ICOM 4015: Advanced Programming

Lecture 14

Chapter Fourteen: Sorting and Searching

ICOM 4015 Fall 2008

Big Java by Cay Horstmann

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

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

11	9	17	5	12

Sorting an Array of Integers

- 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

ch14/selsort/SelectionSorter.java

```
/**
   This class sorts an array, using the selection sort
   algorithm
*/
public class SelectionSorter
{
   /**
      Constructs a selection sorter.
      @param anArray the array to sort
   */
   public SelectionSorter(int[] anArray)
   {
      a = anArray;
   }

   /**
      Sorts the array managed by this selection sorter.
   */
   public void sort()
   {
```

Continued

ch14/selsort/SelectionSorter.java (cont.)

```
      for (int i = 0; i < a.length - 1; i++)
      {
         int minPos = minimumPosition(i);
         swap(minPos, i);
      }
   }

   /**
      Finds the smallest element in a tail range of the array.
      @param from the first position in a to compare
      @return the position of the smallest element in the
      range a[from] . . . a[a.length - 1]
   */
   private int minimumPosition(int from)
   {
      int minPos = from;
      for (int i = from + 1; i < a.length; i++)
         if (a[i] < a[minPos]) minPos = i;
      return minPos;
   }
```

Continued

ch14/selsort/SelectionSorter.java (cont.)

```
   /**
      Swaps two entries of the array.
      @param i the first position to swap
      @param j the second position to swap
   */
   private void swap(int i, int j)
   {
      int temp = a[i];
      a[i] = a[j];
      a[j] = temp;
   }

   private int[] a;
}
```

ch14/selsort/SelectionSortDemo.java

```java
import java.util.Arrays;

/**
   This program demonstrates the selection sort algorithm by
   sorting an array that is filled with random numbers.
*/
public class SelectionSortDemo
{
   public static void main(String[] args)
   {
      int[] a = ArrayUtil.randomIntArray(20, 100);
      System.out.println(Arrays.toString(a));

      SelectionSorter sorter = new SelectionSorter(a);
      sorter.sort();

      System.out.println(Arrays.toString(a));
   }
}
```

File ArrayUtil.java

Typical 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 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:

1	5	4	3	2	6

1	2	3	4	5	6

1	2	3	4	5	6

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*

ch14/selsort/StopWatch.java

```java
/**
   A stopwatch accumulates time when it is running. You can
   repeatedly start and stop the stopwatch. You can use a
   stopwatch to measure the running time of a program.
*/
public class StopWatch
{
   /**
      Constructs a stopwatch that is in the stopped state
      and has no time accumulated.
   */
   public StopWatch()
   {
      reset();
   }

   /**
      Starts the stopwatch. Time starts accumulating now.
   */
   public void start()
   {
      if (isRunning) return;
```

Continued

ch14/selsort/StopWatch.java (cont.)

```
         isRunning = true;
         startTime = System.currentTimeMillis();
      }

      /**
         Stops the stopwatch. Time stops accumulating and is
         is added to the elapsed time.
      */
      public void stop()
      {
         if (!isRunning) return;
         isRunning = false;
         long endTime = System.currentTimeMillis();
         elapsedTime = elapsedTime + endTime - startTime;
      }

      /**
         Returns the total elapsed time.
         @return the total elapsed time
      */
      public long getElapsedTime()
      {
```

Continued

ch14/selsort/StopWatch.java (cont.)

```java
      if (isRunning)
      {
         long endTime = System.currentTimeMillis();
         return elapsedTime + endTime - startTime;
      }
      else
         return elapsedTime;
   }

   /**
      Stops the watch and resets the elapsed time to 0.
   */
   public void reset()
   {
      elapsedTime = 0;
      isRunning = false;
   }

   private long elapsedTime;
   private long startTime;
   private boolean isRunning;
}
```

ch14/selsort/SelectionSortTimer.java

```java
import java.util.Scanner;

/**
   This program measures how long it takes to sort an
   array of a user-specified size with the selection
   sort algorithm.
*/
public class SelectionSortTimer
{
   public static void main(String[] args)
   {
      Scanner in = new Scanner(System.in);
      System.out.print("Enter array size: ");
      int n = in.nextInt();

      // Construct random array

      int[] a = ArrayUtil.randomIntArray(n, 100);
      SelectionSorter sorter = new SelectionSorter(a);

