Задача A2. Анализ MERGE+INSERTION SORT

id посылки: 349348328
github: https://github.com/kurokolover/algorithms_hm/tree/main/A2_solution

1 Модификация алгоритма MERGE SORT

Добавим к стандартной реализации сортировки слиянием дополнительный параметр `threshold`, задающий длину подмассива, при которой рекурсивный вызов заменяется сортировкой вставками. При `threshold = 0` алгоритм совпадает с обычной сортировкой слиянием.

1.1 Сортировка вставками

```
void insertion_sort(std::vector<long long>& a, int l, int r) {
    for (int i = l + 1; i < r; ++i) {
        long long key = a[i];
        int j = i - 1;
        while (j >= l && a[j] > key) {
            a[j + 1] = a[j];
            --j;
        }
        a[j + 1] = key;
    }
}
```

1.2 Стандартная сортировка слиянием

```
void merge_sort(std::vector<long long>& a,
                std::vector<long long>& tmp,
                int l, int r) {
    if (r - l <= 1) return;
    int m = l + (r - l) / 2;
    merge_sort(a, tmp, l, m);
    merge_sort(a, tmp, m, r);
    int i = l, j = m, k = l;
    while (i < m && j < r) {
        if (a[i] <= a[j]) tmp[k++] = a[i++];
        else tmp[k++] = a[j++];
    }
    while (i < m) tmp[k++] = a[i++];
    while (j < r) tmp[k++] = a[j++];
    for (int t = l; t < r; ++t) a[t] = tmp[t];
}
```

1.3 Гибридный алгоритм MERGE+INSERTION SORT

```
void hybrid_merge_sort(std::vector<long long>& a,
                       std::vector<long long>& tmp,
                       int l, int r,
                       int threshold) {
```

```
    int len = r - l;
    if (len <= 1) return;
    if (len <= threshold) {
        insertion_sort(a, l, r);
        return;
    }
    int m = l + len / 2;
    hybrid_merge_sort(a, tmp, l, m, threshold);
    hybrid_merge_sort(a, tmp, m, r, threshold);
    int i = l, j = m, k = l;
    while (i < m && j < r) {
        if (a[i] <= a[j]) tmp[k++] = a[i++];
        else tmp[k++] = a[j++];
    }
    while (i < m) tmp[k++] = a[i++];
    while (j < r) tmp[k++] = a[j++];
    for (int t = l; t < r; ++t) a[t] = tmp[t];
}
```

В задаче A2i использовалось значение `threshold = 15`. В рамках анализа дополнительно рассматривались значения

$$\texttt{threshold} \in \{0, 5, 10, 15, 30, 50\}.$$

2 Подготовка тестовых данных

2.1 Базовые массивы

Для уменьшения влияния генерации на результаты и для удобства повторных измерений создаются три базовых массива максимальной длины $N_{\max} = 100000$, заполненные целыми числами:

1. массив случайных значений в диапазоне $[0, 10000]$;
2. массив, содержащий те же элементы, отсортированные по невозрастанию;
3. массив, отсортированный по возрастанию, в котором затем случайным образом переставлена часть элементов (почти отсортированный массив).

2.2 Генерация массивов нужного размера

Для теста алгоритма на массиве длины n и заданного типа выбирается случайный отрезок длины n из соответствующего базового массива. Это позволяет получать массивы разных размеров, сохраняя общую структуру данных.

```
enum class ArrayType { Random, Reversed, AlmostSorted };

class ArrayGenerator {
    std::mt19937_64 rng;
    std::vector<long long> base_random;
    std::vector<long long> base_reversed;
    std::vector<long long> base_almost;
public:
    ArrayGenerator(int max_n = 100000,
```

```
                    long long lo = 0,
                    long long hi = 10000)
        : rng(std::random_device{}()) {
        std::uniform_int_distribution<long long> dist(lo, hi);
        base_random.resize(max_n);
        for (int i = 0; i < max_n; ++i) base_random[i] = dist(rng);
        base_reversed = base_random;
        std::sort(base_reversed.begin(), base_reversed.end(),
                  std::greater<long long>());
        base_almost = base_random;
        std::sort(base_almost.begin(), base_almost.end());
        int swaps = static_cast<int>(0.05 * max_n);
        std::uniform_int_distribution<int> pos(0, max_n - 1);
        for (int i = 0; i < swaps; ++i) {
            int x = pos(rng);
            int y = pos(rng);
            std::swap(base_almost[x], base_almost[y]);
        }
    }

    std::vector<long long> get_array(int n, ArrayType type) {
        const std::vector<long long>* src = nullptr;
        if (type == ArrayType::Random) src = &base_random;
        else if (type == ArrayType::Reversed) src = &base_reversed;
        else src = &base_almost;
        int max_start = static_cast<int>(src->size()) - n;
        std::uniform_int_distribution<int> start_dist(0, max_start);
        int start = start_dist(rng);
        return std::vector<long long>(src->begin() + start,
                                      src->begin() + start + n);
    }
};
```

