Диасамидзе Таир БПИ236

SET 3 Задача A2

ID посылки: 292081520

Ссылка на GitHub

Реализация класса ArrayGenerator

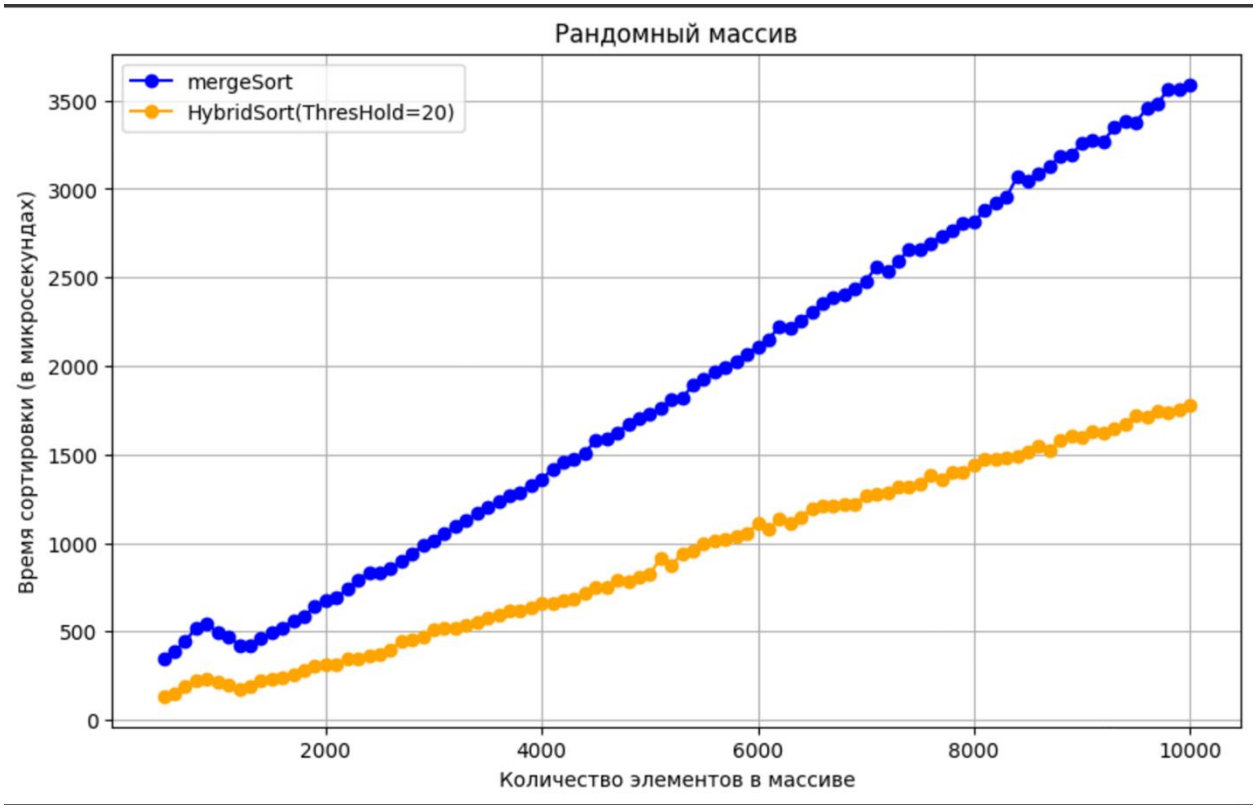
```cpp
#ifndef ARRAY_GENERATOR_H
#define ARRAY_GENERATOR_H

#include <vector>
#include <random>
#include <algorithm>
#include <iostream>

class ArrayGenerator {
    private:
        std::vector<int> array;
        std::mt19937 rng;

    public:
        ArrayGenerator() {
            std::random_device rd;
            rng = std::mt19937(rd());
            std::uniform_int_distribution<int> dist(0, 6000);

            array.resize(10000);
            for (int& num : array) {
                num = dist(rng);
            }
        }

        std::vector<int> getRandomArray(int size) {
            return std::vector<int>(array.begin(), array.begin() + size);
        }

        std::vector<int> getReversedArray(int size) {
            std::vector<int> array = getRandomArray(size);
            std::sort(array.rbegin(), array.rend());
            return array;
        }

        std::vector<int> getNearlySortedArray(int size, int swaps = 10) {
            std::vector<int> array = getRandomArray(size);
            std::sort(array.begin(), array.end());
```

```cpp
        std::vector<int> getNearlySortedArray(int size, int swaps = 10) {
            std::vector<int> array = getRandomArray(size);
            std::sort(array.begin(), array.end());

            for (int i = 0; i < swaps; ++i) {
                int idx1 = rng() % size;
                int idx2 = rng() % size;
                std::swap(array[idx1], array[idx2]);
            }

            return array;
        }

};

#endif // ARRAY_GENERATOR_H
```

