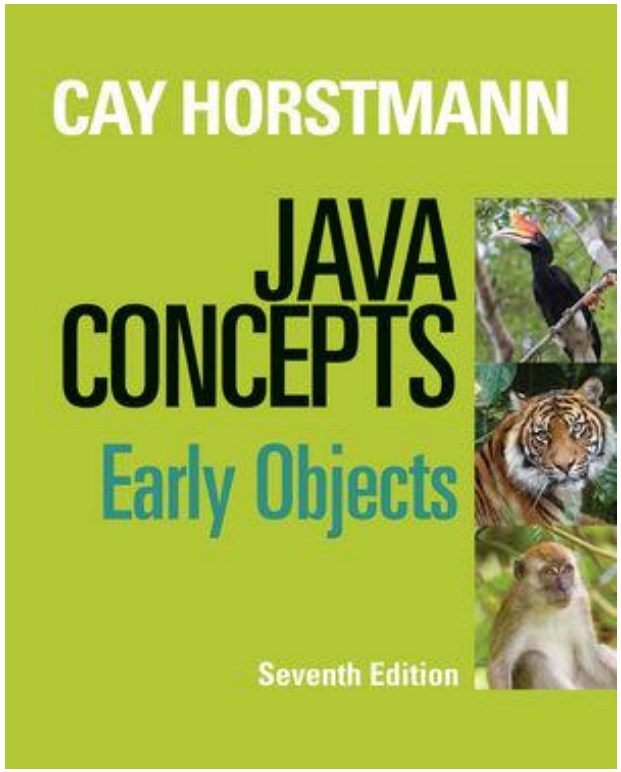
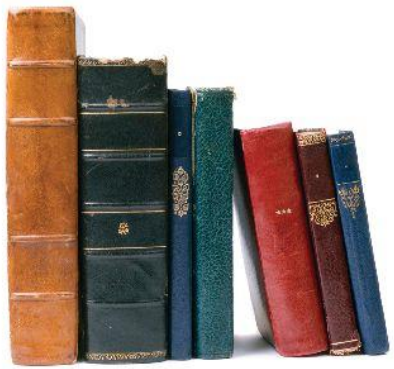


Chapter 14 – Sorting and Searching

Chapter Goals



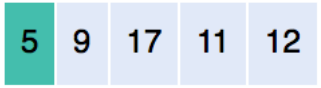
- To study several sorting and searching algorithms
- To appreciate that algorithms for the same task can differ widely in performance
- To understand the big-Oh notation
- To estimate and compare the performance of algorithms
- To write code to measure the running time of a program

Selection Sort

- A sorting algorithm rearranges the elements of a collection so that they are stored in sorted order.
- Selection sort sorts an array by repeatedly finding the smallest element of the unsorted tail region and moving it to the front.
- Slow when run on large data sets.
- Example: sorting an array of integers

Sorting an Array of Integers

1. Find the smallest and swap it with the first element



1. Find the next smallest. It is already in the correct place

1. Find the next smallest and swap it with first element of unsorted portion

2. Repeat

1. When the unsorted portion is of length 1, we are done

Selection Sort

© Zone Creative/iStockphoto.

In selection sort, pick the smallest element and swap it with the first one. Pick the smallest element of the remaining ones and swap it with the next one, and so on.

section_1/SelectionSorter.java

```
1  /**
2     The sort method of this class sorts an array, using the selection
3     sort algorithm.
4  */
5  public class SelectionSorter
6  {
7     /**
8        Sorts an array, using selection sort.
9        @param a the array to sort
10    */
11    public static void sort(int[] a)
12    {
13       for (int i = 0; i < a.length - 1; i++)
14       {
15          int minPos = minimumPosition(a, i);
16          ArrayUtil.swap(a, minPos, i);
17       }
18    }
19
```

Continued

section_1/SelectionSorter.java

```
20      /**
21         Finds the smallest element in a tail range of the array.
22         @param a the array to sort
23         @param from the first position in a to compare
24         @return the position of the smallest element in the
25         range a[from] . . . a[a.length - 1]
26      */
27      private static int minimumPosition(int[] a, int from)
28      {
29         int minPos = from;
30         for (int i = from + 1; i < a.length; i++)
31         {
32            if (a[i] < a[minPos]) { minPos = i; }
33         }
34         return minPos;
35      }
36   }
```

section_1/SelectionSortDemo.java

```
1  import java.util.Arrays;
2
3  /**
4     This program demonstrates the selection sort algorithm by
5     sorting an array that is filled with random numbers.
6  */
7  public class SelectionSortDemo
8  {
9     public static void main(String[] args)
10    {
11       int[] a = ArrayUtil.randomIntArray(20, 100);
12       System.out.println(Arrays.toString(a));
13
14       SelectionSorter.sort(a);
15
16       System.out.println(Arrays.toString(a));
17    }
18 }
19
20
```

Typical Program Run:

```
[65, 46, 14, 52, 38, 2, 96, 39, 14, 33, 13, 4, 24, 99, 89, 77, 73, 87, 36, 81]
[2, 4, 13, 14, 14, 24, 33, 36, 38, 39, 46, 52, 65, 73, 77, 81, 87, 89, 96, 99]
```

Self Check 14.1

Why do we need the `temp` variable in the `swap` method? What would happen if you simply assigned `a[i]` to `a[j]` and `a[j]` to `a[i]`?

Answer: Dropping the `temp` variable would not work. Then `a[i]` and `a[j]` would end up being the same value.

Self Check 14.2

What steps does the selection sort algorithm go through to sort the sequence 6 5 4 3 2 1?

Answer:

Self Check 14.3

How can you change the selection sort algorithm so that it sorts the elements in descending order (that is, with the largest element at the beginning of the array)?

Answer: In each step, find the maximum of the remaining elements and swap it with the current element (or see Self Check 4).

Self Check 14.4

Suppose we modified the selection sort algorithm to start at the end of the array, working toward the beginning. In each step, the current position is swapped with the minimum. What is the result of this modification?