```

Continued

ch14/selsort/SelectionSortTimer.java (cont.)

```
      // Use stopwatch to time selection sort

      StopWatch timer = new StopWatch();

      timer.start();
      sorter.sort();
      timer.stop();

      System.out.println("Elapsed time: "
            + timer.getElapsedTime() + " milliseconds");
   }
}
```

ch14/selsort/SelectionSortTimer.java (cont.)

Output:

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

Selection Sort on Various Size Arrays*

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

* Obtained with a Pentium processor, 2 GHz, Java 6, Linux

Selection Sort on Various Size Arrays

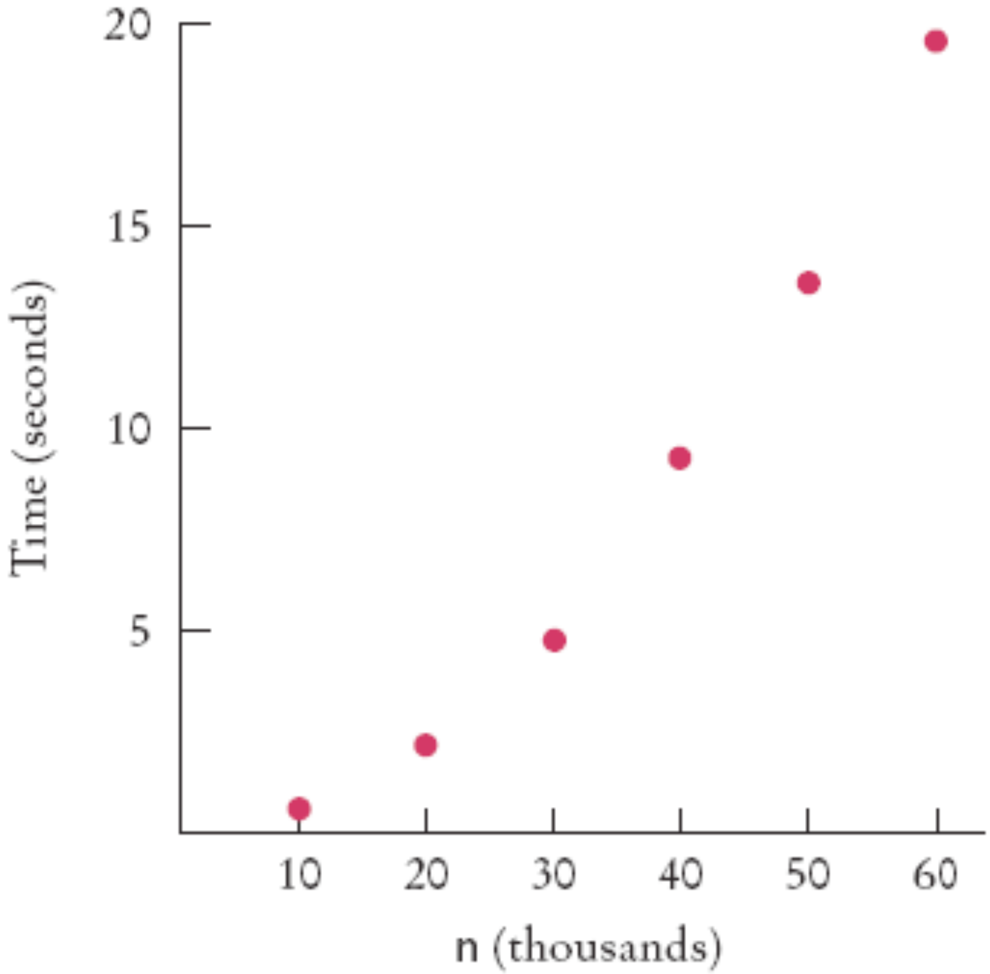


Figure 1 Time Taken by Selection Sort

Selection Sort on Various Size Arrays

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

Self Check 14.3

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 36 seconds.

