Implement Recursive-Merge Sort algorithm

```
//Merge Sort Recursive
void mergeRecursive(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 mergeSortRecursive(int arr[], int l, int r){
    if(l < r){
        int m = l + (r - l) / 2;

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

        mergeRecursive(arr, l, m, r);
    }
}
```

Implement Iterative-Merge Sort algorithm (non-recursive)

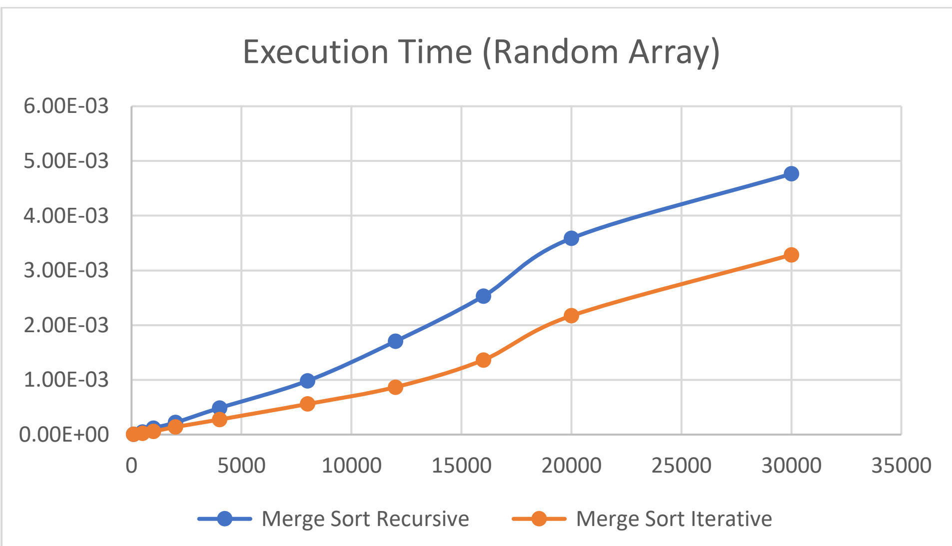
```
//Merge Sort Iterative
void mergeIterative(int arr[], int l, int mid, int r) {
    int n1 = mid - l + 1;
    int n2 = r - mid;
    int L[n1], R[n2];
    for (int i = 0; i < n1; i++) {
        L[i] = arr[l + i];
    }
    for (int j = 0; j < n2; j++) {
        R[j] = arr[mid + 1 + j];
    }
    int 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 mergeSortIterative(int arr[], int n){
    for (int curr_size = 1; curr_size <= n - 1; curr_size = 2 * curr_size) {
        for (int left_start = 0; left_start < n - 1; left_start += 2 *
curr_size) {
            int mid = min(left_start + curr_size - 1, n - 1);
            int right_end = min(left_start + 2 * curr_size - 1, n - 1);
            mergeIterative(arr, left_start, mid, right_end);
        }
    }
}
```

Compare the time between iterative and non-iterative merge sort implementation. Plot the time graph for iterative and non-iterative merge sort

For Random Array

Array Size	Merge Sort Recursive	Merge Sort Iterative
100	8.00E-06	5.00E-06
500	4.50E-05	2.60E-05
1000	0.00012	5.80E-05
2000	0.000221	0.000139
4000	0.000486	0.000275
8000	0.000982	0.000559
12000	0.001705	0.000868
16000	0.00253	0.001364
20000	0.003587	0.002172
30000	0.004765	0.003285