Реализация класса SortTester

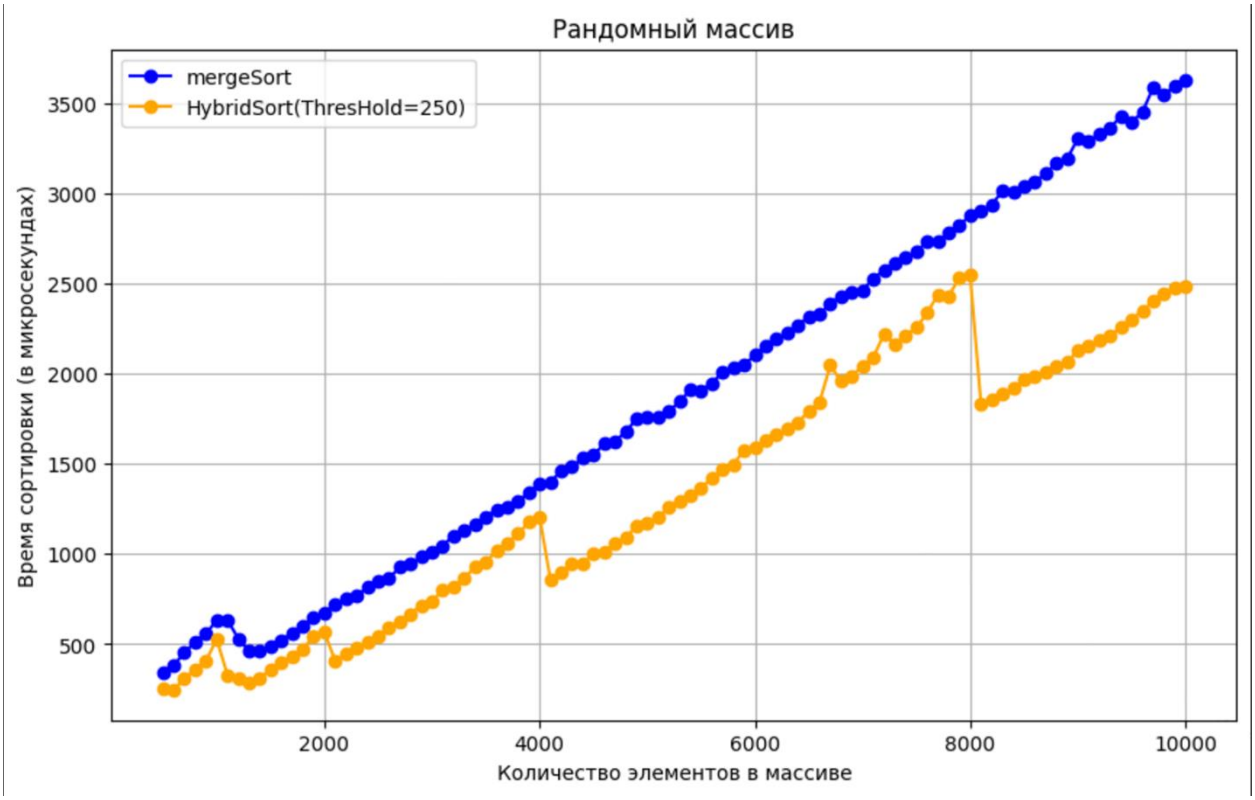
```cpp
#ifndef SORT_TESTER_H
#define SORT_TESTER_H

#include <chrono>
#include "ArrayGenerator.h"
#include "Algorithms.h"

class SortTester {
    public:
        long long measureMergeSort(std::vector<int> array) {
            auto start = std::chrono::high_resolution_clock::now();
            mergeSort(array, 0, array.size() - 1);
            auto elapsed = std::chrono::high_resolution_clock::now() - start;
            long long msec = std::chrono::duration_cast<std::chrono::microseconds>(elapsed).count();
            return msec;
        }

        long long measureHybridSort(std::vector<int> array, int threshold) {
            auto start = std::chrono::high_resolution_clock::now();
            hybridMergeSort(array, 0, array.size() - 1, threshold);
            auto elapsed = std::chrono::high_resolution_clock::now() - start;
            long long msec = std::chrono::duration_cast<std::chrono::microseconds>(elapsed).count();
            return msec;
        }
};

#endif // SORT_TESTER_H
```

