CHAPTER 14

SORTING AND SEARCHING

(based on material from slides accompanying
Horstmann: Java for Everyone: Late Objects,
John Wiley and Sons Inc, with updates by Simon Jones)

Slides by Rick Giles

Chapter Goals

- To study several searching and sorting algorithms
- To appreciate that algorithms for the same task can differ widely in performance
- To see how to measure the running time of a program
- To see how to organize (potentially) widely useful methods in a separate class
- Contents:
  - Searching: Linear and Binary
  - Sorting: Selection and Merge

Searching

- **Linear search:** Examines all values in an array until it finds a match or reaches the end
- Also called **sequential search**
- Number of array element "visits" for a linear search of an array of *n* elements:
  - The average search visits *n*/2 elements
  - The maximum visits is *n*
- So, the time (steps) taken to search is proportional to *n*
- Advantage: Does not require that the values in the array are arranged in any particular way

`linearSearch` method

```
/**
 * Finds value in array a, using the linear search
 * algorithm.
 * Returns the index at which the value occurs, or -1
 * if it does not occur in the array
 */
public static int linearSearch(int[] a, int value)
{
   for (int i = 0; i < a.length; i++)
   {
      if (a[i] == value) { return i; }
   }
   return -1;
}
```

- Standard, familiar algorithm with for loop
- Note: Special return value to indicate "Not found"

LinearSearchDemo.java

```
public static void main(String[] args)
{
   // Construct a random array
   int[] a = ArrayUtil.randomIntArray(20, 100);
   System.out.println(Arrays.toString(a));
   // Search for number, until -1 entered
   Scanner in = new Scanner(System.in);
   while (true)
   {
      System.out.print("Enter number, -1 to quit: ");
      int n = in.nextInt();    // Search for?
      if (n < 0) { break; }    // Quit signal entered
      int pos = linearSearch(a, n);
      System.out.println("Found in position " + pos);
   }
}
```

Note

Note

Helper method: randomIntArray

```
private static Random generator = new Random();

/**
 * Creates an array of given length,
 * filled with random values in range 0 .. n-1
*/
public static int[] randomIntArray(int length, int n)
{
   int[] a = new int[length];
   for (int i = 0; i < a.length; i++)
   {
      a[i] = generator.nextInt(n);
   }
   return a;
}
```

ArrayUtil.java

- `randomIntArray` is (potentially) widely useful
  - So is packaged in a separate *library class* in a *separate file* (***not predefined by Java***)

```
/** This class contains utility methods for array
 *  manipulation.
*/
public class ArrayUtil
{
   private static Random generator = new Random();

   public static int[] randomIntArray(int length, int n)
   { … }
}
```

This is the first example of defining a simple *new class*
- *Improves the program structure*

Overall program structure

- We have *two Java files*: one main class, and one library class:
- In LinearSearchDemo.java:

```
public class LinearSearchDemo
{
   public static void main(String[] args)
   { … }
   public static int linearSearch(...)
   { … }
}
```

- And in ArrayUtil.java:

```
public class ArrayUtil
{
   private static Random generator = new Random();
   public static int[] randomIntArray(int length, int n)
   { … }
}
```

Sample linear search run

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
```


Binary Search (1)

- 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
- Overhead: May have to sort the array first
  - But not too expensive if many searches are then carried out

Binary Search (2)

- To search for 15:

[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

- 15 ≠ 17: We don't have a match

binarySearch **method**

```
/**
  * Finds a value in a range of a sorted array,
  * using the binary search algorithm.
  * Searches range   a[low] ... a[high]
  * Returns the index at which the value occurs,
  * or -1 if it does not occur in the array
  */
public static int binarySearch(int[] a,
                               int low, int high, int value)
{
   ...
}
```

- Recursive method
- `low` and `high` move *towards* each other
  - Check mid-position -> choose half
  - If they cross over: Value not found

Continued

binarySearch method

```
public static int binarySearch(int[] a,
                               int low, int high, int value)
{
   if (low > high)     // Crossed over?
   { return -1; }      // Yes: not found
   else                // No: still searching
   {
      int mid = (low + high) / 2;    // Locate middle

      if (a[mid] == value) { return mid; }    // Found!

