CHAPTER 5

Searching and Sorting

When people collect and work with data, they eventually want to search for specific items within the collection or sort the collection for presentation or easy access. Searching and sorting are two of the most common applications found in computer science. In this chapter, we explore these important topics and study some of the basic algorithms for use with sequence structures. The searching problem will be discussed many times throughout the text as it can be applied to collections stored using many different data structures, not just sequences. We will also further explore the sorting problem in Chapters 12 and 13 with a discussion of more advanced sorting algorithms.

5.1 Searching

Searching is the process of selecting particular information from a collection of data based on specific criteria. You are familiar with this concept from your experience in performing web searches to locate pages containing certain words or phrases or when looking up a phone number in the telephone book. In this text, we restrict the term searching to refer to the process of finding a specific item in a collection of data items.

The search operation can be performed on many different data structures. The ***sequence search***, which is the focus in this chapter, involves finding an item within a sequence using a ***search key*** to identify the specific item. A ***key*** is a unique value used to identify the data elements of a collection. In collections containing simple types such as integers or reals, the values themselves are the keys. For collections of complex types, a specific data component has to be identified as the key. In some instances, a key may consist of multiple components, which is also known as a ***compound key***.

5.1.1 The Linear Search

The simplest solution to the sequence search problem is the sequential or ***linear search*** algorithm. This technique iterates over the sequence, one item at a time, until the specific item is found or all items have been examined. In Python, a target item can be found in a sequence using the `in` operator:

```
if key in theArray :
   print( "The key is in the array." )
else :
   print( "The key is not in the array." )
```

The use of the `in` operator makes our code simple and easy to read but it hides the inner workings. Underneath, the `in` operator is implemented as a linear search. Consider the unsorted 1-D array of integer values shown in Figure 5.1(a). To determine if value 31 is in the array, the search begins with the value in the first element. Since the first element does not contain the target value, the next element in sequential order is compared to value 31. This process is repeated until the item is found in the sixth position. What if the item is not in the array? For example, suppose we want to search for value 8 in the sample array. The search begins at the first entry as before, but this time every item in the array is compared to the target value. It cannot be determined that the value is not in the sequence until the entire array has been traversed, as illustrated in Figure 5.1(b).

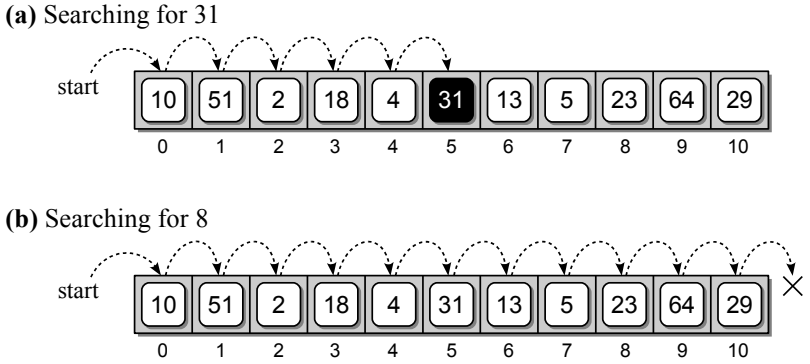


Figure 5.1: Performing a linear search on an unsorted array: (a) the target item is found and (b) the item is not in the array.

Finding a Specific Item

The function in Listing 5.1 implements the sequential search algorithm, which results in a boolean value indicating success or failure of the search. This is the same operation performed by the Python `in` operator. A count-controlled loop is used to traverse through the sequence during which each element is compared against the target value. If the item is in the sequence, the loop is terminated and `True` is returned. Otherwise, a full traversal is performed and `False` is returned after the loop terminates.

Listing 5.1 Implementation of the linear search on an unsorted sequence.

```
1  def linearSearch( theValues, target ) :
2    n = len( theValues )
3    for i in range( n ) :
4       # If the target is in the ith element, return True
5      if theValues[i] == target
6        return True
7
8    return False   # If not found, return False.
```

To analyze the sequential search algorithm for the worst case, we must first determine what conditions constitute the worst case. Remember, the worst case occurs when the algorithm performs the maximum number of steps. For a sequential search, that occurs when the target item is not in the sequence and the loop iterates over the entire sequence. Assuming the sequence contains n items, the linear search has a worst case time of $O(n)$.

Searching a Sorted Sequence

A linear search can also be performed on a sorted sequence, which is a sequence containing values in a specific order. For example, the values in the array illustrated in Figure 5.2 are in ascending or increasing numerical order. That is, each value in the array is larger than its predecessor.

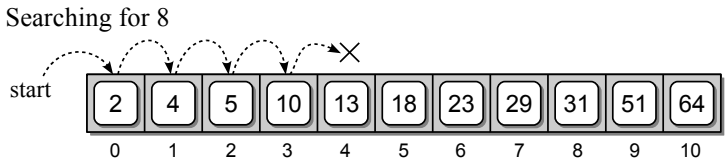


Figure 5.2: The linear search on a sorted array.

A linear search on a sorted sequence works in the same fashion as that for the unsorted sequence, with one exception. It's possible to terminate the search early when the value is not in the sequence instead of always having to perform a complete traversal. For example, suppose we want to search for 8 in the array from Figure 5.2. When the fourth item, which is value 10, is examined, we know value 8 cannot be in the sorted sequence or it would come before 10. The implementation of a linear search on a sorted sequence is shown in Listing 5.2 on the next page.

The only modification to the earlier version is the inclusion of a test to determine if the current item within the sequence is larger than the target value. If a larger value is encountered, the loop terminates and `False` is returned. With the modification to the linear search algorithm, we have produced a better version, but the time-complexity remains the same. The reason is that the worst case occurs when the value is not in the sequence and is larger than the last element. In this case, we must still traverse the entire sequence of n items.

Listing 5.2 Implementation of the linear search on a sorted sequence.

```
1  def sortedLinearSearch( theValues, item ) :
2    n = len( theValues )
3    for i in range( n ) :
4       # If the target is found in the ith element, return True
5      if theValues[i] == item :
6        return True
7       # If target is larger than the ith element, it's not in the sequence.
8      elif theValues[i] > item :
9        return False
10
11   return False   # The item is not in the sequence.
```

Finding the Smallest Value

Instead of searching for a specific value in an unsorted sequence, suppose we wanted to search for the smallest value, which is equivalent to applying Python's `min()` function to the sequence. A linear search is performed as before, but this time we must keep track of the smallest value found for each iteration through the loop, as illustrated in Listing 5.3.

To prime the loop, we assume the first value in the sequence is the smallest and start the comparisons at the second item. Since the smallest value can occur anywhere in the sequence, we must always perform a complete traversal, resulting in a worst case time of $O(n)$.

Listing 5.3 Searching for the smallest value in an unsorted sequence.

```
1  def findSmallest( theValues ):
2    n = len( theValues )
3     # Assume the first item is the smallest value.
4    smallest = theValues[0]
5     # Determine if any other item in the sequence is smaller.
6    for i in range( 1, n ) :
7      if theList[i] < smallest :
8        smallest = theValues[i]
9
10   return smallest       # Return the smallest found.
```

5.1.2 The Binary Search

The linear search algorithm for a sorted sequence produced a slight improvement over the linear search with an unsorted sequence, but both have a linear time-complexity in the worst case. To improve the search time for a sorted sequence, we can modify the search technique itself.

Consider an example where you are given a stack of exams, which are in alphabetical order, and are asked to find the exam for "Jessica Roberts." In performing

this task, most people would not begin with the first exam and flip through one at a time until the requested exam is found, as would be done with a linear search. Instead, you would probably flip to the middle and determine if the requested exam comes alphabetically before or after that one. Assuming Jessica's paper follows alphabetically after the middle one, you know it cannot possibly be in the top half of the stack. Instead, you would probably continue searching in a similar fashion by splitting the remaining stack of exams in half to determine which portion contains Jessica's exam. This is an example of a ***divide and conquer*** strategy, which entails dividing a larger problem into smaller parts and conquering the smaller part.

Algorithm Description

The ***binary search*** algorithm works in a similar fashion to the process described above and can be applied to a sorted sequence. The algorithm starts by examining the middle item of the sorted sequence, resulting in one of three possible conditions: the middle item is the target value, the target value is less than the middle item, or the target is larger than the middle item. Since the sequence is ordered, we can eliminate half the values in the list when the target value is not found at the middle position.

