Chapter 2.

Getting Started

Insertion Sort

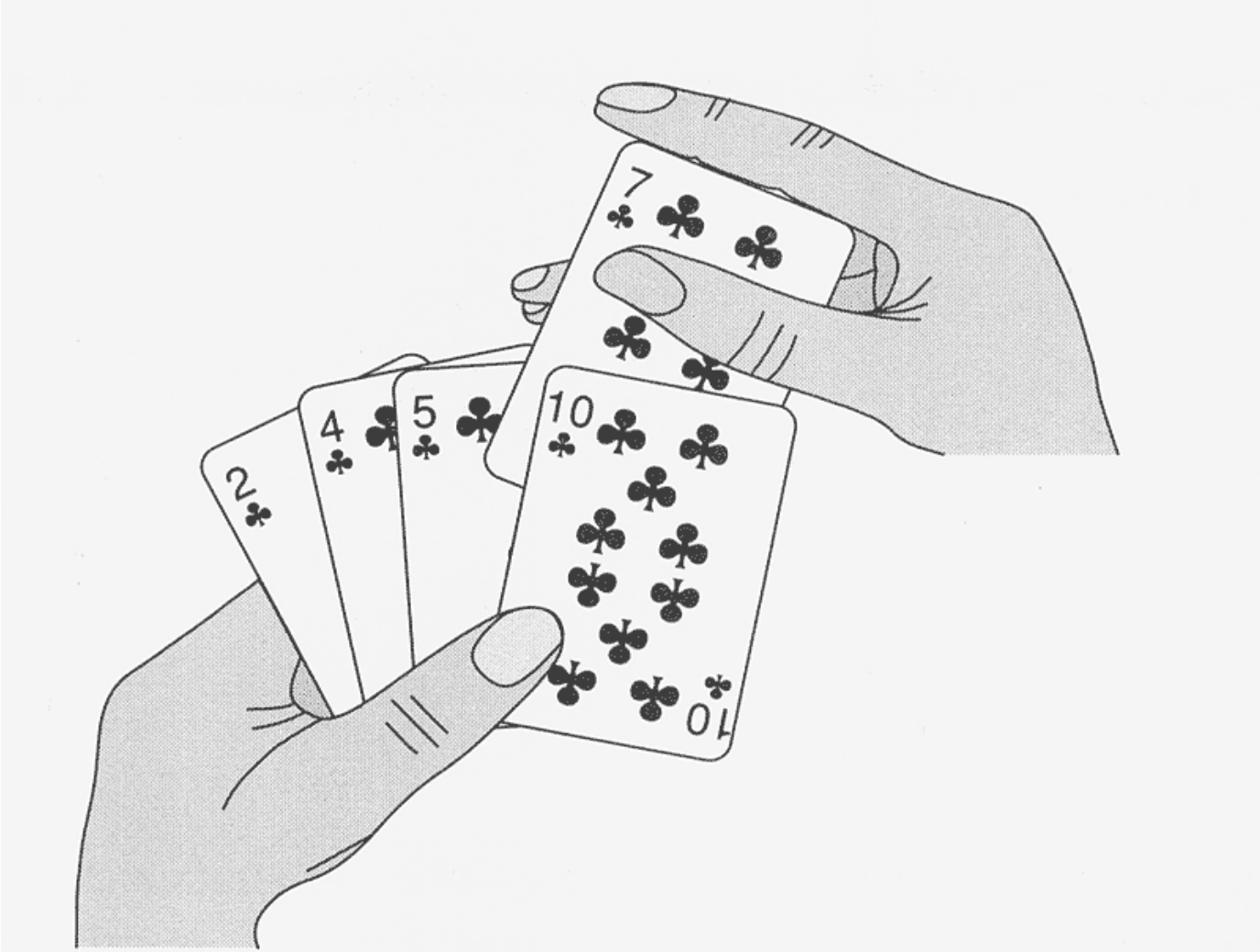

Figure 2.1 Sorting a hand of cards using insertion sort.

