the recursive version is terribly inefficient. In that case, avoid it, unless of course you can't come up with an iterative algorithm. As you'll see later in the chapter, sometimes there just isn't a good solution.

13.3 Sorting Algorithms

The sorting problem provides a nice test bed for the algorithm design techniques we have been discussing. Remember, the basic sorting problem is to take a list and rearrange it so that the values are in increasing (actually, nondecreasing) order.

13.3.1 Naive Sorting: Selection Sort

Let's start with a simple "be the computer" approach to sorting. Suppose you have a stack of index cards, each with a number on it. The stack has been shuffled, and you need to put the cards back in order. How would you accomplish this task?

There are any number of good systematic approaches. One simple method is to look through the deck to find the smallest value and then place that value at the front of the stack (or perhaps in a separate stack). Then you could go through and find the smallest of the remaining cards and put it next in line, etc. Of course, this means that you'll also need an algorithm for finding the smallest remaining value. You can use the same approach we used for finding the max of a list (see Chapter 7). As you go through, you keep track of the smallest value seen so far, updating that value whenever you find a smaller one.

The algorithm I just described is called *selection sort*. Basically, the algorithm consists of a loop and each time through the loop, we select the smallest of the remaining elements and move it into its proper position. Applying this idea to a list of n elements, we proceed by finding the smallest value in the list and putting it into the 0^{th} position. Then we find the smallest remaining value (from positions 1–(n-1)) and put it in position 1. Next, the smallest value from positions 2–(n-1) goes into position 2, etc. When we get to the end of the list, everything will be in its proper place.

There is one subtlety in implementing this algorithm. When we place a value into its proper position, we need to make sure that we do not accidentally lose the value that was originally stored in that position. For example, if the smallest item is in position 10, moving it into position 0 involves an assignment:

```
nums[0] = nums[10]
```

But this wipes out the value currently in `nums[0]`; it really needs to be moved to another location in the list. A simple way to save the value is to swap it with the one that we are moving. Using simultaneous assignment, the statement

```
nums[0], nums[10] = nums[10], nums[0]
```

places the value from position 10 at the front of the list, but preserves the original first value by stashing it into location 10.

Using this idea, it is a simple matter to write a selection sort in Python. I will use a variable called `bottom` to keep track of which position in the list we are currently filling, and the variable `mp` will be used to track the location of the smallest remaining value. The comments in this code explain this implementation of selection sort:

```
def selSort(nums):
    # sort nums into ascending order

    n = len(nums)

    # For each position in the list (except the very last)
    for bottom in range(n-1):
        # find the smallest item in nums[bottom]..nums[n-1]

        mp = bottom                  # bottom is smallest initially
        for i in range(bottom+1,n):  # look at each position
            if nums[i] < nums[mp]:   # this one is smaller
                mp = i               #     remember its index

        # swap smallest item to the bottom
        nums[bottom], nums[mp] = nums[mp], nums[bottom]
```

One thing to notice about this algorithm is the accumulator for finding the minimum value. Rather than actually storing the minimum seen so far, `mp` just remembers the position of the minimum. A new value is tested by comparing the item in position `i` to the item in position `mp`. You should also notice that `bottom` stops at the second-to-last item in the list. Once all of the items up to the last have been put in the proper place, the last item has to be the largest, so there is no need to bother looking at it.

The selection sort algorithm is easy to write and works well for moderately sized lists, but it is not a very efficient sorting algorithm. We'll come back and analyze it after we've developed another algorithm.

13.3.2 Divide and Conquer: Merge Sort

As discussed above, one technique that often works for developing efficient algorithms is the divide-and-conquer approach. Suppose a friend and I were working together trying to put our deck of cards in order. We could divide the problem up by splitting the deck of cards in half with one of us sorting each of the halves. Then we just need to figure out a way of combining the two sorted stacks.

