Chapter 9

Sorting

QUN JIN
Waseda University, Japan
jin@waseda.jp

In this chapter, we discuss sorting algorithms. We begin by introducing three basic sorting methods: Selection Sort, Insertion Sort and Bubble Sort. These methods are easy to understand and easy to program, but they are not very efficient. We then discuss three more efficient sorting methods: Quick Sort, Heap Sort and Merge Sort, and briefly describe two more improved sorting methods: Shell Sort and Radix Sort. Finally, we give the performance comparison of all these sorting methods.

9.1. Basic Sorting

First, let us consider what sorting is. Sorting is an operation that arranges the data in some appropriate order according to some key. We have many examples in our daily life. An English dictionary is a good example, in which all words are arranged in alphabetic order. A phone directory is another one, in which phone numbers are listed in some order.

There are two kinds of sorting algorithms: *Internal Sort* and *External Sort.* Algorithms for sorting data collections that fit into main memory (for example, an array in C language) are called *internal sorting.* Selection Sort, Insertion Sort, Bubble Sort, Quick Sort and Heap Sort are internal sorting. On the other hand, algorithms for sorting data collections from tape or disk are called *external sorting.* Merge Sort is external sorting.

Now let us see how to sort a collection of data. Before discussing the basic ideas of sorting algorithms, we have to describe some general terminologies.

Data compositions: files of records containing keys.

File: collection of records.

Record: collection of fields.

Field: collection of characters.

Key: uniquely identifies a record in a file, used to control the sort.

The basic sorting operations that arrange the data in a specific order according to a specific key are *comparison* and *exchange*. That is, compare two items of the data and exchange their places in the data if they are not in the proper order by the specific key.

9.1.1. *Selection Sort*

Selection Sort is one of the simplest algorithms. The basic idea is to first select successive elements in the ascending order and place them into their proper position. Then find the smallest element from the array, and exchange it with the first element. Find the second smallest element, and exchange it with the second element; and so on. Repeat these procedures until the data is sorted in the proper order.

The following program is an example of implementation in C language for Selection Sort.

```
void selection_sort(int array[], int n)
{
 int i, j, min, tmp;
 for (i = 0; i < n; i++) {
  min = i;
  for (j = i + 1; j < n; j++) {
   if (array[j] < array[min]) min = j;
  }
  tmp = array[min];
  array[min] = array[i];
  array[i] = tmp;
 }
}
```

For each i from 1 to $n-1$, it compares array$[i]$ with array $[i+1], \ldots,$ array $[n]$, and exchanges array $[i]$ with the smallest one of them.

Quiz: Generate an array of 10 random integers, and write a complete C program using Selection Sort to sort the array into increasing order.

[Hint: Refer to the Web Page or CD-ROM to see the sorting process.]

9.1.2. *Insertion Sort*

Insertion Sort is almost as simple as Selection Sort, but a little more flexible. The basic idea is to sort a collection of records by inserting them into an already sorted file. The procedure can be described as follows. First, take elements one by one, and then insert the element in its proper position among those already taken and sorted in a new collection; repeat these processes until all elements are taken and sorted in the proper order.

The following program is an example implementation of the above sorting procedure.

```
void insertion_sort(int array[], int n)
{
 int i, j, tmp;
 for (i = 1; i < n; i++) {
  tmp = array[i];
  for (j = i; array[j – 1]>tmp; j––) {
   array[j] = array[j – 1];
  }
  array[j] = tmp;
 }
}
```

For each i from 2 to n, the elements array[1], ..., array $[i]$ are sorted by placing array $[i]$ into its proper position among the sorted list of elements in array [1], ..., array $[i-1]$.

Quiz: Generate an array of 10 random integers, and write a complete C program using Insertion Sort to sort the array into increasing order.

[Hint: Refer to the Web Page or CD-ROM to see the sorting process.]

9.1.3. *Bubble Sort*

The third elementary sorting algorithm introduced here is called Bubble Sort. The basic idea of this algorithm is to let greater elements "*bubble*" to the right-hand side of the sorted file (or array). The sorting process is given as follows. First, compare two elements in the neighbor. If they are not in the proper order, exchange them (that is, swap them). Therefore, the greatest element "bubbles" to the most right. Next, compare two elements in the neighbor, except the greatest one or greater ones that have already been sorted on the right. If they are not in order, exchange them (swap), so that a greater element "bubbles" to the right. Repeat the above processes until all elements are sorted in the proper order.

