Data Structure and Algorithm CSE-225

Dr. Mohammad Abu Yousuf
yousuf@juniv.edu

Some Sorting Algorithm

Insertion sort (1)

Input: A sequence of n numbers $\langle a_1, a_2, \ldots, a_n \rangle$.

Output: A permutation (reordering) $\langle a'_1, a'_2, \ldots, a'_n \rangle$ of the input sequence such that $a'_1 \leq a'_2 \leq \cdots \leq a'_n$.

The numbers that we wish to sort are also known as the ***keys***. Although conceptually we are sorting a sequence, the input comes to us in the form of an array with n elements.

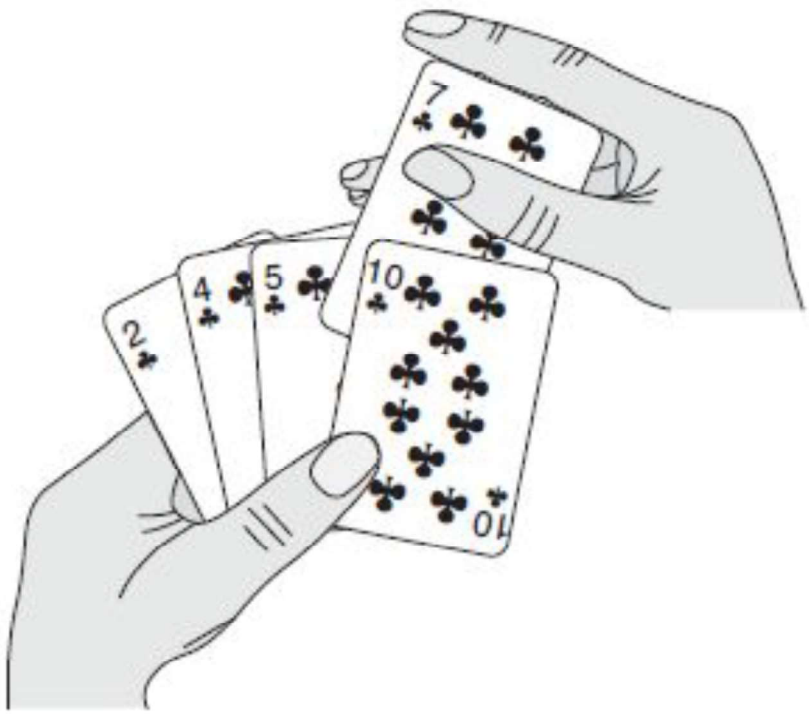


Figure 2.1 Sorting a hand of cards using insertion sort.

Insertion sort (2)

- Pseudocode for insertion sort as a procedure called INSERTIONSORT, which takes as a parameter an array A[1…n] containing a sequence of length n that is to be sorted. (In the code, the number n of elements in A is denoted by A.*length.)*
- *The algorithm sorts the input numbers* ***in place: it rearranges the*** numbers within the array A, with at most a constant number of them stored outside the array at any time.
- The input array A contains the sorted output sequence when the INSERTION-SORT procedure is finished.

Insertion sort (3)

