

As an example, let us consider the problem of computing the sum of n numbers a_0, \dots, a_{n-1} . If $n > 1$, we can divide the problem into two instances of the same problem: to compute the sum of the first $\lfloor n/2 \rfloor$ numbers and to compute the sum of the remaining $\lceil n/2 \rceil$ numbers. (Of course, if $n = 1$, we simply return a_0 as the answer.) Once each of these two sums is computed by applying the same method recursively, we can add their values to get the sum in question:

$$a_0 + \dots + a_{n-1} = (a_0 + \dots + a_{\lfloor n/2 \rfloor - 1}) + (a_{\lfloor n/2 \rfloor} + \dots + a_{n-1}).$$

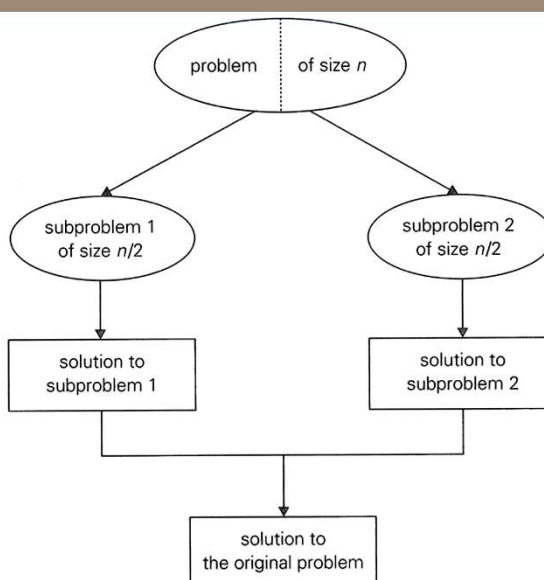


FIGURE 5.1 Divide-and-conquer technique (typical case).

As mentioned above, in the most typical case of divide-and-conquer a problem's instance of size n is divided into two instances of size $n/2$. More generally, an instance of size n can be divided into b instances of size n/b , with a of them needing to be solved. (Here, a and b are constants; $a \geq 1$ and $b > 1$.) Assuming that size n is a power of b to simplify our analysis, we get the following recurrence for the running time $T(n)$:

$$T(n) = aT(n/b) + f(n), \quad (5.1)$$

where $f(n)$ is a function that accounts for the time spent on dividing an instance of size n into instances of size n/b and combining their solutions. (For the sum example above, $a = b = 2$ and $f(n) = 1$.)

Recurrence (5.1) is called the *general divide-and-conquer recurrence*. Obviously, the order of growth of its solution $T(n)$ depends on the values of the constants a and b and the order of growth of the function $f(n)$. The efficiency analysis of many divide-and-conquer algorithms is greatly simplified by the following theorem (see Appendix B).

Master Theorem

If $f(n) \in \Theta(n^d)$ where $d \geq 0$ in recurrence (5.1), then

$$T(n) \in \begin{cases} \Theta(n^d) & \text{if } a < b^d, \\ \Theta(n^d \log n) & \text{if } a = b^d, \\ \Theta(n^{\log_b a}) & \text{if } a > b^d. \end{cases}$$

Analogous results hold for the O and Ω notations, too.

For example, the recurrence for the number of additions $A(n)$ made by the divide-and-conquer sum-computation algorithm (see above) on inputs of size $n = 2^k$ is

$$A(n) = 2A(n/2) + 1.$$

Thus, for this example, $a = 2$, $b = 2$, and $d = 0$; hence, since $a = b^d$,

$$A(n) \in \Theta(n^{\log_b a}) = \Theta(n^{\log_2 2}) = \Theta(n).$$

5.1 Mergesort

ALGORITHM *Mergesort* ($A[0..n-1]$)

//Sorts array $A[0..n-1]$ by recursive mergesort

//Input: An array $A[0..n-1]$ of orderable elements

//Output: Array $A[0..n-1]$ sorted in nondecreasing order

if $n > 1$

 copy $A[0.. \lfloor n/2 \rfloor - 1]$ to $B[0.. \lfloor n/2 \rfloor - 1]$

 copy $A[\lfloor n/2 \rfloor .. n - 1]$ to $C[0.. \lceil n/2 \rceil - 1]$

Mergesort($B[0.. \lfloor n/2 \rfloor - 1]$)

