Chapter 2: Getting Started

Exercise 2.1 – 1
Using Figure 2.2 as a model, illustrate the operation of INSERTION-SORT on the array A = {31, 41, 59, 26, 41, 58}.

- Changes made to the array in each iteration of for loop is as follows:

1. While loop not entered, hence no change in the input array.

31	41	59	26	41	58

2. While loop not entered, hence no change in the input array.

31	41	59	26	41	58

3. While loop entered, array changes after each iteration shown below:

31	41	59	59	41	58

31	41	41	59	41	58

31	31	41	59	41	58

26	31	41	59	41	58

4. While loop entered, array changes after each iteration shown below:

26	31	41	59	59	58

26	31	41	41	59	58

5. While loop entered, array changes after each iteration shown below:

26	31	41	41	59	59

26	31	41	41	58	59

Chapter 2: Getting Started

Exercise 2.1 – 2

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

```
- INSERTION-SORT(A)
      for j = 2 to A.length
          key = A[j]
          i = j – 1

          while i > 0 and A[i] < key
              A[i + 1] = A[i]
              i = i – 1

          A[i + 1] = key
```

Exercise 2.1 – 3

Consider a searching problem:

Input: A sequence of n numbers A = {a_1, a_2, …, a_n} and a value v.

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

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

```
- LINEAR-SEARCH(A, v)
      for i = 1 to A.length
          if A[i] = v
              return i

      return NIL
```

- Initialization: Before the first iteration, we can say that the value was not found in the subarray A[1 … i – 1] (an empty subarray). If the first element turns out to be the value we are looking for, we simply return its index (i.e. 1), if not, we move on to the next iteration.
- Maintenance: Before each iteration, subarray A[1 … i – 1] represents visited elements, neither of which equaled the target value. So, we continue our search i onwards.
- Termination: The loop is terminated either when we find the target value, or when i = A.length + 1, in which case we have scanned the entire array.

Chapter 2: Getting Started

Exercise 2.1 – 4

Consider the problem of adding two 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.

- Input: Two n-sized arrays A and B containing bits of two binary integers a and b respectively.
  Output: An array C of size (n + 1) containing bits of binary integer (a + b).
- ADD(A, B)

```
        Create array C of length (n + 1)
        carry_over = 0

        for i = n downto 1
                C[i + 1] = (A[i] + B[i] + carry_over) mod 2;

                if A[i] + B[i] + carry_over > 1
                        carry_over = 1
                else
                        carry_over = 0

        C[1] = carry_over
        return C
```

Chapter 2: Getting Started

Exercise 2.2 – 1

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

- To express the given function in terms of big theta notation, we need to first get rid of lower order terms ($100n^2$, 100n and 3) and coefficient attached to the leading term (1/1000). We are now left with n^3, and therefore, the function can be expressed as $\Theta(n^3)$.

Exercise 2.2 – 2

Consider sorting n numbers 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.

```
- SELECTION-SORT(A)
      for i = 1 to n – 1
          smallest_element_index = i

          for j = i + 1 to n
              if A[j] < A[smallest_element_index]
                  smallest_element_index = j

          swap A[i] and A[smallest_element_index]
```

- Initialization: Before the first iteration of the outer for loop, the subarray A[1 … i – 1] is sorted (empty subarray).
- Maintenance: In each iteration, smallest element of the remaining subarray is swapped with the element at position i. Therefore, before an iteration starts, subarray A[1 … i – 1] contains i – 1 smallest elements of the input sequence in their sorted position.
- Termination: The loop terminates when i equals n. At this point, subarray A[1 … n – 1] contains all n – 1 smallest elements in their sorted positions, leaving the largest value to take position n.
- Unlike insertion sort, the worst and best-case running time of selection sort is the same, $\Theta(n^2)$. This is because for each iteration of the outer loop, we scan the entire A[i + 1 … n] subarray to find the next smallest element.
- Number of times inner loop is executed = $\sum_{i=1}^{n-1} i$.

Chapter 2: Getting Started

Exercise 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 answer.

- If the element being searched is present in the sequence, on an average, equal number of elements would be present on either side of it. We will therefore need to iterate through n / 2 elements to finally arrive at the target element's position. This would result in Θ(n) time complexity.
- Worst-case happens when the value we are looking for is not present in the array. In this case we check all the elements, resulting in the running time Θ(n).

Exercise 2.2 – 4

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

- We can modify any algorithm to have a good best-case running time by adding conditions. If these conditions are satisfied, we can simply return the precomputed result, or move to the next iteration, etc.

Chapter 2: Getting Started

Exercise 2.3 – 1
Using figure 2.4 as a model, illustrate the operation of merge sort on the array A = {3, 41, 52, 26, 38, 57, 9, 49}.

