id посылки: 349314210

гит: https://github.com/Polinushkin/SET3-A2.git

- a2i.cpp - реализация merge + insertion sort
- a2.cpp - реализация ArrayGenerator и SortTester (проводит серию замеров времени для стандартного merge sort и merge+insertion sort)
- merge_sort_*.csv, hybrid_sort_*.csv - результаты замеров (размер массива и среднее время выполнения в микросекундах)
- graphs.py - код для создания графиков
- *.png - графики (ниже будет описание каждого)

a2i.cpp:

a2i.cpp

```cpp
#include <iostream>
#include <vector>
#include <algorithm>

using namespace std;

#define ll long long
#define ld long double
#define FOR(i, s, f) for(ll (i) = (s); (i) < (f); (i)++)

// Слияние для merge sort
void merge(vector<int>& arr, int left, int mid, int right) {
    int n1 = mid - left + 1;
    int n2 = right - mid;

    vector<int> L(n1), R(n2);

    FOR(i, 0, n1) {
        L[i] = arr[left + i];
    }
    FOR(j, 0, n2) {
        R[j] = arr[mid + 1 + j];
    }

    int i = 0, j = 0, k = left;
    while (i < n1 && j < n2) {
        if (L[i] <= R[j]) {
            arr[k] = L[i];
            i++;
        } else {
            arr[k] = R[j];
            j++;
        }
        k++;
    }

```

```
    while (i < n1) {
        arr[k] = L[i];
        i++;
        k++;
    }

    while (j < n2) {
        arr[k] = R[j];
        j++;
        k++;
    }
}

void mergeSort(vector<int>& arr, int left, int right) {
    if (left < right) {
        int mid = left + (right - left) / 2;
        mergeSort(arr, left, mid);
        mergeSort(arr, mid + 1, right);
        merge(arr, left, mid, right);
    }
}

void insertionSort(vector<int>& arr, int left, int right) {
    FOR(i, left + 1, right + 1) {
        int key = arr[i];
        int j = i - 1;
        while (j >= left && arr[j] > key) {
            arr[j + 1] = arr[j];
            j--;
        }
        arr[j + 1] = key;
    }
}

```

```cpp
void hybridMergeSort(vector<int>& arr, int left, int right, int threshold) {
    if (right - left + 1 < threshold) {
        insertionSort(arr, left, right);
        return;
    }
    if (left < right) {
        int mid = left + (right - left) / 2;
        hybridMergeSort(arr, left, mid, threshold);
        hybridMergeSort(arr, mid + 1, right, threshold);
        merge(arr, left, mid, right);
    }
}

void solve() {
    int n;
    cin >> n;
    if (n == 0) {
        return;
    }
    vector<int> arr(n);
    FOR(i, 0, n) {
        cin >> arr[i];
    }

    int threshold = 15;
    hybridMergeSort(arr, 0, n - 1, threshold);

    FOR(i, 0, n) {
        cout << arr[i];
        if (i < n - 1) cout << " ";
    }
    cout << "\n";
}

```

```cpp
int main() {
    cin.tie(nullptr)->sync_with_stdio(false);
    solve();
    return 0;
}
```

