Introduction to Computer Science with Java

Chapter 20 – Recursive Sorting and Algorithmic Complexity

In this chapter we will learn about two recursive sorting algorithms, quick sort and merge sort, which are significantly more efficient than the iterative sorting algorithms we saw in the last chapter. We will also examine the algorithmic complexity of sorting techniques, and algorithmic complexity in general. We will see how to conduct experiments measuring the efficiency of sorting algorithms.

Chapter Learning Outcomes

Upon completion of this chapter students should be able to:

- describe the concepts of *a priori* and *a posteriori* perspectives in analyzing the performance of algorithms.
- describe how the quicksort algorithm works and create Java software that implements the algorithm to sort an array.
- describe how the merge sort algorithm works and create Java software that implements the algorithm to sort an array.
- describe the concepts of temporal complexity and spatial complexity of an algorithm, and what a recurrence relation is and why it is important in analyzing temporal and spatial complexity.
- describe asymptotic notation, including big O notation, big omega notation, and big theta notation, and how they are used to describe algorithms.
- describe the concepts of recursion overhead, linear recursion, and logarithmic recursion, and how the overhead for linear recursion compares to the overhead for logarithmic recursion.
- describe the concept of software benchmarking and how it is used to analyze algorithms.
- Conduct benchmarking to measure the actual performance of given algorithms.

20.1 Putting the Science in Computer Science

In this chapter, we will look at the performance of algorithms from both *a priori* and a *posteriori* perspectives. The terms *A priori* and *a posteriori* come from the Latin for *before* and *after*. Used in reference to knowledge, a priori generally refers to “prior knowledge” or “knowledge before the fact”, while *a posteriori* refers to “knowledge after the fact”.

A priori knowledge is knowledge of an event based on thought, particularly logic and theory. It is what we can hypothesize or imagine about an event before we actually experience it. **A posteriori** knowledge is based on experiencing an event; it is empirical knowledge based on practice, experimentation and research. Of course the two are related and one depends on the other; prior knowledge is based on prior experience, while understanding something we experience is based on thought and logic. In computer science, *a priori* knowledge of an algorithm means theorizing how the algorithm will perform even before we run the algorithm, while *a posteriori* knowledge depends on conducting experiments to study how an algorithm actually performs.

Consider the following algorithm:

```
nested for loops

for( int row  =0; row  < 5; row ++)  {

   for( int col =0; col < 10; col++)
      print "*";

   println();

} // end for row
```

How many stars (asterisks) do you think this algorithm will print if we write and run a proper Java program based on the algorithm? The answer is 50. The outer loop – the *row* loop – executes five times, and each time the outer loop executes, the inner loop – the *col* loop – executes 10 times. Based on the pseudocode above, we can reckon, or calculate, that that there will be five rows with 10 stars in each row. Five times ten is fifty.

We can figure out all of this before ever running the algorithm, based on logic and our prior knowledge of programming (and mathematics). This is an *a priori* analysis of an algorithm. This is what people do all the time. When we make decisions, we are trying to figure out what will happen before it actually happens and base our decisions on that knowledge. Which candidate should get my vote? Should I fly or drive? Pasta or chicken? Is this the one for me?

After we run the algorithm and see how it actually works, we will have *a posteriori* knowledge of its performance. This is the value of experience. Our logic could be incorrect, incomplete, or flawed in some other way. It is only theoretical. Our experience should prove or disprove our theory.

Prior knowledge leads to ideas or theories, which are refined through experience. This is the scientific process. Develop a theory based on prior experience and logic, test the theory through new experiences, and then refine the theory.

In this chapter we will do this with sorting algorithms. We will put the *science* in computer science.

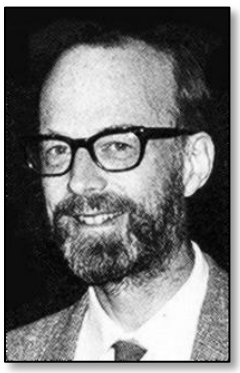
Of course, you have been doing this since the first day of class. You have been developing your own ideas about how an algorithm should work, designing code based on these ideas, and then experiencing what happens when you try to run the code. Some of your theories have been incorrect or incomplete. Hopefully there have been no disasters, but as you have learned computer programming, you have, from time to time,

probably experienced code that produces unexpected results. You have had to debug your programs, at least some of the time.

Yet, if you look at the big picture, most of your ideas about programming have been correct. Even when it can take hours or the help of someone else to get a program to run, most of your ideas about what will happen when you run the code have been correct. Usually there are only a few minor things that cause errors when you try to run a program, and even some of these are due more to the way you type than to the way you think. You have been working as a computer scientist – theorizing and experimenting with algorithms – since you started programming, and you have been getting better at it as time goes by.

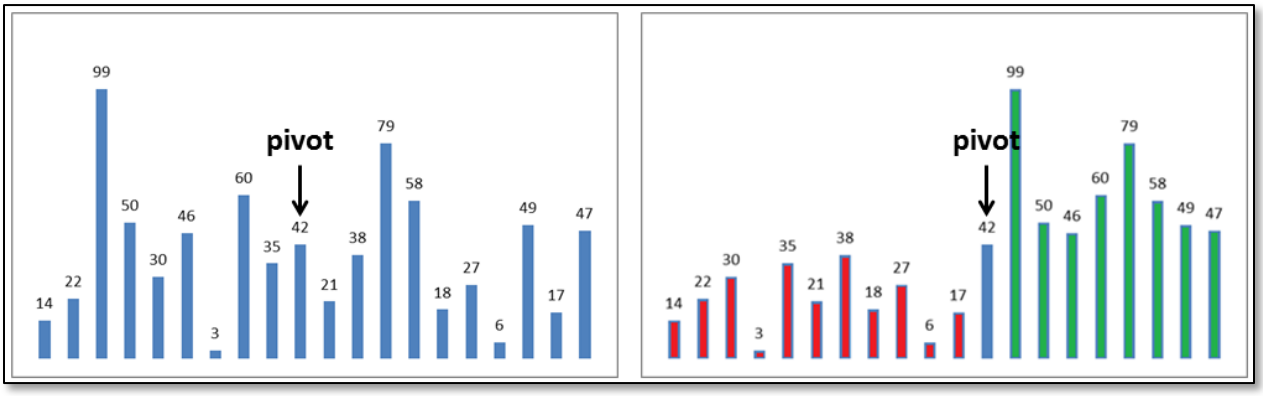
So, if we have been theorizing and experimenting all along, then how is this chapter different? In this chapter we will apply the scientific process to questions about the complexity of algorithms. What is the relationship between the amount of time it takes a program to run and the size of the data set the program is processing? What is the relationship between the amount of space a program uses and the size of the data set it is processing? We will examine, in theory and through experimentation, the temporal and spatial complexity of algorithms. But first, we will learn about recursive sorting methods, quick sort and merge sort, so we can compare these to the iterative sorting methods we learned in the last chapter.

20.2 Quicksort



The *quick sort* algorithm, also known by the one word term *quicksort,* was first developed by Tony Hoare in 1960 (Sir Charles Antony Richard Hoare, FRS, online at: http://www.cs.ox.ac.uk/people/tony.hoare). Sir Tony is an interesting character who started out in the Liberal Arts studying Classics at Oxford and got into Computer Science while trying to find the best way to translate one human language into another. In the process of translation, he found it necessary to sort large dictionaries of various languages so they could more quickly be referenced by translation algorithms. He developed quicksort as a way to do this.

The quicksort algorithm works by partitioning a large set of data into two smaller sets, then moving data to make sure that everything in the set on one side of the partition is less than or equal to everything in the set on the other side of the partition. One data element is chosen to serve as the pivot in the middle. The two sets are normally a low set on the left side of the pivot and a high set on the right side of the pivot. We then apply the quicksort algorithm recursively to each of these two sets.



The image above shows a set of 20 randomly generated integers between 1 and 100 ($1 \le X \le 100$) on the left, and the result of the first quicksort partition of the set on the right. The 10^{th} value, 42, was chosen as the pivot. After partitioning, all values to the left of the pivot are less than or equal to the pivot and all values to the right of the pivot are greater than or equal to the pivot. The process would then be repeated recursively for the set less than the pivot and then for the set greater than the pivot, and so on, until all of the elements are in the correct order.

The use of bars to illustrate sorting techniques is common in Web animations, such as the illustration of quicksort on the Wikipedia page at: *http://en.wikipedia.org/wiki/Quicksort*.

