Задача A2

Зарина Рамазанова

SET 3, ноябрь 2025

Этап 1

Реализация класса ArrayGenerator:

```cpp
class ArrayGenerator {
private:
    std::random_device rd;
    std::mt19937 gen;

public:
    ArrayGenerator() : gen(rd()) {}

    std::vector<int> generateRandomArray(size_t size) {
        std::vector<int> array(size);
        std::uniform_int_distribution<int> dis(0, 6000);

        for (size_t i = 0; i < size; ++i) {
            array[i] = dis(gen);
        }

        return array;
    }

    std::vector<int> generateReverseSortedArray(size_t size) {
        std::vector<int> array = generateRandomArray(size);
        std::sort(array.begin(), array.end(), std::greater<int>());
        return array;
    }

    std::vector<int> generateAlmostSortedArray(size_t size, int swapPairs = 10) {
        std::vector<int> array = generateRandomArray(size);
        std::sort(array.begin(), array.end());

        std::uniform_int_distribution<size_t> indexDis(0, size - 1);

        for (int i = 0; i < swapPairs; ++i) {
            size_t idx1 = indexDis(gen);
            size_t idx2 = indexDis(gen);
            std::swap(array[idx1], array[idx2]);
        }

        return array;
    }

    std::vector<int> getSubarray(const std::vector<int>& source, size_t size) {
        return std::vector<int>(source.begin(), source.begin() + size);
    }
};
```

Этап 2

При замере была использована следующая реализация MergeSort:

```cpp
void merge(std::vector<int>& arr, int left, int mid, int right) {
    int n1 = mid - left + 1;
    int n2 = right - mid;

    std::vector<int> L(n1), R(n2);

    for (int i = 0; i < n1; i++)
        L[i] = arr[left + i];
    for (int j = 0; j < n2; j++)
        R[j] = arr[mid + 1 + j];

    int i = 0, j = 0, k = left;
    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 mergeSort(std::vector<int>& arr, int left, int right) {
    if (left >= right) return;

    int mid = left + (right - left) / 2;
    mergeSort(arr, left, mid);
    mergeSort(arr, mid + 1, right);
    merge(arr, left, mid, right);
}

void mergeSort(std::vector<int>& arr) {
    if (arr.empty()) return;
    mergeSort(arr, 0, arr.size() - 1);
}
```

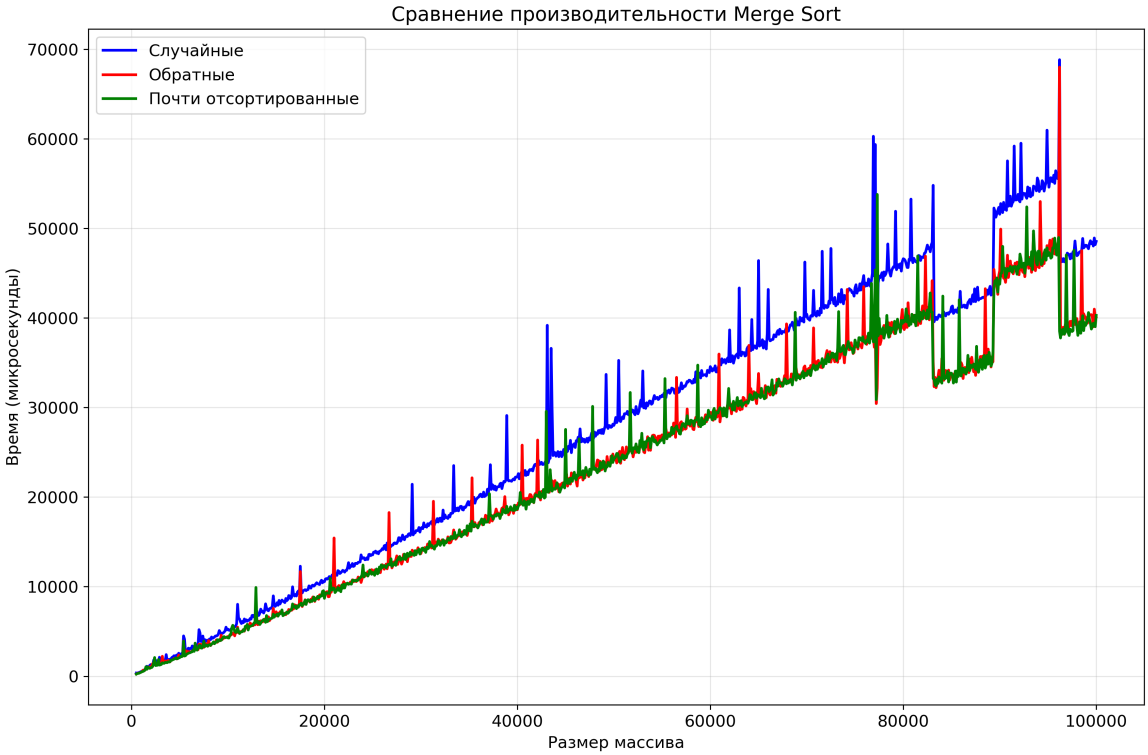
Для реализации представления графиков был реализован main() с записью результатов эксперимента с каждым видом массива в csv-файлы:

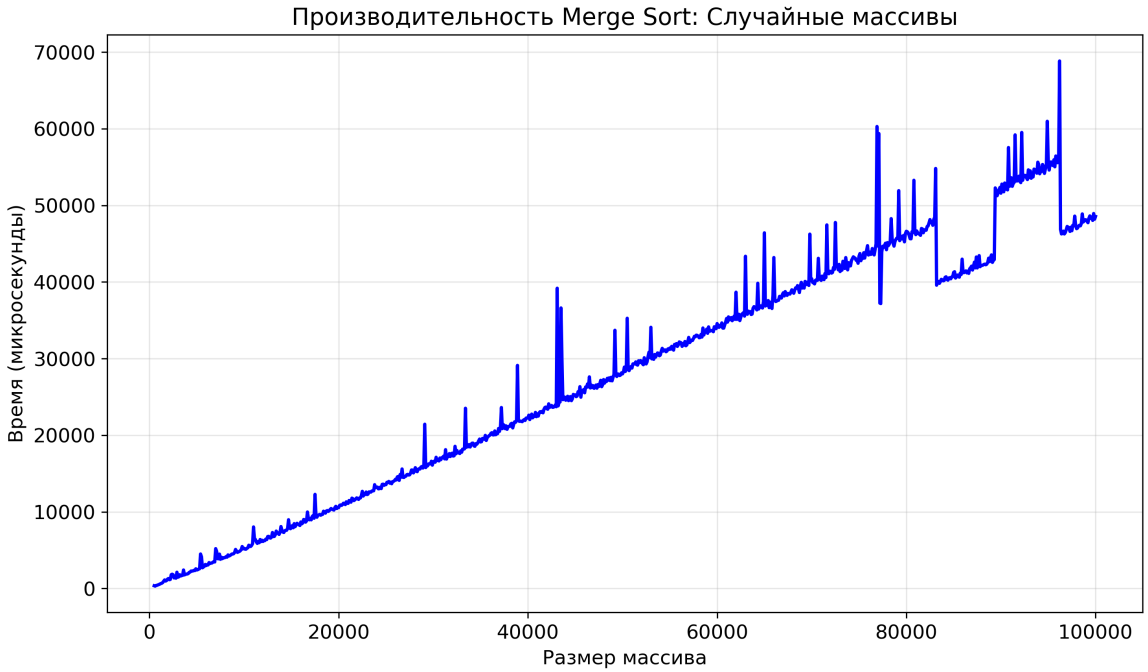
```cpp
    int main() {
    ArrayGenerator generator;
    const size_t MAX_SIZE = 100000;
    const size_t MIN_SIZE = 500;
    const size_t STEP = 100;

    // Генерируем базовые массивы максимального размера
    auto randomBase = generator.generateRandomArray(MAX_SIZE);
    auto reverseBase = generator.generateReverseSortedArray(MAX_SIZE);
    auto almostBase = generator.generateAlmostSortedArray(MAX_SIZE, 50);

    // Файлы для сохранения результатов
    std::ofstream randomFile("merge_sort_random.csv");
```

