Design and Analysis of Algorithms (CS206)

Assignment - 3

U19CS012

1. Given the following algorithms, answer the questions.

• Merge sort: Sorting Problem

<u>Input</u>: A Sequence of Unsorted 'n' numbers, **a1,a2,…,an**

<u>Output</u>: A Permutation (Reordering) (**a1',a2',…,an'**) of Input Sequence
such that **a1'≤ a2'≤ … ≤ an'**

1. Write a program to sort an array, *arr*, consisting *n* numbers using the divide and conquer approach - Use only merge sort.

(1) The divide step should split the array into two (nearly) equal sub-arrays.
(2) The divide step should split the array into three (nearly) equal sub-arrays

1.1. (T) Write pseudocodes to design the algorithms for above mentioned computational problem. Both algorithms should sort the data by dividing them into two and three (nearly) equal sub-arrays respectively.

(1) The divide step should split the array into two (nearly) equal sub-arrays.

A.) MergeSort2 Function

```
• Merge_Sort2(arr,low,high)

1. if low < high
2.    int mid = low + (high-low)/2
// Call this Function to Recursively Divide into Smaller Sub-array [l,m]
3.    Merge_Sort2(arr, low, mid);
// Call this Function to Recursively Divide into Smaller Sub-array [m+1,h]
4.    Merge_Sort2(arr, mid + 1, high);
// Merge the Both Sorted Array
5.    Merge2(arr, low, mid, high);
6. return
```

B.) Merge2 Function

```
// To Merge Two Sorted Array
• Merge2(arr, low, mid, high)
// Create a Temp Array of size high-low+1
    tmp(high - low + 1, 0);
// Crawlers for Temp
    i = low, j = mid + 1, k = 0;

    while i <= mid AND j <= high
      if (arr[i] <= arr[j])
         tmp[k] = arr[i];
         k++;
         i++;
      else
         tmp[k] = arr[j];
         k++;
         j++;

// Remaining Elements in Second Interval
    while i <= mid
         tmp[k] = arr[i];
         i++;
         k++;

// Remaining Elements in Second Interval
    while j <= high
         tmp[k] = arr[j];
         j++;
         k++;

// Copy Temp Array to Original Array
    for i = low to high
       arr[i] = tmp[i - low];
```

(2) The divide step should split the array into three (nearly) equal sub-arrays

A.) MergeSort3 Function

```
• Merge_Sort3(vll &arr, ll low, ll high)

// BASE CASE : 1 Element
    if (high - low < 2)
        return;

    mid1 = low + ((high - low) / 3);
    mid2 = low + 2 * ((high - low) / 3) + 1;

// Call this Function to Recursively Divide into Smaller Sub-array [l,m1)
    Merge_Sort3(arr, low, mid1);
// Call this Function to Recursively Divide into Smaller Sub-array [m1,m2)
    Merge_Sort3(arr, mid1, mid2);
// Call this Function to Recursively Divide into Smaller Sub-array [m1,high)
    Merge_Sort3(arr, mid2, high);
// Merge the Both Sorted Array
    Merge3(arr, low, mid1, mid2, high);
    return;
```

B.) Merge3 Function

```
// To Merge Two Sorted Array
• Merge3(arr, low, mid1, mid2, high)
// Create a Temp Array of size high-low+1
    tmp(high - low + 1, 0);

// Crawlers for Temp
    i = low, j = mid1, k = mid2, id = 0;

    while (i < mid1 && j < mid2 && k < high)
        if (arr[i] < arr[j])
            if (arr[i] < arr[k])
                tmp[id++] = arr[i++];
            else
                tmp[id++] = arr[k++];
        // arr[j] < arr[i]
        else
            if (arr[j] < arr[k])
                tmp[id++] = arr[j++];
            else
                tmp[id++] = arr[k++];
```

```
// i & j
    while ((i < mid1) && (j < mid2))
        if (arr[i] < arr[j])
            tmp[id++] = arr[i++];
        else
            tmp[id++] = arr[j++];

// j & k
    while ((j < mid2) && (k < high))
        if (arr[j] < arr[k])
            tmp[id++] = arr[j++];
        else
            tmp[id++] = arr[k++];

// i & k
    while ((i < mid1) && (k < high))
        if (arr[i] < arr[k])
            tmp[id++] = arr[i++];
        else
            tmp[id++] = arr[k++];

// Copy Remaining Elements
// [0,mid1)
    while (i < mid1)
        tmp[id++] = arr[i++];

// [mid1,mid2)
    while (j < mid2)
        tmp[id++] = arr[j++];

// [mid2,high)
    while (k < high)
        tmp[id++] = arr[k++];

// Copy Temp Array to Original Array
    for (i = low; i < high; i++)
        arr[i] = tmp[i - low];
```

1.2. (T) Analyze the time complexity of both algorithms (split the array into two and three sub-arrays) using the recursion tree method.

A.) Merge Sort Analysis by Dividing into Two Parts

U19CS012

Merge-sort (A, p, r) // T(n) (Assume)

if $p < r$ $+ c$

$q = \lfloor (p+r)/2 \rfloor$ $+ c$

Merge-sort (A, p, q) $+ T(n/2)$

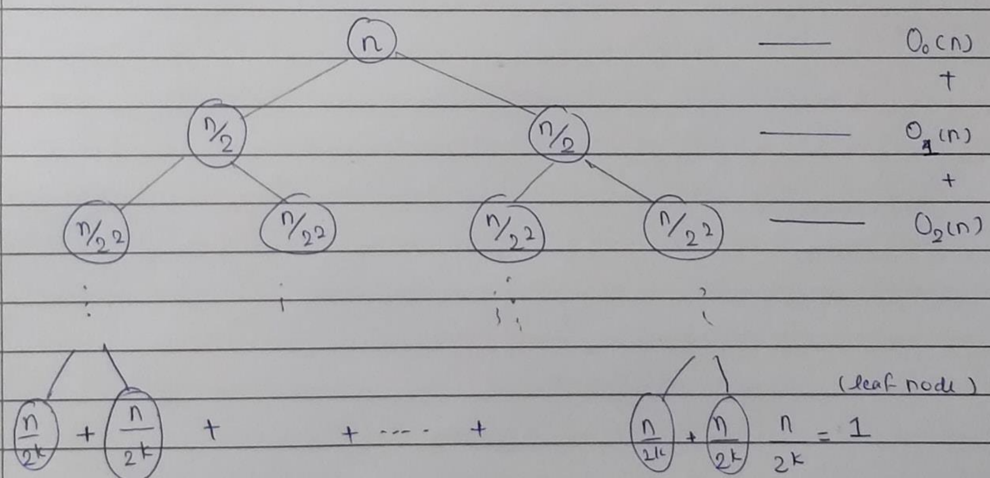
Merge-sort (A, q+1, r) $+ T(n/2)$

Merge (A, p, q, r) $+ \Theta(n)$

(∵ in function merge it take linear time)