A very good short video describing and illustrating quicksort can be found online at the CS animated Website: *http://www.csanimated.com/animation.php?t=Quicksort*. I strongly encourage you to view this video, perhaps more than once, to understand how the quicksort algorithm works.

Quicksort is a classic example of a *recursive divide and conquer* algorithm. A **recursive divide and conquer algorithm** breaks a large problem into smaller, non-overlapping parts, then solves the smaller problems by recursively repeating the breakdown into smaller parts until a base case is reached in which an obvious, easy, or efficient solution can be applied. The small solutions are then reassembled into a big solution. In some algorithms we don't even need to reassemble the parts – the algorithm is complete when each solution at the smallest level is complete.

In the case of quicksort, we keep partitioning each data set into smaller data sets until we have data sets with at most one element (one or zero elements). This is the base case for the recursion. We don't go any further. A set with only one element is a pretty easy set to sort. The sets that have been partitioned into sets with at most one element will form a larger set that is in order because the process of quicksort at each level of recursion makes sure that all of the elements in the low end set are less than or equal to the pivot, which is less than or equal to the elements in the high end set.

The java code for a version quicksort is shown below. It is included with the chapter files as ***QuickSortDemo.***

```
/* QuickSortDemo.java
 *
 * CSCI 112 - Spring 2014
 *
 * This code provides an example of quicksort.  The method partion() chooses the
 * middle element as a pivot, then splits a set of integers into a low end set
 * with each element < the pivot and a high end set with
 * each element >=  the pivot.
 *
 * This process is repeated recursively for the left and right sets until all
 * elements are in order.
 *
 * To pick a pivot randomly, change line 34 to be:
 *        pivot index = startIndex + (int)(Math.random() * ((EndIndex- StratIndex) + 1))
 *
 * last edited march 14, 2014 by C. Herbert
 */
package quicksortdemo;

public class QuickSortDemo {
```

```
    public static void main(String[] args) {

        /* a sample 20-element array is hardcoded here.
         * You can replace this with code to read data from a file.
         */
        int[] a = {14, 22, 99, 50, 30, 46, 3, 60, 35, 42, 21, 38, 79, 58, 18, 27, 6, 49, 17, 47};

        // print the array before sorting
        // you might want to change this for long arrays
        System.out.println("Here is the array before sorting:");
        printtArray(a);

        // start the sort on the entire array -- a[0] to a[length-1]
        quickSort(a, 0, a.length - 1);

        // print the array after sorting
        // remove this for very large arrays
        System.out.println("Here is the array after sorting:");
        printtArray(a);

    } // end main()
    //*************************************************************************

    // a method to print the elements of an integer array on one line
    public static void printtArray(int[] a) {

        // iterate and print the array on one line
        for (int i = 0; i < a.length; i++) {
            System.out.print(a[i] + " ");
        }

        System.out.println();

    } // end printArray()
    //*************************************************************************

    // the recursive quicksort method, which calls the partition method
    public static void quickSort(int[] a, int startIndex, int endIndex) {
        int pivotIndex;      // the index of pivot returned by the quicksort partition

        // if the set has more than one element, then partition
        if (startIndex < endIndex) {
            // partition and return the pivotIndex
            pivotIndex = partition(a, startIndex, endIndex);
            // quicksort the low set
            quickSort(a, startIndex, pivotIndex - 1);
            // quiclsort the high set
            quickSort(a, pivotIndex + 1, endIndex);
        } // end if
    } // end quickSort()
    //*************************************************************************

    // This method performs quicksort partitioning on a set of elements in an array.
    public static int partition(int[] a, int startIndex, int endIndex) {

        int pivotIndex;             // the index of the chosen pivot element
        int pivot;                  // the value of the chosen pivot
        int midIndex = startIndex;  // boundary element between high and low sets
```

```
        // select the center element in the set as the pivot by integer averaging
        pivotIndex = (startIndex + endIndex) / 2;
        pivot = a[pivotIndex];

        // put the pivot at the end of the set so it is out of the way
        swap(a, pivotIndex, endIndex);

        // iterate the set, up to but not including last element
        for (int i = startIndex; i < endIndex; i++) {
            // if a[i] is less than the pivot
            if (a[i] < pivot) {

                // put a[i] in the low half and increment current Index
                swap(a, i, midIndex);
                midIndex = midIndex + 1;
            } // end if
        } // end for

        // partitioning complete -- move pivot from end to middle
        swap(a, midIndex, endIndex);

        // return index of pivot
        return midIndex;

    } // end partition
    //*************************************************************************
    // This method swaps two elements in an integer array
    public static void swap(int[] a, int first, int second) {

        int c;  // a catalyst variable used for the swap

        c = a[first];
        a[first] = a[second];
        a[second] = c;

    } // end Swap()
    //*************************************************************************

} // end class QuickSortDemo
```

The *main()* method above calls the *quicksort()* method for the entire array to be sorted. *quicksort()* is a recurasive method that calls *partition()*, which is the method that does the work of sorting. *Partition()* partitions the array into two sets, one with all elements less than the pivot and one with all elements greater than or equal to the pivot. It returns the pivot – but don't forget, arrays are objects passed by reference; so really, the entire array is returned, with the low set, pivot and high set.

Recursively, quicksort then calls itself for each part of the array – the set before the pivot and the set after the pivot. This continues until we have a collection of single-element sets in the correct order.

One of the critical aspects of quicksort is how the pivot is chosen. If a poor pivot is chosen, then quicksort is barely faster than iterative sorts for some data sets. If a good pivot is chosen, quicksort can be among the fastest of all sorting methods.

In the partition method above, the middle element in the list to be sorted is chosen as the pivot. Here are four ways to choose a quicksort pivot:

1. Just use the first (or last) element in the set as a pivot.
   This was a simple technique used in early versions of quicksort. The problem with this is that an attempt to sort already sorted data or nearly sorted data could result in a worst-case run of quicksort.
2. Choose the middle (median) element in the set to be sorted.
   This works better than choosing the first element if the set is sorted or nearly sorted, but occasionally the middle element is not a good pivot.
3. Choose a random pivot. Most of the time, this works, but occasionally the worst possible pivot is chosen.
4. Look at the first, last and middle element and choose the one in the middle of these three. This is a method that was suggested by Dr. Robert Sedqwick, whose doctoral dissertation was about the quicksort algorithm. It generally has very good results.

One of the exercises at the end of this chapter addresses the choice of quicksort pivot.

20.3 Merge Sort

Merge sort is a recursive sorting algorithm introduced by Herman Goldstine and John Von Neumann in the 1940's. Goldstine was one of the original authors of the proposal for ENIAC. He introduced Von Neumann to the ENIAC project before both went on to become professors in the institute for Advanced Study at Princeton University. Goldstine lived in the Philadelphia area for most of his adult life, and served as president of the American Philosophical Society that had been founded in Philadelphia by Benjamin Franklin.

Merge sort is a recursive divide and conquer sorting algorithm that breaks one set with many elements into many one-element sets, then sorts as it merges the one-element sets back into one set with many elements.

It works by breaking a set in half, then breaking the halves in half, and so on recursively, until the set becomes a collection of one-element sets. The one-element sets are then merged into two-element sets in order, those two-element sets are merged into four-element sets, and so on, until we have a sorted version of the original set. The work of sorting occurs as smaller sets are combined into larger sets.

In most versions of merge sort, as two small sets are merged and sorted into one larger set, the smaller sets are first copied to another array, then put back in place in the original array in order as they are merged.

The diagram on the right shows two sorted four-element subsets being merged into one eight-element set.

