Chapter 7: Advanced Sorting

1. Shellsort
2. Partitioning
3. Quicksort
4. Radix Sort

1

Overview

- Simple sorting in Chapter 3, the bubble, selection, and insertion sorts, are easy to implement but are rather slow.
- In Chapter 6, the mergesort runs much faster than the simple sorts, but requires twice as much space as the original array; this is often a serious drawback.

- Shellsort
- Quicksort is based on the idea of partitioning
- Quicksort

2

Shellsort

- Named for Donald L. Shell, who discovered it in 1959.
- It's based on the insertion sort
  - but adds a new feature that dramatically improves the insertion sort's performance.
- The Shellsort is good for medium-sized arrays, perhaps up to a few thousand items, depending on the particular implementation.
  - However, see the cautionary notes in Chapter 15 about how much data can be handled by a particular algorithm.
- It's not quite as fast as quicksort and other O(N*logN) sorts, so it's not optimum for very large files.
- However, it's much faster than the $O(N^2)$ sorts (like the selection sort and the insertion sort) and it's very easy to implement: the code is short and simple.

3

Shellsort

- The worst-case performance is not significantly worse than the average performance.
  - We'll see later in this chapter that the worst-case performance for quicksort can be much worse unless precautions are taken.
- Some experts recommend starting with a Shellsort for almost any sorting project, and only changing to a more advanced sort, like quicksort, if Shellsort proves too slow in practice.

4

Insertion Sort: Too Many Copies

- Because Shellsort is based on the insertion sort, let's review the relevant section of Chapter 3.
  - Partway through the insertion sort, the items to the left of a marker are internally sorted (sorted among themselves) and items to the right are not.
  - The algorithm removes the item at the marker and stores it in a temporary variable.
  - Then, beginning with the item to the left of the newly vacated cell, it shifts the sorted items right one cell at a time, until the item in the temporary variable can be reinserted in sorted order.

5

The Problem With The Insertion Sort

- Suppose a small item is on the far right, where the large items should be.
  - To move this small item to its proper place on the left, all the intervening items (between where it is and where it should be) must be shifted one space right. This is close to N copies, just for one item.
  - Not all the items must be moved a full N spaces, but the average item must be moved N/2 spaces, which takes N times N/2 shifts for a total of $N^2/2$ copies.
  - Thus the performance of insertion sort is $O(N^2)$.
- This performance could be improved if we could somehow move a smaller item many spaces to the left without shifting all the intermediate items individually.

6

N-Sorting

- The Shellsort achieves these large shifts by insertion-sorting widely spaced elements.
- Once these are sorted, it sorts somewhat less widely spaced elements, and so on.
- The spacing between elements for these sorts is called the *increment* and is traditionally represented by the letter h.

7

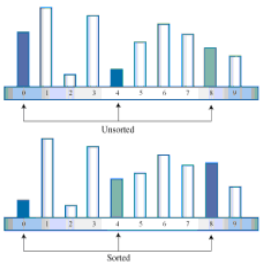


Figure 7.1: 4-sorting 0, 4, and 8

Shows the first step in the process of sorting a 10-element array with an increment of 4. Here the elements 0, 4, and 8 are sorted.

8

- Once 0, 4, and 8 are sorted, the algorithm shifts over one cell and sorts 1, 5, and 9.
- This process continues until all the elements have been *4-sorted*, which means that all items spaced 4 cells apart are sorted among themselves.
- The process is shown (using a more compact visual metaphor) in Figure 7.2.
- After the complete 4-sort, the array can be thought of as comprising four subarrays: (0,4,8), (1,5,9), (2,6), and (3,7), each of which is completely sorted.
- These subarrays are interleaved, but otherwise independent.

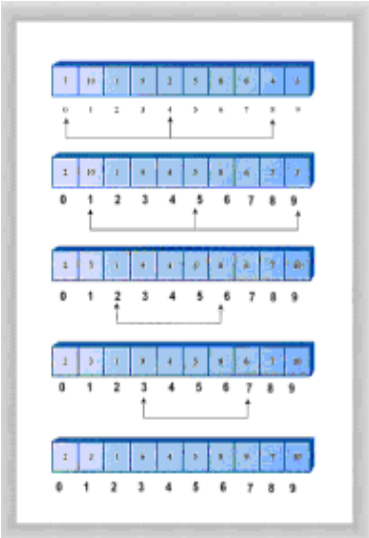


Figure 7.2: A complete 4-sort

9

- Notice that, in this particular example, at the end of the 4-sort no item is more than 2 cells from where it would be if the array were completely sorted.
- This is what is meant by an array being "almost" sorted and is the secret of the Shellsort.
- By creating interleaved, internally sorted sets of items, we minimize the amount of work that must be done to complete the sort.
- Now, as we noted in Chapter 3, the insertion sort is very efficient when operating on an array that's almost sorted.
  - If it only needs to move items one or two cells to sort the file, it can operate in almost O(N) time.
- Thus after the array has been 4-sorted, we can 1-sort it using the ordinary insertion sort.
- The combination of the 4-sort and the 1-sort is much faster than simply applying the ordinary insertion sort without the preliminary 4-sort.

10

Diminishing Gaps

- We've shown an initial interval—or gap—of 4 cells for sorting a 10-cell array.
- For larger arrays the gap should start out much larger.
- The interval is then repeatedly reduced until it becomes 1.
  - For instance, an array of 1,000 items might be 364-sorted, then 121-sorted, then 40-sorted, then 13-sorted, then 4-sorted, and finally 1-sorted.
- The sequence of numbers used to generate the intervals (in this example 364, 121, 40, 13, 4, 1) is called the *interval sequence* or *gap sequence*.
- The particular interval sequence shown here, attributed to Knuth, is a popular one.
  - In reversed form, starting from 1, it's generated by the recursive expression

    ```
    h = 3*h + 1
    ```

    where the initial value of h is 1.

- There are other approaches to generating the interval sequence (Later).

11

How The Shellsort Works Using Knuth's Sequence

- In the sorting algorithm, the sequence-generating formula is first used in a short loop to figure out the initial gap.
- A value of 1 is used for the first value of h, and the h=h*3+1 formula is applied to generate the sequence 1, 4, 13, 40, 121, 364, and so on.
- This process ends when the gap is larger than the array.
- For a 1,000-element array, the 7th number in the sequence, 1093, is too large.
- Thus we begin the sorting process with the 6th-largest number, creating a 364-sort.
- Then, each time through the outer loop of the sorting routine, we reduce the interval using the inverse of the formula previously given:

  ```
  h = (h–1) / 3
  ```

  as shown in the third column of Table 7.1.
- This inverse formula generates the reverse sequence 364, 121, 40, 13, 4, 1.
- Starting with 364, each of these numbers is used to n-sort the array.
- When the array has been 1-sorted, the algorithm is done.

12

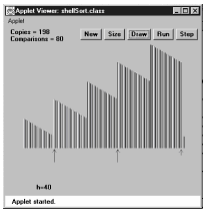
The ShellSort Workshop Applet

- In single-stepping through the algorithm, you'll notice that the explanation we gave in the last section is slightly simplified.
  - The sequence for the 4-sort is not actually (0,4,8), (1,5,9), (2,6), and (3,7).
  - Instead the first two elements of each group of three are sorted first, then the first two elements of the second group, and so on.
  - Once the first two elements of all the groups are sorted, the algorithm returns and sorts three-element groups.
  - The actual sequence is (0,4), (1,5), (2,6), (3,7), (0,4,8), (1,5,9).
- It might seem more obvious for the algorithm to 4-sort each complete subarray first: (0,4), (0,4,8), (1,5), (1,5,9), (2,6), (3,7), but the algorithm handles the array indices more efficiently using the first scheme.

