SET 3. Task A2

Algorithms. Merge Sort. Insertion Sort

ID посылки: `292994197`
GitHub: https://github.com/vasyukov1/HSE-FCS-SE-2-year/tree/main/Algorithms/Homework/SET_3/A2

- `a2i.cpp` - реализация Merge + Insertion Sort
- `a2.cpp` - решение задачи из ЛМС
- `merge_sort_random.txt` - результаты Merge Sort рандомного массива
- `merge_sort_reversed.txt` - результаты Merge Sort раверсивного массива
- `merge_sort_nearly_sorted.txt` - результаты Merge Sort почти отсортированного массива
- `hybrid_sort_random.txt` - результаты Merge + Insertion Sort рандомного массива
- `hybrid_sort_reversed.txt` - результаты Merge + Insertion Sort раверсивного массива
- `hybrid_sort_nearly_sorted.txt` - результаты Merge + Insertion Sort почти отсортированного массива
- `graphs.py` - построение графиков

Реализация Merge + Insertion Sort

```cpp
#include <iostream>
#include <vector>
using std::cout;
using std::cin;
using std::vector;

void merge(vector<int>& array, int left, int mid, int right);
void mergeSort(vector<int>& array, int left, int right);
void insertionSort(vector<int>& array, int left, int right);
void hybridMergeSort(vector<int>& array, int left, int right, int threshold);

int main() {
    std::ios::sync_with_stdio(false);
    std::cin.tie(nullptr);

    // Enter the array size
    int n;
    cin >> n;

    // Creation of array
    vector<int> array;
    for (int i = 0; i < n; ++i) {
        int el;
        cin >> el;
        array.push_back(el);
    }

    // 'threshol' - the number after which merge sort is applied
    int threshold = 15;
    hybridMergeSort(array, 0, n - 1, threshold);

    // Output array
    for (int i = 0; i < n; ++i) {
        cout << array[i] << ' ';
    }

    return 0;
}

void merge(vector<int>& array, int left, int mid, int right) {
    int n1 = mid - left + 1;
    int n2 = right - mid;

    // Divide the main array to 2 arrays
    vector<int> array1(n1);
    vector<int> array2(n2);

    // Fill arrays
    for (int i = 0; i < n1; ++i) {
        array1[i] = array[left + i];
    }
    for (int j = 0; j < n2; ++j) {
```

```
        array2[j] = array[mid + j + 1];
    }

    // Merge arrays to the main array
    int i = 0;
    int j = 0;
    int k = left;
    while (i < n1 && j < n2) {
        if (array1[i] < array2[j]) {
            array[k++] = array1[i++];
        } else {
            array[k++] = array2[j++];
        }
    }
    while (i < n1) {
        array[k++] = array1[i++];
    }
    while (j < n2) {
        array[k++] = array2[j++];
    }
}

void mergeSort(vector<int>& array, int left, int right) {
    if (left < right) {
        int mid = left + (right - left) / 2;
        mergeSort(array, left, mid);
        mergeSort(array, mid + 1, right);
        merge(array, left, mid, right);
    }
}

void insertionSort(vector<int>& array, int left, int right) {
    for (int i = left + 1; i <= right; ++i) {
        int el = array[i];
        int j = i - 1;
        // If previous element is greater than current, lift it up
        while (el < array[j]) {
            array[j + 1] = array[j];
            --j;
        }
        array[j + 1] = el;
    }
}

void hybridMergeSort(vector<int>& array, int left, int right, int threshold) {
    // If the array size less than threshol, use insertion sort
    if (right - left + 1 < threshold) {
        insertionSort(array, left, right);
        return;
    }
    // Use merge sort
    if (left < right) {
        int mid = left + (right - left) / 2;
        mergeSort(array, left, mid);
        mergeSort(array, mid + 1, right);
        merge(array, left, mid, right);
    }
}
```

Этап 1. Подготовка тестовых данных

Класс `ArrayGenerator` для создания массива случайных чисел.

