CHAPTER 12

Advanced Sorting

We introduced the sorting problem in Chapter 5 and explored three basic sorting algorithms, but there are many others. Most sorting algorithms can be divided into two categories: comparison sorts and distribution sorts. In a ***comparison sort***, the data items can be arranged in either ascending (from smallest to largest) or descending (from largest to smallest) order by performing pairwise logical comparisons between the sort keys. The pairwise comparisons are typically based on either *numerical order* when working with integers and reals or *lexicographical order* when working with strings and sequences. A ***distribution sort***, on the other hand, distributes or divides the sort keys into intermediate groups or collections based on the individual key values. For example, consider the problem of sorting a list of numerical grades based on their equivalent letter grade instead of the actual numerical value. The grades can be divided into groups based on the corresponding letter grade without having to make comparisons between the numerical values.

The sorting algorithms described in Chapter 5 used nested iterative loops to sort a sequence of values. In this chapter, we explore two additional comparison sort algorithms, both of which use recursion and apply a divide and conquer strategy to sort sequences. Many of the comparison sorts can also be applied to linked lists, which we explore along with one of the more common distribution sorts.

12.1 Merge Sort

The ***merge sort*** algorithm uses the divide and conquer strategy to sort the keys stored in a mutable sequence. The sequence of values is recursively divided into smaller and smaller subsequences until each value is contained within its own subsequences. The subsequences are then merged back together to create a sorted sequence. For illustration purposes, we assume the mutable sequence is a list.

12.1.1 Algorithm Description

The algorithm starts by splitting the original list of values in the middle to create two sublists, each containing approximately the same number of values. Consider the list of integer values at the top of Figure 12.1. This list is first split following the element containing value 18. These two sublists are then split in a similar fashion to create four sublists and those four are split to create eight sublists.

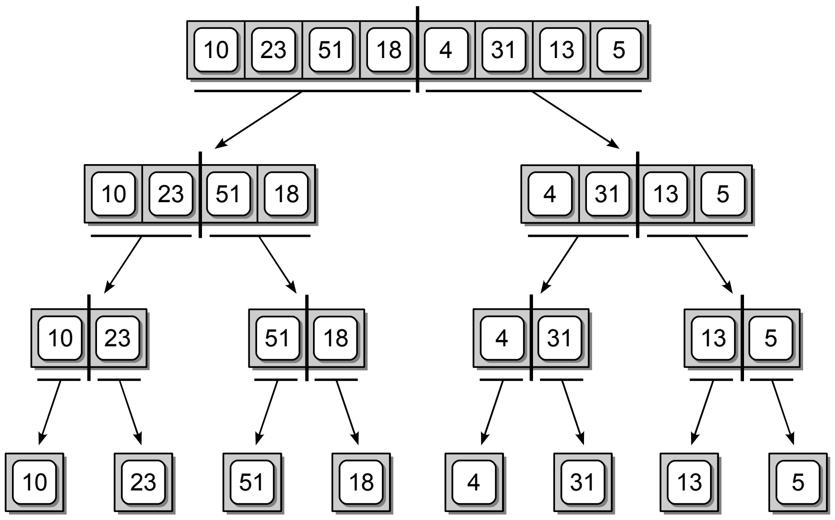


Figure 12.1: Recursively splitting a list until each element is contained within its own list.

After the list has been fully subdivided into individual sublists, the sublists are then merged back together, two at a time, to create a new sorted list. These sorted lists are themselves merged to create larger and larger lists until a single sorted list has been constructed. During the merging phase, each pair of sorted sublists are merged to create a new sorted list containing all of the elements from both sublists. This process is illustrated in Figure 12.2.

12.1.2 Basic Implementation

Given a basic description of the merge sort algorithm from an abstract view, we now turn our attention to the implementation details. There are two major steps in the merge sort algorithm: dividing the list of values into smaller and smaller sublists and merging the sublists back together to create a sorted list. The use of recursion provides a simple solution to this problem. The list can be subdivided by each recursive call and then merged back together as the recursion unwinds.

Listing 12.1 illustrates a simple recursive function for use with a Python list. If the supplied list contains a single item, it is by definition sorted and the list is simply returned, which is the base case of the recursive definition. If the list contains multiple items, it has to be split to create two sublists of approximately

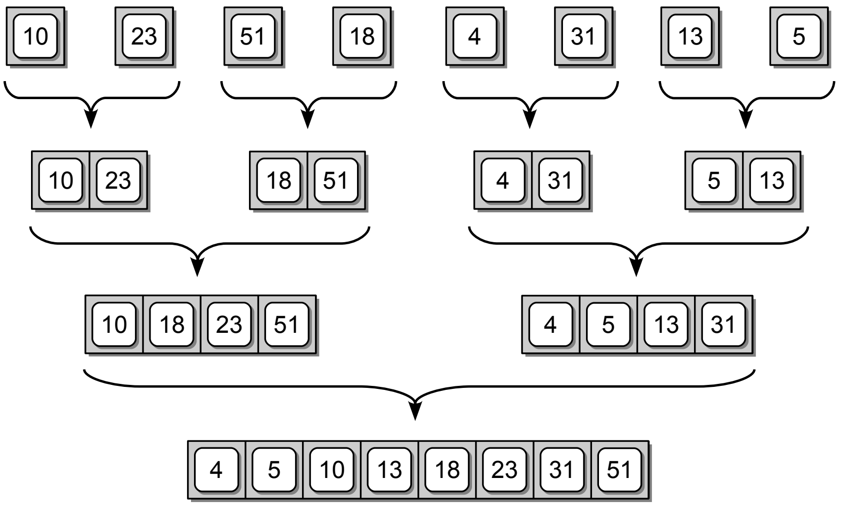


Figure 12.2: The sublists are merged back together to create a sorted list.

equal size. The split is handled by first computing the midpoint of the list and then using the slice operation to create two new sublists. The left sublist is then passed to a recursive call of the `pythonMergeSort()` function. That portion of the list will be processed recursively until it is completely sorted and returned. The right half of the list is then processed in a similar fashion. After both the left and right sublists have been ordered, the two lists are merged using the `mergeSortedLists()` function from Section 5.3.2. The new sorted list is returned.

Listing 12.1 Implementation of the merge sort algorithm for use with Python lists.

```
# Sorts a Python list in ascending order using the merge sort algorithm.
def pythonMergeSort( theList ):
   # Check the base case - the list contains a single item.
  if len(theList) <= 1 :
    return theList
  else :
     # Compute the midpoint.
    mid = len(theList) // 2

     # Split the list and perform the recursive step.
    leftHalf = pythonMergeSort( theList[ :mid ] )
    rightHalf = pythonMergeSort( theList[ mid: ] )

     #Merge the two ordered sublists.
    newList = mergeOrderedLists( leftHalf, rightHalf )
    return newList
```

The `pythonMergeSort()` function provides a simple recursive implementation of the merge sort algorithm, but it has several disadvantages. First, it relies on the use of the slice operation, which prevents us from using the function to sort an array of values since the array structure does not provide a slice operation. Second, new physical sublists are created in each recursive call as the list is subdivided. We learned in Chapter 4 that the slice operation can be time consuming since a new list has to be created and the contents of the slice copied from the original list. A new list is also created each time two sublists are merged during the unwinding of the recursion, adding yet more time to the overall process. Finally, the sorted list is not contained within the same list originally passed to the function as was the case with the sorting algorithms presented earlier in Chapter 5.

12.1.3 Improved Implementation

We can improve the implementation of the merge sort algorithm by using a technique similar to that employed with the binary search algorithm from Chapter 5. Instead of physically creating sublists when the list is split, we can use index markers to specify a subsequence of elements to create virtual sublists within the original physical list as was done with the binary search algorithm. Figure 12.3 shows the corresponding index markers used to split the sample list from Figure 12.1. The use of virtual sublists eliminates the need to repeatedly create new physical arrays or Python list structures during each recursive call.

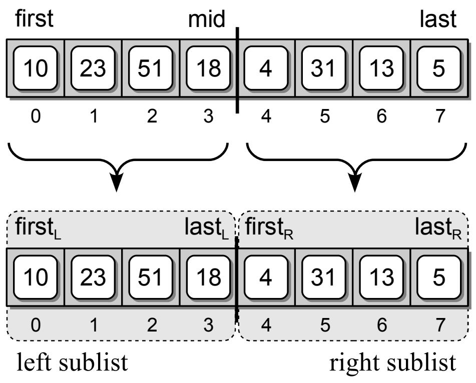


Figure 12.3: Splitting a list of values into two virtual sublists.

The new implementation of the merge sort algorithm is provided in Listing 12.2. The `recMergeSort()` function is very similar to the earlier implementation since both require the same steps to implement the merge sort algorithm. The difference is that `recMergeSort()` works with virtual sublists instead of using the slice operation to create actual sublists. This requires two index variables, `first` and `last`, for indicating the range of elements within the physical sublist that comprise the virtual sublist.

