**CHAPTER 7**

**Sorting**

**7.1**

**7.2** *O*(*N*), because the *while* loop terminates immediately. Of course, accidentally changing the test to include equalities raises the running time to quadratic for this type of input.

**7.3** The inversion that existed between *a*[*i*] and *a*[*i* + *k*] is removed. This shows at least one inversion is removed. For each of the *k* − 1 elements *a*[*i* + 1], *a*[*i* + 2], . . . , *a*[*i* + *k* − 1], at most two inversions can be removed by the exchange. This gives a maximum of 2(*k* − 1) + 1 = 2*k* − 1.

**7.4**

**7.5 (a)** Θ(*N*2). The 2-sort removes at most only three inversions at a time; hence the algorithm is Ω(*N*2). The 2-sort is two insertion sorts of size *N*/2, so the cost of that pass is *O*(*N*2). The 1-sort is also *O*(*N*2), so the total is *O*(*N*2).

**7.6** Part (a) is an extension of the theorem proved in the text. Part (b) is fairly complicated; see reference [12].

**7.7** See reference [12].

**7.8** Use the input specified in the hint. If the number of inversions is shown to be Ω(*N*2), then the bound follows, since no increments are removed until an *ht*/2 sort. If we consider the pattern formed *hk* through *h*2*k*−1, where *k* = *t*/2 + 1, we find that it has length *N* = *hk*(*hk* + 1) − 1, and the number of inversions is roughly *hk*4/24, which is Ω(*N*2).

**7.9 (a)** *O*(*N* log *N*). No exchanges, but each pass takes *O*(*N*).

**(b)** *O*(*N* log *N*). It is easy to show that after an *hk* sort, no element is farther than *hk* from its rightful position. Thus if the increments satisfy *hk*+ 1≤*chk* for a constant *c*, which implies *O*(log *N*) increments, then the bound is *O*(*N* log *N*).

**7.10 (a)** No, because it is still possible for consecutive increments to share a common factor. An example is the sequence 1, 3, 9, 21, 45, *ht*+ 1 = 2*ht* + 3.

**(b)** Yes, because consecutive increments are relatively prime. The running time becomes *O*(*N*3/2).

**7.11** The input is read in as

142, 543, 123, 65, 453, 879, 572, 434, 111, 242, 811, 102

The result of the heapify is

879, 811, 572, 434, 543, 123, 142, 65, 111, 242, 453, 102

879 is removed from the heap and placed at the end. We’ll place it in italics to signal that it is not part of the heap. 102 is placed in the hole and bubbled down, obtaining

811, 543, 572, 434, 453, 123, 142, 65, 111, 242, 102, *879*

Continuing the process, we obtain

572, 543, 142, 434, 453, 123, 102, 65, 111, 242, *811*, *879*

543, 453, 142, 434, 242, 123, 102, 65, 111, *572*, *811*, *879*

453, 434, 142, 111, 242, 123, 102, 65, *543*, *572*, *811*, *879*

434, 242, 142, 111, 65, 123, 102, *453*, *543*, *572*, *811*, *879*

242, 111, 142, 102, 65, 123, *434*, *453*, *543*, *572*, *811*, *879*

142, 111, 123, 102, 65, *242*, *434*, *453*, *543*, *572*, *811*, *879*

123, 111, 65, 102, *142*, *242*, *434*, *453*, *543*, *572*, *811*, *879*

111, 102, 65, *123*, *142*, *242*, *434*, *453*, *543*, *572*, *811*, *879*

102, 65, *111*, *123*, *142*, *242*, *434*, *453*, *543*, *572*, *811*, *879*

65, *102*, *111*, *123*, *142*, *242*, *434*, *453*, *543*, *572*, *811*, *879*

7.12 *N* log *N*

7.13 Heapsort uses at least (roughly) *N* log *N* comparisons on any input, so there are no particularly good inputs. This bound is tight; see the paper by Schaeffer and Sedgewick [18]. This result applies for almost all variations of heapsort, which have different rearrangement strategies. See Y. Ding and M. A. Weiss, “Best Case Lower Bounds for Heapsort,” *Computing* 49 (1992).

7.14 If the root is stored in position *low*, then the left child of node *i* is stored at position 2*i* + 1 − *low*. This requires a small change to the heapsort code.

