CONCEPT EXERCISES

11.1 Trace the execution of each of the six sort methods – Insertion Sort, Selection Sort, Bubble Sort, Merge Sort, Quick Sort and Radix Sort – with the following array of values:

10 90 45 82 71 96 82 50 33 43 67

Insertion Sort

Here is the array after each pass; a '**]**' is to the right of the sorted subarray:

10 90**]** 45 82 71 96 82 50 33 43 67

10 45 90**]** 82 71 96 82 50 33 43 67

10 45 82 90**]** 71 96 82 50 33 43 67

10 45 71 82 90**]** 96 82 50 33 43 67

10 45 71 82 90 96**]** 82 50 33 43 67

10 45 71 82 82 90 96**]** 50 33 43 67

10 45 50 71 82 82 90 96**]** 33 43 67

10 33 45 50 71 82 82 90 96**]** 43 67

10 33 43 45 50 71 82 82 90 96**]** 67

10 33 43 45 50 67 71 82 82 90 96**]**

Selection Sort

Here is the array after each pass; a '**]**' is to the right of the sorted subarray:

10**]** 90 45 82 71 96 82 50 33 43 67

10 33**]** 45 82 71 96 82 50 90 43 67

10 33 43**]** 82 71 96 82 50 90 45 67

10 33 43 45**]** 71 96 82 50 90 82 67

10 33 43 45 50**]** 96 82 71 90 82 67

10 33 43 45 50 67**]** 82 71 90 82 96

10 33 43 45 50 67 71**]** 82 90 82 96

10 33 43 45 50 67 71 82**]** 90 82 96

10 33 43 45 50 67 71 82 82**]** 90 96

10 33 43 45 50 67 71 82 82 90**]** 96

The entire array is now sorted.

Bubble Sort

Here is the array after each pass; a '**[**' is to the left of the sorted subarray:

10 45 82 71 90 82 50 33 43 67 **[**96

10 45 71 82 82 50 33 43 67 **[**90 96

10 45 71 82 50 33 43 67 **[**82 90 96

10 45 71 50 33 43 67 **[**82 82 90 96

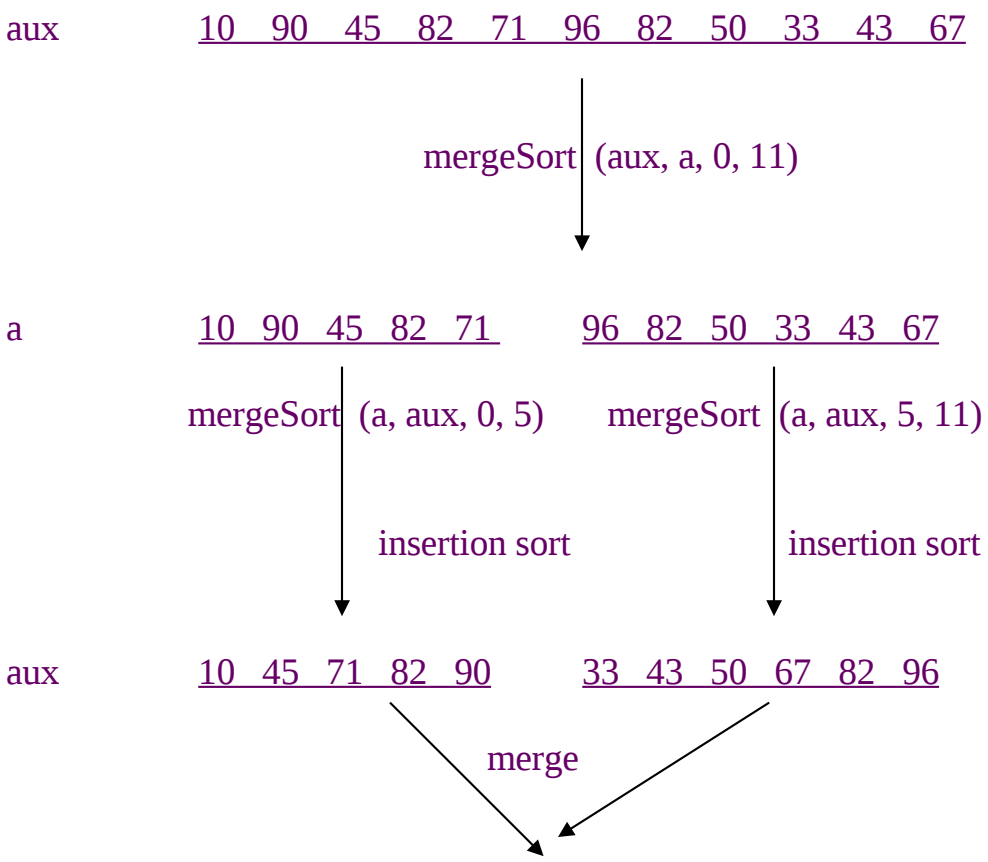
10 45 50 33 43 67 **[**71 82 82 90 96

10 45 33 43 50 **[**67 71 82 82 90 96

10 33 43 45 **[**50 67 71 82 82 90 96

A final pass produces no swaps, and the array is sorted.

Merge Sort



Quick Sort

The initial pivot, v, has the value 67.

After the first outer-loop iteration:

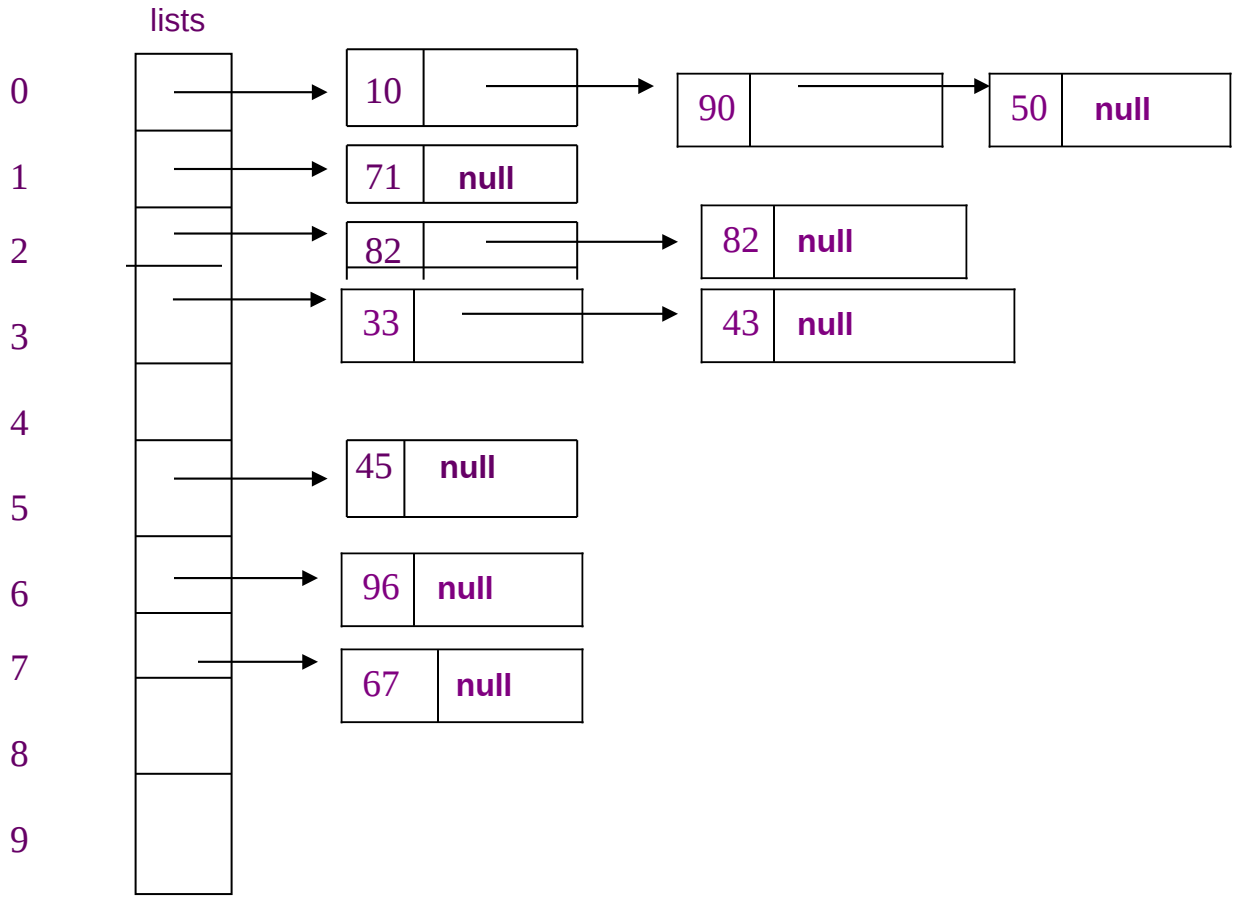
10 43 45 33 50 67 96 82 71 82 90

After Insertion Sort is applied to the subarrays:

10 33 43 45 50 67 71 82 82 90 96

Radix Sort

After each element is appended to a linked list according to the units digit:

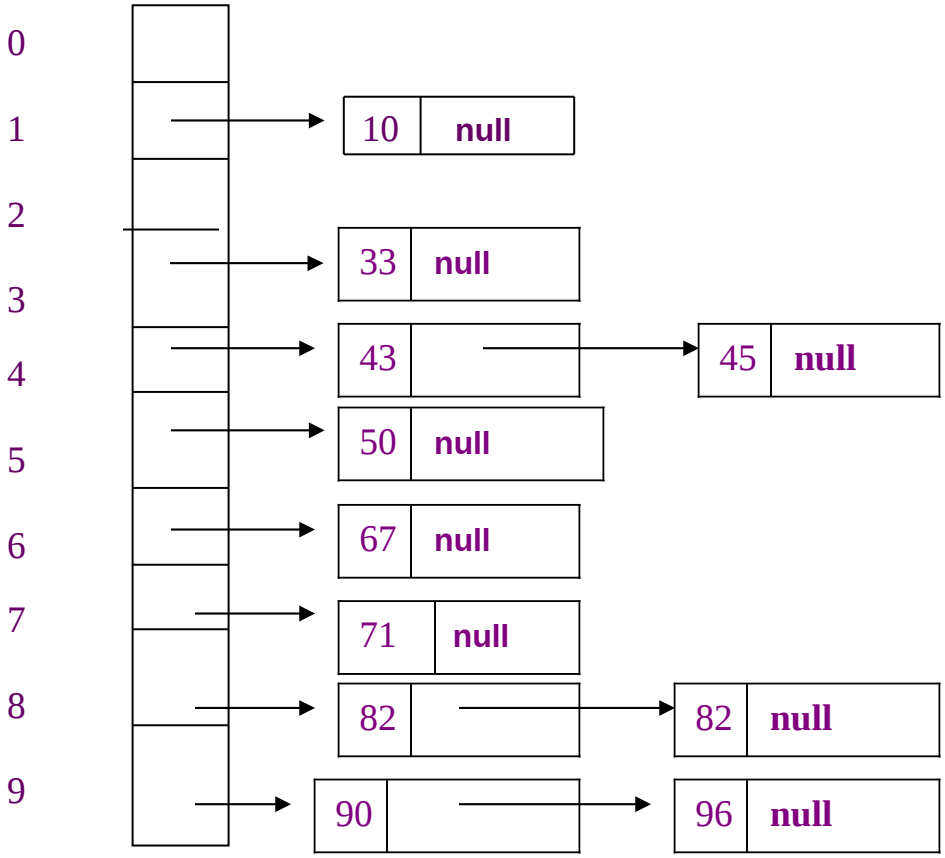


After storing the lists, consecutively, back in the array:

10 90 50 71 82 82 33 43 45 96 67

After each element is stored in a linked list corresponding to its tens digit:

lists



After storing the lists, consecutively, back in the array:

10 33 43 45 50 67 71 82 82 90 96

11.2. a. For each sort method, rearrange the list of values in Concept Exercise 11.1 so that the minimum number of element-comparisons would be required to sort the array.

Insertion Sort: 10 33 43 45 50 67 71 82 82 90 96

Selection Sort: (Any arrangement will require the same number of element-comparisons.)

Bubble Sort: 10 33 43 45 50 67 71 82 82 90 96

Merge Sort: 10 33 43 45 50 67 71 82 82 90 96

Quick Sort: 10 33 43 45 50 67 71 82 82 90 96

Radix Sort: (No matter how the elements are arranged, there are no element-comparisons.)

b. For each sort method, rearrange the list of values in Concept Exercise 11.1 so that the maximum number of element-comparisons would be required to sort the sequence.