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

Recursion Tree



$[k = \log_2(n)]$

$$O\left(\sum_{i=0}^{k} 2^i \frac{n}{(2^i)}\right) = O\left(\sum_{i=0}^{k} n\right) = O(k.n)$$

$$= O(\log_2(n) * n)$$

∴ Time complexity for Merge sort $= \Theta(n \log_2(n))$

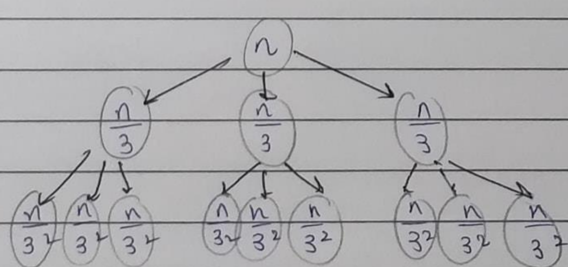
B.) Merge Sort Analysis by Dividing into Three Parts

U19CS012

Following the similar logic
(for 3 parts)

```
Merge-sort3( arr, low, high)                        // T(n)
    if( high-low)< 2                                 + c
        return
    mid1 = low + ((high-low)/3 );                    + c
    mid2 = low + 2*((high-low)/3 );                  + c

    Merge-sort3 ( arr, low, mid1 );                  T(n/3)
    Merge-sort 3 ( arr, mid1, mid2);                 T(n/3)
    Merge-sort 3 ( arr, mid2, high);                 T(n/3)
    Merge ( arr, low, mid1, mid2, high);             Θ(n)
```

$$T(n) = 3T\left(\frac{n}{3}\right) + \Theta(n)$$



time
cn
$3\left(\frac{n}{3}\right)$
$3^2\left(\frac{n}{3}\right)$

$\frac{n}{3^k}$ ---- ---- ---- $\frac{n}{3^k}$

leaf Node

$\frac{n}{3^k} = 1 \quad [k = \log_3 n]$

$$O\left(\sum_{i=0}^{k} 3^i \frac{n}{3^i}\right) = O\left(\sum_{i=0}^{k} n\right) = O(k \cdot n)$$

$$= O(\log_3(n) * n)$$

∴ Time complexity for Merge sort = $\Theta(n \log_3(n))$

1.3. (L) Provide the details of Hardware/Software you used to implement algorithms and to measure the time.

Hardware Details of My Laptop:

PARAMETER	LAPTOP CONFIGURATION
Operating System	Microsoft Windows **10**.0.19042
Processor	Intel(R) Core(TM) i5-10210U [Core **i5 10th Gen**]
CPU	**1.60GHz**, 2112 Mhz, **4** Core(s), 8 Logical Processor(s)
System Type	x64-based PC [**64 Bit**]
RAM	**8.00** GB
Hard Drive/SSD	512 GB **SSD**

Software Used:

PARAMETER	LAPTOP CONFIGURATION
Code Editor	**Visual Studio** Code [Version 1.52]
Compiler	gcc (MinGW.org **GCC-8.2.0-5**) 8.2.0
Time	Measured using **chrono** Library in C++
Programming Language Used	**C++**

1.4. (L) Submit the code (complete programs).

A.) ***Merge Sort Program*** by Dividing into **Two Parts**

```
// HEADERS AND NAMESPACE
#include <bits/stdc++.h>
// INSTEAD OF ALL THESE
#include <iostream>
// For Creating File
#include <fstream>
#include <vector>
// For set - precision
#include <iomanip>
// For Time Calculation
#include <chrono>
// For File Name and Output File Name
```

```cpp
#include <string>

using namespace std;
using namespace std::chrono;

// COMMONLY USED TYPES
typedef long long ll;
typedef vector<ll> vll;

// Basic Algorithm Implementation of Merge Sort
// To Merge Two Sorted Array
void merge(vll &arr, ll low, ll mid, ll high)
{
    // Create a Temp Array
    vll tmp(high - low + 1, 0);

    // Crawlers for Temp
    ll i = low, j = mid + 1, k = 0;

    while (i <= mid && j <= high)
    {
        if (arr[i] <= arr[j])
        {
            tmp[k] = arr[i];
            k++;
            i++;
        }
        else
        {
            tmp[k] = arr[j];
            k++;
            j++;
        }
    }

    // Remaining Elements in Second Interval
    while (i <= mid)
    {
        tmp[k] = arr[i];
        i++;
        k++;
    }

    // Remaining Elements in Second Interval
    while (j <= high)
    {
        tmp[k] = arr[j];
        j++;
        k++;
    }
```

```cpp
    // Copy Temp Array to Original Array
    for (i = low; i <= high; i++)
    {
        arr[i] = tmp[i - low];
    }
}

// Real Merge Sort Function
void merge_sort(vll &arr, ll low, ll high)
{

    if (low < high)
    {
        ll mid = low + (high - low) / 2; // To Avoid Overflow
        // Call this Function to Recursively Divide into Smaller Sub-array [l,m]
        merge_sort(arr, low, mid);
        // Call this Function to Recursively Divide into Smaller Sub-array [m+1,h]
        merge_sort(arr, mid + 1, high);
        // Merge the Both Sorted Array
        merge(arr, low, mid, high);
    }

    return;
}

int main()
{
    // For Read & Write from "Input File" and  Return Output to "Output" File
    freopen("output.txt", "w", stdout);

    // EDIT THIS FILE NUMBER , LIMIT and Number of Times File Runs
    int file_no = 1;
    int limit = 10;
    int each_file_runs = 2;

    for (; file_no <= limit; file_no++)
    {
        string inp_file = "File";
        string num = to_string(file_no);
        string ext = ".txt";
        inp_file += num;
        inp_file += ext;

        ifstream File;
        File.open(inp_file);

        vector<ll> arr;

        ll number, idx = 0;
```

```cpp
    while (!File.eof())
    {
        File >> number;
        arr.push_back(number);
    }

    ll Best_Duration = 0, Worst_Duration = 0, Average_Duration = 0;
    auto start = high_resolution_clock::now();
    auto end = high_resolution_clock::now();
    auto time_taken = duration_cast<nanoseconds>(end - start);
    ll n1 = arr.size();
    for (int f = 0; f < each_file_runs; f++)
    {
        // ------------------------AVERAGE CASE [O(n^2)]----------------------------

        start = high_resolution_clock::now();
        // Function Here
        merge_sort(arr, 0, arr.size() - 1);
        // Function Ends here
        end = high_resolution_clock::now();
        time_taken = duration_cast<nanoseconds>(end - start);
        Average_Duration += time_taken.count();

        // ------------------------BEST CASE [0(n^2)]----------------------------
        // The Array is Already Sorted from Average Case, So it Becomes out Best Case
        // sort(arr.begin(), arr.end());
        start = high_resolution_clock::now();
        // Function Here
        merge_sort(arr, 0, arr.size() - 1);
        // Function Ends here
        end = high_resolution_clock::now();
        time_taken = duration_cast<nanoseconds>(end - start);
        Best_Duration += time_taken.count();

        // ------------------------WORST CASE [0(n^2)]----------------------------
        // This will Reverse the Sorted Array, Therfore we will Get the Worst Case

        reverse(arr.begin(), arr.end());
        // sort(arr.begin(), arr.end(), greater<ll>());
        start = high_resolution_clock::now();
        // Function Here
        merge_sort(arr, 0, arr.size() - 1);
        // Function Ends here
        end = high_resolution_clock::now();
        time_taken = duration_cast<nanoseconds>(end - start);
        Worst_Duration += time_taken.count();
    }

    cout << "--------------------------------------------------------" << endl;
    cout << inp_file << endl;
```

```
        cout << "AVERAGE CASE : ";
        double avg = (double)Average_Duration / (double)each_file_runs;
        avg *= 1e-9;
        cout << fixed << avg << setprecision(9);
        cout << " seconds" << endl;
        cout << "BEST CASE    : ";
        double best = (double)Best_Duration / (double)each_file_runs;
        best *= 1e-9;
        cout << fixed << best << setprecision(9);
        cout << " seconds" << endl;
        cout << "WORST CASE   : ";
        double worst = (double)Worst_Duration / (double)each_file_runs;
        worst *= 1e-9;
        cout << fixed << worst << setprecision(9);
        cout << " seconds" << endl;
    }

    return 0;
}
```

