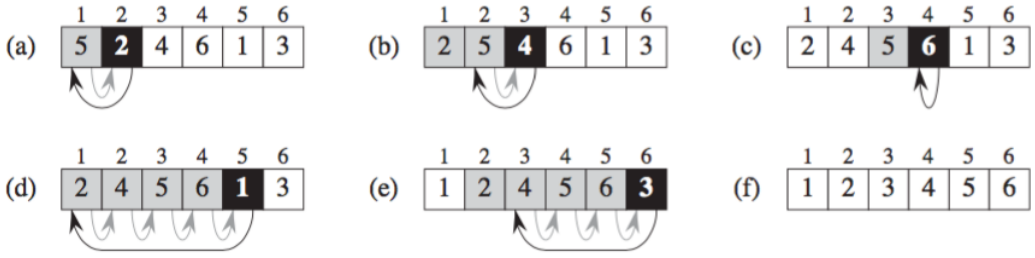
Lecture 2: Getting Started

Insertion Sort

- Our first algorithm is Insertion Sort
  - Solves the sorting problem
  - 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 $a'_1 \leq a'_2 \ldots, \leq a'_n$.
- We use pseudocode to to describe algorithms
  - Similar to C, C++, Java, Python, Pascal
  - Very descriptive
  - Sometimes sentences in English
- We will describe insertion sort with pseudocode
- How does insertion sort work?
  - Similar to the way we sort a hand of playing cards
  - Start with empty left hand, cards face down on the table
  - Take one card at a time and insert it to its correct position in the left hand
    - Compare card with cards already in the hand
    - From right to left
  - We always keep our cards in the left hand sorted
- Analysis of Insertion Sort
  - Execution time depends on the input
  - Input size
  - Is the input partially ordered?
- Input size
  - Number of elements in the input
  - A vector, the number of elements
  - Graphs, number of vertices, number of edges
- Insertion sort with input $A = \langle 5, 2, 4, 6, 1, 3 \rangle$
  - j indicates current card being inserted
  - $A[1 \ldots j-1]$, currently sorted hand
  - $A[j+1 \ldots n]$, cards still to be sorted
- Insertion sort works the way many people sort a hand of playing cards. We start with an empty left hand and the cards face down on the table. We then remove one card at a time from the table and insert it into the correct position in the left hand. To find the correct position for a card, we compare it with each of the cards already in the hand, from right to left.

```
INSERTION-SORT(A)
1. for j = 2 to A.length
2.     key = A[j]
3.     i = j − 1
4.     while i > 0 and A[i] > key
5.         A[i + 1] = A[i]
6.         i = i − 1
7.     A[i + 1] = key
```

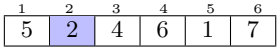
- We present our pseudocode for insertion sort as a procedure called Insertion-Sort, which takes as a parameter an array $A[1 \ldots n]$ containing a sequence of length n that is to be sorted. (In the code, the number n of elements in A is denoted by $A.length$.) The algorithm sorts the input numbers 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.



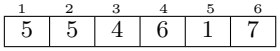
The operation of INSERTION-SORT on the array $A = \langle 5, 2, 4, 6, 1, 3 \rangle$. Array indices appear above the rectangles, and values stored in the array positions appear within the rectangles. (a)–(e) The iterations of the for loop of lines 1–7. In each iteration, the black rectangle holds the 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 7. (f) The final sorted array.

Example 1. We will use INSERTION-SORT to sort the sequence $A = \langle 5, 7, 9, 4, 6 \rangle$.

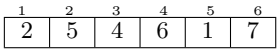
$j = 2$



$key = 2$
$i = 1$
$1 > 0$ and $5 > 2$ (True)

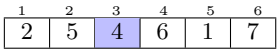


$A[2] = 5$
$i = 0$
$0 > 0$ (False)

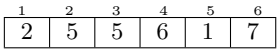


$A[1] = 2$

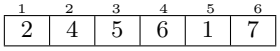
$j = 3$



$key = 4$
$i = 2$
$2 > 0$ and $5 > 4$ (True)

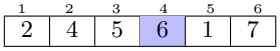


$A[3] = 5$
$i = 1$
$1 > 0$ and $2 > 4$ (False)

