# АиСД-1: Set 3

Хромова Елизавета, группа БПИ2310

A2

**Реализация ArrayGenerator.**

```cpp
class ArrayGenerator {
public:
    ArrayGenerator() {
        initializeBaseArrays();
        generateSubArrays();
    }

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

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

    std::vector<std::vector<int>> getAlmostSorted() const {
        return almostSortedArrays;
    }

private:
    std::vector<std::vector<int>> randomArrays;
    std::vector<std::vector<int>> sortedArrays;
    std::vector<std::vector<int>> almostSortedArrays;
    std::mt19937 generator;

    void initializeBaseArrays()  {
        std::random_device rand_dev;
        generator = std::mt19937(rand_dev());
        std::uniform_int_distribution<> distr(1, 6000);

        randomArrays.emplace_back(generateRandomArray(10000, distr));
        sortedArrays.emplace_back(randomArrays[0]);
        almostSortedArrays.emplace_back(randomArrays[0]);

        std::stable_sort(sortedArrays[0].begin(), sortedArrays[0].end());
        reverseArray(sortedArrays[0]);

        std::stable_sort(almostSortedArrays[0].begin(), almostSortedArrays[0].end());
        reverseArray(almostSortedArrays[0]);
        introduceRandomSwaps(almostSortedArrays[0], distr, 10);
    }
```

```
48
49      void generateSubArrays() {
50          std::uniform_int_distribution<> distr(0, 10000 - 1);
51
52          for (int size = 500; size <= 10000; size += 100) {
53              int range = 10000 - size;
54              int start = (range > 0) ? distr(generator) % range : 0;
55
56              randomArrays.push_back(copySubArray(randomArrays[0], start, size));
57              sortedArrays.push_back(copySubArray(sortedArrays[0], start, size));
58              almostSortedArrays.push_back(copySubArray(almostSortedArrays[0], start, size));
59          }
60      }
61
62      std::vector<int> generateRandomArray(int size, std::uniform_int_distribution<> &distr) {
63          std::vector<int> array(size);
64          for (int &val : array) {
65              val = distr(generator);
66          }
67          return array;
68      }
69
70      void reverseArray(std::vector<int> &array)  {
71          std::reverse(array.begin(), array.end());
72      }
73
74      void introduceRandomSwaps(std::vector<int> &array, std::uniform_int_distribution<> &distr, int swaps) {
75          int n = array.size();
76          for (int i = 0; i < swaps; ++i) {
77              std::swap(array[distr(generator) % n], array[distr(generator) % n]);
78          }
79      }
80
81      std::vector<int> copySubArray(const std::vector<int> &source, int start, int size) {
82          return std::vector<int>(source.begin() + start, source.begin() + start + size);
83      }
84  };
```

**Реализация SortTester.**

```
86  class SortTester {
87  public:
88      long long testMergeSort(std::vector<int>& A) {
89          auto start = std::chrono::high_resolution_clock::now();
90          mergeSort(A, 0, static_cast<int>(A.size() - 1));
91          auto elapsed = std::chrono::high_resolution_clock::now() - start;
92          long long msec = std::chrono::duration_cast<std::chrono::microseconds>(elapsed).count();
93          return msec;
94      }
95
96      long long testModMergeSort(std::vector<int>& A) {
97          auto start = std::chrono::high_resolution_clock::now();
98          modMergeSort(A, 0, static_cast<int>(A.size() - 1));
99          auto elapsed = std::chrono::high_resolution_clock::now() - start;
00          long long msec = std::chrono::duration_cast<std::chrono::microseconds>(elapsed).count();
01          return msec;
02      }
03  };
```

Для генерации тестов создан класс ArrayGenerator, в котором создаются три группы массивов: с рандомными числами, отсортированный и почти отсортированный (отсортированный с несколькими поменяными числами).

Время выполнения сортировок mergeSort И merge+insertionSort замеряется в классе SortTester стандартным способом. Измерения времни проводились 20 раз, на графиках приведены усредненные результаты.