B.) ***Merge Sort Program*** by Dividing into **Three Parts**

```
// HEADERS AND NAMESPACE
#include <bits/stdc++.h>
// INSTEAD OF ALL THESE
#include <iostream>
// For Creating File
#include <fstream>
#include <vector>
// For set - precision
#include <iomanip>
// For Time Calculation
#include <chrono>
// For File Name and Output File Name
#include <string>

using namespace std;
using namespace std::chrono;

// COMMONLY USED TYPES
typedef long long ll;
typedef vector<ll> vll;

// Basic Algorithm Implementation of Merge Sort
// To Merge Two Sorted Array
void merge(vll &arr, ll low, ll mid1, ll mid2, ll high)
{
    // Create a Temp Array
```

```
    vll tmp(high - low + 1, 0);

    // Crawlers for Temp
    ll i = low, j = mid1, k = mid2, id = 0;

    while (i < mid1 && j < mid2 && k < high)
    {
        if (arr[i] < arr[j])
        {
            if (arr[i] < arr[k])
            {
                tmp[id++] = arr[i++];
            }
            else
            {
                tmp[id++] = arr[k++];
            }
        }
        // arr[j] < arr[i]
        else
        {
            if (arr[j] < arr[k])
            {
                tmp[id++] = arr[j++];
            }
            else
            {
                tmp[id++] = arr[k++];
            }
        }
    }

    // i & j
    while ((i < mid1) && (j < mid2))
    {
        if (arr[i] < arr[j])
        {
            tmp[id++] = arr[i++];
        }
        else
        {
            tmp[id++] = arr[j++];
        }
    }
    // j & k
    while ((j < mid2) && (k < high))
    {
        if (arr[j] < arr[k])
        {
            tmp[id++] = arr[j++];
```

```cpp
        }
        else
        {
            tmp[id++] = arr[k++];
        }
    }
    // i & k
    while ((i < mid1) && (k < high))
    {
        if (arr[i] < arr[k])
        {
            tmp[id++] = arr[i++];
        }
        else
        {
            tmp[id++] = arr[k++];
        }
    }

    // Copy Remaining Elements
    // [0,mid1)
    while (i < mid1)
    {
        tmp[id++] = arr[i++];
    }
    // [mid1,mid2)
    while (j < mid2)
    {
        tmp[id++] = arr[j++];
    }
    // [mid2,high)
    while (k < high)
    {
        tmp[id++] = arr[k++];
    }

    // Copy Temp Array to Original Array
    for (i = low; i < high; i++)
    {
        arr[i] = tmp[i - low];
    }
}

// Real Merge Sort Function
void merge_sort(vll &arr, ll low, ll high)
{
    // BASE CASE : 1 Element
    if (high - low < 2)
    {
        return;
```

```cpp
    }

    ll mid1 = low + ((high - low) / 3);
    ll mid2 = low + 2 * ((high - low) / 3) + 1;

    // Call this Function to Recursively Divide into Smaller Sub-array [l,m1)
    merge_sort(arr, low, mid1);
    // Call this Function to Recursively Divide into Smaller Sub-array [m1,m2)
    merge_sort(arr, mid1, mid2);
    // Call this Function to Recursively Divide into Smaller Sub-array [m1,high)
    merge_sort(arr, mid2, high);
    // Merge the Both Sorted Array
    merge(arr, low, mid1, mid2, high);

    return;
}

int main()
{
    // For Read & Write from "Input File" and  Return Output to "Output" File
    freopen("output.txt", "w", stdout);

    // EDIT THIS FILE NUMBER , LIMIT and Number of Times File Runs
    int file_no = 1;
    int limit = 10;
    int each_file_runs = 3;

    for (; file_no <= limit; file_no++)
    {
        string inp_file = "File";
        string num = to_string(file_no);
        string ext = ".txt";
        inp_file += num;
        inp_file += ext;

        ifstream File;
        File.open(inp_file);

        vector<ll> arr;

        ll number, idx = 0;
        while (!File.eof())
        {
            File >> number;
            arr.push_back(number);
        }

        ll Best_Duration = 0, Worst_Duration = 0, Average_Duration = 0;
        auto start = high_resolution_clock::now();
        auto end = high_resolution_clock::now();
```

```cpp
    auto time_taken = duration_cast<nanoseconds>(end - start);
    ll n1 = arr.size();
    for (int f = 0; f < each_file_runs; f++)
    {
        // ------------------------AVERAGE CASE [O(n^2)]---------------------------

        start = high_resolution_clock::now();
        // Function Here
        merge_sort(arr, 0, arr.size());
        // Function Ends here
        end = high_resolution_clock::now();
        time_taken = duration_cast<nanoseconds>(end - start);
        Average_Duration += time_taken.count();

        // ------------------------BEST CASE [0(n^2)]-----------------------------
        // The Array is Already Sorted from Average Case, So it Becomes out Best Case
        // sort(arr.begin(), arr.end());
        start = high_resolution_clock::now();
        // Function Here
        merge_sort(arr, 0, arr.size() - 1);
        // Function Ends here
        end = high_resolution_clock::now();
        time_taken = duration_cast<nanoseconds>(end - start);
        Best_Duration += time_taken.count();

        // ------------------------WORST CASE [0(n^2)]-----------------------------
        // This will Reverse the Sorted Array, Therfore we will Get the Worst Case

        reverse(arr.begin(), arr.end());
        // sort(arr.begin(), arr.end(), greater<ll>());
        start = high_resolution_clock::now();
        // Function Here
        merge_sort(arr, 0, arr.size() - 1);
        // Function Ends here
        end = high_resolution_clock::now();
        time_taken = duration_cast<nanoseconds>(end - start);
        Worst_Duration += time_taken.count();
    }

    cout << "---------------------------------------------------------" << endl;
    cout << inp_file << endl;
    cout << "AVERAGE CASE : ";
    double avg = (double)Average_Duration / (double)each_file_runs;
    avg *= 1e-9;
    cout << fixed << avg << setprecision(9);
    cout << " seconds" << endl;
    cout << "BEST CASE    : ";
    double best = (double)Best_Duration / (double)each_file_runs;
    best *= 1e-9;
    cout << fixed << best << setprecision(9);
```

