**Problem Statement:**

Design and implement the **Merge Sort algorithm** using the **Divide and Conquer method** for a given input. Determine the time required to sort the list for large values of *n*.

**Brief About the Problem:**

Sorting is a fundamental problem in computer science, with applications in searching, optimization, and data organization. **Merge Sort** is a stable and efficient sorting algorithm that follows the **Divide and Conquer** paradigm. It is particularly useful for sorting large datasets due to its **consistent O(n log n) time complexity**.

1. **How Merge Sort Works:**
   * **Divide:** The array is recursively split into two equal (or nearly equal) halves.
   * **Conquer:** Each half is sorted recursively using Merge Sort.
   * **Merge:** The sorted halves are merged back together in sorted order.
2. **Why Merge Sort?**
   * Guarantees **O(n log n) time complexity** in all cases (best, average, and worst).
   * Performs well for **large datasets** and **linked lists**, where quick access to elements is not required.
   * More predictable performance compared to Quick Sort, which can degrade to **O(n²) in the worst case**.

The goal is to implement **Merge Sort**, analyze its **time complexity**, and determine the sorting time for **large values of n**.

**Algorithm:**

1. **Divide:**
   * If the array has one element, return (base case).
   * Find the middle index and split the array into two halves.
2. **Conquer:**
   * Recursively sort both halves.
3. **Merge:**
   * Compare elements from both halves and merge them in sorted order.

**Time Complexity Analysis:**

| **Case** | **Merge Sort Time Complexity** |
| --- | --- |
| Best Case | O(n log n) |
| Average Case | O(n log n) |
| Worst Case | O(n log n) |

Since Merge Sort always splits the array and merges in **O(n)** time, its time complexity remains **O(n log n) in all cases**. This makes it an **efficient and reliable** sorting algorithm for large datasets, even when compared to Quick Sort.

#include <iostream>

#include <chrono>

#include <cstdlib> // For rand and srand

#include <ctime> // For time

using namespace std;

using namespace std::chrono;

// Merge function (without vector)

void merge(int arr[], int left, int mid, int right) {

int n1 = mid - left + 1;

int n2 = right - mid;

int\* L = new int[n1];

int\* R = new int[n2];

for (int i = 0; i < n1; i++)

L[i] = arr[left + i];

for (int i = 0; i < n2; i++)

R[i] = arr[mid + 1 + i];

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++;

}

delete[] L;

delete[] R;

}

// Merge Sort function (without vector)

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 n;

cout << "Enter the number of elements: ";

cin >> n;

int\* arr = new int[n]; // Dynamically allocate the array

// Generate random numbers (without algorithm)

srand(time(0)); // Seed the random number generator

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

arr[i] = rand() % 10000; // Generates numbers between 0 and 9999

}

auto start = high\_resolution\_clock::now();

mergeSort(arr, 0, n - 1);

auto stop = high\_resolution\_clock::now();

auto duration = duration\_cast<nanoseconds>(stop - start);

cout << "Time taken for Merge Sort: " << duration.count() << " nanoseconds" << endl;

// Optional: Print the first few elements to verify (for smaller n)

/\*

if (n <= 20) {

cout << "Sorted array: ";

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

cout << arr[i] << " ";

}

cout << endl;

}

\*/

delete[] arr; // Deallocate the dynamically allocated array. VERY IMPORTANT!

return 0;

}



