****

**NAME: -** Darji Akshatkumar Hiteshbhai

**RollNo: -** 23MCD001

**Branch: -** M.tech-CSE**(Data Science)**

**Subject: -** Complexity Theory & Algorithms

**Practical-2**

**Aim:** Perform Merge Sort, External Merge sort for the input size 10000, 50000 and 100000 for Ascending, Descending & Random order array. Plot the chart of the output data and do the analysis which algorithm is best and justify your reason.

**Code for Merge sort-**

#include <bits/stdc++.h>

using namespace std;

using namespace std::chrono;

void acending(vector<int> &arr, int n)

{

    for (int i = 0; i < n; i++)

    {

        arr[i] = i;

    }

}

void decending(vector<int> &arr, int n)

{

    for (int i = 0; i < n; i++)

    {

        arr[i] = n - i - 1;

    }

}

void random(vector<int> &arr, int n)

{

    for (int i = 0; i < n; i++)

    {

        arr[i] = rand() % n;

    }

}

void merge(vector<int> &arr, int low, int mid, int high)

{

    vector<int> temp;

    int left = low;

    int right = mid + 1;

    while (left <= mid && right <= high)

    {

        if (arr[left] <= arr[right])

        {

            temp.push\_back(arr[left]);

            left++;

        }

        else

        {

            temp.push\_back(arr[right]);

            right++;

        }

    }

    while (left <= mid)

    {

        temp.push\_back(arr[left]);

        left++;

    }

    while (right <= high)

    {

        temp.push\_back(arr[right]);

        right++;

    }

    for (int i = low; i <= high; i++)

    {

        arr[i] = temp[i - low];

    }

}

void mergesort(vector<int> &arr, int low, int high)

{

    if (low >= high)

        return;

    int mid = floor((low + high) / 2);

    mergesort(arr, low, mid);

    mergesort(arr, mid + 1, high);

    merge(arr, low, mid, high);

}

void mergesort\_a(vector<int> &arr, int n)

{

    acending(arr, n);

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

    mergesort(arr, 0, n - 1);

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

    duration<double> total = end - start;

    for (int i = 0; i < 200; i++)

    {

        cout << arr[i] << " ";

    }

    cout << endl;

    cout << "Total time taken by merge sort for " << n << " elements in Ascending order is: " << total.count() << endl;

    cout << "--------------------------------------------------------------------------------------------------------";

    cout << endl;

}

void mergesort\_d(vector<int> &arr, int n)

{

    decending(arr, n);

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

    mergesort(arr, 0, n - 1);

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

    duration<double> total = end - start;

    for (int i = 0; i < 200; i++)

    {

        cout << arr[i] << " ";

    }

    cout << endl;

    cout << "Total time taken by merge sort for " << n << " elements in Decending order is: " << total.count() << endl;

    cout << "--------------------------------------------------------------------------------------------------------";

    cout << endl;

}

void mergesort\_r(vector<int> &arr, int n)

{

    random(arr, n);

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

    mergesort(arr, 0, n - 1);

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

    duration<double> total = end - start;

    for (int i = 0; i < 200; i++)

    {

        cout << arr[i] << " ";

    }

    cout << endl;

    cout << "Total time taken by merge sort for " << n << " elements in random order is: " << total.count() << endl;

    cout << "--------------------------------------------------------------------------------------------------------";

    cout << endl;

}

int main()

{

    int n;

    cout << "Enter the size of an Array: ";

    cin >> n;

    vector<int> arr(n);

    for (int i = 0; i <= 3; i++)

    {

        if (i == 1)

        {

            mergesort\_a(arr, n);

        }

        if (i == 2)

        {

            mergesort\_d(arr, n);

        }

        if (i == 3)

        {

            mergesort\_r(arr, n);

