VIT®
Vellore Institute of Technology
(Deemed to be University under section 3 of UGC Act, 1956)

Ex.No: 3 Date : 01.08.2024	Implementation of Merge sort Algorithms with experiments on the analysis and the efficiency	Name : Venkatesan M Reg.No : 22BAI1259

Aim

Implementation of Merge sort Algorithms with Divide-Conquer-Combine approach with experiments on the analysis and the efficiency.

Algorithm

Merge Sort is a divide-and-conquer algorithm that splits an array into halves, sorts each half, and then merges them back together in a sorted manner.

Steps:

1. **Base Case Check**:
    - If the array contains one or zero elements, it is already sorted. No further action is needed.
2. **Divide**:
    - Calculate the middle index of the array to split it into two halves. This can be done by taking the left index (`l`) and the right index (`r`) and computing the middle index `m` as

$$m = l + \frac{(r-l)}{2}$$

3. **Conquer**:
    - Recursively apply the Merge Sort algorithm to the left half of the array from index `l` to `m`.
    - Recursively apply the Merge Sort algorithm to the right half of the array from index `m + 1` to `r`.

4. **Combine**:
    - Once the two halves are sorted, merge them together into a single sorted array.

Merging Process

1. **Create Temporary Arrays**:
    - Create two temporary arrays to hold the left and right halves of the array being merged. These arrays will be `L[]` for the left half and `R[]` for the right half.

2. **Copy Elements**:
    - Copy the elements from the left subarray into the temporary array `L[]`.
    - Copy the elements from the right subarray into the temporary array `R[]`.

3. **Merge Temporary Arrays**:
    - Initialize three pointers: `i` for `L[]`, `j` for `R[]`, and `k` for the original array `arr[]`.
    - Compare the elements of `L[]` and `R[]`. Insert the smaller element into the original array `arr[]`.
    - Move the pointer forward in the temporary array from which the element was taken.
    - Repeat this process until all elements from `L[]` and `R[]` have been inserted into `arr[]`.

4. **Copy Remaining Elements**:
    - If there are any remaining elements in `L[]`, copy them into `arr[]`.
    - If there are any remaining elements in `R[]`, copy them into `arr[]`.

Pseudo code

```
MergeSort(arr[], l, r)
    if l < r
        // Find the middle point
        m = l + (r - l) / 2

        // Sort first and second halves
        MergeSort(arr, l, m)
        MergeSort(arr, m + 1, r)

        // Merge the sorted halves
        Merge(arr, l, m, r)

Merge(arr[], l, m, r)
    n1 = m - l + 1
    n2 = r - m

    // Create temporary arrays
    L[n1], R[n2]

    // Copy data to temporary arrays
    for i = 0 to n1 - 1
        L[i] = arr[l + i]
    for j = 0 to n2 - 1
        R[j] = arr[m + 1 + j]

    // Merge the temporary arrays back into arr[l..r]
```

```
i = 0, j = 0, k = l
while i < n1 and j < n2
   if L[i] <= R[j]
      arr[k] = L[i]
      i++
   else
      arr[k] = R[j]
      j++
   k++

// Copy the remaining elements of L[], if any
while i < n1
   arr[k] = L[i]
   i++
   k++

// Copy the remaining elements of R[], if any
while j < n2
   arr[k] = R[j]
   j++
   k++
```

Problem Statement 1

Translate the merge-sort algorithm (A) discussed in the class into a program (P) and execute the same for the given inputs.

Input

- 0,1,2,3,4,5,
- 5, 5.5, 6, 3.723, 1.23, 8.88

Source Code

```
#include <iostream>
#include <vector>
#include <chrono>
#include <cmath>

using namespace std;

// Merge function
void merge(vector<double>& arr, int left, int mid, int right) {
    int n1 = mid - left + 1;
    int n2 = right - mid;
    vector<double> L(n1), R(n2);

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

    int i = 0, j = 0, k = left;
    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;
    }
}

// Merge Sort function
void mergeSort(vector<double>& 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);
    }
}

// Function to calculate time and sort
double sortAndMeasureTime(vector<double>& arr) {
    auto start = chrono::high_resolution_clock::now();
    mergeSort(arr, 0, arr.size() - 1);
    auto end = chrono::high_resolution_clock::now();
```

```
    chrono::duration<double> duration = end - start;
    return duration.count();
}

int main() {
    vector<vector<double>> inputs = {
        {0, 1, 2, 3, 4, 5},
        {5, 5.5, 6, 3.723, 1.23, 8.88},
        // Add more input sets of varying sizes for experiments
    };

    cout << "Size\tT(n)\tt(n)\n";
    for (auto& input : inputs) {
        double tn = input.size() * log2(input.size());
        vector<double> temp = input; // Make a copy to sort
        double t = sortAndMeasureTime(temp);
        cout << input.size() << "\t" << tn << "\t" << t << "\n";
    }

    return 0;
}
```

