Chapter 02
Searching and Sorting Techniques

1. Linear search, Binary search,

Searching:

Searching means to find whether a particular value is present in an array or not. If the value is present in the array, then searching is said to be successful and the searching process gives the location of that value in the array. However, if the value is not present in the array, the searching process displays an appropriate message and in this case searching is said to be unsuccessful. There are two popular methods for searching the array elements: *linear search* and *binary search*.

The algorithm that should be used depends entirely on how the values are organized in the array. For example, if the elements of the array are arranged in ascending order, then binary search should be used, as it is more efficient for sorted lists in terms of complexity.

Linear search:

Linear search, also called as sequential search, is a very simple method used for searching an array for a particular value. It works by comparing the value to be searched with every element of the array one by one in a sequence until a match is found. Linear search is mostly used to search an unordered list of elements (array in which data elements are not sorted). For example, if an array A[] is declared and initialized as,

```
int A[] = {10, 8, 2, 7, 3, 4, 9, 1, 6, 5};
```

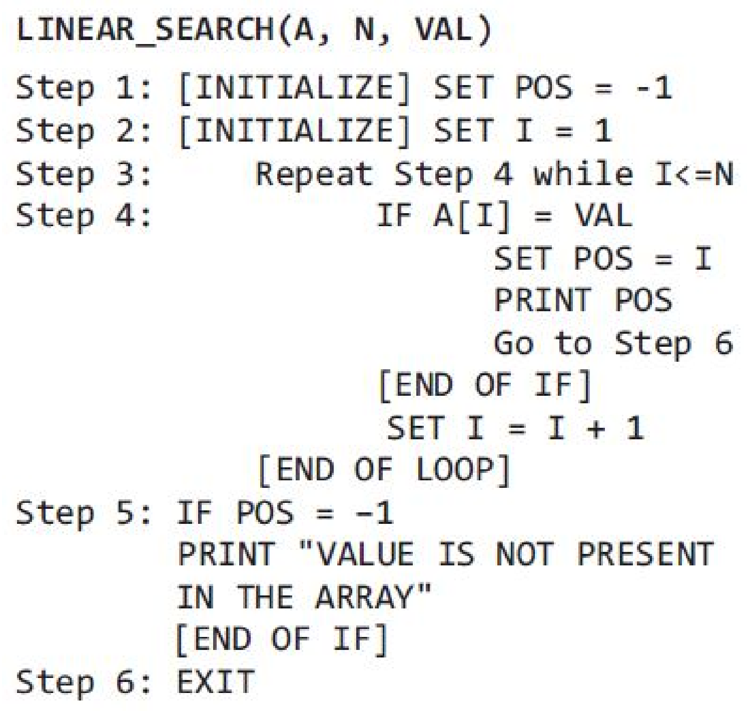
and the value to be searched is VAL = 7, then searching means to find whether the value ‘7’ is present in the array or not. If yes, then it returns the position of its occurrence. Here, POS = 3 (index starting from 0).

Algorithm: Linear Search

Input: A: List of Element

VAL: Search Element

Output: POS



```
LINEAR_SEARCH(A, N, VAL)
Step 1: [INITIALIZE] SET POS = -1
Step 2: [INITIALIZE] SET I = 1
Step 3:     Repeat Step 4 while I<=N
Step 4:             IF A[I] = VAL
                        SET POS = I
                        PRINT POS
                        Go to Step 6
                    [END OF IF]
                    SET I = I + 1
            [END OF LOOP]
Step 5: IF POS = -1
        PRINT "VALUE IS NOT PRESENT
        IN THE ARRAY"
        [END OF IF]
Step 6: EXIT
```

In Steps 1 and 2 of the algorithm, we initialize the value of POS and I. In Step 3, a while loop is executed that would be executed till I is less than N (total number of elements in the array). In Step 4, a check is made to see if a match is found between the current array element and VAL. If a match is found, then the position of the array element is printed, else the value of I is incremented to match the next element with VAL. However, if all the array elements have been compared with VAL and no match is found, then it means that VAL is not present in the array.

Complexity of Linear Search Algorithm:

Linear search executes in O(n) time where n is the number of elements in the array. Obviously, the best case of linear search is when VAL is equal to the first element of the array. In this case, only one comparison will be made. Likewise, the worst case will happen when either VAL is not present in the array or it is equal to the last element of the array. In both the cases, n comparisons will have to be made. However, the performance of the linear search algorithm can be improved by using a sorted array.

Write a program to search an element in an array using the linear search technique.

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

#define size 20 // Added so the size of the array can be altered more easily

int main(int argc, char *argv[])
{
	int arr[size], num, i, n, found = 0, pos = -1;
	printf("\n Enter the number of elements in the array : ");
	scanf("%d", &n);
	printf("\n Enter the elements: ");
	for(i=0;i<n;i++)
	{
		scanf("%d", &arr[i]);
	}
	printf("\n Enter the number that has to be searched : ");
	scanf("%d", &num);
	for(i=0;i<n;i++)
	{
		if(arr[i] == num)
		{
			found =1;
			pos=i;
			printf("\n %d is found in the array at position= %d", num,i+1);

			/* +1 added in line 23 so that it would display the number in
			the first place in the array as in position 1 instead of 0 */

			break;
		}
	}
	if (found == 0)
		printf("\n %d does not exist in the array", num);
	return 0;
}
```

Binary Search:

Binary search is a searching algorithm that works efficiently with a sorted list. The mechanism of binary search can be better understood by an analogy of a telephone directory. When we are searching for a particular name in a directory, we first open the directory from the middle and then decide whether to look for the name in the first part of the directory or in the second part of the directory. Again, we open some page in the middle and the whole process is repeated until we finally find the right name.

Take another analogy. How do we find words in a dictionary? We first open the dictionary somewhere in the middle. Then, we compare the first word on that page with the desired word whose meaning we are looking for. If the desired word comes before the word on the page, we look in the first half of the dictionary, else we look in the second half. Again, we open a page in the first half of the dictionary and compare the first word on that page with the desired word and repeat the same procedure until we finally get the word. The same mechanism is applied in the binary search. Now, let us consider how this mechanism is applied to search for a value in a sorted array.

Consider an array A[] that is declared and initialized as

```
int A[] = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
```

and the value to be searched is VAL = 9. The algorithm will proceed in the following manner.

BEG = 0, END = 10, MID = (0 + 10)/2 = 5

Now, VAL = 9 and A[MID] = A[5] = 5

A[5] is less than VAL, therefore, we now search for the value in the second half of the array.

So, we change the values of BEG and MID.

Now, BEG = MID + 1 = 6, END = 10, MID = (6 + 10)/2 =16/2 = 8

VAL = 9 and A[MID] = A[8] = 8

A[8] is less than VAL, therefore, we now search for the value in the second half of the segment.

So, again we change the values of BEG and MID.

Now, BEG = MID + 1 = 9, END = 10, MID = (9 + 10)/2 = 9

Now, VAL = 9 and A[MID] = 9.

In this algorithm, we see that BEG and END are the beginning and ending positions of the segment that we are looking to search for the element. MID is calculated as (BEG + END)/2. Initially, BEG =lower_bound and END = upper_bound. The algorithm will terminate when A[MID] = VAL. When the algorithm ends, we will set POS = MID. POS is the position at which the value is present in the array.

However, if VAL is not equal to A[MID], then the values of BEG, END, and MID will be changed depending on whether VAL is smaller or greater than A[MID].

(a) If VAL < A[MID], then VAL will be present in the left segment of the array. So, the value of END will be changed as END = MID – 1.

(b) If VAL > A[MID], then VAL will be present in the right segment of the array. So, the value of BEG will be changed as BEG = MID + 1.

Finally, if VAL is not present in the array, then eventually, END will be less than BEG. When this happens, the algorithm will terminate and the search will be unsuccessful.

Algorithm: BINARY_SEARCH
Input: A: Element List
VAL: Search element
Output: POS

```
BINARY_SEARCH(A, lower_bound, upper_bound, VAL)
Step 1: [INITIALIZE] SET BEG = lower_bound
          END = upper_bound, POS = - 1
Step 2: Repeat Steps 3 and 4 while BEG <= END
Step 3:           SET MID = (BEG + END)/2
Step 4:           IF A[MID] = VAL
                        SET POS = MID
                        PRINT POS
                        Go to Step 6
                  ELSE IF A[MID] > VAL
                        SET END = MID - 1
                  ELSE
                        SET BEG = MID + 1
                  [END OF IF]
        [END OF LOOP]
Step 5: IF POS = -1
            PRINT "VALUE IS NOT PRESENT IN THE ARRAY"
        [END OF IF]
Step 6: EXIT
```

In Step 1, we initialize the value of variables, BEG, END, and POS. In Step 2, a while loop is executed until BEG is less than or equal to END. In Step 3, the value of MID is calculated. In Step 4, we check if the array value at MID is equal to VAL (item to be searched in the array). If a match is found, then the value of POS is printed and the algorithm exits. However, if a match is not found, and if the value of A[MID] is greater than VAL, the value of END is modified, otherwise if A[MID] is greater than VAL, then the value of BEG is altered. In Step 5, if the value of POS = –1, then VAL is not present in the array and an appropriate message is printed on the screen before the algorithm exits.

Complexity of Binary Search Algorithm:

The complexity of the binary search algorithm can be expressed as f(n), where n is the number of elements in the array. The complexity of the algorithm is calculated depending on the number of comparisons that are made. In the binary search algorithm, we see that with each comparison, the size of the segment where search has to be made is reduced to half. Thus, we can say that, in order to locate a particular value in the array, the total number of comparisons that will be made is given as $2^{f(n)} > n$ or $f(n) = \log_2 n$

Write a program to search an element in an array using binary search.