a2.cpp:

```cpp
#include <iostream>
#include <vector>
#include <algorithm>
#include <random>
#include <chrono>
#include <fstream>
#include <iomanip>

using namespace std;

#define ll long long
#define ld long double
#define FOR(i, s, f) for(ll (i) = (s); (i) < (f); (i)++)

void merge(vector<int>& arr, int left, int mid, int right);
void mergeSort(vector<int>& arr, int left, int right);
void insertionSort(vector<int>& arr, int left, int right);
void hybridMergeSort(vector<int>& arr, int left, int right, int threshold);

class ArrayGenerator {
private:
    vector<int> baseRandomArray;
    vector<int> baseReversedArray;
    vector<int> baseSortedArray;
    int maxLength;
    int minRange;
    int maxRange;

    mutable mt19937 gen;

    void generateBaseArrays() {
        uniform_int_distribution<int> dist(minRange, maxRange);

        baseRandomArray.reserve(maxLength);
        baseReversedArray.reserve(maxLength);
        baseSortedArray.reserve(maxLength);

        FOR(i, 0, maxLength) {
            int val = dist(gen);
            baseRandomArray.push_back(val);
            baseSortedArray.push_back(val);
        }
        sort(baseSortedArray.begin(), baseSortedArray.end());
        baseReversedArray = baseSortedArray;
        reverse(baseReversedArray.begin(), baseReversedArray.end());
    }

public:
    ArrayGenerator(unsigned int seed = random_device{}()) : maxLength(100000), minRange(0), maxRange(10000), gen(seed) {
        generateBaseArrays();
    }

    // Возвращает подмассив из сгенерированного случайного массива
    vector<int> getRandomArray(int size) const {
        if (size > maxLength) size = maxLength;
        return vector<int>(baseRandomArray.begin(), baseRandomArray.begin() + size);
    }

    // Возвращает подмассив из обратно отсортированного массива
    vector<int> getReversedArray(int size) const {
        if (size > maxLength) size = maxLength;
        return vector<int>(baseReversedArray.begin(), baseReversedArray.begin() + size);
    }
```

```
    // Возвращает отсортированный массив с случайными перестановками
    vector<int> getNearlySortedArray(int size, int swaps) const {
        if (size > maxLength) size = maxLength;
        vector<int> arr(baseSortedArray.begin(), baseSortedArray.begin() + size);

        uniform_int_distribution<int> dist(0, size - 1);
        FOR(i, 0, swaps) {
            swap(arr[dist(gen)], arr[dist(gen)]);
        }
        return arr;
    }
};


class SortTester {
private:
    const int numTrials = 10;

public:
    ld measureMergeSortTime(vector<int>& originalArray) {
        ld totalTime = 0.0;
        FOR(trial, 0, numTrials) {
            vector<int> arrayToSort = originalArray;
            auto start = chrono::high_resolution_clock::now();
            mergeSort(arrayToSort, 0, arrayToSort.size() - 1);
            auto end = chrono::high_resolution_clock::now();
            totalTime += chrono::duration<ld, micro>(end - start).count();
        }
        return totalTime / numTrials;
    }

    ld measureHybridMergeSortTime(vector<int>& originalArray, int threshold) {
        ld totalTime = 0.0;
        FOR(trial, 0, numTrials) {
            vector<int> arrayToSort = originalArray;
            auto start = chrono::high_resolution_clock::now();
            hybridMergeSort(arrayToSort, 0, arrayToSort.size() - 1, threshold);
            auto end = chrono::high_resolution_clock::now();
            totalTime += chrono::duration<ld, micro>(end - start).count();
        }
        return totalTime / numTrials; // В микросекундах
    }
};

void merge(vector<int>& arr, int left, int mid, int right) {
    int n1 = mid - left + 1;
    int n2 = right - mid;

    vector<int> L(n1), R(n2);

    FOR(i, 0, n1) L[i] = arr[left + i];
    FOR(j, 0, n2) R[j] = arr[mid + 1 + j];

    int i = 0, j = 0, k = left;
    while (i < n1 && j < n2) {
        if (L[i] <= R[j]) {
            arr[k++] = L[i++];
        } else {
            arr[k++] = R[j++];
        }
    }
    while (i < n1) arr[k++] = L[i++];
    while (j < n2) arr[k++] = R[j++];
}
```

```cpp
void mergeSort(vector<int>& arr, int left, int right) {
    if (left < right) {
        int mid = left + (right - left) / 2;
        mergeSort(arr, left, mid);
        mergeSort(arr, mid + 1, right);
        merge(arr, left, mid, right);
    }
}

void insertionSort(vector<int>& arr, int left, int right) {
    FOR(i, left + 1, right + 1) {
        int key = arr[i];
        int j = i - 1;
        while (j >= left && arr[j] > key) {
            arr[j + 1] = arr[j];
            j--;
        }
        arr[j + 1] = key;
    }
}

void hybridMergeSort(vector<int>& arr, int left, int right, int threshold) {
    if (right - left + 1 < threshold) {
        insertionSort(arr, left, right);
        return;
    }
    if (left < right) {
        int mid = left + (right - left) / 2;
        hybridMergeSort(arr, left, mid, threshold);
        hybridMergeSort(arr, mid + 1, right, threshold);
        merge(arr, left, mid, right);
    }
}

