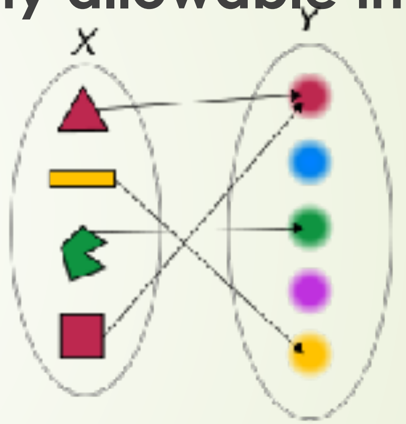
COT 5407: Introduction to Algorithms

Giri NARASIMHAN

www.cs.fiu.edu/~giri/teach/5407S19.html

Definitions

Abstract Problem: defines a function from any allowable input to a corresponding output



Instance of a Problem: a specific input to abstract problem

Algorithm: well-defined computational procedure that takes an instance of a problem as input and produces the correct output

An Algorithm must <u>halt</u> on every input with <u>correct</u> output.

Sorting

- Input is a sequence of n items that can be compared.
- Output is an ordered list of those n items
  - I.e., a reordering or permutation of the input items such that the items are in sorted order
- Fundamental problem that has received a lot of attention over the years.
- Used in many applications.
- Scores of different algorithms exist.
- Task: To compare algorithms
  - On what bases?
    - Time
    - Space
    - Other

Sorting Algorithms

- Number of Comparisons
- Number of Data Movements
- Additional Space Requirements

Sorting Algorithms

- SelectionSort
- InsertionSort
- BubbleSort
- ShakerSort
- MergeSort
- HeapSort
- QuickSort
- Bucket & Radix Sort
- Counting Sort

Worst-Case Time Analysis

- Two Techniques:
  1. **Counts and Summations**:
     - Count number of steps from pseudocode and add
  2. **Recurrence Relations**:
     - Use invariant, write down recurrence relation and solve it
- We will use big-Oh notation to write down time and space complexity (for both worst-case & average-case analyses).
- Compute worst possible time of all input instances of length N.

Definition of big-Oh

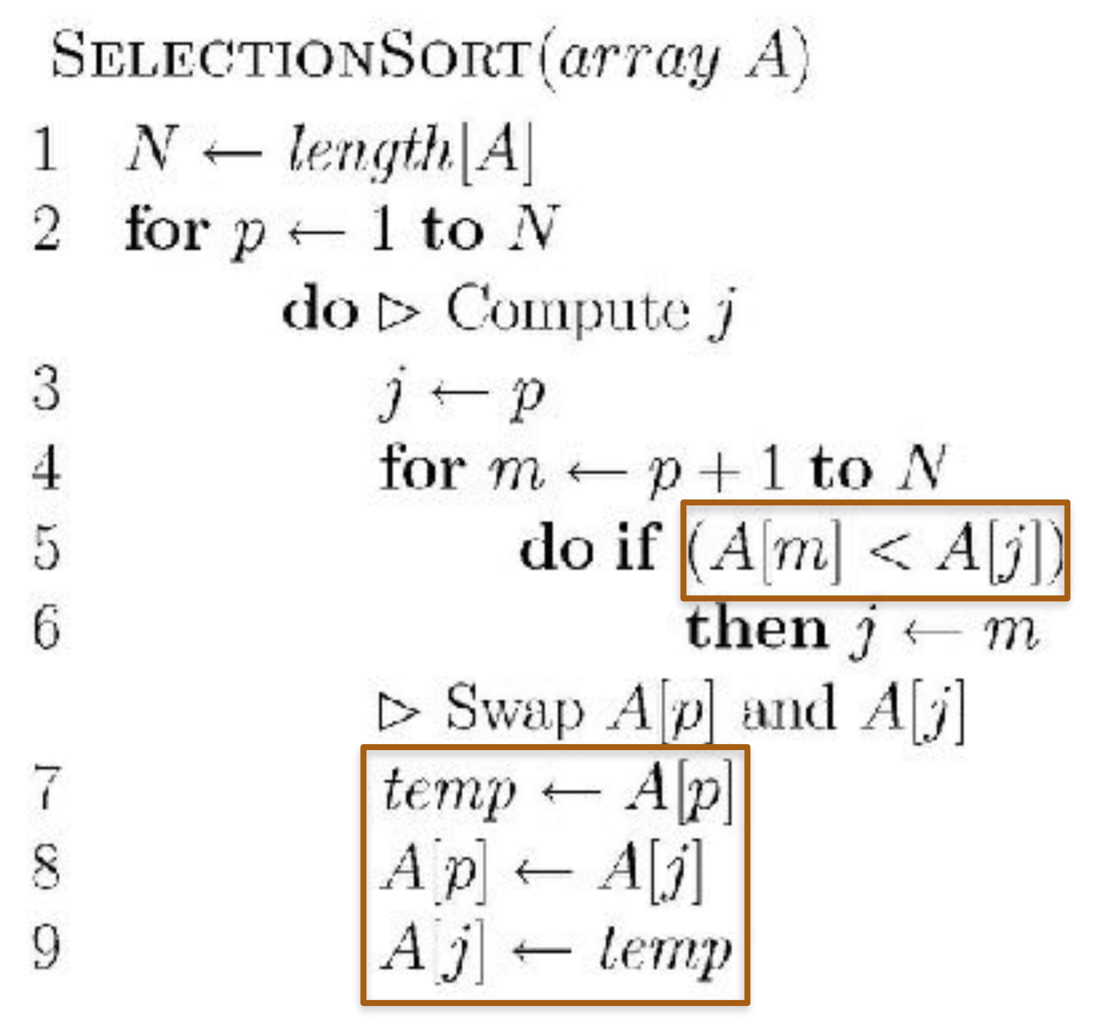
- We say that
  - $F(n) = O(G(n))$

If there exists two <u>positive</u> constants, c and n_0, such that

  - For all $n \geq n_0$, we have $F(n) \leq c\, G(n)$

- Thus, to show that $F(n) = O(G(n))$, you need to find two positive constants that satisfy the condition mentioned above
- Also, to show that $F(n) \neq O(G(n))$, you need to show that for any value of c, there does not exist a positive constant n_0 that satisfies the condition mentioned above

