Unit IV
Searching and Sorting:

Searching is the process of finding specific element in given list of values. The searching is said to be either successful of unsuccessful based on whether the searched item is found or not.

There are 2 ways of searching in given list viz. linear search and binary search.

Linear or Sequential Search:

This is the simplest method of searching required data in given list. In this method, each element in the list is traversed and checked whether it is the required element or not. The search starts with the first element and goes on sequentially with next elements till the required element found.

```
procedure LINEAR_SEARCH (array, key)

   for each item in the array
      if match element == key
         return element's index
      end if
   end for

end procedure
```

```
#include <stdio.h>
int LINEAR_SEARCH(int inp_arr[], int size, int val)
{
    int i;
    for (i = 0; i < size; i++)
        if (inp_arr[i] == val)

            return i;
    return -1;
}

int main(void)
{
    int arr[] = { 10, 20, 30, 40, 50, 100, 70,90,80 };
    int key = 100;
    int size = sizeof(arr)/sizeof(arr[0]);
```

```
    int res = LINEAR_SEARCH(arr, size, key);
    if (res == -1)
    printf("ELEMENT NOT FOUND!!");
    else
    printf("Item is present at index %d", res);
    return 0;
}
```

Binary Search:

Binary search is the quickest and efficient algorithm for searching in a list. But the binary search requires the all the elements sorted in the list. In this searching, the key to be searched is identified and checked whether it falls in left sub list or right sub list of the middle of list, and further divides that sub list into further sub lists and search for the key.

Binary Search is a search algorithm that is used to find the position of an element (target value) in a sorted array. The array should be sorted prior to applying a binary search. Binary search is also known by these names, logarithmic search, binary chop, half interval search.

Working

The binary search algorithm works by comparing the element to be searched by the middle element of the array and based on this comparison follows the required procedure.

Case 1 − element = middle, the element is found return the index.

Case 2 − element > middle, search for the element in the sub-array starting from middle+1 index to n.

Case 3 − element < middle, search for element in the sub-array starting from 0 index to middle -1.

ALGORITHM:

Parameters inital_value , end_value

Step 1 : Find the middle element of array. using ,

middle = initial_value + end_value / 2 ;

Step 2 : If middle = element, return ‘element found’ and index.

Step 3 : if middle > element, call the function with end_value = middle - 1 .

Step 4 : if middle < element, call the function with start_value = middle + 1 .

Step 5 : exit.

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

int iterativeBinarySearch(int array[], int start_index, int end_index, int element)
{
  int middle;
  while (start_index <= end_index)
  {
    // middle = start_index + (end_index - start_index )/2;
    middle=(start_index+end_index) /2;

    if (array[middle] == element)
        return middle;

    if (array[middle] < element)
        start_index = middle + 1;
    else
        end_index = middle - 1;
  }
  return -1;
}
void main(){
  int arr[] = {1, 4, 7, 9, 16, 56, 70};
  int n = sizeof(arr)/sizeof(arr[0]);
  int element = 16;
  int found_index;
  clrscr();

   found_index = iterativeBinarySearch(arr, 0, n-1, element);

  if(found_index == -1 )
    printf("Element not found in the array ");

  else
    printf("Element %d found at index : %d",element, found_index);