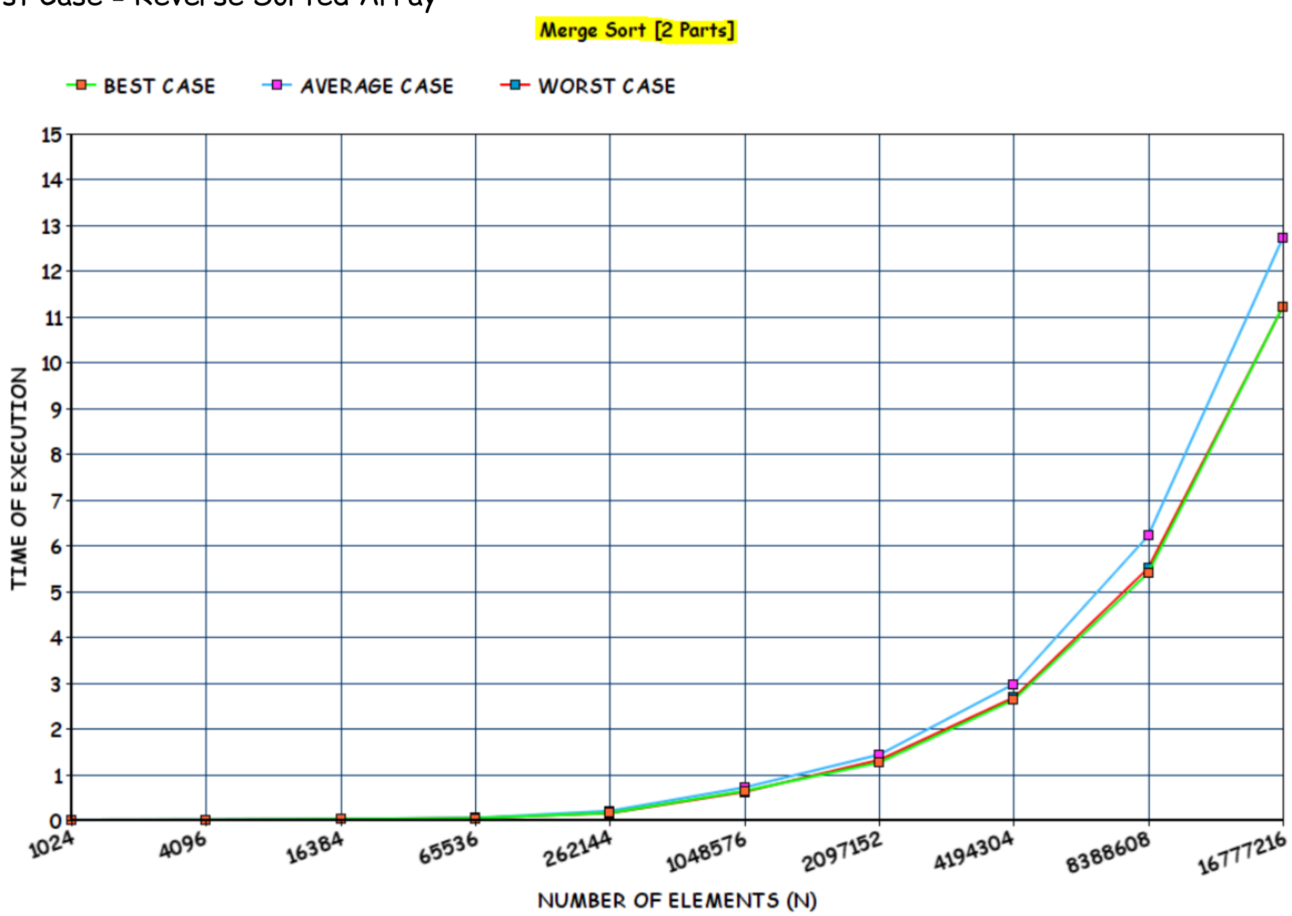
```cpp
        cout << " seconds" << endl;
        cout << "WORST CASE    : ";
        double worst = (double)Worst_Duration / (double)each_file_runs;
        worst *= 1e-9;
        cout << fixed << worst << setprecision(9);
        cout << " seconds" << endl;
    }

    return 0;
}
```

1.5. (L) Measure the best-case time, average-case time and worst-case time of the above two algorithms for all ten files (Assignment 1). Plot a graph.

