Chapter 11 Sorting

- One of the most common computer operations is sorting, that is, putting a collection of elements in order.
- From one-time sorts to highly efficient sorts for frequently used mailing lists and dictionaries, the ability to choose among various sort methods is an important skill in every programmer's repertoire.

11.1 Introduction

- Our focus in this chapter will be on comparison based sorts.
- Comparison based sorts entail comparing elements to other elements. Comparisons are not necessary if we know, in advance, the final position of each element.
- For example if we start with an unsorted list of 100 distinct integers in the range 0…99, we know without any comparison that integer 0 must end up in position 0, and so on.
- The best-known non-comparison based algorithm in Radix sort.
- All of the sorting algorithms presented in this chapter are generic algorithms, that is, they have no calling object and operate on the parameter that specifies the collection to be sorted. (static)
- Two of the sort methods, Merge Sort and Quick Sort, are included in the Java Collection Framework.
- Parameter list may an array of primitive types, objects, or a List object.
- In estimating the efficiency of a sorting method, our primary concern will be worstTime(n). In some applications, such as the national defense and life-support systems, the worst case performance of sort methods is critical.
- Some algorithms, as we will see, are really fast on average but on worst case performance they are extremely slow.
- Another criterion important to sorting algorithm space requirement and method stability.
- A **stable** sort method preserves the relative order of equal elements.

  For example:

  Suppose we have an array of students in which each student consists of a last name and the total quality points for that student. Then we sort by quality points:

  Before sorting:

  [("Balan", 28), ("Wang", 28)]

  After sorting:

  [("Balan", 28), ("Wang", 28)]

11.2 Simple Sorts

- First, each sort method will be illustrated using the following collection of 20 elements:

  59 46 32 80 46 55 50 43 44 81 12 95 17 80 75 33 40 61 16 87

- In each case, we will sort an array of int values into ascending order, and duplicates will be allowed.
- Few if the simple sorting methods are easy to develop, but they have slow execution time when n is large.

11.2.1 Insertion Sort

- The basic idea behind the Insertion Sort is to repeatedly sort out-of-place elements down into their proper indexes in an array.
- Given an array x of int values, x[1] is inserted relative to x[0], and x[0] and x[1] will be swapped if x[0] > x[1].
- At this point, we have x[0] <= x[1]. Then x[2] will be inserted related to the x[0] and x[1].

Example:

59 46 32 80 46 55 50 43 44 81 12 95 17 80 75 33 40 61 16 87

First we place x[1] related to x[0].

46 59 32 80 46 55 50 43 44 81 12 95 17 80 75 33 40 61 16 87

Then we place x[2] relative to the x[0] and x[1].

32 46 59 80 46 55 50 43 44 81 12 95 17 80 75 33 40 61 16 87

Next, we place x[3], where it belongs relative to the sorted subarray x[0], …x[2].

32 46 59 80 46 55 50 43 44 81 12 95 17 80 75 33 40 61 16 87

This process continues until we have an int variable i in the range 1 through 19, and we place x[i] into proper position relative to the sorted subarray.

```
public static void   insertionSort (int [ ] x)
{

        for( int i = 1; i < x.length; i++)
                for(int k = I; k>0 && x[k-1] > x[k]; k--)
                        swap(x, k, k-1)
}
```

```
public static void swap (int [] x, int a, int b)
{
	int t = x[a];
	x[a] = x[b];
	x[b] = t;
}
```

- Average time and worst time for insertion sort is quadratic in n.
- In the Java Collection Framework, Insertion sort is used for sorting subarrays of fewer than 7 elements.
- For small subarrays, other sort methods –usually faster than Insertion Sort – are actually slower because their powerful machinery is designed for larger-sized arrays.
- The choice of 7 for the cutoff is based on empirical studies described in Bentley [1993].
- The cutoff will depend of machine-dependent characteristics.
- Because Insertion sort's inner loop, which does not swaps equal methods, Insertion sort is stable.

11.2.2 Selection Sort

- The simplest of all sort algorithms is Selection Sort.
- Given an array x of int values, swap the smallest element with the element at index 0. Then swap the second smallest element with the element at index 1 and so on.

59 46 32 80 46 55 50 43 44 81 **12** 95 17 80 75 33 40 61 16 87

↑

The smallest value in the array, 12, is swapped with the value 59 at index 0.

12 46 32 80 46 55 50 43 44 81 59 95 17 80 75 33 40 61 **16** 87

↑

Now 16, the smallest of the values from index 1 on, is swapped with the value 46 at index 1:

12 16 32 80 46 55 50 43 44 81 59 95 **17** 80 75 33 40 61 46 87

↑

Then 17, the smallest of the values from index 2 on, is swapped with the value 32 at index 2.

