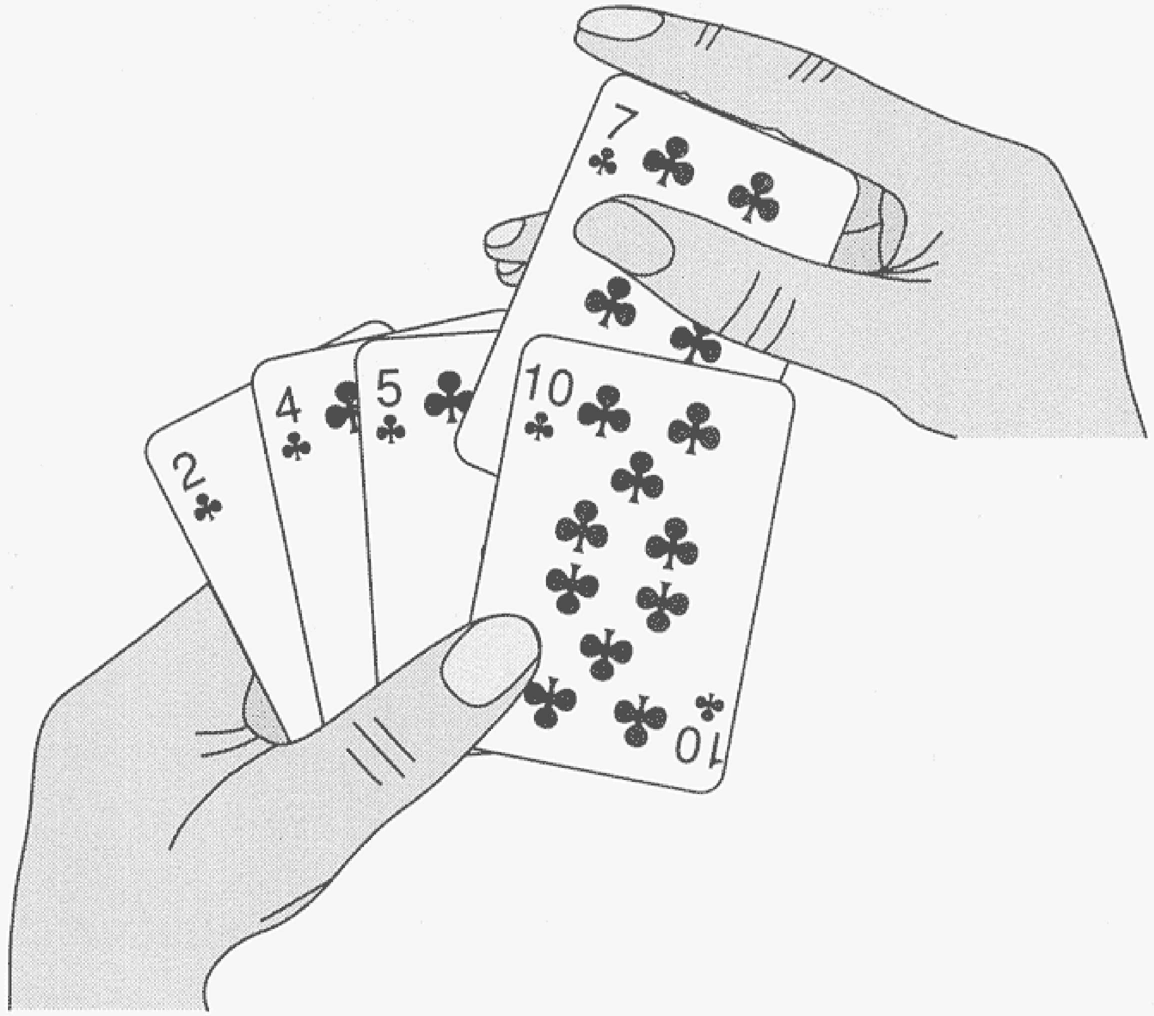


Figure 2.1 Sorting a hand of cards using insertion sort.

Sorting

- Input is a list of n items that can be compared.
- Output is an ordered list of those n items.
- 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

SelectionSort

Array Position		0	1	2	3	4	5
Initial State		8	5	9	2	6	3
After Iteration 1		2	5	9	8	6	3
After Iteration 2		2	3	9	8	6	5
After Iteration 3		2	3	5	8	6	9
After Iteration 4		2	3	5	6	8	9
After Iteration 5		2	3	5	6	8	9

Psuedocode

- Convention about statements
- Indentation
- Comments
- Parameters -- passed by value not reference
- And/or are short-circuiting

SelectionSort

```
$\textsc{SelectionSort}(array\ A)$
1  $N \leftarrow length[A]$
2  for $p \leftarrow 1$ to $N$
3        do Compute $j$, the index of the
            smallest item in $A[p..N]$
4           Swap $A[p]$ and $A[j]$
```

SelectionSort

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

SelectionSort: Algorithm Invariants

- iteration k:
  - the k smallest items are in correct location
- NEED TO PROVE THE INVARIANT!!

How to prove invariants & correctness

- **Initialization**: prove it is true at start
- **Maintenance**: prove it is maintained within iterative control structures
- **Termination**: show how to use it to prove correctness

Algorithm Analysis

- Worst-case time complexity
- (Worst-case) space complexity
- Average-case time complexity

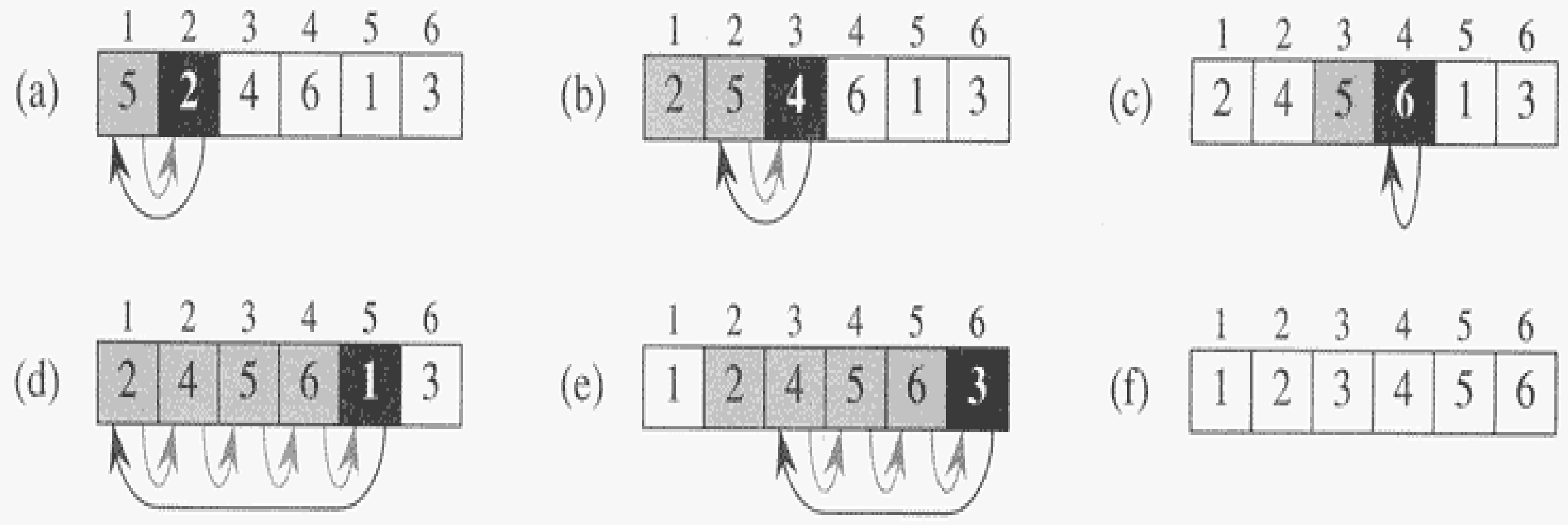
SelectionSort

```
SELECTIONSORT($array\ A$)
1   $N \leftarrow length[A]$
2   for $p \leftarrow 1$ to $N$
          do ▷ Compute $j$
3             $j \leftarrow p$
4             for $m \leftarrow p + 1$ to $N$
5                  do if $(A[m] < A[j])$
6                        then $j \leftarrow m$
              ▷ Swap $A[p]$ and $A[j]$
7             $temp \leftarrow A[p]$
8             $A[p] \leftarrow A[j]$
9             $A[j] \leftarrow temp$
```