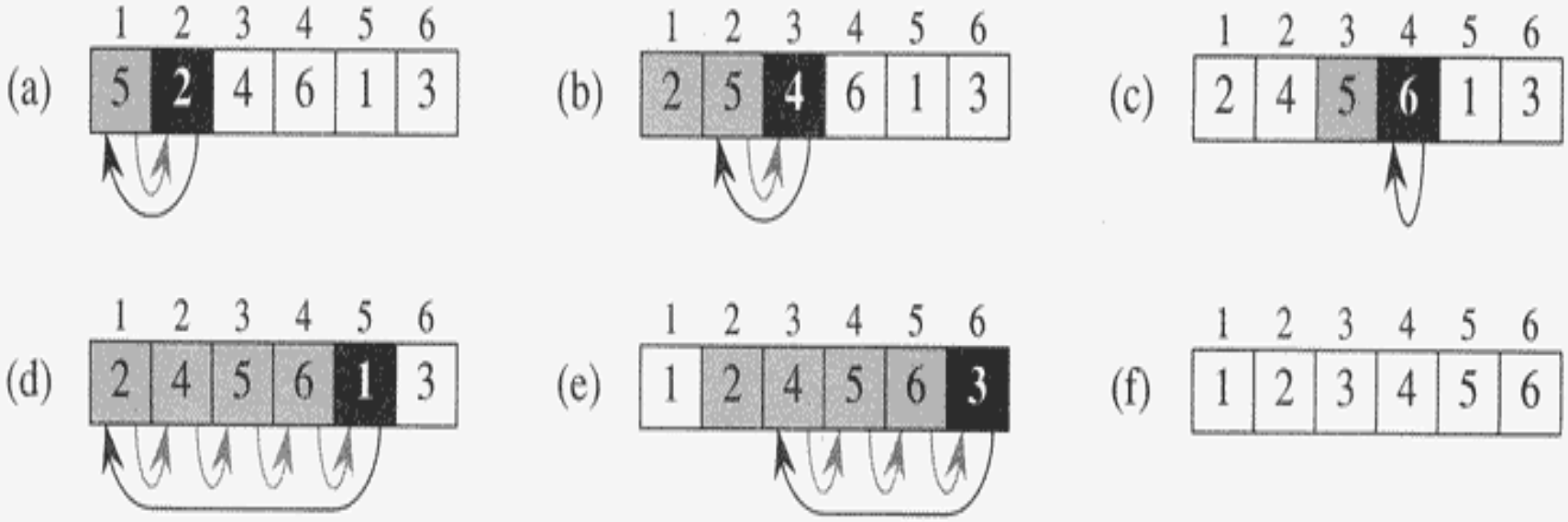

Figure 2.2 The operation of INSERTION-SORT on the array $A = \langle 5, 2, 4, 6, 1, 3 \rangle$. Array indices appear above the rectangles, and values stored in the array positions appear within the rectangles. **(a)–(e)** The iterations of the **for** loop of lines 1–8. In each iteration, the black rectangle holds the key taken from $A[j]$, which is compared with the values in shaded rectangles to its left in the test of line 5. Shaded arrows show array values moved one position to the right in line 6, and black arrows indicate where the key is moved to in line 8. **(f)** The final sorted array.

```
INSERTION-SORT(A)
1  for j ← 2 to length[A]
2       do key ← A[j]
3          ▷ Insert A[j] into the sorted sequence A[1 .. j − 1].
4          i ← j − 1
5          while i > 0 and A[i] > key
6               do A[i + 1] ← A[i]
7                  i ← i − 1
8          A[i + 1] ← key
```

Loop invariants and the correctness of insertion sort

Correctness

Often Use a loop invariant:

***Loop invariant*: At the start of each iteration of the "outer" **for** loop**

-- the loop indexed by j -- the subarray $A[1 \,..\, J\text{-}1]$ consists of the elements originally in $A[1 \,..\, j\text{-}1]$ but in sorted order.

To use a loop invariant to prove correctness, show three things about it:

Initialization:

It is true prior to the first iteration of the loop.

Maintenance:

If it is true before an iteration of the loop, it remains true before the next iteration.

