elements, so in some special cases, they could number fewer than those made by the selection sort.

Insertion Sort

In most cases, the insertion sort is the best of the elementary sorts described in this chapter. It still executes in $O(N^2)$ time, but it's about twice as fast as the bubble sort and somewhat faster than the selection sort in normal situations. It's also not too complex, although it's slightly more involved than the bubble and selection sorts. It's often used as the final stage of more sophisticated sorts, such as quicksort.

Insertion Sort on the Football Players

To begin the insertion sort, start with the football players lined up in random order. (They wanted to play a game, but clearly they've got to wait until the picture can be taken.) It's easier to think about the insertion sort if you begin in the middle of the process, when part of the team is sorted.

Partial Sorting

You can use your handy ball to mark a place in the middle of the line. The players to the left of this marker are **partially sorted**. This means that they are sorted among themselves; each one is taller than the person to their left. The players, however, aren't necessarily in their final positions because they may still need to be moved when previously unsorted players are inserted between them.

Note that partial sorting did not take place in the bubble sort and selection sort. In these algorithms, a group of data items was completely sorted into their final positions at any given time; in the insertion sort, one group of items is only partially sorted.

The Marked Player

The player where the marker ball is, whom we call the "marked" player, and all the players to the right are as yet unsorted. Figure 3-8 shows the process. At the top, the players are unsorted. The next row in the figure shows the situation three steps later. The marker ball is put in front of a player who has stepped forward.

What you're going to do is insert the marked player in the appropriate place in the (partially) sorted group. To do this, you need to shift some of the sorted players to the right to make room. To provide a space for this shift, the marked player is taken out of line. (In the program, this data item is stored in a temporary variable.) This step is shown in the second row of Figure 3-8.

