CHAPTER 10

Organizing Data

> Search is an unsolved problem. We have a good 90 to 95% of the solution, but there is a lot to go in the remaining 10%.
>
> Marissa Mayer, President and CEO of Yahoo!
> *Los Angeles Times interview (2008)*

In this age of "big data," we take search algorithms for granted. Without web search algorithms that sift through billions of pages in a fraction of a second, the web would be practically useless. Similarly, large data repositories, such as those maintained by the U.S. Geological Survey (USGS) and the National Institutes of Health (NIH), would be useless without the ability to search for specific information. Even the operating systems on our personal computers now supply integrated search capabilities to help us navigate our increasingly large collections of files.

To enable fast access to these data, they must be organized in an efficient ***data structure***. Hidden data structures in the implementations of the list and dictionary abstract data types enable their methods to access and modify their contents quickly. (The data structure behind a dictionary was briefly explained in Tangent 7.2.) In this chapter, we will explore one of the simplest ways to organize data—maintaining it in a sorted list—and the benefits this can provide. We will begin by developing a significantly faster search algorithm that can take advantage of knowing that the data is sorted. Then we will design three algorithms to sort data in a list, effectively creating a sorted list data structure. If you continue to study computer science, you can look forward to seeing many more sophisticated data structures in the future that enable a wide variety of efficient algorithms.

10.1 BINARY SEARCH

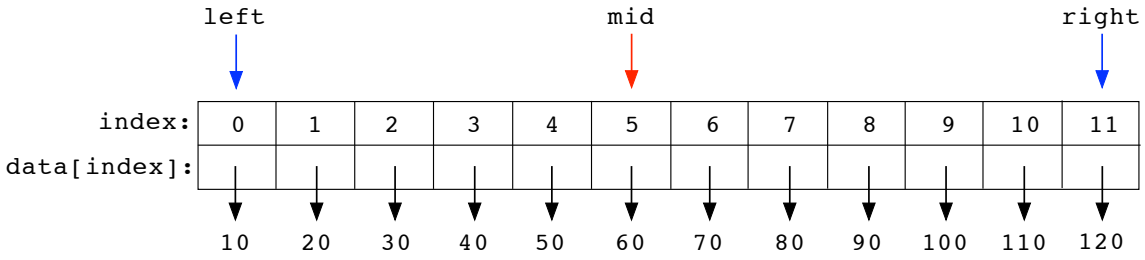
The spelling checkers that are built into most word processing programs work by searching through a list of English words, seeking a match. If the word is found, it is considered to be spelled correctly. Otherwise, it is assumed to be a misspelling. These word lists usually contain about a quarter million entries. If the words in the list are in no particular order, then we have no choice but to search through it one item at a time from the beginning, until either we happen to find the word we seek or we reach the end. We previously encountered this algorithm, called a *linear search* (or *sequential search*) because it searches in a linear fashion from beginning to end, and has linear time complexity.

Now let's consider the improvements we can make if the word list has been sorted in alphabetical order, as they always are. If we use a linear search on a sorted list, we know that we can abandon the search if we reach a word that is alphabetically after the word we seek. But we can do even better. Think about how we would search a physical, bound dictionary for the word "espresso." Since "E" is in the first half of the alphabet, we might begin by opening the book to a point about 1/4 of the way through. Suppose that, upon doing so, we find ourselves on a page containing words beginning with the letter "G." We would then flip backwards several pages, perhaps finding ourselves on a page on which the last word is "eagle." Next, we would flip a few pages forward, and so on, continuing to hone in on "espresso" until we find it.

Reflection 10.1 *How can we apply this idea to searching a sorted list?*

We can search a sorted list in a similar way, except that we usually do not know much about the distribution of the list's contents, so it is hard to make that first guess about where to start. In this case, the best strategy is to start in the middle. After comparing the target item to the middle item, we continue searching in the half of the list that must contain the target alphabetically. Because we are effectively dividing the list into two halves in each step, this algorithm is called *binary search*.

For example, suppose we wanted to search for the number 70 in the following sorted list of numbers. (We will use numbers instead of words in our example to save space.)



As we hone in on our target, we will update two variables named `left` and `right` to keep track of the first and last indices of the sublist that we are still considering. In addition, we will maintain a variable named `mid` that is assigned to the index of the middle value of this sublist. (When there are two middle values, we choose the

Tangent 10.1: Databases

A database is a structured file (or set of files) that contains a large amount of searchable data. The most common type of database, called a ***relational database***, stores its data in tables. Each row in a table has a unique *key* that can be used to search for that row. For example, the tables below represent a small portion of the earthquake data that we worked with in Section 7.4. The key in each table is underlined.

Earthquakes

QuakeID	Latitude	Longitude	Mag	NetID
nc72076126	40.1333	-123.863	1.8	NC
ak10812068	59.8905	-151.2392	2.5	AK
nc72076101	37.3242	-122.1015	1.8	NC
ci11369570	34.3278	-116.4663	1.2	CI
ci11369562	35.0418	-118.3227	1.4	CI
ci11369546	32.0487	-115.0075	3.2	CI

Networks

NetID	NetName
AK	Alaska Regional
CI	Southern California
NC	Northern California
US	US National
UW	Pacific Northwest

The table on the left contains information about individual earthquakes, each of which is identified with a `QuakeID`. The last column in the left table contains a two-letter `NetID` that identifies the preferred source of information about that earthquake. The table on the right contains the names associated with each `NetID`.

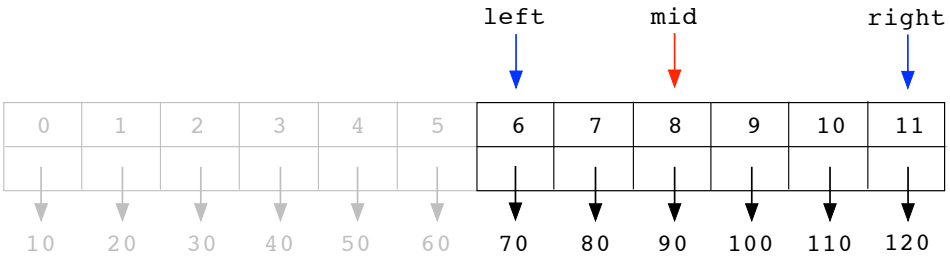
Relational databases are queried using a programming language called SQL. A simple SQL query looks like this:

```
select Mag from Earthquakes where QuakeID = 'nc72076101'
```

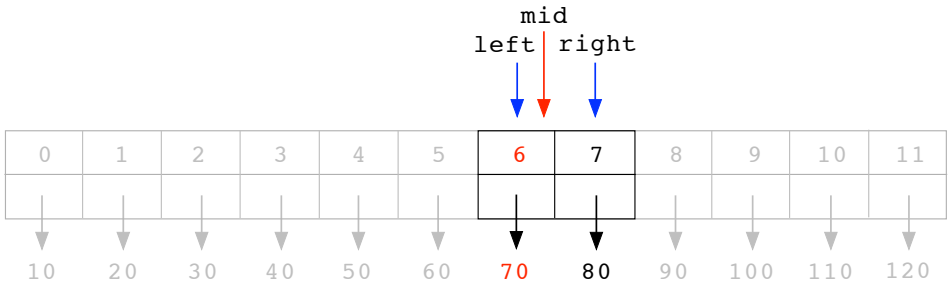
This query is asking for the magnitude (`Mag`), from the `Earthquakes` table, of the earthquake with `QuakeID nc72076101`. The response to this query would be the value `1.8`. Searching a table quickly for a particular key is facilitated by an *index*. An index is data structure that maps keys to rows in a table (similar to a Python dictionary). The keys in the index can be maintained in sorted order so that any key, and hence any row, can be found quickly using a binary search. (But database indices are more commonly maintained in a hash table or a specialized data structure called a B-tree.)

leftmost one.) In each step, we will compare the target item to the item at index `mid`. If the target is equal to this middle item, we return `mid`. Otherwise, we either set `right` to be `mid - 1` (to hone in on the left sublist) or we set `left` to be `mid + 1` (to hone in on the right sublist).

In the list above, we start by comparing the item at index `mid` (60) to our target item (70). Then, because $70 > 60$, we decide to narrow our search to the second half of the list. To do this, we assign `left` to `mid + 1`, which is the index of the item immediately after the middle item. In this case, we assign `left` to `5 + 1 = 6`, as shown below.



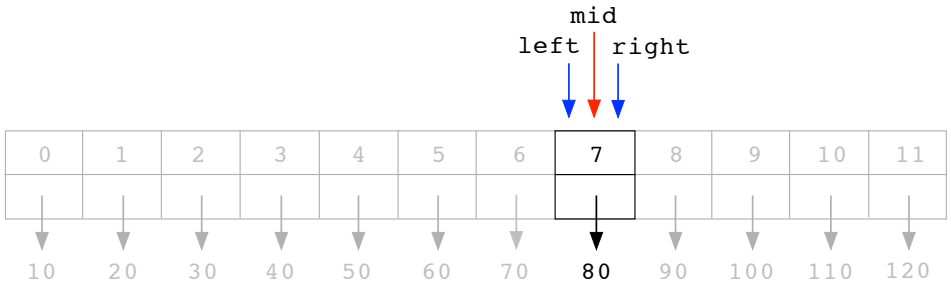
Then we update `mid` to be the index of the middle item in this sublist between `left` and `right`, in this case, 8. Next, since 70 is less than the new middle value, 90, we discard the second half of the sublist by assigning `right` to `mid - 1`, in this case, `8 - 1 = 7`, as shown below.



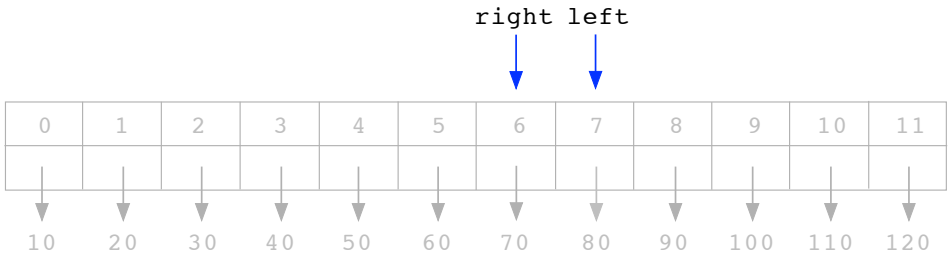
Then we update `mid` to be 6, the index of the "middle" item in this short sublist. Finally, since the item at index `mid` is the one we seek, we return the value of `mid`.