График №1 - ThresHold = 20, сгенерирован рандомный массив



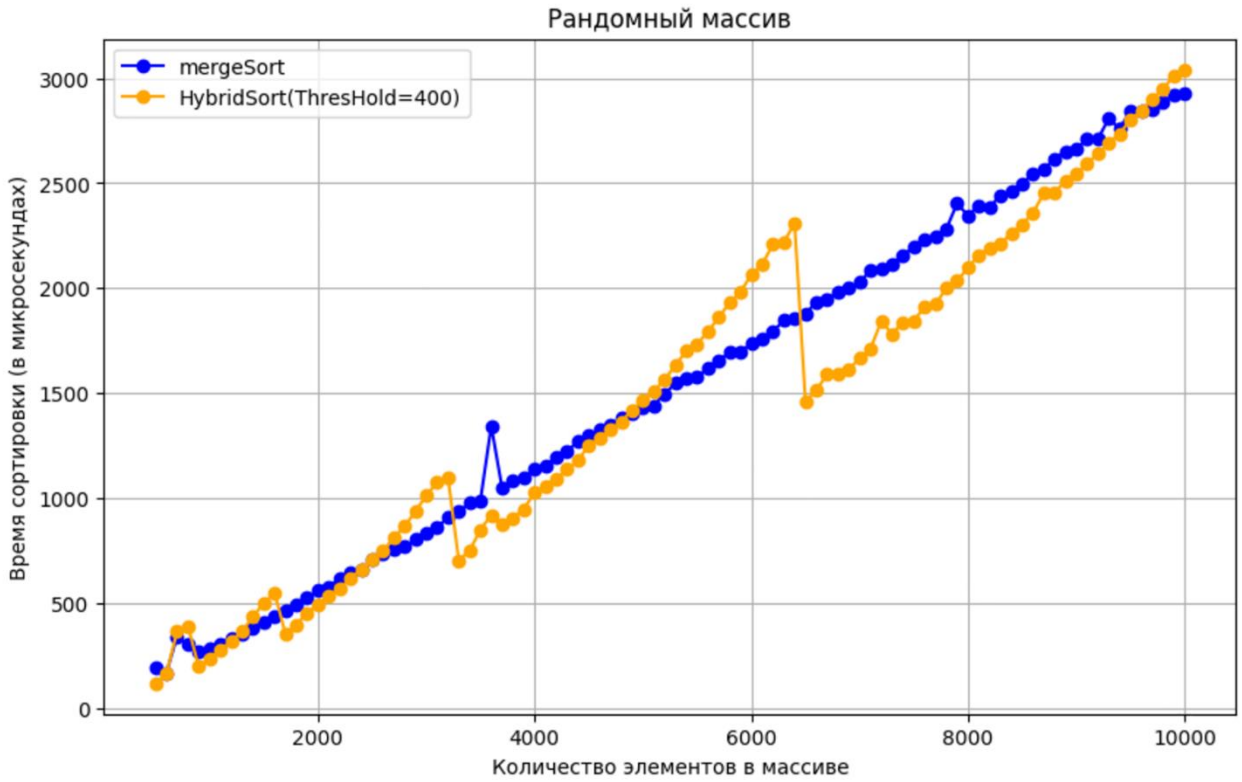
Заметим, что при ThresHold = 20 и малых входных данных алгоритмы сортировок достаточно отличаются по времени. Однако по увеличению входных данных HybridSort (merge + insertion) работает быстрее, чем MergeSort.