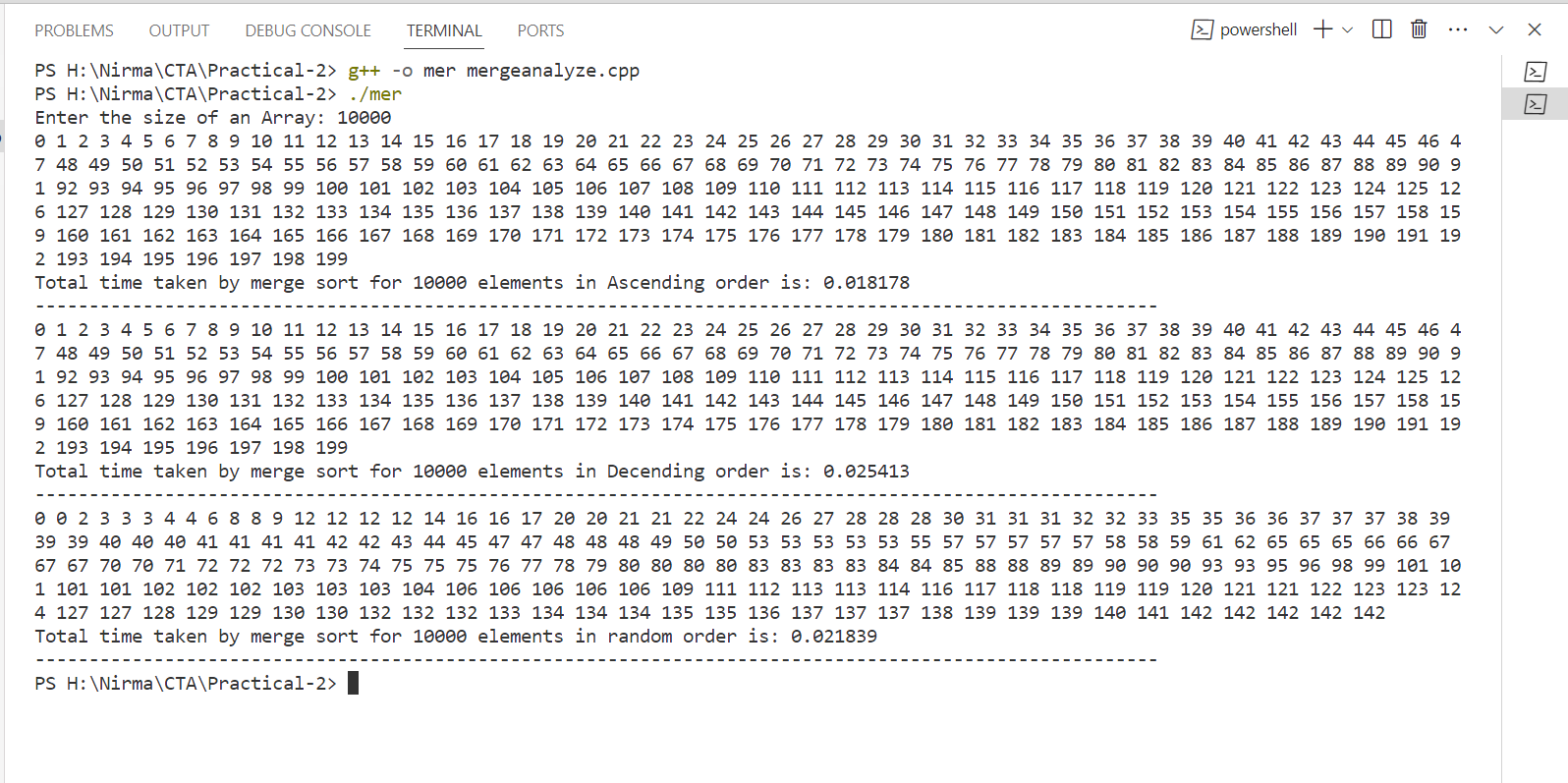
        }

    }

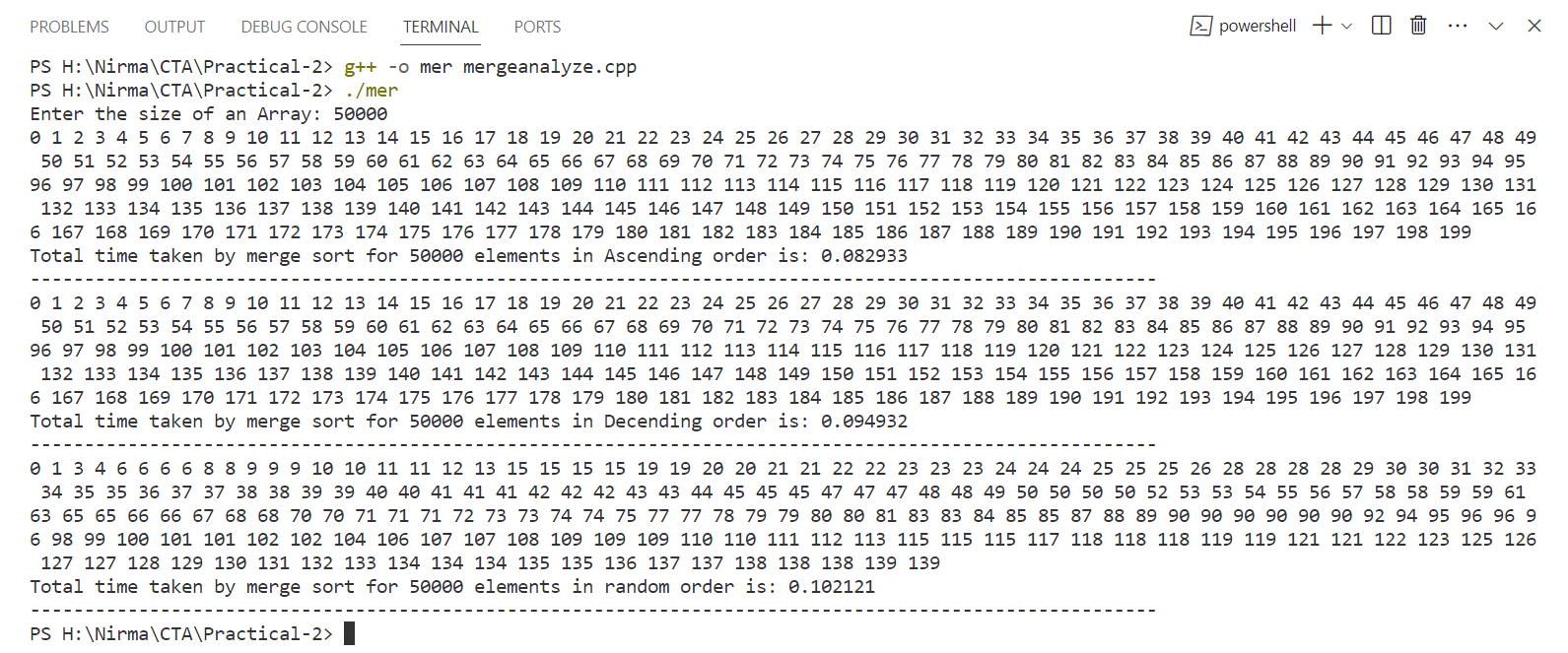
}

* **Output**

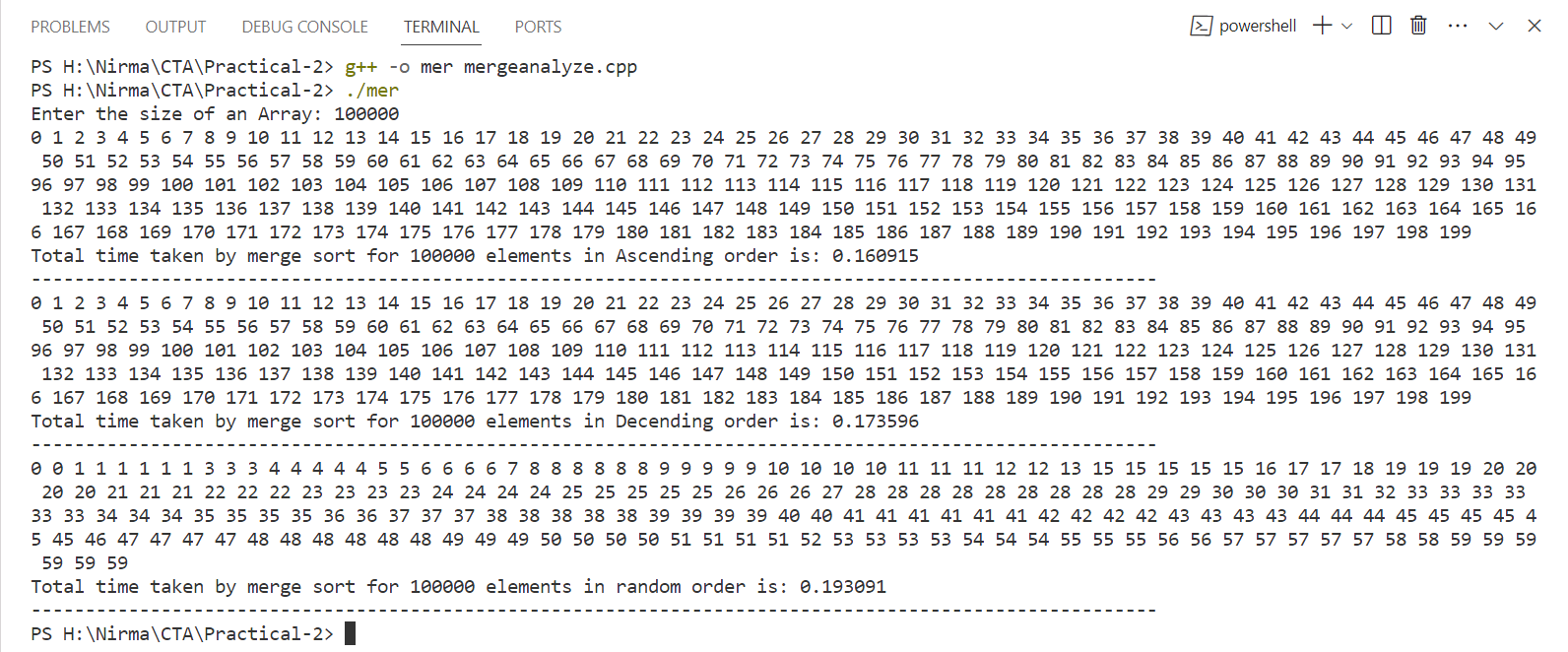
For array size = 10000



For array size = 50000



For array size = 100000



* **Graphs & Output Data**

Here are the One tables that describes the output generated by the above code for ascending, descending and random array for size 10000, 50000 & 100000.

|  |  |  |  |
| --- | --- | --- | --- |
|  | **Ascending Order** | **Descending Order** | **Random Order** |
| **10000** | **0.01817** | **0.02541** | **0.02183** |
| **50000** | **0.08293** | **0.09493** | **0.1021** |
| **100000** | **0.1609** | **0.1735** | **0.193** |

* In above graph x-axis contains the time in second, y-axis contains the order of array for 10000, 50000, and 100000 input size.
* **Analysis**

Merge sort is dividing and conquer algorithm so from my data as per the array size increase the time taken by merge sort algorithm increases in all three cases that is ascending, descending and random order array. Now for the 10000 elements array for ascending order it takes lesser time than all because the no of swaps and comparisons are less then other order because in other order the no of swaps and comparisons are more then the ascending order.

**Code for External Merge sort-**

#include <bits/stdc++.h>

using namespace std;

using namespace std::chrono;

const int CHUNK\_SIZE = 1000;

// Function to generate a random input file with n elements

void generateAscendingInput(const string &inputFile, int n)

{

    ofstream outFile(inputFile);

    if (!outFile)

    {

        cerr << "Error: Cannot open output file for writing." << endl;

        return;

    }

    for (int i = 1; i <= n; ++i)

    {

        outFile << i << endl;

    }

    outFile.close();

}

void generateDescendingInput(const string &inputFile, int n)

{

    ofstream outFile(inputFile);

    if (!outFile)

    {

        cerr << "Error: Cannot open output file for writing." << endl;

        return;

    }

    for (int i = n; i >= 1; --i)

    {

        outFile << i << endl;

    }

    outFile.close();

}

void generateRandomInput(const string &inputFile, int n)

{

    ofstream outFile(inputFile);

    if (!outFile)

    {

        cerr << "Error: Cannot open output file for writing." << endl;

        return;

    }

    srand(static\_cast<unsigned int>(time(nullptr)));

    for (int i = 0; i < n; ++i)

    {

        int value = rand() % 10000; // Adjust the range as needed

        outFile << value << endl;

    }

    outFile.close();

}

// Function to merge sorted chunks

void mergeChunks(const vector<string> &chunkFiles, const string &outputFile)

{

    vector<ifstream> chunkStreams;

    for (const auto &chunkFile : chunkFiles)

    {

        chunkStreams.emplace\_back(chunkFile);

    }

    vector<int> currentValues(chunkStreams.size());

    priority\_queue<pair<int, int>, vector<pair<int, int>>, greater<>> minHeap;

    // Initialize currentValues with the first element from each chunk

    for (int i = 0; i < chunkStreams.size(); ++i)

    {

        if (chunkStreams[i] >> currentValues[i])

        {

            minHeap.emplace(currentValues[i], i);

        }

    }

    ofstream outFile(outputFile);

    while (!minHeap.empty())

    {

        auto [value, chunkIndex] = minHeap.top();

        minHeap.pop();

        outFile << value << endl;

        if (chunkStreams[chunkIndex] >> currentValues[chunkIndex])

