**PRACTICAL – 1**

**Implement Binary searching with and without recursion. Also display its execution time.**

**About Binary search:**

Binary Search is a searching algorithm for finding an element's position in a sorted array. In this approach, the element is always searched in the middle of a portion of an array. Binary search compares the target value to the middle element of the array. If they are not equal, the half in which the target cannot lie is eliminated and the search continues on the remaining half, again taking the middle element to compare to the target value, and repeating this until the target value is found. If the search ends with the remaining half being empty, the target is not in the array.

Binary search runs in [logarithmic time](https://en.wikipedia.org/wiki/Time_complexity#Logarithmic_time) in the [worst case](https://en.wikipedia.org/wiki/Best,_worst_and_average_case), making {\displaystyle O(\log n)}comparisons, where {\displaystyle n} is the number of elements in the array. Binary search is faster than [linear search](https://en.wikipedia.org/wiki/Linear_search) except for small arrays.

Binary Search Algorithm can be implemented in the following two ways

1. Iterative Method
2. Recursive Method

**Iteration Method**

binarySearch(arr, x, low, high)

repeat till low = high

mid = (low + high)/2

if (x == arr[mid])

return mid

else if (x > arr[mid]) // x is on the right side

low = mid + 1

else // x is on the left side

high = mid – 1

**Recursive Method (The recursive method follows the divide and conquers approach)**

binarySearch(arr, x, low, high)

if low > high

return False

else

mid = (low + high) / 2

if x == arr[mid]

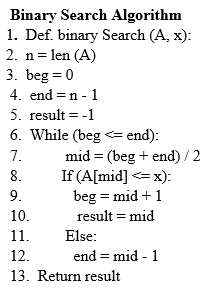
return mid

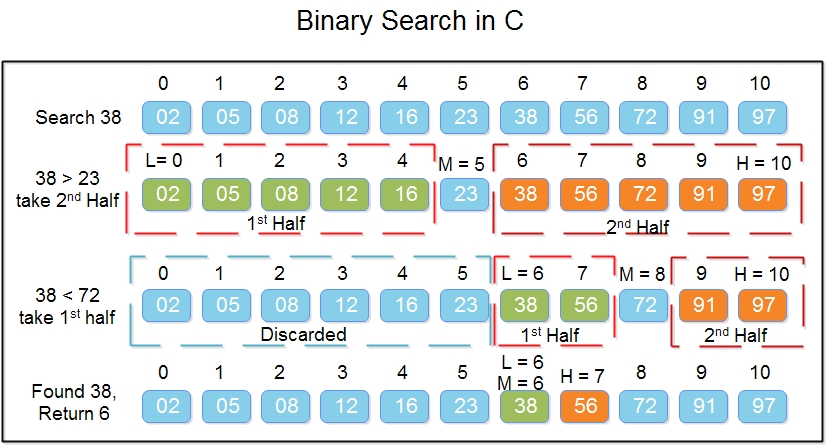
else if x > arr[mid] // x is on the right side

return binarySearch(arr, x, mid + 1, high)

else // x is on the right side

return binarySearch(arr, x, low, mid - 1)





**C Code:**

#include <stdio.h>

#include<sys/time.h>

#define MAX\_LEN 10

struct timeval tt;

struct timezone tz;

void b\_search\_nonrecursive(int l[],int num,int ele)

{

int l1,i,j, flag = 0;

l1 = 0;

i = num-1;

while(l1 <= i)

{

j = (l1+i)/2;

if( l[j] == ele)

{

printf("\nThe element %d is present at position %d in list\n",ele,j+1);

flag =1;

break;}

else

if(l[j] < ele)

l1 = j+1;

else

i = j-1;

}

if( flag == 0)

printf("\nThe element %d is not present in the list\n",ele);

}

int b\_search\_recursive(int l[],int arrayStart,int arrayEnd,int a){

int m,pos;

if (arrayStart<=arrayEnd){

m=(arrayStart+arrayEnd)/2;

if (l[m]==a)

return m;

else if (a<l[m])

return b\_search\_recursive(l,arrayStart,m-1,a);

else

return b\_search\_recursive(l,m+1,arrayEnd,a)}

return -1;

}

int main()