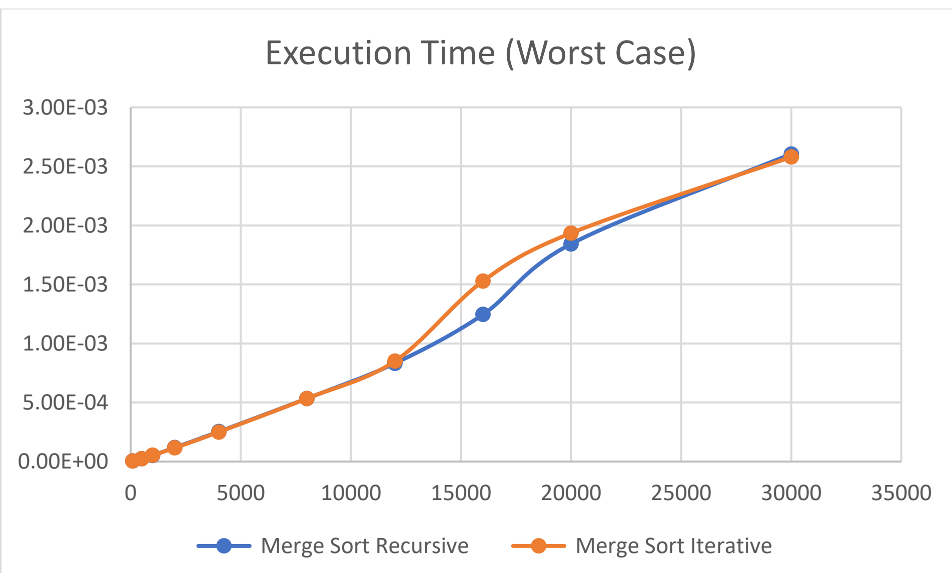
Based on the plotted graph, we can observe that the iterative implementation of Merge Sort is consistently faster than the recursive implementation for all array sizes. This is because the recursive implementation incurs additional overhead due to the recursive function calls that are made during the sorting process. Each recursive call creates a new stack frame, which requires additional memory allocation and deallocation, and incurs a cost for the function call itself. As the array size increases, the number of recursive calls also increases, leading to a higher overhead and longer execution time.

In contrast, the iterative implementation of Merge Sort avoids the overhead associated with recursive function calls by using loops to divide and sort the array. This can make it faster than the recursive implementation, especially for larger arrays.

Therefore, based on the plotted graph and the explanation above, we can conclude that the iterative implementation of Merge Sort is faster than the recursive implementation because it avoids the overhead associated with recursive function calls. The iterative implementation, on the other hand, can handle large arrays efficiently and can be a better choice in cases where performance is critical.

For Worst Case:

Array Size	Merge Sort Recursive	Merge Sort Iterative
100	5.00E-06	5.00E-06
500	2.50E-05	2.50E-05
1000	5.20E-05	5.40E-05
2000	0.000119	0.000115
4000	0.000253	0.00025
8000	0.000534	0.000534
12000	0.000832	0.00085
16000	0.001245	0.001528
20000	0.001843	0.001935
30000	0.002605	0.002579



When we use arrays with elements in descending order, we observed that the execution times for the recursive and iterative implementations of Merge Sort are closer together. This is because in a descending order array, the initial partitioning of the array in Merge Sort may not result in smaller subarrays that are as balanced as they would be in a random array or an ascending order array.

In a descending order array, each partitioning results in one subarray that is much larger than the other. As a result, the recursive calls made in the recursive implementation and the loops used in the iterative implementation will need to perform more comparisons and swaps on the larger subarray, leading to a similar execution time for both implementations.

Furthermore, when the array is sorted in descending order, it is closer to being "sorted" in the opposite direction of how Merge Sort naturally sorts. This means that both the recursive and iterative implementations of Merge Sort will need to perform more swaps and comparisons in order to sort the array compared to a random or ascending order array, which can increase the execution time for both implementations.

Therefore, the use of descending order arrays can result in execution times that are closer together for both implementations, due to the nature of the array being sorted in a way that is not as conducive to the natural behavior of the algorithm.