```

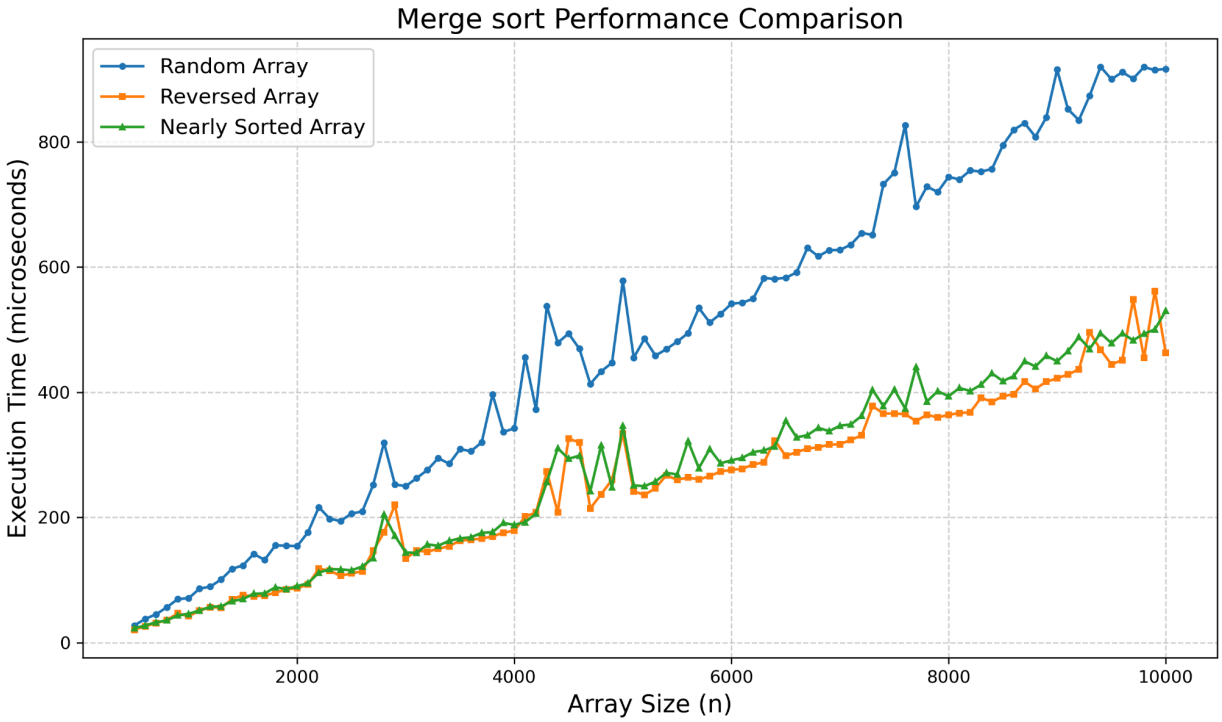
```cpp
void runExperiments() {
    ArrayGenerator generator(42);
    SortTester tester;

    const string merge_random_file = "merge_sort_random.csv";
    const string merge_reversed_file = "merge_sort_reversed.csv";
    const string merge_nearly_sorted_file = "merge_sort_nearly_sorted.csv";
    const string hybrid_random_file = "hybrid_sort_random.csv";
    const string hybrid_reversed_file = "hybrid_sort_reversed.csv";
    const string hybrid_nearly_sorted_file = "hybrid_sort_nearly_sorted.csv";

    ofstream f_merge_random(merge_random_file);
    ofstream f_merge_reversed(merge_reversed_file);
    ofstream f_merge_nearly_sorted(merge_nearly_sorted_file);
    ofstream f_hybrid_random(hybrid_random_file);
    ofstream f_hybrid_reversed(hybrid_reversed_file);
    ofstream f_hybrid_nearly_sorted(hybrid_nearly_sorted_file);

    if (!f_merge_random.is_open() || !f_merge_reversed.is_open() || !f_merge_nearly_sorted.is_open() ||
        !f_hybrid_random.is_open() || !f_hybrid_reversed.is_open() || !f_hybrid_nearly_sorted.is_open()) {
        cerr << "Ошибка при открытии файлов !!" << endl;
        return;
    }

    f_merge_random << "n,time_us\n";
    f_merge_reversed << "n,time_us\n";
    f_merge_nearly_sorted << "n,time_us\n";

    vector<int> thresholds = {5, 10, 15, 20, 30, 50};
    f_hybrid_random << "n";
    for (int t : thresholds) f_hybrid_random << ",time_us_th_" << t;
    f_hybrid_random << "\n";

    f_hybrid_reversed << "n";
    for (int t : thresholds) f_hybrid_reversed << ",time_us_th_" << t;
    f_hybrid_reversed << "\n";
```

