Programming Logic and Design
Ninth Edition

Chapter 8

Advanced Data Handling Concepts

Objectives

In this chapter, you will learn about:

- The need for sorting data
- The bubble sort algorithm
- Sorting multifield records
- The insertion sort algorithm
- Multidimensional arrays
- Indexed files and linked lists

Understanding the Need for Sorting Data

- **Sequential order**
  - Placed in order based on the value of some field
    - Alphabetical or numerical
  - Ascending order (arranging records from A to Z or lowest to highest)
  - Descending order (arranging records from Z to A or highest to lowest)
  - The **median** value in a list is the value of the middle item when the values are listed in order
    - not the same as the arithmetic average, or **mean**

Understanding the Need for Sorting Data (continued -1)

- When computers sort data
  - Always use numeric values when making comparisons between values
  - Letters are translated into numbers using coding schemes such as ASCII, Unicode, or EBCDIC
    - “B” is numerically one greater than “A”
    - “y” is numerically one less than “z”
  - Whether “A” is represented by a number that is greater or smaller than the number representing “a” depends on your system

Understanding the Need for Sorting Data (continued -2)

- Professional programmers might never have to write a program that sorts data
  - Organizations can purchase prewritten sorting programs
  - Many popular language compilers come with built-in methods that can sort data for you
- Beneficial to understand the sorting process

Using the Bubble Sort Algorithm

- An **algorithm** is a list of instructions that accomplish a task, such as a sort
- **Bubble sort**
  - Items in a list are compared with each other in pairs
  - Items are then swapped based on which is larger or smaller
  - Sometimes called a sinking sort
    - Ascending sorts put the smaller item on top so the largest item sinks to the bottom
    - Descending sorts put the larger item on top so the smallest item sinks to the bottom

Understanding Swapping Values

- To **swap values** stored in two variables:
  - Exchange their values
  - Set the first variable equal to the value of the second
  - Set the second variable equal to the value of the first
- There is a trick to swapping any two values

Understanding Swapping Values (continued -1)

- Example
  - `num score1 = 90`
  - `num score2 = 85`
- This is what could go wrong
  - If you first assign `score1` to `score2` using a statement such as `score2 = score1`
  - Both `score1` and `score2` hold 90, and the value 85 is lost
- Must create a temporary variable to hold one of the scores
  - `temp = score2` (`temp` and `score2` both contain 85)
  - `score2 = score1` (`score2` contains 90)
  - `score1 = temp` (`score1` contains 85)

Understanding Swapping Values (continued -2)

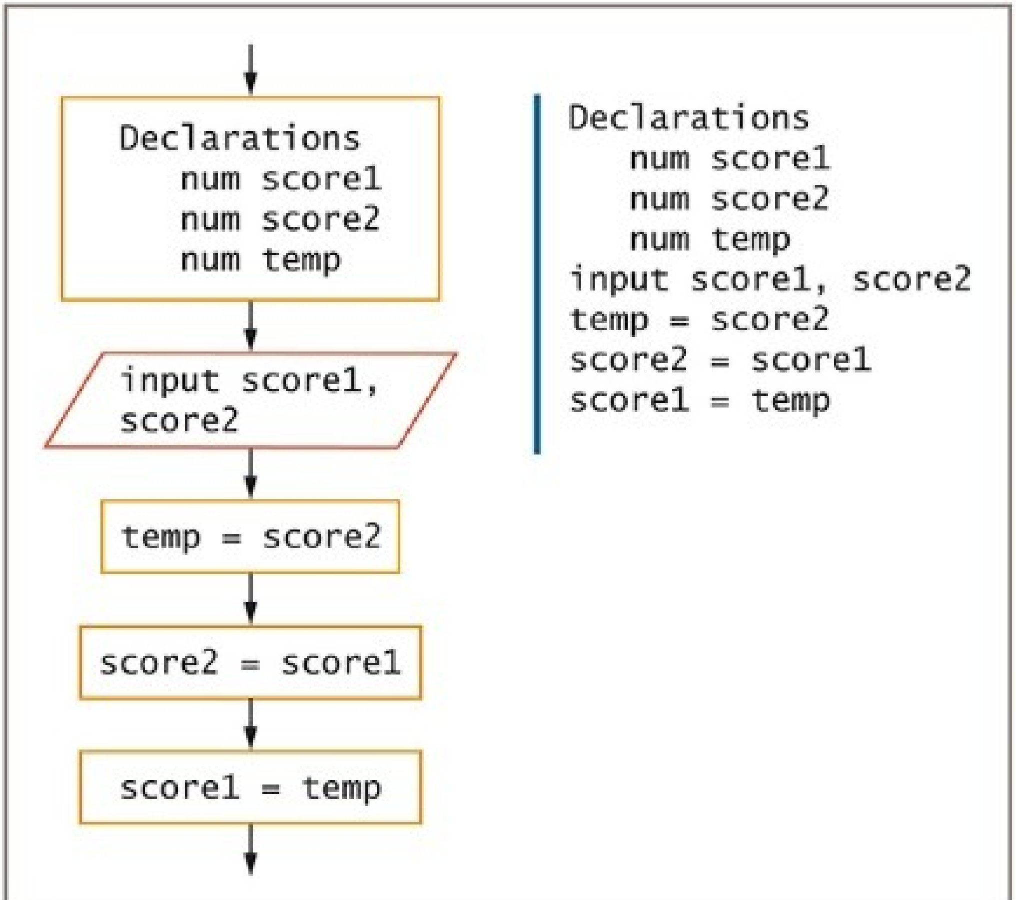
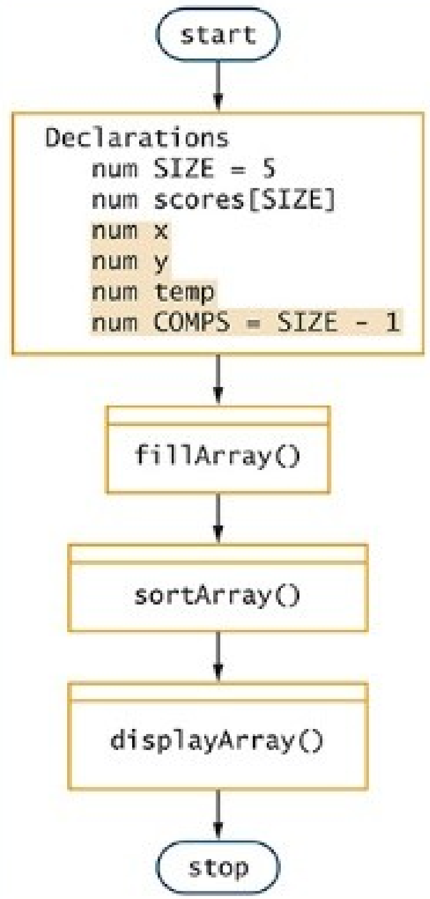


Figure 8-1 Program segment that swaps two values

Understanding the Bubble Sort




```
start
   Declarations
      num SIZE = 5
      num scores[SIZE]
      num x
      num y
      num temp
      num COMPS = SIZE - 1
   fillArray()
   sortArray()
   displayArray()
stop
```

Figure 8-2 Mainline logic for program that accepts, sorts, and displays scores

Understanding the Bubble Sort (continued -1)