$O(n^2)$ time

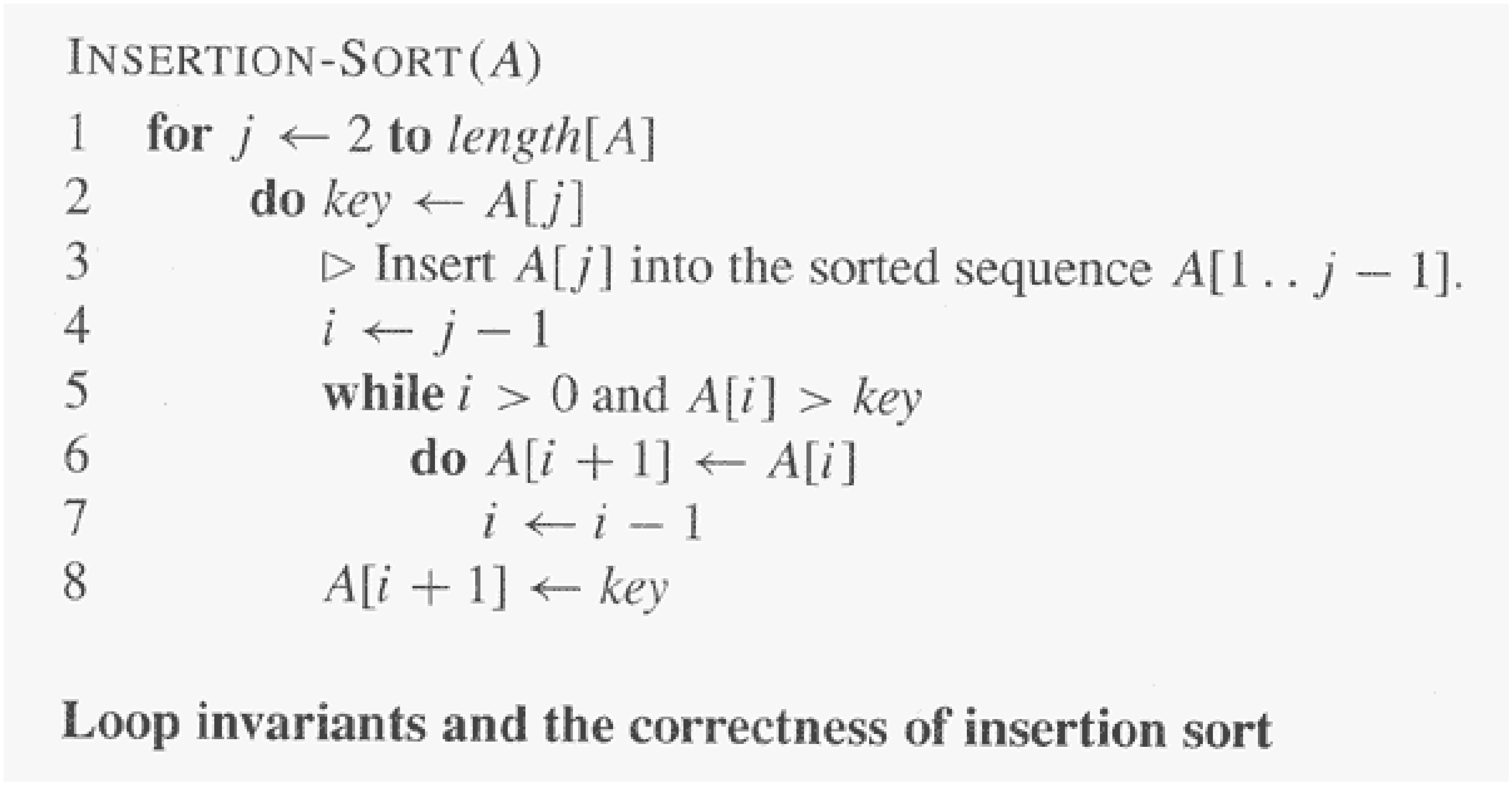
$O(1)$ space

InsertionSort



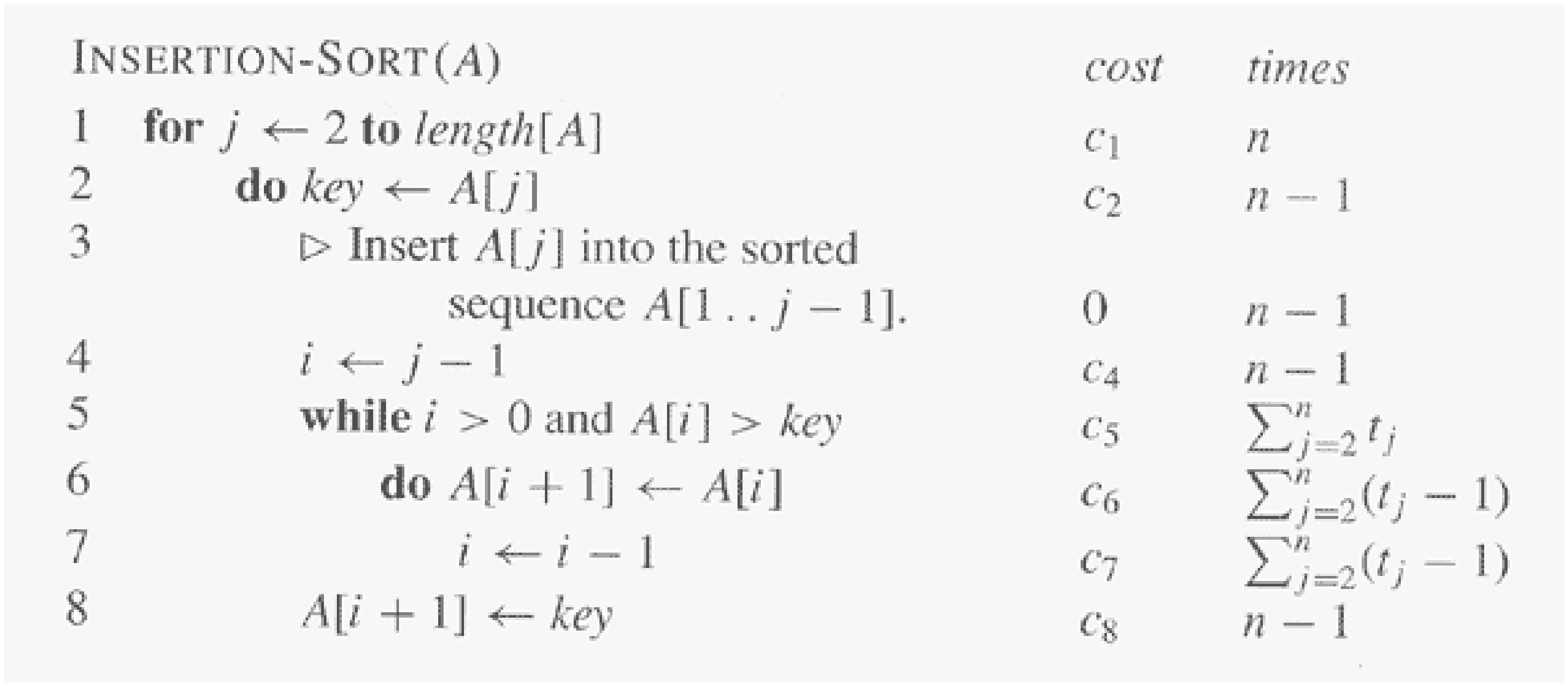
(a) 1 2 3 4 5 6 | 5 2 4 6 1 3
(b) 1 2 3 4 5 6 | 2 5 4 6 1 3
(c) 1 2 3 4 5 6 | 2 4 5 6 1 3
(d) 1 2 3 4 5 6 | 2 4 5 6 1 3
(e) 1 2 3 4 5 6 | 1 2 4 5 6 3
(f) 1 2 3 4 5 6 | 1 2 3 4 5 6