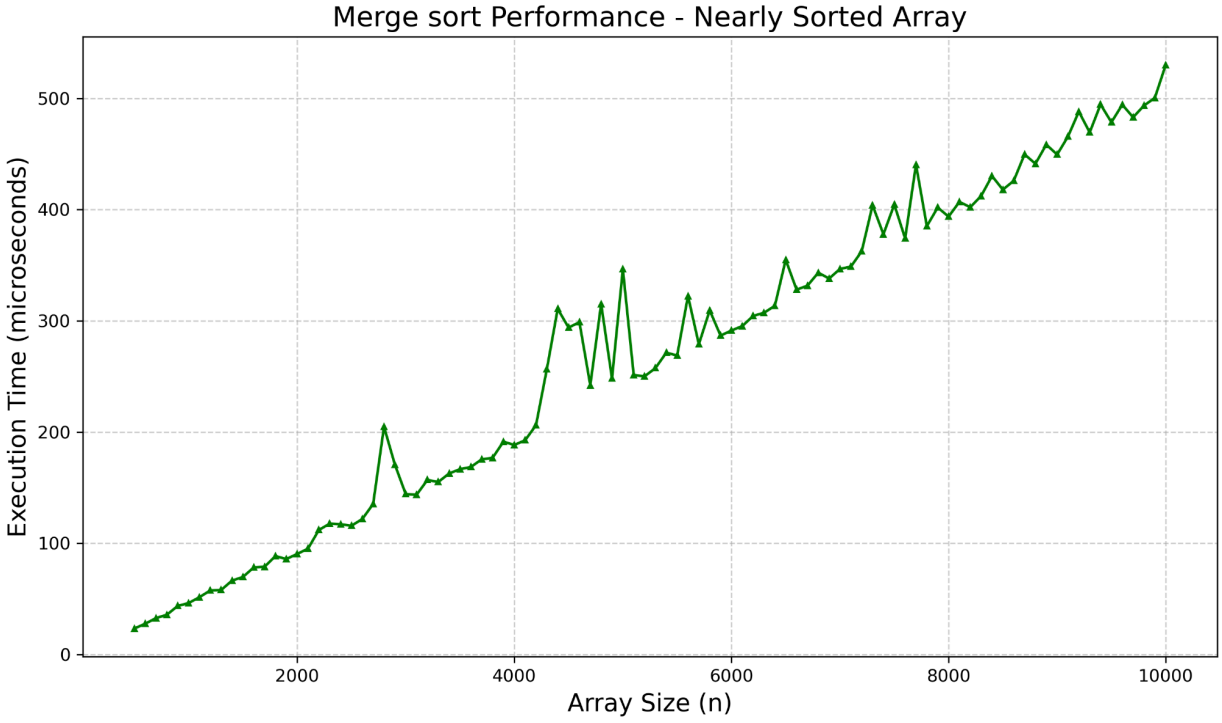
```
    f_hybrid_nearly_sorted << "n";
    for (int t : thresholds) f_hybrid_nearly_sorted << ",time_us_th_" << t;
    f_hybrid_nearly_sorted << "\n";

    for (int n = 500; n <= 10000; n += 100) { // Можно заменить на 100000 позже
        cout << "Processing size: " << n << endl;
        auto random_arr = generator.getRandomArray(n);
        auto reversed_arr = generator.getReversedArray(n);
        auto nearly_sorted_arr = generator.getNearlySortedArray(n, n / 20);

        // merge - замеры
        ld time_merge_random = tester.measureMergeSortTime(random_arr);
        ld time_merge_reversed = tester.measureMergeSortTime(reversed_arr);
        ld time_merge_nearly_sorted = tester.measureMergeSortTime(nearly_sorted_arr);

        f_merge_random << n << "," << fixed << setprecision(2) << time_merge_random << "\n";
        f_merge_reversed << n << "," << fixed << setprecision(2) << time_merge_reversed << "\n";
        f_merge_nearly_sorted << n << "," << fixed << setprecision(2) << time_merge_nearly_sorted << "\n";

        // hybrid - замеры
        f_hybrid_random << n;
        f_hybrid_reversed << n;
        f_hybrid_nearly_sorted << n;

        for (int threshold : thresholds) {
            ld time_hybrid_random = tester.measureHybridMergeSortTime(random_arr, threshold);
            ld time_hybrid_reversed = tester.measureHybridMergeSortTime(reversed_arr, threshold);
            ld time_hybrid_nearly_sorted = tester.measureHybridMergeSortTime(nearly_sorted_arr, threshold);
            f_hybrid_random << "," << fixed << setprecision(2) << time_hybrid_random;
            f_hybrid_reversed << "," << fixed << setprecision(2) << time_hybrid_reversed;
            f_hybrid_nearly_sorted << "," << fixed << setprecision(2) << time_hybrid_nearly_sorted;
        }
        f_hybrid_random << "\n";
        f_hybrid_reversed << "\n";
        f_hybrid_nearly_sorted << "\n";
    }
```

