D06
PROGRAMMING with JAVA

Ch19 – Sorting & Searching

PowerPoint presentation, created by Angel A. Juan - ajuanp(@)gmail.com,
for accompanying the book "Big Java", by Cay S. Horstmann (2006)

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 learn how to estimate and compare the performance of algorithms
- To learn how to measure the running time of a program

Sort Algorithm #1: Selection Sort

- A sorting algorithm rearranges the elements of a collection so that they are stored in sorted order.
- Example: Consider the following array of integers: 11 9 17 5 12

  Find the smallest and swap it with the first element: 5 9 17 11 12

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

  5 9 17 11 12

  Find the next smallest and swap it with first element of unsorted portion: 5 9 11 17 12

  Repeat: 5 9 11 12 17

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

  5 9 11 12 17

File SelectionSorter.java

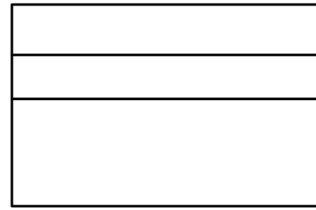
```
01: /**
02:    This class sorts an array, using the selection sort
03:    algorithm
04: */
05: public class SelectionSorter
06: {
07:    /**
08:       Constructs a selection sorter.
09:       @param anArray the array to sort
10:    */
11:    public SelectionSorter(int[] anArray)
12:    {
13:       a = anArray;
14:    }
15:
16:    /**
17:       Sorts the array managed by this selection sorter.
18:    */
```

This class encapsulates a sorting algorithm. Whenever we want to sort an array of integers, we will do the following:

1. To create a new object of this class, passing to it the array to be sort
2. To invoke the `sort` method on this new object

Continued

File SelectionSorter.java

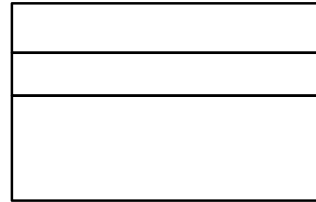


```
19:     public void sort()
20:     {
21:        for (int i = 0; i < a.length - 1; i++)
22:        {
23:           int minPos = minimumPosition(i);
24:           swap(minPos, i);
25:        }
26:     }
27:
28:     /**
29:        Finds the smallest element in a tail range of the array.
30:        @param from the first position in a to compare
31:        @return the position of the smallest element in the
32:        range a[from] . . . a[a.length - 1]
33:     */
34:     private int minimumPosition(int from)
35:     {
```

The public method sort uses two private methods: minimumPosition() and swap()

Continued

File SelectionSorter.java



```
36:         int minPos = from;
37:         for (int i = from + 1; i < a.length; i++)
38:            if (a[i] < a[minPos]) minPos = i;
39:         return minPos;
40:      }
41:
42:      /**
43:         Swaps two entries of the array.
44:         @param i the first position to swap
45:         @param j the second position to swap
46:      */
47:      private void swap(int i, int j)
48:      {
49:         int temp = a[i];
50:         a[i] = a[j];
51:         a[j] = temp;
52:      }
```

```
53:
54:      private int[] a;
55:  }
```

The array to be sorted is stored as a class instance field (see the class constructor)

File `SelectionSortTester.java`

```
01: /**
02:    This program tests the selection sort algorithm by
03:    sorting an array that is filled with random numbers.
04: */
05: public class SelectionSortTester
06: {
07:    public static void main(String[] args)
08:    {
09:       int[] a = ArrayUtil.randomIntArray(20, 100);
10:       ArrayUtil.print(a);
11:
12:       SelectionSorter sorter = new SelectionSorter(a);
13:       sorter.sort();
14:
15:       ArrayUtil.print(a);
16:    }
17: }
18:
19:
```

The `ArrayUtil` class will be implemented next

File ArrayUtil.java

ArrayUtil
-generator
+randomIntArray() : int +print()

```
01: import java.util.Random;
02:
03: /**
04:    This class contains utility methods for array
05:    manipulation.
06: */
07: public class ArrayUtil
08: {
09:    /**
10:       Creates an array filled with random values.
11:       @param length the length of the array
12:       @param n the number of possible random values
13:       @return an array filled with length numbers between
14:       0 and n - 1
15:    */
16:    public static int[] randomIntArray(int length, int n)
17:    {
```