- The Shellsort is actually not very efficient with only 10 items, making almost as many swaps and comparisons as the insertion sort.
- However, with 100 bars the improvement becomes significant.

13

The ShellSort Workshop Applet
(100 inversely sorted bars)



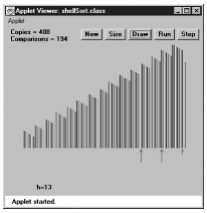


Figure 7.4: After the 40-sort

Figure 7.5: After the 13-sort

With each new value of h, the array becomes more nearly sorted.

14

Why Is The Shellsort So Much Faster (than the insertion sort)

- When h is large, the number of items per pass is small, and items move long distances. This is very efficient.
- As h grows smaller, the number of items per pass increases, but the items are already closer together, which is more efficient for the insertion sort.
- It's the combination of these trends that makes the Shellsort so effective.

- Notice that later sorts (small values of h) don't undo the work of earlier sorts (large values of h).
- An array that has been 40-sorted remains 40-sorted after a 13-sort, for example.
- If this wasn't so the Shellsort couldn't work.

15

```
// The Java code for the Shellsort is scarcely more complicated than for the insertion sort.
// Starting with the insertion sort, you substitute h for 1 in appropriate places and add the
// formula to generate the interval sequence.
//
// shellSort.java
// demonstrates shell sort
// to run this program: C>java ShellSortApp
//--------------------------------------------------------------
class ArraySh
   {
   private long[] theArray;          // ref to array theArray
   private int nElems;               // number of data items
//--------------------------------------------------------------
   public ArraySh(int max)           // constructor
      {
      theArray = new long[max];      // create the array
      nElems = 0;                    // no items yet
      }
//--------------------------------------------------------------
   public void insert(long value)    // put element into array
      {
      theArray[nElems] = value;      // insert it
      nElems++;                      // increment size
      }
//--------------------------------------------------------------
   public void display()             // displays array contents
      {
      System.out.print("A=");
      for(int j=0; j<nElems; j++)    // for each element,
         System.out.print(theArray[j] + " ");  // display it
      System.out.println("");
      }
```

16

```
//--------------------------------------------------------------
   public void shellSort()
      {
      int inner, outer;
      long temp;

      int h = 1;                    // find initial value of h
      while(h <= nElems/3)
         h = h*3 + 1;               // (1, 4, 13, 40, 121, ...)

      while(h>0)                    // decreasing h, until h=1
         {
                                    // h-sort the file
      for(outer=h; outer<nElems; outer++)
         {
         temp = theArray[outer];
         inner = outer;
                                    // one subpass (eg 0, 4, 8)
         while(inner > h-1 && theArray[inner-h] >=  temp)
            {
            theArray[inner] = theArray[inner-h];
            inner -= h;
            }
         theArray[inner] = temp;
         }  // end for
      h = (h-1) / 3;                // decrease h
      }  // end while(h>0)
   }  // end shellSort()
//--------------------------------------------------------------
   }  // end class ArraySh
```

17

```
////////////////////////////////////////////////////////////////
class ShellSortApp
   {
   public static void main(String[] args)
      {
      int maxSize = 10;             // array size
      ArraySh arr;
      arr = new ArraySh(maxSize);   // create the array

      for(int j=0; j<maxSize; j++)  // fill array with
         {                          // random numbers
         long n = (int)(java.lang.Math.random()*99);
         arr.insert(n);
         }
      arr.display();                // display unsorted array
      arr.shellSort();              // shell sort the array
      arr.display();                // display sorted array
      }  // end main()
   }  // end class ShellSortApp
////////////////////////////////////////////////////////////////
```

You can change `maxSize` to higher numbers, but don't go too high; 10,000 items take a fraction of a minute to sort.

In `main()` we create an object of type `ArraySh`, capable of holding 10 items, fill it with random data, display it, Shellsort it, and display it again.

Here's some sample output:

```
A=20 89 6 42 55 59 41 69 75 66
A=6 20 41 42 55 59 66 69 75 89
```

18

Other Interval Sequences

- Picking an interval sequence is a bit of a black art.
- Our discussion so far used the formula h=h*3+1 to generate the interval sequence, but other interval sequences have been used with varying degrees of success.
- The only absolute requirement is that the diminishing sequence ends with 1, so the last pass is a normal insertion sort.

19

Interval Sequences

- In Shell's original paper, he suggested an initial gap of N/2, which was simply divided in half for each pass. Thus the descending sequence for N=100 is 50, 25, 12, 6, 3, 1.
  - This approach has the advantage that you don't need to calculate the sequence before the sort begins to find the initial gap; you just divide N by 2.
  - However, this turns out not to be the best sequence. Although it's still better than the insertion sort for most data, it sometimes degenerates to $O(N^2)$ running time, which is no better than the insertion sort.

20

Interval Sequences

- A better approach is to divide each interval by 2.2 instead of 2.
- For n=100 this leads to 45, 20, 9, 4, 1.
- This is considerably better than dividing by 2, as it avoids some worst case circumstances that lead to $O(N^2)$ behavior.
- Some extra code is needed to ensure that the last value in the sequence is 1, no matter what N is.
- This gives results comparable to Knuth's sequence shown in the listing.

21

Interval Sequences

- Another possibility for a descending sequence (from Flamig; see Appendix B, "Further Reading") is:

```
if(h < 5)
  h = 1;
else
  h = (5*h-1) / 11;
```

22

Interval Sequences

- It's generally considered important that the numbers in the interval sequence are relatively prime; that is, they have no common divisors except 1.
  - This makes it more likely that each pass will intermingle all the items sorted on the previous pass.
  - The inefficiency of Shell's original N/2 sequence is due to its failure to adhere to this rule.
- Invent a gap sequence of your own that does just as well (or possibly even better) than those shown.
  - Whatever it is, it should be quick to calculate so as not to slow down the algorithm.

23

Efficiency of the ShellSort

- No one so far has been able to analyze the Shellsort's efficiency theoretically, except in special cases.
- Based on experiments, there are various estimates, which range from $O(N^{3/2})$ down to $O(N^{7/6})$.
- Table 7.2 shows some of these estimated O() values, compared with the slower insertion sort and the faster quicksort.
- The theoretical times corresponding to various values of N are shown.
- Note that $N^{x/y}$ means the y^{th} root of N raised to the x power. Thus if N is 100, $N^{3/2}$ is the square root of 100^3, which is 1,000. Also, $(\log N)^2$ means the log of N, squared. This is often written $\log^2 N$, but that's easy to confuse with $\log_2 N$, the logarithm to the base 2 of N.

24

Efficiency of the ShellSort

O() Value	Type of Sort	10 Items	100 Items	1,000 Items	10,000 Items
N^2	Insertion, etc.	100	10,000	1,000,000	100,000,000
$N^{3/2}$	Shellsort	32	1,000	32,000	1,000,000
$N*(\log N)^2$	Shellsort	10	400	9,000	160,000
$N^{5/4}$	Shellsort	18	316	5,600	100,000
$N^{7/6}$	Shellsort	14	215	3,200	46,000
N*logN	Quicksort, etc.	10	200	3,000	40,000

Table 7.2: Estimates of ShellSort Running Time

For most data, the higher estimates, such as $N^{3/2}$, are probably more realistic.