Consider the task of searching for value 10 in the sorted array from Figure 5.2. We first determine which element contains the middle entry. As illustrated in Figure 5.3, the middle entry contains 18, which is greater than our target of 10. Thus, we can discard the upper half of the array from consideration since 10 cannot possibly be in that part. Having eliminated the upper half, we repeat the process on the lower half of the array. We then find the middle item of the lower half and compare its value to the target. Since that entry, which contains 5, is less than the target, we can eliminate the lower fourth of the array. The process is repeated on the remaining items. Upon finding value 10 in the middle entry from among those remaining, the process terminates successfully. If we had not found the target, the process would continue until either the target value was found or we had eliminated all values from consideration.

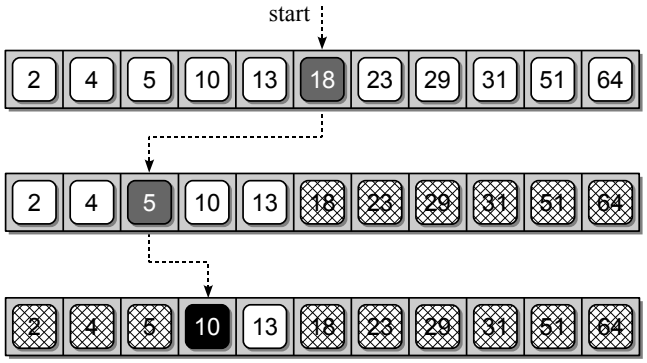


Figure 5.3: Searching for 10 in a sorted array using the binary search.

Implementation

The Python implementation of the binary search algorithm is provided in Listing 5.4. The variables `low` and `high` are used to mark the range of elements in the sequence currently under consideration. When the search begins, this range is the entire sequence since the target item can be anywhere within the sequence. The first step in each iteration is to determine the midpoint of the sequence. If the sequence contains an even number of elements, the mid point will be chosen such that the left sequence contains one less item than the right. Figure 5.4 illustrates the positioning of the `low`, `high`, and `mid` markers as the algorithm progresses.

Listing 5.4 Implementation of the binary search algorithm.

```
1  def binarySearch( theValues, target ) :
2     # Start with the entire sequence of elements.
3    low = 0
4    high = len(theValues) - 1
5
6     # Repeatedly subdivide the sequence in half until the target is found.
7    while low <= high :
8       # Find the midpoint of the sequence.
9      mid = (high + low) // 2
10      # Does the midpoint contain the target?
11     if theValues[mid] == target :
12       return True
13      # Or does the target precede the midpoint?
14     elif target < theValues[mid] :
15       high = mid - 1
16      # Or does it follow the midpoint?
17     else :
18       low = mid + 1
19
20    # If the sequence cannot be subdivided further, we're done.
21   return False
```

After computing the midpoint, the corresponding element in that position is examined. If the midpoint contains the target, we immediately return `True`. Otherwise, we determine if the target is less than the item at the midpoint or greater. If it is less, we adjust the `high` marker to be one less than the midpoint and if it is greater, we adjust the `low` marker to be one greater than the midpoint. In the next iteration of the loop, the only portion of the sequence considered are those elements between the `low` and `high` markers, as adjusted. This process is repeated until the item is found or the `low` marker becomes greater than the `high` marker. This condition occurs when there are no items left to be processed, indicating the target is not in the sorted sequence.

Run Time Analysis

To evaluate the efficiency of the the binary search algorithm, assume the sorted sequence contains n items. We need to determine the maximum number of times the

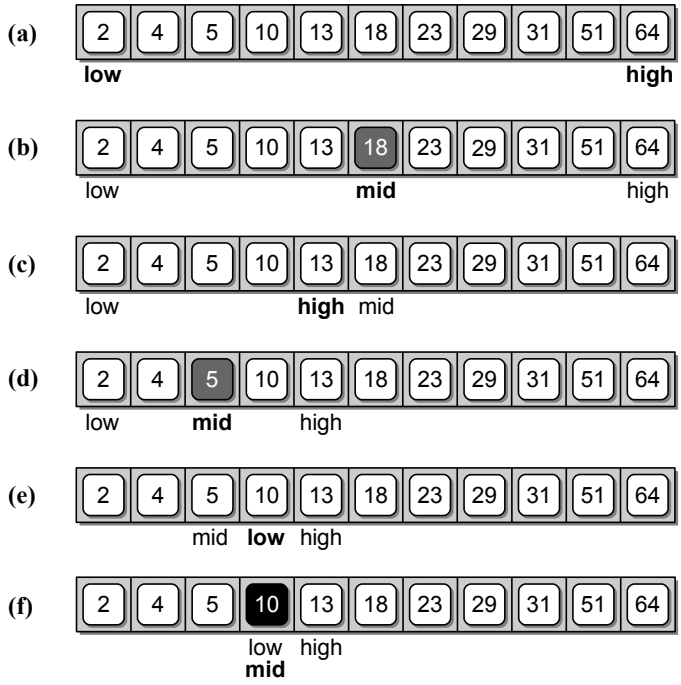


Figure 5.4: The steps performed by the binary search algorithm in searching for 10: (a) initial range of items, (b) locating the midpoint, (c) eliminating the upper half, (d) midpoint of the lower half, (e) eliminating the lower fourth, and (f) finding the target item.

`while` loop is executed. The worst case occurs when the target value is not in the sequence, the same as for the linear search. The difference with the binary search is that not every item in the sequence has to be examined before determining the target is not in the sequence, even in the worst case. Since the sequence is sorted, each iteration of the loop can eliminate from consideration half of the remaining values. As we saw earlier in Section 4.1.2, when the input size is repeatedly reduced by half during each iteration of a loop, there will be $\log n$ iterations in the worst case. Thus, the binary search algorithm has a worst case time-complexity of $O(\log n)$, which is more efficient than the linear search.

5.2 Sorting

Sorting is the process of arranging or ordering a collection of items such that each item and its successor satisfy a prescribed relationship. The items can be simple values, such as integers and reals, or more complex types, such as student records or dictionary entries. In either case, the ordering of the items is based on the value of a ***sort key***. The key is the value itself when sorting simple types or it can be a specific component or a combination of components when sorting complex types.

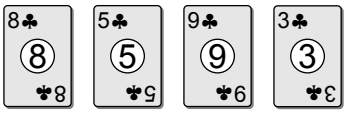
We encounter many examples of sorting in everyday life. Consider the listings of a phone book, the definitions in a dictionary, or the terms in an index, all of which are organized in alphabetical order to make finding an entry much easier. As we

saw earlier in the chapter, the efficiency of some applications can be improved when working with sorted lists. Another common use of sorting is for the presentation of data in some organized fashion. For example, we may want to sort a class roster by student name, sort a list of cities by zip code or population, rank order SAT scores, or list entries on a bank statement by date.

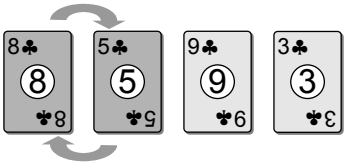
Sorting is one of the most studied problems in computer science and extensive research has been done in this area, resulting in many different algorithms. While Python provides a `sort()` method for sorting a list, it cannot be used with an array or other data structures. In addition, exploring the techniques used by some of the sorting algorithms for improving the efficiency of the sort problem may provide ideas that can be used with other types of problems. In this section, we present three basic sorting algorithms, all of which can be applied to data stored in a mutable sequence such as an array or list.

5.2.1 Bubble Sort

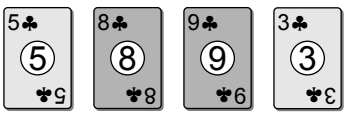
A simple solution to the sorting problem is the ***bubble sort*** algorithm, which rearranges the values by iterating over the list multiple times, causing larger values to bubble to the top or end of the list. To illustrate how the bubble sort algorithm works, suppose we have four playing cards (all of the same suit) that we want to order from smallest to largest face value. We start by laying the cards out face up on a table as shown here:



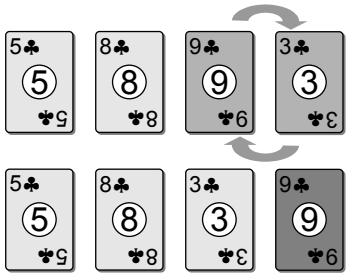
The algorithm requires multiple passes over the cards, with each pass starting at the first card and ending one card earlier than on the previous iteration. During each pass, the cards in the first and second positions are compared. If the first is larger than the second, the two cards are swapped.



