Chapter 7

Searching and Sorting

There are basically two aspects of computer programming. One is data organization also commonly called as data structures. Till now we have seen about data structures and the techniques and algorithms used to access them. The other part of computer programming involves choosing the appropriate algorithm to solve the problem. Data structures and algorithms are linked each other. After developing programming techniques to represent information, it is logical to proceed to manipulate it. This chapter introduces this important aspect of problem solving.

Searching is used to find the location where an element is available. There are two types of search techniques. They are:

1. Linear or sequential search
2. Binary search

Sorting allows an efficient arrangement of elements within a given data structure. It is a way in which the elements are organized systematically for some purpose. For example, a dictionary in which words is arranged in alphabetical order and telephone director in which the subscriber names are listed in alphabetical order. There are many sorting techniques out of which we study the following.

1. Bubble sort
2. Quick sort
3. Selection sort and
4. Heap sort

There are two types of sorting techniques:

1. Internal sorting
2. External sorting

If all the elements to be sorted are present in the main memory then such sorting is called **internal sorting** on the other hand, if some of the elements to be sorted are kept on the secondary storage, it is called **external sorting**. Here we study only internal sorting techniques.

7.1. Linear Search:

This is the simplest of all searching techniques. In this technique, an ordered or unordered list will be searched one by one from the beginning until the desired element is found. If the desired element is found in the list then the search is successful otherwise unsuccessful.

Suppose there are ‘n’ elements organized sequentially on a List. The number of comparisons required to retrieve an element from the list, purely depends on where the element is stored in the list. If it is the first element, one comparison will do; if it is second element two comparisons are necessary and so on. On an average you need [(n+1)/2] comparison’s to search an element. If search is not successful, you would need ’n’ comparisons.

The time complexity of linear search is ***O(n)***.

Algorithm:

Let array a[n] stores n elements. Determine whether element ‘x’ is present or not.

```
linsrch(a[n], x)
{
        index = 0;
        flag = 0;
        while (index < n) do
        {
                if (x == a[index])
                {
                        flag = 1;
                        break;
                }
                index ++;
        }
        if(flag == 1)
                printf(“Data found at %d position“, index);
        else
                printf(“data not found”);

}
```

Example 1:

Suppose we have the following unsorted list: 45, 39, 8, 54, 77, 38, 24, 16, 4, 7, 9, 20

If we are searching for:
45, we’ll look at 1 element before success
39, we’ll look at 2 elements before success
8, we’ll look at 3 elements before success
54, we’ll look at 4 elements before success
77, we’ll look at 5 elements before success
38 we’ll look at 6 elements before success
24, we’ll look at 7 elements before success
16, we’ll look at 8 elements before success
4, we’ll look at 9 elements before success
7, we’ll look at 10 elements before success
9, we’ll look at 11 elements before success
20, we’ll look at 12 elements before success

For any element not in the list, we’ll look at 12 elements before failure.

Example 2:

Let us illustrate linear search on the following 9 elements:

Index	*0*	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*
Elements	-15	-6	0	7	9	23	54	82	101

Searching different elements is as follows:

1. Searching for x = 7Search successful, data found at 3rd position.

2. Searching for x = 82Search successful, data found at 7th position.

3. Searching for x = 42Search un-successful, data not found.

7.1.1. A non-recursive program for Linear Search:

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

main()
{
	int number[25], n, data, i, flag = 0;
	clrscr();
	printf("\n Enter the number of elements: ");
	scanf("%d", &n);
	printf("\n Enter the elements: ");
	for(i = 0; i < n; i++)
		scanf("%d", &number[i]);
	printf("\n Enter the element to be Searched: ");
	scanf("%d", &data);
	for( i = 0; i < n; i++)
	{
		if(number[i] == data)
		{
			flag = 1;
			break;
		}
	}
	if(flag == 1)
		printf("\n Data found at location: %d", i+1);
	else
		printf("\n Data not found ");
}
```

7.1.2. A Recursive program for linear search:

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

void linear_search(int a[], int data, int position, int n)
{
	if(position < n)
```

```
        {
                if(a[position] == data)
                        printf("\n Data Found at %d ", position);
                else
                        linear_search(a, data, position + 1, n);
        }
        else
                printf("\n Data not found");
}

void main()
{
        int a[25], i, n, data;
        clrscr();
        printf("\n Enter the number of elements: ");
        scanf("%d", &n);
        printf("\n Enter the elements: ");
        for(i = 0; i < n; i++)
        {
                scanf("%d", &a[i]);
        }
        printf("\n Enter the element to be seached: ");
        scanf("%d", &data);
        linear_search(a, data, 0, n);
        getch();
}
```

7.2. BINARY SEARCH

If we have 'n' records which have been ordered by keys so that $x_1 < x_2 < \dots < x_n$. When we are given a element 'x', binary search is used to find the corresponding element from the list. In case 'x' is present, we have to determine a value 'j' such that a[j] = x (successful search). If 'x' is not in the list then j is to set to zero (un successful search).

