complexity of all these techniques is proportional to n, where n is the number of data elements. Hence, these search techniques are also called as *quantity-dependent search techniques*.

Searching and sorting are the two very important operations performed most commonly on a large amount of information. We have studied various algorithms to search a record, and we notice that it is much easier to find any information that is organized in some proper order. For example, if we want to find any name in the telephone directory, which contains names in any random order, we perhaps have to go through the whole directory sequentially to find the name. Similarly, consider the trouble we might have to take to search for a book in a library where the books are placed anywhere without any order. We can imagine the ease if these books are assigned a specific position and are shelved in a specific order. In general, sorting is performed in business data-processing applications to retrieve information more efficiently. Let us see more details of sorting and methods associated with it. More details on the hash table are covered in Chapter 11.

9.3 SORTING

One of the fundamental problems in computer science is ordering a list of items. There are plenty of solutions to this problem, commonly known as *sorting algorithms*. Some sorting algorithms are simple and iterative, such as the bubble sort. Others such as the quick sort are extremely complicated but produce lightning-fast results.

Sorting is the operation of arranging the records of a table according to the key value of each record, or it can be defined as the process of converting an unordered set of elements to an ordered set.

A table or a file is an ordered sequence of records $r[1]$, $r[2]$, …, $r[n]$, each containing a key $k[1]$, $k[2]$, … , $k[n]$. This key is usually one of the fields of the entire record. The table is said to be sorted on the key if $i < j$ implies that $k[i]$ precedes $k[j]$ in some ordering on the keys.

9.3.1 Types of Sorting

Sorting algorithms are divided into two categories: internal and external sorts.

If all the records to be sorted are kept internally in the main memory, they can be sorted using an internal sort. However, if there are a large number of records to be sorted, they must be kept in external files on auxiliary storage. They have to be sorted using external sort.

Internal Sorting

Any sort algorithm that uses main memory exclusively during the sorting is called as an *internal sort algorithm*. This assumes high-speed and random access to all data members. All the methods described in this chapter assume that all the data is stored in high-speed main memory of the computer and are therefore internal sorting techniques, except for merge sort. Internal sorting is faster than external sorting. The various internal sorting techniques are the following:

1. Bubble sort
2. Insertion sort
3. Selection sort
4. Quick sort
5. Heap sort
6. Shell sort
7. Bucket sort
8. Radix sort
9. File sort
10. Merge sort

External Sorting

Any sort algorithm that uses external memory, such as tape or disk, during the sorting is called as an *external sort algorithm*. *Merge sort* uses external memory. Do note that the other algorithms may read the initial values from a magnetic tape or write sorted values to a disk, but this is not using external memory during the sort.

Most of the methods to be described involve the movement of records within the table. For example, consider Fig. 9.2(a) where a table of four records is shown. Figure 9.2(b) shows a sorted table, which results when the table of Fig. 9.2(a) is sorted in an increasing order on the numeric key.

	Key	Other fields
#1	13	Shalu
#2	10	Gilda
#3	20	Raj
#4	5	Usha

(a)

Key	Other fields
5	Usha
10	Gilda
13	Shalu
20	Raj

(b)

Fig. 9.2 Movement of records within tables (a) Before sorting (b) After sorting

In this case, the actual records are moved from one place to another in the table.

In certain applications, the records can be quite long, and it is very expensive to move the actual data. One way to reduce record movement is to use an auxiliary table of pointers, each pointing to one record of the table to be sorted. Then, we can move these pointers instead of moving the actual records. For example, consider Fig. 9.3 which contains a table to be sorted and shows sorting using pointers.

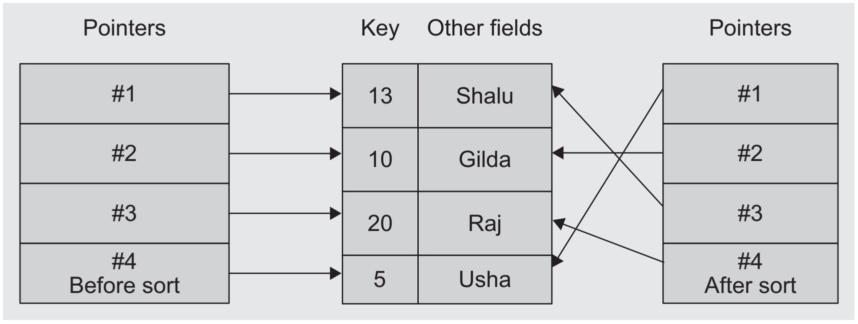


Fig. 9.3 Sorting with pointers

The table at the left is the initial table of pointers. These pointers are adjusted during the sorting process to produce the final table of pointers as on the right of the original table.

We may note the actual records in the table are not moved. While describing the algorithms ahead, we assume that we are moving the actual records, and we will only sort the keys.

9.3.2 General Sort Concepts

Let us now discuss some general terms related to sorting.

Sort Order

Data can be ordered either in ascending or in descending order. The order in which the data is organized, either ascending or descending, is called *sort order*. For example, the percentages of marks obtained by students in the examination are organized in descending order to decide ranks, whereas the names in the telephone directory are organized alphabetically in ascending order.

Sort Stability

A sorting method is said to be stable if at the end of the method, identical elements occur in the same relative order as in the original unsorted set. While sorting, we must take care of the special case—when two or more of the records have the same key, it is important to preserve the order of records in this case of duplicate keys. A sorting algorithm is said to be stable if it preserves the order for all records with duplicate keys; that means, if for all records *i* and *j* is such that *k*[*i*] is equal to *k*[*j*] and if *r*[*i*] precedes to *r*[*j*] in the unsorted table, then *r*[*i*] precedes to *r*[*j*] in the sorted table too. Bubble sort, selection sort, and insertion sort are the stable sort methods. Example 9.2 illustrates examples of both stable and unstable sort methods.

EXAMPLE 9.2 Consider the following unsorted sequence of marks to be sorted in descending order. Sort this sequence using the stable and unstable sort methods.

Name	Uma	Saurabh	Sanika	Kasturi	Ashish	Harsha	Lelo
Marks	80	90	93	95	83	90	83

Solution The stable sort method will sort the sequence as

Name	Kasturi	Sanika	**Saurabh**	**Harsha**	**Ashish**	**Lelo**	Uma
Marks	95	93	**90**	**90**	**83**	**83**	80

whereas, the unstable sort method may sort the same sequence as

Name	Kasturi	Sanika	**Harsha**	**Saurabh**	**Lelo**	**Ashish**	Uma
Marks	95	93	**90**	**90**	**83**	**83**	80

Sort Efficiency

Each sorting method may be analysed depending on the amount of time necessary for running the program and the amount of space required for the program. The amount of time for running a program is proportional to the number of key comparisons and the movement of records or the movement of pointers to records.

Sort efficiency is a measure of the relative efficiency of a sort. It is usually an estimate of the number of comparisons and data movement required to sort the data. We will discuss various sorting algorithms in Sections 9.3.3–9.3.12. While analysing our sorting methods, we will concentrate on these aspects of the sorting algorithms. We will start with simple methods such as bubble sort, selection sort, and insertion sort and proceed to more complex and efficient ones such as quick sort, shell sort, and bucket sort.

Passes

During the sorted process, the data is traversed many times. Each traversal of the data is referred to as a *sort pass*. Depending on the algorithm, the sort pass may traverse the whole list or just a section of the list. In addition, the characteristic of a sort pass is the placement of one or more elements in a sorted list.

9.3.3 Bubble Sort

The bubble sort is the oldest and the simplest sort in use. Unfortunately, it is also the slowest. The bubble sort works by comparing each item in the list with the item next to it and swapping them if required. The algorithm repeats this process until it makes a pass all the way through the list without swapping any items (in other words, all items are in the correct order). This causes larger values to 'bubble' to the end of the list while smaller values 'sink' towards the beginning of the list. In brief, the bubble sort derives its name from the fact that the smallest data item bubbles up to the top of the sorted array. Figure 9.4 demonstrates the bubble technique by showing numbers and their moves during each pass.