Answer: The modified algorithm sorts the array in descending order.

Profiling the Selection Sort Algorithm

- We want to measure the time the algorithm takes to execute:
  - Exclude the time the program takes to load
  - Exclude output time
- To measure the running time of a method, get the current time immediately before and after the method call.
- We will create a `StopWatch` class to measure execution time of an algorithm:
  - It can start, stop and give elapsed time
  - Use `System.currentTimeMillis` method
- Create a `StopWatch` object:
  - Start the stopwatch just before the sort
  - Stop the stopwatch just after the sort
  - Read the elapsed time

section_2/StopWatch.java

```
1  /**
2     A stopwatch accumulates time when it is running. You can
3     repeatedly start and stop the stopwatch. You can use a
4     stopwatch to measure the running time of a program.
5  */
6  public class StopWatch
7  {
8     private long elapsedTime;
9     private long startTime;
10    private boolean isRunning;
11
12    /**
13       Constructs a stopwatch that is in the stopped state
14       and has no time accumulated.
15    */
16    public StopWatch()
17    {
18       reset();
19    }
20
```

Continued

section_2/StopWatch.java

```
21      /**
22          Starts the stopwatch. Time starts accumulating now.
23      */
24      public void start()
25      {
26          if (isRunning) { return; }
27          isRunning = true;
28          startTime = System.currentTimeMillis();
29      }
30
31      /**
32          Stops the stopwatch. Time stops accumulating and is
33          is added to the elapsed time.
34      */
35      public void stop()
36      {
37          if (!isRunning) { return; }
38          isRunning = false;
39          long endTime = System.currentTimeMillis();
40          elapsedTime = elapsedTime + endTime - startTime;
41      }
42
```

Continued

section_2/StopWatch.java

```
43      /**
44         Returns the total elapsed time.
45         @return the total elapsed time
46      */
47      public long getElapsedTime()
48      {
49         if (isRunning)
50         {
51            long endTime = System.currentTimeMillis();
52            return elapsedTime + endTime - startTime;
53         }
54         else
55         {
56            return elapsedTime;
57         }
58      }
59
60      /**
61         Stops the watch and resets the elapsed time to 0.
62      */
63      public void reset()
64      {
65         elapsedTime = 0;
66         isRunning = false;
67      }
68   }
```

section_2/SelectionSortTimer.java

```
1  import java.util.Scanner;
2
3  /**
4     This program measures how long it takes to sort an
5     array of a user-specified size with the selection
6     sort algorithm.
7  */
8  public class SelectionSortTimer
9  {
10    public static void main(String[] args)
11    {
12       Scanner in = new Scanner(System.in);
13       System.out.print("Enter array size: ");
14       int n = in.nextInt();
15
16       // Construct random array
17
18       int[] a = ArrayUtil.randomIntArray(n, 100);
19
```

Continued

section_2/SelectionSortTimer.java

```
20          // Use stopwatch to time selection sort
21
22          StopWatch timer = new StopWatch();
23
24          timer.start();
25          SelectionSorter.sort(a);
26          timer.stop();
27
28          System.out.println("Elapsed time: "
29                + timer.getElapsedTime() + " milliseconds");
30       }
31    }
32
33
```

Program Run:

```
Enter array size: 50000
Elapsed time: 13321 milliseconds
```

Selection Sort on Various Size Arrays

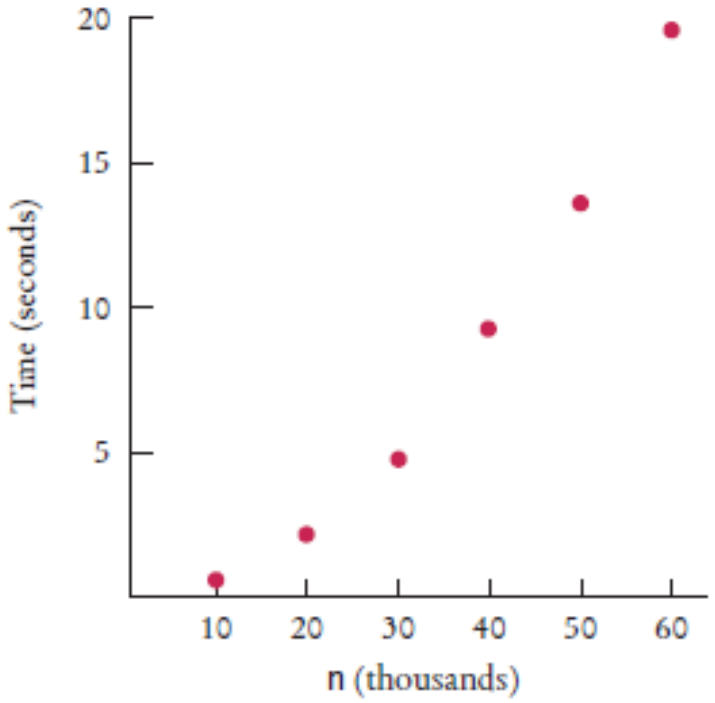


Figure 1 Time Taken by Selection Sort

n	Milliseconds
10,000	786
20,000	2,148
30,000	4,796
40,000	9,192
50,000	13,321
60,000	19,299

Doubling the size of the array more than doubles the time needed to sort it.

Self Check 14.5

Approximately how many seconds would it take to sort a data set of 80,000 values?

Answer: Four times as long as 40,000 values, or about 37 seconds.

Self Check 14.6

Look at the graph in Figure 1. What mathematical shape does it resemble?

Answer: A parabola.

