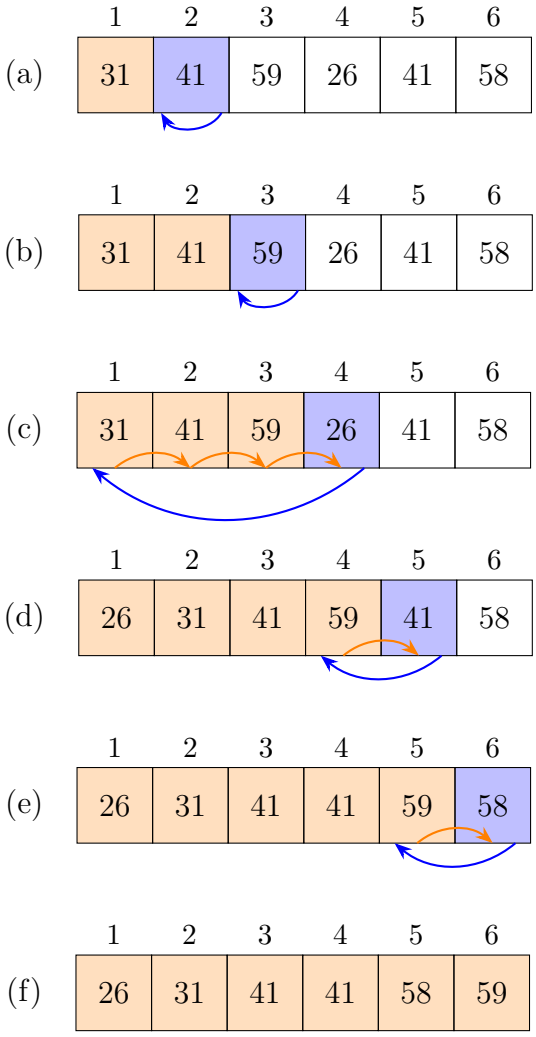
Exercises in Section 2.1

1. Using Figure 2.2 as a model, illustrate the operation of INSERTION-SORT on an array initially containing the sequence $\langle 31, 41, 59, 26, 41, 58 \rangle$.

Solution



2. Consider the procedure SUM-ARRAY on the facing page. It computes the sum of the n numbers in array $A[1:n]$. State a loop invariant for this procedure, and use its initialization, maintenance, and termination properties to show that the SUM-ARRAY procedure returns the sum of the numbers in $A[1:n]$.

Solution

Algorithm 1 SUM-ARRAY(A, n)

```
1: sum = 0
2: for i = 1 to n do
3:     sum = sum + A[i]
4: end for
5: return sum
```

Loop invariant: At the start of each iteration, sum is equal to the sum of the first i numbers in A.
Initialization: Before the first iteration, $sum = 0$.
Maintenance: Suppose that at the start of the i-th iteration, $sum = \sum_{j=1}^{i-1} A[j]$. Then at the start of the $(i+1)$-th iteration, $sum = \sum_{j=1}^{i} A[j]$.
Termination: When $i = n$, the loop terminates. Then $sum = \sum_{j=1}^{n} A[j]$.

3. Rewrite the INSERTION-SORT procedure to sort into monotonically decreasing instead of monotonically increasing order.

Solution

Algorithm 2 INSERTION-SORT(A, n)

```
1: for i = 2 to n do
2:     key = A[i]
3:     j = i − 1
4:     while j > 0 and A[j] < key do
5:         A[j + 1] = A[j]
6:         j = j − 1
7:     end while
8:     A[j + 1] = key
9: end for
```

4. Consider the *searching problem*: **Input:** A sequence of n numbers $\langle a_1, a_2, \ldots, a_n \rangle$ stored in array $A[1:n]$ and a value x. **Output:** An index i such that $x = A[i]$, or the special value NIL if x does not appear in A. Write pseudocode for *linear search*, which scans through the array from beginning to end, looking for x. Using a loop invariant, prove that your algorithm is correct. Make sure that your loop invariant fulfills the three necessary properties.

Solution

Algorithm 3 LINEAR-SEARCH(A, n, x)

```
1: for i = 1 to n do
2:     if A[i] = x then
3:         return x
4:     end if
5: end for
6: return NIL
```

5. Consider the problem of adding two n-bit binary integers a and b, stored in two n-element arrays $A[0:n-1]$ and $B[0:n-1]$, where each element is either 0 or 1, $a = \sum_{i=0}^{n-1} A[i] \cdot 2^i$ and $b = \sum_{i=0}^{n-1} B[i] \cdot 2^i$. The sum $c = a + b$ of the two integers should be stored in binary form in an $(n+1)$-element array $C[0:n]$, where $c = \sum_{i=0}^{n} C[i] \cdot 2^i$. Write a procedure ADD-BINARY-INTEGERS that takes as input arrays A and B, along with the length n, and returns array C holding the sum.