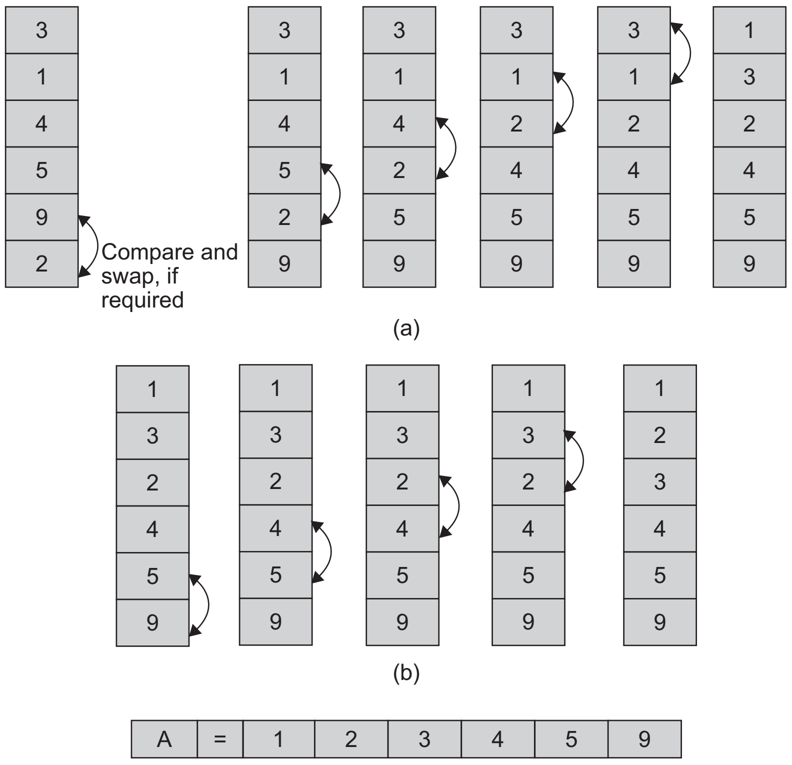


Fig. 9.4 Bubble sort (a) Pass 1 (i = 1) (b) Pass 2 (i = 2) (c) the resultant sorted array after pass (n − 1) (i = 5),

Algorithm 9.5 depicts the logic behind bubble sort.

ALGORITHM 9.5

```
1. Let A be the array to be sorted
2. for i = 1 to n - 1
   for j = 0 to n - i
   begin
      if A[j] > A[j+1] then
         Swap A[j] with A[j + 1] as follows
         temp = A[j]
         A[j] = A[j + 1]
         A[j + 1] = temp
      end
   end
3. stop
```

Figure 9.5 illustrates a bubble sort using an array of size 7.

76	**67**	36	55	23	14	06

(a)

Array		**76**	**67**	36	55	23	14	6
Pass 1	Step 1	67	**76**	**36**	55	23	14	6
	Step 2	67	36	**76**	**55**	23	14	6
	Step 3	67	36	55	**76**	**23**	14	6
	Step 4	67	36	55	23	**76**	**14**	6
	Step 5	67	36	55	23	14	**76**	**6**
	Step 6	**67**	**36**	55	23	14	6	**76**
Pass 2	Step 1	36	**67**	**55**	23	14	6	76
	Step 2	36	55	**67**	**23**	14	6	76
	Step 3	36	55	23	**67**	**14**	6	76
	Step 4	36	55	23	14	**67**	**6**	76
	Step 5	**36**	**55**	23	14	6	**67**	**76**
Pass 3	Step 1	36	**55**	**23**	14	6	67	76
	Step 2	36	23	**55**	**14**	6	67	76
	Step 3	36	23	14	**55**	**6**	67	76
	Step 4	**36**	**23**	14	6	**55**	**67**	**76**
Pass 4	Step 1	23	**36**	**14**	6	55	67	76
	Step 2	23	14	**36**	**6**	55	67	76
	Step 3	**23**	**14**	6	**36**	**55**	**67**	**76**
Pass 5	Step 1	14	**23**	**6**	36	55	67	76
	Step 2	**14**	**6**	**23**	**36**	**55**	**67**	**76**
Pass 6	Step 1	6	14	23	36	55	67	76
Final sorted array		**6**	**14**	**23**	**36**	**55**	**67**	**76**

(b)

Fig. 9.5 Example of bubble sorting (a) Initial array (b) Final sorted array with passes

Program Code 9.7 illustrates the bubble sort function.

PROGRAM CODE 9.7

```
// function for bubble sort for array A having n elements
void bubblesort(int A[max], int n)
{
   int i, j,temp;
   for(i = 1; i < n; i++)      // number of passes
   {
      for(j = 0; j < n - i; j++)     // j varies from 0 to
                                    // n - i
      {
         if( A[j] > A[j + 1] )    // compare two successive
                                 // numbers
         {
            temp = A[j];      // swap A[j] with A[j + 1]
            A[j] = A[j + 1];
            A[j + 1] = temp;
         }
      }
   }
}
```

For descending order of sorting, only the comparison condition should be changed in Program Code 9.7.

```
if(A[j] < A[j + 1] )   // change as  <
{
   temp = A[j];      // swap A[j] with A[j + 1]
   A[j] = A[j + 1];
   A[j + 1] = temp;
}
```

Analysis of Bubble Sort

The algorithm begins by comparing the top item of the array with the next and swapping them if necessary. After $n - 1$ comparisons, the largest among a total of n items descends to the bottom of the array, that is, to the n^{th} location. The process is then repeated to the remaining $n - 1$ items in the array. For n data items, the method requires $n(n - 1)/2$ comparisons and on an average, almost one-half as many swaps. The bubble sort, therefore, is very inefficient in large sorting jobs.

The analysis of this routine is a bit difficult. If we do not stop iterations when the array is sorted, the analysis is simple.

The number of comparisons made at each of the iterations is as follows:
$(n - 1)$ comparisons in the first iteration, $(n - 2)$ comparisons in the second iteration, ..., one comparison in the last iteration
This totals up to

$$(n - 1) + (n - 2) + (n - 3) + \ldots + 1 = n(n - 1)/2$$

Thus, the total number of comparisons is $n(n - 1)/2$, which is $O(n^2)$.
Hence, the time complexity for each of the cases is given by the following:

1. Average case complexity = $O(n^2)$
2. Best case complexity = $O(n^2)$
3. Worst case complexity = $O(n^2)$

9.3.4 Insertion Sort

The insertion sort works just like its name suggests—it inserts each item into its proper place in the final list. The simplest implementation of this requires two list structures: the source list and the list into which the sorted items are inserted.

Let us consider a list $L = \{3, 6, 9, 14\}$. Given this sorted list, we need to insert a new element 5 in it. The commonly used process would involve the following steps:

1. Compare the new element 5 and the last element 14
2. Shift 14 right to get 3, 6, 9, ,14
3. Shift 9 right to get 3, 6, ,9, 14
4. Shift 6 right to get 3, ,6, 9, 14
5. Insert 5 to get 3, 5, 6, 9, 14

These steps could be coded as the following piece of code:

```
// insert t into a[0:i - 1]
int j;
// let X be the element to be inserted
// shift elements from the last member to right by one position
// till you get a smaller one
for(j = i - 1; j >= 0 && X < a[j]; j--)
   a[j + 1] = a[j];
// Insert t at j + 1 location
a[j + 1] = X;
```

These steps when done for each element of the list are to be sorted by considering another list and starting with one element in it. The steps for inserting an element in the sorted list can then be repeatedly used to yield the sorted list. Let us consider the following list of numbers: $L = \{7, 3, 5, 6, 1\}$. The following steps are required to sort this list.

1. Start with 7 and insert 3 => 3, 7
2. Insert 5 => 3, 5, 7

3. Insert 6 => 3, 5, 6, 7
4. Insert 1 => 1, 3, 5, 6, 7

The piece of code needed to do this will look like

```
for(int i = 1; i < n; i++)
{  // insert a[i] into a[0:i - 1]
   // code to insert comes here
}
```

After adding the code for insertion we have already built, the resultant code will be

```
for(int i = 1; i < n; i++)
{   // insert a[i] into a[0:i - 1]
   int t = a[i];
   int j;
   for(j = i - 1; j >= 0 && t < a[j]; j--)
      a[j + 1] = a[j];
   a[j + 1] = t;
}
```

To save memory, most implementations use an in-place sort that works by moving the current item past the already sorted items and repeatedly swapping it with the preceding item until it is in place. The main idea behind the insertion sort is to insert the i^{th} element, in the i^{th} pass, into $A(1)$, $A(2)$, ..., $A(i)$, in the right place. Algorithm 9.6 lists the steps for insertion sort.