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 $j \leftarrow 2$ to $length[A]$
2       do $key \leftarrow A[j]$
3          ▷ Insert $A[j]$ into the sorted sequence $A[1 \,..\, j-1]$.
4          $i \leftarrow j - 1$
5          while $i > 0$ and $A[i] > key$
6              do $A[i+1] \leftarrow A[i]$
7                 $i \leftarrow i - 1$
8          $A[i+1] \leftarrow key$
```

Loop invariants and the correctness of insertion sort



	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 $A[1\,..\,j-1]$.	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	$n-1$

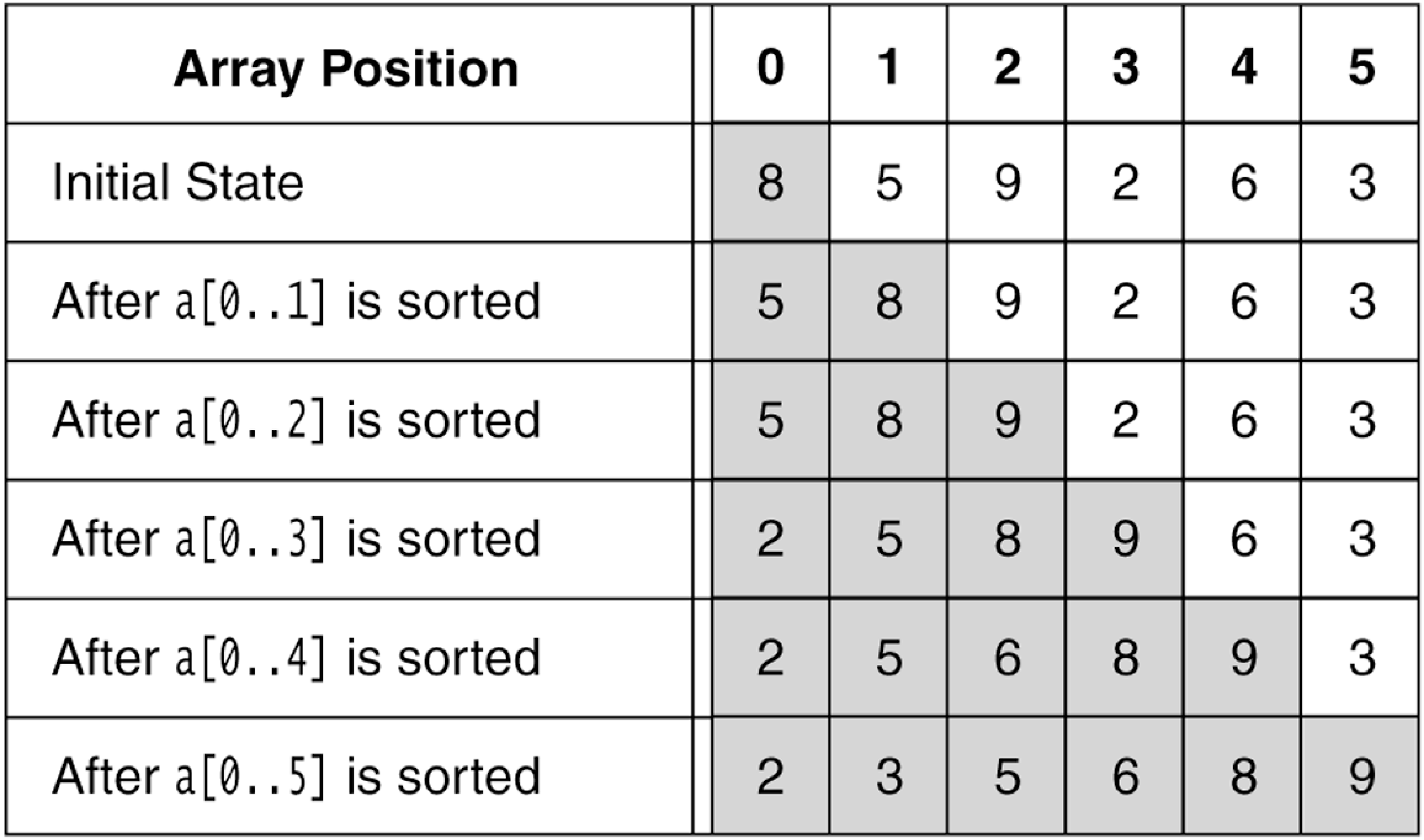
$O(n^2)$ time

$O(1)$ space

InsertionSort: Algorithm Invariant

- iteration k:
  - the first k items are in sorted order.

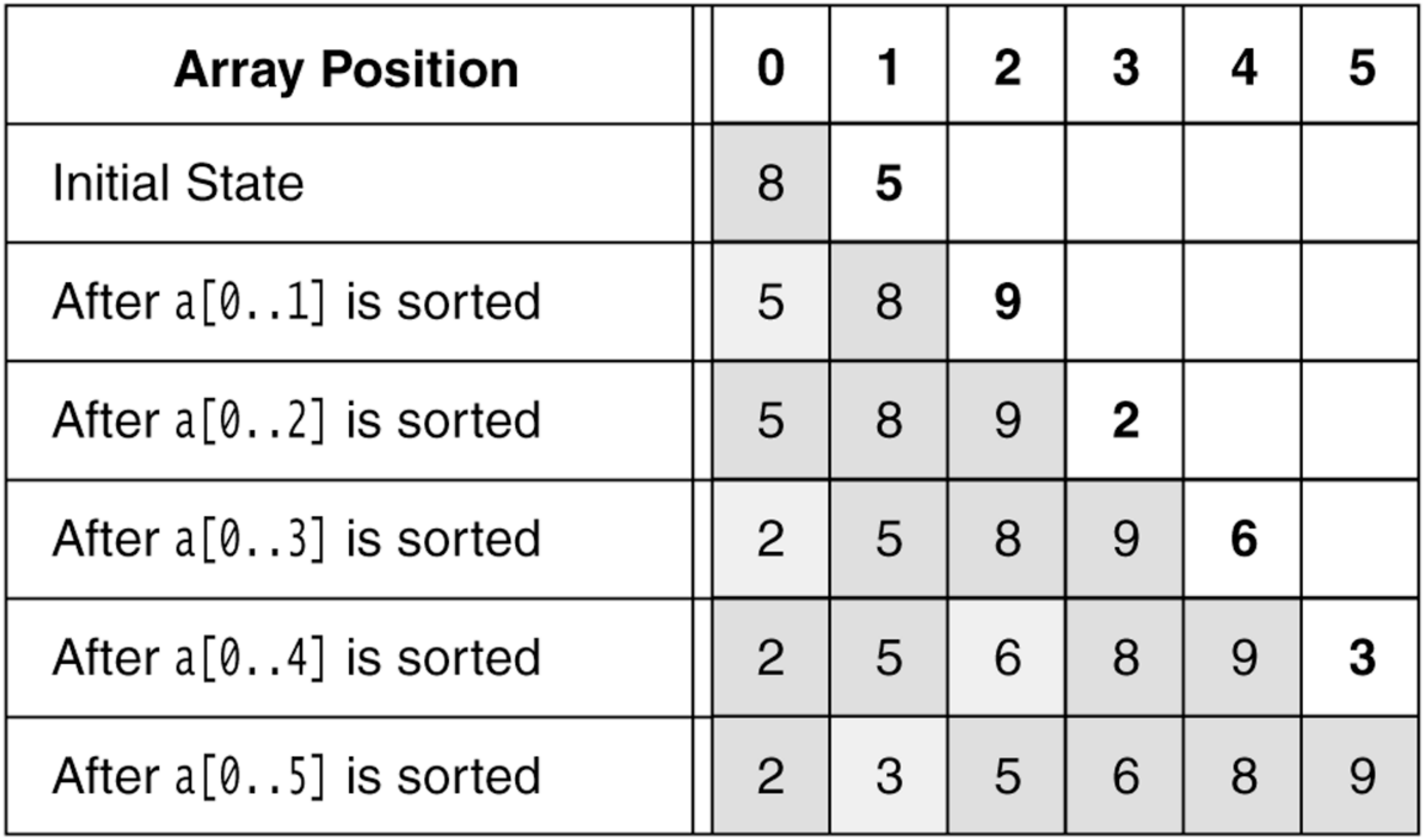
Figure 8.3
