Merge Sort

By: Dr. Surjeet Kumar

Dept. of Computer Application

Source: "Introduction to Algorithms" PHI 3rd Edition

by Thomas H. Cormen & Others.

The divide-and-conquer approach

Many useful algorithms are ***recursive*** in structure: to solve a given problem, they call themselves recursively one or more times to deal with closely related subproblems. These algorithms typically follow a ***divide-and-conquer*** approach: they break the problem into several subproblems that are similar to the original problem but smaller in size, solve the subproblems recursively, and then combine these solutions to create a solution to the original problem.

The divide-and-conquer paradigm involves three steps at each level of the recursion:

Divide the problem into a number of subproblems that are smaller instances of the same problem.

Conquer the subproblems by solving them recursively. If the subproblem sizes are small enough, however, just solve the subproblems in a straightforward manner.

Combine the solutions to the subproblems into the solution for the original problem.

The ***merge sort*** algorithm closely follows the divide-and-conquer paradigm. Intuitively, it operates as follows.

Divide: Divide the n-element sequence to be sorted into two subsequences of $n/2$ elements each.

Conquer: Sort the two subsequences recursively using merge sort.

Combine: Merge the two sorted subsequences to produce the sorted answer.

The recursion "bottoms out" when the sequence to be sorted has length 1, in which case there is no work to be done, since every sequence of length 1 is already in sorted order.

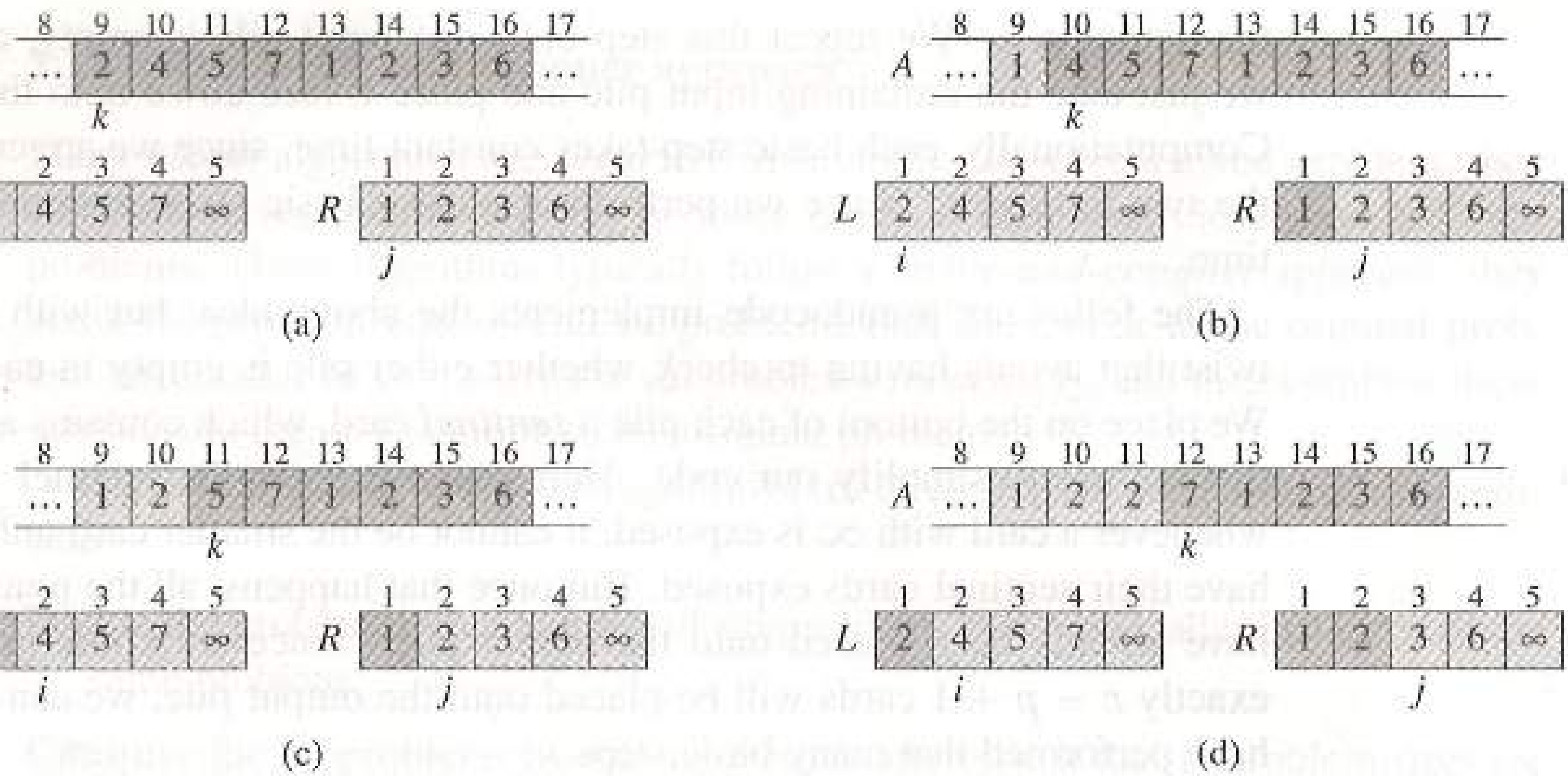
The key operation of the merge sort algorithm is the merging of two sorted sequences in the "combine" step. We merge by calling an auxiliary procedure MERGE(A, p, q, r), where A is an array and p, q, and r are indices into the array such that $p \leq q < r$. The procedure assumes that the subarrays $A[p \,..\, q]$ and $A[q + 1 \,..\, r]$ are in sorted order. It ***merges*** them to form a single sorted subarray that replaces the current subarray $A[p \,..\, r]$.