- Changes made to the array after each merge operation is shown below:

3	41	52	26	38	57	9	49

3	41	26	52	38	57	9	49

3	26	41	52	9	38	49	57

3	9	26	38	41	49	52	57

Exercise 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.

```
- MERGE(A, p, q, r)
      n1 = q – p + 1
      n2 = r – q
      let L[1 … n1] and R[1 … n2] be new arrays

      for i = 1 to n1
          L[i] = A[p + i – 1]

      for j = 1 to n2
          R[j] = A[q + j]

      i = 1
      j = 1
      k = p
      while i < n1 and j < n2
          if L[i] < R[j]
              A[k++] = L[i++]
          else
              A[k++] = R[j++]

      while i < n1
          A[k++] = L[i++]

      while j < n2
          A[k++] = R[j++]
```

Chapter 2: Getting Started

Exercise 2.3 – 3

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

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

Is T(n) = n lgn.

- Using mathematical induction:
- Step 1 (base step):
  When k = 1,
  T(n) = T(2) = 2 (given)
  n * lg(n) = 2 * lg(2) = 2
  Therefore, condition holds for k = 1.
- Step 2 (inductive step):
  Let us assume that the condition holds for some k > 1,
  $T(2^k) = 2^k * \lg(2)^k$

  Let us now prove that the condition holds for k + 1,
  $T(2^{k+1}) = 2 * T(2^{k+1} / 2) + 2^{k+1}$
  $= 2 * T(2^k) + 2^{k+1}$
  Substituting the value of $T(2^k)$,
  $T(2^{k+1}) = 2 * (2^k * \lg(2)^k) + 2^{k+1}$
  $= 2^{k+1} * \lg(2)^k + 2^{k+1}$
  $= 2^{k+1} * (\lg(2^k) + 1)$
  $= 2^{k+1} * (\lg(2^k) + \lg 2)$
  $= 2^{k+1} * [\lg(2^k * 2)]$
  $= 2^{k+1} * \lg 2^{k+1}$
- We can see that the condition holds for k + 1 too. I therefore conclude that the solution to the given recurrence is n * lg(n).

Chapter 2: Getting Started

Exercise 2.3 – 4

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

```
-   INSERTION-SORT(A, index)
        if index > 1
            INSERTION-SORT(A, index – 1)
            INSERT-ELEMENT(A, index)

    INSERT-ELEMENT(A, index)
        while index > 1 and A[index – 1] > A[index ]
            swap A[index – 1] and A[index]
```

- Let the time taken to solve a problem of size "n" be T(n). Therefore, it would take T(n – 1) units of time to solve a problem of size "n – 1". We also know that procedure "INSERT-ELEMENT" takes Θ(n) time. Using this data, we get the recurrence:

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

Chapter 2: Getting Started

Exercise 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 v and eliminate half 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 Θ(lgn).

```
- BINARY-SEARCH(A, v)
        leftIndex = 1
        rightIndex = A.length

        while leftIndex <= rightIndex
            middleIndex = ⌊(leftIndex + rightIndex) / 2⌋

            if A[middleIndex] == v]
                return middleIndex

            if A[middleIndex] < v
                leftIndex = middleIndex + 1
            else
                rightIndex = middleIndex – 1

        return NIL
```

- Let the running time of the algorithm for input of size n be T(n), therefore,
  T(n) = T (n / 2) + c
  Using master theorem, the solution we get for the above recurrence is Θ(lg(n)).
- Let us solve it intuitively. In worst case, the value we are searching for is not present in the array. Since the algorithm works by continuously dividing an array into two (almost) equal parts and excluding one of them from consideration, it can at most make lg(n) divisions.

Exercise 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 … 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 Θ(nlgn).

- The while loop of lines 5 - 7 inserts j^{th} element in its correct position in the sorted subarray A[1 … j – 1].
- In a worst-case scenario, binary search will enable us in finding the element's position in lg(j) time, which is a big improvement. But to actually insert it at that position, we will still need j – 1 swaps.
- Therefore, replacing the loop with binary search will not help improve the worst-case running time of the algorithm.

Chapter 2: Getting Started

Exercise 2.3 – 7

Describe a Θ(nlgn) – 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.

- We first sort all the elements in ascending order using merge-sort and then initiate two variables a and b, with values 1 and n respectively.
- These variables point to the first and last element of the input sequence.
- If the sum of elements with indices a and b is greater than x, we decrement b to make it point to the second last element. If their sum is less than x, we increment a to make it point to the second element. This process continues until a becomes greater than or equal to b.
- If no such pair is found, we simply return False.
- FIND-PAIR(S, x)

```
        MERGE-SORT(S)

        leftIndex = 1
        rightIndex = S.length

        while leftIndex < rightIndex
                if S[leftIndex] + S[rightIndex] == x]
                        return True

                else if S[leftIndex] + S[rightIndex] > x
                        rightIndex = rightIndex - 1
                else
                        leftIndex = leftIndex + 1

        return False
```

-

Chapter 2: Getting Started

Problem 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?**

- We have seen that it takes $\Theta(k^2)$ worst-case time for insertion sort to sort an array of length k. If we have n / k such arrays, it will take, $\Theta((n / k) * k^2) = \Theta(n * k)$ worst-case time.
- If we make the suggested changes to our merge sort algorithm, we will be left with n / k subarrays, each of length k at the bottom most level of recursion tree. At this point instead of dividing the subarrays further, we sort them using insertion sort. We will therefore have lg(n / k) levels in total and not lg(n) + 1.
  At each of these levels, we perform merge operation which still takes $\Theta(n)$ time thereby resulting $\Theta(n * \lg(n / k))$.
- For the modified algorithm to have the same asymptotic running time as standard merge sort, $\Theta(n * k + n * \lg(n / k))$ will have to be a subset of $\Theta(n * \lg(n))$. Therefore, the maximum value k can take is lg(n) to satisfy this condition.
- In practice, we can initialize k as 1 and compare standard merge sort algorithm with the modified one. We can then repeat this procedure, every time incrementing k by 1, until our modified algorithm runs slower than the standard one.