На каждом массиве тестируются обе сортировки, результат измерения времени записываются

в файл. На основе результатов получены графики ниже.

На графиках сравниваются время выполнения mergeSort и merge+insertionSort с указанным **над графиком** порогом перехода от insertion к merge.

### Almost

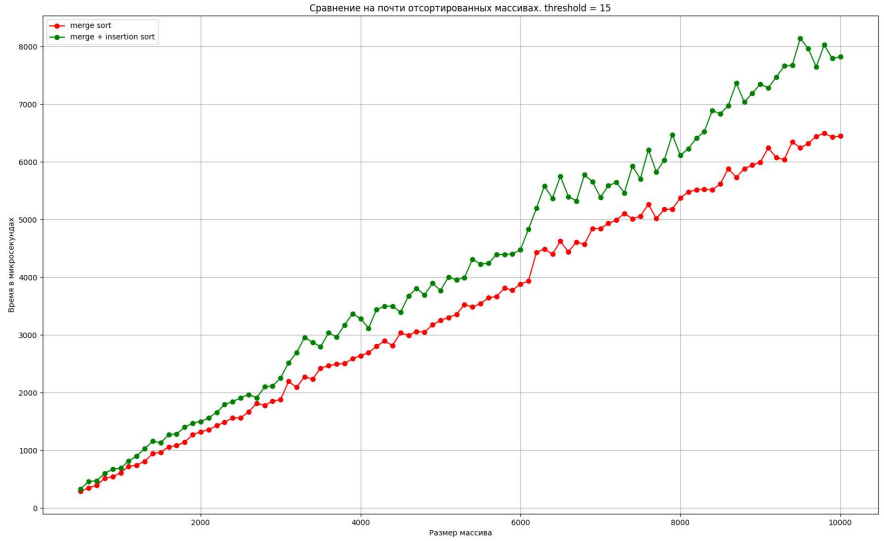


Рис. 1: threshold = 15