3 Инфраструктура измерений

3.1 Класс `SortTester`

Измерение времени работы алгоритмов выполняется с помощью стандартной библиотеки `std::chrono`. Для уменьшения случайных колебаний каждый эксперимент повторяется несколько раз, после чего вычисляется среднее время.

```
struct SortResult {
    int n;
    double ms;
};

class SortTester {
    int trials;
public:
    explicit SortTester(int trials_ = 500) : trials(trials_) {}

    template <typename SortFunc>
    std::vector<SortResult> run(ArrayGenerator& gen,
```

```
                                ArrayType type,
                                SortFunc sort_func,
                                int n_min,
                                int n_max,
                                int step) {
        std::vector<SortResult> res;
        for (int n = n_min; n <= n_max; n += step) {
            double sum_ms = 0.0;
            for (int t = 0; t < trials; ++t) {
                auto a = gen.get_array(n, type);
                std::vector<long long> tmp(n);
                auto start =
                    std::chrono::high_resolution_clock::now();
                sort_func(a, tmp);
                auto elapsed =
                    std::chrono::high_resolution_clock::now() - start;
                double ms =
                    std::chrono::duration_cast<
                        std::chrono::microseconds>(elapsed).count()
                    / 1000.0;
                sum_ms += ms;
            }
            res.push_back({n, sum_ms / trials});
        }
        return res;
    }
};
```

В экспериментах параметры были выбраны так:

- длина массивов: от 500 до 10000 с шагом 100;
- количество прогонов для каждого размера и конфигурации: 500;
- значения `threshold`: $0, 5, 10, 15, 30, 50$;
- типы данных: случайные, обратно отсортированные и почти отсортированные массивы.

