Исаев Кирилл БПИ2310
SET 3
Задание A2.

ID посылки: 292682209
Ссылка на GitHub
Реализация ArrayGenerator:

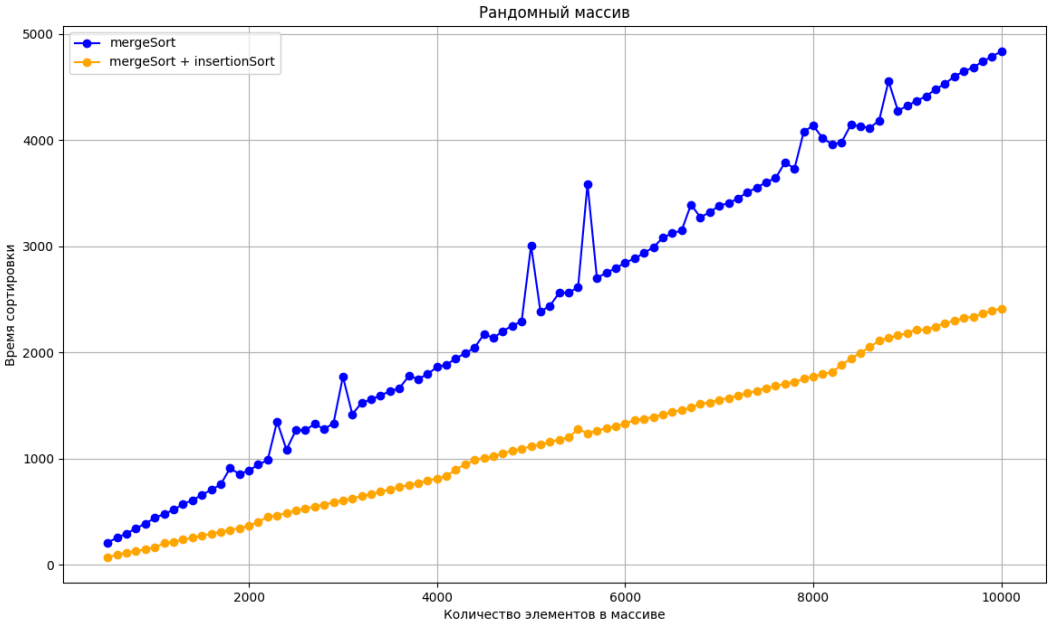
```
class ArrayGenerator {
public:
    ArrayGenerator() {
        randomArrays.resize( new_size: ARRAY_AMOUNT);
        sortedArrays.resize( new_size: ARRAY_AMOUNT);
        semiSortedArrays.resize( new_size: ARRAY_AMOUNT);
        fillArrays();
    }

    vector<int> generateRandomArray(int arraySize) {
        random_device rand_dev;
        mt19937 generator( sd: rand_dev());
        uniform_int_distribution<> distr( a: 1,  b: 6000);

        vector<int> array( n: arraySize);
        for (int i = 0; i < arraySize; ++i) {
            array[i] = distr( &: generator);
        }
        return array;
    }

    vector<int> generateSortedArray(int arraySize) {
        vector<int> array( n: arraySize);
        array = generateRandomArray(arraySize);
        sort( first: array.begin(),  last: array.end(),  comp: greater<int>());
        return array;
    }

    vector<int> generateSemiSortedArray(int arraySize) {
        random_device rand_dev;
        mt19937 generator( sd: rand_dev());
        uniform_int_distribution<> distr( a: 1,  b: 6000);

        vector<int> array( n: arraySize);
        array = generateRandomArray(arraySize);
        sort( first: array.begin(),  last: array.end());
        for (int i = 0; i < 20; ++i) {
            swap( &: array[distr( &: generator) % arraySize],  &: array[distr( &: generator)% arraySize]);
        }
        return array;
```

```
    void fillArrays() {
        int ind = 0;
        for (int size = 500; size <= 10000; size += 100) {
            randomArrays[ind] = generateRandomArray( arraySize: size);
            sortedArrays[ind] = generateSortedArray( arraySize: size);
            semiSortedArrays[ind] = generateSemiSortedArray( arraySize: size);
            ++ind;
        }
    }

    vector<vector<int>> getRandomArrays() {
        return randomArrays;
    }

    vector<vector<int>> getSortedArrays() {
        return sortedArrays;
    }

    vector<vector<int>> getSemiSortedArrays() {
        return semiSortedArrays;
    }

private:
    const int ARRAY_AMOUNT = 96;
    vector<vector<int>> randomArrays;
    vector<vector<int>> sortedArrays;
    vector<vector<int>> semiSortedArrays;
};
```