Analyzing the Performance of the Selection Sort Algorithm

- In an array of size *n*, count how many times an array element is visited:
  - To find the smallest, visit *n* elements + 2 visits for the swap
  - To find the next smallest, visit *(n* - 1) elements + 2 visits for the swap
  - The last term is 2 elements visited to find the smallest + 2 visits for the swap

Analyzing the Performance of the Selection Sort Algorithm

- The number of visits:
  - $n + 2 + (n - 1) + 2 + (n - 2) + 2 + \ldots + 2 + 2$
  - This can be simplified to $n^2/2 + 5n/2 - 3$
  - $5n/2 - 3$ is small compared to $n^2/2$ – so let's ignore it
  - Also ignore the 1/2 – it cancels out when comparing ratios

Analyzing the Performance of the Selection Sort Algorithm

- The number of visits is of the order n^2 .
- Computer scientists use the big-Oh notation to describe the growth rate of a function.
- Using big-Oh notation: The number of visits is $O(n^2)$.
- Multiplying the number of elements in an array by **2** multiplies the processing time by **4**.
- To convert to big-Oh notation: locate fastest-growing term, and ignore constant coefficient.

Self Check 14.7

If you increase the size of a data set tenfold, how much longer does it take to sort it with the selection sort algorithm?

Answer: It takes about 100 times longer.

Self Check 14.8

How large does n need to be so that $1/2\ n^2$ is bigger than $5/2\ n - 3$?

Answer: If n is 4, then n^2 is 8 and $5/2\ n - 3$ is 7.

Self Check 14.9

Section 7.3.6 has two algorithms for removing an element from an array of length n. How many array visits does each algorithm require on average?

Answer: The first algorithm requires one visit, to store the new element. The second algorithm requires $T(p) = 2 \times (n - p - 1)$ visits, where p is the location at which the element is removed. We don't know where that element is, but if elements are removed at random locations, on average, half of the removals will be above the middle and half below, so we can assume an average p of $n / 2$ and $T(n) = 2 \times (n - n / 2 - 1) = n - 2$.

Self Check 14.10

Describe the number of array visits in Self Check 9 using the big-Oh notation.

Answer: The first algorithm is $O(1)$, the second $O(n)$.

Self Check 14.11

What is the big-Oh running time of checking whether an array is already sorted?

Answer: We need to check that `a[0]` ≤ `a[1]`, `a[1]` ≤ `a[2]`, and so on, visiting $2n - 2$ elements. Therefore, the running time is $O(n)$.

Self Check 14.12

Consider this algorithm for sorting an array. Set *k* to the length of the array. Find the maximum of the first *k* elements. Remove it, using the second algorithm of Section 7.3.6. Decrement *k* and place the removed element into the k^{th} position. Stop if *k* is 1. What is the algorithm's running time in big-Oh notation?

Continued

Self Check 14.12

Answer: Let n be the length of the array. In the kth step, we need k visits to find the minimum. To remove it, we need an average of $k - 2$ visits (see Self Check 9). One additional visit is required to add it to the end. Thus, the kth step requires $2k - 1$ visits. Because k goes from n to 2, the total number of visits is

$$2n - 1 + 2(n - 1) - 1 + \ldots + 2 \cdot 3 - 1 + 2 \cdot 2 - 1 =$$
$$2(n + (n - 1) + \ldots + 3 + 2 + 1 - 1) - (n - 1) =$$
$$n(n + 1) - 2 - n + 1 = n^2 - 3$$

(because $1 + 2 + 3 + \ldots + (n - 1) + n = n(n + 1) / 2$)

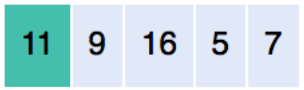
Therefore, the total number of visits is $O(n^2)$.

Common Big-Oh Growth Rates

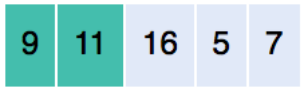
Table 1 Common Big-Oh Growth Rates	
Big-Oh Expression	Name
$O(1)$	Constant
$O(\log(n))$	Logarithmic
$O(n)$	Linear
$O(n \log(n))$	Log-linear
$O(n^2)$	Quadratic
$O(n^3)$	Cubic
$O(2^n)$	Exponential
$O(n!)$	Factorial

Insertion Sort

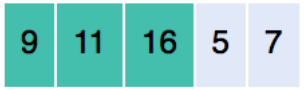
- Assume initial sequence `a[0] . . . a[k]` is sorted (`k = 0`):



- Add `a[1]`; element needs to be inserted before 11



- Add `a[2]`



- Add `a[3]`

- Finally, add `a[4]`

Insertion Sort

```
public class InsertionSorter
{
   /**
      Sorts an array, using insertion sort.
      @param a the array to sort
   */
   public static void sort(int[] a)
   {
      for (int i = 1; i < a.length; i++)
      {
         int next = a[i];
         // Move all larger elements up
         int j = i;
         while (j > 0 && a[j - 1] > next)
         {
            a[j] = a[j - 1];
            j--;
         }
         // Insert the element
         a[j] = next;
      }
   }
}
```

Insertion Sort

- Insertion sort is the method that many people use to sort playing cards. Pick up one card at a time and insert it so that the cards stay sorted.

© Kirby Hamilton/iStockphoto.

- Insertion sort is an $O(n^2)$ algorithm.

