Отчет a2

ID посылки: 349100065
GitHub: https://github.com/sshpv/Algorithms_SE2/tree/main/A2

- `a2i.cpp` — реализация гибридного алгоритма **Merge + Insertion Sort** для подзадачи **A2i** с порогом `threshold = 15`.
- `sort_algorithms.hpp` — реализация стандартного `mergeSort` и гибридного `hybridMergeSort` (merge + insertion).
- `array_generator.hpp` — класс **ArrayGenerator** для генерации трёх типов тестовых массивов.
- `sort_tester.hpp` — класс **SortTester** для замеров времени работы алгоритмов.
- `a2.cpp` — основная программа запускающая и сохраняющая серию экспериментов
- `merge_random.csv`, `merge_reversed.csv`, `merge_nearly.csv` — результаты стандартного MERGE SORT для:
  - случайных массивов,
  - обратно отсортированных,
  - почти отсортированных.
- `hybrid_random.csv`, `hybrid_reversed.csv`, `hybrid_nearly.csv` — результаты гибридного MERGE+INSERTION SORT для разных значений порога `threshold`.
- `graphs_2.py` — колабовский файл с построением графиков по csv файликам на питоне

Код a2

```cpp
#include <iostream>
#include <fstream>
#include <vector>
#include "array_generator.hpp"
#include "sort_tester.hpp"

int main() {
    std::ios::sync_with_stdio(false);
    std::cin.tie(nullptr);

    ArrayGenerator generator(100000, 0, 6000);
    SortTester tester;

    const int minN = 500;
    const int maxN = 100000;
    const int step = 100;
```

```
    const int trialsSmall = 5;   // для маленьких массивов
    const int trialsMedium = 3;  // для средних
    const int trialsLarge = 1;   // для больших

    std::vector<int> thresholds;
    thresholds.push_back(5);
    thresholds.push_back(10);
    thresholds.push_back(15);
    thresholds.push_back(20);
    thresholds.push_back(30);
    thresholds.push_back(50);

    std::ofstream mergeRandom("merge_random.csv");
    std::ofstream mergeReversed("merge_reversed.csv");
    std::ofstream mergeNearly("merge_nearly.csv");

    mergeRandom  << "n,time_us\n";
    mergeReversed<< "n,time_us\n";
    mergeNearly  << "n,time_us\n";

    std::ofstream hybridRandom("hybrid_random.csv");
    std::ofstream hybridReversed("hybrid_reversed.csv");
    std::ofstream hybridNearly("hybrid_nearly.csv");

    auto write_header = [&](std::ofstream &out) {
        out << "n";
        for (int th : thresholds) {
            out << ",t" << th;
        }
        out << "\n";
    };

    write_header(hybridRandom);
    write_header(hybridReversed);
    write_header(hybridNearly);

    for (int n = minN; n <= maxN; n += step) {
        int trials;
        if (n < 5000) {
            trials = trialsSmall;
        } else if (n < 30000) {
            trials = trialsMedium;
        } else {
            trials = trialsLarge;
        }

        std::vector<int> baseRandom   = generator.randomArray(n);
        std::vector<int> baseReversed = generator.reversedArray(n);
        std::vector<int> baseNearly   = generator.nearlySortedArray(n, n /
20);
```

```
        long long tMergeRandom   = tester.measureMergeSort(baseRandom,
trials);
        long long tMergeReversed = tester.measureMergeSort(baseReversed,
trials);
        long long tMergeNearly   = tester.measureMergeSort(baseNearly,
trials);

        mergeRandom   << n << "," << tMergeRandom   << "\n";
        mergeReversed << n << "," << tMergeReversed << "\n";
        mergeNearly   << n << "," << tMergeNearly   << "\n";

        hybridRandom << n;
        hybridReversed << n;
        hybridNearly << n;

        for (int th : thresholds) {
            long long tHRand  = tester.measureHybridSort(baseRandom,   th,
trials);
            long long tHRev   = tester.measureHybridSort(baseReversed, th,
trials);
            long long tHNear  = tester.measureHybridSort(baseNearly,   th,
trials);

            hybridRandom   << "," << tHRand;
            hybridReversed << "," << tHRev;
            hybridNearly   << "," << tHNear;
        }

        hybridRandom   << "\n";
        hybridReversed << "\n";
        hybridNearly   << "\n";

        std::cerr << "Done n = " << n << " (trials = " << trials << ")\n";
    }

    return 0;
}
```

Изначальная реализация гибридного алгоритма (a2i)

```
#include <iostream>
#include <vector>

using namespace std;

const int PAR = 15;

void insertion_sort(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;
    }
}

void merge_sort(vector<long long> &a, vector<long long> &tmp, int l, int
r) {
    int len = r - l;
    if (len <= PAR) {
        insertion_sort(a, l, r);
        return;
    }

    int m = l + (r - l) / 2;
    merge_sort(a, tmp, l, m);
    merge_sort(a, tmp, m, r);

    int i = l;
    int j = m;
    int 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];
    }
}

int main() {
    ios::sync_with_stdio(false);
    cin.tie(nullptr);

    int n;
```

```
    if (!(cin >> n)) {
        return 0;
    }

    vector<long long> a(n);
    for (int i = 0; i < n; ++i) {
        cin >> a[i];
    }

    if (n > 1) {
        vector<long long> tmp(n);
        merge_sort(a, tmp, 0, n);
    }

    for (int i = 0; i < n; ++i) {
        if (i) cout << ' ';
        cout << a[i];
    }
    cout << '\n';

    return 0;
}
```

Этап 1. Подготовка тестовых данных

Надо получить три типа массивов целых чисел для размеров от 500 до 100000 (шаг 100), значения от 0 до 6000

Реализация `array_generator.hpp`:

- В конструкторе `ArrayGenerator(int maxLen, int mn, int mx)`:
  - генерируется один базовый массив длины `maxLen = 100000`;
  - элементы — равномерно в `[0, 6000]` (`std::mt19937`, `std::uniform_int_distribution`)
- Для каждого размера `n` используются первые `n` элементов базового массива

Типы массивов:

1. `randomArray(n)` — префикс базового случайного массива длины `n`
2. `reversedArray(n)` — префикс длины `n`, отсортированный по возрастанию и перевёрнутый (по невозрастанию)
3. `nearlySortedArray(n, swaps)` — префикс длины `n`, отсортированный по возрастанию, затем выполняется `swaps = n / 20` случайных обменов (≈5% пар)

Параметры: размеры 500–100000, шаг 100, диапазон значений 0–6000, три требуемых типа массивов.