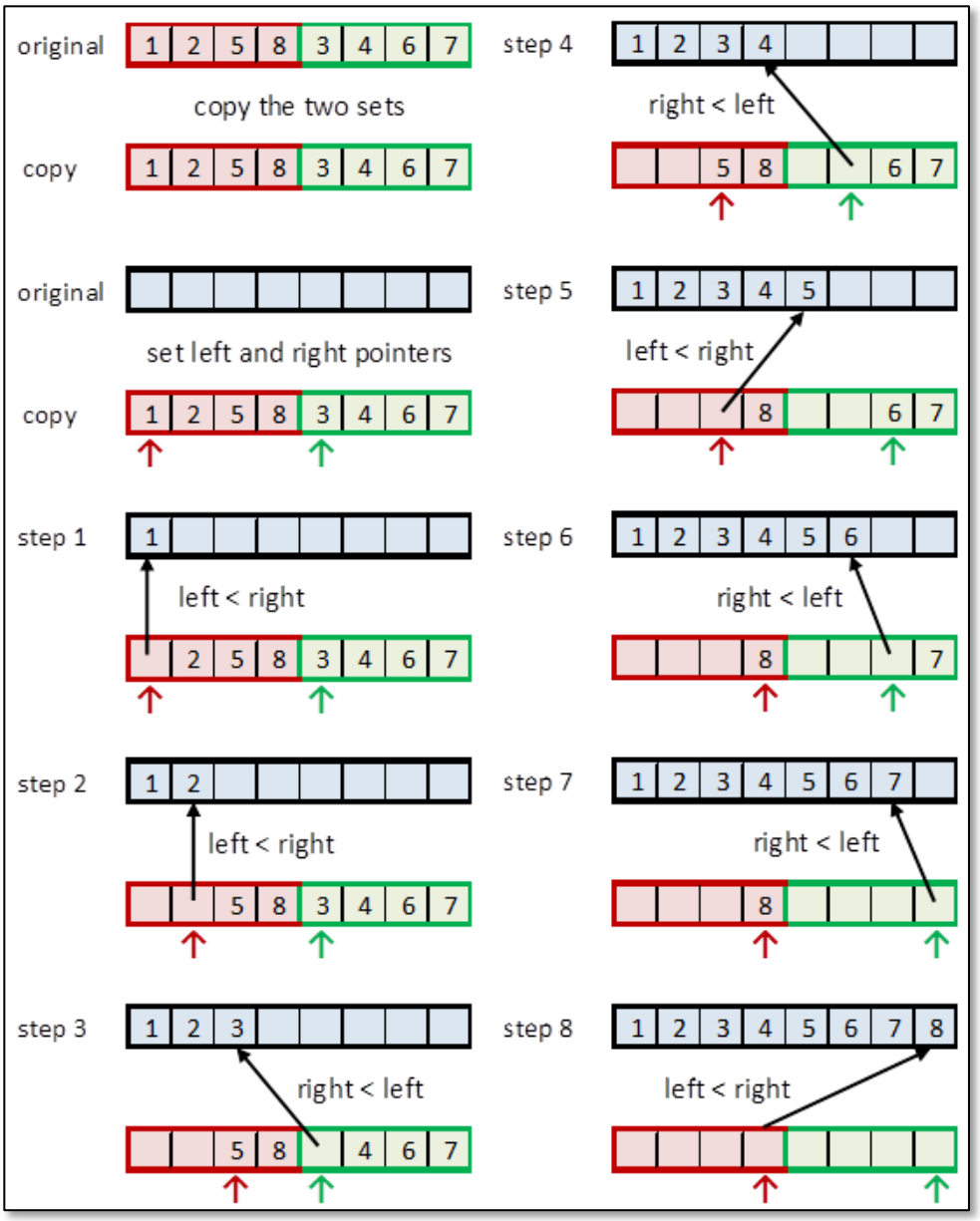
First, a copy of the subsets is made in a new array.

Left and right pointers are set up for the first element in the left subset copy and the first element in the right subset copy. The elements in each subset are already in order.

The sorting merge works by moving the lowest remaining value from either of the two subset copies back to the original set

If the left pointer's value is less than or equal to the right pointer's value, then the left element is moved up and the left pointer is incremented. If not, then the right element is moved up and the right pointer is incremented.

This continues until all elements have been moved back to the original set.



The java code for a version of merge sort is shown below. It is included with the chapter files as ***MergeSortDemo.***

```
/* MergeSortDemo.java
 * CSCI 112 Spring 2014 Semester
 * last edited March 11, 2014 by C. Herbert
 *
 * This application demonstrates a merge sort algorithm performed on an array
 * of 9 digits hardcoded into the program.  The array is displayed on the Console
 * before and after the sort.
 *
 * This code is written for clarity.  Changes can be made to improve effeciency.
 *
 * The accompanying software - MergeSortDemo.java - uses the same mergeSort() and
 * merge() methods to sort an array of  1 million randomly generated integers
 * of up to six digits each.
 */

package mergesortdemo;

public class MergeSortDemo {

    public static void main(String[] arg) {

         // a[] is an integer array to be sorted
        int[] a = {1, 6, 2, 5, 8, 4, 3, 9, 7};

        // empty temporary array, the same size and type as a[]
        int[] temp = new int[a.length];

        // display array before sorting
        displayArray(a);

        // sort the entire array
        mergeSort(a, temp, 0, (a.length - 1));

        // display array after sorting
        displayArray(a);

    }// end main()
    //*****************************************************************

    public static void mergeSort(int[] a, int[] temp, int low, int high) {
        //  low is the low index of the part of the array to be sorted
        //  high is the high index of the part of the array to be sorted

        int mid;  // the middle of the array - last item in low half

        // if high > low then there is more than one item in the list to be sorted
        if (high > low) {

            // split into two halves and mergeSort each part
```

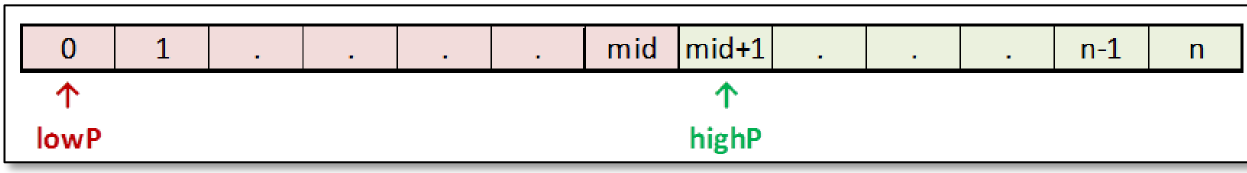
```
            // find middle (last element in low half)
            mid = (low+high)/2;
            mergeSort(a, temp, low, mid );
            mergeSort(a, temp, mid+1, high);

            // merge the two halves back together, sorting while merging
            merge(a, temp, low, mid, high);
        } // end if

        return;
    }// end mergeSort()
    //*****************************************************************


    /* This method merges the two halves of the set being sorted back together.
     * the low half goes from a[low] to a[mid]
     * the high half goes from a[mid+1] to a[high]
     * (High and low only refer to index numbers, not the values in the array.)
     *
     * The work of sorting occurs as the two halves are merged back into one
     * sorted set.
     *
     * This version of mergesort copies the set to be sorted into the same
     * locations in a temporary array, then sorts them as it puts them back.
     * Some versions of mergesort sort into the temporary array then copy it back.
     */
    public static void merge(int[] a, int[] temp, int low, int mid, int high) {
        //  low is the low index of the part of the array to be sorted
        //  high is the high index of the part of the array to be sorted
        //  mid is the middle of the array - last item in low half

        // copy the two sets from a[] to the same locations in the temporary array
        for (int i = low; i <= high; i++) {
            temp[i] = a[i];
        }

        //set up necessary pointers
        int lowP = low;          // pointer to current item in low half
        int highP = mid + 1;     // pointer to current item in high half
        int aP = low;            // pointer to where each item will be put back in a[]

        // while the pointers have not yet reached the end of either half
        while ((lowP <= mid) && (highP <= high)) {

            // if current item in low half <= current item in high half
            if (temp[lowP] <= temp[highP]) {
                // move item at lowP back to array a and increment low pointer
                a[aP] = temp[lowP];
                lowP++;
            } else {
                // move item at highP back to array a and increment high pointer
                a[aP] = temp[highP];
                highP++;
            } // end if..else

            // increment pointer for location in original array
            aP++;
        } // end while
```

```
        /* When the while loop is done, either the low half or the high half is done.
         * We now simply move back everything in the half not yet done.
         * Remember, each half is already in order itself.
         */

        // if lowP is at end of low half, then low half is done, move rest of high half.
        if (lowP > mid)
            for (int i = highP; i <= high; i++) {
                a[aP] = temp[i];
                aP++;
            } // end for
        else // high half is done, move rest of low half

            for (int i = lowP; i <= mid; i++) {
                a[aP] = temp[i];
                aP++;
            }// end for

        return;
    } // end merge()
    // ************************************************************

    public static void displayArray(int[] a) {

        for (int i = 0; i < a.length; i++)
            System.out.print(a[i] + " ");
        System.out.println();

    } // end displayArray()

} // end class MergeSortDemo
```

The merge sort method recursively continues to break an array or part of an array into two halves, until the part being considered has only one element. Its parameters are the name of the array and the name of the temporary array to be used, along with the index of the lowest (leftmost) element and the index of highest (rightmost) element in the part of the array to be sorted. If high < low then there is more than one element, so the method breaks the set into two parts and calls itself again for each part. If high = low, then there is only one item in the set and the method does nothing but return.

