Experiment No. 3
To implement Merge Sort
Name : Vaidehi D. Gadag
Branch/Div.: Comps-1 (C47)
Date of Performance: 22/02/2024
Date of Submission: 29/02/2024

Experiment No. 3

Title: Merge Sort

Aim: To study, implement and Analyze Merge Sort Algorithm

Objective: To introduce the methods of designing and analyzing algorithms

Theory:

The merge sort algorithm closely follows the divide-and-conquer paradigm. Intuitively, it operates as follows:

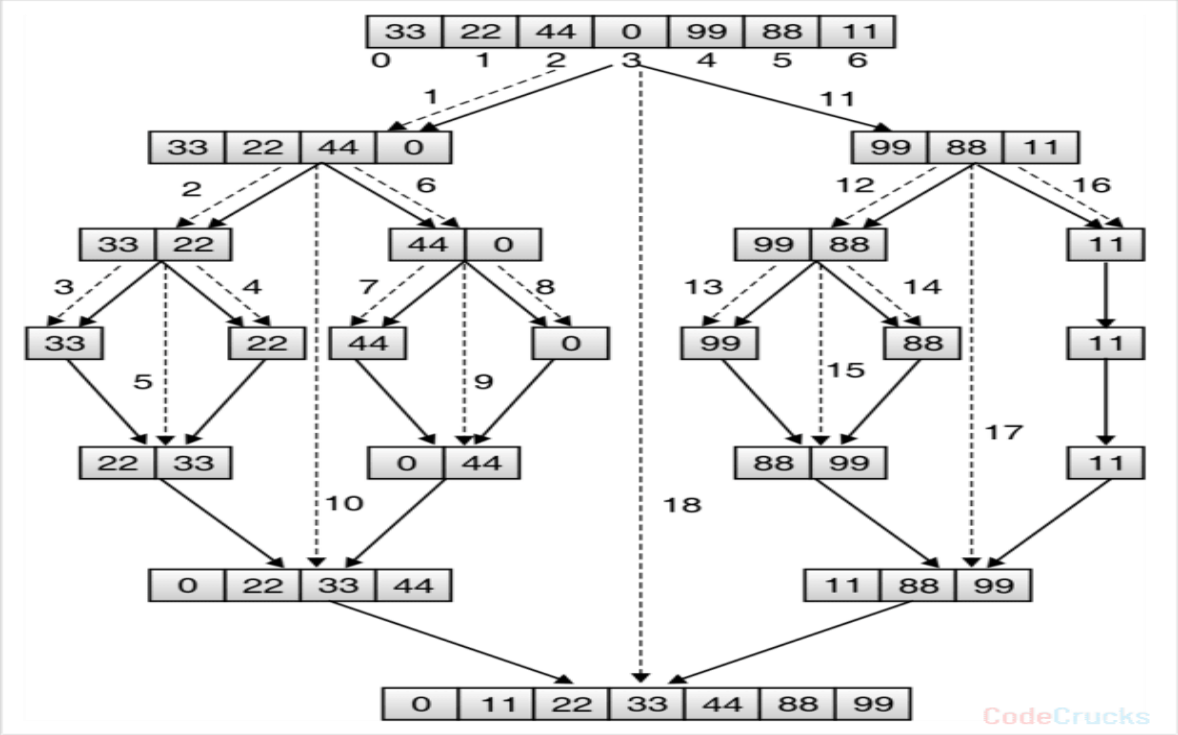
1. Divide: Divide the n-element sequence to be sorted into two sub sequences of n=2 elements each.
2. Conquer: Sort the two sub sequences recursively using merge sort.
3. Combine: Merge the two sorted sub sequences to produce the sorted answer.

During the Merge sort process the object in the collection are divided into two collections. To split a collection, Merge sort will take the middle of the collection and split the collection into its left and its right part. The resulting collections are again recursively sorted via the Merge sort algorithm.

Once the sorting process of the two collections is finished, the result of the two collections is combined. To combine both collections Merge sort start at each collection at the beginning. It pick the object which is smaller and inserts this object into the new collection. For this collection it now selects the next elements and selects the smaller element from both collection.