класс ArrayGenerator

```
#ifndef ARRAY_GENERATOR_HPP
#define ARRAY_GENERATOR_HPP

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

class ArrayGenerator {
public:
    ArrayGenerator(int maxLength = 100000, int minVal = 0, int maxVal =
6000)
            : maxLen(maxLength), minValue(minVal), maxValue(maxVal),
              rng(std::random_device{}()), dist(minVal, maxVal)
    {
        base.reserve(maxLen);
        for (int i = 0; i < maxLen; ++i) {
            base.push_back(dist(rng));
        }
    }

    // Случайный массив размера n
    std::vector<int> randomArray(int n) const {
        if (n > maxLen) n = maxLen;
        return std::vector<int>(base.begin(), base.begin() + n);
    }

    // Массив размера n по невозрастанию
    std::vector<int> reversedArray(int n) const {
        if (n > maxLen) n = maxLen;
        std::vector<int> v(base.begin(), base.begin() + n);
        std::sort(v.begin(), v.end());
        std::reverse(v.begin(), v.end());
        return v;
    }

    // Почти отсортированный
    std::vector<int> nearlySortedArray(int n, int swaps) {
        if (n > maxLen) n = maxLen;
        std::vector<int> v(base.begin(), base.begin() + n);
        std::sort(v.begin(), v.end());

        if (n == 0) return v;
        std::uniform_int_distribution<int> posDist(0, n - 1);
        for (int i = 0; i < swaps; ++i) {
            int i1 = posDist(rng);
            int i2 = posDist(rng);
            std::swap(v[i1], v[i2]);
```

```
        }
        return v;
    }

private:
    int maxLen;
    int minValue;
    int maxValue;
    std::vector<int> base;

    mutable std::mt19937 rng;
    mutable std::uniform_int_distribution<int> dist;
};

#endif // ARRAY_GENERATOR_HPP
```

Этап 2. Эмпирический анализ стандартного MERGE SORT

Надо замерить время работы классического рекурсивного MERGE SORT с доп. памятью на трёх типах массивов.

Реализация (`sort_algorithms.hpp`):

- `mergeSort(std::vector<int>& a)`:
  - выделяется буфер `tmp(a.size())`
  - рекурсивная функция `mergeSortRec(a, tmp, l, r)`:
    - если `r - l <= 1` — выход
    - рекурсивные вызовы для `[l, m)` и `[m, r)`
    - слияние в `tmp`, копирование обратно в `a[l..r)`

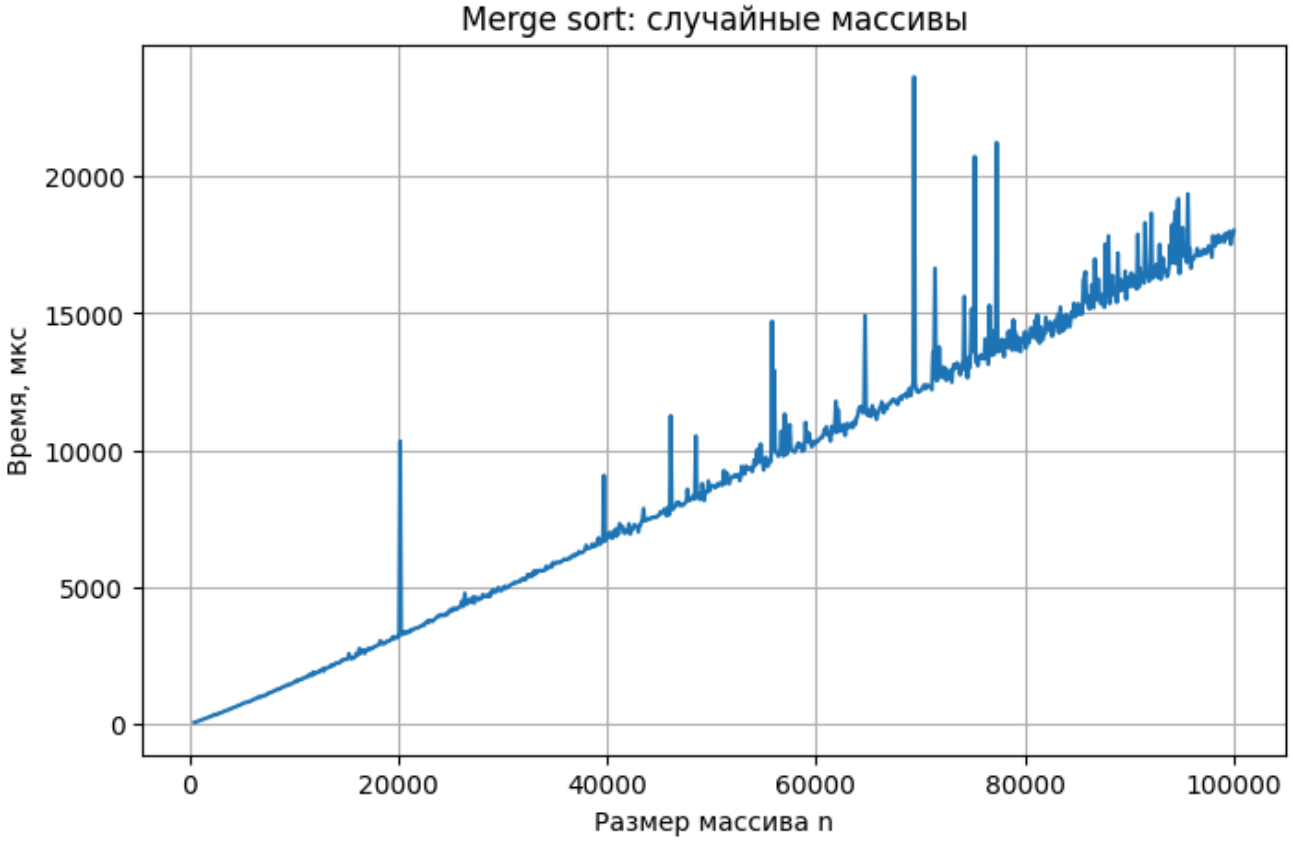
Замеры (`sort_tester.hpp`, `a2_experiments.cpp`):

- `measureMergeSort(const std::vector<int>& data, int trials)`:
  - на каждом trial копируется `data` → `arr`
  - время замеряется через `std::chrono::high_resolution_clock`
  - возвращается среднее время в микросекундах
- Число повторов:
  - `n < 5000` → 5
  - `5000 ≤ n < 30000` → 3
  - `n ≥ 30000` → 1
- Для каждого `n` и каждого типа массива (random, reversed, nearly) результаты записываются в:
  - `merge_random.csv`
  - `merge_reversed.csv`

- `merge_nearly.csv` формат: `n,time_us`

1) График: _Merge sort - случайные массивы

(из `merge_random.csv`)
Зависимость времени работы стандартного `mergeSort` от размера массива `n` на случайных данных (ось X — `n`, ось Y — время в мкс)