Termination:

When the loop terminates, the invariant—usually along with the reason that the loop terminated—gives us a useful property that helps show that the algorithm is correct.

	INSERTION-SORT(A)	*cost*	*times*
1	**for** $j \leftarrow 2$ **to** $length[A]$	c_1	n
2	**do** $key \leftarrow A[j]$	c_2	$n-1$
3	▷ Insert $A[j]$ into the sorted sequence $A[1 \ldots j-1]$.	0	$n-1$
4	$i \leftarrow j-1$	c_4	$n-1$
5	**while** $i > 0$ and $A[i] > key$	c_5	$\sum_{j=2}^{n} t_j$
6	**do** $A[i+1] \leftarrow A[i]$	c_6	$\sum_{j=2}^{n} (t_j - 1)$
7	$i \leftarrow i-1$	c_7	$\sum_{j=2}^{n} (t_j - 1)$
8	$A[i+1] \leftarrow key$	c_8	$n-1$

Analyzing algorithms

Random-access machine(RAM) model

How do we analyze an algorithm's running time?

The time taken by an algorithm depends on the input

Input size: depends on the problem being studied.

Running time: on a particular input, it is the number of primitive operations(steps) executed.

Analysis of *insertion sort*

The running time of the algorithm is:

$\sum$ **(cost of statement) x (# of times statement is executed)**
all statements

t_j = # of times that while loop test is executed for that value of *j*.

Best case: the array is already sorted (all $t_j = 1$)

Worst case:

the array is in reverse order ($t_j = j$).

The worst case running time gives a guaranteed upper bound on the running time for any input.

Average case:

On average, the key in *A[j]* is less than half the elements in *A[1 .. j-1]* and it's greater than the other half. ($t_j = j/2$).

.. continued

Order of Growth

The *abstraction* to ease analysis and focus on the important features.

Look only at the leading term of the formula for running time.

Drop lower-order terms.

Ignore the constant coefficient in the leading term.

Example: $an^2 + bn + c = \Theta(n^2)$

Drop lower-order terms $\Rightarrow$ an^2

Ignore constant coefficient $\Rightarrow$ n^2

The worst case running time $T(n)$ grows like n^2; it does ***not equal*** n^2.

The running time is $\Theta(n^2)$ to capture the notion that the order of growth is n^2.

We consider one algorithm is ***more efficient*** than another if its worst case running time has a smaller order of growth.

Designing algorithms

Divide and Conquer

Divide the problem into a number of *subproblems.*

Conquer the *subproblems* by solving them recursively.

Base case:

If the subproblems are small enough,
just solve them.

Combine the *subproblem solutions* to give
a solution to the original problem.

Cf.) **Incremental method** – insertion sort.

Merge Sort

A sorting algorithm based on divide and conquer.

The worst-case running time:

merge sort < insertion sort in its order of growth

To sort $A[p \,..\, r]$:

Divide by splitting into two subarrays $A[p \,..\, q]$ and $A[q+1 \,..\, r]$, where q is the halfway point of $A[p \,..\, r]$.

Conquer by recursively sorting the two subarrays $A[p \,..\, q]$ and $A[q+1 \,..\, r]$.

Combine by merging the two sorted subarrays $A[p \,..\, q]$ and $A[q+1 \,..\, r]$ to produce a single sorted subarray $A[p \,..\, r]$.

To accomplish this step, we'll define a procedure

MERGE*(A, p, q, r)*.

