Sorting Algorithms

Chapter 8: Selection Sort
Chapter 9: Merge Sort, Radix Sort

Others:
(b) A comparison of three representative hardware sorting units, R. Marcelino et al.;
Objectives

• Sort an array into
  - Categories: Radix Sort
  - Total Order: Selection sort, Merge Sort

• Assess efficiency of sorting methods
  - discuss relative efficiencies of various methods
Contents

- Organizing Java Methods That Sort an Array
- Radix Sort
- Selection Sort
- Merge Sort
- Comparing the Algorithms
- Other: Insertion Sort, Shell Sort, Quick Sort
Sorting

• Arranging things into
  ▪ Categories
  ▪ Total order either ascending or descending

• Methods
  ▪ Manual methods
  ▪ Machines
  ▪ Algorithms

• Modeling sorting algorithms In Java
  ▪ static methods to sort objects of an array
  ▪ static methods to sort linked chain of nodes
Sorting an Array

• For an array to be sortable, objects must be comparable
  ▪ Must implement interface Comparable

```java
<T extends Comparable<T>>
```

• We could begin our class with

```java
public class SortArray
{
    public static <T extends Comparable<T>> void sort(T[] a, int n)
    {
        ...
    }
```
Figure 8-1 The class **Gadget** is derived from the class **Widget**, which implements the interface **Comparable**

Ex. Which groups are sortable?
(a.) Widgets  
(b.) Gadgets  
(c.) Neither  
(d.) Both
Contents

• Organizing Java Methods That Sort an Array
• Radix Sort
• Selection Sort
• Merge Sort
• Comparing the Algorithms
• Other: Insertion Sort, Shell Sort, Quick Sort
Radix Sort

• Previously seen sorts on objects that can be compared
• Radix sort does not use comparison
  ▪ Looks for matches in certain categories
  ▪ Places items in “buckets”
• Origin may be from punched card sorting machines
  ▪ Starting from 1900 for census applications!

1949 IBM 082 Card Sorter
http://en.wikipedia.org/wiki/IBM_card_sorter
Figure 9-9
Radix sort: (a) Original array and buckets after 1st, 2nd, 3rd distributions;
Q. Are buckets queues or stacks?
Radix Pseudocode

**Algorithm** radixSort(a, first, last, maxDigits)

// Sorts the array of positive decimal integers a[first..last] into ascending order;
// maxDigits is the number of digits in the longest integer.

for (i = 0 to maxDigits - 1)
{
    Clear bucket[0], bucket[1], ..., bucket[9]
    for (index = first to last)
    {
        digit = digit i of a[index]
        Place a[index] at end of bucket[digit]
    }
    Place contents of bucket[0], bucket[1], ..., bucket[9] into the array a
}

Exercise 9-4. Trace the steps of radixSort during sorting of the following array into ascending order:
(a) 21, 12, 11, 32, 23, 33, 22
(b) 6340, 1243, 291, 3, 6325, 68, 5227, 1638

Exercise B: What is time-complexity of radixSort for N numbers, each with k digits.

Exercise C: Sketch a proof of correctness for radixSort for k-digit numbers.
Contents

• Organizing Java Methods That Sort an Array
• Radix Sort
• Selection Sort
• Merge Sort
• Comparing the Algorithms
• Other: Insertion Sort, Shell Sort, Quick Sort
Selection Sort

• Suppose a priority queue was available
  ▪ How may one use it to implement sorting?

• Example of sorting books by height
  ▪ Take all books off shelf
  ▪ Select shortest, replace on shelf
  ▪ Continue until all books

• Alternative
  ▪ Look down shelf, select shortest
  ▪ Swap first with selected shortest
  ▪ Move to second slot, repeat process
Figure 8-2 Before and after exchanging the shortest book and the first book
Exercise A. Complete trace for array with 2, 8, 10, 15, 5.

Exercise B: What is time-complexity of selectionSort for N numbers.

Exercise C: Trace steps of selectionSort during sorting of following array: 21, 12, 11, 32, 23, 33, 22

Exercise D: Sketch a proof of correctness for selectionSort for N numbers.
public class SortArray {

    public static < T extends Comparable < ? super T > >
    void selectionSort (T [] a, int n) {
        for (int index = 0 ; index < n - 1 ; index++) {
            int indexOfNextSmallest = getIndexOfSmallest (a, index, n - 1);
            swap (a, index, indexOfNextSmallest); }
    }

    private static < T extends Comparable < ? super T > >
    int getIndexOfSmallest (T [] a, int first, int last) {
        T min = a [first];
        int indexOfMin = first;
        for (int index = first + 1 ; index <= last ; index++) {
            if (a [index].compareTo (min) < 0) {
                min = a [index]; indexOfMin = index;
            }
        }
    }

    private static void swap (Object [] a, int i, int j) {
        Object temp = a [i]; a [i] = a [j]; a [j] = temp;
    }
}

Copyright ©2012 by Pearson Education, Inc. All rights reserved
Contents

• Organizing Java Methods That Sort an Array
• Radix Sort
• Selection Sort
• Merge Sort
• Comparing the Algorithms
• Other: Insertion Sort, Shell Sort, Quick Sort
Merge Sort in the Java Class Library