Reflection 10.2 *What would have happened if we were looking for a non-existent number like 72 instead?*

If we were looking for 72 instead of 70, all of the steps up to this point would have been the same, except that when we looked at the middle item in the last step, it would not have been equal to our target. Therefore, picking up from where we left off, we would notice that 72 is greater than our middle item 70, so we update `left` to be the index after `mid`, as shown below.



Now, since `left` and `right` are both equal to 7, `mid` must be assigned to 7 as well. Then, since 72 is less than the middle item, 80, we continue to blindly follow the algorithm by assigning `right` to be one less than `mid`.



At this point, since `right` is to the left of `left` (i.e., `left > right`), the sublist framed by `left` and `right` is empty! Therefore, 72 must not be in the list, and we return −1.

This description of the binary search algorithm can be translated into a Python function in a very straightforward way:

```
def binarySearch(keys, target):
    """Find the index of target in a sorted list of keys.

    Parameters:
        keys:   a sorted list of keys
        target: a key for which to search

    Return value: an index of target in keys or -1 if not found
    """

    left = 0
    right = len(keys) - 1
    while left <= right:
        mid = (left + right) // 2
        if target < keys[mid]:
            right = mid - 1
        elif target > keys[mid]:
            left = mid + 1
        else:
            return mid
    return -1
```

Notice that we have named our list parameter `keys` (instead of the usual `data`) because, in real databases (see Tangent 10.1), we typically try to match a unique *key* associated with the item we are seeking. For example, if we search for "Cumberbatch" in a phone directory, we are looking for a directory entry in which the last name (the key) matches "Cumberbatch;" we are not expecting the entire directory entry to match "Cumberbatch." When the search term is found, we return the entire directory entry that corresponds to this key. In our function, we return the index at which the key was found which, if we had data associated with the key, might provide us with enough information to find it in an associated data structure. We will look at an example of this in Section 10.2.

List length n	Worst case comparisons c
1	1
2	2
4	3
8	4
16	5
⋮	⋮
$2^{10} = 1{,}024$	11
⋮	⋮
$2^{20} \approx 1$ million	21
⋮	⋮
$2^{30} \approx 1$ billion	31

Table 10.1 The worst case number of comparisons in a binary search.

Reflection 10.3 *Write a* `main` *function that calls* `binarySearch` *with the list that we used in our example. Search for* `70` *and* `72`*.*

Reflection 10.4 *Insert a statement in the* `binarySearch` *function, after* `mid` *is assigned its value, that prints the values of* `left`*,* `right`*, and* `mid`*. Then search for more target values. Do you see why* `mid` *is assigned to the printed values?*

Efficiency of iterative binary search

How much better is binary search than linear search? When we analyzed the linear search in Section 6.7, we counted the worst case number of comparisons between the target and a list item, so let's perform the same analysis for binary search. Since the binary search contains a `while` loop, we will need to think more carefully this time about when the worst case happens.

Reflection 10.5 *Under what circumstances will the binary search algorithm perform the most comparisons between the target and a list item?*

In the worst case, the `while` loop will never execute `return mid`, instead iterating until `left > right`, rendering the `while` loop condition `False`. This happens when `target` is not found in the list.

Suppose we have a very short list with length $n = 4$. In the worst case, we first look at the item in the middle of this list, and then are faced with searching a sublist with length 2. Next, we look at the middle item of this sublist and, upon not finding the item, search a sublist of length 1. After one final comparison to this single item, the algorithm will return `-1`. So we needed a total of 3 comparisons for a list of length 4.

Reflection 10.6 *Now what happens if we double the size of the list to* $n = 8$*?*

After we compare the middle item in a list with length $n = 8$ to our target, we

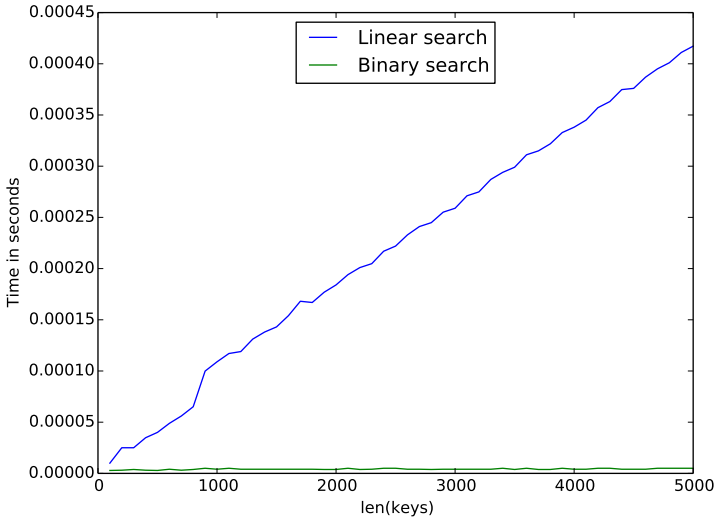


Figure 10.1 Execution times of linear search and binary search on small sorted lists.

are left with a sublist with length 4. We already know that a list with length 4 requires 3 comparisons in the worst case, so a list with length 8 must require $3+1=4$ comparisons in the worst case. Similarly, a list with length 16 must require only one more comparison than a list with length 8, for a total of 5. And so on. This pattern is summarized in Table 10.1. Notice that a list with over a billion items requires at most 31 comparisons!

Reflection 10.7 *In general, for list of length n, how many comparisons are necessary in the worst case?*

In each row of the table, the length of the list (n) is 2 raised to the power of 1 less than the number of comparisons (c), or

$$n = 2^{c-1}.$$

Therefore, for a list of size n, the binary search requires

$$c = \log_2 n + 1$$

comparisons in the worst case. So binary search is a ***logarithmic-time algorithm*** or, equivalently, an algorithm with $\mathcal{O}(\log n)$ time complexity. Since the time complexity of linear search is $\mathcal{O}(n)$, this means that linear search is *exponentially* slower than binary search.

This is a degree of speed-up that is *only* possible through algorithmic refinement; a faster computer simply cannot have this kind of impact. Figure 10.1 shows a comparison of actual running times of both search algorithms on some small lists. The time required by binary search is barely discernible as the red line parallel to the x-axis. But the real power of binary search becomes evident on very long lists.

As suggested by Table 10.1, a binary search takes almost no time at all, even on huge lists, whereas a linear search, which must potentially examine every item, can take a very long time.

A spelling checker

Now let's apply our binary search to the spelling checker problem. We will write a program that reads an alphabetized word list, and then allows someone to repeatedly enter a word to see if it is spelled correctly.

A list of English words can be found on computers running Mac OS X or Linux in the file `/usr/share/dict/words`, or one can be downloaded from the book website. This list is already sorted if you consider an uppercase letter to be equivalent to its lowercase counterpart. (For example, "academy" usually directly precedes "Acadia" in this file.) However, as we saw in Chapter 6, Python considers uppercase letters to come before lowercase letters, so we actually still need to sort the list to have it match Python's definition of "sorted." For now, we can use the `sort` method; in the coming sections, we will develop our own sorting algorithms.

The following function implements our spelling checker, using the `binarySearch` function (highlighted).

```
def spellcheck():
    """Repeatedly ask for a word to spell-check and print the result.

    Parameters: none

    Return value: None
    """

    dictFile = open('/usr/share/dict/words', 'r', encoding = 'utf-8')
    dictionaryWords = [ ]
    for word in dictFile:
        dictionaryWords.append(word[:-1])  # remove newline before append
    dictFile.close()
    dictionaryWords.sort()

    word = input('Enter a word to spell-check (q to quit): ')
    while word != 'q':
        index = binarySearch(dictionaryWords, word) # search for word
        if index != -1:                             # word was found
            print(word, 'is spelled correctly.')
        else:                                       # word was not found
            print(word, 'is not spelled correctly.')
        print()

        word = input('Enter a word to spell-check (q to quit): ')
```

The function begins by opening the word list file and reading each word (one word

per line) into a list. After all of the words have been read, we sort the list. Then a `while` loop repeatedly prompts for a word until the letter `q` is entered. Notice that we ask for a word before the `while` loop to initialize the value of `word`, and then again at the bottom of the loop to set up for the next iteration. In each iteration, we call the binary search function to check if the word is contained in the list. If the word is found (`index != -1`), we assume it is spelled correctly.

Reflection 10.8 *Combine the* `spellcheck` *function with the* `binarySearch` *function in a program. Run the program to try it out.*

Recursive binary search

You may have noticed that the binary search algorithm displays a high degree of self-similarity. In each step, the problem is reduced to solving a subproblem involving half of the original list. In particular, the problem of searching for a key between indices `left` and `right` is reduced to the subproblem of searching between `left` and `mid - 1`, or the subproblem of searching between `mid + 1` and `right`. Therefore, binary search is a natural candidate for a recursive algorithm. In the following function, we add `left` and `right` as parameters because they define the subproblem that is being solved.

```
def binarySearch(keys, target, left, right):
    """Recursively find the index of target in a sorted list of keys.

    Parameters:
        keys:   a sorted list of keys
        target: a value for which to search

    Return value: an index of target in keys or -1 if not found
    """

    if left > right:              # base case 1: not found
        return -1

    mid = (left + right) // 2
    if target == keys[mid]:       # base case 2: found
        return mid

    if target < keys[mid]:        # recursive cases
        return binarySearch(keys, target, left, mid - 1)  # left half
    else:
        return binarySearch(keys, target, mid + 1, right) # right half
```

Like the recursive linear search from Section 9.4, this function needs two base cases. In the first base case, when the list is empty (`left > right`), we return `-1`. In the second base case, if `target == keys[mid]`, we return `mid`. If neither of these cases holds, we solve one of the two subproblems recursively. If the `target` is less than the middle item, we recursively call the binary search with `right` set to `mid - 1`. Or, if

the `target` is greater than the middle item, we recursively call the binary search with `left` set to `mid + 1`.