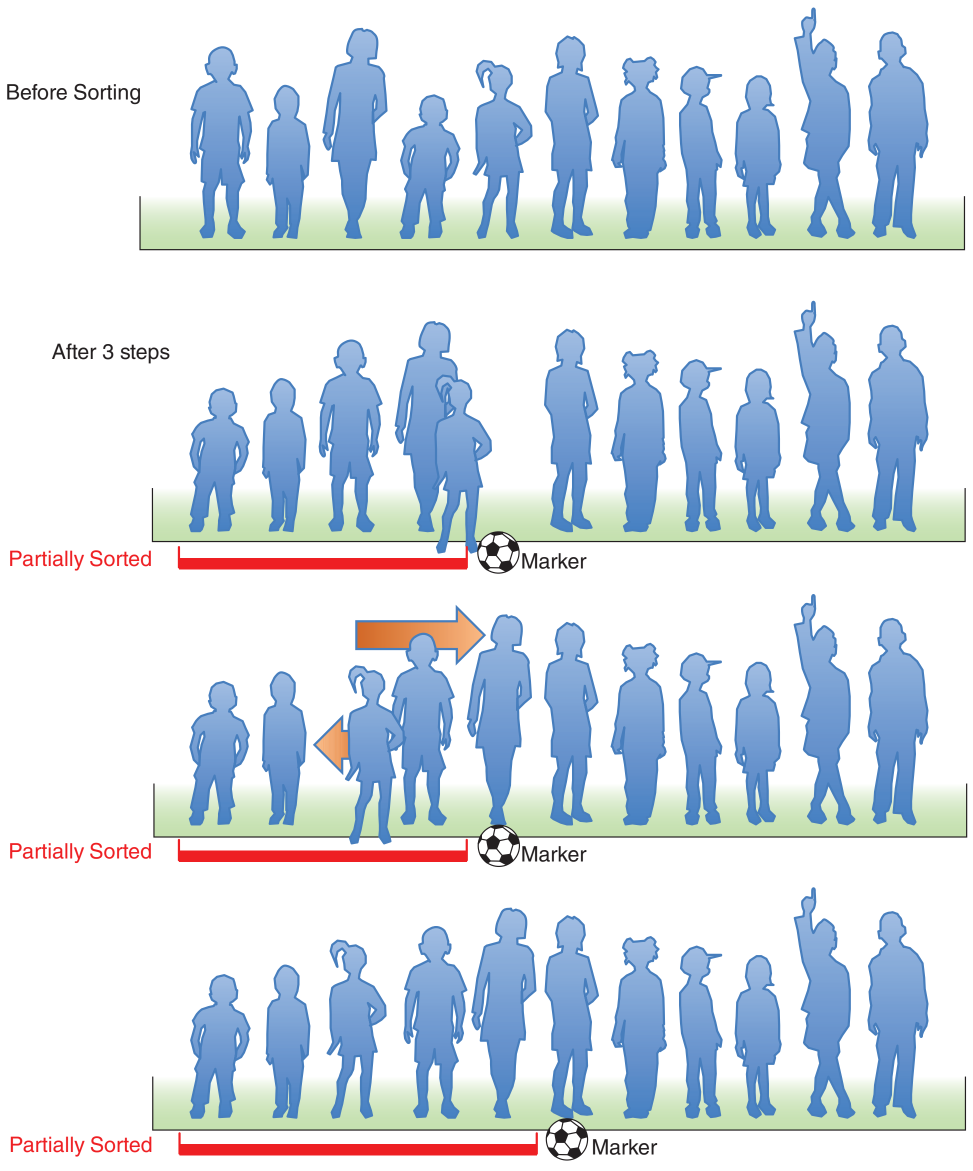

FIGURE 3-8 The insertion sort on football players

Now you shift the sorted players to make room. The tallest sorted player moves into the marked player's spot, the next-tallest player into the tallest player's spot, and so on as shown by the arrows in the third row of the figure.

When does this shifting process stop? Imagine that you and the marked player are walking down the line to the left. At each position you shift another player to the right, but you also compare the marked player with the player about to be shifted. The shifting process stops when you've shifted the last player that's taller than the marked player. The last shift opens up the space where the marked player, when inserted, will be in sorted order. This step is shown in the bottom row of Figure 3-8.

Now the partially sorted group is one player bigger, and the unsorted group is one player smaller. The marker ball is moved one space to the right, so it's again in front of the leftmost unsorted player. This process is repeated until all the unsorted players have been inserted (hence the name *insertion* sort) into the appropriate place in the partially sorted group.

The Insertion Sort in the SimpleSorting Visualization Tool

Return to the SimpleSorting Visualization tool and create another 11-cell array of random values. Then select the Insertion Sort button to start the sorting process. As in the other sort operations, the algorithm puts up two arrows: one for the outer and one for the inner loops. The arrow labeled "outer" is the equivalent of the marker ball in sorting the players.

In the example shown in Figure 3-9, the outer arrow is pointing at the fifth cell in the array. It already copied item 61 to the `temp` variable below the array. It has also copied item 94 into the fifth cell and is starting to copy item 85 into the fourth. The four cells to the left of the outer arrow are partially sorted; the cells to its right are unsorted. Even while it decides where the marked player should go and there are extra copies of one of them, the left-hand cells remain partially sorted. This matches what happens in the computer memory when you copy the array item to a temporary variable.

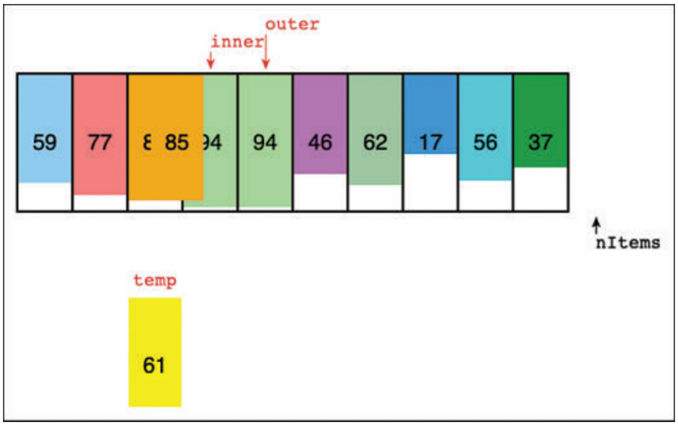

FIGURE 3-9 In the middle of an insertion sort in the SimpleSorting Visualization tool

The inner arrow starts at the same cell as the outer arrow and moves to the left, shifting items taller than the marked temp item to the right. In Figure 3-9, item 85 is being copied to where inner points because it is larger than item 61. After that copy is completed, inner moves one cell to the left and finds that item 77 needs to be moved too. On the next step for inner, it finds item 59 to its left. That item is smaller than 61, and the marked item is copied from temp back to the inner cell, where item 77 used to be.