25

Partitioning

- Partitioning is the underlying mechanism of quicksort.
- It's also a useful operation on its own.
- To *partition* data is to divide it into two groups, so that all the items with a key value higher than a specified amount are in one group, and all the items with a lower key value are in another.

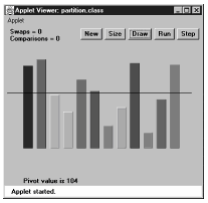
26

Partitioning data: Situations

- Divide your personnel records into two groups: employees who live within 15 miles of the office and those who live farther away.
- A school administrator might want to divide students into those with grade point averages higher and lower than 3.5, so as to know who deserves to be on the Dean's list.

27

Workshop Applet: Partition



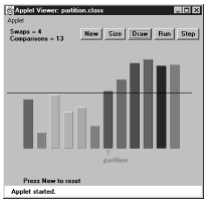


Figure 7.6: Twelve bars before partitioning **Figure 7.7:** Twelve bars after partitioning

- The horizontal line represents the *pivot value*. This is the value used to determine into which of the two groups an item is placed.
- Items with a key value less than the pivot value go in the left part of the array, and those with a greater (or equal) key go in the right part. (In the section on quicksort, we'll see that the pivot value can be the key value of an actual data item, called the pivot. For now, it's just a number.)
- The arrow labeled *partition* points to the leftmost item in the right (higher) subarray. This value is returned from the partitioning method, so it can be used by other methods that need to know where the division is.

28

Workshop Applet: Partition (100 bars)

- Run it
- The `leftScan` and `rightScan` pointers will zip toward each other, swapping bars as they go. When they meet, the partition is complete.

29

Partition

- You can choose any value you want for the pivot value, depending on why you're doing the partition
  - such as choosing a grade point average of 3.5.
- For variety, the Workshop applet chooses a random number for the pivot value (the horizontal black line) each time New or Size is pressed, but the value is never too far from the average bar height.
- After being partitioned, the data is by no means sorted;
  - it has simply been divided into two groups.
  - However, it's more sorted than it was before.
  - It doesn't take much more trouble to sort it completely (next section).
- Partitioning is not *stable*.
  - That is, each group is not in the same order it was originally.
  - In fact, partitioning tends to reverse the order of some of the data in each group.

30

```
// partition.java
// partitionIt() method partitions an array.
// to run this program: C>java PartitionApp
////////////////////////////////////////////////////////////////
class ArrayPar
   {
   private long[] theArray;          // ref to array theArray
   private int nElems;               // number of data items
//--------------------------------------------------------------
   public ArrayPar(int max)          // constructor
      {
      theArray = new long[max];      // create the array
      nElems = 0;                    // no items yet
      }
//--------------------------------------------------------------
   public void insert(long value)    // put element into array
      {
      theArray[nElems] = value;      // insert it
      nElems++;                      // increment size
      }
//--------------------------------------------------------------
   public int size()                 // return number of items
      { return nElems; }
//--------------------------------------------------------------
   public void display()             // displays array contents
      {
      System.out.print("A=");
      for(int j=0; j<nElems; j++)    // for each element,
         System.out.print(theArray[j] + " ");  // display it
      System.out.println("");
      }
//--------------------------------------------------------------
```

31

```
 public int partitionIt(int left, int right, long pivot)
      {
      int leftPtr = left - 1;           // right of first elem
      int rightPtr = right + 1;         // left of pivot
      while(true)
         {
         while(leftPtr < right &&       // find bigger item
               theArray[++leftPtr] < pivot)
            ;  // (nop)

         while(rightPtr > left &&       // find smaller item
               theArray[--rightPtr] > pivot)
            ;  // (nop)
         if(leftPtr >= rightPtr)        // if pointers cross,
            break;                      //    partition done
         else                           // not crossed, so
            swap(leftPtr, rightPtr);    //    swap elements
         }  // end while(true)
       return leftPtr;                  // return partition
       }  // end partitionIt()
//--------------------------------------------------------------
 public void swap(int dex1, int dex2)  // swap two elements
     {
     long temp;
     temp = theArray[dex1];             // A into temp
     theArray[dex1] = theArray[dex2];   // B into A
     theArray[dex2] = temp;             // temp into B
     }  // end swap()
//--------------------------------------------------------------
   }  // end class ArrayPar
```

32

```
////////////////////////////////////////////////////////////////
class PartitionApp
   {
   public static void main(String[] args)
      {
      int maxSize = 16;            // array size
      ArrayPar arr;                // reference to array
      arr = new ArrayPar(maxSize);  // create the array

      for(int j=0; j<maxSize; j++)  // fill array with
         {                          // random numbers
         long n = (int)(java.lang.Math.random()*199);
         arr.insert(n);
         }
      arr.display();               // display unsorted array

      long pivot = 99;             // pivot value
      System.out.print("Pivot is " + pivot);
      int size = arr.size();
                                   // partition array
      int partDex = arr.partitionIt(0, size-1, pivot);

      System.out.println(", Partition is at index " + partDex);
      arr.display();               // display partitioned array
      }  // end main()
   }  // end class PartitionApp
////////////////////////////////////////////////////////////////
```


Partition Workshop Applet

- The `main()` routine creates an `ArrayPar` object that holds 16 items of type `double`.
- The pivot value is fixed at 99.
- The routine inserts 16 random values into `ArrayPar`, displays them, partitions them by calling the `partitionIt()` method, and displays them again. Here's some sample output:

```
A=149 192 47 152 159 195 61 66 17 167 118 64 27 80 30
  105
Pivot is 99, partition is at index 8
A=30 80 47 27 64 17 61 66 195 167 118 159 152 192 149
  105
```

- You can see that the partition is successful: The first eight numbers are all smaller than the pivot value of 99; the last eight are all larger.
- Notice that the partitioning process doesn't necessarily divide the array in half as it does in this example; that depends on the pivot value and key values of the data.
- There may be many more items in one group than in the other.

The Partition Algorithm

- The partitioning algorithm works by starting with two pointers, one at each end of the array.
  - We use the term *pointers* to mean indices that point to array elements, not C++ pointers.
- The pointer on the left, `leftPtr`, moves toward the right, and the one of the right, `rightPtr`, moves toward the left.
- Notice that `leftPtr` and `rightPtr` in the `partition.java` program correspond to `leftScan` and `rightScan` in the Partition Workshop applet.

- Actually, `leftPtr` is initialized to one position to the left of the first cell, and `rightPtr` to one position to the right of the last cell, because they will be incremented and decremented, respectively, before they're used.

35

The Partition Algorithm: Stopping and Swapping

- When `leftPtr` encounters a data item smaller than the pivot value, it keeps going, because that item is in the right place.
- However, when it encounters an item larger than the pivot value, it stops.
- Similarly, when `rightPtr` encounters an item larger than the pivot, it keeps going, but when it finds a smaller item, it also stops.
- Two inner `while` loops, the first for `leftPtr` and the second for `rightPtr`, control the scanning process.
- A pointer stops because its `while` loop exits.