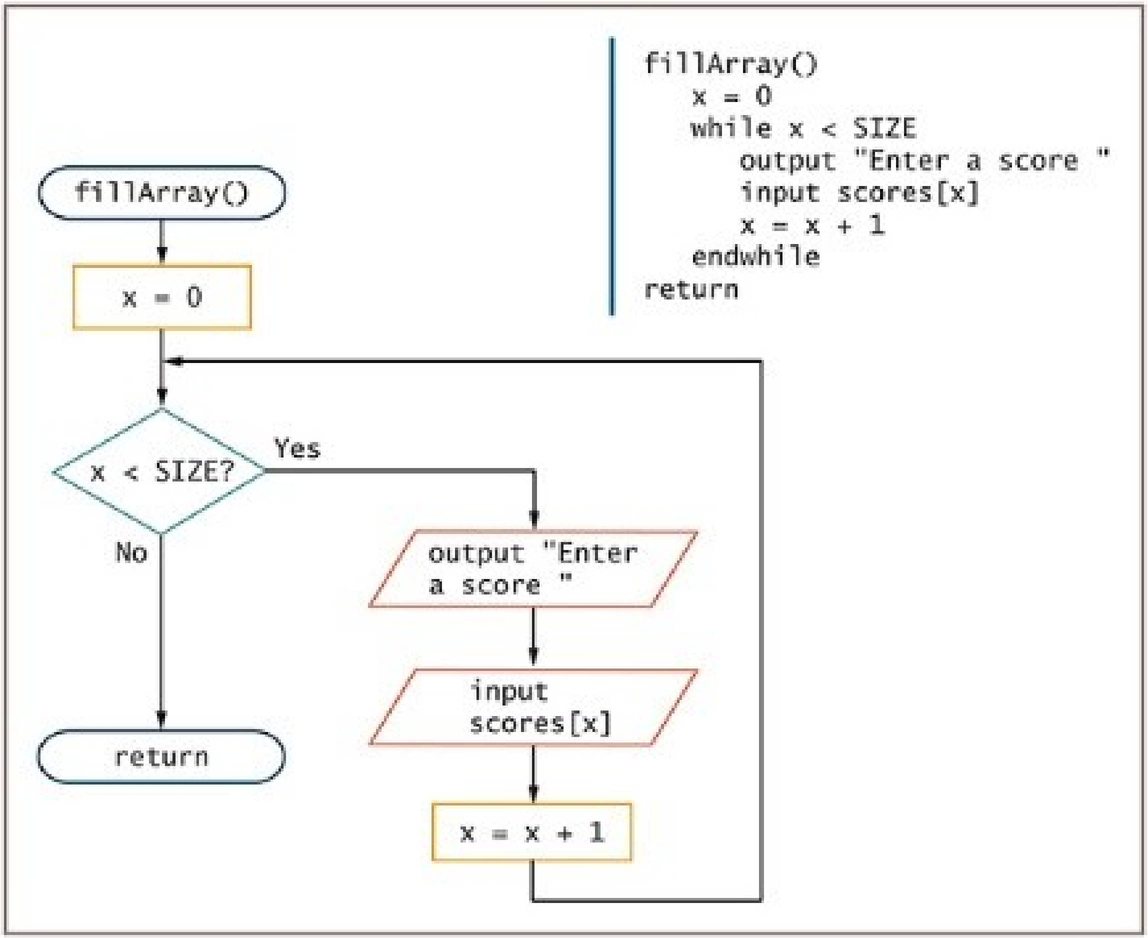


Figure 8-3 The fillArray() method

Understanding the Bubble Sort (continued -2)

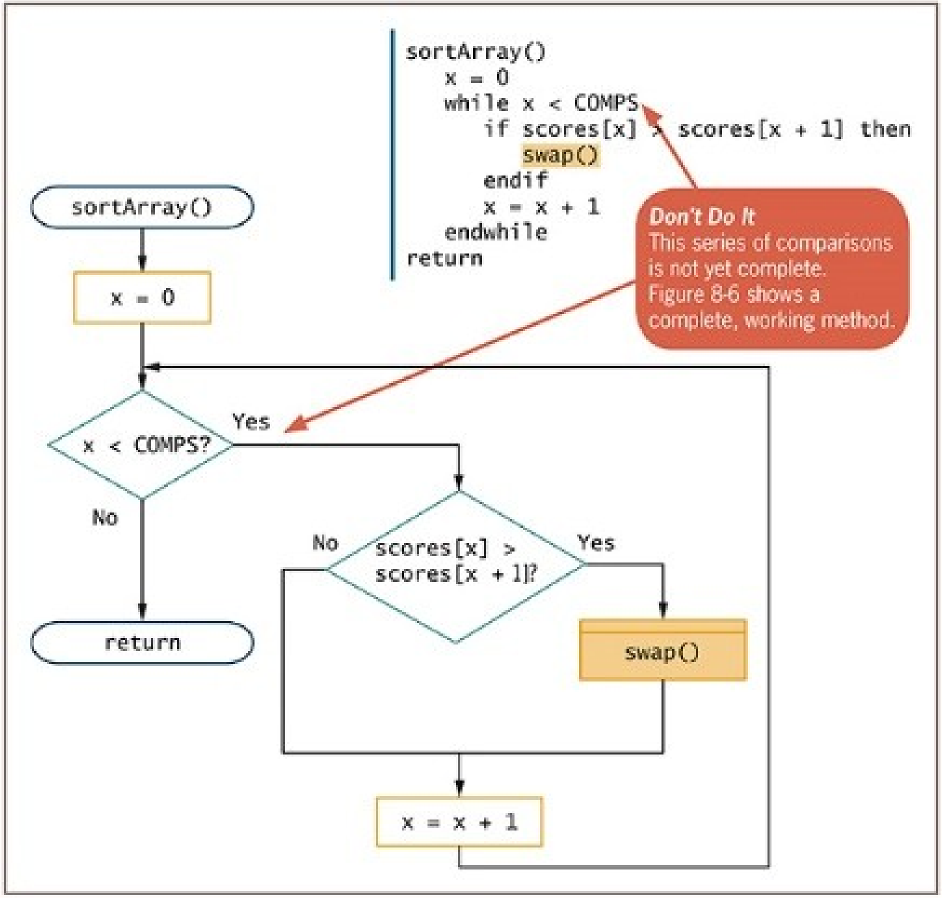
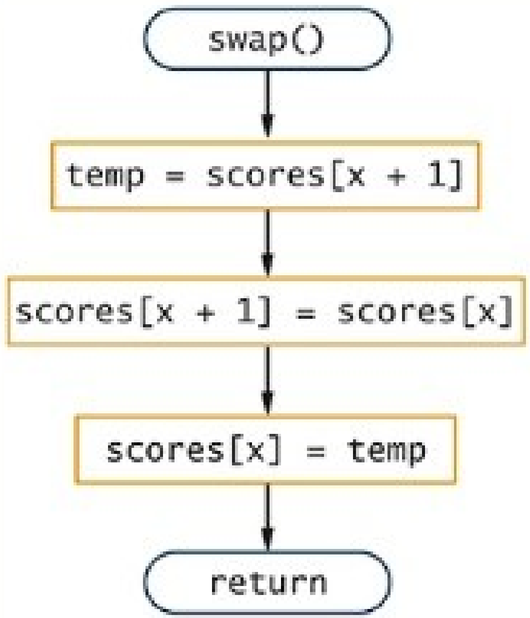


Figure 8-4 The incomplete sortArray() method

Understanding the Bubble Sort (continued -3)




```
swap()
   temp = scores[x + 1]
   scores[x + 1] = scores[x]
   scores[x] = temp
return
```

Figure 8-5 The swap() method

Understanding the Bubble Sort (continued -4)

- Initial list
  - `score[0] = 90`
  - `score[1] = 85`
  - `score[2] = 65`
  - `score[3] = 95`
  - `score[4] = 75`

- 90 and 65 are switched
  - `score[0] = 85`
  - `score[1] = 65`
  - `score[2] = 90`
  - `score[3] = 95`
  - `score[4] = 75`

- 85 and 90 are switched
  - `score[0] = 85`
  - `score[1] = 90`
  - `score[2] = 65`
  - `score[3] = 95`
  - `score[4] = 75`

- No change: 90 and 95 in order
  - `score[0] = 85`
  - `score[1] = 65`
  - `score[2] = 90`
  - `score[3] = 95`
  - `score[4] = 75`

Understanding the Bubble Sort (continued -5)

- 75 and 95 are switched
  - `score[0] = 85`
  - `score[1] = 65`
  - `score[2] = 90`
  - `score[3] = 75`
  - `score[4] = 95`