Our MERGE procedure takes time $\Theta(n)$, where $n = r - p + 1$ is the total number of elements being merged, and it works as follows. Returning to our card-playing motif, suppose we have two piles of cards face up on a table. Each pile is sorted, with the smallest cards on top. We wish to merge the two piles into a single sorted output pile, which is to be face down on the table. Our basic step consists of choosing the smaller of the two cards on top of the face-up piles, removing it from its pile (which exposes a new top card), and placing this card face down onto utput pile. We repeat this step until one input pile is empty, at which time ıst take the remaining input pile and place it face down onto the output pile. putationally, each basic step takes constant time, since we are comparing just wo top cards. Since we perform at most n basic steps, merging takes $\Theta(n)$

e following pseudocode implements the above idea, but with an additional that avoids having to check whether either pile is empty in each basic step. lace on the bottom of each pile a ***sentinel*** card, which contains a special value we use to simplify our code. Here, we use ∞ as the sentinel value, so that ever a card with ∞ is exposed, it cannot be the smaller card unless both piles their sentinel cards exposed. But once that happens, all the nonsentinel cards already been placed onto the output pile. Since we know in advance that ly $r - p + 1$ cards will be placed onto the output pile, we can stop once we performed that many basic steps.

```
GE(A, p, q, r)
  n1 = q - p + 1
  n2 = r - q
  let L[1 .. n1 + 1] and R[1 .. n2 + 1] be new arrays
  for i = 1 to n1
      L[i] = A[p + i - 1]
  for j = 1 to n2
      R[j] = A[q + j]
  L[n1 + 1] = ∞
  R[n2 + 1] = ∞
  i = 1
  j = 1
  for k = p to r
      if L[i] ≤ R[j]
          A[k] = L[i]
          i = i + 1
      else A[k] = R[j]
          j = j + 1
```

detail, the MERGE procedure works as follows. Line 1 computes the length n_1 ne subarray $A[p\,..\,q]$, and line 2 computes the length n_2 of the subarray $+ 1\,..\,r]$. We create arrays L and R ("left" and "right"), of lengths $n_1 + 1$ $n_2 + 1$, respectively, in line 3; the extra position in each array will hold the nel. The **for** loop of lines 4–5 copies the subarray $A[p\,..\,q]$ into $L[1\,..\,n_1]$. the **for** loop of lines 6–7 copies the subarray $A[q + 1\,..\,r]$ into $R[1\,..\,n_2]$. s 8–9 put the sentinels at the ends of the arrays L and R. Lines 10–17, illus-