SelectionSort – Worst-case analysis



```
SELECTIONSORT(array A)
N ← length[A]
for p ← 1 to N
        do ▷ Compute j
            j ← p
            for m ← p + 1 to N
                do if (A[m] < A[j])
                    then j ← m
            ▷ Swap A[p] and A[j]
            temp ← A[p]
            A[p] ← A[j]
            A[j] ← temp
```

N-p comparisons

3 data movements

SelectionSort: Worst-Case Analysis

Learn how to sum series

- **Data Movements**

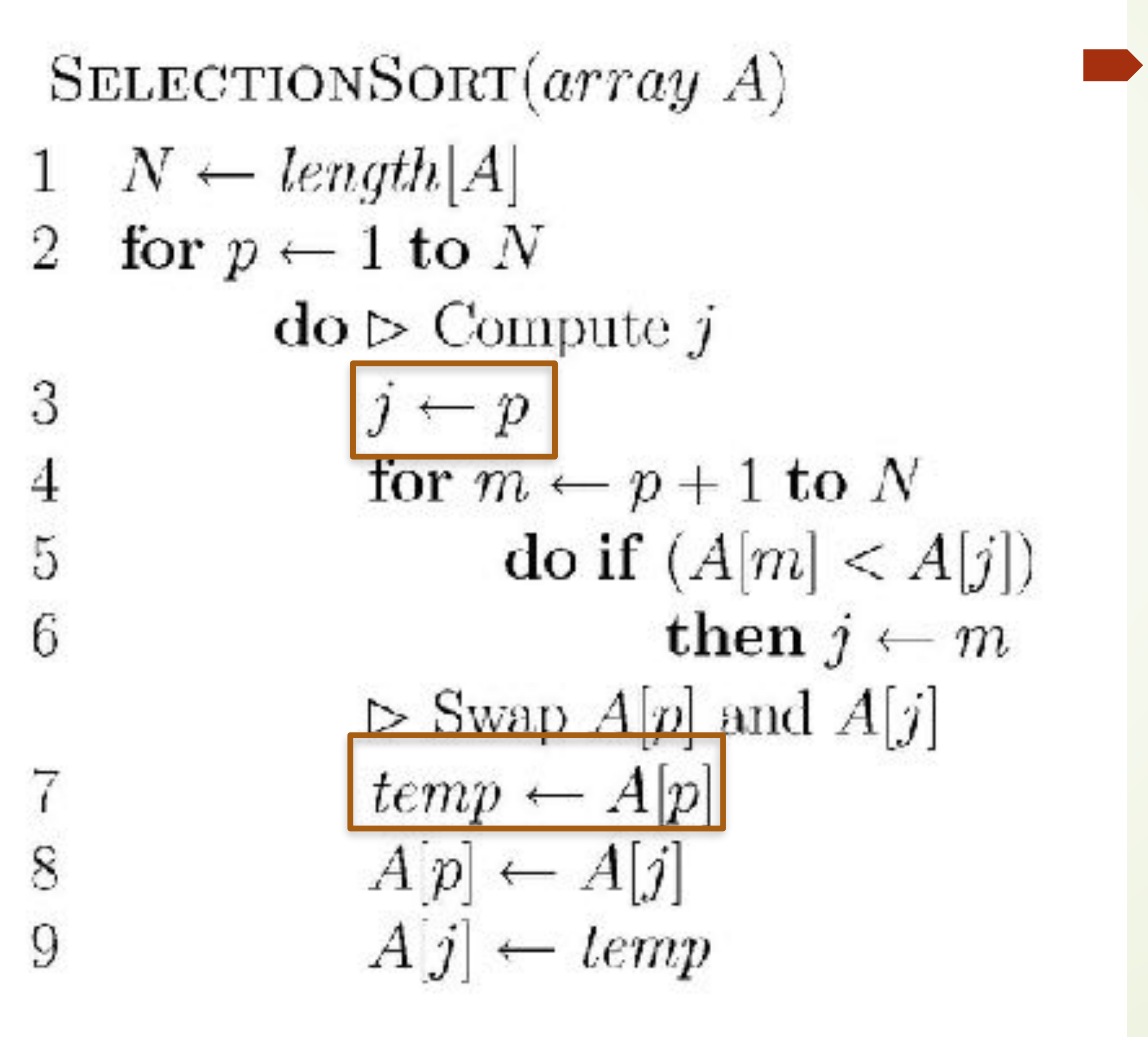
$$= \sum_{p=1}^{N} 3 = 3 \times N = O(N)$$

- **Number of Comparisons**

$$\begin{aligned} &= \sum_{p=1}^{N} (N-p) \\ &= \sum_{p=1}^{N} N - \sum_{p=1}^{N} p \\ &= (N \times N) - (N)(N+1)/2 \\ &= O(N^2) \end{aligned}$$

- **Time Complexity = $O(N^2)$**

SelectionSort – Space Complexity



```
SELECTIONSORT(array $A$)
$N \leftarrow length[A]$
for $p \leftarrow 1$ to $N$
        do ▷ Compute $j$
           $\boxed{j \leftarrow p}$
           for $m \leftarrow p + 1$ to $N$
               do if $(A[m] < A[j])$
                     then $j \leftarrow m$
           ▷ Swap $A[p]$ and $A[j]$
           $\boxed{temp \leftarrow A[p]}$
           $A[p] \leftarrow A[j]$
           $A[j] \leftarrow temp$
```

- Temp Space
  - No extra arrays or data structures
  - O(1)

Invariant for SelectionSort

- An appropriate invariant has a parameter related to the progress of the algorithm (e.g., iteration number)
- An appropriate invariant helps in proving algorithm is correct
- "At the end of iteration p, the p smallest items are in their correct location"

