CHAPTER 8

Algorithms

(Solutions to Practice Set)

Review Questions

1. An algorithm is an ordered set of unambiguous steps that produces a result and terminates in a finite time.
2.
   a. **Sequence**: An algorithm is a sequence of instructions, which can be a simple instruction or either of the other two constructs.
   b. **Decision** (selection): Tests a condition. If the result of testing is true, it follows a sequence of instructions; if it is false, it follows a different sequence of instructions.
   c. **Repetition**: This construct repeats a set of instructions.
3. Universal Modeling Language (UML) is a pictorial representation of an algorithm. It hides all of the details of an algorithm in an attempt to give the big picture; it shows how the algorithm flows from beginning to end.
4. Pseudocode is an English-like representation of an algorithm.
5. A sorting algorithm arranges data according to their values.
6. The three types of sorting algorithms discussed in this chapter are the selection sort, the bubble sort, and the insertion sort.
7. A searching algorithm finds a particular item (target) among a list of data.
8. The two major searching algorithms discussed in this chapter are the sequential search and the binary search. The sequential search is normally used for searching unsorted lists while the binary search is used for searching sorted lists.
9. An algorithm is iterative if it uses a loop construct to perform a repetitive task.
10. An algorithm is recursive if it calls itself, that is, if the algorithm appears within its own definition.

Multiple-Choice Questions

11. d	12. c	13. b	14. a	15. c	16. a
17. c	18. a	19. b	20. b	21. d	22. c
23. a	24. b	25. b	26. a	27. b	28. c

Exercises

29. The value of **Sum** after each iteration is shown below:

Iteration	Data item	Sum = 0
1	20	**Sum** = **0** + 20 = 20
2	12	**Sum** = **20** + 12 = 32
3	70	**Sum** = **32** + 70 = 102
4	81	**Sum** = **102** + 81 = 183
5	45	**Sum** = **183** + 45 = 228
6	13	**Sum** = **228** + 13 = 241
7	81	**Sum** = **241** + 81 = 322
After exiting the loop		**Sum = 322**

30. The value of **Product** after each iteration is shown below:

Iteration	Data item	Product = 1
1	2	**Product** = **1** × 2 = 2
2	12	**Product** = **2** × 12 = 24
3	8	**Product** = **24** × 8 = 192
4	11	**Product** = **192** × 11 = 2112
5	10	**Product** = **2112** × 10 = 21120
6	5	**Product** = **21120** × 5 = 105600
7	20	**Product** = **105600** × 20 = 2112000
After exiting the loop		**Product = 2112000**

31. The value of **Largest** after each iteration is shown below:

Iteration	Data item	Largest = $-\infty$
1	18	**Largest = 18**
2	12	**Largest = 18**
3	8	**Largest = 18**
4	20	**Largest = 20**
5	10	**Largest = 10**
6	32	**Largest = 32**
7	5	**Largest = 32**
After exiting the loop		**Largest = 32**

32. The value of **Smallest** after each iteration is shown below:

Iteration	Data item	Smallest = + ∞
1	18	**Smallest = 18**
2	3	**Smallest = 3**
3	11	**Smallest = 3**
4	8	**Smallest = 3**
5	20	**Smallest = 3**
6	1	**Smallest = 1**
7	2	**Smallest = 1**
After exiting the loop		**Smallest = 1**

33. The status of the list and the location of the wall after each pass is shown below:

Pass	List								
	14 |	7	23	31	40	56	78	9	2
1	2 |	7	23	31	40	56	78	9	14
2	2	7 |	23	31	40	56	78	9	14
3	2	7	9 |	31	40	56	78	23	14
4	2	7	9	14 |	40	56	78	23	31
5	2	7	9	14	23 |	56	78	40	31
6	2	7	9	14	23	31 |	78	40	56
7	2	7	9	14	23	78	40 |	78	56
8	2	7	9	14	23	78	40	56 |	78

34. The status of the list and the location of the wall after each pass is shown below:

Pass	List								
	14 |	7	23	31	40	56	78	9	2
1	2 |	14	7	23	31	40	56	78	9
2	2	7 |	14	9	23	31	40	56	78
3	2	7	9 |	14	23	31	40	56	78
4	2	7	9	14 |	23	31	40	56	78
5	2	7	9	14	23 |	31	40	56	78
6	2	7	9	14	23	31 |	40	56	78
7	2	7	9	14	23	31	40 |	56	78
8	2	7	9	14	23	31	40	56 |	78