Output

Size	T(n)	t(n)
6	15.51	2.511e-06
6	15.51	1.82e-06

Problem Statement 2

Run the code with different inputs and record the output of your execution in the form of a table as follows. Here n is the size of the problem . Since T(n) ∈ θ(n.lgn), Take T(n) as (n ∗ lg n). t(n) is the time taken by your program for executing the input of size n, as computed through your code. particular input Choose 10 different values for the size of the problem , n in an increasing magnitude and run your code. Please note the difference between T(n) and t(n).

Source Code

```
#include <iostream>
#include <ctime>
#include <cstdlib>
#include <cmath>

using namespace std;

// Merge function
void merge(int arr[], int p, int q, int r) {
   int n1 = q - p + 1;
   int n2 = r - q;
   int L[n1], M[n2];

   for (int i = 0; i < n1; i++)
      L[i] = arr[p + i];
   for (int j = 0; j < n2; j++)
      M[j] = arr[q + 1 + j];

   int i = 0, j = 0, k = p;
   while (i < n1 && j < n2) {
```

```
        if (L[i] <= M[j]) {
            arr[k] = L[i];
            i++;
        } else {
            arr[k] = M[j];
            j++;
        }
        k++;
    }

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

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

// Merge Sort function
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);
    }
```

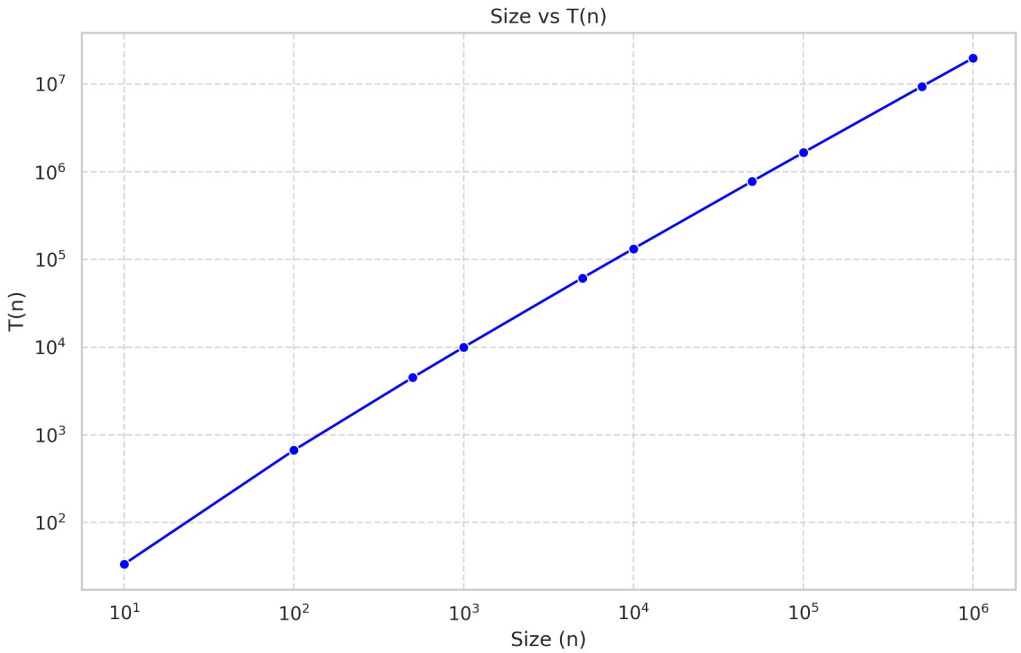
```
}

// Measure execution time
void measureExecutionTime(int n) {
    int* arr = new int[n];
    for (int i = 0; i < n; ++i) {
        arr[i] = rand() % 100;
    }

    clock_t start = clock();
    mergeSort(arr, 0, n - 1);
    clock_t end = clock();

    double duration = double(end - start) / CLOCKS_PER_SEC;
    double tn = n * log2(n);
    cout << "Size: " << n << ", T(n): " << tn << ", t(n): " << duration << " seconds" << endl;

    delete[] arr;
}

int main() {
    srand(time(0));
    int sizes[] = {10, 100,500, 1000, 5000,10000, 50000,100000,500000,1000000};
    for (int size : sizes) {
        measureExecutionTime(size);
    }
    return 0;
}
```