ALGORITHM 9.6

```
1. Set J = 2, where J is an integer
2. Check if list (J) < list (J – 1): if so interchange them; set J = J –1
   and repeat step (2) until J = 1
3. Set J = 3, 4, 5,. . ., N and keep on executing step (2)
```

The following steps in Example 9.3 essentially define the insertion sort as applied to sorting into ascending order an array list containing *N* elements:

EXAMPLE 9.3 Consider the given unsorted array. Sort this array in ascending order using insertion sort.

	Original unsorted array						
Elements	76	67	36	55	23	14	6
Index	0	1	2	3	4	5	6

Solution Pass 1: Consider the first list is sorted, and insert the second number 67 in the first list.

	Sorted array	Unsorted array					
Elements	76	67	36	55	23	14	6
Index	0	1	2	3	4	5	6

Pass 2: Insert number 36 in the first list.

	Sorted array		Unsorted array				
Elements	67	76	36	55	23	14	6
Index	0	1	2	3	4	5	5

Pass 3: Insert number 55 in the first list.

	Sorted array			Unsorted array			
Elements	36	67	76	55	23	14	6
Index	0	1	2	3	4	5	6

Pass 4: Insert number 23 in the first list.

	Sorted array				Unsorted array		
Elements	36	55	67	76	23	14	6
Index	0	1	2	3	4	5	6

Pass 5: Insert number 14 in the first list.

	Sorted array					Unsorted array	
Elements	23	36	55	67	76	14	6
Index	0	1	2	3	4	5	6

Pass 6: Insert number 6 in the first list.

	Sorted array						Unsorted array
Elements	14	23	36	55	67	76	6
Index	0	1	2	3	4	5	6

The final sorted array is

	Sorted array						
Elements	6	14	23	36	55	67	76
Index	0	1	2	3	4	5	6

Program Code 9.8 defines the `InsertionSort()` function.

PROGRAM CODE 9.8

```
void InsertionSort(int A[], int n)
{
   int i, j, element;
   for(i = 1; i < n; i++)
   {
      element = A[i];
      // insert ith element in 0 to i − 1 array
      j = i;
      while((j > 0) && (A[j − 1] > element))
      //compare if A[j − 1] > element
      {
         A[j] = A[j − 1];      // shift elements
         j = j − 1;
      }
      A[j] = element;      // place element at jth position
   }
}
```

Analysis of Insertion Sort

Although the insertion sort is almost always better than the bubble sort, the time required in both the methods is approximately the same, that is, it is proportional to n^2, where n is the number of data items in the array.

The total number of comparisons is given as follows:

$$(n-1)+(n-2)+\dots+1=(n-1)\times n/2$$

which is $O(n^2)$.

If the data is initially sorted, only one comparison is made on each pass so that the sort time complexity is $O(n)$. The number of interchanges needed in both the methods is on an average $(n^2)/4$, and in the worst case is about $(n^2)/2$.

When the data is already partially ordered, the insertion sort will normally take less time than the bubble sort. The insertion sort is highly efficient if the array is already in an almost sorted order. Example 9.4 provides a pictorial representation of the insertion sort.

EXAMPLE 9.4 Figure 9.6 is an example of insertion sort.

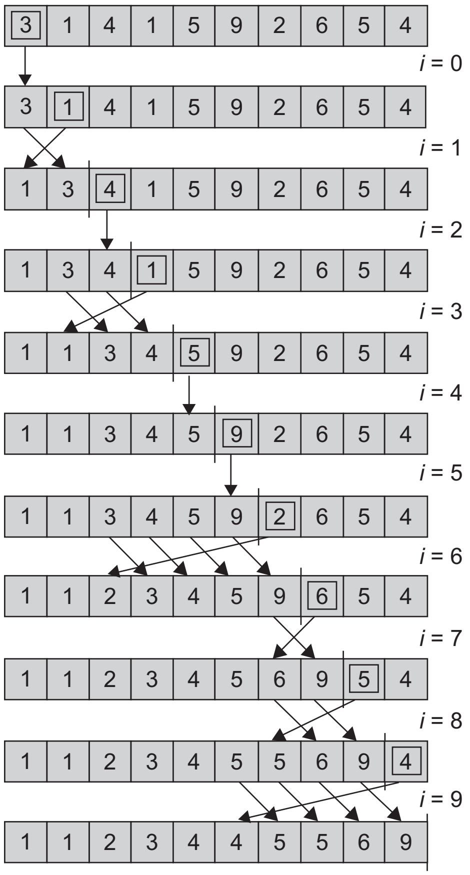


Fig. 9.6 Insertion sorting

9.3.5 Selection Sort

The selection sort algorithms construct the sorted sequence, one element at a time, by adding elements to the sorted sequence *in order*. At each step, the next element to be added to the sorted sequence is selected from the remaining elements.

Because the elements are added to the sorted sequence in order, they are always added at one end. This makes the selection sorting different from the insertion sorting. In insertion sorting, the elements are added to the sorted sequence in an arbitrary order. Therefore, the position in the sorted sequence at which each subsequent element is inserted is arbitrary.

Both selection and insertion sorts sort the arrays *in-place*.

In this method, we sort a set of unsorted elements in two steps. In the first step, find the smallest element in the structure. In the second step, swap the smallest element with the element at the first position. Then, find the next smallest element and swap with the element at the second position. Repeat these steps until all elements get arranged at proper positions.

This is illustrated in Program Code 9.9.

PROGRAM CODE 9.9

```
void SelectionSort(int A[], int n)
{
   int i, j;
   int minpos, temp;
   for(i = 0; i < n - 1; i++)
   {
      minpos = i;
      for(j = i + 1; j < n; j++)
      //find the position of min element as minpos from
      //i + 1 to n - 1
      {
         if(A[j] < A[minpos])
            minpos = j;
      }
      if(minpos != i)
      {
         temp = A[i];
         // swap the ith element and minpos element
         A[i] = A[minpos];
         A[minpos] = temp;
      }
   }
}
```

Look at following array of unsorted integers. The working of selection sort is shown in Table 9.4 with the resultant array after each pass where the updated values of index variable *i* and minpos after each pass are indicated.

	Original unsorted array						
Elements	76	67	36	55	23	14	6
Index	0	1	2	3	4	5	6

Table 9.4 Selection sort

Index	0	1	2	3	4	5	6	*i*	minpos
	76	67	36	55	23	14	6	0	6
Pass 1	6	67	36	55	23	14	76	1	5
Pass 2	6	14	36	55	23	67	76	2	4
Pass 3	6	14	23	55	36	67	76	3	4
Pass 4	6	14	23	36	55	67	76	4	4
Pass 5	6	14	23	36	55	67	76	5	5
Sorted array	6	14	23	36	55	67	76		

The same can be done in reverse order also to arrange the elements. That is, first find the largest element in the structure. In the second step, swap the largest element with the element at the last position. Then, find the next largest element and swap with the element at the last but one position, and so on. Let us have look at one more example on the working of selection sort.

EXAMPLE 9.5 Figure 9.7 shows an unsorted array and the sorting process with the resultant array after each pass.