4 Результаты

Порог threshold = 0

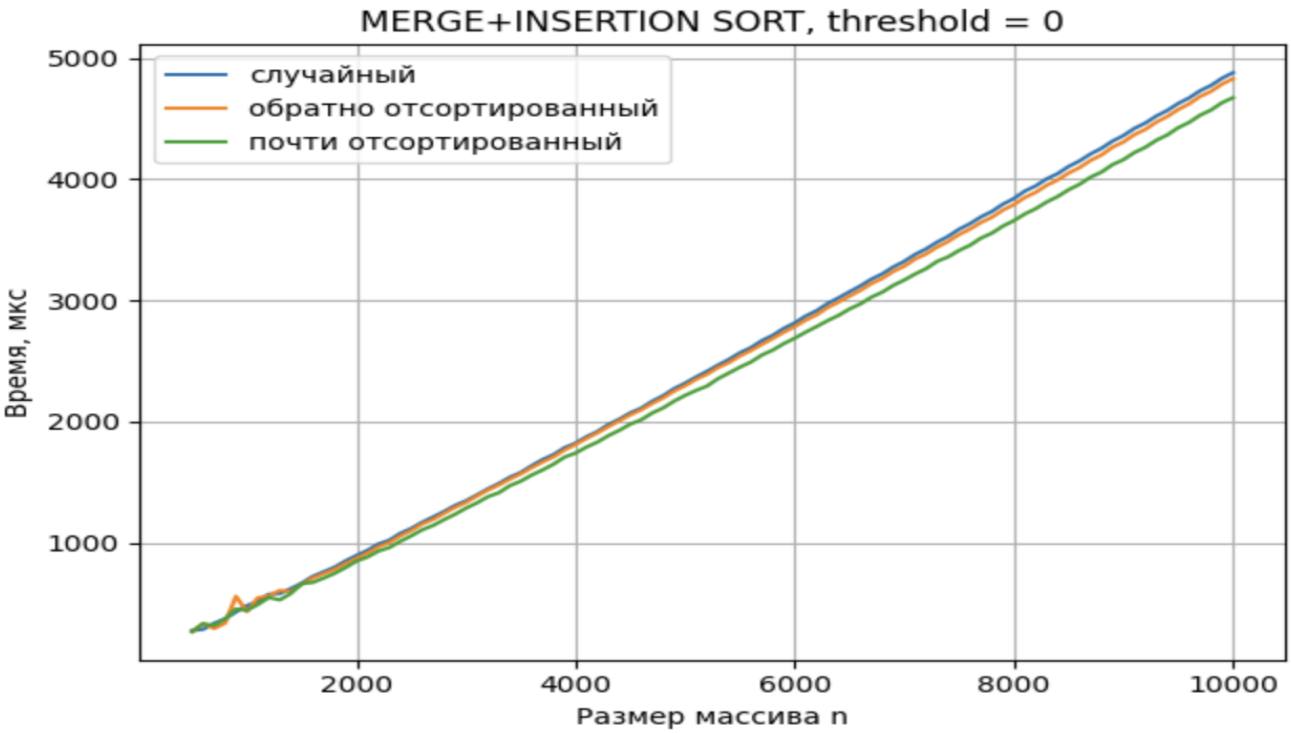

Рис. 1: Время работы сортировки слиянием при `threshold = 0`

Порог threshold = 5

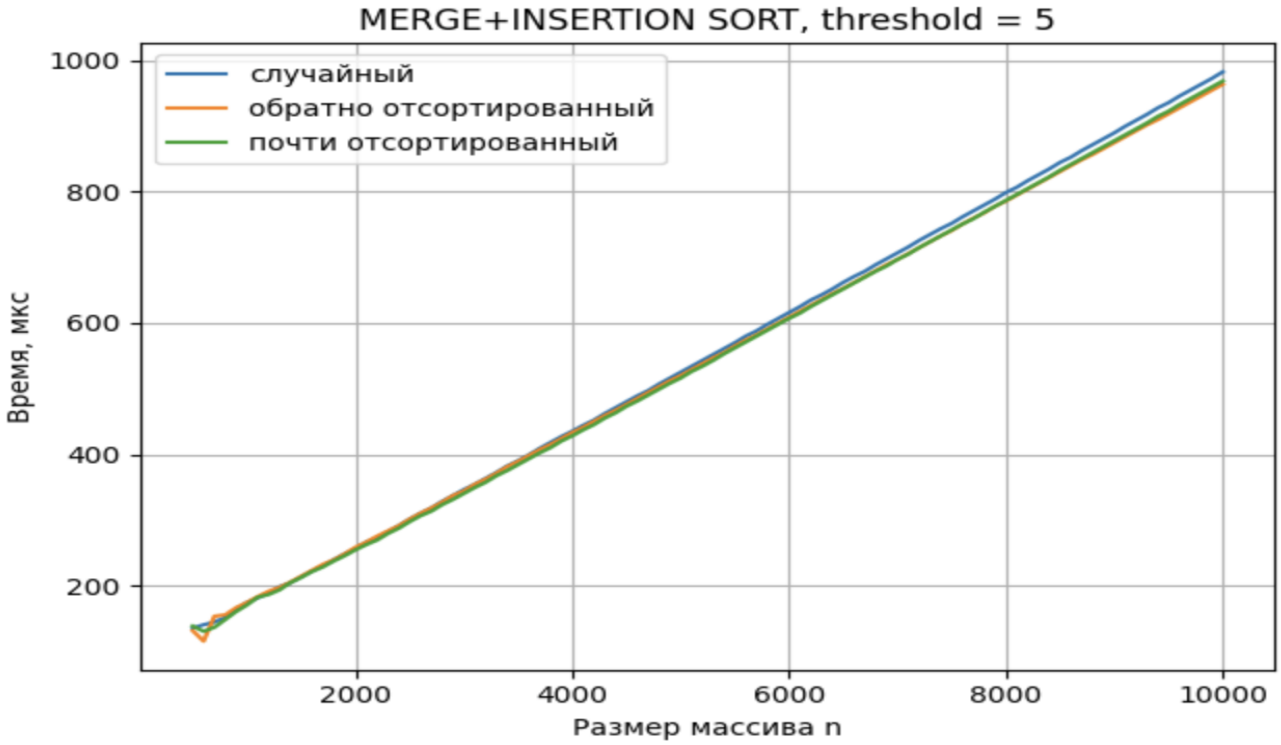

