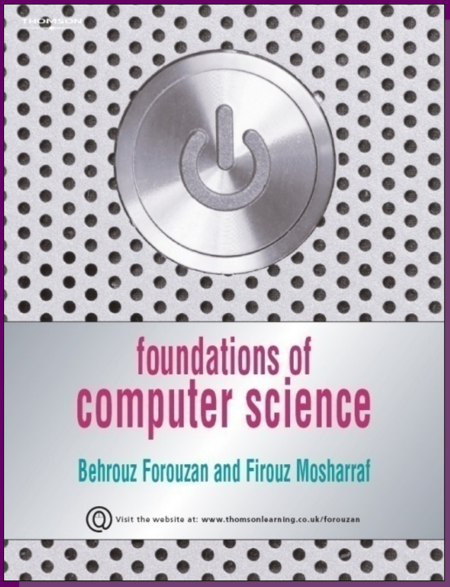
8
Algorithms



Foundations of Computer Science © Cengage Learning

Objectives

After studying this chapter, the student should be able to:

- Define an algorithm and relate it to problem solving.
- Define three construct and describe their use in algorithms.
- Describe UML diagrams and pseudocode and how they are used in algorithms.
- List basic algorithms and their applications.
- Describe the concept of sorting and understand the mechanisms behind three primitive sorting algorithms.
- Describe the concept of searching and understand the mechanisms behind two common searching algorithms.
- Define subalgorithms and their relations to algorithms.
- Distinguish between iterative and recursive algorithms

8-1 CONCEPT

In this section we informally define an **algorithm** and elaborate on the concept using an example.

Informal definition

An informal definition of an algorithm is:

Algorithm: a step-by-step method for solving a problem or doing a task.

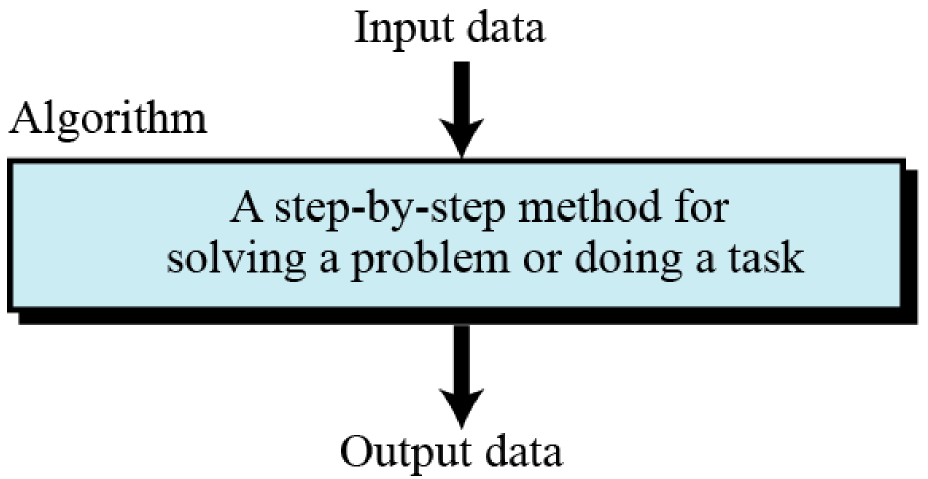


Figure 8.1 **Informal definition of an algorithm used in a computer**

Example

We want to develop an algorithm for finding the largest integer among a list of positive integers.
The algorithm should find the largest integer among a list of any values (for example 5, 1000, 10,000, 1,000,000). The algorithm should be general and not depend on the number of integers.

To solve this problem, we need an intuitive approach. First use a small number of integers (for example, five), then extend the solution to any number of integers.

Figure 8.2 shows one way to solve this problem. The algorithm receives a list of five integers as input and gives the largest integer as output.

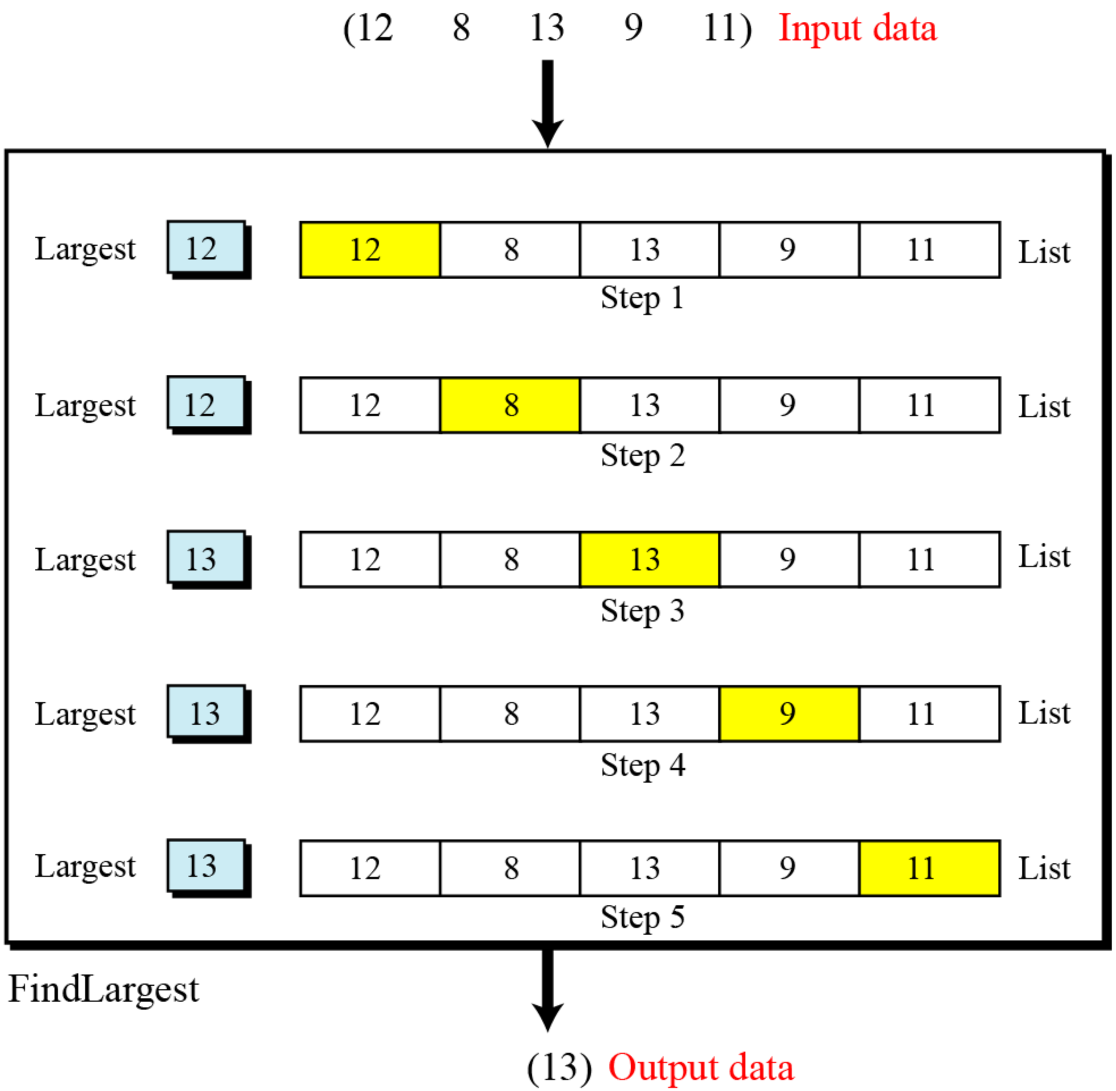


Figure 8.2 **Finding the largest integer among five integers**

Defining actions

Figure 8.2 does not show what should be done in each step. We can modify the figure to show more details.