- `getRandomArray(int size)` - функция, которая создаёт массив размера `size` со случайными числами.
- `getReversedArray(int size)` - функция, которая создаёт массив размера `size` отсортированный в обратном порядке (от наибольшего к меньшему).
- `getNearlySortedArray(int size, int swaps)` - функция, которая создаёт отсортированный массив размера `size`, но с `swaps` неотсортирванными элементами.

```
class ArrayGenerator {
private:
    vector<int> baseArray;
    int maxLength;
    int minRange;
    int maxRange;
```

```
    void generateBaseArray() {
        std::mt19937 gen(static_cast<unsigned>(std::time(nullptr)));
        std::uniform_int_distribution<int> dist(minRange, maxRange);
        for (int i = 0; i <= maxLength; ++i) {
            baseArray.push_back(dist(gen));
        }
    }

public:
    ArrayGenerator() : maxLength(10000), minRange(0), maxRange(6000) {
        generateBaseArray();
    }

    vector<int> getRandomArray(int size) {
        if (size > maxLength) {
            size = maxLength;
        }
        return vector<int> (baseArray.begin(), baseArray.begin() + size);
    }

    vector<int> getReversedArray(int size) {
        if (size > maxLength) {
            size = maxLength;
        }
        vector<int> reversedArray(baseArray.begin(), baseArray.begin() + size);
        std::sort(reversedArray.rbegin(), reversedArray.rend());
        return reversedArray;
    }

    vector<int> getNearlySortedArray(int size, int swaps) {
        if (size > maxLength) {
            size = maxLength;
        }
        vector<int> nearlySortedArray(baseArray.begin(), baseArray.begin() + size);
        std::sort(nearlySortedArray.begin(), nearlySortedArray.end());

        std::mt19937 gen(static_cast<unsigned>(std::time(nullptr)));
        std::uniform_int_distribution<int> dist(0, size - 1);

        for (int i = 0; i < swaps; ++i) {
            std::swap(nearlySortedArray[dist(gen)], nearlySortedArray[dist(gen)]);
        }

        return nearlySortedArray;
    }
};
```

Этап 2. Эмпирический анализ стандартного алгоритма MERGE SORT

Класс `SortTester` для подсчёта времени (в миллисекундах), за которое выполняются алгоритмы Merge Sort и Merge + Insertion Sort.