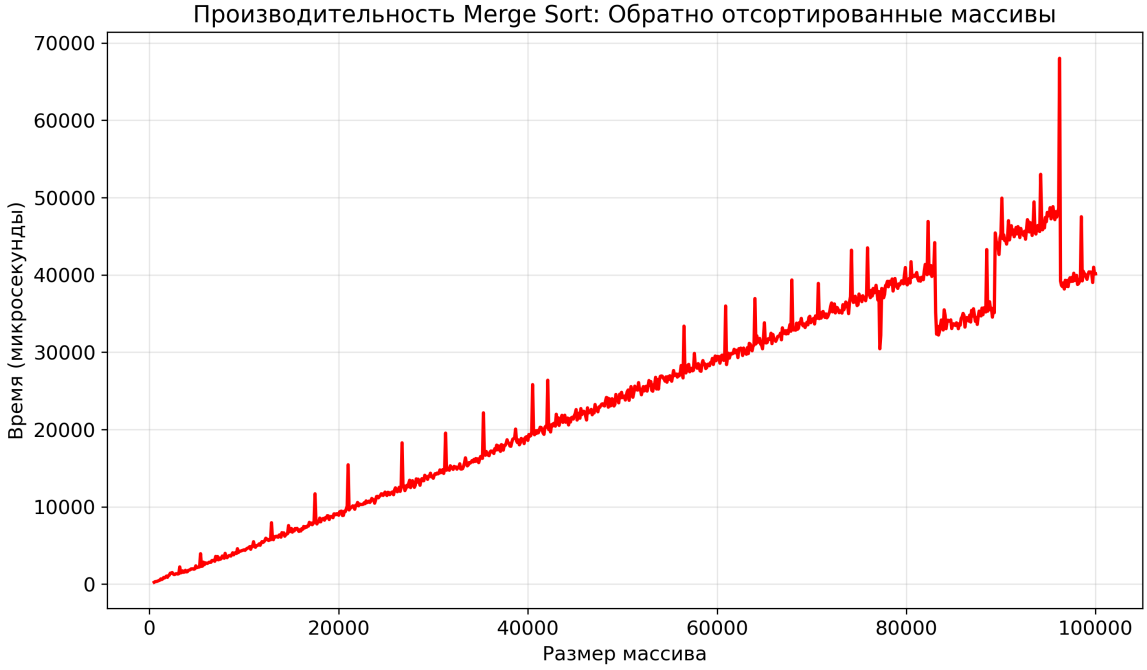
```
    std::ofstream reverseFile("merge_sort_reverse.csv");
    std::ofstream almostFile("merge_sort_almost.csv");

    randomFile << "size,time(ms)\n";
    reverseFile << "size,time(ms)\n";
    almostFile << "size,time(ms)\n";


    for (size_t size = MIN_SIZE; size <= MAX_SIZE; size += STEP) {
        // Случайные массивы
        auto randomArray = generator.getSubarray(randomBase, size);
        auto start = std::chrono::high_resolution_clock::now();
        mergeSort(randomArray);
        auto elapsed = std::chrono::high_resolution_clock::now() - start;
        long long randomTime = std::chrono::duration_cast<std::chrono::microseconds>(elapsed).count();
        randomFile << size << "," << randomTime << "\n";

        // Обратно отсортированные массивы
        auto reverseArray = generator.getSubarray(reverseBase, size);
        start = std::chrono::high_resolution_clock::now();
        mergeSort(reverseArray);
        elapsed = std::chrono::high_resolution_clock::now() - start;
        long long reverseTime = std::chrono::duration_cast<std::chrono::microseconds>(elapsed).count()
    ;
        reverseFile << size << "," << reverseTime << "\n";

        // Почти отсортированные массивы
        auto almostArray = generator.getSubarray(almostBase, size);
        start = std::chrono::high_resolution_clock::now();
        mergeSort(almostArray);
        elapsed = std::chrono::high_resolution_clock::now() - start;
        long long almostTime = std::chrono::duration_cast<std::chrono::microseconds>(elapsed).count();
        almostFile << size << "," << almostTime << "\n";
    }

    randomFile.close();
    reverseFile.close();
    almostFile.close();

    std::cout << "Результаты сохранены в CSV файлы." << std::endl;

    return 0;
}
```