In Binary search we jump into the middle of the file, where we find key a[mid], and compare 'x' with a[mid]. If x = a[mid] then the desired record has been found. If x < a[mid] then 'x' must be in that portion of the file that precedes a[mid]. Similarly, if a[mid] > x, then further search is only necessary in that part of the file which follows a[mid].

If we use recursive procedure of finding the middle key a[mid] of the un-searched portion of a file, then every un-successful comparison of 'x' with a[mid] will eliminate roughly half the un-searched portion from consideration.

Since the array size is roughly halved after each comparison between 'x' and a[mid], and since an array of length 'n' can be halved only about $\log_2 n$ times before reaching a trivial length, the worst case complexity of Binary search is about $\log_2 n$.

Algorithm:

Let array a[n] of elements in increasing order, n ≥ 0, determine whether 'x' is present, and if so, set j such that x = a[j] else return 0.

```
binsrch(a[], n, x)
{
        low = 1; high = n;
        while (low ≤ high) do
        {
                mid =  (low + high)/2
                if (x < a[mid])
                        high = mid – 1;
                else if (x > a[mid])
                        low = mid + 1;
                else return mid;
        }
        return 0;
}
```

low and *high* are integer variables such that each time through the loop either 'x' is found or *low* is increased by at least one or *high* is decreased by at least one. Thus we have two sequences of integers approaching each other and eventually *low* will become greater than *high* causing termination in a finite number of steps if 'x' is not present.

Example 1:

Let us illustrate binary search on the following 12 elements:

Index	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*10*	*11*	*12*
Elements	4	7	8	9	16	20	24	38	39	45	54	77

If we are searching for x = 4: (This needs 3 comparisons)
low = 1, high = 12, mid = 13/2 = 6, check 20
low = 1, high = 5, mid = 6/2 = 3, check 8
low = 1, high = 2, mid = 3/2 = 1, check 4, ***found***

If we are searching for x = 7: (This needs 4 comparisons)
low = 1, high = 12, mid = 13/2 = 6, check 20
low = 1, high = 5, mid = 6/2 = 3, check 8
low = 1, high = 2, mid = 3/2 = 1, check 4
low = 2, high = 2, mid = 4/2 = 2, check 7, ***found***

If we are searching for x = 8: (This needs 2 comparisons)
low = 1, high = 12, mid = 13/2 = 6, check 20
low = 1, high = 5, mid = 6/2 = 3, check 8, ***found***

If we are searching for x = 9: (This needs 3 comparisons)
low = 1, high = 12, mid = 13/2 = 6, check 20
low = 1, high = 5, mid = 6/2 = 3, check 8
low = 4, high = 5, mid = 9/2 = 4, check 9, ***found***

If we are searching for x = 16: (This needs 4 comparisons)
low = 1, high = 12, mid = 13/2 = 6, check 20
low = 1, high = 5, mid = 6/2 = 3, check 8
low = 4, high = 5, mid = 9/2 = 4, check 9
low = 5, high = 5, mid = 10/2 = 5, check 16, ***found***

If we are searching for x = 20: (This needs 1 comparison)
low = 1, high = 12, mid = 13/2 = 6, check 20, ***found***

If we are searching for x = 24: (This needs 3 comparisons)
low = 1, high = 12, mid = 13/2 = 6, check 20
low = 7, high = 12, mid = 19/2 = 9, check 39
low = 7, high = 8, mid = 15/2 = 7, check 24, ***found***

If we are searching for x = 38: (This needs 4 comparisons)
low = 1, high = 12, mid = 13/2 = 6, check 20
low = 7, high = 12, mid = 19/2 = 9, check 39
low = 7, high = 8, mid = 15/2 = 7, check 24
low = 8, high = 8, mid = 16/2 = 8, check 38, ***found***

