ASSIGNMENT 4 PROBLEM STATEMENT

Problem Statement : Implement Quick Sort and compare with Merge Sort. Tasks:

a. Use different pivot selection strategies

b. Compare execution time for sorted, reverse-sorted, and random data

QUICK SORT

ALGORITHM

Start

If low < high, then

Partition the array to find pivot position:

p ← PARTITION(A, low, high)

Recursively apply Quick Sort on left subarray:

QUICK_SORT(A, low, p − 1)

Recursively apply Quick Sort on right subarray:

QUICK_SORT(A, p + 1, high)

End If

Stop

CODE

```
// quick sort algorithm
#include <bits/stdc++.h>
using namespace std;
int divide(vector<int> &arr, int low, int high) {
    // Select last element as the pivot
    int pivot = arr[high];
    // Index of elemment just before the last element it is used for swapping
    int i = (low - 1);
    for(int j = low; j <= high - 1; j++) {
        // If current element is smaller than or equal to pivot, swap
```

```
        if (arr[j] <= pivot) {
            i++;
            swap(arr[i], arr[j]);
        }
    }
    // Put pivot to its position
    swap(arr[i + 1], arr[high]);
    // Return the point of partition
    return (i + 1);
}
void quickSort(vector<int> &arr, int low, int high) {

    if (low < high) {
        // pi is Partitioning Index
        int pi = divide(arr, low, high);
        // Separately sort elements before and after the Partition Index
        quickSort(arr, low, pi - 1);
        quickSort(arr, pi + 1, high);
    }
}
int main() {
    int n;
    cout<<"Enter number of array elements : \n";
    cin>>n;
    cout<<"Enter Array elements : \n";
    vector<int> arr(n);
    for(int i = 0; i < n; i++){
        cin>>arr[i];
    }
```

```
    quickSort(arr, 0, n - 1);
    cout<<"Sorted array : ";
    for (auto i : arr) {
        cout << i << " ";
    }
    return 0;
}
```

RESULT

Enter number of array elements : 5

Enter Array elements : 10 4 8 12 5

Sorted array : 4 5 8 10 12

COMPLEXITY

Memory: 3.432 Mb

Time: 0.0000 secs

Space Complexity:

Best Case	Worst Case	Average Case
O(log n)	O(n)	O(log n)

Time Complexity:

Best Case	Worst Case	Average Case
O(n log n)	$O(n^2)$	O(n log n)

MERGE SORT

ALGORITHM

```
BEGIN
  l1 ← low
  l2 ← mid + 1
  i ← low

  WHILE l1 ≤ mid AND l2 ≤ high DO
    IF arr[l1] ≤ arr[l2] THEN
      b[i] ← arr[l1]
      l1 ← l1 + 1
    ELSE
      b[i] ← arr[l2]
      l2 ← l2 + 1
    END IF
    i ← i + 1
  END WHILE
  WHILE l1 ≤ mid DO
    b[i] ← arr[l1]
    l1 ← l1 + 1
    i ← i + 1
  END WHILE
  WHILE l2 ≤ high DO
    b[i] ← arr[l2]
    l2 ← l2 + 1
    i ← i + 1
  END WHILE
  FOR i ← low TO high DO
    arr[i] ← b[i]
  END FOR
```

END

CODE

```
#include <iostream>
using namespace std;
int b[10];
void merging(int arr[], int low, int mid, int high){
  int l1, l2, i;
  for(l1 = low, l2 = mid + 1, i = low; l1 <= mid && l2 <= high; i++) {
    if(arr[l1] <= arr[l2])
      b[i] = arr[l1++];
    else
      b[i] = arr[l2++];
  }
  while(l1 <= mid)
    b[i++] = arr[l1++];
  while(l2 <= high)
    b[i++] = arr[l2++];
  for(i = low; i <= high; i++)
    arr[i] = b[i];
}
void sort(int arr[], int low, int high){
  int mid;
  if(low < high) {
    mid = (low + high) / 2;
    sort(arr, low, mid);
    sort(arr, mid + 1, high);
    merging(arr, low, mid, high);
  } else {
    return;
  }
```

```
}
int main(){
   int n;
   cout<<"Enter total number of elements :";
   cin>>n;
   cout<<"\nEnter "<<n<<" array elements : ";
   int arr[n];
   for(int i = 0; i < n; i++){
      cin>>arr[i];
   }
   cout << "\nArray before sorting\n";
   for(int i = 0; i < n; i++)
     cout<<arr[i]<<" ";
   sort(arr, 0, n - 1);
   cout<< "\nArray after sorting\n";
   for(int i = 0; i < n; i++)
     cout<<arr[i]<<" ";
}
```

ANALYZE COMPLEXITY

Memory: 3.436 Mb

Time: 0.0000 secs

Time Complexity

Best Case	Worst Case	Average Case
O(n log n)	O(n log n)	O(n log n)

Space Complexity

Auxiliary Case	Recursion Case
O(n)	O(n log n)

COMPARISON: MERGE SORT VS QUICK SORT

Parameter	Merge Sort	Quick Sort
Best Case Time	O(n log n)	O(n log n)
Average Case Time	O(n log n)	O(n log n)
Worst Case Time	O(n log n)	$O(n^2)$
Space Complexity	O(n) (extra temporary arrays)	O(log n) (recursion stack, in-place)
Stability	Stable	Not stable (by default)
In-place	No (needs extra memory)	Yes (in-place partitioning)
Performance on Sorted Data	Same O(n log n)	Can degrade to $O(n^2)$ with poor pivot
Practical Speed	Slightly slower (more memory operations)	Usually faster in practice