Basic action of insertion sort (the shaded part is sorted)



Array Position	0	1	2	3	4	5
Initial State	8	5	9	2	6	3
After a[0..1] is sorted	5	8	9	2	6	3
After a[0..2] is sorted	5	8	9	2	6	3
After a[0..3] is sorted	2	5	8	9	6	3
After a[0..4] is sorted	2	5	6	8	9	3
After a[0..5] is sorted	2	3	5	6	8	9

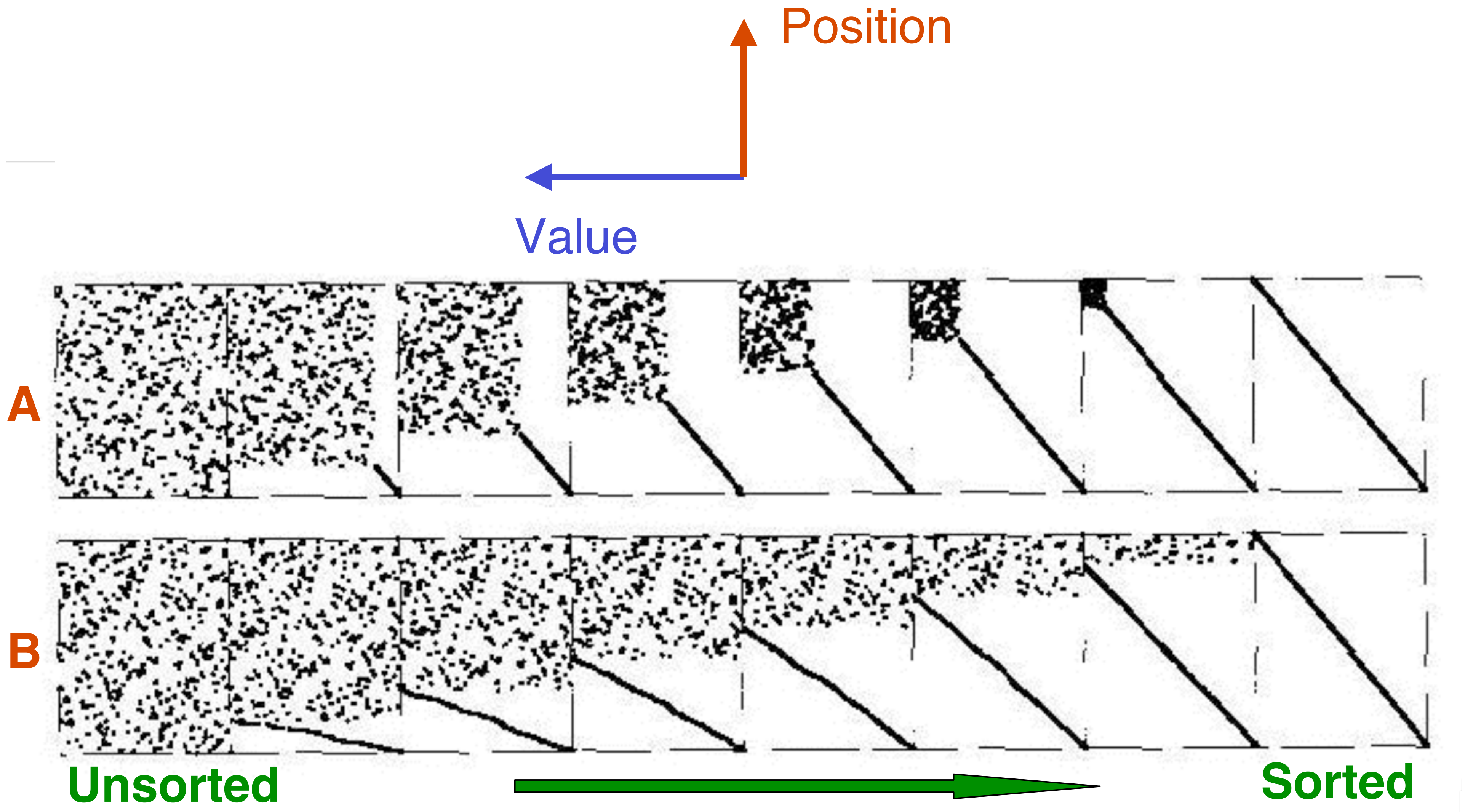
Figure 8.4

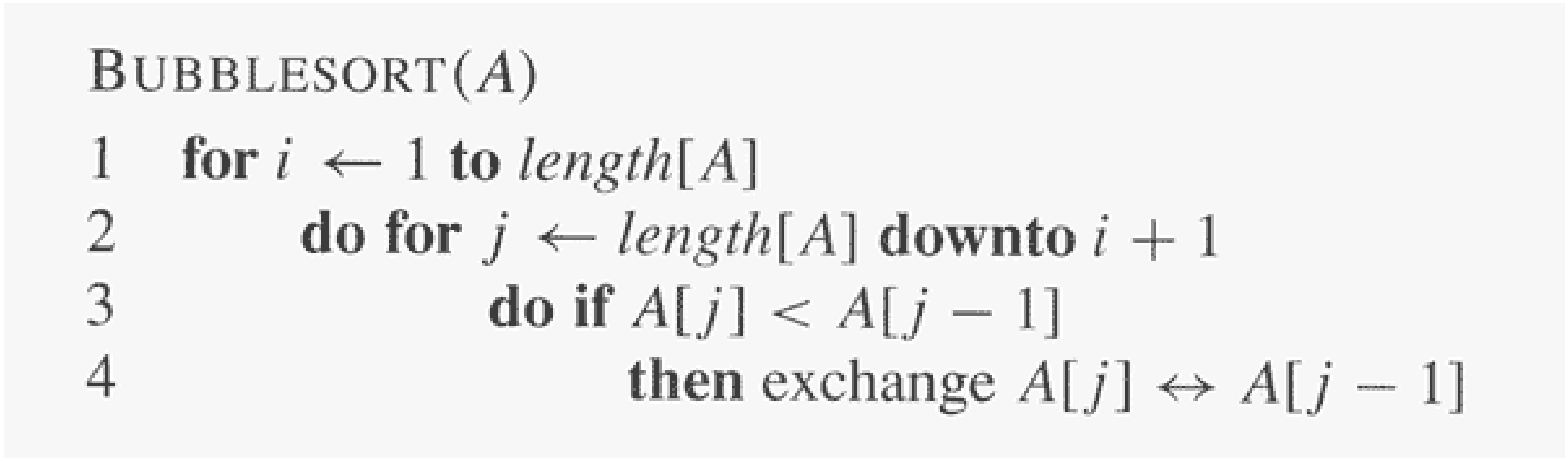
A closer look at the action of insertion sort (the dark shading indicates the sorted area; the light shading is where the new element was placed).



Array Position	0	1	2	3	4	5
Initial State	8	**5**				
After `a[0..1]` is sorted	5	8	**9**			
After `a[0..2]` is sorted	5	8	9	**2**		
After `a[0..3]` is sorted	2	5	8	9	**6**	
After `a[0..4]` is sorted	2	5	6	8	9	**3**
After `a[0..5]` is sorted	2	3	5	6	8	9

Visualizing Algorithms 1





