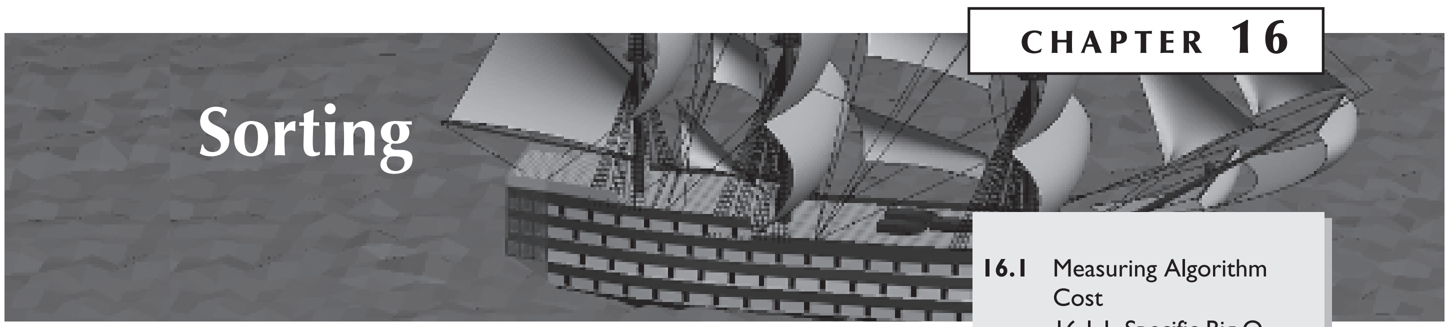
CHAPTER 16

Sorting



Chapter Objectives

This chapter discusses:

- A technique for comparing the performance of algorithms
- A range of algorithms for sorting a collection of data
- Application areas in which these algorithms are most appropriate

First, we will digress from the main thread of problem solving to discuss an "engineering algebra" for measuring the cost of an algorithm in terms of the amount of work done. Then we will consider a number of sorting algorithms, using this technique to assess their relative merits.

16.1 Measuring Algorithm Cost

How many times do you ask yourself, “Just how good is my algorithm?” Probably not very often, if ever. After all, we have been creating relatively simple programs that work on a small, finite set of data. Our functions execute and return an answer within a second or two (except for the recursive Fibonacci function on numbers over 25). You may have noticed that some of the image processing scripts take a number of seconds to run. However, as the problems become more complex and the volume of data increases, we need to consider whether we are solving the problem in the most efficient manner. In extreme cases, processes that manipulate huge amounts of data like the inventory of a large warehouse or a national telephone directory might be possible only with highly efficient algorithms.

Big O is an algebra that permits us to express how the amount of work done in solving a problem relates to the amount of data being processed. It is a gross simplification for software engineering analysis purposes, based on some sound but increasingly complex theory.

Technical Insight 16.1

Interested readers should look up little-O, Big-Ω, little-ω, and Big-Θ.

Big O is a means of estimating the worst case performance of a given algorithm when presented with a certain number of data items, usually referred to as N. In fact, the actual process attempts to determine the limit of the relationship between the work done by an algorithm and N as N approaches infinity.

We report the Big O of an algorithm as O (<expression in terms of N>). For example, O(1) describes the situation where the computing cost is independent of the size of the data, O(N) describes the situation where the computing cost is directly proportional to the size of the data, and O(2^N) describes the situation where the computing cost doubles each time one more piece of data is added. At this point, we should also observe some simplifying assumptions:

- We are not concerned with constant multipliers on the Big O of an algorithm. As rapidly as processor performance and languages are improving, multiplicative improvements can be achieved merely by acquiring the latest hardware or software. Big O is a concept that reports qualitative algorithm improvement. Therefore, we choose to ignore constant multipliers on Big O analyses.
- We are concerned with the performance of algorithms as N approaches infinity. Consequently, when the Big O is expressed as the sum of multiple terms, we keep only the term with the fastest growth rate.

16.1.1 Specific Big O Examples

On the basis of algorithms we have already discussed, we will look at examples of the most common Big O cases.

O(1)—Independent of N O(1) describes the ideal case of an algorithm or logical step whose amount of work is independent of the amount of data. The most obvious example is accessing or modifying an entry in a vector. Since all good languages permit direct access to elements of a vector, the work of these simple operations is independent of the size of the vector.

O(N)—Linear with N O(N) describes an algorithm whose performance is linearly related to N. Copying a cell array of size N is an obvious example, as is searching for a specific piece of data in such a cell array. One might argue that occasionally one would find the data as the first element. There is an equal chance that we would be unlucky and find the item as the last element. On average, we would claim that the performance of this search is the mean of these numbers: (N+1) / 2. However, applying the simplification rules above, we first reject the 1 as being N to a lower power, leaving N/2, and then reject the constant multiplier, leaving O(N) for a linear search.