- Here's a simplified version of the code that scans for out-of-place items:

```
while( theArray[++leftPtr] < pivot ) // find bigger item
   ; // (nop)
while( theArray[--rightPtr] > pivot ) // find smaller item
   ; // (nop)
swap(leftPtr, rightPtr); // swap elements
```

36

The Partition Algorithm

- The first `while` loop exits when an item larger than pivot is found; the second loop exits when an item smaller than pivot is found.
- When both these loops exit, both `leftPtr` and `rightPtr` point to items that are in the wrong part of the array, so these items are swapped.
- After the swap, the two pointers continue on, again stopping at items that are in the wrong part of the array and swapping them.
- All this activity is nested in an outer `while` loop, as can be seen in the `partitionIt()` method in Listing 7.2.
- When the two pointers eventually meet, the partitioning process is complete and this outer `while` loop exits.

37

Partition Workshop Applet 100 bars

- The pointers in action: Represented by blue arrows, start at opposite ends of the array and move toward each other, stopping and swapping as they go.
- The bars between them are unpartitioned; those they've already passed over are partitioned.
- When they meet, the entire array is partitioned.

38

Handling Unusual Data

- If we were sure that there was a data item at the right end of the array that was smaller than the pivot value, and an item at the left end that was larger, the simplified while loops previously shown would work fine.
- Unfortunately, the algorithm may be called upon to partition data that isn't so well organized.
  - If all the data is smaller than the pivot value, for example, the `leftPtr` variable will go all the way across the array, looking in vain for a larger item, and fall off the right end, creating an `array index out of bounds` exception.
  - A similar fate will befall `rightPtr` if all the data is larger than the pivot value.
- To avoid these problems, extra tests must be placed in the `while` loops to check for the ends of the array: `leftPtr<right` in the first loop, and `rightPtr>left` in the second.
  - This can be seen in context in Listing 7.2.
- In the section on quicksort, we'll see that a clever pivot-selection process can eliminate these end-of-array tests. Eliminating code from inner loops is always a good idea if you want to make a program run faster.

39

Delicate Code

- The code in the `while` loops is rather delicate. For example, you might be tempted to remove the increment operators from the inner `while` loops and use them to replace the nop statements. (Nop refers to a statement consisting only of a semicolon, and means *no operation*.)

  For example, you might try to change this:

```
while(leftPtr < right && theArray[++leftPtr] < pivot)
   ; // (nop)
```

  to this:

```
while(leftPtr < right && theArray[leftPtr] < pivot)
   ++leftPtr;
```

  and similarly for the other inner `while` loop.

  - This would make it possible for the initial values of the pointers to be `left` and `right`, which is somewhat clearer than `left-1` and `right+1`.

40

Delicate Code

- However, these changes result in the pointers being incremented only when the condition is satisfied.
- The pointers must move in any case, so two extra statements within the outer `while` loop would be required to bump the pointers. The nop version is the most efficient solution.

41

Efficiency of the Partition Algorithm

- The partition algorithm runs in O(N) time.
- It's easy to see this when running the Partition Workshop applet:
  - the two pointers start at opposite ends of the array and move toward each other at a more or less constant rate, stopping and swapping as they go.
  - When they meet, the partition is complete
  - If there were twice as many items to partition, the pointers would move at the same rate, but they would have twice as far to go (twice as many items to compare and swap), so the process would take twice as long.
  - Thus the running time is proportional to N.

42

Efficiency of the Partition Algorithm

- More specifically, for each partition there will be N+1 or N+2 comparisons.
- Every item will be encountered and used in a comparison by one or the other of the pointers, leading to N comparisons, but the pointers overshoot each other before they find out they've "crossed" or gone beyond each other, so there are one or two extra comparisons before the partition is complete.
- The number of comparisons is independent of how the data is arranged (except for the uncertainty between 1 and 2 extra comparisons at the end of the scan).

43

Efficiency of the Partition Algorithm

- The number of swaps, however, does depend on how the data is arranged.
- If it's inversely ordered and the pivot value divides the items in half, then every pair of values must be swapped, which is N/2 swaps.
- Remember in the Partition Workshop applet that the pivot value is selected randomly, so that the number of swaps for inversely sorted bars won't always be exactly N/2.

44

Efficiency of the Partition Algorithm

- For random data, there will be fewer than N/2 swaps in a partition, even if the pivot value is such that half the bars are shorter and half are taller.
  - This is because some bars will already be in the right place (short bars on the left, tall bars on the right).
- If the pivot value is higher (or lower) than most of the bars, there will be even fewer swaps because only those few bars that are higher (or lower) than the pivot will need to be swapped.
- On average, for random data, about half the maximum number of swaps take place.

45

Efficiency of the Partition Algorithm

- Although there are fewer swaps than comparisons, they are both proportional to N.
- Thus the partitioning process runs in O(N) time.
  - Running the Workshop applet, we can see that for 12 random bars there are about 3 swaps and 14 comparisons, and for 100 random bars there are about 25 swaps and 102 comparisons.

46

Quicksort

- Quicksort is undoubtedly the most popular sorting algorithm
  - in the majority of situations, it's the fastest, operating in O(N*logN) time.
    - (This is only true for *internal* or in-memory sorting; for sorting data in disk files other methods may be better.)
- Discovered by C.A.R. Hoare in 1962.
- To understand quicksort
  - Be familiar with the partitioning algorithm
  - Basically the quicksort algorithm operates by partitioning an array into two subarrays, and then calling itself to quicksort each of these subarrays.
  - However, there are some embellishments we can make to this basic scheme.
    - These have to do with the selection of the pivot and the sorting of small partitions.
    - We'll examine these refinements after we've looked at a simple version of the main algorithm.
- It's difficult to understand what quicksort is doing before you understand how it does it
  - so we'll reverse our usual presentation and show the Java code for quicksort before presenting the quicksort Workshop applet.

47

```
public void recQuickSort(int left, int right)
      {
      if(right-left <= 0)              // if size <= 1,
          return;                      //    already sorted
      else                             // size is 2 or larger
         {
                                      // partition range
         int partition = partitionIt(left, right);
         recQuickSort(left, partition-1);   // sort left side
         recQuickSort(partition+1, right);  // sort right side
         }
      }  // end recQuickSort()
```

48

Quicksort

- There are three basic steps:
  1. Partition the array or subarray into left (smaller keys) and right (larger keys) groups.
  2. Call ourselves to sort the left group.
  3. Call ourselves again to sort the right group.
- After a partition, all the items in the left subarray are smaller than all those on the right.
- If we then sort the left subarray and sort the right subarray, the entire array will be sorted.
- How do we sort these subarrays? By calling ourself.

49

Quicksort

- The arguments to the `recQuickSort()` method determine the left and right ends of the array (or subarray) it's supposed to sort.
- The method first checks whether this array consists of only one element.
  - If so, the array is by definition already sorted, and the method returns immediately.
  - This is the base case in the recursion process.
- If the array has two or more cells, the algorithm calls the `partitionIt()` method to partition it.
  - This method returns the index number of the *partition*: the left element in the right (larger keys) subarray.
  - The partition marks the boundary between the subarrays.
  - This is shown in Figure 7.8.