Reflection 10.9 *Repeat Reflections 10.3 and 10.4 with the recursive binary search function. Does the recursive version "look at" the same values of* `mid`*?*

*Efficiency of recursive binary search

Like the iterative binary search, this algorithm has logarithmic, or $\mathcal{O}(\log n)$, time complexity. But, as with the recursive linear search, we have to derive this result differently using a recurrence relation. Let $T(n)$ denote the worst case number of comparisons between the target and a list item in a binary search when the length of the list is n. In the recursive `binarySearch` function, there are two such comparisons before reaching a recursive function call. As we did in Section 9.4, we will simply represent the number of comparisons before each recursive call with the constant c. When $n = 0$, we reach a base case with no recursive calls, so $T(0) = c$.

Reflection 10.10 *How many more comparisons are there in a recursive call to* `binarySearch`*?*

Since each recursive call divides the size of the list under consideration by (about) half, the size of the list we are passing into each recursive call is (about) $n/2$. Therefore, the number of comparisons in each recursive call must be $T(n/2)$. The total number of comparisons is then

$$T(n) = T(n/2) + c.$$

Now we can use the same substitution method that we used with recursive linear search to arrive at a closed form expression in terms of n. First, since $T(n) = T(n/2) + c$, we can substitute $T(n)$ with $T(n/2) + c$:

$$T(n)$$
$$\downarrow$$
$$T(n/2) + c$$

Now we need to replace $T(n/2)$ with something. Notice that $T(n/2)$ is just $T(n)$ with $n/2$ substituted for n. Therefore, using the definition of $T(n)$ above,

$$T(n/2) = T(n/2/2) + c = T(n/4) + c.$$

Similarly,

$$T(n/4) = T(n/4/2) + c = T(n/8) + c$$

and

$$T(n/8) = T(n/8/2) + c = T(n/16) + c.$$

This sequence of substitutions is illustrated in Figure 10.2. Notice that the denominator under the n at each step is a power of 2 whose exponent is the multiplier in

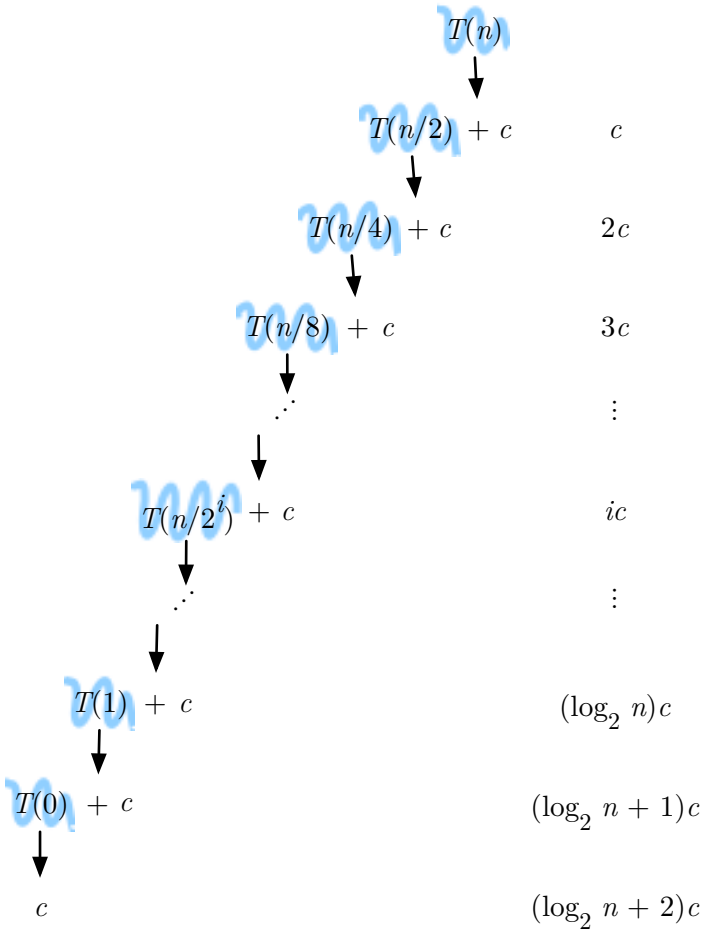


Figure 10.2 An illustration of how to derive a closed form for the recurrence relation $T(n) = T(n/2) + c$.

front of the accumulated c's at that step. In other words, for each denominator 2^i, the accumulated value on the right is $i \cdot c$. When we finally reach $T(1) = T(n/n)$, the denominator has become $n = 2^{\log_2 n}$, so $i = \log_2 n$ and the total on the right must be $(\log_2 n)c$. Finally, we know that $T(0) = c$, so the total number of comparisons is

$$T(n) = (\log_2 n + 2)\, c.$$

Therefore, as expected, the recursive binary search is a $\mathcal{O}(\log n)$ algorithm.

Exercises

10.1.1* Modify both of the binary search functions so that, when the target is not found, the functions also print the values in `keys` that would have been on either side of the target if it were in the list.

10.1.2* When the value of `target` is less than `keys[mid]` in the binary search algorithms, they next search the sublist between indices `left` and `mid - 1`. Would the algorithms still work if they searched between `left` and `mid` instead? Why or why not?

10.1.3. Similar to the previous exercise, suppose the binary search algorithms next searched the sublist between `mid` and `right` (instead of between `mid + 1` and `right`) when the `target` is greater than `keys[mid]`. Would the algorithms still work? Why or why not?

10.1.4. Write a function that takes the name of a text file as a parameter and returns the number of misspelled words in the file. Use the `wordTokens` function from Section 6.1 to get the list of words in the text file.

10.1.5. Write a function that takes three parameters—`minLength`, `maxLength`, and `step`—and produces a plot like Figure 10.1 comparing the worst case running times of binary search and linear search on lists with length `minLength`, `minLength + step`, `minLength + 2 * step`, ..., `maxLength`. Use a slice of the list derived from `list(range(maxLength))` as the sorted list for each length. To produce the worst case behavior of each algorithm, search for an item that is not in the list (e.g., –1).

10.1.6. The function below plays a guessing game against the pseudorandom number generator. What is the worst case number of guesses necessary for the function to win the game for any value of `n`, where `n` is a power of 2? Explain your answer.

```
import random
def guessingGame(n):
    secret = random.randrange(1, n + 1)
    left = 1
    right = n
    guessCount = 1
    guess = (left + right) // 2
    while guess != secret:
        if guess > secret:
            right = guess - 1
        else:
            left = guess + 1
        guessCount = guessCount + 1
        guess = (left + right) // 2
    return guessCount
```

10.2 SELECTION SORT

Sorting is a well-studied problem, and a wide variety of sorting algorithms have been designed, including the one used by the familiar `sort` method of the `list` class. In this section and the two that follow, we will develop and compare three other common algorithms, named *selection sort*, *insertion sort*, and *merge sort*.

Reflection 10.11 *Before you read further, think about how you would sort a list of items (names, numbers, books, socks, etc.) in some desired order. Write down your algorithm in pseudocode.*

The ***selection sort*** algorithm is so called because, in each step, it selects the next smallest value in the list and places it in its proper sorted position by swapping it with whatever is currently there. For example, consider the list of numbers `[50, 30, 40, 20, 10, 70, 60]`. To sort this list in ascending order, the selection sort algorithm first finds the smallest number, `10`. We want to place `10` in the first position in the list, so we swap it with the number that is currently in that position, `50`, resulting in the modified list

```
[10, 30, 40, 20, 50, 70, 60]
```

Next, we find the second smallest number, `20`, and swap it with the number in the second position, `30`:

```
[10, 20, 40, 30, 50, 70, 60]
```

Then we find the third smallest number, `30`, and swap it with the number in the third position, `40`:

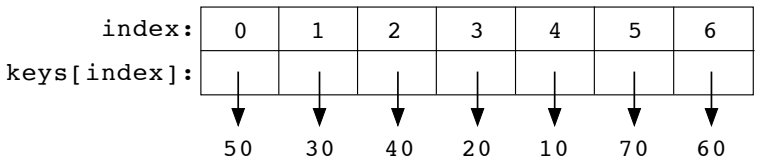
```
[10, 20, 30, 40, 50, 70, 60]
```

Next, we find the fourth smallest number, `40`. But since `40` is already in the fourth position, no swap is necessary. This process continues until we reach the end of the list.

Reflection 10.12 *Work through the remaining steps in the selection sort algorithm. What numbers are swapped in each step?*

Implementing selection sort

Let's look at how we can implement this algorithm, using a more detailed representation of this list which, as before, we will name `keys`:



To begin, we want to search for the smallest value in the list, and swap it with the value at index 0. We have actually already implemented both parts of this step. In Exercise 7.2.5, you may have written a function

```
swap(data, i, j)
```

that swaps the two values with indices `i` and `j` in the list `data`. To use this function to swap items, we will need the index of the smallest value. We already did this back on page 289 in the `minDay` function:

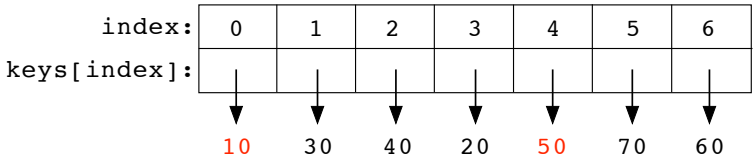
```
minIndex = 0
for index in range(1, len(keys)):
    if keys[index] < keys[minIndex]:
        minIndex = index
```

Once we have the index of the minimum value in `minIndex`, we can swap it with the item at index 0 with:

```
if minIndex != 0:
    swap(keys, 0, minIndex)
```

Reflection 10.13 *Why do we check if* `minIndex != 0` *before calling the* `swap` *function?*

In our example, these steps will find the smallest value, `10`, at index `4`, and then call `swap(keys, 0, 4)`. We first check if `minIndex != 0` so we do not needlessly swap the value in position 0 with itself. This swap results in the following modified list:



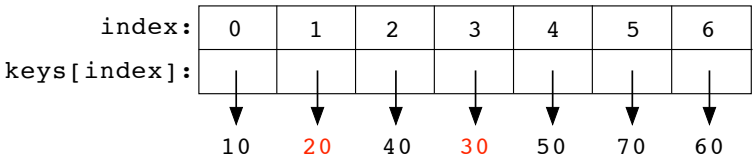
In the next step, we need to do the same thing, but for the second smallest value.

