Індивідуальне завдання 3

**Завдання:**

**1**. Оберіть і програмно реалізуйте один з методів вдосконаленого

сортування з обчисленням кількості порівнянь та переміщень.

**2.** Програмно реалізуйте один з методів зовнішнього сортування з

обчисленням кількості порівнянь та переміщень.

**3.** Протестуйте ці реалізацію в найкращому, найгіршому та

середньому випадках для даних розміром 100, 1000, 10000, 100000,

1000000 елементів. Збережіть отримані кількості порівнянь та

переміщень у підготовленому середовищі та проаналізуйте отримані

графіки. Порівняйте ці методи із вже реалізованими. Зробіть свої

висновки та запишіть їх. Перевірте, чи відображують отримані вами

результати загальні формули трудомісткості та очікувану поведінку

методу для різних початкових послідовностей.

**Генерація випадкового числа**

Оголошую статична змінна seed для збереження початкового значення генератора випадкових чисел. Вона зберігає значення між викликами функції.

**seed = (seed \* 1664525 + 1013904223)% 4294967296**; Це формула для генерації псевдовипадкового чисел

**return (seed % (maxVal - minVal + 1)) + minVal;** Виводить випадкове число у межах заданого діапазону

**Зовнішне**

**Функція для злиття масивів у merge sort**

Функція зливає два відсортованих підмасивів **arr[left..mid]** і **arr[mid+1..right]** в один масив

**comparisons** - лічильник порівнянь

**swaps** - лічильник переміщень

**n1** - розмір лівого підмасиву

**n2** - розмір правого підмасиву

Створюю тимчасові масиви для зберігання елементів з arr

    for (int i = 0; i < n1; i++) {

        leftArr[i] = arr[left + i];

    }

    for (int i = 0; i < n2; i++) {

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

    }

Копіює елементи з arr у leftArr та rightArr

i та j — індекси для тимчасових масивів, k — індекс для вставки у вихідний масив

Порівнюю поточні елементи тимчасових масивів, менший з них вставляю в arr Збільшуються лічильники порівнянь та переміщень

    while (i < n1 && j < n2) {

        comparisons++;

        if (leftArr[i] <= rightArr[j]) {

            arr[k] = leftArr[i];

            i++;

        } else {

            arr[k] = rightArr[j];

            j++;

        }

        k++;

        swaps++;

    }

Копируются оставшиеся элементы из leftArr та rightArr

**Рекурсивний алгоритм для mergeSort**

void mergeSort(int arr[], int left, int right, int &comparisons, int &swaps) {

    if (left < right) {

        int mid = left + (right - left) / 2;

        mergeSort(arr, left, mid, comparisons, swaps);

        mergeSort(arr, mid + 1, right, comparisons, swaps);

        merge(arr, left, mid, right, comparisons, swaps);

    }

}

if (left <right) перевіряє, чи можна розділити масив.

int mid = left + (right - left) / 2; Обчислює середину масиву.

Рекурсивно сортуються лівий та правий підмасиви, потім вони поєднуються за допомогою функції merg

**Генерация массива и запись в файл**

створюється файл filename.

Перевірка успішності відкриття файлу.

    vector<int> randomArray(size);

    vector<int> sortedArray(size);

    vector<int> reversedArray(size);

Створюю масиви для зберігання випадкового, відсортованого та зворотного масиву

Заповнюю файл рандомними числами

    for (int i = 0; i < size; i++) {

        randomArray[i] = generateRandom(minVal, maxVal);

    }

Копіюється і сортується масив завантаженогоArray.

    for (int i = 0; i < size; i++) {

        sortedArray[i] = randomArray[i];

    }

    for (int i = 0; i < size - 1; i++) {

        for (int j = 0; j < size - 1 - i; j++) {

            if (sortedArray[j] > sortedArray[j + 1]) {

                swap(sortedArray[j], sortedArray[j + 1]);

            }

        }

    }

Створюється масив, що перетворюється наArray, який є зворотним відсортованим

    for (int i = 0; i < size; i++) {

        reversedArray[i] = sortedArray[size - 1 - i];

    }

    for (const int &val : randomArray) {

        outFile << val << " ";

    }

    outFile << "\n";

    for (const int &val : sortedArray) {

        outFile << val << " ";

    }

    outFile << "\n";

    for (const int &val : reversedArray) {

        outFile << val << " ";

    }

    outFile << "\n";

    outFile.close();

}

Запис цих масивів у файл

**Читання масиву з файлу**

    string line;

    int currentLine = 0;

    while (getline(inFile, line)) {

        if (++currentLine == lineNumber) {

            istringstream iss(line);

            int value;

            while (iss >> value) {

                arr.push\_back(value);

            }

            break;

        }

    }

    inFile.close();

}

Зчитує потрібний рядок із файлу. Рядок перетворюється на числа, які додаються в масив.