O(logN)—Binary Search Consider searching for a number—say, 86—in a sorted vector such as that shown in Figure 16.1. One could use a linear search without taking advantage of the ordering of the data. However, a better algorithm might be as follows:

1. Go to the middle of the vector (approximately) and compare that element (59) to the number being sought.
2. If this is the desired value, exit with the answer.
3. If the number sought is less than that element, since the data are ordered, we can reject the half of the array to the right of, and including the 59.
4. Similarly, if the number sought is greater than that element, we can reject the half of the array to the left of and including the 59.
5. Repeat these steps with the remaining half vector until either the number is found or the size of the remaining half is zero.

Now consider how much data can be covered by each test—a measure of the work done as shown in Table 16.1.

In general, we can state that the relationship is expressed as follows:

```
N = 2^W
```

7	12	42	59	71	86	104	212

Figure 16.1 *Binary search*

Table 16.1 Work done in a binary search

Work	N
1	**2**
2	**4**
3	**8**
4	**16**
5	**32**
.	.
.	.
W	2^W

However, we need the expression for the work, W, as a function of N. Therefore, we take the log base 2 of each side so that:

```
W = log₂N
```

Now, we realize that we can convert $\log_2 N$ to $\log_x N$ merely by multiplying by $\log_2 x$, a constant that we are allowed to ignore. Consequently, we lose interest in representing the specific base of the logarithm, leaving the work for a binary search as O(log N).

O(N^2)—Proportional to N^2 O(N^2) describes an algorithm whose performance is proportional to the square of N. It is a special case of O(N × M), which describes any operation on an N × M array or image.

O(2^N)—Exponential Growth or Worse Occasionally we run across very nasty implementations of simple algorithms. For example, consider the recursive implementation of the Fibonacci algorithm we discussed in Section 9.6.2. In this implementation, fib(N) = fib(N − 1) + fib(N − 2). So each time we add another term, the previous two terms have to be calculated again, thereby doubling the amount of work. If we double the work when 1 is added to N, in general the Big O is O(2^N). Of course, in the case of this particular algorithm, there is a simple iterative solution with a much preferable performance of O(N).

16.1.2 Analyzing Complex Algorithms

We can easily calculate the Big O of simple algorithms. For more complex algorithms, we determine the Big O by breaking the complex algorithm into simpler abstractions, as we saw in Chapter 10. We would continue that process until the abstractions can be characterized as simple operations on defined collections for which we can determine their Big Os. The Big O of the overall algorithm is then determined from the individual components by combining them according to the following rules:

- If two components are sequential (do A and then do B), you add their Big O expressions
- If components are nested (for each A, do B), you multiply their Big O expressions

For example, we will see the merge sort algorithm in Section 16.2.5. It can be abstracted as follows:

> Perform a binary division of the data (O(logN)) and *then for each* binary step (of which there are O(log(N)), merge all the data items (O(N)).

This has the general form:

> Do A, then for each B, do C

which, according to the rules above, should result in $O_A + O_B * O_C$. The overall algorithm therefore costs O(log N) + O(N) * O(log N). We remove the first term because its growth is slower, leaving O(N log N) as the overall algorithm cost.

16.2 Algorithms for Sorting Data

Generally, sorting a collection of data will organize the data items in such a way that it is possible to search for a specific item using a binary search rather than a linear search. This concept is nice in principle when dealing with simple collections like an array of numbers. However, it is more difficult in practice with real data. For example, telephone books are always sorted by the person's last name. This facilitates searching by last name, but it does not help if you are looking for the number of a neighbor whose name you do not know. That search would require sorting the data by street name.

There are many methods for sorting data. We present five representative samples selected from many sorting algorithms because each has a practical, engineering application. First we describe each algorithm, and then we compare their performance and suggest engineering circumstances in which you would apply each algorithm. Note that in all these algorithms, the comparisons are done using functions (e.g., `gt(...)`, `lt(...)`, or `equals(...)`) rather than mathematical operators. This permits collections containing arbitrarily complex objects to be sorted merely by customizing the comparison functions.

16.2.1 Insertion Sort

Insertion sort is perhaps the most obvious sorting technique. Given the original collection of objects to sort, it begins by initializing an empty

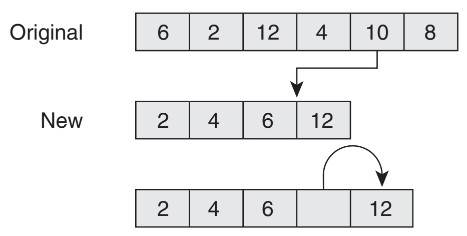


Figure 16.2 *Insertion sort in progress*

collection. For example, if the collection were a vector, you might allocate a new vector of the same size and initialize an "output index" to the start of that vector. Then the algorithm traverses the original vector, inserting each object from that vector in order into the new vector. This usually requires "shuffling" the objects in the new vector to make room for the new object.

Figure 16.2 illustrates the situation where the first four numbers of the original vector have been inserted into the new vector; the algorithm finds the place to insert the next number (10) and then moves the 12 across to make space for it.