MergeSort

- Divide-and-Conquer Strategy
- Divide array into two sublists of roughly equal length
- Sort each sublist “recursively”
- Merge two sorted lists to get final sorted list
  - **Assumption**: Merging is faster than sorting from fresh
- Most of the work is done in merging
- Process described using a tree
  - **Top-down process**: Divide each list into 2 sublists
  - **Bottom-up process**: Merge two sorted sublists into one sorted sublist

MergeSort

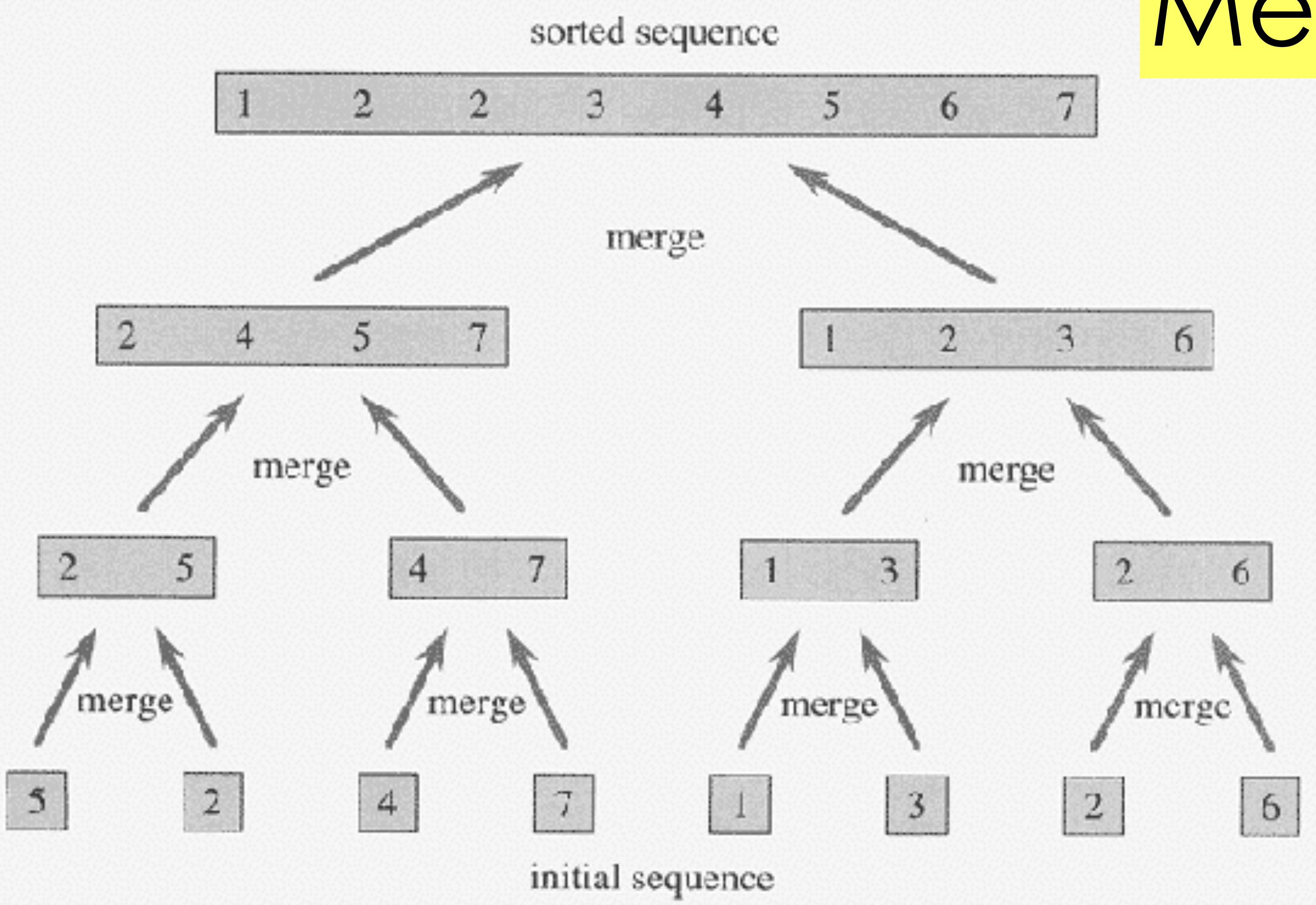


Figure 2.4 The operation of merge sort on the array $A = \langle 5, 2, 4, 7, 1, 3, 2, 6 \rangle$. The lengths of the sorted sequences being merged increase as the algorithm progresses from bottom to top.