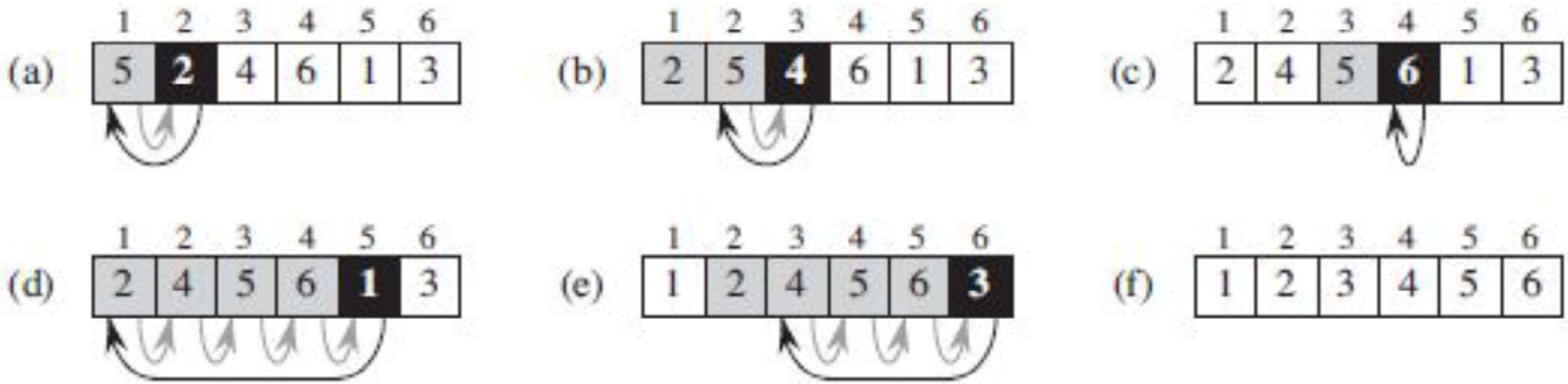


Figure 2.2 The operation of INSERTION-SORT on the array $A = \langle 5, 2, 4, 6, 1, 3 \rangle$. Array indices appear above the rectangles, and values stored in the array positions appear within the rectangles. **(a)–(e)** The iterations of the **for** loop of lines 1–8. In each iteration, the black rectangle 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 moves to in line 8. **(f)** The final sorted array.

Insertion sort (4)

```
INSERTION-SORT(A)
for j = 2 to A.length
    key = A[j]
    // Insert A[j] into the sorted sequence A[1 .. j − 1].
    i = j − 1
    while i > 0 and A[i] > key
        A[i + 1] = A[i]
        i = i − 1
    A[i + 1] = key
```

Insertion sort (5)

Loop invariants and the correctness of insertion sort

Figure 2.2 shows how this algorithm works for $A = \langle 5, 2, 4, 6, 1, 3 \rangle$. The index j indicates the "current card" being inserted into the hand. At the beginning of each iteration of the **for** loop, which is indexed by j, the subarray consisting of elements $A[1 \,..\, j-1]$ constitutes the currently sorted hand, and the remaining subarray $A[j+1 \,..\, n]$ corresponds to the pile of cards still on the table. In fact, elements $A[1 \,..\, j-1]$ are the elements *originally* in positions 1 through $j-1$, but now in sorted order. We state these properties of $A[1 \,..\, j-1]$ formally as a ***loop invariant***:

> At the start of each iteration of the **for** loop of lines 1–8, the subarray $A[1 \,..\, j-1]$ consists of the elements originally in $A[1 \,..\, j-1]$, but in sorted order.

Analyzing algorithms(1)

Analysis of insertion sort:

- The time taken by the INSERTION-SORT procedure depends on the input: sorting a thousand numbers takes longer than sorting three numbers.

- Moreover, INSERTIONSORT can take different amounts of time to sort two input sequences of the same size depending on how nearly sorted they already are.

- In general, the time taken by an algorithm grows with the size of the input, so it is traditional to describe the running time of a program as a function of the size of its input.

- To do so, we need to define the terms “running time” and “size of input” more carefully.

Analyzing algorithms(2)

- The best notion for ***input size depends on the problem being studied. For many*** problems, such as sorting or computing discrete Fourier transforms, the most natural measure is the *number of items in the input—for example, the array size* ***n*** for sorting.

- For many other problems, such as multiplying two integers, the best measure of input size is the *total number of bits needed to represent the input in* ordinary binary notation.

- Sometimes, it is more appropriate to describe the size of the input with two numbers rather than one. For instance, if the input to an algorithm is a graph, the input size can be described by the numbers of vertices and edges in the graph.

Analyzing algorithms(3)

- The ***running time of an algorithm on a particular input is the number of primitive*** operations or "steps" executed.

- It is convenient to define the notion of step so that it is as machine-independent as possible. For the moment, let us adopt the following view. A constant amount of time is required to execute each line of our pseudocode.