Initial call: MERGE-SORT(A, 1, n)

```
MERGE-SORT(A, p, r)
1   if p < r
2       then q ← ⌊(p + r)/2⌋
3            MERGE-SORT(A, p, q)
4            MERGE-SORT(A, q + 1, r)
5            MERGE(A, p, q, r)
```

```
MERGE(A, p, q, r)
 1  n1 ← q − p + 1
 2  n2 ← r − q
 3  create arrays L[1 .. n1 + 1] and R[1 .. n2 + 1]
 4  for i ← 1 to n1
 5        do L[i] ← A[p + i − 1]
 6  for j ← 1 to n2
 7        do R[j] ← A[q + j]
 8  L[n1 + 1] ← ∞
 9  R[n2 + 1] ← ∞
10  i ← 1
11  j ← 1
12  for k ← p to r
13        do if L[i] ≤ R[j]
14              then A[k] ← L[i]
15                   i ← i + 1
16              else A[k] ← R[j]
17                   j ← j + 1
```

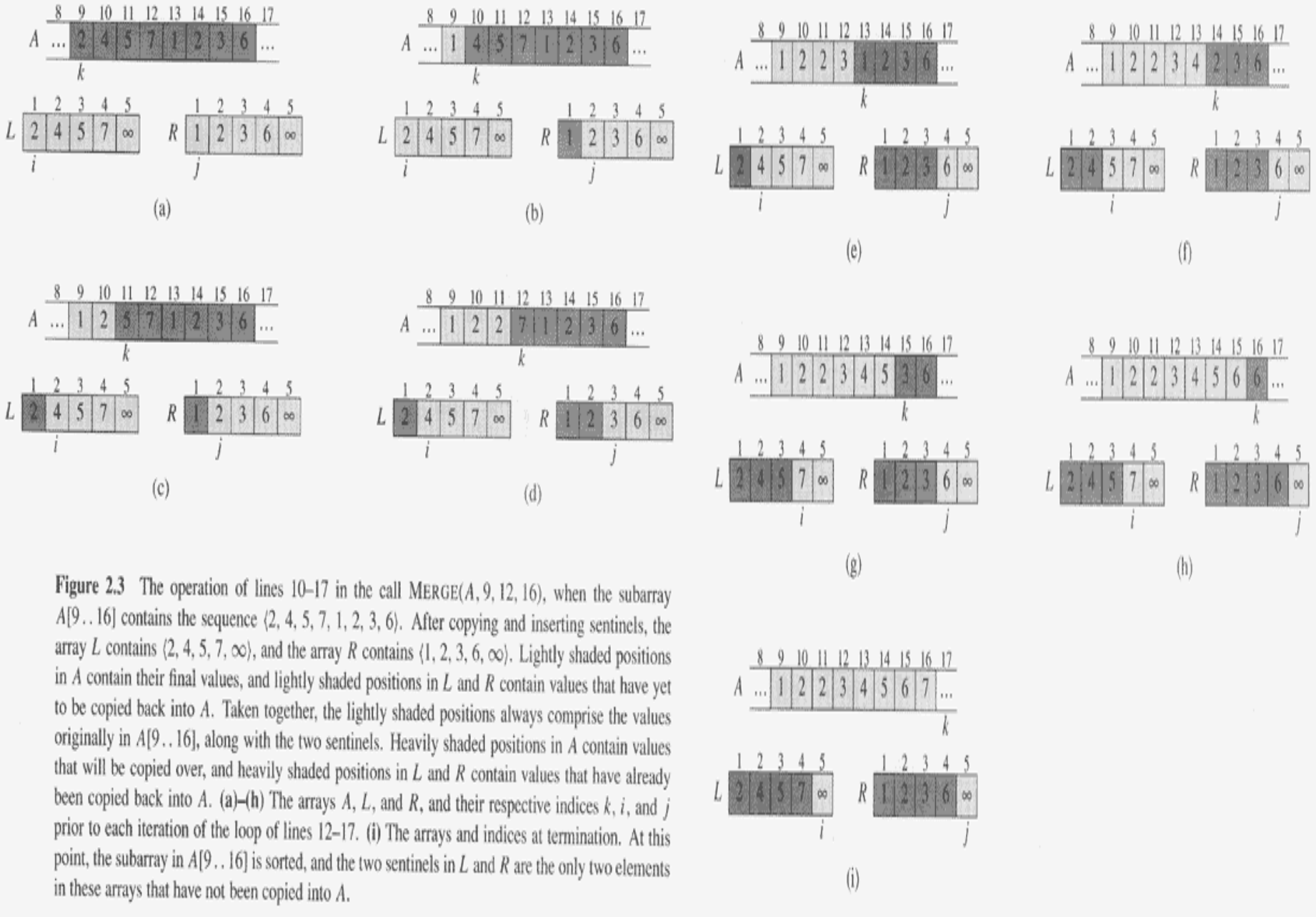

Figure 2.3 The operation of lines 10–17 in the call MERGE(A, 9, 12, 16), when the subarray $A[9..16]$ contains the sequence $\langle 2, 4, 5, 7, 1, 2, 3, 6\rangle$. After copying and inserting sentinels, the array L contains $\langle 2, 4, 5, 7, \infty\rangle$, and the array R contains $\langle 1, 2, 3, 6, \infty\rangle$. Lightly shaded positions in A contain their final values, and lightly shaded positions in L and R contain values that have yet to be copied back into A. Taken together, the lightly shaded positions always comprise the values originally in $A[9..16]$, along with the two sentinels. Heavily shaded positions in A contain values that will be copied over, and heavily shaded positions in L and R contain values that have already been copied back into A. **(a)–(h)** The arrays A, L, and R, and their respective indices k, i, and j prior to each iteration of the loop of lines 12–17. **(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.

Merging: *MERGE(A, p, q, r)*

Input: Array *A* and indices *p, q, r* such that

$p \leq q < r$

Subarray *A[p .. q]* is sorted and subarray *A[q+1 .. r]* is sorted.

By the restrictions on *p, q, r*, neither subarray is empty.

Output: The two subarrays are merge into a single sorted subarray in *A[p .. r].*

T(n) = $\Theta(n)$, where $n=r-p+1$ = the # of elements being merged.

What is ***n***?

The size of a given (sub)problem.

The size of the original problem => the size of a subproblem.