      else if (a[mid] < value )   // Search upper half?
      { return binarySearch(a, mid + 1, high, value); }

      else                       // No, search lower half?
      { return binarySearch(a, low, mid - 1, value); }
   }
}
```


How fast is 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: so *time for n = 1 + time for n/2*
- Overall: time to search n elements is proportional to $\log_2(n)$
- $\log_2(n)$ *is much smaller than n*
- Example: To search 1000 values:
  - Linear search: average 500 visits
  - Binary search: average 10 visits (because $2^{10} = 1024$)
- BUT: The array must be sorted
  - Could sort as required – potentially expensive
  - Or use a linear insertion if only occasional additions to the array

$\log_2(1) = 0$
$\log_2(2) = 1$
$\log_2(4) = 2$
$\log_2(8) = 3$, etc

Selection Sort

- Sorts an array by repeatedly finding the smallest element of the unsorted tail region and moving it to the front
- 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

11	9	17	5	12

→

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

selectionSort method

```
/**
 * Sort an array a, using selection sort.
 * Note: a is rearranged in place.
 */
public static void selectionSort(int[] a)
{
   // Process sections 0 - end, 1 - end, etc
   for (int i = 0; i < a.length - 1; i++)
   {
      int minPos = ArrayUtil.minimumPosition(a, i);
      ArrayUtil.swap(a, minPos, i);
   }
}
```

- Note:
  - `ArrayUtil` library
  - `minimumPosition` in range `i` - end
  - `swap` element `minPos` with element `i`

Helper method: minimumPosition

```
/**
 * Finds the smallest element in a tail range of the
 * array a: checks a[from] ... a[a.length - 1]
 * Returns the position of the smallest
*/
public static int minimumPosition(int[] a, int from)
{
  int minPos = from;
  for (int i = from + 1; i < a.length; i++)
  {
     if (a[i] < a[minPos]) { minPos = i; }
  }
  return minPos;
}
```

Helper method: swap

```
/**
 * Swaps two elements at indices i and j of an array
 */
public static void swap(int[] a, int i, int j)
{
   int temp = a[i];
   a[i] = a[j];
   a[j] = temp;
}
```


ArrayUtil.java

- We now have *three* helper methods:

```
public class ArrayUtil
{
   private static Random generator = new Random();
   public static int[] randomIntArray(int length, int n)
   { … }
   public static void swap(int[] a, int i, int j)
   { … }
   public static int minimumPosition(int[] a, int from)
   { … }
}
```

Main program: SelectionSortDemo

```
/**
 * 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));

      selectionSort(a);

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


SelectionSortDemo.java

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]
```

Profiling the Selection Sort Algorithm

- 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

`StopWatch` slides omitted

SelectionSortTimer.java

An alternative `main` method:

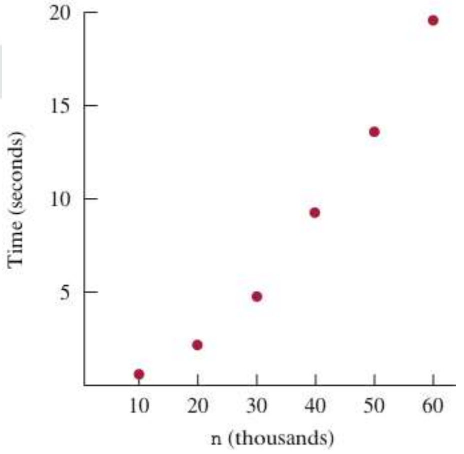
```
public static void main(String[] args)
{
   Scanner in = new Scanner(System.in);
   System.out.print("Enter array size: ");
   int n = in.nextInt();

   // Construct random array
   int[] a = ArrayUtil.randomIntArray(n, 100);

   // Use stopwatch to time selection sort
   StopWatch timer = new StopWatch();

   timer.start();
   selectionSort(a);
   timer.stop();

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

Selection Sort on Various Array Sizes*

Program Run

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

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 2GHz Pentium processor, Java 6, Linux

Performance of Selection Sort Algorithm

- The number of array element "visits" is of the order n^2
  - where *n* is the length of the array
- So: Multiplying the number of elements in an array by **2** multiplies the processing time by **4**
  - bad news!

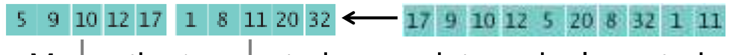
Merge Sort

- Sorts an array by
  - Cutting the array in half
  - ("Recursively") Sorting each half
  - Merging the sorted halves
- Sounds complicated
- But ***much*** more efficient than selection sort

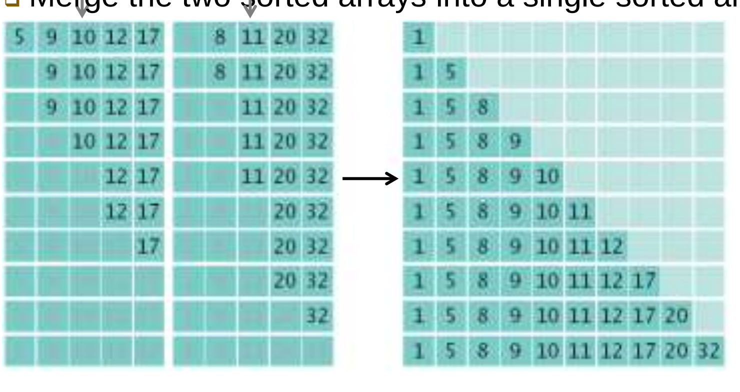
  - engineering trade-off!