50

Quicksort

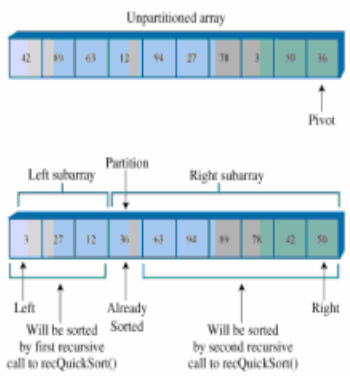


Figure 7.8: Recursive calls sort subarrays

51

Quicksort

- Once the array is partitioned,
  - recQuickSort() calls itself recursively,
    - once for the left part of its array, from `left` to `partition-1`,
    - once for the right part, from `partition+1` to `right`.
- Note that the data item at the index `partition` is not included in either of the recursive calls.
  - Why not?
  - Doesn't it need to be sorted?
  - The explanation lies in how the pivot value is chosen.

52

Quicksort: Choosing a Pivot Value

- What pivot value should the `partitionIt()` method use?
  - Here are some relevant ideas:
    - The pivot value should be the key value of an actual data item; this item is called the *pivot*.
    - You can pick a data item to be the pivot more or less at random. For simplicity, let's say we always pick the item on the right end of the subarray being partitioned.
    - After the partition, if the pivot is inserted at the boundary between the left and right subarrays, it will be in its final sorted position.
- Because the pivot's key value is used to partition the array, then, following the partition,
  - the left subarray holds items smaller than the pivot, and
  - the right subarray holds items larger.
- The pivot starts out on the right, but if it could somehow be placed between these two subarrays, it would be in the right place; that is, in its final sorted position.
- Figure 7.9 shows how this looks with a pivot whose key value is 36.

53

Quicksort: Choosing a Pivot Value

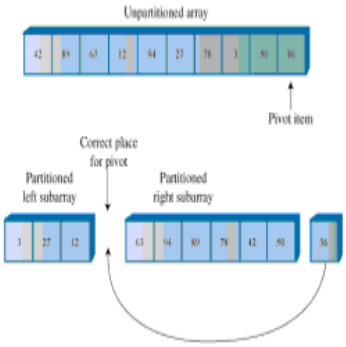


Figure 7.9: The pivot and the subarrays

54

How Do We Move The Pivot To Its Proper Place?

- We could shift all the items in the right subarray to the right one cell to make room for the pivot.
- However, this is inefficient and unnecessary. Remember that
  - all the items in the right subarray, although they are larger than the pivot, are not yet sorted, so they can be moved around within the right subarray without affecting anything.
- Therefore,
  - to simplify inserting the pivot in its proper place, we can simply swap the pivot (36) and the left item in the right subarray, which is 63.
  - This places the pivot in its proper position between the left and right groups.
  - The 63 is switched to the right end, but, because it remains in the right (larger) group, the partitioning is undisturbed.
- This is shown in Figure 7.10.

55

Quicksort

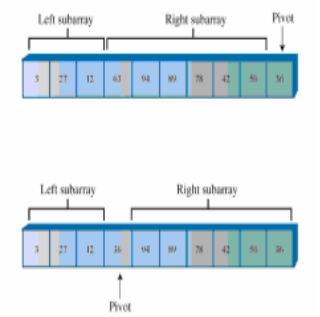


Figure 7.10: Swapping the pivot

- Once it's swapped into the partition's location, the pivot is in its final resting place.
- All subsequent activity will take place on one side of it or on the other, but the pivot itself won't be moved (or indeed even accessed) again.

56

To incorporate the pivot selection process into our `recQuickSort()` method, let's make it an overt statement, and send the pivot value to `partitionIt()` as an argument.

```
public void recQuickSort(int left, int right)
     {
     if(right-left <= 0)              // if size <= 1,
         return;                      //    already sorted
     else                             // size is 2 or larger
        {
        long pivot = theArray[right];      // rightmost item
                                           // partition range
        int partition = partitionIt(left, right, pivot);
        recQuickSort(left, partition-1);   // sort left side
        recQuickSort(partition+1, right);  // sort right side
        }
     }  // end recQuickSort()
```

57

Quicksort

- When we use this scheme of choosing the rightmost item in the array as the pivot, we'll need to modify the `partitionIt()` method to exclude this rightmost item from the partitioning process;
- After all, we already know where it should go after the partitioning process is complete:
  - at the partition, between the two groups.
- Also, once the partitioning process is completed, we need to swap the pivot from the right end into the partition's location.
- Listing 7.3 shows the `quickSort1.java` program, which incorporates these features.

58

```
// quickSort1.java
// demonstrates simple version of quick sort
// to run this program: C>java QuickSort1App
////////////////////////////////////////////////////////////////
class ArrayIns
   {
   private long[] theArray;          // ref to array theArray
   private int nElems;               // number of data items
//--------------------------------------------------------------
   public ArrayIns(int max)          // constructor
      {
      theArray = new long[max];      // create the array
      nElems = 0;                    // no items yet
      }
//--------------------------------------------------------------
   public void insert(long value)    // put element into array
      {
      theArray[nElems] = value;      // insert it
      nElems++;                      // increment size
      }
//--------------------------------------------------------------
   public void display()             // displays array contents
      {
      System.out.print("A=");
      for(int j=0; j<nElems; j++)    // for each element,
         System.out.print(theArray[j] + " ");  // display it
      System.out.println("");
      }
```

59

```
//--------------------------------------------------------------
   public void quickSort()
      {
      recQuickSort(0, nElems-1);
      }
//--------------------------------------------------------------
   public void recQuickSort(int left, int right)
      {
      if(right-left <= 0)              // if size <= 1,
          return;                      //    already sorted
      else                             // size is 2 or larger
         {
         long pivot = theArray[right];      // rightmost item
                                            // partition range
         int partition = partitionIt(left, right, pivot);
         recQuickSort(left, partition-1);   // sort left side
         recQuickSort(partition+1, right);  // sort right side
         }
      }  // end recQuickSort()
```

60

```
//--------------------------------------------------------------
    public int partitionIt(int left, int right, long pivot)
       {
       int leftPtr = left-1;           // left    (after ++)
       int rightPtr = right;           // right-1 (after --)
       while(true)
          {                            // find bigger item
          while( theArray[++leftPtr] < pivot )
             ;  // (nop)
                                       // find smaller item
          while(rightPtr > 0 && theArray[--rightPtr] > pivot)
             ;  // (nop)

          if(leftPtr >= rightPtr)      // if pointers cross,
             break;                    //    partition done
          else                         // not crossed, so
             swap(leftPtr, rightPtr);  //    swap elements
          }  // end while(true)
       swap(leftPtr, right);           // restore pivot
       return leftPtr;                 // return pivot location
       }  // end partitionIt()
//--------------------------------------------------------------
   public void swap(int dex1, int dex2)  // swap two elements
      {
      long temp = theArray[dex1];        // A into temp
      theArray[dex1] = theArray[dex2];   // B into A
      theArray[dex2] = temp;             // temp into B
      }  // end swap(
//--------------------------------------------------------------
   }  // end class ArrayIns
```

61

```
////////////////////////////////////////////////////////////////
class QuickSort1App
   {
   public static void main(String[] args)
      {
      int maxSize = 16;            // array size
      ArrayIns arr;
      arr = new ArrayIns(maxSize);  // create array

      for(int j=0; j<maxSize; j++)  // fill array with
         {                          // random numbers
         long n = (int)(java.lang.Math.random()*99);
         arr.insert(n);
         }
      arr.display();               // display items
      arr.quickSort();             // quicksort them
      arr.display();               // display them again
      }  // end main()
   }  // end class QuickSort1App
////////////////////////////////////////////////////////////////
```

The `main()` routine creates an object of type `ArrayIns`, inserts 16 random data items of type `double` in it, displays it, sorts it with the `quickSort()` method, and displays the results.