- One line may take a different amount of time than another line, but we shall assume that each execution of the *ith* line takes time c_i, where c_i is a constant.

Analyzing algorithms(4)

- We start by presenting the INSERTION-SORT procedure with the time ***"cost"*** of each statement and the number of times each statement is executed.

- For each ***j = 2,3,……,n,*** where ***n = A.length***, we let t_j denote the number of times the while loop test in line 5 is executed for that value of **j**. When a for or while loop exits in the usual way (i.e., due to the test in the loop header), the test is executed one time more than the loop body.
- We assume that comments are not executable statements, and so they take no time.

Analyzing algorithms(5)

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

Analyzing algorithms(6)

The running time of the algorithm is the sum of running times for each statement executed; a statement that takes c_i steps to execute and executes n times will contribute $c_i n$ to the total running time.[6] To compute $T(n)$, the running time of INSERTION-SORT on an input of n values, we sum the products of the *cost* and *times* columns, obtaining

$$\begin{aligned} T(n) &= c_1 n + c_2(n-1) + c_4(n-1) + c_5 \sum_{j=2}^{n} t_j + c_6 \sum_{j=2}^{n} (t_j - 1) \\ &\quad + c_7 \sum_{j=2}^{n} (t_j - 1) + c_8(n-1) \,. \end{aligned}$$

Analyzing algorithms(7)

Even for inputs of a given size, an algorithm's running time may depend on *which* input of that size is given. For example, in INSERTION-SORT, the best case occurs if the array is already sorted. For each $j = 2, 3, \ldots, n$, we then find that $A[i] \leq key$ in line 5 when i has its initial value of $j - 1$. Thus $t_j = 1$ for $j = 2, 3, \ldots, n$, and the best-case running time is

$$\begin{aligned} T(n) &= c_1 n + c_2(n-1) + c_4(n-1) + c_5(n-1) + c_8(n-1) \\ &= (c_1 + c_2 + c_4 + c_5 + c_8)n - (c_2 + c_4 + c_5 + c_8) . \end{aligned}$$

We can express this running time as $an + b$ for *constants* a and b that depend on the statement costs c_i; it is thus a ***linear function*** of n.

If the array is in reverse sorted order—that is, in decreasing order—the worst case results. We must compare each element $A[j]$ with each element in the entire sorted subarray $A[1 \,..\, j-1]$, and so $t_j = j$ for $j = 2, 3, \ldots, n$. Noting that

$$\sum_{j=2}^{n} j = \frac{n(n+1)}{2} - 1$$

and

$$\sum_{j=2}^{n} (j-1) = \frac{n(n-1)}{2}$$

(see Appendix A for a review of how to solve these summations), we find that in the worst case, the running time of INSERTION-SORT is

$$\begin{aligned} T(n) &= c_1 n + c_2(n-1) + c_4(n-1) + c_5\left(\frac{n(n+1)}{2} - 1\right) \\ &\quad + c_6\left(\frac{n(n-1)}{2}\right) + c_7\left(\frac{n(n-1)}{2}\right) + c_8(n-1) \\ &= \left(\frac{c_5}{2} + \frac{c_6}{2} + \frac{c_7}{2}\right)n^2 + \left(c_1 + c_2 + c_4 + \frac{c_5}{2} - \frac{c_6}{2} - \frac{c_7}{2} + c_8\right)n \\ &\quad - (c_2 + c_4 + c_5 + c_8)\,. \end{aligned}$$

We can express this worst-case running time as $an^2 + bn + c$ for constants a, b, and c that again depend on the statement costs c_i; it is thus a ***quadratic function*** of n.

Order of growth

- We used some simplifying abstractions to ease our analysis of the INSERTIONSORT procedure.

- First, we ignored the actual cost of each statement, using the constants c_i to represent these costs.

- Then, we observed that even these constants give us more detail than we really need: we expressed the worst-case running time as $an^2 + bn + c$ for some constants a, b, and c that depend on the statement costs c_i . We thus ignored not only the actual statement costs, but also the abstract costs c_i .

- We shall now make one more simplifying abstraction: it is the rate of growth, or order of growth, of the running time that really interests us.