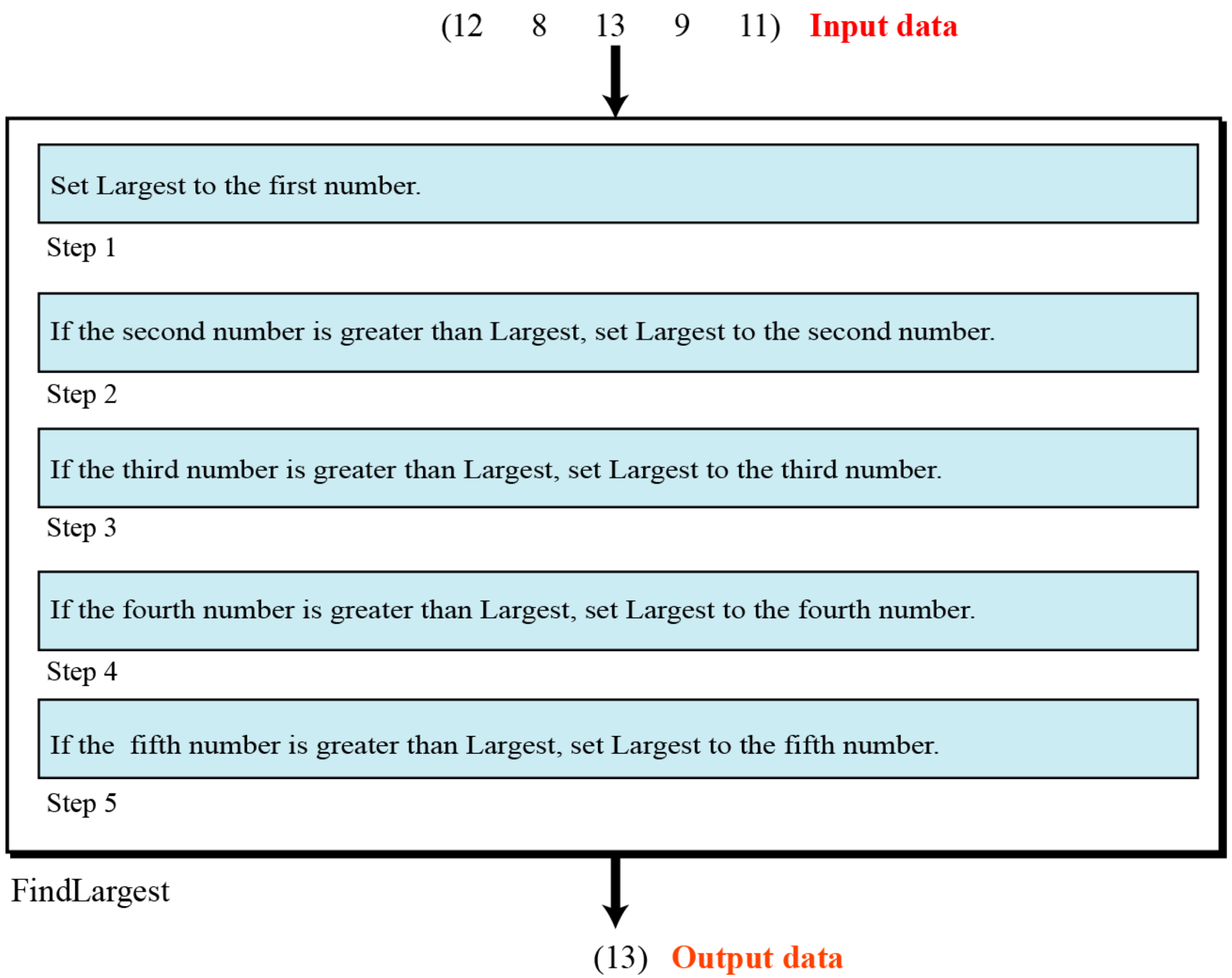


Figure 8.3 **Defining actions in FindLargest algorithm**

Refinement (1)

This algorithm needs refinement to be acceptable to the programming community.

There are **two** problems.

First, the action in the first step is different than those for the other steps.

Second, the wording is not the same in steps 2 to 5. We can easily redefine the algorithm to remove these two inconveniences by changing the wording in steps 2 to 5 to “If the current integer is greater than Largest, set Largest to the current integer.”

Refinement (2)

The reason that the first step is different than the other steps is because Largest is not initialized. If we initialize Largest to $-\infty$ (minus infinity), then the first step can be the same as the other steps, so we add a new step, calling it step 0 to show that it should be done before processing any integers.

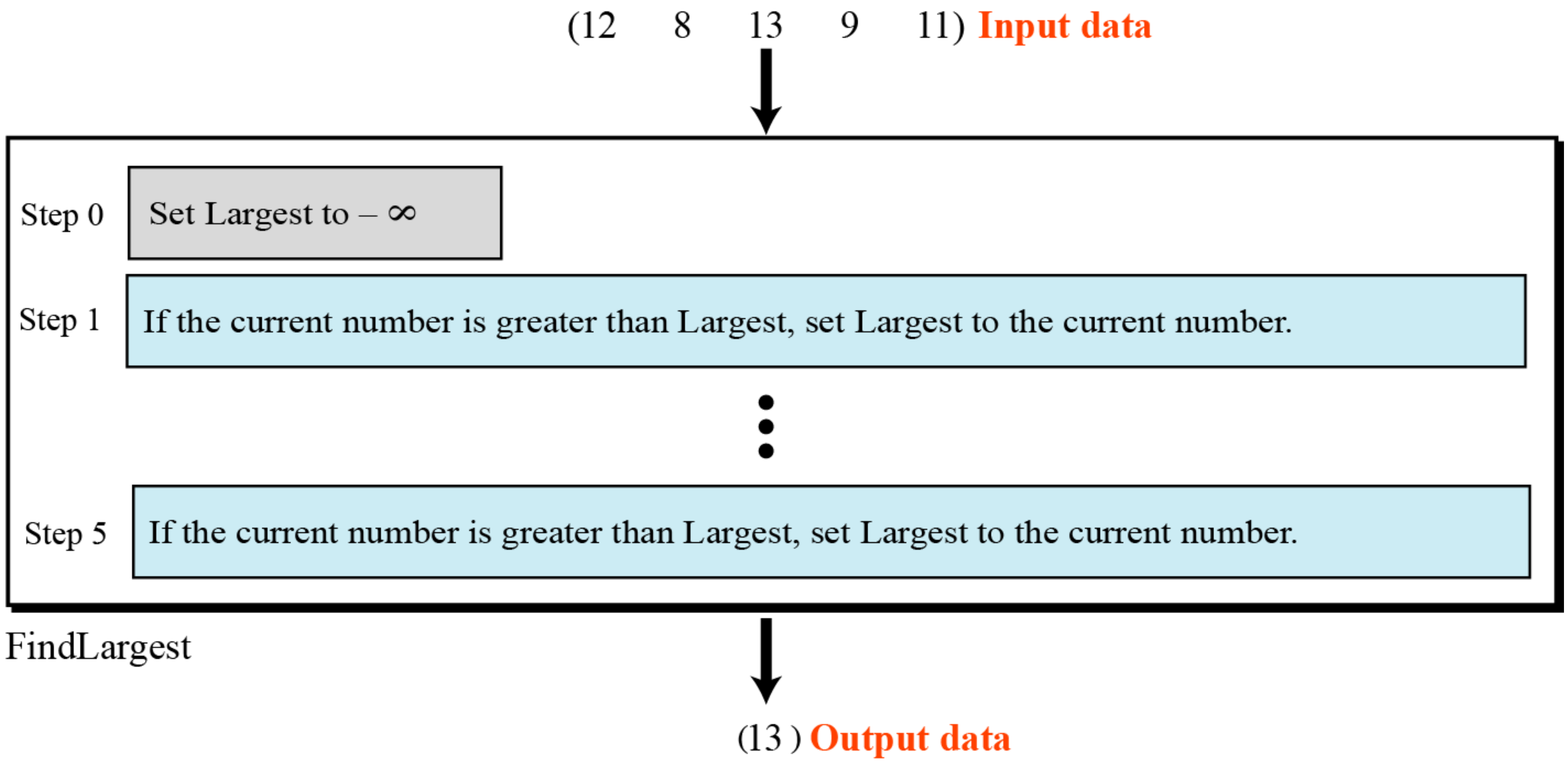


Figure 8.4 **FindLargest refined**

Generalization

Is it possible to generalize the algorithm? We want to find the largest of *n* positive integers, where *n* can be 1000, 1,000,000, or more.

We can follow Figure 8.4 and repeat each step. But if we change the algorithm to a program, then we need to actually type the actions for *n* steps!

There is a better way to do this. We can tell the computer to repeat the steps *n* times. We now include this feature in our pictorial algorithm (Figure 8.5).

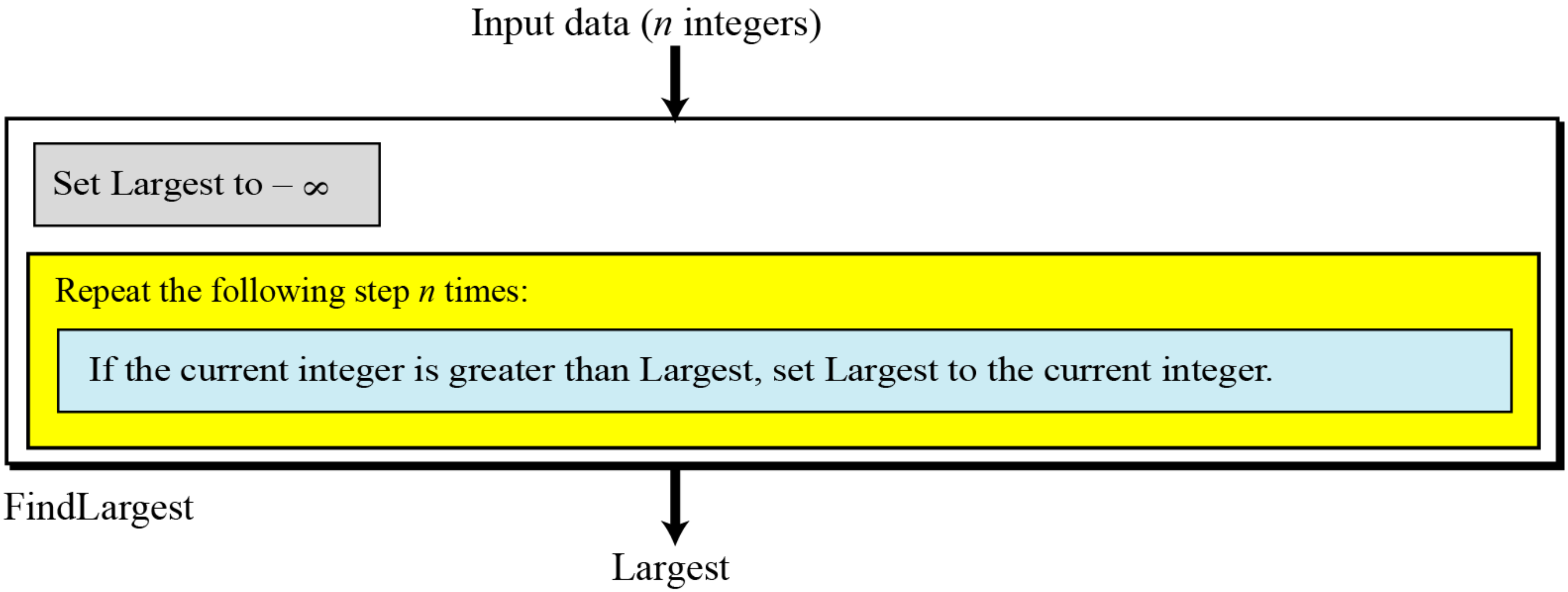


Figure 8.5 **Generalization of FindLargest**

8-2 THREE CONSTRUCTS

Computer scientists have defined **three constructs** for a structured program or algorithm (Figure 8.6). The idea is that a program must be made of a combination of only these three constructs:

1. **sequence**,
2. **decision** (selection)
3. **repetition**

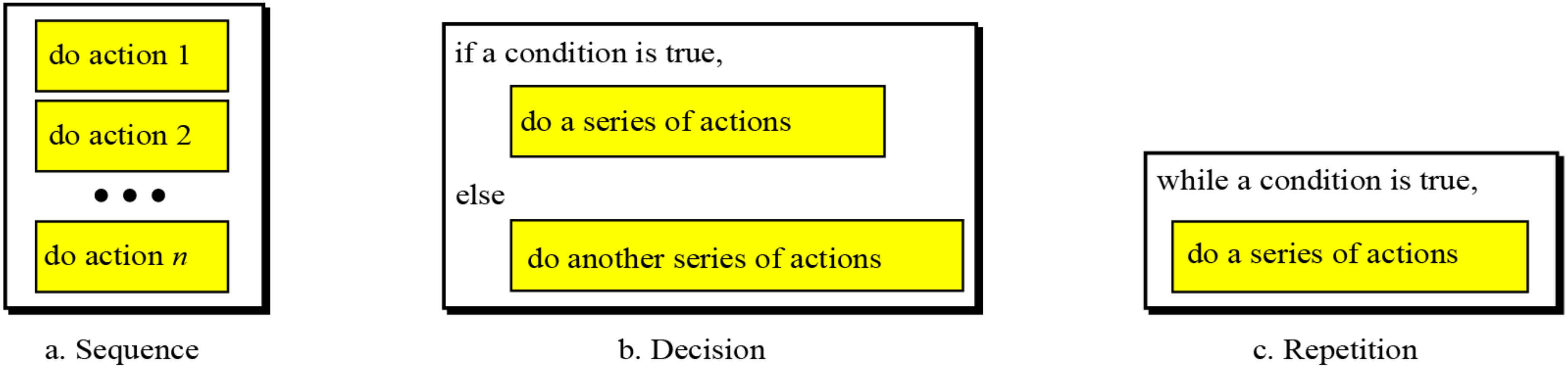


Figure 8.6 **Three constructs**

Sequence

An algorithm, and eventually a program, is a sequence of instructions, which can be a simple instruction or either of the other two constructs.

Decision

We need to test a condition. If the result of testing is true, we follow a sequence of instructions: if it is false, we follow a different sequence of instructions. This is called the decision (selection) construct. (if-else)

Repetition

In some problems, the same sequence of instructions must be repeated. We handle this with the repetition or ***loop*** construct. Finding the largest integer among a set of integers can use a construct of this kind.

8-3 ALGORITHM REPRESENTATION

During the last few decades, tools have been designed for this purpose. Two of these tools, **UML** and **pseudocode**, are presented here.

UML

Unified Modeling Language (UML) is a pictorial representation of an algorithm. It hides all the details of an algorithm in an attempt to give the "big picture" and to show how the algorithm flows from beginning to end.

UML is covered in detail in Appendix B. Here we show only how the three constructs are represented using UML (Figure 8.7).

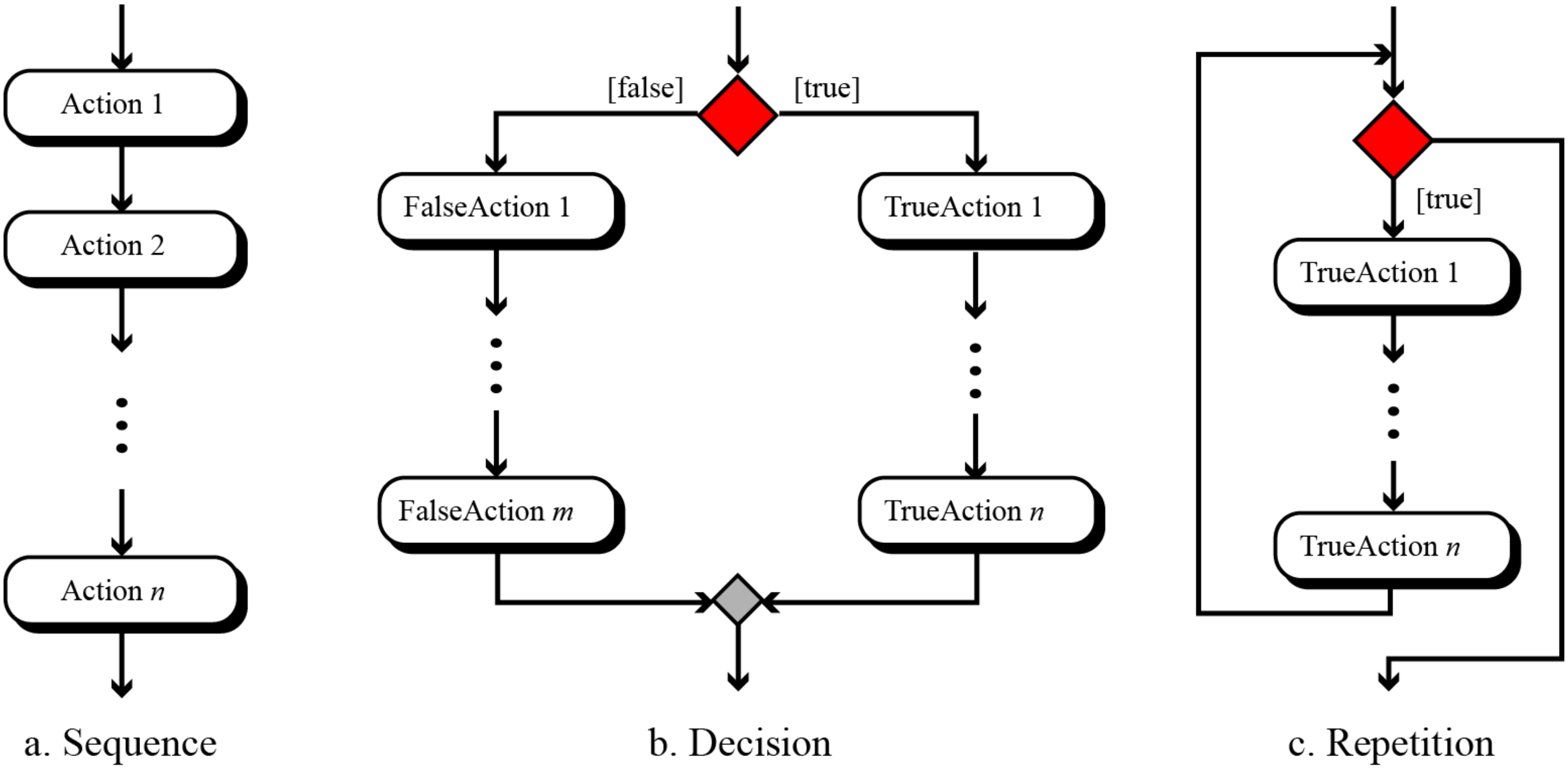


Figure 8.7 UML for three constructs

Pseudocode

Pseudocode is an English-language-like representation of an algorithm. There is no standard for pseudocode—some people use a lot of detail, others use less. Some use a code that is close to English, while others use a syntax like the Pascal programming language.

Pseudocode is covered in detail in Appendix C. Here we show only how the three constructs can be represented by pseudocode (Figure 8.8).

```
action 1
action 2
• • •
action n
```

a. Sequence

```
if (condition)

{
   trueAction(s)
}
 else
{
   falseAction(s)
}
```

b. Decision

```
while (condition)
{
     Action(s)
}
```

c. Repetition

Figure 8.8 **Pseudocode for three constructs**

Example 8.1

Write an algorithm in pseudocode that finds the sum of two integers.

Algorithm 8.1 Calculating the sum of two integers

```
Algorithm: SumOfTwo (first, second)
Purpose: Find the sum of two integers
Pre: Given: two integers (first and second)
Post: None
Return: The sum value
{
    sum ← first + second
    return sum
}
```

Example 8.2

Write an algorithm to change a numeric grade to a pass/no pass grade.

Algorithm 8.2 Assigning pass / no pass grade

Algorithm: Pass/**NoPass** (score)

Purpose: Creates a pass/no pass grade given the score

Pre: Given: the score to be changed to grade

Post: None

Return: The grade

```
{
    if (score ≥ 70)        grade ← "pass"
    else                   grade ← "nopass"
    return grade
}
```

Example 8.3

Write an algorithm to change a numeric grade (integer) to a letter grade.