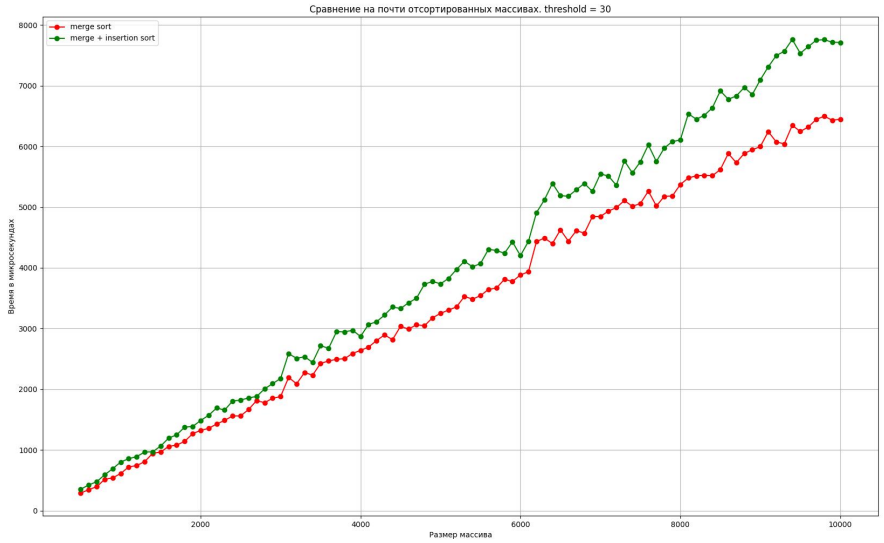


Рис. 2: threshold = 30

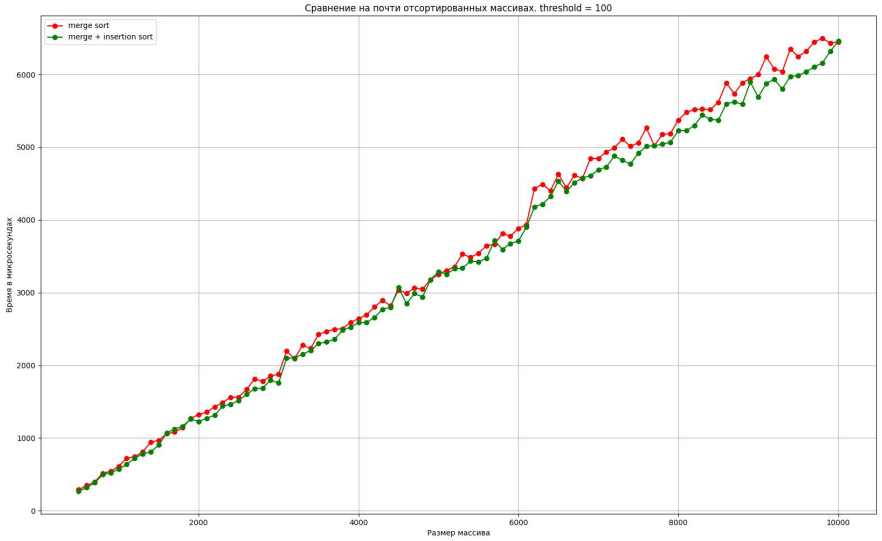


Рис. 3: threshold = 100


Рис. 4: threshold = 500

**Вывод:** в случае с почти отсортированными массивами гибридная сортировка оказалась в среднем эффективнее для всех рассмотернных значениях threshold, но разница гораздо меньше, чем в остальных случаях.