35. The status of the list and the location of the wall after each pass is shown below:

Pass	List								
	14	7	23	31	40	56	78	9	2
1	7	14	23	31	40	56	78	9	2
2	7	14	23	31	40	56	78	9	2
3	7	14	23	31	40	56	78	9	2
4	7	14	23	31	40	56	78	9	2
5	7	14	23	31	40	56	78	9	2
6	7	14	23	31	40	56	78	9	2
7	7	9	14	23	31	40	56	78	2
8	2	7	9	14	23	31	40	56	78

36. The status of the list and the location of the wall after each pass is shown below:

Pass	List							
	7	8	26	44	13	23	98	57
1	7	8	13	44	26	23	98	57
2	7	8	13	23	26	44	98	57
3	7	8	13	23	26	44	98	57

37. The status of the list and the location of the wall after each pass is shown below

Pass	List							
	7	8	26	44	13	23	57	98
1	7	8	13	26	44	23	57	98
2	7	8	13	23	26	44	57	98
3	7	8	13	23	26	44	57	98

38. The status of the list and the location of the wall after each pass is shown below:

Iteration	List							
	3	13	7	26	44	23	98	57
1	3	7	13	26	44	23	98	57
2	3	7	13	23	26	44	98	57
3	3	7	13	23	26	44	98	57

39. The binary search for this problem follows the table shown below. The target (88) is found at index $i = 7$.

first	*last*	*mid*
1	*8*	*4*
5	*8*	*6*
7	*8*	*7*

1	*2*	*3*	*4*	*5*	*6*	*7*	*8*	
8	13	17	26	44	56	88	97	target > 44
				44	56	88	97	target > 56
						88	97	target = 88

40. The binary search for this problem follows the table shown below. The target (20) is not found.

first	*last*	*mid*
1	*6*	*3*
1	*2*	*1*
2	*2*	*2*
2	*1*	*1*

1	*2*	*3*	*4*	*5*	*6*	
17	26	44	56	88	97	target < 44
17	26					target > 17
	26					target < 26

first > *last* → **Target is not in the list**

41. The sequential search follows the table shown below. The target (20) is not found.

Iteration	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*10*	*11*	*12*	
	4	21	36	14	62	91	8	22	7	81	77	10	
1	4	21	36	14	62	91	8	22	7	81	77	10	target ≠ 4
2	4	21	36	14	62	91	8	22	7	81	77	10	target ≠ 21
3	4	21	36	14	62	91	8	22	7	81	77	10	target ≠ 36
4	4	21	36	14	62	91	8	22	7	81	77	10	target ≠ 14
5	4	21	36	14	62	91	8	22	7	81	77	10	target ≠ 62
6	4	21	36	14	62	91	8	22	7	81	77	10	target ≠ 91
7	4	21	36	14	62	91	8	22	7	81	77	10	target ≠ 8
8	4	21	36	14	62	91	8	22	7	81	77	10	target ≠ 22
9	4	21	36	14	62	91	8	22	7	81	77	10	target ≠ 7
10	4	21	36	14	62	91	8	22	7	81	77	10	target ≠ 81
11	4	21	36	14	62	91	8	22	7	81	77	10	target ≠ 77
12	4	21	36	14	62	91	8	22	7	81	77	10	target ≠ 10

Target is not in the list.

42. The binary search follows the table shown below. The target (17) is not found.

first	*last*	*mid*
1	12	6
1	5	3
4	5	4
5	5	5
6	5	5

1	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*10*	*11*	*12*	
4	7	8	10	14	21	22	36	62	77	81	91	target < 21
4	7	8	10	14								target > 8
			10	14								target > 10
				14								target > 14

first > *last* → **Target is not in the list**

43. Iterative evaluation of (6!) = 720 is shown below.:

i	*Factorial*
1	F = **1**
2	F = **1** × 2 = 2
3	F = **2** × 3 = 6
4	F = **6** × 4 = 24
5	F = **24** × 5 = 120
6	F = **120** × 6 = 720
After exiting the loop	F = **720**

44. Recursive evaluation of (6!) = 720 is shown below:

↓	6! = 6 × 5!	6! = 6 × 5! = 6 × **120** = **720**	
↓	5! = 5 × 4!	5! = 5 × 4! = 5 × **24** = **120**	↑
↓	4! = 4 × 3!	4! = 4 × 3! = 4 × **6** = **24**	↑
↓	3! = 3 × 2!	3! = 3 × 2! = 3 × **2** = **6**	↑
↓	2! = 2 × 1!	2! = 2 × 1! = 2 × **1** = **2**	↑
↓	1! = 1 × 0!	1! = 1 × 0! = 1 × **1** = **1**	↑
	0! = **1**		↑

45. Algorithm S8.45 shows the pseudocode for evaluating gcd.

Algorithm S8.45 *Exercise 45*

```
Algorithm: gcd(x, y)
Purpose: Find the greatest common devisor of two numbers
Pre: x, y
Post: None
Return: gcd (x, y)
{
    If (y = 0)          return x
    else                return gcd (y, x mod y)
}
```