Algorithm 8.3 Assigning a letter grade

Algorithm: **LetterGrade** (score)

Purpose: Find the letter grade corresponding to the given score

Pre: Given: a numeric score

Post: None

Return: A letter grade

```
{
    if (100 ≥ score ≥ 90)    grade ← 'A'
    if (80 ≥ score ≥ 89)     grade ← 'B'
    if (70 ≥ score ≥ 79)     grade ← 'C'
    if (60 ≥ score ≥ 69)     grade ← 'D'
    if (0 ≥ score ≥ 59)      grade ← 'F'
    return grade
}
```

Example 8.4

Write an algorithm to find the largest of a set of integers. We do not know the number of integers.

Algorithm 8.4 Finding the largest integer among a set of integers

```
Algorithm: FindLargest (list)

Purpose: Find the largest integer among a set of integers

Pre: Given: the set of integers

Post: None

Return: The largest integer

{
    largest ← – ∞
    while (more integers to check)
    {
        current ← next integer
        if (current > largest)        largest ← current

    }
    return largest
}
```

Example 8.5

Write an algorithm to find the largest of the first 1000 integers in a set of integers.

Algorithm 8.5 Finding the largest integer among the first 1000 integers

```
Algorithm: FindLargest2 (list)

Purpose: Find and return the largest integer among the first 1000 integers

Pre: Given: the set of integers with more than 1000 integers

Post: None

Return: The largest integer

{
    largest ← – ∞
    counter ← 1
    while (counter ≤ 1000)
    {
        current ← next integer
        if (current > largest)        largest ← current
        counter ← counter + 1
    }
    return largest
}
```

8-4 A MORE FORMAL DEFINITION

Now that we have discussed the concept of an algorithm and shown its representation, here is a more formal definition.

i

Algorithm:

An ordered set of unambiguous steps that produces a result and terminates in a finite time.

1. Well-Defined

An algorithm must be a well-defined, ordered set of instructions.

2. Unambiguous steps

Each step in an algorithm must be clearly and unambiguously defined.

If one step is to add two integers, we must define both “integers” as well as the “add” operation: we cannot for example use the same symbol to mean addition in one place and multiplication somewhere else.

3. Produce a result

An algorithm must produce a result, otherwise it is useless. The result can be data returned to the calling algorithm, or some other effect (for example, printing).

4. Terminate in a finite time

An algorithm must terminate (halt). If it does not (that is, it has an infinite loop), we have not created an algorithm. In Chapter 17 we will discuss solvable and unsolvable problems, and we will see that a solvable problem has a solution in the form of an algorithm that terminates.

8-5 BASIC ALGORITHMS

Several algorithms are used in computer science so prevalently that they are considered "**basic**".

We discuss the most common here. This discussion is very general: implementation depends on the language.

Summation

We can add two or three integers very easily, but how can we add many integers? The solution is simple: we use the add operator in a loop (Figure 8.9).

A summation algorithm has three logical parts:

1. Initialization of the sum at the beginning.
2. The loop, which in each iteration adds a new integer to the sum.
3. Return of the result after exiting from the loop.

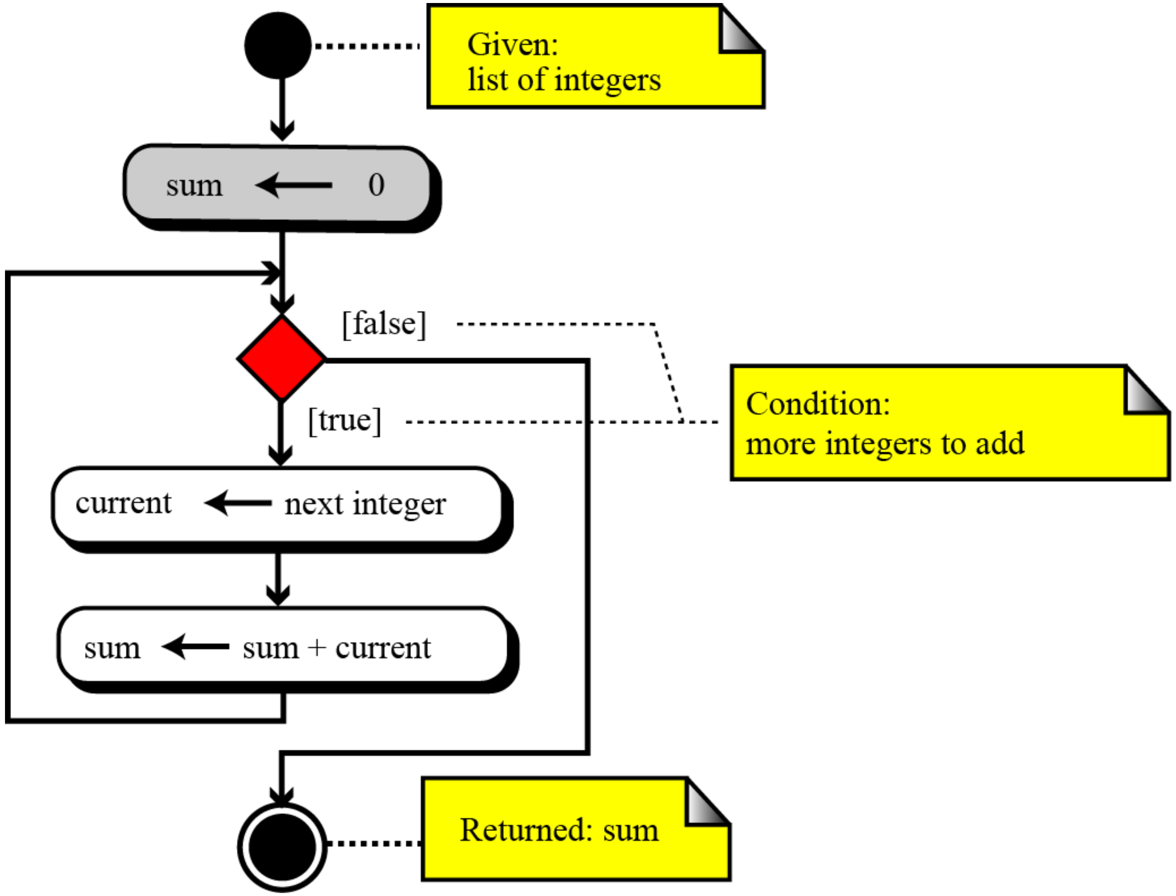


Figure 8.9 **Summation algorithm**

Product

Another common algorithm is finding the product of a list of integers. The solution is simple: use the multiplication operator in a loop (Figure 8.10).

A product algorithm has three logical parts:

1. Initialization of the product at the beginning.
2. The loop, which in each iteration multiplies a new integer with the product.
3. Return of the result after exiting from the loop.

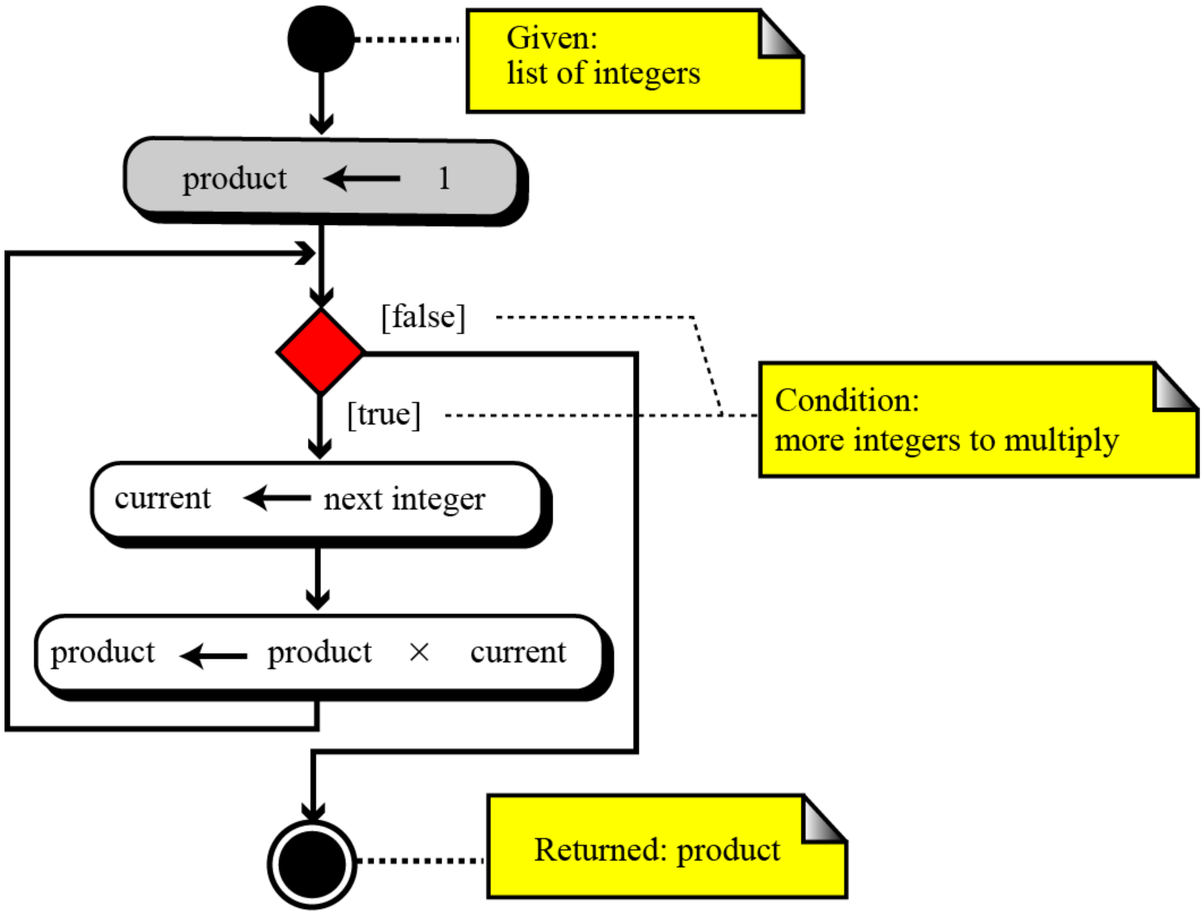


Figure 8.10 **Product algorithm**

Smallest and largest

The idea was to write a decision construct to find the larger of two integers. If we put this construct in a loop, we can find the largest of a list of integers.

