EXPERIMENT NO.02

Roll No.: 24141001
Batch:I1

Title: Quick sort, Merge sort using array as a data structure. Analyse time and space complexity.

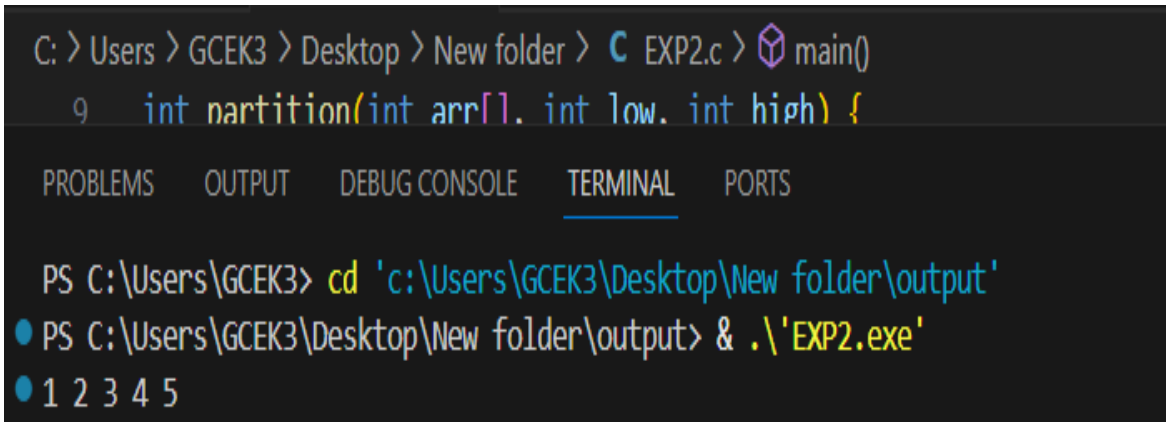
1) Quick sort

Program:

```
#include <stdio.h>

void swap(int* a, int* b) {
   int temp = *a;
   *a = *b;
   *b = temp;
}

int partition(int arr[], int low, int high) {
   int pivot = arr[high];
   int i = low - 1;
   for (int j = low; j < high; j++) {
      if (arr[j] <= pivot) {
         i++;
         swap(&arr[i], &arr[j]);
      }
   }
   swap(&arr[i + 1], &arr[high]);
   return i + 1;
}

void quickSort(int arr[], int low, int high) {
   if (low < high) {
      int pi = partition(arr, low, high);
      quickSort(arr, low, pi - 1);
      quickSort(arr, pi + 1, high);
   }
}

int main() {
   int arr[] = {4, 2, 5, 3, 1};
   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:



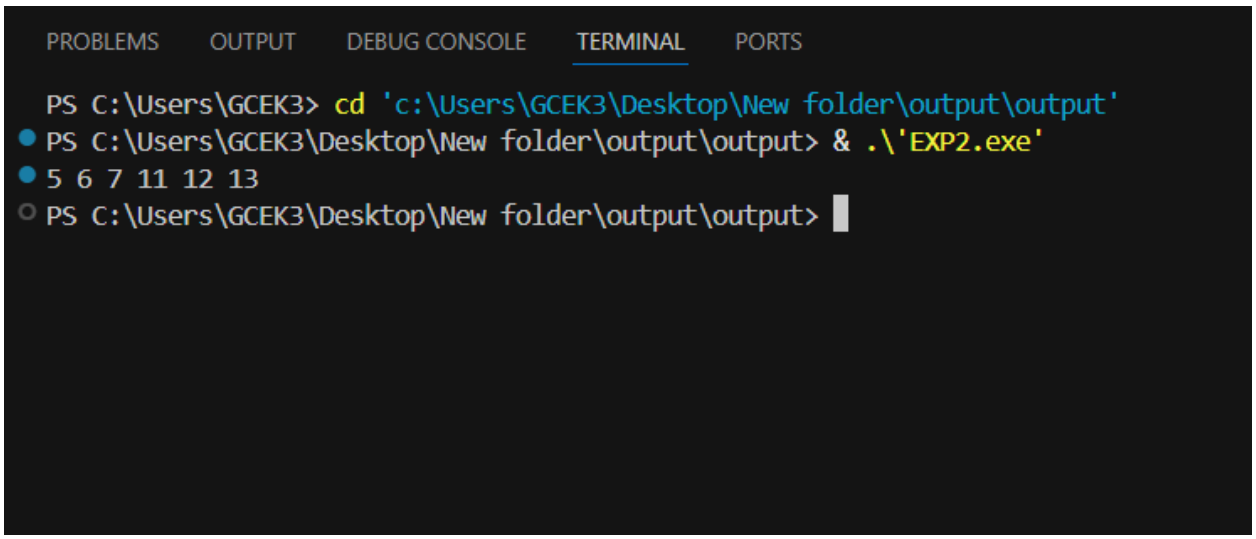
2)Merge sort

Program:

