**EXPERIMENT 3 : MERGE SORT**

**AIM:** Write a program to arrange an array in ascending order using Merge Sort Algorithm

**PSEUDOCODE:**

**void mergeSort(int \*Arr, int start, int end)**

**{**

**if(start < end)**

**{**

**int mid = (start + end) / 2;**

**mergeSort(Arr, start, mid);**

**mergeSort(Arr, mid+1, end);**

**merge(Arr, start, mid, end);**

**}**

**}**

**MergeSort(arr[], p, r)**

**If r > l**

1. **Find the middle point to divide the array into two halves:**

**middle m = p+ (r-p)/2**

**2. Call mergeSort for first half:**

**Call mergeSort(arr, p, 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, p, m, r)**

**EXAMPLE:**



**TIME COMPLEXITY:** Merge Sort is a recursive algorithm and time complexity can be expressed as following recurrence relation.   
**T(n) = 2T(n/2) + O(n)**

Time complexity of Merge Sort is  **O(nLogn)** in all 3 cases (worst, average and best) as merge sort always divides the array into two halves and takes linear time to merge two halves.

**PROG:**

#include<stdio.h>

#include<stdlib.h>

int \*mergeSort(int \*arr, int start, int end)

{

int size = end - start; // Gets the size of the current array

if(size == 1) return arr+start; // If only 1 element is present return that element

int mid = (start + end) / 2; // Get the mid index of the current array

// Divide

int \*lft = mergeSort(arr, start, mid); // Get the left sub-array

int \*rht = mergeSort(arr, mid, end); // Get the right sub-array

int l\_size = (mid - start); // Stores left sub-array size

int r\_size = (end - mid); // Stores right sub-array size

int \*left = malloc(sizeof(int) \* l\_size); // Stores left sub-array

int \*right = malloc(sizeof(int) \* r\_size); // Stores right sub-array

// Copies the content of the left and right sub-array into it's respective variable

for(int i=0;i<l\_size;i++)

{

left[i] = lft[i];

}

for(int i=0;i<r\_size;i++)

{

right[i] = rht[i];

}

// Merge

int l=0, r=0, a=start;

while(l < l\_size && r < r\_size)

{

if(left[l] < right[r])

{

arr[a] = left[l];

l++;

a++;

}

else

{

arr[a] = right[r];

r++;

a++;

}

}

if(l == l\_size)

{

while(r < r\_size)

{

arr[a] = right[r];

r++;

a++;

}

}

else

{

while(l < l\_size)

{

arr[a] = left[l];

l++;

a++;

}

}

return arr+start;

}

int main()

{

int size;

printf("Enter Size of array: ");

scanf("%d", &size);

int \*arr = malloc(sizeof(int) \* size);

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

{

printf("Enter %d Element: ", i+1);

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

}

mergeSort(arr, 0, size);

printf("\nSorted Array: ");

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

{

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

}

printf("\n");

return 0;