8 9 10 11 12 13 14 15 16 17
… 2 4 5 7 1 2 3 6 …
k
2 3 4 5
4 5 7 ∞
R 1 2 3 6 ∞
j
(a)
A … 1 4 5 7 1 2 3 6 …
k
L 2 4 5 7 ∞
i
R 1 2 3 6 ∞
j
(b)
… 1 2 5 7 1 2 3 6 …
k
4 5 7 ∞
i
R 1 2 3 6 ∞
j
(c)
A … 1 2 2 7 1 2 3 6 …
k
L 2 4 5 7 ∞
i
R 1 2 3 6 ∞
j
(d)

Figure: The operation of lines 10–17 in the call MERGE$(A, 9, 12, 16)$, when the subarray 9..16] contains the sequence $\langle 2, 4, 5, 7, 1, 2, 3, 6 \rangle$. After copying and inserting sentinels, the ay L contains $\langle 2, 4, 5, 7, \infty \rangle$, and the array R contains $\langle 1, 2, 3, 6, \infty \rangle$. Lightly shaded positions A contain their final values, and lightly shaded positions in L and R contain values that have yet be copied back into A. Taken together, the lightly shaded positions always comprise the values ginally in $A[9..16]$, along with the two sentinels. Heavily shaded positions in A contain values t will be copied over, and heavily shaded positions in L and R contain values that have already en copied back into A. **(a)–(h)** The arrays A, L, and R, and their respective indices k, i, and j or to each iteration of the loop of lines 12–17.

ted in Figure, perform the $r - p + 1$ basic steps by maintaining the following op invariant:

At the start of each iteration of the **for** loop of lines 12–17, the subarray $A[p..k-1]$ contains the $k - p$ smallest elements of $L[1..n_1 + 1]$ and $R[1..n_2 + 1]$, in sorted order. Moreover, $L[i]$ and $R[j]$ are the smallest elements of their arrays that have not been copied back into A.

We must show that this loop invariant holds prior to the first iteration of the **for** op of lines 12–17, that each iteration of the loop maintains the invariant, and at the invariant provides a useful property to show correctness when the loop rminates.

itialization: Prior to the first iteration of the loop, we have $k = p$, so that the subarray $A[p..k-1]$ is empty. This empty subarray contains the $k - p = 0$ smallest elements of L and R, and since $i = j = 1$, both $L[i]$ and $R[j]$ are the smallest elements of their arrays that have not been copied back into A.