The process of combining two sorted lists into a single sorted result is called *merging*. The basic outline of our divide-and-conquer algorithm, called *merge sort*, looks like this:

```
Algorithm: merge sort nums

split nums into two halves
sort the first half
sort the second half
merge the two sorted halves back into nums
```

The first step in the algorithm is simple; we can just use list slicing to handle that. The last step is to merge the lists together. If you think about it, merging is pretty simple. Let's go back to our card stack example to flesh out the details. Since our two stacks are sorted, each has its smallest value on top. Whichever of the top values is the smallest will be the first item in the merged list. Once the smaller value is removed, we can look at the tops of the stacks again, and whichever top card is smaller will be the next item in the list. We just continue this process of placing the smaller of the two top values into the big list until one of the stacks runs out. At that point, we finish out the list with the cards from the remaining stack.

Here is a Python implementation of the merge process. In this code, `lst1` and `lst2` are the smaller lists, and `lst3` is the larger list where the results are placed. In order for the merging process to work, the length of `lst3` must be equal to the sum of the lengths of `lst1` and `lst2`. You should be able to follow this code by studying the accompanying comments:

```
def merge(lst1, lst2, lst3):
    # merge sorted lists lst1 and lst2 into lst3

    # these indexes keep track of current position in each list
    i1, i2, i3 = 0, 0, 0  # all start at the front
```

```
    n1, n2 = len(lst1), len(lst2)

    # Loop while both lst1 and lst2 have more items
    while i1 < n1 and i2 < n2:
        if lst1[i1] < lst2[i2]:  # top of lst1 is smaller
            lst3[i3] = lst1[i1]  #  copy it into current spot in lst3
            i1 = i1 + 1
        else:                    # top of lst2 is smaller
            lst3[i3] = lst2[i2]  #  copy it into current spot in lst3
            i2 = i2 + 1
        i3 = i3 + 1              # item added to lst3, update position

    # Here either lst1 or lst2 is done. One of the following loops will
    # execute to finish up the merge.

    # Copy remaining items (if any) from lst1
    while i1 < n1:
        lst3[i3] = lst1[i1]
        i1 = i1 + 1
        i3 = i3 + 1
    # Copy remaining items (if any) from lst2
    while i2 < n2:
        lst3[i3] = lst2[i2]
        i2 = i2 + 1
        i3 = i3 + 1
```

OK, now we can slice a list into two, and if those lists are sorted, we know how to merge them back into a single list. But how are we going to sort the smaller lists? Well, let's think about it. We are trying to sort a list, and our algorithm requires us to sort two smaller lists. This sounds like a perfect place to use recursion. Maybe we can use `mergeSort` itself to sort the two lists. Let's go back to our recursion guidelines to develop a proper recursive algorithm.

In order for recursion to work, we need to find at least one base case that does not require a recursive call, and we also have to make sure that recursive calls are always made on smaller versions of the original problem. The recursion in our `mergeSort` will always occur on a list that is about half as large as the original, so the latter property is automatically met. Eventually, our lists will be very small, containing only a single item or not items at all. Fortunately, these lists are already sorted! Voilà, we have a base case. When the length of the list is less than two, we do nothing, leaving the list unchanged.

Given our analysis, we can update the merge sort algorithm to make it properly recursive:

```
if len(nums) > 1:
    split nums into two halves
    mergeSort the first half
    mergeSort the second half
    merge the two sorted halves back into nums
```

Then we can translate this algorithm directly into Python code:

```
def mergeSort(nums):
    # Put items of nums in ascending order
    n = len(nums)
    # Do nothing if nums contains 0 or 1 items
    if n > 1:
        # split into two sublists
        m = n // 2
        nums1, nums2 = nums[:m], nums[m:]
        # recursively sort each piece
        mergeSort(nums1)
        mergeSort(nums2)
        # merge the sorted pieces back into original list
        merge(nums1, nums2, nums)
```

You might try tracing this algorithm with a small list (say eight elements), just to convince yourself that it really works. In general, though, tracing through recursive algorithms can be tedious and often not very enlightening.

Recursion is closely related to mathematical induction, and it requires practice before it becomes comfortable. As long as you follow the rules and make sure that every recursive chain of calls eventually reaches a base case, your algorithms *will* work. You just have to trust and let go of the grungy details. Let Python worry about that for you!