Insertion Sort: 96 90 82 82 71 67 50 45 43 33 10

Selection Sort: (Any arrangement will require the same number of element-comparisons.)

Bubble Sort: 96 90 82 82 71 67 50 45 43 33 10

Merge Sort: 90 82 67 45 33 96 82 71 50 43 10

Quick Sort: 33 43 45 50 67 10 96 71 82 82 90

Radix Sort: (No matter how the elements are arranged, there are no element-comparisons.)

11.3 Suppose you want a sort method whose worstTime(*n*) is linear-logarithmic in *n*, but requires only linear-in-*n* time for an already sorted collection. Which one of the sorts in this chapter has those properties?

Merge Sort has those properties.

11.4 For the optimized Quick Sort in Section 11.4.3.1, find an arrangement of the integers 0 **. . .** 49 for which the first partition will produce a subarray of size 4 and a subarray of size 44. Recall that because the number of values is greater than 40, the pivot is the "super-median", that is, the median of the three median-of-threes.

The indexes used in the calculation of the median of medians are 0, 6, 12, 19, 25, 31, 37, 43, and 49. We will store values at those indexes so that the median of medians will be 4; then after the first partition, the left subarray will have 4 elements and the

right subarray will have 44 elements. The following values, stored consecutively at indexes 0, 6, 12, and so on, will produce 4 as the median of medians:

0, 1, 2, 3, 4, 5, 6, 7, 8

The median of the first three values is 1, the median of the second three values is 4, and the median of the third three values is 7. The median of 1, 4, and 7 is 4, as desired.

11.6 Show that seven comparisons are sufficient to sort any collection of five elements.

Hint: Compare the first and second elements. Compare the third and fourth elements. Compare the two larger elements from the earlier comparisons. With three comparisons, we have an ordered chain of three elements, with the fourth element less than (or equal to) one of the elements in the chain. Now compare the fifth element to the middle element in the chain. Complete the sorting in three more comparisons. Note that ceil($\log_2 5!$) = 7, so some collections of five elements cannot be sorted with 6 comparisons.

Compare the first and second elements. Compare the third and fourth elements. Compare the two larger elements from the earlier comparisons. With three comparisons, we have an ordered chain of three elements, with the other element less than (or equal to) one of the elements in the chain. We can now label these four elements a, b, c, and d, with a <= b <= c, and (in the worst case) d <= c. Now compare the fifth element, e, to b. If e <= b, compare a and e to get a, b, c, and e in sequence; otherwise, compare c and e to get a, b, c, and e in sequence. Now compare d to the middle element in a, b and e. If d is <= that element, compare d to the smallest element of a, b, and e to get the sequence of five elements in order. Otherwise, compare d to the largest element of a, b, and e to get the sequence of five elements in order.

11.9 Consider the following, consecutive improvements to Insertion Sort:

a. Replace the call to the method swap with in-line code:

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

        for (int i = 1; i < x.length; i++)
                for (int j = i; j > 0 && x [j -1] > x [j]; j--)
```

```
        {
                temp = x [j];
                x [j] = x [j – 1];
                x [j – 1] = temp;
        } // inner for
} // method insertionSort
```

b. Notice that in the inner loop in part a, temp is repeatedly assigned the original value of x [i]. For example, suppose the array x has

32 46 59 80 35

and j starts at 4. Then 35 hops its way down the array, from index 4 to index 1. The only relevant assignment from temp is that last one. Instead, we can move the assignments to and from temp out of the inner loop:

```
int temp,
        j;

    for (int i = 1; i < x.length; i++)
    {
    temp = x [i];

          for (j = i; j > 0 && x [j -1] > temp; j--)
        x [j] = x [j – 1];
      x [j] = temp;
} // outer for
```

Will these changes affect the estimates for worstTime(*n*) and averageTime(*n*)?

The changes have no effect on the estimates for worstTime(*n*) or averageTime(*n*) because they have no effect on the number of inner-loop iterations.

PROGRAMMING EXERCISES

11.1 For Concept Exercise 11.9, conduct a timing experiment to estimate the run-time effect of the changes made.