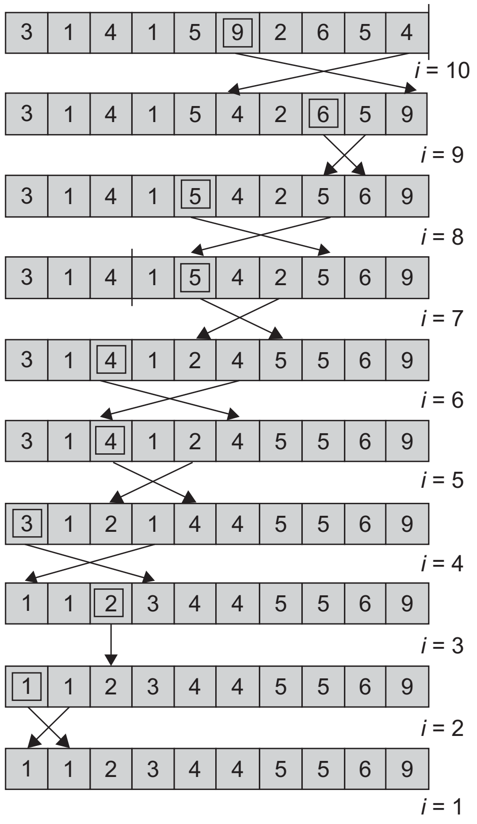


Fig. 9.7 Selection sort sample run

Analysis of Selection Sort

In Program Code 9.9, we can note that there are two loops, one nested within the other. During the first pass, $(n - 1)$ comparisons are made. In the second pass, $(n - 2)$ comparisons are made. In general, for the i^{th} pass, $(n - i)$ comparisons are required.

The total number of comparisons is as follows:

$$(n - 1) + (n - 2) + \ldots + 1 = n(n - 1)/2$$

Therefore, the number of comparisons for the selection sort is proportional to n^2, which means that it is $O(n^2)$. The different cases are as follows:

Average case: $O(n^2)$ Best case: $O(n^2)$ Worst case: $O(n^2)$

The maximum number of interchanges required is $(n - 1)$ as there is utmost one interchange required for each pass. However, the actual number of interchanges depends on the ordering of the original table, because if the smallest key is already at its proper place, the algorithm makes no interchanges.

The selection sort and insertion sort are more efficient than bubble sort. Selection sort is recommended for lists. When records are large, the keys are simple as the selection sort requires lesser swaps than the insertion sort and more comparisons than the insertion sort. If the records are small and the keys are difficult to compare, insertion sort is recommended.

9.3.6 Quick Sort

Quick sort is based on the divide-and-conquer strategy. This sort technique initially selects an element called as *pivot* that is near the middle of the list to be sorted, and then the items on either side are moved so that the elements on one side of pivot are smaller and on the other side are larger. Now, the pivot is at the right position with respect to the sorted sequence. These two steps, selecting the pivot and arranging the elements on either side of pivot, are now applied recursively to both the halves of the list till the list size reduces to one.

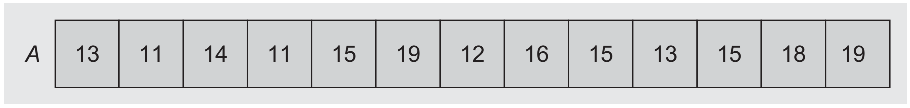
Quick sort is thus an in-place, divide-and-conquer-based, massively recursive sort technique. This technique reduces unnecessary swaps and moves the element at a great distance in one move.

To choose the pivot, there are several strategies. The popular way is considering the first element as the pivot.

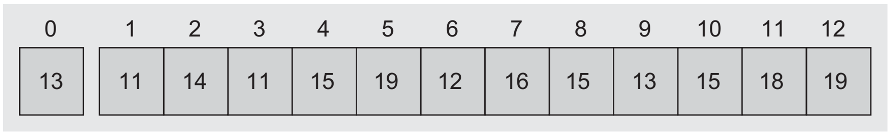
Thus, the recursive algorithm consists of four steps:

1. If the array size is 1, return immediately.
2. Pick an element in the array to serve as a ‘pivot’ (usually the left-most element in the list).
3. Partition the array into two parts—one with elements smaller than the pivot and the other with elements larger than the pivot by traversing from both the ends and performing swaps if needed.
4. Recursively repeat the algorithm for both partitions.

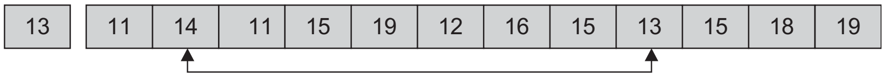
Let us consider an example. Let the list of numbers to be sorted be {13, 11, 14, 11, 15, 19, 12, 16, 15, 13, 15, 18, 19}. Now, the first element 13 becomes pivot. We need to place 13 at a proper location so that all elements to its left are smaller and the right are greater.

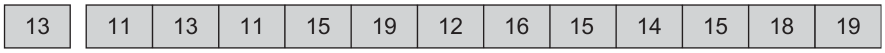


Initially, the array is pivoted about its first element *A*[pivot] = 13.

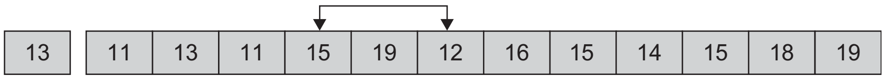


Let us first find the elements larger than the pivot, that is, 13. In addition, let us find the last element not larger than the pivot. These elements are in positions 2 and 9. Let us swap those.

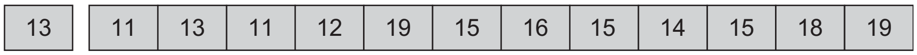




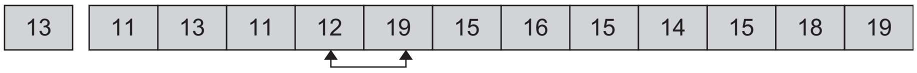
Let us again start scanning from both the directions.



The elements 12 and 15 are to be swapped to get the following sequence:



Let us repeat the steps to get the following sequence:



Here, the lower and upper bounds have crossed. So let us now swap the pivot-with element 12.

12	11	13	11	13	19	15	16	15	14	15	18	19

Here, we get two partitions as represented in the following sequence:

12	11	13	11

13

19	15	16	15	14	15	18	19

Recursively applying similar steps to each sub-list on the right and left side of the pivot, we get,

11	11	12	13

13

15	15	15	16	18	19	19

This is the final sorted array.
Algorithm 9.7 is written by assuming an array `A` with locations `A[Low]` to `A[High]` to be sorted.

ALGORITHM 9.7

```
Repeat process till low < high
1. Select pivot = A[Low], pivot location P = low
2. i = low and j = high;
3. Increment index i till A[i] >= pivot
4. Decrement index j till A[i] <= pivot
5. Swap A[i] with A[j]
6. Repeat steps 4, 5, 6 till i < j
7. if i < j
       Swap a[P] with a[j]
8. call Quicksort(low, j – 1)
9. call Quicksort(j + 1, high)
10. Stop
```

With the first seven steps of the process, the elements lesser than the key value are placed at the left side and the elements greater than the key value are placed at the right side of the key value.

Choice of Pivot We can choose any entry in the list as the pivot. The choice of the first entry as pivot is popular but often a poor choice. If the list is already sorted, then there will be no element less than the first element selected as pivot, and so one of the sub-lists will be empty.

Hence, we choose a pivot near the centre of the list, in the hope that our choice will position the list in such a manner that about half the elements will come on each side of the pivot.

The choice of the pivot near the centre is also arbitrary, and hence, it is not necessary that it will always divide the list into half. A good way to choose a pivot is to use a random number generator to choose the position of the next pivot in each of the activations of quick sort. Quick sort is illustrated in Program Code 9.10.