{

int l[MAX\_LEN], num, ele,f,l1,a,i;

int ch,pos,t1,t2,t3,t4;

printf("\nEnter the number of elements : ");

scanf("%d",&num);

printf("\nEnter the elements:\n");

for(i=0;i<num;i++)

scanf("%d",&l[i]);

printf("\nElements present in the list are:\n\n");

for(i=0;i<num;i++)

printf("%d\t",l[i]);

printf("\n\nEnter the element you want to search:\n\n");

scanf("%d",&ele);

printf("\nRecursive method:\n");

gettimeofday(&tt,&tz);

t1=tt.tv\_usec;

pos=b\_search\_recursive(l,0,num,ele);

gettimeofday(&tt,&tz);

t2=tt.tv\_usec;

if(pos==-1){

printf("Element is not found");

}

else{

printf("The element %d is present at position %d in list\n",ele,pos+1);

}

printf("\n\t Time taken is %d\n",t2-t1);

printf("\nNon-Recursive method:\n");

gettimeofday(&tt,&tz);

t3=tt.tv\_usec;

b\_search\_nonrecursive(l,num,ele);

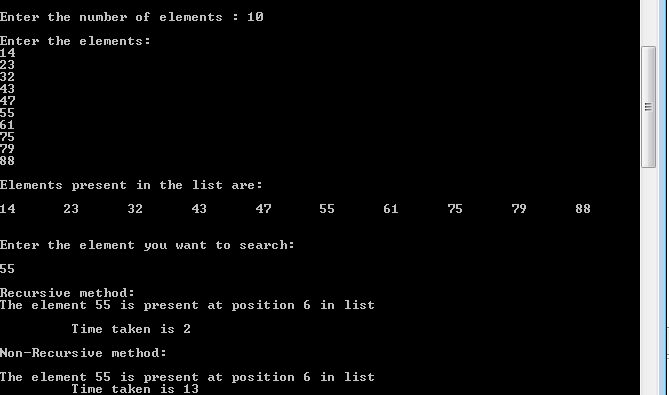
gettimeofday(&tt,&tz);

t4=tt.tv\_usec;

printf("\t Time taken is %d\n",t4-t3);

}

**Input/Output:**



**PRACTICAL – 2**

**Implement a program which finds minimum number from a given array with and without recursion. Also display its execution time.**

**About minimum number:**

As name suggest the program is used for find the minimum number in the given array.

Minimum number can be implemented in the following ways like with recursion and without recursion as below:

**Recursive approach to find the Minimum element in the array**

**Approach:**

* Get the array for which the minimum is to be found
* Recursively find the minimum according to the following:
  + Recursively traverse the array from the end
  + **Base case:** If the remaining array is of length 1, return the only present element i.e. arr[0]

if(n == 1)

return arr[0];

* **Recursive call:** If the base case is not met, then call the function by passing the array of one size less from the end, i.e. from arr[0] to arr[n-1].
* **Return statement:** At each recursive call (except for the base case), return the minimum of the last element of the current array (i.e. arr[n-1]) and the element returned from the previous recursive call.

return min(arr[n-1], recursive\_function(arr, n-1));

* Print the returned element from the recursive function as the minimum element

**Pseudocode for Recursive function:**

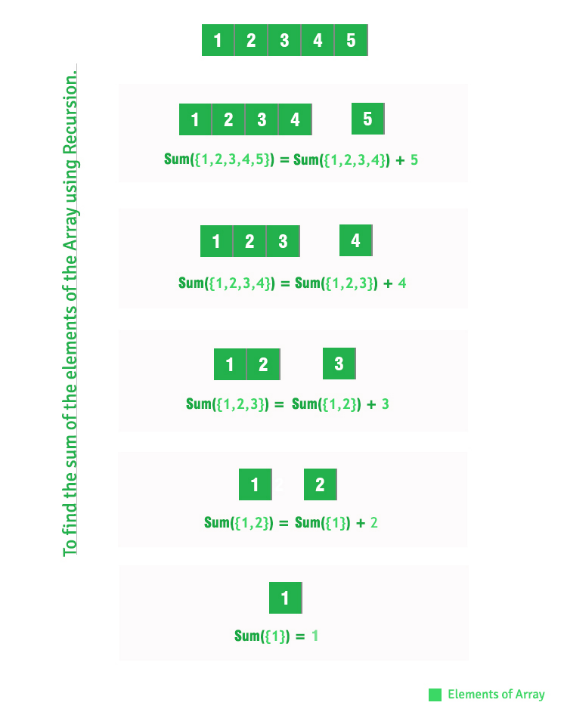
If there is single element, return it.