7.15 First the sequence {3, 1, 4, 1} is sorted. To do this, the sequence {3, 1} is sorted. This involves sorting {3} and {1}, which are base cases, and merging the result to obtain {1, 3}. The sequence {4, 1} is likewise sorted into {1, 4}. Then these two sequences are merged to obtain {1, 1, 3, 4}. The second half is sorted similarly, eventually obtaining {2, 5, 6, 9}. The merged result is then easily computed as {1, 1, 2, 3, 4, 5, 6, 9}.

7.16 Mergesort can be implemented nonrecursively by first merging pairs of adjacent elements, then pairs of two elements, then pairs of four elements, and so on.

**void mergesort( vector<Comparable> & a )**

**{**

**int n = a.size( );**

**vector<Comparable> tmpArray( n );**

**for( int subListSize = 1; subListSize < n; subListSize \*= 2 )**

**{**

**int part1Start = 0;**

**while( part1Start + subListSize < n - 1 )**

**{**

**int part2Start = part1Start + subListSize;**

**int part2End = min( n, part2Start + subListSize - 1 );**

**merge( a, tmpArray, part1Start, part2Start, part2End );**

**part1Start = part2End + 1;**

**}**

**}**

**}**

**7.17** The merging step always takes Θ(*N*) time, so the sorting process takes Θ(*N* log *N*) time on all inputs.

**7.18** See reference [12] for the exact derivation of the worst case of mergesort.

**7.19** The original input is

3, 1, 4, 1, 5, 9, 2, 6, 5, 3, 5

After sorting the first, middle, and last elements, we have

3, 1, 4, 1, 5, 5, 2, 6, 5, 3, 9

Thus the pivot is 5. Hiding it gives

3, 1, 4, 1, 5, 3, 2, 6, 5, 5, 9

The first swap is between two fives. The next swap has *i* and *j* crossing. Thus the pivot is swapped back with *i*:

3, 1, 4, 1, 5, 3, 2, 5, 5, 6, 9

We now recursively quicksort the first eight elements:

3, 1, 4, 1, 5, 3, 2, 5

Sorting the three appropriate elements gives

1, 1, 4, 3, 5, 3, 2, 5

Thus the pivot is 3, which gets hidden:

1, 1, 4, 2, 5, 3, 3, 5

The first swap is between 4 and 3:

1, 1, 3, 2, 5, 4, 3, 5

The next swap crosses pointers, so is undone; *i* points at 5, and so the pivot is swapped:

1, 1, 3, 2, 3, 4, 5, 5

A recursive call is now made to sort the first four elements. The pivot is 1, and the partition does not make any changes. The recursive calls are made, but the subfiles are below the cutoff, so nothing is done. Likewise, the last three elements constitute a base case, so nothing is done. We return to the original call, which now calls quicksort recursively on the right-hand side, but again, there are only three elements, so nothing is done. The result is

1, 1, 3, 2, 3, 4, 5, 5, 5, 6, 9

which is cleaned up by insertion sort.

**7.20 (a)** *O*(*N* log *N*) because the pivot will partition perfectly.

**(b)** Again, *O*(*N* log *N*) because the pivot will partition perfectly.

**(c)** *O*(*N* log *N*); the performance is slightly better than the analysis suggests because of the median-of-three partition and cutoff.

**7.21 (a)** If the first element is chosen as the pivot, the running time degenerates to quadratic in the first two cases. It is still *O*(*N* log *N*) for random input.

**(b)** The same results apply for this pivot choice.

**(c)** If a random element is chosen, then the running time is *O*(*N* log *N*) expected for all inputs, although there is an *O*(*N*2) worst case if very bad random numbers come up. There is, however, an essentially negligible chance of this occurring. Chapter 10 discusses the randomized philosophy.

**(d)** This is a dangerous road to go; it depends on the distribution of the keys. For many distributions, such as uniform, the performance is *O*(*N* log *N*) on average. For a skewed distribution, such as with the input {1, 2, 4, 8, 16, 32, 64, . . . ,} the pivot will be consistently terrible, giving quadratic running time, independent of the ordering of the input.

**7.22 (a)** *O*(*N* log *N*) because the pivot will partition perfectly.

**(b)** Sentinels need to be used to guarantee that *i* and *j* don’t run past the end. The running time will be Θ(*N*2) since the partitioning step will put all but the pivot in *S*1, because *i* won’t stop until it hits the sentinel.