Реализация SortTester:

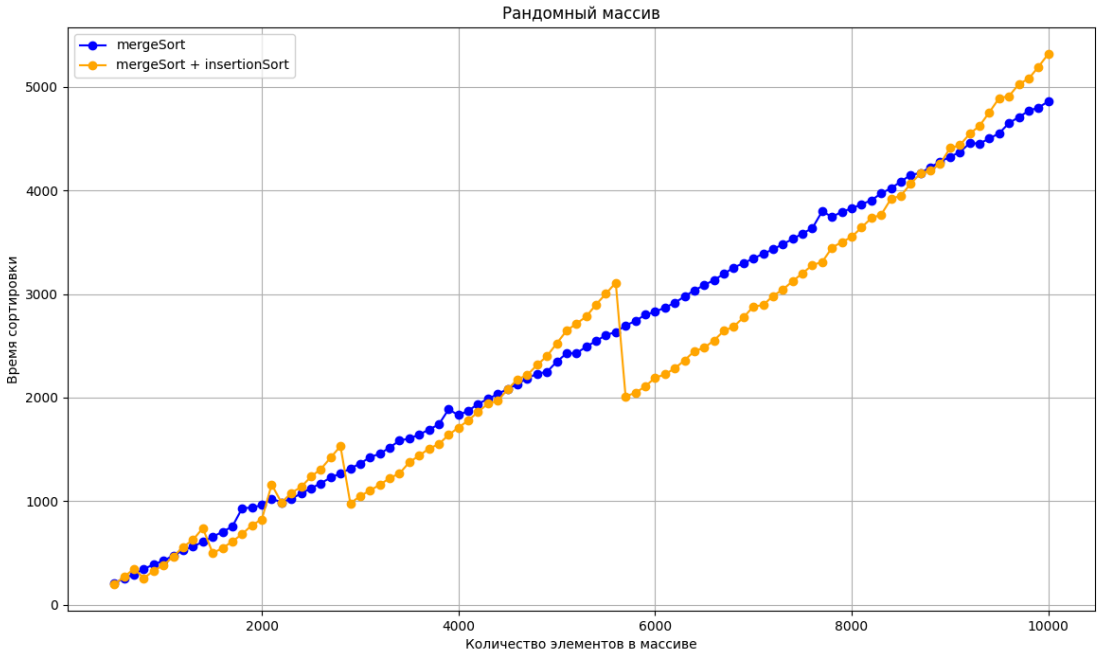
```
class SortTester {
public:
    long long Test(vector<int> array) {
        auto start : time_point<...> = std::chrono::high_resolution_clock::now();
        mergeSort( &: array, left: 0, right: static_cast<int>(array.size()));
        auto elapsed : duration<...> = std::chrono::high_resolution_clock::now() - start;
        long long msec = std::chrono::duration_cast<std::chrono::microseconds>( d: elapsed).count();
        return msec;
    }
    long long TestHybrid(vector<int> array) {
        auto start : time_point<...> = std::chrono::high_resolution_clock::now();
        mergeSortHybrid( &: array, left: 0, right: static_cast<int>(array.size()));
        auto elapsed : duration<...> = std::chrono::high_resolution_clock::now() - start;
        long long msec = std::chrono::duration_cast<std::chrono::microseconds>( d: elapsed).count();
        return msec;
    }
};
```

График №1(threshold = 15, сгенерированы рандомные массивы)