```
#include <stdio.h>
#define size 10 // Added to make changing size of array easier

int smallest(int arr[], int k, int n); // Added to sort array

void selection_sort(int arr[], int n); // Added to sort array

int main(int argc, char *argv[])
{
        int arr[size], num, i, n, beg, end, mid, found=0;
        printf("\n Enter the number of elements in the array: ");
        scanf("%d", &n);
        printf("\n Enter the elements: ");
        for(i=0;i<n;i++)
        {
                scanf("%d", &arr[i]);
        }
        selection_sort(arr, n); // Added to sort the array
        printf("\n The sorted array is: \n");
        for(i=0;i<n;i++)
                printf(" %d\t", arr[i]);
        printf("\n\n Enter the number that has to be searched: ");
        scanf("%d", &num);
        beg = 0, end = n-1;
        while(beg<=end)
        {
                mid = (beg + end)/2;
                if (arr[mid] == num)
                {
                        printf("\n %d is present in the array at position %d", num, mid+1);
                        found =1;
                        break;
                }
                else if (arr[mid]>num)
                        end = mid-1;
                else
                        beg = mid+1;
        }
        if (beg > end && found == 0)
                printf("\n %d does not exist in the array", num);
        return 0;
}

int smallest(int arr[], int k, int n)
{
        int pos = k, small=arr[k], i;
        for(i=k+1;i<n;i++)
        {
                if(arr[i]< small)
                {
                        small = arr[i];
                        pos = i;
                }
        }
        return pos;
}


void selection_sort(int arr[],int n)
{
        int k, pos, temp;
        for(k=0;k<n;k++)
        {
                pos = smallest(arr, k, n);
                temp = arr[k];
                arr[k] = arr[pos];
                arr[pos] = temp;
        }
}
```

2. Hashing – Definition, hash functions, Collision:

The elements are not stored according to the *value* of the key. So in this case, we need a way to convert a five-digit key number to a two-digit array index. We need a function, which will do the transformation. In this case, we will use the term *hash table* for an array and the function that will carry out the transformation will be called a *hash function.*

HASH TABLEs:

Hash table is a data structure in which keys are mapped to array positions by a hash function. In the example discussed here we will use a hash function that extracts the last two digits of the key. Therefore, we map the keys to array locations or array indices. A value stored in a hash table can be searched in O(1) time by using a hash function which generates an address from the key (by producing the index of the array where the value is stored).

Figure 15.3 shows a direct correspondence between the keys and the indices of the array. This concept is useful when the total universe of keys is small and when most of the keys are actually used from the whole set of keys. This is equivalent to our first example, where there are 100 keys for 100 employees. However, when the set K of keys that are actually used is smaller than the universe of keys (U), a hash table consumes less storage space. The storage requirement for a hash table is O(k), where k is the number of keys actually used.

In a hash table, an element with key k is stored at index h(k) and not k. It means a hash function h is used to calculate the index at which the element with key k will be stored. This process of mapping the keys to appropriate locations (or indices) in a hash table is called *hashing*.

Figure 15.4 shows a hash table in which each key from the set K is mapped to locations generated by using a hash function. Note that keys k_2 and k_6 point to the same memory location. This is known as *collision*. That is, when two or more keys map to the same memory location, a collision is said to occur. Similarly, keys k_5 and k_7 also collide. The main goal of using a hash function is to reduce the range of array indices that have to be handled. Thus, instead of having U values, we just need K values, thereby reducing the amount of storage space required.

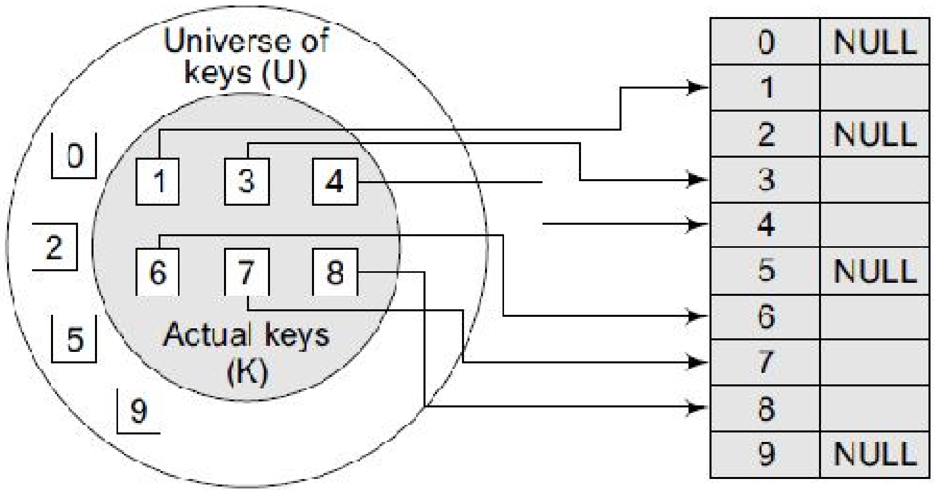


Figure 15.3 Direct relationship between key and index in the array

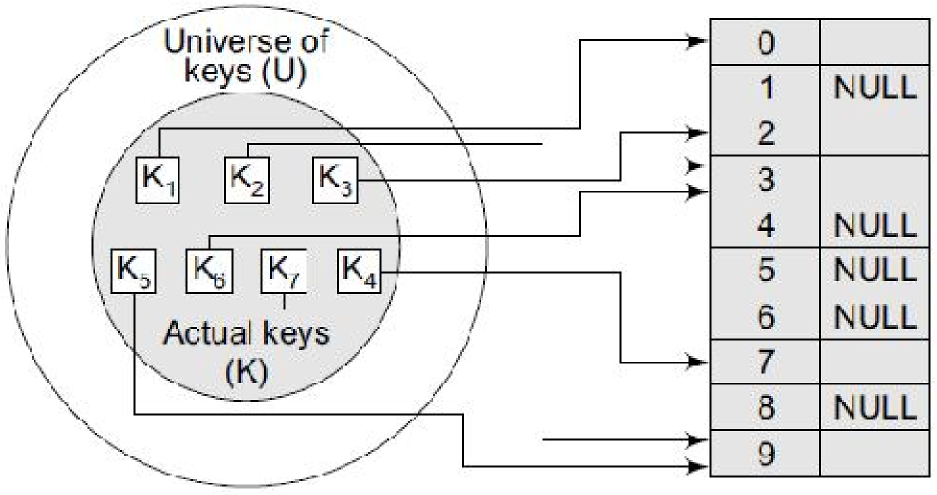


Figure 15.4 Relationship between keys and hash table index

HASH FUNCTIONs:

A hash function is a mathematical formula which, when applied to a key, produces an integer which can be used as an index for the key in the hash table. The main aim of a hash function is that elements should be relatively, randomly, and uniformly distributed. It produces a unique set of integers within some suitable range in order to reduce the number of collisions. In practice, there is no hash function that eliminates collisions completely. A good hash function can only minimize the number of collisions by spreading the elements uniformly throughout the array. In this section, we will discuss the popular hash functions which help to minimize collisions. But before that, let us first look at the properties of a good hash function.

Properties of a Good Hash Function

- ***Low cost*** The cost of executing a hash function must be small, so that using the hashing technique becomes preferable over other approaches. For example, if binary search algorithm can search an element from a sorted table of n items with $\log_2 n$ key comparisons, then the hash function must cost less than performing $\log_2 n$ key comparisons.

- ***Determinism*** A hash procedure must be deterministic. This means that the same hash value must be generated for a given input value. However, this criteria excludes hash functions that depend on external variable parameters (such as the time of day) and on the memory address of the object being hashed (because address of the object may change during processing).

- ***Uniformity*** A good hash function must map the keys as evenly as possible over its output range. This means that the probability of generating every hash value in the output range should roughly be the same. The property of uniformity also minimizes the number of collisions.

DIFFERENT HASH FUNCTIONS

In this section, we will discuss the hash functions which use numeric keys. However, there can be cases in real-world applications where we can have alphanumeric keys rather than simple numeric keys. In such cases, the ASCII value of the character can be used to transform it into its equivalent numeric key. Once this transformation is done, any of the hash functions given below can be applied to generate the hash value.

1 Division Method:

It is the most simple method of hashing an integer x. This method divides x by M and then uses the remainder obtained. In this case, the hash function can be given as

$h(x) = x \bmod M$

The division method is quite good for just about any value of M and since it requires only a single division operation, the method works very fast. However, extra care should be taken to select a suitable value for M.

For example, suppose M is an even number then h(x) is even if x is even and h(x) is odd if x is odd. If all possible keys are equi-probable, then this is not a problem. But if even keys are more likely than odd keys, then the division method will not spread the hashed values uniformly.

Generally, it is best to choose M to be a prime number because making M a prime number increases the likelihood that the keys are mapped with a uniformity in the output range of values. M should also be not too close to the exact powers of 2. If we have

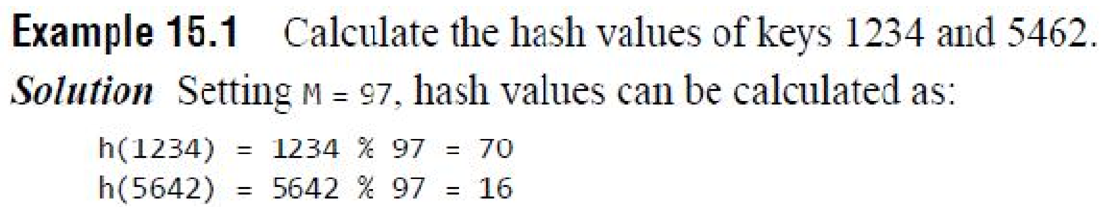
$h(x) = x \bmod 2_k$

then the function will simply extract the lowest k bits of the binary representation of x.

The division method is extremely simple to implement. The following code segment illustrates how to do this:

```
int const M = 97; // a prime number
int h (int x)
{ return (x % M); }
```

A potential drawback of the division method is that while using this method, consecutive keys map to consecutive hash values. On one hand, this is good as it ensures that consecutive keys do not collide, but on the other, it also means that consecutive array locations will be occupied. This may lead to degradation in performance.



Example 15.1 Calculate the hash values of keys 1234 and 5462.

Solution Setting M = 97, hash values can be calculated as:

```
h(1234) = 1234 % 97 = 70
h(5642) = 5642 % 97 = 16
```

2 Multiplication Method:

The steps involved in the multiplication method are as follows:

Step 1: Choose a constant A such that $0 < A < 1$.

Step2: Multiply the key k by A.

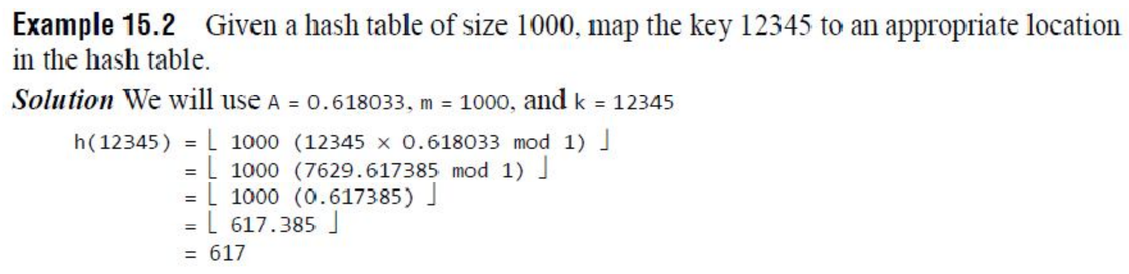
Step 3: Extract the fractional part of kA.

Step 4: Multiply the result of Step 3 by the size of hash table (m).

Hence, the hash function can be given as:

h(k) = I m (kA mod 1) ˚

where (kA mod 1) gives the fractional part of kA and m is the total number of indices in the hash table. The greatest advantage of this method is that it works practically with any value of A. Although the algorithm works better with some values, the optimal choice depends on the characteristics of the data being hashed. Knuth has suggested that the best choice of A is
" (sqrt5 – 1) /2 = 0.6180339887



Example 15.2 Given a hash table of size 1000, map the key 12345 to an appropriate location in the hash table.
Solution We will use A = 0.618033, m = 1000, and k = 12345

$$\begin{aligned} h(12345) &= \lfloor 1000\ (12345 \times 0.618033 \bmod 1) \rfloor \\ &= \lfloor 1000\ (7629.617385 \bmod 1) \rfloor \\ &= \lfloor 1000\ (0.617385) \rfloor \\ &= \lfloor 617.385 \rfloor \\ &= 617 \end{aligned}$$

3 Mid-Square Method:

The mid-square method is a good hash function which works in two steps:

Step 1: Square the value of the key. That is, find k_2.

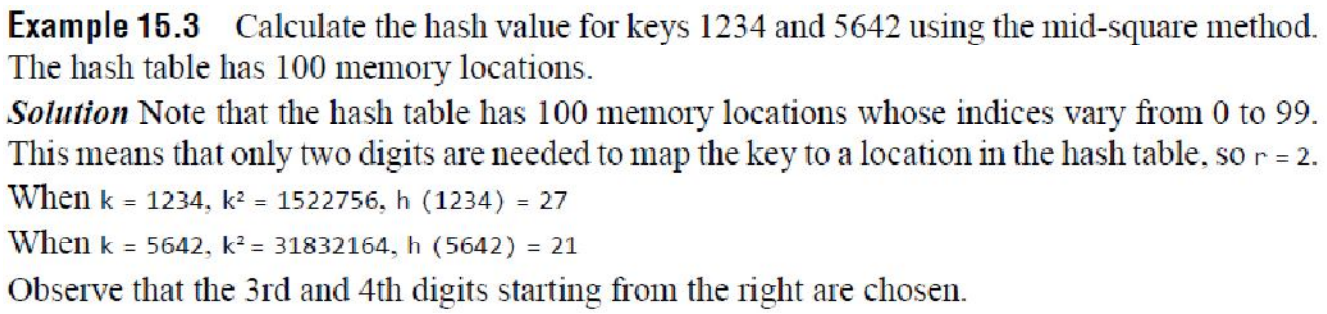
Step 2: Extract the middle r digits of the result obtained in Step 1.

The algorithm works well because most or all digits of the key value contribute to the result. This is because all the digits in the original key value contribute to produce the middle digits of the squared value. Therefore, the result is not dominated by the distribution of the bottom digit or the top digit of the original key value.

In the mid-square method, the same r digits must be chosen from all the keys. Therefore, the hash function can be given as:

h(k) = s

where s is obtained by selecting r digits from k^2.



Example 15.3 Calculate the hash value for keys 1234 and 5642 using the mid-square method. The hash table has 100 memory locations.
Solution Note that the hash table has 100 memory locations whose indices vary from 0 to 99. This means that only two digits are needed to map the key to a location in the hash table, so r = 2.
When k = 1234, k^2 = 1522756, h (1234) = 27
When k = 5642, k^2 = 31832164, h (5642) = 21
Observe that the 3rd and 4th digits starting from the right are chosen.

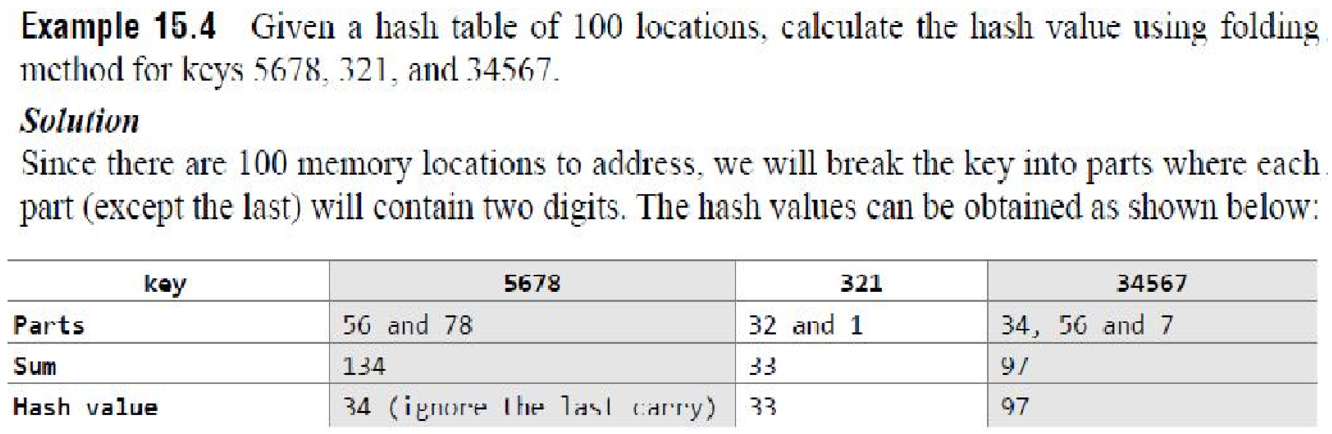
4 Folding Method:

The folding method works in the following two steps:

Step 1: Divide the key value into a number of parts. That is, divide k into parts k_1, k_2, ..., k_n, where each part has the same number of digits except the last part which may have lesser digits than the other parts.

Step 2: Add the individual parts. That is, obtain the sum of $k_1 + k_2 + ... + k_n$. The hash value is produced by ignoring the last carry, if any.

Note that the number of digits in each part of the key will vary depending upon the size of the hash table. For example, if the hash table has a size of 1000, then there are 1000 locations in the hash table. To address these 1000 locations, we need at least three digits; therefore, each part of the key must have three digits except the last part which may have lesser digits.



Example 15.4 Given a hash table of 100 locations, calculate the hash value using folding method for keys 5678, 321, and 34567.

Solution

Since there are 100 memory locations to address, we will break the key into parts where each part (except the last) will contain two digits. The hash values can be obtained as shown below:

key	5678	321	34567
Parts	56 and 78	32 and 1	34, 56 and 7
Sum	134	33	97
Hash value	34 (ignore the last carry)	33	97

COLLISIONS:

As discussed earlier in this chapter, collisions occur when the hash function maps two different keys to the same location. Obviously, two records cannot be stored in the same location. Therefore, a method used to solve the problem of collision, also called *collision resolution technique*, is applied. The two most popular methods of resolving collisions are:

1. Open addressing
2. Chaining

Collision Resolution by Open Addressing:

Once a collision takes place, open addressing or closed hashing computes new positions using a probe sequence and the next record is stored in that position. In this technique, all the values are stored in the hash table. The hash table contains two types of values: *sentinel values* (e.g., –1) and *data values*. The presence of a sentinel value indicates that the location contains no data value at present but can be used to hold a value.

When a key is mapped to a particular memory location, then the value it holds is checked. If it contains a sentinel value, then the location is free and the data value can be stored in it. However, if the location already has some data value stored in it, then other slots are examined systematically in the forward direction to find a free slot. If even a single free location is not found, then we have an OVERFLOW condition.

The process of examining memory locations in the hash table is called *probing*. Open addressing technique can be implemented using linear probing, quadratic probing, double hashing, and rehashing.

Collision Resolution by Chaining:

In chaining, each location in a hash table stores a pointer to a linked list that contains all the key values that were hashed to that location. That is, location l in the hash table points to the head of the linked list of all the key values that hashed to l. However, if no key value hashes to l, then location l in the hash table contains NULL. Figure 15.5 shows how the key values are mapped to a location in the hash table and stored in a linked list that corresponds to that location.

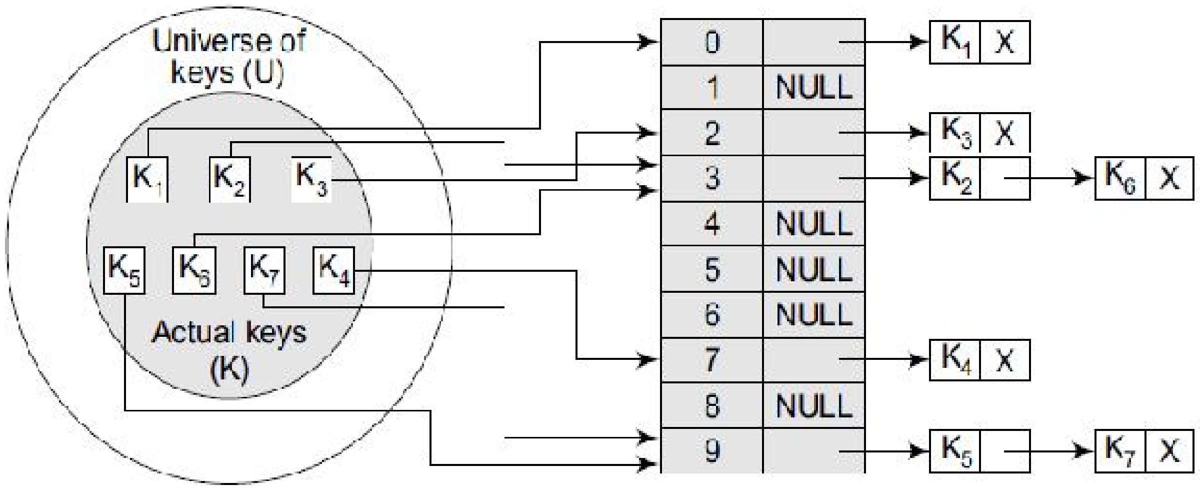


Figure 15.5 Keys being hashed to a chained hash table

APPLICATIONS OF HASHING:

Hash tables are widely used in situations where enormous amounts of data have to be accessed to quickly search and retrieve information. A few typical examples where hashing is used are given here.

Hashing is used for database indexing. Some database management systems store a separate file known as the index file. When data has to be retrieved from a file, the key information is first searched in the appropriate index file which references the exact record location of the data in the database file. This key information in the index file is often stored as a hashed value.

3. Bubble sort:

Bubble sort is a very simple method that sorts the array elements by repeatedly moving the largest element to the highest index position of the array segment (in case of arranging elements in ascending order). In bubble sorting, consecutive adjacent pairs of elements in the array are compared with each other. If the element at the lower index is greater than the element at the higher index, the two elements are interchanged so that the element is placed before the bigger one. This process will continue till the list of unsorted elements exhausts.

This procedure of sorting is called bubble sorting because elements ‘bubble’ to the top of the list. Note that at the end of the first pass, the largest element in the list will be placed at its proper position (i.e., at the end of the list).

Note If the elements are to be sorted in descending order, then in first pass the smallest element is moved to the highest index of the array.

Technique:

The basic methodology of the working of bubble sort is given as follows:

(a) In Pass 1, A[0] and A[1] are compared, then A[1] is compared with A[2], A[2] is compared with A[3], and so on. Finally, A[N–2] is compared with A[N–1]. Pass 1 involves n–1 comparisons and places the biggest element at the highest index of the array.

(b) In Pass 2, A[0] and A[1] are compared, then A[1] is compared with A[2], A[2] is compared with A[3], and so on. Finally, A[N–3] is compared with A[N–2]. Pass 2 involves n–2 comparisons and places the second biggest element at the second highest index of the array.

(c) In Pass 3, A[0] and A[1] are compared, then A[1] is compared with A[2], A[2] is compared with A[3], and so on. Finally, A[N–4] is compared with A[N–3]. Pass 3 involves n–3 comparisons and places the third biggest element at the third highest index of the array.

(d) In Pass n–1, A[0] and A[1] are compared so that A[0]<A[1]. After this step, all the elements of the array are arranged in ascending order.

Example To discuss bubble sort in detail, let us consider an array A[] that has the following elements:

A[] = {30, 52, 29, 87, 63, 27, 19, 54}

Pass 1:

(a) Compare 30 and 52. Since 30 < 52, no swapping is done.

(b) Compare 52 and 29. Since 52 > 29, swapping is done.

    30, 29, 52, 87, 63, 27, 19, 54

(c) Compare 52 and 87. Since 52 < 87, no swapping is done.

(d) Compare 87 and 63. Since 87 > 63, swapping is done.

    30, 29, 52, 63, 87, 27, 19, 54

(e) Compare 87 and 27. Since 87 > 27, swapping is done.

    30, 29, 52, 63, 27, 87, 19, 54

(f) Compare 87 and 19. Since 87 > 19, swapping is done.

    30, 29, 52, 63, 27, 19, 87, 54

(g) Compare 87 and 54. Since 87 > 54, swapping is done.

    30, 29, 52, 63, 27, 19, 54, 87

Observe that after the end of the first pass, the largest element is placed at the highest index of the array. All the other elements are still unsorted.

Pass 2:

(a) Compare 30 and 29. Since 30 > 29, swapping is done.

    29, 30, 52, 63, 27, 19, 54, 87

(b) Compare 30 and 52. Since 30 < 52, no swapping is done.

(c) Compare 52 and 63. Since 52 < 63, no swapping is done.

(d) Compare 63 and 27. Since 63 > 27, swapping is done.

29, 30, 52, 27, 63, 19, 54, 87

(e) Compare 63 and 19. Since 63 > 19, swapping is done.

29, 30, 52, 27, 19, 63, 54, 87

(f) Compare 63 and 54. Since 63 > 54, swapping is done.

29, 30, 52, 27, 19, 54, 63, 87

Observe that after the end of the second pass, the second largest element is placed at the second highest index of the array. All the other elements are still unsorted.

Pass 3:

(a) Compare 29 and 30. Since 29 < 30, no swapping is done.

(b) Compare 30 and 52. Since 30 < 52, no swapping is done.

(c) Compare 52 and 27. Since 52 > 27, swapping is done.

29, 30, 27, 52, 19, 54, 63, 87

(d) Compare 52 and 19. Since 52 > 19, swapping is done.

29, 30, 27, 19, 52, 54, 63, 87

(e) Compare 52 and 54. Since 52 < 54, no swapping is done.

Observe that after the end of the third pass, the third largest element is placed at the third highest index of the array. All the other elements are still unsorted.

Pass 4:

(a) Compare 29 and 30. Since 29 < 30, no swapping is done.

(b) Compare 30 and 27. Since 30 > 27, swapping is done.

29, 27, 30, 19, 52, 54, 63, 87

(c) Compare 30 and 19. Since 30 > 19, swapping is done.

29, 27, 19, 30, 52, 54, 63, 87

(d) Compare 30 and 52. Since 30 < 52, no swapping is done.

Observe that after the end of the fourth pass, the fourth largest element is placed at the fourth highest index of the array. All the other elements are still unsorted.

Pass 5:

(a) Compare 29 and 27. Since 29 > 27, swapping is done.

27, 29, 19, 30, 52, 54, 63, 87

(b) Compare 29 and 19. Since 29 > 19, swapping is done.

27, 19, 29, 30, 52, 54, 63, 87

(c) Compare 29 and 30. Since 29 < 30, no swapping is done.

Observe that after the end of the fifth pass, the fifth largest element is placed at the fifth highest index of the array. All the other elements are still unsorted.

Pass 6:

(a) Compare 27 and 19. Since 27 > 19, swapping is done.

19, 27, 29, 30, 52, 54, 63, 87

(b) Compare 27 and 29. Since 27 < 29, no swapping is done.

Observe that after the end of the sixth pass, the sixth largest element is placed at the sixth largest index of the array. All the other elements are still unsorted.

Pass 7:

(a) Compare 19 and 27. Since 19 < 27, no swapping is done.

In this algorithm, the outer loop is for the total number of passes which is N–1. The inner loop will be executed for every pass. However, the frequency of the inner loop will decrease with every pass because after every pass, one element will be in its correct position. Therefore, for every pass, the inner loop will be executed N–I times, where N is the number of elements in the array and I is the count of the pass.

Complexity of Bubble Sort:

The complexity of any sorting algorithm depends upon the number of comparisons. In bubble sort, we have seen that there are N–1 passes in total. In the first pass, N–1 comparisons are made to place the highest element in its correct position. Then, in Pass 2, there are N–2 comparisons and the second highest element is placed in its position. Therefore, to compute the complexity of bubble sort, we need to calculate the total number of comparisons. It can be given as:

f(n) = (n – 1) + (n – 2) + (n – 3) + ... + 3 + 2 + 1

f(n) = n (n – 1)/2

f(n) = n2/2 + O(n) = O(n2)

Therefore, the complexity of bubble sort algorithm is O(n2). It means the time required to execute bubble sort is proportional to n2, where n is the total number of elements in the array.

Programming Example:

Write a program to enter n numbers in an array. Redisplay the array with elements being sorted in ascending order.