        {

            minHeap.emplace(currentValues[chunkIndex], chunkIndex);

        }

    }

    for (const auto &chunkFile : chunkFiles)

    {

        remove(chunkFile.c\_str()); // Clean up temporary chunk files

    }

}

// Function to perform external merge sort

double externalMergeSort(const string &inputFile, const string &outputFile, int maxElementsPerChunk)

{

    ifstream inFile(inputFile);

    if (!inFile)

    {

        cerr << "Error: Cannot open input file." << endl;

        return -1.0;

    }

    int chunkNumber = 0;

    vector<string> chunkFiles;

    auto start\_time = high\_resolution\_clock::now(); // Start measuring time

    while (!inFile.eof())

    {

        vector<int> chunkData;

        chunkData.reserve(maxElementsPerChunk);

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

        {

            int value;

            if (inFile >> value)

            {

                chunkData.push\_back(value);

            }

            else

            {

                break;

            }

        }

        sort(chunkData.begin(), chunkData.end());

        string chunkFile = "chunk\_" + to\_string(chunkNumber) + ".tmp";

        ofstream chunkOutFile(chunkFile);

        for (int value : chunkData)

        {

            chunkOutFile << value << endl;

        }

        chunkFiles.push\_back(chunkFile);

        ++chunkNumber;

    }

    inFile.close();

    // Merge sorted chunks

    mergeChunks(chunkFiles, outputFile);

    auto end\_time = high\_resolution\_clock::now(); // Stop measuring time

    auto total\_time = duration<double>(end\_time - start\_time);

    return total\_time.count();

}

int main()

{

    int maxElementsPerChunk = CHUNK\_SIZE;

    int totalElements;

    cout << "Number of elements in the input file : ";

    cin >> totalElements;

    cout << "Choose the input order:" << endl;

    cout << "1. Ascending Order" << endl;

    cout << "2. Descending Order" << endl;

    cout << "3. Random Order" << endl;

    int choice;

    cout << "Enter your choice (1/2/3): ";

    cin >> choice;

    string inputFile, outputFile;

    switch (choice)

    {

    case 1:

        inputFile = "input\_ascending.txt";

        outputFile = "output\_ascending.txt";

        generateAscendingInput(inputFile, totalElements);

        break;

    case 2:

        inputFile = "input\_descending.txt";

        outputFile = "output\_descending.txt";

        generateDescendingInput(inputFile, totalElements);

        break;

    case 3:

        inputFile = "input\_random.txt";

        outputFile = "output\_random.txt";

        generateRandomInput(inputFile, totalElements);

        break;

    default:

        cout << "Invalid choice. Exiting." << endl;

        return 1;

    }

    double elapsedTime = externalMergeSort(inputFile, outputFile, maxElementsPerChunk);

    cout << endl

         << "Total Time taken by External Merge sort for " << totalElements << " elements is: " << elapsedTime << " seconds" << endl;

    cout << endl

         << "External merge sort complete. Sorted data is stored in " << outputFile << endl;

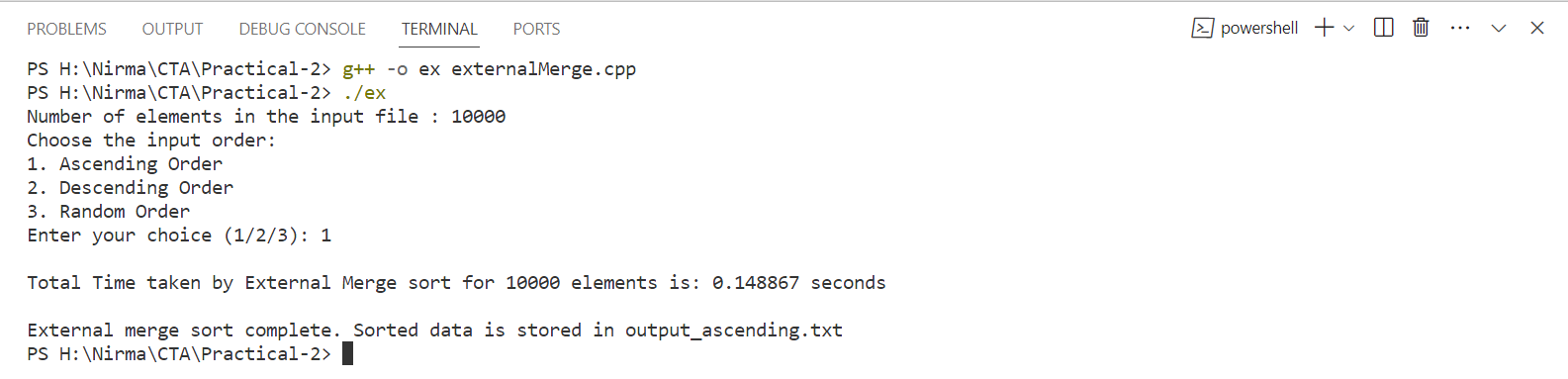
    return 0;