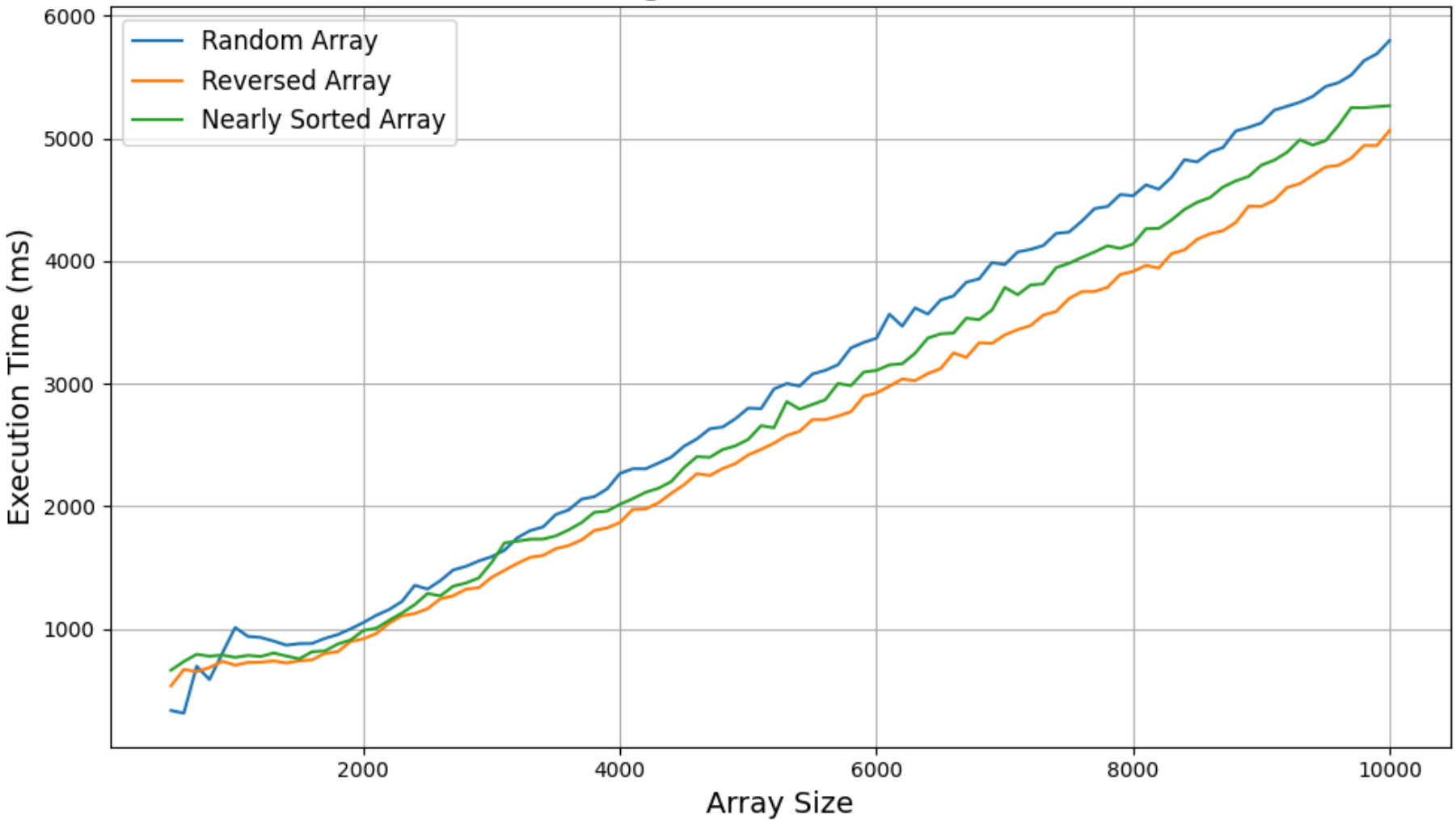
- `measureMergeSortTime` - функция для Merge Sort.
- `measureHybridMergeSortTime` - функция для Merge + Insertion Sort.

```
class SortTester {
public:
    double measureMergeSortTime(vector<int>& array, int size) {
        const int numTrials = 10;
        long long totalTime = 0;

        for (int trial = 0; trial < numTrials; ++trial) {
            vector<int> arrayToSort = array;
            auto start = std::chrono::high_resolution_clock::now();
            mergeSort(arrayToSort, 0, arrayToSort.size() - 1);
            auto elapsed = std::chrono::high_resolution_clock::now() - start;
            long long msec = std::chrono::duration_cast<std::chrono::milliseconds>(elapsed).count();
            totalTime += std::chrono::duration_cast<std::chrono::microseconds>(elapsed).count();
        }

        return static_cast<double> (totalTime) / numTrials;
    }

    double measureHybridMergeSortTime(vector<int>& array, int size, int threshold) {
```