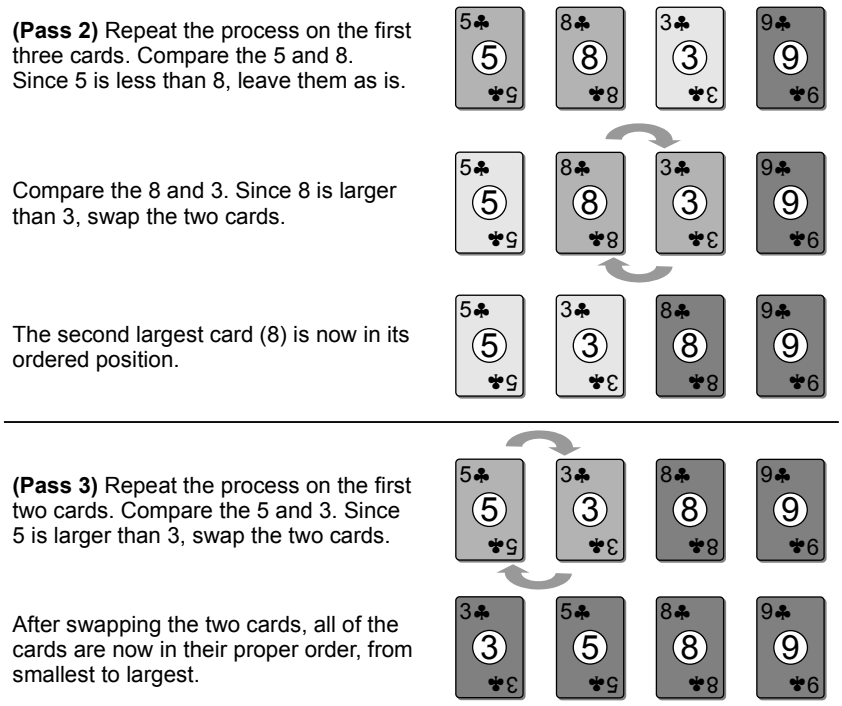
Next, the cards in positions two and three are compared. If the first one is larger than the second, they are swapped. Otherwise, we leave them as they were.



This process continues for each successive pair of cards until the card with the largest face value is positioned at the end.



The next two passes over the cards are illustrated below. In the second pass the card with the second largest face value is positioned in the next-to-last position. In the third and final pass, the first two cards will be positioned correctly.



Listing 5.5 provides a Python implementation of the bubble sort algorithm. Figure 5.5 illustrates the swaps performed during the first pass of the algorithm when applied to an array containing 11 integer values. Figure 5.6 shows the ordering of the values within the array after each iteration of the outer loop.

The efficiency of the bubble sort algorithm only depends on the number of keys in the array and is independent of the specific values and the initial arrangement of those values. To determine the efficiency, we must determine the total number of iterations performed by the inner loop for a sequence containing n values. The outer loop is executed $n-1$ times since the algorithm makes $n-1$ passes over the sequence. The number of iterations for the inner loop is not fixed, but depends on the current iteration of the outer loop. On the first pass over the sequence, the inner loop executes $n-1$ times; on the second pass, $n-2$ times; on the third, $n-3$ times, and so on until it executes once on the last pass. The total number

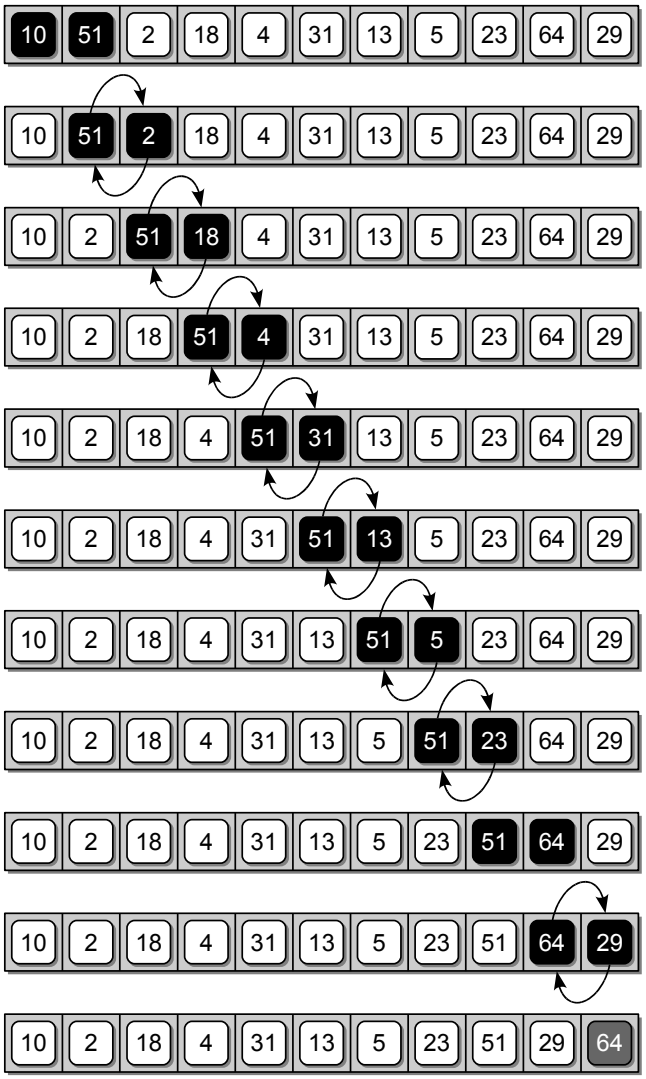


Figure 5.5: First complete pass of the bubble sort algorithm, which places 64 in its correct position. Black boxes represent values being compared; arrows indicate exchanges.

Listing 5.5 Implementation of the bubble sort algorithm.

```
1  # Sorts a sequence in ascending order using the bubble sort algorithm.
2  def bubbleSort( theSeq ):
3    n = len( theSeq )
4     # Perform n-1 bubble operations on the sequence
5    for i in range( n - 1 ) :
6       # Bubble the largest item to the end.
7      for j in range( i + n - 1 ) :
8        if theSeq[j] > theSeq[j + 1] :  # swap the j and j+1 items.
9          tmp = theSeq[j]
10         theSeq[j] = theSeq[j + 1]
11         theSeq[j + 1] = tmp
```

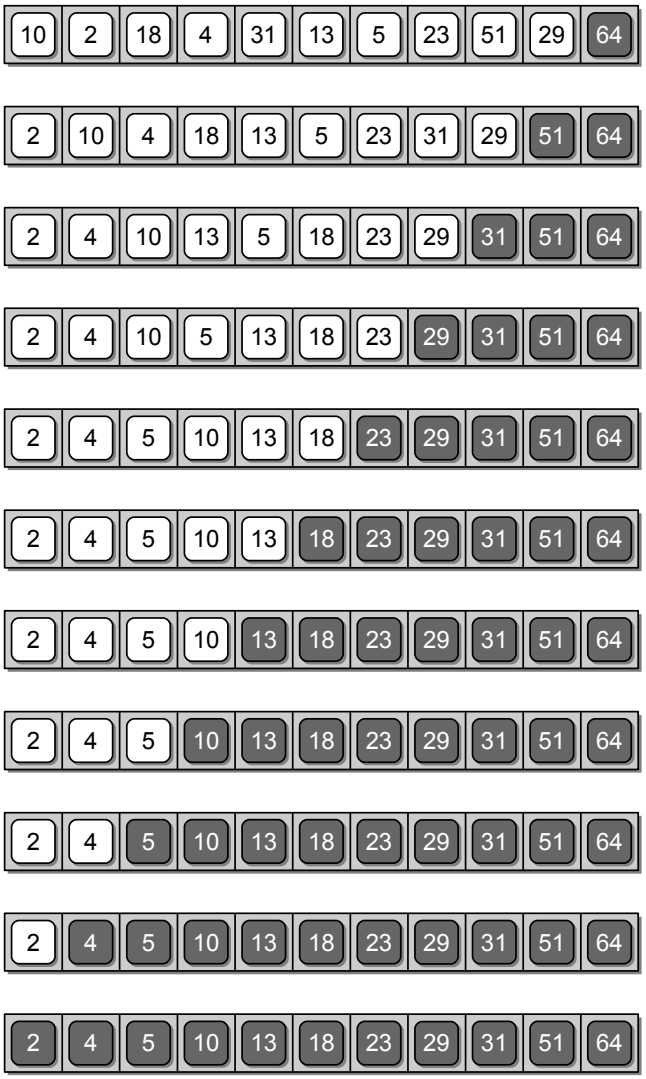


Figure 5.6: Result of applying the bubble sort algorithm to the sample sequence. The gray boxes show the values that are in order after each outer-loop traversal.

of iterations for the inner loop will be the sum of the first $n - 1$ integers, which equals

$$\frac{n(n-1)}{2} - n = \frac{1}{2}n^2 + \frac{1}{2}n$$

resulting in a run time of $O(n^2)$. Bubble sort is considered one of the most inefficient sorting algorithms due to the total number of swaps required. Given an array of keys in reverse order, a swap is performed for every iteration of the inner loop, which can be costly in practice.