If we are searching for x = 39: (This needs 2 comparisons)
low = 1, high = 12, mid = 13/2 = 6, check 20
low = 7, high = 12, mid = 19/2 = 9, check 39, ***found***

If we are searching for x = 45: (This needs 4 comparisons)
low = 1, high = 12, mid = 13/2 = 6, check 20
low = 7, high = 12, mid = 19/2 = 9, check 39
low = 10, high = 12, mid = 22/2 = 11, check 54
low = 10, high = 10, mid = 20/2 = 10, check 45, ***found***

If we are searching for x = 54: (This needs 3 comparisons)
low = 1, high = 12, mid = 13/2 = 6, check 20
low = 7, high = 12, mid = 19/2 = 9, check 39
low = 10, high = 12, mid = 22/2 = 11, check 54, ***found***

If we are searching for x = 77: (This needs 4 comparisons)
low = 1, high = 12, mid = 13/2 = 6, check 20
low = 7, high = 12, mid = 19/2 = 9, check 39
low = 10, high = 12, mid = 22/2 = 11, check 54
low = 12, high = 12, mid = 24/2 = 12, check 77, ***found***

The number of comparisons necessary by search element:

> 20 – requires 1 comparison;
> 8 and 39 – requires 2 comparisons;
> 4, 9, 24, 54 – requires 3 comparisons and
> 7, 16, 38, 45, 77 – requires 4 comparisons

Summing the comparisons, needed to find all twelve items and dividing by 12, yielding 37/12 or approximately 3.08 comparisons per successful search on the average.

Example 2:

Let us illustrate binary search on the following 9 elements:

Index	*0*	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*
Elements	-15	-6	0	7	9	23	54	82	101

Solution:

The number of comparisons required for searching different elements is as follows:

1. If we are searching for x = 101: (Number of comparisons = 4)

low	high	mid
1	9	5
6	9	7
8	9	8
9	9	9
		found

2. Searching for x = 82: (Number of comparisons = 3)

low	high	mid
1	9	5
6	9	7
8	9	8
		found

3. Searching for x = 42: (Number of comparisons = 4)

low	high	mid
1	9	5
6	9	7
6	6	6
7	6	not found

4. Searching for x = -14: (Number of comparisons = 3)

low	high	mid
1	9	5
1	4	2
1	1	1
2	1	not found

Continuing in this manner the number of element comparisons needed to find each of nine elements is:

Index	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*
Elements	-15	-6	0	7	9	23	54	82	101
Comparisons	*3*	*2*	*3*	*4*	*1*	*3*	*2*	*3*	*4*

No element requires more than 4 comparisons to be found. Summing the comparisons needed to find all nine items and dividing by 9, yielding 25/9 or approximately 2.77 comparisons per successful search on the average.

There are ten possible ways that an un-successful search may terminate depending upon the value of x.

If x < a(1), a(1) < x < a(2), a(2) < x < a(3), a(5) < x < a(6), a(6) < x < a(7) or a(7) < x < a(8) the algorithm requires 3 element comparisons to determine that 'x' is not present. For all of the remaining possibilities BINSRCH requires 4 element comparisons.

Thus the average number of element comparisons for an unsuccessful search is:

$$(3 + 3 + 3 + 4 + 4 + 3 + 3 + 3 + 4 + 4) / 10 = 34/10 = 3.4$$

Time Complexity:

The time complexity of binary search in a successful search is O(log n) and for an unsuccessful search is O(log n).

7.2.1. A non-recursive program for binary search:

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

main()
{
	int number[25], n, data, i, flag = 0, low, high, mid;
	clrscr();
	printf("\n Enter the number of elements: ");
	scanf("%d", &n);
	printf("\n Enter the elements in ascending order: ");
	for(i = 0; i < n; i++)
		scanf("%d", &number[i]);
	printf("\n Enter the element to be searched: ");
	scanf("%d", &data);
	low = 0; high = n-1;
	while(low <= high)
	{
		mid = (low + high)/2;
		if(number[mid] == data)
		{
			flag = 1;
			break;
		}
		else
		{
			if(data < number[mid])
				high = mid - 1;
			else
				low = mid + 1;
		}
	}
	if(flag == 1)
		printf("\n Data found at location: %d", mid + 1);
	else
		printf("\n Data Not Found ");
}
```

7.2.2. A recursive program for binary search:

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

void bin_search(int a[], int data, int low, int high)
{
	int mid ;
	if( low <= high)
	{
		mid = (low + high)/2;
		if(a[mid] == data)
			printf("\n Element found at location: %d ", mid + 1);
		else
		{
			if(data < a[mid])
				bin_search(a, data, low, mid-1);
			else
```

```
                        bin_search(a, data, mid+1, high);
                }
        }
        else
                printf("\n Element not found");
}
void main()
{
        int a[25], i, n, data;
        clrscr();
        printf("\n Enter the number of elements: ");
        scanf("%d", &n);
        printf("\n Enter the elements in ascending order: ");
        for(i = 0; i < n; i++)
                scanf("%d", &a[i]);
        printf("\n Enter the element to be searched: ");
        scanf("%d", &data);
        bin_search(a, data, 0, n-1);
        getch();
}
```