46.

a. The following table show the evaluation of **gcd** (7, 41) = 7:

x	***y***	***x*** mod ***y***	Result
7	41	7	**gcd** (7, 41) = **gcd** (41, 7)
41	7	0	**gcd** (41, 7) = **gcd** (7, 0)
7	0		**gcd** (7, 0) = **7**

b. The following table show the evaluation of **gcd** (12, 100) = 4:

x	*y*	*x* mod *y*	Result
12	100	12	**gcd** (12, 100) = **gcd** (100, 12)
100	12	4	**gcd** (100, 12) = **gcd** (12, 4)
12	4	0	**gcd** (12, 4) = **gcd** (4, 0)
4	0		**gcd** (4, 0) = 4

c. The following table show the evaluation of **gcd** (80, 4)

x	*y*	*x* mod *y*	Result
80	4	0	**gcd** (80, 4) = **gcd** (4, 0)
4	0		**gcd** (4, 0) = 4

d. The following table show the evaluation of **gcd** (17, 29) = 1:

x	*y*	*x* mod *y*	Result
17	29	17	**gcd** (17, 29) = **gcd** (29, 17)
29	17	12	**gcd** (29, 17) = **gcd** (17, 12)
17	12	5	**gcd** (17, 12) = **gcd** (12, 5)
12	5	2	**gcd** (12, 5) = **gcd** (5, 2)
5	2	1	**gcd** (5, 2) = **gcd** (2, 1)
2	1	0	**gcd** (2, 1) = **gcd** (1, 0)
1	0		**gcd** (1, 0) = 1

47. Algorithm S8.47 shows the pseudocode for evaluating combination.

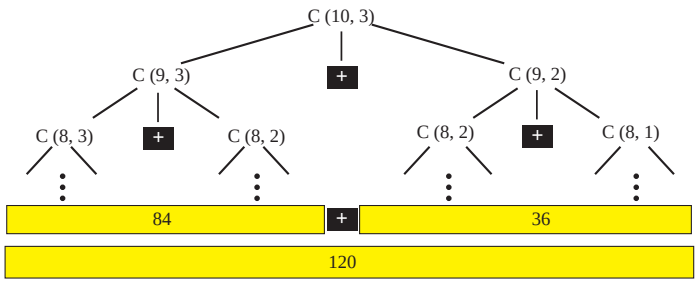
Algorithm S8.47 *Exercise 47*

```
Algorithm: Combination(n, k)
Purpose: It finds the combination of n objects k at a time
Pre: Given: n and k
Post: None
Return: C(n, k)
{
        If (k = 0 or n = k)          return 1
        else                         return C(n − 1, k) + C(n − 1, k − 1)
}
```

48.

a. Figure S8.48a shows the evaluation of C(10, 3).

Figure S8.48a



b. C(5, 5) = 1

c. C(2, 7) does not exist

d. C(4, 3) = C(3, 3) + C(3, 2) = 1 + [C(2, 2) + C(2, 1)]
= 1 + [1 + C(1, 1) + C(1, 0)] = 1 + [1 + 1 + 1] = 4

49. Algorithm S8.49 shows the pseudocode for evaluating Fibonacci sequence.

Algorithm S8.49 *Exercise 49*

```
Algorithm: Fibonacci(n)
Purpose: It finds the elements of Fibonacci sequence
Pre: Given: n
Post: None
Return: Fibonacci (n)
{
    If (n = 0)          return 0
    If (n = 1)          return 1
    else                return (Fibonacci (n − 1) + Fibonacci(n − 2))
}
```

50.