Merge Sort

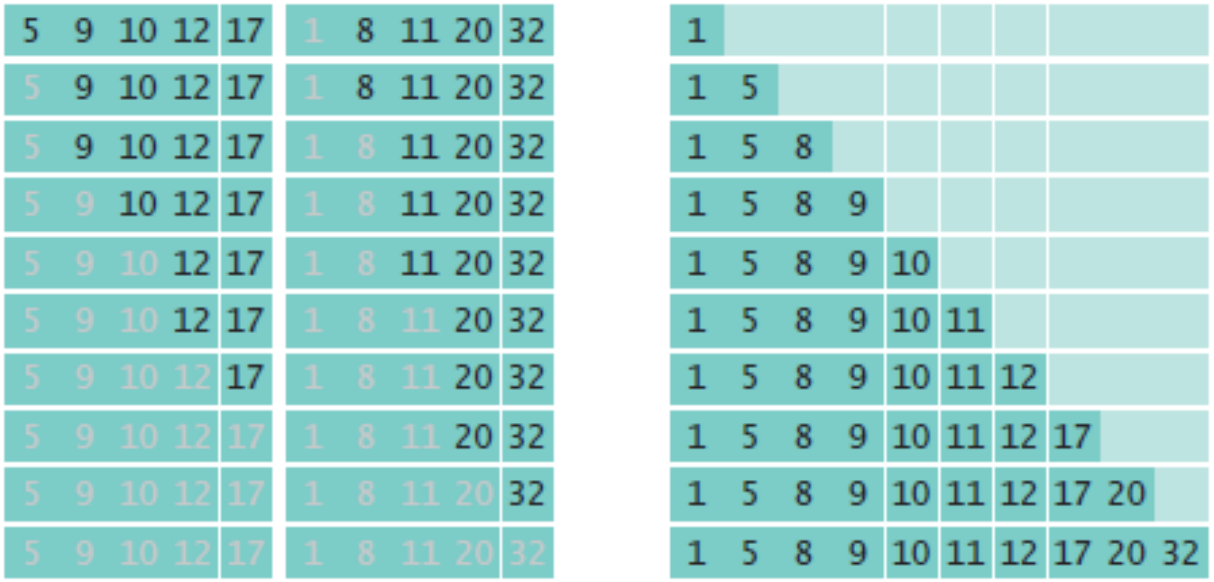
- Sorts an array by
  - Cutting the array in half
  - Recursively sorting each half
  - Merging the sorted halves
- Dramatically faster than the selection sort In merge sort, one sorts each half, then merges the sorted halves.

© Rich Legg/iStockphoto.

Merge Sort Example

- Divide an array in half and sort each half

- Merge the two sorted arrays into a single sorted array



Merge Sort

```
public static void sort(int[] a)
{
   if (a.length <= 1) { return; }
   int[] first = new int[a.length / 2];
   int[] second = new int[a.length - first.length];
   // Copy the first half of a into first, the second half into second
   . . .
   sort(first);
   sort(second);
   merge(first, second, a);
}
```

section_4/MergeSorter.java

```
1  /**
2     The sort method of this class sorts an array, using the merge
3     sort algorithm.
4  */
5  public class MergeSorter
6  {
7     /**
8        Sorts an array, using merge sort.
9        @param a the array to sort
10    */
11    public static void sort(int[] a)
12    {
13       if (a.length <= 1) { return; }
14       int[] first = new int[a.length / 2];
15       int[] second = new int[a.length - first.length];
16       // Copy the first half of a into first, the second half into second
17       for (int i = 0; i < first.length; i++)
18       {
19          first[i] = a[i];
20       }
21       for (int i = 0; i < second.length; i++)
22       {
23          second[i] = a[first.length + i];
24       }
25       sort(first);
26       sort(second);
27       merge(first, second, a);
28    }
29
```

Continued

section_4/MergeSorter.java

```
30      /**
31         Merges two sorted arrays into an array
32         @param first the first sorted array
33         @param second the second sorted array
34         @param a the array into which to merge first and second
35      */
36      private static void merge(int[] first, int[] second, int[] a)
37      {
38         int iFirst = 0; // Next element to consider in the first array
39         int iSecond = 0; // Next element to consider in the second array
40         int j = 0; // Next open position in a
41
42         // As long as neither iFirst nor iSecond is past the end, move
43         // the smaller element into a
44         while (iFirst < first.length && iSecond < second.length)
45         {
46            if (first[iFirst] < second[iSecond])
47            {
48               a[j] = first[iFirst];
49               iFirst++;
50            }
51            else
52            {
53               a[j] = second[iSecond];
54               iSecond++;
55            }
56            j++;
57         }
58
```

Continued

section_4/MergeSorter.java

```
59          // Note that only one of the two loops below copies entries
60          // Copy any remaining entries of the first array
61          while (iFirst < first.length)
62          {
63             a[j] = first[iFirst];
64             iFirst++; j++;
65          }
66          // Copy any remaining entries of the second half
67          while (iSecond < second.length)
68          {
69             a[j] = second[iSecond];
70             iSecond++; j++;
71          }
72       }
73    }
```

section_4/MergeSortDemo.java

```
1  import java.util.Arrays;
2
3  /**
4     This program demonstrates the merge sort algorithm by
5     sorting an array that is filled with random numbers.
6  */
7  public class MergeSortDemo
8  {
9     public static void main(String[] args)
10    {
11       int[] a = ArrayUtil.randomIntArray(20, 100);
12       System.out.println(Arrays.toString(a));
13
14       MergeSorter.sort(a);
15
16       System.out.println(Arrays.toString(a));
17    }
18 }
19
```

Typical Program Run:

```
[8, 81, 48, 53, 46, 70, 98, 42, 27, 76, 33, 24, 2, 76, 62, 89, 90, 5, 13, 21]
[2, 5, 8, 13, 21, 24, 27, 33, 42, 46, 48, 53, 62, 70, 76, 76, 81, 89, 90, 98]
```

Self Check 14.13

Why does only one of the two `while` loops at the end of the `merge` method do any work?

Answer: When the preceding `while` loop ends, the loop condition must be `false`, that is, `iFirst >= first.length` or `iSecond >= second.length` (De Morgan's Law).

Self Check 14.14

Manually run the merge sort algorithm on the array 8 7 6 5 4 3 2 1.

Answer:

First sort 8 7 6 5.
Recursively, first sort 8 7.
Recursively, first sort 8. It's sorted.
Sort 7. It's sorted.
Merge them: 7 8.
Do the same with 6 5 to get 5 6.
Merge them to 5 6 7 8.
Do the same with 4 3 2 1: Sort 4 3 by sorting 4 and 3 and merging them to 3 4.
Sort 2 1 by sorting 2 and 1 and merging them to 1 2.
Merge 3 4 and 1 2 to 1 2 3 4.
Finally, merge 5 6 7 8 and 1 2 3 4 to 1 2 3 4 5 6 7 8.

Self Check 14.15

The merge sort algorithm processes an array by recursively processing two halves. Describe a similar recursive algorithm for computing the sum of all elements in an array.