}
```

Q. Explain Different Types of searches

Hashing and collision resolution:
Hashing is a technique used to store and retrieve data quickly. It is mainly used in searching applications. In hashing, a special function called a hash function is used to convert a key value into an index of an array. This index represents the location where the data will be stored.
A hash function takes the key as input and produces an integer value known as a hash value. This hash value determines the position of the element in the hash table.
The main advantage of hashing is that it allows very fast searching, insertion, and deletion operations. Ideally, hashing provides constant time complexity for searching.

Example
Suppose we have a hash table of size 10 and a key value 25.

If the hash function is h(key) = key mod table_size
Then h(25) = 25 mod 10 = 5. So the element will be stored in index position 5 of the table.
However, sometimes two different keys may produce the same hash value. This situation is called a collision.

Collision:
A collision occurs when two or more keys are mapped to the same location in the hash table. Since a single location can store only one element, a method must be used to resolve this conflict. These methods are called collision resolution techniques.
Example
Consider table size = 10
Hash function = key mod 10
Key 25 → 25 mod 10 = 5
Key 35 → 35 mod 10 = 5
Both elements try to store in index 5, so a collision occurs.
Collision Resolution Techniques: There are several methods used to resolve collisions. The most common methods are: Separate Chaining, Linear Probing, Quadratic Probing, and Double Hashing. The different Advantage of these techniques are it provides better distribution of keys, reduces clustering significantly, etc.

Q. Explain Hashing (5 marks)
Q. Explain Collision point (5 marks)
Q. Explain about hashing and collision point

Sorting:

Sorting is the process of arranging things in a specific order. It becomes easy to identify required data if the data is sorted. There are different sorting algorithms existing each used in a specific situation.

The sorting are classified into two types. Internal sorting and External sorting. The sorting in which the data is swapped and stored within the reserved memory are called internal sorting and those that cannot accommodate data within the reserved memory during sorting are called external sorting.

Selection Sort:

In this sorting, each element is selected and that selected element is compared with every element from its next element till end, and the smallest element is stored into selected indexed element.

```
begin selectionSort(list)

        select each element of list
        compare with all elements from next element
    if list[selected index] > list[comparing index]
          swap(list[selected index], list[comparing index])

end SelectionSort
```

```
#include<stdio.h>
#include<conio.h>
void selectionSort(int[],int);
void main()
{
        int a[]={5,12,7,4,1,3};
        int i;
        int n=sizeof(a)/sizeof(int);
        clrscr();
        selectionSort(a,n);
        for(i=0;i<n;i++)
                printf("%d ",a[i]);

}
```

```
void selectionSort(int a[],int n)
{
        int i,j,t;
        for(i=0;i<n-1;i++)
        for(j=i+1;j<n;j++)
                if(a[i]>a[j])
                {
                        t=a[i];
                        a[i]=a[j];
                        a[j]=t;
                }
}
```

Bubble Sort :

In this the heavier data values are moved down and the lighter values move up as an air bubble in the water, so it is called bubble sort.

In this each element is compared with its next element and the heavier element moved to later indexes and lighter elements move to previous indexes.

Algorithm:

```
begin BubbleSort(list)

   for all elements of list
      if list[i] > list[i+1]
         swap(list[i], list[i+1])