Once all elements from both collections have been inserted in the new collection, Merge sort has successfully sorted the collection. To avoid the creation of too many collections, typically one new collection is created and the left and right side are treated as different collections.

Example: Sort the sequence <33,22,44,0,99,88,11> using Merge Sort



Algorithm and Complexity:

```
1   Algorithm MergeSort1(low, high)
2   // The global array a[low : high] is sorted in nondecreasing order
3   // using the auxiliary array link[low : high]. The values in link
4   // represent a list of the indices low through high giving a[ ] in
5   // sorted order. A pointer to the beginning of the list is returned.
6   {
7       if ((high − low) < 15) then
8           return InsertionSort1(a, link, low, high);
9       else
10      {
11          mid := ⌊(low + high)/2⌋;
12          q := MergeSort1(low, mid);
13          r := MergeSort1(mid + 1, high);
14          return Merge1(q, r);
15      }
16  }
```

Algorithm 3.10 Merge sort using links

```
1   Algorithm Merge1(q, r)
2   // q and r are pointers to lists contained in the global array
3   // link[0 : n]. link[0] is introduced only for convenience and need
4   // not be initialized. The lists pointed at by q and r are merged
5   // and a pointer to the beginning of the merged list is returned.
6   {
7       i := q; j := r; k := 0;
8       // The new list starts at link[0].
9       while ((i ≠ 0) and (j ≠ 0)) do
10      { // While both lists are nonempty do
11          if (a[i] ≤ a[j]) then
12          { // Find the smaller key.
13              link[k] := i; k := i; i := link[i];
14              // Add a new key to the list.
15          }
16          else
17          {
18              link[k] := j; k := j; j := link[j];
19          }
20      }
21      if (i = 0) then link[k] := j;
22      else link[k] := i;
23      return link[0];
24  }
```

Algorithm 3.11 Merging linked lists of sorted elements

Recurrence Relation for Merger Sort:

T(n) = 1 for n=1

T(n) = 2T(n/2) + n for n>1… (1)

Solve by Substitution method:

Solving original recurrence for n/2,

T(n/2) = 2T(n/4) + n/2

Substituting this in equation (1),

T(n) = 2[2T(n/4) + n/2] + n

= 2^2 T(n/2^2) + 2n .

$T(n) = 2^k T(n/2^k) + k.n$ … (2)

Let us consider that k grows up to $\log_2 n$,

Let $n/2^k = 1$

$n = 2^k$

$k = \log_2 n$

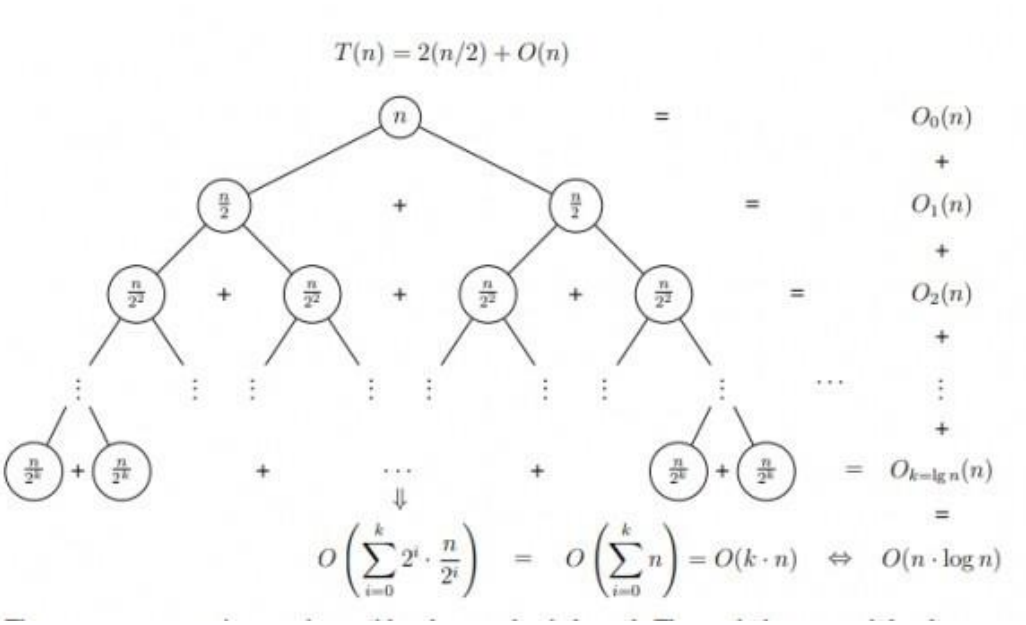
$n = 2^k$

Substitute these values in equation (2)