Answer: If the array size is 1, return its only element as the sum. Otherwise, recursively compute the sum of the first and second subarray and return the sum of these two values.

Analyzing the Merge Sort Algorithm

- In an array of size n, count how many times an array element is visited.
- Assume n is a power of 2: $n = 2^m$.
- Calculate the number of visits to create the two sub-arrays and then merge the two sorted arrays:
  - 3 visits to merge each element or $3n$ visits
  - $2n$ visits to create the two sub-arrays
  - total of $5n$ visits

Analyzing the Merge Sort Algorithm

- Let $T(n)$ denote the number of visits to sort an array of n elements then
  - $T(n) = T(n / 2) + T(n / 2) + 5n$ or
  - $T(n) = 2T(n / 2) + 5n$
- The visits for an array of size $n / 2$ is: $T(n / 2) = 2T(n / 4) + 5\ n / 2$
  - So $T(n) = 2 \times 2T(\ n /4) +5n + 5n$
- The visits for an array of size $n / 4$ is: $T(n / 4) = 2T(n / 8) + 5\ n / 4$
  - So $T(n) = 2 \times 2 \times 2T(n / 8) + 5n + 5n + 5n$

Analyzing the Merge Sort Algorithm

- Repeating the process k times: $T(n) = 2^k T(n / 2^k) + 5nk$
- Since $n = 2^m$, when $k = m$: $T(n) = 2^m T(n / 2^m) + 5nm$
- $T(n) = nT(1) + 5nm$
- $T(n) = n + 5n\log_2(n)$

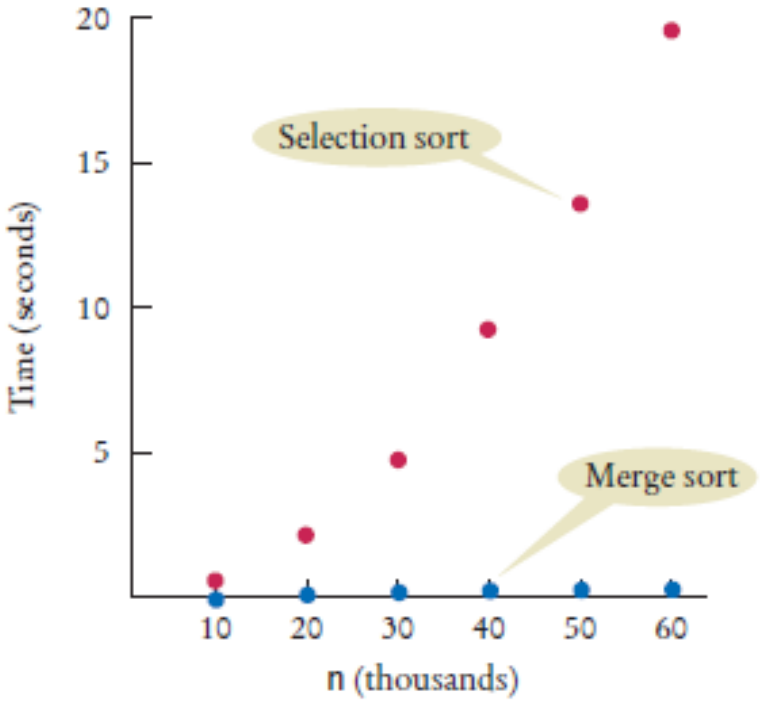
Analyzing the Merge Sort Algorithm

- To establish growth order:
  - Drop the lower-order term n
  - Drop the constant factor 5
  - Drop the base of the logarithm since all logarithms are related by a constant factor
  - We are left with $n \log(n)$
- Using big-Oh notation: number of visits is $O(n \log(n))$.

Merge Sort Vs Selection Sort

- Selection sort is an $O(n^2)$ algorithm.
- Merge sort is an $O(n \log(n))$ algorithm.
- The $n \log(n)$ function grows much more slowly than n^2.

Merge Sort Timing vs. Selection Sort



n	Merge Sort (milliseconds)	Selection Sort (milliseconds)
10,000	40	786
20,000	73	2,148
30,000	134	4,796
40,000	170	9,192
50,000	192	13,321
60,000	205	19,299

Self Check 14.16

Given the timing data for the merge sort algorithm in the table at the beginning of this section, how long would it take to sort an array of 100,000 values?

Answer: Approximately 100,000 × log(100,000) / 50,000 × log(50,000) = 2 × 5 / 4.7 = 2.13 times the time required for 50,000 values. That's 2.13 × 97 milliseconds or approximately 409 milliseconds.

Self Check 14.17

If you double the size of an array, how much longer will the merge sort algorithm take to sort the new array?

Answer: $(2n \log(2n) / n \log(n)) = 2(1 + \log(2) / \log(n))$.
For $n > 2$, that is a value < 3.

The Quicksort Algorithm

- No temporary arrays are required.
  1. Divide and conquer Partition the range
  2. Sort each partition
- In quicksort, one partitions the elements into two groups, holding the smaller and larger elements. Then one sorts each group.

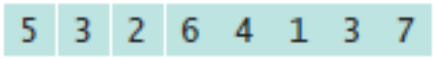
© Christopher Futcher/iStockphoto.