Listing 16.1 shows the MATLAB code for insertion sort on a vector of numbers. The algorithm works for any data collection for which the function `lt(A,B)` compares two instances.

Listing 16.1 The insertion sort function

```
function b = insertionsort(a)
% This function sorts a column vector,
% using the insertion sort algorithm
    b = []; i = 1; sz = length(a);
    while i <= sz
        b = insert(b, a(i,1) );
        i = i + 1;
    end
end
function a = insert(a, v)
% insert the value v into column vector a
    i = 1; sz = length(a); done = false;
    while i <= sz
        if lt(v, a(i,1))
            done = true;
            a = [a(1:i-1); v; a(i:end)];
            break;
        end
        i = i + 1;
    end
    if ~done
        a(sz+1, 1) = v;
    end
end
```

In Listing 16.1:

> Line 2: Initializes the result and the `while` loop parameters.
>
> Line 4: Calls the helper function to insert the latest value into the output vector.
>
> Lines 8–21: The helper function that inserts a new value into a vector and returns that vector.

Later we will refer to the selection sort algorithm that is similar in concept to insertion sort. Rather than sorting as the new data are put into the new vector, however, the selection sort algorithm repeatedly finds and deletes the smallest item in the original vector and puts it directly into the new vector.

Both insertion sort and selection sort are $O(N^2)$ if used to sort a whole vector.

16.2.2 Bubble Sort

Where insertion sort is easy to visualize, it is normally implemented by creating a new collection and growing that new collection as the algorithm proceeds. Bubble sort is conceptually the easiest sorting technique to visualize and is usually accomplished by rearranging the items in a collection in place. Given the original collection of N objects to sort, it makes $(N - 1)$ major passes through the data. The first major pass examines all N objects in a minor pass, and subsequent passes reduce the number of examinations by 1. On each minor pass through the data, beginning with the first data item and moving incrementally through the data, the algorithm checks to see whether the next item is smaller than the current one. If so, the two items are swapped in place in the array.

At the end of the first major pass, the largest item in the collection has been moved to the end of the collection. After each subsequent major pass, the largest remaining item is found at the end of the remaining items. The process repeats until on the last major pass, the first two items are compared and swapped if necessary. Figure 16.3 illustrates a bubble sort of a short vector. On the first pass, the value 98 is moved completely across the vector to the rightmost position. On the next pass, the 45 is moved into position. On the third pass, the 23 reaches the right position, and the last pass finishes the sort.

Listing 16.2 shows the MATLAB code for bubble sort on a vector of numbers. The algorithm works for any data type for which the function `gt(A,B)` compares two instances. Since bubble sort performs $(N - 1) * (N - 1)/2$ comparisons on the data, it is also $O(N^2)$. Some implementations use a flag to determine whether any swaps occurred on the last major pass and terminate the algorithm if none occurred. However, the efficiency

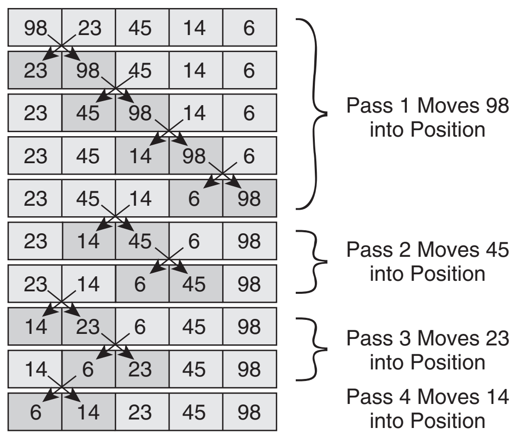


Figure 16.3 *Bubble sort*

Listing 16.2 Bubble sort

```
 1. function bubblesort()
    %  This function sorts the column array b in place,
    %  using the bubble sort algorithm
 2.     global b
 3.     N = length(b);
 4.     right = N-1;
 5.     for in = 1:(N-1)
 6.         for jn = 1:right
 7.             if gt(b(jn), b(jn+1))
 8.                 tmp = b(jn); % swap b(jn) with b(jn+1)
 9.                 b(jn) = b(jn+1);
10.                 b(jn+1) = tmp;
11.             end
12.         end
13.         right = right - 1;
14.     end
15. end
```

gained by stopping the algorithm early has to be weighed against the cost of setting and testing a flag whenever a swap is accomplished.

In Listing 16.2:

Line 2: In order to be able to access the array in place, we pass it as a global variable instead of as a parameter.

Lines 3–4: Since each pass puts the largest element in place, we can reduce the item count by 1 each time. This initializes the size of the first pass.

Lines 4–14: Show the loop for the $N - 1$ major passes.

Lines 6–12: Show the loop for each major pass.

