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, ν)
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. Integers are stored in little endian format.

```
   AddIntegers(A, B)
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.

```
   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 $A = \langle 3, 41, 52, 26, 38, 57, 9, 49 \rangle$

3	9	26	38	41	49	52	57
3	26	41	52	9	38	49	57
3	41	26	52	38	57	9	49
3	41	52	26	38	57	9	49

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)
1  | n_1 = q − p + 1
2  | n_2 = r − q
3  | let L[1, ..., n_1] and R[1, ..., n_2] be new arrays
4  | for i = 1 to n_1 do
5  | |  L[i] = A[p + i − 1]
6  | for j = 1 to n_2 do
7  | |  R[j] = A[q + j]
8  | i = 1
9  | j = 1
10 | for k = p to r do
11 | |  if q + j > r or L[i] ≤ R[j] then
12 | |  |  A[k] = L[i]
13 | |  |  i = i + 1
14 | |  else
15 | |  |  A[k] = R[j]
16 | |  |  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 $T(n) = n \lg n$.

The base case is trivial, since $T(2) = 2 \lg 2 = 2$. To prove that it holds for $n > 2$ using mathematical induction, we need to show that if it holds for $n - 1$, it also holds for n. From the recurrence, $T(n) = 2T(n/2) + n$. But by inductive hypothesis, $T(n/2) = (n/2) \lg(n/2)$, so we get that:

$$\begin{aligned} T(n) &= 2T(n/2) + n \\ &= 2(n/2) \lg(n/2) + n \\ &= n \lg(n/2) + n \\ &= n(\lg(n) - \lg(2)) + n \\ &= n \lg(n) - n + n \\ &= n \lg(n). \end{aligned}$$

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 worst-case 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, ν)
1  |  if s > e then
2  |  |  return NIL
3  |  m = ⌊(s + e)/2⌋
4  |  if ν > A[m] then
5  |  |  BinarySearch(A, m + 1, e, ν)
6  |  else if ν > A[m] then
7  |  |  BinarySearch(A, s, m − 1, ν)
8  |  else
9  |  |  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 $\Theta(n \lg n) + n \times \Theta(\lg n) = \Theta(n \lg n)$.

Problems

2-1 ***Insertion sort on small arrays in merge sort***

Although merge sort runs in $\Theta(n \lg n)$ worst-case time and insertion sort runs in $\Theta(n^2)$ worst-case time, the constant factors in insertion sort can make it faster in practice for small problem sizes on many machines. Thus, it makes sense to ***coarsen*** the leaves of the recursion by using insertion sort within merge sort when subproblems become sufficiently small. Consider a modification to merge sort in which n/k sublists of length k are sorted using insertion sort and then merged using the standard merging mechanism, where k is a value to be determined.

a. Show that insertion sort can sort the n/k sublists, each of length k, in $\Theta(nk)$ worst-case time.

b. Show how to merge the sublists in $\Theta(n \lg(n/k))$ worst-case time.

c. Given that the modified algorithm runs in $\Theta(nk + n \lg(n/k))$ worst-case time, what is the largest value of k as a function of n for which the modified algorithm has the same running time as standard merge sort, in terms of Θ-notation?

d. How should we choose k in practice?

(a) Sort n/k sublists of length k with insertion sort takes $n/k \cdot \Theta(k^2) = \Theta(n/k \cdot k^2) = \Theta(nk)$.

(b) The naive solution is to extend the standard merging procedure to merge n/k sublists at the same time, instead of two. Since there is n/k sublists, in each iteration the algorithm takes $\Theta(n/k)$ to select the lowest element among all the sublists. Since there are n elements (thus n iterations), the total complexity is $n \cdot \Theta(n/k) = \Theta(n^2/k)$.

We can accomplish the requested $\Theta(n \lg(n/k))$ complexity by merging the sublists pairwise, rather than merging them all at the same time. Lets first consider the case in which the number of sublists is even. In the first level there will be $n/(2k)$ pairs of sublists to merge and, since each sublist has length k, each merge will take $\Theta(2k)$. Thus, the first level will take $n/(2k) \cdot \Theta(2k) = \Theta(n)$. The next level will have half the number of sublists and will take $n/(4k) \cdot \Theta(4k) = \Theta(n)$. Since the number of sublists is reduced by two on each level, the total number of levels will be $\lg(n/k)$. Thus, the total cost is $\Theta(n) \cdot \lg(n/k) = \Theta(n \lg(n/k))$. When the number of sublists is odd, it will need one additional level to merge the remaining sublist. Thus, the total cost is $\Theta(n \lg(\lceil n/k \rceil))$.

(c) When $k = 1$ (smallest possible value for k), the modified algorithm takes $\Theta(n \cdot 1 + n \lg(n/1)) = \Theta(n + n \lg n)$. When k grows, the first term grows and the second term decreases. Thus, since the second term can not be greater than $n \lg n$, we just need to pay attention to the first term. The algorithm then takes more than $\Theta(n \lg n)$ when $nk > n \lg n \to k > \lg n$. Thus, the largest value of k is $\lg n$.