12 16 17 80 46 55 50 43 44 81 59 95 **32** 80 75 33 40 61 46 87

↑

During the 19th loop iteration 87 will be swapped with value 95 at index 18 and whole array will be sorted:

12 16 17 32 33 40 43 44 46 46 50 55 59 61 75 80 80 81 87 95

```
public static void selectSort (int [ ] x)
{
    for(int i = 0 ; i < x.length -1; i++)
    {
       int pos = i;
       for(int k = i+1 ; k < x.length -1; k++)
        if(x[k] < x[pos])
          pos = k;
         swap(x, i, pos);
     }
}
```

- Selection sort always takes quadratic time in n.
- Selection sort is not stable.

11.2.3 Bubble Sort

Warning: Do not use this method.

- Information on Bubble Sort is provided to illustrate a very inefficient algorithm with an appealing name.
- Given an array x of int values, compare each element to the next element in the array, swapping when necessary.
- The largest value will be at index x.length -1.
- Once you reach the end of the array, start back at the beginning and compare and swap elements.
- To avoid needless comparisons, go only as far as the last interchange from the previous iteration.
- Continue until no more swaps can be made.

59 46 32 80 46 55 50 43 44 81 12 95 17 80 75 33 40 61 16 87

Because 59 is greater than 46, those two elements are swapped.

46 59 32 80 46 55 50 43 44 81 12 95 17 80 75 33 40 61 16 87

Then, 59 and 32 are swapped, 59 and 80 are not swapped, 80 and 46 are swapped in index 4 and so on.

46 **32** **59** 46 55 50 43 44 80 12 81 17 80 75 33 40 61 16 87 95

The last swap during the first iteration was of the element 95 and 87 at index 18 and 19.
During the second iteration, the final comparison is made between elements in index 17 and 18.

32 46 46 55 50 43 44 59 12 80 17 80 75 33 40 61 16 81 87 95

The last swap during the second iteration was of the elements 18 and 16 at index 16 and 17, so in the third iteration, the final comparison will be between the elements at index 15 and 16.

Finally after 18 iterations, and many swaps, we end up with

12 16 17 32 33 40 43 44 46 46 50 55 59 61 75 80 80 81 87 95

```
public static void bubleSort (int [ ] x)
{
    Int finalSwapPos = x.length -1, swapPos;

     while(finalSwapPos > 0)
     {
         swapPos = 0;
         for( int i =  0; i < finalSwapPos; i++)
             if(x[ i ] > x[ i+1])
             {
                swap(x, i, i+1)
                swapPos = I;
             }
   }
}
```

11.3 The Comparator Interface

- Insertion Sort, Selection Sort and Bubble sort produce an array of int values in ascending order.
- Also, we can easily modify those methods to sort in descending order. The same we can do to sort arrays of other primitive types, such as long and double.
- What about sorting objects?
- For objects in a class that implements the **Comparable** interface, we can sort by “natural” ordering.

```
public static void selectionSort(Object [ ] x)
```

We will replace the line

```
if(x [ k ] < x [pos])
```

with

```
if(((Comparable)x[ k ]).compareTo(x [pos]) < 0)
```

- Using the Comparable interface with a compareTo method sorting will be made lexicographically.
- If **names** is an array of String objects, we can sort names into lexicographical order with the call

  selectionSort(names);

  [Ben, Brent]

- What if we want do sort array by object length instead of the natural order.
- Then, we can develop a class, such as ByLength that will implement Comparator interface and provide a definition of the compare method.
- Page 466.
- Advantage of using a Comparator object is that no changes need to be made to the element class: the compare method’s parameters are two elements to be ordered.
- Leaving the element class unchanged is especially valuable when, as with String class, users are prohibited from modifying the class.
- Selection Sort page. 467

11.4 How fast can we Sort?

- If we apply Insertion Sort, Selection Sort or even Bubble Sort, worstTime(n) and averageTime(n) are quadratic in n.

11.4.1 Merge Sort

- The Merge Sort algorithm, in the Arrays class of the package java.util., sorts a collection of objects.
- To introduce Merge Sort we will apply to simplifying assumptions:
    1) We assume the objects to be sorted are in an array.
    2) We assume the ordering is to be accomplished through the Comparable interface.

- The basic idea behind the Merge Sort is to keep splitting the n-element in two until, at some step, each of the subarrays has size less than 7. (the choice of 7 is based on run-time experiments)
- Then, Insertion Sort is applied to each of two small sized subarrays, and the two, sorted subarrays are merged together into sorted double sized subarrays.
- Eventually, that subarray is merged with another sorted, double – sized subarray to produce a sorted, quadruple –sized subarray.
- This process continues until, finally, two sorted subarrays of size n/2 are merged back into the original array.