Inference

Size of the Inputs	T(n)	t(n) (seconds)
10	33.22	2e-06
100	664.39	1e-05
500	4482.89	5.6e-05
1000	9965.78	0.000103
5000	61438.6	0.000567
10000	132877	0.001194
50000	780482	0.006375
100000	1.66096e+06	0.013109
500000	9.46578e+06	0.071423
1000000	1.99316e+07	0.14

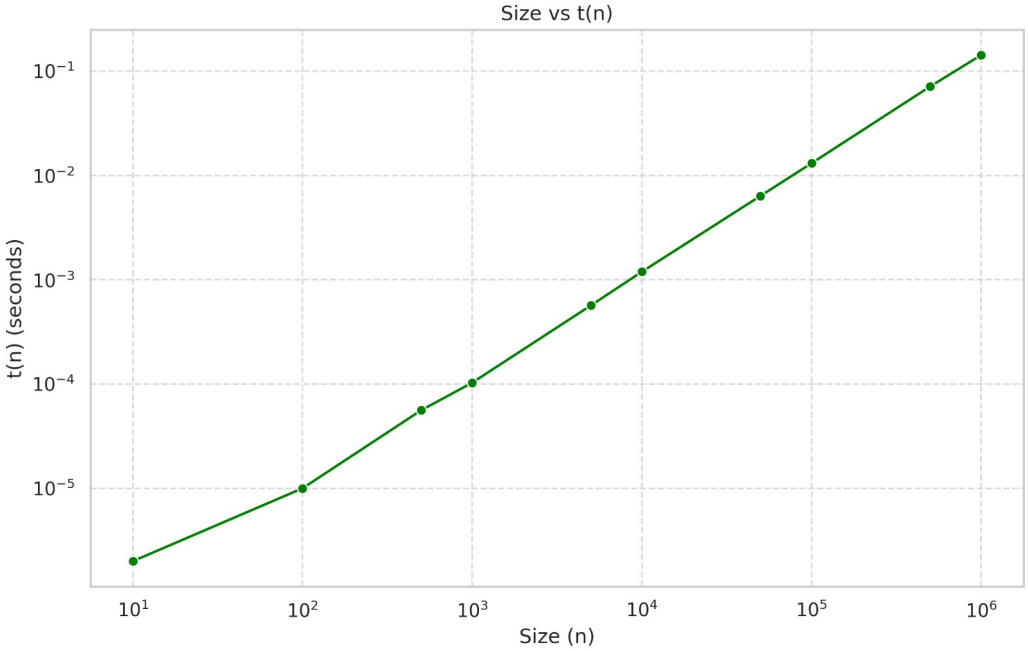
Graph between varying size of inputs and time

Plot a graph n V s T(n).



This plot shows a logarithmic increase of $T(n)$ versus the size of the input n. Given that $T(n)$ is calculated as $n\log(n)$, the curve indicating a logarithmic increase in time with respect to n, suggesting that the time complexity aligns with $O(n\log n)$.

Plot a graph n V s t(n).



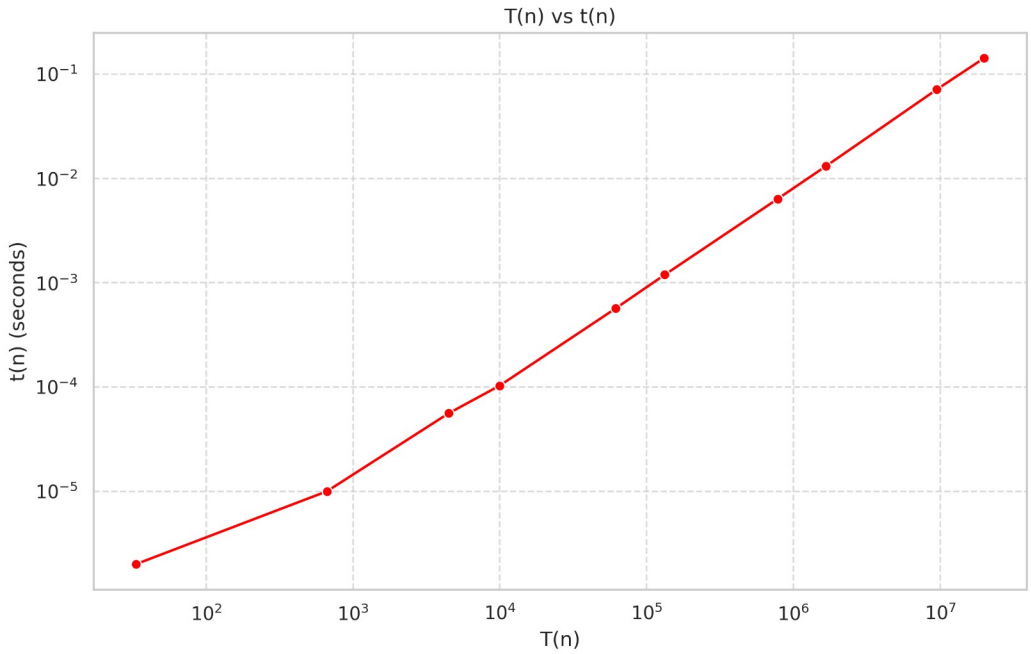
$t(n)$ represents the actual execution time per operation or per input size. As the input size n increases, $t(n)$ increases, but not as steeply as $T(n)$ because $t(n)$ includes more factors (e.g., system overhead, I/O).