Рис. 2: Время работы гибридной сортировки при `threshold = 5`

Порог threshold = 10

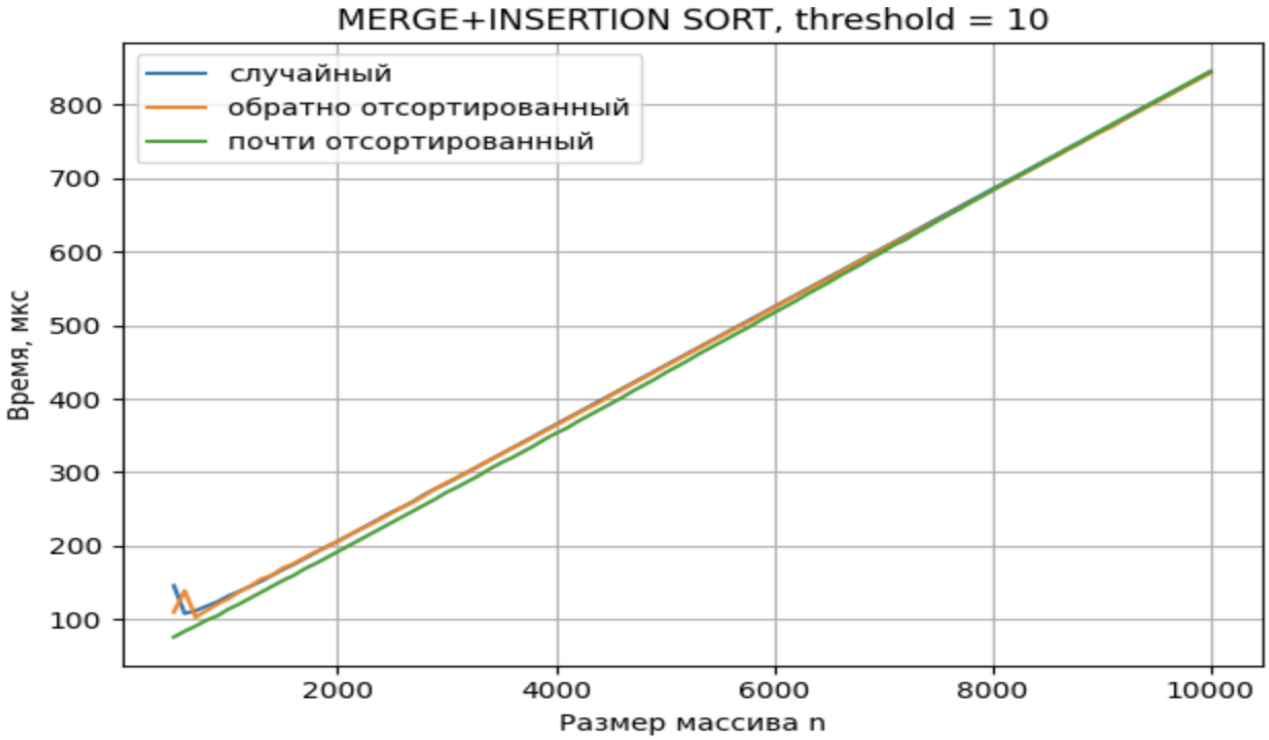

Рис. 3: Время работы гибридной сортировки при `threshold = 10`

Порог threshold = 15