График №2 - ThresHold = 250, сгенерирован рандомный массив



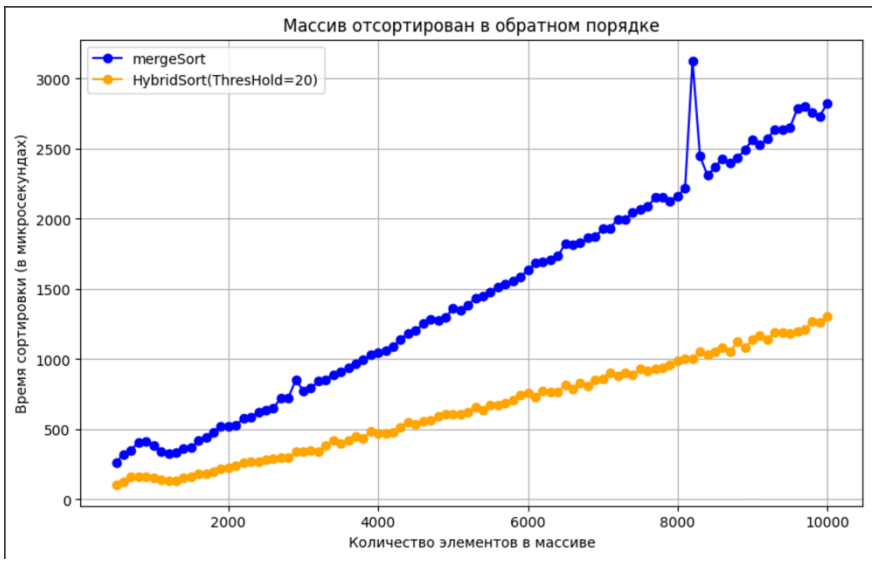
Заметим, что при ThresHold = 250 и малых входных данных размером до примерно 4000 алгоритмы сортировок не сильно отличаются по времени. Однако до сих пор HybridSort работает быстрее.

График №3 - ThresHold = 400, сгенерирован рандомный массив



Заметим, что при ThresHold = 400 HybridSort уже начинает иногда проигрывать по времени. Поэтому при такой константе лучше стоит использовать обычный MergeSort.

График №4 - ThresHold = 20, сгенерированы отсортированные массивы в обратном порядке

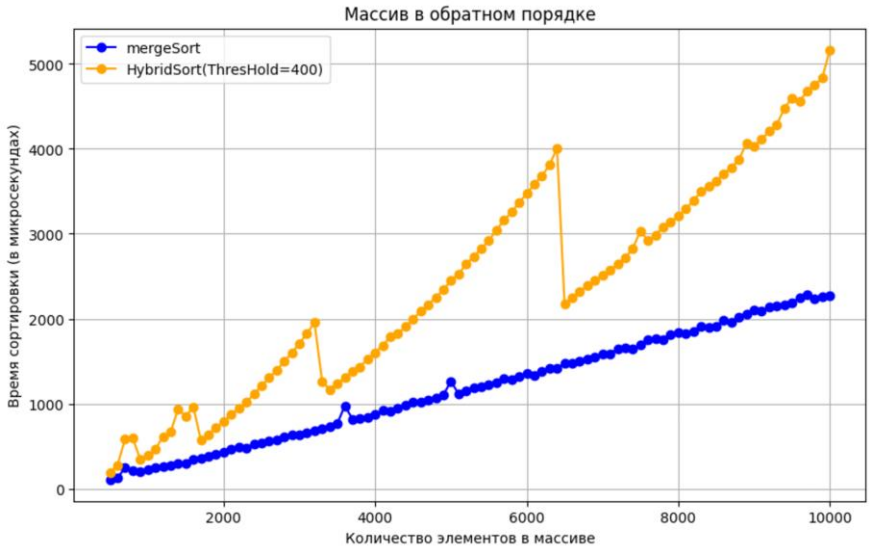


Можно сказать, что ситуация похоже на то, как на графике №1: HybridSort работает быстрее, чем MergeSort.