Continued

File ArrayUtil.java


```
18:         int[] a = new int[length];
19:         for (int i = 0; i < a.length; i++)
20:            a[i] = generator.nextInt(n);
21: 
22:         return a;
23:      }
24: 
25:      /**
26:         Prints all elements in an array.
27:         @param a the array to print
28:      */
29:      public static void print(int[] a)
30:      {
```

Continued

File ArrayUtil.java

ArrayUtil
-generator
+randomIntArray() : int +print()

```
31:         for (int e : a)
32:            System.out.print(e + "  ");
33:         System.out.println();
34:      }
35:
36:      private static Random generator = new Random();
37:   }
38:
```

Note that instantiation takes place here. An alternative way would be to instantiate inside the `randomIsArray()` method

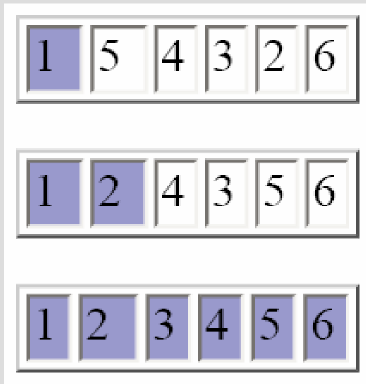
Output:

```
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

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

Answers

1. Dropping the temp variable would not work. Then `a[i]` and `a[j]` would end up being the same value.
2. 

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
- 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

File StopWatch.java


```
01: /**
02:     A stopwatch accumulates time when it is running. You can
03:     repeatedly start and stop the stopwatch. You can use a
04:     stopwatch to measure the running time of a program.
05: */
06: public class StopWatch
07: {
08:    /**
09:       Constructs a stopwatch that is in the stopped state
10:       and has no time accumulated.
11:    */
12:    public StopWatch()
13:    {
14:       reset();
15:    }
16:
```

Continued

File StopWatch.java

StopWatch
-elapsedTime : long -startTime : long -isRunning : bool
+start() +stop() +getElapsedTime() : long +reset()

```
17:     /**
18:        Starts the stopwatch. Time starts accumulating now.
19:     */
20:     public void start()
21:     {
22:        if (isRunning) return;
23:        isRunning = true;
24:        startTime = System.currentTimeMillis();
25:     }
26:
27:     /**
28:        Stops the stopwatch. Time stops accumulating and is
29:        is added to the elapsed time.
30:     */
31:     public void stop()
32:     {
```

Continued

File StopWatch.java


```
33:        if (!isRunning) return;
34:        isRunning = false;
35:        long endTime = System.currentTimeMillis();
36:        elapsedTime = elapsedTime + endTime - startTime;
37:     }
38:
39:     /**
40:        Returns the total elapsed time.
41:        @return the total elapsed time
42:     */
43:     public long getElapsedTime()
44:     {
45:        if (isRunning)
46:        {
47:           long endTime = System.currentTimeMillis();
48:           return elapsedTime + endTime - startTime;
49:        }
```

Continued

File StopWatch.java

StopWatch

-elapsedTime : long
-startTime : long
-isRunning : bool

+start()
+stop()
+getElapsedTime() : long
+reset()

```
50:         else
51:             return elapsedTime;
52:     }
53:
54:     /**
55:         Stops the watch and resets the elapsed time to 0.
56:     */
57:     public void reset()
58:     {
59:         elapsedTime = 0;
60:         isRunning = false;
61:     }
62:
63:     private long elapsedTime;
64:     private long startTime;
65:     private boolean isRunning;
66: }
```

File SelectionSortTimer

```
01: import java.util.Scanner;
02:
03: /**
04:    This program measures how long it takes to sort an
05:    array of a user-specified size with the selection
06:    sort algorithm.
07: */
08: public class SelectionSortTimer
09: {
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:
```

Continued

File SelectionSortTimer