Mergesort($C[0.. \lceil n/2 \rceil - 1]$)

Merge(B, C, A) //see below

The *merging* of two sorted arrays can be done as follows. Two pointers (array indices) are initialized to point to the first elements of the arrays being merged. The elements pointed to are compared, and the smaller of them is added to a new array being constructed; after that, the index of the smaller element is incremented to point to its immediate successor in the array it was copied from. This operation is repeated until one of the two given arrays is exhausted, and then the remaining elements of the other array are copied to the end of the new array.

ALGORITHM *Merge*($B[0..p-1]$, $C[0..q-1]$, $A[0..p+q-1]$)
 //Merges two sorted arrays into one sorted array
 //Input: Arrays $B[0..p-1]$ and $C[0..q-1]$ both sorted
 //Output: Sorted array $A[0..p+q-1]$ of the elements of B and C
 $i \leftarrow 0; j \leftarrow 0; k \leftarrow 0$
 while $i < p$ and $j < q$ do
 if $B[i] \leq C[j]$
 $A[k] \leftarrow B[i]; i \leftarrow i + 1$
 else $A[k] \leftarrow C[j]; j \leftarrow j + 1$
 $k \leftarrow k + 1$
 if $i = p$
 copy $C[j..q-1]$ to $A[k..p+q-1]$
 else copy $B[i..p-1]$ to $A[k..p+q-1]$

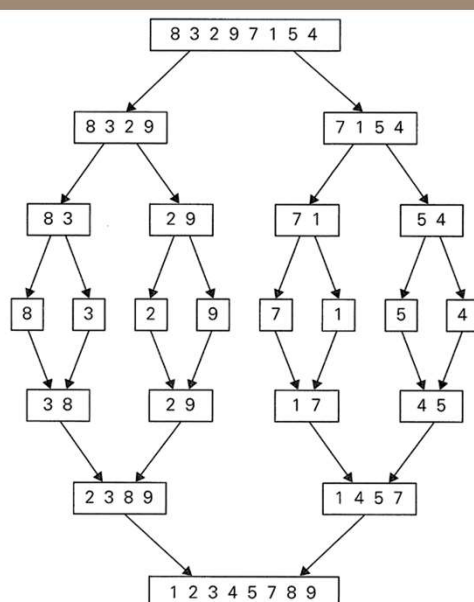


FIGURE 5.2 Example of mergesort operation.

How efficient is mergesort? Assuming for simplicity that n is a power of 2, the recurrence relation for the number of key comparisons $C(n)$ is

$$C(n) = 2C(n/2) + C_{\text{merge}}(n) \quad \text{for } n > 1, C(1) = 0.$$

Let us analyze $C_{\text{merge}}(n)$, the number of key comparisons performed during the merging stage. At each step, exactly one comparison is made, after which the total number of elements in the two arrays still needing to be processed is reduced by 1. In the worst case, neither of the two arrays becomes empty before the other one contains just one element (e.g., smaller elements may come from the alternating arrays). Therefore, for the worst case, $C_{\text{merge}}(n) = n - 1$, and we have the recurrence

$$C_{\text{worst}}(n) = 2C_{\text{worst}}(n/2) + n - 1 \quad \text{for } n > 1, C_{\text{worst}}(1) = 0.$$

Hence, according to the Master Theorem, $C_{\text{worst}}(n) \in \Theta(n \log n)$ (why?). In fact, it is easy to find the exact solution to the worst-case recurrence for $n = 2^k$:

$$C_{\text{worst}}(n) = n \log_2 n - n + 1.$$

There are two main ideas leading to several variations of mergesort. First, the algorithm can be implemented bottom up by merging pairs of the array's elements, then merging the sorted pairs, and so on. (If n is not a power of 2, only slight bookkeeping complications arise.) This avoids the time and space overhead of using a stack to handle recursive calls. Second, we can divide a list to be sorted in more than two parts, sort each recursively, and then merge them together. This scheme, which is particularly useful for sorting files residing on secondary memory devices, is called *multiway mergesort*.