Finding the smallest integer among a list of integers is similar, with two minor differences.

First, we use a decision construct to find the smaller of two integers.

Second, we initialize with a very large integer instead of a very small one.

Sorting

One of the most common applications in computer science is sorting, which is the process by which data is arranged according to its values.

In this section, we introduce three sorting algorithms: **selection sort**, **bubble sort** and **insertion sort**.

These three sorting algorithms are the foundation of faster sorting algorithms used in computer science today.

Selection sorts

In a **selection sort**, the list to be sorted is divided into two sublists—sorted and unsorted—which are separated by an imaginary wall.

We find the smallest element from the unsorted sublist and swap it with the element at the beginning of the unsorted sublist. After each selection and swap, the imaginary wall between the two sublists moves one element ahead.

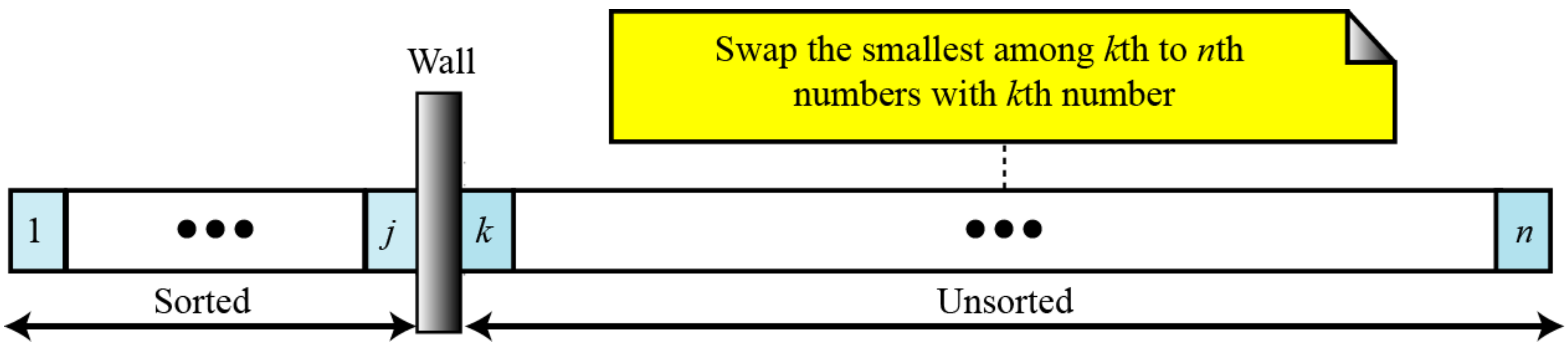


Figure 8.11 **Selection sort**

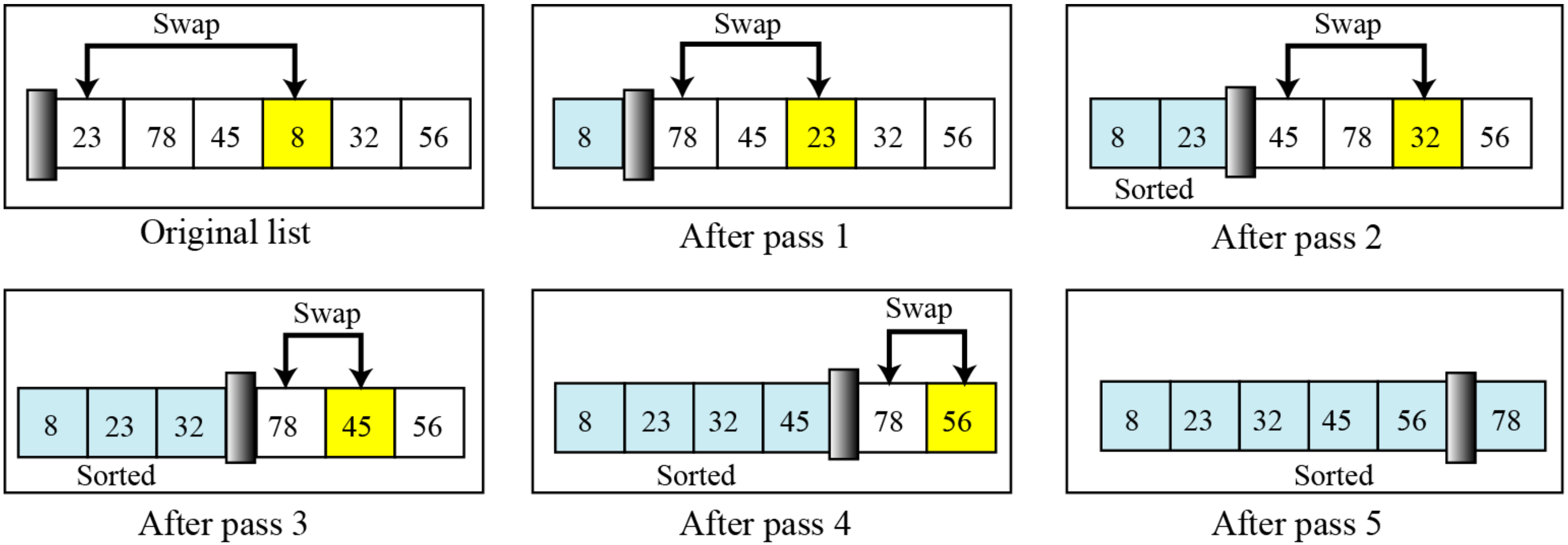


Figure 8.12 **Example of selection sort**

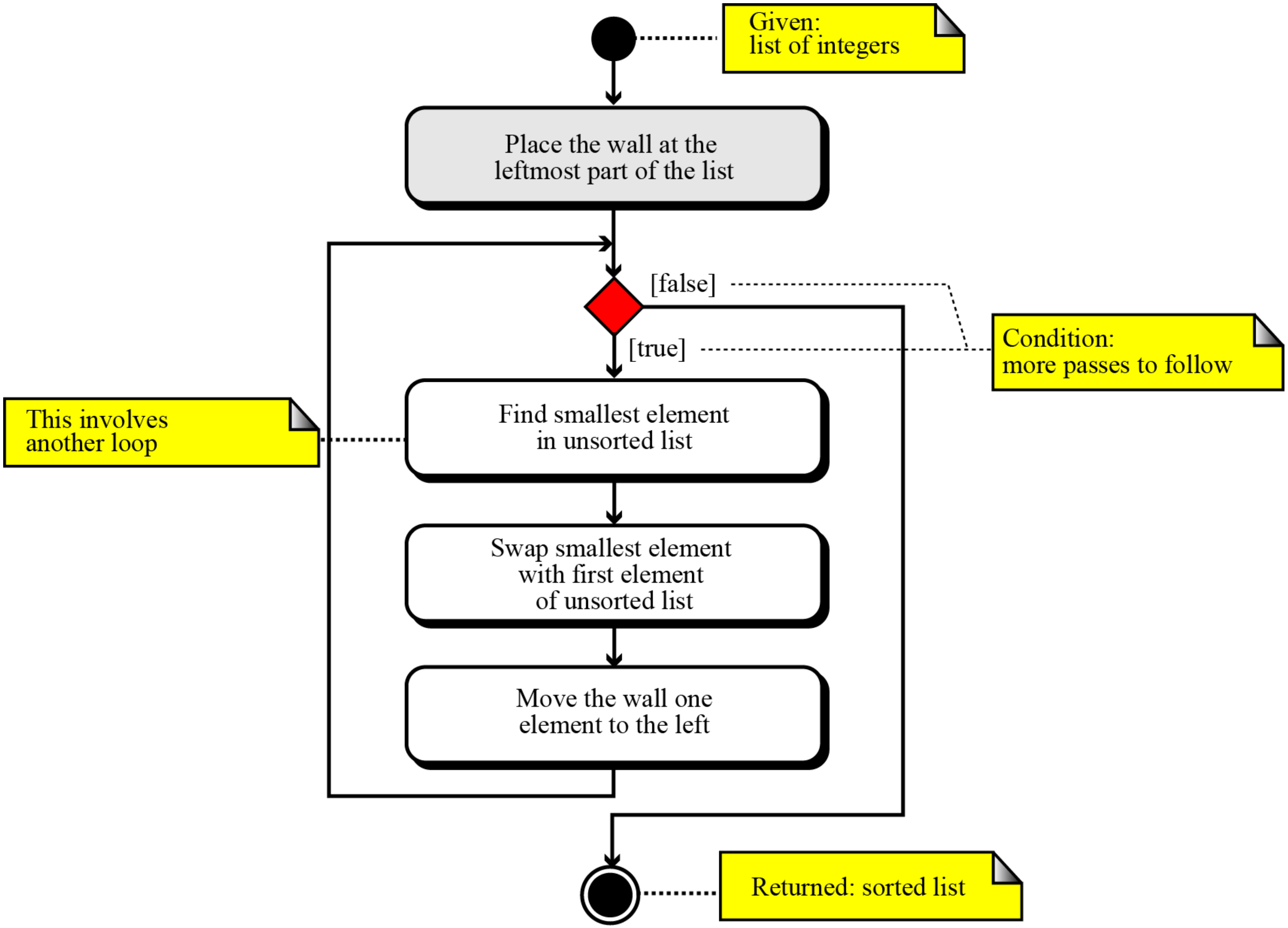


Figure 8.13 **Selection sort algorithm**

```
Function selectionSort(Type data[1..n])
Index i, j, max
   For i from 1 to n do
     max = i
        For j from i + 1 to n do
         If data[j] > data[max] then
          max = j
          Exchange data[i] and data[max]
End
```

Bubble sorts

In the bubble sort method, the list to be sorted is also divided into two sublists—sorted and unsorted.