```
18:         int[] a = ArrayUtil.randomIntArray(n, 100);
19:         SelectionSorter sorter = new SelectionSorter(a);
20:
21:         // Use stopwatch to time selection sort
22:
23:         StopWatch timer = new StopWatch();
24:
25:         timer.start();
26:         sorter.sort();
27:         timer.stop();
28:
29:         System.out.println("Elapsed time: "
30:               + timer.getElapsedTime() + " milliseconds");
31:     }
32: }
33:
34:
```

Output:

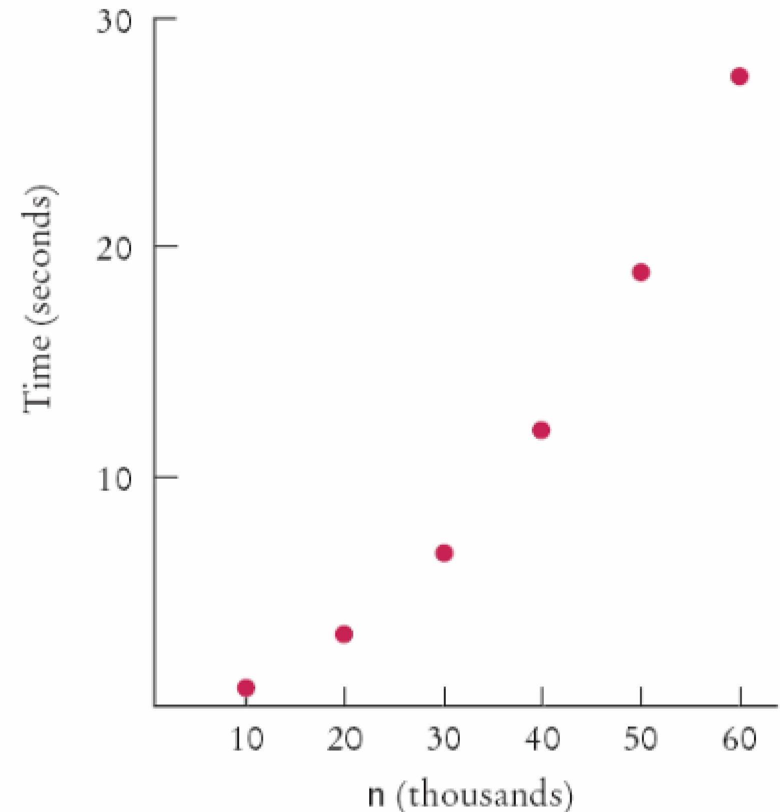
```
Enter array size: 100000
Elapsed time: 27880 milliseconds
```

Selection Sort on Various Size Arrays*

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

n	Milliseconds
10,000	772
20,000	3,051
30,000	6,846
40,000	12,188
50,000	19,015
60,000	27,359

Figure 1:
Time Taken by Selection Sort



*Obtained with a Pentium processor, 1.2 GHz, Java 5.0, Linux

Self Check

3. Approximately how many seconds would it take to sort a data set of 80,000 values?
4. Look at the graph in Figure 1. What mathematical shape does it resemble?

Answers

3. Four times as long as 40,000 values, or about 50 seconds.
4. 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
- 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

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

- Big-Oh notation "$f(n) = O(g(n))$" expresses that *f* grows no faster than *g*
- To convert to big-Oh notation: locate fastest-growing term, and ignore constant coefficient

Self Check

5. If you increase the size of a data set tenfold, how much longer does it take to sort it with the selection sort algorithm?
6. How large does n need to be so that $n^2/2$ is bigger than $5n^2/2 - 3$?

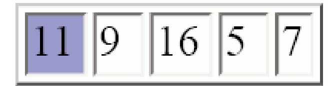
Answers

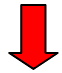
5. It takes about 100 times longer.
6. If n is 4, then $n^2/2$ is 8 and $5n^2/2 - 3$ is 7.

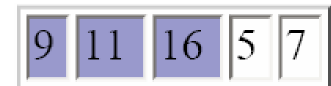
Sort Algorithm #2: Insertion Sort

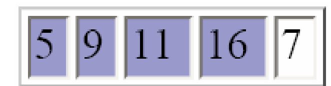
- Assume initial sequence `a[0] a[1] . . . a[k]` of an array is already sorted (when the algorithm starts, we set `k = 0`)
- We enlarge the initial sequence by inserting the next array element, `a[k+1]`, at the proper location
- When we reach the end of the array, the sorting process is complete
- It can be proved that insertion sort is an $O(n^2)$ algorithm (on the same order of efficiency as selection sort)

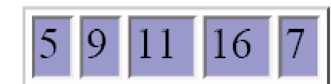
If the array is already sorted (data sets are often partially sorted) → its performance is $O(n)$











File InsertionSorter.java