```
BUBBLESORT($A$)
1  for $i \leftarrow 1$ to $length[A]$
2      do for $j \leftarrow length[A]$ downto $i + 1$
3          do if $A[j] < A[j-1]$
4              then exchange $A[j] \leftrightarrow A[j-1]$
```

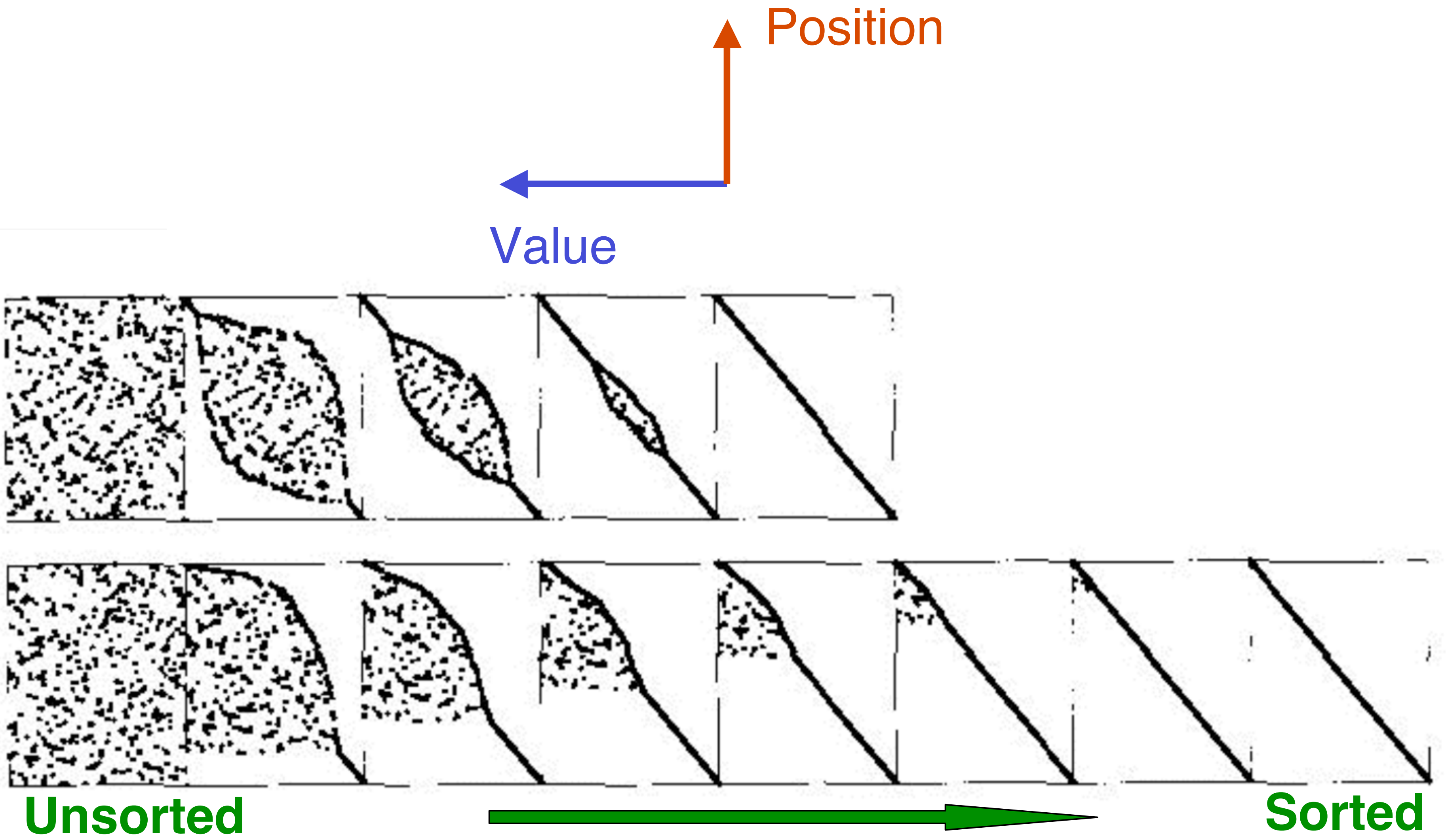
O(n^2) time

O(1) space

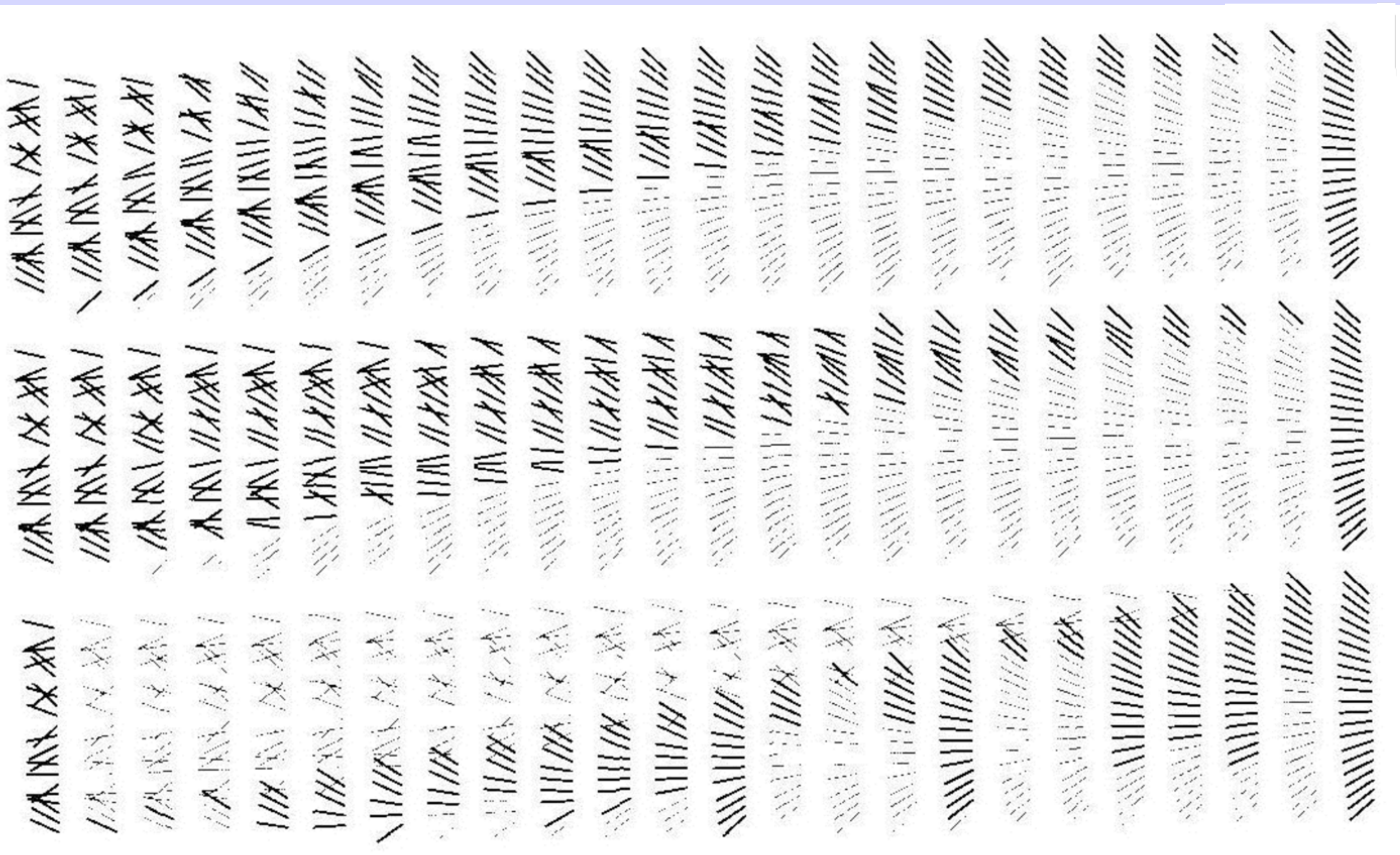
BubbleSort: Algorithm Invariant

- In each pass, every item that does not have a smaller item after it, is moved as far up in the list as possible.
- Iteration k:
  - k smallest items are in the correct location.

Visualizing Algorithms 2



Visualizing Comparisons 3



Animation Demos

http://cg.scs.carleton.ca/~morin/misc/sortalg/

Comparing O(n^2) Sorting Algorithms

- InsertionSort and SelectionSort (and ShakerSort) are roughly twice as fast as BubbleSort for small files.