a. F(2) = F(1) + F(0) = 1 + 0 = 1

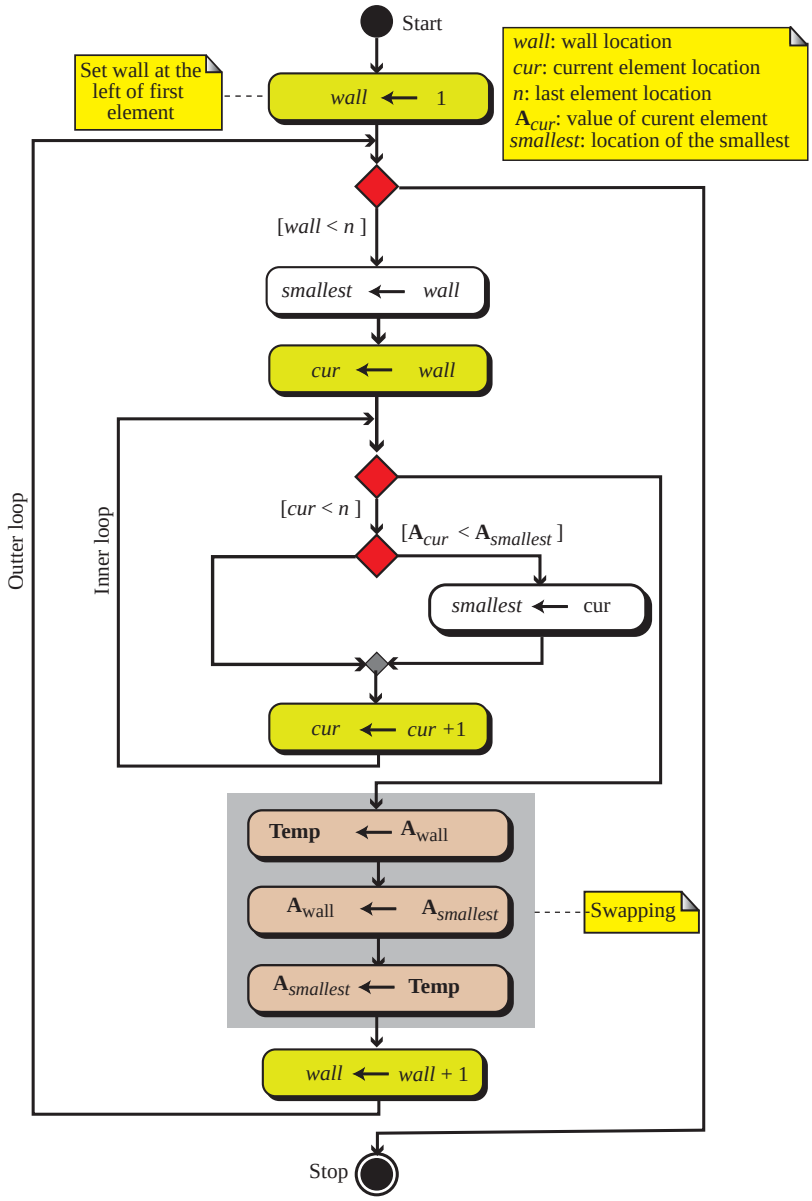
b. F(3) = F(2) + F(1) = [F(1) + F(0)] + F(1) = [1 + 0] + 1 = 2

c. F(4) = F(3) + F(2) = 2 + 1 = 3, based on solutions to parts *a* and *b*.

d. F(5) = F(4) + F(3) = 3 + 2 = 5, based on solutions to parts *b* and *c*.

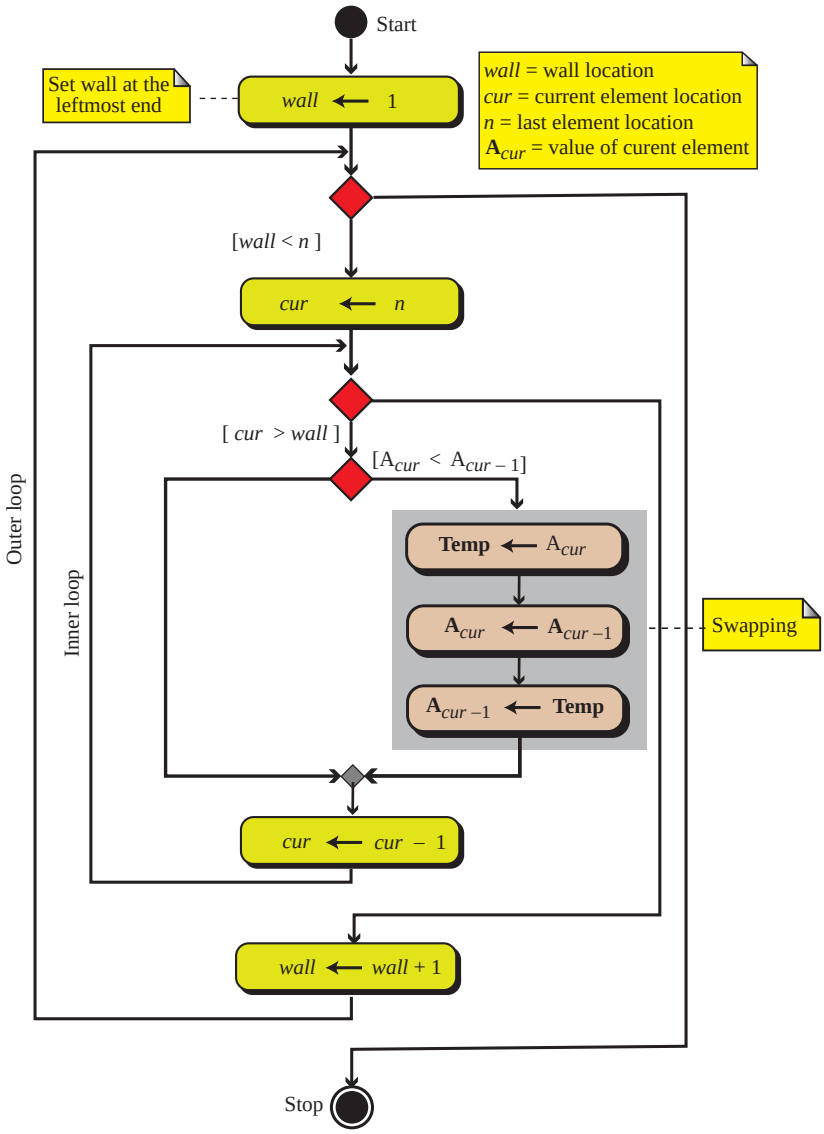
51. The UML for the selection sort is shown in Figure S8.51. The inner loop finds the location of the smallest element in the unsorted list. The three instructions after the inner loop swap this element with the first element in the unsorted list. Before searching for the smallest element, we assume that the first element in the unsorted list is the smallest one.