Self Check 14.4

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
- 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 14.5

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

Answer: It takes about 100 times longer.

Self Check 14.6

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

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

Insertion Sort

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

11	9	16	5	7

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

9	11	16	5	7

- Add a[2]

9	11	16	5	7

- Add a[3]

5	9	11	16	7

- Finally, add a[4]

5	9	11	16	7

ch14/insertionsort/InsertionSorter.java

```
/**
   This class sorts an array, using the insertion sort
   algorithm
*/
public class InsertionSorter
{
   /**
      Constructs an insertion sorter.
      @param anArray the array to sort
   */
   public InsertionSorter(int[] anArray)
   {
      a = anArray;
   }

   /**
      Sorts the array managed by this insertion sorter
   */
   public void sort()
   {
      for (int i = 1; i < a.length; i++)
      {
```

Continued

ch14/insertionsort/InsertionSorter.java (cont.)

```
            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;
         }
      }

   private int[] a;
}
```

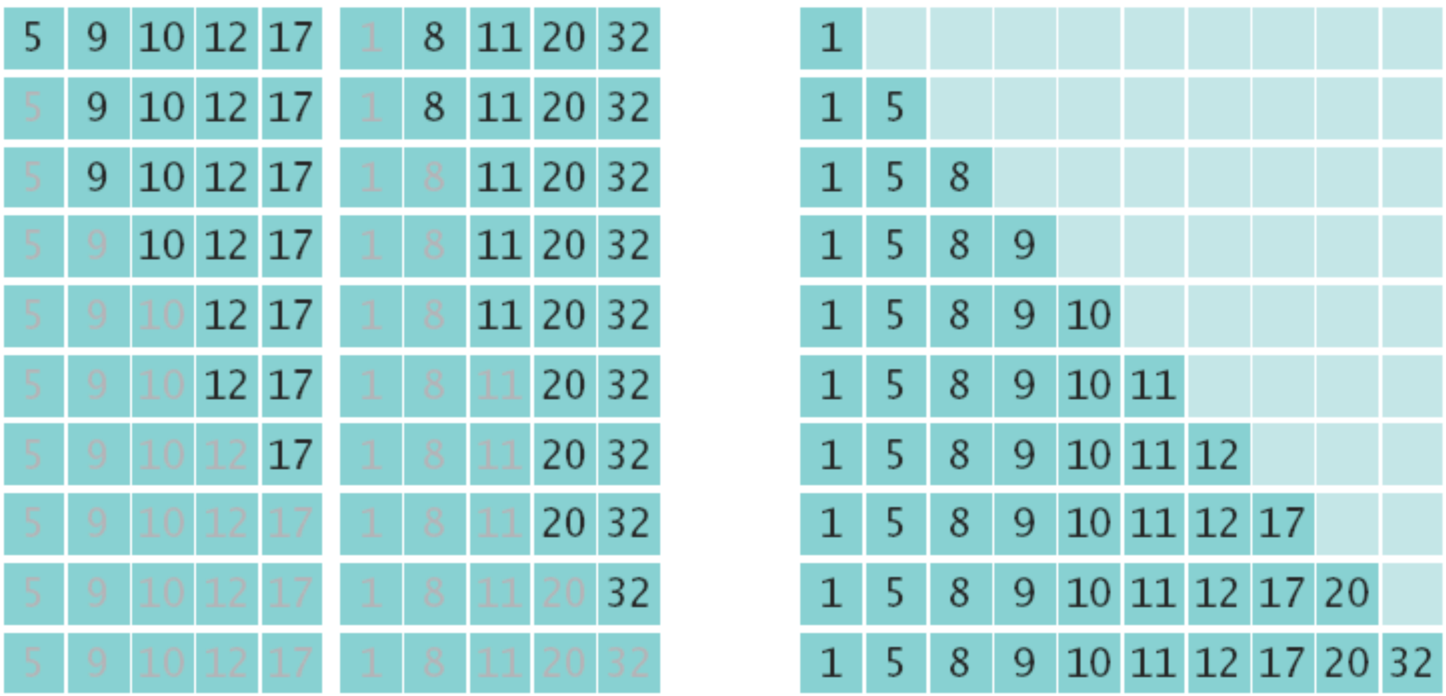
Merge Sort

- Sorts an array by
  - *Cutting the array in half*
  - *Recursively sorting each half*
  - *Merging the sorted halves*
- Dramatically faster than the selection sort