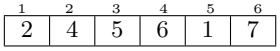


$A[2] = 4$

$j = 4$

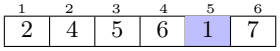


$key = 6$
$i = 3$
$3 > 0$ and $5 > 6$ (False)

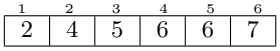


$A[4] = 6$

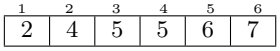
$j = 5$



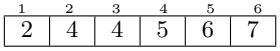
$key = 1$
$i = 4$
$4 > 0$ and $6 > 1$ (True)



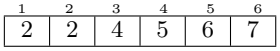
$A[5] = 6$
$i = 3$
$3 > 0$ and $5 > 1$ (True)



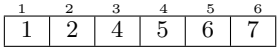
$A[4] = 5$
$i = 2$
$2 > 0$ and $4 > 1$ (True)



$A[3] = 4$
$i = 1$
$1 > 0$ and $2 > 1$ (True)

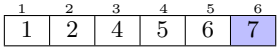


$A[2] = 3$
$i = 1$
$1 > 0$ and $1 > 1$ (False)

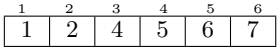


$A[1] = 1$

$j = 7$



$key = 7$
$i = 6$
$6 > 0$ and $6 > 7$ (False)



$A[7] = 7$

Insertion Sort - Loop Invariant

At the beginning of each iteration of the for loop, which is indexed by j, the subarray consisting of elements $A[1 \ldots j-1]$ constitutes the currently sorted hand, and the remaining subarray $A[j+1 \ldots n]$ corresponds to the pile of cards still on the table. In fact, elements $A[1 \ldots j-1]$ are the elements originally in positions 1 through $j-1$, but now in sorted order. We state these properties of $A[1 \ldots j-1]$ formally as a loop invariant:

> At the start of each iteration of the for loop of lines 1-7, the subarray $A[1 \ldots j-1]$ consists of the elements originally in $A[1 \ldots j-1]$, but in sorted order.

Initialization: We start by showing that the loop invariant holds before the first loop iteration, when $j = 2$. The subarray $A[1 \ldots j-1]$, therefore, consists of just the single element $A[1]$, which is in fact the original element in $A[1]$. Moreover, this subarray is sorted (trivially, of course), which shows that the loop invariant holds prior to the first iteration of the loop.

Maintenance: The body of the for loop works by moving $A[j-1]$, $A[j-2]$, $A[j-3]$, and so on by one position to the right until it finds the proper position for $A[j]$ (lines 4-6), at which point it inserts the value of $A[j]$ (line 7). The subarray $A[1 \ldots j]$ then consists of the elements originally in $A[1 \ldots j]$, but in sorted order. Incrementing j for the next iteration of the for loop then preserves the loop invariant.

Termination: Finally, we examine what happens when the loop terminates. The condition causing the for loop to terminate is that $j > A.length = n$. Because each loop iteration increases j by 1, we must have $j = n+1$ at that time. Substituting $n+1$ for j in the wording of loop invariant, we have that the subarray $A[1 \ldots n]$ consists of the elements originally in $A[1 \ldots n]$, but in sorted order. Observing that the subarray $A[1 \ldots n]$ is the entire array, we conclude that the entire array is sorted. Hence, the algorithm is correct.

Analyzing Insertion Sort

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.

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

$$T(n) = c_1 n + c_2(n-1) + c_3(n-1) + c_4 \sum_{j=2}^{n} t_j + c_5 \sum_{j=2}^{n} (t_j - 1) + c_6 \sum_{j=2}^{n} (t_j - 1) + c_7(n-1).$$

Best Case

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

$$\begin{aligned} T(n) &= c_1 n + c_2(n-1) + c_3(n-1) + c_4(n-1) + c_7(n-1) \\ &= (c_1 + c_2 + c_3 + c_4 + c_7)n - (c_2 + c_3 + c_4 + c_7) \\ &= O(n). \end{aligned}$$