This implementation of the merge sort algorithm requires the use of a temporary array when merging the sorted virtual sublists. Instead of repeatedly creating a new array and later deleting it each time sublists are merged, we can create a single array and use it for every merge operation. Since this array is needed inside the `mergeVirtualLists()` function, it has to either be declared as a global

Listing 12.2 Improved implementation of the merge sort algorithm.

```
# Sorts a virtual subsequence in ascending order using merge sort.

def recMergeSort( theSeq, first, last, tmpArray ):
 # The elements that comprise the virtual subsequence are indicated
 # by the range [first...last]. tmpArray is temporary storage used in
 # the merging phase of the merge sort algorithm.

   # Check the base case: the virtual sequence contains a single item.
  if first == last :
    return;
  else :
     # Compute the mid point.
    mid = (first + last) // 2

     # Split the sequence and perform the recursive step.
    recMergeSort( theSeq, first, mid, tmpArray )
    recMergeSort( theSeq, mid+1, last, tmpArray )

     # Merge the two ordered subsequences.
    mergeVirtualSeq( theSeq, first, mid+1, last+1, tmpArray )
```

variable or created and passed into the recursive function before the first call. Our implementation uses the latter approach.

The `mergeVirtualSeq()` function, provided in Listing 12.3 on the next page, is a modified version of `mergeSortedLists()` from Section 5.3.2. The original function was used to create a new list that contained the elements resulting from merging two sorted lists. This version is designed to work with two virtual mutable subsequences that are stored adjacent to each other within the physical sequence structure, `theSeq`. Since the two virtual subsequences are always adjacent within the physical sequence, they can be specified by three array index variables: `left`, the index of the first element in the left subsequence; `right`, the index of the first element in the right subsequence; and `end`, the index of the first element following the end of the right subsequence. A second difference in this version is how the resulting merged sequence is returned. Instead of creating a new list structure, the merged sequence is stored back into the physical structure within the elements occupied by the two virtual subsequences.

The `tmpArray` argument provides a temporary array needed for intermediate storage during the merging of the two subsequences. The array must be large enough to hold all of the elements from both subsequences. This temporary storage is needed since the resulting sorted sequence is not returned by the function but instead is copied back to the original sequence structure. During the merging operation, the elements from the two subsequences are saved into the temporary array. After being merged, the elements are copied from the temporary array back to the original structure. We could create a new array each time the function is called, which would then be deleted when the function terminates. But that requires additional overhead that is compounded by the many calls to the `mergeVirtualSeq()` function during the execution of the recursive `recMergeSort()` function. To reduce

Listing 12.3 Merging two ordered virtual sublists.

```python
# Merges the two sorted virtual subsequences: [left..right) [right..end)
# using the tmpArray for intermediate storage.

def mergeVirtualSeq( theSeq, left, right, end, tmpArray ):
   # Initialize two subsequence index variables.
  a = left
  b = right
   # Initialize an index variable for the resulting merged array.
  m = 0
   # Merge the two sequences together until one is empty.
  while a < right and b < end :
    if theSeq[a] < theSeq[b] :
      tmpArray[m] = theSeq[a]
      a += 1
    else :
      tmpArray[m] = theSeq[b]
      b += 1
    m += 1

   # If the left subsequence contains more items append them to tmpArray.
  while a < right :
    tmpArray[m] = theSeq[a]
    a += 1
    m += 1

   # Or if right subsequence contains more, append them to tmpArray.
  while b < end :
    tmpArray[m] = theSeq[b]
    b += 1
    m += 1

   # Copy the sorted subsequence back into the original sequence structure.
  for i in range( end - left ) :
    theSeq[i+left] = tmpArray[i]
```

this overhead, implementations of the the merge sort algorithm typically allocate a single array that is of the same size as the original list and then simply pass the array into the `mergeVirtualSeq()` function. The use of the temporary array is illustrated in Figure 12.4 with the merging of the two subsequences lists formed from the second half of the original sequence.

The implementation of the earlier sorting algorithms only required the user to supply the array or list to be sorted. The `recMergeSort()` function, however,

NOTE

Wrapper Functions. A *wrapper function* is a function that provides a simpler and cleaner interface for another function and typically provides little or no additional functionality. Wrapper functions are commonly used with recursive functions that require additional arguments since their initial invocation may not be as natural as an equivalent sequential function.

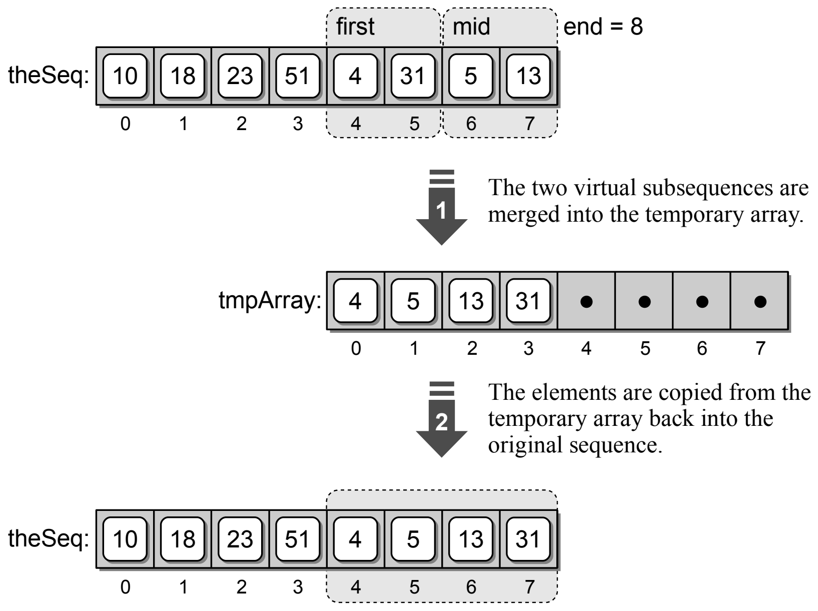

first
mid
end = 8
theSeq:
10 18 23 51 4 31 5 13
0 1 2 3 4 5 6 7
1
The two virtual subsequences are merged into the temporary array.
tmpArray:
4 5 13 31 ● ● ● ●
0 1 2 3 4 5 6 7
2
The elements are copied from the temporary array back into the original sequence.
theSeq:
10 18 23 51 4 5 13 31
0 1 2 3 4 5 6 7

Figure 12.4: A temporary array is used to merge two virtual subsequences.

requires not only the sequence structure but also the index markers and a temporary array. These extra arguments may not be as intuitive to the user as simply passing the sequence to be sorted. In addition, what happens if the user supplies incorrect range values or the temporary array is not large enough to merge the largest subsequence? A better approach is to provide a wrapper function for `recMergeSort()` such as the one shown in Listing 12.4. The `mergeSort()` function provides a simpler interface as it only requires the array or list to be sorted. The wrapper function handles the creation of the temporary array needed by the merge sort algorithm and initiates the first call to the recursive function.

Listing 12.4 A wrapper function for the new implementation of the merge sort algorithm.

```
# Sorts an array or list in ascending order using merge sort.
def mergeSort( theSeq ):
  n = len( theSeq )
   # Create a temporary array for use when merging subsequences.
  tmpArray = Array( n )
   # Call the private recursive merge sort function.
  recMergeSort( theSeq, 0, n-1, tmpArray )
```

12.1.4 Efficiency Analysis

We provided two implementations for the merge sort algorithm: one that can only be used with lists and employs the slice operation, and another that can be used with arrays or lists but requires the use of a temporary array in merging virtual subsequences. Both implementations run in $O(n \log n)$ time. To see how we obtain this result, assume an array of n elements is passed to `recMergeSort()` on the first invocation of the recursive function. For simplicity, we can let n be a power of 2, which results in subsequences of equal size each time a list is split.

As we saw in Chapter 10, the running time of a recursive function is computed by evaluating the time required by each function invocation. This evaluation only includes the time of the steps actually performed in the given function invocation. The recursive steps are omitted since their times will be computed separately.

We can start by evaluating the time required for a single invocation of the `recMergeSort()` function. Since each recursive call reduces the size of the problem, we let m represent the number of keys in the subsequence passed to the current instance of the function (n represents the size of the entire array). When the function is executed, either the base case or the divide and conquer steps are performed. The base case occurs when the supplied sequence contains a single item ($m = 1$), which results in the function simply returning without having performed any operations. This of course only requires $O(1)$ time. The dividing step is also a constant time operation since it only requires computing the midpoint to determine where the virtual sequence will be split. The real work is done in the conquering step by the `mergeVirtualLists()` function. This function requires $O(m)$ time in the worst case where m represents the total number of items in both subsequences. The analysis for the merging operation follows that of the `mergeSortedLists()` from Chapter 5 and is left as an exercise. Having determined the time required of the various operations, we can conclude that a single invocation of the `recMergeSort()` function requires $O(m)$ time given a subsequence of m keys.

The next step is to determine the total time required to execute all invocations of the recursive function. This analysis is best described using a recursive call tree. Consider the call tree in Figure 12.5, which represents the merge sort algorithm when applied to a sequence containing 16 keys. The values inside the function call boxes show the size of the subsequence passed to the function for that invocation. Since we know a single invocation of the `recMergeSort()` function requires $O(m)$

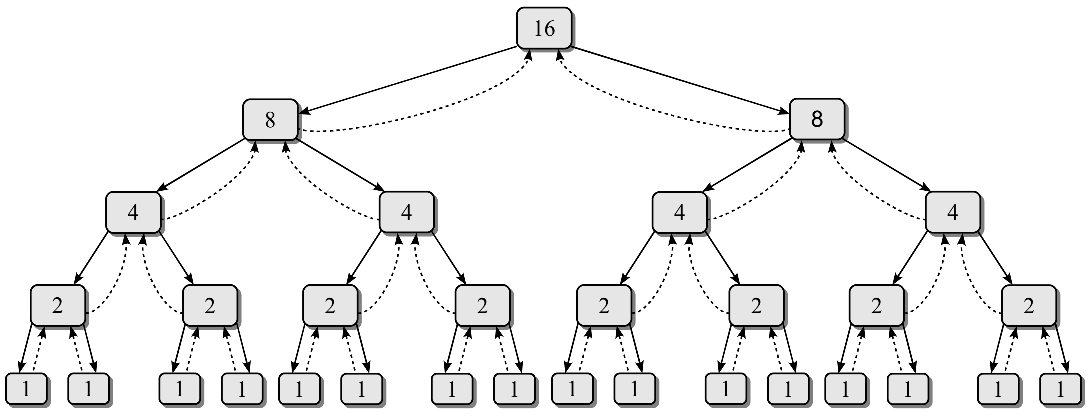


Figure 12.5: Recursive call tree for the merge sort algorithm for $n = 16$.

time, we can determine the time required for each instance of the function based on the size of the subsequence processed by that instance.