Else return minimum of following.

a) Last Element

b) Value returned by recursive call

for n-1 elements.



**C Code:**

//190450131018

//find minimum using DAC

#include<stdio.h>

#include<sys/time.h>

int a[100];

struct timeval tt;

struct timezone tz;

int minw(int A[], int n)

{

int min=A[0],i;

for(i=0;i<n;i++)

{

if(A[i] < min)

min = A[i];

}

return min;}

int recmin(int A[], int beg, int end){

int mid, min1, min2;

if(beg==end)

return A[beg];

else if(beg==end-1)

{

if(A[beg]>A[end])

return A[end];

else

return A[beg];

}

else{

mid=(beg+end)/2;

min1 = recmin(A, beg, mid);

min2 = recmin(A, mid+1, end);

if(min1>min2)

return min2;

else

return min1;}}

int main(){

int n,i;

int max,min,min1,t1,t2,t3,t4;

printf("Enter the number of elements in array:");

scanf("%d",&n);

printf("\nEnter the elements:\n");

for(i=0;i<n;i++)

{

scanf("%d",&a[i]);

}

gettimeofday(&tt,&tz);

t1=tt.tv\_usec;

min1 = minw(a,n);

gettimeofday(&tt,&tz);

t2=tt.tv\_usec;

printf("\nMin value without recursion:%d",min1);

printf("\t Time taken is %d\n",t2-t1);

gettimeofday(&tt,&tz);

t3=tt.tv\_usec;

min=recmin(a,0,n-1);

gettimeofday(&tt,&tz);

t4=tt.tv\_usec;

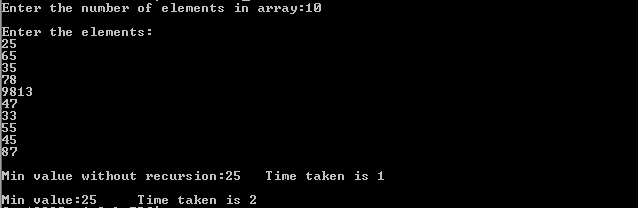
printf ("\nMin value:%d",min);

printf ("\t Time taken is %d\n",t4-t3);

return 0;

}

**Input/Output:**



**PRACTICAL – 3**

**Implement Quick sort with and without recursion. Also display its execution time.**

**About Quick search:**

Quick sort is a sorting algorithm that uses the divide and conquer strategy. In this method division is dynamically carried out. the three steps of quick sort are as follows:

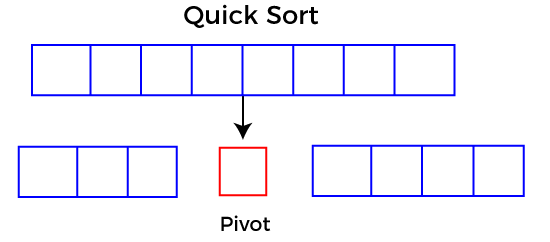
**Divide:** In Divide, first pick a pivot element. After that, partition or rearrange the array into two sub-arrays such that each element in the left sub-array is less than or equal to the pivot element and each element in the right sub-array is larger than the pivot element.

**Conquer:** Recursively sort the two-sub array.

**Combine:** Combine the already sorted array.

Quicksort picks an element as pivot, and then it partitions the given array around the picked pivot element. In quick sort, a large array is divided into two arrays in which one holds values that are smaller than the specified value (Pivot), and another array holds the values that are greater than the pivot.

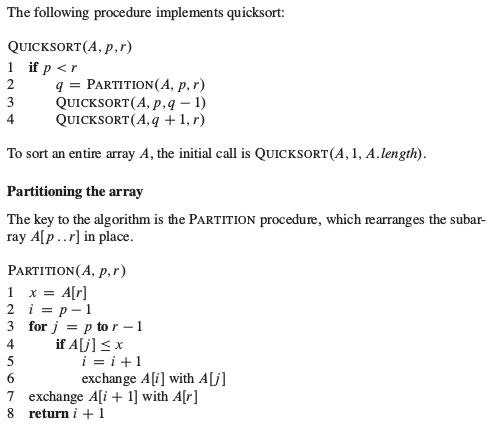
After that, left and right sub-arrays are also partitioned using the same approach. It will continue until the single element remains in the sub-array.