- No change: 90 and 75 in order
  - `score[0] = 65`
  - `score[1] = 85`
  - `score[2] = 75`
  - `score[3] = 90`
  - `score[4] = 95`

- Back to top: 65 and 85 switch
  - `score[0] = 65`
  - `score[1] = 85`
  - `score[2] = 90`
  - `score[3] = 75`
  - `score[4] = 95`

- 75 and 85 switch: sorted!
  - `score[0] = 65`
  - `score[1] = 75`
  - `score[2] = 85`
  - `score[3] = 90`
  - `score[4] = 95`

Understanding the Bubble Sort

(continued -6)

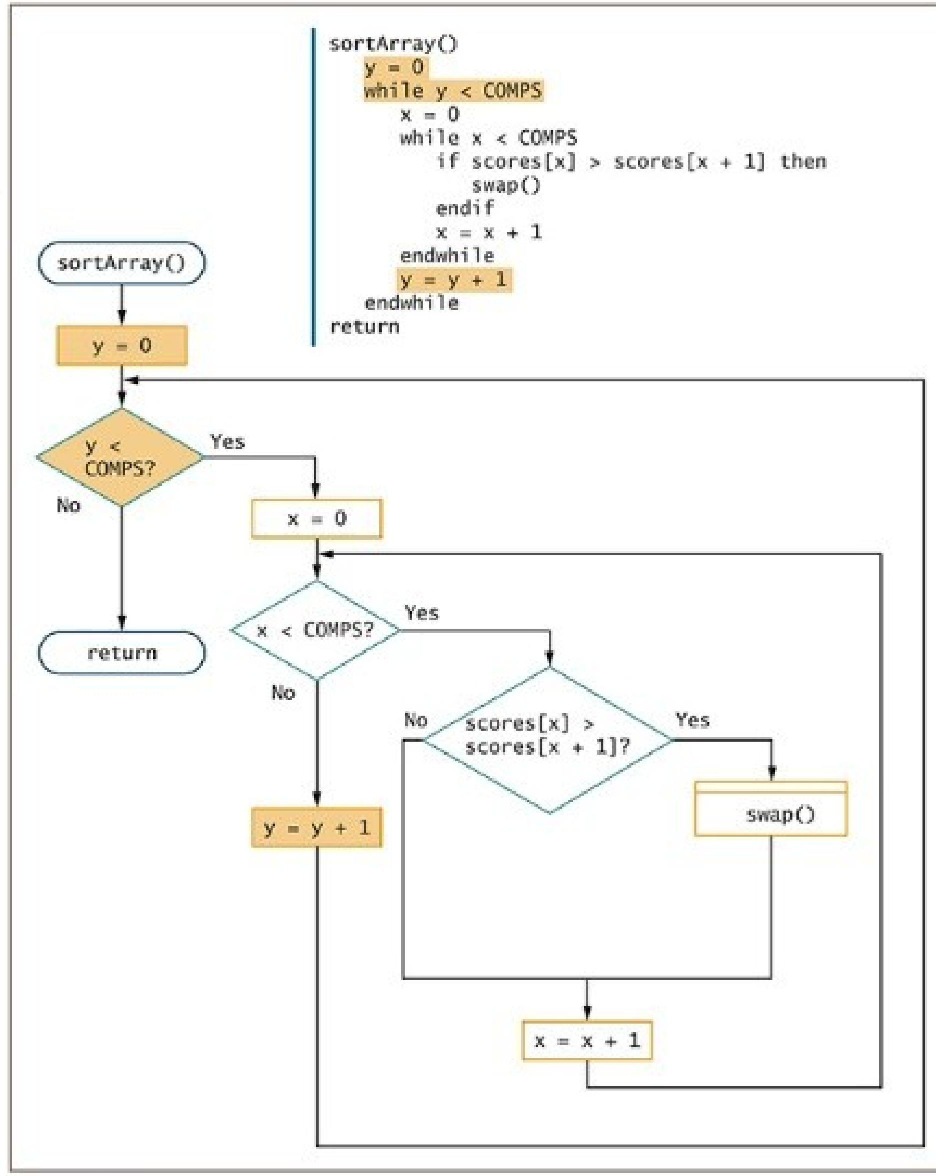
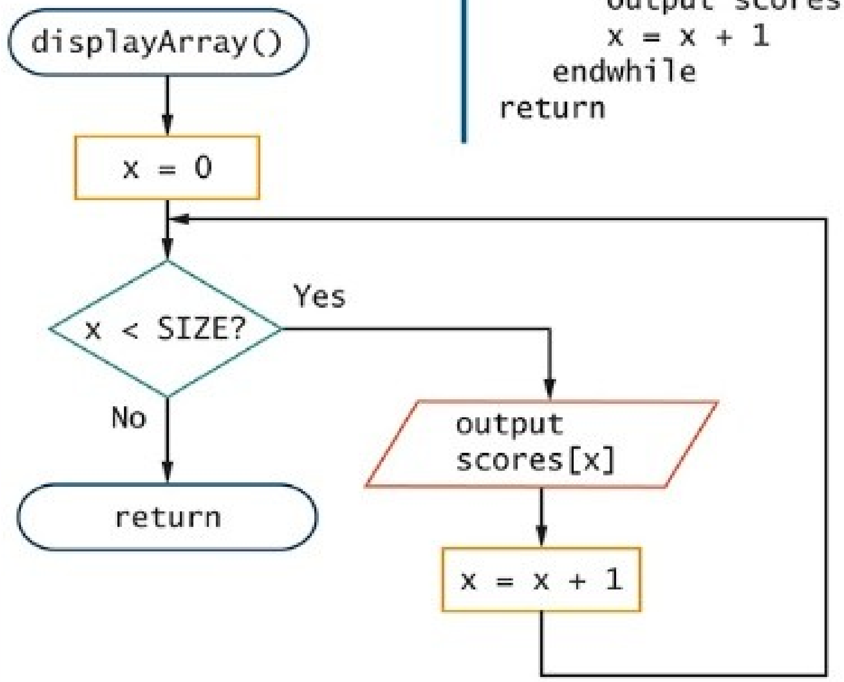


Figure 8-6 The completed sortArray() method

Understanding the Bubble Sort (continued -7)




```
displayArray()
   x = 0
   while x < SIZE
      output scores[x]
      x = x + 1
   endwhile
return
```

Figure 8-7 The displayArray() method

Understanding the Bubble Sort (continued -8)

- Nested loops
  - Inner loop swaps out-of-order pairs
  - Outer loop goes through the list multiple times
- General rules
  - Greatest number of pair comparisons is one less than the number of elements in the array
  - Number of times you need to process the list of values is one less than the number of elements in the array

Sorting a List of Variable Size

- Might not know how many array elements will hold valid values
- Keep track of the number of elements stored in an array
  - Store number of elements
  - Compute comparisons as number of elements - 1

Sorting a List of Variable Size

(continued -1)

