EXPERIMENT NO.02

Name:-Shivani Kolekar

Roll no:24141031

Batch: I2

Title: Merge Sort and Quick Sort with there time and space complexity.

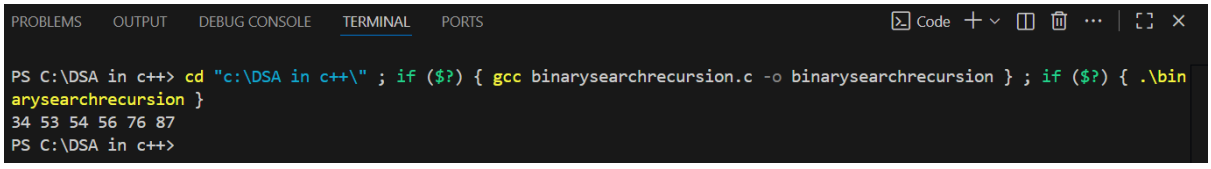
Merge Sort:

```
// C program for the implementation of merge sort
#include <stdio.h>
#include <stdlib.h>
// Merges two subarrays of arr[].
// First subarray is arr[left..mid]
// Second subarray is arr[mid+1..right]
void merge(int arr[], int left, int mid, int right) {
  int i, j, k;
  int n1 = mid - left + 1;
  int n2 = right - mid;
  // Create temporary arrays
  int leftArr[n1], rightArr[n2];
  // Copy data to temporary arrays
  for (i = 0; i < n1; i++)
    leftArr[i] = arr[left + i];
  for (j = 0; j < n2; j++)
    rightArr[j] = arr[mid + 1 + j];
  // Merge the temporary arrays back into arr[left..right]
  i = 0;
  j = 0;
  k = left;
```

```
    while (i < n1 && j < n2) {
        if (leftArr[i] <= rightArr[j]) {
            arr[k] = leftArr[i];
            i++;
        }
        else {
            arr[k] = rightArr[j];
            j++;
        }
        k++;
    }
    // Copy the remaining elements of leftArr[], if any
    while (i < n1) {
        arr[k] = leftArr[i];
        i++;
        k++;
    }
    // Copy the remaining elements of rightArr[], if any
    while (j < n2) {
        arr[k] = rightArr[j];
        j++;
        k++;
    }
}
// The subarray to be sorted is in the index range [left-right]
void mergeSort(int arr[], int left, int right) {
    if (left < right)
{
```

```
        // Calculate the midpoint
        int mid = left + (right - left) / 2
        // Sort first and second halves
        mergeSort(arr, left, mid);
        mergeSort(arr, mid + 1, right);

        // Merge the sorted halves
        merge(arr, left, mid, right);
    }
}

int main() {
    int arr[] =  {54,53,56,87,34,76
};
    int n = sizeof(arr) / sizeof(arr[0]);
            // Sorting arr using mergesort
    mergeSort(arr, 0, n - 1);
    for (int i = 0; i < n; i++)
        printf("%d ", arr[i]);
    return 0;
}
```

OUTPUT:



Applications:

1. Data Processing

- Large Data Sets: Merge Sort is ideal for sorting massive datasets that cannot fit into memory, as it works well with external storage.
- Database Management: Used to sort records in databases for efficient querying and retrieval.

2. Gaming

- Game Development: Helps in sorting and merging game elements, such as ranking players or organizing game assets, to enhance gameplay performance.

3. E-Commerce

- Product Ranking: Used to sort products based on price, ratings, or popularity for a seamless user experience.
- Inventory Management: Efficiently organizes inventory data for quick access and updates.

4. File Systems

- File Merging: Merge Sort is used in file systems to merge sorted files, such as during multi-way merge operations in external sorting.

5. Multithreading

- Parallel Processing: Its divide-and-conquer approach makes it suitable for parallel execution, improving performance in multi-core systems.

6. Genomics

- DNA Sequencing: Used to sort and merge DNA sequences in bioinformatics for Analysis and research.

Time Complexity:

- **Best Case:** O(n log n), When the array is already sorted or nearly sorted.
- **Average Case:** O(n log n), When the array is randomly ordered.
- **Worst Case:** O(n log n), When the array is sorted in reverse order.

Auxiliary Space: O(n), Additional space is required for the temporary array used during merging.

Limitations:

1. **Space Complexity**:
   Merge Sort requires additional memory for temporary arrays during the merging process. Its space complexity is O(n)O(n)O(n), which can be a drawback for systems with limited memory.
2. **Recursive Overhead**:
   Merge Sort uses recursion, which can lead to significant overhead due to function calls, especially for very large datasets. This can also cause stack overflow in environments with limited stack size.
3. **Not In-Place**:
   Unlike some other sorting algorithms (e.g., Quick Sort), Merge Sort is not an in-place algorithm, as it requires extra space for merging.
4. **Performance on Small Datasets**:
   For smaller datasets, Merge Sort may be slower compared to simpler algorithms like Insertion Sort, due to its higher constant factors and overhead.
5. **Complex Implementation**:
   The implementation of Merge Sort is more complex compared to simpler algorithms like Bubble Sort or Selection Sort, making it harder to debug and maintain.