A.) ***Merge Sort Program*** by Dividing into **Two Parts**

FILE	No. of Elements	BEST CASE [in sec]	AVERAGE CASE [in sec]	WORST CASE [in sec]
1	1024 = 2^10	0.000000000	0.000519000	0.000000000
2	4096 = 2^12	0.003997500	0.008046500	0.003997500
3	16384 = 2^14	0.017086500	0.017533500	0.017251500
4	65536 = 2^16	0.037438500	0.043737500	0.042738500
5	262144 = 2^18	0.154819000	0.198323000	0.147636500
6	1048576 = 2^20	0.634084000	0.711899500	0.616180000
7	2097152 = 2^21	1.255310000	1.427379000	1.311771000
8	4194304 = 2^22	2.629683500	2.973717500	2.677120500
9	8388608 = 2^23	5.407570500	6.219963500	5.513057500
10	16777216 = 2^24	11.222652500	12.724297000	11.212041500

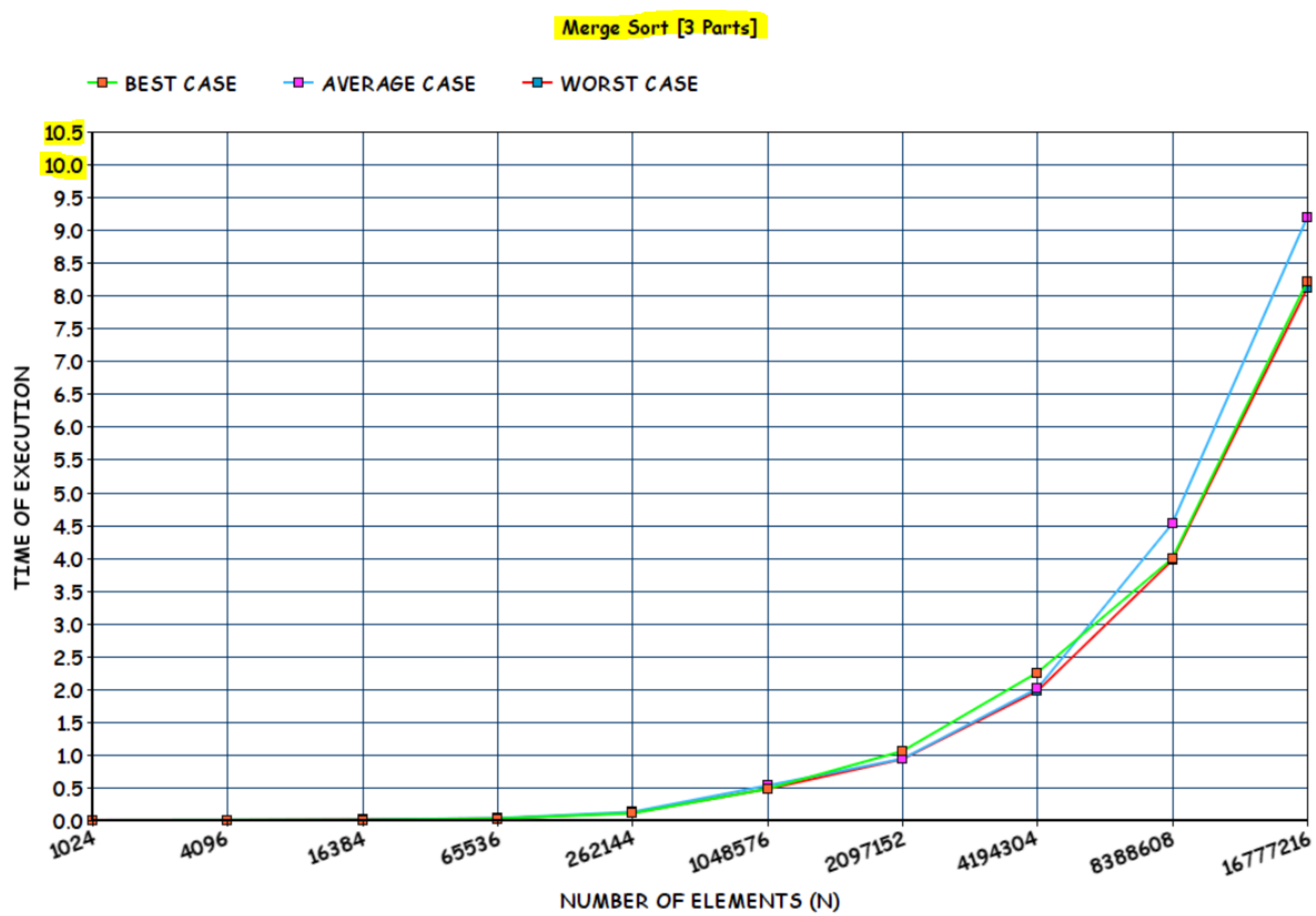
*Worst Case = Reverse Sorted Array



B.) ***Merge Sort Program*** by Dividing into **Three Parts**

FILE	No. of Elements	BEST CASE [in sec]	AVERAGE CASE [in sec]	WORST CASE [in sec]
1	1024 = 2^10	0.000000000	0.000332	0.001713000
2	4096 = 2^12	0.003036000	0.002839667	0.001595667
3	16384 = 2^14	0.003331667	0.007095667	0.010413333
4	65536 = 2^16	0.024336000	0.030064667	0.025325667
5	262144 = 2^18	0.105656333	0.126667667	0.111899000
6	1048576 = 2^20	0.477917333	0.528992333	0.476208667
7	2097152 = 2^21	1.054969667	0.936536667	0.931854000
8	4194304 = 2^22	2.252637667	2.009962000	1.972199000
9	8388608 = 2^23	3.994126333	4.524697667	3.966805333
10	16777216 = 2^24	8.216317333	9.190697000	8.110807000

