Chapter 7: Sorting

7.1

Original	3	1	4	1	5	9	2	6	5
after P=2	1	3	4	1	5	9	2	6	5
after P=3	1	3	4	1	5	9	2	6	5
after P=4	1	1	3	4	5	9	2	6	5
after P=5	1	1	3	4	5	9	2	6	5
after P=6	1	1	3	4	5	9	2	6	5
after P=7	1	1	2	3	4	5	9	6	5
after P=8	1	1	2	3	4	5	6	9	5
after P=9	1	1	2	3	4	5	5	6	9

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 $2(k-1)+1 = 2k-1$.

7.4

Original	9	8	7	6	5	4	3	2	1
after 7-sort	2	1	7	6	5	4	3	9	8
after 3-sort	2	1	4	3	5	7	6	9	8
after 1-sort	1	2	3	4	5	6	7	8	9

7.5 (a) $\Theta(N^2)$. The 2-sort removes at most only three inversions at a time; hence the algorithm is $\Omega(N^2)$. 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 [11].

7.7 See reference [11].

7.8 Use the input specified in the hint. If the number of inversions is shown to be $\Omega(N^2)$, then the bound follows, since no increments are removed until an $h_{t/2}$ sort. If we consider the pattern formed h_k through h_{2k-1}, where $k = t/2+1$, we find that it has length $N = h_k(h_k+1)$–1, and the number of inversions is roughly ${h_k}^4/24$, which is $\Omega(N^2)$.

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 h_k sort, no element is farther than h_k from its rightful position. Thus if the increments satisfy $h_{k+1} \le c h_k$ 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, $h_{t+1} = 2h_t + 3$.

(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 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 [16]. 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).

7.13 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}.

7.14 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. This is implemented in Fig. 7.1.

7.15 The merging step always takes $\Theta(N)$ time, so the sorting process takes $\Theta(N \log N)$ time on all inputs.

7.16 See reference [11] for the exact derivation of the worst case of mergesort.

7.17 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

```
void
Mergesort( ElementType A[ ], int N )
{
        ElementType *TmpArray;
        int SubListSize, Part1Start, Part2Start, Part2End;

        TmpArray = malloc( sizeof( ElementType ) * N );
        for( SubListSize = 1; SubListSize < N; SubListSize *= 2 )
        {
                Part1Start = 0;
                while( Part1Start + SubListSize < N - 1 )
                {
                        Part2Start = Part1Start + SubListSize;
                        Part2End = min( N, Part2Start + SubListSize - 1 );
                        Merge( A, TmpArray, Part1Start, Part2Start, Part2End );
                        Part1Start = Part2End + 1;
                }
        }
}
```

Fig. 7.1.

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.

7.18 (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.

7.19 (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.

7.20 (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 $\Theta(N^2)$ since, because i won't stop until it hits the sentinel, the partitioning step will put all but the pivot in S_1.

(c) Again a sentinel needs to be used to stop j. This is also $\Theta(N^2)$ because the partitioning is unbalanced.

7.21 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.

7.22 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 $\Omega(N^2)$ 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.

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

7.25 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.

7.26 (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.

7.27 A decision tree would have N leaves, so $\lceil \log N \rceil$ comparisons are required.

7.28 $\log N! \approx N \log N - N \log e$.

7.29 (a) $\binom{2N}{N}$.

(b) The information-theoretic lower bound is $\log \binom{2N}{N}$. Applying Stirling's formula, we can estimate the bound as $2N - \frac{1}{2}\log N$. A better lower bound is known for this case: $2N-1$ comparisons are necessary. Merging two lists of different sizes M and N likewise requires at least $\log \binom{M+N}{N}$ comparisons.

7.30 It takes $O(1)$ to insert each element into a bucket, for a total of $O(N)$. It takes $O(1)$ to extract each element from a bucket, for $O(M)$. We waste at most $O(1)$ examining each empty bucket, for a total of $O(M)$. Adding these estimates gives $O(M+N)$.

7.31 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.

7.32 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.

7.33 (a) $\lceil \log 4! \rceil = 5$.

(b) Compare and exchange (if necessary) a_1 and a_2 so that $a_1 \geq a_2$, 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.

7.34 (a) $\lceil \log 5! \rceil = 7$.

(b) Compare and exchange (if necessary) a_1 and a_2 so that $a_1 \geq a_2$, and repeat with a_3 and a_4 so that $a_3 \geq a_4$. Compare and exchange (if necessary) the two winners, a_1 and a_3. Assume without loss of generality that we now have $a_1 \geq a_3 \geq a_4$, and $a_1 \geq a_2$. (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.

7.38 (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 [20].

7.39 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 L_L leaves, and the right subtree has L_R leaves. By the inductive hypothesis, the total depth of the leaves (which is their average times their number) in the left subtree is $L_L(1 + \log L_L)$, and the total depth of the right subtree's leaves is $L_R(1 + \log L_R)$ (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 $L + L_L \log L_L + L_R \log L_R$. Since $f(x) = x \log x$ is convex for $x \geq 1$, we know that $f(x) + f(y) \geq 2f((x+y)/2)$. Thus, the total depth of all the leaves is at least $L + 2(L/2)\log(L/2) \geq L + L(\log L - 1) \geq L \log L$. Thus the average leaf depth is at least $\log L$.