Full Source Code

```
#include <iostream>
#include <cstdlib>
#include <ctime>
#include <chrono>
#include <fstream>

using namespace std;


//Generate a Array with Random Values
int* generateRandomArray(int size){
    int* arr = new int[size];
    srand(time(NULL)); // seed the random number generator
    for(int i=0; i<size; i++){
        arr[i] = rand(); // generate a random number
    }
    return arr;
}

//Generate Decreasing Array
int* generateDecreasingArray(int size){
    int* arr = new int[size];
    for(int i=0; i<size; i++){
        arr[i] = size - i;
    }
    return arr;
}

//Merge Sort Recursive
void mergeRecursive(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 mergeSortRecursive(int arr[], int l, int r){
    if(l < r){
        int m = l + (r - l) / 2;

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

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

//Merge Sort Iterative
void mergeIterative(int arr[], int l, int mid, int r) {
    int n1 = mid - l + 1;
    int n2 = r - mid;
    int L[n1], R[n2];
```

```
    for (int i = 0; i < n1; i++) {
        L[i] = arr[l + i];
    }
    for (int j = 0; j < n2; j++) {
        R[j] = arr[mid + 1 + j];
    }
    int 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 mergeSortIterative(int arr[], int n){
    for (int curr_size = 1; curr_size <= n - 1; curr_size = 2 * curr_size) {
        for (int left_start = 0; left_start < n - 1; left_start += 2 *
curr_size) {
            int mid = min(left_start + curr_size - 1, n - 1);
            int right_end = min(left_start + 2 * curr_size - 1, n - 1);
            mergeIterative(arr, left_start, mid, right_end);
        }
    }
}



//get Execution Time in millisecond Merge Sort Recursive
double getExecutionTimeMergeSortRecursive(int arr[], int size){
    auto start = chrono::high_resolution_clock::now();
    mergeSortRecursive(arr, 0, size - 1);
    auto end = chrono::high_resolution_clock::now();
    auto duration = chrono::duration_cast<chrono::microseconds>(end - start);
    return duration.count()/ 1000000.0;
}

//get Execution Time in millisecond Merge Sort Iterative
double getExecutionTimeMergeSortIterative(int arr[], int size){
    auto start = chrono::high_resolution_clock::now();
    mergeSortIterative(arr, size);
    auto end = chrono::high_resolution_clock::now();
```

```cpp
    auto duration = chrono::duration_cast<chrono::microseconds>(end - start);
    return duration.count()/ 1000000.0;
}

//Write to CSV file Array Size and Execution Time for Merge Sort Recursive
and Iterative
void writeToFile(int size[], double timeMergeSortRecursive[], double
timeMergeSortIterative[], string const& fileName){
    ofstream file;
    file.open(fileName);
    file << "Array Size, Merge Sort Recursive, Merge Sort Iterative" << endl;
    for(int i = 0; i < 10; i++){
        file << size[i] << "," << timeMergeSortRecursive[i] << "," <<
timeMergeSortIterative[i] << endl;
    }
    file.close();
}

//print array
void printArray(int arr[], int size){
    for(int i = 0; i < size; i++){
        cout << arr[i] << " ";
    }
    cout << endl;
}


int main() {
    int const number_of_arrays = 10;
    int size_of_array[number_of_arrays] =
{100,500,1000,2000,4000,8000,12000,16000, 20000, 30000};

    double executionTimeMergeSortRecursive_Random[number_of_arrays];
    double executionTimeMergeSortIterative_Random[number_of_arrays];

    double executionTimeMergeSortRecursive_Worst[number_of_arrays];
    double executionTimeMergeSortIterative_Worst[number_of_arrays];


    for(int i = 0; i<number_of_arrays; i++){
        int *random_array = generateRandomArray(size_of_array[i]);
        executionTimeMergeSortRecursive_Random[i] =
getExecutionTimeMergeSortRecursive(random_array, size_of_array[i]);
        executionTimeMergeSortIterative_Random[i] =
getExecutionTimeMergeSortIterative(random_array, size_of_array[i]);

        int *decendingArray = generateDecreasingArray(size_of_array[i]);
        executionTimeMergeSortRecursive_Worst[i] =
getExecutionTimeMergeSortRecursive(decendingArray, size_of_array[i]);
        executionTimeMergeSortIterative_Worst[i] =
getExecutionTimeMergeSortIterative(decendingArray, size_of_array[i]);

    }

    writeToFile(size_of_array, executionTimeMergeSortRecursive_Worst,
executionTimeMergeSortIterative_Worst, "mergeSortWorst.csv");
    writeToFile(size_of_array, executionTimeMergeSortRecursive_Random,
```

```
executionTimeMergeSortIterative_Random, "mergeSortRandom.csv");


    return 0;
}
```