Worst Case

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 $A[1 \ldots j-1]$, and so $t_j = j$ for $j = 2, 3, \ldots n$. We find that in the worst case, the running time of Insertion-Sort is

$$\begin{aligned} T(n) &= c_1 n + c_2(n-1) + c_3(n-1) + c_4 \left(\frac{n(n+1)}{2} - 1 \right) \\ &\quad + c_5 \left(\frac{n(n-1)}{2} - 1 \right) + c_6 \left(\frac{n(n-1)}{2} - 1 \right) + c_7(n-1) \\ &= \left(\frac{c_4}{2} + \frac{c_5}{2} + \frac{c_6}{2} \right) n^2 + \left(c_1 + c_2 + c_3 + \frac{c_4}{2} - \frac{c_5}{2} - \frac{c_6}{2} + c_7 \right) n \\ &\quad - (c_2 + c_3 + c_4 + c_7) \\ &= O(n^2). \end{aligned}$$

Analyzing Algorithms

- Predicting the resources that the algorithm requires
- Mostly computational time
- We analyze several candidates and choose the best one (most efficient) for the problem at hand
- Need to know the model of the implementation technology
- We assume a generic one processor
- Random-access machine (RAM), one instruction after the other with no concurrent operations
- We measure running time
- Number of primitive operations (steps) executed
- Need to define the notion of step in a way independent from the computer. Constant time to execute each pseudocode line. Add, Multiply, divide, floor, remainder, and so on. They actually take different time but we treat it as the same and as constant time.
- Order of Growth
  - We simplify the analysis of algorithms
    - We use constants to represent the cost of lines of code
  - Another simplification (abstraction)
    - We only care about the growing rate or order of growth
    - We only consider the leading term in a formula (i.e. an^2)
  - We also ignore constant coefficients
    - They are less significant than the growing rate
  - Worst case for InsertionSort: $\theta(n^2)$
  - Theta of n-squared
  - We say that an algorithm is more efficient than another one if its worst case running time has a lower order of growth
    - There might be inconsistencies for short input sizes but not for large ones
  - Many techniques to design algorithms
  - InsertionSort follows an incremental approach
  - Other techniques such as Divide and conquer
    - 3 steps in each level of recursion
    - Divide the problem in subproblems
    - Conquer the problems by recursively solving them, if the problem is trivial (small enough) it solves it directly
    - Combine the solutions to obtain the solution to the original problem

Designing Algorithms

- Many techniques to design algorithms
  - InsertionSort follows an incremental approach
  - Other techniques such as Divide and conquer
- Divide and Conquer
  - 3 steps in each level of recursion
  - Divide the problem in subproblems
  - Conquer the problems by recursively solving them, if the problem is trivial (small enough) it solves it directly
  - Combine the solutions to obtain the solution to the original problem
- Many algorithms are recursive in their structure
  - Recursive algorithms call themselves once or more times to solve similar subproblems
  - They follow a divide and conquer approach
    - Divide the problem in similar subproblems but smaller
    - Solve the problems recursively
    - Combine the solutions to create the solution to the original problem

The Merge Sort Algorithm

- MergeSort follows the divide-and-conquer paradigm
  - Divides the sequence of n-elements to sort in 2 subsequences of $n/2$ elements each
  - Sorts the 2 subsequences recursively using MergeSort
  - Combines, merges the 2 sorted subsequences to produce the sorted solution
- Initially A is the input array, $p = 1$, $q = length[A]$, $p \leq q < r$

MERGE(A, p, q, r)

1. $n_1 = q - p + 1$
2. $n_2 = r - q$
3. let $L[1 \cdots n_1 + 1]$ and $R = [1 \cdots n_2 + 1]$ be new arrays
4. **for** $i = 1$ **to** n_1
5. $\quad L[i] = A[p + i - 1]$
6. **for** $j = 1$ **to** n_2
7. $\quad R[j] = A[q + j]$
8. $L[n_1 + 1] = \infty$
9. $R[n_2 + 1] = \infty$
10. $i = 1$
11. $j = 1$
12. **for** $k = p$ **to** r
13. $\quad$ **if** $L[i] \leq R[j]$
14. $\quad\quad A[k] = L[i]$
15. $\quad\quad i = i + 1$
16. $\quad$ **else** $A[k] = R[j]$
17. $\quad\quad j = j + 1$

MERGE SORT(A, p, r)

1. if $p < r$
2. $\quad q = (p + r)/2$
3. $\quad$ MERGE-SORT(A, p, q)
4. $\quad$ MERGE-SORT$(A,\ q + 1,\ r)$
5. $\quad$ MERGE(A, p, q, r)