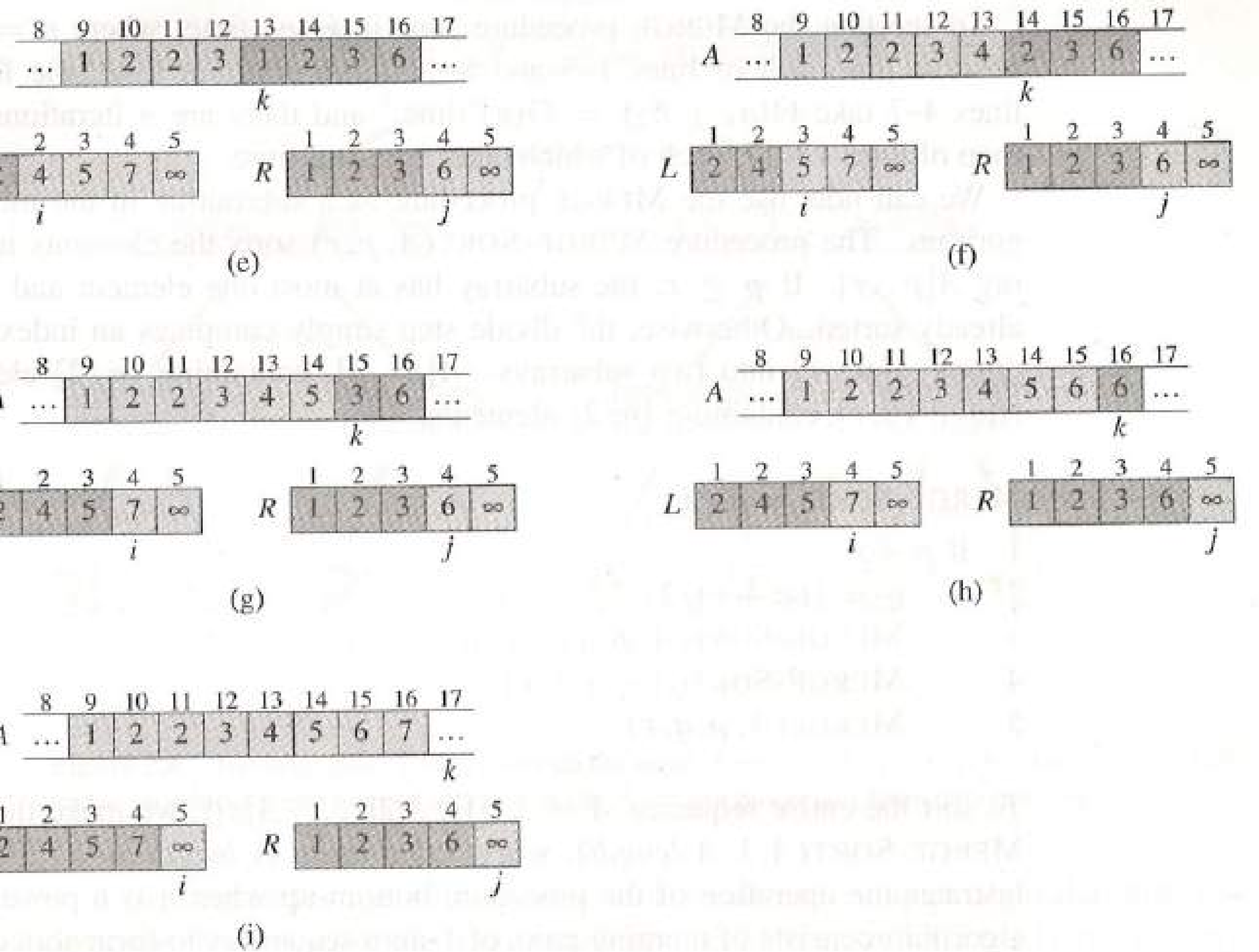

(e)
(f)
(g)
(h)
(i)

Figure , continued (i) The arrays and indices at termination. At this point, the subarray in $A[9..16]$ is sorted, and the two sentinels in L and R are the only two elements in these arrays that have not been copied into A.

Maintenance: To see that each iteration maintains the loop invariant, let us first suppose that $L[i] \leq R[j]$. Then $L[i]$ is the smallest element not yet copied back into A. Because $A[p..k-1]$ contains the $k-p$ smallest elements, after line 14 copies $L[i]$ into $A[k]$, the subarray $A[p..k]$ will contain the $k-p+1$ smallest elements. Incrementing k (in the **for** loop update) and i (in line 15) reestablishes the loop invariant for the next iteration. If instead $L[i] > R[j]$, then lines 16–17 perform the appropriate action to maintain the loop invariant.

Termination: At termination, $k = r+1$. By the loop invariant, the subarray $A[p..k-1]$, which is $A[p..r]$, contains the $k-p = r-p+1$ smallest elements of $L[1..n_1+1]$ and $R[1..n_2+1]$, in sorted order. The arrays L and R together contain $n_1+n_2+2 = r-p+3$ elements. All but the two largest have been copied back into A, and these two largest elements are the sentinels.