- InsertionSort is the best for very small files.
- O(n^2) sorting algorithms are **NOT** useful for large random files.
- If comparisons are very expensive, then among the O(n^2) sorting algorithms, insertionsort is best.
- If data movements are very expensive, then among the O(n^2) sorting algorithms, ?? is best.

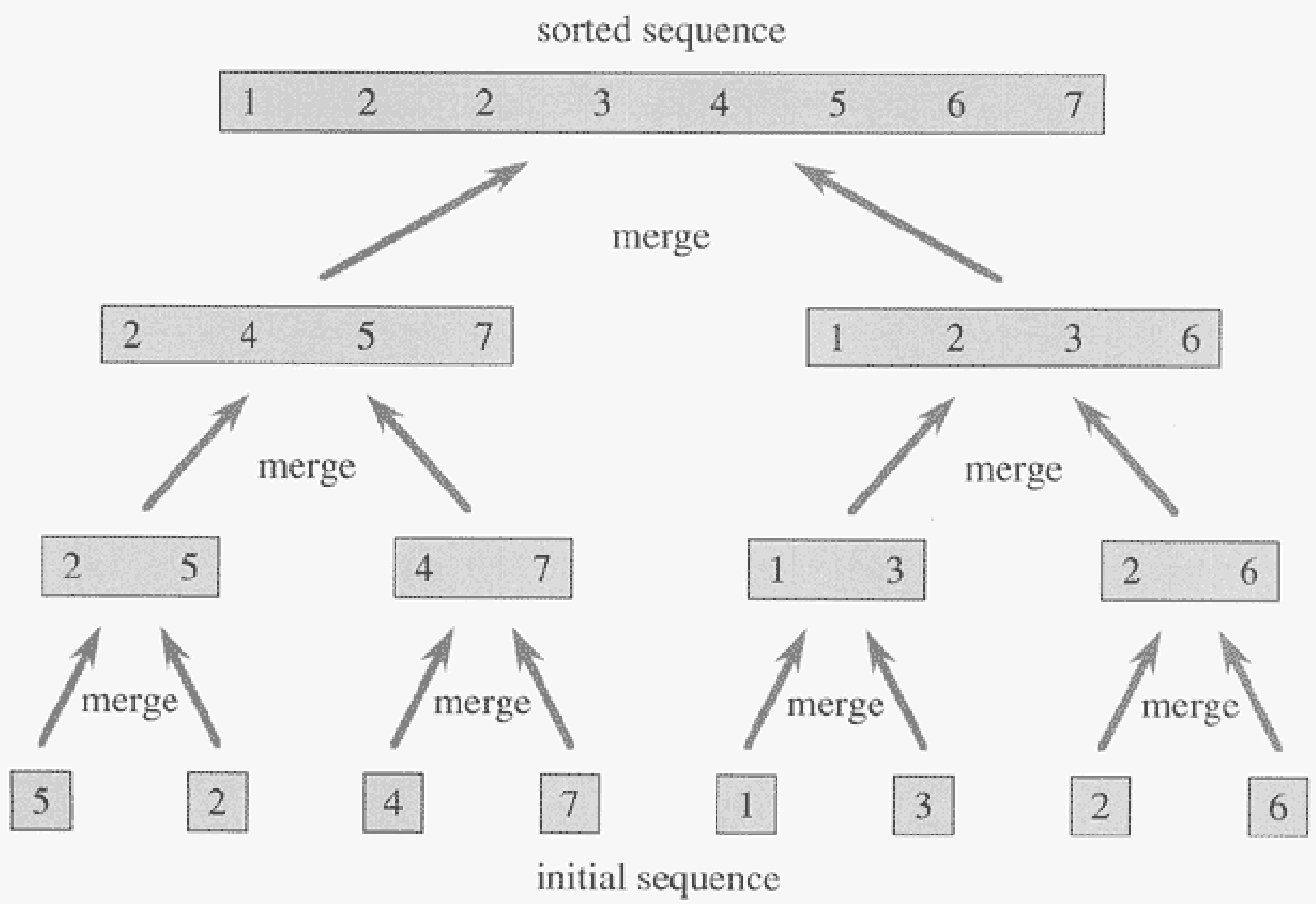
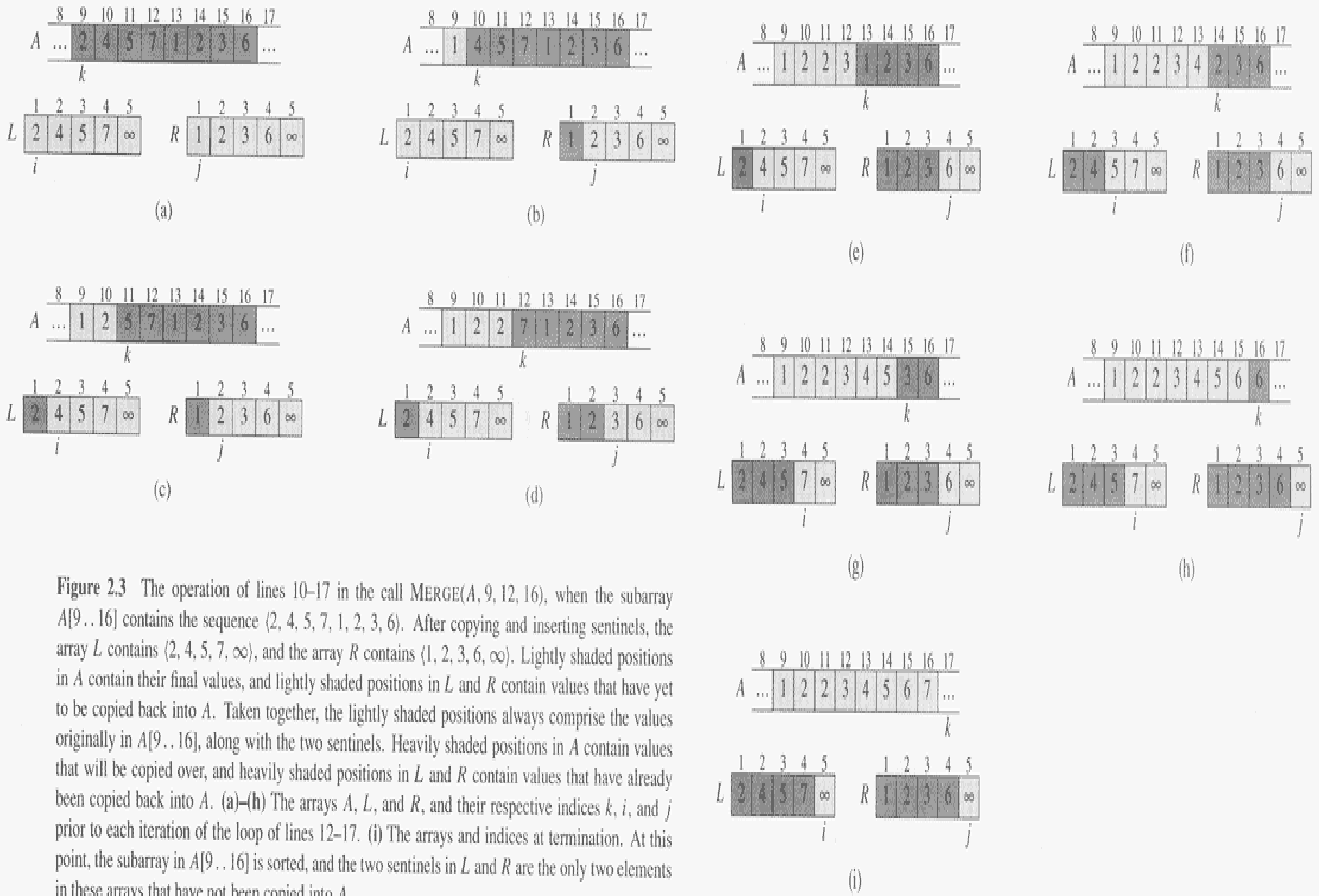
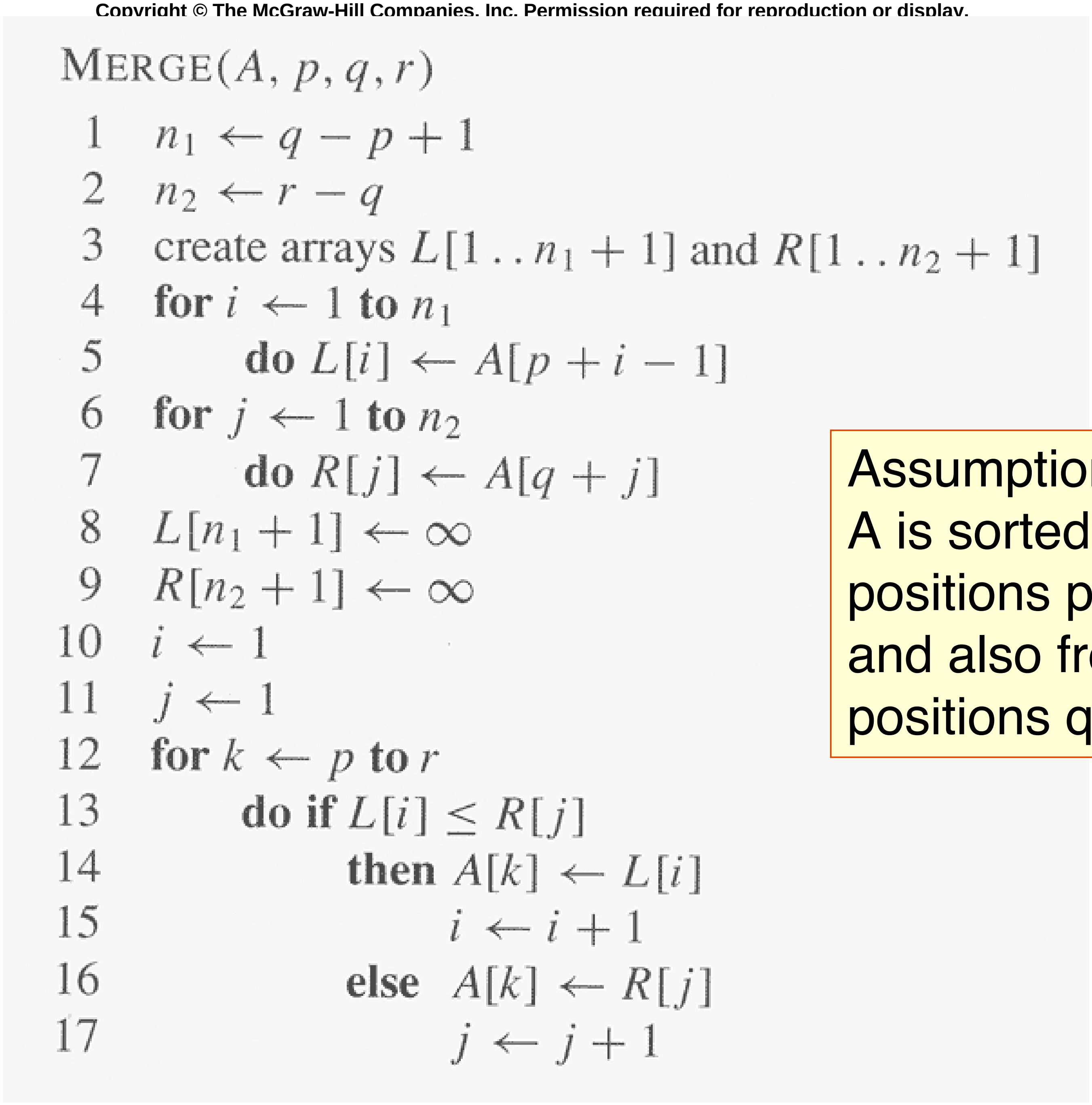


Figure 2.4 The operation of merge sort on the array $A = \langle 5, 2, 4, 7, 1, 3, 2, 6 \rangle$. The lengths of the sorted sequences being merged increase as the algorithm progresses from bottom to top.



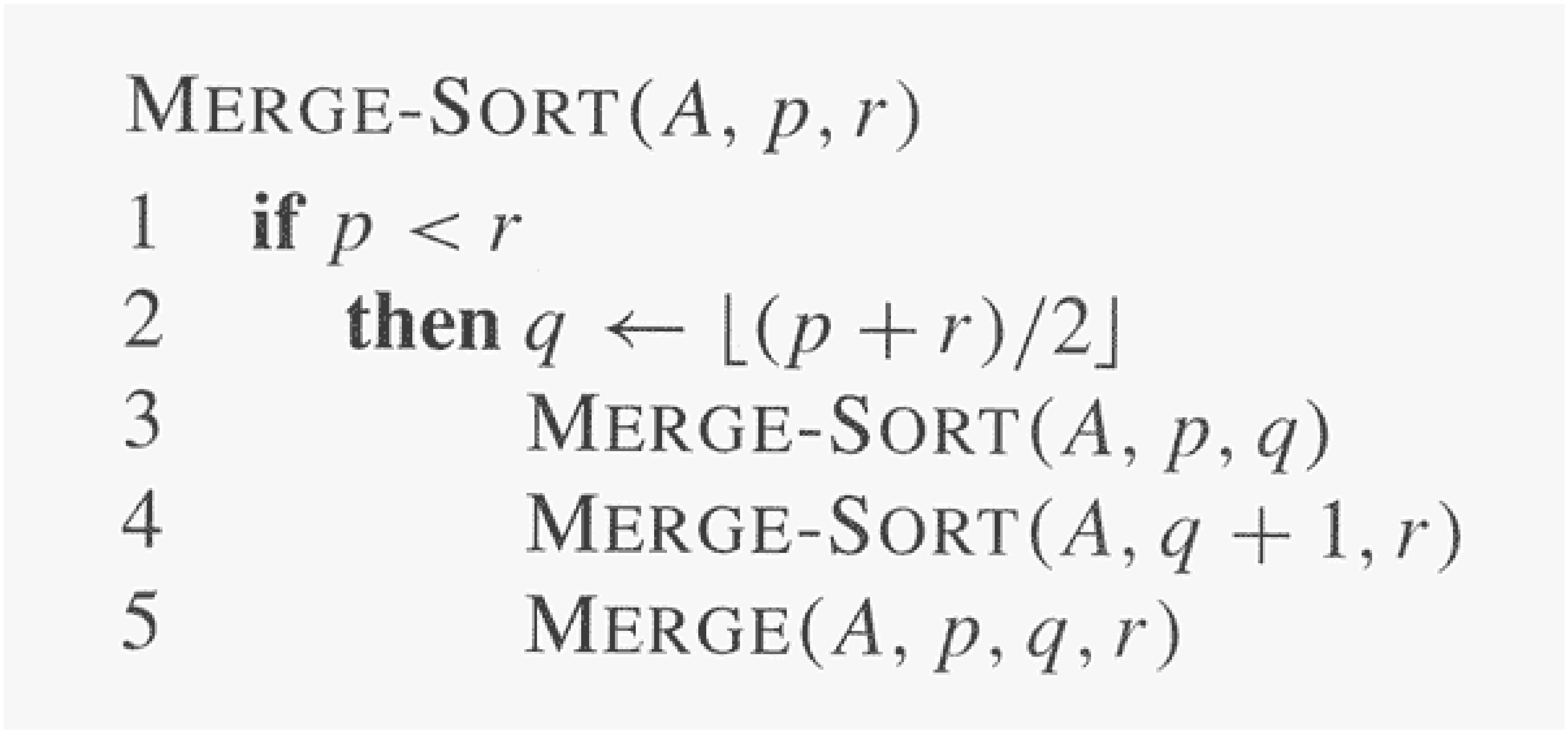
(a) A … 2 4 5 7 1 2 3 6 … k; L 2 4 5 7 ∞ i; R 1 2 3 6 ∞ j