To obtain the total running time of the merge sort algorithm, we need to compute the sum of the individual times. In our sample call tree, where $n = 16$, the first recursive call processes the entire key list. This instance makes two recursive calls, each processing half ($\frac{n}{2}$) of the original key sequence, as shown on the second level of the call tree. The two function instances at the second level of the call tree each make two recursive calls, all of which process one-fourth ($\frac{n}{4}$) of the original key sequence. These recursive calls continue until the subsequence contains a single key value, as illustrated in the recursive call tree.

While each invocation of the function, other than the initial call, only processes a portion of the original key sequence, all n keys are processed at each level. If we can determine how many levels there are in the recursive call tree, we can multiply this value by the number of keys to obtain the final run time. When n is a power of 2, the merge sort algorithm requires $\log n$ levels of recursion. Thus, the merge sort algorithm requires $O(n \log n)$ time since there are $\log n$ levels and each level requires n time. The final analysis is illustrated graphically by the more general recursive call tree provided in Figure 12.6. When n is not a power of 2, the only difference in the analysis is that the lowest level in the call tree will not be completely full, but the call tree will still contain at most $\lceil \log n \rceil$ levels.

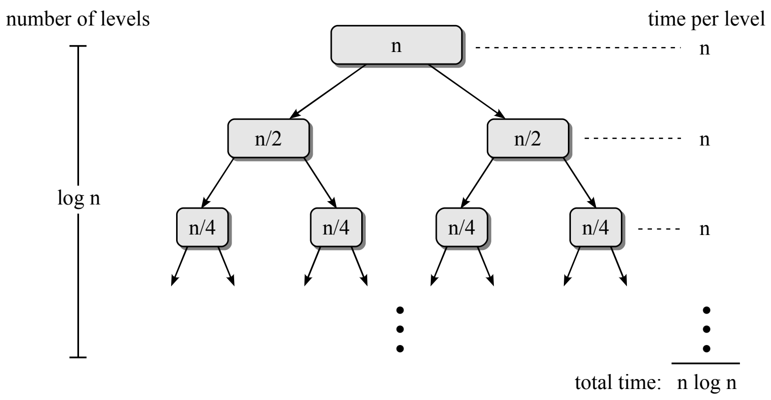


Figure 12.6: Time analysis of the merge sort algorithm.

12.2 Quick Sort

The ***quick sort*** algorithm also uses the divide and conquer strategy. But unlike the merge sort, which splits the sequence of keys at the midpoint, the quick sort partitions the sequence by dividing it into two segments based on a selected *pivot key*. In addition, the quick sort can be implemented to work with virtual subsequences without the need for temporary storage.

12.2.1 Algorithm Description

The quick sort is a simple recursive algorithm that can be used to sort keys stored in either an array or list. Given the sequence, it performs the following steps:

1. The first key is selected as the pivot, p. The pivot value is used to partition the sequence into two segments or subsequences, L and G, such that L contains all keys less than the p and G contains all keys greater than or equal to p.

2. The algorithm is then applied recursively to both L and G. The recursion continues until the base case is reached, which occurs when the sequence contains fewer than two keys.

3. The two segments and the pivot value are merged to produce a sorted sequence. This is accomplished by copying the keys from segment L back into the original sequence, followed by the pivot value and then the keys from segment G. After this step, the pivot key will end up in its proper position within the sorted sequence.

An abstract view of the partitioning step, in which much of the actual work is done, is illustrated in Figure 12.7. You will notice the size of the segments will vary depending on the value of the pivot. In some instances, one segment may not contain any elements. It depends on the pivot value and the relationship between that value and the other keys in the sequence. When the recursive calls return, the segments and pivot value are merged to produce a sorted sequence. This process is illustrated in Figure 12.8.

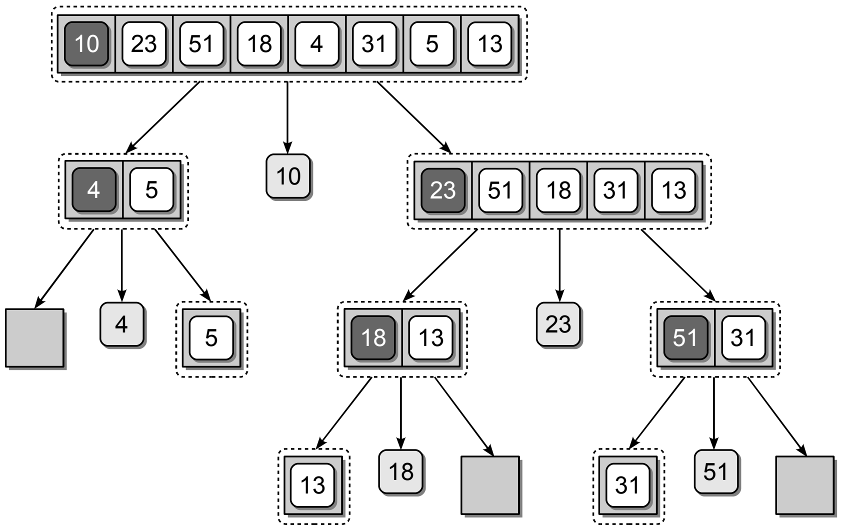


Figure 12.7: An abstract view showing how quick sort partitions the sequence into segments based on the pivot value (shown with a gray background).

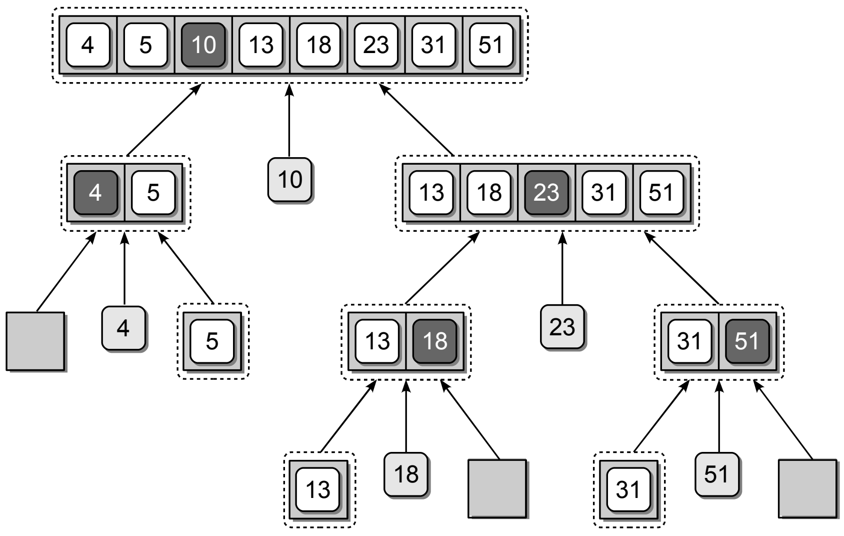


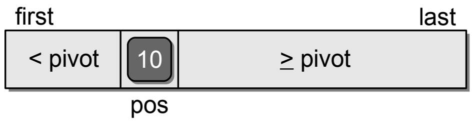
Figure 12.8: An abstract showing how quick sort merges the sorted segments and pivot value back into the original sequence.

12.2.2 Implementation

A simple implementation using the slice operation can be devised for the quick sort algorithm as was done with the merge sort but it would require the use of temporary storage. An efficient solution can be designed to work with virtual subsequences or segments that does not require temporary storage. However, it is not as easily implemented since the partitioning must be done using the same sequence structure.

A Python implementation of the quick sort algorithm is provided in Listing 12.5. The `quickSort()` function is a simple wrapper that is used to initiate the recursive call to `recQuickSort()`. The recursive function is rather simple and follows the enumerated steps described earlier. Note that `first` and `last` indicate the elements in the range [`first`...`last`] that comprise the current virtual segment.