- We therefore consider only the leading term of a formula (e.g., an^2), since the lower-order terms are relatively insignificant for large values of n.

Divide and Conquer algorithms

The divide-and-conquer approach

- For insertion sort, we used an ***incremental approach:*** ***having sorted the sub-array A[1…. j -1],*** we inserted the single element A[j] into its proper place, yielding the sorted sub-array A[1……j] .

- we examine an alternative design approach, known as "divide and conquer,". We'll use divide and conquer to design a sorting algorithm whose worst-case running time is much less than that of insertion sort.

The divide-and-conquer approach

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

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

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

  Combine the solutions to the sub problems into the solution for the original problem.

Merge sort algorithm

- 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

Merge sort algorithm

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 $A[p \,..\, r]$.

Merge sort algorithm

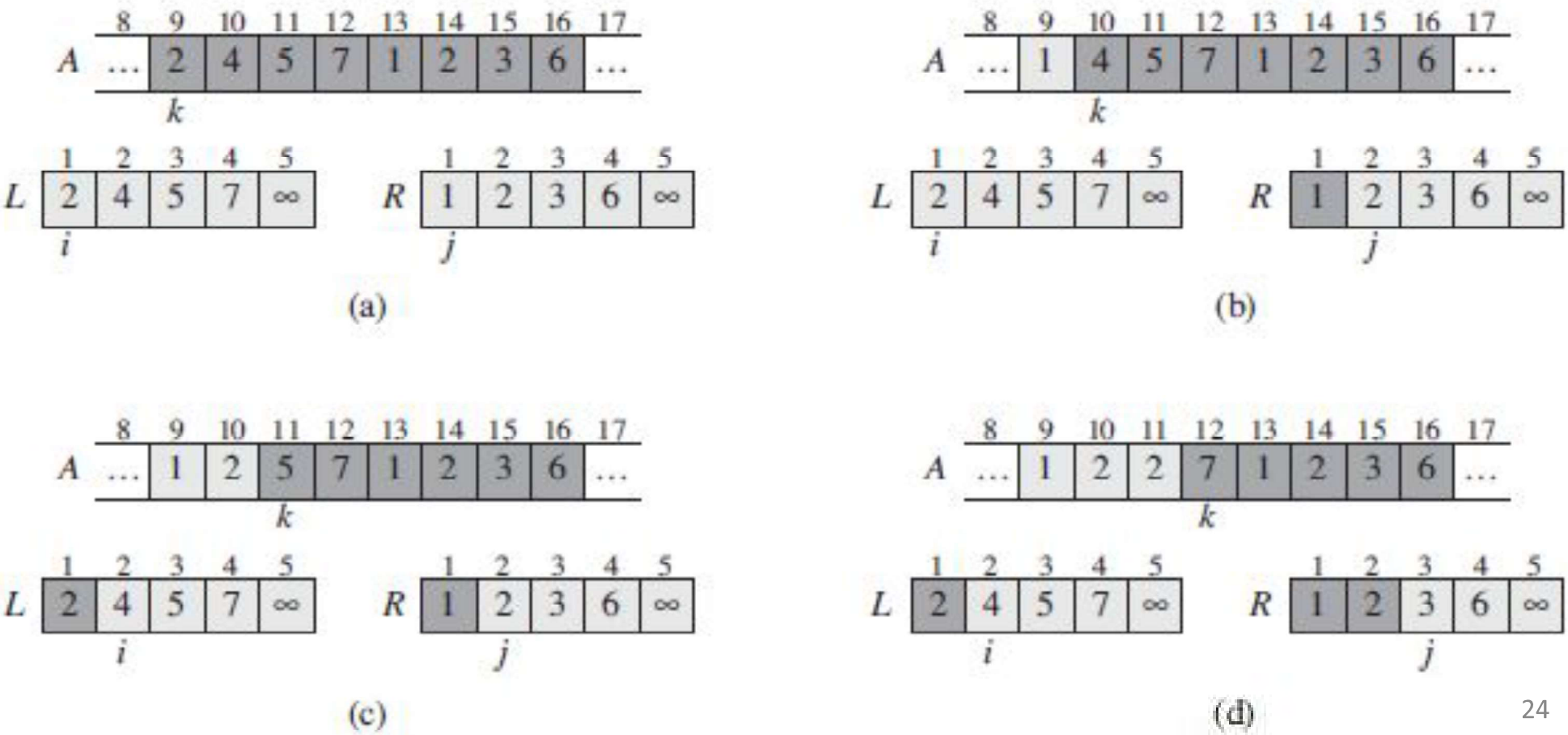
```
MERGE(A, p, q, r)
n1 = q - p + 1
n2 = r - q
let L[1 .. n1 + 1] and R[1 .. n2 + 1] be new arrays
for i = 1 to n1
    L[i] = A[p + i - 1]
for j = 1 to n2
    R[j] = A[q + j]
L[n1 + 1] = ∞