Solution

Algorithm 4 ADD-BINARY-INTEGERS(A, B, C, n)

```
1: Initialize C with all zeros.
2: for i = 0 to n − 1 do
3:     C[i] = A[i] + B[i] + C[i]
4:     if C[i] > 1 then
5:         C[i + 1] = 1
6:         C[i] = C[i] − 2
7:     end if
8: end for
9: return NIL
```

Exercises in Section 2.2

1. Express the function $n^3/1000 + 100n^2 - 100n + 3$ in terms of Θ-notation.

   Solution

   $n^3/1000 + 100n^2 - 100n + 3 = \Theta(n^3)$.

2. Consider sorting n numbers stored in array $A[1:n]$ by first finding the smallest element of $A[1:n]$ and exchanging it with the element in $A[1]$. Then find the smallest element of $A[2:n]$, and exchange it with $A[2]$. Then find the smallest element of $A[3:n]$, and exchange it with $A[3]$. Continue in this manner for the first $n-1$ elements of A. Write pseudocode for this algorithm, which is known as *selection sort*. What loop invariant does this algorithm maintain? Why does it need to run for only the first $n-1$ elements, rather than for all n elements? Give the worst-case running time of selection sort in Θ-notation. Is the best-case running time any better?

   Solution

Algorithm 5 SELECTION-SORT(A, n)

```
1:  for i = 1 to n − 1 do
2:      min = i
3:      for j = i + 1 to n do
4:          if A[j] < A[min] then
5:              min = j
6:          end if
7:      end for
8:      temp = A[i]
9:      A[i] = A[min]
10:     A[min] = temp
11: end for
12: return A
```

Both the worst case running time and the best case running time are $\Theta(n^2)$.

3. Consider linear search again (see Exercise 2.1-4). How many elements of the input array need to be checked on average, assuming that the element being searched for is equally likely

to be any element in the array? How about in the worst case? Using Θ-notation, give the average-case and worst-case running times of the linear search. Justify your answers.

Solution

If each element is equally likely to be any element in the array, then the average number of elements checked is $(n+1)/2$. In the worst case, the number of elements checked is n. Therefore, both the average-case and worst-case running times are $\Theta(n)$.

