

ID ссылки: 292081520

[Ссылка на GitHub](#)

Реализация класса ArrayGenerator

```

1  #ifndef ARRAY_GENERATOR_H
2  #define ARRAY_GENERATOR_H
3
4  #include <vector>
5  #include <random>
6  #include <algorithm>
7  #include <iostream>
8
9  class ArrayGenerator {
10 private:
11     std::vector<int> array;
12     std::mt19937 rng;
13
14 public:
15     ArrayGenerator() {
16         std::random_device rd;
17         rng = std::mt19937(rd());
18         std::uniform_int_distribution<int> dist(0, 6000);
19
20         array.resize(10000);
21         for (int& num : array) {
22             num = dist(rng);
23         }
24     }
25
26     std::vector<int> getRandomArray(int size) {
27         return std::vector<int>(array.begin(), array.begin() + size);
28     }
29
30     std::vector<int> getReversedArray(int size) {
31         std::vector<int> array = getRandomArray(size);
32         std::sort(array.rbegin(), array.rend());
33         return array;
34     }
35
36     std::vector<int> getNearlySortedArray(int size, int swaps = 10) {
37         std::vector<int> array = getRandomArray(size);
38         std::sort(array.begin(), array.end());

```

```

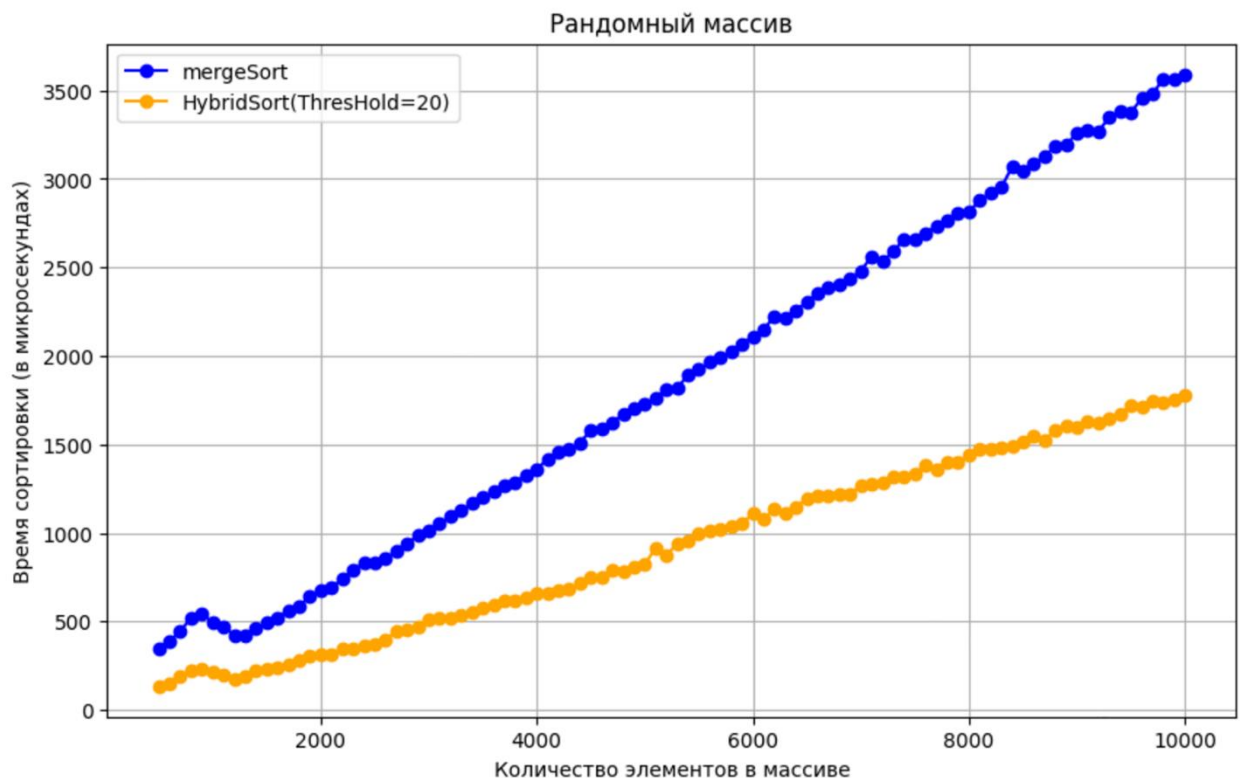
35
36         std::vector<int> getNearlySortedArray(int size, int swaps = 10) {
37             std::vector<int> array = getRandomArray(size);
38             std::sort(array.begin(), array.end());
39
40             for (int i = 0; i < swaps; ++i) {
41                 int idx1 = rng() % size;
42                 int idx2 = rng() % size;
43                 std::swap(array[idx1], array[idx2]);
44             }
45
46             return array;
47         }
48
49     };
50
51 #endif // ARRAY_GENERATOR_H
52