- **Class Arrays** in `java.util` has sort methods
  - `public static void sort(Object[] a)`
  - `public static void sort(Object[] a, int first, int after)`

- These methods use merge sort
  - Merge step skipped if none of entries in left half, greater than entries in right half
Merge Sort

- Divide array into two halves
  - Sort the two halves
  - Merge them into one sorted array
- Uses strategy of “divide and conquer”
  - Divide problem up into two or more distinct, smaller tasks
- Good application for recursion
- Analogy with a tournament
  - For determining all ranks!
Figure 9-2 The major steps in a merge sort

1. Divide the array into two halves.
2. Sort the two halves.
3. Merge the sorted halves into another array.
4. Copy the merged array back into the original array.
Merge Sort Algorithm

**Algorithm** mergeSort(a, tempArray, first, last)
// Sorts the array entries a[first] through a[last] recursively.

if (first < last)
{
    mid = (first + last) / 2
    mergeSort(a, tempArray, first, mid)
    mergeSort(a, tempArray, mid + 1, last)
    *Merge the sorted halves a[first..mid] and a[mid+1..last] using the array tempArray*
Figure 9-1 Merging two sorted arrays into one sorted array
Algorithm to Merge

**Algorithm** merge(a, tempArray, first, mid, last)
// Merges the adjacent subarrays a[first..mid] and a[mid+1..last].

beginHalf1 = first
endHalf1 = mid
beginHalf2 = mid + 1
endHalf2 = last

// While both subarrays are not empty, compare an entry in one subarray with // an entry in the other; then copy the smaller item into the temporary array
index = 0 // next available location in tempArray
while ( (beginHalf1 <= endHalf1) and (beginHalf2 <= endHalf2) )
{
    if (a[beginHalf1] <= a[beginHalf2])
    {
        tempArray[index] = a[beginHalf1]
        beginHalf1++
    }
    else
    {
        tempArray[index] = a[beginHalf2]
        beginHalf2++
    }
    index++
}

// Assertion: One subarray has been completely copied to tempArray.

Copy remaining entries from other subarray to tempArray
Copy entries from tempArray to array a
Figure 9-3 The effect of the recursive calls and the merges during a merge sort
Figure 9-4 A worst-case merge of two sorted arrays is **linear** cost in sum of lengths of two arrays!

First array

```
  2  6
```

Second array

```
  4  8
```

New merged array

```
  2  4  6  8
```

<table>
<thead>
<tr>
<th>Step</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>2 &lt; 4, so copy 2 to new array</td>
</tr>
<tr>
<td>b</td>
<td>6 &gt; 4, so copy 4 to new array</td>
</tr>
<tr>
<td>c</td>
<td>6 &lt; 8, so copy 6 to new array</td>
</tr>
<tr>
<td>d</td>
<td>Copy 8 to new array</td>
</tr>
</tbody>
</table>
Efficiency of Merge Sort

- For \( n = 2^k \) entries
  - In general \( k \) levels of recursive calls are made
  - Note \( k = \log n \)
- Each of \( k \) levels
  - Merge requires at most \( 3n - 1 \) comparisons
  - Calls to merge do at most \( 3n - 2^2 \) operations
- Total across \( k \) levels is \( O(n \times k) \)
  - i.e., \( O(n \log n) \)
Contents

• Organizing Java Methods That Sort an Array
• Radix Sort
• Selection Sort
• Merge Sort
• Comparing the Algorithms
• Other: Insertion Sort, Shell Sort, Quick Sort
Figure 9-10 The time efficiency of various sorting algorithms, expressed in Big Oh notation

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Average Case</th>
<th>Best Case</th>
<th>Worst Case</th>
</tr>
</thead>
<tbody>
<tr>
<td>Radix sort</td>
<td>$O(n)$</td>
<td>$O(n)$</td>
<td>$O(n)$</td>
</tr>
<tr>
<td>Merge sort</td>
<td>$O(n \log n)$</td>
<td>$O(n \log n)$</td>
<td>$O(n \log n)$</td>
</tr>
<tr>
<td>Quick sort</td>
<td>$O(n \log n)$</td>
<td>$O(n \log n)$</td>
<td>$O(n^2)$</td>
</tr>
<tr>
<td>Shell sort</td>
<td>$O(n^{1.5})$</td>
<td>$O(n)$</td>
<td>$O(n^2)$ or $O(n^{1.5})$</td>
</tr>
<tr>
<td>Insertion sort</td>
<td>$O(n^2)$</td>
<td>$O(n^2)$</td>
<td>$O(n^2)$</td>
</tr>
<tr>
<td>Selection sort</td>
<td>$O(n^2)$</td>
<td>$O(n^2)$</td>
<td>$O(n^2)$</td>
</tr>
</tbody>
</table>

Copyright ©2012 by Pearson Education, Inc. All rights reserved
Exercise C: Trace steps of mergeSort during sorting of following array:

21, 12, 11, 32, 23, 33, 22

Exercise D: Sketch a proof of correctness for mergeSort for N numbers.
Iterative Merge Sort

• More difficult than recursive version
  ▪ Recursion controls merging process
  ▪ Iteration would require separate control

• Iterative more efficient in time, space required
  ▪ More difficult to code correctly