```
#include <stdio.h>

int main()
{
        int i, n, temp, j, arr[10];

        printf("\n Enter the number of elements in the array : ");
        scanf("%d", &n);
        printf("\n Enter the elements: ");
        for(i=0;i<n;i++)
        {
                scanf("%d", &arr [i]);
        }
        for(i=0;i<n;i++)
        {
                for(j=0;j<n-i-1;j++)
                {
                        if(arr[j] > arr[j+1])
                        {
                                temp = arr[j];
                                arr[j] = arr[j+1];
                                arr[j+1] = temp;
                        }
                }
        }
        printf("\n The array sorted in ascending order is :\n");
        for(i=0;i<n;i++)
                printf("%d\t", arr[i]);

        return 0;
}
```

```
Output
Enter the number of elements in the array : 10
Enter the elements : 8 9 6 7 5 4 2 3 1 10
The array sorted in ascending order is :
1 2 3 4 5 6 7 8 9 10
```

4. Selection sort:

Selection sort is a sorting algorithm that has a quadratic running time complexity of O(n^2), there by making it inefficient to be used on large lists. Although selection sort performs worse than insertion sort algorithm, it is noted for its simplicity and also has performance advantages over more complicated algorithms in certain situations. Selection sort is generally used for sorting files with very large objects (records) and small keys.

Technique

Consider an array ARR with N elements. Selection sort works as follows:

- First find the smallest value in the array and place it in the first position. Then, find the second smallest value in the array and place it in the second position. Repeat this procedure until the entire array is sorted. Therefore,

- In Pass 1, find the position POS of the smallest value in the array and then swap ARR[POS] and ARR[0]. Thus, ARR[0] is sorted.

- In Pass 2, find the position POS of the smallest value in sub-array of N–1 elements. Swap ARR[POS] with ARR[1]. Now, ARR[0] and ARR[1] is sorted.

- In Pass N–1, find the position POS of the smaller of the elements ARR[N–2] and ARR[N–1]. Swap ARR[POS] and ARR[N–2] so that ARR[0], ARR[1], ..., ARR[N–1] is sorted.

Example 14.4 Sort the array given below using selection sort.

39	9	81	45	90	27	72	18

PASS	POS	ARR[0]	ARR[1]	ARR[2]	ARR[3]	ARR[4]	ARR[5]	ARR[6]	ARR[7]
1	1	9	39	81	45	90	27	72	18
2	7	9	18	81	45	90	27	72	39
3	5	9	18	27	45	90	81	72	39
4	7	9	18	27	39	90	81	72	45
5	7	9	18	27	39	45	81	72	90
6	6	9	18	27	39	45	72	81	90
7	6	9	18	27	39	45	72	81	90

Algorithm: SELECTION_SORT

Input: ARR: Unsorted Element List

Output: ARR: Sorted Element List