```

Реализация класса SortTester

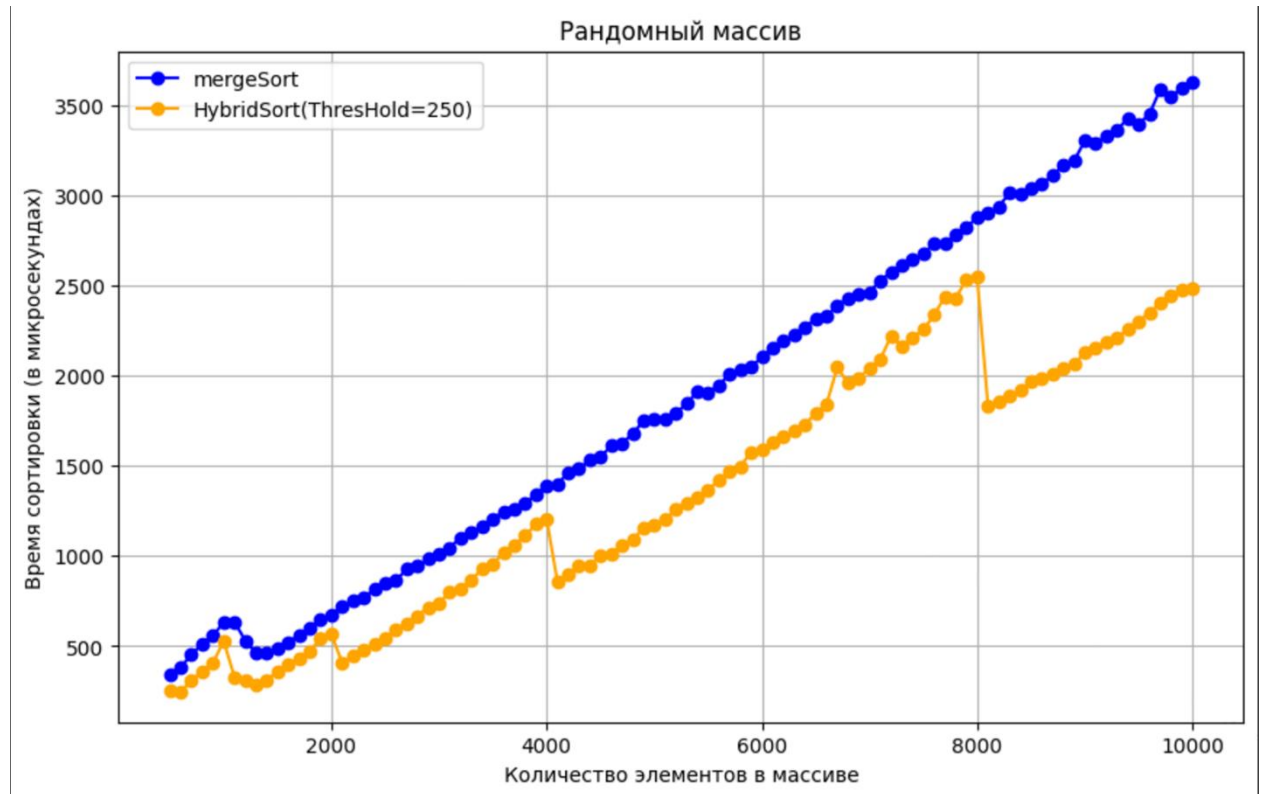
```
1  #ifndef SORT_TESTER_H
2  #define SORT_TESTER_H
3
4  #include <chrono>
5  #include "ArrayGenerator.h"
6  #include "Algorithms.h"
7
8  class SortTester {
9  public:
10     Long Long measureMergeSort(std::vector<int> array) {
11         auto start = std::chrono::high_resolution_clock::now();
12         mergeSort(array, 0, array.size() - 1);
13         auto elapsed = std::chrono::high_resolution_clock::now() - start;
14         Long Long msec = std::chrono::duration_cast<std::chrono::microseconds>(elapsed).count();
15         return msec;
16     }
17
18     Long Long measureHybridSort(std::vector<int> array, int threshold) {
19         auto start = std::chrono::high_resolution_clock::now();
20         hybridMergeSort(array, 0, array.size() - 1, threshold);
21         auto elapsed = std::chrono::high_resolution_clock::now() - start;
22         Long Long msec = std::chrono::duration_cast<std::chrono::microseconds>(elapsed).count();
23         return msec;
24     }
25 };
26
27 #endif // SORT_TESTER_H
```