The Quicksort Algorithm

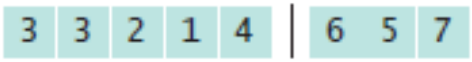
```
public void sort(int from, int to)
{
   if (from >= to) return;
   int p = partition(from, to);
   sort(from, p);
   sort(p + 1, to);
}
```

The Quicksort Algorithm

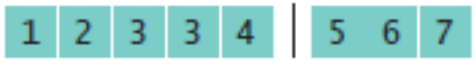
- Starting range



- A partition of the range so that no element in first section is larger than element in second section



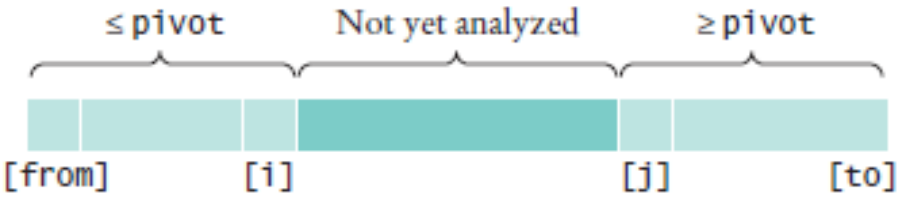
- Recursively apply the algorithm until array is sorted



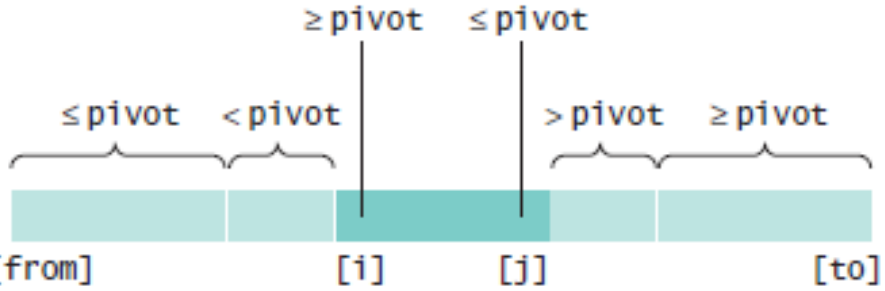
The Quicksort Algorithm

```
private static int partition(int[] a, int from, int to)
{
    int pivot = a[from];
    int i = from - 1;
    int j = to + 1;
    while (i &lt; j)
    {
        i++; while (a[i] < pivot) { i++; }
        j--; while (a[j] > pivot) { j--; }
        if (i &lt; j) { ArrayUtil.swap(a, i, j); }
    }
    return j;
}
```

The Quicksort Algorithm – Partioning



Partitioning a Range



Extending the Partitions

The Quicksort Algorithm

- On average, the quicksort algorithm is an $O(n \log(n))$ algorithm.
- Its worst-case run-time behavior is $O(n^2)$.
- If the pivot element is chosen as the first element of the region,
  - That worst-case behavior occurs when the input set is already sorted

Searching

- **Linear search:** also called **sequential search**
- Examines all values in an array until it finds a match or reaches the end
- Number of visits for a linear search of an array of *n* elements:
  - The average search visits *n*/2 elements
  - The maximum visits is *n*
- A linear search locates a value in an array in *O*(*n*) steps

section_6_1/LinearSearcher.java

```
1  /**
2     A class for executing linear searches in an array.
3  */
4  public class LinearSearcher
5  {
6     /**
7        Finds a value in an array, using the linear search
8        algorithm.
9        @param a the array to search
10       @param value the value to find
11       @return the index at which the value occurs, or -1
12       if it does not occur in the array
13    */
14    public static int search(int[] a, int value)
15    {
16       for (int i = 0; i < a.length; i++)
17       {
18          if (a[i] == value) { return i; }
19       }
20       return -1;
21    }
22 }
```

section_6_1/LinearSearchDemo.java

```
1  import java.util.Arrays;
2  import java.util.Scanner;
3
4  /**
5     This program demonstrates the linear search algorithm.
6  */
7  public class LinearSearchDemo
8  {
9     public static void main(String[] args)
10    {
11       int[] a = ArrayUtil.randomIntArray(20, 100);
12       System.out.println(Arrays.toString(a));
13       Scanner in = new Scanner(System.in);
14
15       boolean done = false;
16       while (!done)
17       {
18          System.out.print("Enter number to search for, -1 to quit: ");
19          int n = in.nextInt();
20          if (n == -1)
21          {
22             done = true;
23          }
24          else
25          {
26             int pos = LinearSearcher.search(a, n);
27             System.out.println("Found in position " + pos);
28          }
29       }
30    }
31 }
```

Continued

section_6_1/LinearSearchDemo.java

Program Run:

```
[46, 99, 45, 57, 64, 95, 81, 69, 11, 97, 6, 85, 61, 88, 29, 65, 83, 88, 45, 88]
Enter number to search for, -1 to quit: 12
Found in position -1
Enter number to search for, -1 to quit: -1
```

Self Check 14.11

Suppose you need to look through 1,000,000 records to find a telephone number. How many records do you expect to search before finding the number?

Answer: On average, you'd make 500,000 comparisons.

Self Check 14.12

Why can't you use a “for each” loop

```
for (int element : a)
```

in the search method?

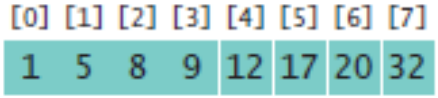
Answer: The search method returns the index at which the match occurs, not the data stored at that location.

Binary Search

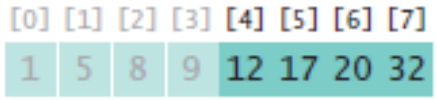
- A binary search locates a value in a **sorted** array by:
  - Determining whether the value occurs in the first or second half
  - Then repeating the search in one of the halves
- The size of the search is cut in half with each step.

Binary Search

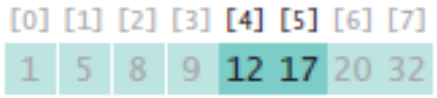
- Searching for 15 in this array



- The last value in the first half is 9
  - So look in the second (darker colored) half

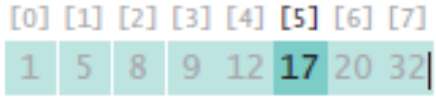


- The last value of the first half of this sequence is 17
  - Look in the darker colored sequence



Binary Search

- The last value of the first half of this very short sequence is 12,
  - This is smaller than the value that we are searching,
  - so we must look in the second half



- 15 ≠ 17: we don't have a match

section_6_2/BinarySearcher.java

```
1  /**
2     A class for executing binary searches in an array.
3  */
4  public class BinarySearcher
5  {
6     /**
7        Finds a value in a range of a sorted array, using the binary
8        search algorithm.
9        @param a the array in which to search
10       @param low the low index of the range
11       @param high the high index of the range
12       @param value the value to find
13       @return the index at which the value occurs, or -1
14       if it does not occur in the array
15    */
```