R[n2 + 1] = ∞
i = 1
j = 1
for k = p to r
    if L[i] ≤ R[j]
        A[k] = L[i]
        i = i + 1
    else A[k]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

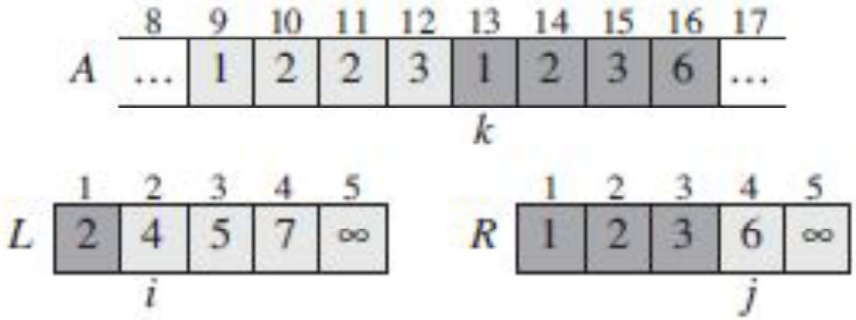
Merge sort algorithm

In detail, the MERGE procedure works as follows. Line 1 computes the length n_1 of the subarray $A[p \,..\, q]$, and line 2 computes the length n_2 of the subarray $A[q+1 \,..\, r]$. We create arrays L and R ("left" and "right"), of lengths $n_1 + 1$ and $n_2 + 1$, respectively, in line 3; the extra position in each array will hold the sentinel. The **for** loop of lines 4–5 copies the subarray $A[p \,..\, q]$ into $L[1 \,..\, n_1]$, and the **for** loop of lines 6–7 copies the subarray $A[q+1 \,..\, r]$ into $R[1 \,..\, n_2]$. Lines 8–9 put the sentinels at the ends of the arrays L and R. Lines 10–17, illustrated in Figure 2.3, perform the $r - p + 1$ basic steps by maintaining the following loop invariant:

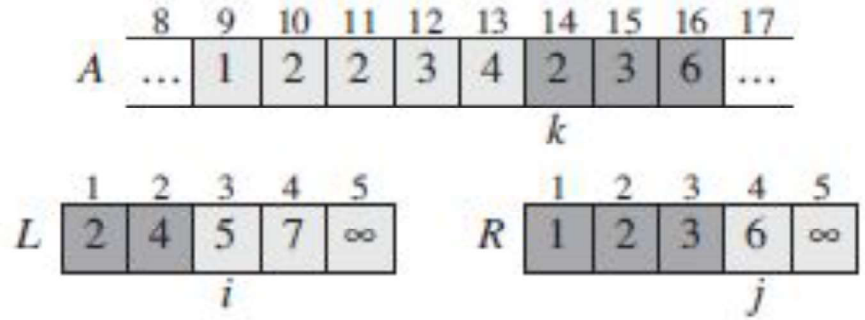
> At the start of each iteration of the **for** loop of lines 12–17, the subarray $A[p \,..\, k-1]$ contains the $k - p$ smallest elements of $L[1 \,..\, n_1 + 1]$ and $R[1 \,..\, n_2 + 1]$, in sorted order. Moreover, $L[i]$ and $R[j]$ are the smallest elements of their arrays that have not been copied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