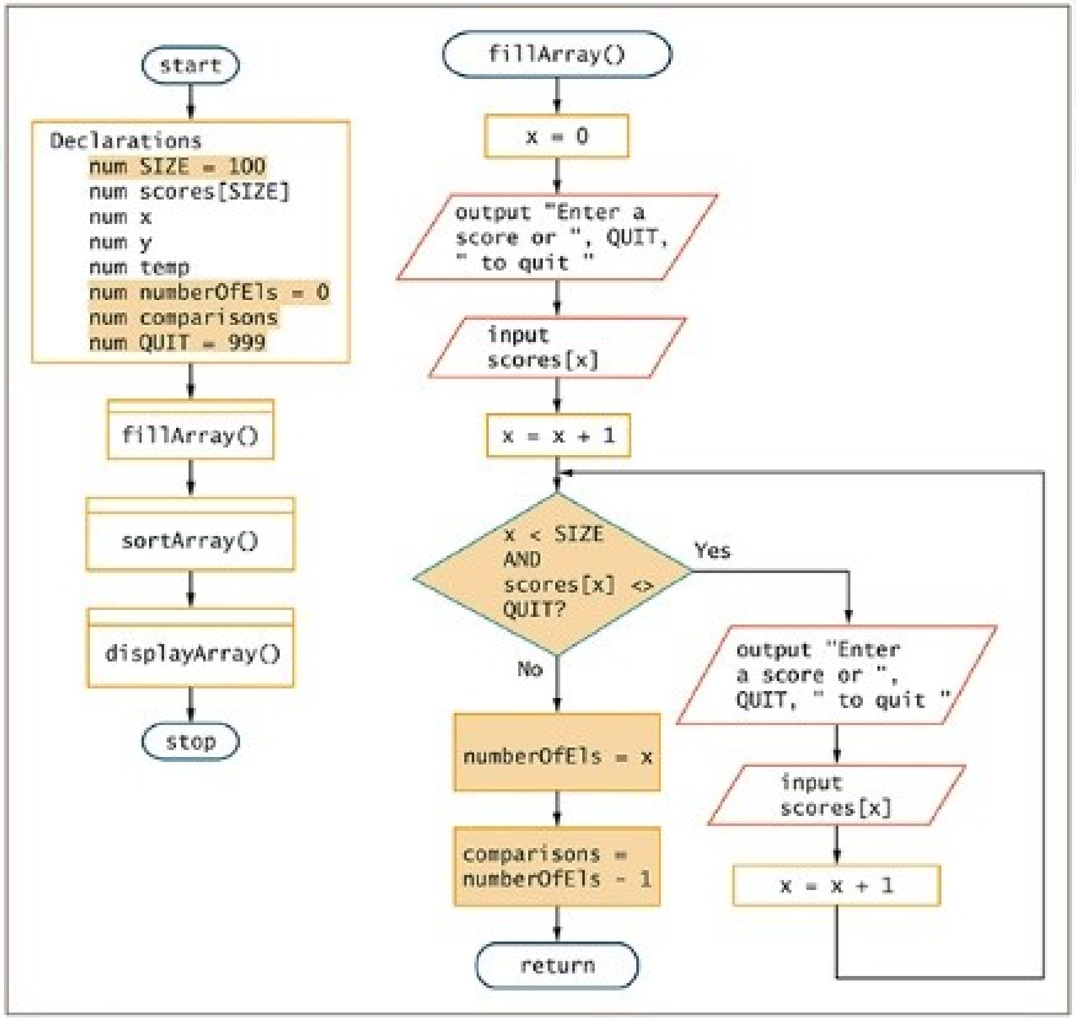


Figure 8-8 Score-sorting application in which number of elements to sort can vary *(continues)*

Sorting a List of Variable Size
(continued -2)

(continued)

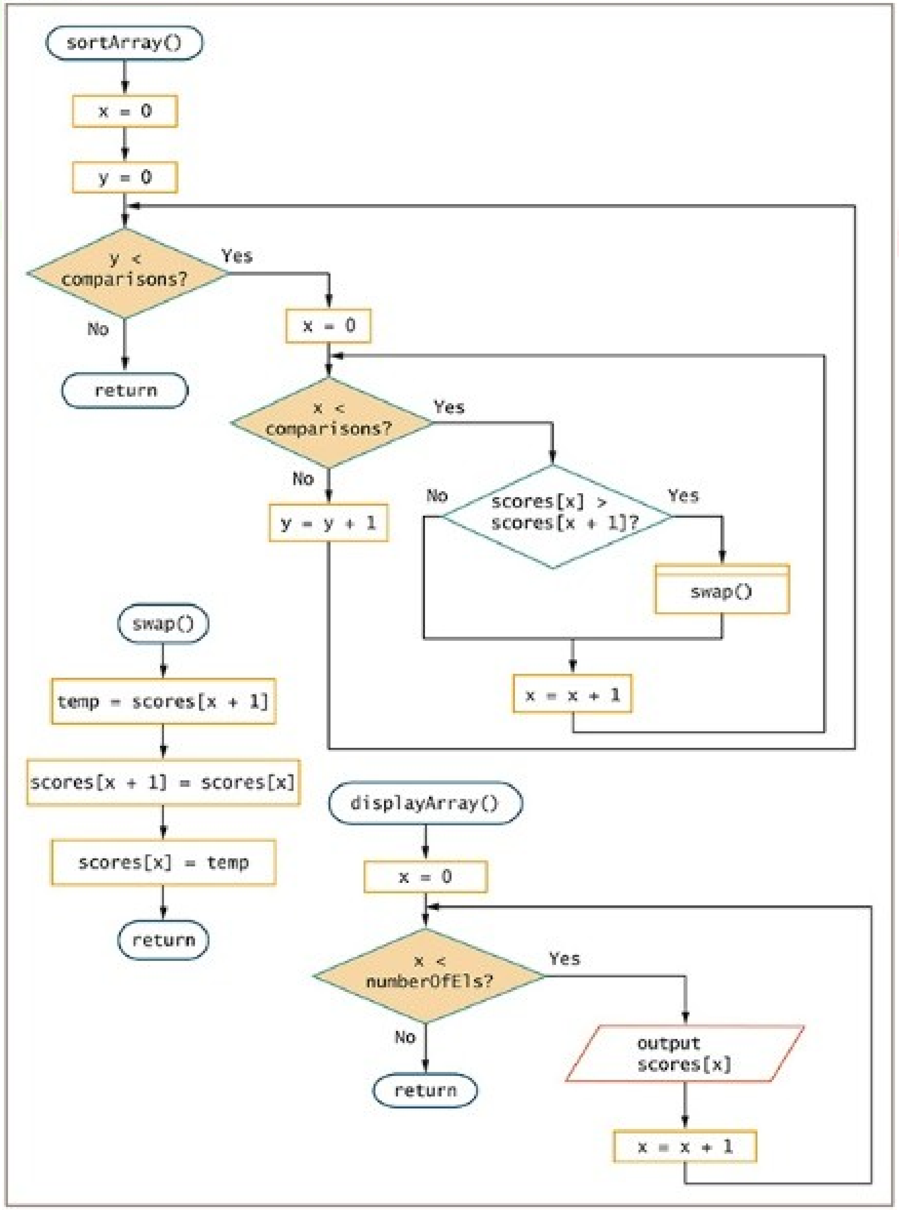


Figure 8-8 Score-sorting application in which number of elements to sort can vary (continues)

Sorting a List of Variable Size

(continued -3)

(continued)

```
start
   Declarations
      num SIZE = 100
      num scores[SIZE]
      num x
      num y
      num temp
      num numberOfEls = 0
      num comparisons
      num QUIT = 999
   fillArray()
   sortArray()
   displayArray()
stop

fillArray()
   x = 0
   output "Enter a score or ", QUIT, " to quit "
   input scores[x]
   x = x + 1
   while x < SIZE AND scores[x] <> QUIT
      output "Enter a score or ", QUIT, " to quit "
      input scores[x]
      x = x + 1
   endwhile
   numberOfEls = x
   comparisons = numberOfEls - 1
return

sortArray()
   x = 0
   y = 0
   while y < comparisons
      x = 0
      while x < comparisons
         if scores[x] > scores[x + 1] then
            swap()
         endif
         x = x + 1
      endwhile
      y = y + 1
   endwhile
return

swap()
   temp = scores[x + 1]
   scores[x + 1] = scores[x]
   scores[x] = temp
return

displayArray()
   x = 0
   while x < numberOfEls
      output scores[x]
      x = x + 1
   endwhile
return
```

Figure 8-8 Score-sorting application in which number of elements to sort can vary