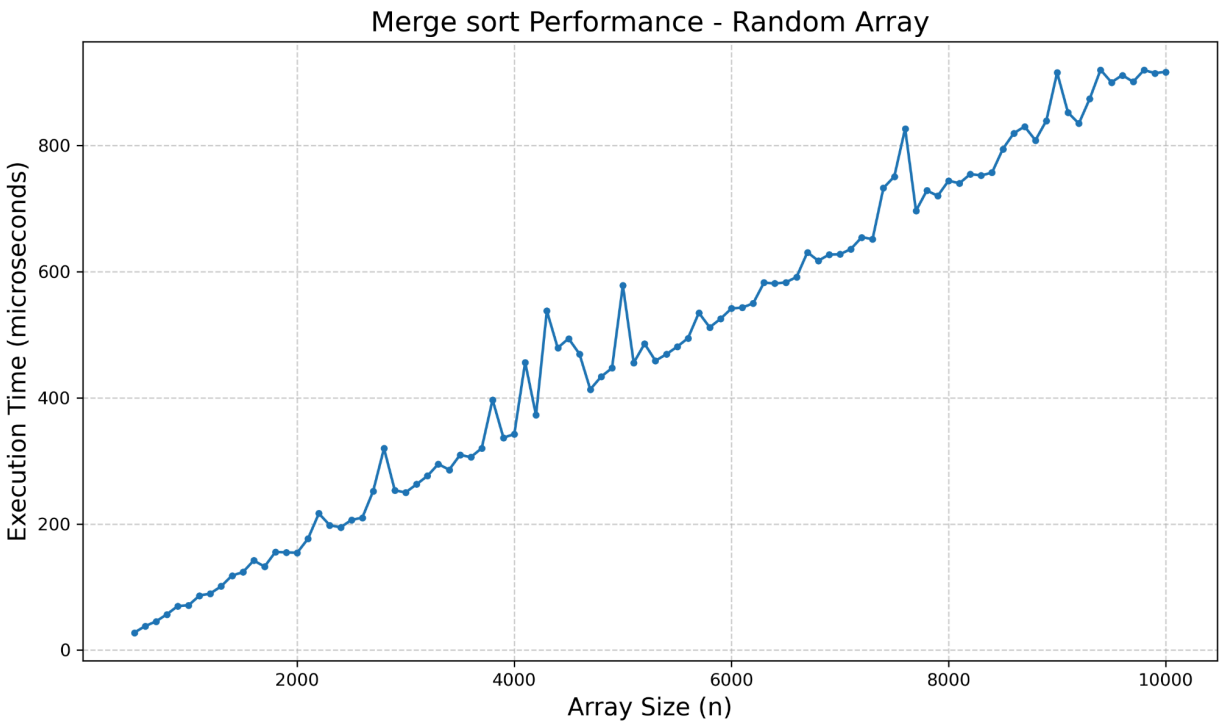
```
    f_merge_random.close();
    f_merge_reversed.close();
    f_merge_nearly_sorted.close();
    f_hybrid_random.close();
    f_hybrid_reversed.close();
    f_hybrid_nearly_sorted.close();
}

int main() {
    cin.tie(nullptr)->sync_with_stdio(false);
    runExperiments();
    return 0;
}
```