**(c)** Again a sentinel needs to be used to stop *j*. This is also Θ(*N*2) because the partitioning is unbalanced.

**7.23** Yes, but it doesn’t reduce the average running time for random input. Using median-of-three partitioning reduces the average running time because it makes the partition more balanced on average.

**7.24** The strategy used here is to force the worst possible pivot at each stage. This doesn’t necessarily give the maximum amount of work (since there are few exchanges, just lots of comparisons), but it does give Ω(*N*2) comparisons. By working backward, we can arrive at the following permutation:

20, 3, 5, 7, 9, 11, 13, 15, 17, 19, 4, 10, 2, 12, 6, 14, 1, 16, 8, 18

A method to extend this to larger numbers when *N* is even is as follows: The first element is *N*, the middle is *N* − 1, and the last is *N* − 2. Odd numbers (except 1) are written in decreasing order starting to the left of center. Even numbers are written in decreasing order by starting at the rightmost spot, always skipping one available empty slot, and wrapping around when the center is reached. This method takes *O*(*N* log *N*) time to generate the permutation, but is suitable for a hand calculation. By inverting the actions of quicksort, it is possible to generate the permutation in linear time.

**7.26** Each recursive call is on a subarray that is less than half the size of the original, giving a logarithmic number of recursive calls.

**7.27** See reference [14].

**7.28** See reference [1].

**7.29**

template<typename Comparable>

void selectionSort(vector<Comparable> & toSort)

{

int minIndex;

for (int i = 0; i < toSort.size(); i++)

{

minIndex = i;

for (int j = i+1; j < toSort.size(); j++)

{

if (toSort[j] < toSort[minIndex])

swap(toSort[j], toSort[i]);

}

}

}

**7.30** This recurrence results from the analysis of the quick selection algorithm. *T*(*N*) = *O*(*N*).

**7.31** Insertion sort and mergesort are stable if coded correctly. Any of the sorts can be made stable by the addition of a second key, which indicates the original position.

**7.32 (d)** *f* (*N*) can be *O*(*N*/log *N*). Sort the *f* (*N*) elements using mergesort in *O*(*f* (*N*) log *f* (*N*)) time. This is *O*(*N*) if *f* (*N*) is chosen using the criterion given. Then merge this sorted list with the already sorted list of *N* numbers in *O*(*N* + *f* (*N*)) = *O*(*N*) time.

**7.33** A decision tree would have *N* leaves, so ⎡log *N*⎤ comparisons are required.

**7.34** log *N*! ≈ *N* log *N* − *N* log *e*.

**7.35 (a) **

**(b)** The information-theoretic lower bound is  Applying Stirling’s formula, we can estimate the bound as 2*N*−*half* log *N*. A better lower bound is known for this case: 2*N* − 1 comparisons are necessary. Merging two lists of different sizes *M* and *N* likewise requires at least  comparisons.

**7.37** Algorithms *A* and *B* require at least three comparisons based on the information-theoretic lower bound. Algorithm *C* requires at least five comparisons, based on Exercise 7.35(b). So the algorithm requires 11 comparisons; but six elements can be sorted in 10 comparisons (for instance, use the results in Exercise 7.42, and use binary search to find the correct place to insert the sixth element).

**7.38** Here is a c++ implementation

/\*\* Exercise738.c++

\* Given N points in the plane, we are to find 4 of them that

\* are collinear. The algorithm needs to be O(n^2 log\_2 n).

\* The algorithm proceeds as follows.

\* 1. For each pair of points, compute the slope/y-intercept.

\* We now have N^2 objects to consider.

\* 2. Sort the pairs of points. Using the STL,

this can be done in O(N^2 log (N^2)) time which

\* equals O(N^2 log N) time (since the coefficient 2 comes out).

\* 3. Once the list of N^2 pairs of points has been sorted, we can

\* find a collinear set of four by a single traversal of this list.

\* This means it can be done in O(N^2) time. We just need to look

\* at three consecutive pairs to find the 4 desired points.