end BubbleSort
```

```
#include<stdio.h>
#include<conio.h>
void bubbleSort(int[],int);
void main()
{
        int a[]={5,12,7,4,1,3};
        int i;
        int n=sizeof(a)/sizeof(int);
        clrscr();
        bubbleSort(a,n);
        for(i=0;i<n;i++)
                printf("%d ",a[i]);
```

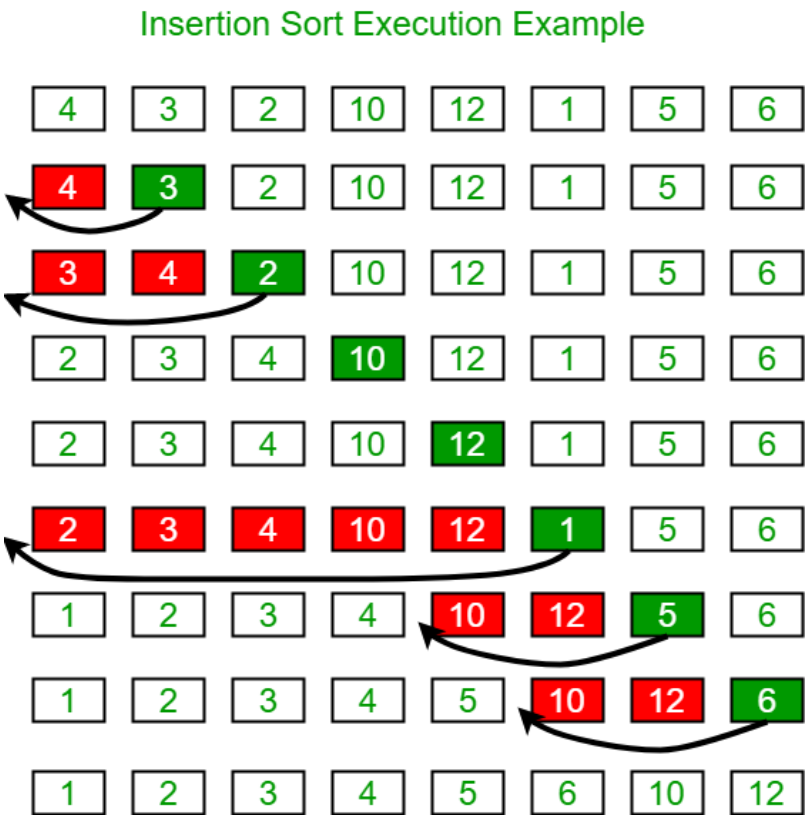
```
}
void bubbleSort(int a[],int n)
{
	int i,j,t;
	for(i=0;i<n-1;i++)
	for(j=0;j<n-i-1;j++)
		if(a[j]>a[j+1]) {
			t=a[j];
			a[j]=a[j+1];
			a[j+1]=t;
		}
}
```

Insertion Sort:

Insertion sort is a simple sorting algorithm that works similar to the way you sort playing cards in your hands. The array is virtually split into a sorted and an unsorted part. Values from the unsorted part are picked and placed at the correct position in the sorted part.

Algorithm

To sort an array of size n in ascending order:
1: Iterate from arr[1] to arr[n] over the array.
2: Compare the current element (key) to its predecessor.
3: If the key element is smaller than its predecessor, compare it to the elements before. Move the greater elements one position up to make space for the swapped element.




```
// C program for insertion sort
#include <math.h>
#include <stdio.h>
/* Function to sort an array using insertion sort*/
void insertionSort(int arr[], int n)
{
        int i, key, j;
        for (i = 1; i < n; i++)
         {
                key = arr[i];
                j = i - 1;

                /* Move elements of arr[0..i-1], that are
                greater than key, to one position ahead
                of their current position */

                while (j >= 0 && arr[j] > key) {
                        arr[j + 1] = arr[j];
                        j -- ;
                }
                arr[j + 1] = key;
        }
```

```
}

// A utility function to print an array of size n
void printArray(int arr[], int n)
{
        int i;
        for (i = 0; i < n; i++)
                printf("%d ", arr[i]);
        printf("\n");
}

/* Driver program to test insertion sort */
int main()
{
        int arr[] = { 12, 11, 13, 5, 6 };
        int n = sizeof(arr) / sizeof(arr[0]);

