Исаев Кирилл БПИ2310
SET 3
Задание A3

ID посылки: [292683125]

[Ссылка на GitHub]

Реализация ArrayGenerator:

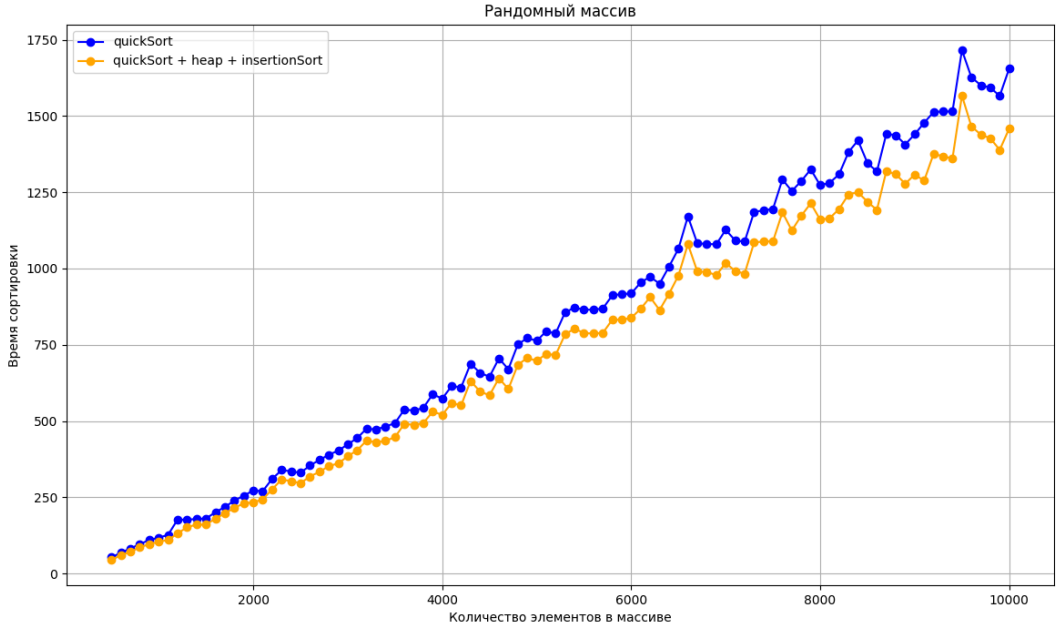
```
class ArrayGenerator {
public:
    ArrayGenerator() {
        randomArrays.resize( new_size: ARRAY_AMOUNT);
        sortedArrays.resize( new_size: ARRAY_AMOUNT);
        semiSortedArrays.resize( new_size: ARRAY_AMOUNT);
        fillArrays();
    }

    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:

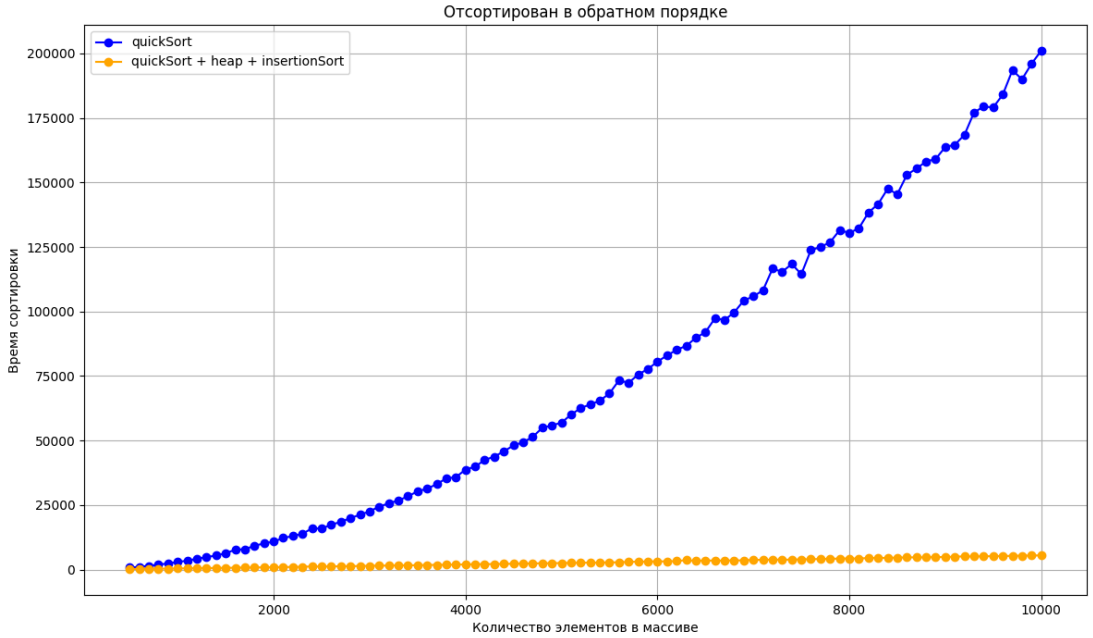
```
class SortTester {
public:
    long long Test(vector<int> array) {
        auto start :time_point<...> = std::chrono::high_resolution_clock::now();
        quickSort( &: array, left: 0, right: static_cast<int>(array.size()) - 1);
        auto elapsed :duration<...> = std::chrono::high_resolution_clock::now() - start;
        long long msec = std::chrono::duration_cast<std::chrono::microseconds>( d: elapsed).count();
        return msec;
    }
    long long TestHybrid(vector<int> array, int limit) {
        auto start :time_point<...> = std::chrono::high_resolution_clock::now();
        quickSortHybrid( &: array, lower: 0, higher: static_cast<int>(array.size()) - 1, limit);
        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(сгенерированы рандомные массивы)