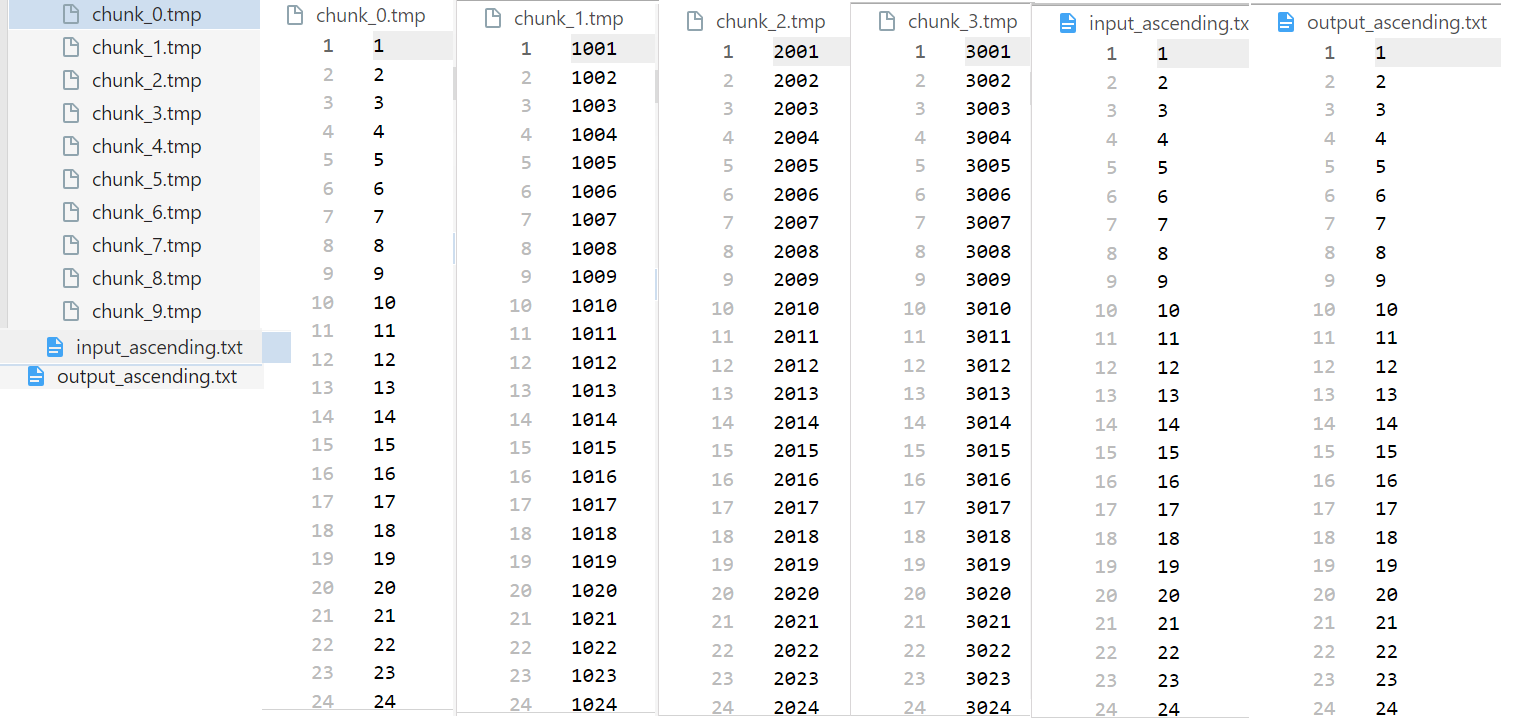
}

* **Output**

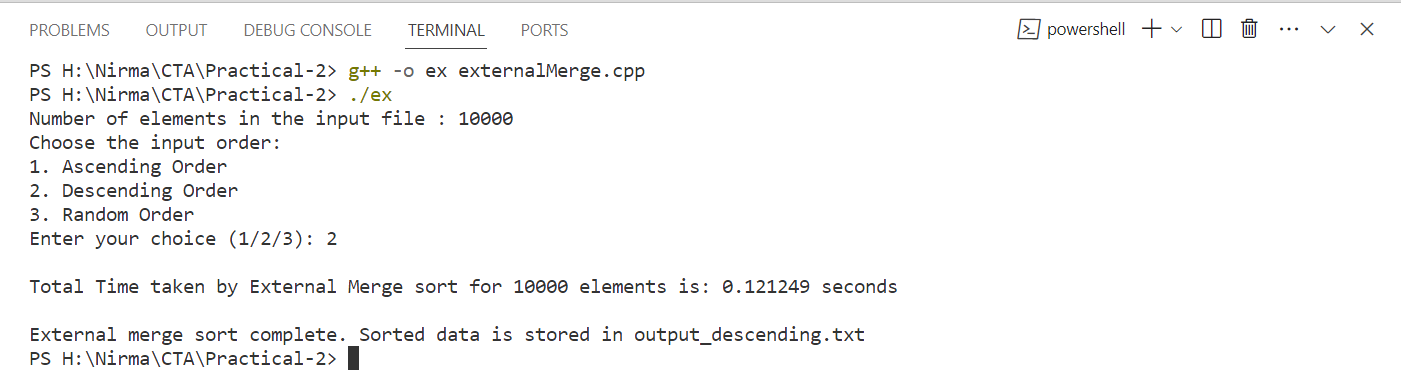
For array size = 10000

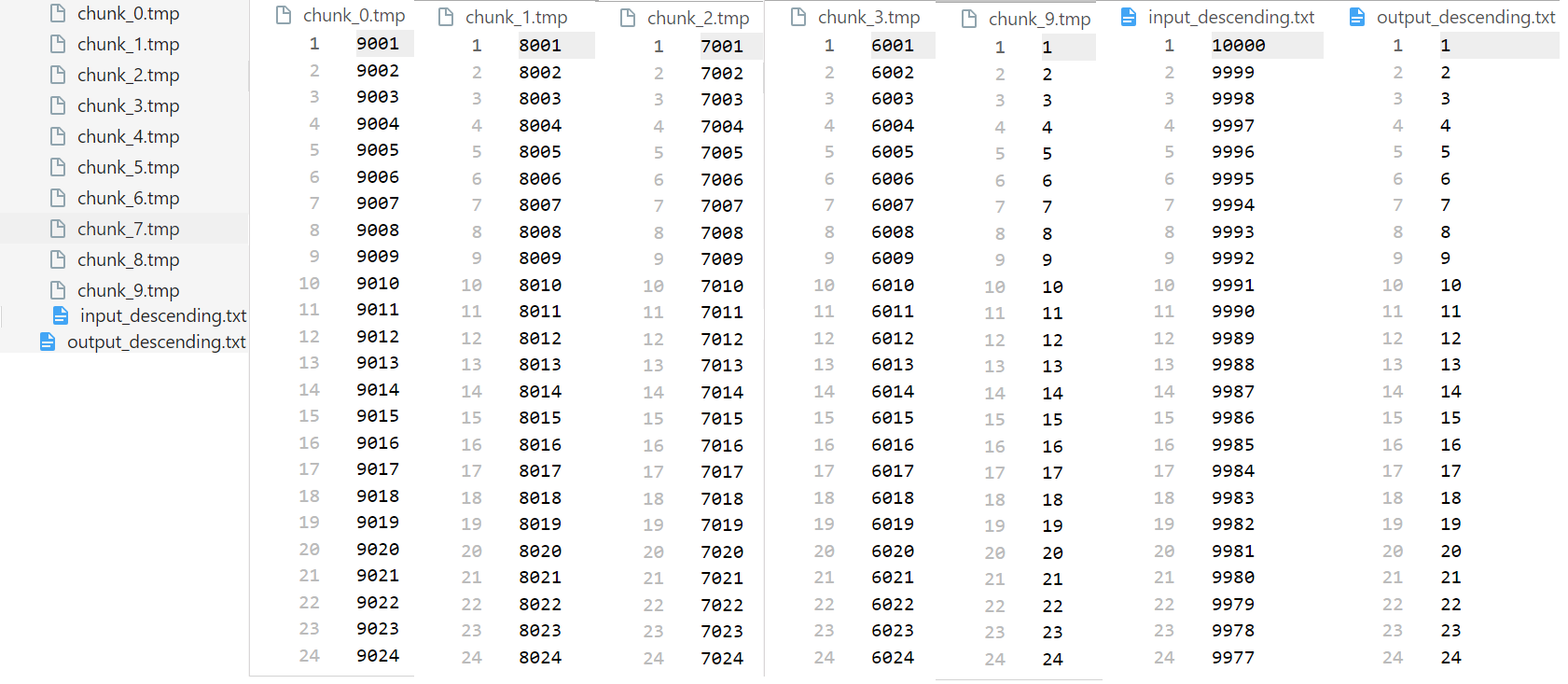
**Ascending Order**



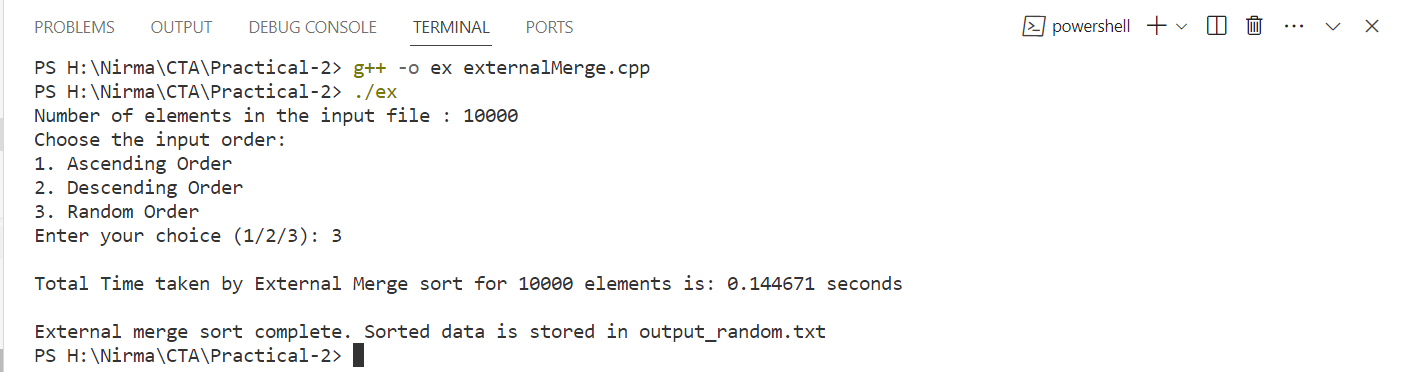


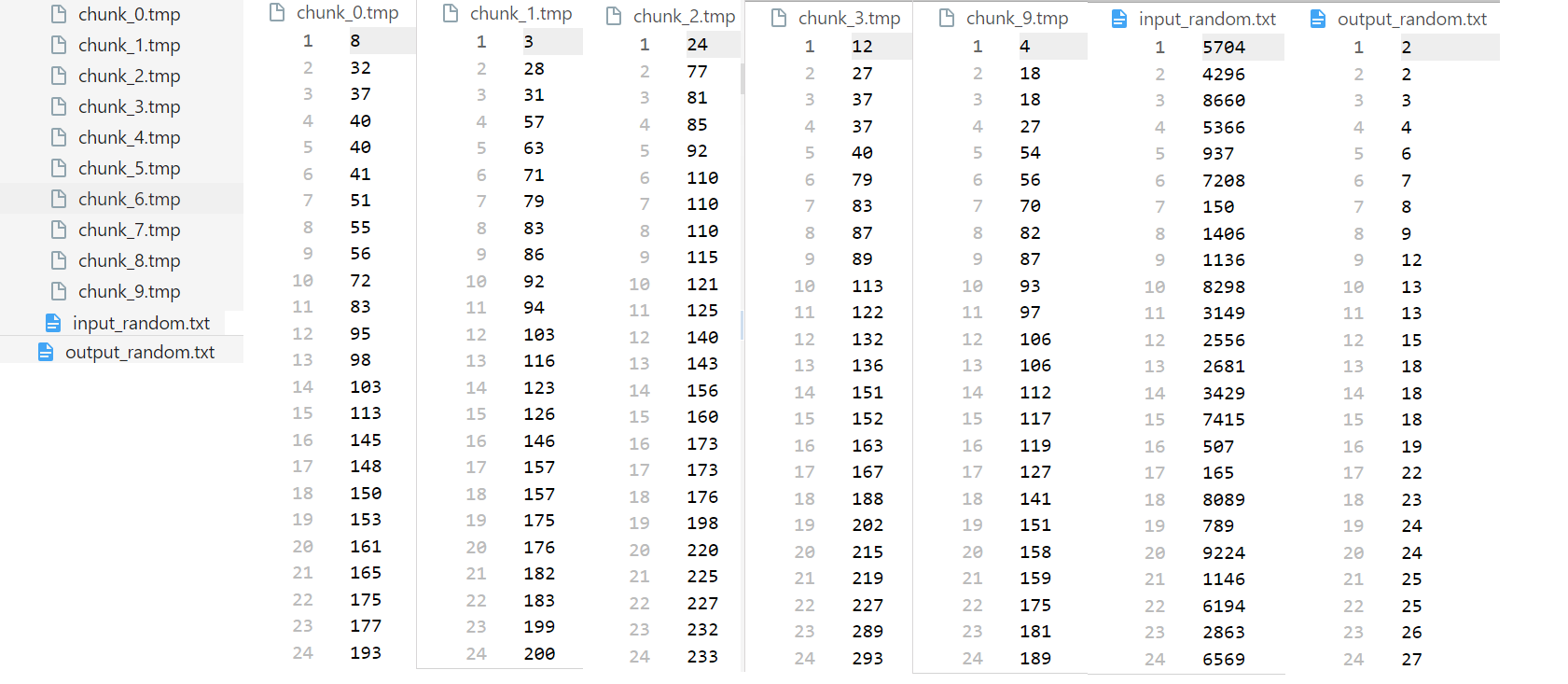
**Descending Order**





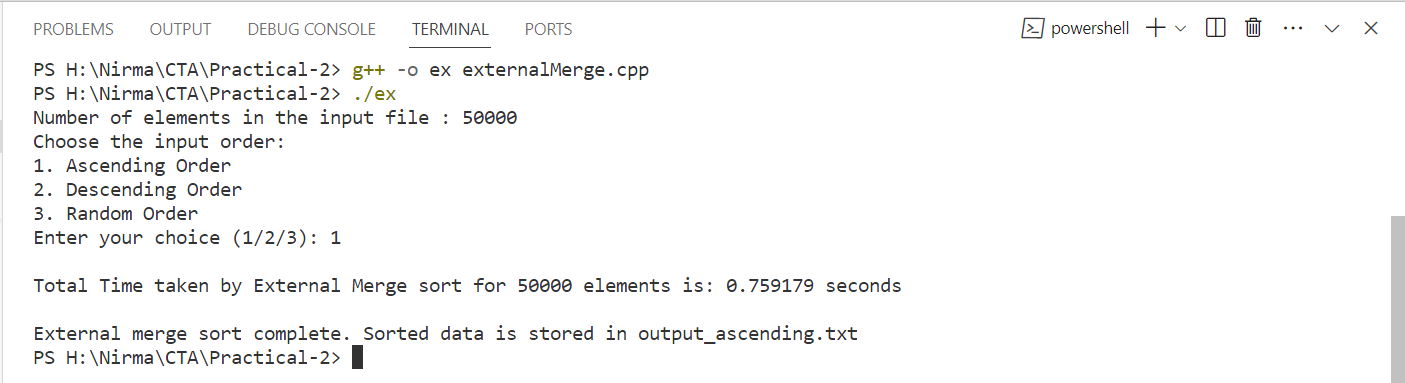
**Random Order**

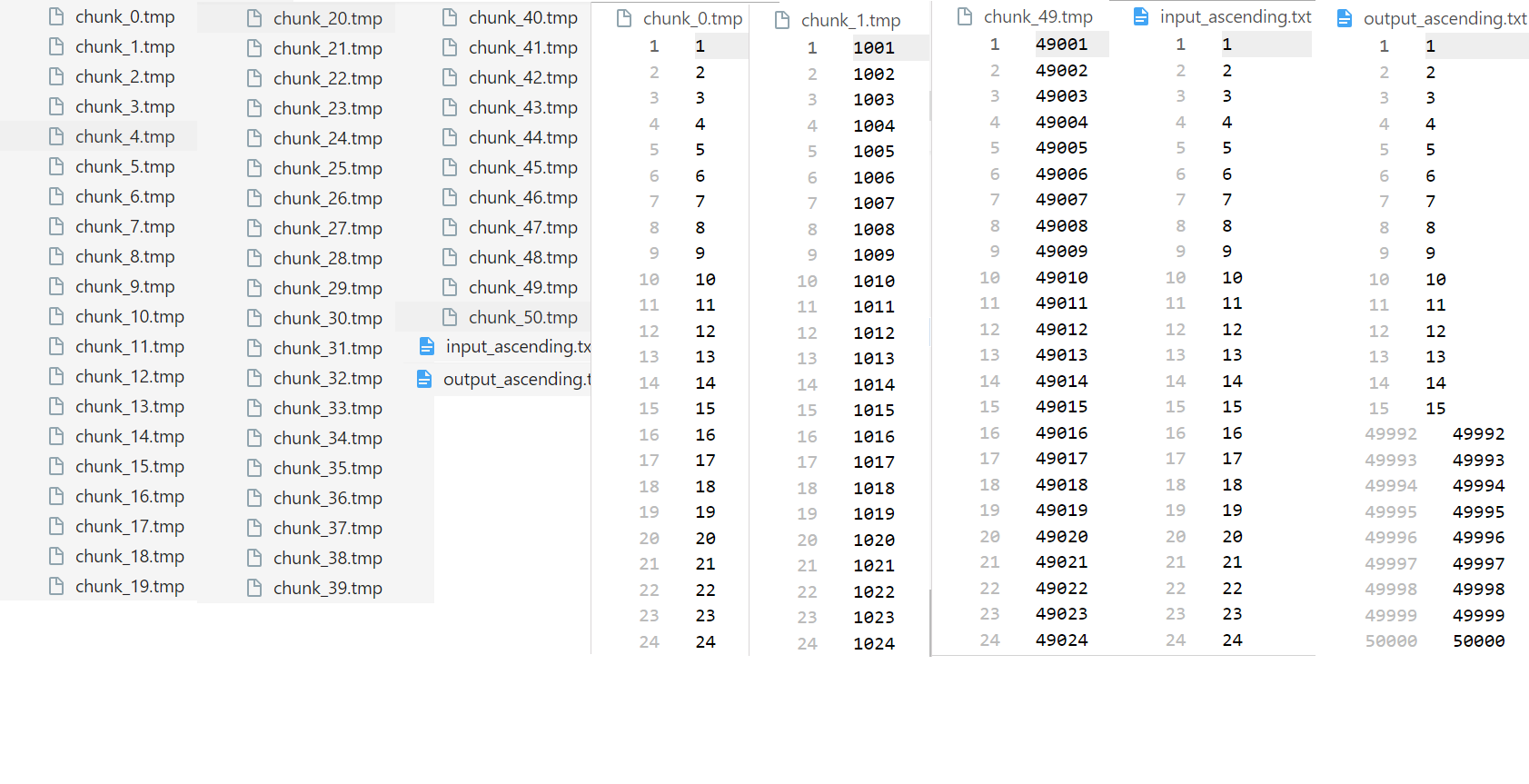
****

****

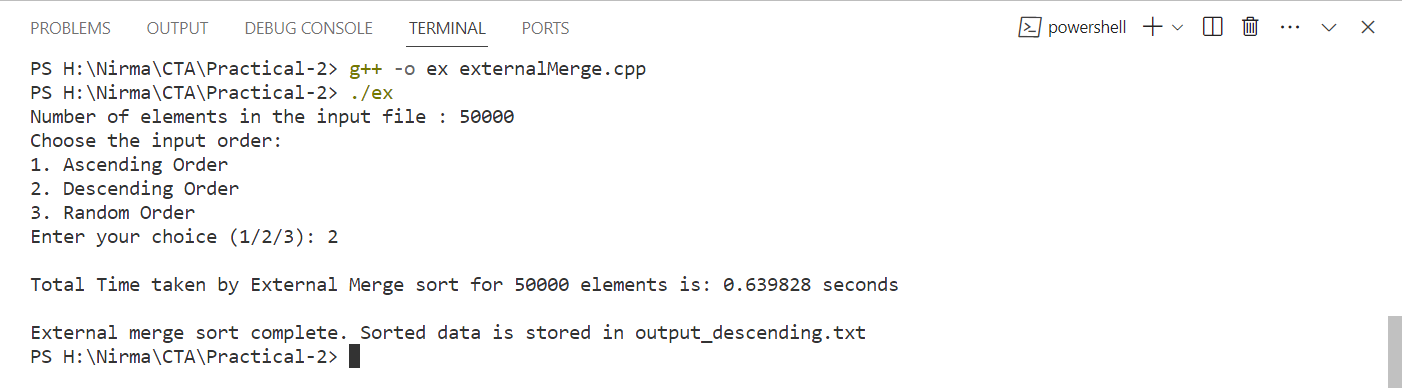
For array size = 50000

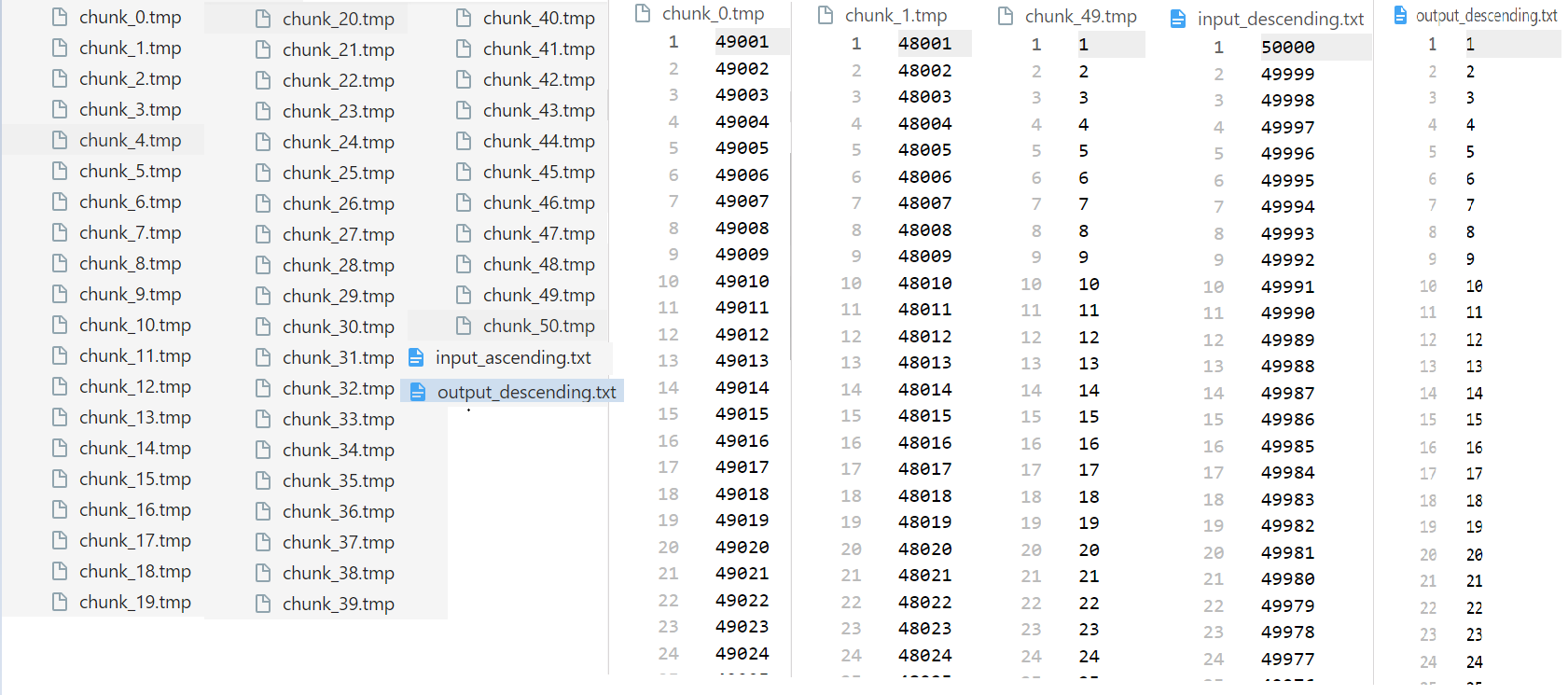