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 void sort()
{
   if (a.length <= 1) return;
   int [] first = new int[a.length / 2];
   int[] second = new int[a.length - first.length];
   System.arraycopy(a, 0, first, 0, first.length);
   System.arraycopy(a, first.length, second, 0,
         second.length);
   MergeSorter firstSorter = new MergeSorter(first);
   MergeSorter secondSorter = new MergeSorter(second);
   firstSorter.sort();
   secondSorter.sort();
   merge(first, second);
}
```

ch14/mergesort/MergeSorter.java

```java
/**
   This class sorts an array, using the merge sort algorithm.
*/
public class MergeSorter
{
   /**
      Constructs a merge sorter.
      @param anArray the array to sort
   */
   public MergeSorter(int[] anArray)
   {
      a = anArray;
   }

   /**
      Sorts the array managed by this merge sorter.
   */
   public void sort()
   {
      if (a.length <= 1) return;
      int[] first = new int[a.length / 2];
      int[] second = new int[a.length - first.length];
```

Continued

ch14/mergesort/MergeSorter.java (cont.)

```
        System.arraycopy(a, 0, first, 0, first.length);
        System.arraycopy(a, first.length, second, 0, second.length);
        MergeSorter firstSorter = new MergeSorter(first);
        MergeSorter secondSorter = new MergeSorter(second);
        firstSorter.sort();
        secondSorter.sort();
        merge(first, second);
    }

    /**
        Merges two sorted arrays into the array managed by this
        merge sorter.
        @param first the first sorted array
        @param second the second sorted array
    */
    private void merge(int[] first, int[] second)
    {
        // Merge both halves into the temporary array

        int iFirst = 0;
            // Next element to consider in the first array
        int iSecond = 0;
```

Continued

ch14/mergesort/MergeSorter.java (cont.)

```
          // Next element to consider in the second array
      int j = 0;
          // Next open position in a

      // As long as neither iFirst nor iSecond past the end, move
      // the smaller element into a
      while (iFirst < first.length && iSecond < second.length)
      {
         if (first[iFirst] < second[iSecond])
         {
            a[j] = first[iFirst];
            iFirst++;
         }
         else
         {
            a[j] = second[iSecond];
            iSecond++;
         }
         j++;
      }

```

Continued

ch14/mergesort/MergeSorter.java (cont.)

```
      // Note that only one of the two calls to arraycopy below
      // copies entries

      // Copy any remaining entries of the first array
      System.arraycopy(first, iFirst, a, j, first.length - iFirst);

      // Copy any remaining entries of the second half
      System.arraycopy(second, iSecond, a, j, second.length - iSecond);
   }

   private int[] a;
}
```

ch14/mergesort/MergeSortDemo.java

```java
import java.util.Arrays;

/**
   This program demonstrates the merge sort algorithm by
   sorting an array that is filled with random numbers.
*/
public class MergeSortDemo
{
   public static void main(String[] args)
   {
      int[] a = ArrayUtil.randomIntArray(20, 100);
      System.out.println(Arrays.toString(a));