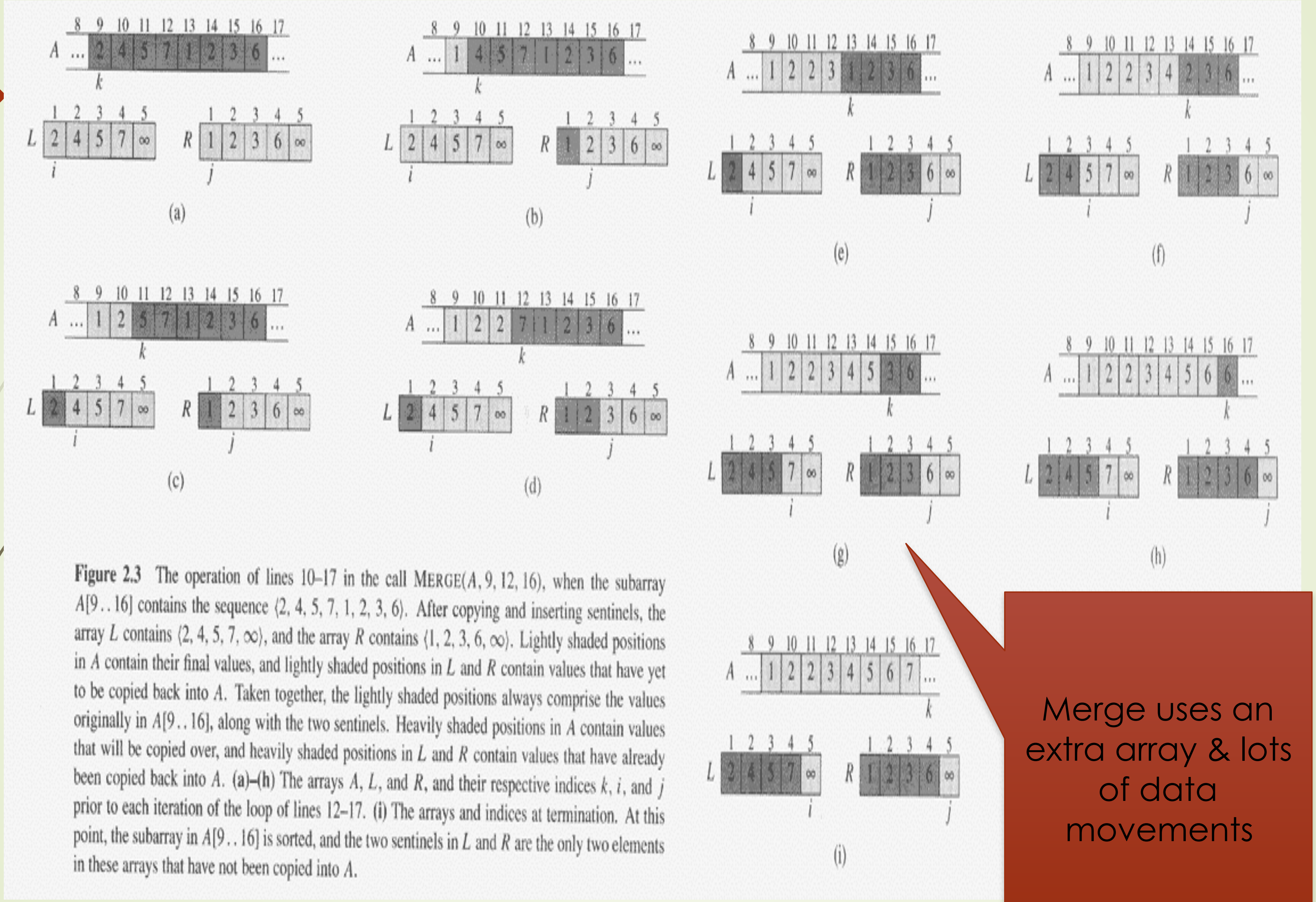
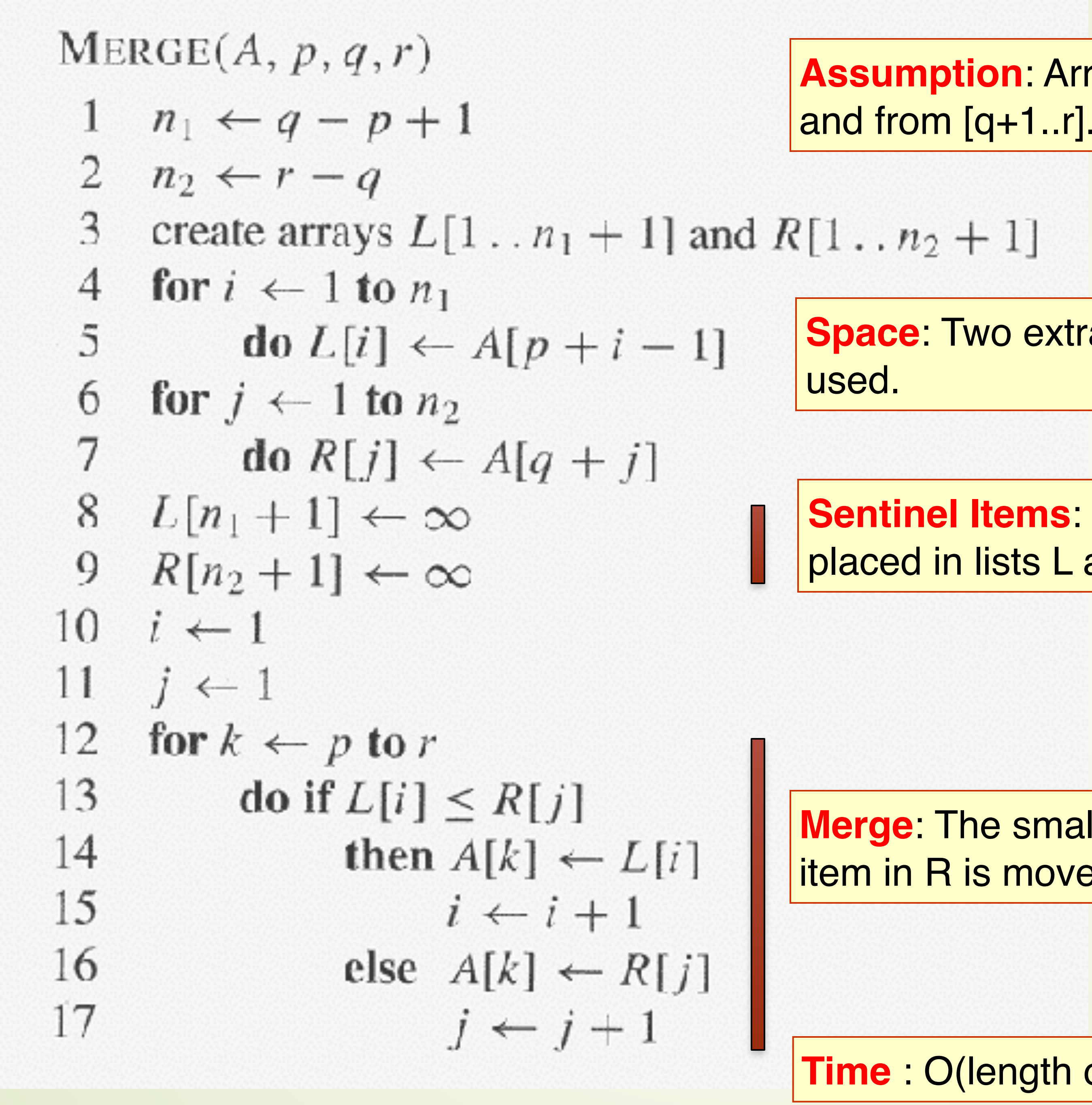


Figure 2.3 The operation of lines 10–17 in the call MERGE$(A, 9, 12, 16)$, when the subarray $A[9..16]$ contains the sequence $\langle 2, 4, 5, 7, 1, 2, 3, 6 \rangle$. After copying and inserting sentinels, the array L contains $\langle 2, 4, 5, 7, \infty \rangle$, and the array R contains $\langle 1, 2, 3, 6, \infty \rangle$. 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.