График №5 - ThresHold = 250, сгенерированы отсортированные массивы в обратном порядке

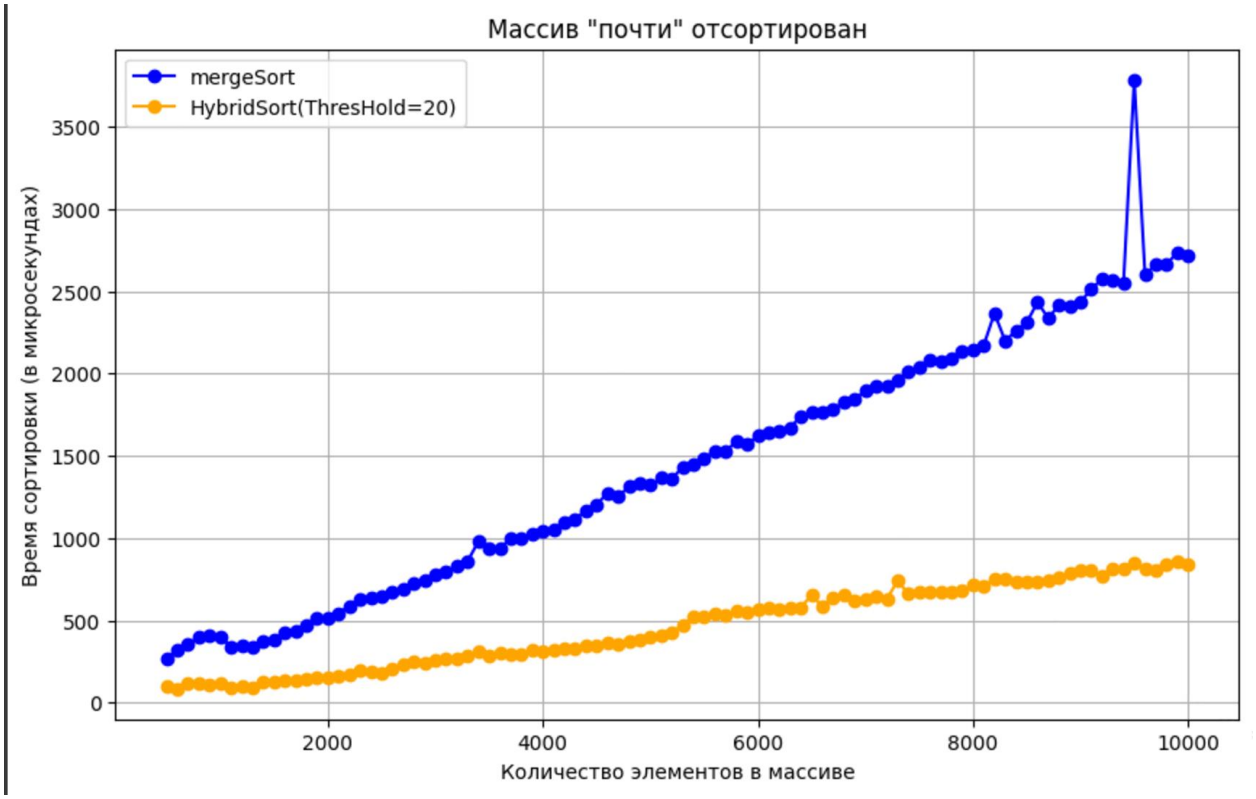
Можно заметить, что гибридный алгоритм при малых входных данных иногда проигрывает обычному MergeSort, а дальше – полностью проигрывает обычному MergeSort. Также HybridSort работает медленнее, чем при ThresHold = 20.

График №6 - ThresHold = 400, сгенерированы отсортированные массивы в обратном порядке



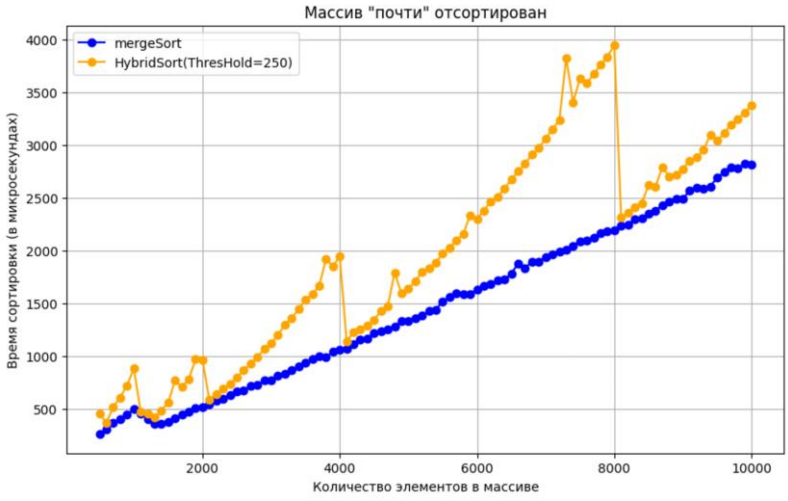
Здесь можно только сказать, что HybridSort полностью проигрывает обычному MergeSort. Поэтому не стоит использовать гибридный алгоритм.