The recurrence relation for Merge Sort is

$$T(n)=2T\left(\frac{n}{2}\right)+O(n)$$

Best-case: $O(n\log n)$
Worst-case: $O(n\log n)$
Average-case: $O(n\log n)$

Plot a graph T(n) V s t(n).



This graph shows a proportional relationship between $T(n)$ and $t(n)$. Since $T(n)$ is a function of $O(n \log n)$ and $t(n)$ is measured time.

Problem Statement 3

Modify the Merge-sort algorithm such that the algorithm takes the input as words (of different lengths) and arrange them in an alphabetical order. Your words will have both lower-case letters as well as the upper-case letters. Compute the time-complexity t(n) of your algorithm in an experimental approach. Compare this algorithm with that of the algorithm which takes n numbers as inputs and decide which consumes minimum time.

Source Code

```
#include <iostream>
#include <vector>
#include <string>
#include <chrono>

using namespace std;

// Merge function for strings
void merge(vector<string>& arr, int left, int mid, int right) {
    int n1 = mid - left + 1;
    int n2 = right - mid;
    vector<string> L(n1), R(n2);

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

    int i = 0, j = 0, k = left;
    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;
    }
}

// Merge Sort function for strings
void mergeSort(vector<string>& 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);
    }
}

// Function to calculate time and sort strings
double sortAndMeasureTime(vector<string>& arr) {
    auto start = chrono::high_resolution_clock::now();
    mergeSort(arr, 0, arr.size() - 1);
```

```
    auto end = chrono::high_resolution_clock::now();
    chrono::duration<double> duration = end - start;
    return duration.count();
}

int main() {
    vector<string> words = {"banana", "Apple", "orange", "grape", "Pineapple"};
    double t = sortAndMeasureTime(words);

    cout << "Sorted words:\n";
    for (const auto& word : words)
        cout << word << " ";
    cout << "\nTime taken: " << t << " seconds\n";

    return 0;
}
```

Sample Input and output

{"banana", "Apple", "orange", "grape", "Pineapple"};

Sorted words:

Apple Pineapple banana grape orange

Time taken: 7.55e-06 seconds

Measure time for Random input

```
#include <iostream>
#include <vector>
#include <string>
#include <chrono>
#include <cmath>
#include <ctime>
#include <cstdlib>

using namespace std;

// Merge function for strings
void merge(vector<string>& arr, int left, int mid, int right) {
    int n1 = mid - left + 1;
    int n2 = right - mid;
    vector<string> L(n1), R(n2);

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

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

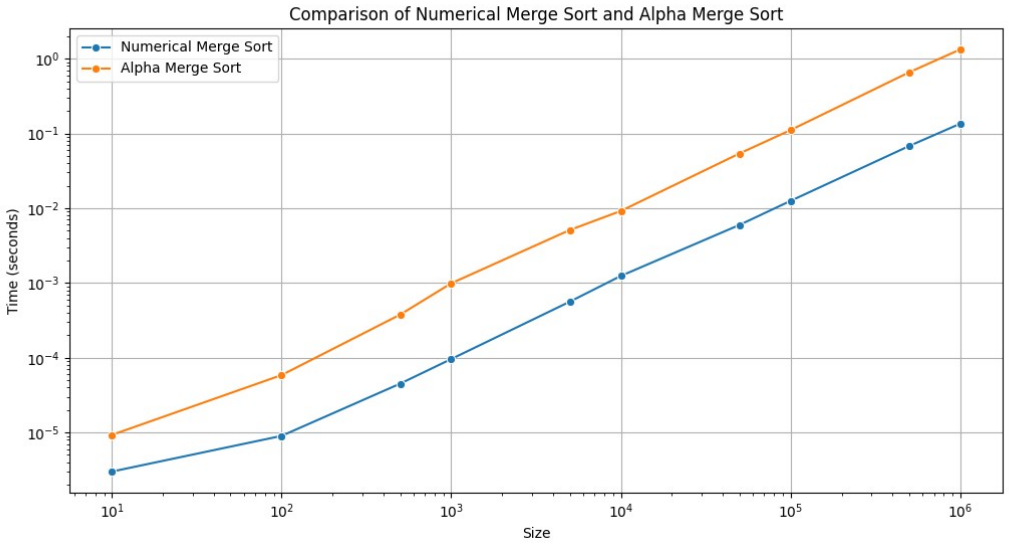
```
    }