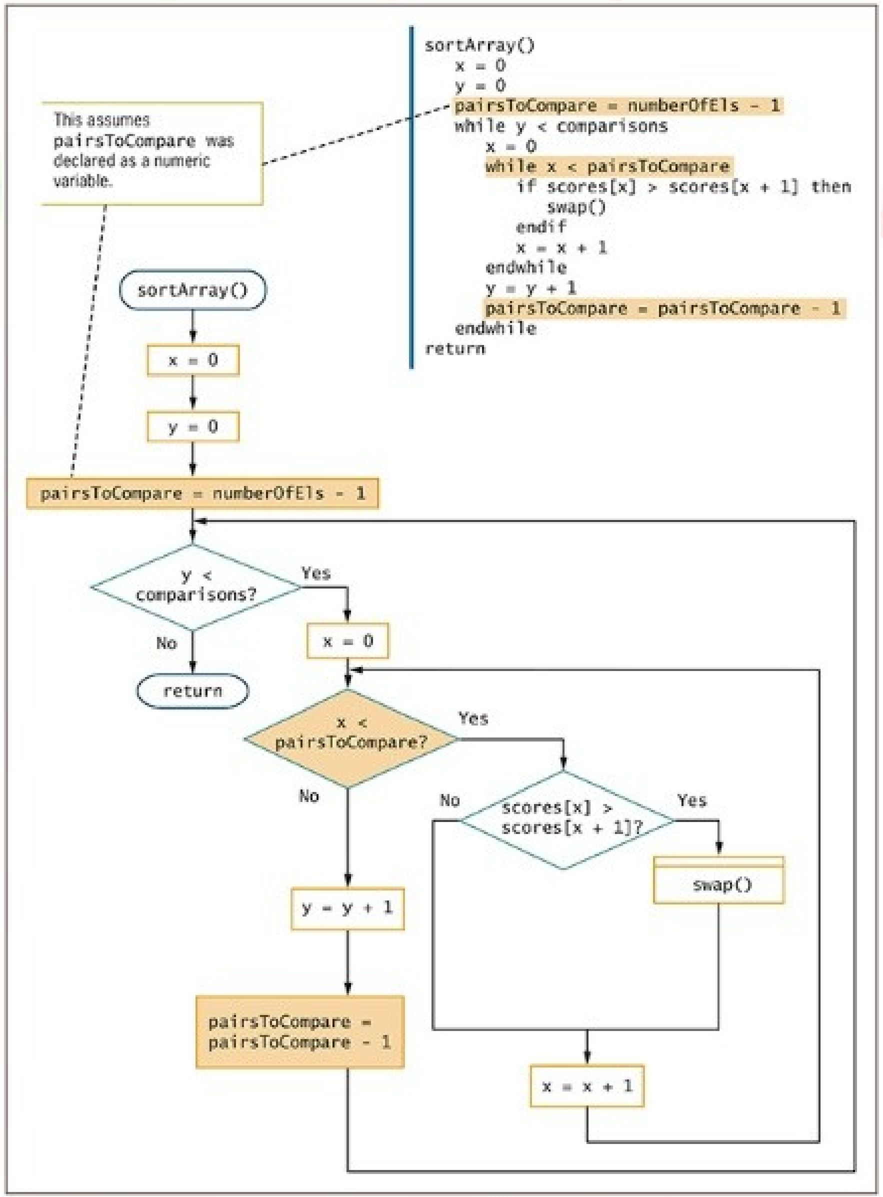
Refining the Bubble Sort to Reduce Unnecessary Comparisons

- If you are performing an ascending sort
  - After you have made one pass through the list, the largest value is guaranteed to be in its correct final position
- Compare every element pair in the list on every pass through the list
  - Comparing elements that are already guaranteed to be in their final correct position
- On each pass through the array
  - Stop your pair comparisons one element sooner

Refining the Bubble Sort to Reduce Unnecessary Comparisons (continued -1)

- Create a new variable `pairsToCompare`
  - Set it equal to the value of `numberOfEls - 1`
  - On each subsequent pass through the list, reduce `pairsToCompare` by `1`

Refining the Bubble Sort to Reduce Unnecessary Comparisons
(continued -2)




```
sortArray()
   x = 0
   y = 0
   pairsToCompare = numberOfEls - 1
   while y < comparisons
      x = 0
      while x < pairsToCompare
         if scores[x] > scores[x + 1] then
            swap()
         endif
         x = x + 1
      endwhile
      y = y + 1
      pairsToCompare = pairsToCompare - 1
   endwhile
return
```

Figure 8-9 Flowchart and pseudocode for sortArray() method using pairsToCompare variable

Refining the Bubble Sort to Reduce Unnecessary Passes

- Create a new variable `didSwap`
  - Set it equal to the value of “No”
  - Each time the `swap()` module executes, change the

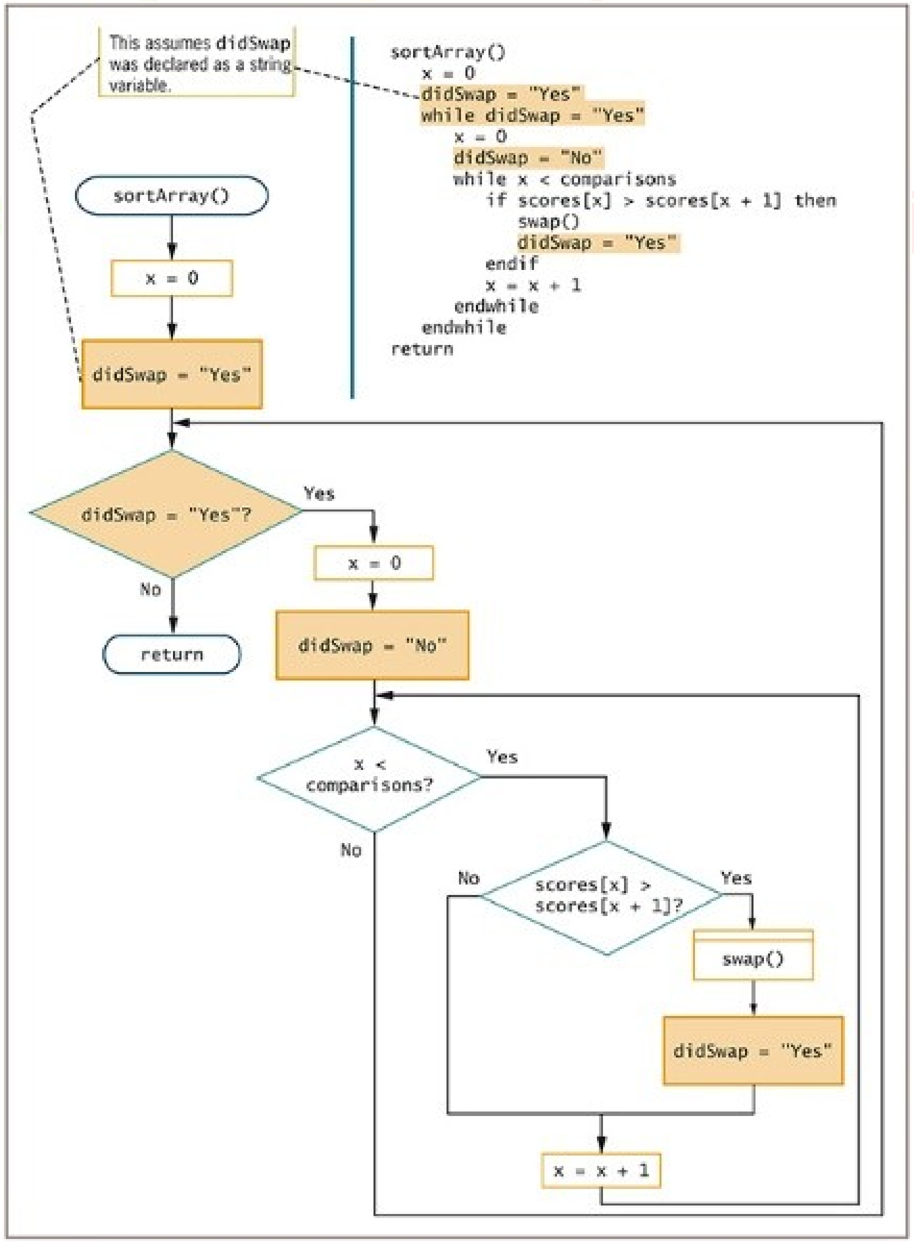


Figure 8-10 Flowchart and pseudocode for sortArray() method using didSwap variable