PROGRAM CODE 9.10

```
#define max 20
void read(int A[max], int n)
{
   int i;
   for(i = 0; i < n; i++)
      cin >> A[i];
}

void display(int A[max], int n)
{
   int i;
   for(i = 0; i < n; i++)
      cout << A[i];
}

void swap(int *x, int *y)
{
   int temp;
   temp = *x;
   *x = *y;
   *y = temp;
}

void qsort(int A[], int low, int high)
{
   int k;
   if(low < high)
   {
      K = partition(A, low, high);
      qsort(A, low, j - 1);
      qsort(A, j + 1, high);
   }
}

int partition(int A[],int low, int high)
{
   int pivot, i, j;
   pivot = A[low];
   j = high + 1; i = low;
   do
   {
      i++;
      while(A[i] < pivot && low <= high)
      do
      {
         j++;
      } while(pivot < A[j]);
      if(i < j)
        swap(A[i],A[j]);
   } while(i < j);
   A[low] = A[j];
   A[j] = pivot;
   return j;
}

main()
{
   int A[max], n;
   int i, choice;
   cout << "Enter number of numbers:";
   cin >> n;
   cout << "Enter numbers:";
   read(A, n);
   qsort(A, 0, n - 1);
   cout << "Sorted array is:";
   display(A, n);
}

/**********************Output************************
 Enter number of numbers: 7
 Enter numbers: 10     5     23     67     20     30     60
 Sorted array is: 5    10    20    23    30    60    67
******************************************************/
```

Analysis of Quick Sort

Now, let us see the efficiency of quick sort. On the first pass, every element in the array is compared to the pivot, so there are n comparisons. The array is then divided into two parts each of size (n/2). We assume that the array is divided into approximately one-half each time. For each of these sub-arrays, (n/2) comparisons are made and four sub-arrays of size (n/4) are formed. So at each level, the number of sub-arrays doubles. It will take $\log_2 n$ divisions if we are dividing the array approximately one-half each time. Therefore, quick sort is $O(n\log_2 n)$ on the average.

If the original array is sorted and array[left] is chosen as a pivot, then order of quick sort turns out to be $O(n^2)$. Therefore, when we choose array[left] as pivot, quick sort works best for files that are completely unsorted and worst for files that are completely sorted. In the case of nearly sorted arrays, choose a random element as a pivot value. The time required to sort the left sub-list and the right sub-lists where we assume that each has the size n/2 is as follows:

$$T(n) = c \times n + 2 \times T(n/2)$$

where c is a constant and $T(n/2)$ is the time required to sort the list of size n/2.

Similarly, the time required to sort the list of size n/2 is equal to the sum of the time required to place the key element at its proper position in the list of size n/2 and the time required to sort the left and right sub-lists each assumed to be of size n/4, $T(n/2)$. This turns out to be in the following form:

$$T(n/2) = c \times n/2 +\!- 2 \times T(n/4)$$

where $T(n/4)$ is the time required to sort the list of size n/4

$$\therefore \quad T(n/4) = c \times n/4 + 2 \times T(n/8)$$

This process continues, and finally we get $T(1) = 1$.

$\therefore \quad T(n) = c \times n + 2(c \times n(n/2) + 2T(n/4))$

$\therefore \quad T(n) = c \times n + c \times n + 4T(n/4)) = 2 \times c \times n + 9T(n/9) = 2 \times c \times n + 9(c \times (n/9) + 2T(n/8))$

$\therefore \quad T(n) = 2 \times c \times n + c \times n + 8T(n/8) = 3 \times c \times n + 8T(n/8)$

$\therefore \quad T(n) = (\log n) \times c \times n + nT(n/n) = (\log n) \times c \times n + nT(1) = n + n \times (\log n) \times c$

$\therefore \quad T(n) \; \alpha \; n\log(n)$

The average complexity of the quick sort algorithm is $O(n\log n)$. However, the worst case time complexity is $O(n^2)$.

9.3.7 Heap Sort

Heap sort is one of the fastest sorting algorithms, which achieves the speed of quick sort and merge sort. The advantages of heap sort are that it does not use recursion, and it is efficient for any data order. There is no worst case scenario in the case of heap sort. We shall discuss heap sort in detail in Chapter 12. Heap sort is a sorting technique that sorts a list of length n with $O(n\log 2(n))$ comparisons and movement of entries, even in the worst case.

Hence, it achieves the worst case bounds better than those of quick sort; and for the list, it is better than merge sort since it needs only a small and constant amount of space apart from the list being sorted. The steps for building a heap sort are as follows:

1. Build the heap tree.
2. Start delete heap operations storing each deleted element at the end of the heap array.

After performing step 2, the order of the elements will be opposite to the order in the heap tree. Hence, if we want the elements to be sorted in ascending order, we need to build the heap tree in descending order—the greatest element will have the highest priority.

Note that we use only one array, treating its parts differently:

1. When building the heap tree, a part of the array will be considered as the heap, and the rest will be the original array.
2. When sorting, a part of the array will be the heap, and the rest will be the sorted array.

Algorithm 9.8 provides the steps followed in sorting data using a heap.

ALGORITHM 9.8

```
1. Build a heap tree with a given set of data
2. Delete root node from heap
   Rebuild the heap after deletion
   Place the deleted node in the output
3. Continue with step 2 until the heap tree is empty
```

Program Code 9.11 illustrates Algorithm 9.8 in C++.

PROGRAM CODE 9.11

```
// reheapup operation is required when a new value is
inserted at the ith location
void reheapdown(int a[], int n, int i)
{
   int temp, j;
   while(2 * i + 1 < n)
   {
      j = 2 * i + 1;     // j index shows the left child of
      the node
      if(j + 1 < n && a[j + 1] > a[j])
      // finding max from left and right child
         j = j + 1;
         if(a[i] > a[j]) break;
         // if root > children then break
```

```
            else
            {
               // swap a[i] with a[j]
               temp = a[i];
               a[i] = a[j];
               a[j] = temp;
               i = j;
            }
         }       // end of while
      }

      Void Heap_Sort (int a[], int n)
      {
         // create heap
         int i, temp;
         for(i = (n - 1)/2; i >= 0; i--)
            reheapdown(a, n, i);
         // delete first value and swap it with last
         while(n > 0)
         {
            //swap first and last element
            temp = a[0];
            a[0] = a[n - 1];
            a[n - 1] = temp;
            n--;      // decrement count
            reheapdown(a, n, 0);
         }
      }

      void main()
      {
         int a[10], n, i;
         cout << "Enter N";
         cin >> n;
         cout << "Enter the elements";
         for(i = 0; i < n; i++)
            cin >> a[i];
         Heap_Sort(a, n);
         cout << "The sorted elements are";
         for(i = 0; i < n; i++)
      cout << a[i];
}
/*************** Output *****************************
Enter N: 12
Enter the elements: 44   33   11   55   77   90   40
60   99   22   88   66
The sorted elements are 11  22  33  40  44  55  60  66
77  88  90  99
******************************************************
```

In each pass of the `while` loop in the function `reheapdown(a,n,0)`, the position i is double; hence, the number of passes cannot exceed $\log(n/i)$. Therefore, the computation time is of the order $O(\log n/i)$.
For building the heap, the `reheapdown` procedure is called $n/2$ times. Hence, the total number of iterations will be as follows:

$$\log(n) + \log(n/2) + ... + \log(n/n/2)$$
$$= \sum_{i=1}^{n/2} \log(n/i)$$
$$= n/21\text{og}(n) - \log(n/2)$$

This turns out to be some constant times n.

If we analyse the processing phase, a heap of size i requires $O(\log_2 i)$ comparisons and interchanges even in the worst case.

Therefore, the required number of comparisons and interchanges, on the average, is

$$\sum_{i=2}^{n} \log_2 i + \frac{1}{2}\sum_{i=2}^{n} \log_2 i$$

This is $(n - 1) \log_2 n$. The worst case is quite comparable to the average case, and the number of comparisons and interchanges in this case is given by the following expression:

$$2(n-1)\ \log_2 n \left(\sum_{i=2}^{n} \log_2 i + \sum_{i=2}^{n} \log_2 i \right)$$

Therefore, heap sort is definitely $O(n\log_2 n)$.

The time complexity analysis of heap sort is as follows:

1. Best case: O(*n*log*n*)
2. Average case: O(*n*log*n*)
3. Worst case: O(*n*log*n*)

9.3.8 Shell sort

The technique used by shell sort (named after its inventor Donald Shell) is interesting. The algorithm is easy to program, and it runs fairly quickly. Its analysis, however, is very difficult. Shell sort is a sorting algorithm, which is an improved version of insertion sort. It makes repetitive use of insertion sort.

In this technique, the elements at a fixed distance are compared. Later, this distance is decremented in the next pass by some value and again the comparisons are made. The fixed distance is called as *gap*. The algorithm begins with the initial gap as *n*/2, where *n* is the total number of elements to be sorted. Later, in the next pass, the gap is modified as *n*/4, *n*/8, and so on till it becomes 1. When gap is 1, it becomes an ordinary insertion sort (Program Code 9.12).