The partitioning step is handled by the `partitionSeq()` function. This function rearranges the keys within the physical sequence structure by correctly positioning the pivot key within the sequence and placing all keys that are less than the pivot to the left and all keys that are greater to the right as shown here:



The final position of the pivot value also indicates the position at which the sequence is split to create the two segments. The left segment consists of the elements between the `first` element and the `pos - 1` element while the right segment consists of the elements between `pos + 1` and `last`, inclusive. The virtual segments

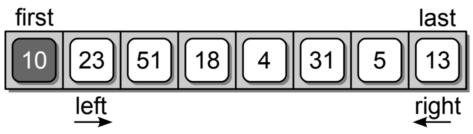
Listing 12.5 Implementation of the quick sort algorithm.

```python
# Sorts an array or list using the recursive quick sort algorithm.
def quickSort( theSeq ):
  n = len( theSeq )
  recQuickSort( theSeq, 0, n-1 )

# The recursive implementation using virtual segments.
def recQuickSort( theSeq, first, last ):
   # Check the base case.
  if first >= last :
    return
  else :
     # Save the pivot value.
    pivot = theSeq[first]

     # Partition the sequence and obtain the pivot position.
    pos = partitionSeq( theSeq, first, last )

     # Repeat the process on the two subsequences.
    recQuickSort( theSeq, first, pos - 1 )
    recQuickSort( theSeq, pos + 1, last )

# Partitions the subsequence using the first key as the pivot.
def partitionSeq( theSeq, first, last ):
   # Save a copy of the pivot value.
  pivot = theSeq[first]

   # Find the pivot position and move the elements around the pivot.
  left = first + 1
  right = last
  while left <= right :
     # Find the first key larger than the pivot.
    while left < right and theSeq[left] < pivot :
      left += 1

     # Find the last key in the sequence that is smaller than the pivot.
    while right >= left and theSeq[right] >= pivot :
      right -= 1

     # Swap the two keys if we have not completed this partition.
    if left < right :
      tmp = theSeq[left]
      theSeq[left] =  theSeq[right]
      theSeq[right] = tmp

   # Put the pivot in the proper position.
  if right != first :
    theSeq[first] = theSeq[right]
    theSeq[right] = pivot

   # Return the index position of the pivot value.
  return right
```

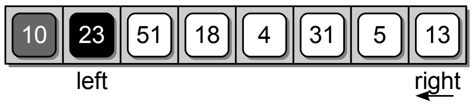
are passed to the recursive calls in lines 19 and 20 of Listing 12.5 using the proper index ranges.

After the recursive calls, the `recQuickSort()` function returns. In the earlier description, the sorted segments and pivot value had to be merged and stored back into the original sequence. But since we are using virtual segments, the keys are already stored in their proper position upon the return of the two recursive calls.

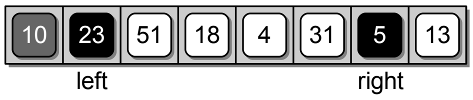
To help visualize the operation of the `partitionSeq()` function, we step through the first complete partitioning of the sample sequence. The function begins by saving a copy of the pivot value for easy reference and then initializes the two index markers, `left` and `right`. The `left` marker is initialized to the first position following the pivot value while the `right` marker is set to the last position within the virtual segment. The two markers are used to identify the range of elements within the sequence that will comprise the left and right segments.



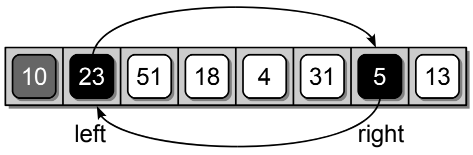
The main loop is executed until one of the two markers crosses the other as they are shifted in opposite directions. The `left` marker is shifted to the right by the loop in lines 32 and 33 of Listing 12.5 until a key value larger than the pivot is found or the `left` marker crosses the `right` marker. Since the `left` marker starts at a key larger than the pivot, the body of the outer loop is not executed if `theSeq` is empty.



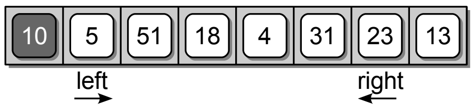
After the `left` marker is positioned, the `right` marker is then shifted to the left by the loop in lines 36 and 37. The marker is shifted until a key value less than or equal to the pivot is located or the marker crosses the `left` marker. The test for less than or equal allows for the correct sorting of duplicate keys. In our example, the `right` marker will be shifted to the position of the 5.



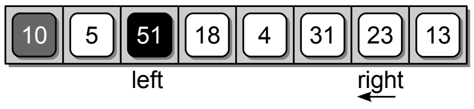
The two keys located at the positions marked by `left` and `right` are then swapped, which will place them within the proper segment once the location of the pivot is found.

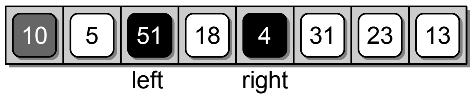


After the two keys are swapped, the two markers are again shifted starting where they left off:

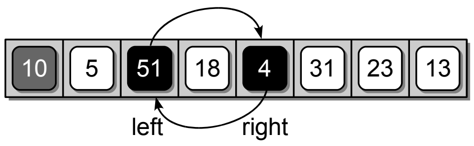


The `left` marker will be shifted to key value 51 and the `right` marker to value 4.

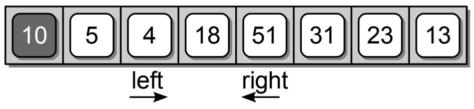


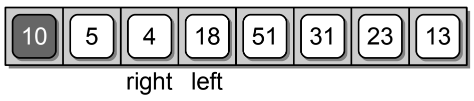


Once the two markers are shifted, the corresponding keys are swapped:

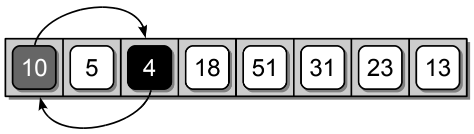


and the process is repeated. This time, the `left` marker will stop at value 18 while the `right` marker will stop at value 4.

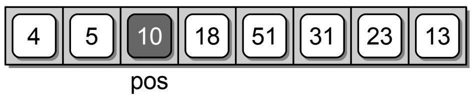




Note that the `right` marker has crossed the left such that `right < left`, resulting in the termination of the outer `while` loop. When the two markers cross, the `right` marker indicates the final position of the pivot value in the resulting sorted list. Thus, the pivot value currently located in the first element and the element marked by `right` have to be swapped:



resulting in value 10 being placed in element number 3, the final sorted position of the pivot within the original sequence:



The `if` statement at line 46 of Listing 12.5 is included to prevent a swap from occurring when the `right` marker is at the same position as the pivot value. This situation will occur when there are no keys in the list that are smaller than the pivot. Finally, the function returns the pivot position for use in splitting the sequence into the two segments.

We are not limited to selecting the first key within the list as the pivot, but it is the easiest to implement. We could have chosen the last key instead. But, in practice, using the first or last key as the pivot is a poor choice especially when a subsequence is already sorted that results in one of the segments being empty. Choosing a key near the middle is a better choice that can be implemented with a few modifications to the code provided. We leave these modifications as an exercise.

12.2.3 Efficiency Analysis

The quick sort algorithm has an average or expected time of $O(n \log n)$ but runs in $O(n^2)$ in the worst case, the analysis of which is left as an exercise. Even though quick sort is quadratic in the worst case, it does approach the average case in many instances and has the advantage of not requiring additional temporary storage as is the case with the merge sort. The quick sort is the commonly used algorithm to implement sorting in language libraries. Earlier versions of Python used quick sort to implement the `sort()` method of the list structure. In the current version of Python, a hybrid algorithm that combines the insertion and merge sort algorithms is used instead.

12.3 How Fast Can We Sort?

The comparison sort algorithms achieve their goal by comparing the individual sort keys to other keys in the list. We have reviewed five sorting algorithms in this chapter and Chapter 5. The first three—bubble, selection, and insertion—have a worst case time of $O(n^2)$ while the merge sort has a worst case time of $O(n \log n)$. The quick sort, the more commonly used algorithm in language libraries, is $O(n^2)$ in the worst case but it has an expected or average time of $O(n \log n)$. The natural question is can we do better than $O(n \log n)$? For a comparison sort, the answer is no. It can be shown, with the use of a decision tree and examining the permutations of all possible comparisons among the sort keys, that the worst case time for a comparison sort can be no better than $O(n \log n)$.

This does not mean, however, that the sorting operation cannot be done faster than $O(n \log n)$. It simply means that we cannot achieve this with a comparison sort. In the next section, we examine a distribution sort algorithm that works in linear time. Distribution sort algorithms use techniques other than comparisons among the keys themselves to sort the sequence of keys. While these distribution algorithms are fast, they are not general purpose sorting algorithms. In other words, they cannot be applied to just any sequence of keys. Typically, these algorithms are used when the keys have certain characteristics and for specific types of applications.

12.4 Radix Sort

Radix sort is a fast distribution sorting algorithm that orders keys by examining the individual components of the keys instead of comparing the keys themselves. For example, when sorting integer keys, the individual digits of the keys are compared from least significant to most significant. This is a special purpose sorting algorithm but can be used to sort many types of keys, including positive integers, strings, and floating-point values.

The radix sort algorithm also known as *bin sort* can be traced back to the time of punch cards and card readers. Card readers contained a number of bins in which punch cards could be placed after being read by the card reader. To sort values punched on cards the cards were first separated into 10 different bins based on the value in the ones column of each value. The cards would then be collected such that the cards in the bin representing zero would be placed on top, followed by the cards in the bin for one, and so on through nine. The cards were then sorted again, but this time by the tens column. The process continued until the cards were sorted by each digit in the largest value. The final result was a stack of punch cards with values sorted from smallest to largest.

12.4.1 Algorithm Description

To illustrate how the radix sort algorithm works, consider the array of values shown at the top of Figure 12.9. As with the card reader version, bins are used to store the various keys based on the individual column values. Since we are sorting positive integers, we will need ten bins, one for each digit.