We show an example implementation in C language for Bubble Sort below.

```
void bubble_sort(int array[], int n)
{
 int i, j, tmp;
 for (i = n; i > 0; i--) {
  for (j = 1; j < i; j++) {
   if (array[j - 1]>array[j]) {
    tmp = array[j - 1];
    array[j - 1] = array[j];
    array[j] = tmp;
   }
  }
 }
}
```

During one pass, an element will be continuously moved to the right-hand side if it is larger than its right neighbor until it encounters a larger element. Then the larger element will be compared with its right neighbor, and moved to the right-hand side if it is larger. On completion of the first pass, the largest element reaches the right end of the array. On the second pass, the second largest element is put into the second position to the right end, and so on.

Quiz: Generate an array of 10 random integers, and write a complete C program using Bubble Sort to sort the array into increasing order.

[Hint: Refer to the Web Page or CD-ROM to see the sorting process.]

9.2. Quick Sort

Invented in 1960 by C. A. R. Hoare, Quick Sort is one of the most popular sorting algorithms that have been well investigated. It has been greatly improved since then. Quick Sort is quite easy to be implemented, and has a good average performance ($N \log N$), but not very stable. The performance in the worst case is N^*N.

The basic idea of Quick Sort is to apply the so-called "*divide-and-conquer*" paradigm, which works based on the mechanism to partition the file into two parts, then recursively sort the parts independently. The detailed procedures are as follows: First choose a pivotal (central) element (for example, first, last or middle element). Elements on either side are then moved so that the elements on one side of the pivot are smaller and

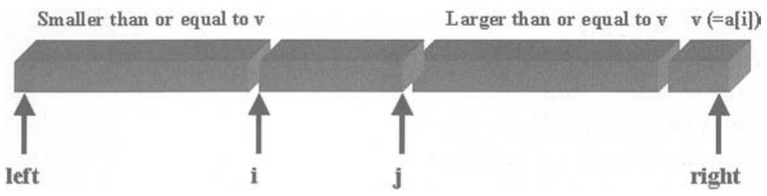

Fig. 9.1. Basic algorithm for Quick Sort.

those on the other side are larger. Apply the above procedures to the two parts until the whole file is sorted in the proper order.

The core part of the basic algorithm is given below.

```
quicksort(int array[], int left, int right) {
        int $i$;
  if (right > left) {
        $i$ = partition(left, right);
        quicksort(array, left, $i - 1$);
        quicksort(array, $i + 1$, right);
         }
         }
```

where partition (left, right) makes the following conditions hold: when the element array[i] is placed in its final place, all elements of array[left], ..., array[$i - 1$] are smaller than or equal to array[i], and all elements of array[$i + 1$], ..., array[right] are larger than or equal to array[i]. That is, array[j] <= array[i] for $j < i$ and array[j] >= array[i] for $j > i$, as shown in Fig. 9.1.

A full implementation for Quick Sort in C language is given as follows.

```
quicksort(int array[], int left, int right) {
  int $v$, $i$, $j$, tmp;
  if (right > left) {
    $v$ = array[right]; $i$ = left −1; $j$ = right;
    for (; ;) {
      while (array[++$i$] < $v$);
      while (array[−−$j$] > $v$);
      if ($i$ >= $j$) break;
      tmp = array[$i$];
      array[$i$] = array[$j$];
      array[$j$] = tmp;
    }
    tmp = array[$i$];
    array[$i$] = array[right];
    array[right] = tmp;
    quicksort(array, left, $i - 1$);
    quicksort(array, $i + 1$, right)
  }
}
```

In this sample program, the variable v holds the current value of the "partitioning elements" array[right] and i and j are the left and right scan pointers, respectively. Quick Sort may be implemented without the recursive approach. The nonrecursive implementation for Quick Sort is given below, by using the stack.

```
quicksort(int array[ ], int $n$) {
  int $i$, left, right;
  left = 1; right = $n$; stackinit( );
  for (;;) {
    while (right > left) {
          $i$ = partition(array, left, right);
      if ($i$-left > right-$i$) {
        push(left);
        push($i$-left);
        left = $i + 1$;
      }
      else {
        push($i + 1$);
        push(right);
        right = $i - 1$;
      }
    if (stackempty( )) break;
    right = pop( );
    left = pop( );
  }
}
```

Quiz: Generate an array of 10 random integers, and write a complete C program using Quick Sort (both Recursive and Nonrecursive versions) to sort the array into increasing order.