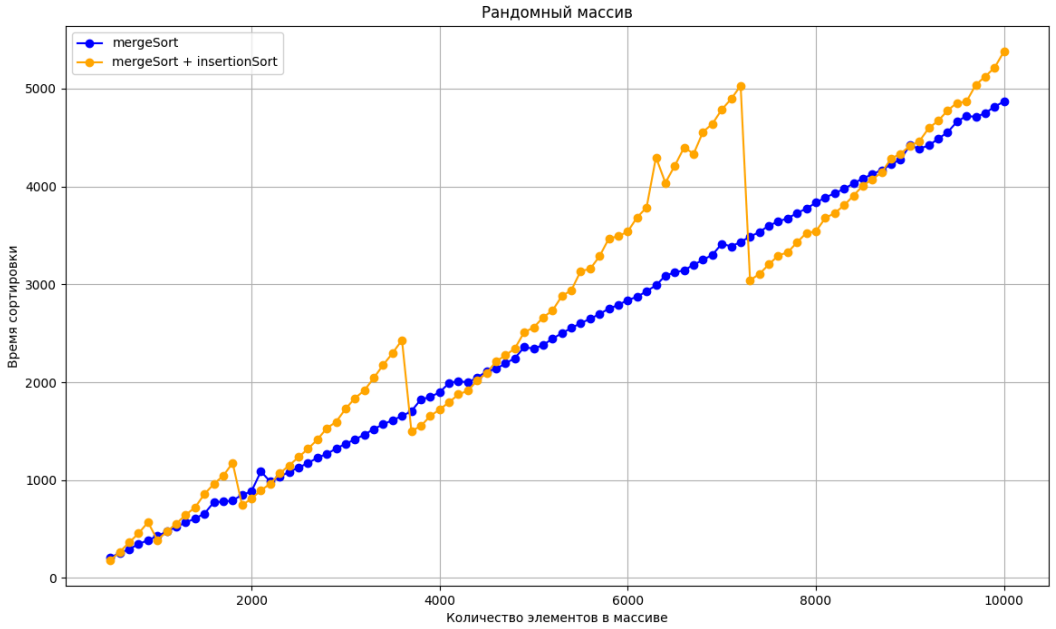
Можем заметить, что при малых входных данных разница в работе алгоритмов не сильно отличается. При этом при увеличении количества входных данных работа алгоритмов начинает сильно отличаться по времени. Скорость сортировки массива размера 10000 у обычного алгоритма = 4835 микросекунд, у гибридного = 2414 микросекунды.

График №2(threshold = 350, сгенерированы рандомные массивы)



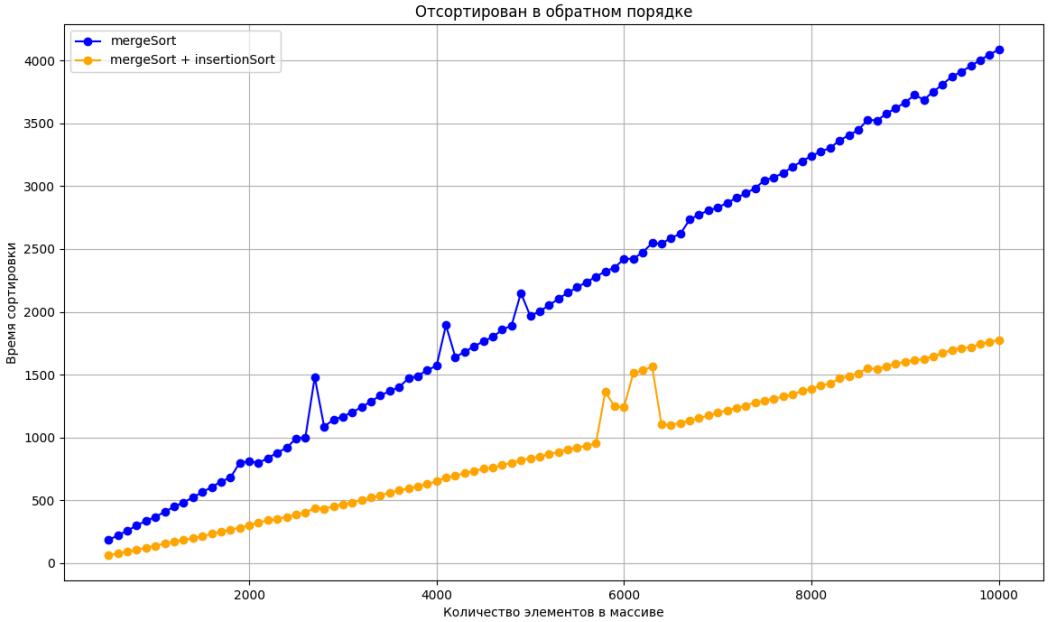
Можно заметить, что при threshold = 350 гибридный алгоритм иногда начинает проигрывать по скорости обычному mergeSort, но он по прежнему в большинстве случаев работает быстрее.