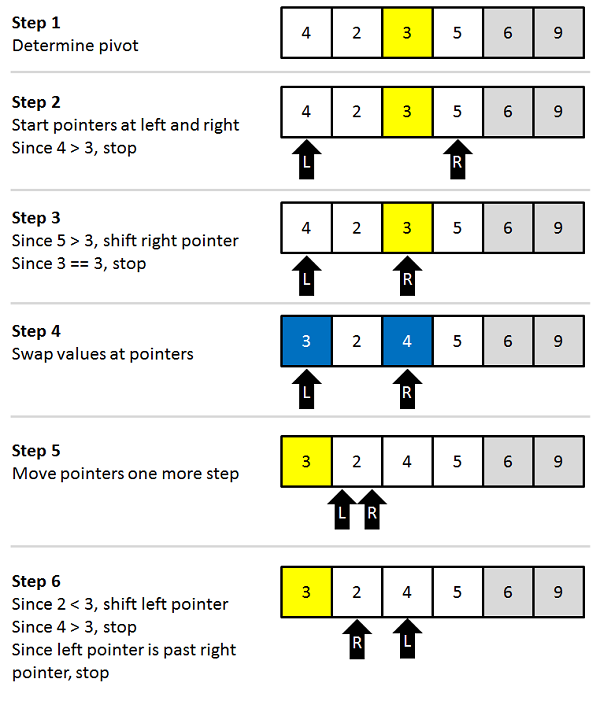


## Choosing the pivot

Picking a good pivot is necessary for the fast implementation of quicksort. However, it is typical to determine a good pivot. Some of the ways of choosing a pivot are as follows -

* Pivot can be random, i.e., select the random pivot from the given array.
* Pivot can either be the rightmost element of the leftmost element of the given array.
* Select median as the pivot element.



****

**C Code:**

//190450131018

//quick sort

#include <stdio.h>

#include<sys/time.h>

struct timeval tt;

struct timezone tz;

void quick\_sort(int [],int ,int);

void quicksortr (int [], int, int);

int main(){

int list[50];

int size, i,t1,t2,t3,t4;

printf("Enter the number of elements: ");

scanf("%d", &size);

printf("Enter the elements to be sorted:\n");

for (i = 0; i < size; i++){

scanf("%d", &list[i]);

}

gettimeofday(&tt,&tz);

t1=tt.tv\_usec;

quicksortr(list, 0, size - 1);

gettimeofday(&tt,&tz);

t2=tt.tv\_usec;

printf("After applying Recursive quick sort\n");

for (i = 0; i < size; i++){

printf("%d ", list[i]); }

printf("\n");

printf("\n\t Time taken is %d\n",t2-t1);

gettimeofday(&tt,&tz);

t3=tt.tv\_usec;

quick\_sort(list,0,size-1);

gettimeofday(&tt,&tz);

t4=tt.tv\_usec;

printf("\nAfter applying non Recursive quick sort: ");

for(i=0;i<size;i++)

printf(" %d",list[i]);

printf("\n");

printf("\t Time taken is %d\n",t4-t3);

return 0;

}

void quick\_sort(int arr[],int low,int high) {

int temp,left,right,x,k;

if(low>=high)

return;

else

{

x=arr[low];

right=low+1;

left = high;

while(right<=left){

while(arr[right]<x && right <= high){

right ++;

}

while(arr[left]>x && left > low){

left--;

}

if(right<left){

temp=arr[right];

arr[right]=arr[left];

arr[left]=temp;

right++;

left--;

}}

arr[low]=arr[left];

arr[left]=x;

quick\_sort(arr,low,left-1);

quick\_sort(arr,left+1,high);

}}

void quicksortr(int list[], int low, int high){

int pivot, i, j, temp;

if (low < high) {

pivot = low;

i = low;

j = high;

while (i < j) {

while (list[i] <= list[pivot] && i <= high) {

i++;

}

while (list[j] > list[pivot] && j >= low){

j--;

}

if (i < j){

temp = list[i];

list[i] = list[j];

list[j] = temp;

} }

temp = list[j];

list[j] = list[pivot];