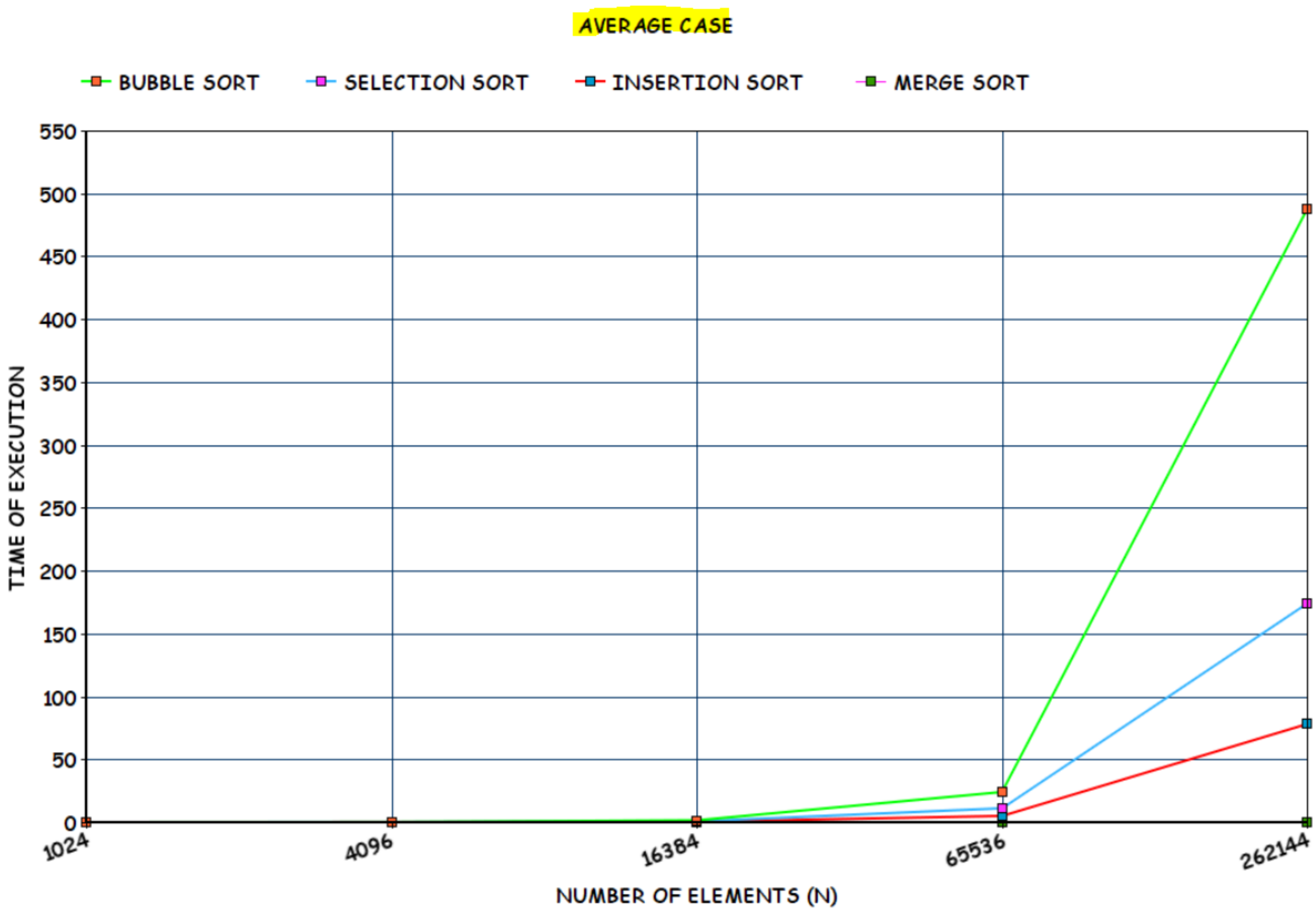
*Worst Case = Reverse Sorted Array



1.7. (L) Compare the average-case performance of bubble sort, selection sort, insertion sort, and merge sort for all ten files. Plot a graph.

AVERAGE CASE

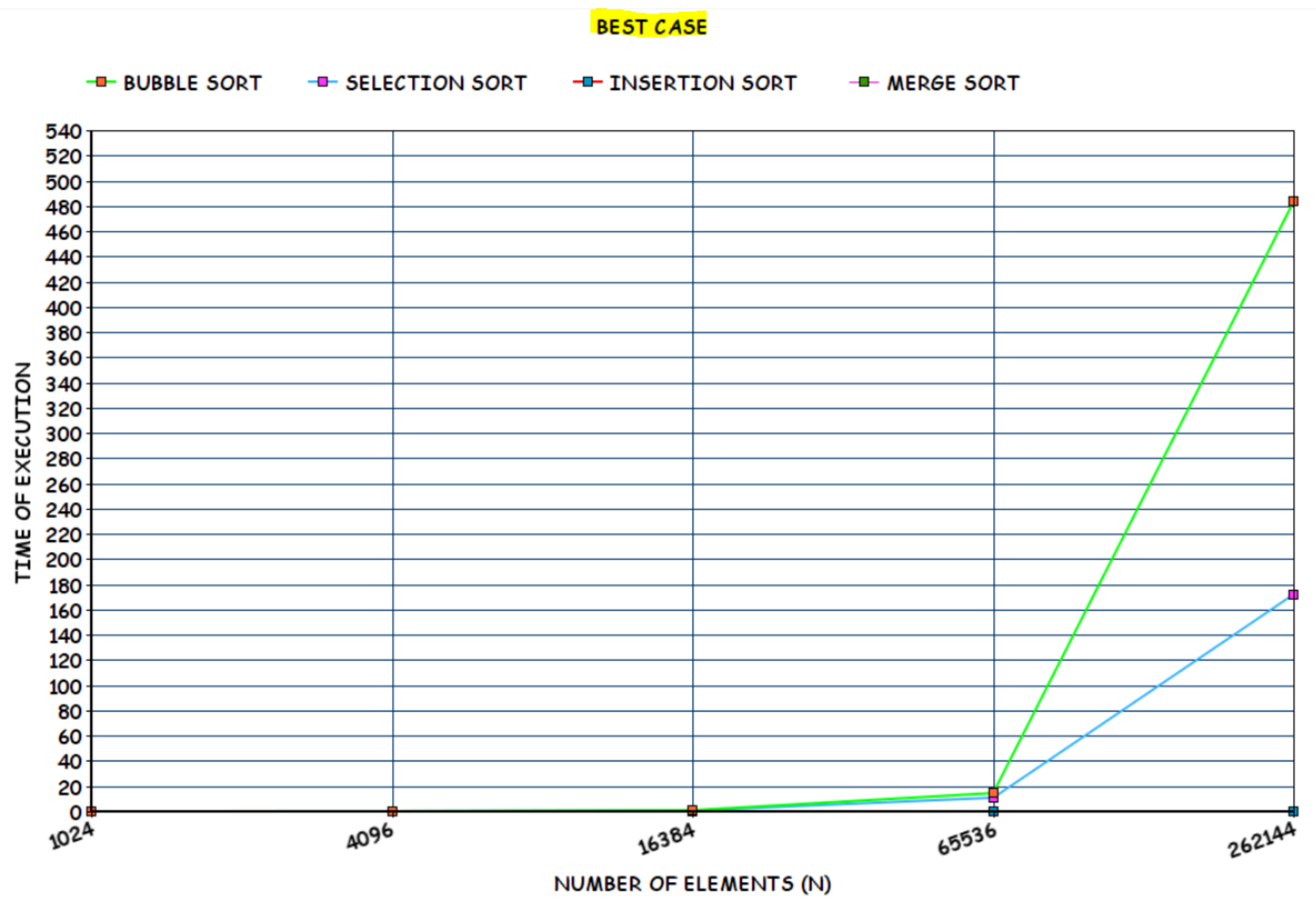
File	No. Of Elements	Bubble Sort	Selection Sort	Insertion Sort	Merge Sort
1	2^10	0.01099400	0.002992000	0.000000000	0.000519000
2	2^12	0.111313000	0.050864000	0.038989000	0.008046500
3	2^14	1.501978000	0.669212000	0.345732500	0.017533500
4	2^16	24.017980000	11.059123000	4.971833000	0.043737500
5	2^18	487.501293000	174.081319000	78.119213000	0.198323000