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

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

// Merge Sort function for strings
void mergeSort(vector<string>& 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);
    }
}

// Function to generate random strings
vector<string> generateRandomStrings(int n) {
    vector<string> arr(n);
    static const char alphanum[] = "ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz";
    for (int i = 0; i < n; ++i) {
        string str;
```

```
        for (int j = 0; j < 5; ++j) {
            str += alphanum[rand() % (sizeof(alphanum) - 1)];
        }
        arr[i] = str;
    }
    return arr;
}

// Measure execution time
void measureExecutionTime(int n) {
    vector<string> arr = generateRandomStrings(n);

    auto start = chrono::high_resolution_clock::now();
    mergeSort(arr, 0, arr.size() - 1);
    auto end = chrono::high_resolution_clock::now();

    chrono::duration<double> duration = end - start;
    double tn = n * log2(n);
    cout << "Size: " << n << ", T(n): " << tn << ", t(n): " << duration.count() << " seconds" << endl;
}

int main() {
    srand(time(0));
    int sizes[] = {10, 100, 500, 1000, 5000, 10000, 50000, 100000, 500000, 1000000};
    for (int size : sizes) {
        measureExecutionTime(size);
    }
    return 0;
}
```

Output

Size: 10, T(n): 33.2193, t(n): 9.28e-06 seconds
Size: 100, T(n): 664.386, t(n): 5.857e-05 seconds
Size: 500, T(n): 4482.89, t(n): 0.00037707 seconds
Size: 1000, T(n): 9965.78, t(n): 0.00098953 seconds
Size: 5000, T(n): 61438.6, t(n): 0.00510676 seconds
Size: 10000, T(n): 132877, t(n): 0.00919159 seconds
Size: 50000, T(n): 780482, t(n): 0.05398 seconds
Size: 100000, T(n): 1.66096e+06, t(n): 0.110983 seconds
Size: 500000, T(n): 9.46578e+06, t(n): 0.657262 seconds
Size: 1000000, T(n): 1.99316e+07, t(n): 1.3299 seconds

Comparison



While both numeric and alphabetic sorting using Merge Sort share the same theoretical complexity $O(n \log n)$, numeric sorting might have a slight performance edge due to simpler comparisons. Alphabetic sorting can introduce additional overhead depending on string length and character encoding, potentially affecting practical execution times.

Time Complexity

Best Case

- best case occurs when the array is already sorted, so no element swaps are necessary. The algorithm performs $\log n$ levels of recursive division, and at each level, it merges n elements. This results in a time complexity of $O(n \log n)$.

Average Case

- The average case complexity is the scenario where the elements are in random order. the algorithm always divides the array into halves and merges them back, regardless of the initial order of the elements. Thus, the average case also has a time complexity of $O(n \log n)$.

Worst Case

- The worst case for Merge Sort occurs when the elements are in reverse order. the number of divisions and merges remains the same. Hence, the time complexity does not change and remains $O(n \log n)$.

Space Complexity

Merge Sort requires additional space for the temporary arrays used during merging $O(n)$Extra space proportional to the size of the input array is needed to hold the left and right subarrays during the merge process.

Results

Merge Sort is an efficient and reliable sorting algorithm with a consistent time complexity of $O(n \log n)$ across all cases. Its divide-and-conquer approach ensures that it can handle large datasets effectively, making it a valuable tool in a variety of applications.