**Ascending Order**



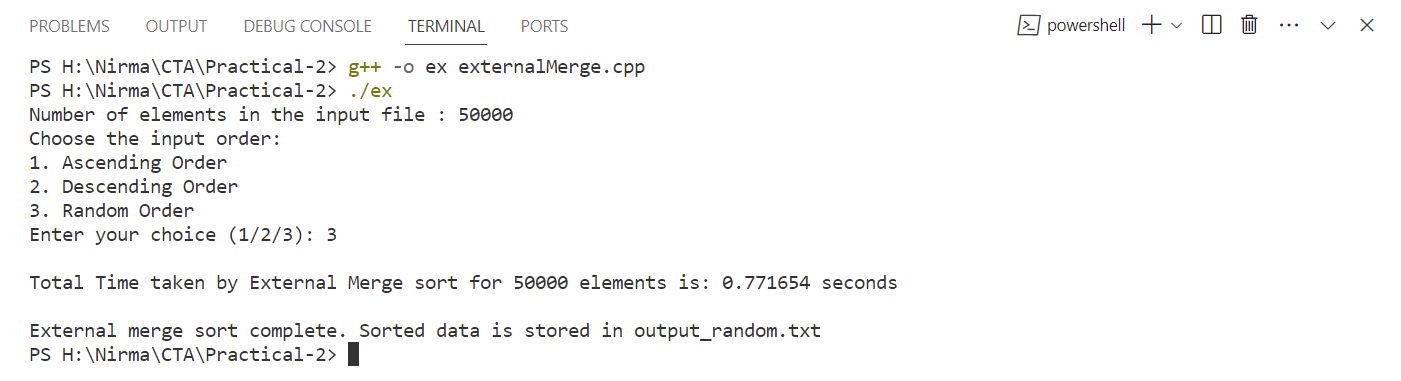


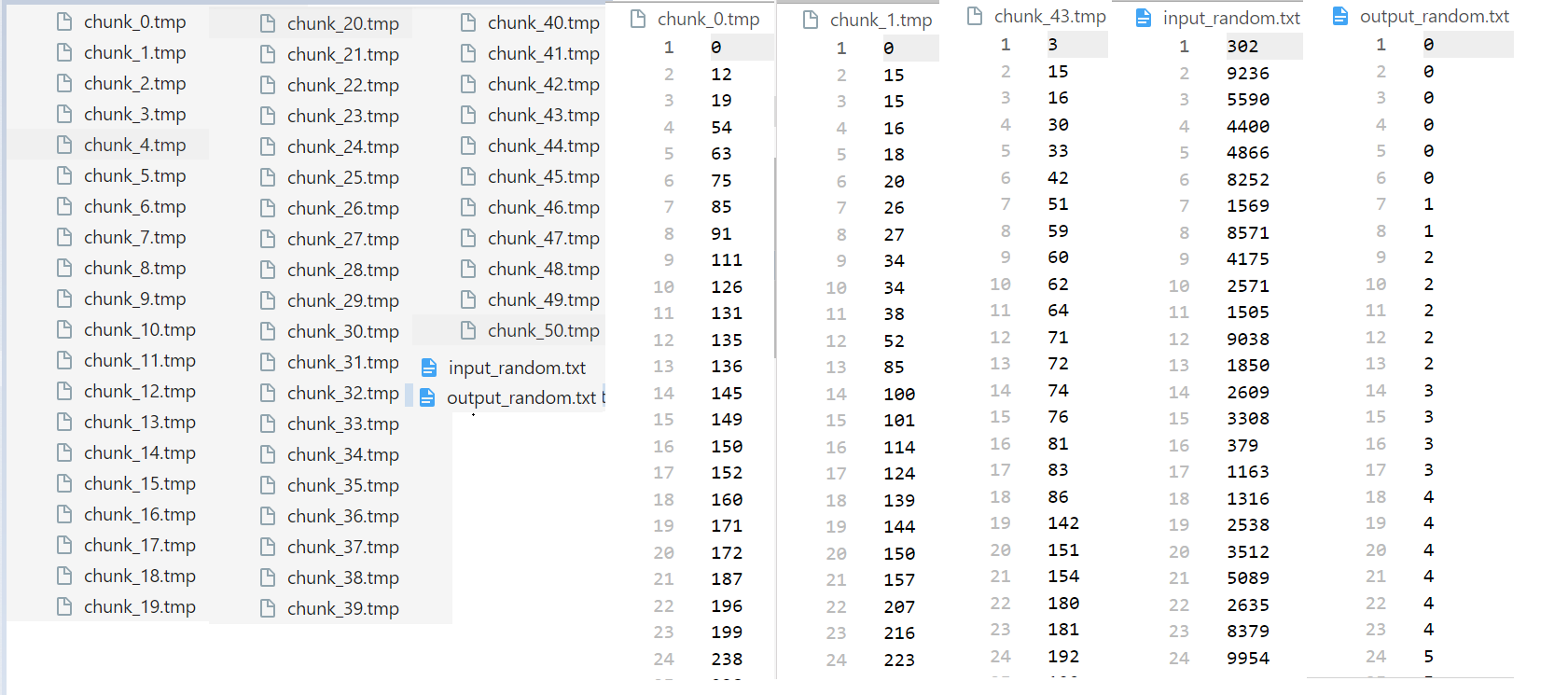
**Descending Order**





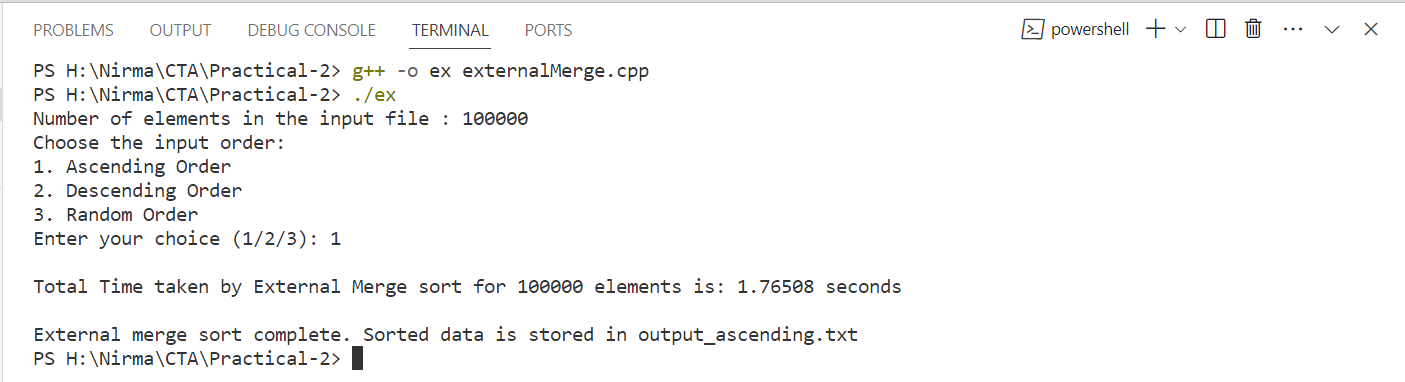
**Random Order**

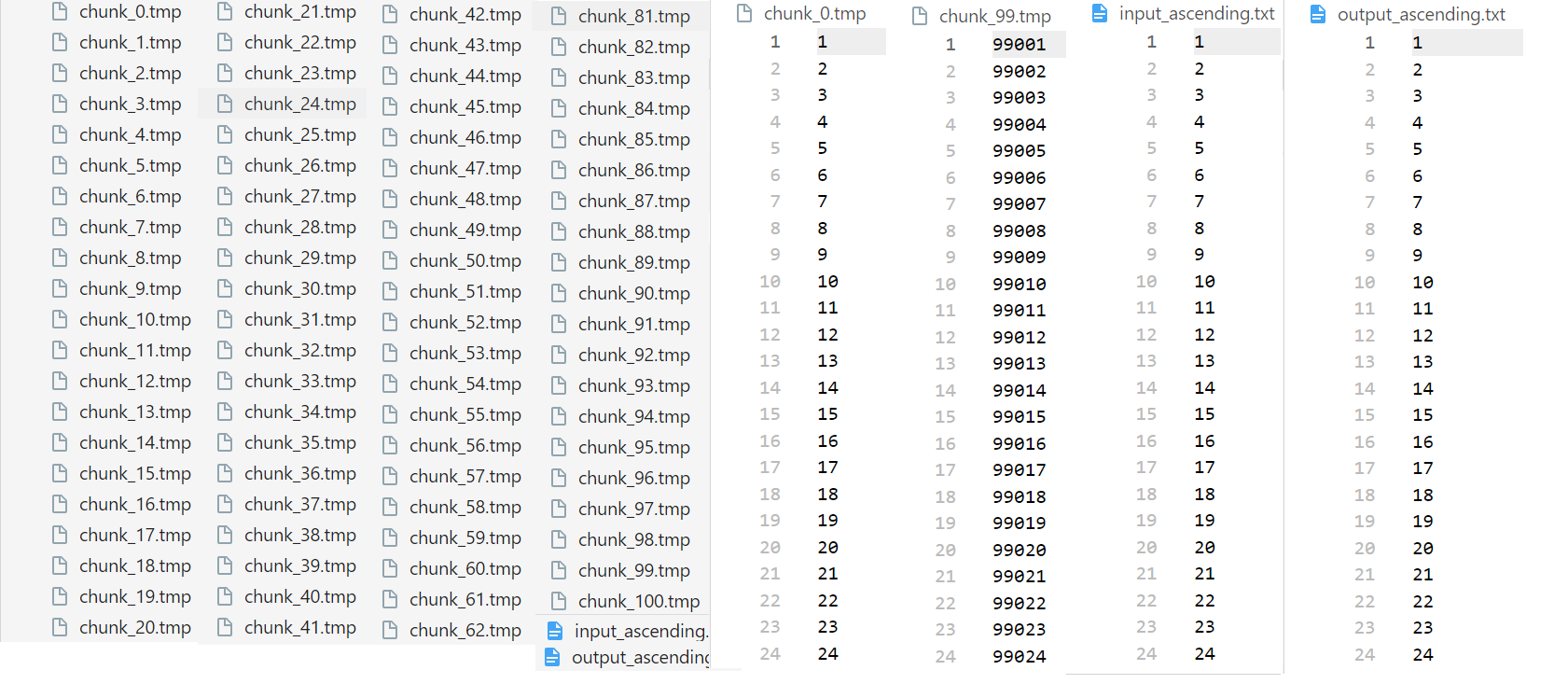
****

****

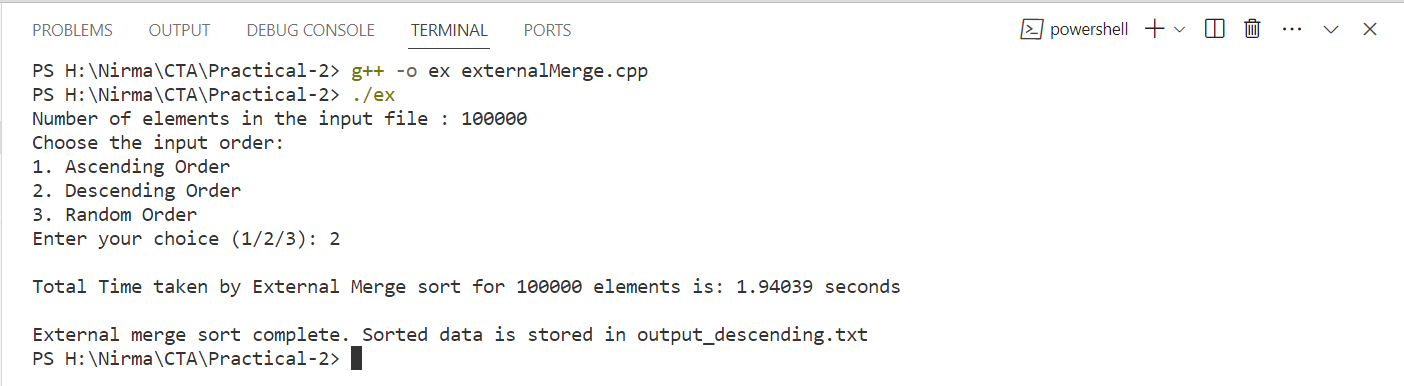
For array size = 100000

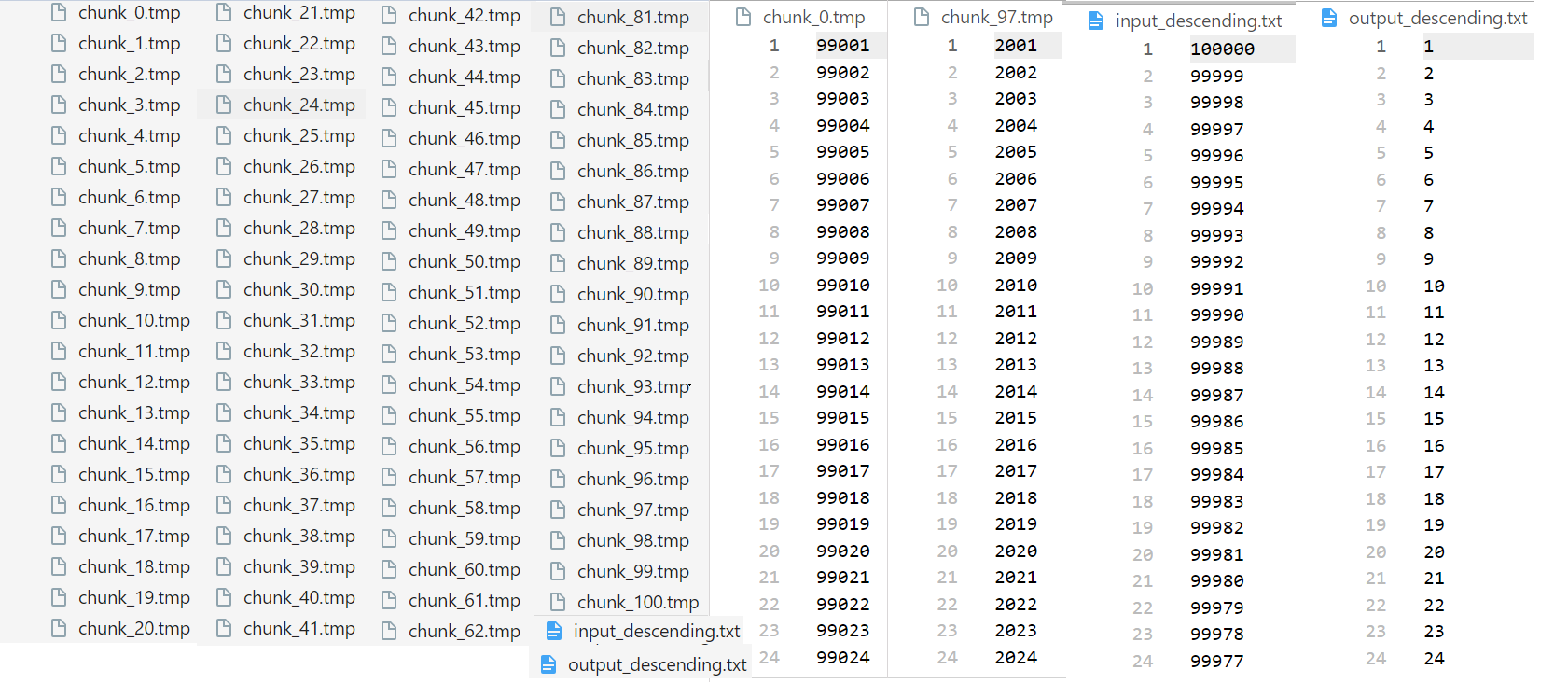
**Ascending Order**



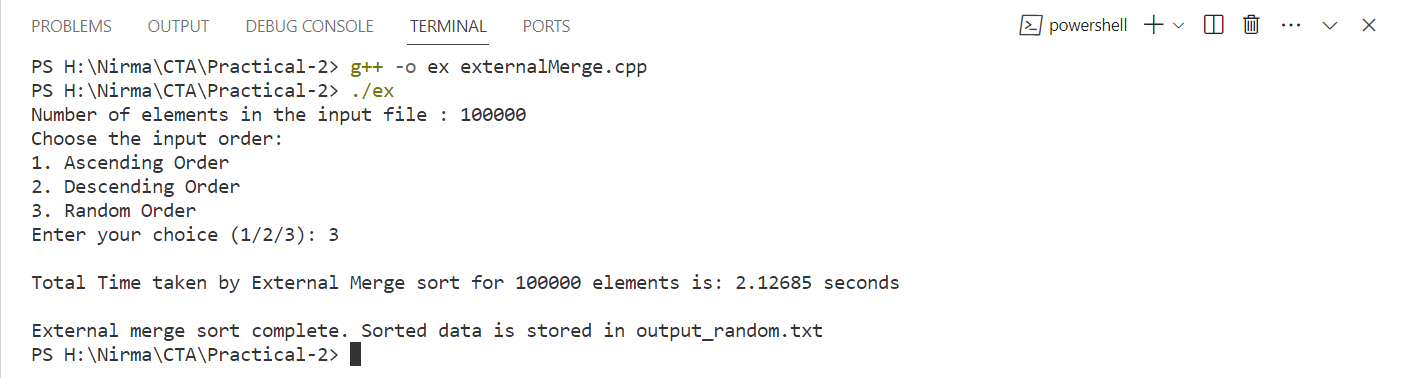


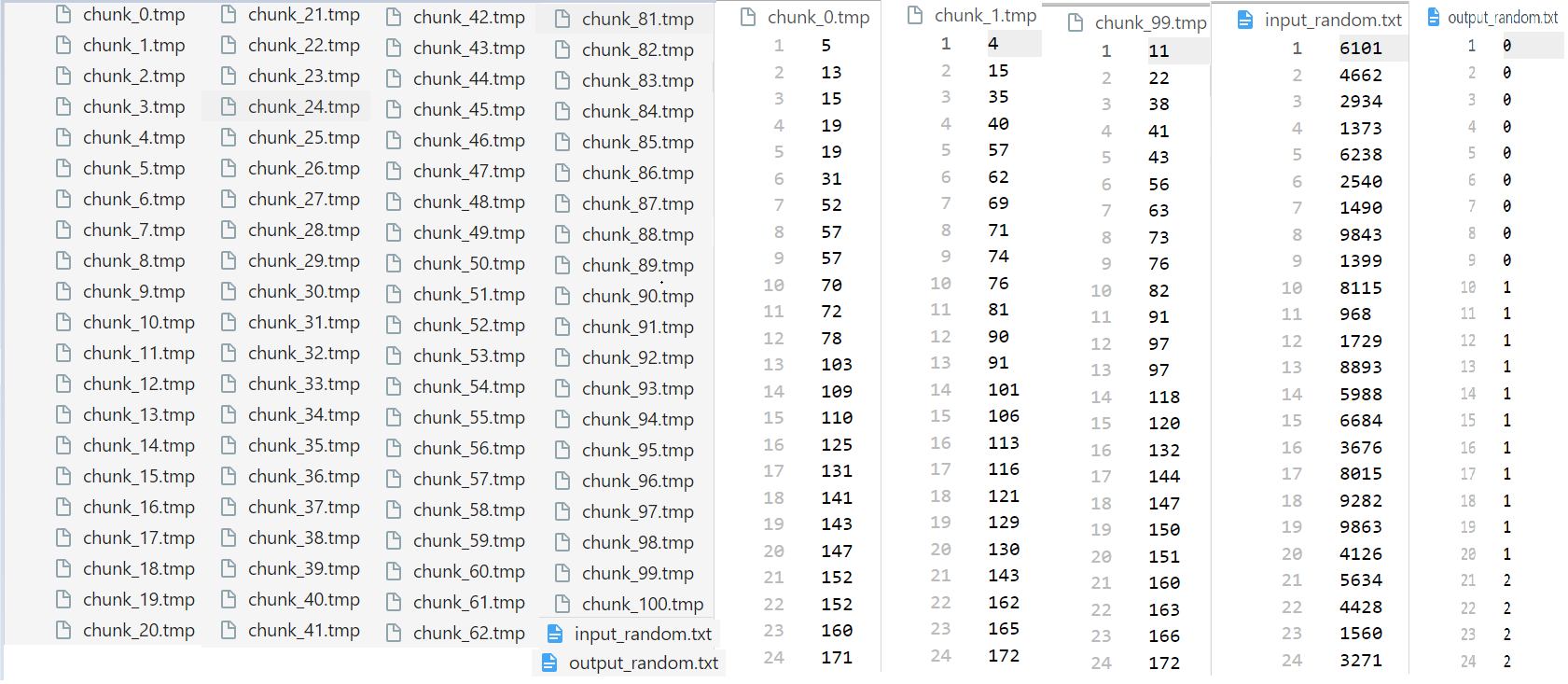