PROGRAM CODE 9.12

```
void shell_sort(int A[], int n)
{
   int temp, gap, i;
   int swapped;
   gap = n/2;
   do
   {
      do
      {
         swapped = 0;
         for(i = 0; i < n - gap; i++)
            if(A[i] > A[i + gap])
            {
               temp = A[i];
               A[i] = A[i + gap];
               A[i + gap] = temp;
               swapped = 1;
            }
      } while(swapped == 1);
   } while((gap = gap/2) >= 1);
}
```

Figure 9.8 illustrates the sample run using shell sort.

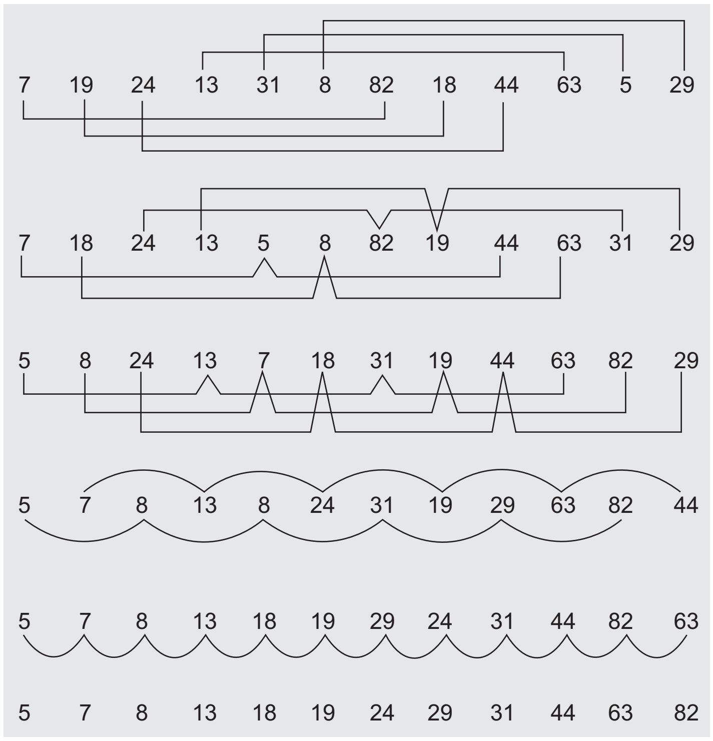


Fig. 9.8 Shell sort sample run

The time complexity of shell sort lies between O($n\log^2 n$) and O($n^{1.5}$).

9.3.9 Bucket Sort

Bucket sort is possibly the simplest distribution sorting algorithm. In bucket sort, initially, a fixed number of buckets are selected. For example, suppose that we are sorting elements from the set of integers in the interval [0, $m - 1$]. The bucket sort uses m buckets or counters. The ith counter/bucket keeps track of the number of occurrences of the ith element of the list. Figure 9.9 illustrates how this is done for $m = 9$.

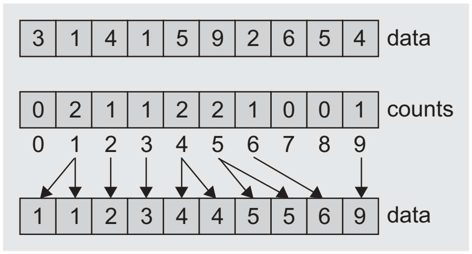


Fig. 9.9 Bucket sort

In Fig. 9.9, the buckets are assumed to be {0, 1, ..., 9}. Therefore, 10 counters are required to keep track of the number of 0s, number of 1s, and so on till 9. A single pass through the data counts the frequency (count indicating number of times the element occurs) of each element. Once the counts have been determined, the sorted sequence is easily obtained. Here, each bucket need not to be sorted again as equal numbers lie in the same bucket. Though this techniques seems to be very simple, the number of buckets required depends on the size of the list to be sorted. Program Code 9.13 illustrates the `BucketSort()` function.

PROGRAM CODE 9.13

```
void BucketSort(int A[], int n)
{
   int i, j;
   int bucket[max];
   //counters/buckets can store numbers maximum 20
   for(i = 0; i < max; i++)
      bucket[i] = 0;
   for(j = 0; j < n; j++)
   {
      ++bucket[A[j]];
      // counting number for each bucket
   }
   for(i = 0, j = 0; i < max; i++)
      for(;bucket[i] > 0; --bucket[i])
      { A[j] = i; j++; }
}
/****************** Output ***************************
 Enter number of numbers: 7
 Enter numbers value < 20: 12 15 07 05 12 09 07
 bucket[12]=1
 bucket[15]=1
 bucket[7]=1
 bucket[5]=1
 bucket[12]=2
 bucket[9]=1
 bucket[7]=2
 Sorted array is 5 7 7 9 12 12 15
*************************************************/
```

9.3.10 Radix Sort

Radix sort is a generalization of bucket sort and works in three steps:

1. Distribute all elements into *m* buckets. Here *m* is a suitable integer, for example, to sort decimal numbers with radix 10. We take 10 buckets numbered as 0, 1, 2, …, 9. For sorting strings, we may need 26 buckets, and so on.
2. Sort each bucket individually.
3. Finally, combine all buckets.

To sort each bucket, we may use any of the other sorting techniques or radix sort recursively. To use radix sort recursively, we need more than one pass depending upon the range of numbers to be sorted. For sorting single digit number, we need only one pass, which is discussed in Section 9.3.9. For sorting numbers with two digits mean ranging between 00 and 99, we would need two passes; for the range from 0 to 999, we would need three passes, and so on.

Let us consider a set of two digit number to be sorted. In the first pass, we would distribute numbers in buckets 0 to 9 using the most significant digit (MSD). Now, in the bucket 0, we have all numbers with MSD 0, and all numbers with MSD 1 are in bucket 1, and so on. In the second pass, the numbers in each bucket would be sorted based on the second most significant digit. The buckets are combined to yield a sorted list.

Let us consider a set of numbers to be sorted {07, 10, 99, 02, 80, 14, 25, 63, 88, 33, 11, 72, 68, 39, 21, 50}. Table 9.5 illustrates a sample run for this list using radix sort.

Table 9.5 Sample run for radix sort

Pass 1	
Bucket	**Numbers**
0	02, 07
1	10, 11, 14
2	21, 25
3	39, 33
4	
5	50
6	68, 63
7	72
8	80, 88
9	99

(Continued)

Table 9.5 (Continued)

Pass 2 (only non-empty buckets are shown for distribution based on the second significant digit)		
Bucket	**Numbers**	
	Buckets based on second significant digit	**Numbers**
0	2	02
	7	07
1	0	10
	1	11
	4	14
2	1	21
	5	25
3	3	33
	9	39
4		
5	0	50
6	3	63
	8	68
7	2	72
8	0	80
	8	88
9	9	99

The amount of space needed by a bucket sort depends on how the buckets are stored. If every bucket is to consist of a set of sequential locations (e.g., an array), then each must be allocated enough space to hold the maximum number of elements that might belong in one bucket, and that is *n*. As the number of buckets increases, the speed of the algorithm increases but so does the amount of space used. Linked lists would be better, which would need the space for *n* elements plus links and a list head for each bucket. Program Code 9.14 illustrates this.

PROGRAM CODE 9.14

```
#define max 20
void radixsort(int A[max], int n)
{
   int i, j, temp;
   int bucket[10][15];
   int count[10], digit, k, p, x, nopass, maxval;
   maxval = A[0];
```

```
    for(i = 0; i < n; i++)
       if(maxval < A[i]) maxval = A[i];
    nopass = 0;
    while(maxval != 0)
    {
       maxval = maxval/10;
       nopass++;
    }
    p = 0;
    do
    {
       x = 1;
       for(i = 0; i < 10; i++) count[i] = 0;
       for(i = 0; i < n ; i++)
       {
          digit = (A[i]/x) % 10;
          bucket[digit]*[count[digit]] = A[i];
          // setting up bucket
          count[digit]++;
        }
        k = 0;
        for(i = 0; i < 10; i++)
        {
           if(count[i] != 0)
           {
              for(j = 0; j < count[i]; j++)
              A[k++] = bucket[i][j];
           }
        }
        cout << "Pass" << p;
        display(A, n);
        x = x * 10;
        p++;
     }while(p < nopass);
}
```

9.3.11 File Sort

The sorting algorithms discussed so far use array to hold and process the data to be sorted that resides in memory. Quite often, voluminous files, such as a master file for all the employees in a large corporation, must exist on external storage devices because of their size. These on-line storage devices, such as tapes and disks, carry with them specific software and hardware considerations relating to the access of stored data.