To see that the MERGE procedure runs in $\Theta(n)$ time, where $n = r - p + 1$, bserve that each of lines 1–3 and 8–11 takes constant time, the **for** loops of nes 4–7 take $\Theta(n_1 + n_2) = \Theta(n)$ time,[7] and there are n iterations of the **for** oop of lines 12–17, each of which takes constant time.

We can now use the MERGE procedure as a subroutine in the merge sort al-orithm. The procedure MERGE-SORT(A, p, r) sorts the elements in the subar-ay $A[p \,..\, r]$. If $p \geq r$, the subarray has at most one element and is therefore lready sorted. Otherwise, the divide step simply computes an index q that par-tions $A[p \,..\, r]$ into two subarrays: $A[p \,..\, q]$, containing $\lceil n/2 \rceil$ elements, and $[q + 1 \,..\, r]$, containing $\lfloor n/2 \rfloor$ elements.[8]

```
ERGE-SORT(A, p, r)
  if p < r
      q = ⌊(p + r)/2⌋
      MERGE-SORT(A, p, q)
      MERGE-SORT(A, q + 1, r)
      MERGE(A, p, q, r)
```

sort the entire sequence $A = \langle A[1], A[2], \ldots, A[n] \rangle$, we make the initial call ERGE-SORT$(A, 1, A.length)$, where once again $A.length = n$. Figure il-trates the operation of the procedure bottom-up when n is a power of 2. The gorithm consists of merging pairs of 1-item sequences to form sorted sequences length 2, merging pairs of sequences of length 2 to form sorted sequences of ngth 4, and so on, until two sequences of length $n/2$ are merged to form the final rted sequence of length n.

nalyzing divide-and-conquer algorithms

hen an algorithm contains a recursive call to itself, we can often describe its nning time by a ***recurrence equation*** or ***recurrence***, which describes the overall nning time on a problem of size n in terms of the running time on smaller inputs. e can then use mathematical tools to solve the recurrence and provide bounds on e performance of the algorithm.