For Example:

59 46 32 80 46 55 87 43 44 81

- We will use auxiliary array aux. page 471.
- Then we clone parameter a into aux.
- With the initial call to the mergeSort, Insertion sort is not performed because high – low >= 7.
- Thus we have to split the array.

```
mergeSort(a, aux, 0, 5);
mergeSort(a, aux, 5, 10);
```

So, Insertion sort is performed on first five elements of aux:

a[0 ….4] = {49, 46, 32, 80, 46}
aux[0…..4] = {32, 46, 46, 59, 80}

The two arrays will be identical until Insertion sort is performed.

a[5 …..9] = {55, 87, 43, 44, 81}
aux[5….9] = {43, 44, 55, 81, 87}

- Upon completion of these two calls to mergeSort, the ten elements of aux, in two sorted subarrays of size 5, are merged back into a, and we are done. More details about merging sort starts on page 471.
- The mergeSort method is an example of the Divide and Conquer design pattern.
- Every divide-and-conquer algorithm has the following characteristics:
    1. The method consists of at least two recursive calls to the method itself;
    2. The recursive calls are independent and can be executed in parallel;
    3. The original task is accomplished by combining the effect of the recursive calls.

11.4.3 Quick Sort

- One of the most efficient and, therefore, widely used sorting algorithms is Quick Sort, developed by C.A.R Hoare [1962].
- The generic algorithm sort is Quick Sort algorithm based on “Engineering a Sort Function”.
- In Arrays class, Quick Sort refers to any method name sort whose parameters is an array of primitive values.
- Merge Sort refers to any method named sort whose parameter is an array of objects.
- Basic idea behind the sort method is this:
    1. First we partition the array x into left subarray and right subarray so that each element in the left subarray is less than or equal to each element in right subarray.
    2. Then we Quick Sort the left and right subarrays and we are done. The sizes of subarrays need not to be the same.

- The first task in partitioning is to choose an element, called a pivot that each element is compared to.
- Elements less than the pivot will end up in the left subarray, and elements greater than the pivot will end up in the right subarray. Elements equal to the pivot may end up in either subarray.
- So, what makes quick sort fast?
- With other sorts, it may take many comparison to put an element in the general area where it belongs.
- With Quick Sort, a partition can move many elements close to where they will finally end up.
- This assumes that the value of the pivot is close to the median of the elements to be partitioned.

  The median of a collection of values is the value that would be in the middle position if the collection were sorted.

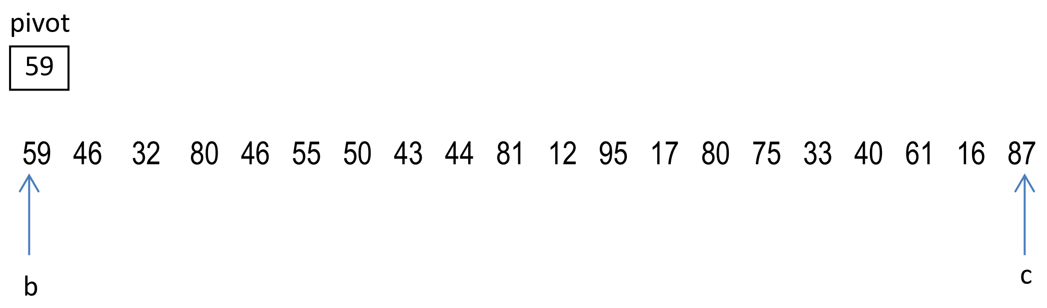
  100 32 77 85 95

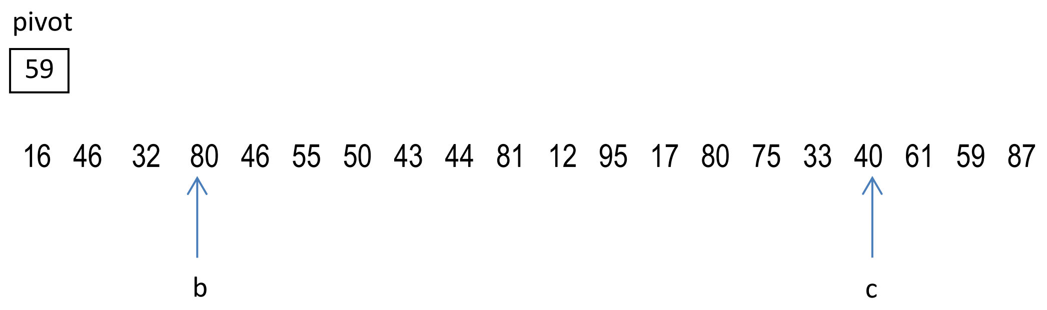
  Median is 85.
  If the collection contains an even number of elements, the median is the average of two values that would be in the middle positions if the collection were sorted.