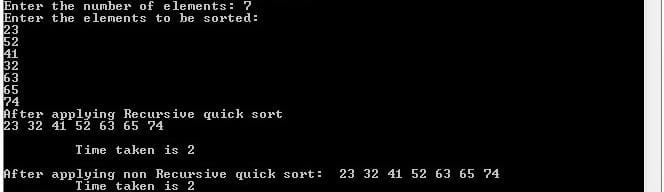
list[pivot] = temp;

quicksortr(list, low, j - 1);

quicksortr(list, j + 1, high);

}}

**Output:**



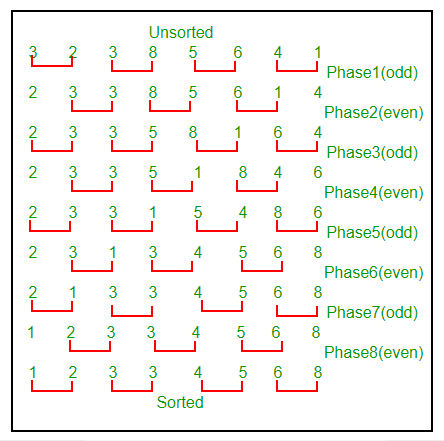
**Practical -8**

**Implement odd-even sorting in C.**

This is basically a variation of bubble-sort. This algorithm is divided into two phases- Odd and Even Phase. The algorithm runs until the array elements are sorted and in each iteration two phases occurs- Odd and Even Phases.

In the odd phase, we perform a bubble sort on odd indexed elements and in the even phase, we perform a bubble sort on even indexed elements.

We demonstrate the above algorithm using the below illustration on the array = {3, 2, 3, 8, 5, 6, 4, 1}



**C program code**

#include<stdio.h>

void main()

{

int i,flag,temp,size;

printf("Enter number of elements in array\n");

scanf("%d",&size);

int a[size];

printf("Enter Elements of Array for Odd Even Sort\n");

for(i=0;i<size;i++)

{

scanf("%d",&a[i]);

}

do{

flag=0;

//For Even Sort

for(i=0;i<size-1;i+=2)

{

//Swapping

if(a[i]>a[i+1])

{

temp=a[i];

a[i]=a[i+1];

a[i+1]=temp;

flag=1;

}

}

//For Odd Sort

for(i=1;i<size-1;i+=2)

{

//Swapping

if(a[i]>a[i+1])

{

temp=a[i];

a[i]=a[i+1];

a[i+1]=temp;

flag=1;

}

}

}while(flag);

printf("Array after Sorting\n");

for(i=0;i<size;i++)

{

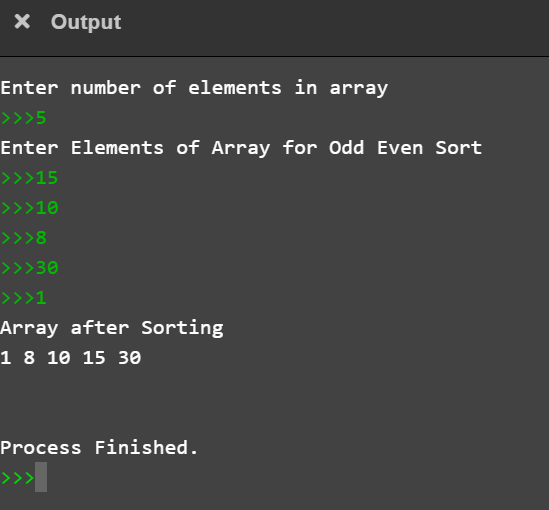
printf("%d ",a[i]);

}

printf("\n");

}

**Output**

****

**Practical -9**

**Implement bitonic sorting in C.**

# **Bitonic Sort Algorithm**

In this article, we will discuss the Bitonic sort Algorithm. Bitonic sort is a parallel sorting algorithm that performs O(n2log n) comparisons. Although the number of comparisons is more than that in any other popular sorting algorithm, it performs better for the parallel implementation because elements are compared in a predefined sequence that must not depend upon the data being sorted. The predefined sequence is called the Bitonic sequence.

To understand the bitonic sort, we first have to understand the **Bitonic sequence**.

In **Bitonic sequence,** elements are first arranged in increasing order, and then after some particular index, they start decreasing.

An array with A[0…i…n-1] is said to be bitonic, if there is an index i, such that –

A[0] < A[1] < A[2] .... A[i-1] < A[i] > A[i+1] > A[i+2] > A[i+3] > ... >A[n-1]

Where, 0 ≤ i ≤ n-1.