```
01: /**
02:    This class sorts an array, using the insertion sort
03:    algorithm
04: */
05: public class InsertionSorter
06: {
07:    /**
08:       Constructs an insertion sorter.
09:       @param anArray the array to sort
10:    */
11:    public InsertionSorter(int[] anArray)
12:    {
13:       a = anArray;
14:    }
15:
16:    /**
17:       Sorts the array managed by this insertion sorter
18:    */
```

This class encapsulates a sorting algorithm. Whenever we want to sort an array of integers, we will do the following:

1. To create a new object of this class, passing to it the array to be sort
2. To invoke the `sort` method on this new object

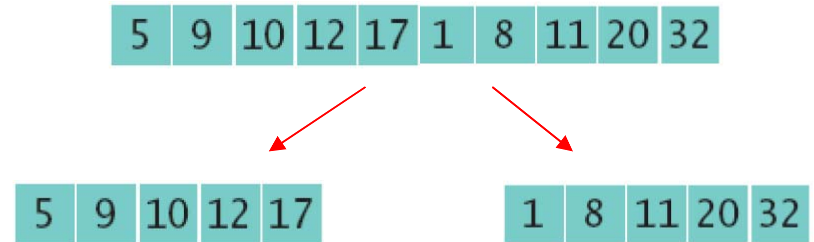
Continued

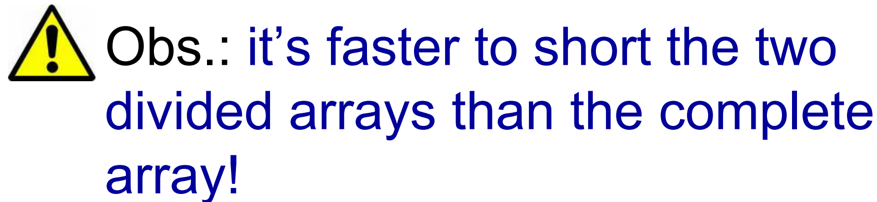
File `InsertionSorter.java`

```
19:    public void sort()
20:    {
21:       for (int i = 1; i < a.length; i++)
22:       {
23:          int next = a[i];
24:          // Move all larger elements up
25:          int j = i;
26:          while (j > 0 && a[j - 1] > next)
27:          {
28:             a[j] = a[j - 1];
29:             j--;
30:          }
31:          // Insert the element
32:          a[j] = next;
33:       }
34:    }
35: 
36:    private int[] a;
37: }
```

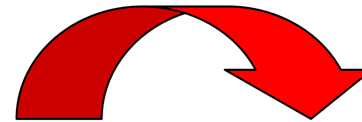
Sort Algorithm #3: Merge Sort

- Example (with only one division): given an array
  - Divide the array in half and sort each half
  - Merge the two sorted arrays into a single sorted array





Obs.: it's faster to short the two divided arrays than the complete array!



5	9	10	12	17		8	11	20	32	1									
	9	10	12	17		8	11	20	32	1	5								
	9	10	12	17			11	20	32	1	5	8							
		10	12	17			11	20	32	1	5	8	9						
			12	17			11	20	32	1	5	8	9	10					
			12	17				20	32	1	5	8	9	10	11				
				17				20	32	1	5	8	9	10	11	12			
								20	32	1	5	8	9	10	11	12	17		
									32	1	5	8	9	10	11	12	17	20	
										1	5	8	9	10	11	12	17	20	32

- Algorithm: Sorts an array by
  - Cutting the array in half
  - Recursively sorting each half
  - Merging the sorted halves
- Merge sort is dramatically faster than the selection sort

File MergeSorter.java

```
01: /**
02:     This class sorts an array, using the merge sort algorithm.
03: */
04: public class MergeSorter
05: {
06:     /**
07:         Constructs a merge sorter.
08:         @param anArray the array to sort
09:     */
10:     public MergeSorter(int[] anArray)
11:     {
12:         a = anArray;
13:     }
14:
15:     /**
16:         Sorts the array managed by this merge sorter.
17:     */
```

This class encapsulates a sorting algorithm. Whenever we want to sort an array of integers, we will do the following:

1. To create a new object of this class, passing to it the array to be sort
2. To invoke the `sort` method on this new object

Continued

File MergeSorter.java