4. How can you modify any sorting algorithm to have a good best-case running time?

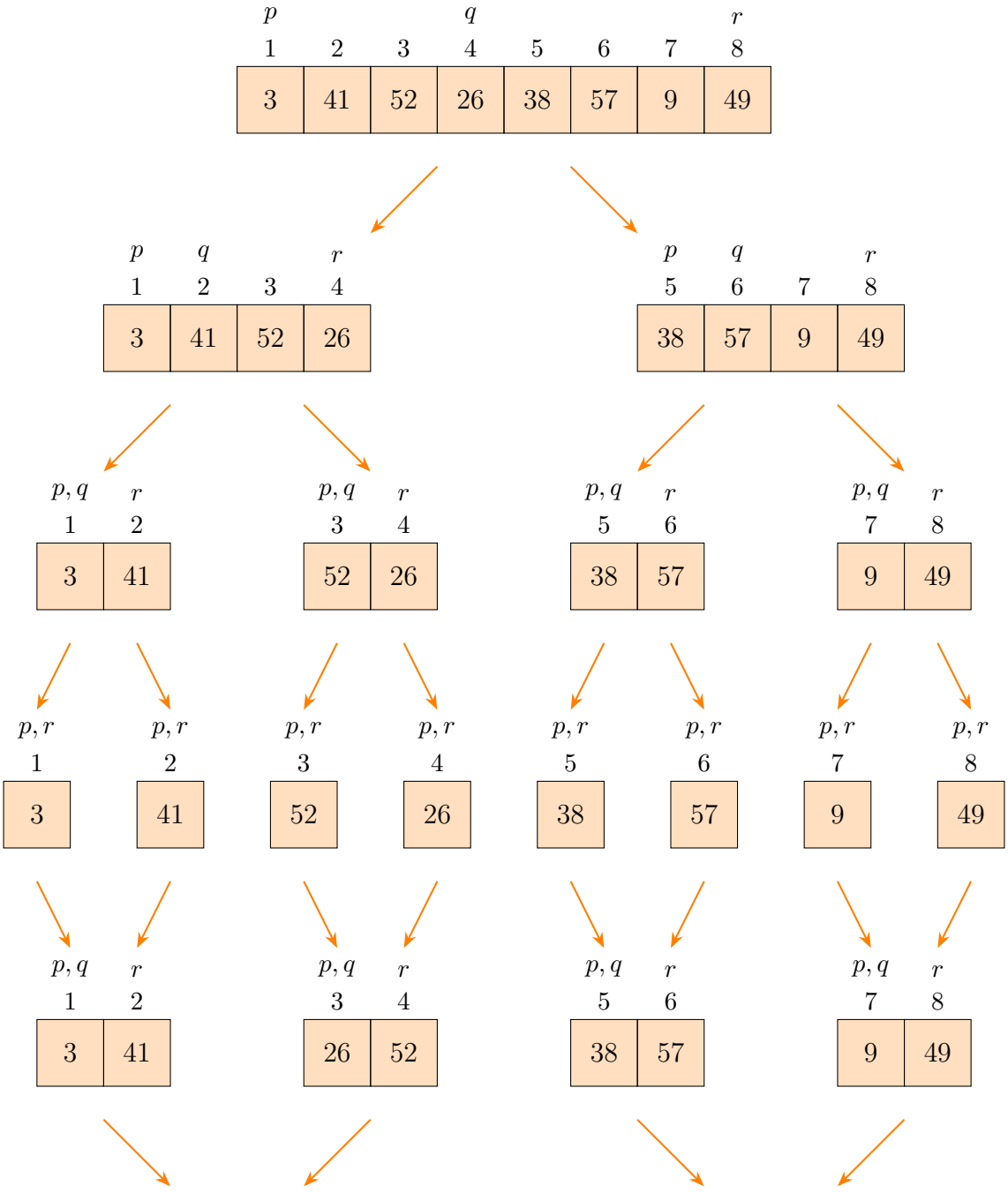
Solution

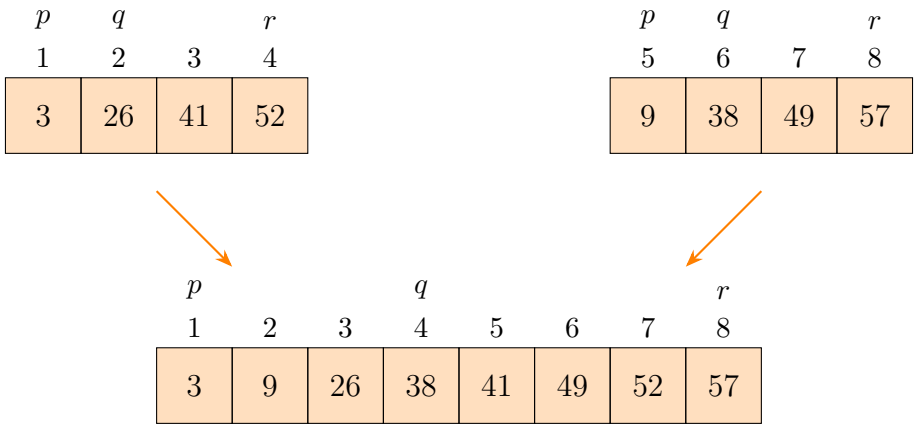
Just add a check to see if the array is already sorted.

Exercises in Section 2.3

1. Using Figure 2.4 as a model, illustrate the operation of merge sort on an array initially containing the sequence $\langle 3, 41, 52, 26, 38, 57, 9, 49 \rangle$.

Solution





2. The test in line 1 of the MERGE-SORT procedure reads "if $p \geq r$" rather than "if $p \neq r$." If MERGE-SORT is called with $p > r$, then the subarray $A[p:r]$ is empty. Argue that as long as the initial call of MERGE-SORT$(A, 1, n)$ has $n \geq 1$, the test "if $p \neq r$" suffices to ensure that no recursive call has $p > r$.

Solution

We need to show that $p \leq r$ in every call of MERGE-SORT. In the initial call, $p = 1$ and $r = n$. Given A is nonempty ($n \geq 1$), $p \leq r$.

Now, suppose $p \leq r$ at the start of an iteration, MERGE-SORT is recursively called if and only if $p < r$ (therefore $p \leq r - 1$). We have

$$q = \lfloor (p + r)/2 \rfloor \geq \lfloor p \rfloor = p,$$

and

$$q = \lfloor (p + r)/2 \rfloor \leq \lfloor ((r - 1) + r)/2 \rfloor = \lfloor r - 1/2 \rfloor = r - 1.$$

Hence, in both recursive calls there is $p \leq r$.