График №1 - ThresHold = 20, сгенерирован случайный массив



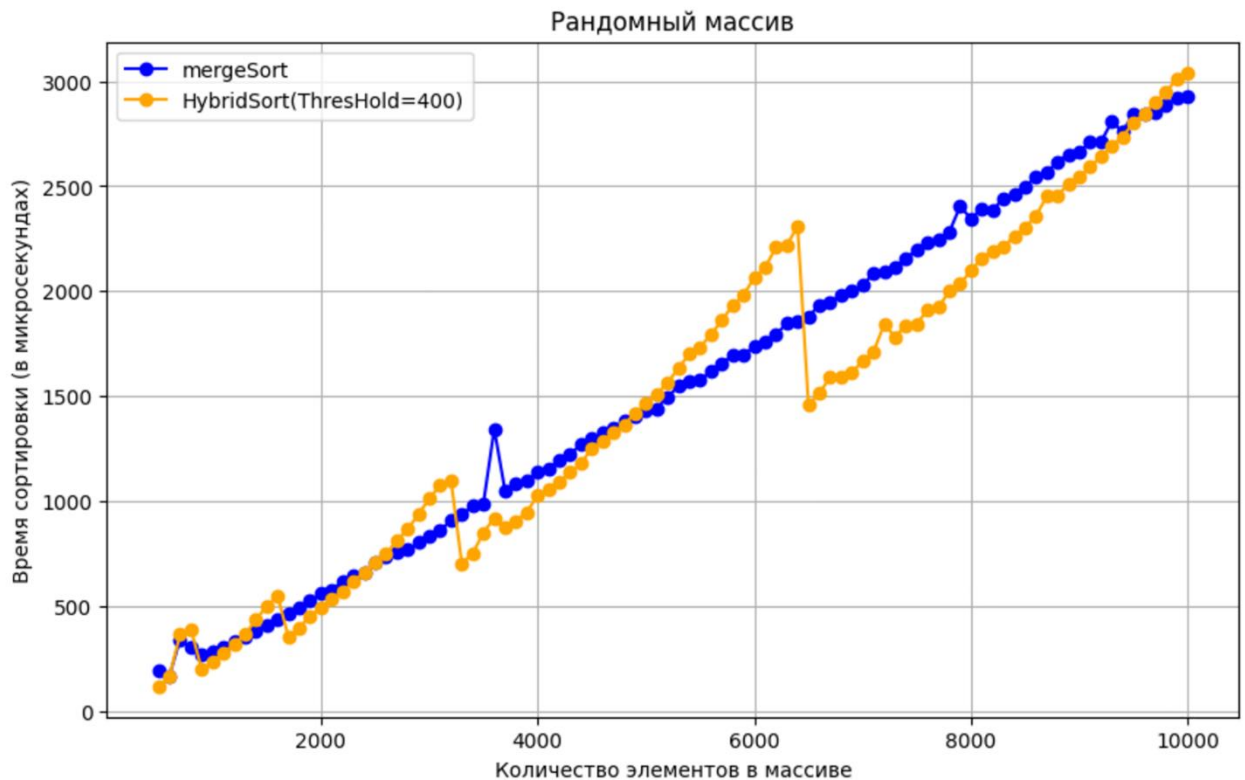
Заметим, что при $\text{ThresHold} = 20$ и малых входных данных алгоритмы сортировок достаточно отличаются по времени. Однако по увеличению входных данных HybridSort (merge + insertion) работает быстрее, чем MergeSort.