1.6. (L) Compare the best-case performance of bubble sort, selection sort, insertion sort, and merge sort for all ten files. Plot a graph.

BEST CASE

File	No. Of Elements	Bubble Sort	Selection Sort	Insertion Sort	Merge Sort
1	2^10	0.008177500	0.001993000	0.000000000	0.000000000
2	2^12	0.057657500	0.048870000	0.000000000	0.003997500
3	2^14	0.913967500	0.667217000	0.005000000	0.017086500
4	2^16	14.563756500	10.864665000	0.000000000	0.037438500
5	2^18	483.525254500	172.253910000	0.001998500	0.154819000

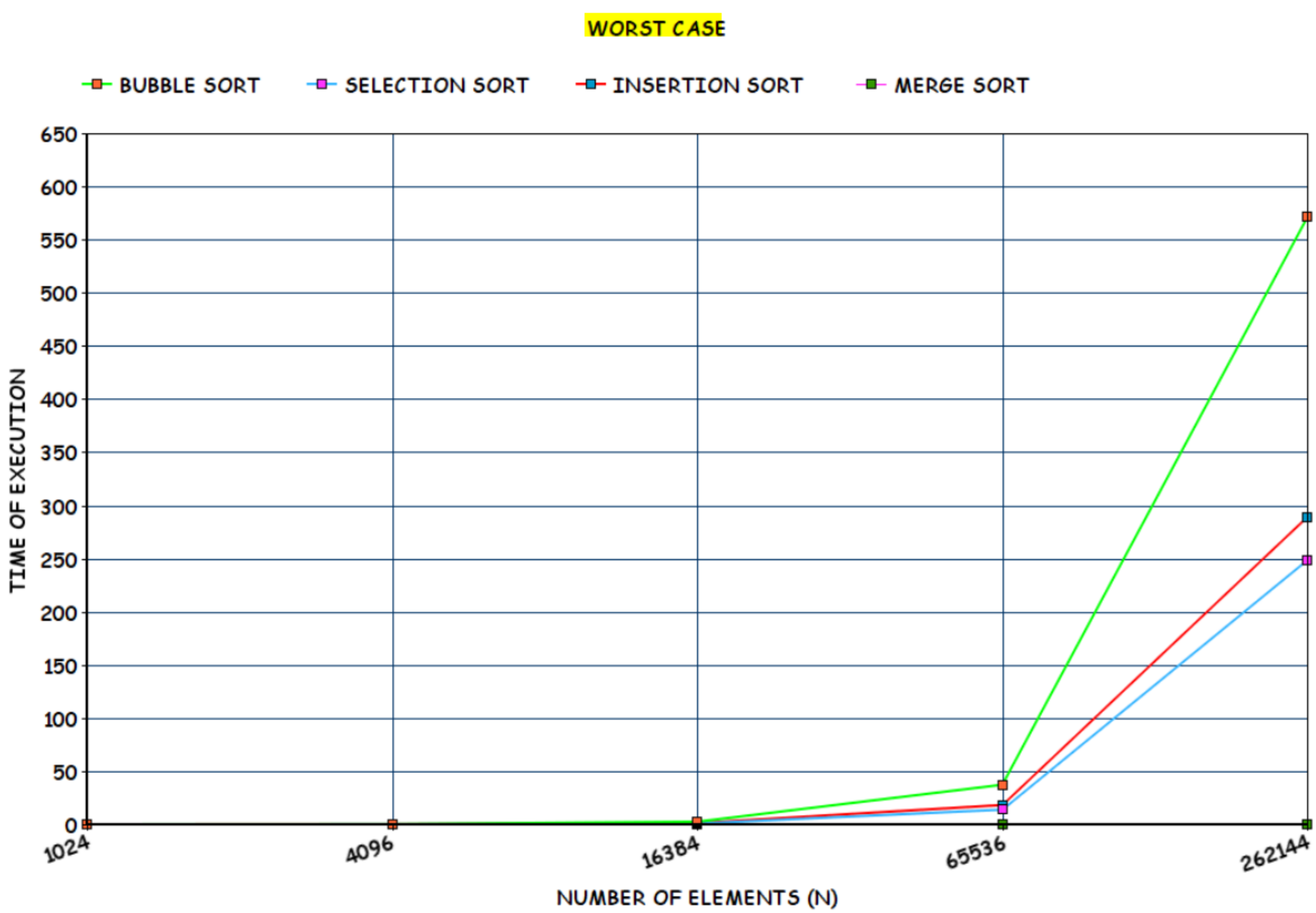


1.8. (L) Compare the worst-case performance of bubble sort, selection sort, insertion sort, and merge sort for all ten files. Plot a graph.

WORST CASE

File	No. Of Elements	Bubble Sort	Selection Sort	Insertion Sort	Merge Sort [Rev Sort]
1	2^10	0.014323500	0.003026000	0.010075500	0.000000000
2	2^12	0.142320000	0.045876000	0.086874000	0.003997500
3	2^14	2.449665000	0.743014000	1.208551500	0.017251500
4	2^16	37.231916500	13.791273000	18.156971000	0.042738500
5	2^18	571.021661500	248.630703000	289.277434500	0.147636500

After File 5 Onwards, It would take a Minimum of 2 hrs for Each File Execution. So Avoided Executing for Rest of the Files.