Before moving directly towards the algorithm of bitonic sort, first, understand the conversion of any random sequence into a bitonic sequence.

How to convert the random sequence into a bitonic sequence?

Consider a sequence A[ 0 ... n-1] of n elements. First, start constructing a Bitonic sequence by using 4 elements of the sequence. Sort the first 2 elements in ascending order and the last 2 elements in descending order, concatenate this pair to form a Bitonic sequence of 4 elements. Repeat this process for the remaining pairs of the element until we find a Bitonic sequence.

Let's understand the process to convert the random sequence into a bitonic sequence using an example.

Suppose the elements of array are - {30, 70, 40, 80, 60, 20, 10, 50}

Bitonic Sort

After conversion, the bitonic sequence that we will get is -

30, 40, 70, 80, 60, 50, 20, 10

Steps to perform Bitonic sort

The steps used to perform the bitonic sort are listed as follows -

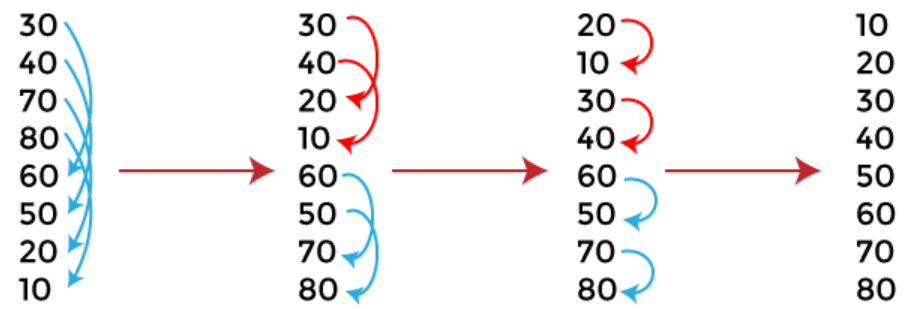
In first step, we have to create the bitonic sequence from the given random sequence as we have done above. It is considered as the first step of the process. After this step, we will get a sequence in which the elements in the first half are arranged in ascending order, while the elements of the second half are arranged in descending order.

Now, in the second step, we have to compare the first element of first half with the first element of second half, then second element of first half with the second element of second half, and so on. Here, we have to swap the elements if any element in the second half is found to be smaller.

After the above step, we got all the elements in the first half smaller than all the elements in the second half. The compare and swap results into the two sequences of n/2 length each. Repeat the process performed in the second step recursively onto every sequence until we get a single sorted sequence of length n.

Now, let's see the entire procedure of bitonic sort with an example. It will be easier to understand the bitonic sort with an example as it makes the explanation clearer and easier.

In the below example, we are using the bitonic sequence given above that we created from a random sequence.



Bitonic Sort

Now, the given array is completely sorted.

**1. Time Complexity**

|  |  |
| --- | --- |
| **Case** | **Time Complexity** |
| **Best Case** | **O(log2n)** |
| **Average Case** | **O(log2n)** |
| **Worst Case** | **O(log2n)** |

The time complexity of bitonic sort is O(log2 n) in all three cases**.**

**2. Space Complexity**

|  |  |
| --- | --- |
| **Space Complexity** | **O(n log2n)** |
| **Stable** | **No** |

#include<stdio.h>

//200450131506

/\*In this function the parameter 'd' represents the sorting direction\*/

void exchange(int a[], int i, int j, int d)

{

int temp;

if (d==(a[i]>a[j]))

{

temp = a[i];

a[i] = a[j];

a[j] = temp;

}

}

void merge(int a[], int beg, int c, int d)

{

int k, i;

if (c > 1)

{

k = c/2;

for (i = beg; i < beg+k; i++)

exchange(a, i, i+k, d);

merge(a, beg, k, d);

merge(a, beg+k, k, d);

}

}

void bitonicSort(int a[],int beg, int c, int d)

{

int k;

if (c>1)

{

k = c/2;

bitonicSort(a, beg, k, 1); // sort in ascending order

bitonicSort(a, beg+k, k, 0); // sort in descending order

merge(a,beg, c, d); //merge the sequence in ascending order

}

}

/\* function to call the bitonicSort() function to sort the given array in ascending order \*/

void sort(int a[], int n, int order)