One of the solutions is to bring only portions of large files into the main memory and sort them. The portion of a file that can reside in main memory is a *block*. The sorted block can be sent back to the external storage medium and the next block can be brought in. Finally, the partially sorted blocks must be merged into a completely sorted file.

Because of the nature of secondary storage devices, bringing a block of data items into the main memory takes a longer time than processing it. For instance, it takes time to position the read–write head over the appropriate track of a disk, and more time for the disk to rotate to bring the correct block to the read–write head. An average input/output operation to and from an auxiliary storage device (not bucketing processing in memory) may take as 200 milliseconds. When we design sort algorithms for files on external media, we must consider this time delay.

There are numerous algorithms used to perform sorts external to the computer's main memory. Among the many external sort methods, the polyphase sort is more efficient in terms of speed and utilization of resources. However, it is more complicated, and therefore, in some situations, the other algorithms could be more applicable. In practice, internal sorts are already supplementing these sorting methods. Thus, a number of records from each tape would be read into the main memory and sorted using an internal sort and then output to the tape rather than one record at a time, as was the case initially. Let us study the merge sort method. Merge sort technique is commonly used for external sort and is suitable for internal sort too.

9.3.12 Merge Sort

The most common algorithm used in external sorting is the merge sort. Merging is the process of combining two or more sorted files into the third sorted file. We can use a technique of merging two sorted lists. Divide and conquer is a general algorithm design paradigm that is used for merge sort. Merge sort has three steps to sort an input sequence S with n elements:

1. Divide—partition S into two sequences $S1$ and $S2$ of about $n/2$ elements each
2. Recur—recursively sort $S1$ and $S2$
3. Conquer—merge $S1$ and $S2$ into a sorted sequence

A file (or sub-file) is divided into two files, $f1$ and $f2$. These two files are then compared, one pair of records at a time, and merged. This is done by writing them on two separate new files $M1$ and $M2$. Elements that do not pair off are simply rewritten into the new files. The records in $M1$ and $M2$ are now blocked with two records in each block. The two blocks (i.e., four records), one from $M1$ and one from $M2$, are merged and written onto the original files $f1$ and $f2$. The length of the blocks in each of $f1$ and $f2$ is now increased to four; the merge process is applied again, and the new files are written to $M1$ and $M2$. The process is continued until one of the two files, $f1$ or $f2$, is empty. Merge sort is a divide-and-conquer algorithm. Note that the function `mergesort()` calls itself recursively. Algorithm 9.9 is derived based on the steps discussed.

ALGORITHM 9.9

```
List mergesort(list L, int n)
{
   if(n == 1)
      return(L);
   else
   {
      split L into two halves L1 and L2;
      return(merge(mergesort(L1, n/2), (mergesort(L2, n/2))
   }
}
```

Time Complexity

Let T(n) be the running time of merge sort on an input list of size n. Then,

$$T(n) < C_1 \text{ (if } n = 1\text{), where } C_1 \text{ is a constant and}$$
$$T(n) < 2T(n/2) + C_2 n$$

Here, $2T(n/2)$ is for two recursive calls, and C_2n is the cost of merging the two sorted lists.

Now, by the substitution method,

$$T(n) = 2T(n/2) + C_2 n$$

If $n = 2^k$ for some k, it can be shown that after k steps

$$T(n) = 2^k T(n/2^k) + C_2 C_2{}^k$$

Hence, for $n = 2^k$

$$T(n) = n\log_2 n$$

That is, $T(n) = O(n\log n)$

Let us implement the merge technique for two arrays instead of working on files. Let us write a routine that accepts two sorted arrays, `A` and `B` containing elements $n1$ and $n2$, respectively and merges them into a third array `C` containing $n3$ elements. Here, the array `A` is from `low` to `mid`, array `B` is from `mid + 1` to `high`, and array `C` gives the merging of `A` and `B` This is shown in Program Code 9.15.

PROGRAM CODE 9.15

```
void merge (int A[],int low, int high, int mid)
{
   int i, j, k, C[max];
   i = low;      // index for first part
   j = mid + 1;      // index for second part
   k = 0;      // index for array C
   while((i <= mid) && (j <= high))
   // merge arrays A & B in array C
   {
      if(A[i] < A[j])
         C[k] = A[i++];
      else
         C[k] = A[j++];
      k++;
   }
   while(i <= mid)
      C[k++] = A[i++];
   while(j <= high)
      C[k++]=A[j++];
   for(i = low, j = 0; i < = high; i++, j++)
    // copy array C contents back to array A
   {
      A[i] = C[j];
   }
}

void MergeSort(int A[], int low, int high)
{
   int mid;
   if(low < high)
   {
      mid = (low + high)/2;
      MergeSort(A, low, mid);
      MergeSort(A, mid + 1, high);
      merge(A, low, high, mid);
   }
}
```

When merge sort is used for files as described in Program Code 9.15, each merge operation requires reading and writing of two files, both of which are on the average about $n/2$ records long. Thus, the total number of blocks read or written in a merge operation is approximately $2n/c$, where c is the number of records in a block. The number of blocks accessed for the whole operation is $O((n(\log_2 n))/c)$, which amounts to $O(\log_2 n)$ passing through the entire original file. This is a considerable improvement over the $O(n)$ passes needed in the preceding algorithms.

9.4 MULTIWAY MERGE AND POLYPHASE MERGE

We have already studied external sorting in Section 9.3.11. It broadly works in the following three steps:

1. Split the data into small sets that fit into main memory.
2. Now sort each of the subsets with a conventional sorting algorithm.
3. Finally merge those so-called runs and get a complete sorted data set.

This merging procedure can obviously be applied to more than two runs at every time and it is called *n*-way merge or multiway merge. The sophisticated multiway merge algorithms include polyphase merge.

A non-balanced *k*-way merge that reduces the number of output files needed by reusing the emptied input file or device as one of the output devices is called *polyphase merge*. This is most efficient if the number of output runs in each output file is different. Combining the run creation and run merging calculations together, we find that the overall complexity is $O(n\log_2 n)$. The repeated merging is referred to as polyphase merging. Polyphase merge sorts are ideal for sorting and merging large files. Two pairs of input and output files are opened as file streams. At the end of each iteration, input files are deleted; output files are closed and reopened as input files. The use of file streams makes it possible to sort and merge files that cannot be loaded into the computer's main memory. It is a method of merging, where the keys are kept in more than one backup store or file. Items are merged from the source files to another file. Whenever one of the source files is exhausted, it immediately becomes the destination of the merge operations from the non-exhausted and earlier destination files. When there is only one file left, the process stops.

9.4.1 Comparison of Ordinary Merge Sort and Polyphase Sort

Typically, a merge sort splits items into sorted runs and then recursively merges each run into larger runs. When there is only one run left, it is termed as the sorted result. An ordinary merge sort could use four working files organized as a pair of input files and a pair of output files. At each iteration, two input files are read. The odd-numbered runs of the two input files are merged to the first output file, and the even-numbered runs are merged to the second output file. When the input is exhausted, the new output files are used as the input for the next iteration. The number of runs decreases by a factor of 2 at each iteration. At each iteration, the same level/phase of merge occurs—a file is either completely read or completely written during the iteration.

If the four files were on four separate tape drives, an ordinary merge sort would provide some interesting details. In the first iteration, only one input drive is used and the other input file is empty. In subsequent iterations, each input drive runs at half speed, while one output drive runs at full speed and the second output drive stands idle waiting for the next run. The situation is even worse when six tape drives are used, out of which at least two stand idle. It would be ideal if the idle drives could be put to more use.