7.3. Bubble Sort:

The bubble sort is easy to understand and program. The basic idea of bubble sort is to pass through the file sequentially several times. In each pass, we compare each element in the file with its successor i.e., X[i] with X[i+1] and interchange two element when they are not in proper order. We will illustrate this sorting technique by taking a specific example. Bubble sort is also called as exchange sort.

Example:

Consider the array x[n] which is stored in memory as shown below:

X[0]	X[1]	X[2]	X[3]	X[4]	X[5]
33	44	22	11	66	55

Suppose we want our array to be stored in ascending order. Then we pass through the array 5 times as described below:

Pass 1: (first element is compared with all other elements).

We compare X[i] and X[i+1] for i = 0, 1, 2, 3, and 4, and interchange X[i] and X[i+1] if X[i] > X[i+1]. The process is shown below:

X[0]	X[1]	X[2]	X[3]	X[4]	X[5]	Remarks
33	44	22	11	66	55	
	22	44				
		11	44			
			44	66		
				55	66	
33	22	11	44	55	66	

The biggest number 66 is moved to (bubbled up) the right most position in the array.

Pass 2: (second element is compared).

We repeat the same process, but this time we don't include X[5] into our comparisons. i.e., we compare X[i] with X[i+1] for i=0, 1, 2, and 3 and interchange X[i] and X[i+1] if X[i] > X[i+1]. The process is shown below:

X[0]	X[1]	X[2]	X[3]	X[4]	Remarks
33	22	11	44	55	
22	33				
	11	33			
		33	44		
			44	55	
22	11	33	44	55	

The second biggest number 55 is moved now to X[4].

Pass 3: (third element is compared).

We repeat the same process, but this time we leave both X[4] and X[5]. By doing this, we move the third biggest number 44 to X[3].

X[0]	X[1]	X[2]	X[3]	Remarks
22	11	33	44	
11	22			
	22	33		
		33	44	
11	22	33	44	

Pass 4: (fourth element is compared).

We repeat the process leaving X[3], X[4], and X[5]. By doing this, we move the fourth biggest number 33 to X[2].

X[0]	X[1]	X[2]	Remarks
11	22	33	
11	22		
	22	33	

Pass 5: (fifth element is compared).

We repeat the process leaving X[2], X[3], X[4], and X[5]. By doing this, we move the fifth biggest number 22 to X[1]. At this time, we will have the smallest number 11 in X[0]. Thus, we see that we can sort the array of size 6 in 5 passes.

For an array of size n, we required (n-1) passes.

7.3.1. Program for Bubble Sort:

```
#include <stdio.h>
#include <conio.h>
void bubblesort(int x[], int n)
{
	int i, j, temp;
	for (i = 0; i < n; i++)
	{
		for (j = 0; j < n–i-1 ; j++)
		{
			if (x[j] > x[j+1])
			{
				temp = x[j];
				x[j] = x[j+1];
				x[j+1] = temp;
			}
		}
	}
}

main()
{
	int i, n, x[25];
	clrscr();
	printf("\n Enter the number of elements: ");
	scanf("%d", &n);
	printf("\n Enter Data:");
	for(i = 0; i < n ; i++)
		scanf("%d", &x[i]);
	bubblesort(x, n);
	printf ("\n Array Elements after sorting: ");
	for (i = 0; i < n; i++)
		printf ("%5d", x[i]);
}
```

Time Complexity:

The bubble sort method of sorting an array of size n requires (n-1) passes and (n-1) comparisons on each pass. Thus the total number of comparisons is $(n-1) * (n-1) = n^2 - 2n + 1$, which is $O(n^2)$. Therefore bubble sort is very inefficient when there are more elements to sorting.