The smallest element is bubbled up from the unsorted sublist and moved to the sorted sublist. After the smallest element has been moved to the sorted list, the wall moves one element ahead.

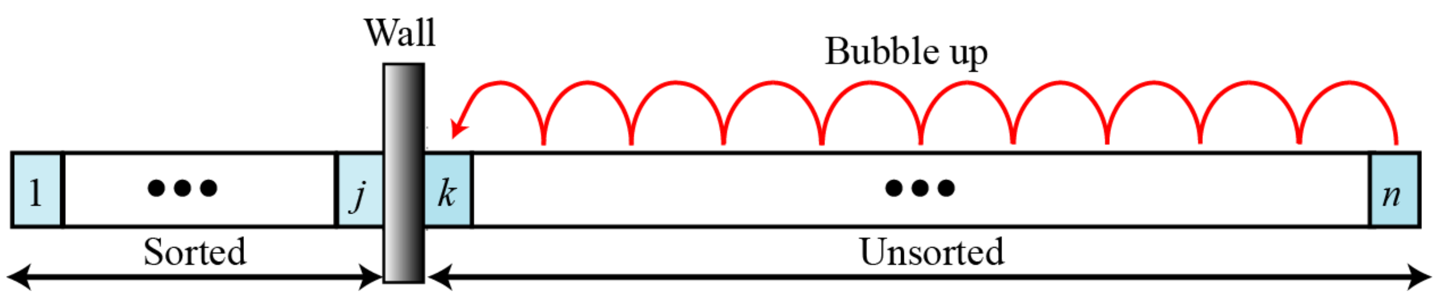


Figure 8.14 **Bubble sort**

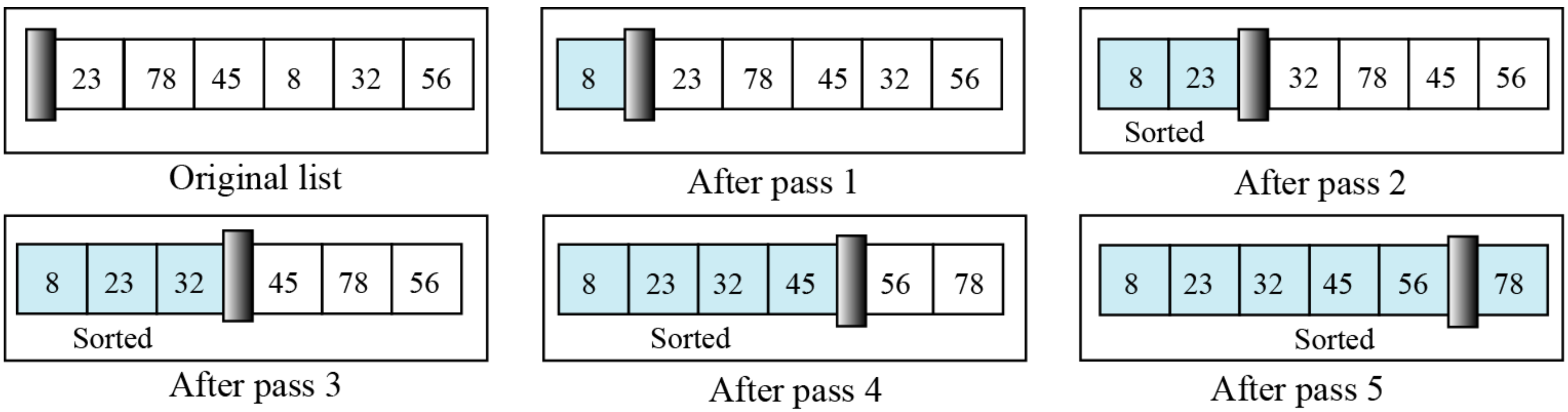


Figure 8.15 **Example of bubble sort**

```
Function bubbleSort(Type data[1..n])
    Index i, j;
    For i from n to 2 do
        For j from 1 to i - 1 do
            If data[j] > data[j + 1] then
                Exchange data[j] and data[j + 1]
End
```

Insertion sorts

The insertion sort algorithm is one of the most common sorting techniques, and it is often used by card players. Each card a player picks up is inserted into the proper place in their hand of cards to maintain a particular sequence.

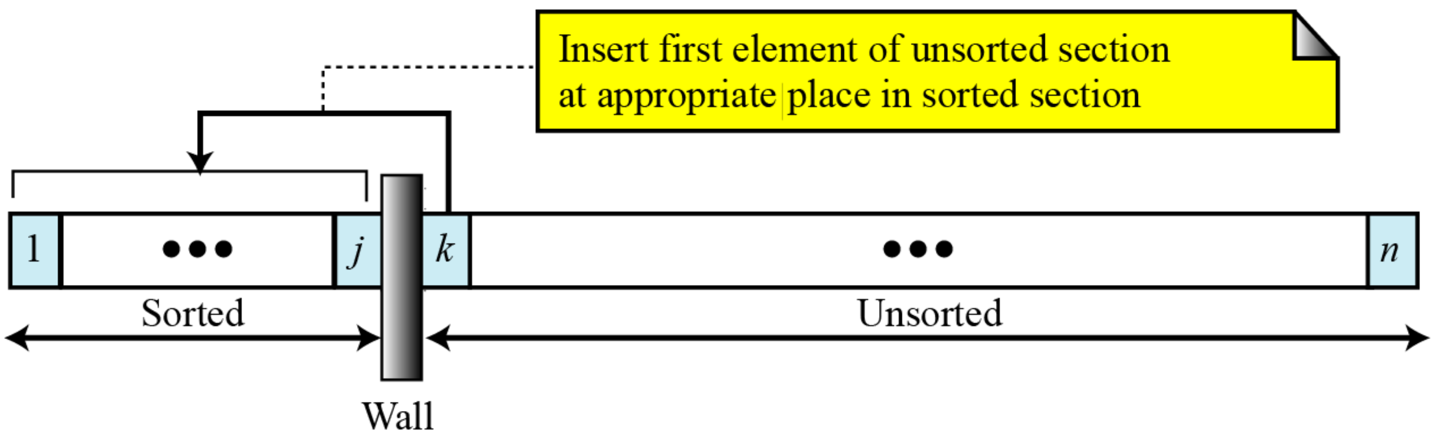


Figure 8.16 **Insertion sort**

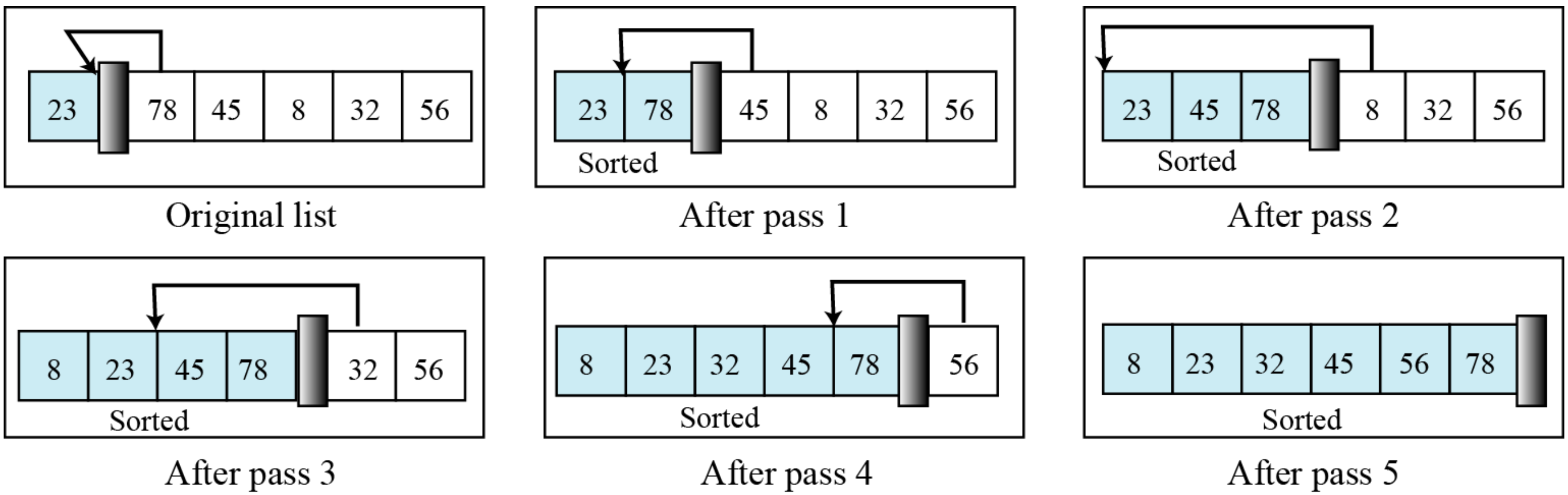


Figure 8.17 **Example of insertion sort**

```
Function insertionSort(Type data[1..n])
  Index i, j;
  Type value;
  For i from 2 to n do
     value = data[i];
     j = i - 1;
      While j >= 1 and data[j] > value do
         data[j + 1] = data[j];
          j = j - 1;
         data[j + 1] = value;
End
```

Searching

Another common algorithm in computer science is searching, which is the process of finding the location of a target among a list of objects.