\*/

#include <vector>

#include <algorithm>

#include <iostream>

using namespace std;

const double EPSILON = 0.000001;

const double INF = 9999999.;

struct Point

{

double x;

double y;

bool operator < (const Point & point)

{

if (x != point.x)

return x < point.x;

else

return y < point.y;

}

bool operator == (const Point & point)

{

return (x == point.x && y == point.y);

}

};

class Line

{

private:

Point p1; // point 1

Point p2;// point 2

double slope;

double yInter;

public:

Line(Point one, Point two):p1(one), p2(two)

{

slope = computeSlope(p1, p2);

if (slope != INF)

yInter = p1.y - slope \* p1.x ;

else

yInter = p1.x; // for vertical lines use x-inter

}

double computeSlope(Point p1, Point p2)

{

if (fabs(p1.x-p2.x) < EPSILON)

return INF;

else

return (p2.y - p1.y)/(p2.x - p1.x);

}

bool operator< (const Line & line)

{

if ( fabs(slope - line.slope) > EPSILON)

return slope < line.slope;

else

return yInter < line.yInter;

}

double getSlope() {return slope;}

double getYInter(){ return yInter;}

Point getP1() {return p1;}

Point getP2() {return p2;}

};

void printPoints(vector<Point> collinear)

{

vector<Point>::iterator it;

int count = 0;

it = unique(collinear.begin(), collinear.end());

for (vector<Point>::iterator itr = collinear.begin(); itr != it; itr++)

count++;

if (count > 3)

for (vector<Point>::iterator itr = collinear.begin(); itr != it; itr++)

cout<<"("<<itr->x<<","<<itr->y<<")\n";

}

int main()

{

vector<Point> points;

vector<Line \*> lines;

vector<Point> collinear;

int numPoints;

Line \* line;

cout<<"how many points: ";

cin>>numPoints;

points.resize(numPoints);

for (int i = 0; i < numPoints; i++)

{

cout<<"enter the x and y coordinates for the "<<i<<"th point :";

cin>>points[i].x>>points[i].y;

}

for (int i = 0; i < points.size(); i++)

for (int j = i+1 ; j < points.size(); j++)

{

line = new Line(points[i], points[j]);

lines.push\_back(line);

}

sort(lines.begin(), lines.end());

for (int i = 0; i < lines.size() - 1; i++)

{

if (lines[i]->getSlope() == lines[i+1]->getSlope() &&

lines[i]->getYInter() == lines[i+1]->getYInter())

{

collinear.push\_back(lines[i]->getP1());

collinear.push\_back(lines[i]->getP2());

collinear.push\_back(lines[i+1]->getP1());

collinear.push\_back(lines[i+1]->getP2());

}

else

{

printPoints(collinear);

collinear.clear();

}

}

}

**7.39** First, we will find the smallest element, and then the second smallest. Create a binary tree based on the elements of the array. First, the N values of the array will be the leaves of the tree. The higher levels of the tree will be formed as follows.

For each pair of elements, i.e. the 1st and 2nd, the 3rd and 4th, the 5th and 6th, etc., determine which one is smaller, and make this value the parent of the two nodes. When this is finished, you now have a new level of the tree with N/2 nodes.

Create a new higher level L-1 of the binary tree by taking each pair at level L of the tree, seeing which number is smaller, and creating a parent node with this value. This procedure of creating new levels of the tree completes once you have the root of the tree, which is the smallest value in the array.

As an illustration, consider the array: 5, 4, 7, 3, 2, 6, 1, 8.

Initially we have just eight leaves. Then, we create the next higher level of the tree to obtain:

4 3 2 1

5 4 7 3 2 6 1 8

And we create the next level of the tree:

3 1

4 3 2 1

5 4 7 3 2 6 1 8

Finally, add the root:

1

3 1

4 3 2 1

5 4 7 3 2 6 1 8

In order to find the smallest value, we needed to perform n/2 + n/4 + n/8 + ... + 1 comparisons, which in this case worked out to 4+2+1 comparisons. The number of comparisons needed to find the smallest element is N-1.

To determine the 2nd smallest number, this must be one of the values that was directly compared with the 1st smallest value at some point in the tree. For example, in the tree above, the 2nd smallest number must be 3, 2 or 8, because these numbers was compared with 1.

The number of values to consider is based on the height of the tree (log2(n)), and the number of comparisons needed to find the smallest of these is 1 fewer, which is log2(n) – 1.

**7.40** Here is a Java implementation:

import java.util.Random;

public class Exercise740

{

public static int numComparisons;

public static final int N = 1048576; // Let N be some power of 2.