7.4. Selection Sort:

Selection sort will not require no more than n-1 interchanges. Suppose x is an array of size n stored in memory. The selection sort algorithm first selects the smallest element in the array x and place it at array position 0; then it selects the next smallest element in the array x and place it at array position 1. It simply continues this procedure until it places the biggest element in the last position of the array.

The array is passed through (n-1) times and the smallest element is placed in its respective position in the array as detailed below:

Pass 1: Find the location j of the smallest element in the array x [0], x[1], ... x[n-1], and then interchange x[j] with x[0]. Then x[0] is sorted.

Pass 2: Leave the first element and find the location j of the smallest element in the sub-array x[1], x[2], ... x[n-1], and then interchange x[1] with x[j]. Then x[0], x[1] are sorted.

Pass 3: Leave the first two elements and find the location j of the smallest element in the sub-array x[2], x[3], ... x[n-1], and then interchange x[2] with x[j]. Then x[0], x[1], x[2] are sorted.

Pass (n-1): Find the location j of the smaller of the elements x[n-2] and x[n-1], and then interchange x[j] and x[n-2]. Then x[0], x[1], ... x[n-2] are sorted. Of course, during this pass x[n-1] will be the biggest element and so the entire array is sorted.

Time Complexity:

In general we prefer selection sort in case where the insertion sort or the bubble sort requires exclusive swapping. In spite of superiority of the selection sort over bubble sort and the insertion sort (there is significant decrease in run time), its efficiency is also $O(n^2)$ for n data items.

Example:

Let us consider the following example with 9 elements to analyze selection Sort:

1	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*Remarks*
65	70	75	80	50	60	55	85	45	find the first smallest element
i								j	swap a[i] & a[j]
45	70	75	80	50	60	55	85	65	find the second smallest element
	i			j					swap a[i] and a[j]
45	**50**	75	80	70	60	55	85	65	Find the third smallest element
		i				j			swap a[i] and a[j]
45	**50**	**55**	80	70	60	75	85	65	Find the fourth smallest element
			i		j				swap a[i] and a[j]
45	**50**	**55**	**60**	70	80	75	85	65	Find the fifth smallest element
				i				j	swap a[i] and a[j]
45	**50**	**55**	**60**	**65**	80	75	85	70	Find the sixth smallest element
					i			j	swap a[i] and a[j]
45	**50**	**55**	**60**	**65**	**70**	75	85	80	Find the seventh smallest element
						i j			swap a[i] and a[j]
45	**50**	**55**	**60**	**65**	**70**	**75**	85	80	Find the eighth smallest element
							i	J	swap a[i] and a[j]
45	**50**	**55**	**60**	**65**	**70**	**75**	**80**	**85**	The outer loop ends.

7.4.1. Non-recursive Program for selection sort:

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

void selectionSort( int low, int high );

int a[25];

int main()
{
        int num, i= 0;
        clrscr();
        printf( "Enter the number of elements: " );
        scanf("%d", &num);
        printf( "\nEnter the elements:\n" );
        for(i=0; i < num; i++)
                scanf( "%d", &a[i] );
        selectionSort( 0, num - 1 );
        printf( "\nThe elements after sorting are: " );
        for( i=0; i< num; i++ )
                printf( "%d    ", a[i] );
        return 0;
}

void selectionSort( int low, int high )
{
        int i=0, j=0, temp=0, minindex;
        for( i=low; i <= high; i++ )
        {
                minindex = i;
                for( j=i+1; j <= high; j++ )
                {
                        if( a[j] < a[minindex] )
                                minindex = j;
                }
                temp = a[i];
                a[i] = a[minindex];
                a[minindex] = temp;
        }
}
```

7.4.2. Recursive Program for selection sort:

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

int x[6] = {77, 33, 44, 11, 66};
selectionSort(int);

main()
{
        int i, n = 0;
        clrscr();
        printf (" Array Elements before sorting: ");
        for (i=0; i<5; i++)
```

```
                printf ("%d ", x[i]);
        selectionSort(n);                       /* call selection sort */
        printf ("\n Array Elements after sorting: ");
        for (i=0; i<5; i++)
                printf ("%d ", x[i]);
}

selectionSort( int n)
{
        int k, p, temp, min;
        if (n== 4)
                return (-1);
        min = x[n];
        p = n;
        for (k = n+1; k<5; k++)
        {
                if (x[k] <min)
                {
                        min = x[k];
                        p = k;
                }
        }
        temp = x[n];            /* interchange x[n] and x[p] */
        x[n] = x[p];
        x[p] = temp;
        n++ ;
        selectionSort(n);
}
```

7.5. Quick Sort:

The quick sort was invented by Prof. C. A. R. Hoare in the early 1960's. It was one of the first most efficient sorting algorithms. It is an example of a class of algorithms that work by "divide and conquer" technique.