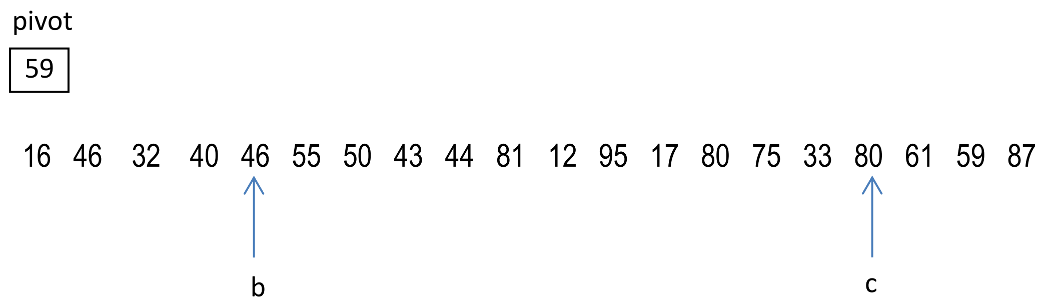
- Suppose we have a collection,

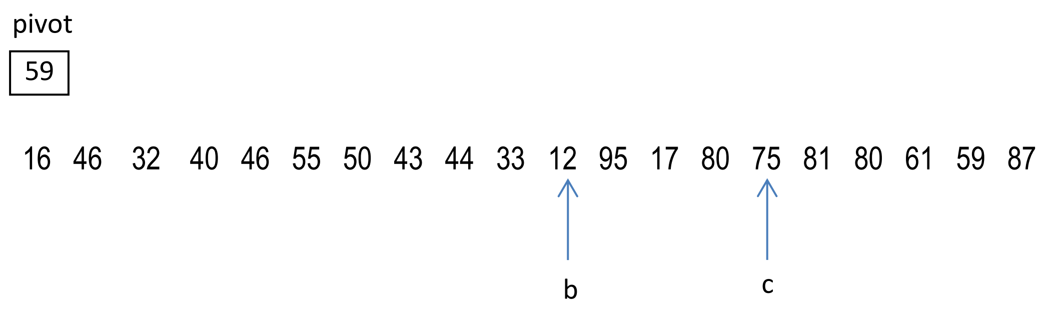
59 46 32 80 46 55 50 43 44 81 12 95 17 80 75 33 40 61 16 87

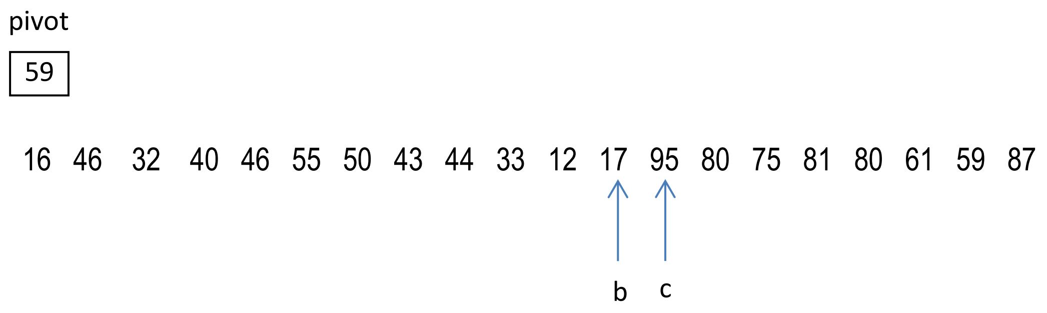
- We now want to move the left subarray all the elements that are less than 59 and move to the right subarray all the elements that are greater than 59.
- To accomplish the partitioning, we create two counters: b which starts at off and moves upward, and c which starts at off+ len-1 and moves downward.











- After partitioning, the left subarray consists of elements from index off through c, and the right subarray consists of elements from index b through off+len-1. The pivot need not end up in either subarray.

11.5 Radix Sort

- Radix Sort is unlike the other sorts presented in this chapter. The sorting is based on the internal representation of the elements to be sorted, not on comparisons between elements.
- Thus, the restriction that worstTime(n) can be no better than linear-logarithmic in n no longer applies.
- Radix sort was widely used on electromechanical punched-cards sorters that appear in old FBI movies.
- Suppose we want to sort an array of non-negative interferes of at most two decimal digits.
- The representation is in base 10, also, referred as radix 10.
- In addition to the array to be sorted, we also have an array, lists, of 10 linked lists, with one linked for each of the ten possible digit values.