Section 2.1 – Insertion sort

2.1-1 Using Figure 2.2 as a model, illustrate the operation of INSERTION-SORT on the array $A = \langle 31, 41, 59, 26, 41, 58 \rangle$.

(a) 31 41 59 26 41 58

(b) 31 41 59 26 41 58

(c) 31 41 59 26 41 58

(d) 26 31 41 59 41 58

(e) 26 31 41 41 59 58

(f) 26 31 41 41 58 59

2.1-2 Rewrite the INSERTION-SORT procedure to sort into non-increasing instead of non-decreasing order.

The pseudocode is stated below.

Input: Array A

```
1 for j = 2 to A.length do
2 |   key = A[j]
3 |   i = j − 1
4 |   while i > 0 and A[i] > key do
5 |   |   A[i + 1] = A[i]
6 |   |   i = i − 1
7 |   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.

The pseudocode is stated below.

Input : Array A
Desired value ν
Output: Index i or NIL

```
1 for i = 1 to A.length do
2 |   if A[i] == ν then
3 |   |   return i
4 return NIL
```

Here is the *loop invariant*. At the start of each iteration of the **for** loop of lines 1–3, the algorithm assures that the subarray $A[1, \ldots, i-i]$ does not contain the element ν. Within each iteration, if $A[i]$ corresponds to the ν element, its index is returned.

- **Initialization.** Before the **for** loop, $i = 1$ and $A[1, \ldots, i-1]$ constains no element (therefore does not contain ν).
- **Maintenance.** The body of the **for** loop verifies if $A[i]$ corresponds to the ν element. If the element correspond to ν, its index is returned. Otherwise, incrementing i for the next iteration of the **for** loop then preserves the loop invariant.
- **Termination.** The **for** loop can terminate in one of the following conditions: (1) $A[i] = \nu$, which means that ν was found and its index is returned; (2) $i > A.length$ and, since each loop iteration increases i by 1, at that time we have $i = A.length + 1$ which assures (from the previous property) that $A[1, \ldots, A.length]$ does not contain the element ν.

2.1-4 Consider the problem of adding two n-bit binary integers, stored in two n-element arrays A and B. The sum of the two integers should be stored in binary form in an $(n+1)$-element array C. State the problem formally and write pseudocode for adding the two integers.

The pseudocode is stated below.

Input : Two integers, stored in two n-bit arrays (little endian) $A = \langle a_1, a_2, \ldots, a_n \rangle$ and $B = \langle b_1, b_2, \ldots, b_n \rangle$.
Output: A $(n+1)$-bit array $C = \langle c_1, c_2, \ldots, c_{n+1} \rangle$ storing the sum of the two aforementioned integers.

```
1 let C[1, ..., n + 1] be a new array
2 C[1] = 0
3 for i = 1 to A.length do
4 |   s = A[i] + B[i] + C[i]
5 |   C[i] = s  mod 2
6 |   C[i + 1] = s / 2
7 return C
```

Section 2.2 – Analyzing algorithms

2.2-1 Express the function $n^3/1000 - 100n^2 - 100n + 3$ in terms of Θ-notation.

$\Theta(n^3)$.

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.

The pseudocode is stated below.

Input: Array A.

```
1 for i = 1 to A.length − 1 do
2 |   smallest = i
3 |   for j = i + 1 to A.length do
4 |   |   if A[j] < A[smallest] then
5 |   |   |   smallest = j
6 |   tmp = A[i]
7 |   A[i] = A[smallest]
8 |   A[smallest] = tmp
```

Here is the *loop invariant*. At the start of each iteration of the **for** loop of lines 1–8, the subarray $A[1, \ldots, i-i]$ consists of the $(i-1)$ smallest elements of the array A in sorted order.

- **Initialization.** Before the **for** loop, $i = 1$ and $A[1, \ldots, i-1]$ constains no element.
- **Maintenance.** The body of the **for** loop looks on the subarray $A[i+1, \ldots, A.length]$ for a element that is smaller than $A[i]$. If a smaller element is found, their positions in A are exchanged. Since the subarray $A[1, \ldots, i-1]$ already contains the i smallest elements of A, the smaller element between $A[i]$ and $A[i+1, \ldots, A.length]$ is the i-th smallest element of A, which maintains our *loop invariant* for the subarray $[1, \ldots, i]$.
- **Termination.** The condition causing the **for** loop to terminate is that $i == A.length - 1$. At that time, $i = A.length = n$. Since (from the previous property) the subarray $A[1, \ldots, n-1]$ consists of the $(n-1)$ smaller elements A, the lasting element $A[n]$ can only be the n-th smaller element.

It needs to run only for the first $(n-1)$ element because, after that, the subarray $A[1, \ldots, n-1]$ consists of the $(n-1)$ smaller elements of A and the n-th element is already in the correct position.

Regardless of the content of the input array A, for $i = 1, 2, \ldots, (A.length - 1)$ the algorithm will always look for the i-th element in the whole subarray $A = [i+1, A.length]$. Thus, the algorithm takes $\Theta(n^2)$ for every input.

2.2-3 Consider linear search again (see Exercise 2.1-3). How many elements of the input sequence need to be checked on the average, assuming that the element being searched for is equally likely to be any element in the array? How about in the worst case? What are the average-case and worst-case running times of linear search in Θ-notation? Justify your answers.

Lets consider an array of size n, where each element is taken from the set $1, \ldots, k$. If k is not a function of n, its a constant. In the average case, each comparison has probability $1/k$ to find the element that is being searched, resulting in an average of k comparisons. Thus, in the average case, as a function of the input size, the algorithm takes $\Theta(k) = \Theta(1)$. The worst case occurs when $k >= n$, which takes $\Theta(n)$.

2.2-4 How can we modify almost any algorithm to have a good best-case running time?

Verify if the input is already solved. If it is solved, do nothing. Otherwise, solve it with some algorithm.