The process starts by distributing the values among the various bins based on the digits in the ones column, as illustrated in step (a) of Figure 12.9. If keys have duplicate digits in the ones column, the values are placed in the bins in the order that they occur within the list. Thus, each duplicate is placed behind the keys already stored in the corresponding bin, as illustrated by the keys in bins 1, 3, and 8.

After the keys have been distributed based on the least significant digit, they are gathered back into the array, one bin at a time, as illustrated in step (b) of Figure 12.9. The keys are taken from each bin, without rearranging them, and inserted into the array with those in bin zero placed at the front, followed by those in bin one, then bin two, and so on until all of the keys are back in the sequence.

At this point, the keys are only partially sorted. The process must be repeated again, but this time the distribution is based on the digits in the tens column. After distributing the keys the second time, as illustrated in step (c) of Figure 12.9, they

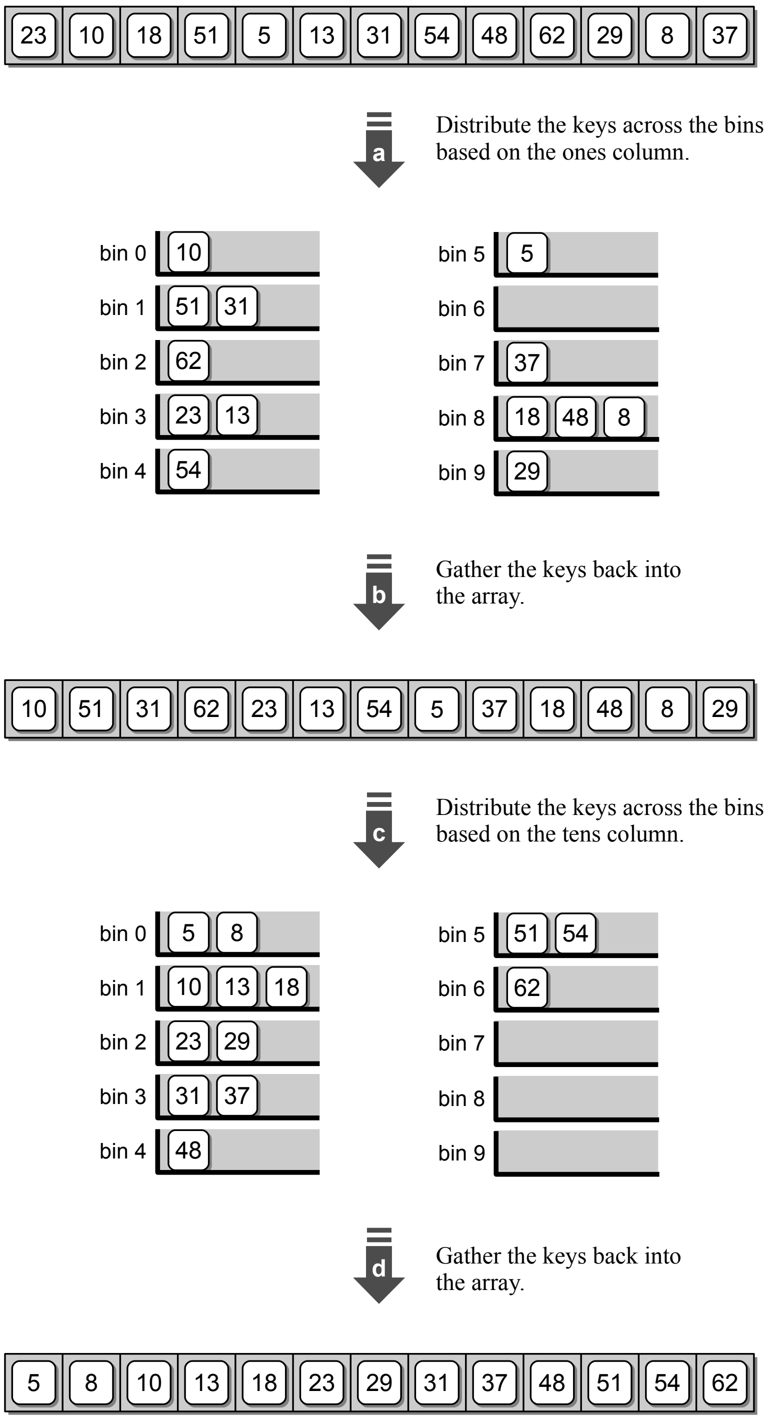


Figure 12.9: Sorting an array of integer keys using the radix sort algorithm.

are once again gathered back into the array, one bin at a time as shown in step (d). The result is a correct ordering of the keys from smallest to largest, as shown at the bottom of Figure 12.9.

In this example, the largest value (62) only contains two digits. Thus, we had to distribute and then gather the keys twice, once for the ones column and once for the tens column. If the largest value in the list had contain additional digits, the process would have to be repeated for each digit in that value.

12.4.2 Basic Implementation

The radix sort, as indicated earlier, is not a general purpose algorithm. Instead, it's used in special cases such as sorting records by zip code, Social Security number, or product codes. The sort keys can be represented as integers, reals, or strings. Different implementations are required, however, since the individual key components (digits or characters) differ based on the type of key. In addition, we must know the maximum number of digits or characters used by the largest key in order to know the number of iterations required to distribute the keys among the bins.

In this section, we implement a version of the radix sort algorithm for use with positive integer values stored in a mutable sequence. First, we must decide how to represent the bins used in distributing the values. Consider the following points related to the workings of the algorithm:

- The individual bins store groups of keys based on the individual digits.
- Keys with duplicate digits (in a given column) are stored in the same bin, but following any that are already there.
- When the keys are gathered from the bins, they have to be stored back into the original sequence. This is done by removing them from the bins in a first-in first-out ordering.

You may notice the bins sound very much like queues and in fact they can be represented as such. Adding a key to a bin is equivalent to enqueuing the key while removing the keys from the bins to put them back into the sequence is easily handled with the dequeue operation. Since there are ten digits, we will need ten queues. The queues can be stored in a ten-element array to provide easy management in the distribution and gathering of the keys. Our implementation of the radix sort algorithm is provided in Listing 12.6.

The function takes two arguments, the list of integer values to be sorted and the maximum number of digits possible in the largest key value. Instead of relying on the user to supply the number of digits, we could easily have searched for the largest key value in the sequence and then computed the number of digits in that value.

The implementation of the radix sort uses two loops nested inside an outer loop. The outer `for` loop iterates over the columns of digits with the number of iterations based on the user-supplied `numDigits` argument. The first nested loop in lines 19–21 distributes the keys across the bins. Since the queues are stored in

Listing 12.6 Implementation of the radix sort using an array of queues.

```
 1 # Sorts a sequence of positive integers using the radix sort algorithm.
 2
 3 from llistqueue import Queue
 4 from array import Array
 5
 6 def radixSort( intList, numDigits ):
 7   # Create an array of queues to represent the bins.
 8   binArray = Array( 10 )
 9   for k in range( 10 ):
10     binArray[k] = Queue()
11
12   # The value of the current column.
13   column = 1
14
15   # Iterate over the number of digits in the largest value.
16   for d in range( numDigits ):
17
18     # Distribute the keys across the 10 bins.
19     for key in intList :
20       digit = (key // column) % 10
21       binArray[digit].enqueue( key )
22
23     # Gather the keys from the bins and place them back in intList.
24     i = 0
25     for bin in binArray :
26       while not bin.isEmpty() :
27         intList[i] = bin.dequeue()
28         i += 1
29
30     # Advance to the next column value.
31     column *= 10
```

the ten-element array, the distribution is easily handled by determining the bin or corresponding queue to which each key has to be added (based on the digit in the current column being processed) and enqueuing it in that queue. To extract the individual digits, we can use the following arithmetic expression:

```
digit = (key // columnValue) % 10
```

where `column` is the value $(1, 10, 100, \ldots)$ of the current column being processed. The variable is initialized to 1 since we work from the least-significant digit to the most significant. After distributing the keys and then gathering them back into the sequence, we can advance to the next column by simply multiplying the current value by 10, as is done at the bottom of the outer loop in line 31.

The second nested loop, in lines 24–28, handles the gathering step. To remove the keys from the queues and place them back into the sequence, we must dequeue all of the keys from each of the ten queues and add them to the sequence in successive elements starting at index position zero.

This implementation of the radix sort algorithm is straightforward, but it requires the use of multiple queues. To result in an efficient implementation, we must use the Queue ADT implemented as a linked list or have direct access to the underlying list in order to use the Python list version.

12.4.3 Efficiency Analysis

To evaluate the radix sort algorithm, assume a sequence of n keys in which each key contains d components in the largest key value and each component contains a value between 0 and $k-1$. Also assume we are using the linked list implementation of the Queue ADT, which results in $O(1)$ time queue operations.

The array used to store the k queues and the creation of the queues themselves can be done in $O(k)$ time. The distribution and gathering of the keys involves two steps, which are performed d times, one for each component:

- The distribution of the n keys across the k queues requires $O(n)$ time since an individual queue can be accessed directly by subscript.
- Gathering the n keys from the queues and placing them back into the sequence requires $O(n)$ time. Even though the keys have to be gathered from k queues, there are n keys in total to be dequeued resulting in the `dequeue()` operation being performed n times.

The distribution and gathering steps are performed d times, resulting in a time of $O(dn)$. Combining this with the initialization step we have an overall time of $O(k+dn)$. The radix sort is a special purpose algorithm and in practice both k and d are constants specific to the given problem, resulting in a linear time algorithm. For example, when sorting a list of integers, $k = 10$ and d can vary but commonly $d < 10$. Thus, the sorting time depends only on the number of keys.