The quick sort algorithm partitions the original array by rearranging it into two groups. The first group contains those elements less than some arbitrary chosen value taken from the set, and the second group contains those elements greater than or equal to the chosen value. The chosen value is known as the *pivot* element. Once the array has been rearranged in this way with respect to the *pivot*, the same partitioning procedure is recursively applied to each of the two subsets. When all the subsets have been partitioned and rearranged, the original array is sorted.

The function partition() makes use of two pointers up and down which are moved toward each other in the following fashion:

1. Repeatedly increase the pointer 'up' until a[up] >= pivot.
2. Repeatedly decrease the pointer 'down' until a[down] <= pivot.
3. If down > up, interchange a[down] with a[up]
4. Repeat the steps 1, 2 and 3 till the 'up' pointer crosses the 'down' pointer. If 'up' pointer crosses 'down' pointer, the position for pivot is found and place pivot element in 'down' pointer position.

The program uses a recursive function quicksort(). The algorithm of quick sort function sorts all elements in an array 'a' between positions 'low' and 'high'.

1. It terminates when the condition low >= high is satisfied. This condition will be satisfied only when the array is completely sorted.

2. Here we choose the first element as the 'pivot'. So, pivot = x[low]. Now it calls the partition function to find the proper position j of the element x[low] i.e. pivot. Then we will have two sub-arrays x[low], x[low+1], ... x[j-1] and x[j+1], x[j+2], . . . x[high].

3. It calls itself recursively to sort the left sub-array x[low], x[low+1], ... x[j-1] between positions low and j-1 (where j is returned by the partition function).

4. It calls itself recursively to sort the right sub-array x[j+1], x[j+2], . . x[high] between positions j+1 and high.

The time complexity of quick sort algorithm is of ***O(n log n)***.

Algorithm

Sorts the elements a[p], ... ,a[q] which reside in the global array a[n] into ascending order. The a[n + 1] is considered to be defined and must be greater than all elements in a[n]; a[n + 1] = +

```
quicksort (p, q)
{
          if ( p < q ) then
          {
               call j = PARTITION(a, p, q+1);     // j is the position of the partitioning element
               call quicksort(p, j – 1);
               call quicksort(j + 1 , q);
           }
}

partition(a, m, p)
{
          v = a[m]; up = m; down = p;          // a[m] is the partition element
          do
          {
                    repeat
                              up = up + 1;
                    until (a[up] ≥ v);

                    repeat
                              down = down – 1;
                    until (a[down] ≤ v);
                    if (up < down) then call interchange(a, up,
           down); } while (up ≥ down);

          a[m] = a[down];
          a[down] = v;
          return (down);
}
```

```
interchange(a, up, down)
{
        p = a[up];
        a[up] = a[down];
        a[down] = p;
}
```

Example:

Select first element as the pivot element. Move 'up' pointer from left to right in search of an element larger than pivot. Move the 'down' pointer from right to left in search of an element smaller than pivot. If such elements are found, the elements are swapped.

This process continues till the 'up' pointer crosses the 'down' pointer. If 'up' pointer crosses 'down' pointer, the position for pivot is found and interchange pivot and element at 'down' position.

Let us consider the following example with 13 elements to analyze quick sort:

1	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*10*	*11*	*12*	*13*	*Remarks*
38	08	16	06	79	57	24	56	02	58	04	70	45	
pivot				up						down			swap up & down
pivot				04						79			
pivot					up			down					swap up & down
pivot					02			57					
pivot						down	up						swap pivot & down
(24	08	16	06	04	02)	**38**	(56	57	58	79	70	45)	
pivot					down	up							swap pivot & down
(02	08	16	06	04)	**24**								
pivot, down	up												swap pivot & down
02	(08	16	06	04)									
	pivot	up		down									swap up & down
	pivot	04		16									
	pivot		down	Up									
	(06	04)	**08**	(16)									swap pivot & down
	pivot	down	up										
	(04)	**06**											swap pivot & down
	04 pivot, down, up												
				16 pivot, down, up									
(02	**04**	**06**	**08**	**16**	**24)**	38							