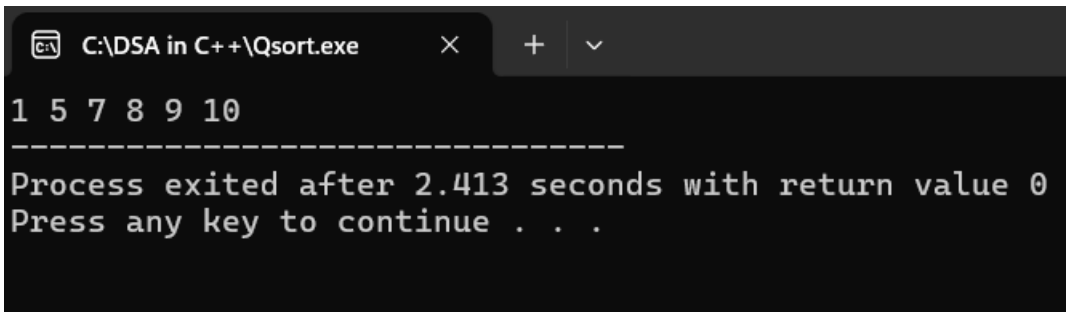
Quick Sort:

```
#include <stdio.h>

void swap(int* a, int* b);

// Partition function

int partition(int arr[], int low, int high) {

  // Choose the pivot

  int pivot = arr[high];

  // Index of smaller element and indicates

  // the right position of pivot found so far

  int i = low - 1;


  // Traverse arr[low..high] and move all smaller
```

```
    // elements to the left side. Elements from low to
    // i are smaller after every iteration
    for (int j = low; j <= high - 1; j++) {
        if (arr[j] < pivot) {
            i++;
            swap(&arr[i], &arr[j]);
        }
    }
    // Move pivot after smaller elements and
    // return its position
    swap(&arr[i + 1], &arr[high]);
    return i + 1;
}

// The QuickSort function implementation
void quickSort(int arr[], int low, int high) {
    if (low < high) {
        // pi is the partition return index of pivot
        int pi = partition(arr, low, high);

        // Recursion calls for smaller elements
        // and greater or equals elements
        quickSort(arr, low, pi - 1);
        quickSort(arr, pi + 1, high);
    }
}

void swap(int* a, int* b) {
```

```
    int t = *a;
    *a = *b;
    *b = t;
]
int main() {
    int arr[] = {10, 7, 8, 9, 1, 5};
    int n = sizeof(arr) / sizeof(arr[0]);

    quickSort(arr, 0, n - 1);
    for (int i = 0; i < n; i++) {
        printf("%d ", arr[i]);
    }
    return 0;
}
```

OUTPUT:



Time Complexity:

- Best Case: ($\Omega(n \log n)$), Occurs when the pivot element divides the array into two equal halves.
- Average Case ($\theta(n \log n)$), On average, the pivot divides the array into two parts, but not necessarily equal.
- Worst Case: ($O(n^2)$), Occurs when the smallest or largest element is always chosen as the pivot (e.g., sorted arrays).

Auxiliary Space: O(n), due to recursive call stack

Applications :

- Database Management: Quick Sort efficiently sorts large volumes of data like customer records, transaction logs, and product inventories, helping database systems like MySQL and PostgreSQL perform faster searches and indexing. Its in-place sorting minimizes memory overhead, making it ideal for database operations.
- File and Directory Sorting: Operating systems and file management utilities use Quick Sort to organize files by name, date, size, or other attributes due to its speed and cache-friendly behavior, especially when handling large directories.
- Sorting in Programming Libraries: Many standard libraries implement Quick Sort or its variants for sorting primitive data types. For example, C++ STL's std::sort uses a hybrid introsort which begins with Quick Sort, while Java's Arrays.sort() for primitives relies on dual-pivot Quick Sort for superior average-case performance.
- Parallel and Distributed Processing: Quick Sort's divide-and-conquer nature enables straightforward parallelization. Large datasets can be partitioned and sorted concurrently on multi-core processors or distributed systems, which is valuable in Big Data frameworks like Apache Spark and Hadoop.
- Scientific and Numerical Computations: Quick Sort supports priority queues and other data structures central to scientific research requiring accurate and efficient sorting, often as part of larger algorithmic workflows.
- E-commerce and Search Engines: Sorting product listings by price, rating, or popularity relies on Quick Sort for rapid data organization. Similarly, search engines use it to rank and present results based on relevance efficiently.

Limitations:

1. Instability

- Quick Sort is not a stable sorting algorithm, meaning it does not preserve the relative order of elements with equal keys. This can be problematic in scenarios where stability is required.

2. Worst-Case Time Complexity

- In its worst-case scenario (e.g., when the pivot is always the smallest or largest element), Quick Sort has a time complexity of $O(n^2)$. This can occur if the input array is already sorted or nearly sorted, depending on the pivot selection strategy.

3. Recursive Nature

- Quick Sort is implemented using recursion, which can lead to stack overflow for very large datasets if the recursion depth becomes too high.

4. Pivot Selection Sensitivity

- The efficiency of Quick Sort heavily depends on the choice of the pivot. A poor pivot selection can degrade performance significantly.

5. Memory Usage

- Although Quick Sort is an in-place sorting algorithm, its recursive calls require additional stack space. In the worst case, this can lead to O(n) extra space usage.

6. Not Suitable for Small Datasets

- For small datasets, Quick Sort may not be as efficient as simpler algorithms like Insertion Sort due to its overhead.