(d) It depends of the constant factors of insertion sort and merge sort. Since the cost of these constants may vary between different machines, in practice one should choose the largest value of k in which insertion sort is faster then merge sort in a given machine.

2-2 ***Correctness of bubblesort***

Bubblesort is a popular, but inefficient, sorting algorithm. It works by repeatedly swapping adjacent elements that are out of order.

```
   BubbleSort(A)
1  | for i = 1 to A.length − 1 do
2  | |  for j = A.length downto i + 1 do
3  | |  |  if A[j] < A[j − 1] then
4  | |  |  |  exchange A[j] with A[j − 1]
```

a. Let A' denote the output of BubbleSort(A). To prove that BubbleSort is correct, we need to prove that it terminates and that

$$A'[1] \leq A'[2] \leq \cdots \leq A'[n].$$

where $n = A.length$. In order to show that BubbleSort actually sorts, what else do we need to prove?

b. State precisely a loop invariant for the **for** loop in lines 2–4, and prove that this loop invariant holds. Your proof should use the structure of the loop invariant proof presented in this chapter.

c. Using the termination condition of the loop invariant proved in part (b), state a loop invariant **for** the for loop in lines 1–4 that will allow you to prove in- equality (2.3). Your proof should use the structure of the loop invariant proof presented in this chapter.

d. What is the worst-case running time of bubblesort? How does it compare to the running time of insertion sort?

(a) A' must be a permutation of A.

(b) Here is the *loop invariant*. At the start of each iteration j of the for loop of lines 2–4, $A[j]$ is the smallest element of the subarray $A[j, \ldots, A.length]$.

- **Initialization.** Prior to the first iteration of the loop, $j = n = A.length$, so the subarray $A[j, \ldots, A.length]$ has only one element, and $A[j]$ is therefore the smallest element of the subarray $A[j, \ldots, A.length]$.
- **Maintenance.** To see that each iteration maintains the loop invariant, let's suppose that $A[j-1] > A[j]$. Because $A[j]$ is the smallest element of the subarray $A[j, \ldots, A.length]$, after line 4 exchanges the position of the elements $A[j]$ and $A[j-1]$, $A[j-1]$ will be the smallest element of the subarray $A[j-1, \ldots, A.length]$. Incrementing j (in the for loop update) reestablishes the loop invariant for the next iteration. If instead $A[j-1] < A[j]$, nothing needs to be done and $A[j-1]$ is already the smallest element of the subarray $A[j-1, \ldots, A.length]$.
- **Termination.** At termination, $j = i$. By the loop invariant $A[i]$ is the smallest element of the subarray $A[i, \ldots, A.length]$.

(c) Here is the *loop invariant*. At the start of each iteration i of the for loop of lines 1–4, the subarray $A[1, \ldots, i-1]$ consists of the i smallest elements of A in sorted order.

- **Initialization.** Prior to the first iteration of the loop, we have $i = 1$, so that the subarray $A[1, \ldots, i-1]$ is empty. This empty subarray contains the $i - 1 = 0$ smallest elements of A in sorted order.
- **Maintenance.** In each iteration i, the subarray $A[1, \ldots, i-1]$ constains the $i-1$ smallest elements of A in sorted order. After the for loop of lines 2–4, $A[i]$ will be the smallest element of the subarray $A[i, \ldots, A.length]$ and thus the i-th smallest element of A. This implies that $A[1, \ldots, i]$ will contain the i smallest elements of A in sorted order. Incrementing i (in the for loop update) reestablishes the loop invariant for the next iteration.
- **Termination.** At termination, $i = A.length$. By the loop invariant the subarray $A[1, \ldots, A.length - 1]$ consists of the smallest elements of A in sorted order. Since $A[A.length]$ can only be the largest element of A, it is already in its correct position and the subarray $A[1, \ldots, A.length]$ consists of the elements of A in sorted order.

(d) The worst running time of Bubble-Sort is $\Theta(n^2)$, which is the same of Insertion-Sort. However, the best running time of Insertion-Sort is $\Theta(n)$ (when the array is already sorted) and Bubble-Sort runs always in $\Theta(n^2)$.

2-3 ***Correctness of Horner's rule***

The following code fragment implements Horner's rule for evaluating a polynomial

$$\begin{aligned} P(x) &= \sum_{k=0}^{n} a_k x^k \\ &= a_0 + x(a_1 + x(a_2 + \cdots + x(a_{n-1} + x a_n) \cdots)), \end{aligned}$$

given the coefficients $a_0, a_1, \ldots, a_n$ and a value for x:

```
1 y = 0
2 for i = n downto 0 do
3 |   y = a_i + x · y
```

a. In terms of Θ-notation, what is the running time of this code fragment for Horner's rule?

b. Write pseudocode to implement the naive polynomial-evaluation algorithm that computes each term of the polynomial from scratch. What is the running time of this algorithm? How does it compare to Horner's rule?

c. Consider the following loop invariant:
At the start of each iteration of the **for** loop of lines 2–3,

$$y = \sum_{k=0}^{n-(i+1)} a_{k+i+1} x^k.$$

Interpret a summation with no terms as equaling 0. Following the structure of the loop invariant proof presented in this chapter, use this loop invariant to show that, at termination, $y = \sum_{k=0}^{n} a_k x^k$.

d. Conclude by arguing that the given code fragment correctly evaluates a polynomial characterized by the coefficients $a_0, a_1, \ldots, a_n$.