```
18:    public void sort()
19:    {
20:       if (a.length <= 1) return;
21:       int[] first = new int[a.length / 2];
22:       int[] second = new int[a.length - first.length];
23:       System.arraycopy(a, 0, first, 0, first.length);
24:       System.arraycopy(a, first.length, second, 0,
             second.length);
25:       MergeSorter firstSorter = new MergeSorter(first);
26:       MergeSorter secondSorter = new MergeSorter(second);
27:       firstSorter.sort();
28:       secondSorter.sort();
29:       merge(first, second);
30:    }
31:
```

Continued

File MergeSorter.java

```
32:    /**
33:       Merges two sorted arrays into the array managed by
34:       this merge sorter.
35:       @param first the first sorted array
36:       @param second the second sorted array
37:    */
38:    private void merge(int[] first, int[] second)
39:    {
40:       // Merge both halves into the temporary array
41:
42:       int iFirst = 0;
43:          // Next element to consider in the first array
44:       int iSecond = 0;
45:          // Next element to consider in the second array
46:       int j = 0;
47:          // Next open position in a
48:
```

Continued

File MergeSorter.java

```
49:        // As long as neither iFirst nor iSecond past the
50:        // end, move the smaller element into a
51:        while (iFirst < first.length && iSecond < second.length)
52:        {
53:           if (first[iFirst] < second[iSecond])
54:           {
55:              a[j] = first[iFirst];
56:              iFirst++;
57:           }
58:           else
59:           {
60:              a[j] = second[iSecond];
61:              iSecond++;
62:           }
63:           j++;
64:        }
65:
```

Continued

File MergeSorter.java

```
66:        // Note that only one of the two calls to arraycopy
67:        // below copies entries
68:
69:        // Copy any remaining entries of the first array
70:        System.arraycopy(first, iFirst, a, j,
              first.length - iFirst);
71:
72:        // Copy any remaining entries of the second half
73:        System.arraycopy(second, iSecond, a, j,
              second.length - iSecond);
74:     }
75:
76:     private int[] a;
77: }
```

File `MergeSortTester.java`

```
01: /**
02:    This program tests the merge sort algorithm by
03:    sorting an array that is filled with random numbers.
04: */
05: public class MergeSortTester
06: {
07:    public static void main(String[] args)
08:    {
09:       int[] a = ArrayUtil.randomIntArray(20, 100);
10:       ArrayUtil.print(a);
11:       MergeSorter sorter = new MergeSorter(a);
12:       sorter.sort();
13:       ArrayUtil.print(a);
14:    }
15: }
16:
```

Output:

```
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

7. Why does only one of the two `arraycopy` calls at the end of the `merge` method do any work?
8. Manually run the merge sort algorithm on the array:

   8 7 6 5 4 3 2 1

Answers

7. 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).
Then first.length - iFirst <= 0 or iSecond.length - iSecond <= 0.
```

8. 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.

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, i.e.: $n = 2^m$ (this assumption does not affect the outcome of the computation)
- 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
  - $T(n) = T(n/2) + T(n/2) + 5n = 2T(n/2) + 5n$
- The visits for an array of size $n/2$ is $T(n/2) = 2T(n/4) + 5n/2$
  - So $T(n) = 2 \times 2T(n/4) + 5n + 5n$
- The visits for an array of size $n/4$ is $T(n/4) = 2T(n/8) + 5n/4$
  - So $T(n) = 2 \times 2 \times 2\,T(n/8) + 5n + 5n + 5n$
- Repeating the process k times: $T(n) = 2^k T(n/2^k) + 5nk$

  since $n = 2^m$, when $k = m$: $T(n) = 2^m T(n/2^m) + 5nm = nT(1) + 5nm = n + 5n \log_2(n)$
- To establish growth order: (a) Drop the lower-order term n, (b) drop the constant factor 5, and (c) 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

n	Merge Sort (milliseconds)	Selection Sort (milliseconds)
10,000	31	772
20,000	47	3,051
30,000	62	6,846
40,000	80	12,188
50,000	97	19,015
60,000	113	27,359

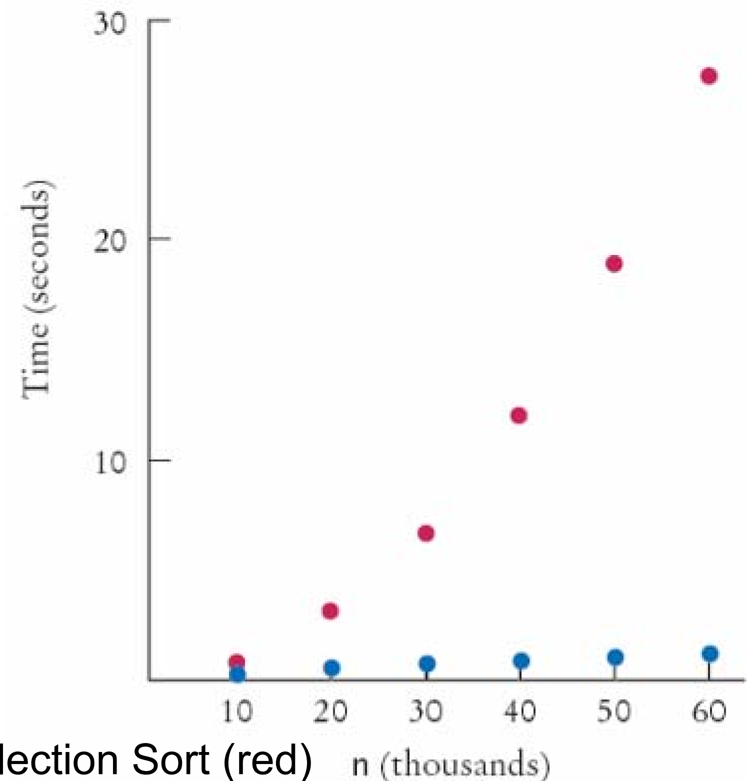


Figure 2:
Merge Sort Timing (blue) versus Selection Sort (red)

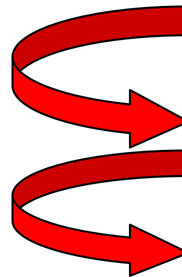
Sort Algorithm #4: Quicksort

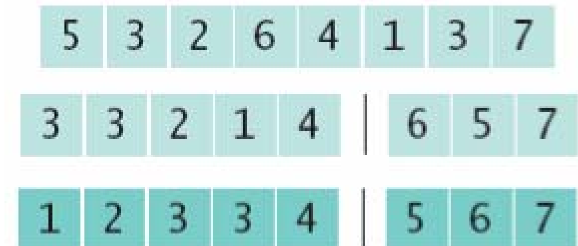
- Quicksort is a commonly used algorithm (O($n$log(n)) on average) that has the advantage over merge sort that no temporary arrays are required to sort and merge the partial results
- Like the merge sort, it is based on the strategy of divide and conquer
- To sort a range `a[from] … a[to]` of the array `a`, do the following:
  1. Partitioning the range: first rearrange the elements in the range so that no elements in the range `a[from] … a[p]` is larger than any element in the range `a[p+1] … a[to]` (we'll see later how to obtain such a partition)
  2. Sorting each partition: next, sort each partition by recursively applying the same algorithm on the two partitions (that sorts the entire range, because the largest element in the first partition is at most as large as the smallest element in the second partition)

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

(1) Partitioning the range

(2) Sorting each partition





Sort Algorithm #4: Quicksort

- How to obtain the partitions?:
  - Pick an element from the range and call it the pivot
  - Now form two regions `a[from]` … `a[i]`, consisting of values at most as large as the pivot and `a[j]` … `a[to]`, consisting of values at least as large as the pivot. The region `a[i+1]` … `a[j-1]` consists of values that haven't been analyzed yet
  - Then, keep incrementing `i` while `a[i] < pivot` and keep decrementing `j` while `a[j] > pivot`
  - Now swap the values in positions `i` and `j`, increasing both areas once more
  - Keep going while `i < j`

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

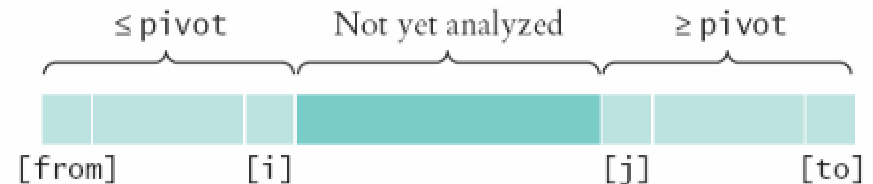


Figure 3: Partitioning a Range

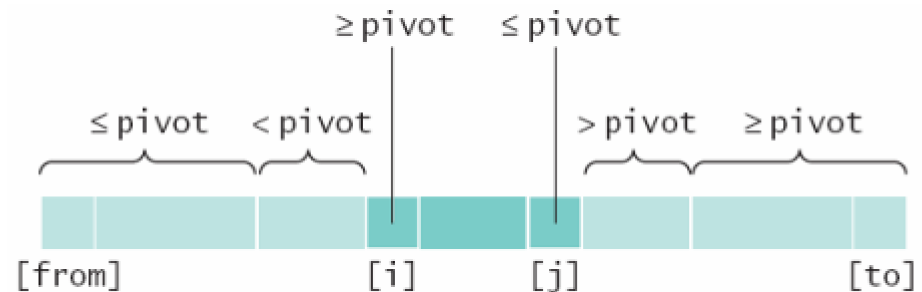


Figure 4: Extending the Partitions

Sorting in a Java Program

The `Arrays` class implements static sort methods to sort arrays of integers and floating-point numbers

- Example: To sort an array of integers:

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

- That `sort` method uses the **`Quicksort`** algorithm

Self Check

9. 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?
10. Suppose you have an array `double[]` values in a Java program. How would you sort it?

Answers

9. 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 207 milliseconds.
10. By calling `Arrays.sort(values)`.

Search Algorithm #1: Linear Search

- If you want to find a number in a sequence of values (array) that occur in arbitrary order, there is nothing you can do to speed up the search. You must simply look through all elements until you have found a match or until you reach the end. This is called a linear or sequential search
- Number of visits for a linear search of an array of n elements:
  - The average search visits $n/2$ elements (assuming that the desired element is present in the sequence)
  - The maximum visits is n (if the desired element is not present in the sequence)

Therefore, a linear search locates a value in an array in $O(n)$ steps

File LinearSearcher.java