Try creating a larger array filled with random values. Run the insertion sort and watch the updates to the arrows and items. Everything to the left of the outer arrow remains sorted by height. The inner arrow moves to the left, shifting items until it finds the right cell to place the item that was marked (copied from the outer cell).

Eventually, the outer arrow arrives at the right end of the array. The inner arrow moves to the left from that right edge and stops wherever it is appropriate to insert the last item. It might finish anywhere in the array. That's a little different than the bubble and selection sorts where the outer and inner loop variables finish together.

Notice that, occasionally, the item copied from outer to temp is copied right back to where it started. That occurs when outer happens to point at an item larger than every item to its left. A similar thing happens in the selection sort when the outer arrow points at an item smaller than every item to its right; it ends up doing a swap with itself. Doing these extra data moves might seem inefficient, but adding tests for these conditions could add its own inefficiencies. An experiment at the end of the chapter asks you to explore when it might make sense to do so.

Python Code for Insertion Sort

Let's now look at the `insertionSort()` method for the `Array` class as shown in Listing 3-3. You still need two loops, but the inner loop hunts for the insertion point using a `while` loop this time.

LISTING 3-3 The `insertionSort()` method of the `Array` class

```
def insertionSort(self):                      # Sort by repeated inserts
   for outer in range(1, self.__nItems):      # Mark one element
      temp = self.__a[outer]                  # Store marked elem in temp
      inner = outer                           # Inner loop starts at mark
      while inner > 0 and temp < self.__a[inner-1]: # If marked
         self.__a[inner] = self.__a[inner-1] # elem smaller, then
         inner -= 1                           # shift elem to right
      self.__a[inner] = temp                  # Move marked elem to 'hole'
```

In the outer `for` loop, `outer` starts at 1 and moves right. It marks the leftmost unsorted array element. In the inner `while` loop, `inner` starts at `outer` and moves left to `inner-1`, until either `temp` is smaller than the array element there, or it can't go left any further. Each pass through the `while` loop shifts another sorted item one space right.

Invariants in the Insertion Sort

The data items with smaller indices than `outer` are partially sorted. This is true even at the beginning when `outer` is 1 because the single item in cell 0 forms a one-element sequence that is always sorted. As items shift, there are multiple copies of one of them, but they all remain in partially sorted order.

Efficiency of the Insertion Sort

How many comparisons and copies does this algorithm require? On the first pass, it compares a maximum of one item. On the second pass, it's a maximum of two items, and so on, up to a maximum of N–1 comparisons on the last pass. (We ignore the check for `inner > 0` and count only the intra-element comparisons.) This adds up to

$$1 + 2 + 3 + \ldots + N-1 = N\times(N-1)/2$$

On each pass, however, the number of items actually compared before the insertion point is found is, on average, half of the partially sorted items, so you can divide by 2, which gives

$$N\times(N-1)/4$$

The number of copies is approximately the same as the number of comparisons because it makes one copy for each comparison but the last. Copying one item is less time-consuming than a swap of two items, so for random data this algorithm runs twice as fast as the bubble sort and faster than the selection sort, on average.

In any case, like the other sort routines in this chapter, the insertion sort runs in $O(N^2)$ time for random data.

For data that is already sorted or almost sorted, the insertion sort does much better. When data is in order, the condition in the `while` loop is never true, so it becomes a simple statement in the outer loop, which executes N–1 times. In this case, the algorithm runs in O(N) time. If the data is almost sorted, the insertion sort runs in almost O(N) time, which makes it a simple and efficient way to order an array that is only slightly out of order.