Sorting Multifield Records

- Sorting data stored in parallel arrays
  - Each array must stay in sync

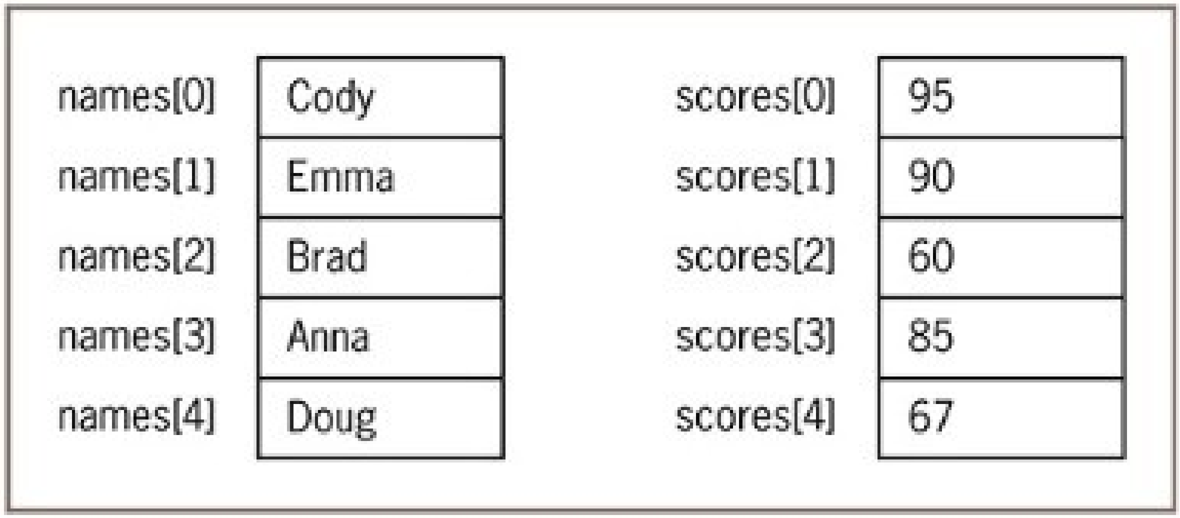
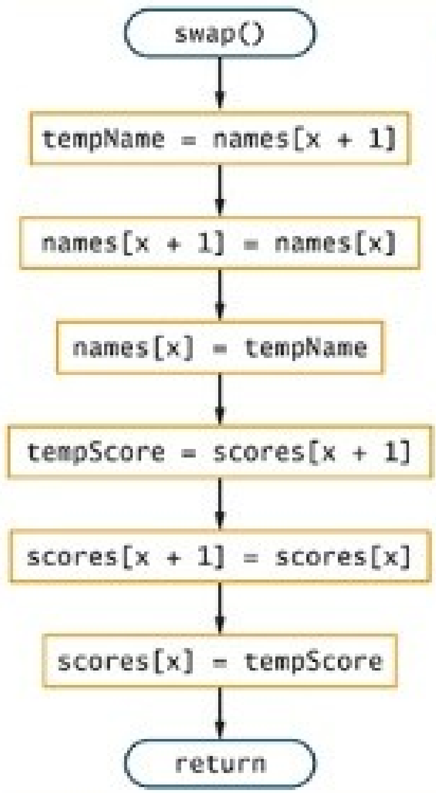


Figure 8-11 Appearance of `names` and `scores` arrays in memory

Sorting Multifield Records

(continued -1)




```
swap()
   tempName = names[x + 1]
   names[x + 1] = names[x]
   names[x] = tempName
   tempScore = scores[x + 1]
   scores[x + 1] = scores[x]
   scores[x] = tempScore
return
```

Figure 8-12 The swap() method for a program that sorts student names and retains their correct scores

Sorting Multifield Records

(continued -2)

- Sorting records as a whole
  - Create groups called record structures or classes
    - Will be covered in detail in Chapters 10 and 11

```
class StudentRecord
   string name
   num score
endClass
```

  - Defining a class allows you to sort by either name or score

Other Sorting Algorithms

- **Insertion sort**
  - Look at each element one at a time
  - If element is out of order relative to any of the items earlier in the list, move each earlier item down one slot
  - Then insert the element at the empty slot
  - Similar to the technique you would most likely use to sort a group of objects manually

Other Sorting Algorithms
(continued -1)

The insertion sort algorithm

4	3	1	5	2

Start with the second element. Compare 3 and 4. They are out of order, so swap them.

3	4	1	5	2

Compare 1 and 3. They are out of order. So save 1 in a temporary variable, move 4 and 3 down, and insert 1 where 3 was.

1	3	4	5	2

Compare 5 and 1. They are in order, so do nothing. Compare 5 and 3. They are in order, so do nothing. Compare 5 and 4. They are in order, so do nothing.

1	3	4	5	2

Compare 2 and 1. They are in order so do nothing. Compare 2 and 3, they are out of order, so save 2 in a temporary variable, move 3,4, and 5 down, and insert 2 where 3 used to be.

1	2	3	4	5

The sort is complete.

Figure 8-13 The insertion sort algorithm

Other Sorting Algorithms (continued-2)

- **Selection sort**
  - Uses two sublists containing
    - Values already sorted
    - Values not yet sorted
  - Repeatedly look for the smallest value in the unsorted sublist
  - Swap it with the item at the beginning
  - Then add that element to the end of the sorted sublist

Other Sorting Algorithms
(continued -3)