График №2 - $\text{ThresHold} = 250$, сгенерирован случайный массив



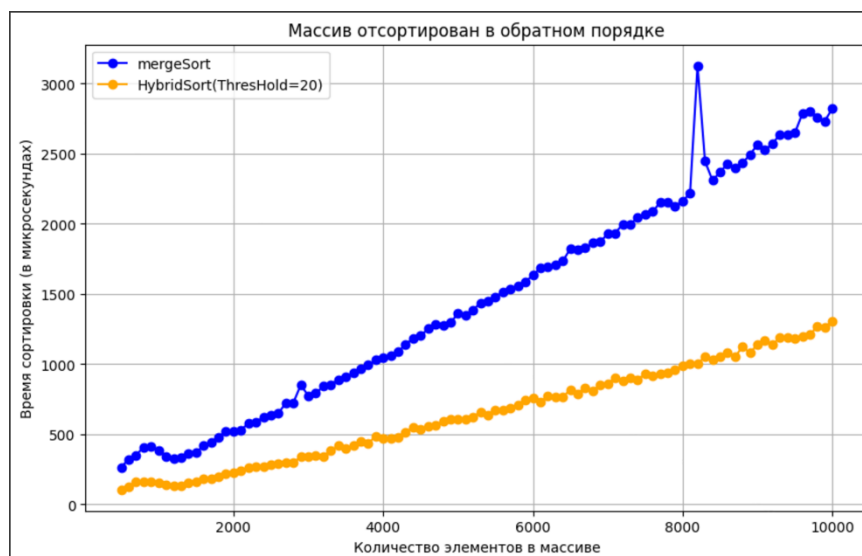
Заметим, что при $\text{ThresHold} = 250$ и малых входных данных размером до примерно 4000 алгоритмы сортировок не сильно отличаются по времени. Однако до сих пор HybridSort работает быстрее.

График №3 - ThresHold = 400, сгенерирован случайный массив



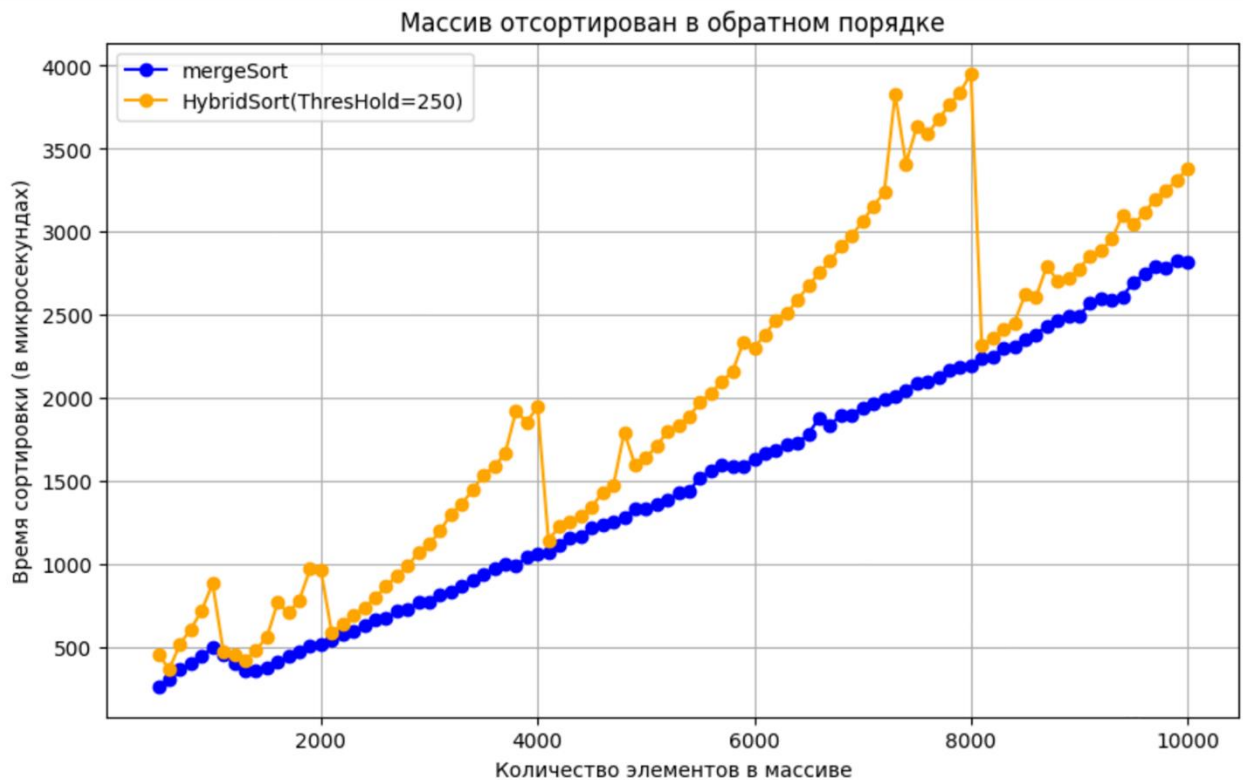
Заметим, что при ThresHold = 400 HybridSort уже начинает иногда проигрывать по времени. Поэтому при такой константе лучше стоит использовать обычный MergeSort.