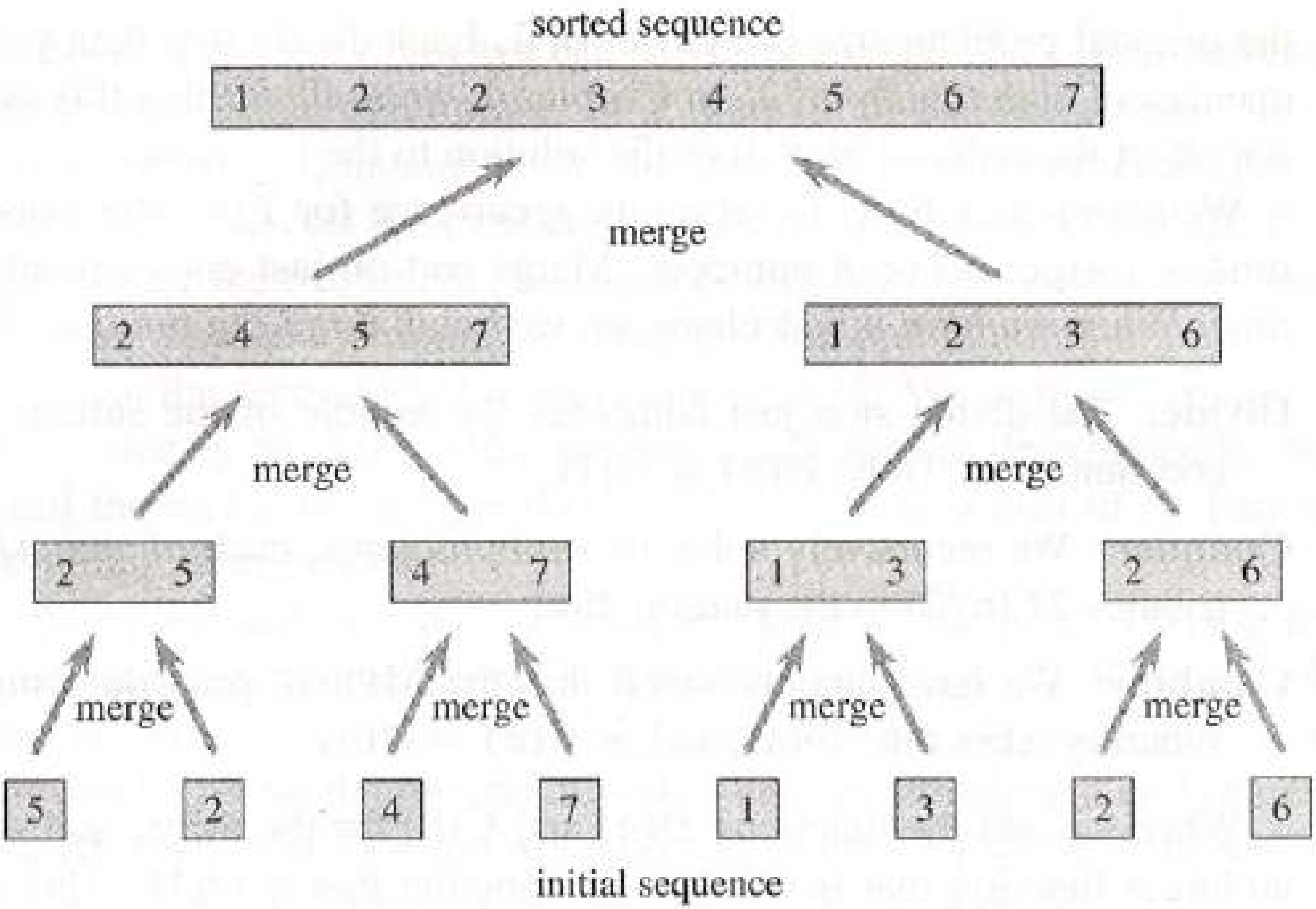


Figure The operation of merge sort on the array $A = \langle 5, 2, 4, 7, 1, 3, 2, 6 \rangle$. The lengths of the sorted sequences being merged increase as the algorithm progresses from bottom to top.

A recurrence for the running time of a divide-and-conquer algorithm falls out
om the three steps of the basic paradigm. As before, we let $T(n)$ be the running
ne on a problem of size n. If the problem size is small enough, say $n \le c$
r some constant c, the straightforward solution takes constant time, which we
ite as $\Theta(1)$. Suppose that our division of the problem yields a subproblems,
ch of which is $1/b$ the size of the original. (For merge sort, both a and b are 2,
t we shall see many divide-and-conquer algorithms in which $a \ne b$.) It takes
ne $T(n/b)$ to solve one subproblem of size n/b, and so it takes time $aT(n/b)$
solve a of them. If we take $D(n)$ time to divide the problem into subproblems
d $C(n)$ time to combine the solutions to the subproblems into the solution to the
iginal problem, we get the recurrence

$$(n) = \begin{cases} \Theta(1) & \text{if } n \le c , \\ aT(n/b) + D(n) + C(n) & \text{otherwise} . \end{cases}$$

nalysis of merge sort

lthough the pseudocode for MERGE-SORT works correctly when the number of
ements is not even, our recurrence-based analysis is simplified if we assume that

iginal problem size is a power of 2. Each divide step then yields two subse-
ces of size exactly $n/2$. we shall see that this assumption does not affect the
of growth of the solution to the recurrence.

reason as follows to set up the recurrence for $T(n)$, the worst-case running
of merge sort on n numbers. Merge sort on just one element takes constant
When we have $n > 1$ elements, we break down the running time as follows.

e: The divide step just computes the middle of the subarray, which takes
nstant time. Thus, $D(n) = \Theta(1)$.

quer: We recursively solve two subproblems, each of size $n/2$, which con-
butes $2T(n/2)$ to the running time.

bine: We have already noted that the MERGE procedure on an n-element
barray takes time $\Theta(n)$, and so $C(n) = \Theta(n)$.