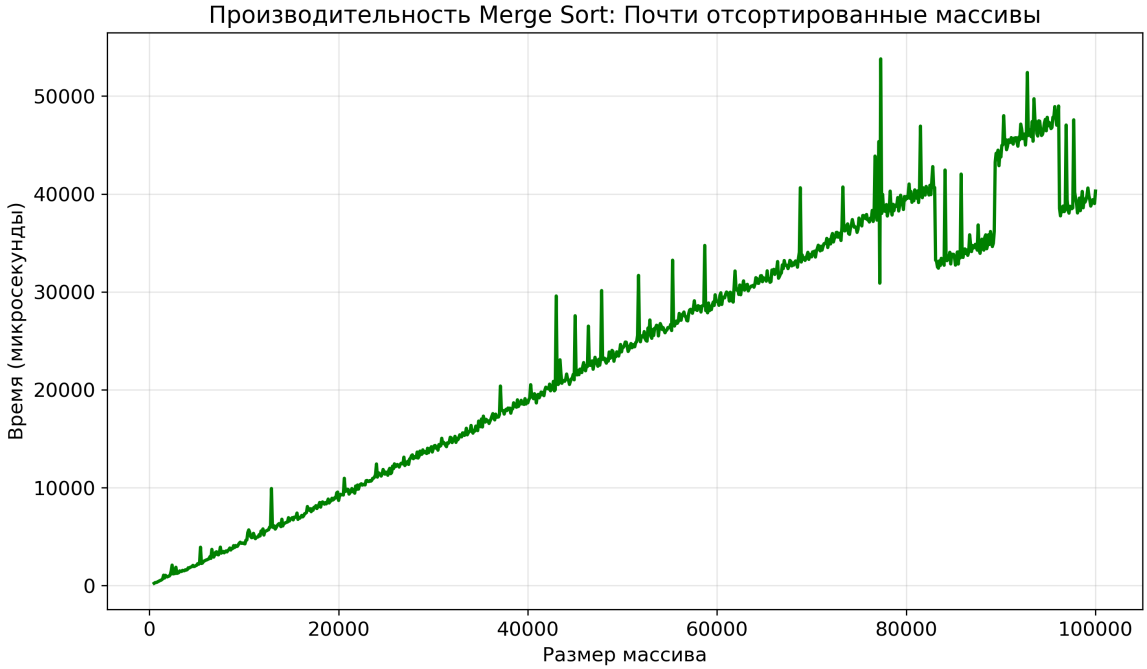
Графики

Были получены следующие графики:









Этап 3

Реализация класса SortTester:

```
    class SortTester {
private:
    ArrayGenerator generator;

    // Insertion Sort для небольших массивов
    void insertionSort(std::vector<int>& arr, int left, int right) {
        for (int i = left + 1; i <= right; i++) {
            int key = arr[i];
            int j = i - 1;

            while (j >= left && arr[j] > key) {
```

```
                arr[j + 1] = arr[j];
                j--;
            }
            arr[j + 1] = key;
        }
    }

    // Гибридный Merge Sort
    void hybridMergeSort(std::vector<int>& arr, int left, int right, int threshold) {
        if (left >= right) return;

        // Если размер подмассива меньше порога, используем Insertion Sort
        if (right - left + 1 <= threshold) {
            insertionSort(arr, left, right);
            return;
        }

        int mid = left + (right - left) / 2;
        hybridMergeSort(arr, left, mid, threshold);
        hybridMergeSort(arr, mid + 1, right, threshold);
        merge(arr, left, mid, right);
    }

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

        std::vector<int> L(n1), R(n2);

        for (int i = 0; i < n1; i++)
            L[i] = arr[left + i];
        for (int j = 0; j < n2; j++)
            R[j] = arr[mid + 1 + j];

        int i = 0, j = 0, k = left;
        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++;
        }
    }

public:
    SortTester() {}

    // Тестирование гибридного алгоритма с заданным порогом
    long long testHybridSort(const std::vector<int>& originalArray, int threshold) {
        std::vector<int> arr = originalArray; // Копируем массив
        auto start = std::chrono::high_resolution_clock::now();
```

```
78
79          if (!arr.empty()) {
80              hybridMergeSort(arr, 0, arr.size() - 1, threshold);
81          }
82
83          auto elapsed = std::chrono::high_resolution_clock::now() - start;
84          return std::chrono::duration_cast<std::chrono::microseconds>(elapsed).count();
85      }
86
87      // Тестирование стандартного Merge Sort для сравнения
88      long long testStandardMergeSort(const std::vector<int>& originalArray) {
89          std::vector<int> arr = originalArray;
90          auto start = std::chrono::high_resolution_clock::now();
91
92          if (!arr.empty()) {
93              standardMergeSort(arr, 0, arr.size() - 1);
94          }
95
96          auto elapsed = std::chrono::high_resolution_clock::now() - start;
97          return std::chrono::duration_cast<std::chrono::microseconds>(elapsed).count();
98      }
99
100 private:
101     void standardMergeSort(std::vector<int>& arr, int left, int right) {
102         if (left >= right) return;
103
104         int mid = left + (right - left) / 2;
105         standardMergeSort(arr, left, mid);
106         standardMergeSort(arr, mid + 1, right);
107         merge(arr, left, mid, right);
108     }
109 };
```