График №3(threshold = 450, сгенерированы рандомные массивы)



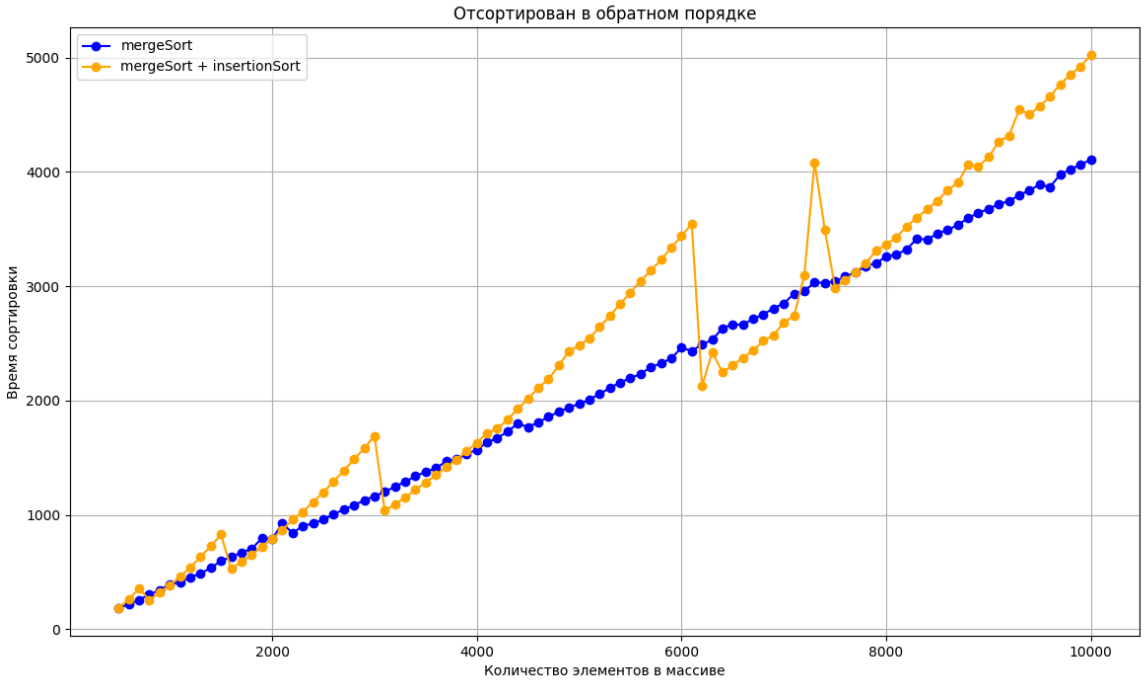
При threshold = 450 заметно, что теперь гибридный алгоритм почти всегда проигрывает обычному по скорости. Поэтому приблизительно при такой константе уже стоит использовать обычный mergeSort, нежели его гибридную реализацию.

График №4(threshold = 15, сгенерированы отсортированные в обратном порядке массивы)



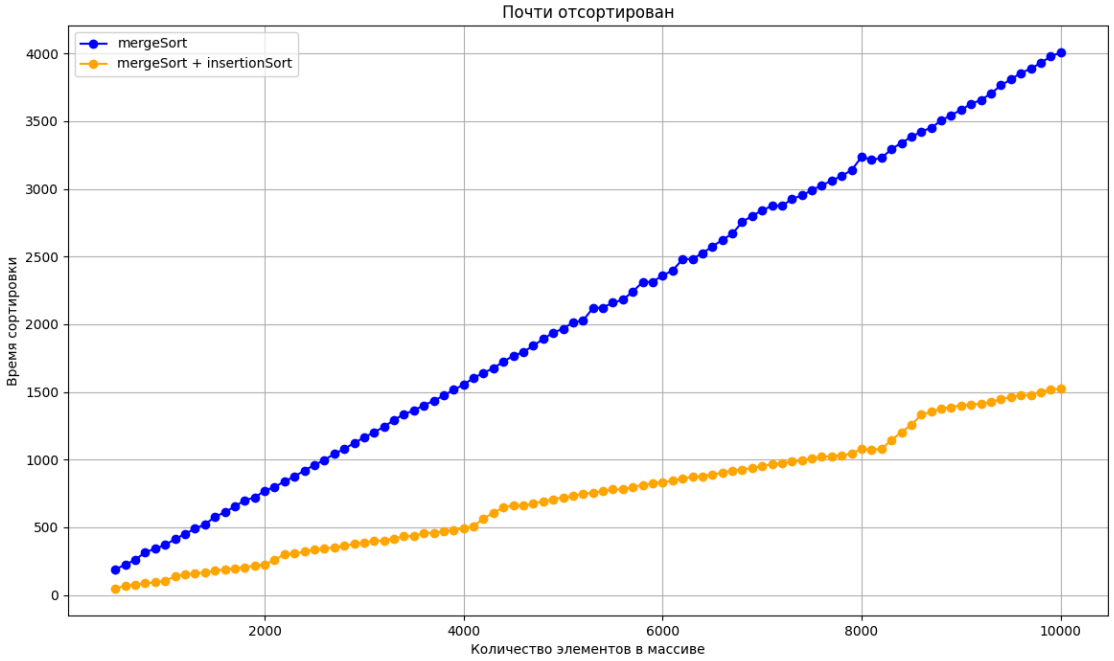
Если сравнивать с предыдущими результатами, то этот вид массивов оба алгоритма сортируют быстрее, чем просто случайно сгенерированные массивы(4835 микросекунд и 2414 микросекунд, против 4087 и 1776). Если сравнивать два алгоритма между собой, то результат точно такой же, как на Графике№1.