Figure S8.51 *Exercise 51*



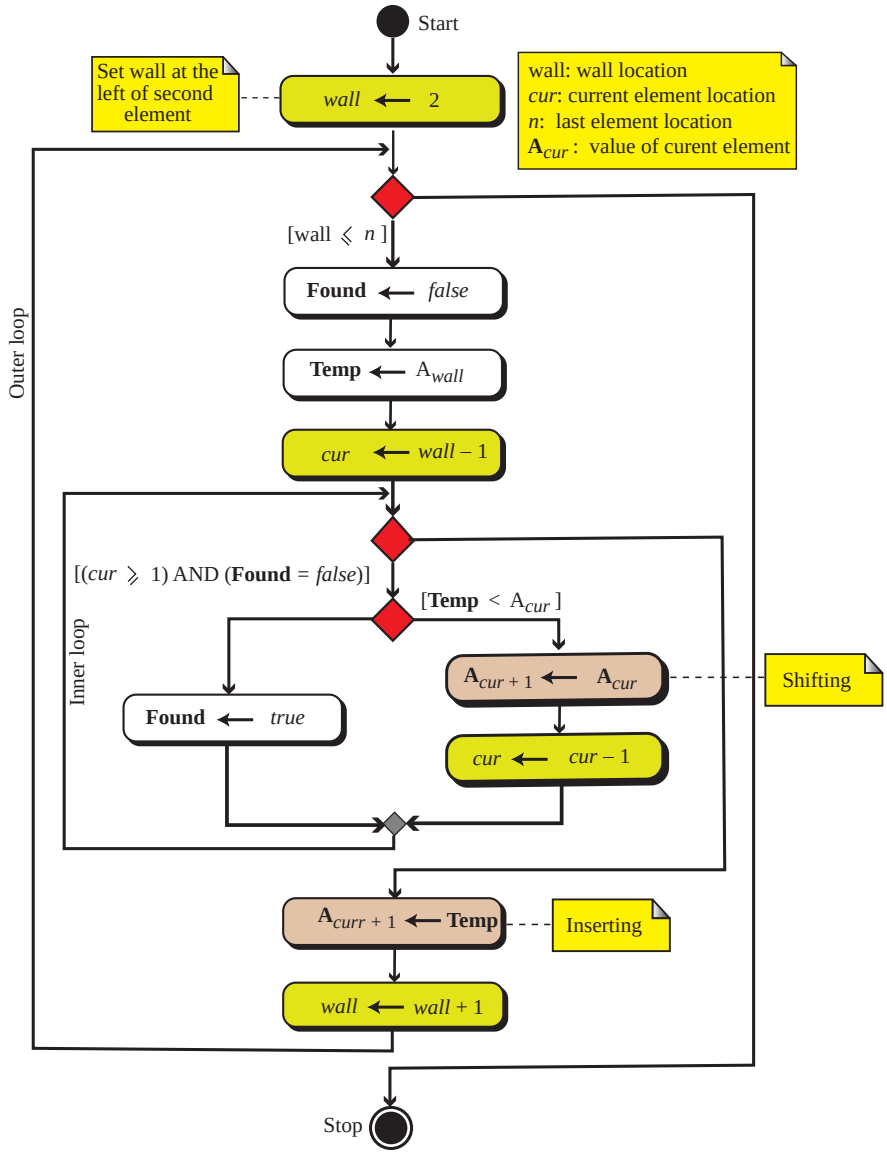
52. The UML for bubble sort is shown in Figure S8.52. Bubbling is performed by an inside loop. The conditional swapping performed by the inside loop bubbles the smaller items to the leftmost of the unsorted list. Note that the outside loop is iterated when the wall reaches the left of the rightmost element. When there is only one element in the unsorted list, the list is already sorted.