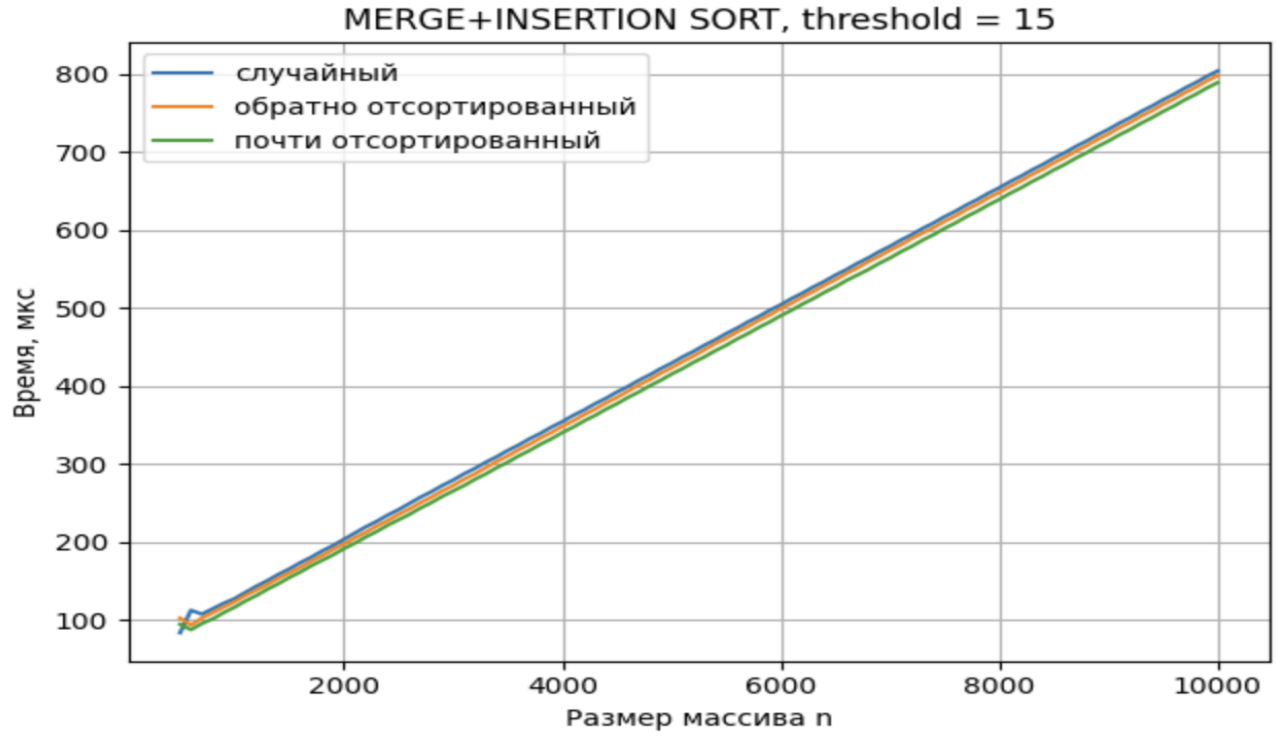

Рис. 4: Время работы гибридной сортировки при `threshold = 15`

Порог threshold = 30

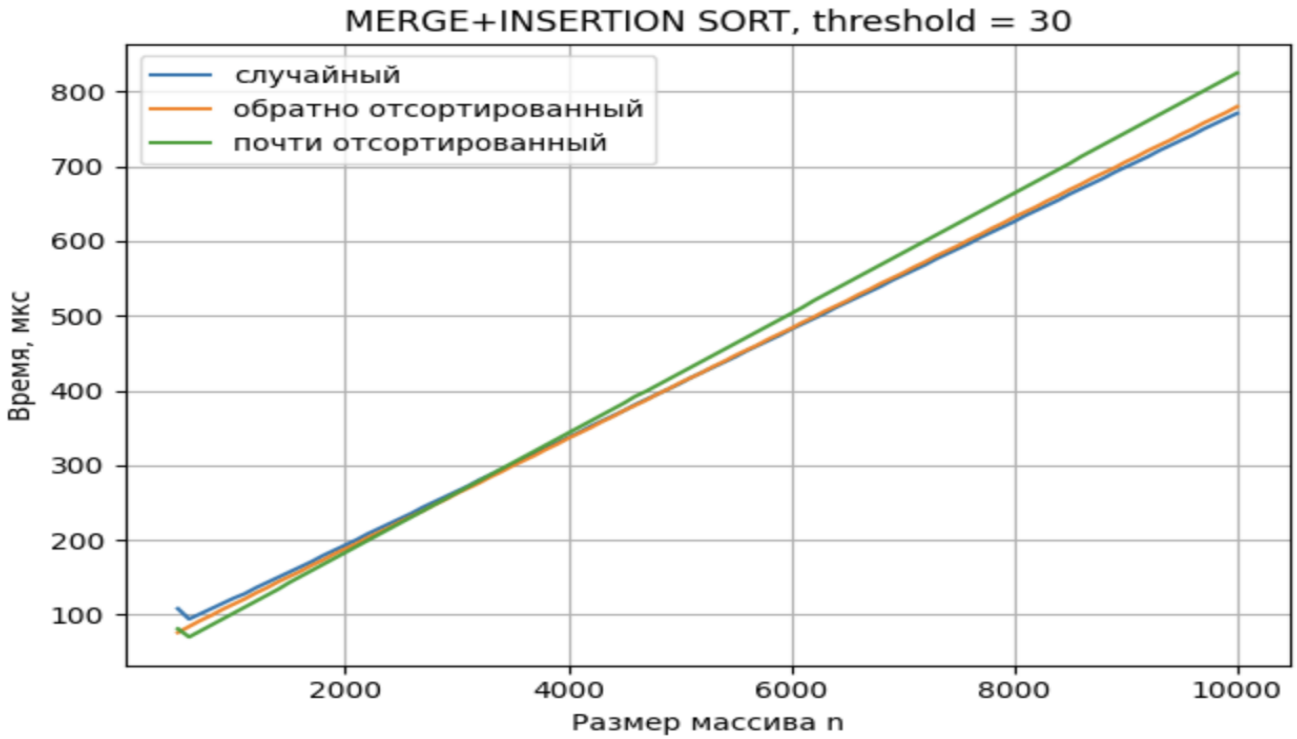

Рис. 5: Время работы гибридной сортировки при `threshold = 30`