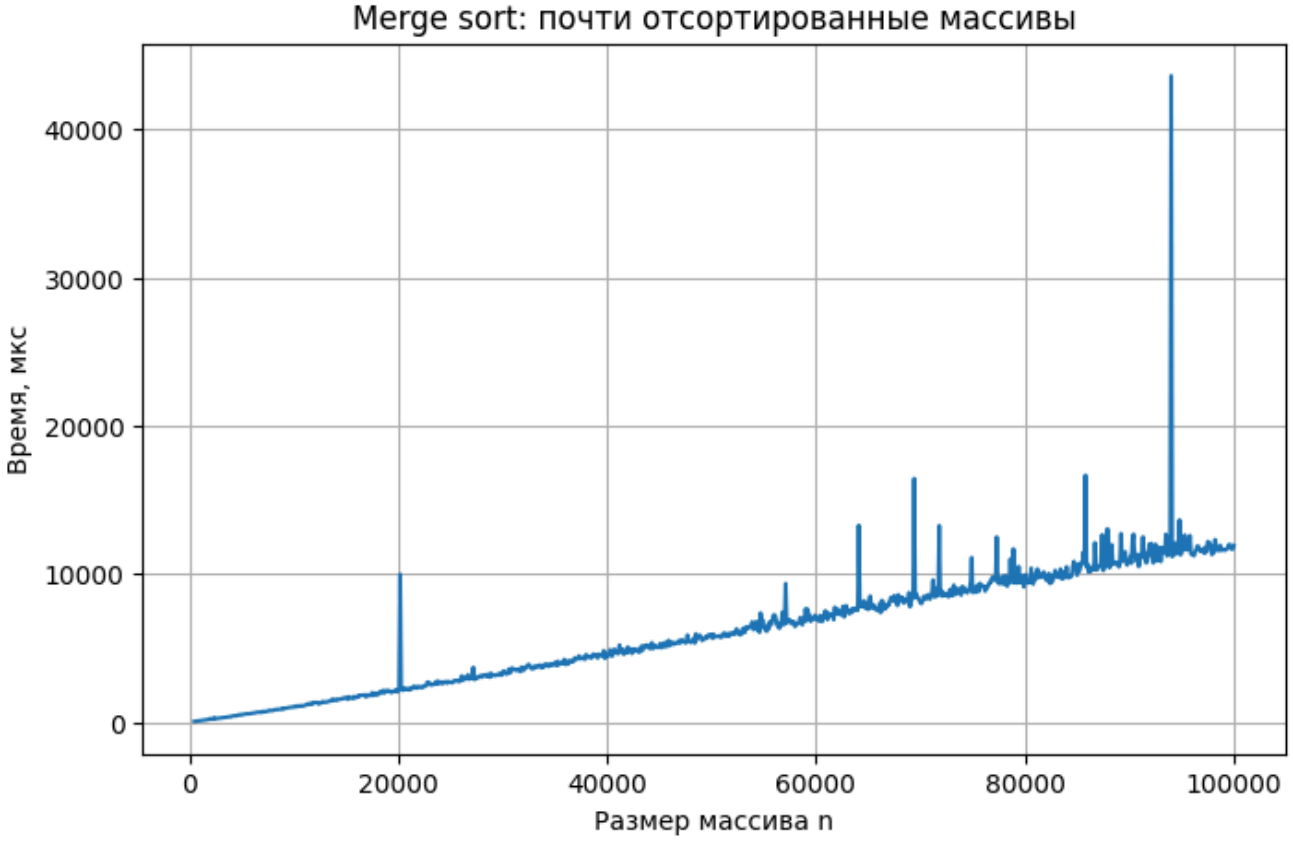
2) График: *Merge sort - обратно отсортированные массивы*

(из `merge_reversed.csv`)
Время `mergeSort` в зависимости от `n` для массивов, отсортированных по

невозрастанию

3) График: *Merge sort - почти отсортированные массивы*

(из `merge_nearly.csv`)
Время `mergeSort` в зависимости от `n` для почти отсортированных массивов (немного случайных обменов в отсортированном массиве).



Вывод по Этапу 2:

- Время растёт примерно как $O(n \log n)$
- Отличия между тремя типами массивов небольшие: Merge sort мало зависит от начального порядка

Этап 3. Эмпирический анализ гибридного MERGE+INSERTION SORT

Надо оценить влияние порога `threshold` в гибридном алгоритме MERGE+INSERTION SORT на время работы.

Реализация (`sort_algorithms.hpp`):

- `hybridMergeSort(std::vector<int>& a, int threshold)`:
  - общий буфер `tmp` как в merge sort
  - `hybridMergeSortRec(a, tmp, l, r, threshold)`:
    - если `r - l <= threshold` → `insertionSort(a, l, r)` и выход
    - иначе — рекурсивное разделение и слияние как в стандартном MERGE SORT
- `insertionSort(a, l, r)` — классическая сортировка вставками на подмассиве `[l, r)`

Замеры (`sort_tester.hpp`, `a2_experiments.cpp`):

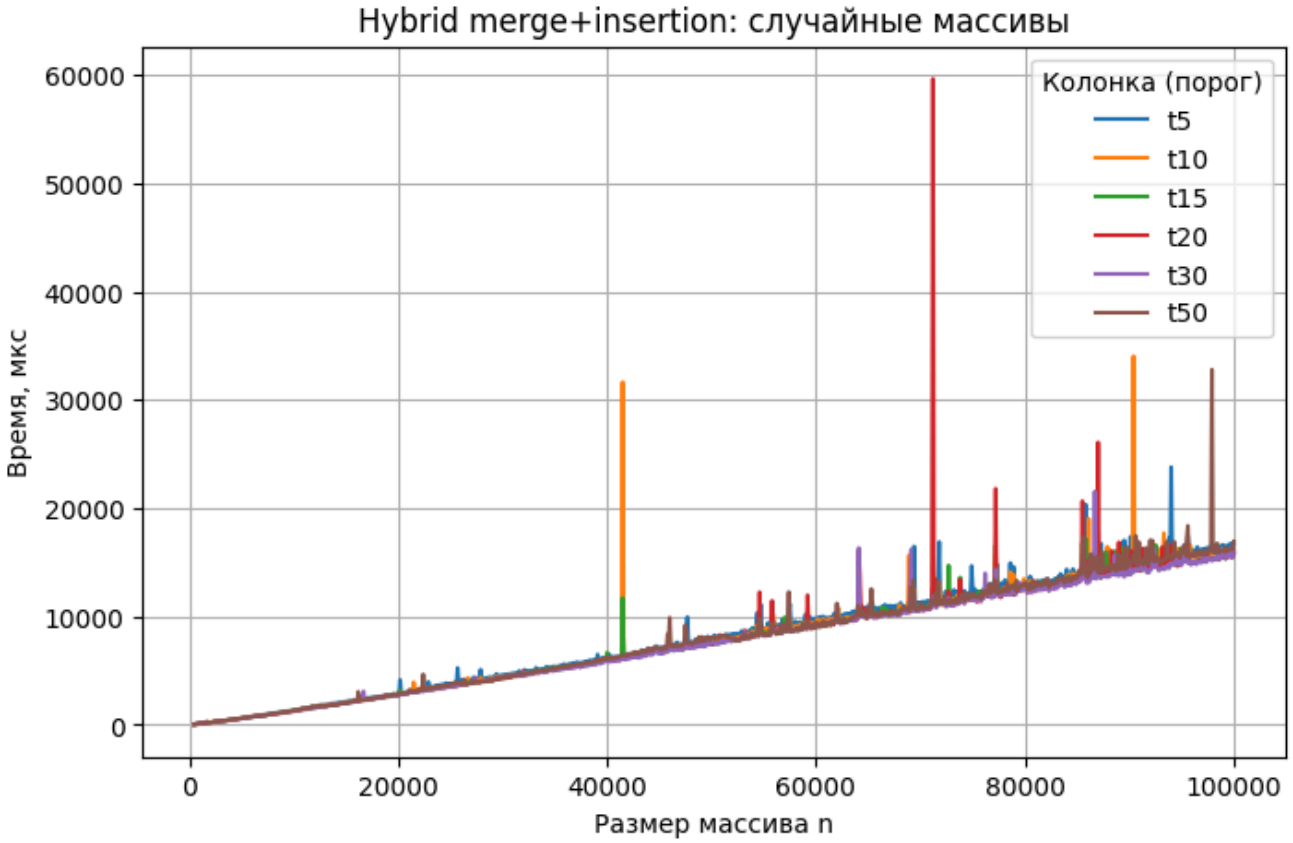
- `measureHybridSort(const std::vector<int>& data, int threshold, int trials)`:
  - такая же схема, как у `measureMergeSort`
- Используемые пороги: `threshold ∈ {5, 10, 15, 20, 30, 50}`
- Для каждого `n`, для трёх типов массивов и для каждого порога:
  - снимается среднее время
  - результаты записываются в:
    - `hybrid_random.csv`
    - `hybrid_reversed.csv`
    - `hybrid_nearly.csv` формат: `n,t5,t10,t15,t20,t30,t50`