Merge uses an extra array & lots of data movements



```
MERGE($A, p, q, r$)
$n_1 \leftarrow q - p + 1$
$n_2 \leftarrow r - q$
create arrays $L[1 \,..\, n_1 + 1]$ and $R[1 \,..\, n_2 + 1]$
for $i \leftarrow 1$ to $n_1$
        do $L[i] \leftarrow A[p + i - 1]$
for $j \leftarrow 1$ to $n_2$
        do $R[j] \leftarrow A[q + j]$
$L[n_1 + 1] \leftarrow \infty$
$R[n_2 + 1] \leftarrow \infty$
$i \leftarrow 1$
$j \leftarrow 1$
for $k \leftarrow p$ to $r$
        do if $L[i] \le R[j]$
                then $A[k] \leftarrow L[i]$
                        $i \leftarrow i + 1$
                else $A[k] \leftarrow R[j]$
                        $j \leftarrow j + 1$
```

Assumption: Array A is sorted from [p..q] and from [q+1..r].

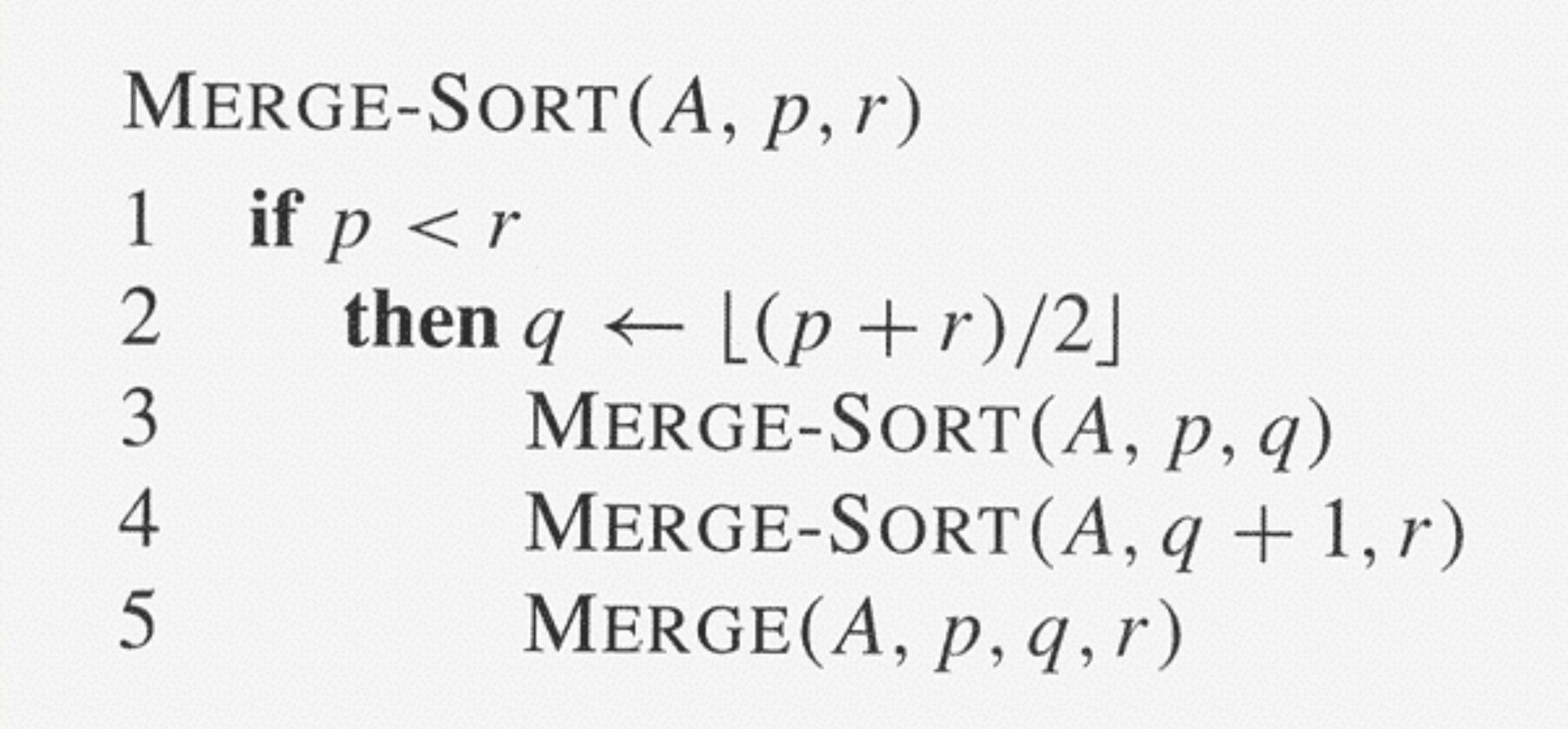
Space: Two extra arrays L and R are used.

Sentinel Items: Two sentinel items placed in lists L and R.