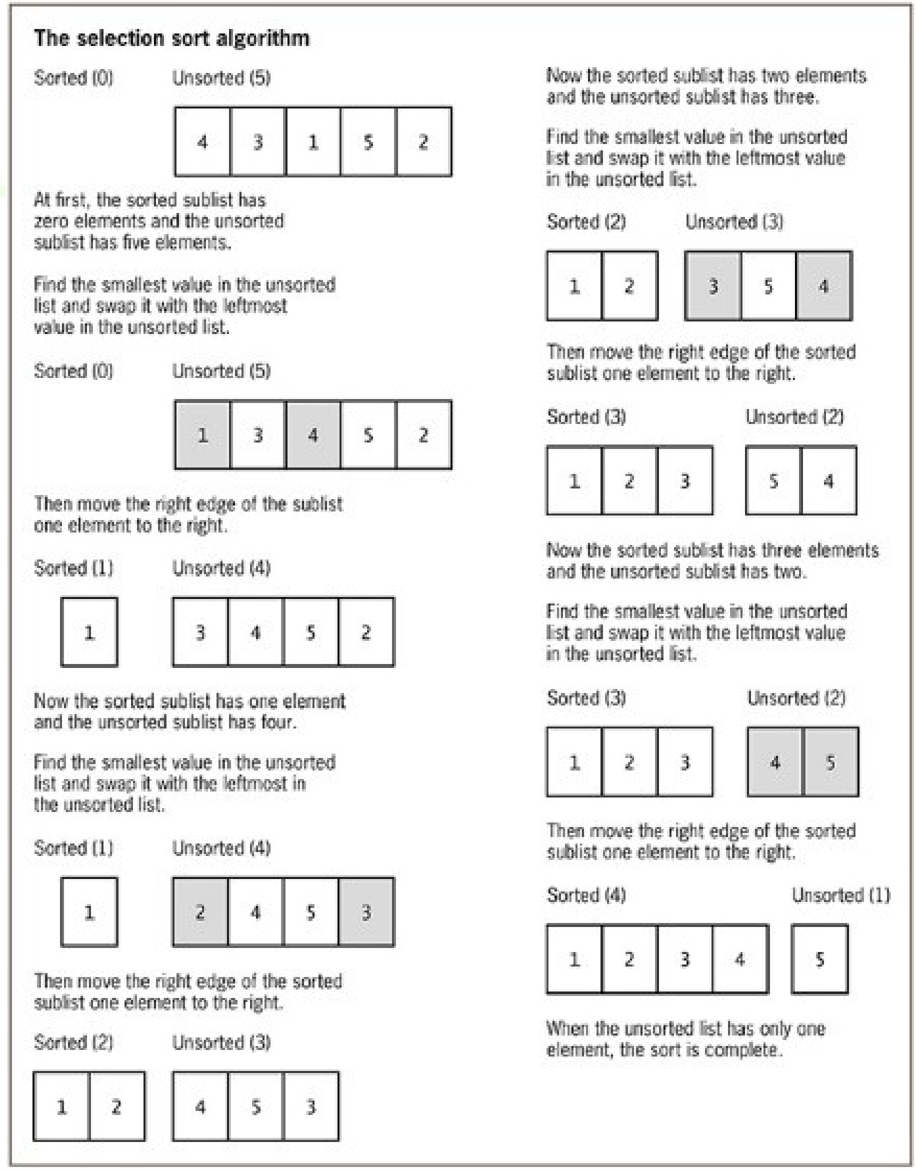


Figure 8-14 The selection sort algorithm

Using Multidimensional Arrays

- **One-dimensional array**
  - **Single-dimensional array**
  - Elements accessed using a single subscript
- Data can be stored in a table that has just one dimension
  - Height
- If you know the vertical position of a one-dimensional array's element, you can find its value
- **Multidimensional array**
  - requires more than one subscript to access an element

Using Multidimensional Arrays (continued -1)

- **Two-dimensional array**
  - Contains two dimensions: height and width
  - Location of any element depends on two factors
- Example
  - Apartment building with five floors
  - Each floor has studio apartments and one- and two-bedroom apartments
  - To determine a tenant’s rent, you need to know two pieces of information:
    - The floor
    - The number of bedrooms in the apartment

Using Multidimensional Arrays (continued -2)

- Each element in a two-dimensional array requires two subscripts to reference it
  - Row and column
- Declare a two-dimensional array
  - Type two sets of brackets after the array type and name
  - First square bracket holds the number of rows
  - Second square bracket holds the number of columns

Using Multidimensional Arrays (continued -3)

- Access a two-dimensional array value
  - First subscript represents the row
  - Second subscript represents the column
  - `RENT_BY_FLOOR_AND_BDRMS[0][0]` is 350
- Mathematicians often call a two-dimensional array a **matrix** or a **table**

Using Multidimensional Arrays (continued -4)

A Single-Dimensional Array

```
num RENTS_BY_FLR[5] =
  350, 400, 475, 600, 1000
```

350
400
475
600
1000

A Two-Dimensional Array

```
num RENTS_BY_FLR_AND_BDRMS[5][3] =
        {350, 390, 435},
        {400, 440, 480},
        {475, 530, 575},
        {600, 650, 700},
        {1000, 1075, 1150}
```

350	390	435
400	440	480
475	530	575
600	650	700
1000	1075	1150

Figure 8-15 One- and two-dimensional arrays in memory

Using Multidimensional Arrays (continued -5)

- **Three-dimensional arrays**
  - Supported by many programming languages
  - Apartment building
    - Number of floors
    - Different numbers of bedrooms available in apartments on each floor
    - Building number
- Examples
  - `RENT_BY_3_FACTORS[floor][bedrooms][building]`
  - `RENT_BY_3_FACTORS[0][1][2]`

Using Multidimensional Arrays

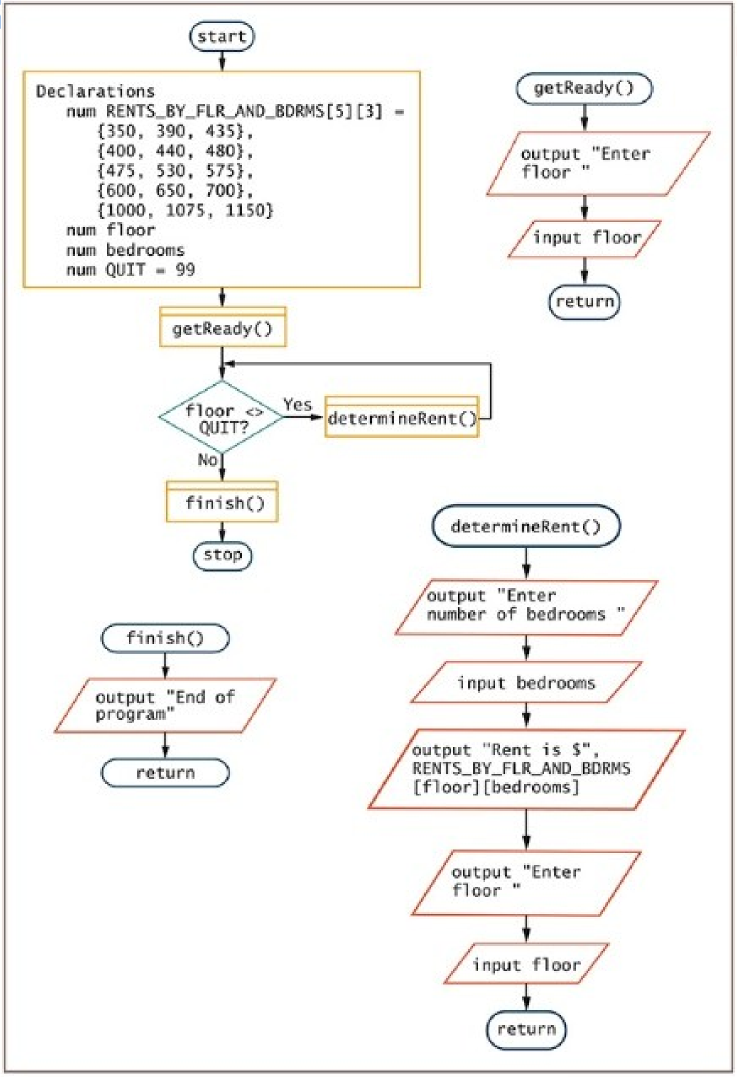


Figure 8-16 A program that determines rents (*continues*)

Using Multidimensional Arrays (continued -7)

(continued)

```
start
   Declarations
      num RENTS_BY_FLR_AND_BDRMS[5][3] = {350, 390, 435},
                                         {400, 440, 480},
                                         {475, 530, 575},
                                         {600, 650, 700},
                                         {1000, 1075, 1150}
      num floor
      num bedrooms
      num QUIT = 99
   getReady()
   while floor <> QUIT
      determineRent()
   endwhile
   finish()
stop

getReady()
   output "Enter floor "
   input floor
return

determineRent()
   output "Enter number of bedrooms "
   input bedrooms
   output "Rent is $", RENTS_BY_FLR_AND_BDRMS[floor][bedrooms]
   output "Enter floor "
   input floor
return

finish()
   output "End of program"
return
```

Figure 8-16 A program that determines rents

Using Multidimensional Arrays (continued -8)