5.2 Quicksort

Quicksort is the other important sorting algorithm that is based on the divide-and-conquer approach. Unlike mergesort, which divides its input elements according to their position in the array, quicksort divides them according to their values. We already encountered this idea of an array partition in Section 4.5, where we discussed the selection problem. A *partition* is an arrangement of the array's elements so that all the elements to the left of some element $A[s]$ are less than or equal to $A[s]$, and all the elements to the right of $A[s]$ are greater than or equal to it:

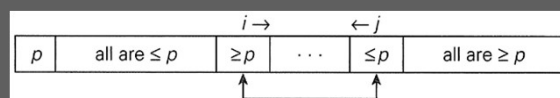
$$\underbrace{A[0] \dots A[s-1]}_{\text{all are } \leq A[s]} \quad A[s] \quad \underbrace{A[s+1] \dots A[n-1]}_{\text{all are } \geq A[s]}$$

ALGORITHM *Quicksort*($A[l..r]$)

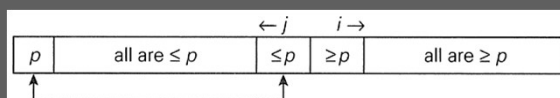
//Sorts a subarray by quicksort

//Input: Subarray of array $A[0..n-1]$, defined by its left and// right indices l and r //Output: Subarray $A[l..r]$ sorted in nondecreasing orderif $l < r$ $s \leftarrow \text{Partition}(A[l..r])$ // s is a split position*Quicksort*($A[l..s-1]$)*Quicksort*($A[s+1..r]$)

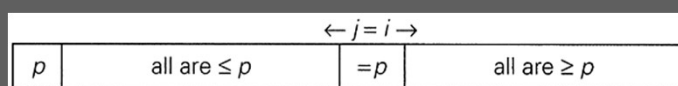
After both scans stop, three situations may arise, depending on whether or not the scanning indices have crossed. If scanning indices i and j have not crossed, i.e., $i < j$, we simply exchange $A[i]$ and $A[j]$ and resume the scans by incrementing i and decrementing j , respectively:



If the scanning indices have crossed over, i.e., $i > j$, we will have partitioned the subarray after exchanging the pivot with $A[j]$:



Finally, if the scanning indices stop while pointing to the same element, i.e., $i = j$, the value they are pointing to must be equal to p (why?). Thus, we have the subarray partitioned, with the split position $s = i = j$:



ALGORITHM *HoarePartition*($A[l..r]$)

```
//Partitions a subarray by Hoare's algorithm, using the first
// element as a pivot
//Input: Subarray of array  $A[0..n - 1]$ , defined by its left and
// right indices  $l$  and  $r$  ( $l < r$ )
//Output: Partition of  $A[l..r]$ , with the split position returned
// as this function's value
 $p \leftarrow A[l]$ 
 $i \leftarrow l; j \leftarrow r + 1$ 
```

```

repeat
  repeat  $i \leftarrow i + 1$  until  $A[i] \geq p$ 
  repeat  $j \leftarrow j - 1$  until  $A[j] \leq p$ 
  swap( $A[i], A[j]$ )
until  $i \geq j$ 
swap( $A[i], A[j]$ ) //undo last swap when  $i \geq j$ 
swap( $A[l], A[j]$ )
return  $j$ 