Use this technique when we analyze ***recursive algorithm***

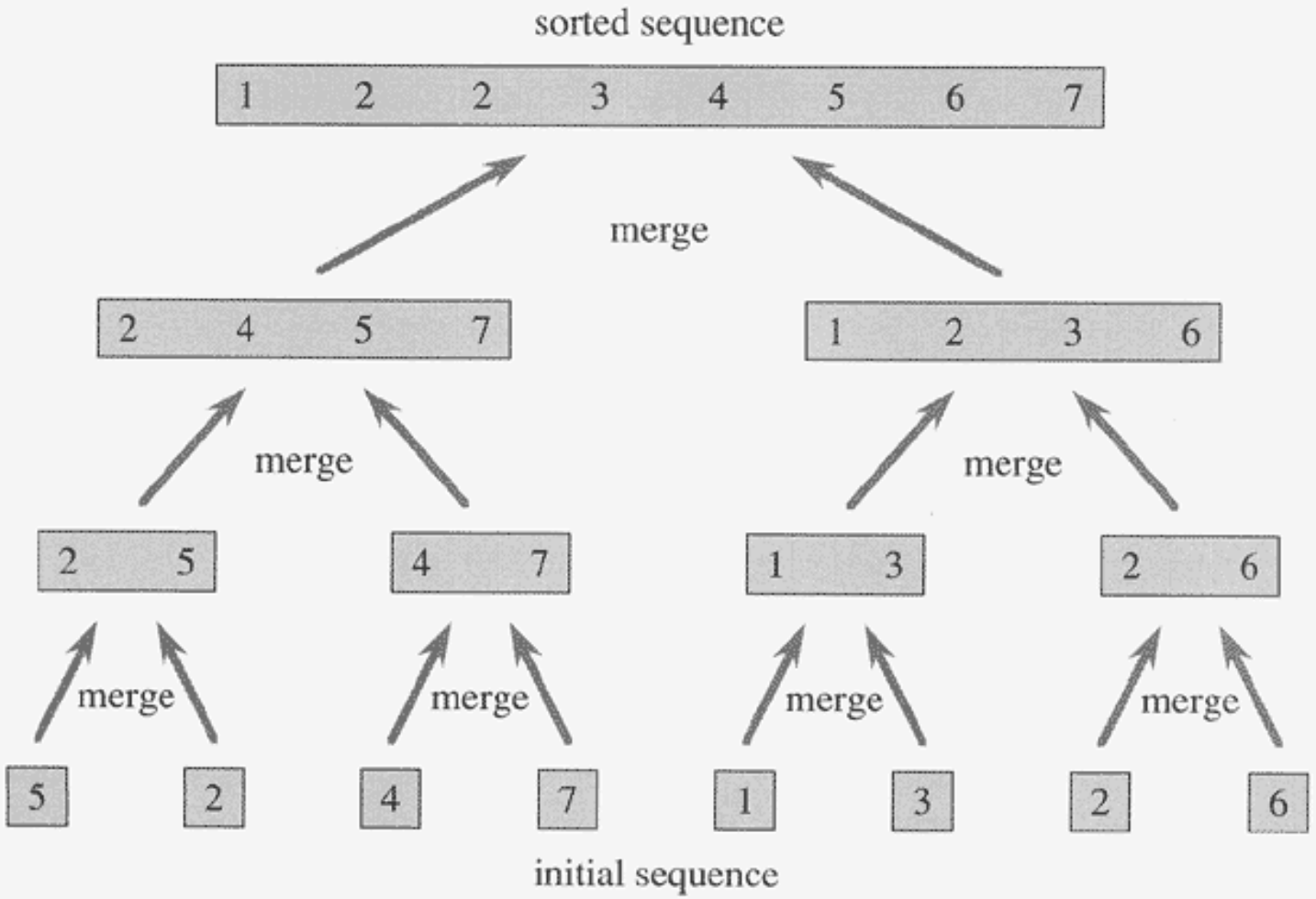

Figure 2.4 The operation of merge sort on the array $A = \langle 5, 2, 4, 7, 1, 3, 2, 6 \rangle$. The lengths of the sorted sequences being merged increase as the algorithm progresses from bottom to top.

Analyzing Divide-and-Conquer Algorithms

Use a ***recurrence*** (equation) to describe the running time of a divide-and-conquer algorithm.

Let $T(n)$ = *running time on a problem of a size* n.

If the problem size is small enough(say, $n \leq c$ for some constant c), we have a ***base case*** – $c(=\Theta(1))$.

Otherwise, suppose that we divide into a subproblems, each $1/b$ the size of the original. (In merge sort, $a=b=2$.)

Let $D(n)$ be the time to *divide* a size-*n* problem.

There are a subproblems to solve, each of size n/b

$\Rightarrow$ each subproblem takes $T(n/b)$ time to solve

$\Rightarrow$ we spend $aT(n/b)$ time solving subproblems.

Let $C(n)$ be the time to *combine* solutions.

We get the recurrence:

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

Analyzing Merge Sort – Use a Recurrence.

For simplicity, assume that $n = 2^k$

The base case: when $n = 1$, $T(n) = \Theta(1)$.

When n ≥ 2, time for merge sort steps:

Divide: Just compute q as the average of p and r ⇒ $D(n) = \Theta(1)$.

Conquer: Recursively solve 2 subproblems, each of size $n/2$ ⇒ $2T(n/2)$

Combine: MERGE on an n *element subarray takes* $\Theta(n)$ time ⇒ $C(n) = \Theta(n)$.

Since $D(n) + C(n) = \Theta(1) + \Theta(n) = \Theta(n)$, the recurrence for merge sort running time is:

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

Solving the merge-sort *recurrence*: $T(n) = \Theta(n \log_2 n)$

Let c be a constant for T(n) of the base case and of the time per array element for the divide and conquer steps.

Rewirte the recurrence as

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

Draw a recursion tree, which shows successive expansions of the recurrence.

