Merge Sort

Aim: The aim of this lab practical is to implement and analyze the efficiency of the Merge Sort algorithm in sorting a collection of elements in ascending order.

Theory:
Merge sort is a popular sorting algorithm that follows the divide and conquer paradigm. It works by dividing the input array into two halves, sorting each half recursively, and then merging the sorted halves to produce the final sorted array.

Pseudocode:

```
MergeSort(arr):
    if length of arr <= 1:
        return arr

    mid = length of arr / 2
    left_half = arr[0:mid]
    right_half = arr[mid:end]

    left_half = MergeSort(left_half)
    right_half = MergeSort(right_half)

   result = Merge(left_half, right_half)

    return result

Merge(left, right):
    merged = []
    i = j = 0

    while i < length of left and j < length of right:
        if left[i] <= right[j]:
            append left[i] to merged
            increment i
        else:
            append right[j] to merged
            increment j

    append remaining elements of left from index i to merged
    append remaining elements of right from index j to merged

    return merged
```

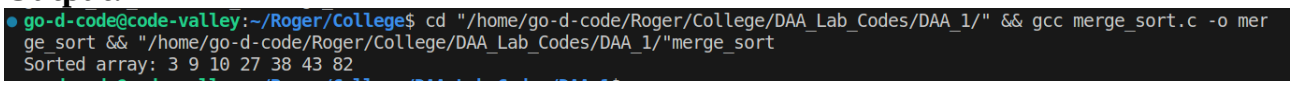
Code:

```
#include <stdio.h>
#include <stdlib.h>

void merge(int arr[], int l, int m, int r) {
    int i, j, k;
    int n1 = m - l + 1;
```

```
    int n2 = r - m;

    int L[n1], R[n2];

    for (i = 0; i < n1; i++)
        L[i] = arr[l + i];
    for (j = 0; j < n2; j++)
        R[j] = arr[m + 1 + j];

    i = 0;
    j = 0;
    k = l;
    while (i < n1 && j < n2) {
        if (L[i] <= R[j]) {
            arr[k] = L[i];
            i++;
        } else {
            arr[k] = R[j];
            j++;
        }
        k++;
    }

    while (i < n1) {
        arr[k] = L[i];
        i++;
        k++;
    }

    while (j < n2) {
        arr[k] = R[j];
        j++;
        k++;
    }
}

void mergeSort(int arr[], int l, int r) {
    if (l < r) {
        int m = l + (r - l) / 2;

        mergeSort(arr, l, m);
        mergeSort(arr, m + 1, r);

        merge(arr, l, m, r);
    }
}

int main() {
    int myArray[] = {38, 27, 43, 3, 9, 82, 10};

    int size = sizeof(myArray) / sizeof(myArray[0]);
```

```
    mergeSort(myArray, 0, size - 1);

    printf("Sorted array: ");
    for (int i = 0; i < size; i++) {
        printf("%d ", myArray[i]);
    }
    printf("\n");

    return 0;
}
```

Output:



Algorithm Steps:

1. MergeSort(arr):
   - If the length of `arr` is 1 or less, return `arr`.
   - Calculate the middle index (`mid`) of `arr`.
   - Split `arr` into `left_half` and `right_half`.
   - Recursively call `MergeSort` on `left_half` and `right_half`.
   - Merge the sorted `left_half` and `right_half` using the `Merge` function.
   - Return the merged result.

2. Merge(left, right):
   - Initialize an empty array `merged`.
   - Initialize two pointers, `i` and `j`, to 0.
   - Compare elements from `left` and `right`.
   - Append the smaller element to `merged` and increment the corresponding pointer.
   - Continue this process until one of the arrays is exhausted.
   - Append any remaining elements from `left` and `right` to `merged`.
   - Return the merged array.

Example:

1. Initial array: [38, 27, 43, 3, 9, 82, 10]

2. First recursion (divide):
   - Split the array into two halves: left_half = [38, 27, 43] and right_half = [3, 9, 82, 10].
   - Recursively apply Merge Sort to left_half and right_half.

   Left Half:
   - Split into [38] and [27, 43].
   - Recursively apply Merge Sort to [27, 43].
   - Split [27] and [43]. Since both have one element, they are considered sorted.

   Right Half:
   - Split into [3] and [9, 82, 10].
   - Recursively apply Merge Sort to [9, 82, 10].
   - Split [9] and [82, 10].
   - Recursively apply Merge Sort to [82, 10].

- Split [82] and [10]. Both are sorted.

3. Merge steps:
   - Merge [27] and [43] to get [27, 43].
   - Merge [38] and [27, 43] to get [27, 38, 43].
   - Merge [9] and [82, 10] to get [9, 10, 82].

   - Merge [27, 38, 43] and [9, 10, 82] to get [9, 10, 27, 38, 43, 82].

4. Final sorted array: [9, 10, 27, 38, 43, 82]

Time Complexity:

- Best Case: O(n log n) - The array is consistently divided into halves, and the merging step takes linear time.
- Average Case: O(n log n) - Similar to the best case, the divide-and-conquer approach dominates the time complexity.
- Worst Case: O(n log n) - Even in the worst case, the time complexity remains O(n log n) due to the consistent division of the array.

Space Complexity:
- The space complexity is O(n) due to the additional space required for the temporary arrays during the merge process. In the worst case, log n recursive calls are made, and each call requires O(n) space for the temporary arrays.