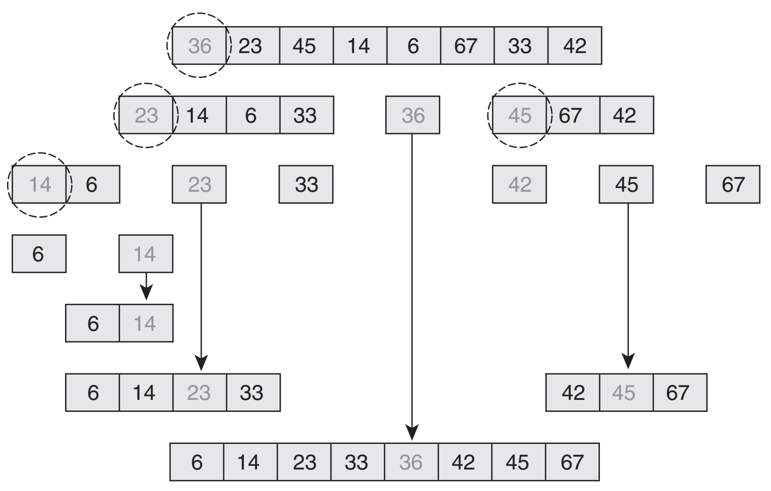


Figure 16.4 *Quick sort*

Lines 8–10: Swap the current item with its neighbor. By doing this in place, the largest item is always considered the current item until it reaches the end.

Line 13: Shortens the row after each major pass because the largest item in the last pass is placed at the right-hand end.

16.2.3 Quick Sort

As its name suggests, the quick sort algorithm is one of the fastest sorting algorithms. Like Bubble Sort, it is designed to sort an array "in place." The quick sort algorithm is recursive and uses an elegant approach to subdividing the original vector. Figure 16.4 illustrates this process. The algorithm proceeds as follows:

- The terminating condition occurs when the vector is of length 1, which is obviously sorted.
- A "pivot point" is then chosen. Some sophisticated versions go to a significant amount of effort to calculate the most effective pivot point. We are content to choose the first item in the vector.
- The vector is then subdivided by moving all of the items less than the pivot to its left and all those greater than the pivot to its right, thereby placing the pivot in its final location in the resulting vector.
- The items to the left and right of the pivot are then recursively sorted by the same algorithm.
- The algorithm always converges because these two halves are always shorter than the original vector.

Listing 16.3 shows the code for the quick sort algorithm. The partitioning algorithm looks a little messy, but it is just performing the

Listing 16.3 Quick sort

```
 1. function a = quicksort(a, from, to)
    % This function sorts a column array,
    % using the quick sort algorithm
 2.     if from < to
 3.         [a p] = partition(a, from, to);
 4.         a = quicksort(a, from, p);
 5.         a = quicksort(a, p + 1, to);
 6.     end
 7. end
 8. function [a lower] = partition(a, from, to)
    % This function partitions a column array
 9.     pivot = a(from); i = from - 1; j = to + 1;
10.     while i < j
11.         i = i + 1;
12.         while lt(a(i), pivot)
13.             i = i + 1;
14.         end
15.         j = j - 1;
16.         while gt(a(j), pivot)
17.             j = j - 1;
18.         end
19.         if (i < j)
20.             temp = a(i); % this section swaps
21.             a(i) = a(j); % a(i) with a(j)
22.             a(j) = temp;
23.         end
24.     end
25.     lower = j;
26. end
27.
```

array adjustments. It starts with `i` and `j` outside the vector to the left and right. Then it keeps moving each toward the middle as long as the values at `i` and `j` are on the proper side of the pivot. When this process stops, `i` and `j` are the indices of data items that are out of order. They are swapped, and the process is repeated until `i` crosses past `j`. Quick sort is O(N log N). As with the previous techniques, this algorithm applies to collections of any data type for which the functions `lt(A,B)` and `gt(A,B)` compare two instances.

In Listing 16.3:

Line 1: Each recursive call is provided with the vector to sort and the range of indices to sort. These are initially `from = 1` and `to = length(a)`.

Line 2: The terminating condition for the recursion is when the vector to sort has size 1—that is, when `from == to`.

Line 3: The partition function performs three roles—it places the pivot in the right place, moves the smaller and larger values to the correct sides, and returns the location of the pivot to permit the recursive partitioning.

Lines 4 and 5: Show recursive calls to sort the left and right parts of the vector.

Lines 8–26: Show the helper function.

Line 9: Initializes the variables.

Lines 10–24: The outer loop continues until `i` passes `j`.

Lines 11–14: Skip `i` forward over all the items less than the pivot.

Lines 15–18: Skip `j` backward over all the elements greater than the pivot.

Lines 20–22: If `i < j`, `i` is indexing an item greater than the pivot, and `j` is indexing an item less than the pivot. By swapping the contents of `a(i)` and `a(j)`, we rectify both inequities and can continue the inner loop.

Line 25: When the loop exits, both `i` and `j` are indexing the pivot.

There is one performance caution about quick sort. Its speed depends on the randomness of the data. If the data are mostly sorted, its performance reduces to O(N^2).