Vhen we add the functions $D(n)$ and $C(n)$ for the merge sort analysis, we are ing a function that is $\Theta(n)$ and a function that is $\Theta(1)$. This sum is a linear ction of n, that is, $\Theta(n)$. Adding it to the $2T(n/2)$ term from the "conquer" gives the recurrence for the worst-case running time $T(n)$ of merge sort:

$$n) = \begin{cases} \Theta(1) & \text{if } n = 1 , \\ 2T(n/2) + \Theta(n) & \text{if } n > 1 . \end{cases}$$

we shall see the "master theorem," which we can use to show $T(n)$ is $\Theta(n \lg n)$, where $\lg n$ stands for $\log_2 n$. Because the logarithm func- grows more slowly than any linear function, for large enough inputs, merge t, with its $\Theta(n \lg n)$ running time, outperforms insertion sort, whose running e is $\Theta(n^2)$, in the worst case.

We do not need the master theorem to intuitively understand why the solution to recurrence is $T(n) = \Theta(n \lg n)$. Let us rewrite recurrence as

$$n) = \begin{cases} c & \text{if } n = 1 , \\ 2T(n/2) + cn & \text{if } n > 1 , \end{cases}$$

ere the constant c represents the time required to solve problems of size 1 as ll as the time per array element of the divide and combine steps.

For convenience, we as-
ne that n is an exact power of 2. Part (a) of the figure shows $T(n)$, which we
pand in part (b) into an equivalent tree representing the recurrence. The cn term
he root (the cost incurred at the top level of recursion), and the two subtrees of
root are the two smaller recurrences $T(n/2)$. Part (c) shows this process carried
e step further by expanding $T(n/2)$. The cost incurred at each of the two sub-
des at the second level of recursion is $cn/2$. We continue expanding each node
he tree by breaking it into its constituent parts as determined by the recurrence,
il the problem sizes get down to 1, each with a cost of c. Part (d) shows the
ulting ***recursion tree***.

Next, we add the costs across each level of the tree. The top level has total
t cn, the next level down has total cost $c(n/2) + c(n/2) = cn$, the level after
t has total cost $c(n/4)+c(n/4)+c(n/4)+c(n/4) = cn$, and so on. In general,
level i below the top has 2^i nodes, each contributing a cost of $c(n/2^i)$, so that
ith level below the top has total cost $2^i\, c(n/2^i) = cn$. The bottom level has n
des, each contributing a cost of c, for a total cost of cn.

The total number of levels of the recursion tree in Figure is $\lg n + 1$, where is the number of leaves, corresponding to the input size. An informal inductive gument justifies this claim. The base case occurs when $n = 1$, in which case the ee has only one level. Since $\lg 1 = 0$, we have that $\lg n + 1$ gives the correct umber of levels. Now assume as an inductive hypothesis that the number of levels a recursion tree with 2^i leaves is $\lg 2^i + 1 = i + 1$ (since for any value of i, e have that $\lg 2^i = i$). Because we are assuming that the input size is a power 2, the next input size to consider is 2^{i+1}. A tree with $n = 2^{i+1}$ leaves has ne more level than a tree with 2^i leaves, and so the total number of levels is $+ 1) + 1 = \lg 2^{i+1} + 1$.

To compute the total cost represented by the recurrence, we simply add up e costs of all the levels. The recursion tree has $\lg n + 1$ levels, each costing cn, r a total cost of $cn(\lg n + 1) = cn \lg n + cn$. Ignoring the low-order term and e constant c gives the desired result of $\Theta(n \lg n)$.