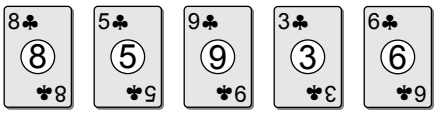
The bubble sort algorithm as implemented in Listing 5.5 always performs n^2 iterations of the inner loop. But what if the sequence is already in sorted order?

In this case, there would be no need to sort the sequence. But our implementation still performs all n^2 iterations because it has no way of knowing the sequence is already sorted.

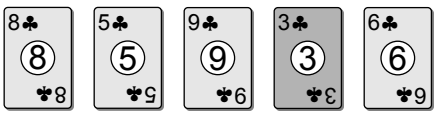
The bubble sort algorithm can be improved by having it terminate early and not require it to perform all n^2 iterations when the sequence is in sorted order. We can determine the sequence is in sorted order when no swaps are performed by the `if` statement within the inner loop. At that point, the function can return immediately without completing the remaining iterations. If a value is out of sorted order, then it will either be smaller than its predecessor in the sequence or larger than its successor at which point the condition of the `if` statement would be true. This improvement, which is left as an exercise, introduces a best case that only requires $O(n)$ time when the initial input sequence is in sorted order.

5.2.2 Selection Sort

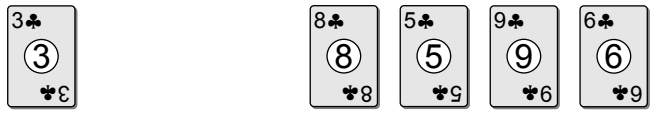
A second sorting algorithm, which improves on the bubble sort and works in a fashion similar to what a human may use to sort a list of values, is known as the ***selection sort***. We can again use the set of playing cards to illustrate the algorithm and start by placing five cards face up on a table that are to be sorted in ascending order.



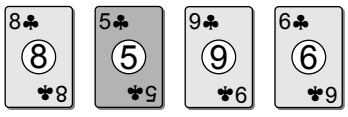
Instead of swapping the cards as was done with the bubble sort, we are going to scan through the cards and select the smallest from among those on the table and place it in our hand. For our set of cards, we identify the 3 as the smallest:



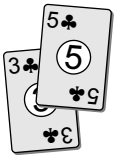
We pick up the 3 and place it in our hand, leaving the remaining cards on the table:

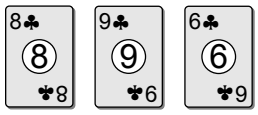


We repeat the process and identify the 5 as the next smallest face value:



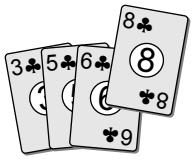
We pick up the 5 and add it to proper sorted position, which will be on the right side since there are no cards with a smaller face value left on the table.

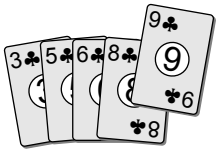

our hand

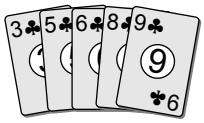

cards on the table

This process is continued until all of the cards have been picked up and placed in our hand in the correct sorted order from smallest to largest.


pick up the next smallest card (6)


pick up the next smallest card (8)


pickup the last card (9)


the resulting hand

The process we used to sort the set of five cards is similar to the approach used by the selection sort algorithm. But when implementing insertion sort in code, the algorithm maintains both the sorted and unsorted values within the same sequence structure. The selection sort, which improves on the bubble sort, makes multiple passes over the sequence, but unlike the bubble sort, it only makes a single swap after each pass. The implementation of the selection sort algorithm is provided in Listing 5.6.

Listing 5.6 Implementation of the selection sort algorithm.

```
# Sorts a sequence in ascending order using the selection sort algorithm.
def selectionSort( theSeq ):
  n = len( theSeq )
  for i in range( n - 1 ):
     # Assume the ith element is the smallest.
    smallNdx = i
     # Determine if any other element contains a smaller value.
    for j in range( i + 1, n ):
      if theSeq[j] < theSeq[smallNdx] :
        smallNdx = j

    # Swap the ith value and smallNdx value only if the smallest value is
    # not already in its proper position. Some implementations omit testing
    # the condition and always swap the two values.
   if smallNdx != i :
      tmp = theSeq[i]
      theSeq[i] = theSeq[smallNdx]
      theSeq[smallNdx] = tmp
```

The process starts by finding the smallest value in the sequence and swaps it with the value in the first position of the sequence. The second smallest value is then found and swapped with the value in the second position. This process continues positioning each successive value by selecting them from those not yet sorted and swapping with the values in the respective positions. Figure 5.7 shows the results after each iteration of the algorithm when applied to the sample array of integers. The grayed boxes represent those items already placed in their proper position while the black boxes show the two values that are swapped.

The selection sort, which makes $n-1$ passes over the array to reposition $n-1$ values, is also $O(n^2)$. The difference between the selection and bubble sorts is that the selection sort reduces the number of swaps required to sort the list to $O(n)$.

5.2.3 Insertion Sort

Another commonly studied sorting algorithm is the ***insertion sort***. Continuing with our analogy of sorting a set of playing cards to illustrate the sorting algorithms, consider five cards stacked in a deck face up:

We pick up the top card from the deck and place it in our hand:

Since this is the first card, there is no decision to be made as to its position. We again pick up the top card from the deck and compare it to the card already in our hand and insert it into its proper sorted position:

After placing the 8 into our hand, the process is repeated. This time, we pick up the 5 and find its position within our hand and insert it in the proper place:

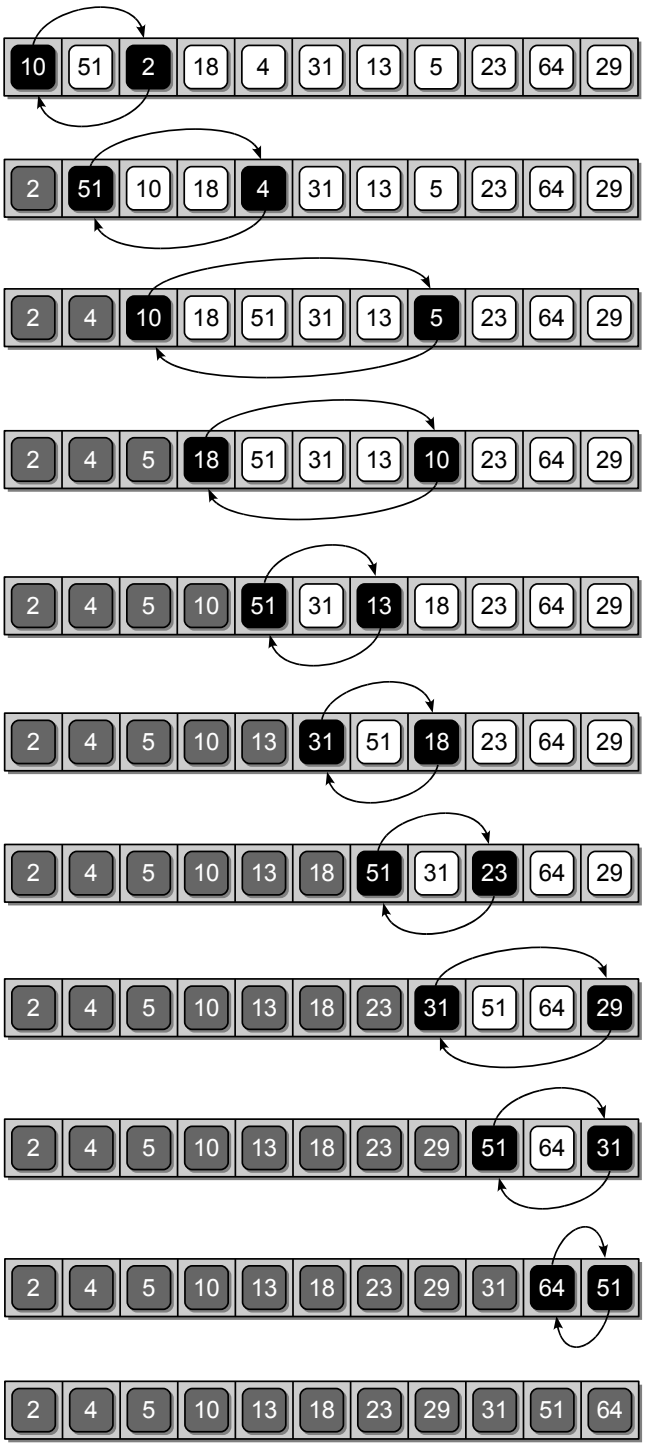
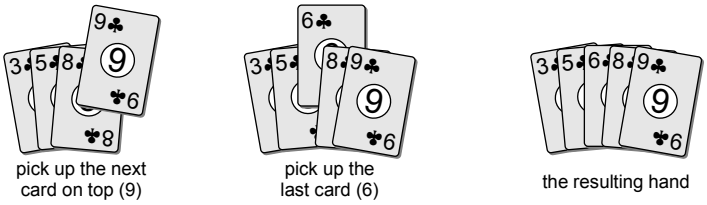


Figure 5.7: Result of applying the selection sort algorithm to our sample array. The gray boxes show the values that have been sorted; the black boxes show the values that are swapped during each iteration of the algorithm.