Reflection 10.14 *How we do we find the second smallest value in the list?*

Notice that, now that the smallest value is "out of the way" at the front of the list, the second smallest value in `keys` must be the smallest value in `keys[1:]`. Therefore, we can use exactly the same process as above, but on `keys[1:]` instead. This requires only four small changes in the code, marked in red below.

```
minIndex = 1
for index in range(2, n):
    if keys[index] < keys[minIndex]:
        minIndex = index
if minIndex != 1:
    swap(keys, 1, minIndex)
```

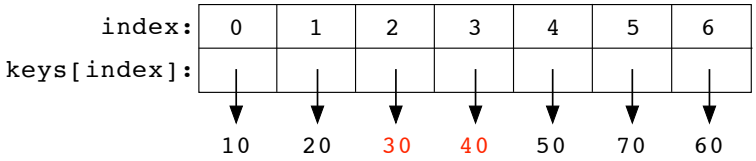
Instead of initializing `minIndex` to 0 and starting the `for` loop at 1, we initialize `minIndex` to 1 and start the `for` loop at 2. Then we swap the smallest value into position 1 instead of 0. In our example list, this will find the smallest value in `keys[1:]`, 20, at index 3. Then it will call `swap(keys, 1, 3)`, resulting in:



Similarly, the next step is to find the index of the smallest value starting at index 2, and then swap it with the value in index 2:

```
minIndex = 2
for index in range(3, n):
    if keys[index] < keys[minIndex]:
        minIndex = index
if minIndex != 2:
    swap(keys, 2, minIndex)
```

In our example list, this will find the smallest value in `keys[2:]`, 30, at index 3. Then it will call `swap(keys, 2, 3)`, resulting in:



We continue by repeating this sequence of steps, with increasing values of the numbers in red, until we reach the end of the list.

To implement this algorithm, we need to situate the loop above in another loop that iterates over the increasing values in red. We can do this by replacing the initial value assigned to `minIndex` in red with a variable named `start`:

```
minIndex = start
for index in range(start + 1, n):
    if keys[index] < keys[minIndex]:
        minIndex = index
if minIndex != start:
    swap(keys, start, minIndex)
```

Then we place these steps inside a `for` loop that has `start` take on all of the integers from `0` to `len(keys) - 2`:

```
1  def selectionSort(keys):
2      """Sort a list in ascending order using the selection sort algorithm.

3      Parameter:
4          keys: a list of keys

5      Return value: None
6      """

7      n = len(keys)
8      for start in range(n - 1):
9          minIndex = start
10         for index in range(start + 1, n):
11             if keys[index] < keys[minIndex]:
12                 minIndex = index
13         if minIndex != start:
14             swap(keys, start, minIndex)
```

Reflection 10.15 *In the outer* `for` *loop of the* `selectionSort` *function, why is the last value of* `start` *equal to* `n - 2` *instead of* `n - 1`*? Think about what steps would be executed if* `start` *were assigned the value* `n - 1` *in the last iteration of the loop.*

Reflection 10.16 *What would happen if we called* `selectionSort` *with the list* `['dog', 'cat', 'Monkey', 'Zebra', 'platypus', 'armadillo']`*? Would it work? If so, in what order would the words be sorted?*

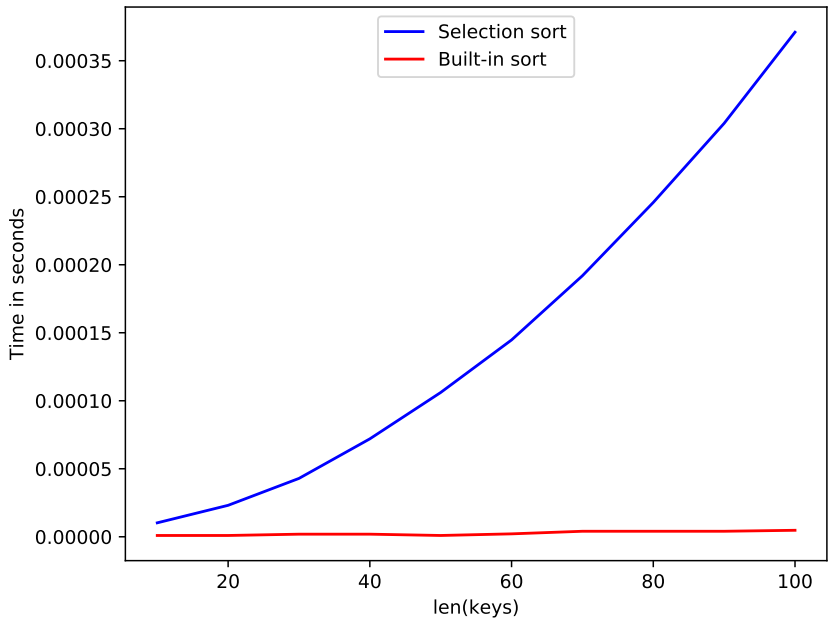


Figure 10.3 Execution times of selection sort and the list `sort` method on small randomly shuffled lists.

Because the comparison operators are defined for both numbers and strings, we can use our `selectionSort` function to sort either kind of data. For example, call `selectionSort` on each of the following lists, and then print the results. (Remember to incorporate the `swap` function from Exercise 7.2.5.)

```
numbers = [50, 30, 40, 20, 10, 70, 60]
animals = ['dog', 'cat', 'Monkey', 'Zebra', 'platypus', 'armadillo']
heights = [7.80, 6.42, 8.64, 7.83, 7.75, 8.99, 9.25, 8.95]
```

Efficiency of selection sort

Next, let's look at the time complexity of the selection sort algorithm. We can derive the asymptotic time complexity by counting the number of times the most frequently executed elementary step executes. In the `selectionSort` function, this is the comparison in the `if` statement on line 11. Since line 11 is in the body of the inner `for` loop that starts on line 10, it will be executed in every iteration of that loop. When `start` is 0, the inner `for` loop on line 10 runs from 1 to $n-1$, for a total of $n-1$ iterations. So line 11 is also executed $n-1$ times. Next, when `start` is 1, the inner `for` loop runs from 2 to $n-1$, a total of $n-2$ iterations, so line 11 is executed $n-2$ times. With each new value of `start`, there is one less iteration of the inner `for` loop. Therefore, the total number of times that line 11 is executed is

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

Where does this sum stop? To find out, we look at the last iteration of the outer `for` loop, when `start` is `n - 2`. In this case, the inner `for` loop runs from `n - 1` to `n - 1`, for only one iteration. So the total number of steps is

$$(n-1)+(n-2)+(n-3)+\cdots+3+2+1.$$

We have encountered this sum a few times before (see Tangent 4.1):

$$(n-1)+(n-2)+(n-3)+\cdots+3+2+1=\frac{n(n-1)}{2}=\frac{1}{2}n^2-\frac{1}{2}n.$$

Ignoring the 1/2 in front of n^2 and the low order term $(1/2)n$, we find that this expression is asymptotically $\mathcal{O}(n^2)$. So selection sort has quadratic time complexity.

Figure 10.3 shows the results of an experiment comparing the running time of selection sort to the `sort` method of the list class. (Exercise 10.3.3 asks you to replicate this experiment.) The parabolic blue curve in Figure 10.3 represents the quadratic time complexity of the selection sort algorithm. The red curve at the bottom of Figure 10.3 represents the running time of the `sort` method. Although this plot compares the algorithms with very small lists, on which both algorithms are very fast, we see a marked difference in the growth rates of the execution times. We will see why the `sort` method is so much faster in Section 10.4.

Querying data

Suppose we want to write a program that allows someone to query the USGS earthquake data that we worked with in Section 7.4. Although we did not use them then, each earthquake was identified by a unique key such as `ak10811825`. The first two characters identify the monitoring network (`ak` represents the Alaska Regional Network) and the last eight digits are a unique ID assigned by the network. Our program will search for a given key, and return the associated latitude, longitude, magnitude, and depth. This associated data is sometimes called ***satellite data*** because it revolves around the key.

To use the efficient binary search algorithm in our program, we need to first sort the data by its keys. When we read this data into memory, we can either read it into parallel lists, as we did in Section 7.4, or we can read it into a table (i.e., a list of lists), as we did in Section 8.1. In this section, we will modify our selection sort algorithm to handle the first option. We will leave the second option as an exercise.

We will read the data into two lists named `ids` and `data`, which are a list of keys and a list of tuples, respectively. Each tuple in the second list will contain the satellite data for one earthquake. By design, these two lists will be *parallel* in the sense that the satellite data in `data[index]` belongs to the earthquake with the key in `ids[index]`. When we sort the earthquakes' keys, we will need to make sure that their associations with the satellite data are maintained. In other words, if, during the sort of the list `ids`, we swap the values in `ids[9]` and `ids[4]`, we also need to swap `data[9]` and `data[4]`.