(b) A … 1 4 5 7 1 2 3 6 … k; L 2 4 5 7 ∞ i; R 1 2 3 6 ∞ j
(c) A … 1 2 5 7 1 2 3 6 … k; L 2 4 5 7 ∞ i; R 1 2 3 6 ∞ j
(d) A … 1 2 2 7 1 2 3 6 … k; L 2 4 5 7 ∞ i; R 1 2 3 6 ∞ j
(e) A … 1 2 2 3 1 2 3 6 … k; L 2 4 5 7 ∞ i; R 1 2 3 6 ∞ j
(f) A … 1 2 2 3 4 2 3 6 … k; L 2 4 5 7 ∞ i; R 1 2 3 6 ∞ j
(g) A … 1 2 2 3 4 5 3 6 … k; L 2 4 5 7 ∞ i; R 1 2 3 6 ∞ j
(h) A … 1 2 2 3 4 5 6 6 … k; L 2 4 5 7 ∞ i; R 1 2 3 6 ∞ j
(i) A … 1 2 2 3 4 5 6 7 … k; L 2 4 5 7 ∞ i; R 1 2 3 6 ∞ j

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$(A, p, q, r)$
$n_1 \leftarrow q - p + 1$
$n_2 \leftarrow r - q$
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 $L[i] \leq R[j]$
          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 positions p to q and also from positions q+1 to r.