This process continues, one card at a time, until all of the cards have been removed from the table and placed into our hand in their proper sorted position.



The insertion sort maintains a collection of sorted items and a collection of items to be sorted. In the playing card analogy, the deck represents the collection to be sorted and the cards in our hand represents those already sorted. When implementing insertion sort in a program, the algorithm maintains both the sorted and unsorted collections within the same sequence structure. The algorithm keeps the list of sorted values at the front of the sequence and picks the next unsorted value from the first of those yet to be positioned. To position the next item, the correct spot within the sequence of sorted values is found by performing a search. After finding the proper position, the slot has to be opened by shifting the items down one position. A Python implementation of the insertion sort algorithm is provided in Listing 5.7.

Listing 5.7 Implementation of the insertion sort algorithm.

```
# Sorts a sequence in ascending order using the insertion sort algorithm.
def insertionSort( theSeq ):
  n = len( theSeq )
   # Starts with the first item as the only sorted entry.
  for i in range( 1, n ) :
     # Save the value to be positioned.
    value = theSeq[i]
     # Find the position where value fits in the ordered part of the list.
    pos = i
    while pos > 0 and value < theSeq[pos - 1] :
       # Shift the items to the right during the search.
      theSeq[pos] = theSeq[pos - 1]
      pos -= 1

     # Put the saved value into the open slot.
    theSeq[pos] = value
```

The `insertionSort()` function starts by assuming the first item is in its proper position. Next, an iteration is performed over the remaining items so each value can be inserted into its proper position within the sorted portion of the sequence. The ordered portion of the sequence is at the front while those yet to be inserted are at the end. The `i` loop index variable marks the separation point between the two parts. The inner loop is used to find the insertion point within the sorted sequence and at the same time, shifts the items down to make room for the next item. Thus, the inner loop starts from the end of the sorted subsequence and

works its way to the front. After finding the proper position, the item is inserted. Figure 5.8 illustrates the application of this algorithm on an array of integer values.

The insertion sort is an example of a sorting algorithm in which the best and worst cases are different. Determining the different cases and the corresponding run times is left as an exercise.

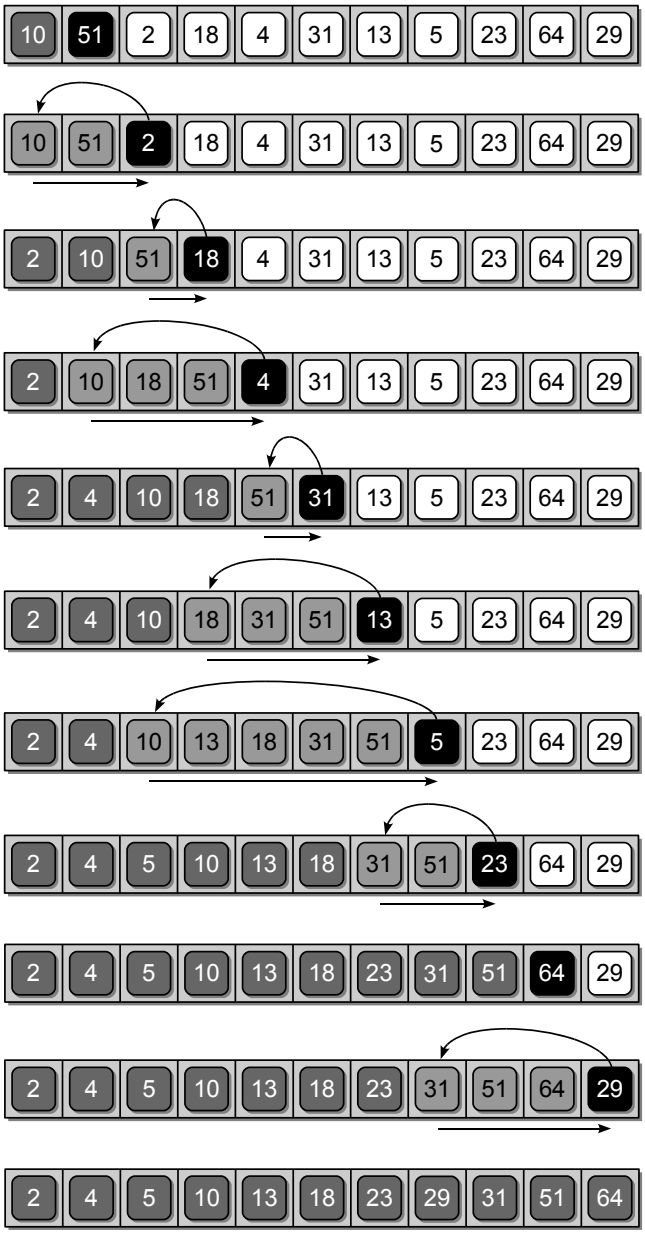


Figure 5.8: Result of applying the insertion sort algorithm to the sample array. The gray boxes show values that have been sorted; the black boxes show the next value to be positioned; and the lighter gray boxes with black text are the sorted values that have to be shifted to the right to open a spot for the next value.

5.3 Working with Sorted Lists

The efficiency of some algorithms can be improved when working with sequences containing sorted values. We saw this earlier when performing a search using the binary search algorithm on a sorted sequence. Sorting algorithms can be used to create a sorted sequence, but they are typically applied to an unsorted sequence in which all of the values are known and the collection remains static. In other words, no new items will be added to the sequence nor will any be removed.

In some problems, like the set abstract data type, the collection does not remain static but changes as new items are added and existing ones are removed. If a sorting algorithm were applied to the underlying list each time a new value is added to the set, the result would be highly inefficient since even the best sorting algorithm requires $O(n \log n)$ time. Instead, the sorted list can be maintained as the collection changes by inserting the new item into its proper position without having to re-sort the entire list. Note that while the sorting algorithms from the previous section all require $O(n^2)$ time in the worst case, there are more efficient sorting algorithms (which will be covered in Chapter 12) that only require $O(n \log n)$ time.

5.3.1 Maintaining a Sorted List

To maintain a sorted list in real time, new items must be inserted into their proper position. The new items cannot simply be appended at the end of the list as they may be out of order. Instead, we must locate the proper position within the list and use the `insert()` method to insert it into the indicated position. Consider the sorted list from Figure 5.3. If we want to add 25 to that list, then it must be inserted at position 7 following value 23.

To find the position of a new item within a sorted list, a modified version of the binary search algorithm can be used. The binary search uses a divide and conquer strategy to reduce the number of items that must be examined to find a target item or to determine the target is not in the list. Instead of returning `True` or `False` indicating the existence of a value, we can modify the algorithm to return the index position of the target if it's actually in the list or where the value should be placed if it were inserted into the list. The modified version of the binary search algorithm is shown in Listing 5.8.

Note the change to the two `return` statements. If the target value is contained in the list, it will be found in the same fashion as was done in the original version of the algorithm. Instead of returning `True`, however, the new version returns its index position. When the target is not in the list, we need the algorithm to identify the position where it should be inserted.

Consider the illustration in Figure 5.9, which shows the changes to the three variables `low`, `mid`, and `high` as the binary search algorithm progresses in searching for value 25. The `while` loop terminates when either the `low` or `high` range variable crosses the other, resulting in the condition `low > high`. Upon termination of the loop, the `low` variable will contain the position where the new value should be placed. This index can then be supplied to the `insert()` method to insert the new

Listing 5.8 Finding the location of a target value using the binary search.

```
# Modified version of the binary search that returns the index within
# a sorted sequence indicating where the target should be located.
def findSortedPosition( theList, target ):
  low = 0
  high = len(theList) - 1
  while low <= high :
    mid = (high + low) // 2
    if theList[mid] == target :
      return mid                    # Index of the target.
    elif target < theList[mid] :
      high = mid - 1
    else :
      low = mid + 1

  return low          # Index where the target value should be.
```

value into the list. The `findOrderedPosition()` function can also be used with lists containing duplicate values, but there is no guarantee where the new value will be placed in relation to the other duplicate values beyond the proper ordering requirement that they be adjacent.