13.3.3 Comparing Sorts

Now that we have developed two sorting algorithms, which one should we use? Before we actually try them out, let's do some analysis. As in the searching problem, the difficulty of sorting a list depends on the size of the list. We need to

figure out how many steps each of our sorting algorithms requires as a function of the size of the list to be sorted.

Take a look back at the algorithm for selection sort. Remember, this algorithm works by first finding the smallest item, then finding the smallest of the remaining items, and so on. Suppose we start with a list of size n. In order to find the smallest value, the algorithm has to inspect each of the n items. The next time around the outer loop, it has to find the smallest of the remaining $n-1$ items. The third time around, there are $n-2$ items of interest. This process continues until there is only one item left to place. Thus, the total number of iterations of the inner loop for the selection sort can be computed as the sum of a decreasing sequence.

$$n + (n-1) + (n-2) + (n-3) + \ldots + 1$$

In other words, the time required by selection sort to sort a list of n items is proportional to the sum of the first n whole numbers. There is a well-known formula for this sum, but even if you do not know the formula, it is easy to derive. If you add the first and last numbers in the series you get $n+1$. Adding the second and second-to-last values gives $(n-1)+2 = n+1$. If you keep pairing up the values working from the outside in, all of the pairs add to $n+1$. Since there are n numbers, there must be $\frac{n}{2}$ pairs. That means the sum of all the pairs is $\frac{n(n+1)}{2}$.

You can see that the final formula contains an n^2 term. That means that the number of steps in the algorithm is proportional to the square of the size of the list. If the size of the list doubles, the number of steps quadruples. If the size triples, it will take nine times as long to finish. Computer scientists call this a *quadratic* or n^2 algorithm.

Let's see how that compares to the merge sort algorithm. In the case of merge sort, we divided a list into two pieces and sorted the individual pieces before merging them together. The real work is done during the merge process when the values in the sublists are copied back into the original list.

Figure 13.3 depicts the merging process to sort the list `[3, 1, 4, 1, 5, 9, 2, 6]`. The dashed lines show how the original list is continually halved until each item is its own list with the values shown at the bottom. The single-item lists are then merged back up into the two-item lists to produce the values shown in the second level. The merging process continues up the diagram to produce the final sorted version of the list shown at the top.

The diagram makes analysis of the merge sort easy. Starting at the bottom level, we have to copy the n values into the second level. From the second to

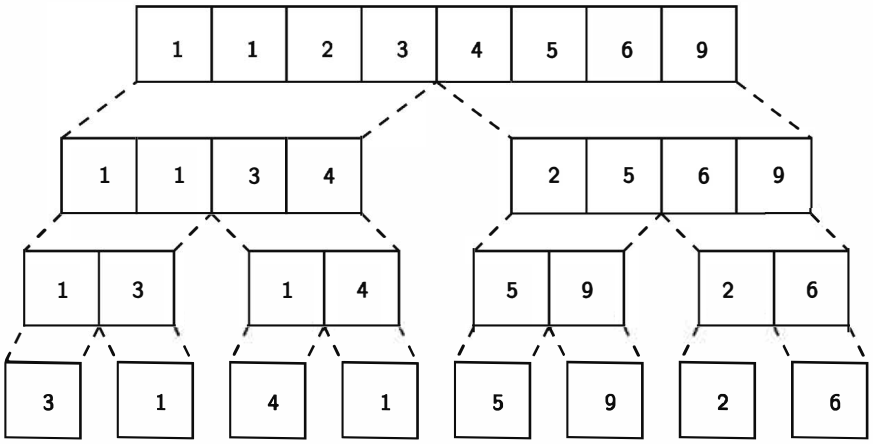

Figure 13.3: Merges required to sort [3, 1, 4, 1, 5, 9, 2, 6]

third level, the n values need to be copied again. Each level of merging involves copying n values. The only question left to answer is how many levels are there? This boils down to how many times a list of size n can be split in half. You already know from the analysis of binary search that this is just $\log_2 n$. Therefore, the total work required to sort n items is $n \log_2 n$. Computer scientists call this an *n log n* algorithm.