4) График: *Hybrid merge+insertion - случайные массивы*

(из `hybrid_random.csv` — линии `t5, t10, t15, t20, t30, t50`)
Время гибридного алгоритма для случайных массивов при разных значениях порога `threshold` (5, 10, 15, 20, 30, 50)

Вывод: Лучшее время на случайных массивах дают пороги примерно 10-20; при `threshold=5` выигрыш небольшой, при `threshold≥30` время начинает заметно расти

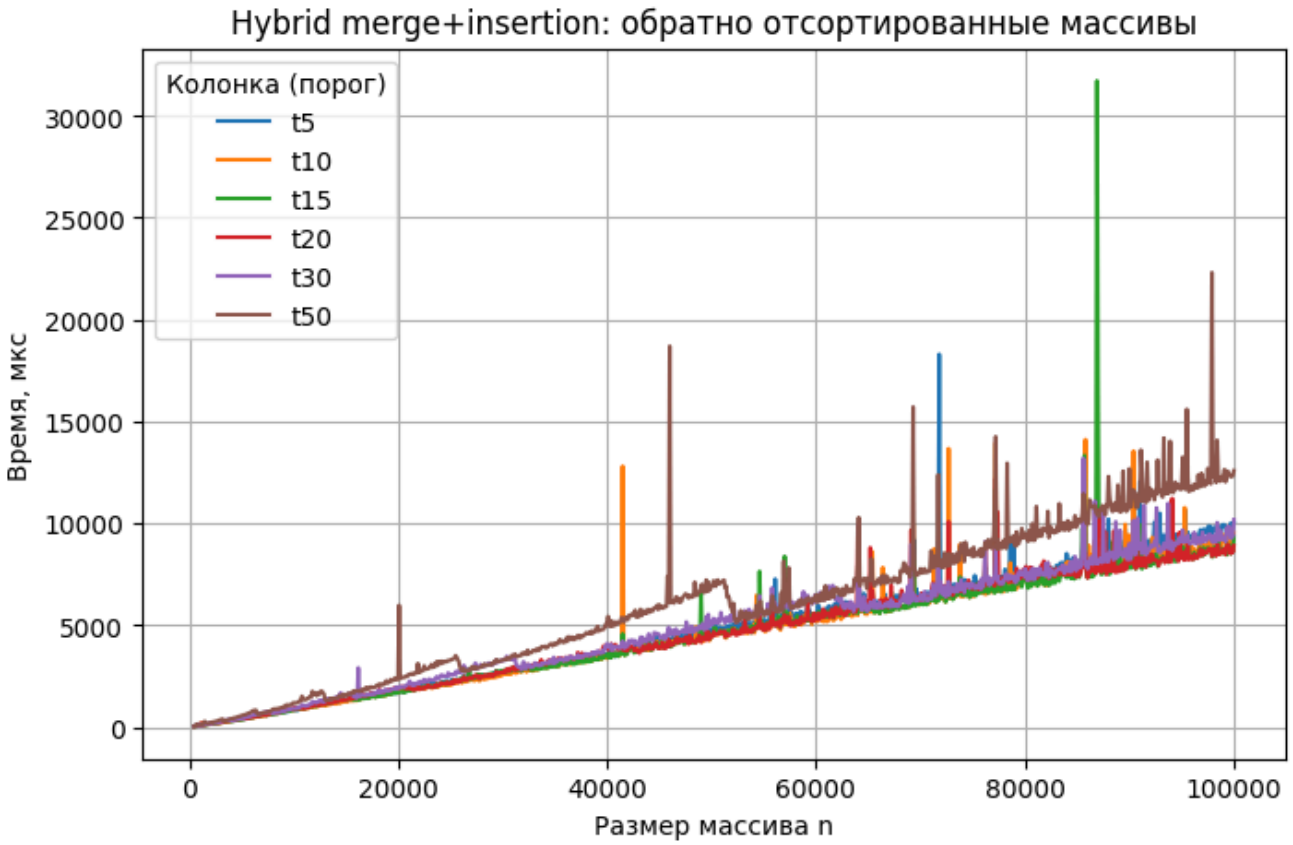
из-за квадратичной части INSERTION SORT



5) График: *Hybrid merge+insertion - обратно отсортированные массивы*

(из `hybrid_reversed.csv`)
Время гибридного алгоритма для обратно отсортированных массивов при разных порогах `threshold`
Вывод: На обратно отсортированных массивах INSERTION SORT работает в худшем случае, поэтому большие пороги (30–50) резко ухудшают время; пороги 10-20 дают

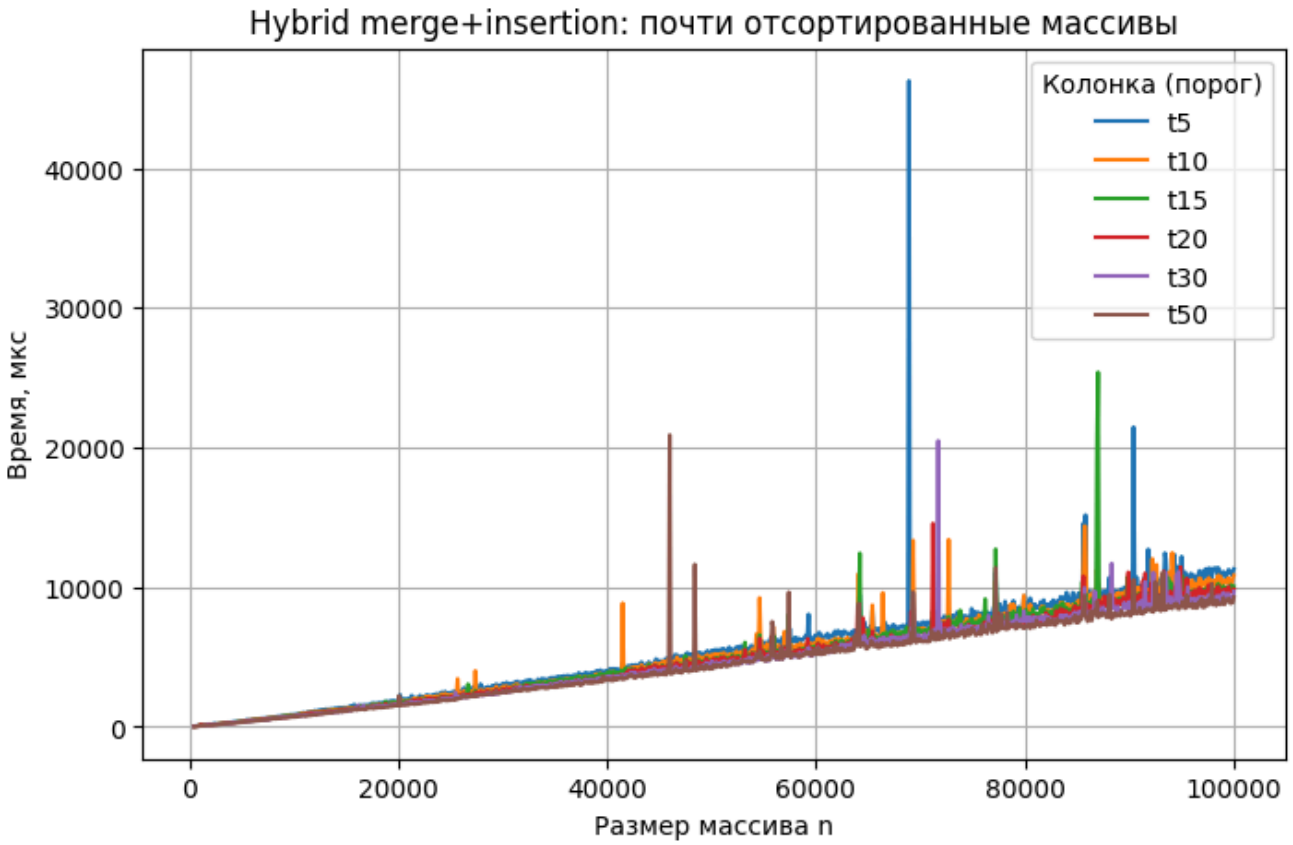
более сбалансированное поведение и близки по времени к чистому MERGE SORT