**Запись результатов в текстовый файл**

void writeResultsToFile(const string &filename, int avgComparisons, int avgSwaps, int bestComparisons, int bestSwaps, int worstComparisons, int worstSwaps) {

    ofstream outFile(filename);

    if (!outFile) {

        cout << "Error: Cannot open file " << filename << endl;

        exit(1);

    }

    outFile << "Average case (comparisons, swaps): " << avgComparisons << ", " << avgSwaps << "\n";

    outFile << "Best case (comparisons, swaps): " << bestComparisons << ", " << bestSwaps << "\n";

    outFile << "Worst case (comparisons, swaps): " << worstComparisons << ", " << worstSwaps << "\n";

    outFile.close();

}

Зберігаються результати порівняння та переміщень для всіх випадків

**Запись результатов в CSV**

void writeResultsToCSV(const string &csvFilename, int avgComparisons, int avgSwaps, int bestComparisons, int bestSwaps, int worstComparisons, int worstSwaps) {

    ofstream csvFile(csvFilename);

    if (!csvFile) {

        cout << "Error: Cannot open CSV file " << csvFilename << endl;

        exit(1);

    }

    csvFile << "Case,Comparisons,Swaps\n";

    csvFile << "Average," << avgComparisons << "," << avgSwaps << "\n";

    csvFile << "Best," << bestComparisons << "," << bestSwaps << "\n";

    csvFile << "Worst," << worstComparisons << "," << worstSwaps << "\n";

    csvFile.close();

}

Відкривається файл CSV

Записуються дані у форматі Case, Comparisons, Swaps.

**Main**

Імена файлів.

Генерація масиву та запис у файл.

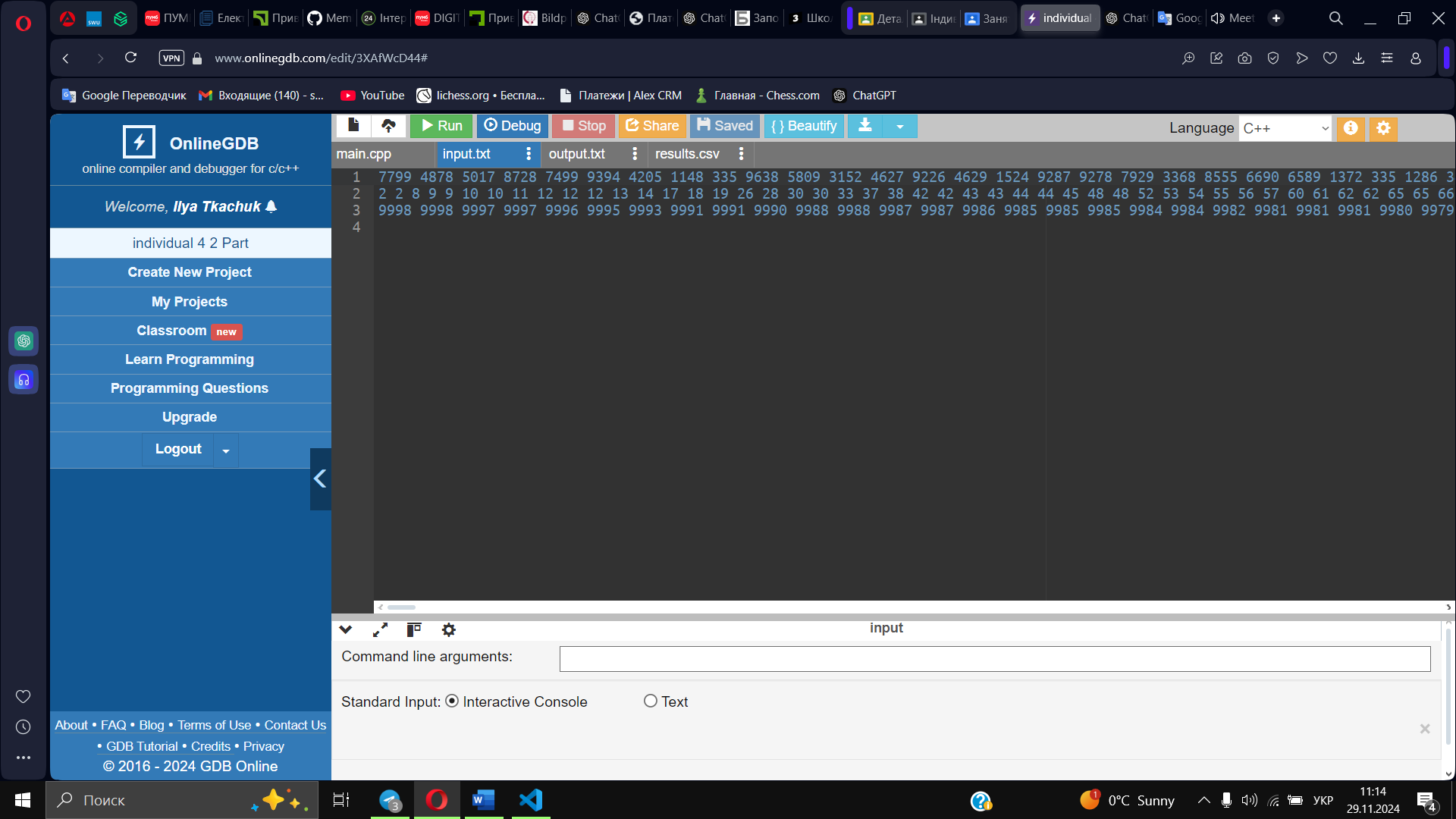
Читання масиву із файлу.