So which is going to be better, the n^2 selection sort or the $n \log n$ merge sort? If the input size is small, the selection sort might be a little faster because the code is simpler and there is less overhead. What happens, though, as n gets larger? We saw in the analysis of binary search that the log function grows *very* slowly ($\log_2 16,000,000 \approx 24$) so $n(\log_2 n)$ will grow much more slowly than $n(n)$.

Empirical testing of these two algorithms confirms this analysis. On my computer, selection sort beats merge sort on lists up to size about 50, which takes around 0.008 seconds. On larger lists, the merge sort dominates. Figure 13.4 shows a comparison of the time required to sort lists up to size 3000. You can see that the curve for selection sort veers rapidly upward (forming half of a parabola), while the merge sort curve looks almost straight (look at the bottom). For 3000 items, selection sort requires over 30 seconds while merge sort completes the task in about $\frac{3}{4}$ of a second. Merge sort can sort a list of 20,000 items in less than six seconds; selection sort takes around 20 minutes. That's quite a difference!

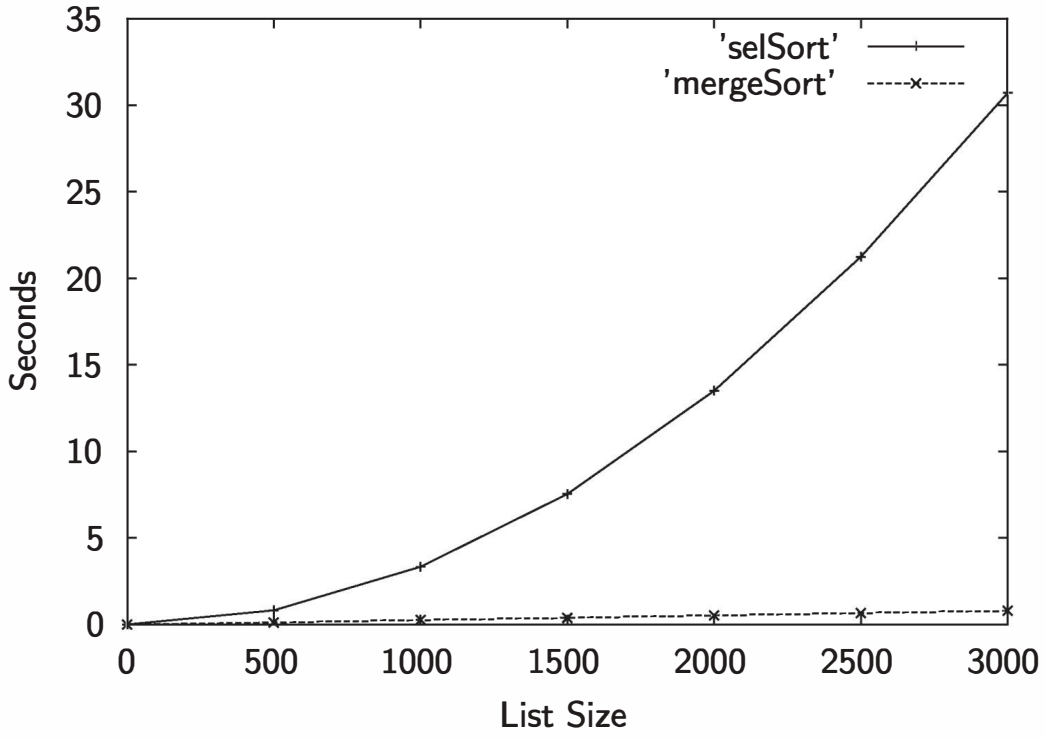

Figure 13.4: Experimental comparison of selection sort and merge sort

13.4 Hard Problems

Using our divide-and-conquer approach, we were able to design good algorithms for the searching and sorting problems. Divide and conquer and recursion are very powerful techniques for algorithm design. However, not all problems have efficient solutions.

13.4.1 Tower of Hanoi

One very elegant application of recursive problem solving is the solution to a mathematical puzzle usually called the Tower of Hanoi or Tower of Brahma. This puzzle is generally attributed to the French mathematician Édouard Lucas, who published an article about it in 1883. The legend surrounding the puzzle goes something like this:

Somewhere in a remote region of the world is a monastery of a very devout