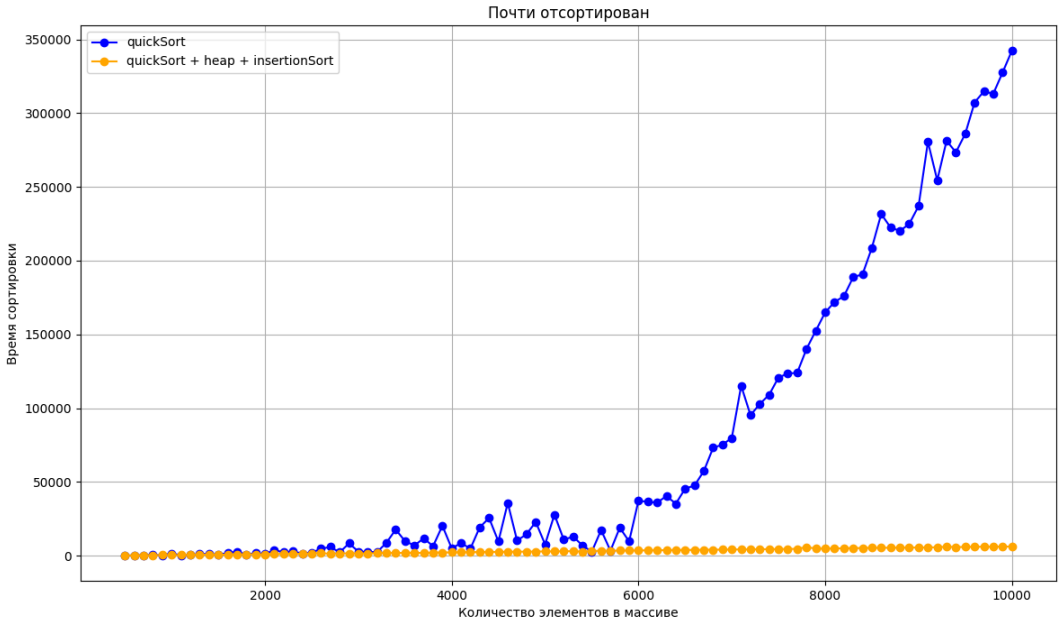
При случайно сгенерированных массивах разница в работе обычного и гибридного алгоритма почти не отличается(тем не менее, quickSort работает медленнее, чем его гибридная версия). Массив размера 10000 quickSort сортирует за 1656 микросекунд, quickSort + heapSort + insertionSort сортирует за 1459 микросекунд.

График№2(сгенерированы массивы, отсортированные в обратном порядке)



По данном графику видно, что гибридный алгоритм работает быстрее, чем обычная реализация quickSort, если массивы будут отсортированы в обратном порядке. При этом, скорость работы обычного алгоритма очень медленная - целых 201065 микросекунд чтобы отсортировать массив размера 10000, в то время, когда гибридный алгоритм этот же объем данных сортирует за 5487 микросекунд.
Следовательно не стоит использовать обычный quickSort в таком кейсе.

График№3(сгенерированы “почти” отсортированные массивы)



В случае “почти” отсортированных массивов скорость работы двух алгоритмов приблизительно равна до входных данных размера ~3000. Обычный алгоритм quickSort всегда случайно выбирает pivot, из-за чего на графике можно увидеть много резких скачков и резких падений(это следствие неудачного выбора pivot’а). Поэтому иногда даже на данных > 3000, обычный алгоритм может отрабатывать приблизительно с такой же скоростью как и гибридный алгоритм. Скорости алгоритмов начинают значительно отличаться при массивах размера ~6000 и больше. При этом скорость quickSort ОЧЕНЬ сильно падает при сортировке массивов размера ~6000 и больше, в то время, как гибридная сортировка ведет себя стабильно. Массив размера 10000 обычный алгоритм обработал за целых 342401 микросекунд, что является ужасным результатом. Гибрид же за 6192 микросекунд. Не стоит использовать quickSort в этом кейсе.