Порог threshold = 50

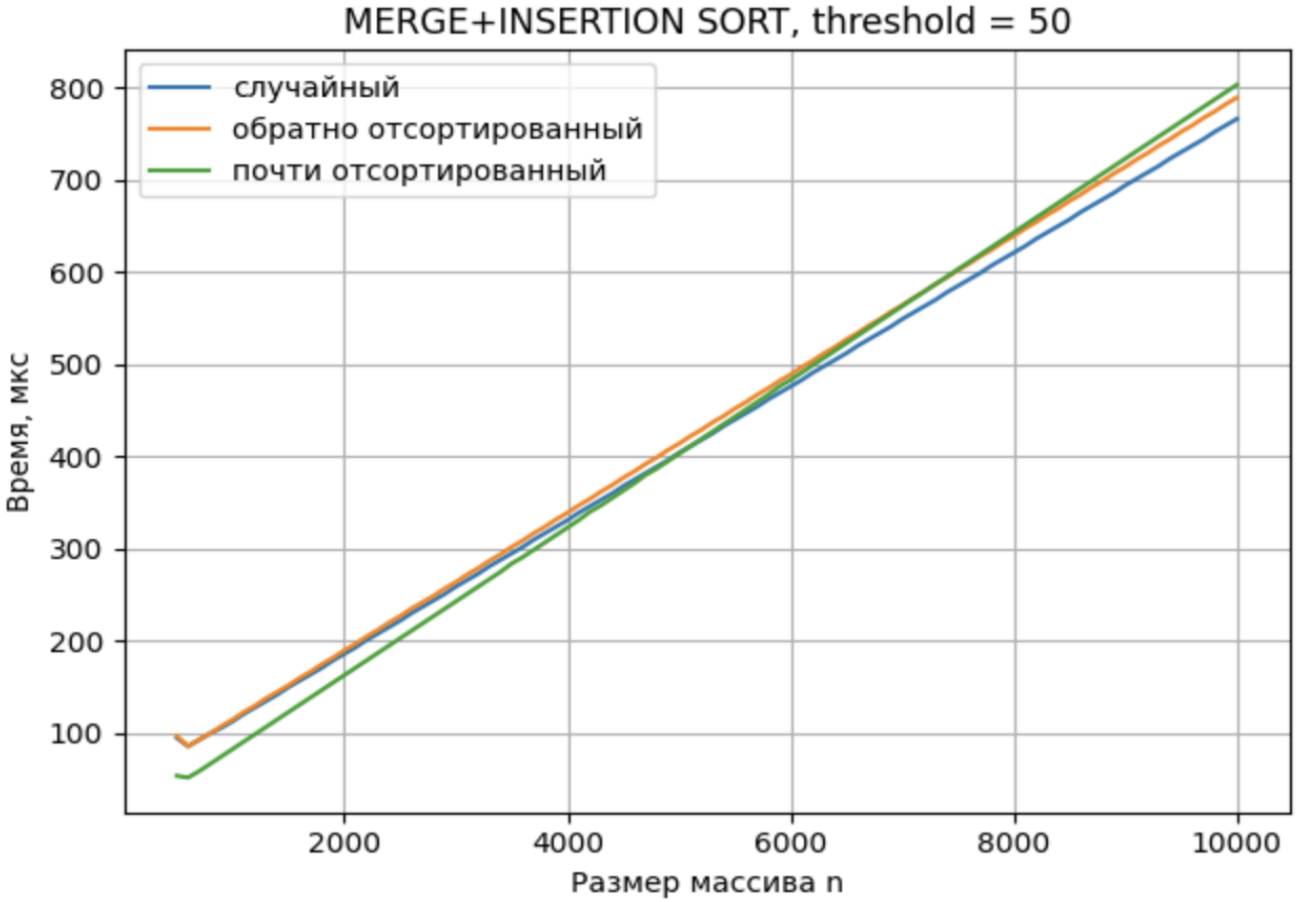

Рис. 6: Время работы гибридной сортировки при `threshold = 50`

Сравнение порогов на случайных массивах (диапазон 500–10000)