        insertionSort(arr, n);
        printArray(arr, n);
        return 0;
}
```

Quick Sort :

Quick sort is a highly efficient sorting algorithm and is based on partitioning of array of data into smaller arrays. A large array is partitioned into two arrays one of which holds values smaller than the specified value, say pivot, based on which the partition is made and another array holds values greater than the pivot value.

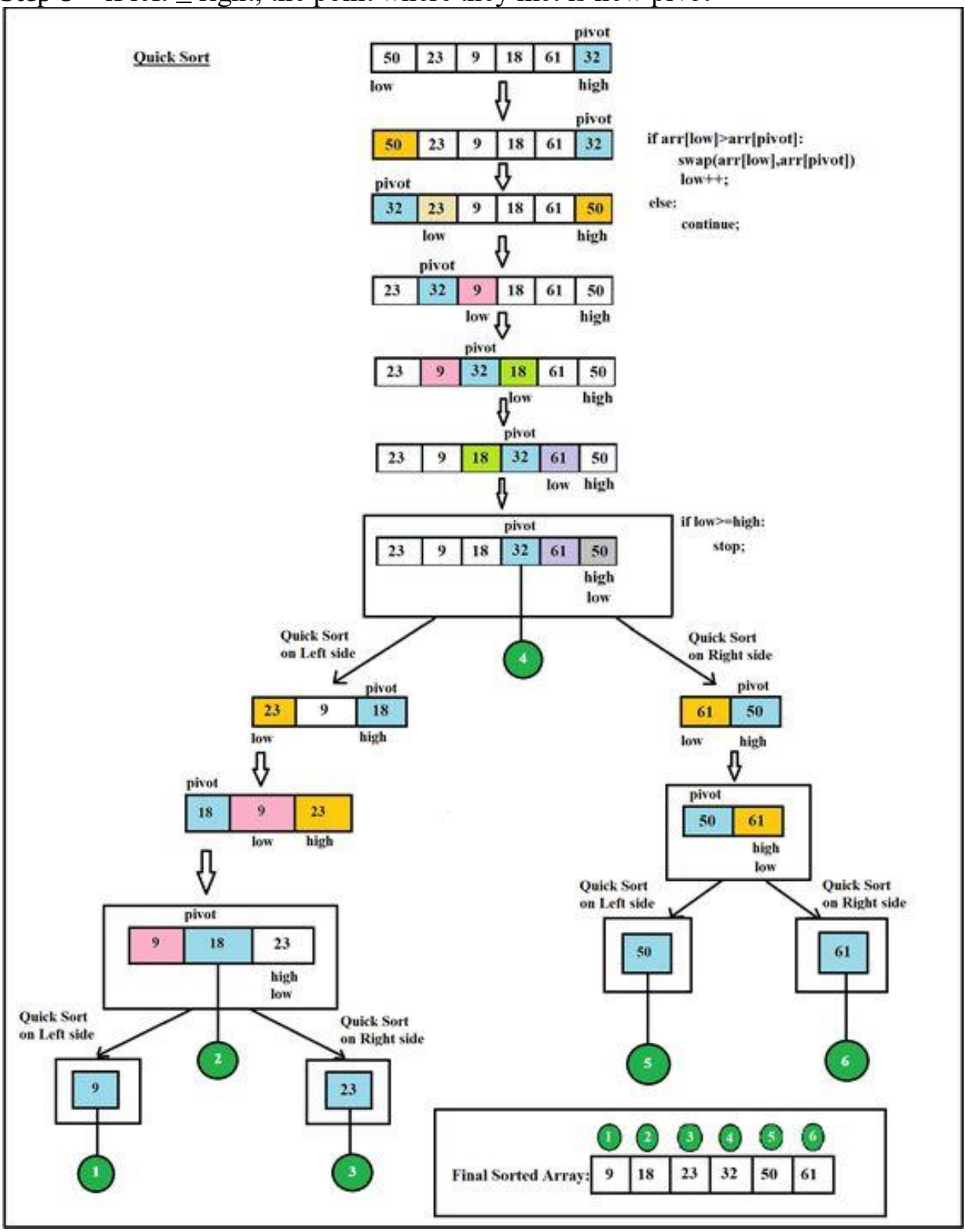
Quicksort partitions an array and then calls itself recursively twice to sort the two resulting subarrays. This algorithm is quite efficient for large-sized data sets as its average and worst-case complexity are $O(n^2)$, respectively.

Quick sort algorithm:
Step 1 − Make the right-most index value pivot
Step 2 − partition the array using pivot value
Step 3 − quicksort left partition recursively
Step 4 − quicksort right partition recursively

Algorithm for partitioning:

Step 1 − Choose the highest index value has pivot
Step 2 − Take two variables to point left and right of the list excluding pivot

Step 3 − left points to the low index
Step 4 − right points to the high
Step 5 − while value at left is less than pivot move right
Step 6 − while value at right is greater than pivot move left
Step 7 − if both step 5 and step 6 does not match swap left and right
Step 8 − if left ≥ right, the point where they met is new pivot



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

// to swap two numbers
void swap(int* a, int* b)
{
   int t = *a;
   *a = *b;
   *b = t;
}

int partition (int arr[], int low, int high)
{
   int pivot = arr[high];  // selecting last element as pivot
   int i = (low - 1);  // index of smaller element
   int j;

   for (j = low; j <= high- 1; j++)
   {
        // If the current element is smaller than or equal to pivot
        if (arr[j] <= pivot)
        {
           i++;    // increment index of smaller element
           swap(&arr[i], &arr[j]);
        }
   }
   swap(&arr[i + 1], &arr[high]);
   return (i + 1);
}
/*
   a[] is the array, p is starting index, that is 0,
   and r is the last index of array.
*/
void quicksort(int a[], int p, int r)
{
   int q;
   if(p < r)
```

```
    {

        q = partition(a, p, r);
        quicksort(a, p, q-1);
        quicksort(a, q+1, r);
    }
}


// function to print the array
void printArray(int a[], int size)
{
    int i;
    for (i=0; i < size; i++)
        printf("%d ", a[i]);
}

void main()
{
    int arr[] = {9, 7, 5, 11, 12, 2, 14, 3, 10, 6};
    int n = sizeof(arr)/sizeof(arr[0]);
    clrscr();
    // call quickSort function
    quicksort(arr, 0, n-1);

    printf("Sorted array: \n");
    printArray(arr, n);
}
```