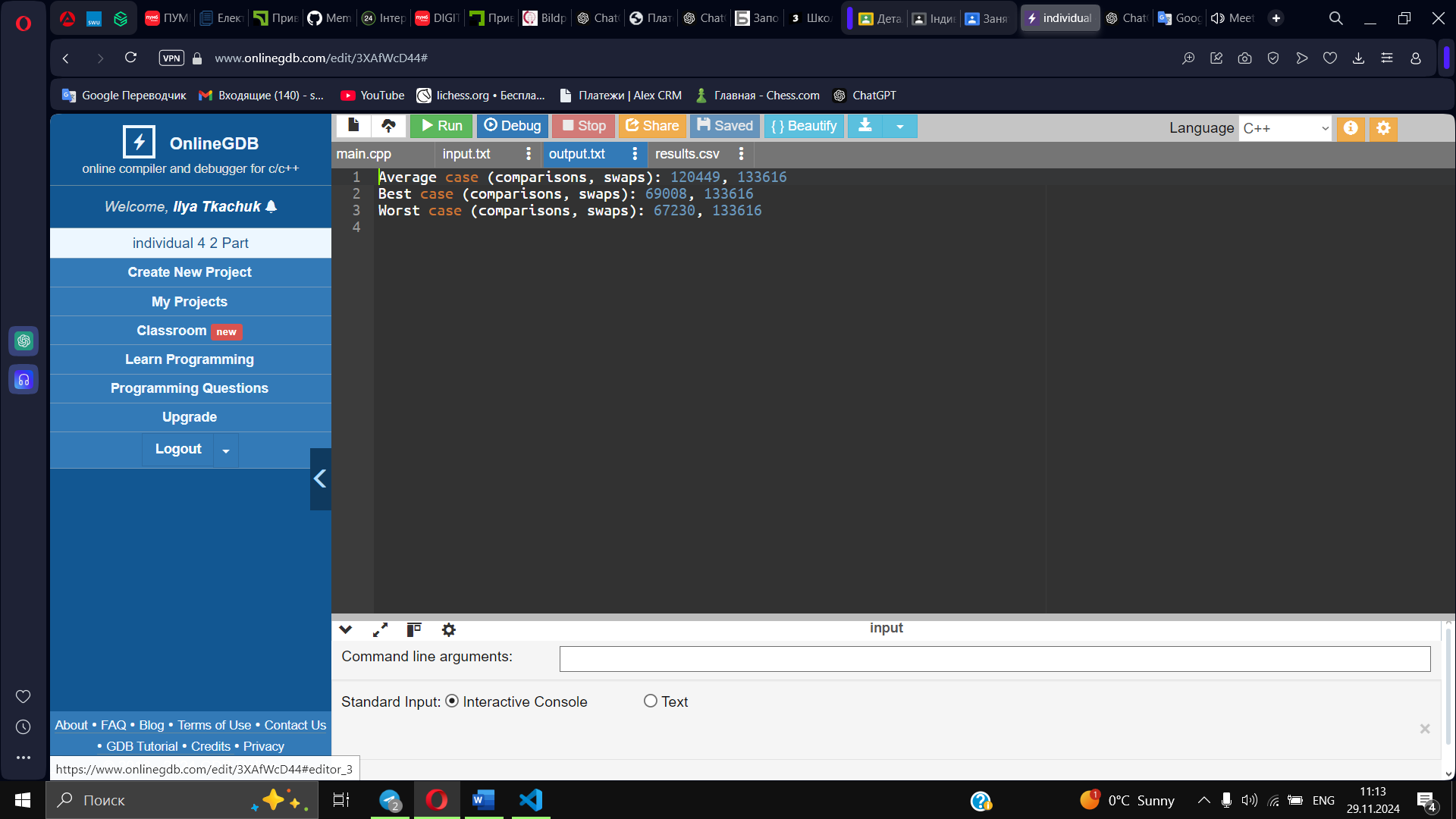
Сортування всіх випадків.