[Hint: Refer to the Web Page or CD-ROM to see the sorting process.]

9.3. Heap Sort

Heap Sort is an elegant and efficient sorting algorithm that has been widely used. Good performance (NlogN) is guaranteed in any case. Another advantage of Heap Sort is that no extra memory is needed in the sorting process. Comparing with Quick Sort, inner loop in Heap Sort is a bit longer, generally about twice as slow as Quick Sort on the average.

Before introducing the algorithm of Heap Sort, we firstly give a brief introduction on the so-called *Priority Queue*. Let us recall a deletion operation in a stack and a queue. It deletes the newest element from a stack, whereas it deletes the oldest element from a queue. Priority queue is a generalization of the stack and the queue, which supports the operations of inserting a new element and deleting the largest element. Application of priority queues includes simulation, numerical computation, job scheduling, etc.

Now, let us see what the heap is. Heap is a data structure to support the priority queue operations. Heap can be represented in a complete binary tree, in which every node satisfies the so-called *heap condition*: the key in each node is larger than (or equal to) the keys in its children (if available); the largest key is in the root. Heap represented as an array could be something like this: root in array[1]; children of root in array[2] and array[3]; ...; children of i in array[$2i$] and array[$2i+1$] (parent of i in array[$i/2$]); and so on.

The basic idea of Heap Sort is simply to build a heap containing the elements to be sorted and then to remove them all in order.

The following program is a function that implements the *down heap* operation on a heap.

```
void down_heap(int array[], int n, int x)
{
 int i, v;
 v = array[x];
 while (x <= n/2) {
   i = x + x;
   if (i<n && array[i] <array[i + 1]) i++;
   if (v>=array[i]) break;
   array[x] = array[i];
   x = i;
 }
 array[x] = v;
}
```

Using the above function, we may have an example implementation for the Heap Sort.

```
void heap_sort(int array[], int n)
{
 int x, tmp;
 for (x = n/2; x >= 0; x--) {
   down_heap(array, n, x);
 }
 while (n > 0) {
   tmp = array[0];
   array[0] = array[n];
   array[n] = tmp;
   down_heap(array, --n, 0);
 }
}
```

Please note here we have assumed the *downheap* function has been modified to take the array and heap size as the first two arguments.

Quiz: Generate an array of ten random integers, and write a complete C program using Heap Sort to sort the array into increasing order.

[Hint: Refer to the Web Page or CD-ROM to see the sorting process.]

9.4. Merge Sort

Merging is the operation of combining two sorted files to make one larger sorted file, while selection, as we have seen in Quick Sort, is the operation of partitioning a file into two independent files. We might consider that selection and merging are complementary operations.

Merger Sort is one of the most common algorithms for external sorting. It has a rather good and stable performance ($N \log N$, even in the worst case), the same as Quick Sort and Heap Sort. One of its major drawbacks is that Merge Sort needs linear extra space for the merge, which means it can only sort half the memory.

Both Quick Sort and Merge Sort apply the "divide-and-conquer" paradigm. As we saw in Quick Sort, a selection procedure was followed by two recursive calls. However, for Merge Sort, two recursive calls was followed by a merging procedure, which could be described as follows: First divide the file in half; sort the two halves recursively; and then merge the two halves together while sorting them.