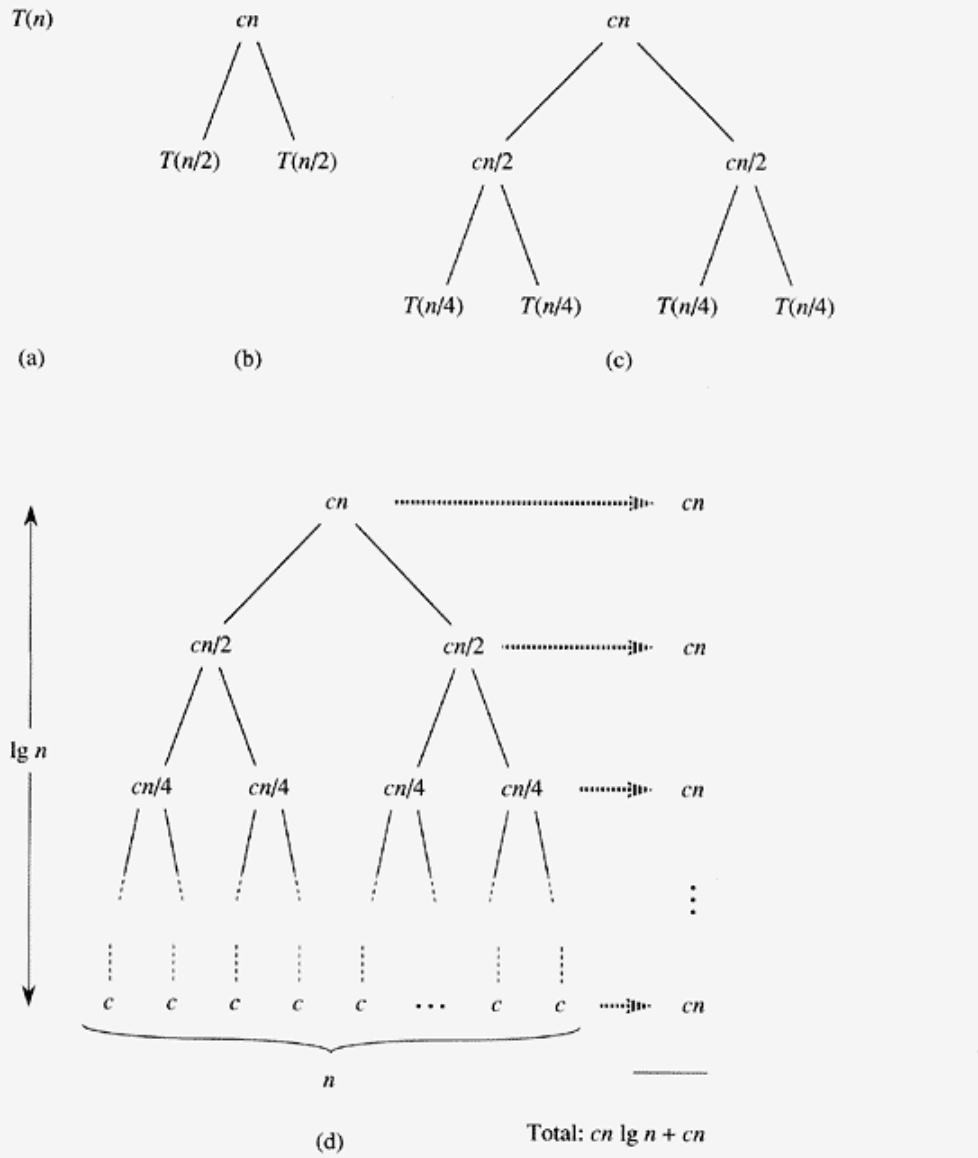

Figure 2.5 The construction of a recursion tree for the recurrence $T(n) = 2T(n/2) + cn$. Part **(a)** shows $T(n)$, which is progressively expanded in **(b)–(d)** to form the recursion tree. The fully expanded tree in part (d) has $\lg n + 1$ levels (i.e., it has height $\lg n$, as indicated), and each level contributes a total cost of cn. The total cost, therefore, is $cn \lg n + cn$, which is $\Theta(n \lg n)$.

BubbleSort:

```
BUBBLESORT(A)
1  for i ← 1 to length[A]
2       do for j ← length[A] downto i + 1
3            do if A[j] < A[j − 1]
4                 then exchange A[j] ↔ A[j − 1]
```