|  |
| --- |
| **Experiment No. 3** |
| **To implement Merge Sort** |
| Date of Performance:22/2/24 |
| Date of Submission: 29/2/24 |

## Experiment No. 3

**Title:** Merge Sort

**Aim:** To study, implement and Analyze Merge Sort Algorithm

**Objective:** To introduce the methods of designing and analyzing algorithms

#### Theory:

The merge sort algorithm closely follows the divide-and-conquer paradigm. Intuitively, it operates as follows:

* + 1. Divide: Divide the n-element sequence to be sorted into two sub sequences of n=2 elements each.
    2. Conquer: Sort the two sub sequences recursively using merge sort.
    3. Combine: Merge the two sorted sub sequences to produce the sorted answer.

During the Merge sort process the object in the collection are divided into two collections. To split a collection, Merge sort will take the middle of the collection and split the collection into its left and its right part. The resulting collections are again recursively sorted via the Merge sort algorithm.

Once the sorting process of the two collections is finished, the result of the two collections is combined. To combine both collections Merge sort start at each collection at the beginning. It pick the object which is smaller and inserts this object into the new collection. For this collection it now selects the next elements and selects the smaller element from both collection.

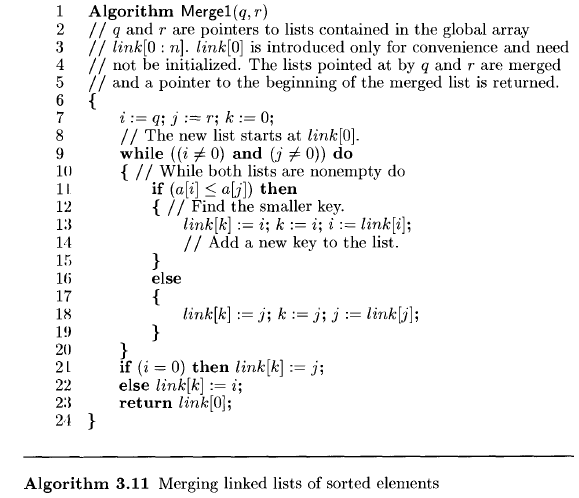
Once all elements from both collections have been inserted in the new collection, Merge sort has successfully sorted the collection. To avoid the creation of too many collections, typically one new collection is created and the left and right side are treated as different collections.

#### A diagram of a tree Description automatically generatedExample: Sort the sequence <33,22,44,0,99,88,11> using Merge Sort

#### 

A screenshot of a computer code

Description automatically generated**Algorithm and Complexity:**

****

Recurrence Relation for Merger Sort:

T(n) = 1 for n=1

T(n) = 2T(n/2) + n  for n>1… (1)

**Solve by Substitution method:**

Solving original recurrence for n/2,

T(n/2) = 2T(n/4) + n/2

Substituting this in equation (1),

T(n) = 2[ 2T(n/4) + n/2 ] + n

= 22 T(n/22) + 2n .

T(n) = 2k T(n/2k) + k.n … (2)

Let us consider that k grows up to log2n,

Let n/2k =1

n = 2k

k = log2n

n = 2k

Substitute these values in equation (2)

T(n) = nT(n/n) + log2n . n

T(n) = O(n.log2n)

A diagram of mathematical equations

Description automatically generated**Solve using recursive tree Method:**

**Code:**

**#include<stdio.h>**

**void Merge(int arr[], int l, int mid, int h) {**

**int merge[h - l + 1];**

**int n1 = l;**

**int n2 = mid + 1;**

**int k = 0;**

**while (n1 <= mid && n2 <= h) {**

**if (arr[n1] < arr[n2]) {**

**merge[k] = arr[n1];**

**k++, n1++;**

**} else {**

**merge[k] = arr[n2];**

**k++, n2++;**

**}**

**}**

**while (n1 <= mid) {**

**merge[k] = arr[n1];**

**k++, n1++;**

**}**

**while (n2 <= h) {**

**merge[k] = arr[n2];**

**k++, n2++;**

**}**

**for (int i = 0, j = l; i < h - l + 1; i++, j++) {**

**arr[j] = merge[i];**

**}**

**}**

**void MergeSort(int arr[], int l, int h) {**

**if (l >= h) {**

**return;**

**}**

**int mid = (l + h) / 2;**

**MergeSort(arr, l, mid);**

**MergeSort(arr, mid + 1, h);**

**Merge(arr, l, mid, h);**

**}**

**int main() {**

**int n;**

**int arr[20];**

**printf("Enter the range: ");**

**scanf("%d", &n);**

**printf("Enter the elements:\n");**

**for (int i = 0; i < n; i++) {**

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

**}**

**MergeSort(arr, 0, n - 1);**

**printf("Sorted Elements are:\n");**

**for (int i = 0; i < n; i++) {**

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

**}**

**return 0;**

**}**

**Output:**

**Enter the range: 10**

**Enter the elements:**

**10 5 8 2 3 6 9 2 4 0**

**Sorted Elements are:**

**0 2 2 3 4 5 6 8 9 10**

#### Conclusion:

Our experiment with Merge Sort has demonstrated its efficiency and scalability for sorting large datasets. Unlike Selection Sort and Insertion Sort, Merge Sort consistently maintains a time complexity of O(n log n), making it highly efficient even for substantial amounts of data. Its divide-and-conquer approach splits the dataset into smaller parts, sorts them individually, and then merges them back together, ensuring a stable and efficient sorting process.

Additionally, Merge Sort's performance remains consistent regardless of the initial order of elements, making it suitable for a wide range of applications. Therefore, Merge Sort emerges as a reliable choice for sorting tasks where performance and scalability are crucial considerations.