The following program is an example in C language for Merge Sort with two recursive calls, where we have used $b[l], \dots, b[r]$ as an auxiliary array to manage the merging operation without sentinels.

```
mergesort(int a[], int l, int r) {
  int i, j, k, m;
  if (r > l) {
    m = (r + l)/2;
    mergesort(a, l, m);
    mergesort(a, m + 1, r);
    for (i = m + 1; i > l; i--) b[i - 1] = a[i - 1];
    for (j = m; j < r; j++) b[r + m - j] = a[j + 1];
    for (k = l; k <= r; k++)
      a[k] = (b[i] < b[j]) ? b[i++] : b[j--];
  }
}
```

A C program example without nonrecursive calls is also given below, using linked lists, in which a bottom–up approach has been applied.

```
struct node *mergesort(struct node *c) {
  int i, n;
  struct node *a, *b, *head; *todo, *t;
  head = (struct node *) malloc(sizeof *head);
  head->next = c;
  a = z;
  for (n = 1; a != head->next; n = n + n) {
    todo = head->next;
    c = head;
    while (todo != z) {
      t = todo; a = t;
      for (i = 1; i < n; i++) t = t->next;
      b = t->next;
      t->next = z;
      t = b;
      for (i = 1; i < n; i++) t = t->next;
      todo = t->next;
      t->next = z;
      c->next = merge(a, b);
      for (i = 1; i <= n + n; i++) c = c->next;
    }
  }
  return head->next;
}
```

Quiz: Generate an array of 10 random integers, and write a complete C program using Merge Sort (both Recursive and Non-recursive versions) to sort the array into increasing order.

[Hint: Refer to the Web Page or CD-ROM to see the sorting process.]

9.5. Shell Sort

Shell Sort is basically an improvement of Insertion Sort. As we have seen before, Insertion Sort is slow since it compares and exchanges only elements in neighbor. Shell Sort solves this problem by allowing comparison and exchange of elements that are far apart to gain speed. The detailed procedure is given as follows: First take every hth element to form a new collection of elements and sort them (using Insertion Sort), which is called h-sort; Then choose a new h with a smaller value, for example, calculated by $h_i+1 = 3^*h_i+1$, or $h_i = (h_i+1-1)/3$; $h_0 = 1$, thus, we have a sequence [..., 1093, 364, 121, 40, 13, 4, 1]; Repeat until $h = 1$, then the file will be sorted in the proper order.

An implementation in C language for Shell Sort is given below.

```
void shell_sort(int array[], int n)
{
 int i, j, k, tmp;
 for (k = 1; k < n/9; k = 3 * k + 1);
 for (; k > 0; k/ = 3) {
  for (i = k; i < n; i++) {
   j = i;
   while (j >= k && array[j - k] > array[j]) {
  tmp = array[j];
  array[j] = array[j - k];
  array[j - k] = tmp;
  j- = k;
   }
  }
 }
}
```

Quiz: Generate an array of 10 random integers, and write a complete C program using Shell Sort to sort the array into increasing order.

[Hint: Refer to the Web Page or CD-ROM to see the sorting process.]

9.6. Radix Sort

The basic idea of Radix Sort is to treat keys as numbers represented in a base-M number system, for example, $M = 2$ or some power of 2, and process with individual digits of the numbers, so that all keys with leading bit 0 appear before all keys with leading bit 1. And among the keys with leading bit 0, all keys with second bit 0 appear before all keys with second bit 1, and so forth. There are two approaches for Radix Sort: Radix Exchange Sort and Straight Radix Sort. The feature of Radix Exchange Sort is that it scans from left to right, that is, scans from the left to find a key which starts with bit 1, and scans from the right to find a key which starts with bit 0, then exchange; continue the process until the scanning pointers cross. This leads to a recursive sorting procedure that is very similar to Quick Sort. On the other hand, Straight Radix Sort, as an alternative method, scans the bits from right to left.

We show a C Program Example for Radix Exchange Sort as follows, which is very similar to the recursive implementation of Quick Sort.

```
void radixexchange(int array[], int l, int r, int b)
{
 int i, j, tmp;
 if (r > l && b >= 0) {
  i = l;
  j = r;
  while (j != i){
    while (bits(array[i], b, 1) == 0 && i < j) i++;
    while (bits(array[j], b, 1) != 0 && j > i) j--;
    tmp = array[i];
    array[i] = array[j];
    array[j] = tmp;
  }
  if (bits(array[r], b, 1) == 0) j++;
  radixexchange(array, l, j - 1, b - 1);
  radixexchange(array, j, r, b - 1);
 }
}
```

Table 9.1. Performance comparison.