// The purpose of main() is to test the max/min algorithms on

// arrays of size N and 3N, where N is a power of 2.

public static void main(String [] args)

{

Random gen = new Random();

int [] a = new int [3\*N];

for (int i = 0; i < 3\*N; ++i)

{

a[i] = gen.nextInt(3\*N);

}

numComparisons = 0;

Pair p = findMaxMin(a, 0, N-1, false);

System.out.printf("Among first %d elements, max = %d, min = %d, # comp = %d\n",

N, p.x, p.y, numComparisons);

numComparisons = 0;

p = findMaxMin(a, 0, 3\*N-1, false);

System.out.printf("Among first %d elements, max = %d, min = %d, # comp = %d\n",

3\*N, p.x, p.y, numComparisons);

// Now, let's try the 2nd algorithm when the size is even but not mult of 4.

numComparisons = 0;

p = findMaxMin(a, 0, N-1, true);

System.out.printf("2nd algorithm: Among first %d elements, max = %d, min = %d, # comp = %d\n",

N, p.x, p.y, numComparisons);

numComparisons = 0;

p = findMaxMin(a, 0, 3\*N-1, true);

System.out.printf("2nd algorithm: Among first %d elements, max = %d, min = %d, # comp = %d\n",

3\*N, p.x, p.y, numComparisons);

}

// Find the max and min of the array segment a[first..last].

// The parameter phaseTwo tells us if we are considering part (c)

// of the exercise, where we must split up the array recursively into

// sizes /2-1 and n/2+1 if n is even but not a multiple of 4.

public static Pair findMaxMin(int [] a, int first, int last, boolean phaseTwo)

{

// If there is logically 1 element

if (first == last)

return new Pair(a[first], a[first]);

// If there are 2 elements

else if (first + 1 == last)

{

++numComparisons;

if (a[first] < a[last])

return new Pair(a[last], a[first]);

else

return new Pair(a[first], a[last]);

}

// Part (c) of the exercise tests a slight imbalance of the halves.

// Recursively call ourselves with sizes n/2-1 and n/2+1.

int midpoint;

if (phaseTwo && (last - first + 1) % 2 == 0 && (last - first + 1) % 4 != 0)

midpoint = first + (last - first + 1)/2 - 1;

// Otherwise split into two halves

else

midpoint = (first + last) / 2;

Pair p1 = findMaxMin(a, first, midpoint, phaseTwo);

Pair p2 = findMaxMin(a, midpoint + 1, last, phaseTwo);

// Compare the two maxes.

numComparisons += 2;

int max = (p1.x > p2.x) ? p1.x : p2.x;

int min = (p1.y < p2.y) ? p1.y : p2.y;

return new Pair(max, min);

}

// The intent of this class is to store the max and min inside

// a single ordered pair object that can be returned from findMaxMin().

// The max comes first.

static class Pair

{

public int x;

public int y;

public Pair(int x, int y)

{

this.x = x;

this.y = y;

}

}

}

// When N = 1,048,576, the number of comparisons is as follows.

// For finding the max/min of N elements: 1,572,862 (about 1.5N)

// For finding the max/min of 3N elements: 5,242,878 (about 5N)

// The results are the same with the second algorithm.

// Note that 3N elements required more than 3 times as many comparisons as N.

// Thus, with a divide and conquer, there is some advantage to starting

// with a size that is a power of two.

**7.42** Note that any two fractions that are not equal differ by at least 1/*N*2. Thus rewrite the fractions as *c*/(*N* + 1)2 + *d*, ignore the *d*, and sort on the basis of *c* using a two-pass radix sort.

**7.43** Look at the middle elements of *A* and *B*. Let *m*1 and *m*2 be these two middle elements; assume *m*1 is the larger (the mirror image is similar). Recursively use the elements in *A* that are larger than *m*1 and the elements in *B* that are smaller than *m*2, making sure that an equal number of elements from *A* and *B* are not included in the recursive problem (include *m*1 or *m*2 if needed to balance). Recursively solve the problem on the subarray consisting of elements larger than the larger of the two middles.

**7.44** We add a dummy *N* + 1*th* element, which we’ll call *maybe*. *maybe* satisfies *false* < *maybe* < *true*. Partition the array around *maybe*, using the standard quicksort techniques. Then swap *maybe* and the *N* + 1*th* element. The first *N* elements are then correctly arranged.