```
MERGE(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
```

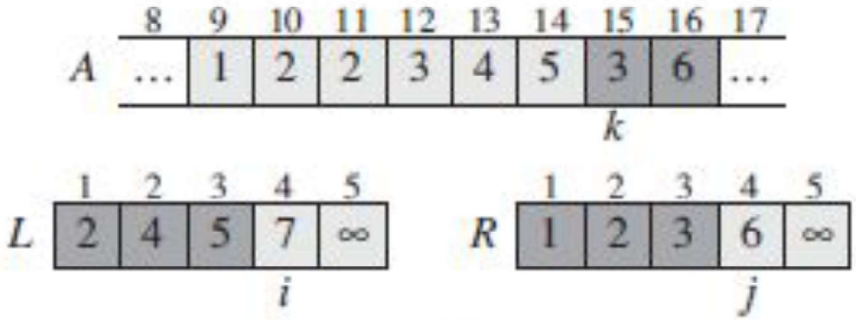



8 9 10 11 12 13 14 15 16 17
A … 1 2 2 3 1 2 3 6 …
k
1 2 3 4 5
L 2 4 5 7 ∞
i
1 2 3 4 5
R 1 2 3 6 ∞
j

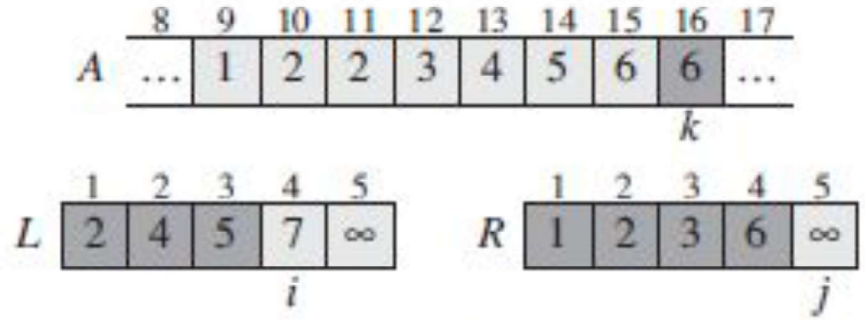
(e)


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

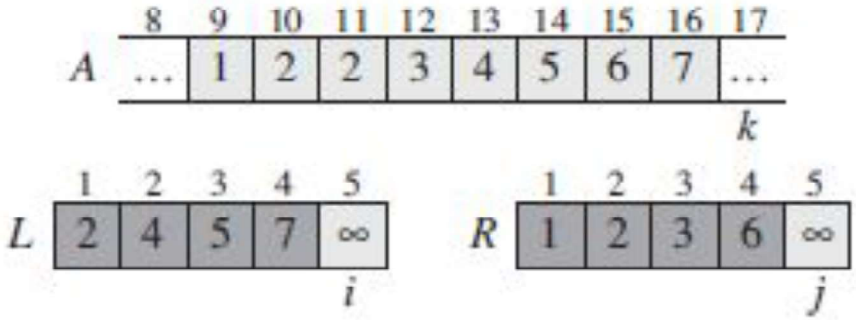
(f)


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

(g)


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

(h)


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

(i)

Merge sort algorithm

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