6) График: *Hybrid merge+insertion - почти отсортированные массивы*

(из `hybrid_nearly.csv`)
Время гибридного алгоритма на почти отсортированных массивах для разных значений `threshold`.
Вывод: Для порогов 10–20 гибридный алгоритм на почти отсортированных данных заметно быстрее других вариантов и стандартного MERGE SORT - INSERTION SORT

очень хорошо отрабатывает на таких входах



Sort Algorithms

```cpp
#ifndef SORT_ALGORITHMS_HPP
#define SORT_ALGORITHMS_HPP

#include <vector>

namespace mysort {

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

    inline void mergeSortRec(std::vector<int> &a,
                             std::vector<int> &tmp,
                             int l, int r) {
        if (r - l <= 1) return;
        int m = l + (r - l) / 2;
```

```
        mergeSortRec(a, tmp, l, m);
        mergeSortRec(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];
    }

    inline void mergeSort(std::vector<int> &a) {
        if (a.size() <= 1) return;
        std::vector<int> tmp(a.size());
        mergeSortRec(a, tmp, 0, static_cast<int>(a.size()));
    }

    // Гибридный
    inline void hybridMergeSortRec(std::vector<int> &a,
                                   std::vector<int> &tmp,
                                   int l, int r,
                                   int threshold) {
        int len = r - l;
        if (len <= threshold) {
            insertionSort(a, l, r);
            return;
        }
        int m = l + len / 2;
        hybridMergeSortRec(a, tmp, l, m, threshold);
        hybridMergeSortRec(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];
    }

    inline void hybridMergeSort(std::vector<int> &a, int threshold) {
        if (a.size() <= 1) return;
        std::vector<int> tmp(a.size());
        hybridMergeSortRec(a, tmp, 0, static_cast<int>(a.size()),
threshold);
    }
```

```
}

#endif // SORT_ALGORITHMS_HPP
```

класс SortTester

```
#ifndef SORT_TESTER_HPP
#define SORT_TESTER_HPP

#include <vector>
#include <chrono>
#include "sort_algorithms.hpp"

class SortTester {
public:
    // Время стандартного merge sort, среднее по trials, в микросекундах
    long long measureMergeSort(const std::vector<int> &data, int trials) {
        using namespace std::chrono;
        long long total = 0;
        for (int t = 0; t < trials; ++t) {
            std::vector<int> arr = data;
            auto start = high_resolution_clock::now();
            mysort::mergeSort(arr);
            auto end = high_resolution_clock::now();
            total += duration_cast<microseconds>(end - start).count();
        }
        return total / trials;
    }

    // Время гибридного merge+insertion сортировки с данным threshold
    long long measureHybridSort(const std::vector<int> &data,
                                int threshold,
                                int trials) {
        using namespace std::chrono;
        long long total = 0;
        for (int t = 0; t < trials; ++t) {
            std::vector<int> arr = data;
            auto start = high_resolution_clock::now();
            mysort::hybridMergeSort(arr, threshold);
            auto end = high_resolution_clock::now();
            total += duration_cast<microseconds>(end - start).count();
        }
        return total / trials;
    }
};

#endif // SORT_TESTER_HPP
```

Вывод по Этапу 3 (кратко):

- Малый порог (5) почти не отличается от MERGE SORT
- Большие пороги (30–50) ухудшают время из-за квадратичности INSERTION SORT, особенно на случайных и обратно отсортированных массивах
- Пороги 10–20 дают наилучшее время, особенно на почти отсортированных данных.

Этап 4. Сравнительный анализ

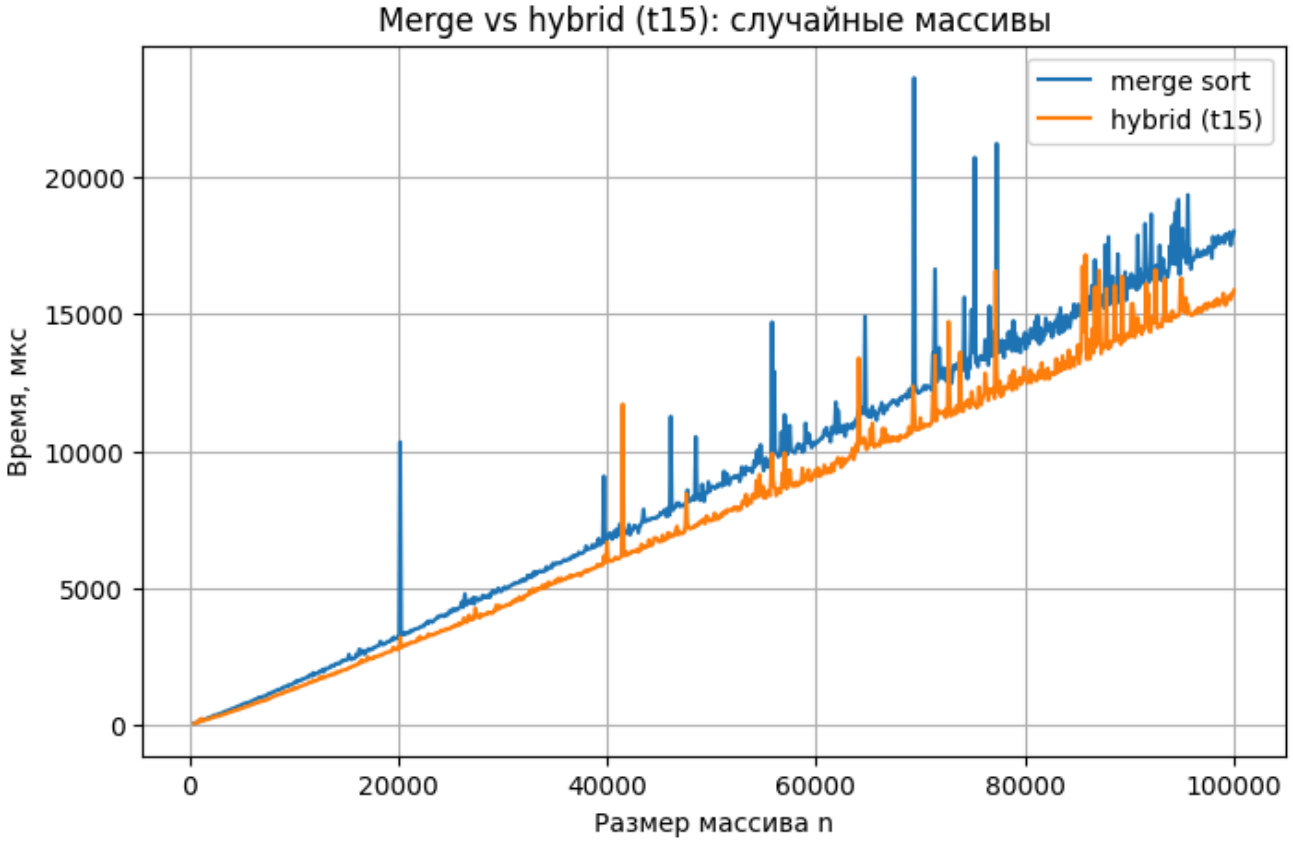
Цель: сравнить стандартный MERGE SORT и гибридный MERGE+INSERTION SORT, подобрать разумный порог `threshold`.