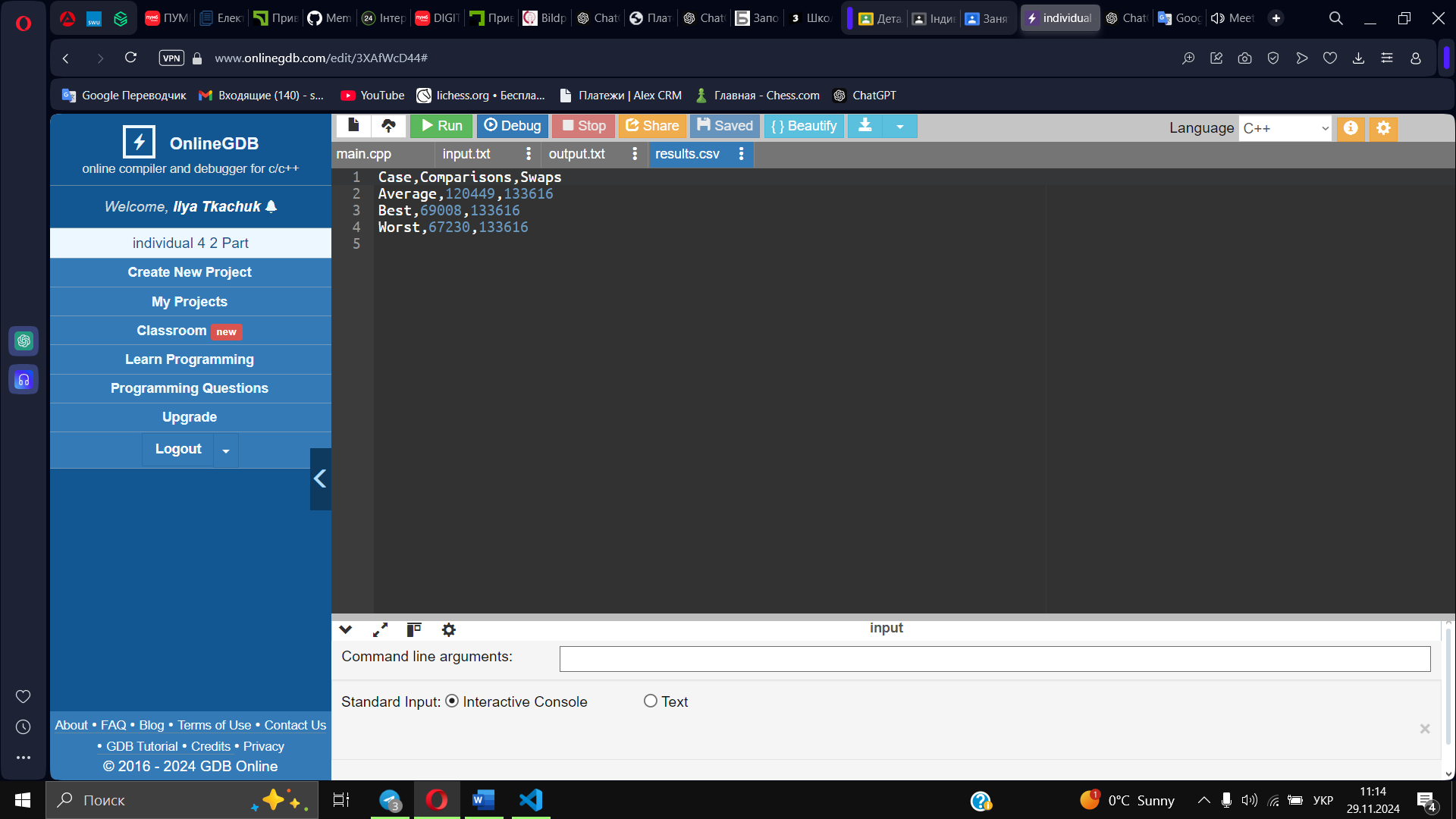
Записує результати у файли.

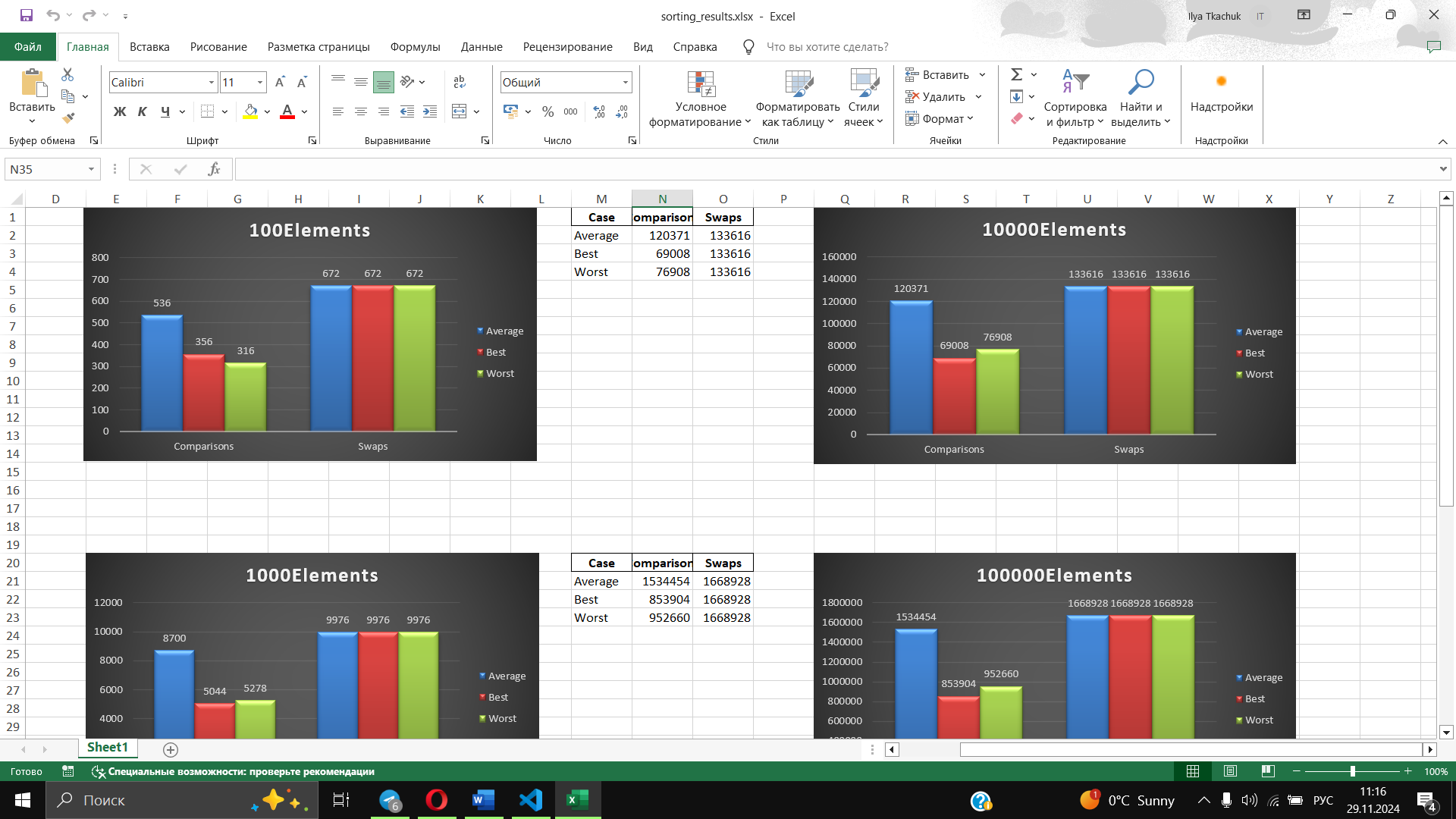
Повідомлення про успішне завершення.

