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.

```
   InsertionSortNonIncreasing($A$)
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.

```
   LinearSearch($A$, $\nu$)
1  for $i = 1$ to $A.length$ do
2      if $A[i] == \nu$ 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. Integers are stored in little endian format.

```
AddIntegers($A$, $B$)
    let $C[1, \dots, n+1]$ be a new array
    $C[1] = 0$
    **for** $i = 1$ **to** $A.length$ **do**
        $s = A[i] + B[i] + C[i]$
        $C[i] = s \mod 2$
        $C[i+1] = s \ / \ 2$
    **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.

```
SelectionSort(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.

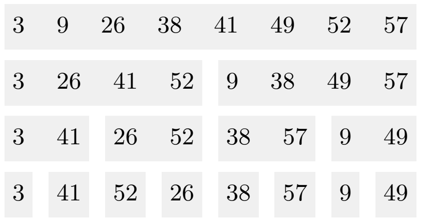
> Lets consider an array of size n, where each element is taken from the set $1, \ldots, k$. If k is not a function of n, its a constant. In the average case, each comparison has probability $1/k$ to find the element that is being searched, resulting in an average of k comparisons. Thus, in the average 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.

Section 2.3 – Analyzing algorithms

2.3-1 Using Figure 2.4 as a model, illustrate the operation of merge sort on the array $A = \langle 3, 41, 52, 26, 38, 57, 9, 49 \rangle$



2.3-2 Rewrite the Merge procedure so that it does not use sentinels, instead stopping once either array L or R has had all its elements copied back to A and then copying the remainder of the other array back into A.

The pseudocode is stated below.

```
Merge(A, p, q, r)
    $n_1 = q - p + 1$
    $n_2 = r - q$
    let $L[1, \ldots, n_1]$ and $R[1, \ldots, n_2]$ be new arrays
    for $i = 1$ to $n_1$ do
        $L[i] = A[p + i - 1]$
    for $j = 1$ to $n_2$ do
        $R[j] = A[q + j]$
    $i = 1$
    $j = 1$
    for $k = p$ to $r$ do
        if $q + j > r$ or $L[i] \leq R[j]$ then
            $A[k] = L[i]$
            $i = i + 1$
        else
            $A[k] = R[j]$
            $j = j + 1$
```

2.3-3 Use mathematical induction to show that when n is an exact power of 2, the solution of the recurrence

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

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

Answer.

2.3-4 We can express insertion sort as a recursive procedure as follows. In order to sort $A[1, \ldots, n]$, we recursively sort $A[1, \ldots, n-1]$ and then insert $A[n]$ into the sorted array $A[1, \ldots, n-1]$. Write a recurrence for the running time of this recursive version of insertion sort.

The recurrence is stated below.

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

It takes $\Theta(n^2)$.

2.3-5 Referring back to the searching problem (see Exercise 2.1-3), observe that if the sequence A is sorted, we can check the midpoint of the sequence against ν and eliminate half of the sequence from further consideration. The ***binary search*** algorithm repeats this procedure, halving the size of the remaining portion of the sequence each time. Write pseudocode, either iterative or recursive, for binary search. Argue that the worst-case running time of binary search is $\Theta(\lg n)$.

The pseudocode is stated below.

```
BinarySearch($A$, $s$, $e$, $\nu$)
    if $s > e$ then
        return NIL
    $m = \lfloor (s+e)/2 \rfloor$
    if $\nu > A[m]$ then
        BinarySearch($A$, $m+1$, $e$, $\nu$)
    else if $\nu > A[m]$ then
        BinarySearch($A$, $s$, $m-1$, $\nu$)
    else
        return $m$
```

In each recursion level, the algorithm compares ν with the central element $A[m]$. If $\nu = A[m]$, the element was found and it just returns the position. If $A[m]$ is bigger (or smaller) than ν, the algorithm discards the left half (or the right half) of the array and continues recursively in the remaining $\lfloor (n-1)/2 \rfloor$ elements. Each recursion element compares ν with a single element of A, thus each level takes $\Theta(1)$. Since the number of elements in the array is halved in each level, there will be at most $\lg n$ recursion levels. The algorithm then takes at most $\lg n \times \Theta(1) = \Theta(\lg n)$.

2.3-6 Observe that 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 $A[1,\ldots,j-1]$. Can we use a binary search (see Exercise 2.3-5) instead to improve the overall worst-case running time of insertion sort to $\Theta(n \lg n)$?

No, because even finding the correct position in $\lg n$, after each search the algorithm will still need to shift up to n the elements to keep the subarray $A[1,\ldots,j]$ sorted. The worst-case running time will remain $\Theta(n^2)$.

2.3-7 ($\star$) Describe a $\Theta(n \lg n)$-time algorithm that, given a set S of n integers and another integer x, determines whether or not there exist two elements in S whose sum is exactly x.

Start by sorting S using MergeSort, which takes $\Theta(n \lg n)$. For each element i of S, $i = 1,\ldots,n$, search the subarray $A[i+1,\ldots,n]$ for the element $\nu = x - S[i]$ using BinarySearch. If ν is found, return its position. Otherwise, continue for the next value of i. It will perform at most n searchs and each search takes $\Theta(\lg n)$. The algorithm then takes $\Theta(n \lg n) + n \times \Theta(\lg n) = \Theta(n \lg n)$.