Figure S8.52 *Exercise 52*



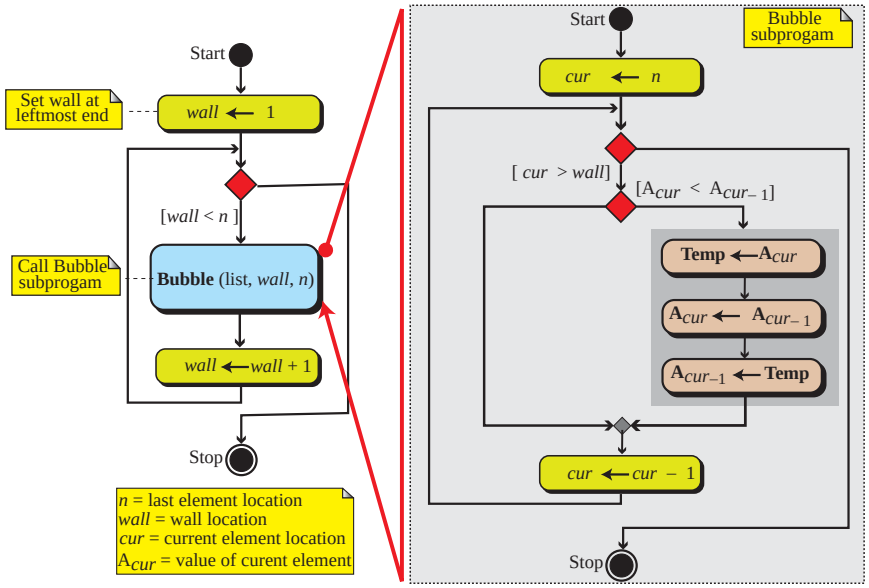
53. The UML for insertion sort is shown in Figure S8.53. The inner loop finds the location of the insertion. We shift the elements in the sorted sublist until we find the appropriate location to insert the element. When the algorithm exits the inner loop, insertion can be done. We have used a true/false value, **Found**, to stop shifting when the location of the insertion is found.

Figure S8.53 *Exercise 53*



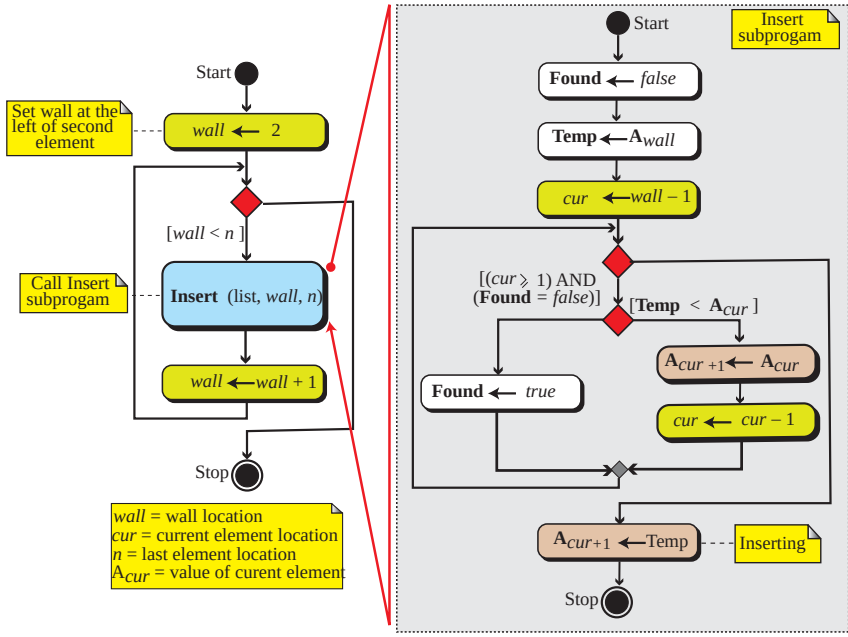
54. The UML is shown in Figure S8.54. The program calls the Bubble subprogram.