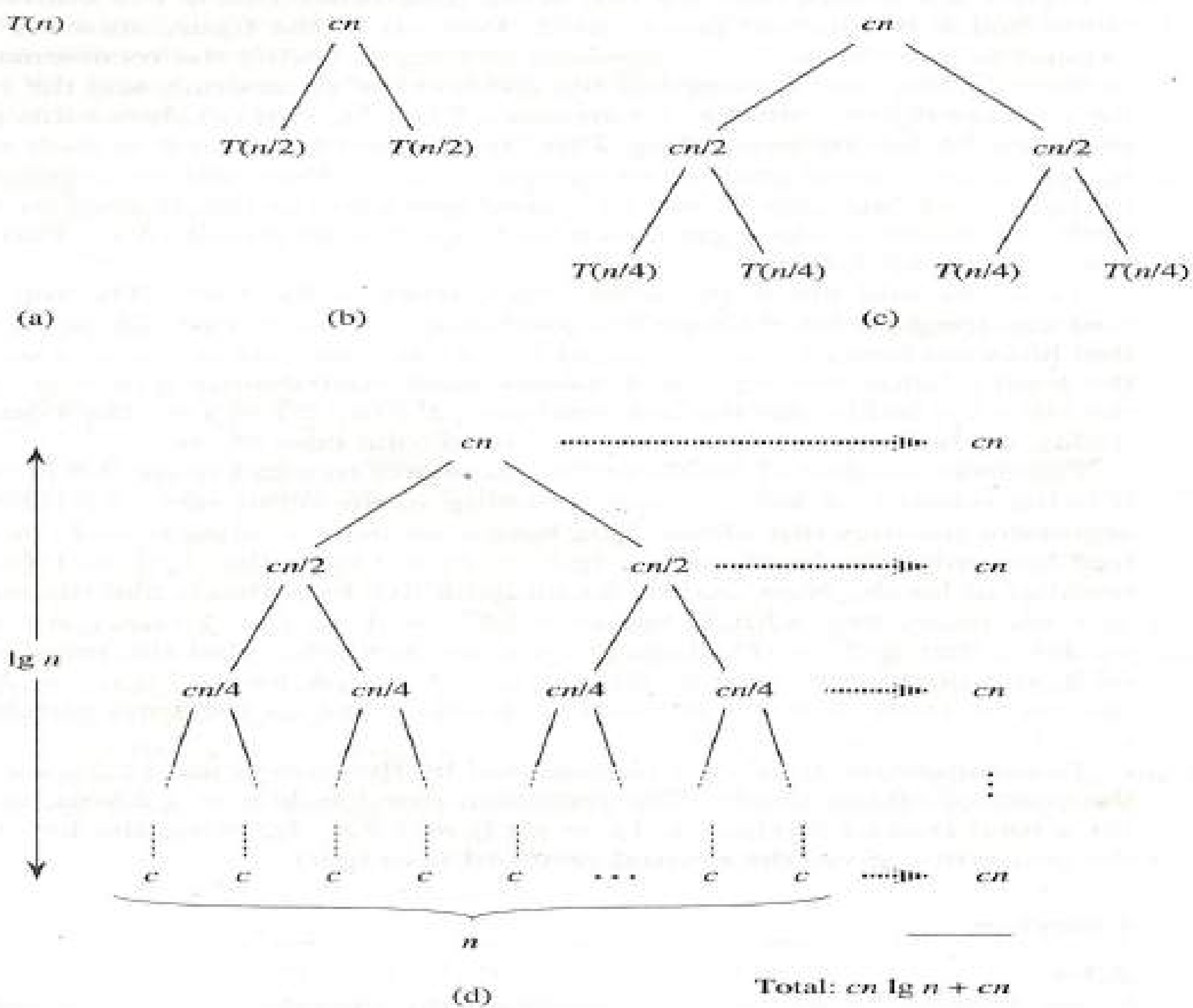


Figure. How to construct a recursion tree for the recurrence $T(n) = 2T(n/2) + cn$. Part (a) shows $T(n)$, which progressively expands in **(b)–(d)** to form the recursion tree. The fully expanded tree in part (d) has $\lg n + 1$ levels (i.e., it has height $\lg n$, as indicated), and each level contributes a total cost of cn. The total cost, therefore, is $cn \lg n + cn$, which is $\Theta(n \lg n)$.