Algorithms	Performance	Comments
Selection Sort	N^2	good for small and partially sorted data
Insertion Sort	N^2	good for almost sorted data
Bubble Sort	N^2	good for $N < 100$
Quick Sort	$N \log N$	excellent
Heap Sort	$N \log N$	excellent
Merge Sort	$\log N$	good for external sort
Shell Sort	$N^{1.5}$	good for medium N
Radix Sort	$N \log N$	excellent

```
unsigned bits(unsigned x, int k, int j)
{
 return (x >> k) & ~ (~ 0 << j);
}
```

Quiz: Generate an array of 10 random integers, and write a complete C program using Radix Sort to sort the array into increasing order.

[Hint: Refer to the Web Page or CD-ROM to see the sorting process.]

9.7. Comparison of Sorting Algorithms

In this section, we give a performance comparison of all these sorting algorithms described in this chapter. As shown in Table 9.1, Selection Sort, Insertion Sort and Bubble Sort have a performance of N^2. Selection Sort is good for small and partially sorted data, while Insertion Sort is good for data that are already almost sorted. On the other hand, Heap Sort is not so good for sorted data, but has a very good performance of $N \log N$. Quick Sort and Radix Sort have the same performance as Heap Sort. They are considered to be excellent among all these sorting algorithms. Bubble Sort is good for $N < 100$, and Shell Sort is considered to be good for medium N, which has a performance of $N^{1.5}$. Finally, Merge Sort has a performance of $\log N$, and is good for external sort.

References

1. R. Sedgewick, *Algorithms in C*, Addison-Wesley, 1990.
2. M. A. Weiss, *Data Structures and Algorithm Analysis in C*, Second Edition, Addison-Wesley, 1997.

3. R. Kruse, C. L. Tondo and B. Leung, *Data Structures and Program Design in C*, Second Edition, Prentice-Hall, 1997.
4. I. T. Adamson, *Data Structures and Algorithms: A First Course*, Springer, 1996.

Appendices

Appendix 9.1.1: *Example of a Complete C Program Source for Selection Sort*

```
#include <stdio.h>
#include <stdlib.h>
#define SIZE 10

void selection_sort(int array[], int n)
{
  int i, j, min, tmp;

  for (i=0; i<n; i++) {
    min = i;

    for (j=i+1; j<n; j++) {
      if (array[j] < array[min]) min = j;
    }

    tmp = array[min];
    array[min] = array[i];
    array[i] = tmp;
  }
}

main ()
{
  int i, j;
  int order[SIZE] = {25, 41, 92, 3, 60, 18, 76, 39, 54, 87};

  printf("Initial Order:\n");
  for (i=0; i<SIZE; i++) {
    printf("%d ", order[i]);
  }
  printf("\n");

  printf("Selection Sort:\n");
  selection_sort(order, SIZE);
```

```
    for (i=0; i<SIZE; i++) {
    printf("%d ", order[i]);
  }
  printf("\n");
}
```

Appendix 9.1.2: *Example of a Complete C Program Source for Insertion Sort*

```
#include <stdio.h>
#include <stdlib.h>
#define SIZE 10

void insertion_sort(int array[], int n)
{
  int i, j, tmp;

  for (i=1; i<n; i++) {
    tmp = array[i];

    for (j=i; array[j-1]>tmp; j--) {
      array[j] = array[j-1];
    }

    array[j] = tmp;
  }
}

main ()
{
  int i, j;
  int order[SIZE] = {25, 41, 92, 3, 60, 18, 76, 39, 54, 87};

  printf("Initial Order:\n");
  for (i=0; i<SIZE; i++) {
    printf("%d ", order[i]);
  }
  printf("\n");

  printf("Insertion Sort:\n");
  insertion_sort(order, SIZE);

  for (i=0; i<SIZE; i++) {
    printf("%d ", order[i]);
  }
  printf("\n");
}
```