(a) Since the body of the for loop of lines 2–3 consists of constant operations, the running time depends on the number of iterations of the loop. The running time is then $\sum_{i=0}^{n} 1 = n + 1 = \Theta(n)$.

(b) Here is the pseudocode of the NaivePolynomialEvaluation algorithm:

```
1 y = 0
2 for i = 0 to n do
3 |   y_i = a_i
4 |   for k = 1 to i do
5 |   |   y_i = y_i x
6 |   y = y + y_i
```

The running time of the above algorithm is $\sum_{i=0}^{n} i = (n(n+1))/2 = n^2/2 - n/2 = \Theta(n^2)$, which is slower than the $\Theta(n)$ running time of Horner's rule.

(c) Here is the loop invariant proof:

- **Initialization.** Prior to the first iteration of the for loop of lines 2–3, we have $y = 0$ and $i = n$. Replacing $i = n$ on the above loop invariant equation we have:

$$y = \sum_{k=0}^{n-n-1} a_{k+n+1} x^k = \sum_{k=0}^{-1} a_{k+n+1} x^k = 0,$$

which correctly corresponds to the initial value of y on line 1.

- **Maintenance.** In each iteration i of the loop, the previous value of y is multiplied by x and incremented by a_i (line 3). Performing these two operations on the above loop invariant equation for an iteration i, we have:

$$a_i + x \cdot \left(\sum_{k=0}^{n-i-1} a_{k+i+1} x^k \right) = a_i + \left(\sum_{k=0}^{n-i-1} a_{k+i+1} x^{k+1} \right) = a_i + \left(\sum_{k=1}^{n-i} a_{k+i} x^k \right) = \left(\sum_{k=0}^{n-i} a_{k+i} x^k \right),$$

which correctly corresponds to the loop invariant equation in the iteration $i - 1$ (next iteration, after iteration i).

- **Termination.** At termination, we have $i = -1$, so that:

$$y = \sum_{k=0}^{n-(-1+1)} a_{k+-1+1} x^k = \sum_{k=0}^{n} a_k x^k.$$

(d) Since the loop invariant holds for all iterations and, at termination, the loop invariant corresponds exactly to the polynomial definition, we can assure that the code fragment correctly evaluates the polynomial characterized by the coefficients $a_0, a_1, \ldots, a_n$.

2-4 ***Inversions***

Let $A[1, \ldots, n]$ be an array of n distinct numbers. If $i < j$ and $A[i] > A[j]$, then the pair (i, j) is called an ***inversion*** of A.

a. List the five inversions of the array $\langle 2, 3, 8, 6, 1 \rangle$.

b. What array with elements from the set {1, 2, . . . , n} has the most inversions? How many does it have?

c. What is the relationship between the running time of insertion sort and the number of inversions in the input array? Justify your answer.

d. Give an algorithm that determines the number of inversions in any permutation on n elements in $\Theta(n \lg n)$ worst-case time. (*Hint: modify merge sort.*)

(a) (1, 4), (1, 5), (2, 5), (3, 5), (4, 5).

(b) $\{n, n-1, n-2, \ldots, 2, 1\}$. It has $\binom{n}{2} = n(n-1)/2$ inversions.

(c) The number of operations of Insertion-Sort in an array A is the same as the number of inversions in A.

(d) The following pseudocode modifies Merge-Sort to count the number of inversions in $\Theta(n \lg n)$.

```
Inversions(A, p, r)
1 | inv = 0
2 | if p < r then
3 | |   q = ⌊(p + r)/2⌋
4 | |   inv = inv + Inversions(A, p, q) + Inversions(A, q + 1, r) + MergeInversions(A, p, q, r)
5 | return inv
```

```
MergeInversions(A, p, q, r)
1  | inv = 0
2  | n_1 = q − p + 1
3  | n_2 = r − q
4  | let L[1, ..., n_1 + 1] and R[1, ..., n_2 + 1] be new arrays
5  | for i = 1 to n_1 do
6  | |  L[i] = A[p + i − 1]
7  | for j = 1 to n_2 do
8  | |  R[j] = A[q + j]
9  | L[n_1 + 1] = ∞
10 | L[n_2 + 1] = ∞
11 | i = 1
12 | j = 1
13 | for k = p to r do
14 | |  if L[i] ≤ R[j] then
15 | |  |  i = i + 1
16 | |  else
17 | |  |  inv = inv + (n_1 − i + 1)
18 | |  |  j = j + 1
19 | return inv
```