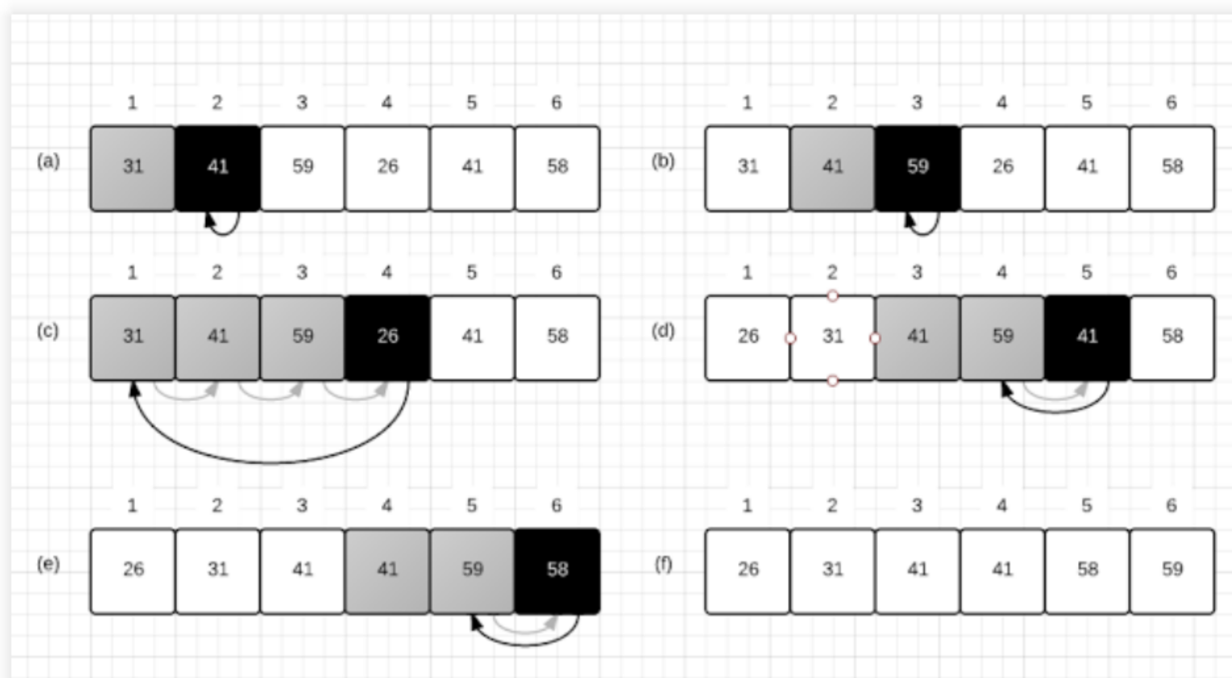
Training Problems #1 Solutions

2.1-1

Using Figure 2.2 as a model, illustrate the operation of INSERTION-SORT on the array $A = \langle 31, 41, 59, 26, 41, 58 \rangle$.

Solution:



2.1-2

Rewrite the INSERTION-SORT procedure to sort into nonincreasing instead of non-decreasing order.

Solution:

```
Insertion-Sort(A, n)
        for j = 2 to n
                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
```

2.1-3

Consider the ***searching problem***:

Input: A sequence of n numbers $A = \langle a_1, a_2, \ldots, a_n \rangle$ and a value ν.

Output: An index i such that $\nu = A[i]$ or the special value NIL if ν does not appear in A.

Write pseudocode for ***linear search***, which scans through the sequence, looking for ν. Using a loop invariant, prove that your algorithm is correct. Make sure that your loop invariant fulfills the three necessary properties.

Solution:

```
LINEAR-SEARCH(A, v)
1. i = 1
2. while i <= A.length
3.    if v == A[i]
4.       return i
5.    i = i + 1
6. return NIL
```

Loop invariant: At the start of each iteration of the while loop of lines 2-5, the algorithm has yet to find value v in subarray A[1..i - 1], which represents the searched elements of array A.

Initialization: We start by showing the loop invariant holds before the first loop iteration, when index i = 1. Since the algorithm has yet to begin searching for value v, subarray A[1..i - 1] correctly consists of zero elements and therefore cannot contain value v. Therefore, the loop invariant holds prior to the first iteration of the loop.

Maintenance: Next, we tackle the second property: showing each iteration maintains the loop invariant. The body of the while loop tests whether value v equals array element A[i] (line 3). If so, it exits with an output of index i. Otherwise, incrementing i for the next iteration of the while loop preserves the loop invariant, as the algorithm will now have fruitlessly searched subarray A[1..i - 1] for value v.

Termination: Finally, we examine what happens when the loop terminates. Condition i > A.length = n causes the while loop to terminate. Since each loop iteration increases i by 1, we must have i = n + 1 at that time. Substituting n + 1 for i in the wording of the loop invariant, we have the subarray A[1..n] consisting of the searched elements of array A. Observing subarray A[1..n] represents the entire array, we conclude v does not exist in A. At this point, the algorithm returns the special value NIL. Hence, the algorithm is correct.

2.2-1

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

Solution:

An easy way to determine the Θ-notation for a function – pick the most dominant term without considering its coefficient. Thus,
$n^3/1000 - 100n^2 - 100n + 3 = \Theta(n^3)$

2.2-2

Consider sorting n numbers stored in array A by first finding the smallest element of A and exchanging it with the element in $A[1]$. Then find the second smallest element of A, and exchange it with $A[2]$. 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 best-case and worst-case running times of selection sort in Θ-notation.

Solution:

```
SELECTION-SORT(A)
n = A.length
for j = 1 to n − 1
    smallest = j
    for i = j + 1 to n
        if A[i] < A[smallest]
            smallest = i
    exchange A[j] with A[smallest]
```

The algorithm maintains the loop invariant that at the start of each iteration of the outer **for** loop, the subarray $A[1 \,..\, j-1]$ consists of the $j-1$ smallest elements in the array $A[1 \,..\, n]$, and this subarray is in sorted order. After the first $n-1$ elements, the subarray $A[1 \,..\, n-1]$ contains the smallest $n-1$ elements, sorted, and therefore element $A[n]$ must be the largest element.

The running time of the algorithm is $\Theta(n^2)$ for all cases.