$T(n) = nT(n/n) + \log_2 n \, . \, n$

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

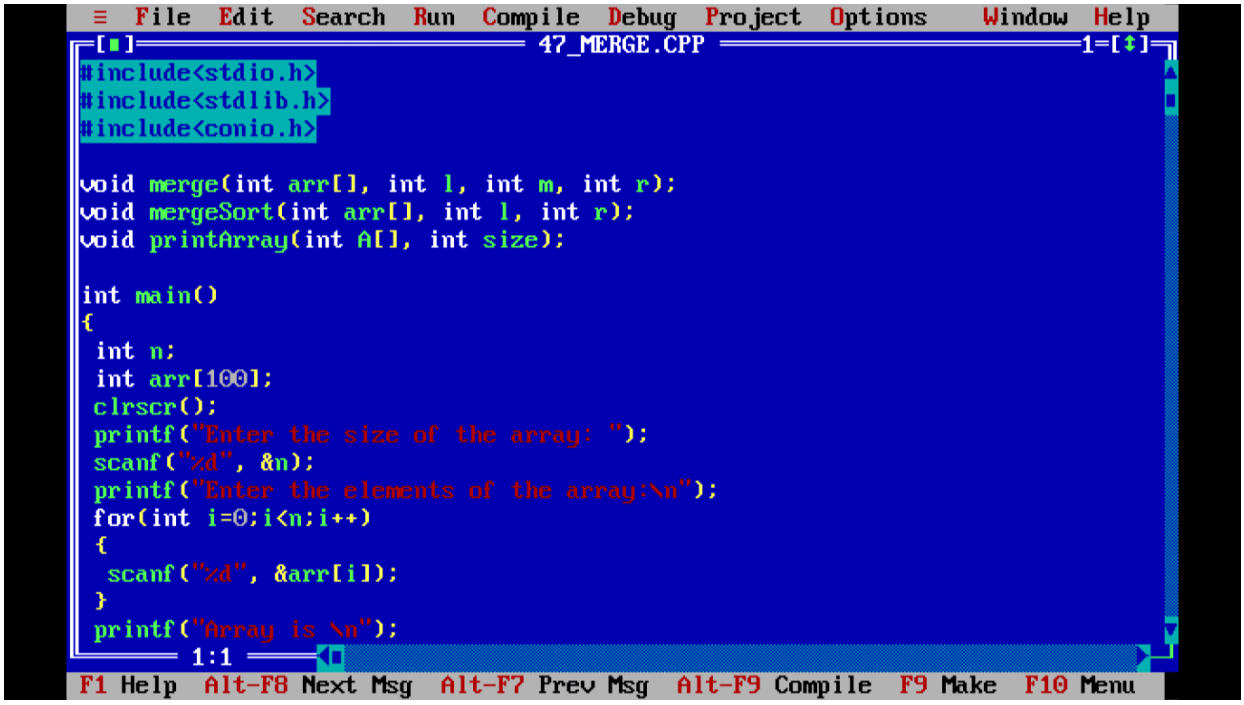
Solve using recursive tree Method:

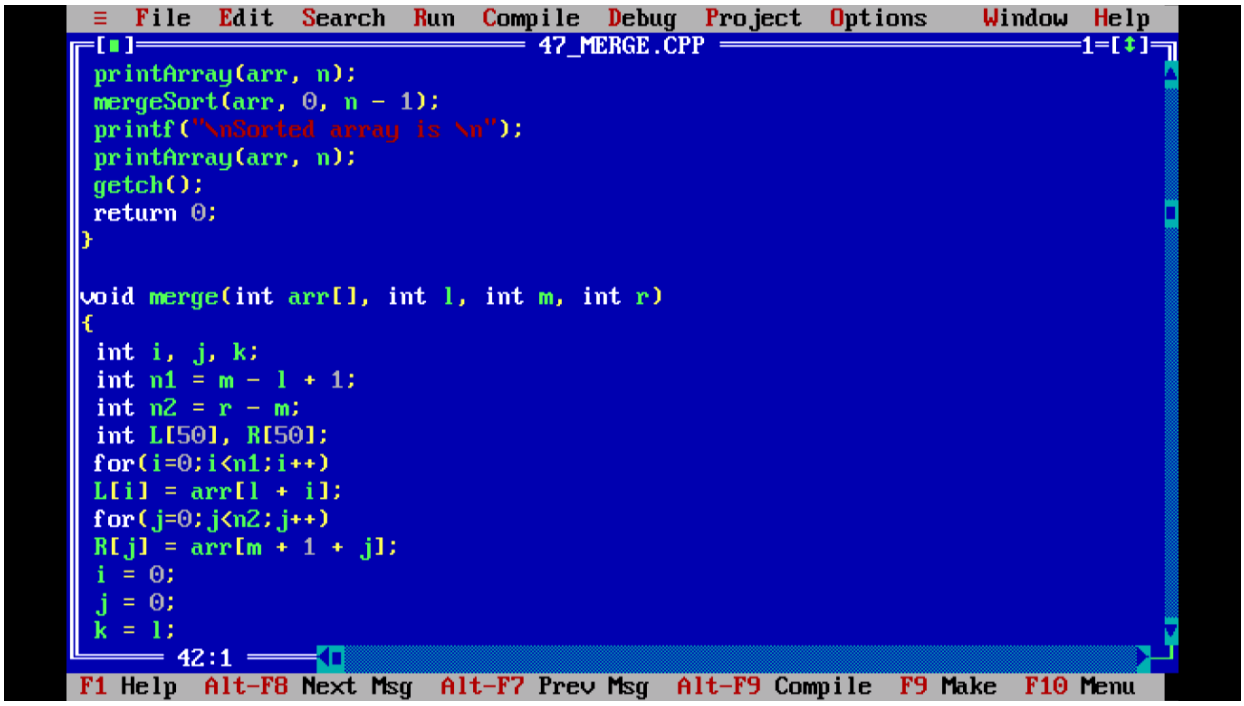


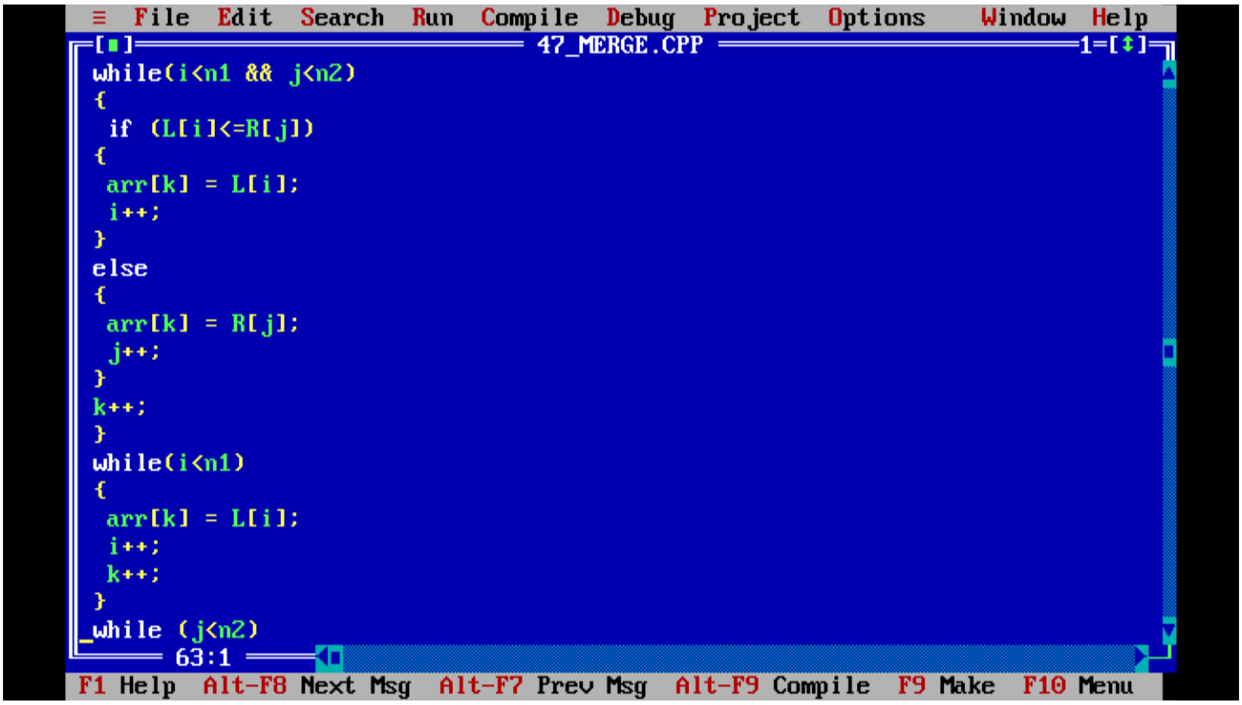
Code:

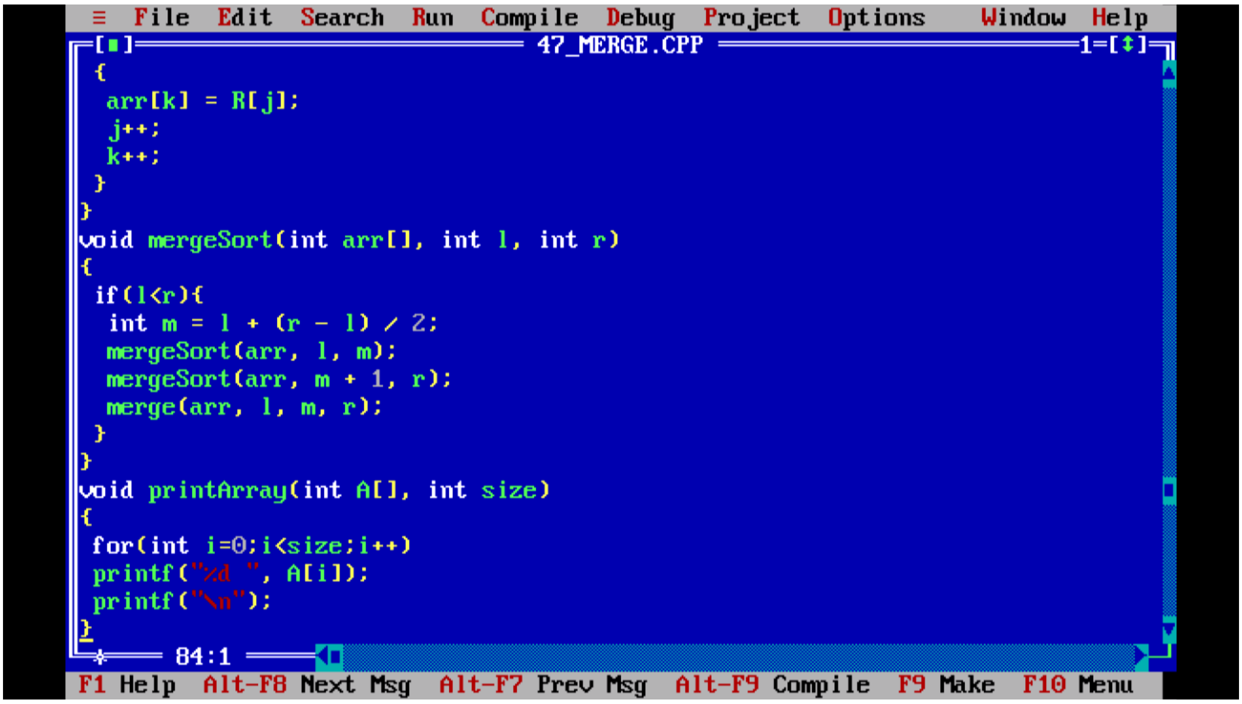
```
#include<stdio.h>
#include<stdlib.h>
#include<conio.h>

void merge(int arr[], int l, int m, int r);
void mergeSort(int arr[], int l, int r);
void printArray(int A[], int size);

int main()
{
 int n;
 int arr[100];
 clrscr();
 printf("Enter the size of the array: ");
 scanf("%d", &n);
 printf("Enter the elements of the array:\n");
 for(int i=0;i<n;i++)
 {
  scanf("%d", &arr[i]);
 }
 printf("Array is \n");
 printArray(arr, n);
 mergeSort(arr, 0, n - 1);
 printf("\nSorted array is \n");
 printArray(arr, n);
 getch();
 return 0;
}

void merge(int arr[], int l, int m, int r)
{
 int i, j, k;
 int n1 = m - l + 1;
 int n2 = r - m;
 int L[50], R[50];
 for(i=0;i<n1;i++)
 L[i] = arr[l + i];
 for(j=0;j<n2;j++)
 R[j] = arr[m + 1 + j];
 i = 0;
 j = 0;
 k = l;
 while(i<n1 && j<n2)
 {
```

```
 if (L[i]<=R[j])
 {
  arr[k] = L[i];
  i++;
 }
 else
 {
  arr[k] = R[j];
  j++;
 }
 k++;
 }
 while(i<n1)
 {
  arr[k] = L[i];
  i++;
  k++;
 }
 while (j<n2)
 {
  arr[k] = R[j];
  j++;
  k++;
 }
}

void mergeSort(int arr[], int l, int r)
{
 if(l<r){
  int m = l + (r - l) / 2;
  mergeSort(arr, l, m);
  mergeSort(arr, m + 1, r);
  merge(arr, l, m, r);
 }
}

void printArray(int A[], int size)
{
 for(int i=0;i<size;i++)
 printf("%d ", A[i]);
 printf("\n");
}
```