```
SMALLEST (ARR, K, N, POS)

Step 1: [INITIALIZE] SET SMALL = ARR[K]
Step 2: [INITIALIZE] SET POS = K
Step 3: Repeat for J = K+1 to N-1
            IF SMALL > ARR[J]
                  SET SMALL = ARR[J]
                  SET POS = J
            [END OF IF]
        [END OF LOOP]
Step 4: RETURN POS
```

```
SELECTION SORT(ARR, N)

Step 1: Repeat Steps 2 and 3 for K = 1
        to N-1
Step 2:     CALL SMALLEST(ARR, K, N, POS)
Step 3:     SWAP A[K] with ARR[POS]
        [END OF LOOP]
Step 4: EXIT
```

In the algorithm, during the Kth pass, we need to find the position POS of the smallest elements from ARR[K], ARR[K+1], ..., ARR[N]. To find the smallest element, we use a variable SMALL to hold the smallest value in the sub-array ranging from ARR[K] to ARR[N]. Then, swap ARR[K] with ARR[POS]. This procedure is repeated until all the elements in the array are sorted.

Complexity of Selection Sort:

Selection sort is a sorting algorithm that is independent of the original order of elements in the array. In Pass 1, selecting the element with the smallest value calls for scanning all n elements; thus, n–1 comparisons are required in the first pass. Then, the smallest value is swapped with the element in the first position. In Pass 2, selecting the second smallest value requires scanning the remaining n – 1 elements and so on.

Therefore,

$(n - 1) + (n - 2) + ... + 2 + 1$
$= n(n - 1) / 2 = O(n^2)$ comparisons

Write a program to sort an array using selection sort algorithm.

```
#include <stdio.h>

int smallest(int arr[], int k, int n);

void selection_sort(int arr[], int n);

int main(int argc, char *argv[])
{
        int arr[10], i, n;
        printf("\n Enter the number of elements in the array: ");
        scanf("%d", &n);
        printf("\n Enter the elements of the array: ");
        for(i=0;i<n;i++)
        {
                scanf("%d", &arr[i]);

        }

        selection_sort(arr, n);

        printf("\n The sorted array is: \n");
        for(i=0;i<n;i++)
                printf(" %d\t", arr[i]);
        return 0;
}

int smallest(int arr[], int k, int n)
{
        int pos = k, small=arr[k], i;
        for(i=k+1;i<n;i++)
        {
                if(arr[i]< small)
                {
                        small = arr[i];
                        pos = i;
                }
        }
        return pos;
}

void selection_sort(int arr[],int n)
{
        int k, pos, temp;
        for(k=0;k<n;k++)
        {
                pos = smallest(arr, k, n);
                temp = arr[k];
                arr[k] = arr[pos];
                arr[pos] = temp;
        }
}
```

5. Insertion sort:

Insertion sort is a very simple sorting algorithm in which the sorted array (or list) is built one element at a time. We all are familiar with this technique of sorting, as we usually use it for ordering a deck of cards while playing bridge. The main idea behind insertion sort is that it inserts each item into its proper place in the final list.

To save memory, most implementations of the insertion sort algorithm work by moving the current data element past the already sorted values and repeatedly interchanging it with the preceding value until it is in its correct place. Insertion sort is less efficient as compared to other more advanced algorithms such as quick sort, heap sort, and merge sort.

Technique:

Insertion sort works as follows:

- The array of values to be sorted is divided into two sets. One that stores sorted values and another that contains unsorted values.

- The sorting algorithm will proceed until there are elements in the unsorted set.

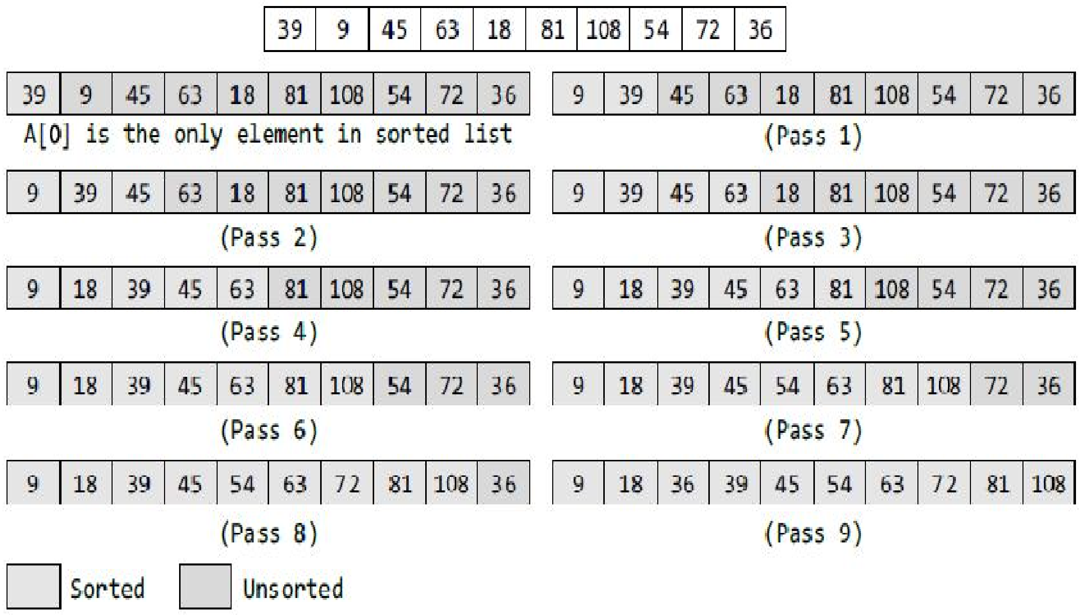
- Suppose there are n elements in the array. Initially, the element with index 0 (assuming LB =0) is in the sorted set. Rest of the elements are in the unsorted set.

- The first element of the unsorted partition has array index 1 (if LB = 0).

- During each iteration of the algorithm, the first element in the unsorted set is picked up and inserted into the correct position in the sorted set.

Example 14.3 Consider an array of integers given below. We will sort the values in the array using insertion sort.

Solution



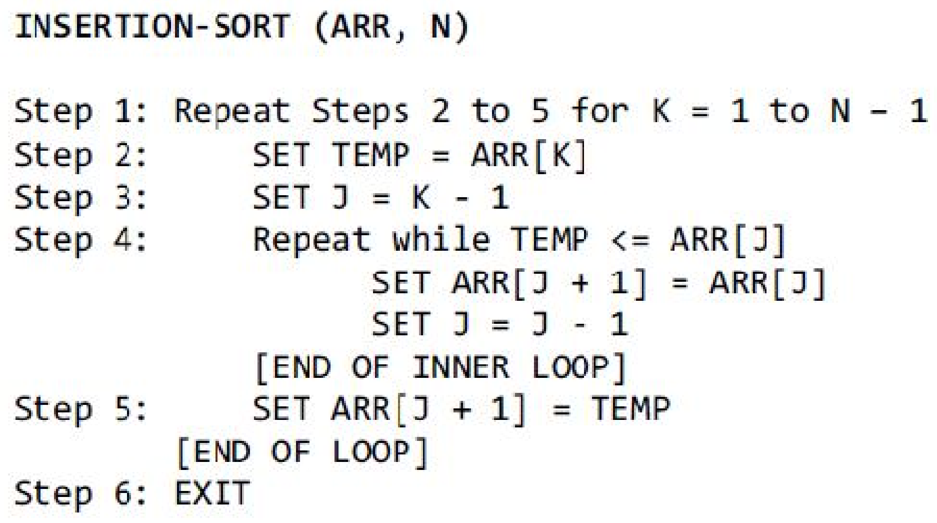
Initially, `A[0]` is the only element in the sorted set. In Pass 1, `A[1]` will be placed either before or after `A[0]`, so that the array `A` is sorted. In Pass 2, `A[2]` will be placed either before `A[0]`, in between `A[0]` and `A[1]`, or after `A[1]`. In Pass 3, `A[3]` will be placed in its proper place. In Pass `N-1`, `A[N-1]` will be placed in its proper place to keep the array sorted.

To insert an element A[K] in a sorted list A[0], A[1], ..., A[K–1], we need to compare A[K] with A[K–1], then with A[K–2], A[K–3], and so on until we meet an element A[J] such that A[J] <= A[K]. In order to insert A[K] in its correct position, we need to move elements A[K–1], A[K–2], ..., A[J] by one position and then A[K] is inserted at the (J+1)th location.

Algorithm: INSERTION_SORT

Input: ARR: Unsorted Element List

Output: ARR: Sorted Element list



```
INSERTION-SORT (ARR, N)

Step 1: Repeat Steps 2 to 5 for K = 1 to N - 1
Step 2:     SET TEMP = ARR[K]
Step 3:     SET J = K - 1
Step 4:     Repeat while TEMP <= ARR[J]
                 SET ARR[J + 1] = ARR[J]
                 SET J = J - 1
            [END OF INNER LOOP]
Step 5:     SET ARR[J + 1] = TEMP
        [END OF LOOP]
Step 6: EXIT
```

In the algorithm, Step 1 executes a for loop which will be repeated for each element in the array. In Step 2, we store the value of the Kth element in TEMP. In Step 3, we set the Jth index in the array. In Step 4, a for loop is executed that will create space for the new element from the unsorted list to be stored in the list of sorted elements. Finally, in Step 5, the element is stored at the (J+1)th location.

Complexity of Insertion Sort:

For insertion sort, the best case occurs when the array is already sorted. In this case, the running time of the algorithm has a linear running time (i.e., O(n)). This is because, during each iteration, the first element from the unsorted set is compared only with the last element of the sorted set of the array. Similarly, the worst case of the insertion sort algorithm occurs when the array is sorted in the reverse order.

In the worst case, the first element of the unsorted set has to be compared with almost every element in the sorted set. Furthermore, every iteration of the inner loop will have to shift the elements of the sorted set of the array before inserting the next element. Therefore, in the worst case, insertion sort has a quadratic running time (i.e., O(n2)). Even in the average case, the insertion sort algorithm will have to make at least (K–1)/2 comparisons. Thus, the average case also has a quadratic running time.

Write a program to sort an array using insertion sort algorithm.