16.2.4 Merge Sort

Merge sort is another O(N log N) algorithm that achieves speed by dividing the original vector into two "equal" halves. It is difficult at best to perform a merge sort in place in a collection. Equality, of course, is not possible when there is an odd number of objects to be sorted, in which case the length of the "halves" will differ by at most 1. The heart of the merge sort algorithm is the technique used to reunite two smaller sorted vectors. This function is called "merge." Its objective is to merge two vectors that have been previously sorted. Since the two vectors are sorted, the smallest object can only be at the front of one of these two vectors. The smallest item is removed from its place and added to the result vector. This merge process continues until one of the two halves is empty, in which case the remaining half (whose values all exceed those in the result vector) is copied into the result.

The merge sort algorithm is shown in Figure 16.5 and proceeds as follows:

- The terminating condition is a vector with length less than 2, which is, obviously, in order
- The recursive part invokes the merge function on the recursive call to merge the two halves of the vector
- The process converges because the halves are always smaller than the original vector

The code for merge sort is shown in Listing 16.4.

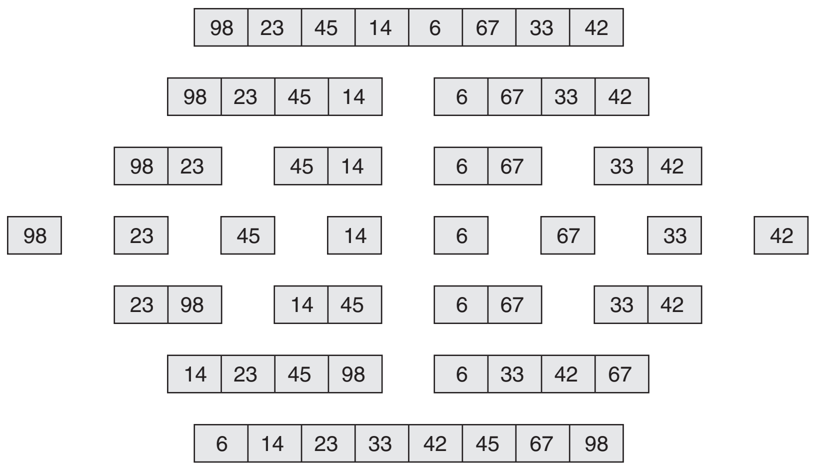


Figure 16.5 *Merge sort*

Listing 16.4 Merge sort

```
function b = mergesort(a)
%  This function sorts a column array,
%  using the merge sort algorithm
   b = a; sz = length(a);
   if sz > 1
        szb2 = floor(sz / 2);
        first = mergesort(a(1 : szb2));
        second = mergesort(a(szb2+1 : sz));
        b = merge(first, second);
    end
end
function b = merge(first, second)
%   Merges two sorted arrays
    i1 = 1; i2 = 1; out = 1;
    % as long as neither i1 nor i2 past the end,
    % move the smaller element into a
    while (i1 <= length(first)) & (i2 <= length(second))
        if lt(first(i1), second(i2))
            b(out,1) = first(i1); i1 = i1 + 1;
        else
            b(out,1) = second(i2); i2 = i2 + 1;
        end
        out = out + 1;
    end
    % copy any remaining entries of the first array
    while i1 <= length(first)
        b(out,1) = first(i1); i1 = i1 + 1; out = out + 1;
    end
    % copy any remaining entries of the second array
    while i2 <= length(second)
        b(out,1) = second(i2); i2 = i2 + 1; out = out + 1;
    end
end
```

In Listing 16.4:

Line 2: Initializes the parameters.

Line 3: The terminating condition is an array of length 1, which does not need sorting.

Line 4: Divides the array in half.

Lines 5 and 6: Sort the halves of the array.

Line 7: Merges the two sorted halves.

Lines 10–26: Show the helper function to merge sorted arrays.

Lines 12–19: This loop repeats until one of the two arrays is used up choosing and removing the smaller element out of the two arrays.

Lines 20–25: Copy the remains of each array to the result.

16.2.5 Radix Sort

A discussion of sorting techniques would not be complete without discussing radix sort, commonly referred to as bucket sort. This is also an O(N log N) algorithm whose most obvious application is for sorting physical piles of papers, such as students' test papers. However, the same principle can be applied to sorting successively on the units, tens and hundreds digit of numbers (hence, the term radix sort). The process begins with a stack of unsorted papers, each with an identifier consisting of a number or a unique name. One pass is made through the stack separating the papers into piles based on the first digit or character of the identifier. Subsequent passes sort each of these piles by subsequent characters or digits until all the piles have a small number of papers that can be sorted by insertion or selection sorts. The piles can then be reassembled in order. Figure 16.6 illustrates the situation at the end of the second sorting pass when piles for the first digit have also been separated by the second digit.