Modifying our selection sort algorithm in this way is actually quite simple. First, we will add a second parameter named `data` that contains the satellite data corresponding to `keys`. The function will still make all of its sorting decisions based on the list of `keys`. But when we swap two values in `keys`, we also swap the corresponding values in `data`. The modified function looks like this (with changes in red):

```
def selectionSort2(keys, data):
    """Sort parallel lists of keys and data values in ascending
       order using the selection sort algorithm.

    Parameters:
        keys: a list of keys
        data: a list of data values corresponding to the keys

    Return value: None
    """

    n = len(keys)
    for start in range(n - 1):
        minIndex = start
        for index in range(start + 1, n):
            if keys[index] < keys[minIndex]:
                minIndex = index
        swap(keys, start, minIndex)
        swap(data, start, minIndex)
```

Once we have the sorted parallel lists `ids` and `data`, we can use binary search to retrieve the index of a particular ID in the list `ids`, and then use that index to retrieve the corresponding satellite data from the list `data`. The following function implements this idea by repeatedly prompting for an earthquake ID.

```
def queryQuakes(ids, data):
    """ (docstring omitted) """

    key = input('Earthquake ID (q to quit): ')
    while key != 'q':
        index = binarySearch(ids, key, 0, len(ids) - 1)
        if index >= 0:
            print('Location: ' + str(data[index][:2]) + '\n' +
                  'Magnitude: ' + str(data[index][3]) + '\n' +
                  'Depth: ' + str(data[index][2]) + '\n')
        else:
            print('An earthquake with that ID was not found.')
        key = input('Earthquake ID (q to quit): ')
```

The `main` function below reads the earthquakes from the file (left as an exercise), sorts the data with our selection sort algorithm for parallel lists, and then calls `queryQuakes`.

```
def main():
    ids, data = readQuakes()    # left as an exercise
    selectionSort2(ids, data)
    queryQuakes(ids, data)
```

Exercises

10.2.1. Can you find a list of length 5 that requires more comparisons in `selectionSort` (on line 11) than another list of length 5? In general, with lists of length n, is there a worst case list and a best case list with respect to comparisons? How many comparisons do the best case and worst case lists require?

10.2.2. Now consider the number of swaps. Can you find a list of length 5 that requires more swaps (on line 14) than another list of length 5? In general, with lists of length n, is there a worst case list and a best case list with respect to swaps? How many swaps do the best case and worst case lists require?

10.2.3* The inner `for` loop of the selection sort function can be eliminated by using two built-in Python functions instead, as shown in the following alternative selection sort implementation.

```
def selectionSortAlt(keys):
    n = len(keys)
    for start in range(n - 1):
        minimum = min(keys[start:])
        minIndex = start + keys[start:].index(minimum)
        if minIndex != start:
            swap(keys, start, minIndex)
```

Is this function more or less efficient than the `selectionSort` function we developed? Explain.

10.2.4. Suppose we already have a list that is sorted in ascending order, and want to insert new values into it. Write a function that inserts an item into a sorted list, maintaining the sorted order, without re-sorting the list.

10.2.5* Write the function

```
readQuakes()
```

that is needed by the program at the end of this section. The function should read earthquake IDs and earthquake satellite data, consisting of latitude, longitude, depth and magnitude, from the data file on the web at

```
http://earthquake.usgs.gov/earthquakes/feed/v1.0/summary/2.5_month.csv
```

and return two parallel lists, as described on page 423. The satellite data for each earthquake should be stored as a tuple of floating point values. For example, the satellite data for a earthquake that occurred at 19.5223 degrees latitude and -155.5753 degrees longitude with magnitude 1.1 and depth 13.6 km should be stored in the tuple `(19.5223, -155.5753, 1.1, 13.6)`.

Use this function to complete a working version of the program on page 425. (Remember to incorporate the recursive binary search and the `swap` function from Exercise 7.2.5.) Look at the above URL in a web browser to find some earthquake IDs for which to search, or do the next exercise to have your program print a list.

10.2.6. Add to the `queryQuakes` function on page 424 the option to print an alphabetical list of all earthquakes, in response to typing `list` for the earthquake ID. The output should look something like this:

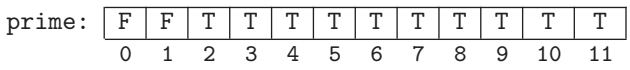
```
Earthquake ID (q to quit): ci37281696
Location: (33.4436667, -116.6743333)
Magnitude: 0.54
Depth: 13.69

Earthquake ID (q to quit): list
    ID             Location          Magnitude  Depth
----------  ---------------------  ---------  -----
ak11406701   (63.2397, -151.4564)         5.5    1.3
ak11406705   (58.9801, -152.9252)        69.2    2.3
ak11406708   (59.7555, -152.6543)        80.0    1.9
   ...

uw60913561   (41.8655, -119.6957)         0.2    2.4
uw60913616   (44.2917, -122.6705)         0.0    1.3
```

10.2.7. An alternative to storing the earthquake data in two parallel lists is to store it in one table (a list of lists). For example, the beginning of a table containing the earthquakes shown in the previous exercise would look like this:

```
[['ak11406701', 63.2397, -151.4564, 5.5, 1.3],
 ['ak11406705', 58.9801, -152.9252, 69.2, 2.3],
 ...
]
```

Rewrite the `readQuakes`, `selectionSort`, `binarySearch`, and `queryQuakes` functions so that they work with the earthquake data stored in this way instead. Your functions should assume that the key for each earthquake is in column 0. Combine your functions into a working program that is driven by a `main` function like the one on page 425.

10.2.8. The Sieve of Eratosthenes is a simple algorithm for generating prime numbers that has a structure that is similar to the nested loops in selection sort. The algorithm begins by initializing a list of n Boolean values named `prime` as follows. (In this case, $n = 12$.)



At the end of the algorithm, we want `prime[index]` to be `False` if `index` is not prime and `True` if `index` is prime. The algorithm continues by initializing a loop `index` variable to 2 (indicated by the arrow below) and then setting the list value of every multiple of 2 to be `False`.

Next, the loop index variable is incremented to 3 and, since `prime[3]` is `True`, the list value of every multiple of 3 is set to be `False`.

F	F	T	T	F	T	F	T	F	F	F	T
0	1	2	3	4	5	6	7	8	9	10	11
			↑								

Next, the loop index variable is incremented to 4. Since `prime[4]` is `False`, we do not need to set any of its multiples to `False`, so we do not do anything.

F	F	T	T	F	T	F	T	F	F	F	T
0	1	2	3	4	5	6	7	8	9	10	11
				↑							

This process continues with the loop index variable set to 5:

F	F	T	T	F	T	F	T	F	F	F	T
0	1	2	3	4	5	6	7	8	9	10	11
					↑						

And so on. How long must the algorithm continue to increment `index` before it has marked all non-prime numbers? Once it is done filling in the list, the algorithm iterates over it one more time to build the list of prime numbers, in this case, `[2, 3, 5, 7, 11]`. Write a function that implements this algorithm to return a list of all prime numbers less than or equal to a parameter `n`.

10.3 INSERTION SORT

Our second sorting algorithm, named *insertion sort*, is familiar to anyone who has sorted a hand of playing cards. Working left to right through our hand, the insertion sort algorithm inserts each card into its proper place with respect to the previously arranged cards. For example, consider our previous list, arranged as a hand of cards:

We start with the second card from the left, 30, and decide whether it should stay where it is or be inserted to the left of the first card. In this case, it should be inserted to the left of 50, resulting in the following slightly modified ordering:

Then we consider the third card from the left, 40. We see that 40 should be inserted between 30 and 50, resulting in the following order.

Next, we consider 20, and see that it should be inserted all the way to the left, before 30.

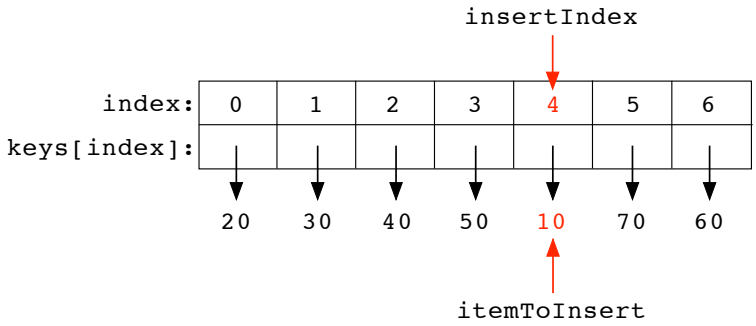
This process continues with 10, 70, and 60, at which time the hand will be sorted.

Implementing insertion sort

To implement this algorithm, we need to repeatedly find the correct location to insert an item among the items to the left, assuming that the items to the left are already sorted. Let's name the index of the item that we wish to insert `insertIndex` and the item itself `itemToInsert`. In other words, we assign

```
itemToInsert = keys[insertIndex]
```

To illustrate, suppose that `insertIndex` is 4 (and `itemToInsert` is 10), as shown below:



We need to compare `itemToInsert` to each of the items to the left, first at index `insertIndex - 1`, then at `insertIndex - 2`, `insertIndex - 3`, etc. When we come to an item that is less than or equal to `itemToInsert` or we reach the beginning of the list, we know that we have found the proper location for the item. This process can be expressed with a `while` loop:

```
index = insertIndex - 1
while index >= 0 and keys[index] > itemToInsert:
    index = index - 1
```

The variable `index` tracks which item we are currently comparing to `itemToInsert`. The value of `index` is decremented while it is still at least zero and the item at position `index` is still greater than `itemToInsert`. When the `while` loop ends, it is because either `index` has reached `-1` or `keys[index] <= itemToInsert`. In either case, we want to insert `itemToInsert` into position `index + 1`. In the example above, we would reach the beginning of the list, so we want to insert `itemToInsert` into position `index + 1` = 0.

To actually insert `itemToInsert` in its correct position, we need to delete `itemToInsert` from its current position, and insert it into position `index + 1`. One option is to use `pop` and `insert`:

```
keys.pop(insertIndex)
keys.insert(index + 1, itemToInsert)
```

In the insertion sort algorithm, we want to repeat this process for each value of `insertIndex`, starting at 1, so we enclose these steps in a `for` loop:

```
def insertionSort_Draft(keys):
    """ (docstring omitted) """

    n = len(keys)
    for insertIndex in range(1, n):
        itemToInsert = keys[insertIndex]
        index = insertIndex - 1
        while index >= 0 and keys[index] > itemToInsert:
            index = index - 1
        keys.pop(insertIndex)
        keys.insert(index + 1, itemToInsert)
```

Although this function is correct, it performs more work than necessary. To see why, think about how the `pop` and `insert` methods must work, based on the picture of the list on page 429. First, to delete (`pop`) `itemToInsert`, which is at position `insertIndex`, all of the items to the right, from position `insertIndex + 1` to position `n - 1`, must be shifted one position to the left. Then, to insert `itemToInsert` into position `index + 1`, all of the items to the right, from position `index + 2` to `n - 1`, must be shifted one position to the right. So the items from position `insertIndex + 1` to position `n - 1` are shifted twice, only to end up back where they started.