Quick Sort

Aim: The aim of this lab practical is to implement and analyze the efficiency of the Quick Sort algorithm in sorting a collection of elements in ascending order.

Theory:

The basic idea behind QuickSort is to partition the array into two parts and then recursively sort each part. The key operation is the partitioning step, where an element, called the pivot, is chosen, and the array is rearranged such that elements smaller than the pivot are on its left, and elements greater than the pivot are on its right.

Pseudocode:

```
QuickSort(arr, low, high):
   if low < high:
      pivot_index = Partition(arr, low, high)

      QuickSort(arr, low, pivot_index - 1)
      QuickSort(arr, pivot_index + 1, high)

Partition(arr, low, high):
   pivot = arr[high]
   i = low - 1

   for j from low to high - 1:
      if arr[j] <= pivot:
         // Swap arr[i+1] and arr[j]
         swap arr[i+1] and arr[j]
         // Move the index of the smaller element
         i = i + 1

   swap arr[i+1] and arr[high]

   return i+1
```

Code:

```
#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 - 1; 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 pivotIndex = partition(arr, low, high);

        quickSort(arr, low, pivotIndex - 1);
        quickSort(arr, pivotIndex + 1, high);
    }
}

int main()
{
    int myArray[] = {64, 25, 12, 22, 11};

    int size = sizeof(myArray) / sizeof(myArray[0]);

    quickSort(myArray, 0, size - 1);

    printf("Sorted array: ");
    for (int i = 0; i < size; i++)
    {
        printf("%d ", myArray[i]);
    }
    printf("\n");
    return 0;
}
```

Output:

```
go-d-code@code-valley:~/Roger/College/DAA_Lab_Codes/DAA_1$ cd "/home/go-d-code/Roger/College/DAA_Lab_Codes/DAA_1/" && gcc quick_sort.c -o quick_sort && "/home/go-d-code/Roger/College/DAA_Lab_Codes/DAA_1/"quick_sort
Sorted array: 11 12 22 25 64
```

Algorithm Analysis:

QuickSort is a highly efficient sorting algorithm that follows a divide-and-conquer approach. QuickSort is known for its widespread use due to its average-case time complexity of O(n log n), making it particularly efficient for large datasets. The algorithm operates by selecting a pivot element from the array and partitioning the elements around it, placing smaller elements to its left and larger elements to its right. The pivot then takes its final sorted position. Through recursive

application of QuickSort to the sub-arrays on the left and right of the pivot, the entire array becomes sorted. While QuickSort performs exceptionally well on average, with a best-case and average-case time complexity of O(n log n), it can degrade to O(n^2) in the worst case when poorly chosen pivots result in unbalanced partitions. Despite this, various pivot selection strategies, such as choosing a random pivot, contribute to mitigating the worst-case scenario. The space complexity of QuickSort is O(log n) on average, determined by the depth of the recursive call stack.

Algorithm Steps:

1. QuickSort(arr, low, high):
   - If low is less than high:
     - Call Partition to get the pivot index.
     - Recursively call QuickSort on the sub-arrays to the left and right of the pivot.

2. Partition(arr, low, high):
   - Choose the pivot, typically the rightmost element (pivot = arr[high]).
   - Initialize the index of the smaller element (i = low - 1).
   - Traverse through the array and rearrange elements such that elements smaller than the pivot are on the left, and elements greater than the pivot are on the right.
   - Swap the pivot with the element at i+1.
   - Return the index i+1 as the pivot's final sorted position.

Example:

1. Choose a Pivot:

Let's choose the last element, 10, as the pivot.

Array: [38, 27, 43, 3, 9, 82, 10]
Pivot: 10

2. Partition the Array:

Partition the array into two sub-arrays - elements less than the pivot and elements greater than the pivot.

[3, 9] | 10 | [38, 27, 43, 82]

Now, the pivot (10) is in its final sorted position.

3. Recursively Sort Sub-arrays:

Recursively apply the QuickSort algorithm to the sub-arrays.

- For the left sub-array ([3, 9]):
  - Choose the pivot (9).
  - Partition: [3] | 9 | []
  - The left sub-array is now sorted.

- For the right sub-array ([38, 27, 43, 82]):
  - Choose the pivot (82).
  - Partition: [38, 27, 43] | 82 | []
  - The right sub-array needs further sorting.

4. Continue Recursion:

Continuing with the right sub-array:

- Choose the pivot (43).
- Partition: [38, 27] | 43 | []
- The right sub-array is now sorted.

5. Final Result:
Combining all the sorted sub-arrays, we get the final sorted array:

[3, 9, 10, 27, 38, 43, 82]

So, the array [38, 27, 43, 3, 9, 82, 10] is sorted using the QuickSort algorithm.

Time Complexity:
- **Best Case:** O(n log n) - This occurs when the partitioning is balanced, and each recursive call processes roughly half of the array.
- **Average Case:** O(n log n) - On average, QuickSort performs well, making O(n log n) comparisons to sort n items.
- **Worst Case:** O(n^2) - This occurs when the pivot selection always results in an unbalanced partition. However, with good pivot selection strategies (e.g., random pivot), the worst-case scenario is unlikely.

Space Complexity:
- O(log n) - The space complexity is determined by the depth of the recursive call stack. In the average case, QuickSort has a logarithmic space complexity.