p.257 7.4

Show the result of running Shellsort on the input 9, 8, 7, 6, 5, 4, 3, 2, 1 using the increments {1, 3, 7}.

Answer

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

p.258 7.14

How would you implement mergesort without using recursion?

```
void   merge( element   list[ ],   element   sorted[ ],   int   i,   int   m,   int   n )
{         /* merges the sorted lists ( list[ i ],…, list[ m ] ) and ( list[ m+1 ], …, list[ n ] ),
             into a single sorted list ( sorted[ i ],…, sorted[ n ] ) */
          int   j, k, t;
          j = m+1;                /* index for the second sublist */
          k = i;                  /* index for the sorted list */
          while ( i <= m && j <= n) { /* while there are elements left in both lists */
              if ( list[i].key <= list[j].key )
                     sorted[k++] = list[i++];
              else
                     sorted[k++] = list[j++];
          }
          if ( i > m) /* merge whatever left to the rest of the list */
               for ( t = j; t <= n; t++ )
                      sorted[ k+t-j ] = list[ t ];
          else
               for ( t = i; t <= m; t++ )
                      sorted[ k+t-i ] = list[ t ];
}

void   merge_pass( element   list[ ],   element   sorted[ ],   int   n,   int   length )
{         /* perform one pass of the merge sort that merges adjacent pairs
             of subfiles from list into sorted.   n is the number of element in the list.
             length is the length of the subfile */
          int   i, j;
          for ( i = 0; i <= n–2 * length; i += 2 * length ) /* merge adjacent pairs */
              merge( list, sorted, i, i + length – 1, i + 2 * length –1 );
          if ( i+length < n ) /* merge the last two pieces */
```

```
            merge( list, sorted, i, i+length–1, n–1);
        else   /* there is only one subfile left */
            for ( j = i; j < n; j++ )   /* simple copy the last piece */
                sorted[ j ] = list[ j ];
}

void   merge_sort( element   list[ ],   int   n )
{
        int   length = 1;   /* current length of subfile being merged */
        element   extra[ MAX_SIZE ];   /* the extra space required */
        while( length < n ) {
            merge_pass( list, extra, n, length ); /* merge list into extra */
            length *= 2;
            merge_pass( extra, list, n, length ); /* merge extra back to list */
            length *= 2;
        }
}
```

p.258   7.20

a. For the quicksort implementation in this chapter, what is the running time when all keys are equal?

b. Suppose we change the partitioning strategy so that neither i nor j stops when an element with the same key as the pivot is found. What fixes need to be made in the code to guarantee that quicksort works, and what is the running time, when all keys are equal?

c. Suppose we change the partitioning strategy so that i stops at an element with the same key as the pivot, but j does not stop in a similar case. What fixes need to be made in the code to guarantee that quicksort works, and when all keys are equal, what is the running time of quicksort?

Answer

a. the running time is O(N log N).

b. change the program line 5 and line 6 into:

```
while( A[+ +i]<=Pivot && i<j ) { }
while( A[– –j]>=Pivot && j>i ) { }
```

the running time is $O(N^2)$ when all keys are equal.

c. change the program line 6 into:

```
while(A[– –j]>=Pivot && j>i) { }
```

the running time is $O(N^2)$ when all keys are equal.

p.259   7.32

Suppose you have an array of N elements, containing three distinct keys, "*true*", "*false*", and "*maybe*". Given an O(N) algorithm to rearrange the list so that all "*false*" elements precede "*maybe*" elements, which in turn precede "*true*" elements. You may use only constant extra space.

```
void   Bucketsort ( ElementType   A[ ],   int   N )
{
    int   Counter[ 3 ];   /* constant extra space */
    int   i, j, k;
    for ( i = 0; i < 3; i++ )
        Counter[ i ] = 0;   /* initialize the counters */
    for ( i = 0, i < N; i++ )   { /* for each key, increment the corresponding counter */
        if ( A[ i ] == false )
            Counter[ 0 ] ++;
        else
            if ( A[ i ] == maybe )
                Counter[ 1 ] ++;
            else
                Counter[ 2 ] ++;
    }
    /* reset the keys */
    k = 0;
    for ( i = 0; i < Counter[ 0 ]; i++ )
        A[ i ] = false;
    k += Counter[ 0 ];
    for ( i = 0; i < Counter[ 1 ]; i++ )
        A[ k+i ] = maybe;
    k += Counter[ 1 ];
    for ( i = 0; i < Counter[ 2 ]; i++ )
        A[ k+i ] = true;
}
```