There are two basic searches for lists: **sequential search** and **binary search**.

Sequential search can be used to locate an item in any list, whereas binary search requires the list first to be sorted.

Sequential search

Sequential search is used if the list to be searched is not ordered. Generally, we use this technique only for small lists, or lists that are not searched often.

In a sequential search, we start searching for the target from the beginning of the list. We continue until we either find the target or reach the end of the list.

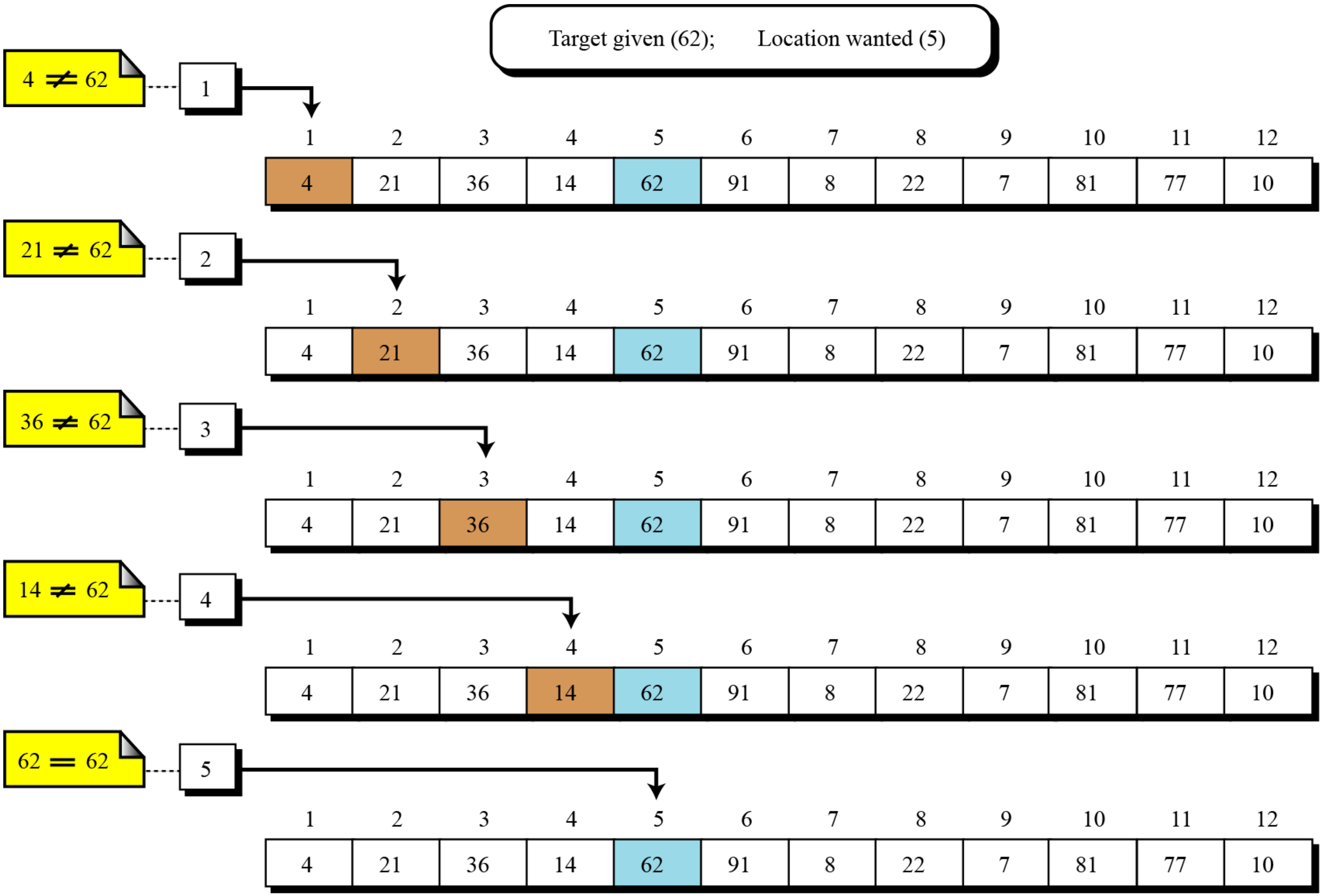


Figure 8.18 **An example of a sequential search**

Binary search

The sequential search algorithm is very slow. If we have a list of a million elements, we must do a million comparisons in the worst case. If the list is not sorted, this is the only solution. If the list is sorted, however, we can use a more efficient algorithm called binary search.

A binary search starts by testing the data in the element at the middle of the list. This determines whether the target is in the first half or the second half of the list. If it is in the first half, there is no need to further check the second half. If it is in the second half, there is no need to further check the first half. In other words, we eliminate half the list from further consideration.

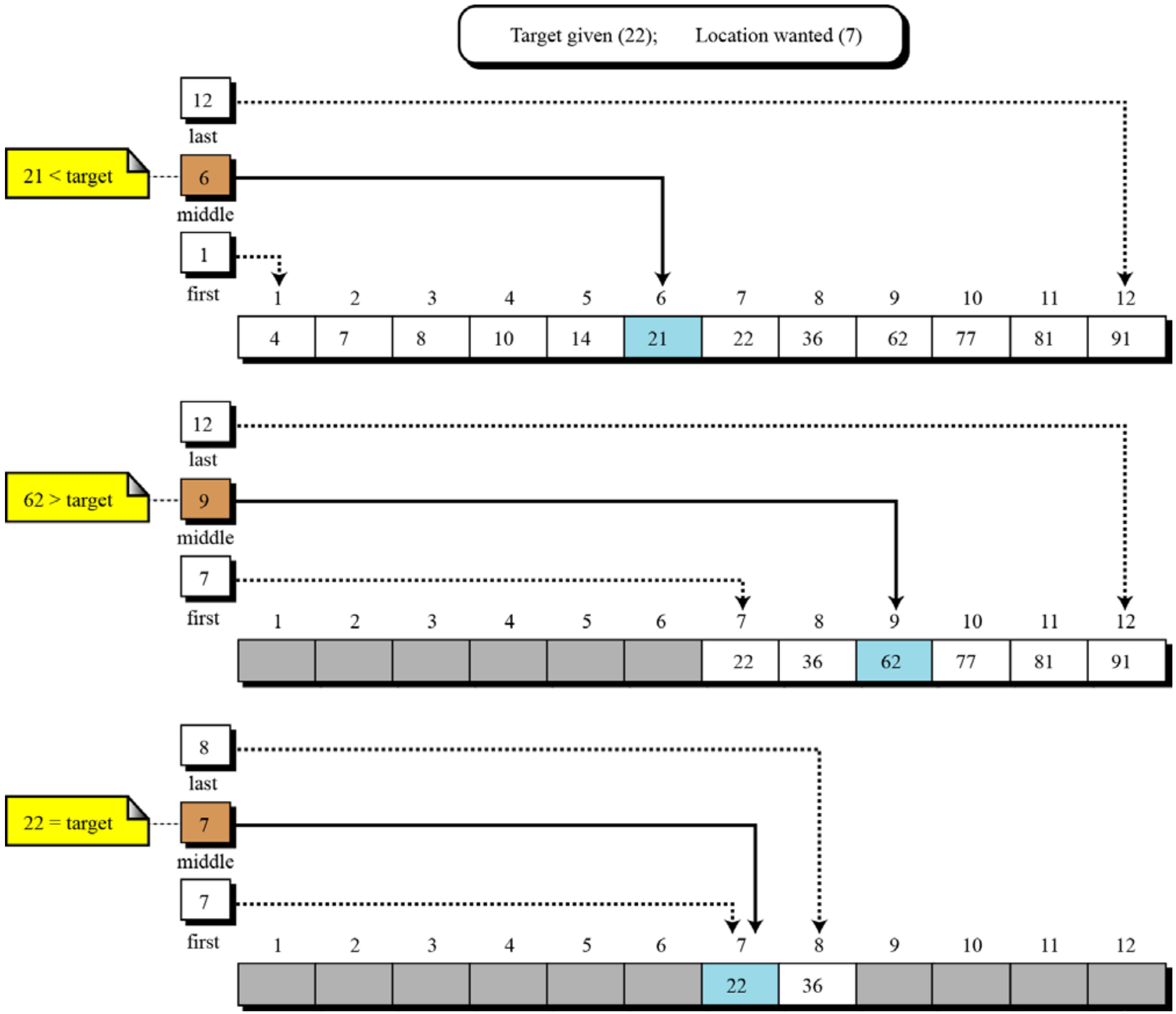


Figure 8.19 Example of a binary search

```
void binarysearch(Type data[1..n], Type search)
{
  Index low = 1;
  Index high = n;
  while (low <= high)
   {
     Index mid = (low + high) / 2;
     if (data[mid] = search)
          { print mid; return; }
     else if (data[mid] > search)
          { high = mid - 1; }
     else if (data[mid] < search) { low = mid + 1; }
    }
   print "Not found";
}
```

8-6 SUBALGORITHMS

The principles of structured programming, however, require that an algorithm be broken into small units called **subalgorithms**.