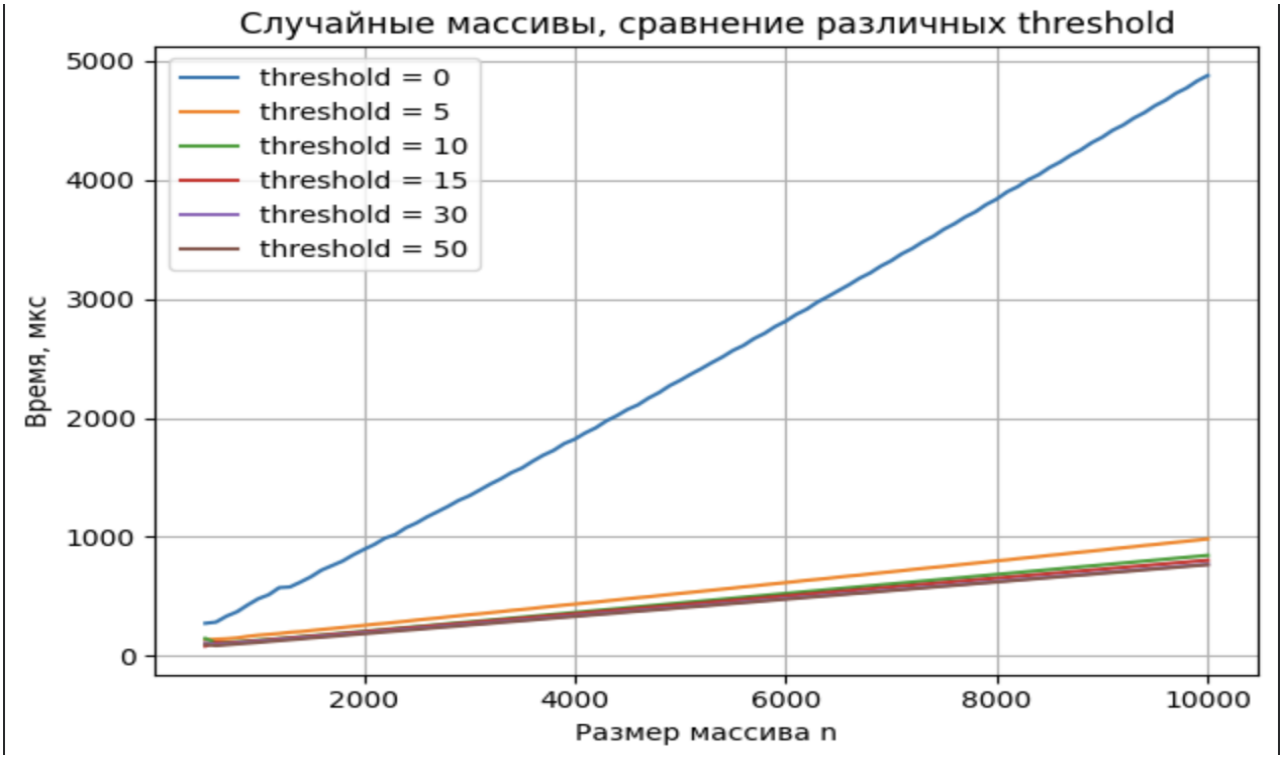

Рис. 7: Сравнение различных значений `threshold` на случайных массивах

Сравнение порогов на случайных массивах (укрупнённый диапазон)