For data arranged in inverse sorted order, however, every possible comparison and shift is carried out, so the insertion sort runs no faster than the bubble sort. Try some experiments with the SimpleSorting Visualization tool. Sort an 8+ element array with the selection sort and then try sorting the result with the selection sort again. Even though the second attempt doesn't make any "real" swaps, it takes the same number of steps to go through the process. Then shuffle the array and sort it twice using the insertion sort. The second attempt at the insertion sort is significantly shorter because the inner loop ends after one comparison.

Python Code for Sorting Arrays

Now let's combine the three sorting algorithms into a single object class to compare them. Starting from the `Array` class introduced in Chapter 2, we add the new methods for swapping and sorting plus the `__str__()` method for displaying the contents of arrays as a string and put them in a module called `SortArray.py`, as shown in Listing 3-4.

LISTING 3-4 The `SortArray.py` module

```
# Implement a sortable Array data structure

class Array(object):
   def __init__(self, initialSize):    # Constructor
      self.__a = [None] * initialSize  # The array stored as a list
      self.__nItems = 0                # No items in array initially

   def __len__(self):                  # Special def for len() func
      return self.__nItems             # Return number of items

   def get(self, n):                   # Return the value at index n
      if 0 <= n and n < self.__nItems: # Check if n is in bounds, and
         return self.__a[n]            # only return item if in bounds

   def set(self, n, value):            # Set the value at index n
      if 0 <= n and n < self.__nItems: # Check if n is in bounds, and
         self.__a[n] = value           # only set item if in bounds

   def swap(self, j, k):               # Swap the values at 2 indices
      if (0 <= j and j < self.__nItems and # Check if indices are in
          0 <= k and k < self.__nItems): # bounds, before processing
         self.__a[j], self.__a[k] = self.__a[k], self.__a[j]

   def insert(self, item):             # Insert item at end
      if self.__nItems >= len(self.__a):     # If array is full,
          raise Exception("Array overflow")      # raise exception
      self.__a[self.__nItems] = item   # Item goes at current end
      self.__nItems += 1               # Increment number of items

   def find(self, item):               # Find index for item
      for j in range(self.__nItems):   # Among current items
         if self.__a[j] == item:       # If found,
            return j                   # then return index to element
      return -1                        # Not found -> return -1

   def search(self, item):             # Search for item
      return self.get(self.find(item)) # and return item if found

   def delete(self, item):             # Delete first occurrence
      for j in range(self.__nItems):   # of an item
         if self.__a[j] == item:       # Found item
```

```
            self.__nItems -= 1         # One fewer at end
            for k in range(j, self.__nItems):  # Move items from
               self.__a[k] = self.__a[k+1]     # right over 1
            return True                # Return success flag

      return False     # Made it here, so couldn't find the item

   def traverse(self, function=print): # Traverse all items
      for j in range(self.__nItems):   # and apply a function
         function(self.__a[j])

   def __str__(self):                  # Special def for str() func
      ans = "["                        # Surround with square brackets
      for i in range(self.__nItems):   # Loop through items
         if len(ans) > 1:              # Except next to left bracket,
            ans += ", "                # separate items with comma
         ans += str(self.__a[i])       # Add string form of item
      ans += "]"                       # Close with right bracket
      return ans

   def bubbleSort(self):               # Sort comparing adjacent vals
      for last in range(self.__nItems-1, 0, -1):  # and bubble up
         for inner in range(last):     # inner loop goes up to last
            if self.__a[inner] > self.__a[inner+1]:  # If elem less
               self.swap(inner, inner+1) # than adjacent value, swap

   def selectionSort(self):            # Sort by selecting min and
      for outer in range(self.__nItems-1): # swapping min to leftmost
         min = outer                   # Assume min is leftmost
         for inner in range(outer+1, self.__nItems): # Hunt to right
            if self.__a[inner] < self.__a[min]: # If we find new min,
               min = inner             # update the min index

         # __a[min] is smallest among __a[outer]...__a[__nItems-1]
         self.swap(outer, min)         # Swap leftmost and min

   def insertionSort(self):            # Sort by repeated inserts
      for outer in range(1, self.__nItems): # Mark one element
         temp = self.__a[outer]        # Store marked elem in temp
         inner = outer                 # Inner loop starts at mark
         while inner > 0 and temp < self.__a[inner-1]: # If marked
            self.__a[inner] = self.__a[inner-1] # elem smaller, then
            inner -= 1                 # shift elem to right
         self.__a[inner] = temp        # Move marked elem to 'hole'
```