Figure S8.54 *Exercise 54*



55. The UML is shown in Figure S8.55. The program calls the Insert subprogram

Figure S8.55 *Exercise 55*



56. Algorithm S8.56 shows the pseudocode for finding the sum of integers.

Algorithm S8.56 *Exercise 56*

```
Algorithm: Summation (list)
Purpose: It finds the sum of integers
Pre: Given: A list of integers
Post: None
Return: Sum of the integers
{
        sum ← 0
        while (more integer to add)
        {
                get next integer
                sum ← sum + (next integer)
        }
        return sum
}
```

57. Algorithm S8.57 shows the pseudocode for finding the product of integers.

Algorithm S8.57 *Exercise 57*

```
Algorithm: Product(list)
Purpose: It finds the product of integers
Pre: Given: A list of integers
Post: None
Return: Product of the integers
{
        product ← 1
        while (more integer to multiply)
        {
                get next integer
                product ← product × (next integer)
        }
        return product
}
```

58. Algorithm S8.58 shows the pseudocode for the selection sort. It follows the UML diagram given in Exercise 51.

Algorithm S8.58 *Exercise 58*

```
Algorithm: SelectionSort(list, n)
Purpose: to sort a list using selection sort method
Pre: Given: A list of numbers
Post: None
Return:
{
        wall ← 1                                    // Set wall at the left of first element
        while (wall < n)                            // Outer loop
        {
                smallest ← wall
                cur ← wall                          // The current item is the one left to the wall
                while (cur < n)                     // Inner loop
                {
                        if (A_cur < A_smallest)             smallest ← cur
                        cur ← cur + 1               // Move the current element
                }
                Temp ← A_wall                       // The next three lines perform swapping
                A_wall ← A_smallest
                A_smallest ← Temp
                wall ← wall + 1                     // Move wall one element to the left
        }
}
```

59. Algorithm S8.59a shows the pseudocode for the selection sort routine that uses a subprogram. Finding the smallest numbers in the unsorted side is performed by a subalgorithm called FindSmallest (Algorithm 8.59b).

Algorithm S8.59a *Exercise 58*

```
Algorithm: SelectionSort(list, n)
Purpose: to sort a list using selection sort method
Pre: Given: A list of numbers
Post: None
Return:
{
        wall ← 1                                    // Set wall at the left of first element
        while (wall < n)
        {
                smallest ← FindSmallest (list, wall, n)     // Call the FindSmallest
                Temp ← A_wall                       // The next three lines perform swapping
                A_wall ← A_smallest
                A_smallest ← Temp
                wall ← wall + 1                     // Move wall one element to the right
        }
        return SortedList
}
```

Algorithm 8.59b *Exercise 59*

```
Algorithm: FindSmallest(list, wall, n)
Purpose: To find the smallest number in an unsorted list
Pre: Given: A list of numbers
Post: None
Return: The location of the smallest element in the unsorted list
{
    smallest ← wall                    // Assume the first element is the smallest one
    cur ← wall                         // The current item is the one left to the wall
    while (cur < n)
    {
        if (A_cur < A_smallest)        smallest ← cur
        cur ← cur + 1                          //Move the current element
    }
    return smallest
}
```

60. Algorithm S8.60 shows the pseudocode for the bubble sort. The algorithm follows the UML diagram in Exercise 52.

Algorithm S8.60 *Exercise 60*

```
Algorithm: BubbleSort(list, n)
Purpose: to sort a list using bubble sort
Pre: Given: A list of N numbers
Post: None
Return:
{
    wall ← 1                           // Set the wall at the leftmost end
    while (wall < n)                   // Outer loop
    {
        cur ← n                        // Start from the end of the list
        while (cur > wall)             // Bubble the smallest to the left of unsorted list
        {
            if (A_cur < A_cur – 1)             // Bubble one location to the left
            {
                Temp ← A_cur
                A_cur ← A_cur–1
                A_cur–1 ← Temp
            }
            cur ← cur – 1
        }
        wall ← wall + 1                // Move the wall one place to the right
    }
}
```

61. Algorithm S8.61a shows the pseudocode for the bubble sort routine that uses a subprogram. The bubbling of the numbers in the unsorted side is performed by a subalgorithm called Bubble (Algorithm 8.61b).

Algorithm S8.61a *Exercise 61*

```
Algorithm: BubbleSort(list, n)
Purpose: to sort a list using bubble sort
Pre: Given: A list of N numbers
Post: None
Return:
{
    wall ← 1                          // Place the wall at the leftmost end of the list
    while (wall < n)
    {
        Bubble(list, wall, n)
        wall ← wall + 1               // Move the wall one place to the right
    }
    return SortedList
}
```

Algorithm 8.61b *Exercise 61*

```
Algorithm: Bubble(list, wall, n)
Purpose: to bubble an unsorted list
Pre: Given: A list, N and location of the wall
Post: None
Return:
{
    cur ← n                       // Start from the end of the list
    while (cur  >  wall))         // Bubble the smallest to the left of unsorted list
    {
        if (A_cur < A_cur – 1)                    // Bubble one location to the left
        {
            Temp ← A_cur
            A_cur ← A_cur–1
            A_cur–1 ← Temp
        }
        cur ← cur – 1
    }
}
```

62. Algorithm S8.62 shows the pseudocode for the insertion sort using UML in Exercise 53. The inner loop finds the location (a hole) in the sorted list to insert the first element of the unsorted list. The actual insertion occurs after the inner loop is exited. We have used a true/false value (Found) to exit the inner loop when the appropriate location for insertion is found. Note that the wall is set to the left of the second element of the unsorted list because if the list has only one element, there is no need for insertion.