There are a number of reasons why this technique is popular for sorting:

- There is a minimal amount of "paper shuffling" or bookkeeping
- The base of the logarithm in the O(N log N) is either 10 (numerical identifier) or 26 (alphabetic identifier), thereby providing a "constant multiplier" speed advantage
- Once the first sorting pass is complete, one can use multi-processing (in the form of extra people) to perform the remaining passes in parallel, thereby reducing the effective performance to O(N) (given sufficient parallel resources)

72 DPI M16_SMIT8708_03_SE_C16.indd Page 380 02/02/12 7:50 PM user-f396 F-396 EQA

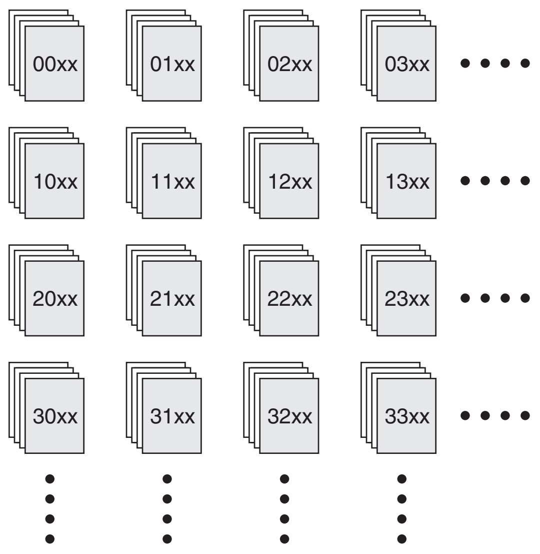


Figure 16.6 *Radix sort*

16.3 Performance Analysis

In order to perform a comparison of the performance of different algorithms, a script was written to perform each sort on a vector of increasing length containing random numbers. The script started with a length of 4 and continued doubling the length until it reached 262,144 (2^{18}). To obtain precise timing measurements, each sort technique was repeated a sufficient number of times to obtain moderately accurate timing measurements with the internal millisecond clock. In order to eliminate common computation costs, it was necessary to measure the overhead cost of the loops themselves and subtract that time from the times of each sort algorithm. Note that in order to show the results of the system internal sort on the same chart, its execution time was multiplied by 1,000.

Figure 16.7 shows a typical plot of the results of this analysis, illustrating the relative power of O(N log N) algorithms versus O(N^2) algorithms. The plot on a log-log scale shows the relative time taken by the selection sort, insertion sort, bubble sort, merge sort, quick sort, and quick sort in place algorithms, together with the internal sort function. Also on the chart are plotted trend lines for O(N^2) and O(N log N) processes. We can make the following observations from this chart:

- Since the scales are each logarithmic, it is tempting to claim that there is “not much difference” between O(N^2) and O(N log N) algorithms. Looking closer, however, it is clear that for around 200,000 items, the O(N^2) sorts are around 100,000 times slower than the O(N log N) algorithms.