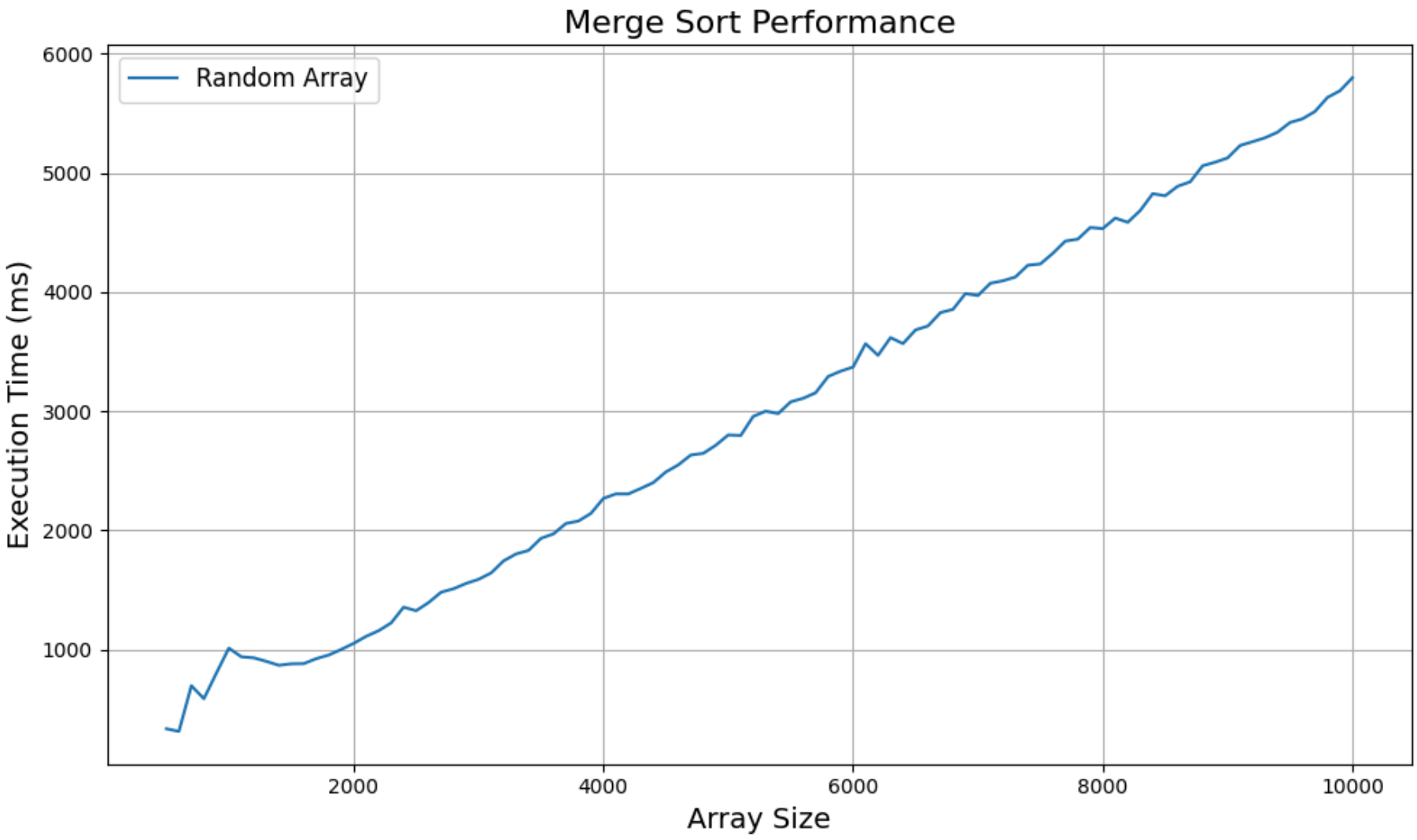
```
        const int numTrials = 10;
        long long totalTime = 0;

        for (int trial = 0; trial < numTrials; ++trial) {
            vector<int> arrayToSort = array;
            auto start = std::chrono::high_resolution_clock::now();
            hybridMergeSort(arrayToSort, 0, arrayToSort.size() - 1, threshold);
            auto elapsed = std::chrono::high_resolution_clock::now() - start;
            long long msec = std::chrono::duration_cast<std::chrono::milliseconds>(elapsed).count();
            totalTime += std::chrono::duration_cast<std::chrono::microseconds>(elapsed).count();
        }

        return static_cast<double> (totalTime) / numTrials;
    }
};
```