      MergeSorter sorter = new MergeSorter(a);
      sorter.sort();
      System.out.println(Arrays.toString(a));
   }
}

```

ch14/mergesort/MergeSortDemo.java (cont.)

Typical 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 14.7

Why does only one of the two `arraycopy` calls 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).

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

Self Check 14.8

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.

Analyzing the Merge Sort Algorithm

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

Merge Sort Timing vs. Selection Sort

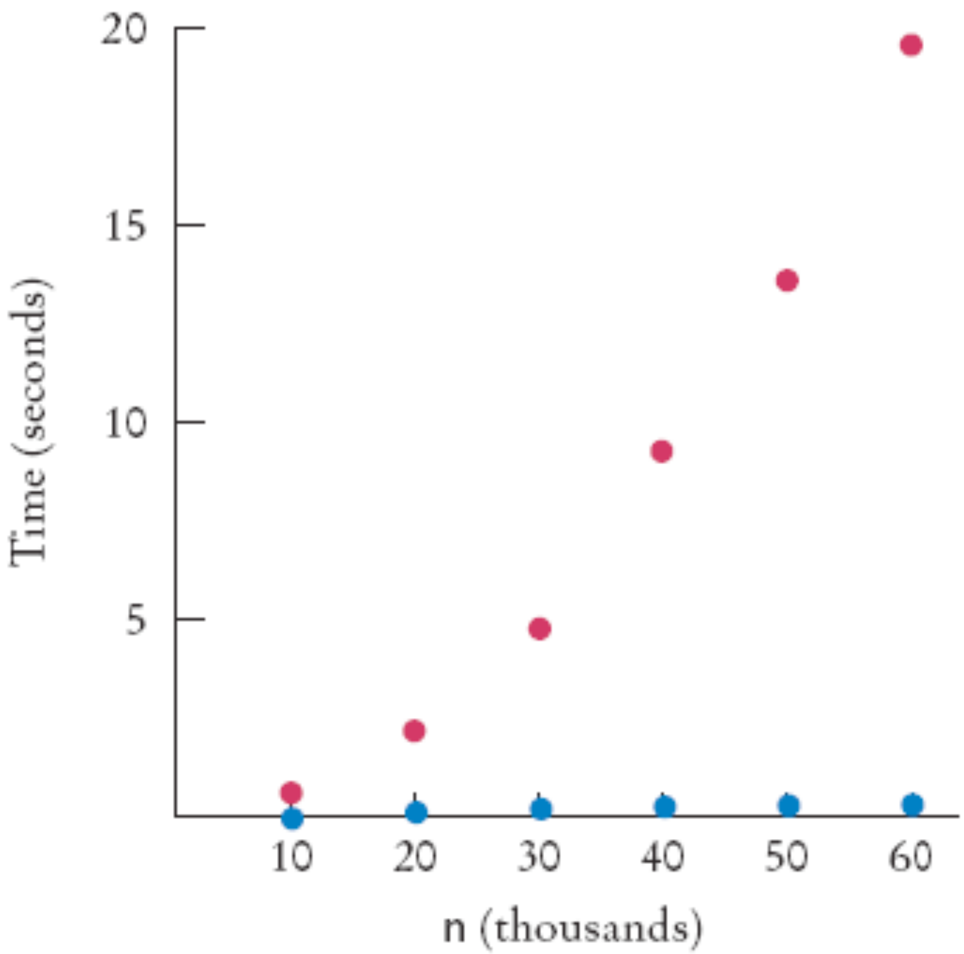


Figure 2

Merge Sort Timing (blue) versus Selection Sort (red)

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:
  - $T(n/2) = 2T(n/4) + 5n/2$
  - *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* $T(n) = 2 \times 2 \times 2T(n/8) + 5n + 5n + 5n$