```c
#include <stdio.h>
#define size 5

void insertion_sort(int arr[], int n);

void main()
{
	int arr[size], i, n;
	printf("\n Enter the number of elements in the array: ");
	scanf("%d", &n);
	printf("\n Enter the elements of the array: ");
	for(i=0;i<n;i++)
	{
		scanf("%d", &arr[i]);
	}
	insertion_sort(arr, n);
	printf("\n The sorted array is: \n");
	for(i=0;i<n;i++)
		printf(" %d\t", arr[i]);
	return 0;
}

void insertion_sort(int arr[], int n)
{
	int i, j, temp;
	for(i=1;i<n;i++)
	{
		temp = arr[i];
		j = i-1;
		while((temp < arr[j]) && (j>=0))
		{
			arr[j+1] = arr[j];
			j--;
		}
		arr[j+1] = temp;
	}
}
```

Output

```
Enter the number of elements in the array : 5
Enter the elements of the array : 500 1 50 23 76
The sorted array is :
1 23 20 76 500 6 7 8 9 10
```

6. Merge sort:

Merge sort is a sorting algorithm that uses the divide, conquer, and combine algorithmic paradigm. Divide means partitioning the n-element array to be sorted into two sub-arrays of n/2 elements. If A is an array containing zero or one element, then it is already sorted. However, if there are more elements in the array, divide A into two sub-arrays, A1 and A2, each containing about half of the elements of A.

Conquer means sorting the two sub-arrays recursively using merge sort.
Combine means merging the two sorted sub-arrays of size n/2 to produce the sorted array of n elements.

Merge sort algorithm focuses on two main concepts to improve its performance (running time):

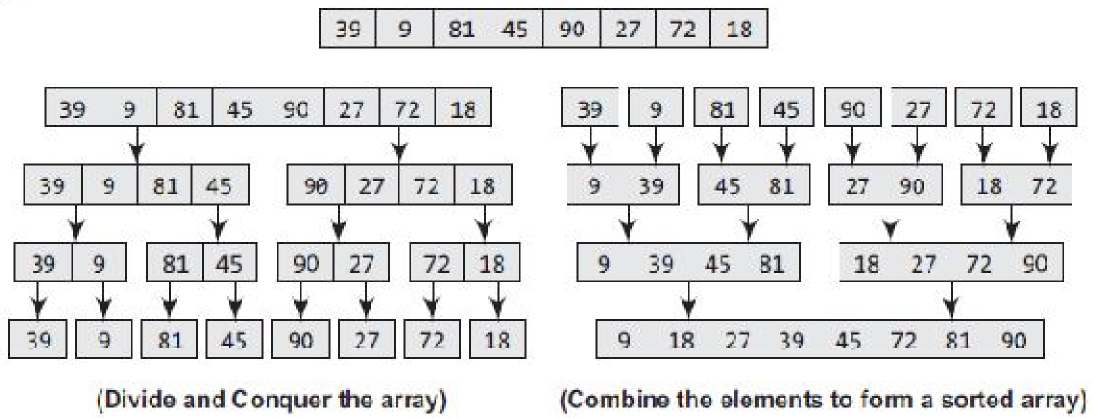
- A smaller list takes fewer steps and thus less time to sort than a large list.
- As number of steps is relatively less, thus less time is needed to create a sorted list from two sorted lists rather than creating it using two unsorted lists.

The basic steps of a merge sort algorithm are as follows:

- If the array is of length 0 or 1, then it is already sorted.
- Otherwise, divide the unsorted array into two sub-arrays of about half the size.
- Use merge sort algorithm recursively to sort each sub-array.
- Merge the two sub-arrays to form a single sorted list.

Example 14.5 Sort the array given below using merge sort.

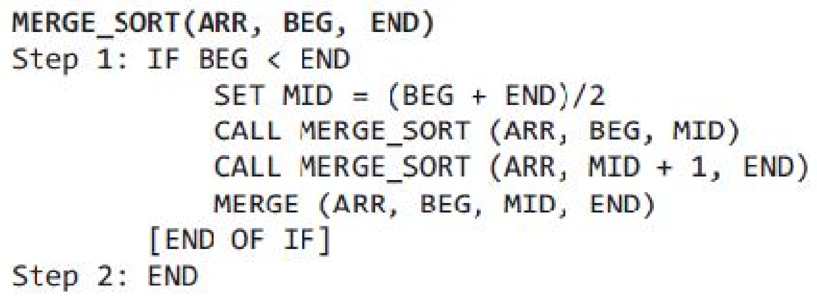
Solution


39 9 81 45 90 27 72 18
9 18 27 39 45 72 81 90
(Divide and Conquer the array)
(Combine the elements to form a sorted array)

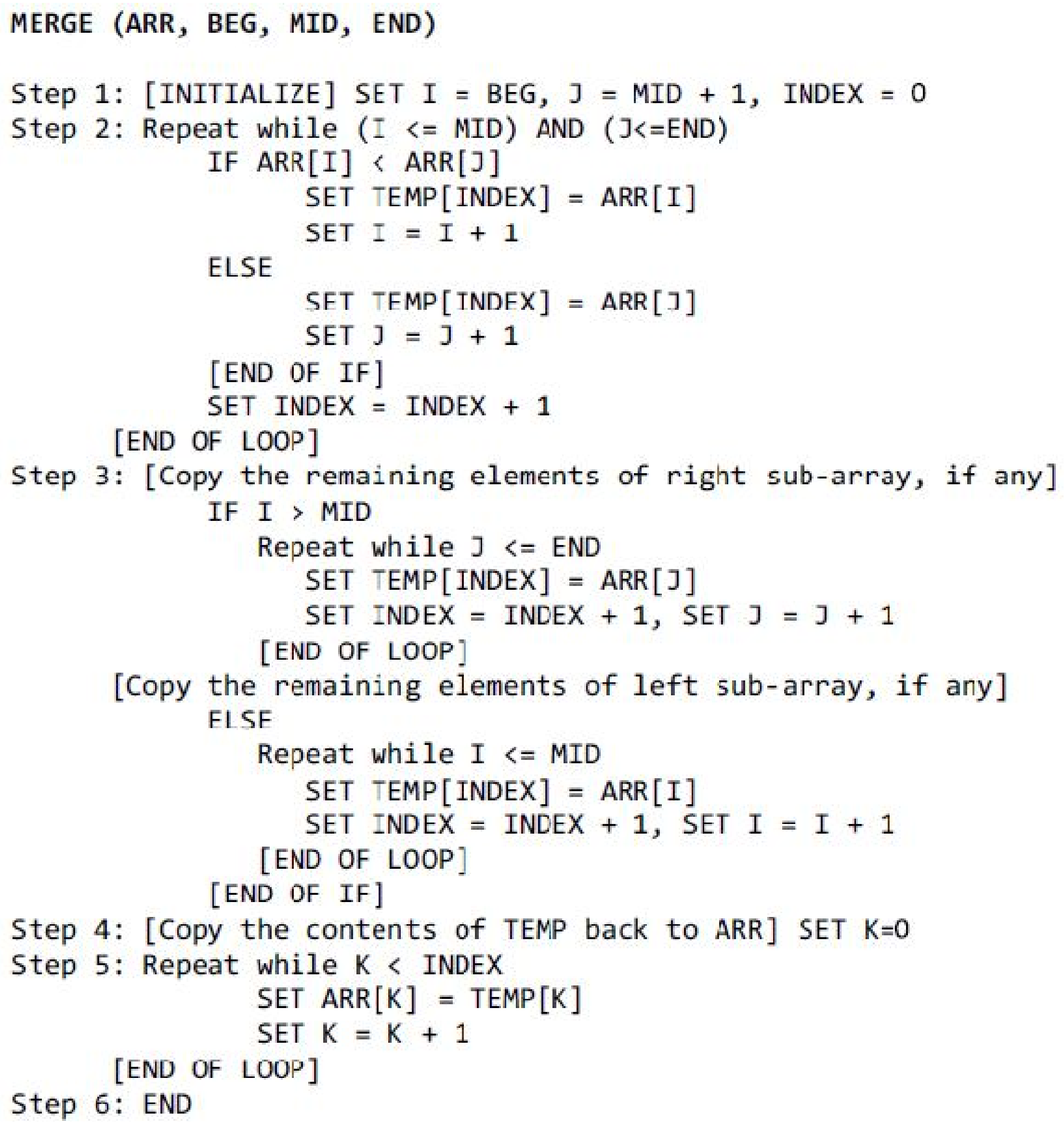
Algorithm: MERGE_SORT

Input: ARR: Unsorted Element List

Output: ARR: Sorted Element List



```
MERGE_SORT(ARR, BEG, END)
Step 1: IF BEG < END
            SET MID = (BEG + END)/2
            CALL MERGE_SORT (ARR, BEG, MID)
            CALL MERGE_SORT (ARR, MID + 1, END)
            MERGE (ARR, BEG, MID, END)
        [END OF IF]
Step 2: END
```



```
MERGE (ARR, BEG, MID, END)

Step 1: [INITIALIZE] SET I = BEG, J = MID + 1, INDEX = 0
Step 2: Repeat while (I <= MID) AND (J<=END)
            IF ARR[I] < ARR[J]
                  SET TEMP[INDEX] = ARR[I]
                  SET I = I + 1
            ELSE
                  SET TEMP[INDEX] = ARR[J]
                  SET J = J + 1
            [END OF IF]
            SET INDEX = INDEX + 1
      [END OF LOOP]
Step 3: [Copy the remaining elements of right sub-array, if any]
            IF I > MID
               Repeat while J <= END
                  SET TEMP[INDEX] = ARR[J]
                  SET INDEX = INDEX + 1, SET J = J + 1
               [END OF LOOP]
      [Copy the remaining elements of left sub-array, if any]
            ELSE
               Repeat while I <= MID
                  SET TEMP[INDEX] = ARR[I]
                  SET INDEX = INDEX + 1, SET I = I + 1
               [END OF LOOP]
            [END OF IF]
Step 4: [Copy the contents of TEMP back to ARR] SET K=0
Step 5: Repeat while K < INDEX
               SET ARR[K] = TEMP[K]
               SET K = K + 1
      [END OF LOOP]
Step 6: END
```

The merge sort algorithm uses a function merge which combines the sub-arrays to form a sorted array. While the merge sort algorithm recursively divides the list into smaller lists, the merge algorithm conquers the list to sort the elements in individual lists. Finally, the smaller lists are merged to form one list. To understand the merge algorithm, consider the figure below which shows how we merge two lists to form one list. For ease of understanding, we have taken two sub-lists each containing four elements. The same concept can be utilized to merge four sub-lists containing two elements, or eight sub-lists having one element each.

Complexity of Algorithm:

The running time of merge sort in the average case and the worst case can be given as O(n log n). Although merge sort has an optimal time complexity, it needs an additional space of O(n) for the temporary array TEMP.

Write a program to implement merge sort.