5.3.2 Merging Sorted Lists

Sometimes it may be necessary to take two sorted lists and merge them to create a new sorted list. Consider the following code segment:

```
listA = [ 2, 8, 15, 23, 37 ]
listB = [ 4, 6, 15, 20 ]
newList = mergeSortedLists( listA, listB )
print( newList )
```

which creates two lists with the items ordered in ascending order and then calls a user-defined function to create and return a new list created by merging the other two. Printing the new merged list produces

```
[2, 4, 6, 8, 15, 15, 20, 23, 37]
```

Problem Solution

This problem can be solved by simulating the action a person might take to merge two stacks of exam papers, each of which are in alphabetical order. Start by choosing the exam from the two stacks with the name that comes first in alphabetical order. Flip it over on the table to start a new stack. Again, choose the exam from the top of the two stacks that comes next in alphabetical order and flip it over and place it on top of first one. Repeat this process until one of the two original stacks is exhausted. The exams in the remaining stack can be flipped over on top of the new stack as they are already in alphabetical order and alphabetically follow the

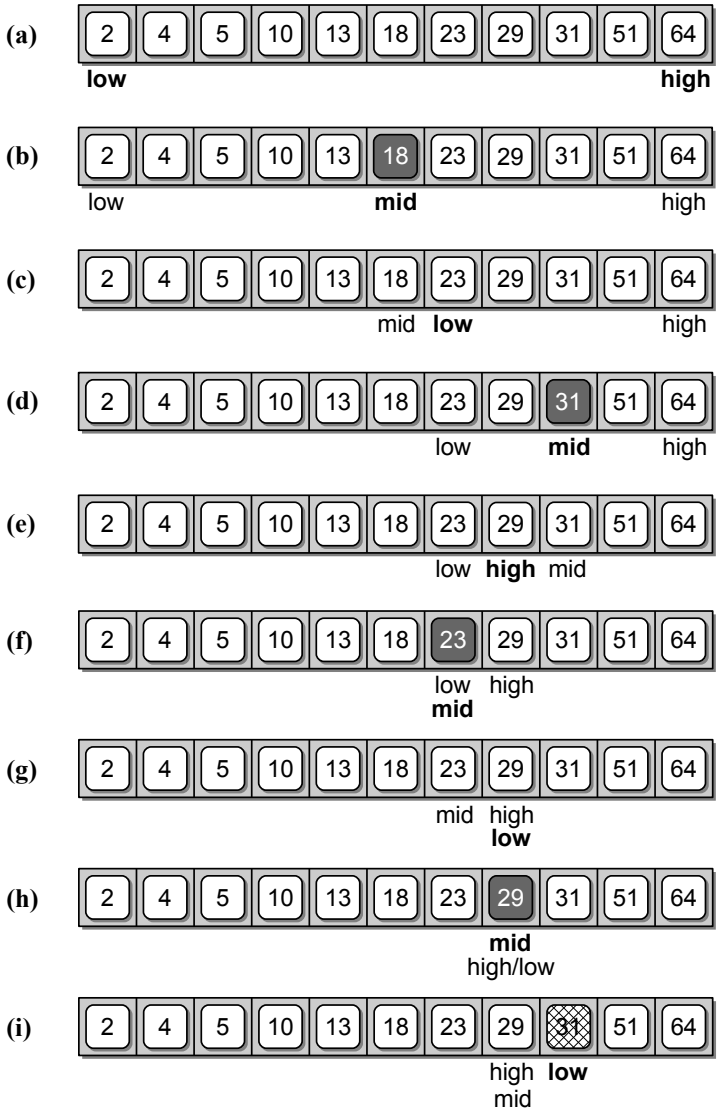


Figure 5.9: Performing a binary search on a sorted list when searching for value 25.

last exam flipped onto the new stack. You now have a single stack of exams in alphabetical order.

A similar approach can be used to merge two sorted lists. Consider the illustration in Figure 5.10, which demonstrates this process on the sample lists created in the example code segment from earlier. The items in the original list are not removed, but instead copied to the new list. Thus, there is no "top" item from which to select the smallest value as was the case in the example of merging two stacks of exams. Instead, index variables are used to indicate the "top" or next value within each list. The implementation of the `mergeSortedLists()` function is provided in Listing 5.9.

The process of merging the two lists begins by creating a new empty list and initializing the two index variables to zero. A loop is used to repeat the process

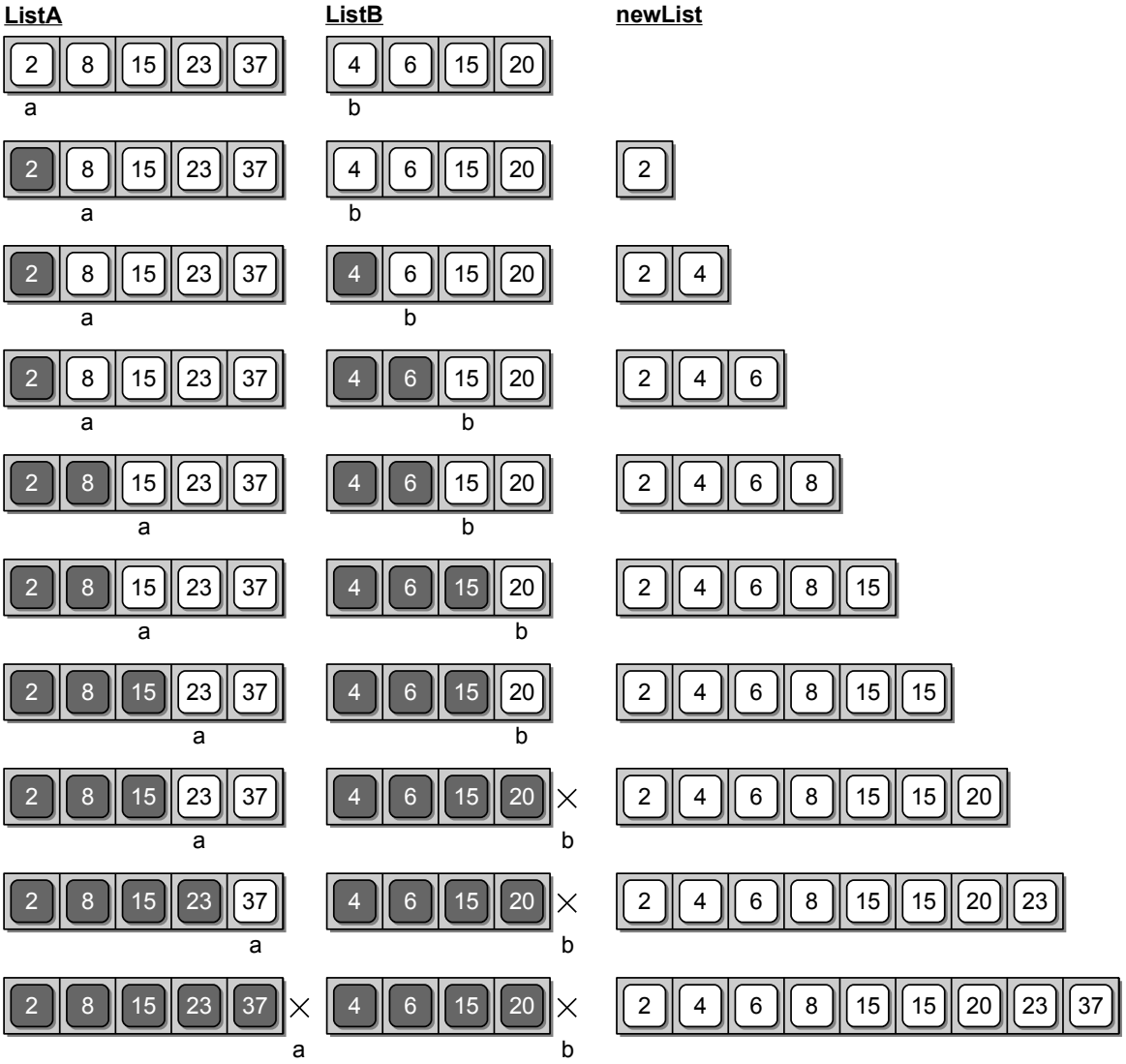


Figure 5.10: The iterative steps for merging two sorted lists into a new sorted list. `a` and `b` are index variables indicating the next value to be merged from the respective list.

of selecting the next largest value to be added to the new merged list. During the iteration of the loop, the value at `listA[a]` is compared to the value `listB[b]`. The largest of these two values is added or appended to the new list. If the two values are equal, the value from `listB` is chosen. As values are copied from the two original lists to the new merged list, one of the two index variables `a` or `b` is incremented to indicate the next largest value in the corresponding list.