Here's some typical output:

```
A=69 0 70 6 38 38 24 56 44 26 73 77 30 45 97 65
A=0 6 24 26 30 38 38 44 45 56 65 69 70 73 77 97
```

62

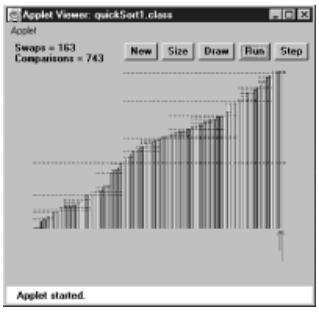
Quicksort

- An interesting aspect of the code in the `partitionIt()` method is that we've been able to remove the test for the end of the array in the first inner `while` loop.
  - This test, seen in the earlier `partitionIt()` method in the `partition.java` program in Listing 7.2, was
    ```
    leftPtr < right
    ```
  - It prevented `leftPtr` running off the right end of the array if there was no item there larger than `pivot`.
  - Why can we eliminate the test?
    - Because we selected the rightmost item as the pivot, so `leftPtr` will always stop there.
- However, the test is still necessary for `rightPtr` in the second `while` loop.
  - Later we'll see how this test can be eliminated as well.
- Choosing the rightmost item as the pivot is thus not an entirely arbitrary choice:
  - it speeds up the code by removing an unnecessary test.
  - Picking the pivot from some other location would not provide this advantage.

63

The quickSort1 Workshop Applet

- **The Big Picture**: sort 100 random bars



- The Details: See textbook

64

The quickSort1 Workshop Applet

- Things to Notice: See textbook

65

Quicksort: Degenerates to O(N^2) Performance

- If you use the quickSort1 Workshop applet to sort 100 inversely sorted bars, you'll see that the algorithm runs much more slowly and that many more dotted horizontal lines are generated, indicating more and larger subarrays are being partitioned.

- What's happening here?

66

Quicksort: Degenerates to O(N^2) Performance

- The problem is in the selection of the pivot.
  - Ideally, the pivot should be the median of the items being sorted.
    - That is, half the items should be larger than the pivot, and half smaller.
  - This would result in the array being partitioned into two subarrays of equal size.
    - Two equal subarrays is the optimum situation for the quicksort algorithm.
  - If it has to sort one large and one small array,
    - it's less efficient because the larger subarray has to be subdivided more times.

67

Quicksort: Degenerates to O(N^2) Performance

- The worst situation results when a subarray with N elements is divided into one subarray with 1 element and the other with N–1 elements.
  - This division into 1 cell and N–1 cells can also be seen in steps 3 and 9 in Figure 7.12.
  - If this 1 and N–1 division happens with every partition, then every element requires a separate partition step.
  - This is in fact what takes place with inversely sorted data:
    - in all the subarrays, the pivot is the smallest item, so every partition results in an N–1 element in one subarray and only the pivot in the other.

68

Quicksort: Degenerates to O(N^2) Performance

- To see this unfortunate process in action,
  - step through the quickSort1 Workshop applet with 12 inversely sorted bars.
    - Notice how many more steps are necessary than with random data.
  - In this situation the advantage gained by the partitioning process is lost and the performance of the algorithm degenerates to O(N^2).
- Besides being slow, there's another potential problem when quicksort operates in O(N^2) time.
  - When the number of partitions increases, the number of recursive function calls also increases.
  - Every function call takes up room on the machine stack.
    - If there are too many calls, the machine stack may overflow and paralyze the system.

69

Quicksort: Degenerates to O(N^2) Performance

- To summarize:
  - In the quickSort1 applet, we select the rightmost element as the pivot.
  - If the data is truly random, this isn't too bad a choice, because usually the pivot won't be too close to either end of the array.
  - However, when the data is sorted or inversely sorted, choosing the pivot from one end or the other is a bad idea.
- Can we improve on our approach to selecting the pivot?

70

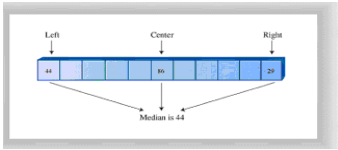
Median of Three Partitioning

- Many schemes have been devised for picking a better pivot.
- The method should be simple but have a good chance of avoiding the largest or smallest value.
- Picking an element at random is simple but—as we've seen—doesn't always result in a good selection.
- However, we could examine all the elements and actually calculate which one was the median.
  - This would be the ideal pivot choice, but the process isn't practical, as it would take more time than the sort itself.

71

Median of Three Partitioning

- A compromise solution is to find the median of the first, last, and middle elements of the array, and use this for the pivot.
  - The *median* or *middle* item is the data item chosen so that exactly half the other items are smaller and half are larger.
- Picking the median of the first, last, and middle elements is called the *median-of-three* approach and is shown here in Figure 7.13.



Median of Three Partitioning

- Finding the median of three items is obviously much faster than finding the median of all the items, and yet it successfully avoids picking the largest or smallest item in cases where the data is already sorted or inversely sorted.
  - There are probably some pathological arrangements of data where the median-of-three scheme works poorly, but normally it's a fast and effective technique for finding the pivot.
- Besides picking the pivot more effectively, the median of three approach has an additional benefit:
  - We can dispense with the `rightPtr>left` test in the second inside `while` loop, leading to a small increase in the algorithm's speed.
  - How is this possible?
    - Because we can use the median-of-three approach to not only select the pivot, but also to sort the three elements used in the selection process.
    - Figure 7.14 in the next slide shows how this looks.

73

Median of Three Partitioning

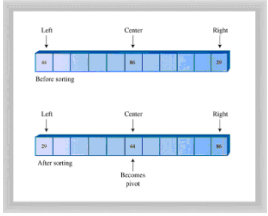


Figure 7.14: Sorting the left, center, and right elements

- Once these three elements are sorted, and the median item is selected as the pivot,
  - we are guaranteed that the element at the left end of the subarray is less than (or equal to) the pivot,
  - and the element at the right end is greater than (or equal to) the pivot.
  - This means that the `leftPtr` and `rightPtr` indices can't step beyond the right or left ends of the array, respectively, even if we remove the `leftPtr>right` and `rightPtr<left` tests.
  - The pointer will stop, thinking it needs to swap the item, only to find that it has crossed the other pointer and the partition is complete.
  - The values at `left` and `right` act as *sentinels* to keep `leftPtr` and `rightPtr` confined to valid array values.

74

Median of Three Partitioning

- Another small benefit:
  - after the left, center, and right elements are sorted, the partition process doesn't need to examine these elements again.
  - The partition can begin at `left+1` and `right-1`, because `left` and `right` have in effect already been partitioned.
  - We know that `left` is in the correct partition because it's on the left and it's less than the pivot,
  - and `right` is in the correct place because it's on the right and it's greater than the pivot.
- Thus, median-of-three partitioning
  - not only avoids $O(N^2)$ performance for already sorted data,
  - it also allows us to speed up the inner loops of the partitioning algorithm
  - and reduce slightly the number of items that must be partitioned.