Chapter 2: Getting Started

Problem 2.2

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
2        for j = A.length downto i + 1
3            if A[j] < A[j – 1]
4                exchange A[j] with A[j – 1]
```

a. **Let A’ denote the output of BOBBLESORT(A). To prove that BUBBLESORT is correct, we need to prove that it terminates and that**
   A’[1] <= A’[2] <= … <= A’[n] (2.3)
   where n = A.length. In order to show that BUBBLESORT actually sorts, what else do we need to prove?
   The next two parts will prove the inequality (2.3).
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 present 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 inequality (2.3). You 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 selection sort?**

- We need to prove that A’ is nothing but a permutation of A. In other words, we need to prove that it contains all the elements of the original array.
- Initialization: Before the first iteration, j stores index of the smallest element in subarray A[j … n]. Since at this point we only have one element in this subarray, j equals n.
- Maintenance: If element at position (j – 1) is greater than the element at position j, we swap the two elements and decrement j. This ensures that j always contains index of the smallest element in subarray A[j … n] after each iteration.
  Termination: The loop is terminated when j equals i. By this time, smallest element of subarray A[i … n] is positioned at i, which is its final position.
- Initialization: Initially, subarray A[1 … i - 1] is an empty subarray, hence already sorted. After the first iteration, smallest element of the remaining subarray is inserted at position i, putting it in its correct sorted position.
  Maintenance: Before each iteration starts, i is incremented by 1 to maintain the sorted characteristic of subarray A[1 … i - 1]. Again, the smallest element of the remaining subarray takes position i.
  Termination: The loop is terminated when i equals n, which means that all the elements have been placed in their correct positions.
- The inner for loop of bubblesort is executed (n – 1), (n – 2) … 3, 2, 1 = $\sum_{i=1}^{n-1} i$ times for all possible permutations of size n, making its best-case running time equal to its worst-case, $\Theta(n^2)$. Therefore, both selection and bubblesort have similar time complexities.

Chapter 2: Getting Started

Problem 2.3

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

P(x) = $\sum_{k=0}^{n} a_k x^k$

$= a_0 + x(a_1 + x(a_2 + \ldots + x(a_{n-1} + xa_n) \ldots))$,

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

```
1   y = 0
2   for i = n downto 0
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 naïve 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 $a_0, a_1, \ldots, a_n$.**

- The for loop is executed n + 1 times. Therefore, the running time of the above code fragment is Θ(n).

-
  ```
  y = 0
  for i = n downto 0
      term = $a_n$

      for j = i downto 1
          term = term * x

      y = y + term
  ```

  The inner for loop in the above pseudocode runs for n, (n – 1), (n – 2), …, 2, 1 = = $\sum_{i=0}^{n} i$ times, giving it time complexity Θ(n^2). Our quadratic solution run much slower than the linear one provided by Horner in case of large input sizes.

Chapter 2: Getting Started

Problem 2.4

Let A[1 … 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 {2, 3, 8, 6, 1}.**
- **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 running time of inversions in the input array? Justify your answer.**
- **d.** **Give an algorithm that determines the inversions in any permutation on n elements in Θ(n * lg(n)) worst-case time. (Hint: Modify merge sort.)**

- The five inversions of the array {2, 3, 8, 6, 1} are (3, 4), (1, 5), (2, 5), (3, 5) and (4, 5).
- An array that contains all the elements of set {1, 2, …, n} in descending order will have most inversions. It will have $\sum_{i=1}^{n-1} i = (n * (n-1)) / 2$ inversions.
- FIND-INVERSIONS(A)

```
        numberOfInversions = 0

        for i = 2 to A.length
            j = i

            while j > 0
                if A[j – 1] > A[j]
                    numberOfInversions = numberOfInversions + 1

        return numberOfInversions
```

  The inner loop is executed $\sum_{i=1}^{n-1} i$ times for all possible permutations of size n. Therefore, the algorithm's best-case running time is same as its worst-case, $\Theta(n^2)$.

  This is not the case in insertion sort. The order of elements of the input array has an impact on its running time. If the input array is already sorted, insertion sort runs in $\Theta(n)$ time.
- In order to find the number of inversions in an array, we need to tweak merge sort a bit. We need to initialize a counter variable (say "numberOfInversions") as 0 outside functions MERGE and MERGE-SORT and add the following code on line number 17 of MERGE subroutine:

  numberOfInversions = numberOfInversions + (n_1 – i)