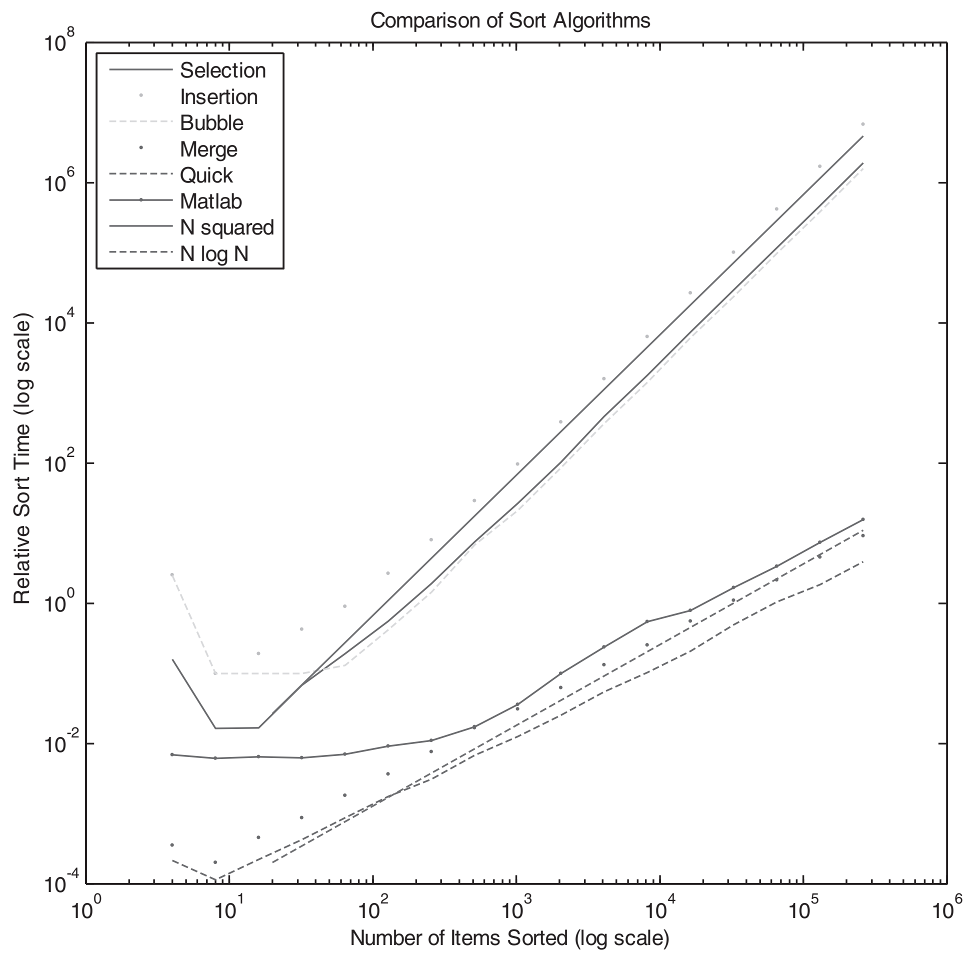


Figure 16.7 *Sort study results*

- The performance of most of the algorithms is extremely erratic below 100 items. If you are sorting small amounts of data, the algorithm does not matter.
- The selection sort, bubble sort, and insertion sort algorithms clearly demonstrate $O(N^2)$ behavior.
- The merge sort and quick sort algorithms seem to demonstrate O(N log N). Notice, however, that the performance of quick sort is slightly better than O(N log N). This slight improvement is due to the fact that once the pivot has been moved, it is in the right place and is eliminated from further sorting passes.
- Clearly, the internal sort function, in addition to being 1,000 times faster than any of the coded algorithms, is closely tracking the O(N log N) performance curve, indicating that it is programmed

with one of the many algorithms that use divide-and-conquer to sort the data as efficiently as possible.

16.4 Applications of Sorting Algorithms

This section discusses the circumstances under which you might choose to use one or another of the sorting algorithms presented above. We assert here without proof that the theoretical lower bound of sorting is O(N log N). Consequently, we should not be looking for a generalized sorting algorithm that improves on this performance. However, within those constraints, there are circumstances under which each of the sorting techniques performs best. As we saw in the analysis above, the internal sort function is blindingly fast and should be used whenever possible. The subsequent paragraphs show the applicability and limitations of the other sort algorithms if you cannot use `sort(...)`.

16.4.1 Using sort(. . .)

The first and most obvious question is why one would not always use the built-in `sort(...)` function. Clearly, whenever that function works, you should use it. Its applicability might seem at first glance to be limited to sorting numbers in an array, and you will come across circumstances when you need to sort more complex items. You might, for example, have a structure array of addresses and telephone numbers that you wish to sort by last name, first name, or telephone number. In this case, it seems that the internal sort program does not help, and you have to create your own sort.

Extracting and Sorting Vectors and Cell Arrays However, a closer examination of the specification of the sort function allows us to generalize the application of `sort(...)` significantly. When you call `sort(v)`, it actually offers you a second result that contains the indices used to sort `v`. So in the case where you have a cell array or a structure array and your sort criteria can be extracted into a vector, you can sort that vector and use the second result, the indexing order, to sort the original array. Furthermore, if you can extract character string data into a cell array of strings, the internal sort function will sort that cell array alphabetically.

For example, consider again the CD collection from Chapter 10. We might want to find the most expensive CD in our collection and then make a list of artists and titles ordered alphabetically by artist. We leave the details of this as an exercise for the reader.

16.4.2 Insertion Sort

Insertion sort is the fastest means of performing incremental sorting. If a small number of new items—say, M—are being added to a sorted collection of size N, the process will be O(M*N), which will be fastest as long as M < log N. For example, consider a national telephone directory with over a billion numbers that must frequently be updated with new listings. Adding a small number of entries (< 20) would be faster with insertion sort than with merge sort, and quick sort would be a disaster because the data are almost all sorted (see below).

16.4.3 Bubble Sort

Bubble sort is the simplest in-place sort to program and is fine for small amounts of data. The major advantage of bubble sort is that in a fine-grained multi-processor environment, if you have N/2 processors available with access to the original data, you can reduce the Big O to O(N).

16.4.4 Quick Sort

As its name suggests, this is the quickest of the sorting algorithms and should normally be used for a full sort. However, it has one significant disadvantage: its performance depends on a fairly high level of randomness in the distribution of the data in the original array. If there is a significant probability that your original data might be already sorted, or partially sorted, your quick sort is not going to be quick. You should use merge sort.

16.4.5 Merge Sort

Since its algorithm does not depend on any specific characteristics of the data, merge sort will always turn in a solid O(N log N) performance. You should use it whenever you suspect that quick sort might get in trouble.

16.4.6 Radix Sort

It is theoretically possible to write the radix sort algorithm to attempt to take advantage of its apparent performance improvements over the more conventional algorithms shown above. However, some practical problems arise:

- In practice, the manipulation of the arrays of arrays necessary to sort by this technique is quite complex
- The performance gained for manual sorts by "parallel processing" stacks using multiple people cannot be realized
- The logic for extracting the character or digit for sorting is going to detract from the overall performance

Therefore, absent some serious parallel processing machines, we recommend that the use of bucket sort be confined to manually sorting large numbers of physical objects.