The merge method first copies the two sets to be merged into the same locations in a temporary array. It then merges the two already sorted parts of the array into a larger part, sorting by putting them in order in the original array as it does so. Some merge sort algorithms merge and sort into the temporary array, then copy the values back afterwards, this method copies first, then moves values back as it merges and sorts.

In the merge method, lowP is a pointer to the elements in the low set (leftmost) and highP is a pointer to the elements in the high set (right most). Whichever value is lower, the one at lowP or the one at highP, will be moved back into the original array in the next location. The pointers are them incremented.



If either pointer reaches the end of its set, then the rest of the remaining set is simply moved back into the original array.

The mergeSort() method will continue to split the an array of ***n*** elements into parts until we have a collection of ***n*** sets with one element each. Each of these one element sets are then merged back into two-elements sets that are sorted by the merge() method, two elements sets are merged into four-elements sets, and so on.

The technique does work for arrays whose size is not a power of 2. Consider what happens to the 13-element array shown below. The diagram shows how the set is split and then reassembled. If an odd number of values needs to be split into two "halves", then the first "half" will have one more element than the second "half". The numbers in the diagram are the index values of the locations in the array.

0	1	2	3	4	5	6	7	8	9	10	11	12
0	1	2	3	4	5	6	7	8	9	10	11	12
0	1	2	3	4	5	6	7	8	9	10	11	12
0	1	2	3	4	5	6	7	8	9	10	11	12
0	1	2	3	4	5	6	7	8	9	10	11	12
0	1	2	3	4	5	6	7	8	9	10	11	12
0	1	2	3	4	5	6	7	8	9	10	11	12
0	1	2	3	4	5	6	7	8	9	10	11	12
0	1	2	3	4	5	6	7	8	9	10	11	12

20.4 The Temporal Complexity of Sorting Algorithms

Merge sort, *quicksort*, and other recursive sorting algorithms are very fast, especially when compared to iterative sorting algorithms such as *insertion sort* and *selection sort*. Experiments conducted by the author using a *Dell Latitude* laptop with *Windows 7*, an *Intel I5 processor,* and 8 gigabytes of memory, yielded the following results for 100 trial runs of the quicksort and merge sort algorithms shown in this chapter, each sorting a randomly generated array of 1 million six-digit integers:

	average	best case	worst case
Quicksort	1.06 sec.	1.05 sec.	1.14 sec.
Merge sort	1.61 sec.	1.59 sec.	1.78 sec.

Insertion sort, Selection sort and Bubble sort each took more than 30 minutes for a single run.

Simple sorting techniques such as selection sort and insertion sort might work well for sorting a hand of cards in a card game, but for very large data sets they are impractical due to their temporal complexity.

The **temporal complexity** of an algorithm is the relationship between the size of an algorithm's data set and the time it takes for the algorithm to finish processing the data. The **spatial complexity** of an algorithm is the relationship between the size of an algorithm's data set and the amount of space (memory) the algorithm uses to finish processing the data. The temporal complexity and spatial complexity of algorithms are studied in the field of computational complexity theory. **Computational complexity theory** is a branch of mathematics and computer science that examines the complexity of problems and algorithms to solve those problems based on the resources needed to solve the problem.

As discussed earlier in this chapter, algorithms can be analyzed theoretically (*a priori*) and experimentally (*a posteriori*). Mathematicians and computer scientists measure the theoretical running time of an algorithm based on the number of operations needed to complete the algorithm. This theoretical analysis of algorithms uses a combination of simple and advanced mathematics, including algebra, calculus, probability, statistics, linear algebra, combinatorics, and so on.

The study of recurrence relations is very useful in examining the complexity of algorithms. A **recurrence relation**, or *recurrence equation*, is an equation that describes the members of a sequence of terms, such as the Fibonacci sequence, defined by the recurrence equation *fib(n) = fib(n-1) + fib(n-2)*.

Computational experts attempt to find a recurrence relation describing the time it takes an algorithm to run based on the size of its data set. This will give us the algorithm's general temporal complexity. The analysis and discovery of an algorithm's recurrence relation requires advanced mathematical skills and is beyond the scope of this class, but certain patterns in an algorithm can give us clues to the temporal complexity of an algorithm. Consider the following three short algorithms:

Algorithm 1

```
n = 1,000
for ( i = 1; i <= n; i++)
   print "*";
```

Algorithm 2

```
n = 1,000
for ( i = 1; i <= n; i++)
   for (j = 1; j <= n; j++)
         print "*";
```

Algorithm 3

```
n = 1,000
while (n > 1)
       {
         print "*";
         n = n/2;
       }
```

How is the running time of each algorithm related to the size of the algorithm's data set? The structure and nature of the algorithm gives us some clues.

In algorithm 1, the loop is repeated 1, 000 times. 1,000 stars are printed. What happens if we change ***n*** to be 2,000, or 5,000, or 10,000? Then 2,000, 5,000, or 10,000 stars would be printed. A simple *a priori* analysis based on the structure of the algorithm is enough to tell us that number of operations the algorithm performs is directly proportional to ***n***. We do not need to conduct an experiment to see this, although we might want to conduct an experiment to verify our theory.

In general, if **o** (the letter o) is the number of operations needed to complete an algorithm, then for algorithm 1, above, ***o = f(n).*** This is a linear relationship, in which ***o*** grows at roughly the same rate as ***n***. There might be some fixed overhead in the algorithm that is not considered, such as a few instructions to get things started, but this fixed number (this constant number) becomes less important as the data set grows. The fact that three, five, or even 20 operations are needed to set things up becomes less important as ***n*** grows to a thousand, a million, or more. It is the number of iterations of the loop – one for each data element – that will be the major factor in determining how many operations the algorithm executes.

In the second algorithm we have two loops – one inside another. The inside loop is executed 1,000 times each time the outside loop is executed, and the outside loop is executed 1,000 times. One thousand times one thousand equals 1 million. The algorithm will print 1 million stars.

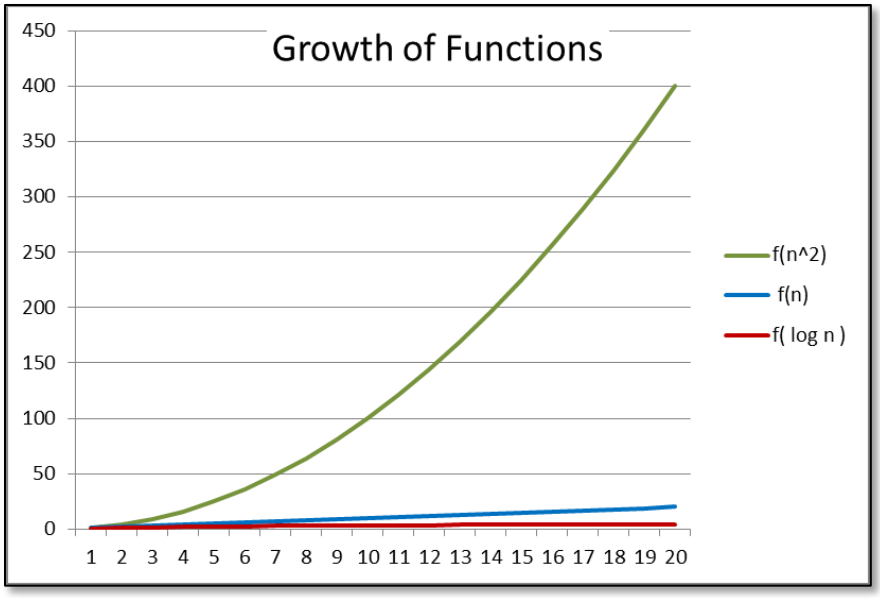
What happens if we change ***n*** to be 2,000, 5,000, or 10,000? Then 4,000,000 (4 million), 25,000,000 (25 million), or 100,000,000 (100 million) stars would be printed. In general, $\boldsymbol{o = f(n^2)}$ for this algorithm – a quadratic relationship. The constant number of operations needed to get things started becomes even less significant as n grows larger for this algorithm. The number of operations the algorithm executes is proportional to $\boldsymbol{n^2}$, which means it grows much quicker than ***n***.