File Edit Search Run Compile Debug Project Options Window Help
47_MERGE.CPP
#include<stdio.h>
#include<stdlib.h>
#include<conio.h>
void merge(int arr[], int l, int m, int r);
void mergeSort(int arr[], int l, int r);
void printArray(int A[], int size);
int main()
{
int n;
int arr[100];
clrscr();
printf("Enter the size of the array: ");
scanf("%d", &n);
printf("Enter the elements of the array:\n");
for(int i=0;i<n;i++)
{
scanf("%d", &arr[i]);
}
printf("Array is \n");
1:1
F1 Help Alt-F8 Next Msg Alt-F7 Prev Msg Alt-F9 Compile F9 Make F10 Menu


File Edit Search Run Compile Debug Project Options Window Help
47_MERGE.CPP
printArray(arr, n);
mergeSort(arr, 0, n - 1);
printf("\nSorted array is \n");
printArray(arr, n);
getch();
return 0;
}
void merge(int arr[], int l, int m, int r)
{
int i, j, k;
int n1 = m - l + 1;
int n2 = r - m;
int L[50], R[50];
for(i=0;i<n1;i++)
L[i] = arr[l + i];
for(j=0;j<n2;j++)
R[j] = arr[m + 1 + j];
i = 0;
j = 0;
k = l;
42:1
F1 Help Alt-F8 Next Msg Alt-F7 Prev Msg Alt-F9 Compile F9 Make F10 Menu