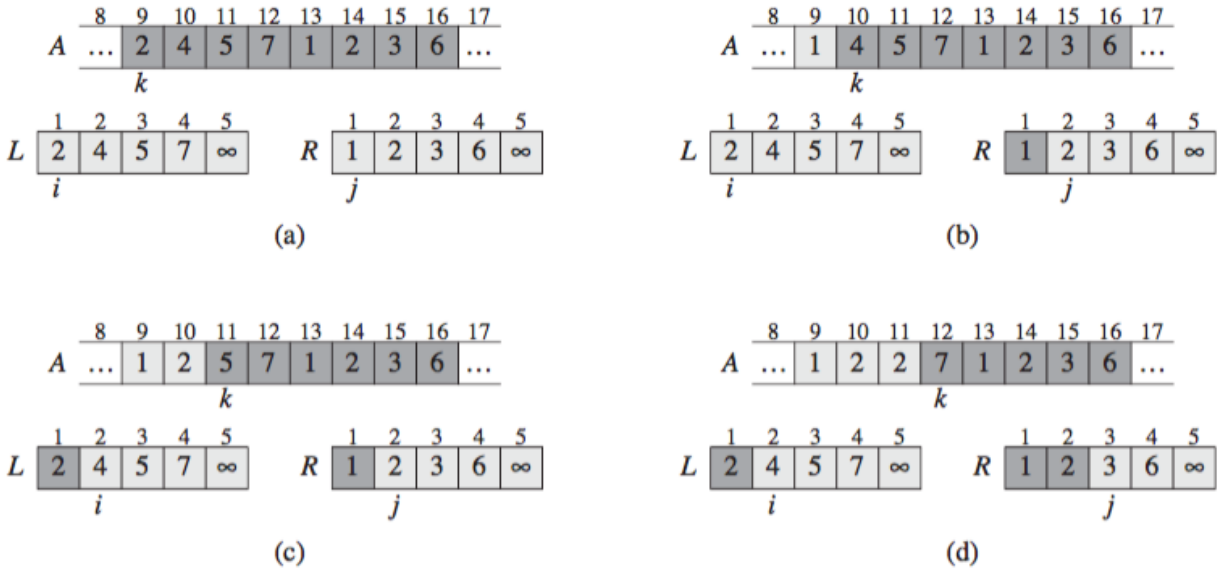
- Recursive algorithm
  - Its execution time is described with a recurrence equation. Describes the execution time of a problem of size n in terms of the execution time of smaller inputs We use mathematical tools to solve recurrences
- A recurrence for running time of a divide-and-conquer algorithm
  - Comes from the 3 steps of the basic paradigm
  - Let $T(n)$ be the running time of a problem of size n
  - If problem enough small (n ¡ c), it takes constant time: $\theta(1)$
- A recurrence for running time of a divide-and-conquer algorithm
  - Each problem division has a subproblems, each $1/b$ of the original size, in MergeSort $a = b = 2$
  - It takes $T(n/b)$ time to solve a subproblem of size n/b
  - It takes $aT(n/b)$ to solve a of them
  - $D(n)$ is the time to divide the problem into subproblems
  - $C(n)$ is the time to combine the solutions