```
// C program for the implementation of merge sort
#include <stdio.h>
#include <stdlib.h>

void merge(int arr[], int left, int mid, int right) {
    int n1 = mid - left + 1;
    int n2 = right - mid;
    int leftArr[n1], rightArr[n2];

    for (int i = 0; i < n1; i++)
        leftArr[i] = arr[left + i];
    for (int j = 0; j < n2; j++)
        rightArr[j] = arr[mid + 1 + j];

    int i = 0, j = 0, k = left;
    while (i < n1 && j < n2) {
        if (leftArr[i] <= rightArr[j]) {
            arr[k++] = leftArr[i++];
        } else {
            arr[k++] = rightArr[j++];
```

```
        }
    }

    while (i < n1)
        arr[k++] = leftArr[i++];
    while (j < n2)
        arr[k++] = rightArr[j++];
}

void mergeSort(int arr[], int left, int right) {
    if (left < right) {
        int mid = left + (right - left) / 2;
        mergeSort(arr, left, mid);
        mergeSort(arr, mid + 1, right);
        merge(arr, left, mid, right);
    }
}

int main() {
    int arr[] = {12, 11, 13, 5, 6, 7};
    int n = sizeof(arr) / sizeof(arr[0]);
    mergeSort(arr, 0, n - 1);
    for (int i = 0; i < n; i++)
        printf("%d ", arr[i]);
    return 0;
}
```

OUTPUT:



- Quick Sort is generally faster and uses less space than Merge Sort for sorting arrays in memory, except in worst-case scenarios.
- Merge Sort guarantees O(n log n) time, but it requires extra memory for temporary arrays.
- Merge Sort is stable, while Quick Sort is not.
- Quick Sort is preferred for small arrays and memory-constrained environments; Merge Sort works best for external (large) sorting tasks.

The best, average, and worst-case time complexities for Quick Sort and Merge Sort vary based on how the algorithms process different input arrangements.

Quick Sort

- Best Case:
    - Occurs when the pivot always splits the array into two equal halves.
    - Time Complexity: O(nlogn)
- Average Case:
    - Occurs for random data when splits are reasonably balanced on average.
    - Time Complexity: O(nlogn)
- Worst Case:
    - Occurs when the pivot is always the smallest or largest element (e.g., already sorted array).
    - Time Complexity: $O(n^2)$

Merge Sort

- Best Case:
    - Happens for any input due to consistent splitting and merging steps.
    - Time Complexity: O(nlogn)
- Average Case:
    - Applies to all random input orders due to the fixed divide-and-merge process.
    - Time Complexity: O(nlogn)
- Worst Case:
    - Even for the most unbalanced splits, merge sort still consistently divides and merges.
    - Time Complexity: O(nlogn)

Quick Sort can perform exceptionally well on average, but with bad pivots, its performance degrades significantly. Merge Sort, however, is consistent for every pattern of input data.

Conclusion:

Both Quick Sort and Merge Sort are important, each suited to different situations. While Quick Sort is fast and efficient for in-place array sorting, Merge Sort provides guaranteed performance and stability, useful for larger and external datasets.