Appendix 9.1.3: *Example of a Complete C Program Source for Bubble Sort*

```
#include <stdio.h>
#include <stdlib.h>
#define SIZE 10

void bubble_sort(int array[], int n)
{
  int i, j, tmp;

  for (i=n; i>0; i--) {
    for (j=1; j<i; j++) {

      if (array[j-1]>array[j]) {
tmp = array[j-1];
array[j-1] = array[j];
array[j] = tmp;
      }

    }
  }
}

main ()
{
  int i, j;
  int order[SIZE] = {25, 41, 92, 3, 60, 18, 76, 39, 54, 87};

  printf("Initial Order:\n");
  for (i=0; i<SIZE; i++) {
    printf("%d ", order[i]);
  }
  printf("\n");

  printf("Bubble Sort:\n");
  bubble_sort(order, SIZE);
  for (i=0; i<SIZE; i++) {
    printf("%d ", order[i]);
  }
  printf("\n");
}
```

Appendix 9.2.1: *Example of a Complete C Program Source for Quick Sort*

1. Recursive Quick Sort

```
#include <stdio.h>
#include <stdlib.h>
#define SIZE 10

void printArray(int[]);

void quick_sort_r(int array[], int l, int r)
{
  int i, j, v, tmp;

  if (l < r) {
    v = array[r];
    i = l-1;
    j = r;

    while (1) {
      while (array[++i] < v);
      while (array[--j] > v);

      if (i>=j) break;

      tmp = array[i];
      array[i] = array[j];
      array[j] = tmp;

      printArray(array);
    }

    tmp = array[i];
    array[i] = array[r];
    array[r] = tmp;
    printArray(array);
    quick_sort_r(array, l, i-1);
    quick_sort_r(array, i+1, r);
  }
}


void printArray(int a[])
{
  int i;

  for (i=0; i<SIZE; i++){
    printf("%d ", a[i]);
  }
  printf("\n");
}

main ()
{
  int i, j;
  int order[SIZE] = {25, 41, 92, 3, 60, 18, 76, 39, 54, 87};

  printf("Initial Order:\n");
  printArray(order);

  printf("Quick Sort:\n");
  quick_sort_r(order, 0, SIZE-1);

  printArray(order);
}
```

2. Nonrecursive Quick Sort

```
#include <stdio.h>
#include <stdlib.h>
#define SIZE 10
#define MAX 100

void push(int);
int pop();
void stackinit();
int stackempty();

int stack[MAX];
```

```
void printArray(int a[])
{
  int i;

  for (i=0; i<SIZE; i++){
    printf("%d ", a[i]);
  }
  printf("\n");
}

void quick_sort(int array[], int n)
{
  int i, j, v, tmp, l=1, r=n;

  stackinit();

  while (1) {
    while (l < r) {
      v = array[r];
      i = l-2;
      j = r;

      while (1) {
while (array[++i] < v);
while (array[--j] > v);

if (i>=j) break;

tmp = array[i];
array[i] = array[j];
array[j] = tmp;

printArray(array);
      }

      tmp = array[i];
      array[i] = array[r];
      array[r] = tmp;
      printArray(array);
      if (r-i < i-l) {
push(l);
push(i-1);
l = i+1;
      } else {
push(i+1);
push(r);
r = i-1;
     }
    }

    if (stackempty()) break;

    r = pop();
    l = pop();
  }
}

void push(int x)
{
  int i;

  for (i=0; i<MAX; i++) {
    if (stack[i] == 0) {
      stack[i] = x;
      return;
    }
  }
}

int pop()
{
  int i, v;

  for (i=0; i<MAX; i++) {
    if (stack[i] == 0 && i>0) {
      v = stack[i-1];
      stack[i-1] = 0;
      return v;
    }
  }

  return NULL;
}

void stackinit()
{
  int i;

  for (i=0; i<MAX; i++){
    stack[i] = 0;
  }
}
```

```
int stackempty()
{
  if (stack[0] == 0) return 1;
  else return 0;
}

main ()
{
  int i, j;
  int order[SIZE] = {25, 41, 92, 3, 60, 18, 76, 39, 54, 87};

  printf("Initial Order:\n");
  printArray(order);

  printf("Quick Sort:\n");
  quick_sort(order, SIZE-1);

  printArray(order);
}
```