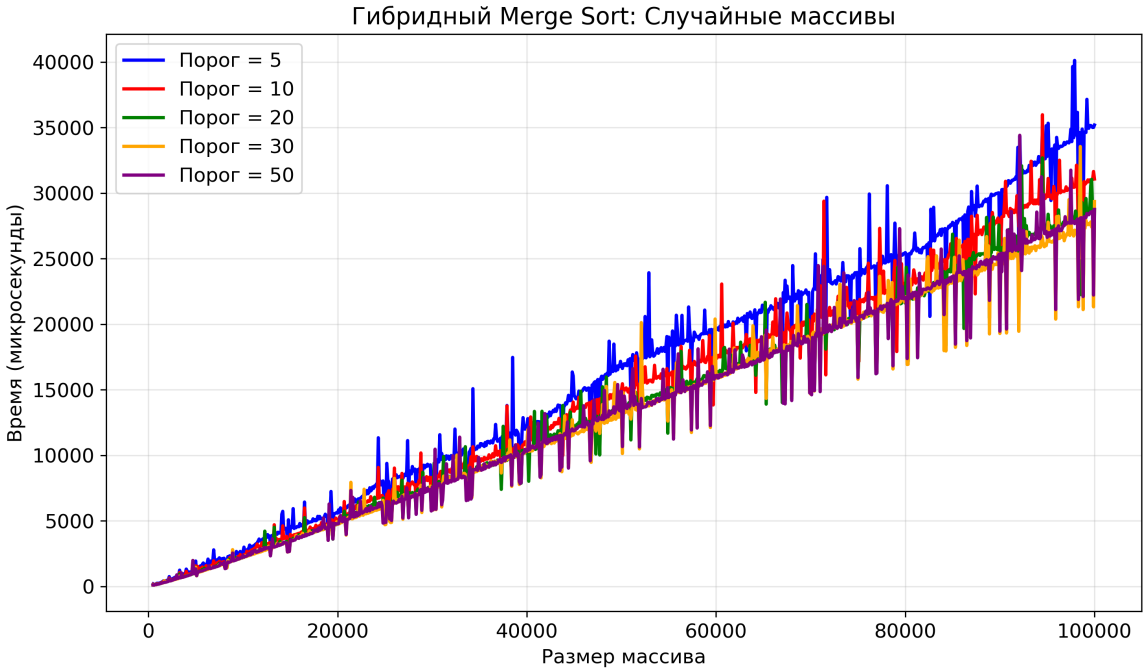
Для реализации представления графиков был реализован main() с записью результатов эксперимента с каждым видом массива в csv-файлы:

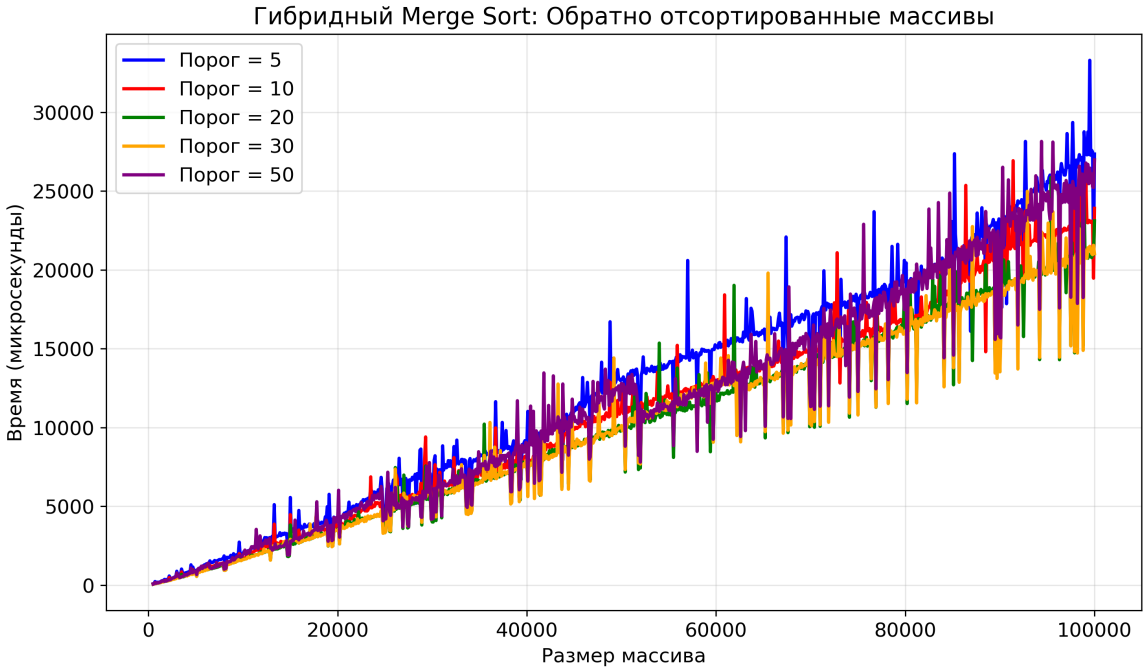
```
1   int main() {
2       ArrayGenerator generator;
3       SortTester tester;
4
5       const size_t MAX_SIZE = 100000;
6       const size_t MIN_SIZE = 500;
7       const size_t STEP = 100;
8
9       // Пороги для переключения на Insertion Sort
10      std::vector<int> thresholds = {5, 10, 20, 30, 50};
11
12      auto randomBase = generator.generateRandomArray(MAX_SIZE);
13      auto reverseBase = generator.generateReverseSortedArray(MAX_SIZE);
14      auto almostBase = generator.generateAlmostSortedArray(MAX_SIZE, 50);
15
16      std::ofstream randomFile("hybrid_sort_random.csv");
17      std::ofstream reverseFile("hybrid_sort_reverse.csv");
18      std::ofstream almostFile("hybrid_sort_almost.csv");
19      std::ofstream comparisonFile("comparison_results.csv");
20
21      randomFile << "size,threshold,time(ms)\n";
22      reverseFile << "size,threshold,time(ms)\n";
23      almostFile << "size,threshold,time(ms)\n";
24      comparisonFile << "size,data_type,algorithm,time(ms)\n";
25
26
27      for (size_t size = MIN_SIZE; size <= MAX_SIZE; size += STEP) {
28
29          // Тестирование для случайных массивов
30          auto randomArray = generator.getSubarray(randomBase, size);
31          long long standardTime = tester.testStandardMergeSort(randomArray);
```