Analyzing 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 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

Sorting in a Java Program

- The `Arrays` class implements a sorting method
- To sort an array of integers

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

- That `sort` method uses the Quicksort algorithm (see Advanced Topic 14.3)

Self Check 14.9

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?

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 × 192 milliseconds or approximately 408 milliseconds.

Self Check 14.10

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

Answer: By calling `Arrays.sort(values)`.

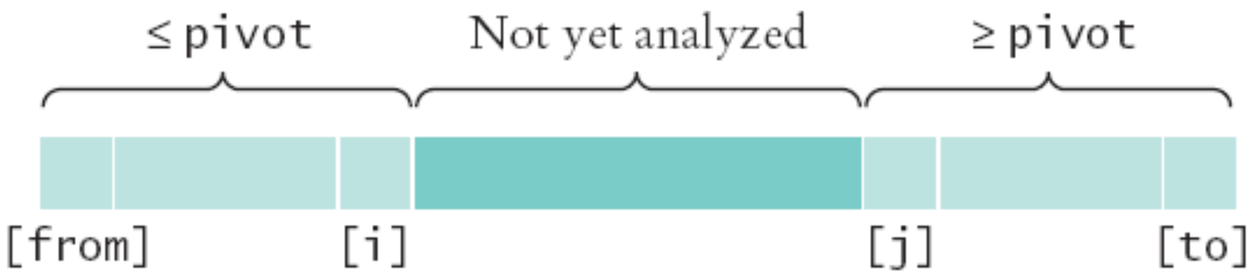
The Quicksort Algorithm

- Divide and conquer
  1. *Partition the range*
  3. *Sort each partition*

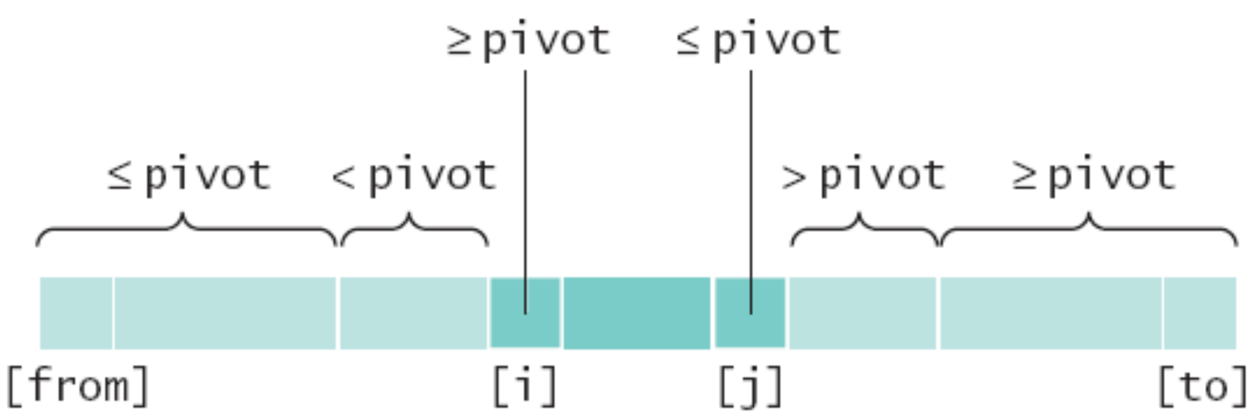
The Quicksort Algorithm

