be used to determine the optimal method; it will not be necessary to write and test a large number of different programs.

Partition without Exchange

On some computers the exchange operation would involve copying one of the items into workspace while the other item overwrites the locations which it occupied. On such a computer it would be advantageous to avoid exchanging altogether, and a method of achieving this is described below.

The item chosen to yield the bound should always be that which occupies the highest-addressed locations of the segment which is to be partitioned. If it is feared that this will have a harmfully non-random result, a randomly chosen item should be initially placed in the highest-addressed locations. The item which yielded the bound is copied into working space. Then the upper and lower pointers are set to their initial values, and the lower pointer starts its upward scan of the store. When it finds an item with key greater than the bound, this item is copied into the locations to which the upper pointer now refers. The upper pointer is stepped down, and proceeds on its downward scan of the data. When it finds an item with key lower than the bound, this item is copied into the locations referred to by the lower pointer. The lower pointer is then stepped up, and the process is repeated until both the pointers are referring to the same item. Then the item which has supplied the bound is copied from working space into the locations to which the pointers refer. Throughout the process, the stationary pointer refers to locations whose contents have been copied elsewhere, while the moving pointer searches for the item to be copied into these locations. The expected number of copying operations is obviously twice the corresponding figure for exchanges.

Cyclic Exchange

On a machine with single-address instructions, which has the facility of exchanging the contents of accumulator and store, it is more economical to perform long sequences of exchanges at one time. A single exchange operation involves reading to the accumulator, exchanging with store, and writing to store, giving $3N$ instructions to perform N exchanges. If these exchanges are performed cyclically all at the same time, one exchange instruction can take the place of a read and a write instruction in all the exchanges except the first and the last. Thus only one read instruction, one write instruction, and $2N - 1$ exchange instructions are required. Further economy is achieved in the case of multi-word items by the fact that the count of words exchanged need be tested only once for each N-fold exchange of each word of the item.

The method of Quicksort allows all exchanges to be saved up until the end of the partitioning process, when they may be executed together in a cyclic movement. In practice, the values of the pointers at the time when they come to a halt are stored in a list for later exchanging. The number of locations which can be spared to hold this list will be a limiting factor in the gain of efficiency.

Optimization of the Key Comparison Loop

Most sorting methods require that a test be made every time that a pointer is stepped, to see whether it has gone outside its possible range. Quicksort is one of the methods which can avoid this requirement by the use of sentinels. Before embarking on the sort, sentinels in the form of items with impossibly large and small key values are placed at each end of the data to be sorted. Now it is possible to remove the pointer test from the key comparison cycle; the test is made only when both pointers are stopped and an exchange is just about to be made. If, at this time, the pointers have not crossed, the exchange is made and the partition process is continued. If they have crossed over, the partition process is at an end.

If the value of the bound is the greatest or the least (or both) of the key values of items in the segment being partitioned, then one (or both) of the pointers will move outside the segment; but no harm can result, provided neither pointer moves outside the area in which the whole mass of data is stored. The upper sentinel, having a key value necessarily greater than that of the bound, will stop the lower pointer, while the lower sentinel will stop the upper pointer. The fact that two extra key comparisons are made on every application of the partition process will be more than compensated on fairly large segments by the omission of pointer comparison from the innermost loop.

Multi-word Keys

When the keys, with respect to which the sorting is performed, extend over more than one computer word, then a long time may be spent on comparing the second and subsequent words of the key. This is a serious problem, since it often happens that a large number of items share a very few values for the first words of their keys. The problem is aggravated when the items are nearly sorted, and it is necessary to make many comparisons between keys which are identical except in their last word. The method described below is due to Mr. P. Shackleton.

The principle of the method is to compare only a single word of the keys on each application of the partitioning process. When it is known that a segment comprises all the items, and only those items, which have key values identical to a given value over their first n words, then, in partitioning this segment, comparison is made of the $(n + 1)$th word of the keys. A variation of the method of partitioning is adopted to ensure that all items with identical values of the key word currently being compared (and consequently identical over earlier words of their keys) are gathered together in one segment as quickly as possible.

The variation consists in altering the criteria which determine the stopping of the pointers. If we ensure that all items with key values equal to the bound are placed in the upper of the resulting segments, then we may associate with each segment its so-called *characteristic value*, which is the greatest value equal to or less than all the key values of the segment (using the expression *key value* to mean the value of the word of the key which will be compared when the segment is partitioned). Furthermore, each segment must contain all the items with key value equal to the characteristic value of the segment. This is easily achieved by making the lower pointer stop whenever it meets an item with key value equal to the bound, so that such an item will be transferred to the upper segment. The value of the bound may obviously be taken as the characteristic value of the upper resulting segment, while the characteristic value of the lower resulting segment is the same as that of the original segment which has just been partitioned. Where this rule does not determine the characteristic values (as in the case of the original mass of data), then no harm will be occasioned by choosing as characteristic value the lowest possible value of the key word.

Now whenever a segment is to be partitioned, the value chosen as the bound is compared with the characteristic value of the segment. If it is greater, partitioning is performed with the modification described in the last paragraph. If, however, they are equal, then it is the upper pointer which is made to stop on encountering an item with key value equal to the bound. Thus all items with key values equal to the characteristic value are collected together in the *lower* resulting segment, and when this segment comes to be partitioned, comparison may be made of the next word of the keys (if any).

The adoption of this refinement means that when the processing of a segment is postponed, the position of the key word which is next to be considered, and the characteristic value for the segment, must be stored together with the positions of the first and last items. On many machines, the extra book-keeping will be justified by the consequent optimization of the innermost comparison loop.