Сравнение (по CSV и графикам):

- Для каждого типа массивов строятся:
  - графики `merge_*` (MERGE SORT)
  - графики `hybrid_*` (зависимость времени от порога)
  - сравнительные графики `T_merge` vs `T_hybrid(threshold)` (например, `t15`)
  - графики ускорения `speedup = T_merge / T_hybrid`

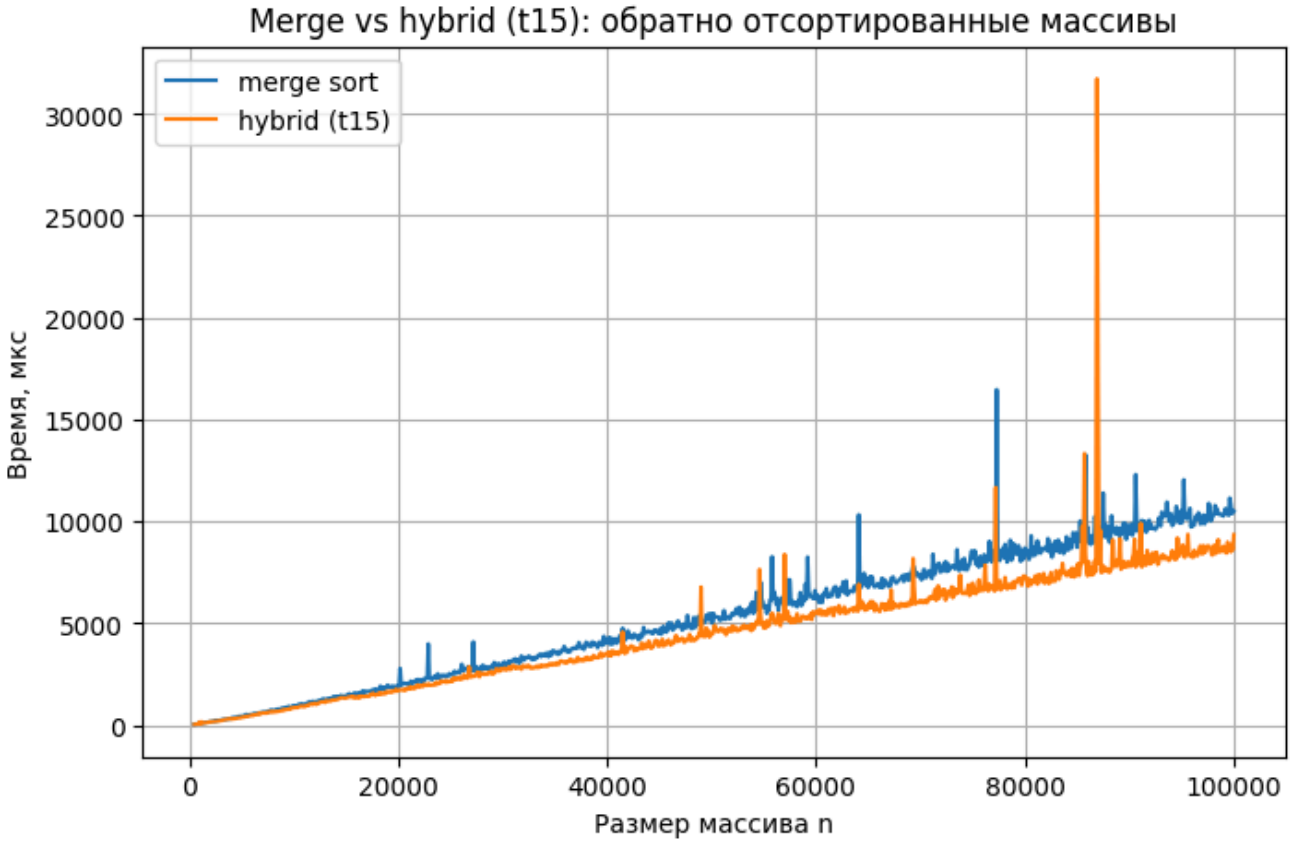
7) График: *Merge vs hybrid (t15) - случайные массивы*

(из `merge_random.csv` + `hybrid_random.csv`)
Кривая времени стандартного `mergeSort` и кривая времени гибридного алгоритма с `threshold=15` для случайных массивов



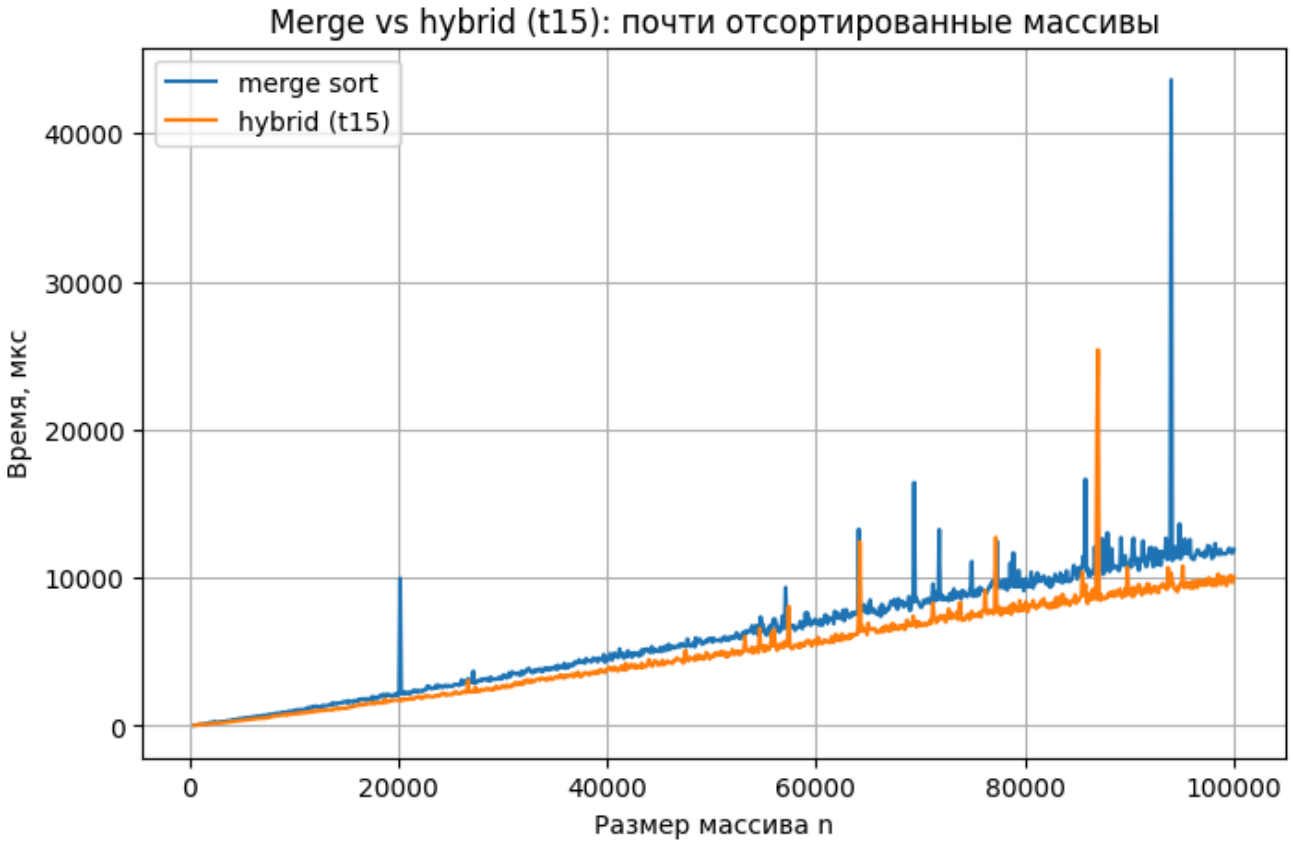
8) График: *Merge vs hybrid (t15) - обратно отсортированные массивы*