The `swap()` method swaps the values in two cells of the array. It ensures that swaps happen only with items that have been inserted in the array and not with allocated but uninitialized cells. Those tests may not be necessary with the sorting methods in this module that already ensure proper indices, but they are a good idea for a general-purpose routine.

We use a separate client program, `SortArrayClient.py`, to test this new module and compare the performance of the different sorting methods. This program uses some other Python modules and features to help in this process and is shown in Listing 3-5.

LISTING 3-5 The `SortArrayClient.py` program

```
from SortArray import *
import random
import timeit

def initArray(size=100, maxValue=100, seed=3.14159):
    """Create an Array of the specified size with a fixed sequence of
       'random' elements"""
    arr = Array(size)                           # Create the Array object
    random.seed(seed)                           # Set random number generator
    for i in range(size):                       # to known state, then loop
        arr.insert(random.randrange(maxValue)) # Insert random numbers
    return arr                                  # Return the filled Array

arr = initArray()
print("Array containing", len(arr), "items:\n", arr)

for test in ['initArray().bubbleSort()',
             'initArray().selectionSort()',
             'initArray().insertionSort()']:
    elapsed = timeit.timeit(test, number=100, globals=globals())
    print(test, "took", elapsed, "seconds", flush=True)

arr.insertionSort()
print('Sorted array contains:\n', arr)
```

The `SortArrayClient.py` program starts by importing the `SortArray` module to test the sorting methods, and the `random` and `timeit` modules to assist. The `random` module provides pseudo-random number generators that can be used to make test data. The `timeit` module provides a convenient way to measure the execution times of Python code.

The test program first defines a function, `initArray()`, that generates an unsorted array for testing. Right after the parameter list, it has a documentation string that helps explain its purpose, creating arrays of a fixed size filled with a sequence of random numbers. Because we'd like to test the different sort operations on the exact same sequence of elements, we want `initArray()` to return a fresh copy of the same array every time it runs, but conforming to the parameters it is given. The function first creates an `Array` of the desired `size`. Next, it initializes the `random` module to a known state by setting the `seed` value. This means that pseudo-random functions produce the same sequence of numbers when called in the same order.

Inside the following `for` loop, the `random.randrange(maxValue)` function generates integers in the range [0, `maxValue`) that are inserted into the array (the math notation *[a, b)* represents a range of numbers that includes *a* but excludes *b*). Then the filled array is returned.

The test program uses the `initArray()` function to generate an array that is stored in the `arr` variable. After printing the initial contents of `arr`, a `for` loop is used to step through the three sort operations to be timed. The loop variable, `test`, is bound to a string on each pass of the loop. The string has the Python expression whose execution time is to be measured. Inside the loop body, the `timeit.timeit()` function call measures the elapsed time of running the `test` expression. It runs the expression a designated `number` of times and returns the total elapsed time in seconds. Running it many times helps deal with the variations in running times due to other operations happening on the computer executing the Python interpreter. The last argument to `timeit.timeit()` is `globals`, which is needed so that the interpreter has all the definitions that have been loaded so far, including `SortArray` and `initArray`. The interpreter uses those definitions when evaluating the Python expression in the `test` variable.