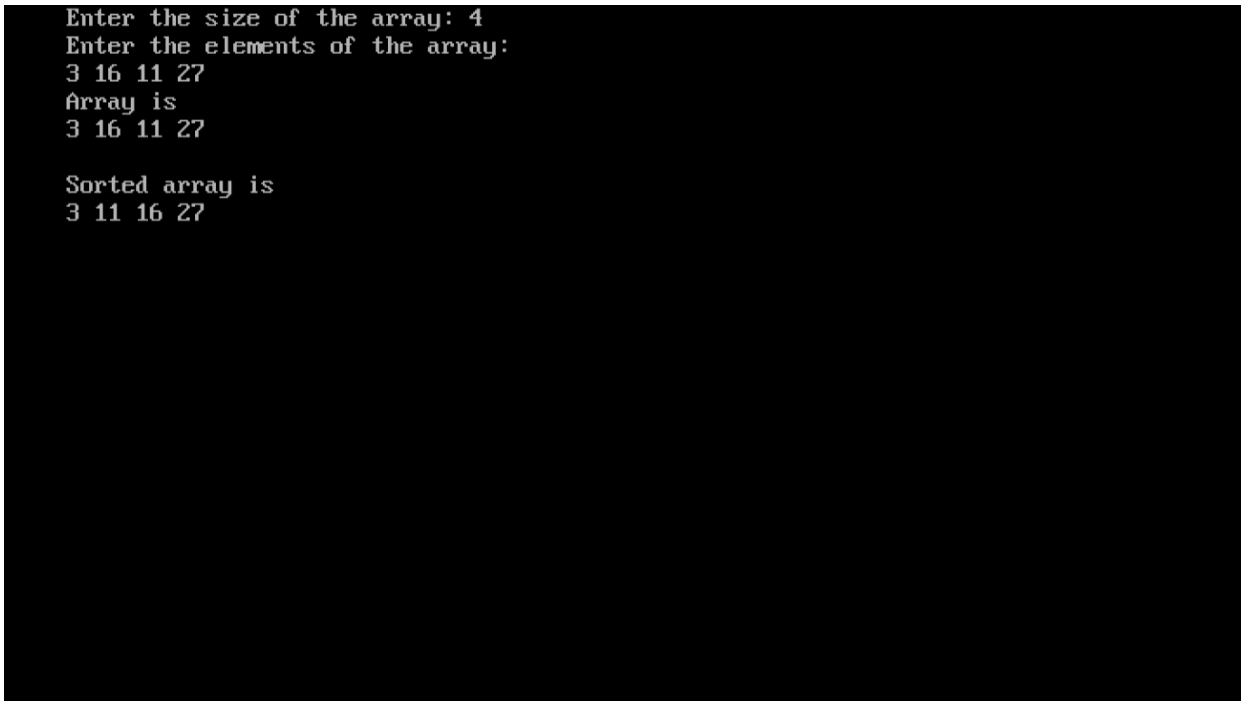

≡ File Edit Search Run Compile Debug Project Options Window Help
47_MERGE.CPP
while(i<n1 && j<n2)
{
if (L[i]<=R[j])
{
arr[k] = L[i];
i++;
}
else
{
arr[k] = R[j];
j++;
}
k++;
}
while(i<n1)
{
arr[k] = L[i];
i++;
k++;
}
while (j<n2)
63:1
F1 Help Alt-F8 Next Msg Alt-F7 Prev Msg Alt-F9 Compile F9 Make F10 Menu


≡ File Edit Search Run Compile Debug Project Options Window Help
47_MERGE.CPP
{
arr[k] = R[j];
j++;
k++;
}
}
void mergeSort(int arr[], int l, int r)
{
if(l<r){
int m = l + (r - l) / 2;
mergeSort(arr, l, m);
mergeSort(arr, m + 1, r);
merge(arr, l, m, r);
}
}
void printArray(int A[], int size)
{
for(int i=0;i<size;i++)
printf("%d ", A[i]);
printf("\n");
}
84:1
F1 Help Alt-F8 Next Msg Alt-F7 Prev Msg Alt-F9 Compile F9 Make F10 Menu

Output:

Enter the size of the array: 4
Enter the elements of the array:
3 16 11 27
Array is
3 16 11 27

Sorted array is
3 11 16 27



Conclusion:

In conclusion, merge sort is an efficient and stable sorting algorithm that utilizes the divide-and-conquer paradigm. It recursively divides the input array into smaller subarrays, sorts them individually, and then merges them back together to form the final sorted array. Merge sort has a time complexity of O(n log n) and is suitable for large datasets. However, it requires additional memory for temporary storage during the merge process, which can be a drawback for systems with limited memory.