12.5 Sorting Linked Lists

The sorting algorithms introduced in the previous sections and earlier in Chapter 5 can be used to sort keys stored in a mutable sequence. But what if we need to sort keys stored in an unsorted singly linked list such as the one shown in Figure 12.10? In this section, we explore that topic by reviewing two common algorithms that can be used to sort a linked list by modifying the links to rearrange the existing nodes.

The techniques employed by any of the three quadratic sorting algorithms—bubble, selection, and insertion—presented in Chapter 5 can be used to sort a linked list. Instead of swapping or shifting the values within the sequence, however, the nodes are rearranged by unlinking each node from the list and then relinking them at a different position. A linked list version of the bubble sort would rearrange the nodes within the same list by leap-frogging the nodes containing larger values over those with smaller values. The selection and insertion sorts, on the other hand, would create a new sorted linked list by selecting and unlinking nodes from the original list and adding them to the new list.

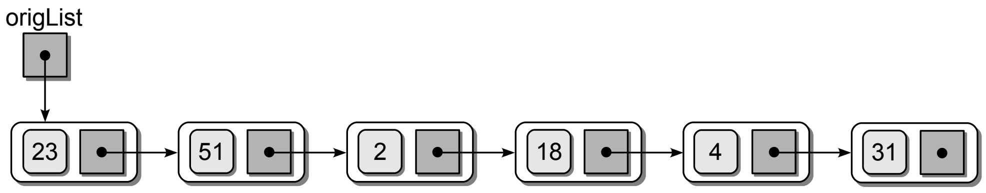


Figure 12.10: An unsorted singly linked list.

12.5.1 Insertion Sort

A simple approach for sorting a linked list is to use the technique employed by the insertion sort algorithm: take each item from an unordered list and insert them, one at a time, into an ordered list. When used with a linked list, we can unlink each node, one at a time, from the original unordered list and insert them into a new ordered list using the technique described in Chapter 6. The Python implementation is shown in Listing 12.7.

To create the sorted linked list using the insertion sort, we must unlink each node from the original list and insert them into a new ordered list. This is done in

Listing 12.7 Implementation of the insertion sort algorithm for use with a linked list.

```python
# Sorts a linked list using the technique of the insertion sort. A
# reference to the new ordered list is returned.

def llistInsertionSort( origList ):
   # Make sure the list contains at least one node.
  if origList is None :
    return None

   # Iterate through the original list.
  newList = None
  while origList is not None :
     # Assign a temp reference to the first node.
    curNode = origList

     # Advance the original list reference to the next node.
    origList = origList.next

     # Unlink the first node and insert into the new ordered list.
    curNode.next = None
    newList = addToSortedList( newList, curNode )

   # Return the list reference of the new ordered list.
  return newList
```

four steps, as illustrated in Figure 12.11 and implemented in lines 11–20. Inserting the node into the new ordered list is handled by the `addToSortedList()` function, which simply implements the operation from Listing 6.10. Figure 12.12 illustrates the results after each of the remaining iterations of the insertion sort algorithm when applied to our sample linked list.

The insertion sort algorithm used with linked lists is $O(n^2)$ in the worst case just like the sequence-based version. The difference, however, is that the items do not have to be shifted to make room for the unsorted items as they are inserted into the sorted list. Instead, we need only modify the links to rearrange the nodes.

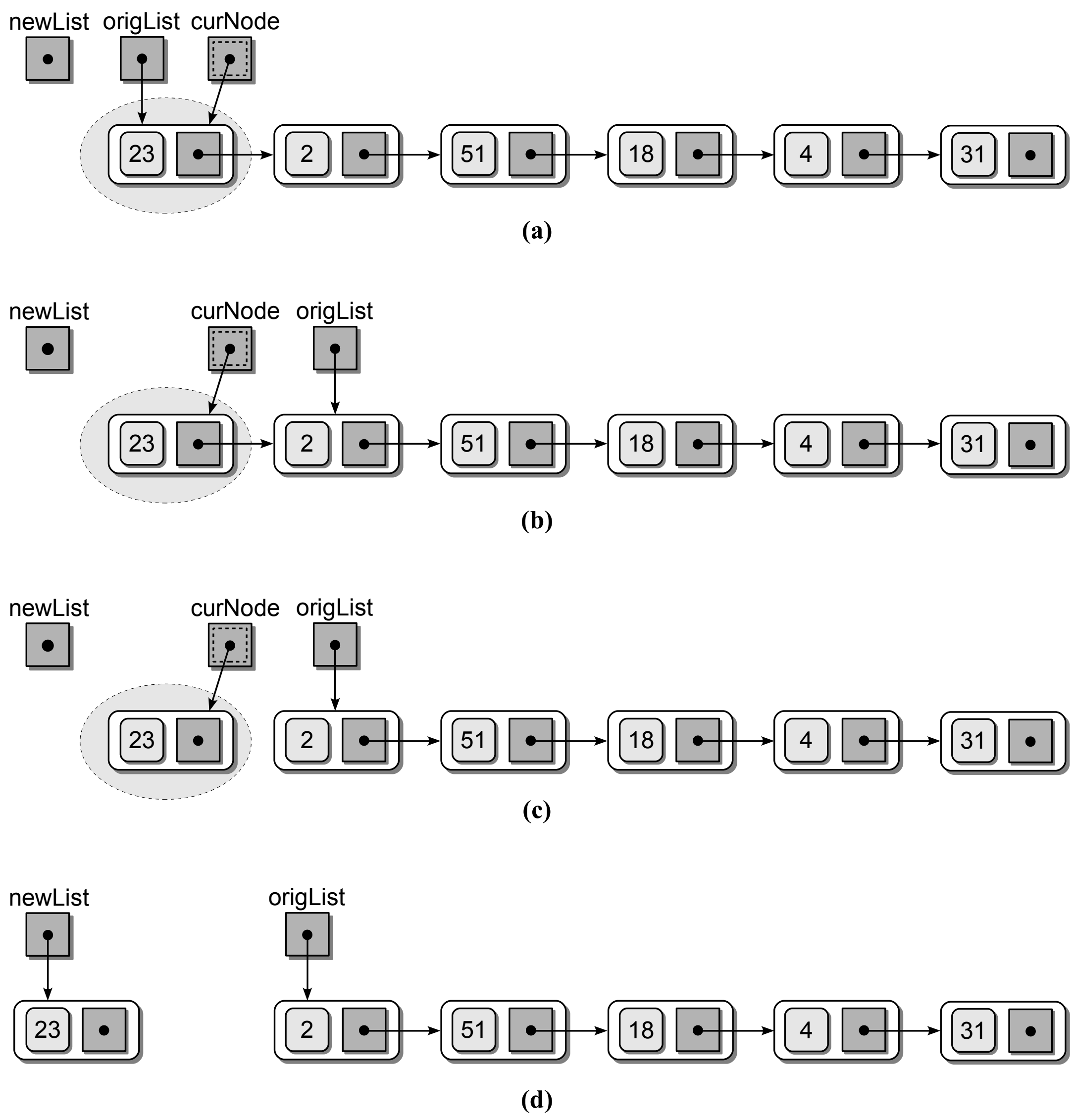


Figure 12.11: The individual steps performed in each iteration of the linked list insertion sort algorithm: (a) assign the temporary reference to the first node; (b) advance the list reference; (c) unlink the first node; and (d) insert the node into the new list.