```
#include <stdio.h>
#define size 100

void merge(int a[], int, int, int);

void merge_sort(int a[],int, int);

int main()
{
	int arr[size], i, n;
	printf("\n Enter the number of elements in the array : ");
	scanf("%d", &n);
	printf("\n Enter the elements of the array: ");
	for(i=0;i<n;i++)
	{
		scanf("%d", &arr[i]);
	}
	merge_sort(arr, 0, n-1);
	printf("\n The sorted array is: \n");
	for(i=0;i<n;i++)
		printf(" %d\t", arr[i]);
	return 0;
}

void merge(int arr[], int beg, int mid, int end)
```

```
{
        int i=beg, j=mid+1, index=beg, temp[size], k;
        while((i<=mid) && (j<=end))
        {
                if(arr[i] < arr[j])
                {
                        temp[index] = arr[i];
                        i++;
                }
                else
                {
                        temp[index] = arr[j];
                        j++;
                }
                index++;
        }
        if(i>mid)
        {
                while(j<=end)
                {
                        temp[index] = arr[j];
                        j++;
                        index++;
                }
        }
        else
        {
                while(i<=mid)
                {
                        temp[index] = arr[i];
                        i++;
                        index++;
                }
        }
        for(k=beg;k<index;k++)
                arr[k] = temp[k];
}

void merge_sort(int arr[], int beg, int end)
{
        int mid;
        if(beg<end)
        {
                mid = (beg+end)/2;
                merge_sort(arr, beg, mid);
                merge_sort(arr, mid+1, end);
                merge(arr, beg, mid, end);
        }
}
```

7. Quick sort:

Quick sort is a widely used sorting algorithm developed by C. A. R. Hoare that makes O(n log n) comparisons in the average case to sort an array of n elements. However, in the worst case, it has a quadratic running time given as O(n2). Basically, the quick sort algorithm is faster than other O(n log n) algorithms, because its efficient implementation can minimize the probability of requiring quadratic time. Quick sort is also known as partition exchange sort.

Like merge sort, this algorithm works by using a divide-and-conquer strategy to divide a single unsorted array into two smaller sub-arrays.

The quick sort algorithm works as follows:

1. Select an element pivot from the array elements.

2. Rearrange the elements in the array in such a way that all elements that are less than the pivot appear before the pivot and all elements greater than the pivot element come after it (equal values can go either way). After such a partitioning, the pivot is placed in its final position. This is called the partition operation.

3. Recursively sort the two sub-arrays thus obtained. (One with sub-list of values smaller than
that of the pivot element and the other having higher value elements.)

Like merge sort, the base case of the recursion occurs when the array has zero or one element because in that case the array is already sorted. After each iteration, one element (pivot) is always in its final position. Hence, with every iteration, there is one less element to be sorted in the array. Thus, the main task is to find the pivot element, which will partition the array into two halves. To understand how we find the pivot element, follow the steps given below. (We take the first element in the array as pivot.)

Technique:

Quick sort works as follows:

1. Set the index of the first element in the array to loc and left variables. Also, set the index of the last element of the array to the right variable. That is, loc = 0, left = 0, and right = n–1 (where n in the number of elements in the array)

2. Start from the element pointed by right and scan the array from right to left, comparing each element on the way with the element pointed by the variable loc. That is, a[loc] should be less than a[right].

    (a) If that is the case, then simply continue comparing until right becomes equal to loc. Once right = loc, it means the pivot has been placed in its correct position.

    (b) However, if at any point, we have a[loc] > a[right], then interchange the two values and jump to Step 3.

    (c) Set loc = right

3. Start from the element pointed by left and scan the array from left to right, comparing each element on the way with the element pointed by loc. That is, a[loc] should be greater than a[left].

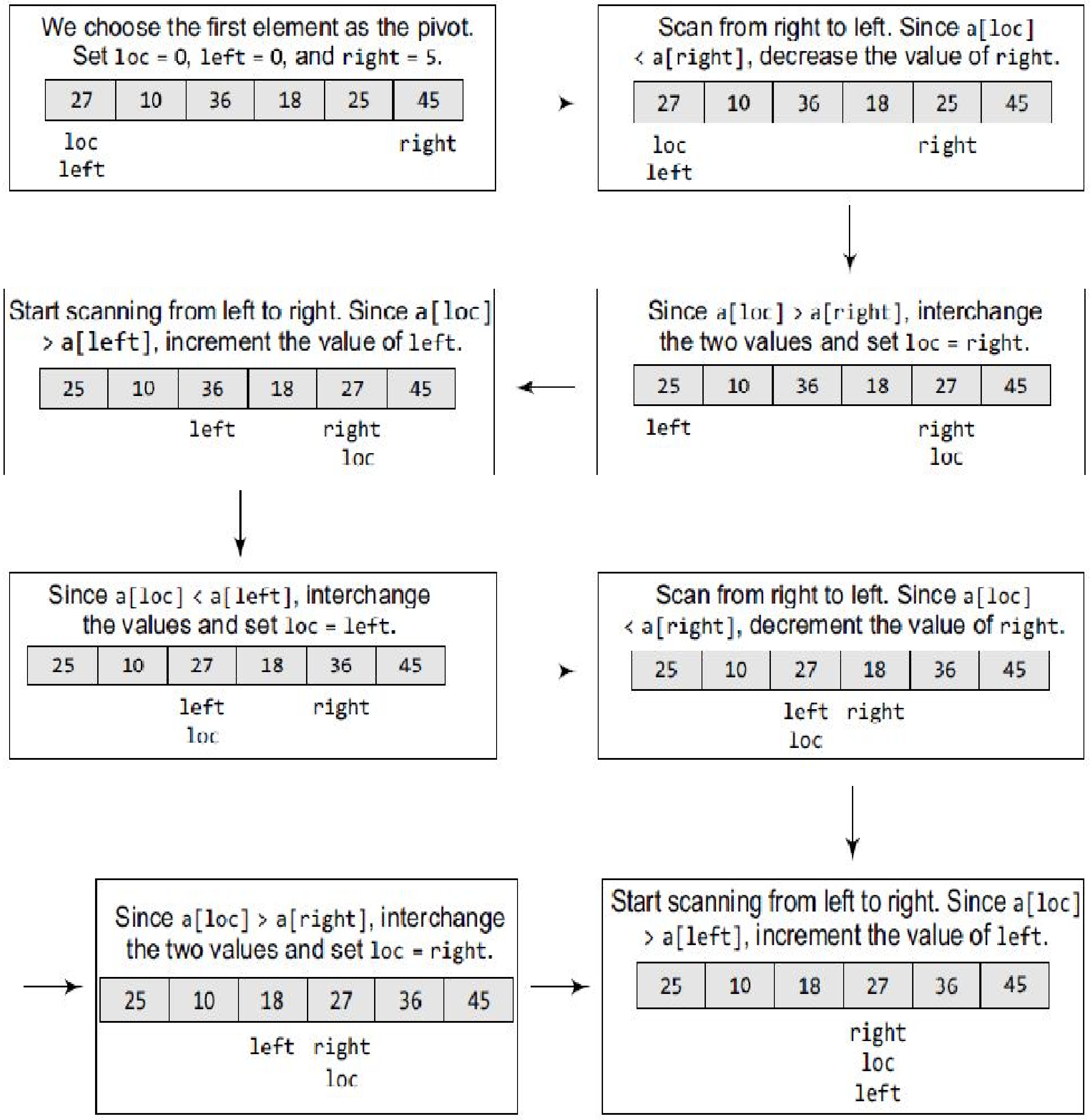
    (a) If that is the case, then simply continue comparing until left becomes equal to loc. Once left = loc, it means the pivot has been placed in its correct position.

    (b) However, if at any point, we have a[loc] < a[left], then interchange the two values and jump to Step 2.

    (c) Set loc = left.

Example 14.6 Sort the elements given in the following array using quick sort algorithm

27	10	36	18	25	45


We choose the first element as the pivot. Set loc = 0, left = 0, and right = 5.

Scan from right to left. Since a[loc] < a[right], decrease the value of right.

Since a[loc] > a[right], interchange the two values and set loc = right.

Start scanning from left to right. Since a[loc] > a[left], increment the value of left.

Since a[loc] < a[left], interchange the values and set loc = left.

Scan from right to left. Since a[loc] < a[right], decrement the value of right.

Since a[loc] > a[right], interchange the two values and set loc = right.

Start scanning from left to right. Since a[loc] > a[left], increment the value of left.

Now left = loc, so the procedure terminates, as the pivot element (the first element of the array, that is, 27) is placed in its correct position. All the elements smaller than 27 are placed before it and those greater than 27 are placed after it.

The left sub-array containing 25, 10, 18 and the right sub-array containing 36 and 45 are sorted in the same manner.

Algorithm: Quick_sort

Input: ARR: Unsorted Element List

Output: ARR: Sorted Element List

```
PARTITION (ARR, BEG, END, LOC)