Merge Sort Example

- Divide an array in half and sort each half



- Merge the two sorted arrays into a single sorted array



Print version of slide

Merge Sort Example

- Divide an array in half and sort each half

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

Animated presentation version of slide

`mergeSort` method

```
public static void mergeSort(int[] a)
{
   if (a.length <= 1) { return; /* Nothing to do */ }
   // Set up two half-size arrays
   int[] first = new int[a.length / 2];
   int[] second = new int[a.length - first.length];
   // Copy halves to first and second
   for (int i = 0; i < first.length; i++)
   {  first[i] = a[i];  }                                <<
   for (int i = 0; i < second.length; i++)
   {  second[i] = a[first.length + i];  }                <<
   // Recursively sort each partition
   mergeSort(first);
   mergeSort(second);
   // And merge back into the original array
   merge(first, second, a);
}
```

merge helper method

```
private static void merge(int[] first, int[] second,
                                           int[] a)
{
    int iFirst = 0;  // Next element in first array
    int iSecond = 0; // Next element in second array
    int j = 0;       // Next open position in a

    // Repeatedly move smallest from first and second
    // into a, until either runs out
    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

merge helper method (cont)

```
    // Now copy remaining entries

    // Note that only one of the two loops below copies

    // Copy any remaining entries of the first array
    while (iFirst < first.length)
    {
        a[j] = first[iFirst];
        iFirst++; j++;
    }

    // Copy any remaining entries of the second array
    while (iSecond < second.length)
    {
        a[j] = second[iSecond];
        iSecond++; j++;
    }
}
```

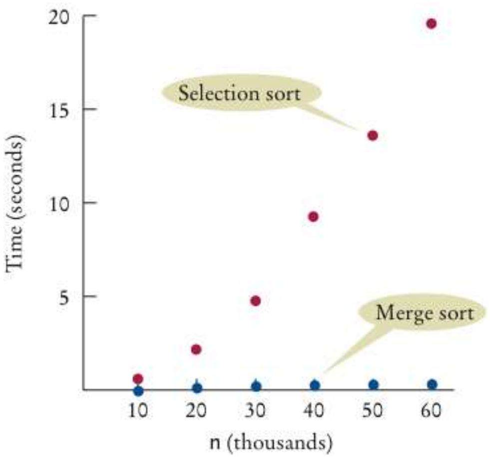
Profiling: MergeSortTimer.java

```
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);
      // Use stopwatch to time merge sort
      StopWatch timer = new StopWatch();
      timer.start();
      mergeSort(a);
      timer.stop();
      System.out.println("Elapsed time: "
         + timer.getElapsedTime() + " milliseconds");
   }
```


Timing the Merge Sort Algorithm (1)

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

Timing the Merge Sort Algorithm (2)



Analyzing the Merge Sort Algorithm

- In an array of size *n*, count how many times an array element is visited
- The overall result is:
  Number of visits is of the order of $n*log_2(n)$
- Selection sort is an order n^2 algorithm - we write $O(n^2)$
- Merge sort is an $O(n*log_2(n))$ algorithm
- $n*log_2(n)$ is much smaller than n^2
- So merge sort performs ***much better*** than selection sort

Sorting and Searching in the Java Libraries

- When you write Java programs, you don't have to implement your own sorting algorithms
- The `Arrays` class contains static `sort` methods
- To sort an array of integers:

```
int[] a = ... ;
Arrays.sort(a);
```
(uses the *Quicksort* algorithm)

- `Arrays` and `Collections` classes contain static `binarySearch` methods
- These methods implement the binary search algorithm, with a useful enhancement:
  - If the value is not found in the array, return -*k* -1, where *k* is the position before which the element should be inserted

Summary: Searching and sorting

- Computer scientists count the "number of visits" to estimate the performance (speed) of an algorithm
- A linear search locates a value in an array in $O(n)$ steps.
- A binary search locates a value in a sorted array in $O(\log(n))$ steps.
- Selection sort is an $O(n^2)$ algorithm. Doubling the data set means a fourfold increase in processing time.
- Merge sort is an $O(n \log(n))$ algorithm. The $n \log(n)$ function grows much more slowly than n^2.

End of section