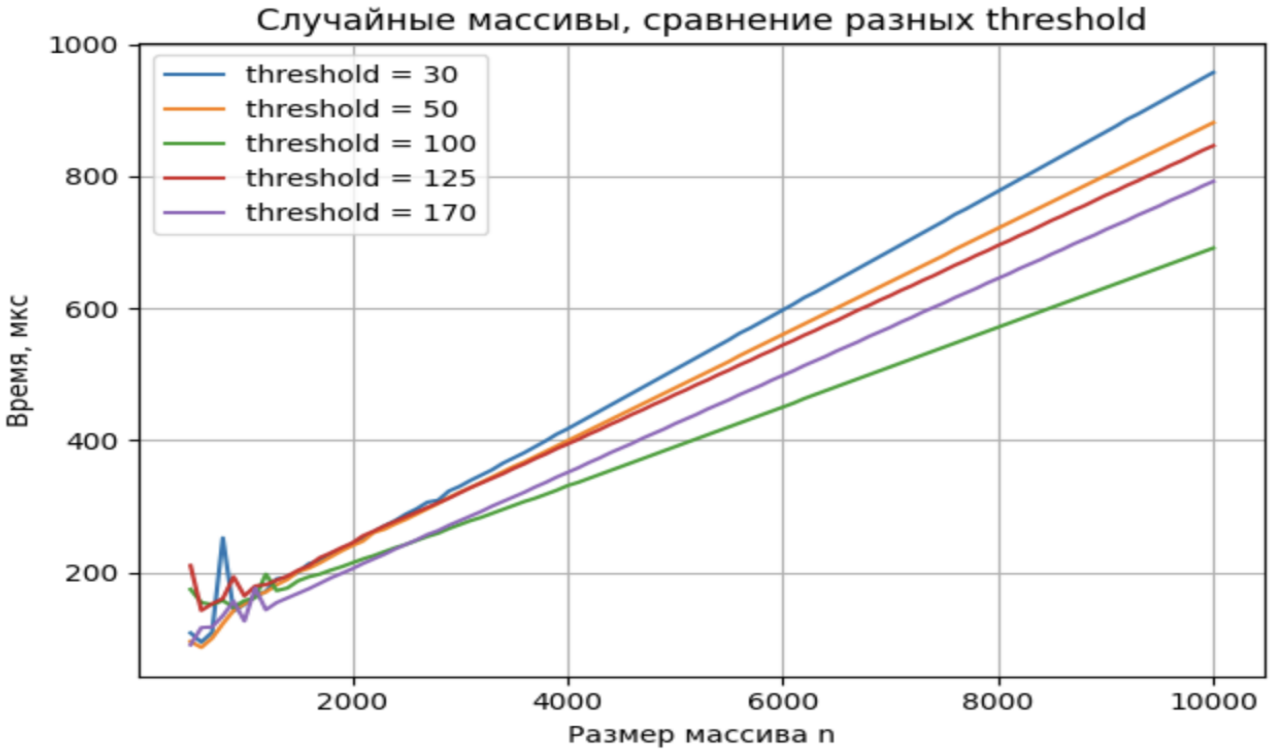

Рис. 8: Сравнение порогов на случайных массивах для больших размеров

5 Вывод

При `threshold = 0` реализуется чистый алгоритм сортировки слиянием. График показывает, что время работы почти не зависит от исходного порядка элементов: кривые для случайных, обратно отсортированных и почти отсортированных массивов близки друг к другу.

При малых, но положительных значениях `threshold` (5 и 10) сортировка вставками начинает частично заменять рекурсивные вызовы сортировки слиянием на нижних уровнях рекурсивного дерева. На графиках заметно общее снижение времени работы, особенно на почти отсортированных массивах. Это объясняется тем, что сортировка вставками обрабатывает близкие к отсортированным данные за время, близкое к линейному.

При `threshold = 15` и `threshold = 30` выигрыш на почти отсортированных массивах становится ещё более заметным, а кривые для различных типов данных сильнее расходятся. В то же время на случайных массивах прирост производительности относительно чистого MERGE SORT постепенно уменьшается.

Для `threshold = 50` квадратичная часть алгоритма (сортировка вставками) затрагивает уже достаточно большие подмассивы. На почти отсортированных данных такой переход даёт максимальный выигрыш, однако на случайных массивах время работы начинает увеличиваться по сравнению с более низкими значениями порога.

Сравнивая графики 7 и 8, можно видеть, что для случайных данных существует диапазон значений `threshold`, в котором время работы гибридного алгоритма минимально. По экспериментальным данным оптимальный порог лежит примерно в интервале от 15 до 70 элементов: при меньших значениях потенциал сортировки вставками используется не полностью, при больших — квадратичный характер алгоритма вставками начинает замедлять обработку крупных массивов.