This process is repeated until all of the values have been copied from one of the two lists, which occurs when `a` equals the length of `listA` or `b` equals the length of `listB`. Note that we could have created and initialized the new list with a sufficient number of elements to store all of the items from both `listA` and `listB`. While that works for this specific problem, we want to create a more general solution that

Listing 5.9 Merging two sorted lists.

```
 # Merges two sorted lists to create and return a new sorted list.
def mergeSortedLists( listA, listB ) :
   # Create the new list and initialize the list markers.
  newList = list()
  a = 0
  b = 0

   # Merge the two lists together until one is empty.
  while a < len( listA ) and b < len( listB ) :
    if listA[a] < listB[b] :
      newList.append( listA[a] )
      a += 1
    else :
      newList.append( listB[b] )
      b += 1

   # If listA contains more items, append them to newList.
  while a < len( listA ) :
    newList.append( listA[a] )
    a += 1

   # Or if listB contains more items, append them to newList.
  while b < len( listB ) :
    newList.append( listB[b] )
    b += 1

  return newList
```

we can easily modify for similar problems where the new list may not contain all of the items from the other two lists.

After the first loop terminates, one of the two lists will be empty and one will contain at least one additional value. All of the values remaining in that list must be copied to the new merged list. This is done by the next two while loops, but only one will be executed depending on which list contains additional values. The position containing the next value to be copied is denoted by the respective index variable `a` or `b`.

Run Time Analysis

To evaluate the solution for merging two sorted list, assume `listA` and `listB` each contain n items. The analysis depends on the number of iterations performed by each of the three loops, all of which perform the same action of copying a value from one of the two original lists to the new merged list. The first loop iterates until all of the values in one of the two original lists have been copied to the third. After the first loop terminates, only one of the next two loops will be executed, depending on which list still contains values.

- The first loop performs the maximum number of iterations when the selection of the next value to be copied alternates between the two lists. This results

in all values from either `listA` or `listB` being copied to the `newList` and all but one value from the other for a total of $2n - 1$ iterations. Then, one of the next two loops will execute a single iteration in order to copy the last value to the `newList`.

- The minimum number of iterations performed by the first loop occurs when all values from one list are copied to the `newList` and none from the other. If the first loop copies the entire contents of `listA` to the `newList`, it will require n iterations followed by n iterations of the third loop to copy the values from `listB`. If the first loop copies the entire contents of `listB` to the `newList`, it will require n iterations followed by n iterations of the second loop to copy the values from `listA`.

In both cases, the three loops are executed for a combined total of $2n$ iterations. Since the statements performed by each of the three loops all require constant time, merging two lists can be done in $O(n)$ time.

5.4 The Set ADT Revisited

The implementation of the Set ADT using a list was quick and rather simple, but several of the operations require quadratic time in the worst case. This inefficiency is due to the linear search used to find an element in the unsorted list that is required by several of the operations. We saw earlier in the chapter the efficiency of the search operation can be improved by using the binary search algorithm. To use the binary search with the Set ADT, the list of elements must be in sorted order and that order must be maintained. The definition of the Set ADT, however, indicates the elements have no particular ordering. While this is true, it does not preclude us from storing the elements in sorted order. It only means there is no requirement that the items must be stored in a particular order.

5.4.1 A Sorted List Implementation

In using the binary search algorithm to improve the efficiency of the set operations, the list cannot be sorted each time a new element is added because it would increase the time-complexity of the `add()` operation. For example, suppose we used one of the sorting algorithms presented earlier in the chapter to sort the list after each element is added. Since those algorithms require $O(n^2)$ time in the worst case, the `add()` operation would then also require quadratic time. Instead, the sorted order must be maintained when new elements are added by inserting each into its proper position. A partial implementation of the Set ADT using a sorted list and the binary search algorithm is provided in Listing 5.10. There are no changes needed in the constructor or the `__len__` method, but some changes are needed in the remaining methods.

Listing 5.10 The `binaryset.py` module.

```
# Implementation of the Set ADT using a sorted list.
class Set :
   # Creates an empty set instance.
  def __init__( self ):
    self._theElements = list()

   # Returns the number of items in the set.
  def __len__( self ):
    return len( self._theElements )

   # Determines if an element is in the set.
  def __contains__( self, element ):
    ndx = self._findPosition( element )
    return ndx < len( self ) and self._theElements[ndx] == element

   # Adds a new unique element to the set.
  def add( self, element ):
    if element not in self :
      ndx = self._findPosition( element )
      self._theElements.insert( ndx, element )

   # Removes an element from the set.
  def remove( self, element ):
    assert element in self, "The element must be in the set."
    ndx = self._findPosition( element )
    self._theElements.pop( ndx )

   # Determines if this set is a subset of setB.
  def isSubsetOf( self, setB ):
    for element in self :
      if element not in setB :
        return False
    return True

   # The remaining methods go here.
   # ......

   # Returns an iterator for traversing the list of items.
  def __iter__( self ):
    return _SetIterator( self._theElements )

   # Finds the position of the element within the ordered list.
  def _findPosition( self, element ):
    low = 0
    high = len( theList ) - 1
    while low <= high :
      mid = (high + low) / 2
      if theList[ mid ] == target :
        return mid
      elif target < theList[ mid ] :
        high = mid - 1
      else :
        low = mid + 1
    return low
```

NOTE

Short-Circuit Evaluation. Most programming languages use *short-circuit evaluation* when testing compound logical expressions. If the result of the compound expression is known after evaluating the first component, the evaluation ends and returns the result. For example, in evaluating the logical expression `a > b and a < c`, if `a > b` is `False`, then there is no need to continue the evaluation of the second component since the overall expression must be `False`.

Basic Operations

Performing a binary search to locate an element in the sorted list or to find the position where an element belongs in the sorted list is needed in several methods. Instead of reimplementing the operation each time, we implement the modified version of the binary search algorithm from Listing 5.8 in the `_findPosition()` helper method. The helper method does not detect nor distinguish between unique and duplicate values. It only returns the index where the element is located within the list or where it should be placed if it were added to the list. Thus, care must be taken when implementing the various methods to check for the existence of an element when necessary.

The `__contains__` method is easily implemented using `_findPosition()`. The index value returned by the helper method indicates the location where the element should be within the sorted list, but it says nothing about the actual existence of the element. To determine if the element is in the set, we can compare the element at the `ndx` position within the list to the target element. Note the inclusion of the condition `ndx < len(self)` within the compound expression. This is needed since the value returned by `_findPosition()` can be one larger than the number of items in the list, which occurs when the target should be located at the end of the list. If this value were directly used in examining the contents of the list without making sure it was in range, an out-of-range exception could be raised. The `__contains__` method has a worst case time of $O(\log n)$ since it uses the binary search to locate the given element within the sorted list.

To implement the `add()` method, we must first determine if the element is unique since a set cannot contain duplicate values. This is done with the use of the `in` operator, which automatically calls the `__contains__` operator method. If the element is not already a member of the set, the `insert()` method is called to insert the new element in its proper position within the ordered list. Even though the `__contains__` method has a better time-complexity when using a sorted list and the binary search, the `add()` operation still requires $O(n)$ time, the proof of which is left as an exercise.

The `remove()` method requires that the target element must be a member of the set. To verify this precondition, an assertion is made using the `in` operator. After which, the `_findPosition()` helper method is called to obtain the location of the element, which can then be used with the `pop()` list method to remove the element from the underlying sorted list. Like the add operation, the `remove()` method still has a worst case time of $O(n)$, the proof of which is left as an exercise.

We can implement the `isSubsetOf()` method in the same fashion as was done in the original version that used the unsorted list as shown in lines 29–33 of Listing 5.10. To evaluate the efficiency of the method, we again assume both sets contain n elements. The `isSubsetOf()` method performs a traversal over the `self` set during which the `in` operator is applied to `setB`. Since the `in` operator requires $O(\log n)$ time and it's called n times, `isSubsetOf()` has a time-complexity of $O(n \log n)$.

A New Set Equals

We could also implement the set equals operation in the same fashion as was done in the original version when using the unsorted list:

```
def __eq__( self, setB ):
  if len( self ) != len( setB ) :
    return False
  else :
    return self.isSubsetOf( setB )
```

But that implementation would have a time-complexity of $O(n \log n)$ since it calls the `isSubsetOf()` method. A more efficient implementation of the equals operation is possible if we compare the elements in the list directly instead of using the `isSubsetOf()` method. Remember, for two sets to be equal, they must contain the exact same elements. Since the lists for both sets are sorted, not only must they contain the same elements, but those elements must be in corresponding positions within the two lists for the sets to be equal.