```

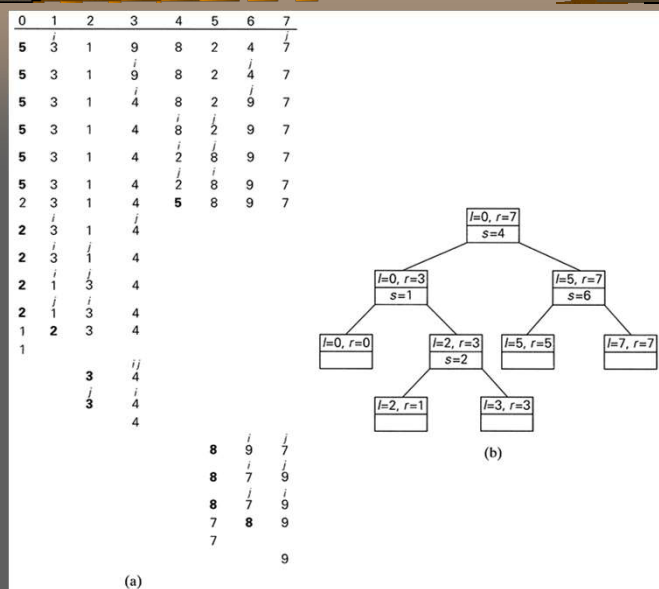


FIGURE 5.3 Example of quicksort operation. (a) Array's transformations with pivots shown in bold. (b) Tree of recursive calls to *Quicksort* with input values l and r of subarray bounds and split position s of a partition obtained.

We start our discussion of quicksort's efficiency by noting that the number of key comparisons made before a partition is achieved is $n + 1$ if the scanning indices cross over and n if they coincide (why?). If all the splits happen in the middle of corresponding subarrays, we will have the best case. The number of key comparisons in the best case satisfies the recurrence

$$C_{best}(n) = 2C_{best}(n/2) + n \quad \text{for } n > 1, C_{best}(1) = 0.$$

According to the Master Theorem, $C_{best}(n) \in \Theta(n \log_2 n)$.

In the worst case, all the splits will be skewed to the extreme: one of the two subarrays will be empty, and the size of the other will be just 1 less than the size of the subarray being partitioned. This unfortunate situation will happen, in particular, for increasing arrays, i.e., for inputs for which the problem is Already solved! Indeed, if $A[0..n - 1]$ is a strictly increasing array and we use $A[0]$ as the pivot, the left-to-right scan will stop on $A[1]$ while the right-to-left scan will go all the way to reach $A[0]$, indicating the split at position 0:

$\leftarrow j$	$i \rightarrow$		
$A[0]$	$A[1]$	\dots	$A[n-1]$

So, after making $n + 1$ comparisons to get to this partition and exchanging the pivot $A[0]$ with itself, the algorithm will be left with the strictly increasing array $A[1..n - 1]$ to sort. This sorting of strictly increasing arrays of diminishing sizes will continue until the last one $A[n - 2..n - 1]$ has been processed. The total Number of key comparisons made will be equal to

$$C_{\text{worst}}(n) = (n + 1) + n + \dots + 3 = \frac{(n + 1)(n + 2)}{2} - 3 \in \Theta(n^2).$$

Thus, the question about the utility of quicksort comes down to its averagecase behavior. Let $C_{\text{avg}}(n)$ be the average number of key comparisons made by quicksort on a randomly ordered array of size n . A partition can happen in any position s ($0 \leq s \leq n - 1$) after $n + 1$ comparisons are made to achieve the partition. After the partition, the left and right subarrays will have s and $n - 1 - s$ elements, respectively. Assuming that the partition split can happen in each position s with the same probability $1/n$, we get the following recurrence relation:

$$C_{\text{avg}}(n) = \frac{1}{n} \sum_{s=0}^{n-1} [(n + 1) + C_{\text{avg}}(s) + C_{\text{avg}}(n - 1 - s)] \quad \text{for } n > 1,$$

$$C_{\text{avg}}(0) = 0, \quad C_{\text{avg}}(1) = 0.$$

Its solution, which is much trickier than the worst- and best-case analyses, turns out to be

$$C_{avg}(n) \approx 2n \ln n \approx 1.39n \log_2 n.$$

Because of quicksort's importance, there have been persistent efforts over the years to refine the basic algorithm. Among several improvements discovered by researchers are:

- better pivot selection methods such as *randomized quicksort* that uses a random element or the *median-of-three* method that uses the median of the leftmost, rightmost, and the middle element of the array
- switching to insertion sort on very small subarrays (between 5 and 15 elements for most computer systems) or not sorting small subarrays at all and finishing the algorithm with insertion sort applied to the entire nearly sorted array

- modifications of the partitioning algorithm such as the three-way partition into segments smaller than, equal to, and larger than the pivot (see Problem 9 in this section's exercises)