In the third algorithm, we have a while loop with ***n*** as the loop control variable. ***n*** starts at 1,000, but is divided in half each time through the loop, until ***n*** is less than one. How many times does the loop execute? How many times can we cut 1,000 in half until we get to one? 1,000, 500, 250, 125, 62.5, 31.25, 15.625, 7.8125, 3.90625, 1.95313, 0.97656 . . . Ten times. We can divide 1,000 by two ten times before we reach one.

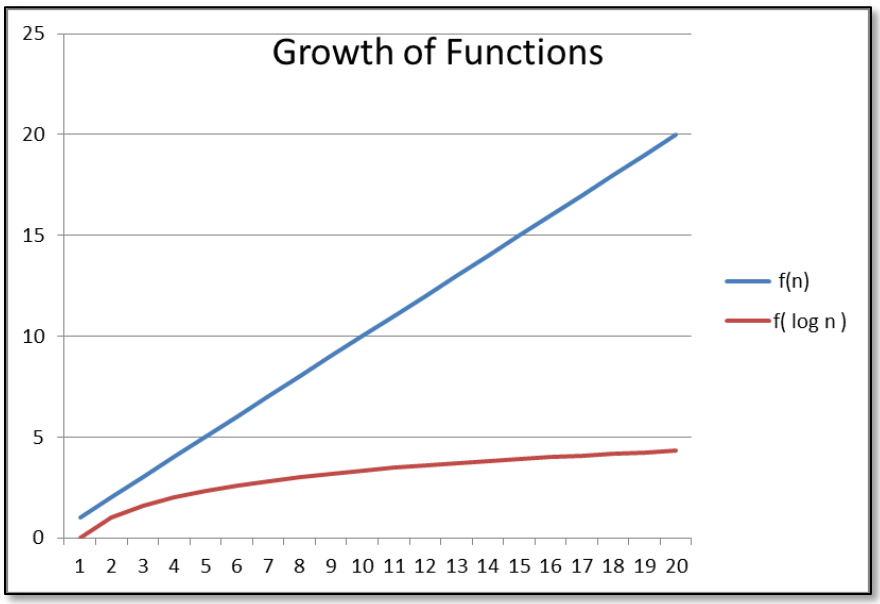
This can be described by using base two logarithms – the $\log_2(1{,}000) \approx 9.97$. We round this up to the next integer because the number of times a loop executes is an integer (We don't execute a loop 9.97 times – it is either 9 times or 10 times.) Algorithm 3 will only print 10 stars.

What happens to algorithm 3 if we change ***n*** to be 2,000, 5,000, or 10,000? Then 11, 13, or 24 stars will be printed, because $\log_2(2{,}000) \approx 10.97$, $\log_2(5{,}000) \approx 12.29$, and $\log_2(10{,}000) \approx 13.29$. In general, $\boldsymbol{o = f(\log(n))}$ for this algorithm – a logarithmic relationship. The number of operations executed by a logarithmic algorithm grows more slowly than ***n***.

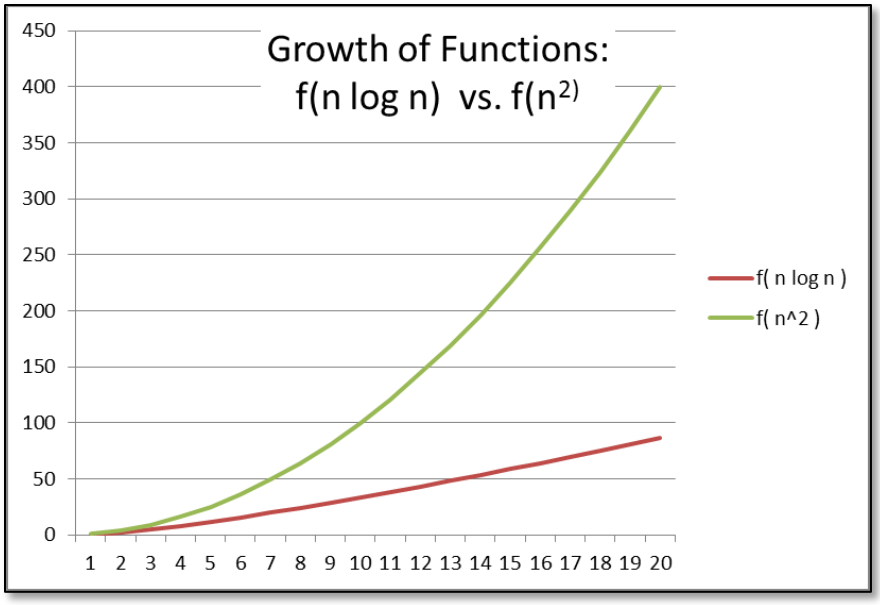
The three functions that describe the algorithms above are shown on the graph below.



As we can see, $f(n^2)$ grows much more quickly than $f(n)$ and $f(\log n)$ grows less quickly than $f(n)$. The graph below shows us a different scale, with a better view of $f(n)$ compared to f (log n). We can see that the growth of $f(\log n)$ starts to level off compared to the growth of $f(n)$ as n becomes larger.



Lastly, we shall see a graph of the growth of $f(n^2)$ compared to the growth of $f(n \log n,)$ which is *n times log(n)*. This is an important comparison in computer science. The temporal complexities of many recursive sorts, such quick sort, merge, sort and heap sort are closest to $f(n \log n)$ while the temporal complexities of many iterative sorts, including insertion sort, selection sort and bubble sort are closest to $f(n^2)$.



The temporal complexity of algorithms with a loop inside a loop are often close to $f(n^2)$, compared to recursive divide and conquer algorithms that repeatedly break a big problem into two (or more) smaller problems, which have a time complexity closer to *f(n log n)*. The graph shows us what happens to these two functions as *n* gets large. For very large data sets, the difference in running time becomes very significant. Even if we can get an $f(n^2)$ algorithm down to $f(\frac{n^2}{2})$, it will still take significantly longer to run for large data sets, with the difference becoming more dramatic as the data set gets bigger.

20.5 Asymptotic Notation

An **asymptote** is a line that approximates the limit of the dependent value (y) of a function y= *f(x)* as its independent value (x) approaches some particular value. Most often, asymptotes show what happens to a function y = *f(x)* as *x* grows larger. The concept of an asymptote has been around since the time of the ancient Greeks, but number theorists, particularly two German Math professors – Paul Bachmann and Edmund Landau – began using what is now known as *asymptotic notation* in the 1890's and early 1900's.

Asymptotic notation in computer science is a method of using general functions to describe the behavior of the complexity of an algorithm as the algorithm's data set grows larger. There are three forms of Asymptotic notation:

- ***Big O*** notation ***O(n)***
- ***Big Omega notation*** ***Ω(n)***
- ***Big Theta*** notation ***Θ(n)***

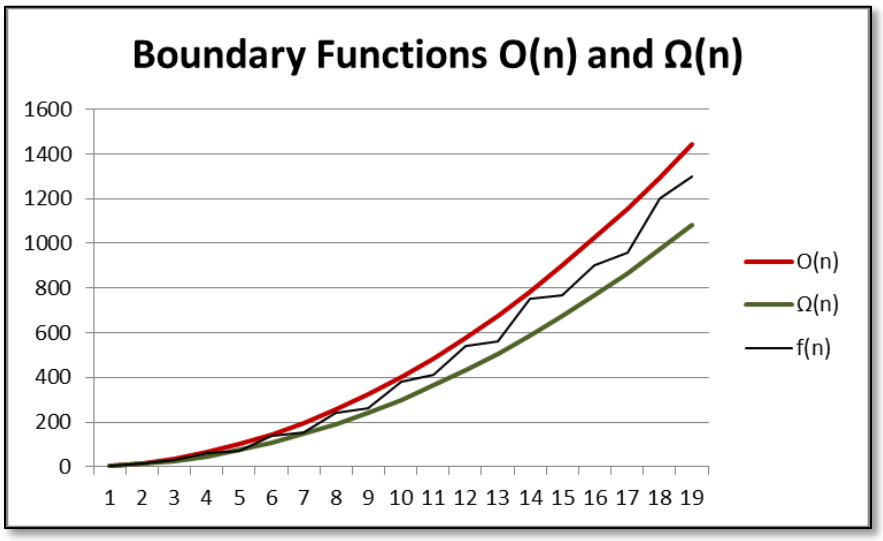
Big O notation O(n)

Big O notation represents an upper bound to the value of a function as it grows. *f(n)* will always be lower than *O(n)*. For example if *O(n)* is $4n^2$, then the function will never be larger than $f(n) = 4n^2$ for any *n*. Ideally, the limiting function for big O of a function should be a tight bound, meaning that the function will not be bigger than big *O(n)*, but will be close to it.

