Answers to End of Section and Review Exercises for Chapter 3

Exercises 3.1

1. Here is the code:

```
problemSize = int(input("Enter the problem size: "))
count = 0
while problemSize > 0:
    problemSize = problemSize / 2
    count += 1
print(count)
```

2. When the problem size doubles, the number of iterations increases by 1. When the problem increases by a factor of 10, the number of iterations increases by 4.
3. The answer depends on what is found in the documentation.

Exercises 3.2

1. a. 2^n, O(n)
   b. $3n^2$, O(n^2)
   c. n^3, O(n^3)
2. Algorithm A does more work on all problem sizes.
3. When n is 16, then n^4 and 2^n are the same: 65,536. When n is 17, n^4 is 83,521 and 2^n is 131,072.

Exercises 3.3

1. Trace of first run:

```
left right midpoint
  0    9      4
  5    9      7
  8    9      8
```

   Trace of second run:

```
left right midpoint
  0    9      4
  0    3      1
```

2. Assume, somewhat unrealistically, that the names are distributed evenly through the list (as many beginning with “A” as with “B,” and so forth) and that there are

thousands of names in the list. Then, on the first pass of the search, the midpoint will be a function of the size of the list and the ordinal value of the target name's first letter. The first pass will thus eliminate many more elements than before, and the other passes, if they are necessary, will have smaller search spaces as well. However, the modified algorithm is still closer to O(lg *n*) than to O(1) in the worst case.

Exercises 3.4

1. A sorted list causes the fewest exchanges in selection sort (0). A list with the largest element as the first one and the rest in ascending order causes the greatest number of exchanges ($n - 1$).
2. The worst-case number of exchanges in selection sort ($n - 1$) does not predominate, so there is little impact on performance. The worst-case number of exchanges in bubble sort (approximately n^2) is similar to the worst-case complexity, so there is significant extra work. The size of the elements has no impact, because references only are exchanged.
3. The configurations of the list that cause the sort to terminate early are relatively rare, so on most of the cases, the outer loop does not terminate early.
4. In partially sorted lists, the insertion points for elements are found more quickly in the inner loop, thus reducing the amount of work done.

Exercises 3.5

1. The strategy of quicksort begins by selecting a pivot element from the current sublist. The algorithm then rearranges the other elements around the pivot, such that the smaller ones are to its left and the larger ones are to its right. After this partition has been accomplished, the algorithm is run recursively on the sublists to the right and left of the pivot. The algorithm halts when the length of its sublist is 1. The total number of comparisons at each level of partitioning is approximately *n*. In the best case, there are $\log_2 n$ levels of partitions, so the best-case complexity of quicksort is O($n \log n$).
2. Quicksort is not O($n \log n$) in all cases. In the worst case, all of the partitions are so uneven that the complexity degrades to O(n^2). For example, this occurs when the list is already sorted and the pivot is always the first element.
3. Selecting an element at random or selecting the median element of the first three elements are two alternative ways of selecting a pivot.
4. Insertion sort performs better as its list is more thoroughly sorted. Because quicksort has already shifted elements into positions that are close to where they should be, running insertion sort on the medium-sized sublists might finish the job faster than continuing quicksort all the way down.
5. Merge sort is O($n \log n$) in the worst case because the subdivisions during recursive descent are always even.

Answers to Review Questions

1. b
2. a
3. b
4. b
5. b
6. a
7. b
8. b
9. b
10. a