```
01: /**
02:     A class for executing linear searches through an array.
03: */
04: public class LinearSearcher
05: {
06:    /**
07:       Constructs the LinearSearcher.
08:       @param anArray an array of integers
09:    */
10:    public LinearSearcher(int[] anArray)
11:    {
12:       a = anArray;
13:    }
14:
15:    /**
16:       Finds a value in an array, using the linear search
17:       algorithm.
```

This class encapsulates a searching algorithm. Whenever we want to search an integer in an array, we will do the following:

1. To create a new object of this class, passing to it the array to be sort
2. To invoke the `search` method on this new object; the explicit parameter of the method will be the integer that we want to search

Continued

File LinearSearcher.java

```
18:         @param v the value to search
19:         @return the index at which the value occurs, or -1
20:         if it does not occur in the array
21:      */
22:      public int search(int v)
23:      {
24:         for (int i = 0; i < a.length; i++)
25:         {
26:            if (a[i] == v)
27:               return i;
28:         }
29:         return -1;
30:      }
31:
32:      private int[] a;
33:   }
```

File LinearSearchTester.java

```
01: import java.util.Scanner;
02:
03: /**
04:     This program tests the linear search algorithm.
05: */
06: public class LinearSearchTester
07: {
08:    public static void main(String[] args)
09:    {
10:       // Construct random array
11:
12:       int[] a = ArrayUtil.randomIntArray(20, 100);
13:       ArrayUtil.print(a);
14:       LinearSearcher searcher = new LinearSearcher(a);
15:
16:       Scanner in = new Scanner(System.in);
17:
```

Continued

File LinearSearchTester.java

```
18:         boolean done = false;
19:         while (!done)
20:         {
21:            System.out.print("Enter number to search for,
                  -1 to quit: ");
22:            int n = in.nextInt();
23:            if (n == -1)
24:               done = true;
25:            else
26:            {
27:               int pos = searcher.search(n);
28:               System.out.println("Found in position " + pos);
29:            }
30:         }
31:      }
32: }
```

Output:

```
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: 11
Found in position 8
```

Self Check

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?
12. Why can't you use a "for each" loop `for (int element : a)` in the search method?

Answers

11. On average, you'd make 500,000 comparisons.
12. The search method returns the index at which the match occurs, not the data stored at that location.

Search Algorithm #2: Binary Search

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

- Example: we would like to see whether the value `15` is in the data set `1 5 8 9 12 17 20 32`:
  1. The last point in the first half of the data set, `a[3]`, is `9`, which is smaller than the value we are looking for → we should look in the second half of the array for a match
  2. Now the last value of the first half of this sequence is `17` → the value must be located in the first half of the subdata set
  3. …
  4. 15 ≠ 17: we don't have a match

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]
1	5	8	9	12	17	20	32

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]
1	5	8	9	12	17	20	32

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]
1	5	8	9	12	17	20	32

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]
1	5	8	9	12	17	20	32

File BinarySearcher.java

```
01: /**
02:    A class for executing binary searches through an array.
03: */
04: public class BinarySearcher
05: {
06:    /**
07:       Constructs a BinarySearcher.
08:       @param anArray a sorted array of integers
09:    */
10:    public BinarySearcher(int[] anArray)
11:    {
12:       a = anArray;
13:    }
14:
15:    /**
16:       Finds a value in a sorted array, using the binary
17:       search algorithm.
```

This class encapsulates a searching algorithm. Whenever we want to search an integer in an array, we will do the following:

1. To create a new object of this class, passing to it the array to be sort
2. To invoke the `search` method on this new object; the explicit parameter of the method will be the integer that we want to search

Continued

File BinarySearcher.java

```
18:         @param v the value to search
19:         @return the index at which the value occurs, or -1
20:         if it does not occur in the array
21:      */
22:      public int search(int v)
23:      {
24:         int low = 0;
25:         int high = a.length - 1;
26:         while (low <= high)
27:         {
28:            int mid = (low + high) / 2;
29:            int diff = a[mid] - v;
30: 
31:            if (diff == 0) // a[mid] == v
32:               return mid;
33:            else if (diff < 0) // a[mid] < v
34:               low = mid + 1;
35:            else
36:               high = mid - 1;
37:         }
38:         return -1;
39:      }
40: 
41:      private int[] a;
42: }
```

Analyzing the Binary Search Algorithm

- 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 → $T(n) = T(n/2) + 1$
- For $n/2$ → $T(n/2) = T(n/4) + 1$
- Substituting into the original equation: $T(n) = T(n/4) + 2$
- This generalizes to: $T(n) = T(n/2^k) + k$
- Assume n is a power of 2, i.e.: $n = 2^m$ where $m = \log_2(n)$ → $T(n) = 1 + \log_2(n)$

Binary search is an $O(\log(n))$ algorithm

Searching a Sorted Array in a Program

The `Arrays` class contains a static **`binarySearch`** method

- The method returns either
  - The index of the element, if element is found
  - Or **`-k-1`** where `k` is the position before which the element should be inserted

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

Self Check

13. 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?
14. Why is it useful that the `Arrays.binarySearch` method indicates the position where a missing element should be inserted?
15. 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?

Answers

13. You would search about 20. (The binary log of 1,024 is 10.)
14. Then you know where to insert it so that the array stays sorted, and you can keep using binary search.
15. Otherwise, you would not know whether a value is present when the method returns 0.

Chapter Summary

- The selection sort algorithm sorts an array by repeatedly finding the smallest element of the unsorted tail region and moving it to the front
- Computer scientists use the big-Oh notation `f(n) = O(g(n))` to express that the function `f` grows no faster than the function `g`
- Selection sorts is an $O(n^2)$ algorithm. Doubling the data set means a fourfold increase in procession time
- Insertion sort is an $O(n^2)$ algorithm
- The merge sort algorithm sorts an array by cutting the array in half, recursively sorting each half, and then merging the sorted halves
- Merge sort is an `O(nlog(n))` algorithm. The `nlog(n)` function grows much more slowly than n^2
- The `Arrays` class implements a sorting method that you should use for your Java programs

Continued

Chapter Summary

- A linear search examines all values in an array until it finds a match or reaches the end
- A linear search locates a value in an array in `O(n)` steps
- 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
- A binary search locates a value in an array in `O(log(n))` steps