A more efficient algorithm only shifts those items that need to be shifted, and does so while we are already iterating over them. The following modified algorithm does just that.

```
1 def insertionSort(keys):
2     """ Sort a list in ascending order using the insertion sort algorithm.

3     Parameter:
4         keys: a list of keys to sort

5     Return value: None
6     """

7     n = len(keys)
8     for insertIndex in range(1, n):
9         itemToInsert = keys[insertIndex]
10        index = insertIndex - 1
11        while index >= 0 and keys[index] > itemToInsert:
12            keys[index + 1] = keys[index]
13            index = index - 1
14        keys[index + 1] = itemToInsert
```

The highlighted assignment statement on line 8 copies each item at position `index` one position to the right. Therefore, when we get to the end of the loop, position `index + 1` is available to store `itemToInsert`.

Reflection 10.17 *To get a better sense of how this works, carefully work through the steps with the three remaining items to be inserted in the illustration on page 429.*

Reflection 10.18 *Write a* `main` *function that calls the* `insertionSort` *function to sort the list from the beginning of this section:* `[50, 30, 40, 20, 10, 70, 60]`.

Efficiency of insertion sort

Is the insertion sort algorithm any more efficient than selection sort? To discover its time complexity, we first need to identify the most frequently executed elementary step(s). In this case, these appear to be the two assignment statements on lines 8–9 in the body of the `while` loop. However, the most frequently executed step is *actually* the test of the `while` loop condition on line 7 because the condition of a

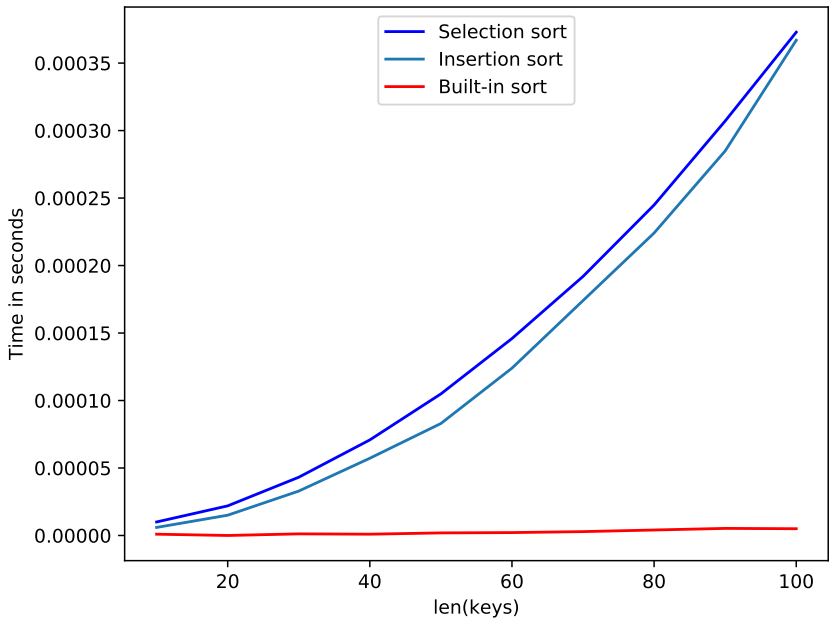


Figure 10.4 Execution times of selection sort, insertion sort, and the `sort` method on small randomly shuffled lists.

`while` loop is always tested when the loop is first reached, and again after each iteration. Therefore, a `while` loop condition is always tested once more than the body of the loop.

Reflection 10.19 *What are the minimum and maximum numbers of times the* `while` *loop condition is tested, for any particular value of* `insertIndex` *in the outer* `for` *loop?*

In the best case, it is possible that the condition is only tested once. This will happen if the item immediately to the left of `itemToInsert` is less than or equal to `itemToInsert`. Since there are $n-1$ iterations of the outer `for` loop, this means that the `while` loop condition could be tested only $n-1$ times in total for the entire algorithm. So, in the best case, insertion sort has linear-time, or $\mathcal{O}(n)$, time complexity.

In the worst case, the `while` loop will always iterate until `index >= 0` is `False`, i.e., until `index == -1`. This will happen if `keys` is initially in reverse order, meaning that `itemToInsert` is always less than every item to its left. Since `index` starts at `insertIndex - 1`, this will cause the `while` loop condition to be tested `insertIndex + 1` times. So in the first iteration of the outer `for` loop, when `insertIndex` is 1, the `while` loop condition is tested `insertIndex + 1` = 2 times. When `insertIndex` is 2, it is tested 3 times. This continues until `insertIndex` is `n - 1`, at which time, the `while` loop condition is tested `n` times. So the total number

of iterations of the `while` loop is $2 + 3 + 4 + \cdots + n$, which is the same as

$$(1 + 2 + 3 + \cdots + n) - 1 = \frac{n(n+1)}{2} - 1 = \frac{1}{2}n^2 + \frac{1}{2}n - 1.$$

Ignoring the constants and slower growing terms, this means that insertion sort is also a quadratic-time, or $\mathcal{O}(n^2)$, algorithm in the worst case.

Figure 10.4 shows the actual running times of the sorting algorithms we have studied so far on very small, randomly shuffled lists. We can see that the running time of insertion sort is almost identical to that of selection sort in practice. Both algorithms are still significantly slower than the built-in `sort` method. We will see why in the next section.

Exercises

10.3.1. Give examples of 10-element lists that require the best case and worst case numbers of comparisons in an insertion sort. How many comparisons are necessary to sort each of these lists?

10.3.2* Write a function that compares the time required to sort a long list of English words using insertion sort to the time required by the `sort` method of the list class. You can use the function `time.time()` function, which returns the number of seconds that have elapsed since January 1, 1970, to record the time required to execute each function. A list of English words can be found on computers running Mac OS X or Linux in the file `/usr/share/dict/words`, or one can be downloaded from the book website. This list is already sorted if you consider an uppercase letter to be equivalent to its lowercase counterpart. However, since Python considers uppercase letters to come before lowercase letters, the list is not really sorted for our purposes. But it is "almost" sorted, which means that insertion sort should perform relatively well. Be sure to make a separate copy of the original list for each sorting algorithm.

How many seconds did each sort require? (Be patient; insertion sort could take several minutes!) If you can be *really* patient, try timing selection sort as well.

10.3.3. Write a function

```
sortPlot(minLength, maxLength, step)
```

that produces a plot like Figure 10.4 comparing the running times of insertion sort, selection sort, and the `sort` method of the list class on shuffled lists with length `minLength`, `minLength + step`, `minLength + 2 * step`, ..., `maxLength`. At the beginning of your function, produce a shuffled list with length `maxLength` with

```
data = list(range(maxLength))
random.shuffle(data)
```

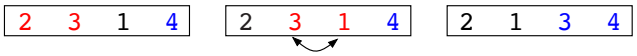
Then time each function for each list length using a new, unsorted slice of this list.

10.3.4* A sorting algorithm is called *stable* if two items with the same value always appear in the sorted list in the same order as they appeared in the original list. Are selection sort and insertion sort stable sorts? Explain your answer in each case.

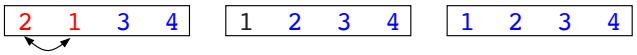
10.3.5. A third simple quadratic-time sorting algorithm is called *bubble sort* because it repeatedly "bubbles" large items toward the end of the list by swapping each item repeatedly with its neighbor to the right if it is larger than this neighbor.

For example, consider the short list `[3, 2, 4, 1]`. In the first pass over the list, bubble sort compares pairs of items, starting from the left, and swaps them if they are out of order. The illustration below depicts in red the items that are compared in the first pass, and the arrows depict which of those pairs are swapped because they are out of order.

At the end of the first pass, the largest item (in blue) is in its correct location. We repeat the process, but stop before the last item.



After the second pass, we know that the two largest items (in blue) are in their correct locations. On this short list, we make just one more pass.



After $n-1$ passes, we know that the last $n-1$ items are in their correct locations. Therefore, the first item must be also, and we are done. Write a function that implements this algorithm.

10.3.6. In the bubble sort algorithm, if no items are swapped during a pass over the list, the list must be in sorted order. The bubble sort algorithm can be made somewhat more efficient by detecting when this happens, and returning early if it does. Write a function that implements this modified bubble sort algorithm. (Hint: replace the outer `for` loop with a `while` loop and introduce a Boolean variable that controls the `while` loop.)

10.3.7. Write a modified version of the insertion sort function that sorts two parallel lists named `keys` and `data`, based on the values in `keys`, like the parallel list version of selection sort on page 424.

10.4 EFFICIENT SORTING

In the preceding sections, we developed two sorting algorithms, but discovered that they were both significantly less efficient than the built-in `sort` method. The `sort` method is based on a recursive sorting algorithm called ***merge sort***.[1]

Merge sort

As illustrated in Figure 10.5(a), merge sort is a *divide and conquer* algorithm, like those from Section 9.5. Divide and conquer algorithms generally consist of three steps:

[1]The Python sorting algorithm, called ***Timsort***, has elements of both merge sort and insertion sort. If you would like to learn more, visit `http://bugs.python.org/file4451/timsort.txt`.