Merge: The smaller of the item in L and item in R is moved to next location in A

Time : O(length of lists)

MergeSort



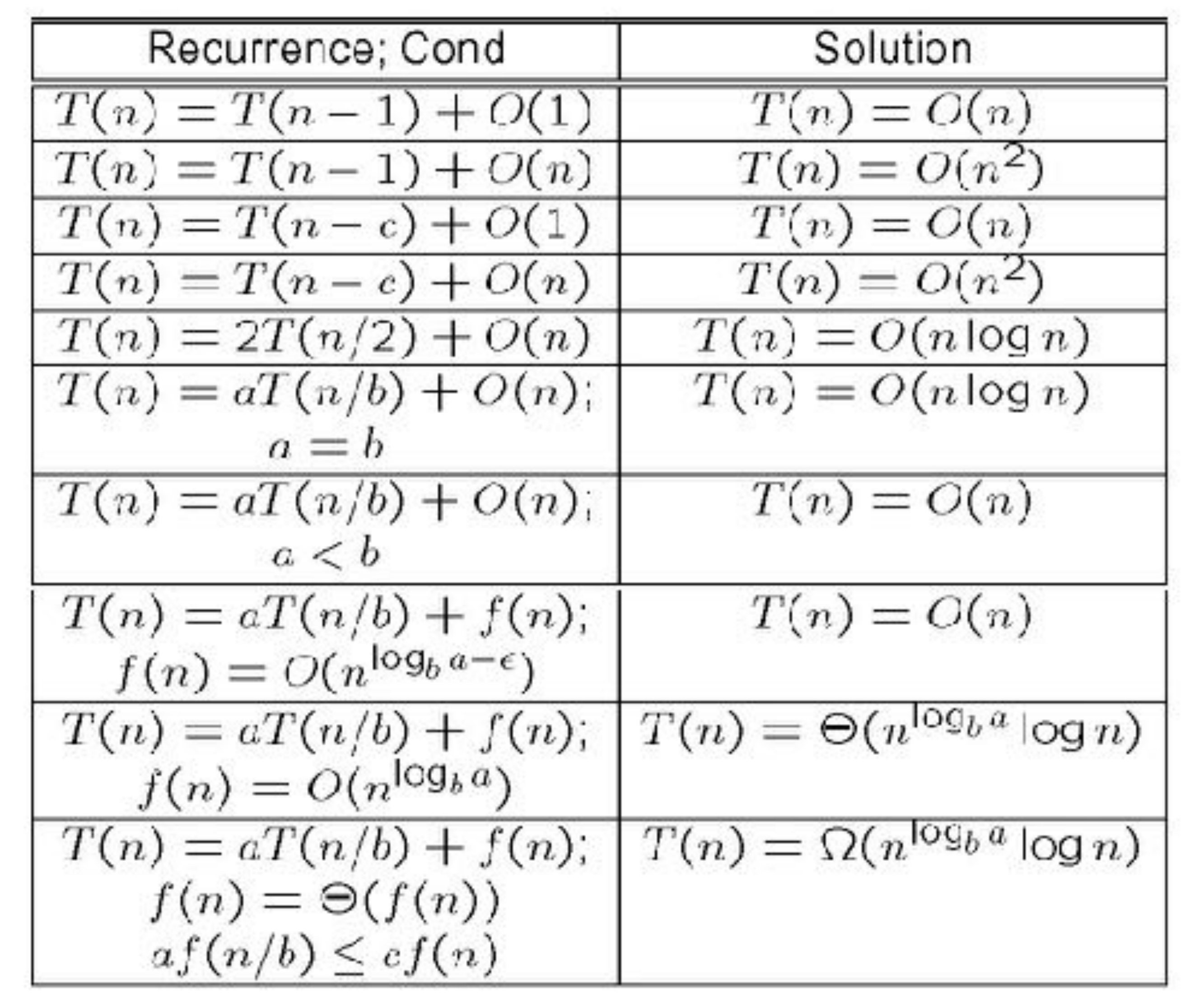
```
MERGE-SORT$(A, p, r)$
    if $p < r$
        then $q \leftarrow \lfloor (p + r)/2 \rfloor$
            MERGE-SORT$(A, p, q)$
            MERGE-SORT$(A, q + 1, r)$
            MERGE$(A, p, q, r)$
```

Time Complexity Recurrence: $T(N) = 2T(N/2) + O(N)$

Invariants, Continued …

- **What is the right invariant for MergeSort?**

Solving Recurrence Relations



Recurrence; Cond	Solution
$T(n) = T(n-1) + O(1)$	$T(n) = O(n)$
$T(n) = T(n-1) + O(n)$	$T(n) = O(n^2)$
$T(n) = T(n-c) + O(1)$	$T(n) = O(n)$
$T(n) = T(n-c) + O(n)$	$T(n) = O(n^2)$
$T(n) = 2T(n/2) + O(n)$	$T(n) = O(n \log n)$
$T(n) = aT(n/b) + O(n);$ $a = b$	$T(n) = O(n \log n)$
$T(n) = aT(n/b) + O(n);$ $a < b$	$T(n) = O(n)$
$T(n) = aT(n/b) + f(n);$ $f(n) = O(n^{\log_b a - \epsilon})$	$T(n) = O(n)$
$T(n) = aT(n/b) + f(n);$ $f(n) = O(n^{\log_b a})$	$T(n) = \Theta(n^{\log_b a} \log n)$
$T(n) = aT(n/b) + f(n);$ $f(n) = \Theta(f(n))$ $af(n/b) \leq cf(n)$	$T(n) = \Omega(n^{\log_b a} \log n)$

Solving Recurrences: Recursion-tree method

- Substitution method fails when a good guess is not available
- Recursion-tree method works in those cases
  - Write down the recurrence as a tree with recursive calls as the children
  - Expand the children
  - Add up each level
  - Sum up the levels
- Useful for analyzing divide-and-conquer algorithms
- Also useful for generating good guesses to be used by substitution method