75

```
// quickSort2.java
// demonstrates quick sort with median-of-three partitioning
// use a separate method, medianOf3(), to sort the left, center, and right elements of a subarray.
// This method returns the value of the pivot, which is then sent to the partitionIt() method.
// to run this program: C>java QuickSort2App
////////////////////////////////////////////////////////////////
class ArrayIns
   {
   private long[] theArray;          // ref to array theArray
   private int nElems;               // number of data items
//--------------------------------------------------------------
   public ArrayIns(int max)          // constructor
      {
      theArray = new long[max];      // create the array
      nElems = 0;                    // no items yet
      }
//--------------------------------------------------------------
   public void insert(long value)    // put element into array
      {
      theArray[nElems] = value;      // insert it
      nElems++;                      // increment size
      }
//--------------------------------------------------------------
   public void display()             // displays array contents
      {
      System.out.print("A=");
      for(int j=0; j<nElems; j++)    // for each element,
         System.out.print(theArray[j] + " ");  // display it
      System.out.println("");
      }
```

76

```
//--------------------------------------------------------------
   public void quickSort()
      {
      recQuickSort(0, nElems-1);
      }
//--------------------------------------------------------------
   public void recQuickSort(int left, int right)
      {
      int size = right-left+1;
      if(size <= 3)                  // manual sort if small
         manualSort(left, right);
      else                           // quicksort if large
         {
         long median = medianOf3(left, right);
         int partition = partitionIt(left, right, median);
         recQuickSort(left, partition-1);
         recQuickSort(partition+1, right);
         }
      }  // end recQuickSort()
//--------------------------------------------------------------
```

77

```
//--------------------------------------------------------------
   public long medianOf3(int left, int right)
      {
      int center = (left+right)/2;
                                     // order left & center
      if( theArray[left] > theArray[center] )
         swap(left, center);
                                     // order left & right
      if( theArray[left] > theArray[right] )
         swap(left, right);
                                     // order center & right
      if( theArray[center] > theArray[right] )
         swap(center, right);

      swap(center, right-1);         // put pivot on right
      return theArray[right-1];      // return median value
      }  // end medianOf3()
//--------------------------------------------------------------
```

78

```
//--------------------------------------------------------------
   public void swap(int dex1, int dex2)  // swap two elements
      {
      long temp = theArray[dex1];        // A into temp
      theArray[dex1] = theArray[dex2];   // B into A
      theArray[dex2] = temp;             // temp into B
      }  // end swap(
//--------------------------------------------------------------
    public int partitionIt(int left, int right, long pivot)
       {
       int leftPtr = left;             // right of first elem
       int rightPtr = right - 1;       // left of pivot

       while(true)
          {
          while( theArray[++leftPtr] < pivot )  // find bigger
             ;                                  //    (nop)
          while( theArray[--rightPtr] > pivot ) // find smaller
             ;                                  //    (nop)
          if(leftPtr >= rightPtr)      // if pointers cross,
             break;                    //    partition done
          else                         // not crossed, so
             swap(leftPtr, rightPtr);  // swap elements
          }  // end while(true)
       swap(leftPtr, right-1);         // restore pivot
       return leftPtr;                 // return pivot location
       }  // end partitionIt()
//--------------------------------------------------------------
```

79

```
//--------------------------------------------------------------
   public void manualSort(int left, int right)
      {
      int size = right-left+1;
      if(size <= 1)
         return;         // no sort necessary
      if(size == 2)
         {               // 2-sort left and right
         if( theArray[left] > theArray[right] )
            swap(left, right);
         return;
         }
      else               // size is 3
         {               // 3-sort left, center, & right
         if( theArray[left] > theArray[right-1] )
            swap(left, right-1);               // left, center
         if( theArray[left] > theArray[right] )
            swap(left, right);                 // left, right
         if( theArray[right-1] > theArray[right] )
            swap(right-1, right);              // center, right
         }
      }  // end manualSort()
//--------------------------------------------------------------
   }  // end class ArrayIns
////////////////////////////////////////////////////////////////
```

- This program uses another new method, `manualSort()`, to sort subarrays of 3 or fewer elements.
- It returns immediately if the subarray is 1 cell (or less), swaps the cells if necessary if the range is 2, and sorts 3 cells if the range is 3.
- The `recQuickSort()` routine can't be used to sort ranges of 2 or 3 because median partitioning requires at least 4 cells.

80

```
////////////////////////////////////////////////////////////////
class QuickSort2App
   {
   public static void main(String[] args)
      {
      int maxSize = 16;            // array size
      ArrayIns arr;                // reference to array
      arr = new ArrayIns(maxSize);  // create the array

      for(int j=0; j<maxSize; j++)  // fill array with
         {                         // random numbers
         long n = (int)(java.lang.Math.random()*99);
         arr.insert(n);
         }
      arr.display();               // display items
      arr.quickSort();             // quicksort them
      arr.display();               // display them again
      }  // end main()
   }  // end class QuickSort2App
////////////////////////////////////////////////////////////////
```

The `main()` routine and the output of `quickSort2.java` are similar to those of `quickSort1.java`.

81

The quickSort2 Workshop Applet

median-of-three partitioning

- This applet is similar to the quickSort1 Workshop applet, but starts off sorting the first, center, and left elements of each subarray and selecting the median of these as the pivot value.
- At least, it does this if the array size is greater than 3.
- If the subarray is 2 or 3 units, the applet simply sorts it "by hand" without partitioning or recursive calls.
- Notice the dramatic improvement in performance when the applet is used to sort 100 inversely ordered bars.
  - No longer is every subarray partitioned into 1 cell and N–1 cells;
  - instead, the subarrays are partitioned roughly in half.
- Other than this improvement for ordered data, the quickSort2 Workshop applet produces results similar to quickSort1.
- It is no faster when sorting random data;
  - it's advantages become evident only when sorting ordered data.

82

Handling Small Partitions

- If you use the median-of-three partitioning method,
  - the quicksort algorithm won't work for partitions of three or fewer items.
  - The number 3 in this case is called a *cutoff* point.
  - In the previous examples we sorted subarrays of 2 or 3 items by hand. Is this the best way?

83

Using an Insertion Sort for Small Partitions

- Another option for dealing with small partitions is to use the insertion sort.
  - When you do this, you aren't restricted to a cutoff of 3.
  - You can set the cutoff to 10, 20, or any other number.
  - It's interesting to experiment with different values of the cutoff to see where the best performance lies.
  - Knuth recommends a cutoff of 9.
  - However, the optimum number depends on your computer, operating system, compiler (or interpreter), and so on.