```
MERGE-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)
```

To sort the entire sequence $A = \langle A[1], A[2], \ldots, A[n] \rangle$, we make the initial call MERGE-SORT$(A, 1, A.length)$, where once again $A.length = n$. Figure 2.4 illustrates the operation of the procedure bottom-up when n is a power of 2. The algorithm consists of merging pairs of 1-item sequences to form sorted sequences of length 2, merging pairs of sequences of length 2 to form sorted sequences of length 4, and so on, until two sequences of length $n/2$ are merged to form the final sorted sequence of length n.

Merge sort algorithm

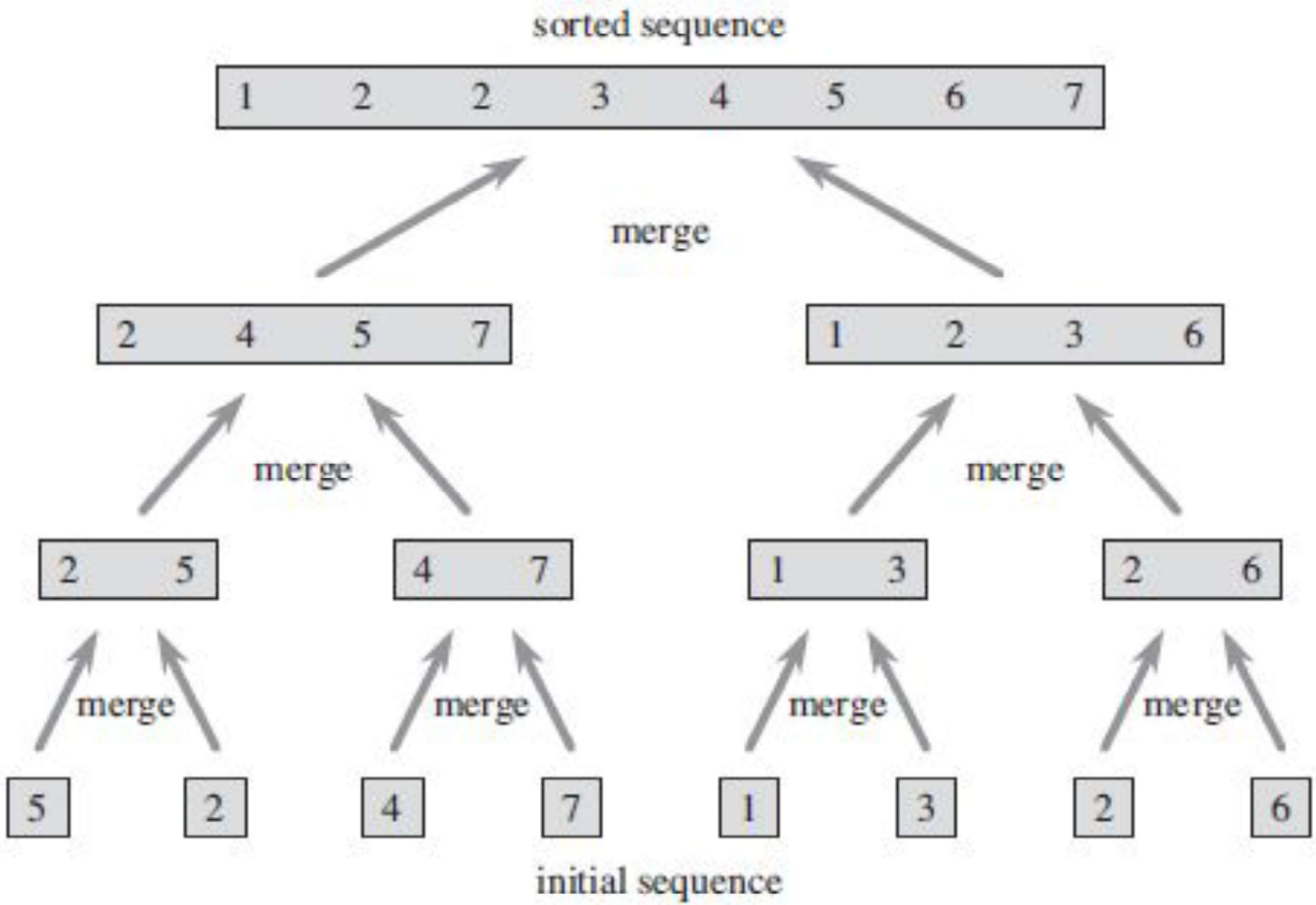


Figure 2.4 The operation of merge sort on the array $A = \langle 5, 2, 4, 7, 1, 3, 2, 6 \rangle$. The lengths of the sorted sequences being merged increase as the algorithm progresses from bottom to top.

Thank you