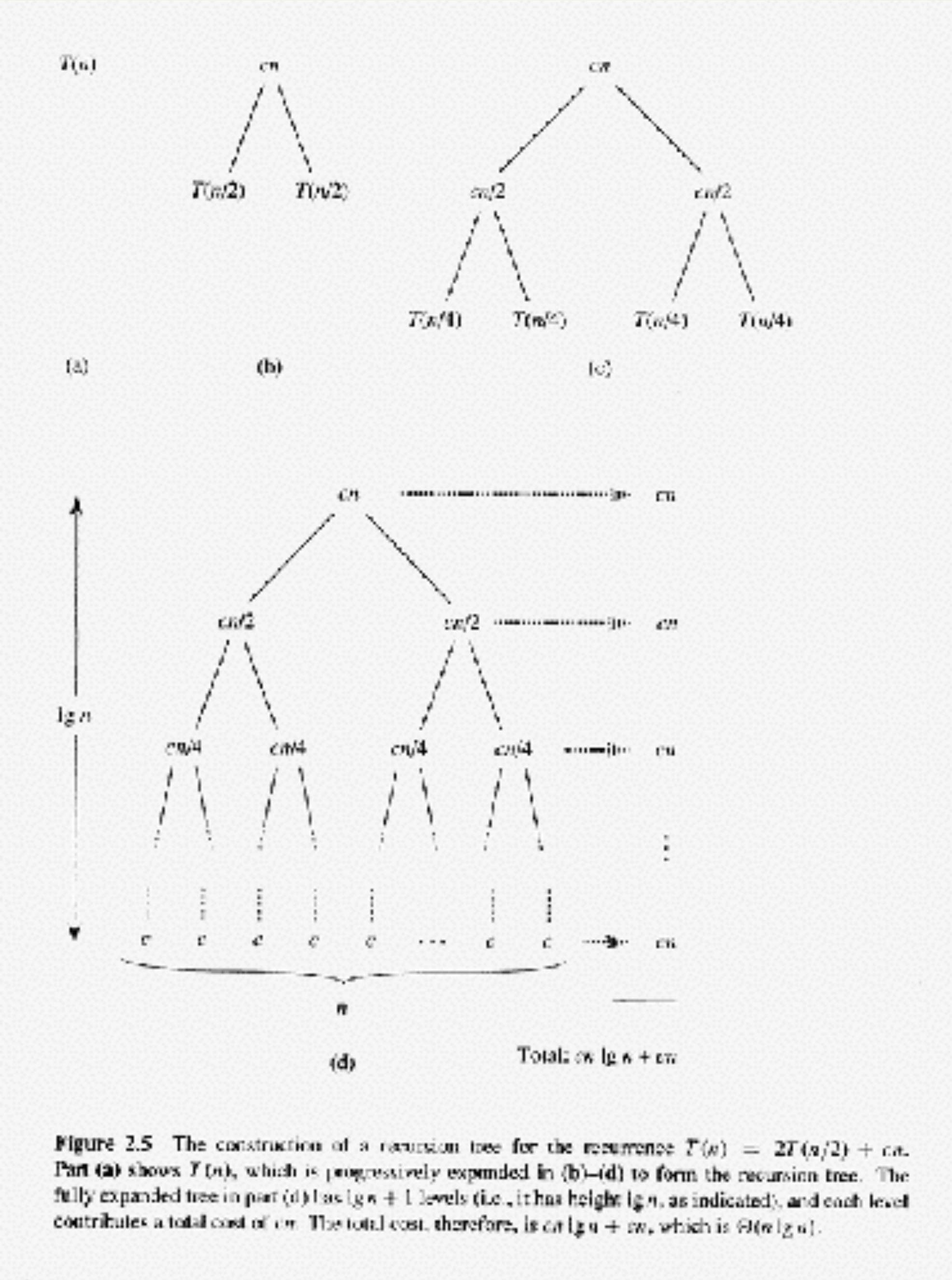


Figure 2.5 The construction of a recursion tree for the recurrence $T(n) = 2T(n/2) + cn$. Part **(a)** shows $T(n)$, which is progressively expanded in **(b)**–**(d)** to form the recursion tree. The fully expanded tree in part (d) has $\lg n + 1$ levels (i.e., it has height $\lg n$, as indicated), and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