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

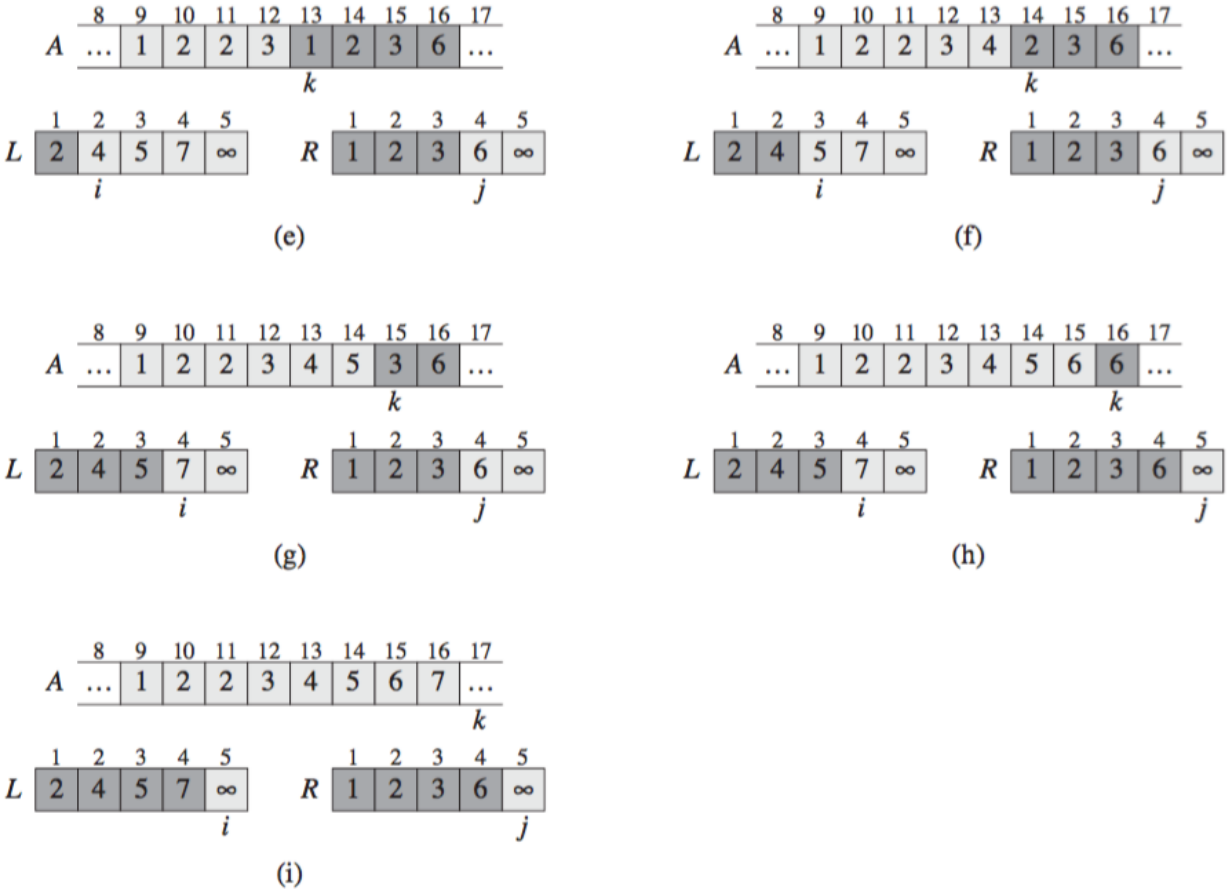
- Analysis of MergeSort
  - We assume problem size as a power of 2
  - Each problem division generates 2 subsequences of size $n/2$
  - Worst time execution time for MergeSort with n numbers
  - MergeSort with only one number → Constant time
  - For $n > 1$ we divide the problem:
  - Divide → Computes the middle of subarray in constant time, $D(n) = \theta(1)$
  - Conquer: Recursively solve 2 subproblems, each of size $n/2$, contributes $2T(n/2)$ to the running time
  - Combine: Merge on n-element subarray takes $\theta(n)$, then $C(n) = \theta(n)$
  - We sum functions $D(n)$ and $C(n)$ for the analysis
  - Sum $\theta(n)$ and $\theta(1)$
  - We add the term $2T(n/2)$ to get the worst case for MergeSort