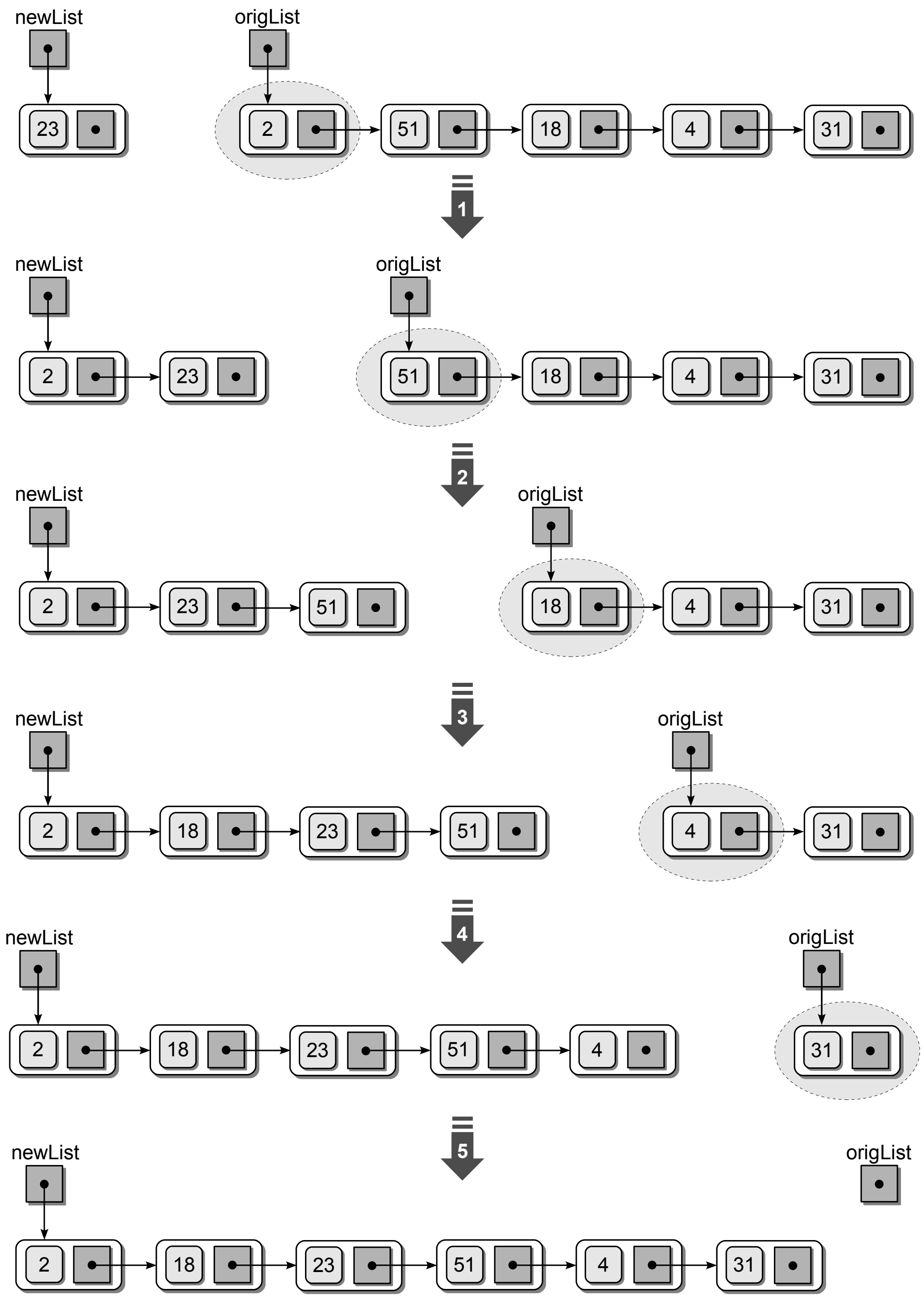


Figure 12.12: The results after each iteration of the linked list insertion sort algorithm.

12.5.2 Merge Sort

The merge sort algorithm is an excellent choice for sorting a linked list. Unlike the sequence-based version, which requires additional storage, when used with a linked list the merge sort is efficient in both time and space. The linked list version, which works in the same fashion as the sequence version, is provided in Listing 12.8.

Listing 12.8 The merge sort algorithm for linked lists.

```python
# Sorts a linked list using merge sort. A new head reference is returned.
def llistMergeSort( theList ):

  # If the list is empty (base case), return None.
  if theList is None :
    return None

  # Split the linked list into two sublists of equal size.
  rightList = _splitLinkedList( theList )
  leftList = theList

  # Perform the same operation on the left half...
  leftList = llistMergeSort( leftList )

  # ... and the right half.
  rightList = llistMergeSort( rightList )

  # Merge the two ordered sublists.
  theList = _mergeLinkedLists( leftList, rightList )

  # Return the head pointer of the ordered sublist.
  return theList

# Splits a linked list at the midpoint to create two sublists. The
# head reference of the right sublist is returned. The left sublist is
# still referenced by the original head reference.
def _splitLinkedList( subList ):

  # Assign a reference to the first and second nodes in the list.
  midPoint = subList
  curNode = midPoint.next

  # Iterate through the list until curNode falls off the end.
  while curNode is not None :
    # Advance curNode to the next node.
    curNode = curNode.next

    # If there are more nodes, advance curNode again and midPoint once.
    if curNode is not None :
      midPoint = midPoint.next
      curNode = curNode.next

  # Set rightList as the head pointer to the right sublist.
  rightList = midPoint.next
  # Unlink the right sub list from the left sublist.
  midPoint.next = None
  # Return the right sub list head reference.
  return rightList

# Merges two sorted linked list; returns head reference for the new list.
def _mergeLinkedLists( subListA, subListB ):
  # Create a dummy node and insert it at the front of the list.
  newList = ListNode( None )
  newTail = newList

  # Append nodes to the new list until one list is empty.
  while subListA is not None and subListB is not None :
    if subListA.data <= subListB.data :
      newTail.next = subListA
      subListA = subListA.next
    else :
      newTail.next = subListB
      subListB = subListB.next

    newTail = newTail.next
    newTail.next = None

  # If self list contains more terms, append them.
  if subListA is not None :
    newTail.next = subListA
  else :
    newTail.next = subListB

  # Return the new merged list, which begins with the first node after
  # the dummy node.
  return newList.next
```

The linked list is recursively subdivided into smaller linked lists during each recursive call, which are then merged back into a new ordered linked list. Since the nodes are not contained within a single object as the elements of an array are, the head reference of the new ordered list has to be returned after the list is sorted. To sort a linked list using the merge sort algorithm, the sort function would be called using the statement:

```
theList = llistMergeSort( theList )
```

The implementation in Listing 12.8 includes the recursive function and two helper functions. You will note that a wrapper function is not required with this version since the recursive function only requires the head reference of the list being sorted as the single argument.

Splitting the List

The split operation is handled by the `_splitLinkedList()` helper function, which takes as an argument the head reference to the singly linked list to be split and returns the head reference for the right sublist. The left sublist can still be referenced by the original head reference. To split a linked list, we need to know the midpoint, or more specifically, the node located at the midpoint. An easy way to find the midpoint would be to traverse through the list and count the number of nodes and then iterate the list until the node at the midpoint is located. This is not the most efficient approach since it requires one and a half traversals through the list.

Instead, we can devise a solution that requires one complete list traversal, as shown in lines 27–48 of Listing 12.8. This approach uses two external references, `midPoint` and `curNode`. The two references are initialized with `midPoint` referencing the first node and `curNode` referencing the second node. The two references are advanced through the list using a loop as is done in a normal list traversal, but the `curNode` reference will advance twice as fast as the `midPoint` reference. The traversal continues until `curNode` becomes null, at which point the `midPoint` reference will be pointing to the last node in the left sublist. Figure 12.13 illustrates the traversal required to find the midpoint of our sample linked list.

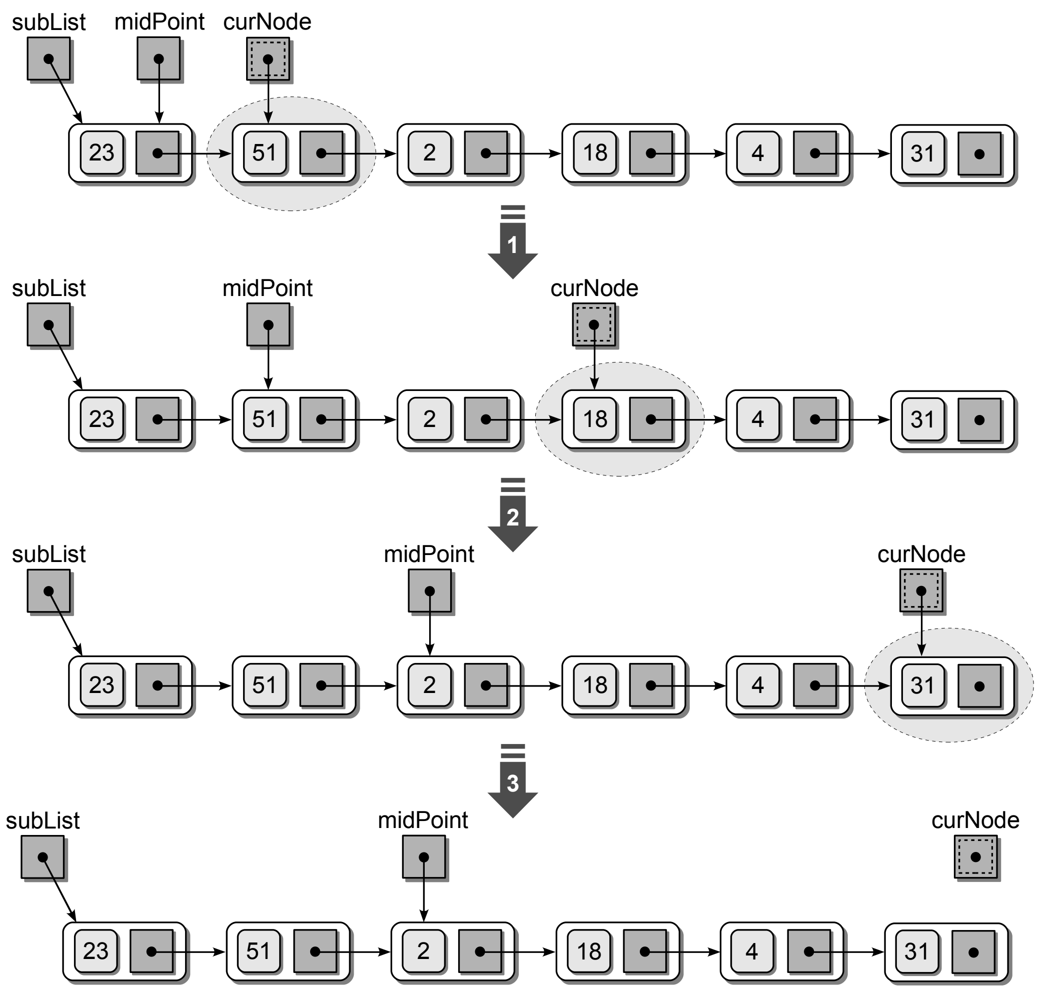


Figure 12.13: Sequence of steps for finding the midpoint in a linked list.