Perfect Three-file Polyphase Merge Sort

It is easiest to look at the polyphase merge from its end conditions and working backwards. At the start of each iteration, there are two input files and one output file. At the end of the iteration, one input file is completely consumed and becomes the output file for the next iteration. The current output file will become an input file for the next iteration. The remaining files (just one in the three-file case) are only partially consumed, and their remaining runs are the input for the next iteration.

In the following instance, File 1 is just emptied, and it becomes the new output file. One run is left on each input tape, and merging those runs together will make the sorted file.

```
File 1 (out):                                <1 run> *       (the sorted file)
File 2 (in): ... | <1 run> *      -->    ... <1 run> | *       (consumed)
File 3 (in):     | <1 run> *                 <1 run> | *       (consumed)
```

Here,

... denotes the possible runs that have already been read
| marks the read pointer of the file
* marks end of file

In the previous iteration, we read from Files 1 and 2. One run is merged from both files before File 1 goes empty. Notice that File 2 is not completely consumed; it has one run left to match the final merge.

```
File 1 (in): ... | <1 run> *            ... <1 run> | *
File 2 (in):     | <2 run> *      -->        <1 run> | <1 run> *
File 3 (out):                                <1 run> *
```

Stepping back another iteration, two runs are merged from Files 1 and 3 before File 3 goes empty.

```
File 1 (in ):     | <3 run> *            ... <2 run> | <1 run> *
File 2 (out):                     -->        <2 run> *
File 3 (in ): ... | <2 run> *                <2 run> | *
```

Moving to the previous iteration, three runs are merged from Files 2 and 3 before File 2 goes empty.

```
File 1 (out):                                <3 run> *
File 2 (in ): ... | <3 run> *     -->    ... <3 run> | *
File 3 (in ):     | <5 run> *                <3 run> | <2 run> *
```

Moving further back, five runs are merged from Files 1 and 2 before File 1 goes empty.

```
File 1 (in ): ... | <5 run> *            ... <5 run> | *
File 2 (in ):     | <8 run> *     -->        <5 run> | <3 run> *
File 3 (out):                                <5 run> *
```

The number of runs merged working backwards—1, 2, 3, 5, …,—reveals a Fibonacci sequence. For everything to work out right, the initial file to be sorted must be distributed to the proper input files, and each input file must have the correct number of runs on it. In the example, this would mean that an input file with 13 runs writes 5 runs to File 1 and 8 runs to File 2.

In practice, the input file might not have a Fibonacci number of runs (which would not be known until after the file has been read). The fix is to pad the input files with dummy runs to obtain the required Fibonacci sequence.

For comparison, the ordinary merge sort combines 16 runs in 4 passes using 4 files. The polyphase merge combines 13 runs in 5 passes using only 3 files. Alternatively, a polyphase merge combines 17 runs in 4 passes using 4 files (sequence: 1, 1, 1, 3, 5, 9, 17, 31, 57, ...).

An iteration (or pass) in an ordinary merge sort involves reading and writing the entire file. An iteration in a polyphase sort does not read or write the entire file, so a typical polyphase iteration takes lesser time than a merge sort iteration.

Two-phase, Multiway Merge Sort

The basic idea behind the two-phase, multiway merge sort is simple, and is described as follows:

Phase 1: Repeat the following until all data items have been visited once.

1. Fill a designated region R of main memory with as many data items as it can hold.
2. Sort the data items in R using an internal sort.
3. Write the sorted data items back to new blocks on disk, which yields a sorted 'sub-list' of the original data items.

Phase 2: At the conclusion of the following steps, a sorted file will emerge.

1. Read a block from each of the sub-lists from Phase 1 into a main memory buffer; in addition, set aside an output buffer.
2. Merge the sub-lists into a sorted file by repeating the following steps as often as necessary.
3. Fill the output buffer by repeatedly selecting the smallest (or the largest, depending on the sorting order) remaining data item in the buffers from the sorted sub-lists. If all of the items in a sub-list buffer have been examined, read the next block for that sub-list (if there is no such block, then do not examine the associated buffer anymore).
4. Write the output buffer to disk and reinitialize the buffer for the next output block.

9.5 COMPARISON OF ALL SORTING METHODS

Table 9.6 compares and comments on the sorting methods discussed in this chapter.

Table 9.6 Comparison of sorting techniques

Sorting method	Technique in brief	Best case	Worst case	Memory requirement	Is stable?	Pros	Cons
Bubble sort	Repeatedly stepping through the list to be sorted, comparing each pair of adjacent items and swapping them if they are in the wrong order	O(n^2)	O(n^2)	No extra space needed	Yes	1. A simple and easy method 2. Efficient for small lists $n > 100$	Highly inefficient for large data
Selection sort	Finds the minimum value in the list and then swaps it with the value in the first position, repeats these steps for the remainder of the list (starting at the second position and advancing each time)	O(n^2)	O(n^2)	No extra space needed	No	1. Recommended for small files 2. Good for partially sorted data	Inefficient for large lists
Insertion sort	Every repetition of insertion sort removes an element from the input data, inserts it into the correct position in the already sorted list until no input elements remain. The choice of which element to remove from the input is arbitrary and can be made using almost any choice of algorithm	O(n)	O(n^2)	No extra space needed	Yes	1. Relatively simple and easy to implement 2. Good for almost sorted data	Inefficient for large lists

(Continued)

Table 9.6 (Continued)

Quick sort	Picks an element, called a pivot, from the list. Reorders the list so that all elements with values less than the pivot come before the pivot, whereas all elements with values greater than the pivot come after it (equal values can go either way). After this partitioning, the pivot is in its final position. This is called the partition operation. Recursively sorts the sub-list of the lesser elements and the sub-list of the greater elements.	$O(n\log_2 n)$	$O(n^2)$	No extra space needed	No	1. Extremely fast 2. Inherently recursive	Very complex algorithm
Shell sort	It is a generalization of insertion sort, which exploits the fact that insertion sort works efficiently on input that is already almost sorted. It improves on insertion sort by allowing the comparison and exchange of elements that are far apart. The last step of shell sort is a plain insertion sort, but by then, the array of data is guaranteed to be almost sorted	$O(n^{1.5})$	$O(n\log^2 n)$	No extra space needed	No	1. It is faster than a quick sort for small arrays 2. Its speed and simplicity makes it a good choice in practice	Slower for sufficiently big arrays

(Continued)

Table 9.6 (Continued)

Sorting method	Technique in brief	Best case	Worst case	Memory requirement	Is stable?	Pros	Cons
Radix sort (most significant digit)	Numbers are placed at proper locations by processing individual digits and by comparing individual digits that share the same significant position.	O(*n*)	O(*n*)	Extra space proportional to *n* is needed	Yes	1. Radix sort is very simple and fast 2. In-Place, recursive, and one of the fastest sorting algorithms for numbers or strings of letters	Radix sort can also take more space than other sorting algorithms since in addition to the array that will be sorted, there needs to be a sub-list for each of the possible digits or letters
Merge sort	If the list is of length 0 or 1, then it is already sorted. Otherwise, the algorithm divides the unsorted list into two sub-lists of about half the size Then, it sorts each sub-list recursively by reapplying the merge sort and then merges the two sub-lists back into one sorted list.	$O(n\log_2 n)$	$O(n\log_2 n)$	Extra space proportional to *n* is needed	Yes	1. Good for external file sorting 2.Can be applied to files of any size	1. It requires twice the memory of the heap sort because of the second array used to store the sorted list. 2. It is recursive, which can make it a bad choice for applications that run on machines with limited memory
Heap sort	Heap sort begins by building a heap out of the data set, and then removing the largest item and placing it at the end of the partially sorted array. After removing the largest item, it reconstructs the heap, removes the largest remaining item, and places it in the next open position from the end of the partially sorted array. This is repeated until there are no items left in the heap and the sorted array is full	$O(n\log_2 n)$	$O(n\log_2 n)$	No extra space needed	No	1. Advantageous as it does not use recursion and that heap sort works just as fast for any data order. That is, there is basically no worst case scenario 2. Heaps work well for small tables and the tables where changes are infrequent	Do not work well for most large tables

*n is the number of data items to be sorted.