График №5(threshold = 190, сгенерированы отсортированные в обратном порядке массивы)



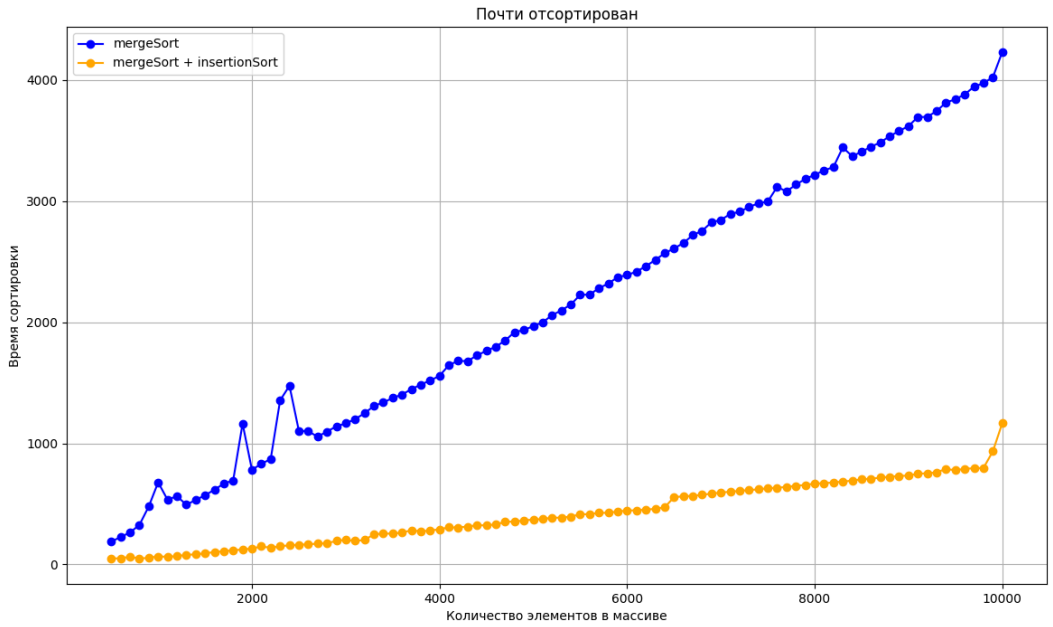
При threshold = 190 заметно, что гибридный алгоритм начинает работать хуже и его уже не стоит использовать(4107 микросекунд у mergeSort и 5022 микросекунды у его гибрида)

График №6(threshold = 15, сгенерированы ”почти” отсортированные массивы)



Если сортировать “почти” отсортированные массивы, то оба этих алгоритма покажут самую быструю скорость сортировки(по сравнению с другими видами массивов). 4009 микросекунд для mergeSort и 1522 микросекунд для гибрида.

График №7(threshold = 200, сгенерированы ”почти” отсортированные массивы)



При повышении threshold скорость работы гибридного алгоритма уменьшается (с 1522 до 1166 микросекунд), скорость работы обычного остается приблизительно той же(на графике значение равно 4231 микросекунде). Значит если массив почти отсортирован, то гибридная алгоритм будет всегда работать лучше, а при повышении threshold становится еще быстрее.