## Random

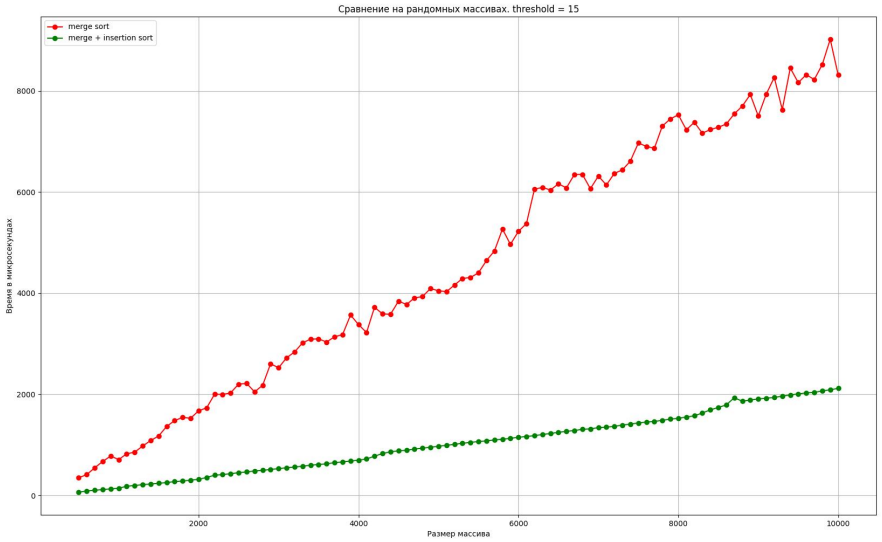


Рис. 5: threshold = 15

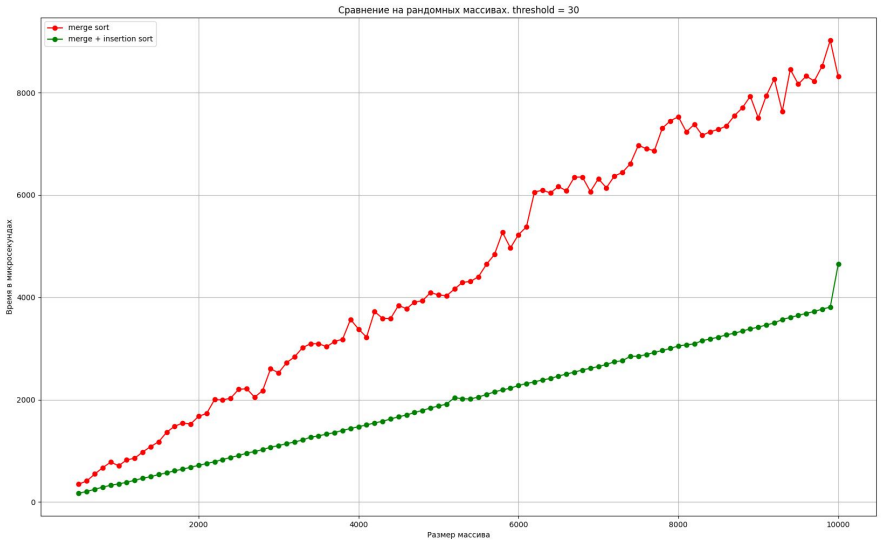


Рис. 6: threshold = 30


Рис. 7: threshold = 100

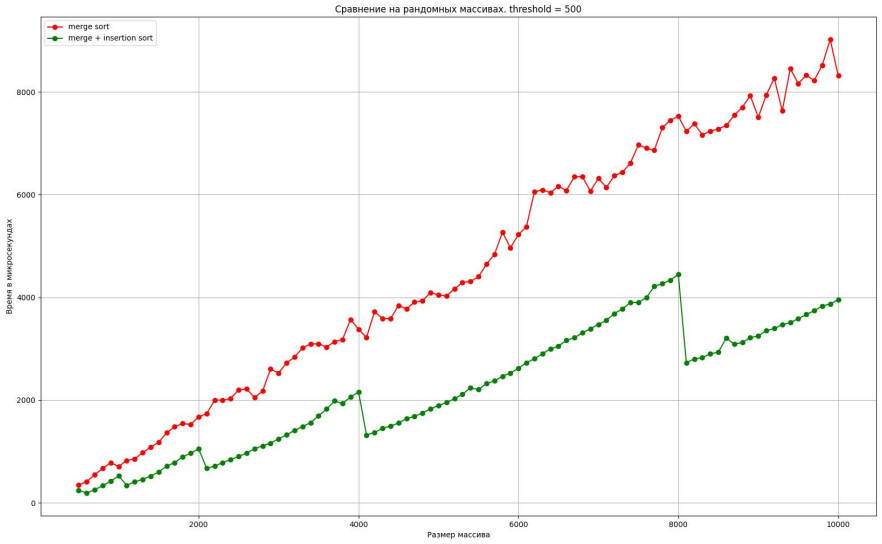


Рис. 8: threshold = 500

**Вывод:** в случае с рандомными массивами гибридная сортировка оказалась в среднем медленне для всех рассмотернных значениях threshold. На малых массивах разница незамента, но чем больше размер массива, тем существенне разница во времени между двумя сортировками.