Note: *Real Worst Case is Calculated in Following Manner:*

The worst case of merge sort will be the one where merge sort will have to do **maximum number of comparisons.**

So I will try building the worst case in bottom up manner:

1. Suppose the array in final step after sorting is `{0,1,2,3,4,5,6,7}`
2. For worst case the array before this step must be `{0,2,4,6,1,3,5,7}` because here left subarray=`{0,2,4,6}` and right subarray=`{1,3,5,7}` will result in maximum comparisons. (*Storing alternate elemets in left and right subarray*)

   Reason: Every element of array will be compared atleast once.
3. Applying the same above logic for left and right subarray for previous steps : For array `{0,2,4,6}` the worst case will be if the previous array is `{0,4}` and `{2,6}` and for array `{1,3,5,7}` the worst case will be for `{1,5}` and `{3,7}`.
4. Now applying the same for previous step arrays: *For worst cases*: `{0,4}` must be `{4,0}` , `{2,6}` must be `{6,2}` , `{1,5}` must be `{5,1}` `{3,7}` must be `{7,3}` . Well if you look clearly this step is **not necessary** because if the size of set/array is 2 then every element will be compared atleast once even if array of size 2 is sorted.

```
Applying Merge Sort using Divide and Conquer

Input array arr[] = [4,0,6,2,5,1,7,3]
                              /   \
                             /     \
                  [4,0,6,2] and [5,1,7,3]
                     / \           / \
                    /   \         /   \
                 [4,0] [6,2]    [5,1] [7,3]
                   |     |        |     |
                   |     |        |     |
                 [0,4] [2,6]    [1,5] [3,7]
                   \     /        \     /
                    \   /          \   /
                 [0,2,4,6]      [1,3,5,7]
                      \            /
                       \          /
                    [0,1,2,3,4,5,6,7]
```

MERGE SORT ALL CASE [THEORATICAL CALCULATION]

FILE	NUMBER OF ELEMENTS	NO OF OPERATIONS $O(N*LOG_2(N))$	APPROX TIME TAKEN [OP/10^8]
FILE 1	1024 = 2^10	1024*10	0.0001024
FILE 2	4096 = 2^12	4096*12	0.00049152
FILE 3	16384 = 2^14	16384*14	0.00229376
FILE 4	65536 = 2^16	65536*16	0.01048576
FILE 5	262144 = 2^18	262144*18	0.04718592
FILE 6	1048576 = 2^20	1048576*20	0.20971520
FILE 7	2097152 = 2^21	2097152*21	0.44040192
FILE 8	4194304 = 2^22	4194304*22	0.92274688
FILE 9	8388608 = 2^23	8388608*23	1.92937984
FILE 10	16777216 = 2^24	16777216*24	4.02653184

For Bubble Sort, Selection Sort and Insertion Sort.

BEST CASE [THEORATICAL CALCULATION]

FILE	NUMBER OF ELEMENTS	NO OF OPERATIONS [CASE] = O(N)	APPROX TIME TAKEN [OP/10^8]
FILE 1	1024 = 2^10	1024	0.00001024
FILE 2	4096 = 2^12	4096	0.00004096
FILE 3	16384 = 2^14	16384	0.00016384
FILE 4	65536 = 2^16	65536	0.00065536
FILE 5	262144 = 2^18	262144	0.00262144
FILE 6	1048576 = 2^20	1048576	0.01048576
FILE 7	2097152 = 2^21	2097152	0.02097152
FILE 8	4194304 = 2^22	4194304	0.04194304
FILE 9	8388608 = 2^23	8388608	0.08388608
FILE 10	16777216 = 2^24	16777216	0.16777216

WORST/AVERAGE CASE [THEORATICAL CALCULATION]

FILE	NUMBER OF ELEMENTS	NO OF OPERATIONS [CASE] = O(N^2)	APPROX TIME TAKEN [OP/10^8]
FILE 1	1024 = 2^10	2^20	0.0104 seconds = 0.01 sec
FILE 2	4096 = 2^12	2^24	0.167 seconds = 0.16 sec
FILE 3	16384 = 2^14	2^28	2.684 seconds = 2.6 sec
FILE 4	65536 = 2^16	2^32	43 seconds = 43 sec
FILE 5	262144 = 2^18	2^36	687 seconds = 11 mins
FILE 6	1048576 = 2^20	2^40	10995 seconds = 3 hrs 3 mins
FILE 7	2097152 = 2^21	2^42	43980 seconds = 12 hrs 13 mins
FILE 8	4194304 = 2^22	2^44	175922 seconds = 2 days 52 hrs 2 mins
FILE 9	8388608 = 2^23	2^46	703687 seconds = 8 days 3 hrs 28 mins
FILE 10	16777216 = 2^24	2^48	2814750 seconds = 32 days 13 hrs 52 mins

CONCLUSION:

Bubble sort: *repeatedly compare neighbor pairs and swap if necessary.*

Selection sort: *repeatedly pick the smallest element to append to the result.*

Insertion sort: *repeatedly add new element to the sorted result.*

Sorting Algorithm	Time Complexity			Space Complexity
	Best Case	Average Case	Worst Case	Worst Case
Bubble Sort	$O(N)$	$O(N^2)$	$O(N^2)$	$O(1)$
Selection Sort	$O(N^2)$	$O(N^2)$	$O(N^2)$	$O(1)$
Insertion Sort	$O(N)$	$O(N^2)$	$O(N^2)$	$O(1)$

Merge Sort:

1.) *If it is only one element in the list it is already sorted, return.*

2.) ***Divide** the list recursively into two halves until it can no more be divided.*

3.) ***Merge** the smaller lists into new list in sorted order.*

- ✓ *Merge Sort is useful for **sorting linked lists**.*
- ✓ *Merge Sort is a **stable sort** which means that the same element in an array maintain their original positions with respect to each other.*
- ✓ *Overall time complexity of Merge sort is O(nLogn).*
  It is more efficient as it is in worst case also the runtime is O(nlogn)
- ✓ *The space complexity of Merge sort is O(n). [Not In-Place]*
- ✓ *This means that this algorithm takes a **lot of space** and may slower down operations for the last data sets.*

SUBMITTED BY:

U19CS012

BHAGYA VINOD RANA