**Descending Order**

****

****

**Random Order**

****

****

* **Graphs & Output Data**

Here are the One tables that describes the output generated by the above code for ascending, descending and random array for size 10000, 50000 & 100000.

|  |  |  |  |
| --- | --- | --- | --- |
|  | **Ascending Order** | **Descending Order** | **Random Order** |
| **10000** | **0.1488** | **0.1212** | **0.1446** |
| **50000** | **0.7591** | **0.6398** | **0.7716** |
| **100000** | **1.765** | **1.9403** | **2.1268** |

* **Analysis**

From above code the conclusion is as per the order the i/o operations are increases with the array size and it also depends on the order of an array if array is sorted so the no of i/o operations are less then the descending and random order data.

* **Comparison of Merge Sort and External Merge Sort**
* Conclusion

Merge sort is used for the in-memory data which can easily fit into the RAM of our Hardware device wherein there are large dataset which can not fit into the main memory so in such cases we use the external merge sort where the data is divided into chunks and sorted that into memory and then merge them. In summary, if you have smaller dataset which can easily fit into main memory then merge sort is efficient but if you have larger data set then external merge sort is efficient.

External merge sort is slower than in memory algorithm like merge sort because read and write operations to external storage to access the elements from file and it is also dependent on the hardware specifications.