Step 1: [INITIALIZE] SET LEFT = BEG, RIGHT = END, LOC = BEG, FLAG = 0
Step 2: Repeat Steps 3 to 6 while FLAG = 0
Step 3: Repeat while ARR[LOC] <= ARR[RIGHT] AND LOC != RIGHT
               SET RIGHT = RIGHT - 1
        [END OF LOOP]
Step 4: IF LOC = RIGHT
               SET FLAG = 1
        ELSE IF ARR[LOC] > ARR[RIGHT]
               SWAP ARR[LOC] with  ARR[RIGHT]
               SET LOC = RIGHT
        [END OF IF]
Step 5: IF FLAG = 0
               Repeat while ARR[LOC] >= ARR[LEFT] AND LOC != LEFT
               SET LEFT = LEFT + 1
               [END OF LOOP]
Step 6:        IF LOC = LEFT
                       SET FLAG = 1
               ELSE IF ARR[LOC] < ARR[LEFT]
                       SWAP ARR[LOC] with  ARR[LEFT]
                       SET LOC = LEFT
               [END OF IF]
        [END OF IF]
Step 7: [END OF LOOP]
Step 8: END
```

```
QUICK_SORT (ARR, BEG, END)

Step 1: IF (BEG < END)
             CALL PARTITION (ARR, BEG, END, LOC)
             CALL QUICKSORT(ARR, BEG, LOC - 1)
             CALL QUICKSORT(ARR, LOC + 1, END)
        [END OF IF]
Step 2: END
```

Complexity of Quick Sort:

In the average case, the running time of quick sort can be given as O(n log n). The partitioning of the array which simply loops over the elements of the array once uses O(n) time. In the best case, every time we partition the array, we divide the list into two nearly equal pieces. That is, the recursive call processes the sub-array of half the size. At the most, only log n nested calls can be made before we reach a sub-array of size 1. It means the depth of the call tree is O(log n). And because at each level, there can only be O(n), the resultant time is given as O(n log n) time.

Practically, the efficiency of quick sort depends on the element which is chosen as the pivot. Its worst-case efficiency is given as O(n2). The worst case occurs when the array is already sorted (either in ascending or descending order) and the left-most element is chosen as the pivot. However, many implementations randomly choose the pivot element. The randomized version of the quick sort algorithm always has an algorithmic complexity of O(n log n).

Write a program to implement quick sort algorithm.

```
#include <stdio.h>

#define size 100

int partition(int a[], int beg, int end);

void quick_sort(int a[], int beg, int end);

int main()
{
	int arr[size], i, n;
	printf("\n Enter the number of elements in the array: ");
	scanf("%d", &n);
	printf("\n Enter the elements of the array: ");
	for(i=0;i<n;i++)
	{
		scanf("%d", &arr[i]);
	}
	quick_sort(arr, 0, n-1);
	printf("\n The sorted array is: \n");
	for(i=0;i<n;i++)
	printf(" %d\t", arr[i]);
	return 0;
}

int partition(int a[], int beg, int end)
{
	int left, right, temp, loc, flag;
	loc = left = beg;
	right = end;
	flag = 0;
	while(flag != 1)
	{
		while((a[loc] <= a[right]) && (loc!=right))
			right--;
		if(loc==right)
```

```
                        flag =1;
                else if(a[loc]>a[right])
                {
                        temp = a[loc];
                        a[loc] = a[right];
                        a[right] = temp;
                        loc = right;
                }
                if(flag!=1)
                {
                        while((a[loc] >= a[left]) && (loc!=left))
                                left++;
                        if(loc==left)
                                flag =1;
                        else if(a[loc] <a[left])
                        {
                                temp = a[loc];
                                a[loc] = a[left];
                                a[left] = temp;
                                loc = left;
                        }
                }
        }
        return loc;
}

void quick_sort(int a[], int beg, int end)
{
        int loc;
        if(beg<end)
        {
                loc = partition(a, beg, end);
                quick_sort(a, beg, loc-1);
                quick_sort(a, loc+1, end);
        }
}
```

8. Radix sort:

Radix sort is a linear sorting algorithm for integers and uses the concept of sorting names in alphabetical order. When we have a list of sorted names, the radix is 26 (or 26 buckets) because there are 26 letters in the English alphabet. So radix sort is also known as bucket sort. Observe that words are first sorted according to the first letter of the name. That is, 26 classes are used to arrange the names, where the first class stores the names that begin with A, the second class contains the names with B, and so on. During the second pass, names are grouped according to the second letter.

After the second pass, names are sorted on the first two letters. This process is continued till the nth pass, where n is the length of the name with maximum number of letters. After every pass, all the names are collected in order of buckets. That is, first pick up the names in the first bucket that contains the names beginning with A. In the second pass, collect the names from the second bucket, and so on.

When radix sort is used on integers, sorting is done on each of the digits in the number. The sorting procedure proceeds by sorting the least significant to the most significant digit. While sorting the numbers, we have ten buckets, each for one digit (0, 1, 2, ⋯, 9) and the number of passes will depend on the length of the number having maximum number of digits.

Algorithm: Radix_Sort

Input: ARR: Unsorted Element List

Output: ARR: Sorted Element List

```
Algorithm for RadixSort (ARR, N)

Step 1: Find the largest number in ARR as LARGE
Step 2: [INITIALIZE] SET NOP = Number of digits in LARGE
Step 3: SET PASS = 0
Step 4: Repeat Step 5 while PASS <= NOP-1
Step 5:          SET I = 0 and INITIALIZE buckets
Step 6:          Repeat Steps 7 to 9 while I<N-1
Step 7:                SET DIGIT  = digit at PASSth place in A[I]
Step 8:                Add A[I] to the bucket numbered DIGIT
Step 9:                INCEREMENT bucket count for bucket numbered DIGIT
                 [END OF LOOP]
Step 10:         Collect the numbers in the bucket
        [END OF LOOP]
Step 11: END
```

Example 14.7 Sort the numbers given below using radix sort.

345, 654, 924, 123, 567, 472, 555, 808, 911

In the first pass, the numbers are sorted according to the digit at ones place. The buckets are pictured upside down as shown below.

Number	0	1	2	3	4	5	6	7	8	9
345						345				
654					654					
924					924					
123				123						
567								567		
472			472							
555						555				
808									808	
911		911								

After this pass, the numbers are collected bucket by bucket. The new list thus formed is used as an input for the next pass. In the second pass, the numbers are sorted according to the digit at the tens place. The buckets are pictured upside down.

Number	0	1	2	3	4	5	6	7	8	9
911		911								
472								472		
123			123							
654						654				
924			924							
345					345					
555						555				
567							567			
808	808									

In the third pass, the numbers are sorted according to the digit at the hundreds place. The buckets are pictured upside down.

Number	0	1	2	3	4	5	6	7	8	9
808									808	
911										911
123		123								
924										924
345				345						
654							654			
555						555				
567						567				
472					472					

The numbers are collected bucket by bucket. The new list thus formed is the final sorted result. After the third pass, the list can be given as

123, 345, 472, 555, 567, 654, 808, 911, 924.

Complexity of Radix Sort:

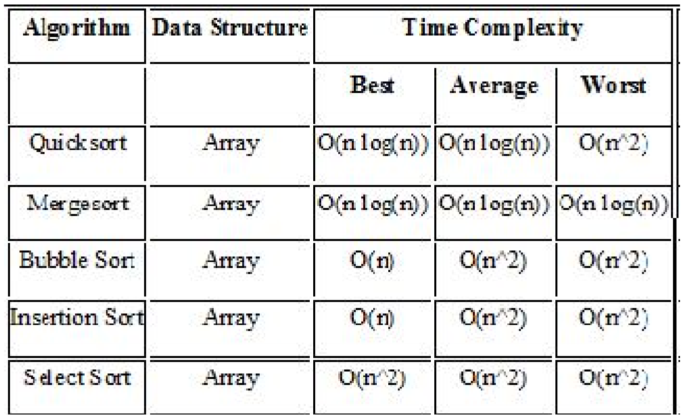
To calculate the complexity of radix sort algorithm, assume that there are n numbers that have to be sorted and k is the number of digits in the largest number. In this case, the radix sort algorithm is called a total of k times. The inner loop is executed n times. Hence, the entire radix sort algorithm takes O(kn) time to execute. When radix sort is applied on a data set of finite size (very small set of numbers), then the algorithm runs in O(n) asymptotic time.

Write a program to implement radix sort algorithm.

```
#include <stdio.h>

#define size 10

int largest(int arr[], int n);

void radix_sort(int arr[], int n);


int main()
{
	int arr[size], i, n;
	printf("\n Enter the number of elements in the array: ");
	scanf("%d", &n);
	printf("\n Enter the elements of the array: ");
	for(i=0;i<n;i++)
	{
		scanf("%d", &arr[i]);
	}
	radix_sort(arr, n);
	printf("\n The sorted array is: \n");
	for(i=0;i<n;i++)
	printf(" %d\t", arr[i]);
	return 0;
}

int largest(int arr[], int n)
{
	int large=arr[0], i;
	for(i=1;i<n;i++)
	{
		if(arr[i]>large)
			large = arr[i];
	}
```

```
        return large;
}

void radix_sort(int arr[], int n)
{
        int bucket[size][size], bucket_count[size];
        int i, j, k, remainder, NOP=0, divisor=1, large, pass;
        large = largest(arr, n);
        while(large>0)
        {
                NOP++;
                large/=size;
        }
        for(pass=0;pass<NOP;pass++) // Initialize the buckets
        {
                for(i=0;i<size;i++)
                        bucket_count[i]=0;
                for(i=0;i<n;i++)
                {
                        // sort the numbers according to the digit at passth place
                        remainder = (arr[i]/divisor)%size;
                        bucket[remainder][bucket_count[remainder]] = arr[i];
                        bucket_count[remainder] += 1;
                }
                // collect the numbers after PASS pass
                i=0;
                for(k=0;k<size;k++)
                {
                        for(j=0;j<bucket_count[k];j++)
                        {
                                arr[i] = bucket[k][j];
                                i++;
                        }
                }
                divisor *= size;
        }
}
}
```

Complexity and analysis:



Algorithm	Data Structure	Time Complexity		
		Best	Average	Worst
Quicksort	Array	O(n log(n))	O(n log(n))	O(n^2)
Mergesort	Array	O(n log(n))	O(n log(n))	O(n log(n))
Bubble Sort	Array	O(n)	O(n^2)	O(n^2)
Insertion Sort	Array	O(n)	O(n^2)	O(n^2)
Select Sort	Array	O(n^2)	O(n^2)	O(n^2)