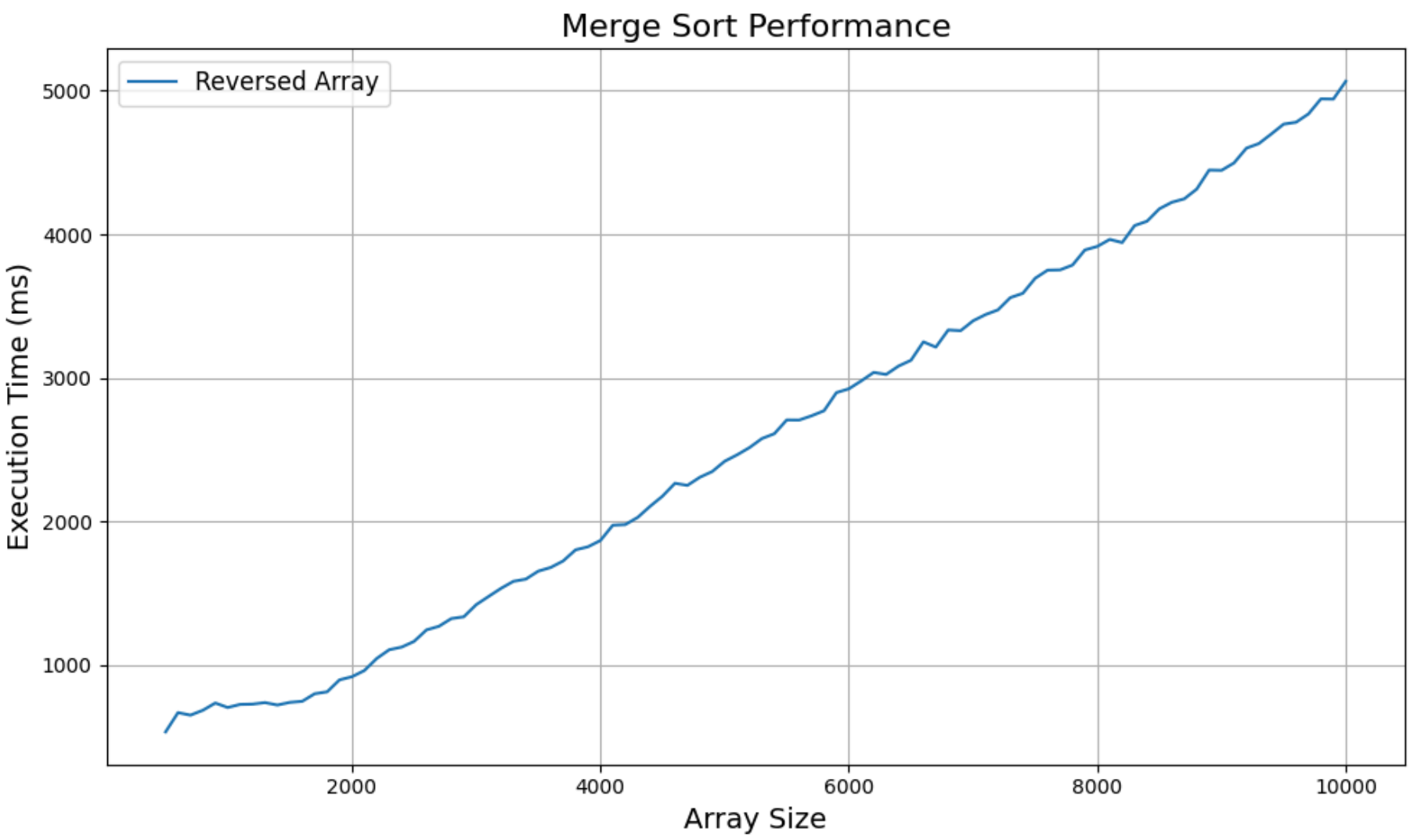
Результаты

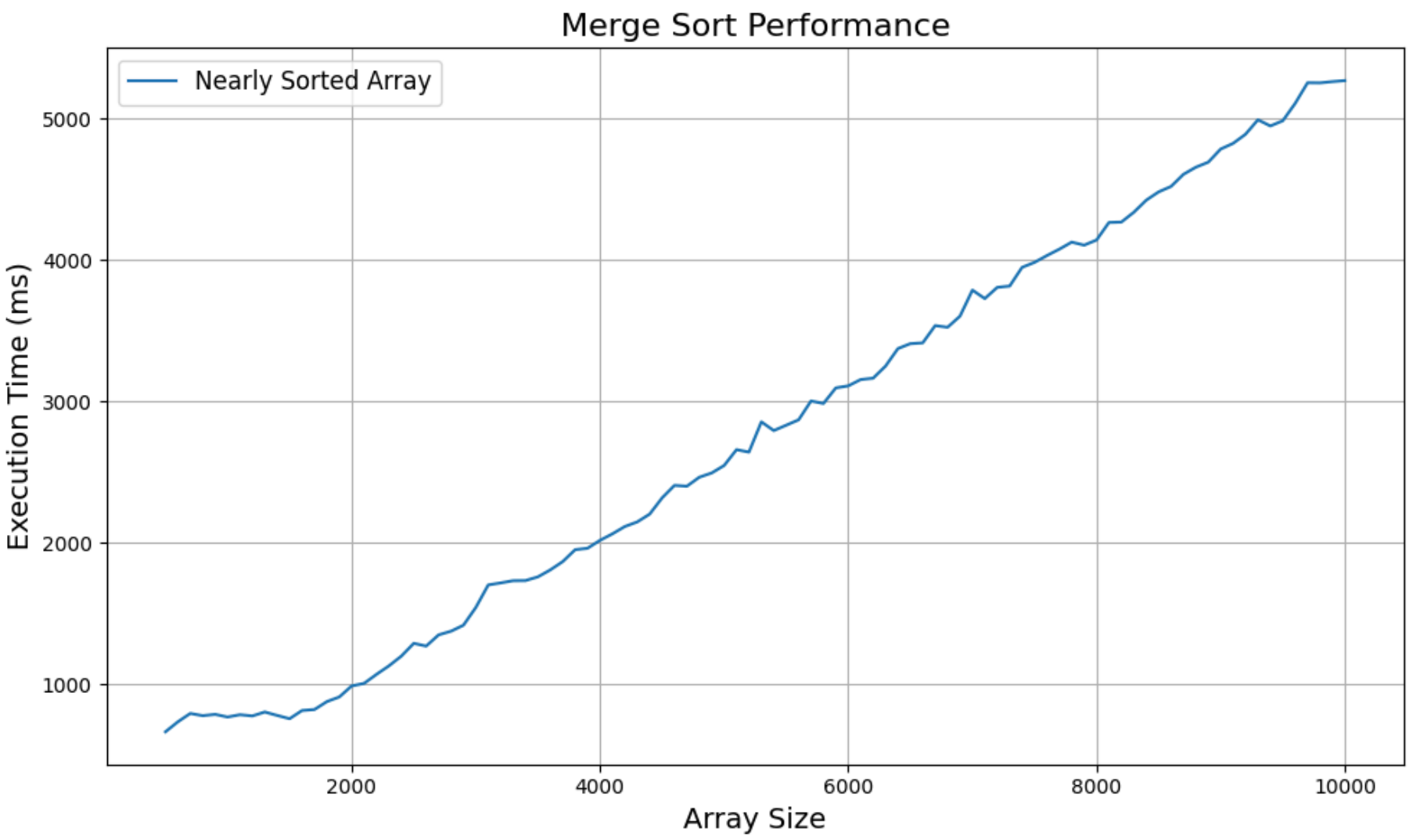
Общий результат использования Merge Sort для 3 типов массивов.
На небольших данных (приблизительно до 700) Merge Sort для массива с рандомными числами работает быстрее, чем для двух остальных случаев. Но чем больше размер массива, тем сортировка реверсивного массива быстрее.