							(56	57	58	79	70	45**)**	
							pivot	up				down	swap up & down
							pivot	45				57	
							pivot	down	up				swap pivot & down
							(45)	**56**	(58	79	70	57)	
							45 pivot, down, up						swap pivot & down
									(58 pivot	79 up	70	57) down	swap up & down
										57		79	
										down	up		
									(57)	**58**	(70	79)	swap pivot & down
									57 pivot, down, up				
											(70	79)	
											pivot, down	up	swap pivot & down
											70		
												79 pivot, down, up	
							(45	**56**	**57**	**58**	**70**	**79)**	
02	**04**	**06**	**08**	**16**	**24**	**38**	**45**	**56**	**57**	**58**	**70**	**79**	

7.5.1. Recursive program for Quick Sort:

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

void quicksort(int, int);
int partition(int, int);
void interchange(int, int);
int array[25];

int main()
{
	int num, i = 0;
	clrscr();
	printf( "Enter the number of elements: " );
	scanf( "%d", &num);
	printf( "Enter the elements: " );
	for(i=0; i < num; i++)
		scanf( "%d", &array[i] );
	quicksort(0, num -1);
	printf( "\nThe elements after sorting are: " );
```

```
        for(i=0; i < num; i++)
                printf("%d ", array[i]);
        return 0;
}

void quicksort(int low, int high)
{
        int pivotpos;
        if( low < high )
        {
                pivotpos = partition(low, high + 1);
                quicksort(low, pivotpos - 1);
                quicksort(pivotpos + 1, high);
        }
}

int partition(int low, int high)
{
        int pivot = array[low];
        int up = low, down = high;

        do
        {
                do
                        up = up + 1;
                while(array[up] < pivot );

                do
                        down = down - 1;
                while(array[down] > pivot);

                if(up < down)
                        interchange(up, down);

        } while(up < down);
        array[low] = array[down];
        array[down] = pivot;
        return down;
}