Each subalgorithm is in turn divided into smaller subalgorithms. A good example is the algorithm for the selection sort in Figure 8.13.

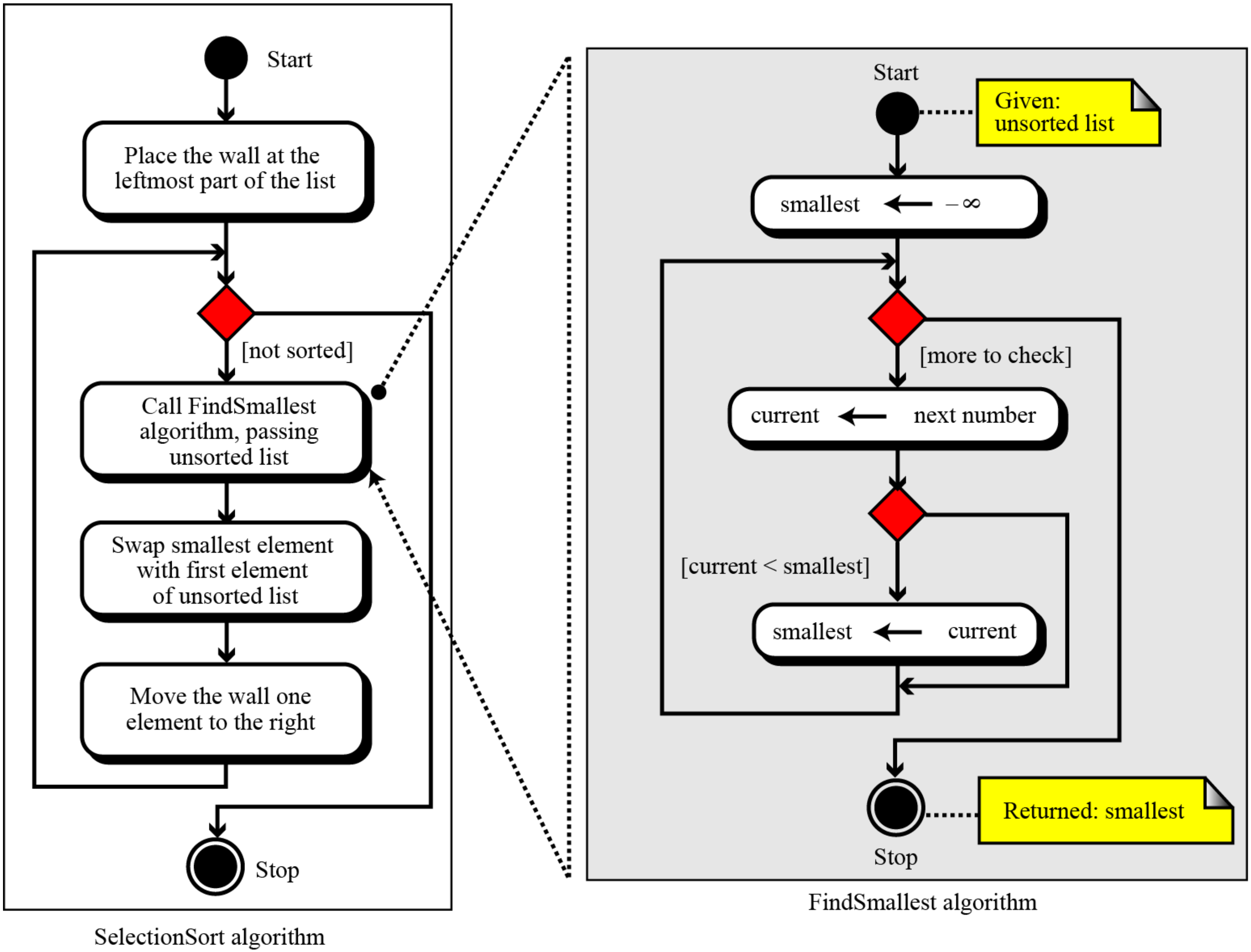


Figure 8.20 **Concept of a subalgorithm**

Structure chart

A structure chart is a high level design tool that shows the relationship between algorithms and subalgorithms.

It is used mainly at the design level rather than at the programming level. We briefly discuss the structure chart in Appendix D.

8-7 RECURSION

In general, there are two approaches to writing algorithms for solving a problem. One uses **iteration**, the other uses **recursion**.

Recursion is a process in which an algorithm calls itself.

Iterative definition

To study a simple example, consider the calculation of a factorial. The factorial of a integer is the product of the integral values from 1 to the integer. The definition is iterative (Figure 8.21). An algorithm is iterative whenever the definition does not involve the algorithm itself.

$$\text{Factorial}(n) = \begin{bmatrix} 1 & & & \text{if } n = 0 \\ \\ n \times (n-1) \times (n-2) & \cdots & 3 \times 2 \times 1 & \text{if } n > 0 \end{bmatrix}$$

Figure 8.21 **Iterative definition of factorial**

Recursive definition

An algorithm is defined recursively whenever the algorithm appears within the definition itself. For example, the factorial function can be defined recursively as shown in Figure 8.22.

$$\text{Factorial}(n) = \begin{cases} 1 & \text{if } n = 0 \\ n \times \text{Factorial}(n-1) & \text{if } n > 0 \end{cases}$$

Figure 8.22 **Recursive definition of factorial**

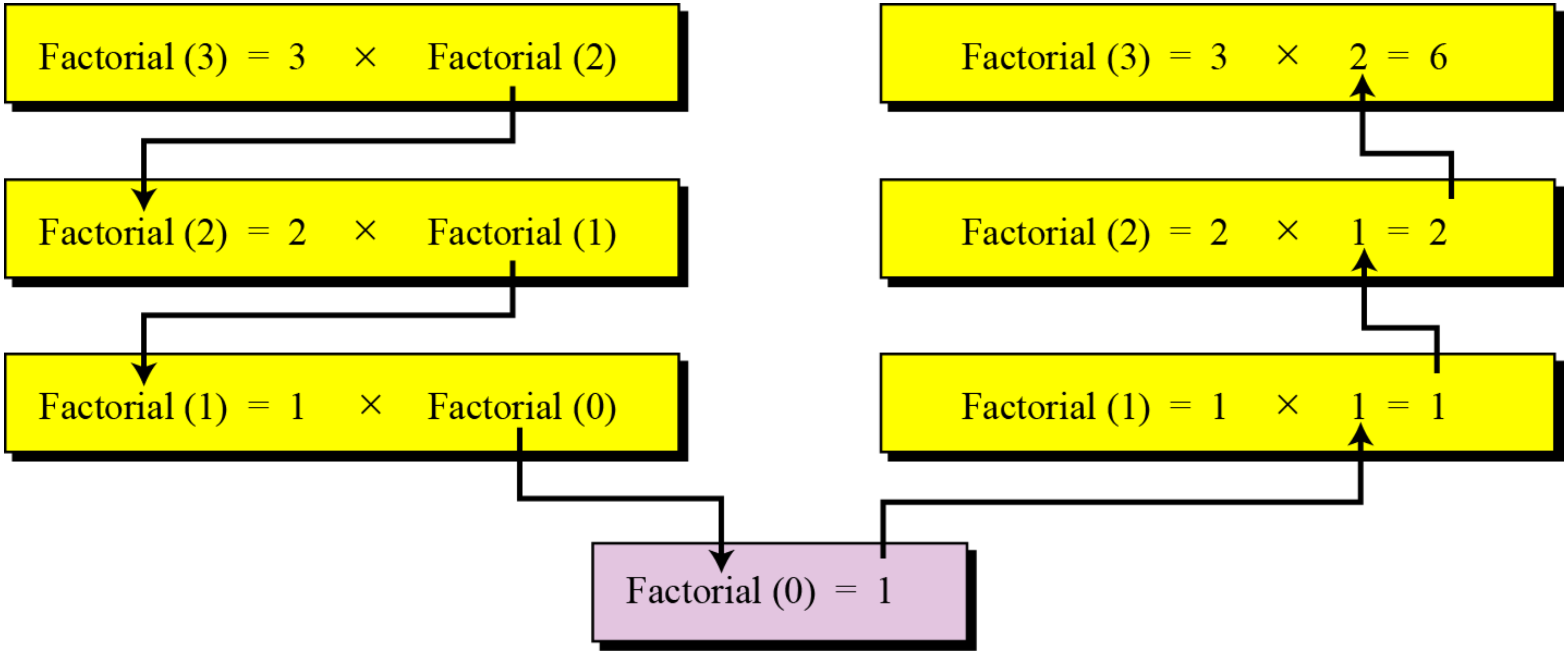


Figure 8.23 **Tracing the recursive solution to the factorial problem**

Iterative solution

This solution usually involves a loop.

Algorithm 8.6 An iterative solution to the factorial problem

```
Algorithm: Factorial (n)
Purpose: Find the factorial of a number using a loop
Pre: Given: n
Post: None
Return: n!
{
        F ← 1
        i ← 1
        while (i ≤ n)
        {
                F ← F × i
                i ← i + 1
        }
        return F
}
```

Recursive solution

The recursive solution does not need a loop, as the recursion concept itself involves repetition.

Algorithm 8.7 Pseudocode for recursive solution of factorial problem

```
Algorithm: Factorial (n)
Purpose: Find the factorial of a number using recursion
Pre: Given: n
Post: None
Return: n!
{
    if (n = 0)          return 1
    else                return n × Factorial (n −1)
}
```