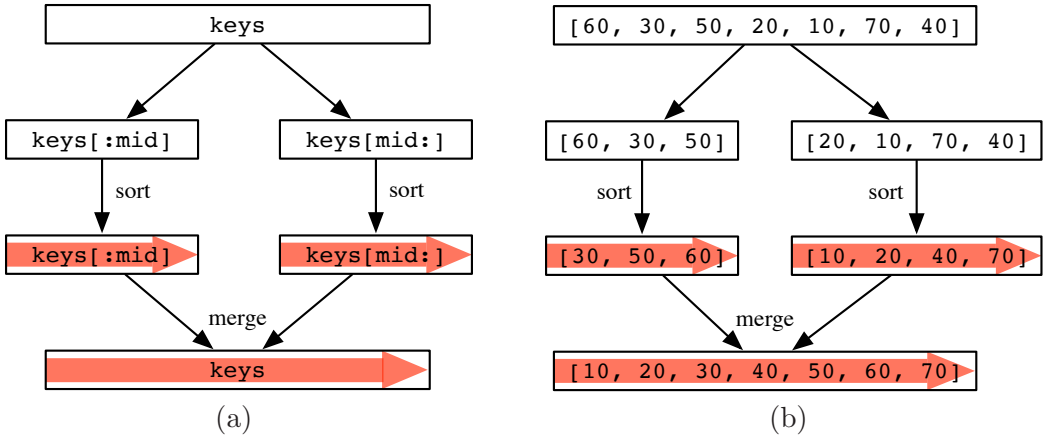


Figure 10.5 Illustrations of merge sort (a) in general and (b) on an example.

1. *Divide* the problem into two or more subproblems.
2. *Conquer* each subproblem recursively.
3. *Combine* the solutions to the subproblems into a solution for the original problem.

Reflection 10.20 *Based on Figure 10.5(a), what are the divide, conquer, and combine steps in the merge sort algorithm?*

The *divide* step of merge sort is very simple: just divide the list in half. The *conquer* step recursively calls the merge sort algorithm on the two halves. The *combine* step merges the two sorted halves into the final sorted list. This elegant algorithm is implemented by the following function:

```
def mergeSort(keys):
    """Sort a list in ascending order using the merge sort algorithm.

    Parameter:
        keys: a list of keys to sort

    Return value: None
    """

    n = len(keys)
    if n > 1:
        mid = n // 2              # divide list in half
        left = keys[:mid]
        right = keys[mid:]

        mergeSort(left)           # recursively sort the left half
        mergeSort(right)          # recursively sort the right half
        merge(left, right, keys)  # merge sorted halves into keys
```

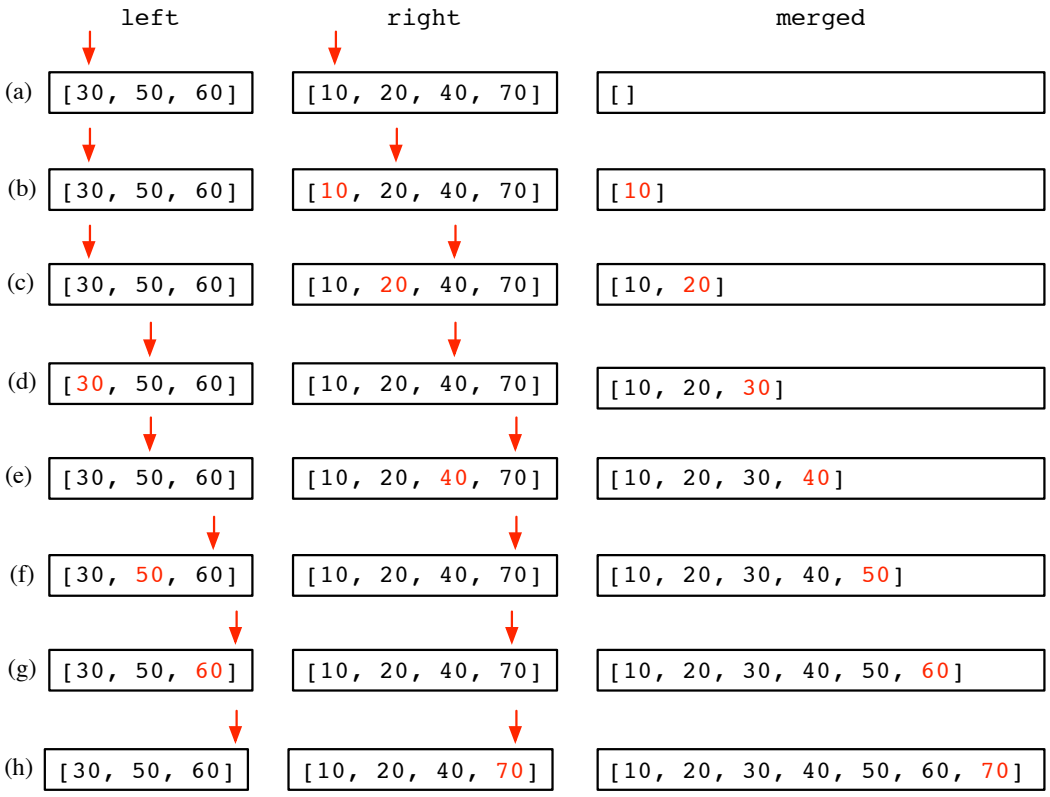


Figure 10.6 An illustration of the merge algorithm with example sublists.

Reflection 10.21 *Where is the base case in this function?*

The base case in this function is implicit; when `n <= 1`, the function just returns because a list containing zero or one values is, of course, already sorted.

To flesh out `mergeSort`, we need to implement the `merge` function. Suppose we want to sort the list `[60, 30, 50, 20, 10, 70, 40]`. As illustrated in Figure 10.5(b), the merge sort algorithm first divides this list into the two sublists `left = [60, 30, 50]` and `right = [20, 10, 70, 40]`. After recursively sorting each of these lists, we have `left = [30, 50, 60]` and `right = [10, 20, 40, 70]`. Now we want to efficiently merge these two sorted lists into one final sorted list. We could, of course, concatenate the two lists and then call merge sort with them. But that would be far too much work; we can do much better!

Because `left` and `right` are sorted, the first item in the merged list must be the minimum of the first item in `left` and the first item in `right`. So we place this minimum item into the first position in the merged list, and remove it from `left` or `right`. The next item in the merged list must again be at the front of `left` or `right`. This process continues until we run out of items in one of the lists.

This algorithm is illustrated in Figure 10.6. Rather than delete items from `left` and

`right` as we append them to the merged list, we will maintain an index for each list to remember the next item to consider. The red arrows in Figure 10.6 represent these indices which, as shown in step (a), start at the left side of each list. In steps (a)–(b), we compare the two front items in `left` and `right`, append the minimum (`10` from `right`) to the merged list, and advance the right index. In steps (b)–(c), we compare the first item in `left` to the second item in `right`, again append the minimum (`20` from `right`) to the merged list, and advance the right index. In steps (c)–(d), we compare the first item in `left` to the third item in `right`, append the minimum (this time, `30` from `left`), and advance the left index. This process continues until one of the indices exceeds the length of its list. In the example, this happens after step (g) when the left index is incremented past the end of `left`. At this point, we extend the merged list with whatever is left over in `right`, as shown in step (h).

Reflection 10.22 *Work through steps (a) through (h) on your own to make sure you understand how the merge algorithm works.*

This merge algorithm is implemented by the following function.

```
def merge(left, right, merged):
    """Merge two sorted lists, left and right, into one sorted list
       named merged.

    Parameters:
        left:   a sorted list
        right:  another sorted list
        merged: the merged sorted list

    Return value: None
    """

    merged.clear()   # clear contents of merged
    leftIndex = 0    # index in left
    rightIndex = 0   # index in right

    while leftIndex < len(left) and rightIndex < len(right):
        if left[leftIndex] <= right[rightIndex]: # left value is smaller
            merged.append(left[leftIndex])
            leftIndex = leftIndex + 1
        else:                                    # right value is smaller
            merged.append(right[rightIndex])
            rightIndex = rightIndex + 1

    if leftIndex >= len(left):              # remaining items are in right
        merged.extend(right[rightIndex:])
    else:                                   # remaining items are in left
        merged.extend(left[leftIndex:])
```

The `merge` function begins by clearing out the contents of the merged list and initializing the indices for the left and right lists to zero. The `while` loop starting

on line 13 constitutes the main part of the algorithm. The loop iterates while both `leftIndex` and `rightIndex` are valid indices in their respective lists. In lines 14–19, the algorithm compares the items at the two indices and appends the smallest to `merged`. When the loop finishes, we know that either `leftIndex >= len(left)` or `rightIndex >= len(right)`. In the first case (lines 20–21), there are still items remaining in `right` to append to `merged`. In the second case (lines 22–23), there are still items remaining in `left` to append to `merged`.

Reflection 10.23 *Write a program that uses the merge sort algorithm to sort the list in Figure 10.5(b).*

Internal vs. external sorting

We have been assuming all along that the data that we want to sort is small enough to fit in a list in a computer's memory. The selection and insertion sort algorithms must have the entire list in memory at once because they potentially pass over the entire list in each iteration of their outer loops. For this reason, they are called *internal sorting algorithms*.

But what if the data is larger than the few gigabytes that can fit in memory all at once? This is routinely the situation with real databases. In these cases, we need an ***external sorting algorithm***, one that can sort data in secondary storage by bringing smaller pieces of it into memory at a time.

The merge sort algorithm can be implemented as an external sorting algorithm. In the `merge` function, each of the sorted halves could reside in a file on disk and the algorithm could just bring the current front items into memory when it needs them. The merged list can also reside in a file on disk; when a new item is added to end of the merged result, it just needs to be written to the merged file after the previous item. Exercise 10.4.7 asks you to write a version of the `merge` function that merges two sorted files in this way.

Efficiency of merge sort

To formally derive the time complexity of merge sort, we would need to set up a recurrence relation like the one for the recursive binary search. But doing so for merge sort is a little more complicated, so let's look at it in a different way. Figure 10.7 illustrates the work that is done through the recursive calls in the algorithm. These recursive calls divide the list into smaller and smaller lists until they reach the base case, as illustrated in the top half of the diagram. The number of elementary steps performed when dividing each list through slicing is proportional to the length of the list, since every element is copied. In each of these "divide" levels, all n items are being copied once in a slicing operation. The number of slices gets larger as we work down toward the base case, and each one is smaller, but the total number of items remains constant at n. So there are $\mathcal{O}(n)$ elementary steps being done in each "divide" level.