void interchange(int i, int j)
{
        int temp;
        temp = array[i];
        array[i] = array[j];
        array[j] = temp;
}
```

Exercises

1. Write a recursive "C" function to implement binary search and compute its time complexity.

2. Find the expected number of passes, comparisons and exchanges for bubble sort when the number of elements is equal to "10". Compare these results with the actual number of operations when the given sequence is as follows: 7, 1, 3, 4, 10, 9, 8, 6, 5, 2.

3. An array contains "n" elements of numbers. The several elements of this array may contain the same number "x". Write an algorithm to find the total number of elements which are equal to "x" and also indicate the position of the first such element in the array.

4. When a "C" function to sort a matrix row-wise and column-wise. Assume that the matrix is represented by a two dimensional array.

5. A very large array of elements is to be sorted. The program is to be run on a personal computer with limited memory. Which sort would be a better choice: Heap sort or Quick sort? Why?

6. Here is an array of ten integers: 5 3 8 9 1 7 0 2 6 4
   Suppose we partition this array using quicksort's partition function and using 5 for the pivot. Draw the resulting array after the partition finishes.

7. Here is an array which has just been partitioned by the first step of quicksort: 3, 0, 2, 4, 5, 8, 7, 6, 9. Which of these elements could be the pivot? (There may be more than one possibility!)

8. Show the result of inserting 10, 12, 1, 14, 6, 5, 8, 15, 3, 9, 7, 4, 11, 13, and 2, one at a time, into an initially empty binary heap.

9. Sort the sequence 3, 1, 4, 5, 9, 2, 6, 5 using insertion sort.

Multiple Choice Questions

1. What is the worst-case time for serial search finding a single item in an array? [ ]
   A. Constant time C. Logarithmic time
   B. Quadratic time D. Linear time

2. What is the worst-case time for binary search finding a single item in an array? [ ]
   A. Constant time C. Logarithmic time
   B. Quadratic time D. Linear time

3. What additional requirement is placed on an array, so that binary search may be used to locate an entry? [ ]
   A. The array elements must form a heap.
   B. The array must have at least 2 entries
   C. The array must be sorted.
   D. The array's size must be a power of two.

4. Which searching can be performed recursively ? [ ]
   A. linear search C. Binary search
   B. both D. none

5. Which searching can be performed iteratively ? [ ]
   A. linear search C. Binary search
   B. both D. none

6. In a selection sort of n elements, how many times is the swap function called in the complete execution of the algorithm? [ ]
   A. 1 C. n – 1
   B. n^2 D. n log n

7. Selection sort and quick sort both fall into the same category of sorting algorithms. What is this category? [ ]
   A. O(n log n) sorts C. Divide-and-conquer sorts
   B. Interchange sorts D. Average time is quadratic

8. Suppose that a selection sort of 100 items has completed 42 iterations of the main loop. How many items are now guaranteed to be in their final spot (never to be moved again)? [ ]
   A. 21 C. 42
   B. 41 D. 43

9. When is insertion sort a good choice for sorting an array? [ ]
   A. Each component of the array requires a large amount of memory
   B. The array has only a few items out of place
   C. Each component of the array requires a small amount of memory
   D. The processor speed is fast

10. What is the worst-case time for quick sort to sort an array of n elements? [ ]
A. O(log n) C. O(n log n)
B. O(n) D. $O(n^2)$

11. Suppose we are sorting an array of eight integers using quick sort, and we have just finished the first partitioning with the array looking like this: [ ]
2 5 1 7 9 12 11 10 Which statement is correct?
A. The pivot could be either the 7 or the 9.
B. The pivot is not the 7, but it could be the 9.
C. The pivot could be the 7, but it is not the 9.
D. Neither the 7 nor the 9 is the pivot

12. What is the worst-case time for heap sort to sort an array of n elements? [ ]
A. O(log n) C. O(n log n)
B. O(n) D. $O(n^2)$

13. Suppose we are sorting an array of eight integers using heap sort, and we have just finished one of the reheapifications downward. The array now looks like this: 6 4 5 1 2 7 8 [ ]
How many reheapifications downward have been performed so far?
A. 1 C. 2
B. 3 or 4 D. 5 or 6

14. Time complexity of inserting an element to a heap of n elements is of the order of [ ]
A. $\log_2 n$ C. $n \log_2 n$
B. n^2 D. n

15. A min heap is the tree structure where smallest element is available at the [ ]
A. leaf C. intermediate parent
B. root D. any where

16. In the quick sort method , a desirable choice for the portioning element will be [ ]
A. first element of list C. median of list
B. last element of list D. any element of list

17. Quick sort is also known as [ ]
A. merge sort C. heap sort
B. bubble sort D. none

18. Which design algorithm technique is used for quick sort . [ ]
A. Divide and conqueror C. backtrack
B. greedy D. dynamic programming

19. Which among the following is fastest sorting technique (for unordered data) [ ]
A. Heap sort C. Quick Sort
B. Selection Sort D. Bubble sort

20. In which searching technique elements are eliminated by half in each pass . [ ]
A. Linear search C. Binary search
B. both D. none

21. Running time of Heap sort algorithm is -----. [ ]
A. $O(\log_2 n)$ C. O(n)
B. A. $O(n \log_2 n)$ D. $O(n^2)$

22. Running time of Bubble sort algorithm is -----. [ ]
A. O($\log_2 n$) C. O(n)
B. A. O(n $\log_2 n$) D. O(n^2)

23. Running time of Selection sort algorithm is -----. [ ]
A. O($\log_2 n$) C. O(n)
B. A. O(n $\log_2 n$) D. O(n^2)

24. The Max heap constructed from the list of numbers 30,10,80,60,15,55 is [ ]
A. 60,80,55,30,10,15 C. 80,55,60,15,10,30
B. 80,60,55,30,10,15 D. none

25. The number of swappings needed to sort the numbers 8,22,7,9,31,19,5,13 in ascending order using bubble sort is [ ]
A. 11 C. 13
B. 12 D. 14

26. Time complexity of insertion sort algorithm in best case is [ ]
A. O($\log_2 n$) C. O(n)
B. A. O(n $\log_2 n$) D. O(n^2)

27. Binary search algorithm performs efficiently on a [ ]
A. linked list C. array
B. both D. none

28. Which is a stable sort ? [ ]
A. Bubble sort C. Quick sort
B. Selection Sort D. none

29. Heap is a good data structure to implement [ ]
A. priority Queue C. linear queue
B. Deque D. none

30. Always Heap is a [ ]
A. complete Binary tree C. Full Binary tree
B. Binary Search Tree D. none