The result of `timeit.timeit()` is stored in the `elapsed` variable and then printed out with `flush=True` so that it doesn't wait until the output buffer is full or input needs to be read before printing. After all three test times are printed, it performs one more sort on the original `arr` array and prints its contents. The results look like this:

```
$ python3 SortArrayClient.py
Array containing 100 items:
 [77, 94, 59, 85, 61, 46, 62, 17, 56, 37, 18, 45, 76, 21, 91, 7, 96, 50, 31, 69, 80,
69, 56, 60, 26, 25, 1, 2, 67, 46, 99, 57, 32, 26, 98, 51, 77, 34, 20, 81, 22, 40,
28, 23, 69, 39, 23, 6, 46, 1, 96, 51, 71, 61, 2, 34, 1, 55, 78, 91, 69, 23, 2, 8,
3, 78, 31, 25, 26, 73, 28, 88, 88, 38, 22, 97, 9, 18, 18, 66, 47, 16, 82, 9, 56, 45,
15, 76, 85, 52, 86, 5, 28, 67, 34, 20, 6, 33, 83, 68]
initArray().bubbleSort() took 0.25465869531035423 seconds
initArray().selectionSort() took 0.11350362841039896 seconds
initArray().insertionSort() took 0.12348053976893425 seconds
Sorted array contains:
 [1, 1, 1, 2, 2, 2, 3, 5, 6, 6, 7, 8, 9, 9, 15, 16, 17, 18, 18, 18, 20, 20, 21, 22,
22, 23, 23, 23, 25, 25, 26, 26, 26, 28, 28, 28, 31, 31, 32, 33, 34, 34, 34, 37, 38,
39, 40, 45, 45, 46, 46, 46, 47, 50, 51, 51, 52, 55, 56, 56, 56, 57, 59, 60, 61, 61,
62, 66, 67, 67, 68, 69, 69, 69, 69, 71, 73, 76, 76, 77, 77, 78, 78, 80, 81, 82, 83,
85, 85, 86, 88, 88, 91, 91, 94, 96, 96, 97, 98, 99]
```

Looking at the output, you can see that the `randrange()` function provided a broad variety of values for the array in a random order. The results of the timing of the sort tests show the bubble sort taking at least twice as much time as the selection and insertion sorts do. The final sorted version of the array confirms that sorting works and shows that there are many duplicate elements in the array.

Stability

Sometimes it matters what happens to data items that have equal keys when sorting. The `SortArrayClient.py` test stored only integers in the array, and equal integers are pretty much indistinguishable. If the array contained complex records, however, it could be very important how records with the same key are sorted. For example, you may have employee records arranged alphabetically by family names. (That is, the family names were used as key values in the sort.) Let's say you want to sort the data by postal code too, but you want all the items with the same postal code to continue to be sorted by family names. This is called a **secondary sort key**. You want the sorting algorithm to shift only what needs to be sorted by the current key and leave everything else in its order after previous sorts using other keys. Some sorting algorithms retain this secondary ordering; they're said to be **stable**. Stable sorting methods also minimize the number of swaps or copy operations.

Some of the algorithms in this chapter are stable. We have included an exercise at the end of the chapter for you to decide which ones are. The answer is not obvious from the output of their test programs, especially on simple integers. You need to review the algorithms to see that swaps and copies only happen on items that need to be moved to be in the right position for the final order. They should also move items with equal keys the minimum number of array cells necessary so that they remain in the same relative order in the final arrangement.

Comparing the Simple Sorts

There's probably no point in using the bubble sort, unless you don't have your algorithm book handy. The bubble sort is so simple that you can write it from memory. Even so, it's practical only if the amount of data is small. (For a discussion of what "small" means and what trade-offs such decisions entail, see Chapter 16, "What to Use and Why.")

Table 3-1 summarizes the time efficiency of each of the sorting methods. We looked at what the algorithms would do when presented with elements whose keys are in random order to determine what they would do in the average case. We also considered what would happen in the worst case when the keys were in some order that would maximize the number of operations. The table includes both the Big O notation followed by the detailed value in square brackets [ ]. The Big O notation is what matters most; it gives the broad classification of the algorithms and tells what will happen for very large N. Typically, we are most interested in the average case behavior because we cannot anticipate what data they will encounter, but sometimes the worst case is the driving factor. For