Continued

section_6_2/BinarySearcher.java

```
16      public static int search(int[] a, int low, int high, int value)
17      {
18         if (low <= high)
19         {
20            int mid = (low + high) / 2;
21
22            if (a[mid] == value)
23            {
24               return mid;
25            }
26            else if (a[mid] < value )
27            {
28               return search(a, mid + 1, high, value);
29            }
30            else
31            {
32               return search(a, low, mid - 1, value);
33            }
34         }
35         else
36         {
37            return -1;
38         }
39      }
40   }
41
```

Binary Search

- Count the number of visits to search a sorted array of size n
  - We visit one element (the middle element) then search either the left or right subarray
  - Thus: $T(n) = T(n/2) + 1$
- If n is $n / 2$, then $T(n / 2) = T(n / 4) + 1$
- Substituting into the original equation: $T(n) = T(n / 4) + 2$
- This generalizes to: $T(n) = T(n / 2^k) + k$

Binary Search

- Assume n is a power of 2, $n = 2^m$ where $m = \log_2(n)$
- Then: $T(n) = 1 + \log_2(n)$
- A binary search locates a value in a sorted array in $O(\log(n))$ steps.

Binary Search

- Should we sort an array before searching?
  - Linear search - O(n)
  - Binary search - O(n log(n))
- If you search the array only once
  - Linear search is more efficient
- If you will make many searches
  - Worthwhile to sort and use binary search

Self Check 14.18

Suppose you need to look through 1,000,000 records to find a telephone number. How many records do you expect to search before finding the number?

Answer: On average, you'd make 500,000 comparisons.

Self Check 14.19

Why can't you use a "for each" loop

```
for (int element : a)
```

in the `search` method?

Answer: The `search` method returns the index at which the match occurs, not the data stored at that location.

Self Check 14.20

Suppose you need to look through a sorted array with 1,000,000 elements to find a value. Using the binary search algorithm, how many records do you expect to search before finding the value?

Answer: You would search about 20. (The binary log of 1,024 is 10.)

Problem Solving: Estimating the Running Time of an Algorithm – Linear time

- Example: an algorithm that counts how many elements have a particular value

```
int count = 0;
for (int i = 0; i < a.length; i++)
{
   if (a[i] == value) { count++; }
}
```

Problem Solving: Estimating the Running Time of an Algorithm – Linear Time

- Pattern of array element visits

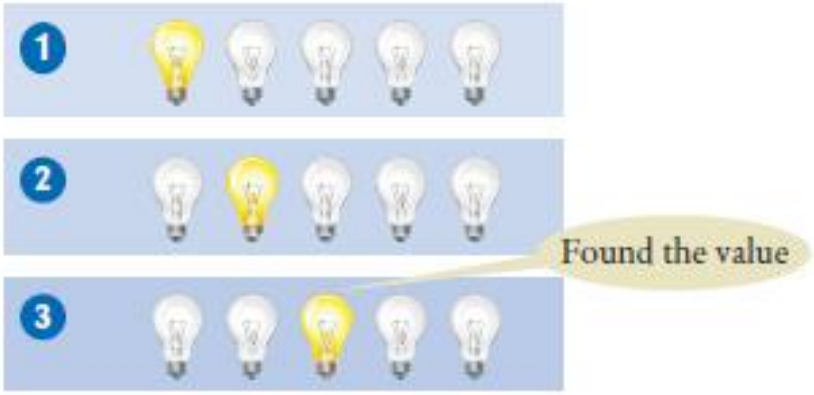
- There are a fixed number of actions in each visit independent of *n*.
- A loop with *n* iterations has *O(n)* running time if each step consists of a fixed number of actions.

Problem Solving: Estimating the Running Time of an Algorithm – Linear Time

- Example: an algorithm to determine if a value occurs in the array

```
boolean found = false;
for (int i = 0; !found && i < a.length; i++)
{
   if (a[i] == value) { found = true; }
}
```

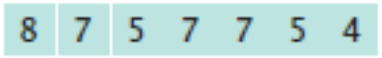
- Search may stop in the middle



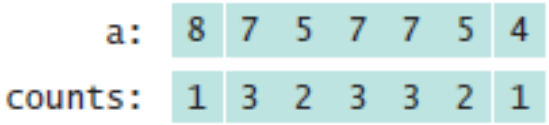
- Still $O(n)$ because we may have to traverse the whole array.

Problem Solving: Estimating the Running Time of an Algorithm – Quadratic Time

- Problem: Find the most frequent element in an array.
- Try it with this array



- Count how often each element occurs.
  - Put the counts in an array



  - Find the maximum count
  - It is 3 and the corresponding value in original array is 7

Problem Solving: Estimating the Running Time of an Algorithm – Quadratic Time

- Estimate how long it takes to compute the counts

```
for (int i = 0; i < a.length; i++)
{
   counts[i] = Count how often a[i] occurs in a
}
```

  - We visit each array element once - $O(n)$
  - Count the number of times that element occurs - $O(n)$
  - Total running time - $O(n^2)$

Problem Solving: Estimating the Running Time of an Algorithm – Quadratic Time

- Three phases in the algorithm
  - Compute all counts. $O(n^2)$
  - Compute the maximum. $O(n)$
  - Find the maximum in the counts. $O(n)$
- A loop with n iterations has $O(n^2)$ running time if each step takes $O(n)$ time. The big-Oh running time for doing several steps in a row is the largest of the big-Oh times for each step.

The Triangle Pattern

- Try to speed up the algorithm for finding the most frequent element.
- Idea - Before counting an element, check that it didn't already occur in the array
  - At each step, the work is *O(i)*
  - In the third iteration, visit a[0] and a[1] again

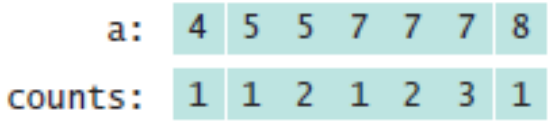
The Triangle Pattern

- $n^2/2$ lightbulbs are visited (light up)
- That is still $O(n^2)$
- A loop with n iterations has $O(n^2)$ running time if the i^{th} step takes $O(i)$ time.