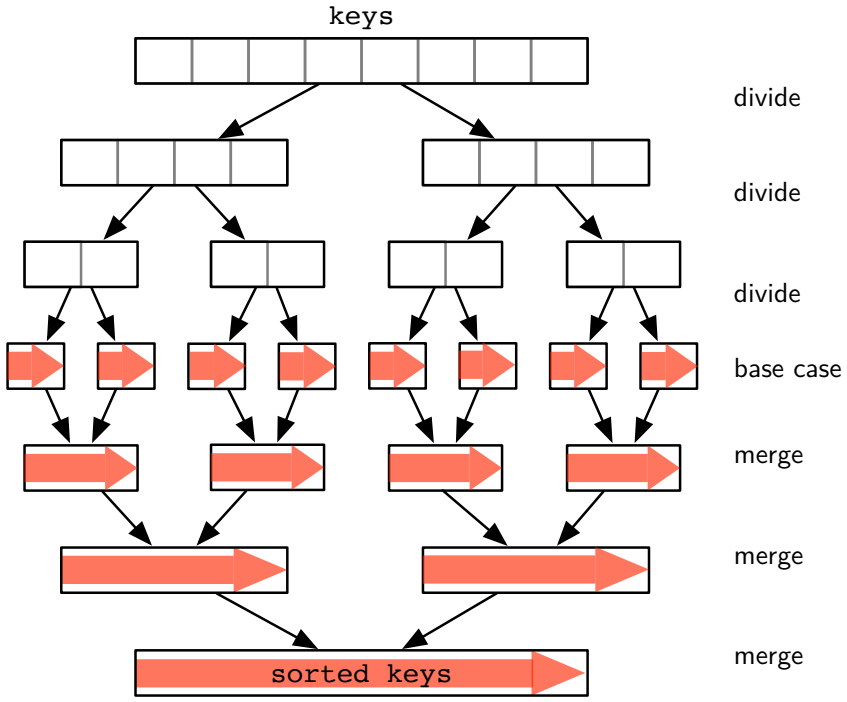


Figure 10.7 An illustration of the work performed by the merge sort algorithm.

Reflection 10.24 *How many "divide" levels are there until the base case is reached when $n = 8$? When $n = 16$? For n in general?*

When $n = 8$, as in the diagram, there are three "divide" levels. If n were doubled to 16, there would be just one more level needed. In general, because the lists are being halved at each level, until each list contains just one item, there must be $\log_2 n$ levels until the base case is reached. Therefore, the total number of elementary steps in the top half of the diagram is proportional to $n \cdot \log_2 n$, or $\mathcal{O}(n \log n)$.

Now let's analyze the number of elementary steps in the bottom half of the diagram. In the base case, each list contains at most one item, so they are sorted, as depicted by the red arrows. Then those short lists are merged into lists that are about twice as long. This merging continues until all of the original items are merged into the final sorted list. The number of "merge" levels in the diagram is the same as the number of "divide" levels because the same process is performed in reverse order. So the total number of elementary steps in the bottom half of the diagram is proportional to $\log_2 n$ times the number of elementary steps performed in each "merge" level.

Reflection 10.25 *About how many elementary steps does the* `merge` *function contain when* `merged` *contains 8 items? 16 items? n items?*

Since all of the items in the `left` and `right` lists are copied to the `merged` list exactly once, the total number of elementary steps in `merge` is proportional to the

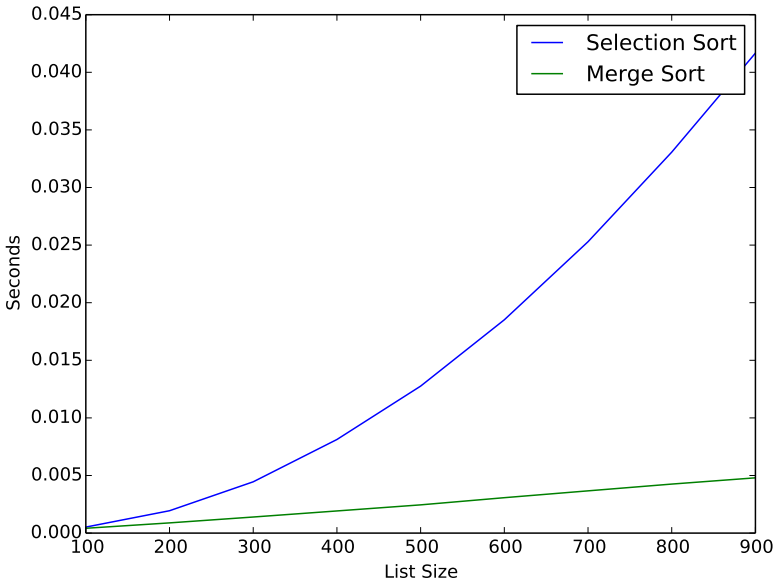


Figure 10.8 A comparison of the execution times of selection sort and merge sort on small randomly shuffled lists.

length of the `merged` list. In each "merge" level of the diagram, since all n items are involved in one merge operation, the combined lengths of the `merged` lists is n. So the total number of elementary steps in each "merge" level must be proportional to n. Therefore, the total number of elementary steps in the bottom half of the diagram is also proportional to $n \cdot \log_2 n$, or $\mathcal{O}(n \log n)$.

Adding the $\mathcal{O}(n \log n)$ from the top half of the diagram to the $\mathcal{O}(n \log n)$ from the bottom half gives us $\mathcal{O}(n \log n)$ elementary steps in total because the "big-oh" notation hides the constant coefficient that comes from this addition. So the time complexity of merge sort is $\mathcal{O}(n \log n)$.

How much faster is this than the quadratic-time selection and insertion sorts? Figure 10.8 illustrates the difference by comparing the merge sort and selection sort functions on small randomly shuffled lists. The merge sort algorithm is *much* faster. Recall that the algorithm behind the built-in `sort` method is based on merge sort, which explains why it was so much faster than our previous sorts. Exercise 10.4.1 asks you to compare the algorithms on much longer lists as well.

Exercises

10.4.1. Suppose the selection sort algorithm requires exactly n^2 steps and the merge sort algorithm requires exactly $n \log_2 n$ steps. About how many times slower is selection sort than merge sort when $n = 100$? $n = 1000$? $n = 1$ million?

10.4.2. Repeat Exercise 10.3.2 with the merge sort algorithm. How does the time required by the merge sort algorithm compare to that of the insertion sort algorithm and the built-in `sort` method?

10.4.3. Add merge sort to the running time plot in Exercise 10.3.3. How does its time compare to the other sorts?

10.4.4. Our `mergeSort` function is a stable sort, meaning that two items with the same value always appear in the sorted list in the same order as they appeared in the original list. However, if we changed the `<=` operator in line 18 of the `merge` function to a `<` operator, it would no longer be stable. Explain why.

10.4.5* We have seen that binary search is exponentially faster than linear search in the worst case. But is it always worthwhile to use binary search over linear search? The answer, as is commonly the case in the "real world," is "it depends." In this exercise, you will investigate this question. Suppose we have an *unordered* list of n items that we wish to search.

(a) If we use the linear search algorithm, what is the time complexity of this search?

(b) If we use the binary search algorithm, what is the time complexity of this search? (Think carefully about this.)

(c) If we perform n (where n is also the length of the list) individual searches of the list, what is the time complexity of the n searches together if we use the linear search algorithm?

(d) If we perform n individual searches with the binary search algorithm, what is the time complexity of the n searches together?

(e) What can you conclude about when it is best to use binary search vs. linear search?

10.4.6. Suppose we have a list of n keys that we anticipate needing to search k times. We have two options: either we sort the keys once and then perform all of the searches using a binary search algorithm or we forgo the sort and simply perform all of the searches using a linear search algorithm. Suppose the sorting algorithm requires exactly $n^2/2$ steps, the binary search algorithm requires $\log_2 n$ steps, and the linear search requires n steps. Assume each step takes the same amount of time.

(a) If the length of the list is $n = 1024$ and we perform $k = 100$ searches, which alternative is better?

(b) If the length of the list is $n = 1024$ and we perform $k = 500$ searches, which alternative is better?

(c) If the length of the list is $n = 1024$ and we perform $k = 1000$ searches, which alternative is better?

10.4.7. Write a function that merges two sorted files into one sorted file. Your function should take the names of the three files as parameters. Assume that all three files contain one string value per line. Your function should not use any lists, instead reading only one item at a time from each input file and writing one item

at a time to the output file. In other words, at any particular time, there should be at most one item from each file assigned to any variable in your function. You will know when you have reached the end of one of the input files when a call to `readline` returns an empty string. There are two files on the book website named `left.txt` and `right.txt` that you can use to test your function.

*10.5 TRACTABLE AND INTRACTABLE ALGORITHMS

This section is available on the book website.

10.6 SUMMARY AND FURTHER DISCOVERY

Sorting and searching are perhaps the most fundamental problems in computer science for good reason. We have seen how simply sorting a list can *exponentially* decrease the time it takes to search it, using the *binary search algorithm*. Since binary search is one of those algorithms that "naturally" exhibits self-similarity, we designed both iterative and recursive algorithms that implement the same idea. We also designed two basic sorting algorithms named *selection sort* and *insertion sort*. Each of these algorithms can sort a short list relatively quickly, but they are both very inefficient when it comes to larger lists. By comparison, the recursive *merge sort* algorithm is very fast. Merge sort has the added advantage of being an *external sorting algorithm*, meaning we can adapt it to sort very large data sets that cannot be brought into a computer's memory all at once.

Although the selection and insertion sort algorithms are quite inefficient compared to merge sort, they are still *tractable*, meaning that they will finish in a "reasonable" amount of time. In fact, all algorithms with time complexities that are polynomial functions of their input sizes are considered to be tractable. On the other hand, exponential-time algorithms are called *intractable* because even when their input sizes are relatively small, they require eons to finish.

Notes for further discovery

This chapter's epigraph is from an interview given by Marissa Mayer to the *Los Angeles Times* in 2008 [21].

The subjects of this chapter are fundamental topics in second-semester computer science courses, and there are many books available that cover them in more detail. A higher-level overview of some of the tricks used to make searching fast can be found in John MacCormick's *Nine Algorithms that Changed the Future* [37].

*10.7 PROJECTS

This section is available on the book website.