Графики по отдельности:







Этап 3. Эмпирический анализ гибридного алгоритма MERGE+INSERTION SORT

В зависимости от разных порогов перехода с `Merge Sort` на `Insertion Sort` получаем графики для `Random Array`, `Reversed Array` и `Nearly Sorted Array`.


Merge + Insertion Sort Performance
Random Array
Execution Time (ms)
0
2000
4000
6000
8000
10000
2000
4000
6000
8000
10000
Array Size
Threshold 5
Threshold 10
Threshold 20
Threshold 30
Threshold 50


Merge + Insertion Sort Performance
Reversed Array
Execution Time (ms)
0
1000
2000
3000
4000
5000
2000
4000
6000
8000
10000
Array Size
Threshold 5
Threshold 10
Threshold 20
Threshold 30
Threshold 50

Этап 4. Сравнительный анализ

1. Преимущество гибридного алгоритма на больших данных

На массивах большого размера гибридный алгоритм `Merge + Insertion Sort` работает быстрее по сравнению с обычным `Merge Sort`. Это происходит из-за того, что на уровнях рекурсии, где размер подмассивов становится достаточно малым, алгоритм вставками `Insertion Sort` более эффективен.

2. Зависимость от порогового значения перехода (threshold)

Однако эффективность гибридного алгоритма зависит от порогового значения `threshold`, определяющего момент переключения с `Merge Sort` на `Insertion Sort`.

Например:

- В некоторых случаях значение `threshold` подобрано неудачно, что приводит к ухудшению времени выполнения. Например, для массива длиной `6600` элементов из категории `Nearly Sorted Array` время работы резко увеличивается из-за слишком раннего переключения на вставки.
- Обратная ситуация возникает для массива длиной `5100` в той же категории, где переход на вставки происходит слишком поздно. Это также ухудшает время выполнения.

3. Наилучший порог перехода

Оптимальное значение `threshold`, при котором гибридный алгоритм демонстрирует стабильные результаты без резких скачков времени работы, оказалось равным `20`.

4. Влияние структуры данных

Скорость выполнения обоих алгоритмов зависит от типа массива:

- Самые быстрые времена работы наблюдаются на массивах, отсортированных в обратном порядке. Это объясняется тем, что в таких массивах меньше всего пересечений в уровнях слияния, что снижает общее количество операций.
- На случайных массивах алгоритмы работают медленнее, так как отсутствует какая-либо упорядоченность, и слияние требует большего числа операций.