**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 |

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

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 31 | 41 | 59 | 26 | 41 | 58 |

1. 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 |

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

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 26 | 31 | 41 | 59 | 59 | 58 |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 26 | 31 | 41 | 41 | 59 | 58 |

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

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 26 | 31 | 41 | 41 | 59 | 59 |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 26 | 31 | 41 | 41 | 58 | 59 |

**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 = {a1,a2, …,an} 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 when we find the target value, or when i = A.length + 1, in which case we have scanned the entire array.

**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

**Exercise 2.2 – 1**

**Express the function n3/1000 – 100n2 – 100n + 3 in terms of -notation.**

* To express the given function in terms of big theta notation, we need to first get rid of lower order terms (100n2, 100n and 3) and coefficient attached to the leading term (1/1000). We are now left with n3, and therefore, the function can be expressed as (n3).

**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 -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: After each iteration, smallest element of the remaining subarray is swapped with the element at position i. Doing this maintains the loop invariant. Before the next iteration starts, the subarray A[1 … i – 1] contains all elements 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, (n2). 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, thereby running the inner loop (n – 1), (n – 2) … 3, 2, 1 = times.

**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 -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. After neglecting coefficients attached to n, running time in such case would be (n).
* 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 (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.

**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++]

**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) = 2 if n = 2**

**2 T(n / 2) + n if n = 2k, for k > 1**

**Is T(n) = n lgn.**

* Using mathematical induction:
* Step 1 (base step):

For n = 21 = 2,

T(n) = T(2)

= 2 … (given)

n \* lg(n) = 2 \* lg(2)

= 2

Therefore, condition holds for n = 2.

* Step 2 (inductive step):

Let us assume that the condition holds for n = 2k,

T(2k) = 2k \* lg2k

Let us now prove that the condition holds for n = 2k + 1,

T(2k + 1) = 2 \* T(2k + 1 / 2) + 2k + 1

= 2 \* T(2k) + 2k + 1

= 2 \* (2k \* lg2k) + 2k + 1

= 2k+1 \* lg2k + 2k + 1

= 2k + 1 \* (lg2k + 1)

= 2k + 1 \* (lg2k + lg2)

= 2k + 1 \* [lg(2k \* 2)]

= 2k + 1 \* lg2k + 1

**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 time (n). Using this data, we get the recurrence:
* T(n) = (1) if n = 1

T(n - 1) + (n) if n > 1

**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 (lg(n)).**

* 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

* In worst case, the value we are searching is either the first or the last element of the array. It takes lg(n) steps for this procedure to reach the first/last element, and therefore has the worst-case running time (lg(n)).

**Exercise 2.3 – 5**

**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 (n \* lg(n)).**

* The while loop of lines 5 - 7 inserts jth element in its correct position in the sorted subarray A[1 … j - 1]. Replacing it with binary search won’t help improve the worst case running time of the algorithm.
* Binary search will enable us in finding the element’s position in lg(j - 1) worst-case time, which is a big improvement. But to actually insert it at that position, we will still need j – 1 swaps in worst case.

**Exercise 2.3 – 6**

**Describe a (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.**

* We can first sort all the elements in ascending order using merge-sort and initiate two variables a and b, with values 1 and n respectively.
* These values refer to the indices of first and last element of the series.
* We can then check if their sum equals x. If their sum is greater than x, we will decrement b to point it to the second last element. If their sum is less than x, we will increment a to point it to the second element. We will continue this process until a < b.
* If no such elements are found, we can 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

**Problem 2.1**

**Although merge sort runs in (n \* lg(n)) worst-case time and insertion sort runs in (n2) 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.**

1. **Show that insertion sort can sort the n / k sublists, each of length k, in (n \* k) worst-case time.**
2. **Show how to merge the sublists in (n \* lg(n / k)) worst-case time.**
3. **Given that the modified algorithm runs in ((n \* k) + 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 -notation?**
4. **How should we choose k in practice?**

* We have seen that it takes (k2) worst-case time for insertion sort to sort an array of length k. If we have n / k such arrays, it will take, ((n / k) \* k2) = (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. At this point when we stop dividing the subarrays further, we have a total of lg(n) – lg(k) = lg(n / k) levels instead of lg(n) + 1.

At each level, we perform merge operation that takes (n) time. Therefore our overall worst-case time complexity becomes (lg(n / k) \* n) = (n \* lg(n / k)).

* 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.

**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]**

1. **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).**

1. **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.**
2. **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.**
3. **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, just in a rearranged manner.
* At all times, index j will contain the smallest element of subarray A[i + 1 … n].

Initialization: We initialize j as n, assuming that last element is the smallest. In the first iteration, we check if it really is smaller than the element appearing before it. If it is, we continue swap the two elements, if not, we continue to the next iteration.

Maintenance: In the next iteration, j is decremented by 1. It still contains the smallest element of subarray A[j … n].

Termination: In the last iteration,

* Initialization: Initially, subarray A[1 … i - 1] is an empty subarray, hence already sorted. After the first iteration, the smallest element of the remaining subarray is swapped with A[1], putting it in its correct sorted position.

Maintenance: When the next iteration starts, the subarray A[1 … i - 1] is again sorted. We need to sort the remaining elements in the array.

Termination: In the last iteration, again, subarray A[1 … i - 1] is sorted, all we need to do is find out if (n – 1)th element is greater than nth. If it is, we swap the two elements, if not, we let them be. Now the entire array is sorted.

* The inner for loop of bubblesort is executed (n – 1) + (n – 2) + … + 2 + 1 = [n \* (n - 1)] / 2 times. Therefore, the worst-case running time of bubblesort is (n2), same as insertion sort. The best-case running time of insertion sort in (n), but the best-case running time of bubblesort is still (n2).

**Problem 2.3**

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

**P(x) =**

**= a0 + x(a1 + x(a2 + … + x(an – 1 + xan) … )),**

**given the coefficients a0, a1, …, an and a value for x:**

**1 y = 0**

**2 for i = n downto 0**

**3 y = ai + x \* y**

1. **In terms of -notation, what is the running time of this code fragment for Horner’s rule?**
2. **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?**
3. **Consider the following loop invariant:**

**At the start of each iteration of the for loop of lines 2 – 3,**

**y =**

**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 =**

1. **Conclude by arguing that the given code fragment correctly evaluates a polynomial characterized by the coefficients a0, a1, …, an.**

* The for loop is executed n + 1 times. Therefore, the running time of the above code fragment is (n).
* y = 0

for i = n downto 0

coefficient = an - i

multiplier = 1

for j = i – 1 downto 1

multiplier = multiplier \* x

y = y + (coefficient \* multiplier)

The inner for loop in the above pseudocode runs for (n – 1), (n – 2), …, 2, 1 times

= = (n2).

**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.**

1. **List the five inversions of the array {2, 3, 8, 6, 1}.**
2. **What array with elements from the set {1, 2, …, n} has the most inversions? How many does it have?**
3. **What is the relationship between the running time of insertion sort and the running time of inversions in the input array? Justify your answer.**
4. **Give an algorithm that determines the inversions in any permutation on n elements in (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 = (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++

return numberOfInversions

The while loop is executed = (n \* (n – 1)) / 2 times, giving us the running time (n2). For each iteration of for loop, while loop is executed i – 1 times. The order of the input array has no impact on its running time and thus we conclude its best-case running time is same as its worst-case running time (n2).

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, it runs in (n) time, which is its best case.

* 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. Every time we swap two elements, we increment this variable by 1. We do this because these elements are an inversion of the array.