84

```
// quickSort3.java
// demonstrates quick sort; uses insertion sort for cleanup
// to run this program: C>java QuickSort3App
////////////////////////////////////////////////////////////////
class ArrayIns
   {
   private long[] theArray;          // ref to array theArray
   private int nElems;               // number of data items
//--------------------------------------------------------------
   public ArrayIns(int max)          // constructor
      {
      theArray = new long[max];      // create the array
      nElems = 0;                    // no items yet
      }
//--------------------------------------------------------------
   public void insert(long value)    // put element into array
      {
      theArray[nElems] = value;      // insert it
      nElems++;                      // increment size
      }
//--------------------------------------------------------------
   public void display()             // displays array contents
      {
      System.out.print("A=");
      for(int j=0; j<nElems; j++)    // for each element,
         System.out.print(theArray[j] + " ");  // display it
      System.out.println("");
      }
//--------------------------------------------------------------
```

85

```
//--------------------------------------------------------------
   public void quickSort()
      {
      recQuickSort(0, nElems-1);
      // insertionSort(0, nElems-1); // the other option
      }
//--------------------------------------------------------------
   public void recQuickSort(int left, int right)
      {
      int size = right-left+1;
      if(size < 10)                  // insertion sort if small
         insertionSort(left, right);
      else                           // quicksort if large
         {
         long median = medianOf3(left, right);
         int partition = partitionIt(left, right, median);
         recQuickSort(left, partition-1);
         recQuickSort(partition+1, right);
         }
      }  // end recQuickSort()
//--------------------------------------------------------------
```

86

```
//--------------------------------------------------------------
   public long medianOf3(int left, int right)
      {
      int center = (left+right)/2;
                                    // order left & center
      if( theArray[left] > theArray[center] )
         swap(left, center);
                                    // order left & right
      if( theArray[left] > theArray[right] )
         swap(left, right);
                                    // order center & right
      if( theArray[center] > theArray[right] )
         swap(center, right);

      swap(center, right-1);           // put pivot on right
      return theArray[right-1];        // return median value
      }  // end medianOf3()
//--------------------------------------------------------------
   public void swap(int dex1, int dex2)  // swap two elements
      {
      long temp = theArray[dex1];        // A into temp
      theArray[dex1] = theArray[dex2];   // B into A
      theArray[dex2] = temp;             // temp into B
      }  // end swap(
//--------------------------------------------------------------
```

87

```
//--------------------------------------------------------------
    public int partitionIt(int left, int right, long pivot)
       {
       int leftPtr = left;             // right of first elem
       int rightPtr = right - 1;       // left of pivot
       while(true)
          {
          while( theArray[++leftPtr] < pivot )  // find bigger
             ;                                  // (nop)
          while( theArray[--rightPtr] > pivot ) // find smaller
             ;                                  // (nop)
          if(leftPtr >= rightPtr)      // if pointers cross,
             break;                    //    partition done
          else                         // not crossed, so
             swap(leftPtr, rightPtr);  // swap elements
          }  // end while(true)
       swap(leftPtr, right-1);         // restore pivot
       return leftPtr;                 // return pivot location
       }  // end partitionIt()
//--------------------------------------------------------------
```

88

```
//--------------------------------------------------------------
                                    // insertion sort
   public void insertionSort(int left, int right)
      {
      int in, out;
                                    //  sorted on left of out
      for(out=left+1; out<=right; out++)
         {
         long temp = theArray[out];    // remove marked item
         in = out;                     // start shifts at out
                                       // until one is smaller,
         while(in>left && theArray[in-1] >= temp)
            {
            theArray[in] = theArray[in-1]; // shift item to right
            --in;                      // go left one position
            }
         theArray[in] = temp;          // insert marked item
         }  // end for
      }  // end insertionSort()
//--------------------------------------------------------------
   }  // end class ArrayIns
////////////////////////////////////////////////////////////////
```

89

```
////////////////////////////////////////////////////////////////
class QuickSort3App
   {
   public static void main(String[] args)
      {
      int maxSize = 16;             // array size
      ArrayIns arr;                 // reference to array
      arr = new ArrayIns(maxSize);  // create the array

      for(int j=0; j<maxSize; j++)  // fill array with
         {                          // random numbers
         long n = (int)(java.lang.Math.random()*99);
         arr.insert(n);
         }
      arr.display();                // display items
      arr.quickSort();              // quicksort them
      arr.display();                // display them again
      }  // end main()
   }  // end class QuickSort3App
////////////////////////////////////////////////////////////////
```

- Using the insertion sort for small subarrays turns out to be the fastest approach on our particular installation, but it is not much faster than sorting subarrays of 3 or fewer cells by hand, as in `quickSort2.java`.
- The numbers of comparisons and copies are reduced substantially in the quicksort phase, but are increased by an almost equal amount in the insertion sort, so the time savings are not dramatic.
- However, it's probably a worthwhile approach if you are trying to squeeze the last ounce of performance out of quicksort.

90

Insertion Sort Following Quicksort

- Another option:
  - Completely quicksort the array without bothering to sort partitions smaller than the cutoff.
  - When quicksort is finished, the array will be almost sorted.
  - Apply the insertion sort to the entire array.
    - The insertion sort is supposed to operate efficiently on almost-sorted arrays
- This approach is recommended by some experts, but on our installation it runs very slowly.
  - The insertion sort appears to be happier doing a lot of small sorts than one big one.

91

Removing Recursion

- Another embellishment recommended by many writers
  - Removing recursion from the quicksort algorithm.
    - This involves rewriting the algorithm to store deferred subarray bounds (`left` and `right`) on a stack,
    - and using a loop instead of recursion to oversee the partitioning of smaller and smaller subarrays.
  - The idea in doing this is to speed up the program by removing method calls.
  - However,
    - this idea arose with older compilers and computer architectures, which imposed a large time penalty for each method call.
    - It's not clear that removing recursion is much of an improvement for modern systems, which handle method calls more efficiently.

92

Efficiency of Quicksort: O(N*logN)

- Generally true of the divide-and-conquer algorithms
  - in which a recursive method divides a range of items into two groups and then calls itself to handle each group.
  - In this situation the logarithm actually has a base of 2: the running time is proportional to $N*\log_2 N$.
- Can get an idea of the validity of this $N*\log_2 N$ running time for quicksort by running one of the quickSort Workshop applets with 100 random bars and examining the resulting dotted horizontal lines.

93

Efficiency of Quicksort: O(N*logN)

- Each dotted line represents an array or subarray being partitioned:
  - the pointers `leftScan` and `rightScan` moving toward each other, comparing each data items and swapping when appropriate.
- We saw in the section on partitioning that a single partition runs in O(N) time.
  - This tells us that the total length of all the lines is proportional to the running time of quicksort.
  - But how long are all the lines?
  - It would be tedious to measure them with a ruler on the screen, but we can visualize them a different way.

94

Efficiency of Quicksort: O(N*logN)

- There is always one line that runs the entire width of the graph, spanning N bars.
  - This results from the first partition.
- There will also be two lines (one below and one above the first line) that have an average length of N/2 bars;
  - together they are again N bars long.
- Then there will be four lines with an average length of N/4 that again total N bars,
- then 8 lines, 16, and so on.
- Figure 7.15 shows how this looks for 1, 2, 4, and 8 lines.

95

Efficiency of Quicksort: O(N*logN)

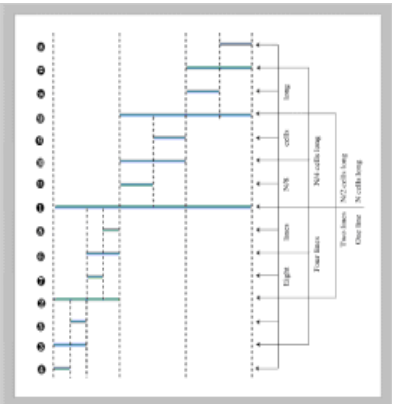


Figure 7.15: Lines correspond to partitions

- In this figure solid horizontal lines represent the dotted horizontal lines in the quicksort applets, and captions like *N/4 cells long* indicate average, not actual, line lengths.
- The circled numbers on the left show the order in which the lines are created.
- Each series of lines (the eight N/8 lines, for example) corresponds to a level of recursion.
- The initial call to `recQuickSort()` is the first level and makes the first line;
- the two calls from within the first call—the second level of recursion—make the next two lines; and so on.
- If we assume we start with 100 cells, the results are shown in Table 7.4.

96