3. State a loop invariant for the **while** loop of lines 12–18 of the MERGE procedure. Show how to use it, along with the **while** loops of lines 20–23 and 24–27, to prove that the MERGE procedure is correct.

Solution

Loop Invariant: At the start of each iteration, elements from index p to $p + i + j - 1$ are sorted, and are smaller than or equal to elements from index $p + i + j$ to $r - 1$.

Initialization: At the start of the first iteration, $i + j = 0$. Vacuously true.

Maintenance: Assume the loop invariant holds for an iteration. Since both L and R are sorted, the smaller element of $L[i]$ and $R[j]$ is smaller than all unsorted elements, and by assumption larger than all sorted elements.

Termination: Elements from index p to $p+i+j-1$ are sorted, with $i = n_L - 1$ or $j = n_R - 1$.

Along with the **while** loops of lines 20–23 and lines 24-27, elements from index p to $r-1$ are sorted, yielding the correctness of the MERGE procedure.

4. Use mathematical induction to show that when $n \geq 2$ is an exact power of 2, the solution of the recurrence

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

is $T(n) = n \lg n$.

Solution

Suppose $n = 2^k$ for some $k \geq 1$.

Base case: If $k = 1$, $T(2) = 2 = 2 \lg 2$.

Inductive step: Assume $T(2^k) = 2^k \lg 2^k$. Then,

$$\begin{aligned} T(2^{k+1}) &= 2T(2^k) + 2^{k+1} \\ &= 2(2^k \lg 2^k) + 2^{k+1} \\ &= 2^{k+1} \lg 2^k + 2^{k+1} \\ &= 2^{k+1} \lg 2^{k+1}. \end{aligned}$$

5. You can also think of insertion sort as a recursive algorithm. In order to sort $A[1:n]$, recursively sort the subarray $A[1:n-1]$ and then insert $A[n]$ into the sorted subarray $A[1:n-1]$. Write pseudocode for this recursive version of insertion sort. Give a recurrence for its worst-case running time.

Solution

$$T(n) = \begin{cases} c_1, & \text{if } n = 1, \\ T(n-1) + c_2 n + c_3, & \text{if } n > 1. \end{cases}$$

Algorithm 6 INSERTION-SORT(A, n)

```
1: if n = 1 then
2:     return
3: end if
4: INSERTION-SORT(A, n − 1)
5: key = A[n]
6: i = n − 1
7: while i > 0 and A[i] < key do
8:     A[i + 1] = A[i]
9:     i = i − 1
10: end while
11: A[i + 1] = key
```

6. Referring back to the searching problem (see Exercise 2.1–4), observe that if the subarray being searched is already sorted, the searching algorithm can check the midpoint of the subarray against v and eliminate half of the subarray from further consideration. The *binary search* algorithm repeats this procedure, halving the size of the remaining portion of the subarray each time. Write pseudocode, either iterative or recursive, for binary search. Argue that the worst-case running time of binary search is $\Theta(\lg n)$.

   Solution

Algorithm 7 BINARY-SEARCH(A, n, x)

```
1: low = 1
2: high = n
3: while low ≤ high do
4:     mid = (low + high)/2
5:     if A[mid] = x then
6:         return mid
7:     else if A[mid] > x then
8:         high = mid − 1
9:     else
10:        low = mid + 1
11:    end if
12: end while
13: return -1
```

$$T(n) = \begin{cases} c_1, & \text{if } n = 1, \\ T(n/2) + c_2, & \text{if } n > 1. \end{cases}$$

There are roughly $\lg n$ recursive calls, so the running time is $c_2 \lg n + c_1 = \Theta(\lg n)$.

7. The **while** loop of lines 5–7 of the INSERTION-SORT procedure in Section 2.1 uses a linear search to scan (backward) through the sorted subarray $A[1:j-1]$. What if insertion sort used a binary search (see Exercise 2.3–6) instead of a linear search? Would that improve the overall worst-case running time of insertion sort to $\Theta(n \lg n)$?

Solution

No. Even we use binary search, we still need to iterate $i - 1$ times to place *key* to the correct place in the worst case.

Extra explanation: Even you use linked list instead of array, the complexity is still $\Theta(n^2)$. That is because binary search does not work for linked list.

8. Describe an algorithm that, given a set S of n integers and another integer x, determines whether S contains two elements that sum to exactly x. Your algorithm should take $\Theta(n \lg n)$ time in the worst case.

Solution

First, sort S using binary search. Time complexity: $\Theta(n \lg n)$.

Then, for each i, find $x - S[i]$ using binary search. Time complexity: $\Theta(n \lg n)$.