The new implementation of the `__eq__` method is provided in Listing 5.11. The two lists are traversed simultaneously during which corresponding elements are compared. If a single instance occurs where corresponding elements are not identical, then the two sets cannot be equal. Otherwise, having traversed the entire list and finding no mismatched items, the two sets must be equal. The new implementation only requires $O(n)$ time since it only involves one complete traversal of the lists. A similar approach can be used to improve the efficiency of the `isSubsetOf()` method to only require $O(n)$ time, which is left as an exercise.

Listing 5.11 New implementation of the `Set` equals method.

```
1  class Set :
2  # ...
3    def __eq__( self, setB ):
4      if len( self ) != len( setB ) :
5        return False
6      else :
7        for i in range( len(self) ) :
8          if self._theElements[i] != setB._theElements[i] :
9            return False
10       return True
```

A New Set Union

The efficiency of the set union operation can also be improved from the original version. Set union using two sorted lists is very similar to the problem of merging two sorted lists that was introduced in the previous section. In that problem, the entire contents of the two sorted lists were merged into a third list. For the Set ADT implemented using a sorted list, the result of the set union must be a new sorted list merged from the unique values contained in the sorted lists used to implement the two source sets.

The implementation of the new `union()` method, which is provided in Listing 5.12, uses a modified version of the `mergeSortedLists()` function. The only modification required is the inclusion of an additional test within the first loop to catch any duplicate values by advancing both index variables and only appending one copy to the new sorted list. The new implementation only requires $O(n)$ time since that is the time required to merge two sorted lists and the new test for duplicates does not increase that complexity. The set difference and set intersection operations can also be modified in a similar fashion and are left as an exercise.

Listing 5.12 New implementation of the `Set` union method.

```
1  class Set :
2  # ...
3    def union( self, setB ):
4      newSet = Set()
5      a = 0
6      b = 0
7       # Merge the two lists together until one is empty.
8      while a < len( self ) and b < len( setB ) :
9        valueA = self._theElements[a]
10       valueB = setB._theElements[b]
11       if valueA < valueB :
12         newSet._theElements.append( valueA )
13         a += 1
14       elif valueA > valueB :
15         newSet._theElements.append( valueB )
16         b += 1
17       else :    # Only one of the two duplicates are appended.
18         newSet._theElements.append( valueA )
19         a += 1
20         b += 1
21
22      # If listA contains more items, append them to newList.
23     while a < len( self ) :
24       newSet._theElements.append( self._theElements[a] )
25       a += 1
26
27      # Or if listB contains more, append them to newList.
28     while b < len( otherSet ) :
29       newSet._theElements.append( setB._theElements[b] )
30       b += 1
31
32     return newSet
```

5.4.2 Comparing the Implementations

The implementation of the Set ADT using an unsorted list was quick and easy, but after evaluating the various operations, it became apparent many of them were time consuming. A new implementation using a sorted list to store the elements of the set and the binary search algorithm for locating elements improved the `__contains__` method. This resulted in better times for the `isSubsetOf()` and `__eq__` methods, but the set union, intersection, and difference operations remained quadratic. After observing several operations could be further improved if they were implemented to directly access the list instead of using the `__contains__` method, we were able to provide a more efficient implementation of the Set ADT. Table 5.1 compares the worst case time-complexities for the Set ADT operations using an unsorted list and the improved sorted list version using the binary search and the list merging operation.

Operation	Unordered	Improved
`s = Set()`	$O(1)$	$O(1)$
`len(s)`	$O(1)$	$O(1)$
`x in s`	$O(n)$	$O(\log n)$
`s.add(x)`	$O(n)$	$O(n)$
`s.isSubsetOf(t)`	$O(n^2)$	$O(n)$
`s == t`	$O(n^2)$	$O(n)$
`s.union(t)`	$O(n^2)$	$O(n)$

Table 5.1: Comparison of the two Set ADT implementations using an unsorted list and the improved sorted list with binary search and list merging.

Exercises

5.1 Given an unsorted list of n values, what is the time-complexity to find the k^th smallest value in the worst case? What would be the complexity if the list were sorted?

5.2 What is the $O(\cdot)$ for the `findSortedPosition()` function in the worst case?

5.3 Consider the new implementation of the `Set` class using a sorted list with the binary search.

(a) Prove or show the worst case time for the `add()` method is $O(n)$.

(b) What is the best case time for the `add()` method?

5.4 Determine the worst case time complexity for each method of the Map ADT implemented in Section 3.2.

5.5 Modify the binary search algorithm to find the position of the first occurrence of a value that can occur multiple times in the ordered list. Verify your algorithm is still $O(\log n)$.

5.6 Design and implement a function to find all negative values within a given list. Your function should return a new list containing the negative values. When does the worst case occur and what is the run time for that case?

5.7 In this chapter, we used a modified version of the `mergeSortedLists()` function to develop a linear time `union()` operation for our Set ADT implemented using a sorted list. Use a similar approach to implement new linear time versions of the `isSubsetOf()`, `intersect()`, and `difference()` methods.

5.8 Given the following list of keys (80, 7, 24, 16, 43, 91, 35, 2, 19, 72), show the contents of the array after each iteration of the outer loop for the indicated algorithm when sorting in ascending order.

(a) bubble sort (b) selection sort (c) insertion sort

5.9 Given the following list of keys (3, 18, 29, 32, 39, 44, 67, 75), show the contents of the array after each iteration of the outer loop for the

(a) bubble sort (b) selection sort (c) insertion sort

5.10 Evaluate the insertion sort algorithm to determine the best case and the worst case time complexities.

Programming Projects

5.1 Implement the Bag ADT from Chapter 1 to use a sorted list and the binary search algorithm. Evaluate the time complexities for each of the operations.

5.2 Implement a new version of the Map ADT from Section 3.2 to use a sorted list and the binary search algorithm.

5.3 The implementation of the Sparse Matrix ADT from Chapter 4 can be improved by storing the `_MatrixElement` objects in a sorted list and using the binary search to locate a specific element. The matrix elements can be sorted based on the row and column indices using an index function similar to that used with a 2-D array stored in a MultiArray. Implement a new version of the Sparse Matrix ADT using a sorted list and the binary search to locate elements.

5.4 Implement a new version of the Sparse Life Grid ADT from Chapter 4 to use a sorted list and the binary search to locate the occupied cells.

5.5 A colormap is a lookup table or color palette containing a limited set of colors. Early color graphics cards could only display up to 256 unique colors at one time. Colormaps were used to specify which 256 colors should be used to display color images on such a device. Software applications were responsible for mapping each color in the image that was to be displayed to a color in the limited color set specified by the colormap. We can define a Colormap ADT for storing a limited set of colors and for use in mapping one of the 16.7+ million colors possible in the discrete RGB color space to a color in the colormap. Given the description below of various operations, implement the Colormap ADT using a 1-D array structure.

- `ColorMap( k )`: Creates a new empty colormap that is capable of storing up to k colors.
- *`length`*`()`: Returns the number of colors currently stored in the colormap.
- *`contains`*`( color )`: Determines if the given color is contained in the colormap.
- `add( color )`: Adds the given color to the colormap. Only one instance of each color can be added to the colormap. In addition, a color cannot be added to a full colormap.
- *`remove`*`( color )`: Removes the given color from the colormap. The color must be contained in the colormap in order to be removed.
- `map( color )`: Maps the given `color` to an entry in the colormap and returns that color. A common approach is to map the `color` to its nearest neighbor in the colormap. The nearest neighbor of a color is the entry in the colormap that has the minimum Euclidean distance squared between the two colors. If there is more than one nearest neighbor in the colormap, only one is returned. In addition, the colormap must contain at least one color in order to perform the mapping operation.
- *`iterator`*`()`: Creates and returns an iterator object that can be used to iterate over the colors in the colormap.

5.6 Evaluate the `map()` method of your implementation of the Colormap ADT from the previous question to determine the worst case time-complexity.

5.7 Colormaps are used in color quantization, which is the process of reducing the number of colors in an image while trying to maintain the original appearance as much as possible. Part of the process recolors an original image using a reduced set of colors specified in a colormap.

(a) Implement the function `recolorImage( image, colormap )`, which produces a new `ColorImage` that results from mapping the color of each pixel in the given `image` to its nearest neighbor in the given `colormap`.

(b) What is the worst case time-complexity for your implementation?