```
import java.util.*;

public class SortTime
```

```
{
   public static void main (String[ ] args)
   {
      new SortTime().run();
   } // method main

   /**
    * Estimates the run-time, on average, for a sort method.
    */
   public void run()
   {
         final String TIME_MESSAGE =
                  "\n\nThe elapsed time in seconds is ";

         final int SENTINEL = -1,
                     SEED = 100;

         final String INPUT_PROMPT =
                  "Please enter the number of integers to be sorted (or " +
                  SENTINEL + " to quit): ";

         final double NANO_FACTOR = 1000000000.0;

         Scanner sc = new Scanner (System.in);

         Random r = new Random (SEED);

         long startTime,
                  finishTime,
                  elapsedTime;

         int n;

         while (true)
         {
                  try
                  {
                           System.out.print (INPUT_PROMPT);
                           n = sc.nextInt();
                           if (n == SENTINEL)
                                    break;
                           int[ ] x = new int [n];
                           for (int k = 0; k < n; k++)
                                    x [k] = r.nextInt (n);

                           startTime = System.nanoTime();
```

```
                    insertionSort1 (x);
                    finishTime = System.nanoTime();
                    elapsedTime = finishTime - startTime;
                    System.out.println (TIME_MESSAGE +
                            (elapsedTime / NANO_FACTOR));
                } // try
                catch (InputMismatchException e)
                {
                    System.out.println (e);
                    sc.nextLine();
                }//catch Exception
        } // while
} // method run

public static void insertionSort1 (int[ ] x)
{
        for (int i = 1; i < x.length; i++)
                for (int j = i; j > 0 && x [j - 1] > x [j]; j--)
                        swap (x, j, j - 1);
} // method insertionSort1

public static void insertionSort2 (int[ ] x)
{
        int temp;

        for (int i = 1; i < x.length; i++)
                for (int j = i; j > 0 && x [j - 1] > x [j]; j--)
                {
                        temp = x[j];
                        x[j] = x[j - 1];
                        x[j - 1] = temp;
                } // inner for
} // method insertionSort2

public static void insertionSort3 (int[ ] x)
{
        int temp,
            j;

        for (int i = 1; i < x.length; i++)
        {
                temp = x [i];
                for (j = i; j > 0 && x [j - 1] > temp; j--)
                        x [j] = x [j - 1];
                x [j] = temp;
```

```
        } // outer for
    } // method insertionSort3

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

Here are some timing results, in seconds:

	insertionSort	improvement 1	improvement 2
n = 50,000	1.93	1.93	1.12
n = 100,000	7.77	7.72	4.48

Clearly, improvement 1 had only a negligible impact on run time, but improvement 2 reduced the run time by about 40%.

11.4 For the original version of Quick Sort in Section 11.4.3, replace the inner-loop conditions from

while (x [b] < v) and **while** (x [c] > v)

to

while (b <= c && x [b] <= v) and **while** (c >= b && x [c] >= v)

Create a main method to apply this version of Quick Sort to the following array of **int** values:

46 59

Explain the results.

m = 1 and v = 59. b is incremented to 2 and the first inner loop terminates. The second inner loop terminates immediately because c is still 1. Because b > c is true, the outer loop is exited, and the first recursive call is made:

```
sort1(x, off, c + 1 - off);
```

But the values of these arguments are the same as in the original call to sort1, so we get an infinite sequence of recursive calls!

→ Lecture 12

Use the Master Theorem to derive $O(\ )$ for the following recurrence relations:

① $T(n) = 3T(n/2) + n^2$

② $T(n) = T(n/2) + 1/2\, n^2 + n$

③ $T(n) = 4T(n/2) + n^2$

④ $T(n) = 2^n T(n/2) + n^n$

⑤ $T(n) = 8T(n/2) + 100n^2$

Solution:

① $a = 3$
$b = 2$
$c = 1$
$d = 2$

$3 < 2^2$
So case I applies and
$T(n) = O(n^d) = O(n^2)$

② $a = 1$
$b = 2$
$c = 1/2$
$d = 2$

$1 < 2^2$
So case I applies and
$T(n) = O(n^d) = O(n^2)$

③ $a = 4$
$b = 2$
$c = 1$
$d = 2$

$4 = 2^2$, so
case II applies and
$T(n) = O(n^d \log n) = O(n^2 \log n)$

④ Master Theorem does not apply. $a\ (2^n)$ is not a constant.

⑤ $a = 8$
$b = 2$
$c = 100$
$d = 2$

$8 > 2^2$, so case III applies

$T(n) = O(n^{\log_b a})$
$= O(n^{\log_2 8}) = O(n^3)$