}

**EXPERIMENT 4 :FINDING MINIMUM AND MAXIMUM ELEMENT**

**AIM:** Write a program to determine minimum and maximum element in an array using Divide and Conquer Approach

**PSEUDOCODE:**

**Algo MaxMin(i, j, max, min)**

**{**

**if(i == j) //SMALL PROB**

**max = min = a[i]**

**elseif (i == j-1) //SMALL PROB**

**{ if(a[i] < a[j])**

**max = a[j] ; min = a[i];**

**else**

**max = a[i] ; min = a[j];**

**}**

**else //NOT SMALL PROB**

**{**

**mid = (i + j) / 2**

**MaxMin(i, mid, max, min)**

**MaxMin(mid+1, j, max1, min1)**

**if(max < max1)**

**max = max1**

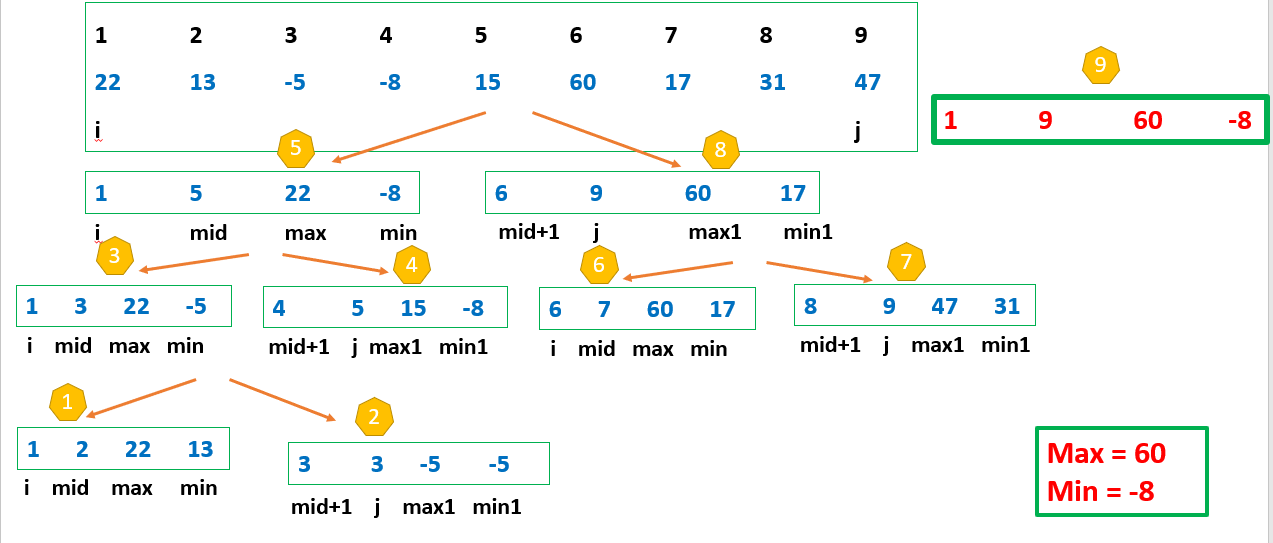
**if(min > min1)**

**min = min1**

**}**

**}**

**EXAMPLE:**



**PROGRAM:**

// Divide and conquer solution to find the minimum and maximum number in an array

void findMinAndMax(vector<int> const &nums, int low, int high, int &min, int &max)

{

    // if the array contains only one element

    if (low == high)                    // common comparison

    {

        if (max < nums[low]) {           // comparison 1

            max = nums[low];

        }

        if (min > nums[high]) {          // comparison 2

            min = nums[high];

        }

        return;

    }

    // if the array contains only two elements

    if (high - low == 1)                // common comparison

    {

        if (nums[low] < nums[high])       // comparison 1

        {

            if (min > nums[low]) {       // comparison 2

                min = nums[low];

            }

            if (max < nums[high]) {      // comparison 3

                max = nums[high];

            }

        }

        else {

            if (min > nums[high]) {      // comparison 2

                min = nums[high];

            }

            if (max < nums[low]) {       // comparison 3

                max = nums[low];

            }

        }

        return;

    }

    // find the middle element

    int mid = (low + high) / 2;

    // recur for the left subarray

    findMinAndMax(nums, low, mid, min, max);

    // recur for the right subarray

    findMinAndMax(nums, mid + 1, high, min, max);

}

int main()

{

    vector<int> nums = { 7, 2, 9, 3, 1, 6, 7, 8, 4 };

    // initialize the minimum element by INFINITY and the

    // maximum element by -INFINITY

    int max = INT\_MIN, min = INT\_MAX;

    int n = nums.size();

    findMinAndMax(nums, 0, n - 1, min, max);

    cout << "The minimum array element is " << min << endl;

    cout << "The maximum array element is " << max << endl;

    return 0;

}

**TIME COMPLEXITY: O(n)**