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



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

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



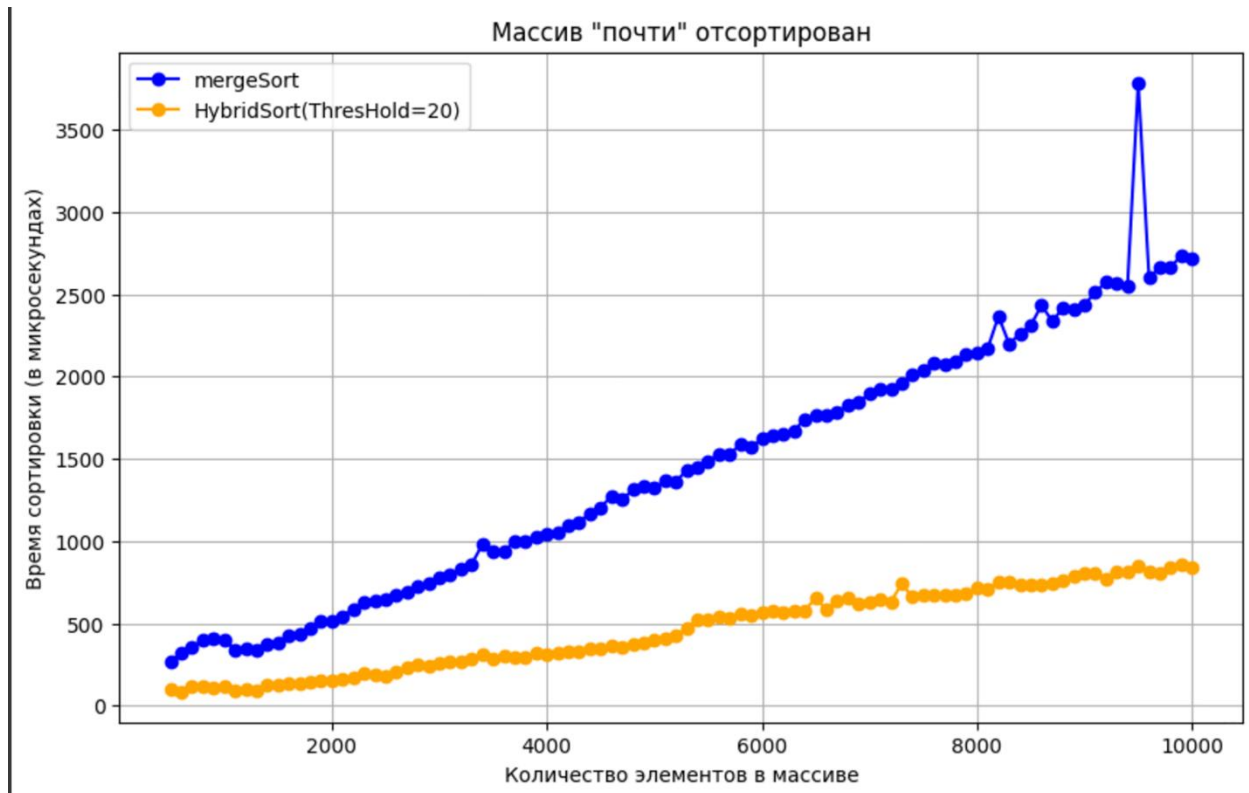
Можно заметить, что гибридный алгоритм при малых входных данных иногда проигрывает обычному MergeSort, а дальше – полностью проигрывает обычному MergeSort. Также HybridSort работает медленнее, чем при ThresHold = 20.