Appendix 9.3.1: *Example of a Complete C Program Source for Heap Sort*

```
#include <stdio.h>
#include <stdlib.h>
#define SIZE 10

void down_heap(int[], int, int);

void heap_sort(int array[], int n)
{
  int x, tmp;

  for (x=n/2; x>=0; x--) {
    down_heap(array, n, x);
  }

  while (n>0) {
    tmp = array[0];
    array[0] = array[n];
    array[n] = tmp;

    down_heap(array, --n, 0);
  }
}
```

```
void down_heap(int array[], int n, int x)
{
  int i, v;
  v = array[x];

  while (x <= n/2) {
    i = x+x;
    if (i<n && array[i]<array[i+1]) i++;

    if (v>=array[i]) break;

    array[x] = array[i];
    x = i;
  }

  array[x] = v;
}

main ()
{
  int i, j;
  int order[SIZE] = {25, 41, 92, 3, 60, 18, 76, 39, 54, 87};

  printf("Initial Order:\n");
  for (i=0; i<SIZE; i++) {
    printf("%d ", order[i]);
  }
  printf("\n");

  printf("Heap Sort:\n");
  heap_sort(order, SIZE-1);

  for (i=0; i<SIZE; i++) {
    printf("%d ", order[i]);
  }
  printf("\n");
}
```

Appendix 9.4.1: *Example of a Complete C Program Source for Merge Sort*

1. Recursive Merge Sort

```
#include <stdio.h>
#include <stdlib.h>
```

```
#define SIZE 10

void printArray(int a[])
{
  int i;

  for (i=0; i<SIZE; i++){
    printf("%d ", a[i]);
  }
  printf("\n");
}

void merge_sort_r(int array[], int l, int r)
{
  int i, j, u, v, tmp[r];

  if (l<r) {
    v = (l+r)/2;
    merge_sort_r(array, l, v);
    merge_sort_r(array, v+1, r);

    for (i=v+1; i>l; i--) tmp[i-1] = array[i-1];
    for (j=v; j<r; j++) tmp[r+v-j] = array[j+1];

    for (u=l; u<=r; u++) {
      if (tmp[i] < tmp[j]) {
array[u] = tmp[i++];
      } else {
array[u] = tmp[j--];
      }
    }
    printArray(array);
  }
}

main ()
{
  int i, j;
  int order[SIZE] = {25, 41, 92, 3, 60, 18, 76, 39, 54, 87};

  printf("Initial Order:\n");
  printArray(order);

  printf("Merge Sort:\n");
  merge_sort_r(order, 0, SIZE-1);
  printArray(order);
}
```

2. Nonrecursive Merge Sort

```
#include <stdio.h>
#include <stdlib.h>
#define SIZE 10

struct node *tail;

struct node *merge(struct node *, struct node *);

struct node *merge_sort(struct node *c)
{
  int i, n;
  struct node *a, *b, *head, *todo, *t;

  head = (struct node *)malloc(sizeof (struct node *));
  head->next = c;
  tail->next = tail;
  a = tail;

  for (n=1; a != head->next; n=n+n) {
    todo = head->next;
    c = head;

    while (todo != tail) {
      t = todo;
      a = t;

      for (i=1; i<n; i++) t = t->next;
      b = t->next;
      t->next = tail;
      t = b;
      for (i=1; i<n; i++) t = t->next;
      todo = t->next;
      t->next = tail;
      c->next = merge(a, b);
      for (i=1; i<=n+n; i++) c = c->next;
    }
  }
  return head->next;
}
```

```
struct node *merge(struct node *a, struct node *b)
{
  struct node *c = tail;

  do
    if (a->key <= b->key) {
      c->next = a;
      c = a;
      c = a->next;

    } else {
      c->next = b;
      c = b;
      b = b->next;

    }
  while (c != tail);

  c = tail->next;
  tail->next = tail;

  return c;
}

void print_stack(struct node *c) {
  while (c != tail) {
    printf("%d ", c->key);
    c = c->next;
  }
  printf("\n");
}
```