Problem Solving: Estimating the Running Time of an Algorithm – Logarithmic Time

- Logarithmic time estimates arise from algorithms that cut work in half in each step.
- Another idea for finding the most frequent element in an array:
  - Sort the array first

  - This is *O(n log(n))* time
- Traverse the array and count how many times you have seen that element:



Problem Solving: Estimating the Running Time of an Algorithm – Logarithmic Time

- The code

```
int count = 0;
for (int i = 0; i < a.length; i++)
{
   count++;
   if (i == a.length - 1 || a[i] != a[i + 1])
   {
      counts[i] = count;
      count = 0;
   }
}
```

Problem Solving: Estimating the Running Time of an Algorithm – Logarithmic Time

- This takes the same amount of work per iteration:
  - visits two elements
  - *2n* which is *O(n)*

- Running time of entire algorithm is $O(n \log(n))$.
- An algorithm that cuts the size of work in half in each step runs in $O(\log(n))$ time.

Self Check 14.21

What is the "light bulb pattern" of visits in the following algorithm to check whether an array is a palindrome?

```
  for (int i = 0; i < a.length / 2; i++)
{
   if (a[i] != a[a.length - 1 - i]) { return false; }
}
return true;
```

Answer:

Self Check 14.22

What is the big-Oh running time of the following algorithm to check whether the first element is duplicated in an array?

```
for (int i = 1; i < a.length; i++)
{
   if (a[0] == a[i]) { return true; }
}
return false;
```

Answer: It is an $O(n)$ algorithm.

Self Check 14.23

What is the big-Oh running time of the following algorithm to check whether an array has a duplicate value?

```
for (int i = 0; i < a.length; i++)
{
   for (j = i + 1; j < a.length; j++)
   {
      if (a[i] == a[j]) { return true; }
   }
}
return false;
```

Answer: It is an $O(n^2)$ algorithm—the number of visits follows a triangle pattern.

Self Check 14.24

Describe an $O(n \log(n))$ algorithm for checking whether an array has duplicates.

Answer: Sort the array, then make a linear scan to check for adjacent duplicates.

Self Check 14.25

What is the big-Oh running time of the following algorithm to find an element in an $n \times n$ array?

```
for (int i = 0; i < n; i++)
{
   for (j = 0; j < n; j++)
   {
      if (a[i][j] == value) { return true; }
   }
}
return false;
```

Answer: It is an $O(n^2)$ algorithm—the outer and inner loops each have n iterations.

Self Check 14.26

If you apply the algorithm of Section 14.7.4 to an $n \times n$ array, what is the big-Oh efficiency of finding the most frequent element in terms of n?

Answer: Because an $n \times n$ array has $m = n^2$ elements, and the algorithm in Section 14.7.4, when applied to an array with m elements, is $O(m \log(m))$, we have an $O(n^2 \log(n))$ algorithm. Recall that $\log(n^2) = 2 \log(n)$, and the factor of 2 is irrelevant in the big-Oh notation.

Sorting and Searching in the Java Library – Sorting

- You do not need to write sorting and searching algorithms
  - Use methods in the Arrays and Collections classes
- The `Arrays` class contains static sort methods.
- To sort an array of integers:

```
int[] a = . . . ;
Arrays.sort(a);
```

  - That sort method uses the Quicksort algorithm (see Special Topic 14.3).
- To sort an `ArrayList`, use `Collections.sort`

```
ArrayList<String> names = . . .;
Collections.sort(names);
```

  - Uses merge sort algorithm

Sorting and Searching in the Java Library – Binary Search

- `Arrays` and `Collections` classes contain static `binarySearch` methods.
  - If the element is not found, returns *-k* - 1
  - Where *k* is the position before which the element should be inserted
- For example

```
int[] a = { 1, 4, 9 };
int v = 7;
int pos = Arrays.binarySearch(a, v);
// Returns -3; v should be inserted before position 2
```

Comparing Objects

- `Arrays.sort` sorts objects of classes that implement `Comparable` interface:

```
public interface Comparable
{
   int compareTo(Object otherObject);
}
```

- The call `a.compareTo(b)` returns
  - A negative number if `a` should come before `b`
  - 0 if `a` and `b` are the same
  - A positive number otherwise

Comparing Objects

- Several classes in Java (e.g. `String` and `Date`) implement `Comparable`.
- You can implement `Comparable` interface for your own classes.
- The `Country` class could implement `Comparable`:

```
public class Country implements Comparable
{
   public int compareTo(Object otherObject)
   {
      Country other = (Country) otherObject;
      if (area < other.area) { return -1; }
      else if (area == other.area) { return 0; }
      else { return 1; }
   }
}
```

Comparing Objects

- You could pass an array of countries to `Arrays.sort`

```
Country[] countries = new Country[n];
// Add countries
Arrays.sort(countries); // Sorts by increasing area
```

Self Check 14.27

Why can't the `Arrays.sort` method sort an array of `Rectangle` objects?

Answer: The `Rectangle` class does not implement the `Comparable` interface.

Self Check 14.28

What steps would you need to take to sort an array of `BankAccount` objects by increasing balance?

Answer: The `BankAccount` class needs to implement the `Comparable` interface. Its `compareTo` method must compare the bank balances.

Self Check 14.29

Why is it useful that the `Arrays.binarySearch` method indicates the position where a missing element should be inserted?

Answer: Then you know where to insert it so that the array stays sorted, and you can keep using binary search.

Self Check 14.30

Why does `Arrays.binarySearch` return $-k - 1$ and not $-k$ to indicate that a value is not present and should be inserted before position k?

Answer: Otherwise, you would not know whether a value is present when the method returns 0.