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



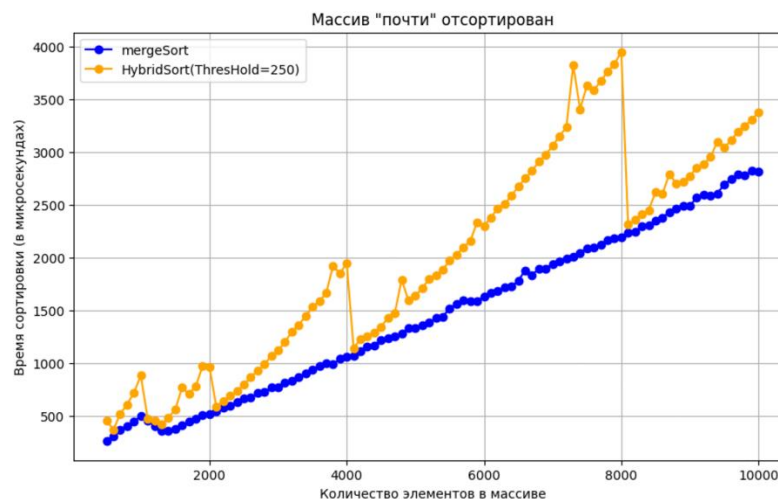
Здесь можно только сказать, что HybridSort полностью проигрывает обычному MergeSort. Поэтому не стоит использовать гибридный алгоритм.

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



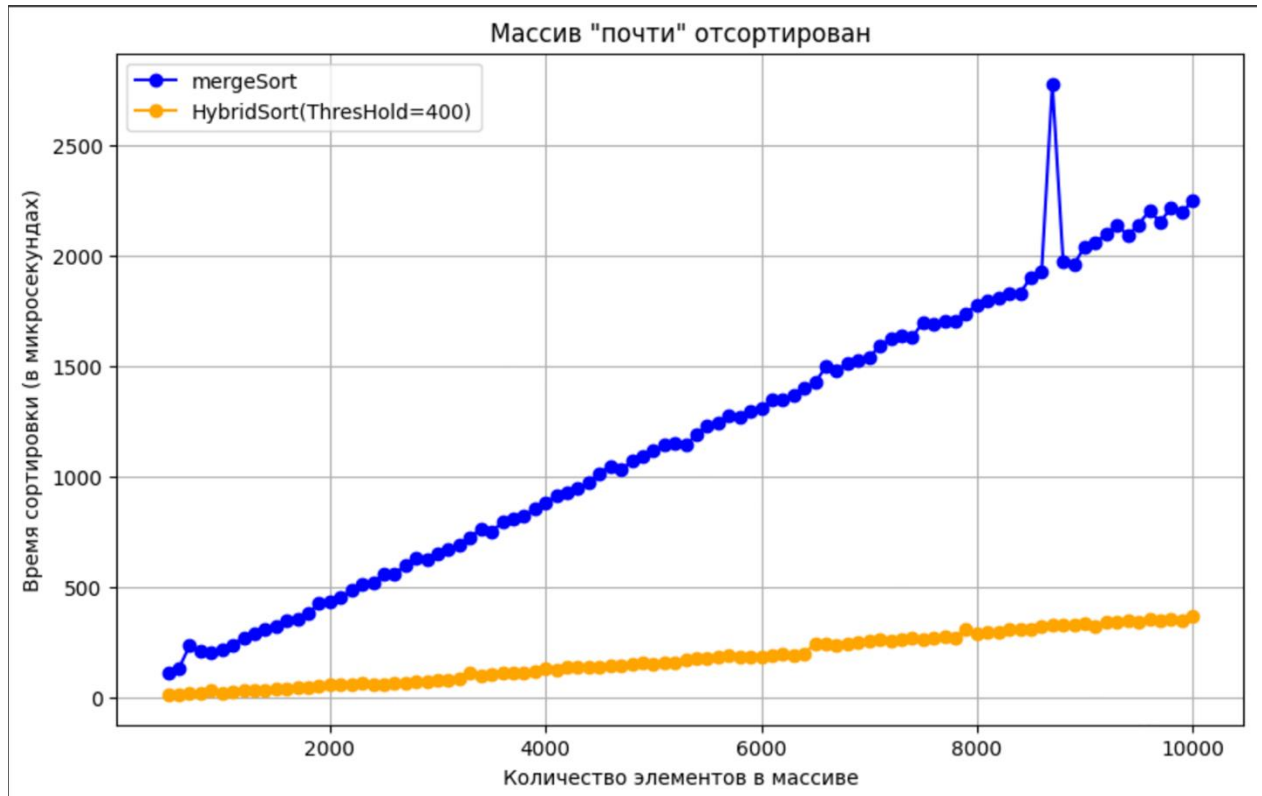
Можно сказать, что при таких вводных данных оба алгоритма работают быстро, однако гибридный алгоритм работает намного быстрее, чем MergeSort.

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



Заметим, что при $\text{ThresHold} = 250$ гибридный алгоритм проигрывает обычному MergeSort.

График №7 - $\text{ThresHold} = 400$, сгенерированы «почти» отсортированные массивы



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