Appendix 9.5.1: *Example of a Complete C Program Source for Shell Sort*

```
#include <stdio.h>
#include <stdlib.h>
#define SIZE 10

void shell_sort(int array[], int n)
{
  int i, j, k, tmp;

  for (k=1; k < n/9; k=3*k+1);
  for (; k>0; k/=3) {
    for (i=k; i<n; i++) {
      j = i;

      while (j>=k && array[j-k] > array[j]) {
tmp = array[j];
array[j] = array[j-k];
array[j-k] = tmp;

j -= k;
      }
    }
  }
}

main ()
{
  int i, j;
  int order[SIZE] = {25, 41, 92, 3, 60, 18, 76, 39, 54, 87};

  printf("Initial Order:\n");
  for (i=0; i<SIZE; i++) {
    printf("%d ", order[i]);
  }
  printf("\n");

  printf("Shell Sort:\n");
  shell_sort(order, SIZE);

  for (i=0; i<SIZE; i++) {
    printf("%d ", order[i]);
  }
  printf("\n");
}
```

Appendix 9.6.1: *Example of a Complete C Program Source for Radix Sort*

```
#include <stdio.h>
#include <stdlib.h>
#define SIZE 10

unsigned bits(unsigned, int, int);

void radixexchange(int array[], int l, int r, int b)
```

```
{
  int i, j, tmp;

  if (r>l && b>=0) {
    i = l;
    j = r;

    while (j != i){
      while (bits(array[i], b, 1) == 0 && i<j) i++;
      while (bits(array[j], b, 1) != 0 && j>i) j--;

      tmp = array[i];
      array[i] = array[j];
      array[j] = tmp;
    }

    if (bits(array[r], b, 1) == 0) j++;
    radixexchange(array, l, j-1, b-1);
    radixexchange(array, j, r, b-1);
  }
}

unsigned bits(unsigned x, int k, int j)
{
  return (x>>k) & ~(~0<<j);
}

main ()
{
  int i, j;
  int order[SIZE] = {25, 41, 92, 3, 60, 18, 76, 39, 54, 87};

  printf("Initial Order:\n");
  for (i=0; i<SIZE; i++) {
    printf("%d ", order[i]);
  }
  printf("\n");

  printf("Radix Sorting...\n");
  radixexchange(order, 0, SIZE-1, 30);

  for (i=0; i<SIZE; i++) {
    printf("%d ", order[i]);
  }
  printf("\n");
}
```

Exercises

Beginner's Exercises

1. Generate an array of 20 random integers. Use Selection Sort to sort the array into increasing order.
2. Use Insertion Sort to sort the array created in Exercise 1 into increasing order.
3. Use Bubble Sort to sort the array created in Exercise 1 into increasing order.
4. Use the recursive Quick Sort to sort the array created in Exercise 1 into increasing order. Output the array before and after sort.
5. Use Heap Sort to sort the array created in Exercise 1 into increasing order.
6. Use the recursive Merge Sort to sort the array created in Exercise 1 into increasing order. Output the array before and after sort.
7. Use Shell Sort to sort the array created in Exercise 1 into increasing order.
8. Use Radix Exchange Sort to sort the array created in Exercise 1 into increasing order.

Intermediate Exercises

9. Write a C program using Selection Sort to sort the following string in alphabetic order.

   AMULTIMEDIACOURSE

10. Use Insertion Sort to sort the string in Exercise 9.
11. Use Bubble Sort to sort the string in Exercise 9.
12. Use Quick Sort to sort the string in Exercise 9.
13. Use Heap Sort to sort the string in Exercise 9.
14. Use Merge Sort to sort the string in Exercise 9.
15. Use Shell Sort to sort the string in Exercise 9.
16. Use Radix Exchange Sort to sort the string in Exercise 9.

Advanced Exercises

17. Use the nonrecursive Quick Sort to sort the array created in Exercise 1 into increasing order. Compare the output of the recursive Quick Sort in Exercise 4.
18. Use the nonrecursive Merge Sort to sort the array created in Exercise 1 into increasing order. Compare the output of the recursive Merge Sort in Exercise 6.

19. Generate an array of 1000 random integers. Use Selection Sort, Insertion Sort, Bubble Sort, Quick Sort, Heap Sort, Merge Sort, Shell Sort, and Radix Sort to sort the array into increasing order. And compare their execution time.