(из `merge_reversed.csv` + `hybrid_reversed.csv`)
Сравнение времени стандартного `mergeSort` и гибридного алгоритма с `threshold=15` на обратно отсортированных массивах



9) График: *Merge vs hybrid (t15) - почти отсортированные массивы*

(из `merge_nearly.csv` + `hybrid_nearly.csv`)
Сравнение времени стандартного `mergeSort` и гибридного алгоритма с `threshold=15`



Наблюдения:

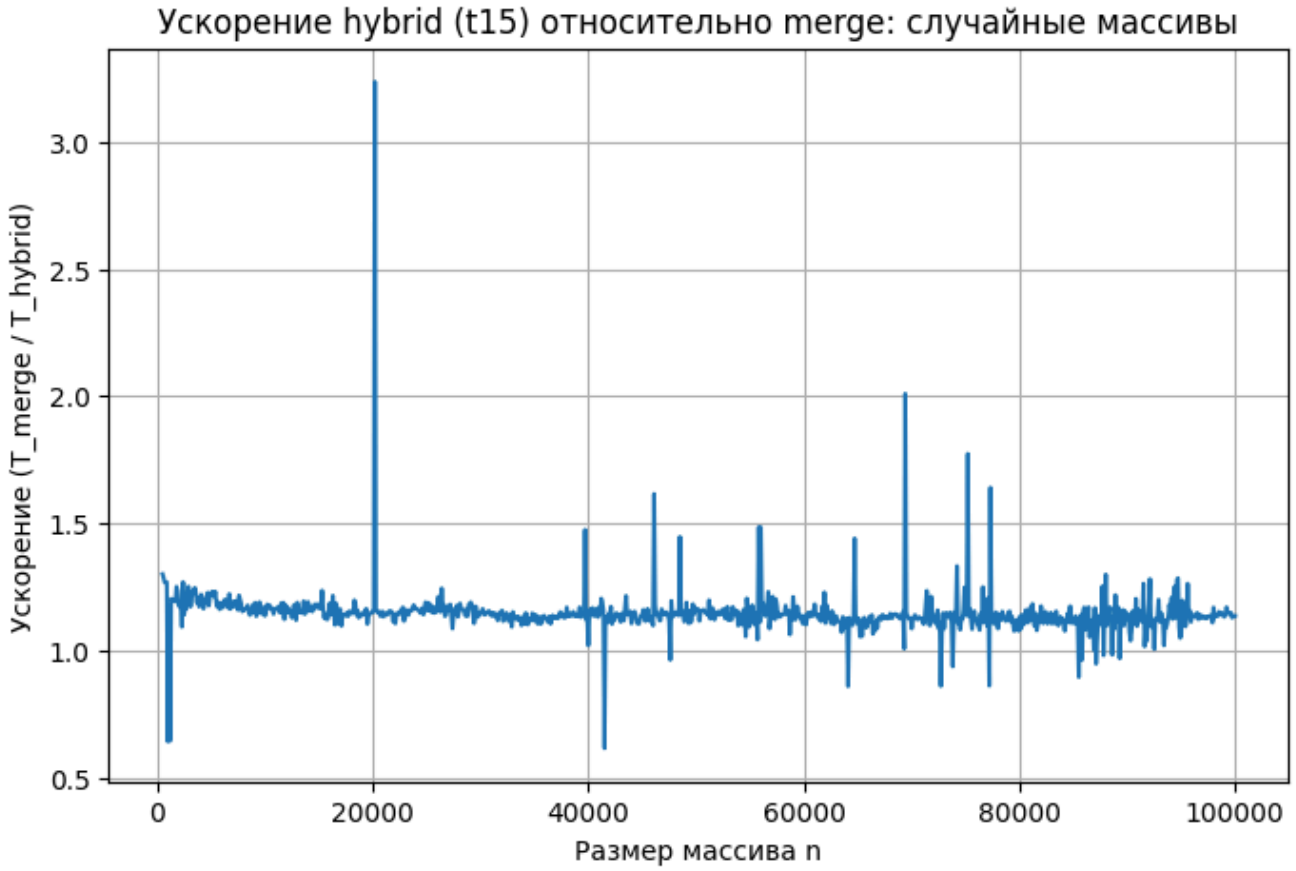
- На **случайных** массивах гибрид с порогами 10–20 стабильно быстрее MERGE SORT
- На **обратно отсортированных** массивах гибрид с порогами 10–20 близок по времени к MERGE SORT; большие пороги заметно хуже
- На **почти отсортированных** массивах гибрид с порогами 10–20 даёт максимальный выигрыш

---

10) График: _Speedup (t15) - случайные массивы

(из `merge_random.csv` и `hybrid_random.csv` — `speedup = T_merge / T_hybrid`)
График ускорения гибридного алгоритма с `threshold=15` относительно MERGE SORT: по оси Y
Вывод: гибридная схема на случайных массивах даёт устойчивое умеренное