Big Omega notation Ω(n)

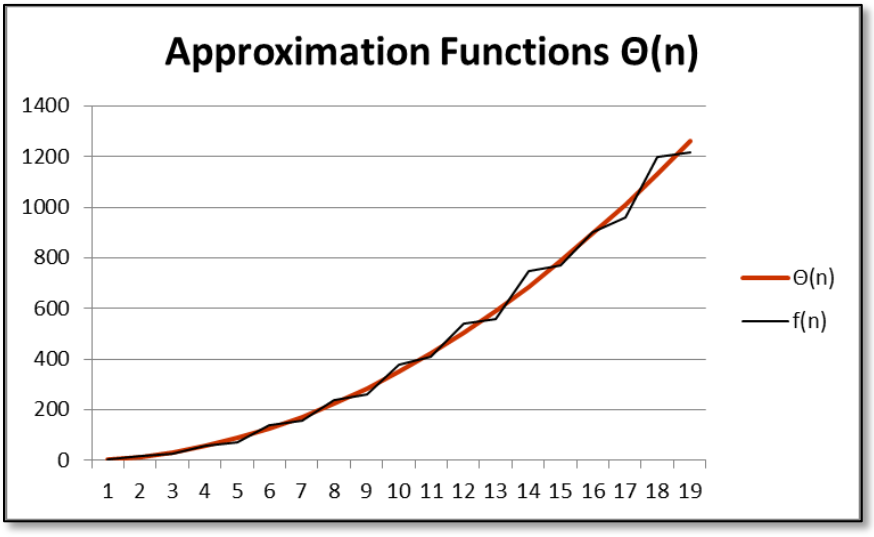
Big omega notation represents a lower bound to the value of a function as it grows. *f(n)* will always be higher than *Ω (n)*. For example if *Ω (n) = 3n²*, then the function will never be less than *f(n) = 3n²* for any *n*. Ideally, the limiting function for big theta of a function should be a tight bound, meaning that the function will not be lower than big *Ω (n)*, but will be close to it.

Together, *O(n)* and *Ω(n)* create an envelope in which the function *f(n)* lies, as shown below.



Big Theta notation Θ(n)

Big theta notation represents a close approximation to the value of a function as it grows. *f(n)* will always be close to *Θ(n)*, relative to the size of *n*. For example if *Θ (n) = 2n²*, then the function will always be close to $2n^2$ and one could say that $2n^2$ is a good approximation of the function, especially as *n* gets larger. Big theta notation takes the place of expressing the complete big O and big omega envelope when expressing the approximate growth of a function. Be careful – often people use or say big O when they mean big *Θ.*



Most often, we express the temporal complexity of an algorithm according to its general magnitude, leaving out smaller terms and coefficients when expressing approximation functions. Is an algorithm linear ($\Theta(n)$), is it $\Theta(n \log n)$, or is it $\Theta(n^2)$? This is usually good enough for designing and choosing algorithms at our level.

20.6 The Spatial Complexity of Sorting Algorithms

We have seen that *Merge sort*, *quicksort*, and other recursive sorting algorithms are faster than iterative sorting algorithms such as *insertion sort* and *selection sort,* but what about the amount of storage space (memory) they use – their spatial complexity?

All algorithms use memory for two things – data and instructions. Data storage space is needed for the data set itself, and for auxiliary data. **Auxiliary data** is data that is need to process other data, such as local variables, other data structures needed to process given data, or temporary copies of a data set or part of a data set. Some auxiliary data does not depend on the size of an algorithm's data set. Consider the first of the three algorithms we saw in the *Section 20.4*, shown again here:

Algorithm 1

```
n = 1,000
for ( i = 1; i <= n; i++)
   print "*";
```

We need data storage space for the variables **n**, and ***i***, but the amount of memory needed for ***n*** and for ***i*** does not change as the value of ***n*** changes. What if ***n*** is 2,000 or 5,000, or 10,000? ***n*** and for ***i*** still each need the same storage space.

In the case of this simple algorithm, the space needed for auxiliary data does not depend on the size of the data set, it is constant.

What about the following algorithm?

Algorithm 4

```
n = 1,000
int a[n];

for ( i = 1; i <= n; i++)
   a[i] = (int)(Math.random()*100+1);
```

In this algorithm, the size of the array depends on *n*. If *n* is changed to 2,000, 5,000 or 10,000 then the size of the array grows accordingly. Memory is needed for the array, plus a small constant amount of memory for *n* and *i.* The size of the array is the major factor in determining the amount of memory needed for the algorithm. The algorithm's spatial complexity is linear, $\Theta(n)$.

Remember, big theta, big O, and big omega notation generally ignore constants and minor terms. A program that has a spatial complexity of $2n+7$ and a program that has a spatial complexity of $1n+8$ would both have linear spatial complexity $\Theta(n)$. We ignore any small constant memory needed.

The spatial complexity of most algorithms tends to be trivial compared to temporal complexity. However, small changes in spatial complexity can be much more critical than changes in temporal complexity in determining if a program will run on a given computer. Consider a program to sort 500 Mb of data on a computer that has 1 Gb of memory. The computer probably needs 250 Mb for the operating system, etc. it still has enough memory to sort the data. Yet, if the sorting algorithm needs to make a copy of the data, then we do not have enough memory.

Just as with time complexity, the nature and structure of an algorithm can tell us something about its spatial complexity. An **in-place** algorithm is an algorithm that can process a large data structure where it is in memory without making a copy of the data. Other algorithms, or other versions of the same algorithm, might need to make one or more copies of the data, but in-place algorithms don't. Insertion sort, Selection sort and Quicksort are all examples of algorithms that can operate in-place. The version of merge sort shown in this chapter makes a copy of the array to be sorted, so it is not an in-place sorting algorithm and requires roughly twice as much memory as quicksort.

Recursion Overhead

The nature of an algorithm's recursion is also critical in determining how much memory it uses. Each time a method is called by an algorithm, memory is needed to keep track of the new method and the method to which the computer should return when the new method is complete. This is known as **subroutine linkage** in the language of operating systems.

Typically, a system stack is created to keep track of subroutine linkage, with a set of data on the stack for each method that has called another method. A **stack** is a last-in first-out data structure, like a stack of dishes. It reverses the order of the items placed on the stack, which is just what we need to keep track of calling and returning from methods in a string of methods that call one another. A **stack frame** is a set of data on the system stack with information about the state of a method that has called another method. Each time a method is called, a stack frame is added to the stack. Each time a program returns from a method, the stack frame is removed from the stack and its data is used by the computer to determine which method to return to, what the values of that method's variables should be, and so on.

This means that each recursive call to a method generates a stack frame. **Recursion overhead** is the extra memory that recursion requires, largely composed of system stack frames for method calls. But, is it enough to cause problems for recursive algorithms?

Each computer has a limited amount of stack space. This author tried the following recursive algorithm on a Dell Latitude with 4Gb or RAM and found that it ran out of stack space after roughly 7,000 recursions.

```
/*
 * RecursiveAddDemo.java
 * This method tests adding by 1 (counting)recursively.  It is used to
 * show that an algorithm with linear tail recursion may run out of memory
 * because of stack space limitations.
 *
 * last edited March 20, 2014 by C. Herbert
 */

package recursiveadddemo;
```

```
public class RecursiveAddDemo {

    public static void main(String[] args) {

        recursiveAdd(1);  // start with the number 1

    } // end main()

public static int recursiveAdd(int i)
{
    System.out.println(i);    // display i
    i = recursiveAdd(i+1);    // the method calls itself for i+1
    return i;
}
} // end class RecursiveAddDemo
```

The recursive add program is an example of linear recursion. **Linear recursion** occurs when an algorithm calls itself once with a subtracted or added value each time. In the case of this algorithm, the method calls itself increasing by one each time a recursion occurs. After about 7,000 calls, the algorithm runs out of stack space. No more stack frames can be added and the algorithm crashes, generating a *StackOverflowError* exception.