Multilevel Storage

Quicksort is well suited to machines with more than one level of storage, for instance a fast-access working store on magnetic cores and a backing store on magnetic discs or drums. The data in the backing store are partitioned repeatedly until each resulting segment may be contained in the fast-access store, in which it may be sorted at high speed.

The partitioning process can be applied quite economically to data held on a magnetic drum or disc backing store. The reason for this is that the movement of the pointers allows serial transfer of information held adjacently in the backing store, and such transfers are usually faster than if more scattered random access were required. This is particularly true if information can only be transferred between the backing store and main store in large blocks. The time lost in searching for information on the backing store may be reduced to insignificant proportions, provided that it does not take an exceptionally long time to search for information at one end of the store immediately after transferring information at the other end. This condition is satisfied by many magnetic drums or disc stores; it is obviously not satisfied by a magnetic-tape store, on which the method of Quicksort cannot usefully be applied.

Conclusion

Quicksort is a sorting method ideally adapted for sorting in the random-access store of a computer. It is equally suited for data held in core storage and data held in high-volume magnetic drum or disc backing stores. The data are sorted *in situ*, and therefore the whole store may be filled with data to be sorted. There is no need to sort simultaneously with input or output.

The number of cycles of the innermost comparison loop is close to the theoretical minimum, and the loop may be made very fast. The amount of data movement within the store is kept within very reasonable bounds. Quicksort is therefore likely to recommend itself as the standard sorting method on most computers with a large enough random-access store to make interna sorting worth while.

Acknowledgement

This paper is published by kind permission of Elliott Brothers (London) Ltd.

Reference

HOARE, C. A. R. (1961). Algorithm 63, Partition: Algorithm 64, Quicksort; *Communications of the ACM*, Vol. 4, p. 321.

Zero-Address Computers

By P. Wegner

The literature on digital computers makes a distinction between one-address, two-address and three-address machines, where the nature of the beast is determined by the number of references to the main random access memory permitted in a single instruction of the basic machine code. After some initial confusion, the one-address machine emerged as the dominant type, and its pre-eminent position has been largely unchallenged during the past five years or so.

I should like to draw attention to the fact that the position of the one-address machine is being challenged by a new type of animal which, on the basis of the above classification, must be called a *zero-address* machine.

The basic arithmetic operations are of the three-operand type. For instance, the operation C = A − B has two operands as input and one operand as output. Three-address machines permit basic machine instructions to refer to the three operands explicitly. Two-address machines refer to two operands explicitly, and one operand (usually the result) implicitly. One-address machines refer to one operand explicitly, and assume that an arithmetic operation, such as ADD, finds its second operand in an independently specified register and stores its result in a second, possibly identical, independently specified register.

A zero-address instruction can be defined as one where the location of all relevant operands is specified by convention, so that no operand need be designated explicitly. Zero-address logical and arithmetic operations are available in computers like the English Electric KDF 9 or the Burroughs B 5000, in which all operands required as input to an arithmetic operation are previously stored in a group of temporary storage registers with last-in-first-out properties, variously known as a *nesting store*, a *stack* or a *pushdown store*. Furthermore, the result of an arithmetic operation on operands in the pushdown store is left in a register in the pushdown store from which it may immediately be used for further zero-address arithmetic operations. For instance, three-address operations, such as addition, perform the addition operation on the two top registers of the pushdown store, reduce the size of the pushdown store by deleting the top register, and store the result in the new top register (previously the second register), where it is available for immediate use for subsequent computation.

By means of a pushdown store it is possible to specify arithmetic and logical operations without explicit reference to an operand. However, a machine which is truly a zero-address machine, requires elimination of references to operands for all operations, including data transmission operations typified by FETCH and STORE. This is accomplished in the Burroughs B 5000 by channelling all references to operands through an operand directory known as the "Program Reference Table." Since grouped data, such as arrays, need be specified only by a single "data descriptor" in such a directory, the number of bits required to reference such a directory will be smaller than the number of bits required to reference the memory as a whole. An indirect addressing technique of this kind eliminates the need to refer to operands explicitly in terms of the memory location which they occupy, so that data transmission instructions of this kind may, in some sense, be regarded as zero-address instructions.

The principal remaining class of memory-address references is that of labels. Labelling, and transfers to labels within a program, may also be dealt with by a program directory technique. A machine like the Burroughs B 5000, which references operands and labels through a program directory, may therefore be regarded as a zero-address computer.

Zero-address computers are more economical in their utilization of memory space for programs than one-address computers, since it is unnecessary to provide space in an instruction for referencing a memory location. For instance, the B 5000 has 12-bit instructions and permits four instructions to a computer word.

Furthermore, the fact that arithmetic and logical operations in such an instruction code are "pure" operations, unencumbered by operands, gives rise to a closer correspondence between mathematical source language and basic machine code than is the case in one-address computers. The constituent + in a mathematical source language has a precise counterpart in the target language; and, in general, source language constituents retain their identity in the target language, although the order of their appearance may be changed. This correspondence leads to simpler and faster translation programs than in the case of one-address machines.

To sum up, there are three principal advantages in a zero-address basic machine code:

1. Machine language instructions can be short, since no explicit reference to operands in a large random-access memory is required.
2. The structure of the machine language is close to the structure of mathematical source languages, leading to fast translation procedures.
3. Execution of sequences of arithmetic and logical operations is speeded up since the number of references to random-access memory is reduced.

In view of these advantages, computers with zero-address arithmetic and logical machine instructions operating through a pushdown store have probably come to stay. The case for zero-address operations for all classes of basic machine instructions is not quite as compelling. However, a consistent zero-address instruction code, permitting only indirect references to locations in the random-access memory, has a great deal to be said for it.