After the midpoint is located, the link between the node referenced by `midPoint` and its successor can be removed, creating two sublists, as illustrated in Figure 12.14. Before the link is removed, a new head reference `rightList` has to be created and initialized to reference the first node in the right sublist. The `rightList` head reference is returned by the function to provide access to the new sublist.

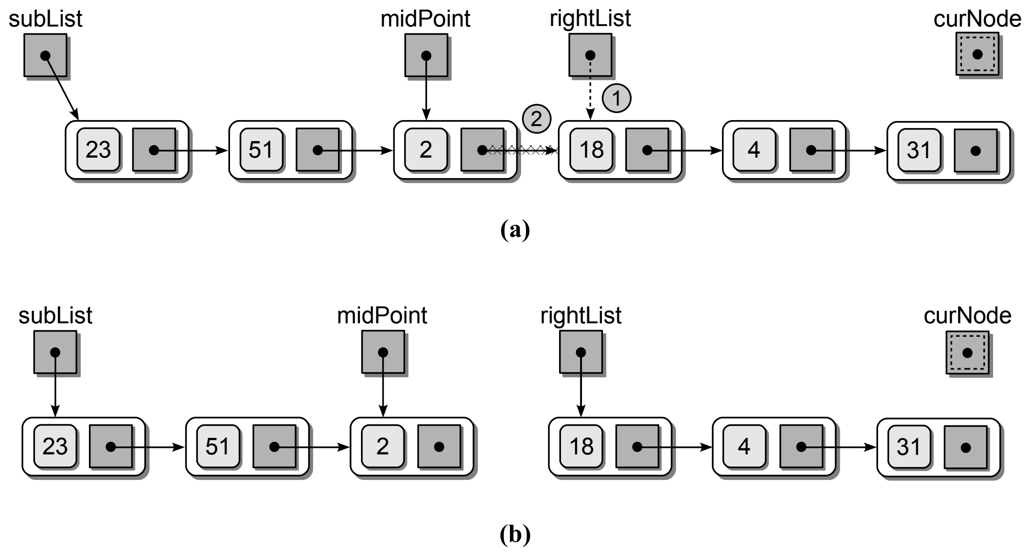


Figure 12.14: Splitting the list after finding the midpoint: (a) link modifications required to unlink the last node of the left sublist from the right sublist and (b) the two sublists resulting from the split.

Merging the Lists

The `_mergeLinkedLists()` function, provided in lines 51–76 of Listing 12.8, manages the merging of the two sorted linked lists. In Chapter 4, we discussed an efficient solution for the problem of merging two sorted Python lists, and earlier in this chapter that algorithm was adapted for use with arrays. The array and Python list versions are rather simple since we can refer to individual elements by index and easily append the values to the sequence structure.

Merging two sorted linked lists requires several modifications to the earlier algorithm. First, the nodes from the two sublists will be removed from their respective list and appended to a new sorted linked list. We can use a tail reference with the new sorted list to allow the nodes from the sublists to be appended in $O(1)$ time. Second, after all of the nodes have been removed from one of the two sublists, we do not have to iterate through the other list to append the nodes. Instead, we can simply link the last node of the new sorted list to the first node in the remaining sublist. Finally, we can eliminate the special case of appending the first node to the sorted list with the use of a ***dummy node*** at the front of the list, as illustrated in Figure 12.15. The dummy node is only temporary and will not be part of the final sorted list. Thus, after the two sublists have been merged, the function returns a reference to the second node in the list (the first real node following the dummy node), which becomes the head reference.

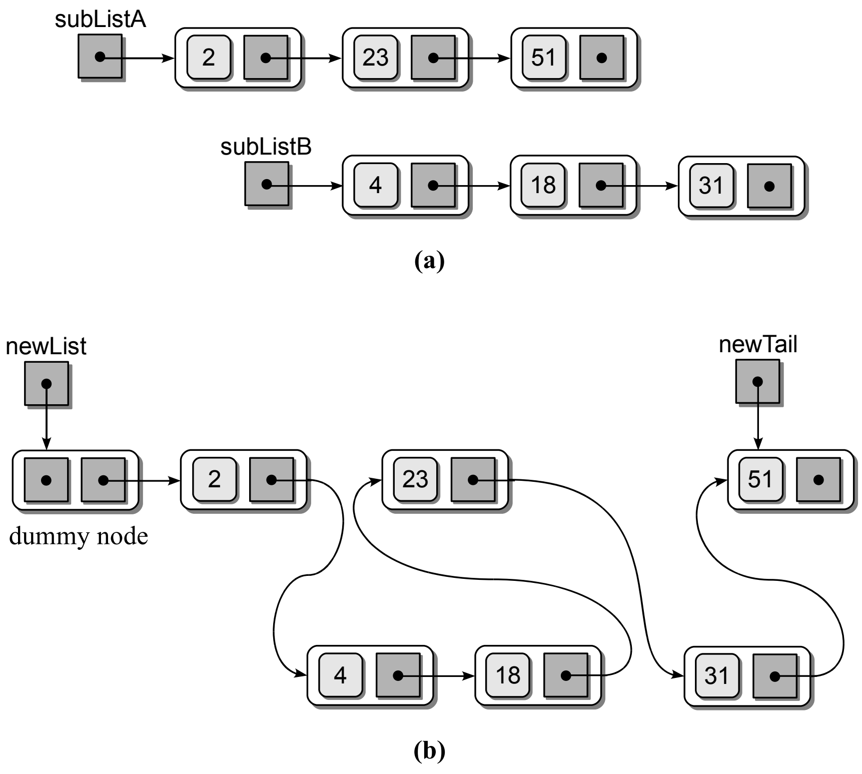


Figure 12.15: Merging two ordered linked lists using a dummy node and tail reference.

The linked list version of the merge sort algorithm is also a $O(n \log n)$ function but it does not require temporary storage to merge the sublists. The analysis of the run time is left as an exercise.

NOTE

Dummy Nodes. A dummy node is a temporary node that is used to simplify link modifications when adding or removing nodes from a linked list. They are called dummy nodes because they contain no actual data. But they are part of the physical linked structure.

Exercises

12.1 Given the following sequence of keys (80, 7, 24, 16, 43, 91, 35, 2, 19, 72), trace the indicated algorithm to produce a recursive call tree when sorting the values in descending order.

(a) merge sort

(b) quick sort

12.2 Do the same as in Exercise 12.1 but produce a recursive call tree when sorting the values in ascending order.

12.3 Show the distribution steps performed by the radix sort when ordering the following list of keys:

(a) $135, 56, 21, 89, 395, 7, 178, 19, 96, 257, 34, 29$

(b) $1.25, 2.46, 0.34, 8.67, 3.21, 1.09, 3.33, 0.02, 5.44, 7.78, 1.93, 4.22$

(c) `"MS", "VA", "AK", "LA", "CA", "AL", "GA", "TN", "WA", "DC"`

12.4 Analyze the quick sort algorithm to show the worst case time is $O(n^2)$.

12.5 Analyze the `mergeVirtualSeq()` function and show that it is a linear time operation in the worst case.

12.6 Analyze the linked list version of the merge sort algorithm to show the worst case time is $O(n \log n)$.

12.7 An important property of sorting algorithms is stability. A sorting algorithm is ***stable*** if it preserves the original order of duplicate keys. Stability is important when sorting a collection that has already been sorted by a *primary key* that will now be sorted by a *secondary key*. For example, suppose we have a sequence of student records that have been sorted by name and now we want to sort the sequence by GPA. Since there can be many duplicate GPAs, we want to order any duplicates by name. Thus, if Smith and Green both have the same GPA, then Green would be listed before Smith. If the sorting algorithm used for this second sort is stable, then the proper ordering can be achieved since Green would appear before Smith in the original sequence.

(a) Determine which of the comparison sorts presented in this chapter and in Chapter 5 are stable sorts.

(b) For any of the algorithms that are not stable, provide a sequence containing some duplicate keys that shows the order of the duplicates is not preserved.

Programming Projects

12.1 Implement the `addToSortedList()` function for use with the linked list version of the insertion sort algorithm.

12.2 Create a linked list version of the indicated algorithm.

(a) bubble sort
(b) selection sort

12.3 Create a new version of the quick sort algorithm that chooses a different key as the pivot instead of the first element.

(a) select the middle element
(b) select the last element

12.4 Write a program to read a list of grade point averages (0.0 – 4.0) from a text file and sort them in descending order. Select the most efficient sorting algorithm for your program.

12.5 Some algorithms are too complex to analyze using simple big-O notation or a representative data set may not be easily identifiable. In these cases, we must actually execute and test the algorithms on different sized data sets and compare the results. Special care must be taken to be fair in the actual implementation and execution of the different algorithms. This is known as an ***empirical analysis***. We can also use an empirical analysis to verify and compare the time-complexities of a family of algorithms such as those for searching or sorting.

Design and implement a program to evaluate the efficiency of the comparison sorts used with sequences by performing an empirical analysis using random numbers. Your program should:

- Prompt the user for the size of the sequence: n.
- Generate a random list of n values (integers) from the range $[0 \ldots 4n]$.
- Sort the original list using each of the sorting algorithms, keeping track of the number of comparisons performed by each algorithm.
- Compute the average number of comparisons for each algorithm and then report the results.

When performing the empirical analysis on a family of algorithms, it is important that you use the same original sequence for each algorithm. Thus, instead of sorting the original sequence, you must make a duplicate copy of the original and sort that sequence in order to preserve the original for use with each algorithm.