```
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



Partitioning a Range



Extending the Partitions

The Quicksort Algorithm

```
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;
}
```

The First Programmer

Babbage's Difference Engine

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

ch14/linsearch/LinearSearcher.java

```java
/**
   A class for executing linear searches through an array.
*/
public class LinearSearcher
{
   /**
      Constructs the LinearSearcher.
      @param anArray an array of integers
   */
   public LinearSearcher(int[] anArray)
   {
      a = anArray;
   }

   /**
      Finds a value in an array, using the linear search
      algorithm.
      @param v the value to search
      @return the index at which the value occurs, or -1
      if it does not occur in the array
   */
```

Continued

ch14/linsearch/LinearSearcher.java (cont.)

```
    public int search(int v)
    {
       for (int i = 0; i < a.length; i++)
       {
          if (a[i] == v)
             return i;
       }
       return -1;
    }

    private int[] a;
}
```

ch14/linsearch/LinearSearchDemo.java

Typical 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 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?

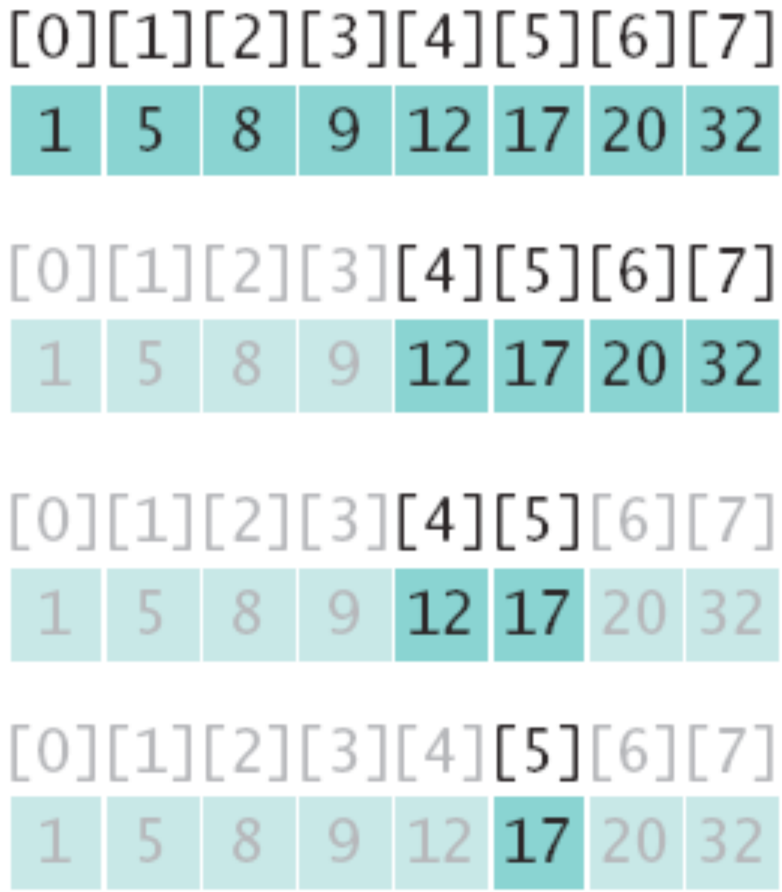
Answer: The search method returns the index at which the match occurs, not the data stored at that location.

Binary Search

- 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*

Binary Search

- To search 15:



- 15 ≠ 17: we don't have a match

ch14/binsearch/BinarySearcher.java

```java
/**
   A class for executing binary searches through an array.
*/
public class BinarySearcher
{
   /**
      Constructs a BinarySearcher.
      @param anArray a sorted array of integers
   */
   public BinarySearcher(int[] anArray)
   {
      a = anArray;
   }

   /**
      Finds a value in a sorted array, using the binary
      search algorithm.
      @param v the value to search
      @return the index at which the value occurs, or -1
      if it does not occur in the array
   */
   public int search(int v)
```

Continued

ch14/binsearch/BinarySearcher.java (cont.)

```java
   {
      int low = 0;
      int high = a.length - 1;
      while (low <= high)
      {
         int mid = (low + high) / 2;
         int diff = a[mid] - v;

         if (diff == 0) // a[mid] == v
            return mid;
         else if (diff < 0) // a[mid] < v
            low = mid + 1;
         else
            high = mid - 1;
      }
      return -1;
   }

   private int[] a;
}
```

Binary Search

- Count the number of visits to search an 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, $n = 2m$ where $m = \log_2(n)$
- Then: $T(n) = 1 + \log_2(n)$
- 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 wherek 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 14.13

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?

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

Self Check 14.14

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 14.15

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*?

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

Sorting Real Data

- `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 is a should come before* `b`
  - *0 if a and b are the same*
  - *A positive number otherwise*

Sorting Real Data

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

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

`compareTo` Method

- The implementation must define a total ordering relationship
  - *Antisymmetric*
    If `a.compareTo(b)` *= 0, then* `b.compareTo(a)` *= 0*
  - *Reflexive*
    `a.compareTo(a) = 0`
  - *Transitive*
    If `a.compareTo(b) = 0` *and* `b.compareTo(c) = 0`*, then* `a.compareTo(c) = 0`

Sorting Real Data

- Once your class implements `Comparable`, simply use the `Arrays.sort` method:

```
Coin[] coins = new Coin[n];
// Add coins
. . .
Arrays.sort(coins);
```

- If the objects are stored in an `ArrayList`, use `Collections.sort`:

```
ArrayList<Coin> coins = new ArrayList<Coin>();
// Add coins
. . .
Collections.sort(coins);
```

- `Collections.sort` uses the merge sort algorithm

Self Check 14.16

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 14.17

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.