16.5 Engineering Example—A Selection of Countries

In the Engineering Application problem in Section 10.5, we attempted to find the best country for a business expansion based on the rate of growth of the GNP for that country versus its population growth. The initial version of the program returned the suggestion that the company should move to Equatorial Guinea. However, when this was presented to the Board of Directors, it was turned down, and you were asked to bring them a list of the best 20 places to give them a good range of selection.

We should make two changes to the algorithm:

- Originally, we used a crude approximation to determine the slope of the population and GNP curves. However, now we know that `polyfit` can perform this slope computation accurately, and we will substitute that computation.
- We will use the internal sort function to find the 20 best countries. The code to accomplish this, a major revision of the code in Chapter 10, is shown in Listing 16.5.

Listing 16.5 Updated world data analysis

```
function doit
    worldData = buildData('World_data.xls');
    n = 20;
    bestn = findBestn(worldData, n);
    fprintf('best %d countries are:\n', n)
    for best = bestn(end:-1:1)
        fprintf('%s\n', worldData(best).name)
    end
end
function bestn = findBestn(worldData, n)
% find the indices of the n best countries
% according to the criterion in the function fold
% we first map world data to add the field growth
    for ndx = 1:length(worldData)
        cntry = worldData(ndx);
        worldData(ndx).growth = fold(cntry);
    end
    % now, sort on this criterion
    values = [worldData.growth];
    [junk order] = sort(values);
    % filter these to keep the best 10
    bestn = order(end-n+1:end);
end
function ans = fold(st)
% s1 is the rate of growth of population
    pop = st.pop(~isnan(st.pop));
    yr = st.year(~isnan(st.pop));
    s1 = slope(yr, pop)/mean(pop);
```

continued on next page

```
        % s2 is the rate of growth of the GDP
23.     gdp = st.gdp(~isnan(st.gdp));
24.     yr = st.year(~isnan(st.gdp));
25.     s2 = slope(yr, gdp)/mean(gdp);
        % Measure of merit is how much faster
        % the gdp grows than the population
26.     ans = s2 - s1;
27. end
28. function sl = slope(x, y)
    % Estimate the slope of a curve
29.     if length(x) == 0 || x(end) == x(1)
30.         error('bad data')
31.     else
32.         coef = polyfit(x, y, 1);
33.         sl = coef(1);
34.     end
35. end
```

In Listing 16.5:

Line 1: Wraps the script in a pseudo-function to allow the helper functions to reside in the same file.

Line 2: Reads in the world data.

Line 3: Selects the number of countries to present.

Line 4: Calls the function that will return the indices of the n best countries.

Lines 5–8: Print the list of country names in reverse order (the best first).

Lines 10–18: Show an updated version of the original function to return the indices of the n best countries.

Lines 11–14: Map the `worldData` structure array, adding to each a field called `growth` that contains the criterion specified in the `fold` function.

Lines 15 and 16: Extract and sort the values of `growth` for each country.

Line 17: Returns the last n countries that will have the highest `growth` values.

Lines 19–27: The `fold` function unchanged from Chapter 10.

Lines 28–35: The modified `slope` function from Chapter 10.

Lines 32 and 33: Use `polyfit` to compute an accurate slope and return it to the calling function.

The results from running this version are shown in Table 16.2. This seems to be an acceptable list of possibilities to take back to the Board of Directors.

Common Pitfalls 16.1

A deceptively simple question arises: Should you expect the `worldData` at line 6 of Listing 16.7 to contain the field growth? Actually, it will not. Although it appears that the function `findBestn` adds this field to `worldData`, it is working with a copy of the `worldData` structure array that is not returned to the calling script.

Table 16.2 Updated world data results

Best 20 countries are:

Estonia	Lebanon
St. Kitts & Nevis	Malta
Albania	Cyprus
Vietnam	Tajikistan
Croatia	Taiwan
Kazakhstan	Korea, Republic of
Azerbaijan	Grenada
Uzbekistan	Ireland
Georgia	Portugal
Dominica	Antigua

Chapter Summary

This chapter discussed:

- A technique for comparing the performance of algorithms
- A range of useful algorithms for sorting a collection of data
- Application areas in which these algorithms are most appropriate

Self Test

Use the following questions to check your understanding of the material in this chapter:

True or False

1. When computing the Big O of sequential operations, you retain only the term that grows fastest with N.
2. All search algorithms have O(N).
3. No sort algorithm can perform better than O(NlogN).
4. All sorting algorithms with O(N^2) traverse the complete data collection N times.
5. Quick sort in reality should be listed as O(N^2).

Fill in the Blanks

1. ________________ is an algebra that permits us to express how the amount of ___________ done in solving a problem relates to the amount ____________ of being processed.
2. Any algorithm that traverses, maps, folds, or filters a collection is O(__________).
3. __________ sort and _____________ sort perform with O(NlogN).
4. __________ sort and _____________ sort are designed to sort the data in place.
5. The system internal `sort(...)` returns the ___________ and a(n) ___________ that allow you to sort any collection from whose elements one can derive a(n) ___________ or ______________________.