Algorithm S8.62 *Exercise 62*

```
Algorithm: InsertionSort(list, n)
Purpose: to sort a list using insertion sort
Pre: Given: A list of N numbers
Post: None
Return:
{
        wall ← 2                                  // Place the wall at the left of second element
        while (wall ≤ N)
        {
                Found ← false
                Temp  ← A_wall
                cur ← wall – 1                    // The current element to be the left of the wall
                while ((cur ≥ 1) AND Found = false))
                {
                        if (Temp < A_cur)         // Shift one location to the left
                        {
                                A_cur + 1 ← A_cur
                                cur  ← cur – 1
                        }
                        else Found ← true
                }
                A_cur +1 ← Temp                   // Insert
                wall ← wall + 1                   // Move the wall one place to the right
        }
}
```

63. Algorithm S8.63a shows the pseudocode for the insertion sort routine that uses a subprogram (Algorithm S8.63b).

Algorithm S8.63a *Exercise 63*

```
Algorithm: InsertionSort(list, n)
Purpose: to sort a list using insertion sort
Pre: Given: A list of N numbers
Post: None
Return: Sorted list
{
        wall ← 2
        while (wall < n)
        {
                Insert (list, wall, n)
                wall ← wall + 1
        }
}
```

Algorithm S8.63b *Exercise 63*

```
Algorithm: Insert(list, wall, n)
Purpose: Insert a number in a sorted list
Pre: Given: A of numbers and location of wall
Post: None
Return:
{
        Found ← false
        Temp ← A_wall
        cur ← wall − 1
        while ((cur ≥ 1) AND Found = false))
        {
                if (Temp < A_cur)
                {
                        A_cur + 1 ← A_cur
                        cur  ← cur − 1
                }
                else    Found ← true
        }
        A_cur +1 ← Temp
}
```

64. Algorithm S8.64 shows the pseudocode for sequential search.

Algorithm S8.64 *Exercise 64*

```
Algorithm: SequentialSearch(list, target, n)
Purpose: Apply a sequential search on a list of N unsorted numbers
Pre: list, target, n
Post: None
Return: flag, i       // flag shows the status of the search (true if target found, false if not)
{
        flag ← false
        i ← 1
        while ((i < n + 1) or (flag = false))
        {
                if (A_i = target)       flag ← true          // A_i is the ith number in the list
                i ← i + 1
        }
        return (flag, i)    // If flag is true, then i is the location of the element found.

}
```

65. Algorithm S8.65 shows the pseudocode for binary search.

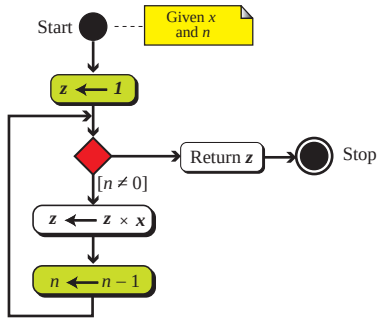
Algorithm S8.65 *Exercise 65*

```
Algorithm: BinarySearch(list, target, n)
Purpose: Apply a binary search a list of n sorted numbers
Pre: list, target, n
Post: None
Return: flag, i       // flag shows the status of the search (true if target found, false if not)

{
        flag ← false
        first ← 1
        last ← n
        while (first ≤ last)
        {
                mid = (first + last) / 2
                if (target < A_mid)          Last ← mid − 1     // A_A is the ith number in the list
                if (target > A_mid)          first ← mid + 1
                if (target = A_mid)          first ← Last + 1   // target is found
        }
        if (target > A_mid)                  i = mid + 1
        if (x ≤ A_mid)                       i = mid
        if (x = A_mid)                       flag ← true
        return (flag, i)
 // If flag is false, i is the location of the smallest number larger than the target
 // If flag is true, i is the location of the target
}
```

66. Figure S8.66 shows the UML for finding the power of an integer with an integral exponent.

Figure S8.66 *Exercise 66*



67. Algorithm S8.67 shows the pseudocode for finding the integer power of an integer.

Algorithm S8.67 *Exercise 67*

```
Algorithm: Power (x, n)
Purpose: Find x^n where x and n are integers
Pre: x, n
Post: None
Return: x^n
{
        z ← 1
        while (n ≠ 1)
        {
                z ← z × x
                n ← n − 1
        }
        return z
}
```