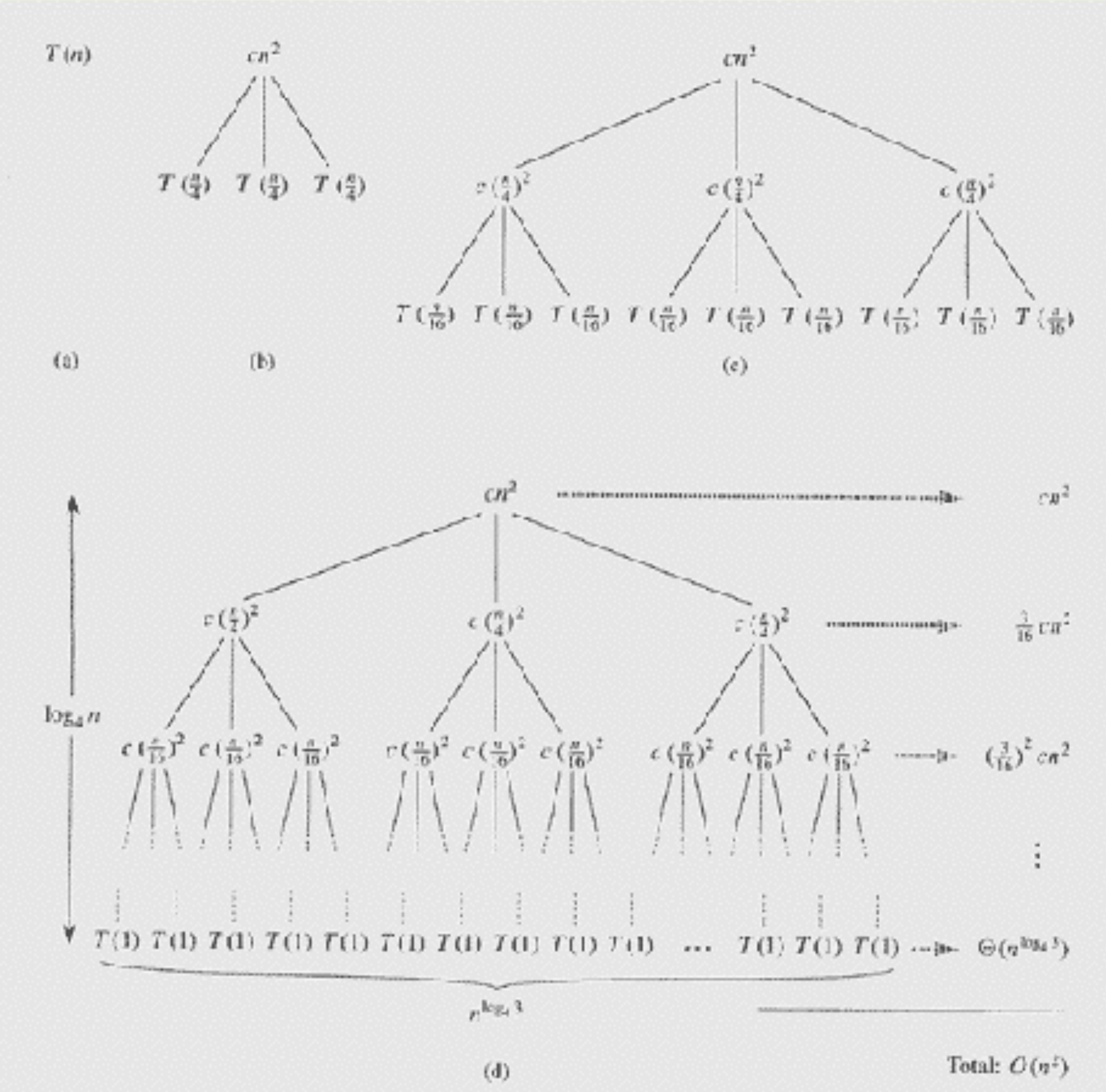


Figure 4.1 The construction of a recursion tree for the recurrence $T(n) = 3T(n/4) + cn^2$. Part (a) shows $T(n)$, which is progressively expanded in (b)–(d) to form the recursion tree. The fully expanded tree in part (d) has height $\log_4 n$ (it has $\log_4 n + 1$ levels).

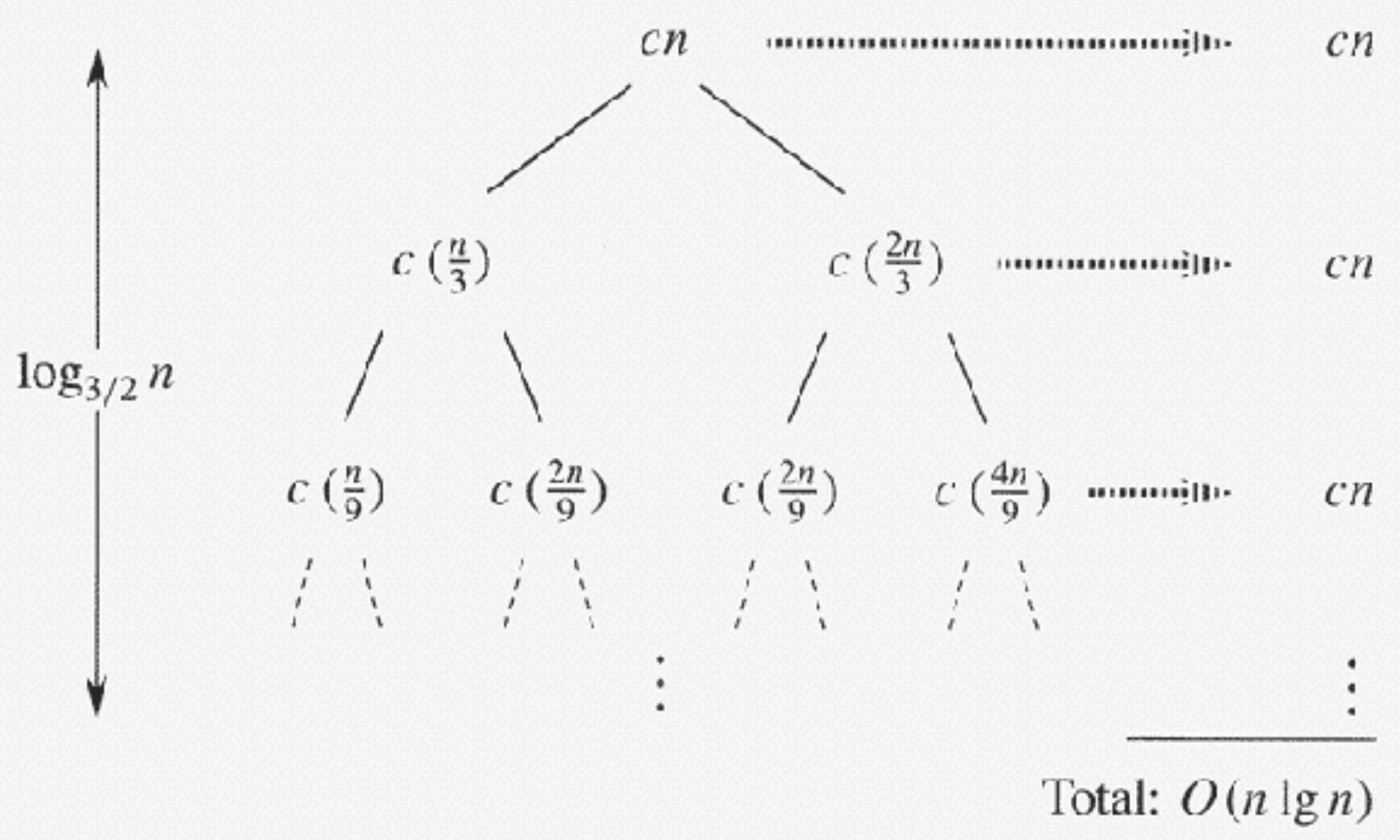


Figure 4.2 A recursion tree for the recurrence $T(n) = T(n/3) + T(2n/3) + cn$.

Solving Recurrences using Master Theorem

Master Theorem:

Let a,b >= 1 be constants, let f(n) be a function, and let

$$T(n) = aT(n/b) + f(n)$$

1. If f(n) = O($n^{\log_b a-e}$) for some constant e>0, then
   - T(n) = Theta($n^{\log_b a}$)
2. If f(n) = Theta($n^{\log_b a}$), then
   - T(n) = Theta($n^{\log_b a}$ log n)
3. If f(n) = Omega($n^{\log_b a+e}$) for some constant e>0, then
   - T(n) = Theta(f(n))