**Файли при розмірі массива 10000**

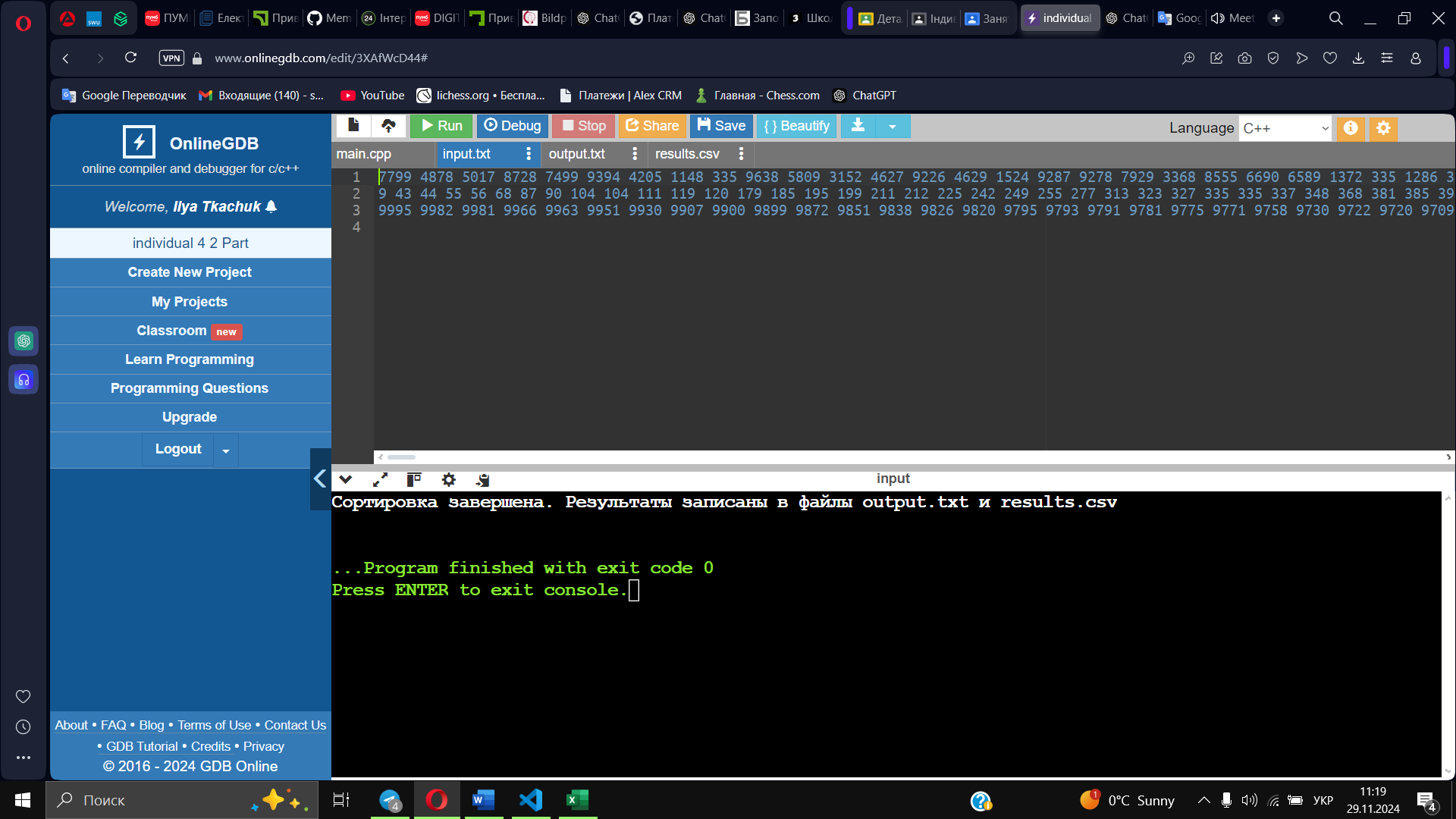


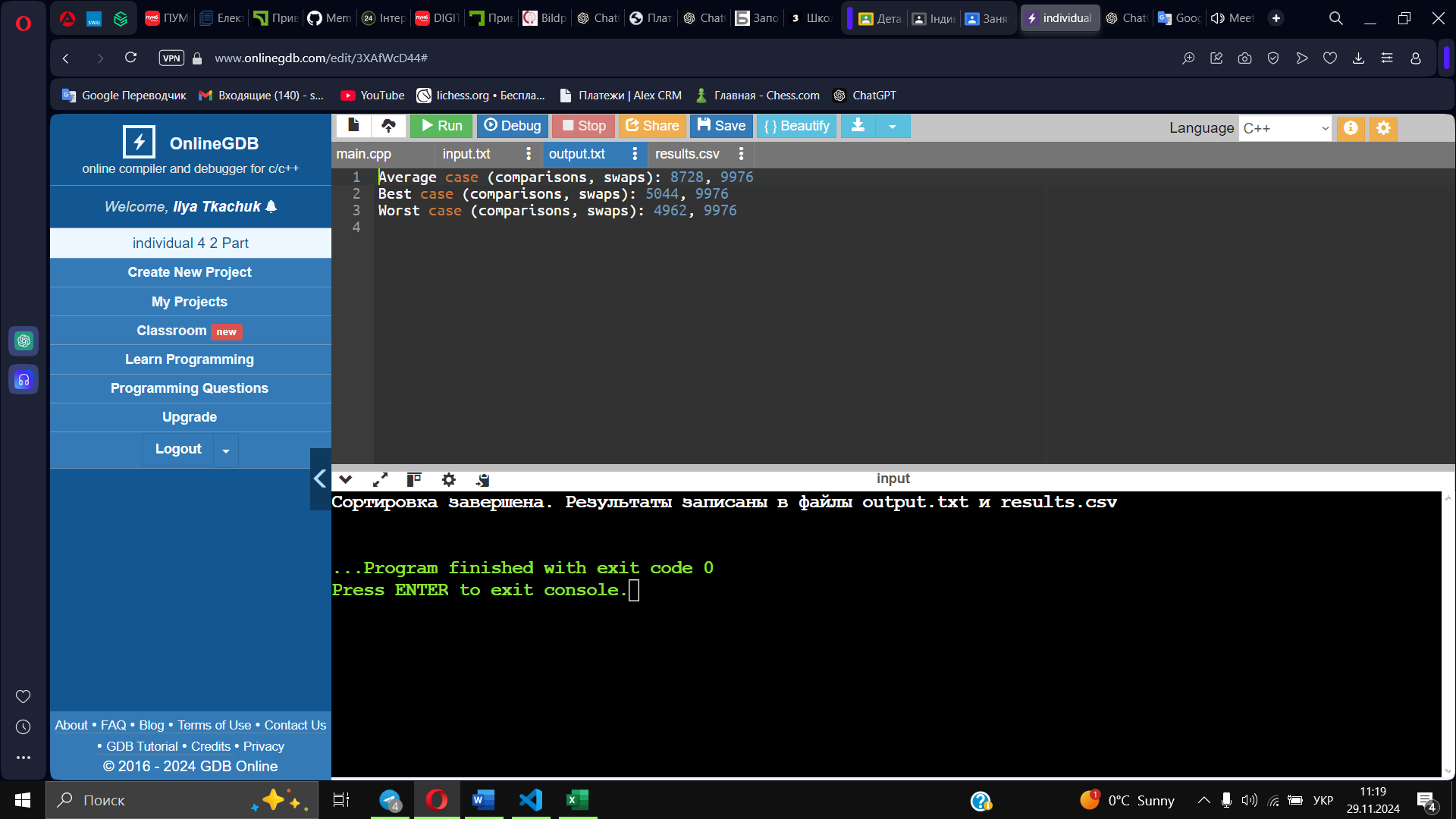


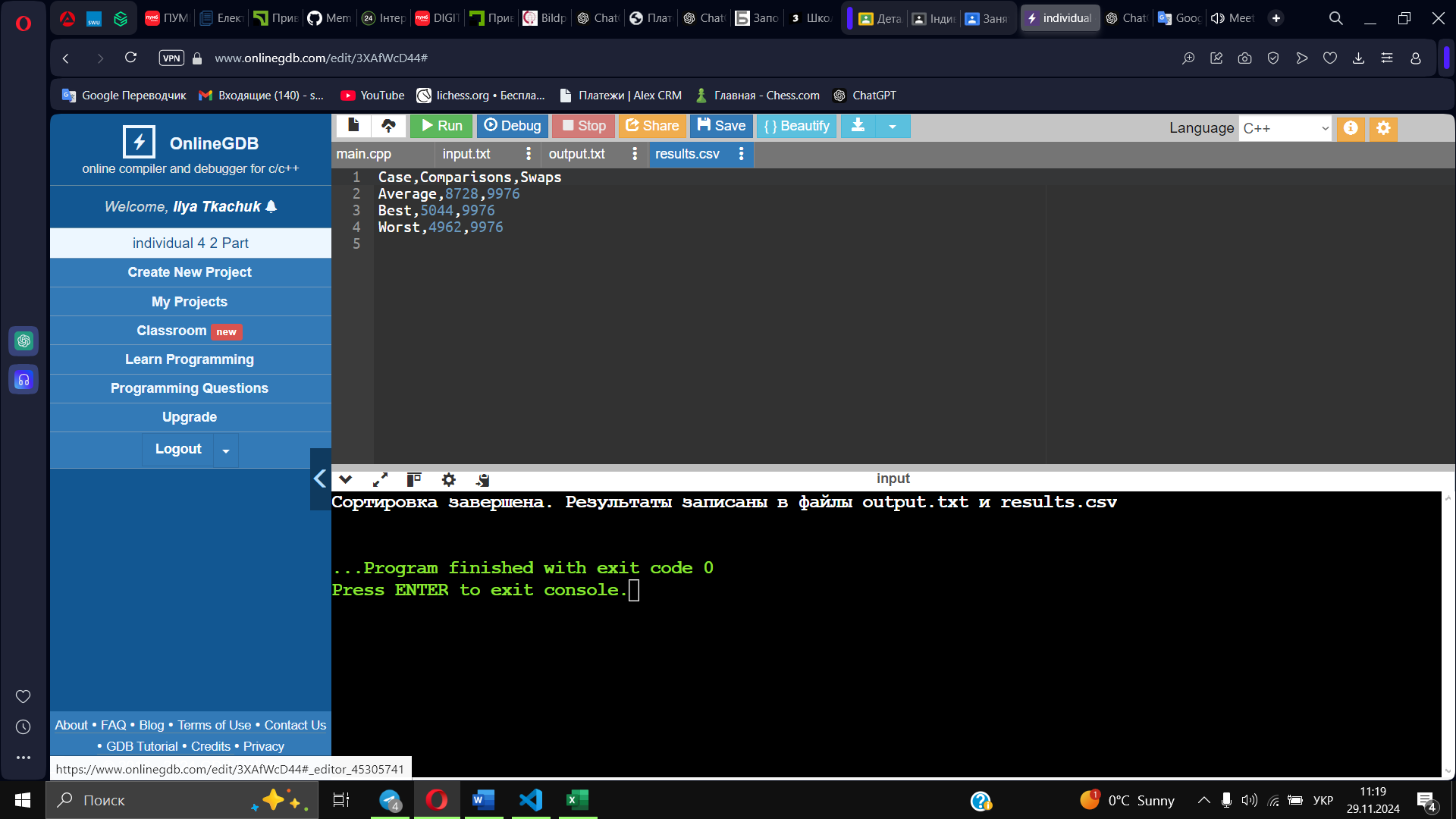




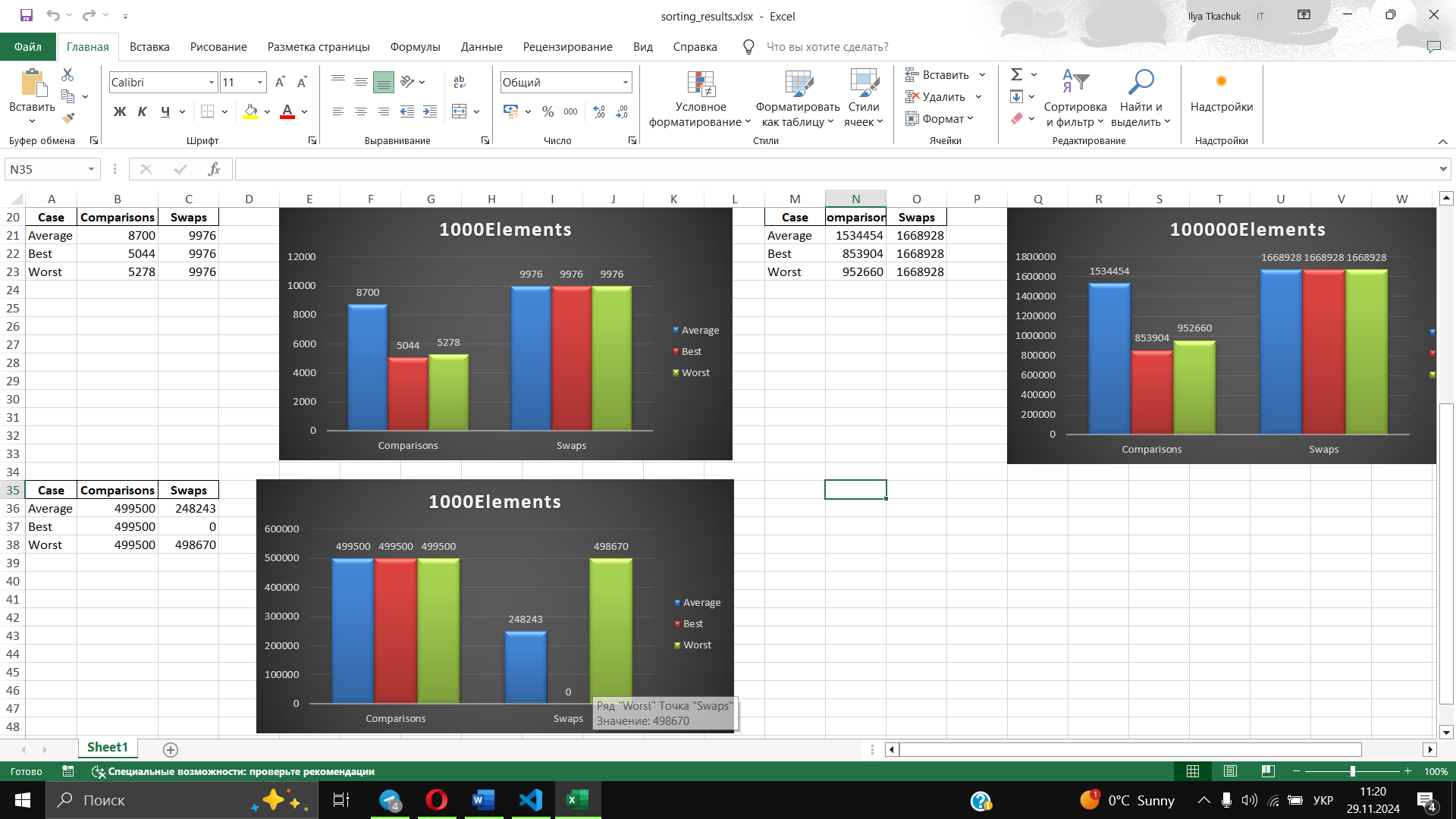
**Файл на 1000 елементів**







**Порівняння результатів сортування злиттям(верхній графік) та сортування бульбашкойю(нижній графік)**



**Внутрішне сортування**

Тут створюється динамічний масив arr розміром size

Масив заповнюється числами від 1 до size. Це буде найкращий випадок алгоритму сортування злиттям, тому що масив вже відсортований. У найкращому разі Merge Sort повинен робити мінімальну кількість порівнянь

Викликається функція сортування злиттям mergeSort. Передаються параметри:

Масив даних (arr.data() - передає покажчик на масив),