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

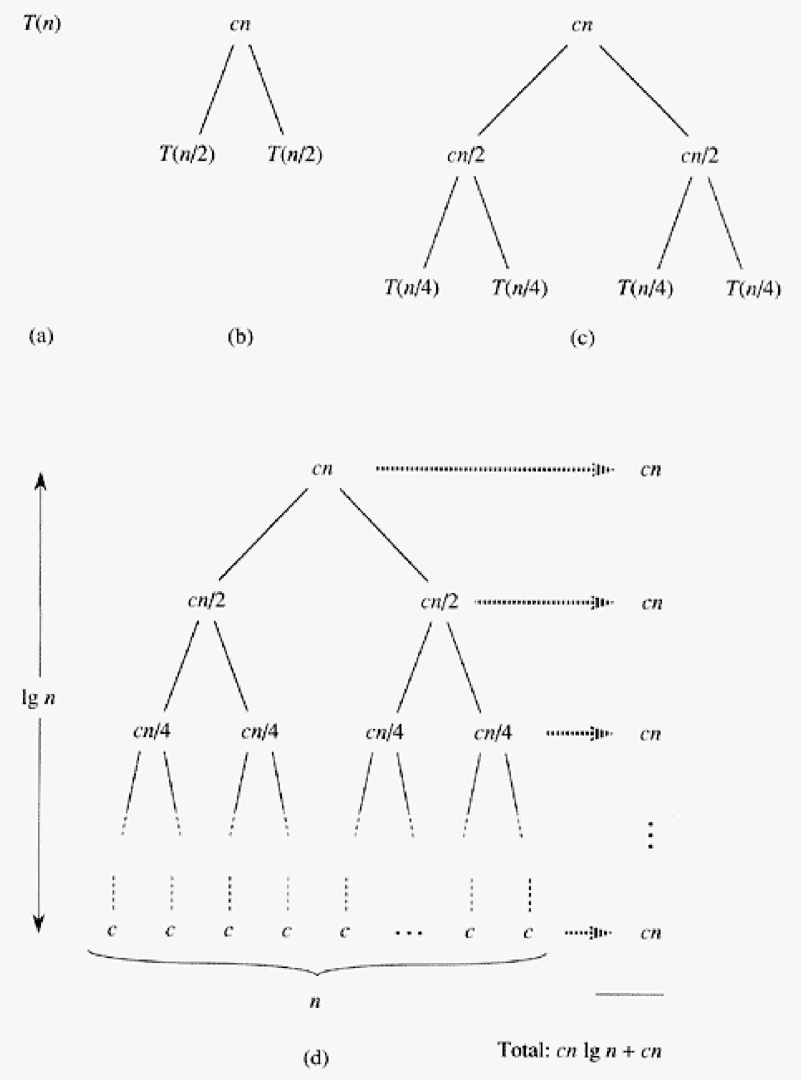


Figure 2.5 The construction of a recursion tree for the recurrence $T(n) = 2T(n/2) + cn$. Part **(a)** shows $T(n)$, which is progressively expanded in **(b)–(d)** to form the recursion tree. The fully expanded tree in part (d) has $\lg n + 1$ levels (i.e., it has height $\lg n$, as indicated), and each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

Problems to think about!

- What is the least number of comparisons you need to sort a list of 3 elements? 4 elements? 5 elements?
- How to arrange a tennis tournament in order to find the tournament champion with the least number of matches? How many tennis matches are needed?