График №7 - ThresHold = 20, сгенерированы «почти» отсортированные массивы



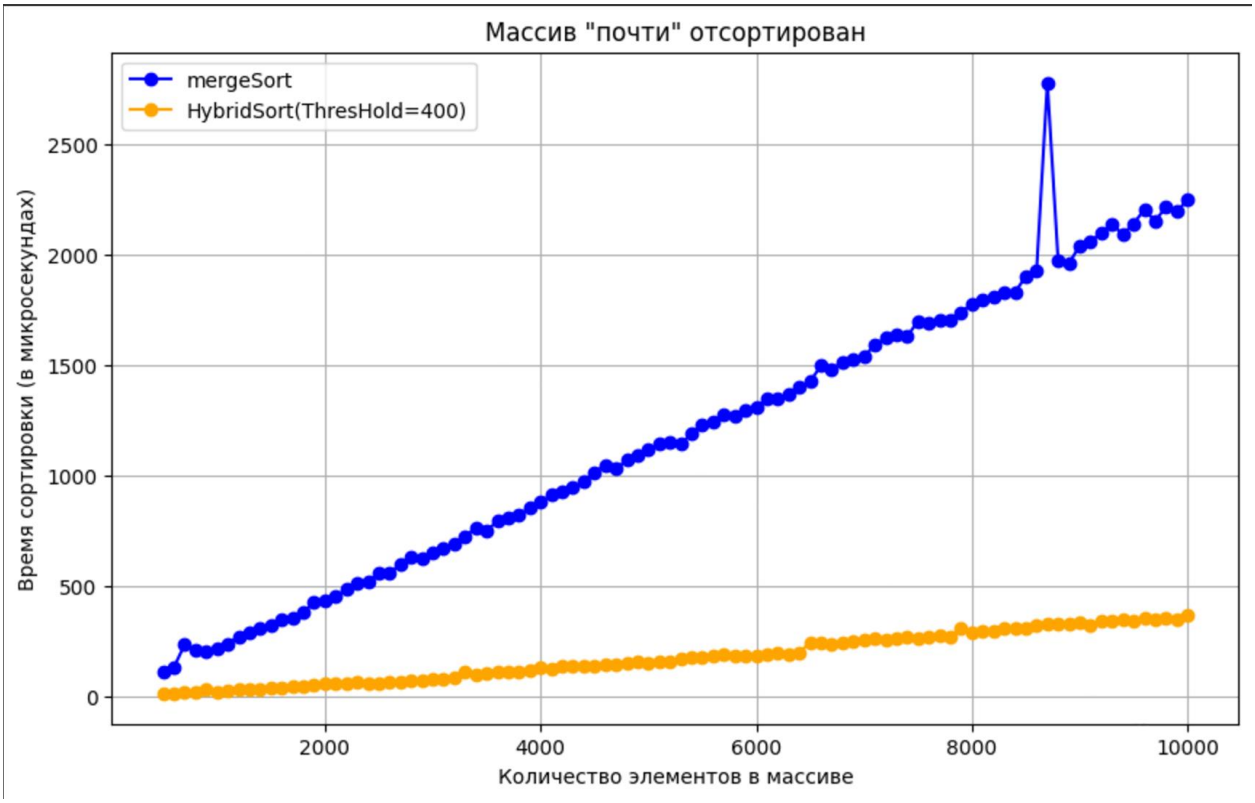
Можно сказать, что при таких вводных данных оба алгоритма работают быстро, однако гибридный алгоритм работает намного быстрее, чем MergeSort.

График №7 - ThresHold = 250, сгенерированы «почти» отсортированные массивы



Заметим, что при ThresHold = 250 гибридный алгоритм проигрывает обычному MergeSort.

График №7 - ThresHold = 400, сгенерированы «почти» отсортированные массивы



Здесь же ситуация другая: при ThresHold = 400 гибридный алгоритм работает намного быстрее. Можно предположить, что при повышении константы гибридный алгоритм будет работать еще быстрее.