Merge Sort :

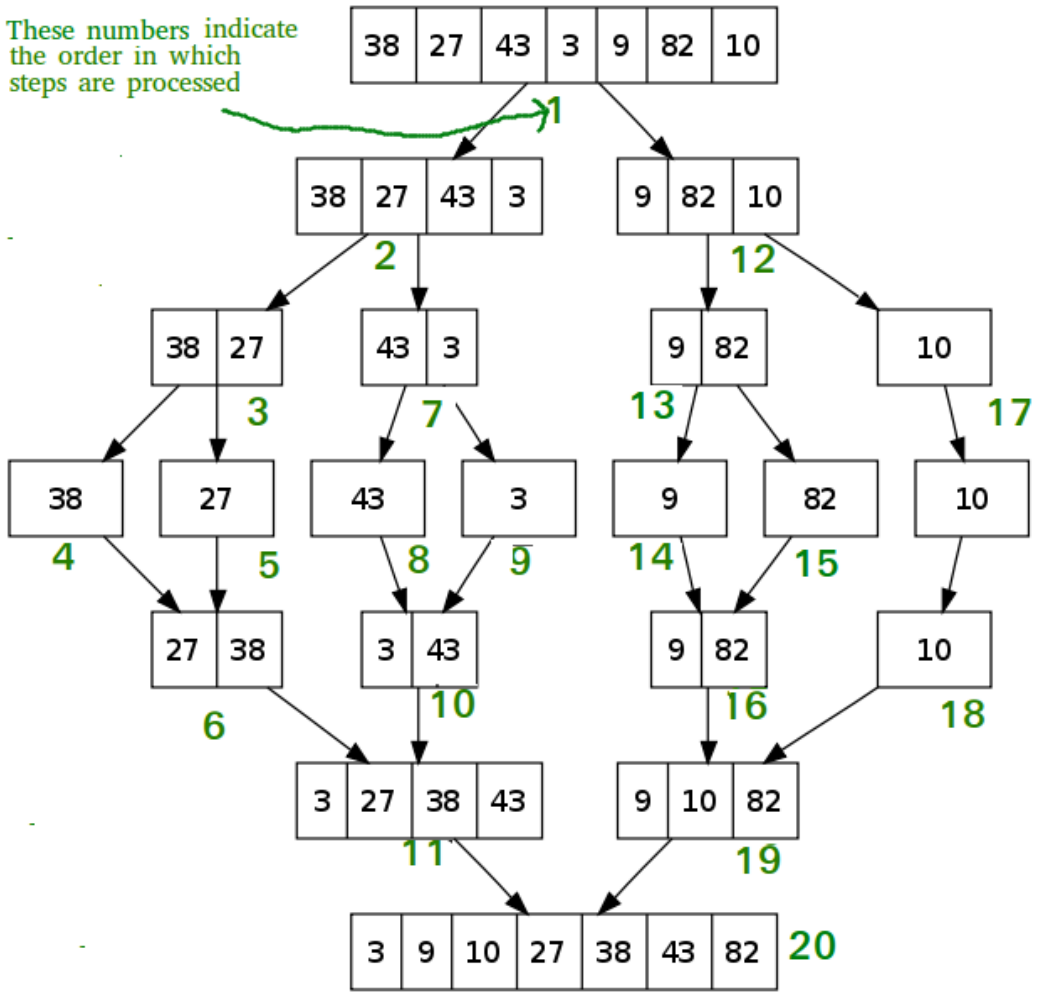
Like QuickSort, Merge Sort is a Divide and Conquer algorithm. It divides the input array into two halves, calls itself for the two halves, and then merges the two sorted halves. The merge() function is used for merging two halves. The merge(arr, l, m, r) is a key process that assumes that arr[l..m] and arr[m+1..r] are sorted and merges the two sorted sub-arrays into one. See the following C implementation for details.