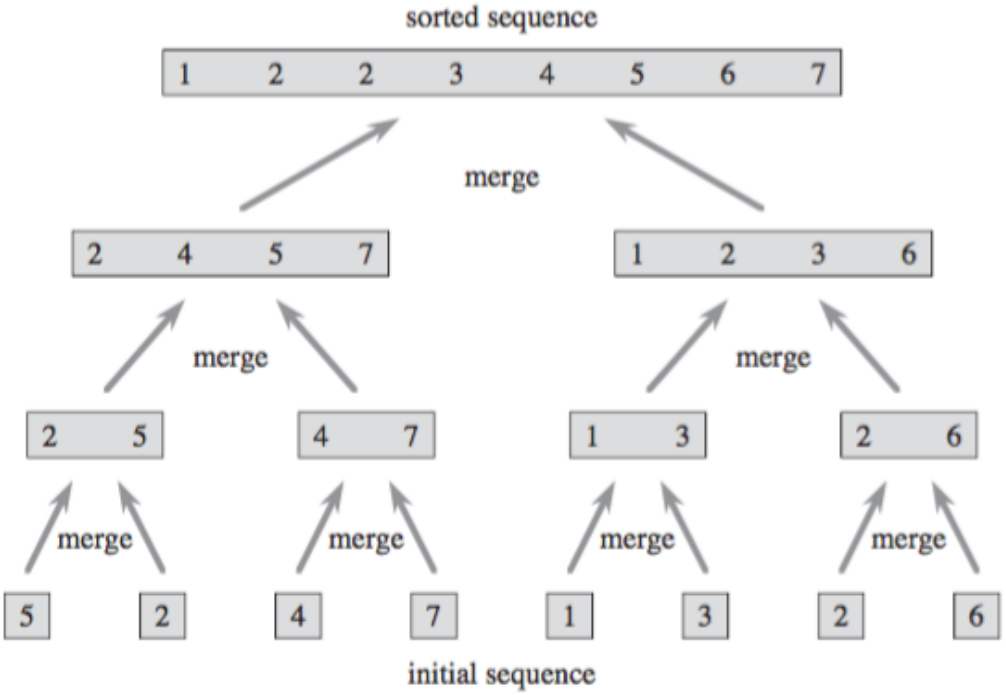
$$T(n) = \begin{cases} \theta(1) & \text{if } n = 1 \\ 2T(n/2) + \theta(n) & n > 1 \end{cases}$$

  - Solution for recurrence: $T(n)$ is $\theta(n \lg n)$
  - For large enough inputs, MergeSort with $\theta(n \lg n)$ is better than InsertionSort with $\theta(n^2)$



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





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

MERGE(A, p, q, r)

1. $n_1 = q - p + 1$
2. $n_2 = r - q$
3. let $L[1 \cdots n_1 + 1]$ and $R = [1 \cdots n_2 + 1]$ be new arrays
4. **for** $i = 1$ **to** n_1
5. $\quad L[i] = A[p + i - 1]$
6. **for** $j = 1$ **to** n_2
7. $\quad R[j] = A[q + j]$
8. $L[n_1 + 1] = \infty$
9. $R[n_2 + 1] = \infty$
10. $i = 1$
11. $j = 1$
12. **for** $k = p$ **to** r
13. $\quad$ **if** $L[i] \le R[j]$
14. $\quad\quad A[k] = L[i]$
15. $\quad\quad i = i + 1$
16. $\quad$ **else** $A[k] = R[j]$
17. $\quad\quad j = j + 1$

In detail, the MERGE procedure works as follows. Line 1 computes the length n_1 of the subarray $A[p \ldots q]$, and line 2 computes the length n_2 of the subarray $A[q+1 \ldots 2]$. We create arrays L and R, 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 $A[p \ldots q]$ into $L[1 \ldots n_1]$, and the for loop of lines 6–7 copies the subarray $A[q+1 \ldots r]$ into $R[1 \ldots n_2]$. Lines 8–9 put the sentinels at the ends of the arrays L and R. Lines 10–17, illustrated above, perform the basic steps by maintaining the following loop invariant:

Loop Invariant
At the start of each iteration of the for loop of lines 12–17, the subarray Ap::k 1 contains the k p smallest elements of L1::n1 C 1 and R1 : : n2 C 1 , in sorted order. Moreover, Li and Rj 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 loop of lines 12–17, that each iteration of the loop maintains the invariant, and that the invariant provides a useful property to show correctness when the loop terminates.

Initialization: Prior to the first iteration of the loop, we have $k = p$, so that the subarray $A[p \ldots k - 1]$ 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.

Maintenance: To see that each iteration maintains the loop invariant, let us first suppose that $L[i] \le R[j]$. Then $L[i]$ is the smallest element not yet copied back into A. Because $A[p \ldots -1]$ contains the $k-p$ smallest elements, after line 14 copies $L[i]$ into $A[k]$, the subarray $A[p \ldots k]$ 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 $A[p \ldots k - 1]$, which is $A[p \ldots r]$, contains the $k - p = r - p + 1$ smallest elements of $L[1 \ldots n_1 + 1$ and $R[1 \ldots n_2 + 1$, 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.