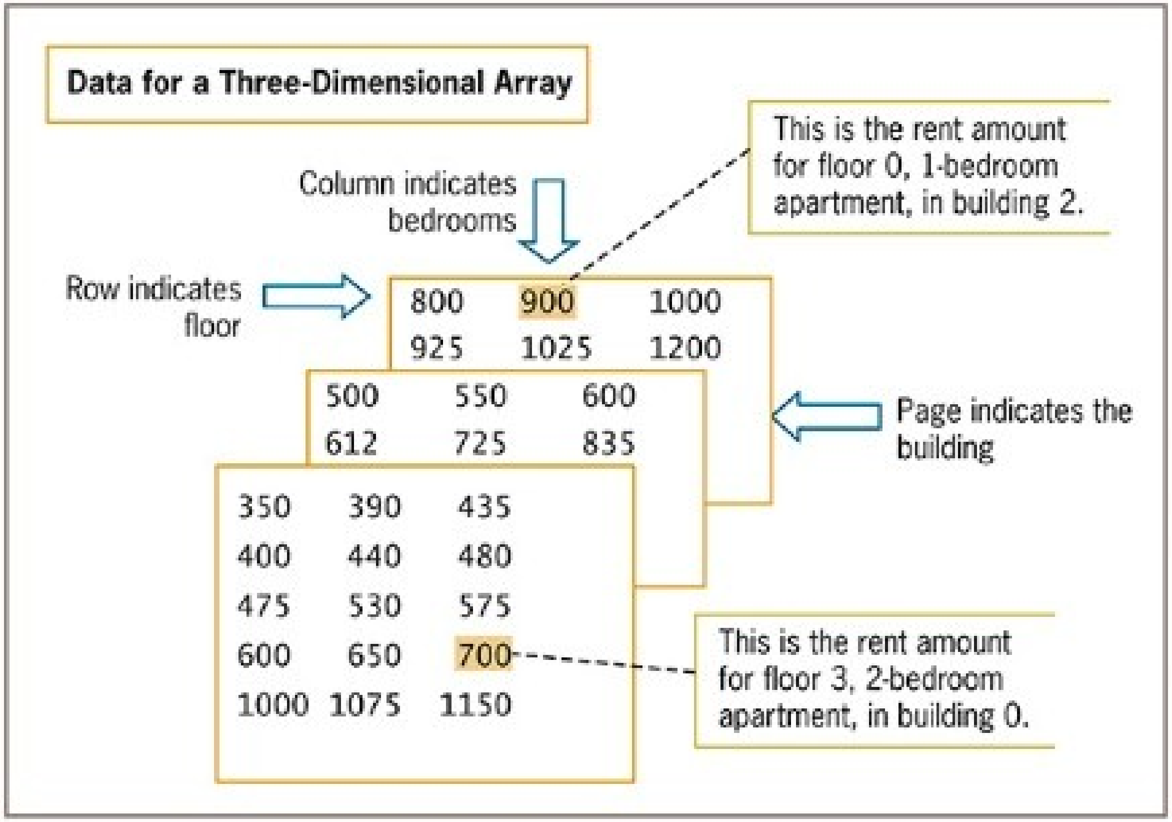


Figure 8-17 Picturing a three-dimensional array

Using Indexed Files and Linked Lists

- Sorting large numbers of data records requires considerable time and computer memory
  - Usually more efficient to store and access records based on their logical order than to sort and access them in their physical order
- **Physical order**
  - Refers to a “real” order for storage
- **Logical order**
  - Virtual order based on any criterion you choose

Using Indexed Files and Linked Lists (continued -1)

- Common method of accessing records in logical order
  - Use an index
  - Identify a key field for each record
- **Key field**
  - A field whose contents make the record unique among all records in a file
  - Example: a unique employee identification number

Using Indexed Files and Linked Lists (continued -2)

- Memory and storage locations have **addresses**
  - Every data record on a disk has a numeric address where it is stored
- **Index** records
  - To store a list of key fields paired with the storage address for the corresponding data record
- Use the index to find the records in order based on their addresses

Using Indexed Files and Linked Lists (continued -3)

- Store records on a **random-access storage device**
  - Records can be accessed in any order
- Each record can be placed in any physical location on the disk
  - Use the index as you would use the index in the back of a book
  - Picture an index based on ID numbers
- When a record is removed from an indexed file
  - It does not have to be physically removed
  - The reference can simply be deleted from the

Using Indexed Files and Linked Lists (continued -4)

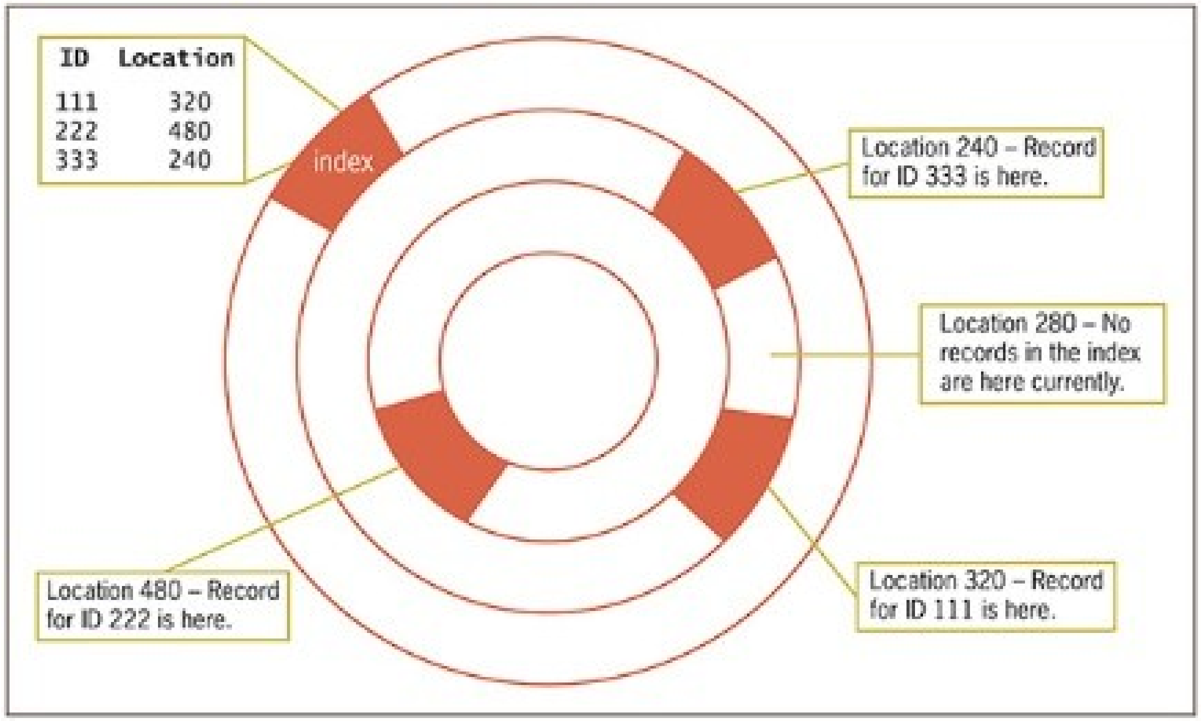


Figure 8-18 An index on a disk that associates ID numbers with disk addresses

Using Linked Lists

- **Linked list**
  - Another way to access records in a desired order, even though they might not be physically stored in that order
  - Create one extra field in every record of stored data
  - Holds the physical address of the next logical record

Address	idNum	name	phoneNum	nextCustAddress
0000	111	Baker	234-5676	7200
7200	222	Vincent	456-2345	4400

- Add a new record to a linked list

Using Linked Lists (continued)

- Remove records from a linked list
  - Records do not need to be physically deleted from the medium on which they are stored
  - Remove link field
- More sophisticated linked lists store two additional fields with each record
  - Address of the next record
  - Address of the previous record
  - List can be accessed either forward or backward

Summary

- Sort data
  - Ascending order
  - Descending order
- Bubble sort
  - Items in a list are compared with each other in pairs
  - When an item is out of order, it swaps values with the item based on which is larger or smaller
- Two possible approaches for sorting multifield records
  - Place related items in parallel arrays

Summary (continued -1)

- Insertion sort looks at each element one at a time
  - Insert tested record in place and move other records down
- Selection sort uses two sublists—values already sorted and values not yet sorted
  - Repeatedly look for the smallest value in the unsorted sublist
  - swap it with the item at the beginning
  - then add that element to the end of the sorted sublist

Summary (continued -2)

- Two-dimensional arrays
  - Use two subscripts when you access an element in a two-dimensional array
- Access data records in a logical order that differs from their physical order
  - Using an index involves identifying a physical address and key field for each record
  - Creating a linked list involves creating an extra field within every record to hold the physical address of the next logical record