Linear recursion is often a bad use of recursion, resulting in too much recursion overhead; it uses too much stack space. Linear recursion can almost always be replaced with a simple iterative algorithm – a loop.

But what about the recursion in quicksort and merge sort? They are examples of divide and conquer algorithms that break a set into two sets, and so on recursively, with the number of levels of recursion active at any one time equal to the depth of dividing things in half. As we saw above, we can divide a set of 1,000 items in half 10 times before we reach 1 item.

Each time the algorithm divides a set in half, it calls a new method, which means it adds a stack frame, but each time it returns from that method it removes a stack frame. When it gets down to the tenth level of recursion it goes no deeper, and must return from that level of recursion before going on to the next sets to be split and merged. Ultimately, for 1,000 items, 1,000 recursive method calls will occur – but only 10 at any one time.

At most, 10 levels of recursion are active at one time for quicksort or merge sort with 1,000 items.

The math of this, as described above, uses logarithms. The $\log_2(1{,}000)$,rounded up, is 10. Quicksort, merge sort, and most recursive divide and conquer algorithms are examples of **logarithmic recursion**, in which the number of active recursive calls, hence the number of frames on the system stack, is equivalent to the base two log of the size of the data set, rounded up to the next integer. For 1,000, data items this is ten. For 1 million data items it is 20, for 1 billion items it is only 30 – far less than the 7,000 that caused a stack overflow error in the case of simple linear recursion above.

The logarithmic recursion of divide and conquer algorithms creates only a small amount of recursion overhead – proportional to the log of the size of the data set – and is not a problem in the same way that the overhead from linear recursion is.

Quicksort is an in-place recursive sorting algorithm whose temporal complexity is $\Theta(n \log n)$. Its spatial complexity is also $\Theta(n \log n)$, because of the stack space it needs. Specifically, its spatial complexity is $1n*(a*log(n))+b$, where a and b are small constants. Merge sort needs more space because it makes a copy of the data; its spatial complexity is $2n*(a*log(n))+b$, where a and b are small constants.

Insertion sort and selection sort do not need stack space for recursion and are both in place algorithms. While their temporal complexity is $\Theta(n^2)$, their spatial complexity is simply $\Theta(n)$.

In summary, even though a recursive divide and conquer sorting algorithm uses more space than an iterative method, it is only a little more space for a few levels of recursion at a time, but the payback for that is a much faster sorting method – thousands or millions of times faster for very large data sets.

20.7 The Actual Running Time of Sorting Software

Theory and practice – mathematical analysis of a sorting algorithm can give us a good theory about the algorithm's temporal complexity, but to find out how long it will actually take to run an algorithm we need to put the theory to practice. We need to write code for the algorithm and run it as computer software.

Asymptotic notation and the analysis of algorithmic complexity are based on the number of operations an algorithm performs. To translate this into actual time, we need to know much more about how long it takes a computer to perform each operation. There are two problems with this – first, it requires knowledge about the innermost workings of the components of a computer system; and second, it is different from one computer to another. So, instead of trying to extend the analysis to develop a detailed theory about the specific time it will take an algorithm to run, we can find out how long algorithms run by actually running them and timing the results.

The actual running time of an algorithm implemented as a computer program depends on many factors. The physical nature of the computer is the biggest factor – how the CPU and memory are arranged, how they work, and how information moved about in a computer system. The physical structure of a computer and how it works are equivalent of a machine's "anatomy and physiology". This is the biggest factor in determining how an algorithm will run on a computer.

In simple terms, the first factor determining the running time of an algorithm is the make and model of computer running the algorithm. This, for practical purposes, is the machine's "anatomy and physiology."

The amount of internal memory, usually a computer's Random Access Memory (RAM), is another important factor in determining the running time of an algorithm. if the entire collection of the programs running on a computer at one time and the data it is using at the same time fit in RAM, then a program will most likely run faster than if the computer needs to access a slower storage device, such as a disk drive. More memory usually means faster program execution.

The choice of an operating system is a significant factor affecting the run time of an algorithm. Some operating systems are faster than others. In January of 2007 Microsoft introduced the Windows Vista operating system to replace Windows XP. By 2009 it had replaced XP with Windows 7. Part of the reason for the quick replacement of Windows Vista was its poor performance compared to its predecessor, Windows XP. Identical code performs differently on identical machines with different operating systems.

The programming language and specific compiler being used also affect how an algorithm will run once it is implemented as a computer program. Java is the world's most widely used programming language largely due to its cross-platform portability, based on the use of a Java Virtual Machine(JVM). That same feature, the JVM, can slow algorithm. Java programs usually run through a local JVM, which communicates with the CPU. This extra step increases the running time for programs.

Converting java software into an executable file that does not use aJVM is one solution, but the best alternative is to write code directly in machine code that has been optimized for particular hardware. This works, but the development time and costs are huge. For most software, the extra run time from using a JVM is minimal, in part because most JVMs are very efficient. But for really important software where timing is critical, the development time and costs for machine-level programing may be justified.

Other software running on the system at the same time could also slow down a program. If the program has to compete with other programs for resources, such as CPU time, then the slowdown could be significant. Turing off all unnecessary software before running a program will usually result in much better run times for the program on a particular system.

So, many factors influence the running time of algorithms, such as the make and model of a computer and its processor, its operating system, how much memory the system has, what programming language is being used, and what other software is running at the same time. All of these things together make up the operating environment for a program.

Software Benchmarking

Software benchmarking is the process of running software in a particular operating environment to study its performance. Usually the person performing the benchmarking tries to isolate one variable for comparison. For example, everything might be the same except for the amount of memory the system has to see how this affects performance. In another example, the same program might be run on identical machines with different operating systems to benchmark the performance of the operating systems.

Two programs that do the same thing with different algorithms can be benchmarked against one another by running them on identical machines many times each and comparing results.

The nature of the data set, in addition to its size, could also be a big factor influencing the run time of an algorithm. How well does the algorithm perform with random data, compared to data that is already sorted, or with datthat is sorted backwards? A thorough theoretical analysis of an algorithm should account for the algorithm's worst case performance, best case performance and average performance with different data sets.

Running a program many times during benchmarking can help to measure the program's worst case, best case and average performance. However, we still need to be careful, it is possible that a worst case performance might not occur while benchmarking an algorithm, so for real world applications, thorough analysis and professional testing of software are needed.

For the purposes of testing our sorting algorithms, we can benchmark sorting techniques by running programs that implement each algorithm hundreds of times and comparing the results. We can run merge sort hundreds of time with random data in a specific environment, then do the same with quicksort and

compare results to see which is faster. We can also do so for questions about how a change in an algorithm affects its run time. For example, we could try quicksort algorithms that pick pivots in different ways and compare the results to see which works best. This is addressed in the exercises at the end of the chapter.

Timing a Method in Java

The Java method *System.nanoTime()* can be used in benchmarking to time an algorithm. It returns the system time in nanoseconds as a long integer. It can be used to read the system clock once before the sort starts and once after it ends. The start time can be subtracted from the end time to get the elapsed time for a method.

Shown below is the code for a program to time the merge sort algorithm from this chapter 100 times, sorting an array of 1 million six-digit numbers each time. It is included with the files for this chapter as the NetBeans project ***MegaMergeSortDemo***. This is an example of software bechmarking.

```
/* MegaMergeSortDemo.java
 * CSCI 112 Spring 2014 Semester
 * last edited March 21, 2014 by C. Herbert
 *
 * This application demonstrates mergesort and timing a mergesort that sorts 1 million
 * randomly generated integers, each up to 5 digits long.
 *
 * It uses the system function nanoTime() to read the system closk in nanoseconds,
 * once before the sort starts and once after it ends. It subtracts the end time from
 * the start time to get the elapsed time in nanoseconds.  This is divided by 1 bilion to
 * gte the time in seconds.
 *
 * The randomly genertated array is written to a data file before the sort and again
 * after the sort.
 *
 * This code is written for clarity.  Changes can be made to improve efficiency.
 */
package megamergesortdemo;

import java.io.*;

public class MegaMergeSortDemo {

    public static void main(String[] arg) throws Exception{

        for (int k = 1; k<= 100; k++)
        {
        int size = 1000000;     // change this number to change the size of the random
array
        int[] a = new int[size];
        int[] temp = new int[a.length]; // empty temporary array, same size and type as a[]

        // fill the array with random integers
        for (int i = 0; i< a.length; i++)
            a[i] = (int)(Math.random()*100000 +1);
```

```
        // get the start time in nanoseconds
        long startTime = System.nanoTime();

        //call mergesort to sort the entire array
        mergeSort(a, temp, 0, (a.length - 1));

        // get the end time in nanoseconds
        long endTime = System.nanoTime();

        // calculate elapsed time in nanoseconds
        long duration = endTime - startTime;

        // print the elapsed time in seconds   (nanaoseconds/ 1 billion)
        // this could be written to a file instead of the sceen.
        System.out.printf("%12.8f %n", (double)duration/100000000) ;

      }// end for

   }// end main()
   //*****************************************************************
```