**7.45** We add a dummy *N* + 1*th* element, which we’ll call *probablyFalse*. *probablyFalse* satisfies *false* < *probablyFalse* < *maybe*. Partition the array around *probablyFalse* as in the previous exercise. Suppose that after the partition, *probablyFalse* winds up in position *i*. Then place the element that is in the *N* + 1*th* slot in position *i*, place *probablyTrue* (defined the obvious way) in position *N* + 1, and partition the subarray from position *i* onward. Finally, swap *probablyTrue* with the element in the *N* + 1*th* location. The first *N* elements are now correctly arranged. These two problems can be done without the assumption of an extra array slot; assuming so simplifies the presentation.

**7.46 (a)** ⎡log4!⎤ = 5.

**(b)** Compare and exchange (if necessary) *a*1 and *a*2 so that *a*1 ≥ *a*2, and repeat with *a*3 and *a*4. Compare and exchange *a*1 and *a*3. Compare and exchange *a*2 and *a*4. Finally, compare and exchange *a*2 and *a*3.

**7.47 (a)** ⎡log5!⎤ = 7.

**(b)** Compare and exchange (if necessary) *a*1 and *a*2 so that *a*1 ≥ *a*2, and repeat with *a*3 and *a*4 so that *a*3 ≥ *a*4. Compare and exchange (if necessary) the two winners, *a*1 and *a*3. Assume without loss of generality that we now have *a*1 ≥ *a*3 ≥ *a*4, and *a*1 ≥ *a*2. (The other case is obviously identical.) Insert *a*5 by binary search in the appropriate place among *a*1 , *a*3, *a*4. This can be done in two comparisons. Finally, insert *a*2 among *a*3 , *a*4 , *a*5. If it is the largest among those three, then it goes directly after *a*1 since it is already known to be larger than *a*1. This takes two more comparisons by a binary search. The total is thus seven comparisons.

**7.51 (a)** For the given input, the pivot is 2. It is swapped with the last element. *i* will point at the second element, and *j* will be stopped at the first element. Since the pointers have crossed, the pivot is swapped with the element in position 2. The input is now 1, 2, 4, 5, 6, . . . , *N* − 1, *N*, 3. The recursive call on the right subarray is thus on an increasing sequence of numbers, except for the last number, which is the smallest. This is exactly the same form as the original. Thus each recursive call will have only two fewer elements than the previous. The running time will be quadratic.

**(b)** Although the first pivot generates equal partitions, both the left and right halves will have the same form as part (a). Thus the running time will be quadratic because after the first partition, the algorithm will grind slowly. This is one of the many interesting tidbits in reference [22].

**7.52** We show that in a binary tree with *L* leaves, the average depth of a leaf is at least log *L*. We can prove this by induction. Clearly, the claim is true if *L* = 1. Suppose it is true for trees with up to *L* − 1 leaves. Consider a tree of *L* leaves with minimum average leaf depth. Clearly, the root of such a tree must have non-null left and right subtrees. Suppose that the left subtree has *LL* leaves, and the right subtree has *LR* leaves. By the inductive hypothesis, the total depth of the leaves (which is their average times their number) in the left subtree is *LL* (1 + log *LL*), and the total depth of the right subtree’s leaves is *LR* (1 + log *LR*) (because the leaves in the subtrees are one deeper with respect to the root of the tree than with respect to the root of their subtree). Thus the total depth of all the leaves is *L* + *LL* log *LL* + *LR* log *LR*. Since *f* (*x*) = *x* log *x* is convex for *x* ≥ 1, we know that *f* (*x*) + *f* (*y*) ≥ 2*f* ((*x* + *y*)/2). Thus the total depth of all the leaves is at least *L* + 2(*L*/2) log (*L*/2) ≥ *L* + *L*(log *L* − 1) ≥ *L* log *L*. Thus the average leaf depth is at least log *L*.

**7.53 (a)** A brute-force search gives a quadratic solution.

**(b)** Sort the items; then run *i* and *j* from opposite ends of the array toward each other, incrementing *i* if *a*[*i*] + *a*[*j*] < *K*, and decrementing *j* if *a*[*i*] + *a*[*j*] > *K*. The sort is *O*(*N* log *N*); the scan is *O*(*N*).

**7.55 (b)** After sorting the items do a scan using an outer loop that considers all positions *p*, and an inner loop that searches for two additional items that sum to *K* − *a*[*p*]. The two loops give a quadratic algorithm.