## Sorted

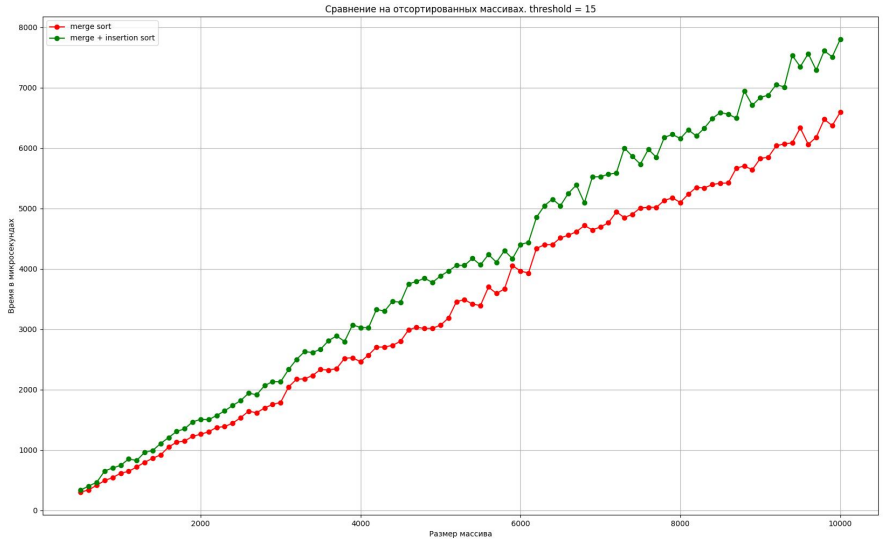


Рис. 9: threshold = 15

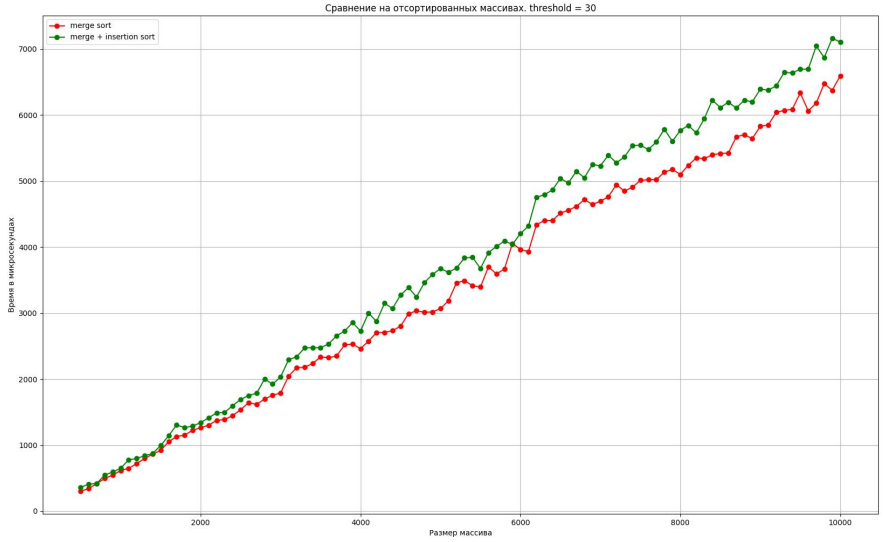


Рис. 10: threshold = 30

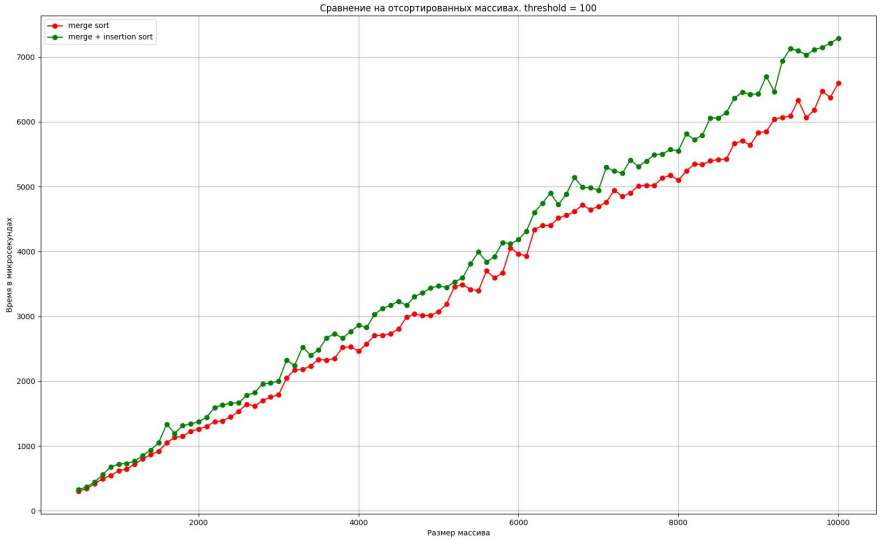


Рис. 11: threshold = 100


Рис. 12: threshold = 500

**Вывод:** в случае с отсортированными массивами гибридная сортировка оказалась в среднем быстрее для всех рассмотернных значениях threshold. Причем чем больше размер массива, тем эффективнее гибридная сортировка в сравнии с обычной merge.

**ID посылки задачи Ai2:** 293167572

**github:** Реализация алгоритмов задачи A2 и другие файлы по задаче тут