{

bitonicSort(a, 0, n, order);

}

void print(int a[], int n) //function to print array elements

{

int i;

for(i = 0; i < n; i++)

{

printf("%d ",a[i]);

}

}

int main()

{

int a[]= {30, 70, 40, 80, 60, 20, 10, 50};

int n = sizeof(a)/sizeof(a[0]);

int order = 1; //It means sorting in increasing order

printf("Before sorting array elements are - \n");

print(a, n);

sort(a, n, order);

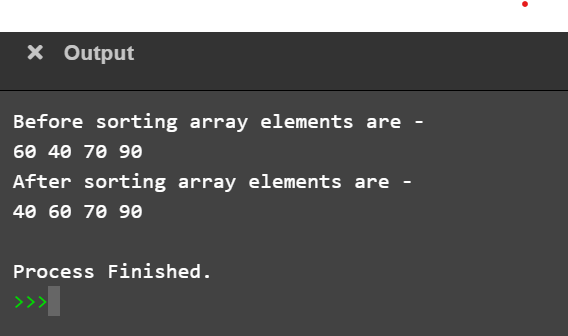
printf("\nAfter sorting array elements are - \n");

print(a, n);

return 0;

}

**Output**

****

**C code –**

//Bitonic Sequence Sorting - Flavour 2//

//20040131506

#include<stdio.h>

#include<conio.h>

#define MAX 8

int bitonic\_seq(int [], int n);

int main()

{

int i,j,k,temp;

int a[8]={56,26,15,12,40,30,76,60};

printf("Array before sorting \n"); for(j=0;j<MAX;j++)

{

printf("%d ", a[j]);

}

//PHASE 1

for(i=0;i<MAX;i++)

{

if(i==0||i==4)

{

if(a[i]>a[i+1])

{

temp=a[i];

a[i]=a[i+1];

a[i+1]=temp;

}

}

if(i==2||i==6)

{

if(a[i]<a[i+1])

{

temp=a[i];

a[i]=a[i+1];

a[i+1]=temp;

}

}

}

printf("\n\nPhase 1:");

for(i=0;i<MAX;i++)

{

printf(" %d ", a[i]);

}

//PHASE 2

for(i=0;i<MAX;i++)

{

if(i==0||i==1)

{

if(a[i]>a[i+2])

{

temp=a[i];

a[i]=a[i+2];

a[i+2]=temp;

}

}

if(i==4||i==5)

{

if(a[i]<a[i+2])

{

temp=a[i];

a[i]=a[i+2];

a[i+2]=temp;

}

}

}

printf("\n\nPhase 2:");

for(i=0;i<MAX;i++)

{

printf(" %d ", a[i]);

}

//PHASE 3

for(i=0;i<MAX;i++)

{

if(i<4)

{

if(a[i]>a[i+1])

{

temp=a[i];

a[i]=a[i+1];

a[i+1]=temp;

}

}

else

{

if(a[i]<a[i+1])

{

temp=a[i];

a[i]=a[i+1];

a[i+1]=temp;

}

}

}

printf("\n\nPhase 3:");

for(i=0;i<MAX;i++)

{

printf(" %d ", a[i]);

}

bitonic\_seq(a,8);

return 0;

}

//function for bitonic sequence

int bitonic\_seq(int a[], int n)

{

int i, temp;

//Phase 4

for(i=0;i<4;i++)

{

if(a[i]>a[i+4])

{

temp=a[i];

a[i]=a[i+4];

a[i+4]=temp;

}

}

printf("\n\nPhase 4:");

for(i=0;i<MAX;i++)

{

printf(" %d ", a[i]);

}

//Phase 5

for(i=0;i<6;i++)

{

if(a[i]>a[i+2])

{

temp=a[i];

a[i]=a[i+2];

a[i+2]=temp;

}

}

printf("\n\nPhase 5:");

for(i=0;i<MAX;i++)

{

printf(" %d ", a[i]);

}

//Phase 6

for(i=0;i<7;i++)

{

if(a[i]>a[i+1])

{

temp=a[i];

a[i]=a[i+1];

a[i+1]=temp;

}

}

printf("\n\nPhase 6:");

for(i=0;i<MAX;i++)

{

printf(" %d ", a[i]);

}

}

**Output :**