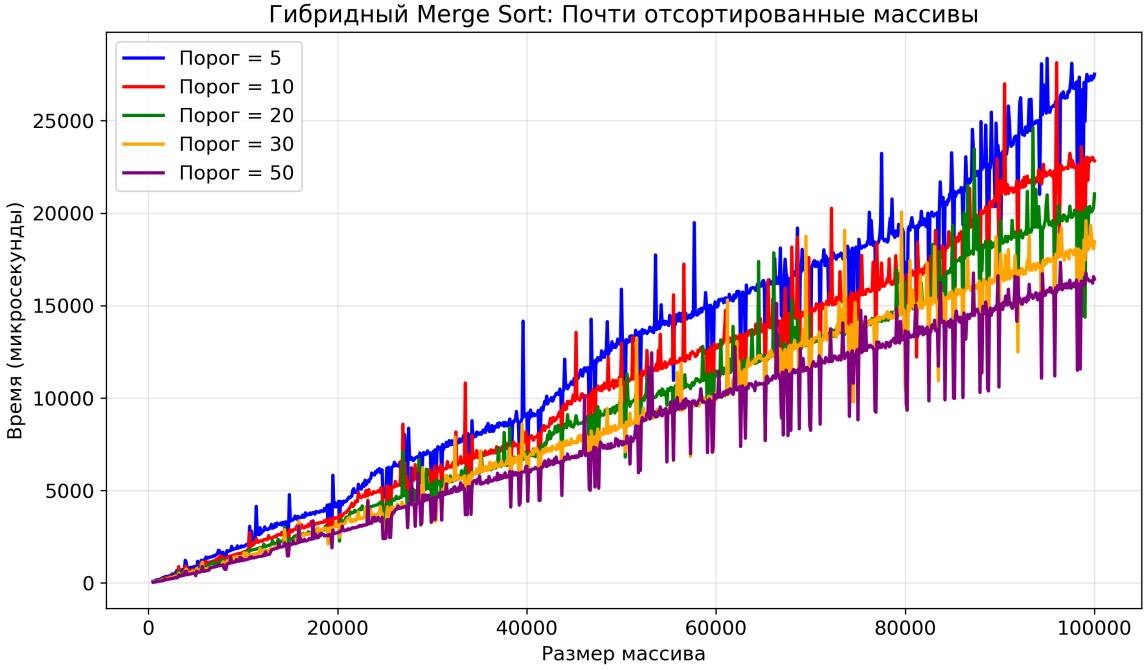
```
        comparisonFile << size << ",random,standard," << standardTime << "\n";

        for (int threshold : thresholds) {
            long long hybridTime = tester.testHybridSort(randomArray, threshold);
            randomFile << size << "," << threshold << "," << hybridTime << "\n";
            comparisonFile << size << ",random,hybrid_" << threshold << "," << hybridTime << "\n";
        }

        // Тестирование для обратно отсортированных массивов
        auto reverseArray = generator.getSubarray(reverseBase, size);
        standardTime = tester.testStandardMergeSort(reverseArray);
        comparisonFile << size << ",reverse,standard," << standardTime << "\n";

        for (int threshold : thresholds) {
            long long hybridTime = tester.testHybridSort(reverseArray, threshold);
            reverseFile << size << "," << threshold << "," << hybridTime << "\n";
            comparisonFile << size << ",reverse,hybrid_" << threshold << "," << hybridTime << "\n";
        }

        // Тестирование для почти отсортированных массивов
        auto almostArray = generator.getSubarray(almostBase, size);
        standardTime = tester.testStandardMergeSort(almostArray);
        comparisonFile << size << ",almost,standard," << standardTime << "\n";

        for (int threshold : thresholds) {
            long long hybridTime = tester.testHybridSort(almostArray, threshold);
            almostFile << size << "," << threshold << "," << hybridTime << "\n";
            comparisonFile << size << ",almost,hybrid_" << threshold << "," << hybridTime << "\n";
        }
    }

    randomFile.close();
    reverseFile.close();
    almostFile.close();
    comparisonFile.close();

    std::cout << "Результаты сохранены в CSV файлы." << std::endl;

    return 0;
}
```

Графики

Были получены следующие графики:







Этап 4

На основе проведенного исследования двух версий сортировки слиянием можно сделать следующие выводы:

1. Стандартный Merge Sort показал стабильную работу на всех типах данных, что подтверждает его теоретическую сложность O(n log n). Однако гибридная версия с Insertion Sort в некоторых случаях работает быстрее.

2. Гибридный алгоритм показал лучшие результаты на почти отсортированных массивах. Это объясняется тем, что Insertion Sort эффективно обрабатывает почти упорядоченные данные.

3. Для случайных массивов лучшим оказался порог 10-20 элементов. Слишком маленький порог (5) вел к лишним переключениям между алгоритмами, а слишком большой (50) не использовал преимущества Insertion Sort.

4. Важным результатом стало определение критического порога. При значении 50 элементов гибридный алгоритм часто работал медленнее стандартного, особенно на больших массивах. Insertion Sort на 50 элементах уже проявляет низкую эффективность, а постоянные проверки порога добавляют накладные расходы.

5. На обратно отсортированных массивах разница между алгоритмами была наименьшей. Лишь при пороге 20-30 элементов гибридная версия показывала небольшое преимущество.

В итоге, гибридный алгоритм может быть быстрее стандартного, но требует правильной настройки порога. Для большинства случаев подходит значение 15-20 элементов.

Прочие данные

Номер посылки на codeforces: 348490365
Ссылка на публичный репозиторий с материалами(в том числе с реализациями классов ArrayGenerator и SortTester): `github.com/sunflowerthu/ADS_A2_SET3`