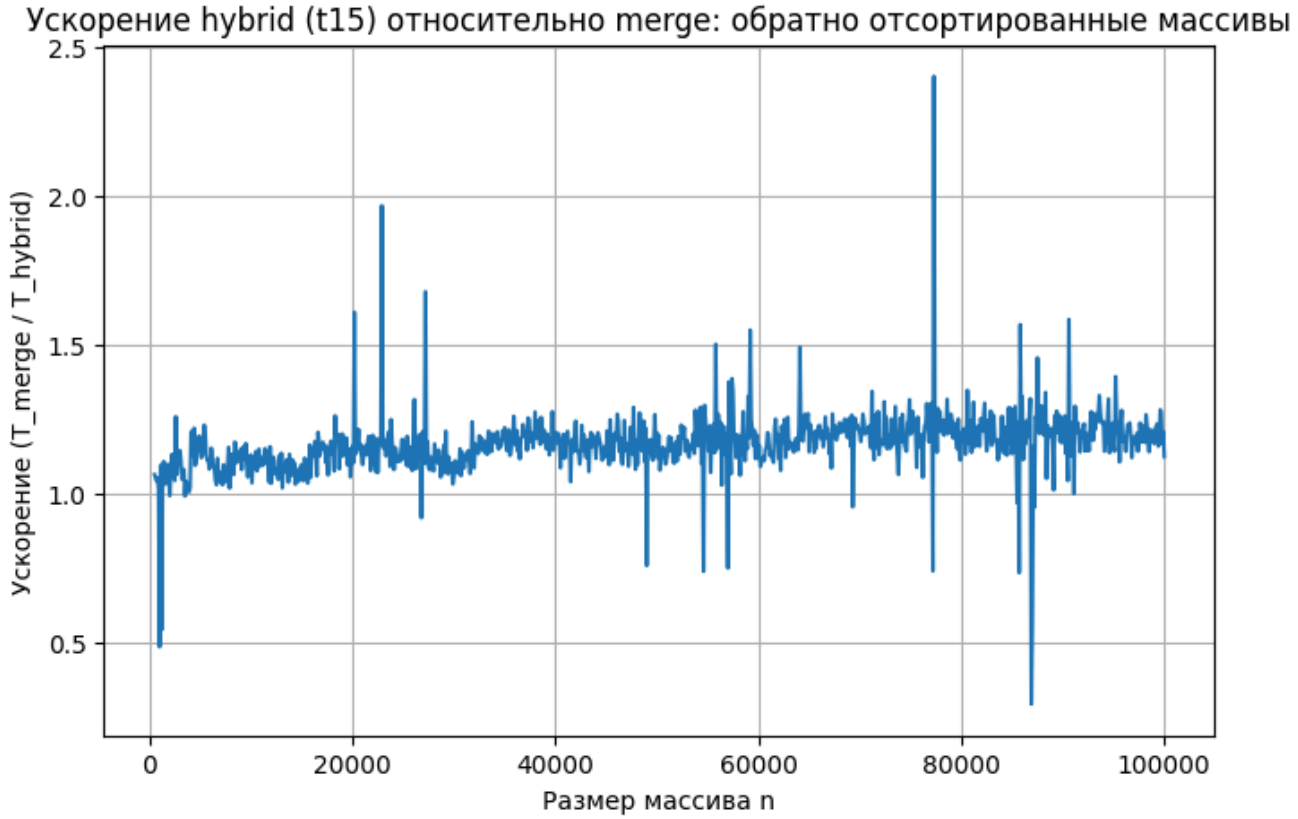
ускорение



11) График: _Speedup (t15) - обратно отсортированные массивы

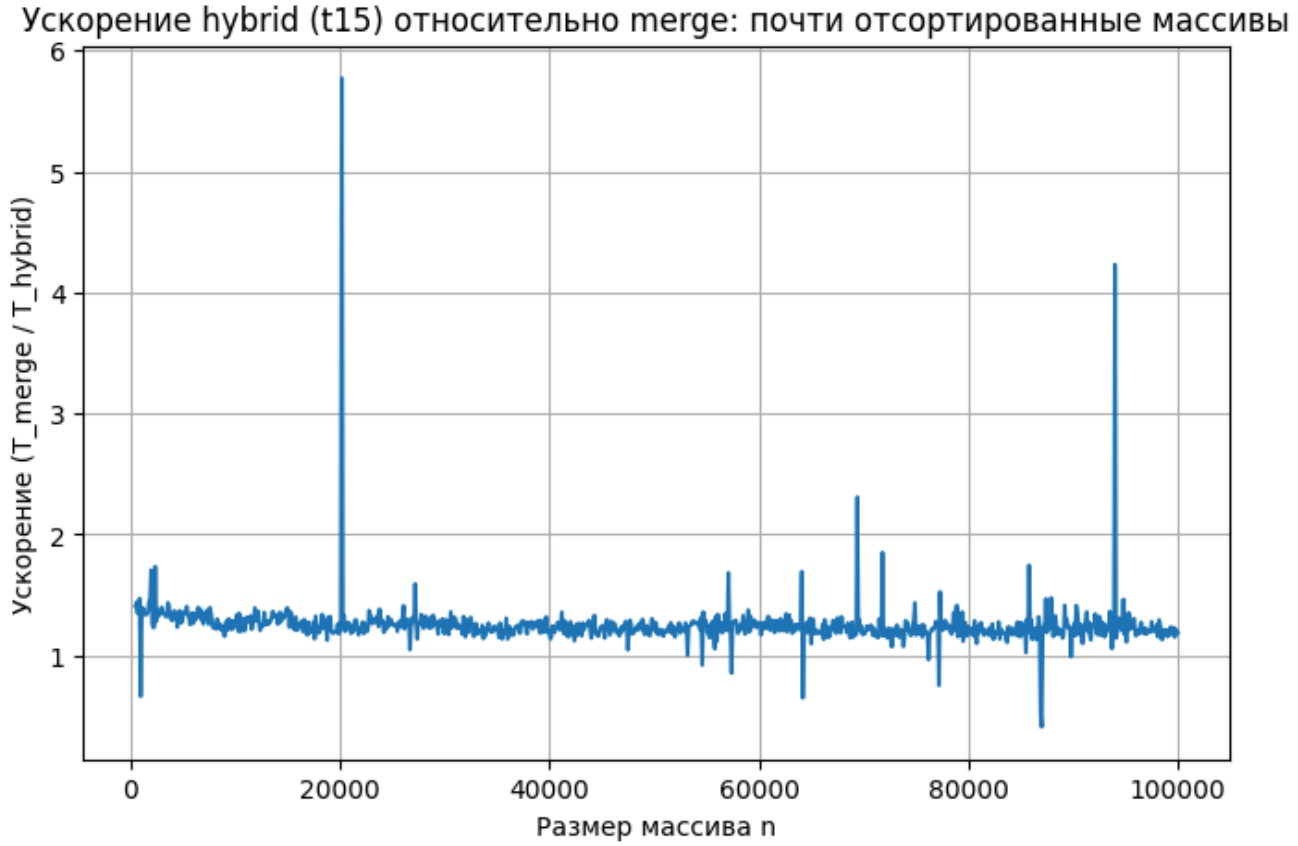
(из `merge_reversed.csv` и `hybrid_reversed.csv`)
Ускорение `T_merge / T_hybrid` для обратно отсортированных массивов при `threshold=15`
Вывод: гибридный алгоритм с порогом 15 примерно равен по скорости чистому

MERGE SORT



12) График: _Speedup (t15) - почти отсортированные массивы

(из `merge_nearly.csv` и `hybrid_nearly.csv`)
Ускорение гибридного алгоритма с `threshold=15` на почти отсортированных массивах
Вывод: гибридный алгоритм значительно быстрее MERGE SORT на почти отсортированных данных



Итоговый вывод:

- Эмпирические замеры на массивах размером от 500 до 100000 показали, что стандартный MERGE SORT ведёт себя в соответствии с теорией $O(n \log n)$ и практически не зависит от типа входных данных
- Гибридный алгоритм с порогом переключения на INSERTION SORT даёт выигрыш по времени на малых подмассивах, но при слишком больших порогах (30–50) квадратичная сложность INSERTION SORT начинает ухудшать результат, особенно на случайных и обратно отсортированных массивах
- Оптимальный диапазон порога по экспериментам - около 10–20 элементов: при таких значениях гибрид стабильно быстрее MERGE SORT на случайных и почти отсортированных массивах и не даёт заметной деградации на обратно отсортированных
- Выбор порога `threshold = 15` в решении A2i таким образом является экспериментально обоснованным компромиссом между скоростью и устойчивостью к типу входных данных