Індекси початку та кінця масиву (0 і size - 1),

Змінні для підрахунку порівнянь та перестановок (best\_comparisons та best\_swaps), які оновлюватимуться в процесі роботи алгоритму.

Алгоритм сортування злиттям виконує рекурсивний поділ масиву та злиття частин, причому в процесі роботи відстежуються кількість порівнянь та перестановок.

Масив заповнюється числами від size до 1 (за спаданням). Це найгірший випадок для Merge Sort, тому що елементи масиву йдуть у зворотному порядку, і сортування злиттям у такому разі має виконати максимальну кількість операцій.

Знову викликається mergeSort, але тепер для масиву, відсортованого у зворотному порядку.

Підраховуються порівняння та перестановки для найгіршого випадку.

Масив знову заповнюється числами від 1 до size, а потім перемішується випадковим чином за допомогою функції random\_shuffle.

Після перемішування масиву, знову викликається mergeSort, щоб підрахувати кількість порівнянь і перестановок у середньому випадку.

Після виконання сортування для всіх трьох випадків (кращий, найгірший, середній) виводяться результати: кількість порівнянь та перестановок для кожного випадку.

**Висновки:**

*Merge Sort має трудомісткість O(n log n), що відображається на результатах:*

*Навіть у найгіршому випадку спостерігається стабільний приріст трудомісткості, що погоджується з теоретичними оцінками.*

*Кількість перестановок не залежить від структури вхідних даних, що типово для Merge Sort.*

*Алгоритм демонструє передбачувану поведінку незалежно від того, чи це найкращий, середній чи найгірший випадок*

*Bubble Sort має трудомісткість O(n²) для порівнянь у всіх випадках, що підтверджується однаковими числами порівнянь.*

*Кількість перестановок демонструє залежність від структури вхідних даних:*

*У найкращому випадку (вже відсортовані дані) перестановок немає.*

*У середньому та найгіршому випадку кількість перестановок значно зростає*

**Порівняння**

*Трудомісткість Merge Sort є значно нижчою порівняно з Bubble Sort для великих масивів, особливо у середньому та найгіршому випадку.*

*Bubble Sort демонструє значну неефективність через квадратичну складність, що помітно на графіках для 100 00 елементів.*

*Merge Sort більш стійкий до початкового стану даних, в той час як Bubble Sort працює значно краще для відсортованих масивів (найкращий випадок).*