The table below shows a typical software benchmarking data report:

Software Benchmarking
Running time of Quick Sort vs. Merge Sort

System Configuration	
Make:	**Dell**
Model:	**E5520**
Processor:	**Intel i5 2520M @ 2.50 Ghz**
RAM:	**8GB**
Op. Sys.:	**Windows 7, 64 bit**
IDE:	**Netbeans 7.3.1**
Language:	**Java 7.45**

	Size of Data Set				
	100,000	200,000	500,000	1,000,000	2,000,000
Quick Sort					
average	0.09349	0.19567	0.51400	1.05610	2.26223
best case	0.09060	0.19012	0.49978	1.03504	2.22343
worst case	0.10651	0.23258	0.76272	1.15449	2.35602
Merge Sort					
average	0.14600	0.29426	0.78689	1.61329	3.40758
best case	0.13421	0.28798	0.76789	1.59936	3.29037
worst case	0.19591	0.35176	0.94432	1.65318	3.80031

The data shows running time in seconds.
100 trial runs of each sort for each size data set were conducted.
The data set consisted of random six-digit integers.

The data in this table is the result of actual benchmarking conducted using the algorithms in this chapter. It is included with the files for the chapter as **benchmarking.xlsx**

Chapter Review

Section 1 of this chapter discussed a scientific approach to determining the most efficient algorithm for a particular task, based on theory and experimentation. Key terms included *a priori* and *a posteriori.*

Section 2 described the quicksort algorithm, an example of a recursive divide and conquer algorithm.

Section 3 introduced a second divide and conquer sorting algorithm, merge sort.

Section 4 discussed the temporal complexity of sorting algorithms. Key terms included: *temporal complexity***,** *spatial complexity***,** *computational complexity theory*, and *recurrence relation*.

Section 5 introduced asymptotic notation and asymptotic analysis from number theory used by computer scientists in analyzing algorithmic complexity. Key terms included *asymptote* and *asymptotic notation* .

Section 6 was about the spatial complexity of sorting algorithms. Key terms included: *auxiliary data***,** *subroutine linkage***,** *stack***,** *stack frame* **,** *recursion overhead* **,** *linear recursion,* and *logarithmic recursion.*

Section 6 discussed the actual running time of software and factors that influence it. Software benchmarking was introduced.

Chapter Questions

1. What is the difference between *a priori* and *a posteriori* analysis of algorithms? How is this related to what you have been doing as you have been creating Java software for programming assignments?
2. How does quicksort work? Who developed quicksort, and for what purpose?
3. What is the pivot in quicksort used for? What method of selecting a quicksort pivot was suggested by at least one expert who studied quicksort?
4. How does merge sort work? Who developed merge sort, and when?
5. How does merge sort work with a data set whose size is not a power of 2? How does it work for with a data set whose size is an odd number?
6. What is a recursive divide and conquer algorithm? What are some examples of such algorithms?
7. What is meant by the concept of the temporal complexity of an algorithm? What is used to measure the theoretical running time of an algorithm?
8. What is a recurrence relation? How are recurrence relations helpful in analyzing the complexity of algorithms?
9. What aren't we using recurrence relations in this course? In the absence of recurrence relations, what can give us clues to the temporal complexity of an algorithm?
10. What does it mean to say that an algorithm has a temporal complexity that is linear? What is the difference in running time between an algorithm with a linear temporal complexity and one that has a temporal complexity of n^2?
11. How many times can we divide a set of 1,000 roughly in half before we end up with a single item? What math functions describes this? What is the temporal complexity of this process?
12. How does the function $f(n^2)$ grow compared to $f(n)$? How does the function $f(\log n)$ grow compared to $f(n)$?

13. What is an asymptote? What are the three forms of asymptotic notation?
14. How are Big O and Big omega notation used together to describe the temporal complexity of an algorithm? How is Big theta notation used to describe the running time of an algorithm?
15. What makes up auxiliary data? What does the size of auxiliary data depend on?
16. How does the spatial complexity of algorithms compare to the temporal complexity of algorithms? Which is more critical in determining if a program will run on a given computer?
17. What is an in-place algorithm? Give examples of sorting algorithms that are in-place and not in-place.
18. What is meant by the term recursion overhead? What makes up most recursion overhead?
19. What is linear recursion? What is logarithmic recursion? How does recursion overhead differ between linear recursive algorithms and logarithmically recursive algorithms?
20. What is meant by the "anatomy and physiology" of a computer system? What are some factors, in addition to the machine's "anatomy and physiology" that affect the actual running time of an algorithm?

Chapter Exercises

1. The choice of a pivot can affect the running time of quicksort. On page 7 in section 20.2 there are four suggestions for ways to select Quicksort pivots.

   Your task is to benchmark the four pivots selection methods listed. You should conduct trials with both randomly generated data, similar to what is done in the *MegaMergeSort()* program in this chapter, and also with data that is in order, both frontwards and backwards. In other words, in addition to testing the pivot choice for random data, test it for data that is already sorted, both in increasing and decreasing order.

   You should submit the code you used for benchmarking, along with a report of your results. You report should describe how you conducted the experiment.

2. This chapter and previous chapters describe five sorting algorithms – bubble sort, selection sort, insertion sort, merge sort and quicksort. This chapter provides the merge sort and quicksort code. The previous chapters provide the bubble sort code and pseudocode for selection sort and insertions sort. Your assignment for the last chapter was to create software for either insertion sort or selection sort.

   Your task here is to conduct benchmarking to compare the algorithms. You should try to run at least four of the five the sorting methods with random data sets of 10,000 20,000 100,000 200,000 1,000,000 and 2,000,000 items, 100 times each. You should stop the program for the first trial of any sorting method that takes more than a minute to run and report this result. If a sorting algorithm takes too long for one trial, such as for 100,000 items, then it is not necessary to complete the trials for larger data sets, simply report the size of the data set for which that algorithm started to take too long.

You should submit a report with your results and conclusions about the experiment. Your report should describe how you conducted the experiment.

3. The sorting methods shown in the textbook are forms of each algorithm written for clarity, not for maximum efficiency. Your task is to choose any sorting algorithm used in the textbook and see if you can change the algorithm to make the method run more quickly. You should conduct 100 sample runs of your method and the original method with several different sizes of data sets. You should report the average, worst case and best case time for each trial.

   You should not use other algorithms, but pick one from the book or from your previous work and modify it.

   Your report should describe how you improved the algorithm and where this idea came from.

4. About a dozen years ago the Python advocate Tim Peters developed *timsort*, a very fast sorting algorithm that is a hybrid algorithm, combining recursive and non-recursive sorting methods. Your task is to do three things:

   1. In your own words, describe the *timsort* algorithm, how it works, and how it compares to other sorting methods in the way it works.

   2. obtain or develop a Java implementation of the *timsort* algorithm and get it to work on a randomly generated array of integers.

   3. Conduct benchmarking similar to that described at the end of this chapter to compare the performance of *quicksort*, *merge sort* and *timsort*. Write a report with your findings.

5. As stated in the chapter, many different factors affect the running time of an algorithm. Your task is to conduct experiments to see how sorting algorithms perform in different environments.

   You should select conduct two sets of experiments:

   First, conduct benchmarking of quicksort and merge sort on at least four different computer systems. Try to determine how factors such as memory, operating system, etc. affect running time fo the algorithms.

   Second, conduct benchmarking of quicksort and merge sort several times on the same system - once with as much software turned off a possible, and then with other programs running - Word, Excel, videos, etc. See if you can determine how different software or combinations of software running at the same time slow down the sorting algorithms the most. You might want to include an Internet connection in this test. Just as with the different software, How does a live connection to a network affect the running time of the algorithms?

   You should submit a report describing your work, your results, and your conclusions.

Contents