Algorithm for merge sort

MergeSort(arr[], l, r)
If r > l

1. Find the middle point to divide the array into two halves:
middle m = l+ (r-l)/2

2. Call mergeSort for first half:
Call mergeSort(arr, l, m)
3. Call mergeSort for second half:
Call mergeSort(arr, m+1, r)
4. Merge the two halves sorted in step 2 and 3:
Call merge(arr, l, m, r)




```
/* C program for Merge Sort */
#include <stdio.h>
#include <stdlib.h>

// Merges two subarrays of arr[].
// First subarray is arr[l..m]
// Second subarray is arr[m+1..r]
void merge(int arr[], int l, int m, int r)
{
```

```
int i, j, k;
int n1 = m - l + 1;
int n2 = r - m;

/* create temp arrays */
int L[30], R[30];

/* Copy data to temp arrays L[] and R[] */
for (i = 0; i < n1; i++)
        L[i] = arr[l + i];
for (j = 0; j < n2; j++)
        R[j] = arr[m + 1 + j];

/* Merge the temp arrays back into arr[l..r]*/
i = 0; // Initial index of first subarray
j = 0; // Initial index of second subarray
k = l; // Initial index of merged subarray
while (i < n1 && j < n2) {
        if (L[i] <= R[j]) {
                arr[k] = L[i];
                i++;
        }
        else {
                arr[k] = R[j];
                j++;
        }
        k++;
}

/* Copy the remaining elements of L[], if there
are any */
while (i < n1) {
        arr[k] = L[i];
        i++;
        k++;
}

/* Copy the remaining elements of R[], if there
are any */
while (j < n2) {
        arr[k] = R[j];
        j++;
        k++;
```

```
        }
}


/* l is for left index and r is right index of the
sub-array of arr to be sorted */
void mergeSort(int arr[], int l, int r)
{
        if (l < r) {
                // Same as (l+r)/2, but avoids overflow for
                // large l and h
                int m = l + (r - l) / 2;

                // Sort first and second halves
                mergeSort(arr, l, m);
                mergeSort(arr, m + 1, r);

                merge(arr, l, m, r);
        }
}

/* UTILITY FUNCTIONS */
/* Function to print an array */
void printArray(int A[], int size)
{
        int i;
        for (i = 0; i < size; i++)
                printf("%d ", A[i]);
        printf("\n");
}

/* Driver code */
int main()
{
        int arr[] = { 12, 11, 13, 5, 6, 7 };
        int arr_size = sizeof(arr) / sizeof(arr[0]);
        clrscr();
        printf("Given array is \n");
        printArray(arr, arr_size);

        mergeSort(arr, 0, arr_size - 1);

        printf("\nSorted array is \n");
```

```
    printArray(arr, arr_size);
    return 0;
}
```