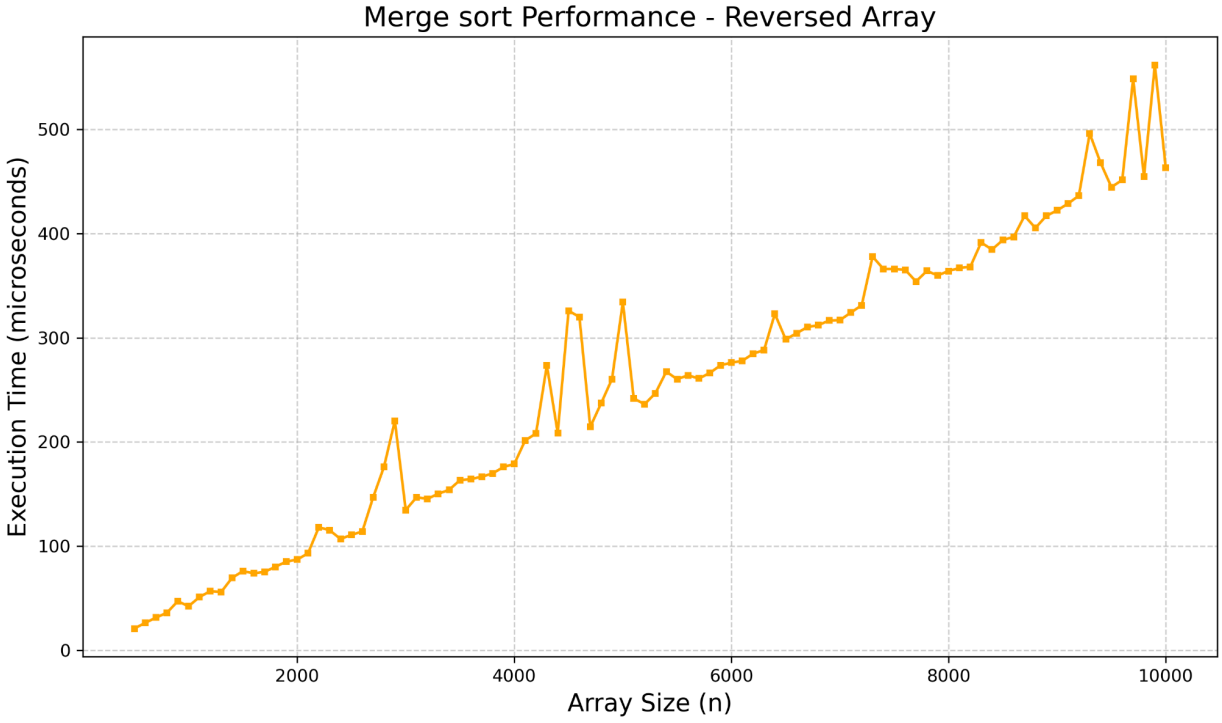
Эмпирический анализ стандартного алгоритма MERGE SORT:

- merge_sort_comparison.png - общий график, показывает зависимость времени выполнения стандартного алгоритма merge sort от размера массива n для 3 разных случаев (случайный, обратно отсортированный, почти отсортированный)
- merge_sort_random.png, merge_sort_reversed.png, merge_sort_nearly_sorted.png - три отдельных графика для разных случаев


Merge sort Performance Comparison
Random Array
Reversed Array
Nearly Sorted Array
Execution Time (microseconds)
0
200
400
600
800
2000
4000
6000
8000
10000
Array Size (n)

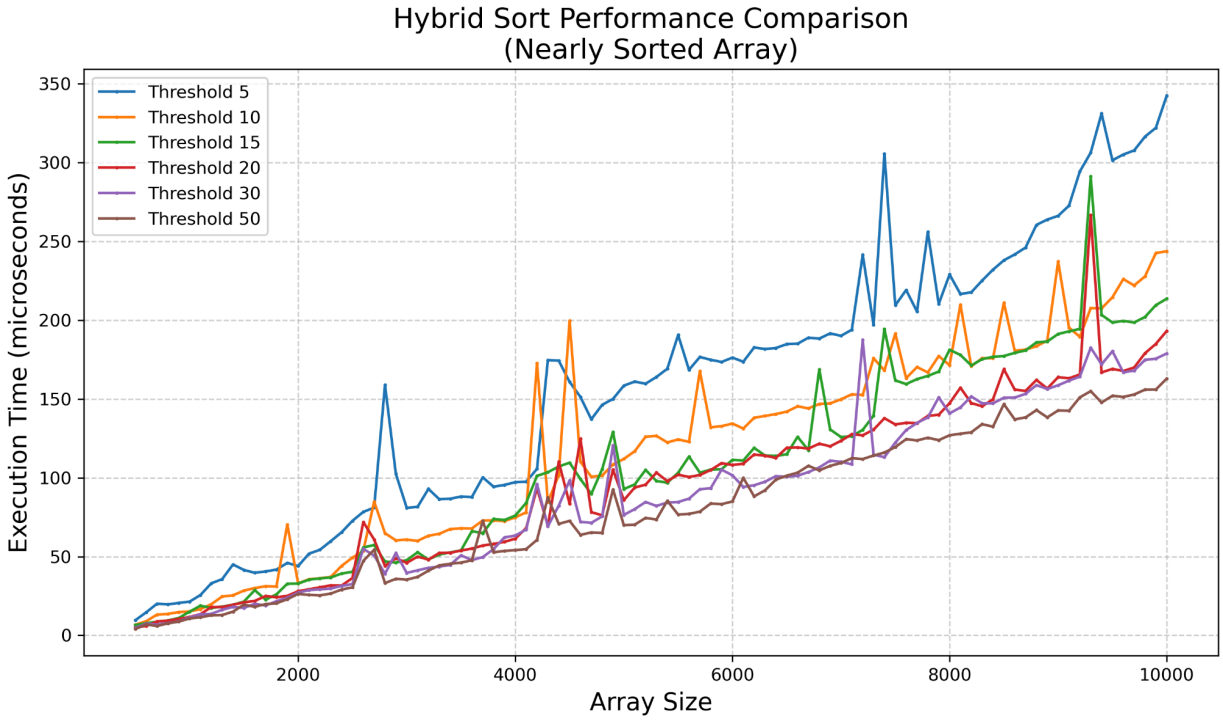

Merge sort Performance - Nearly Sorted Array
Execution Time (microseconds)
0
100
200
300
400
500
2000
4000
6000
8000
10000
Array Size (n)

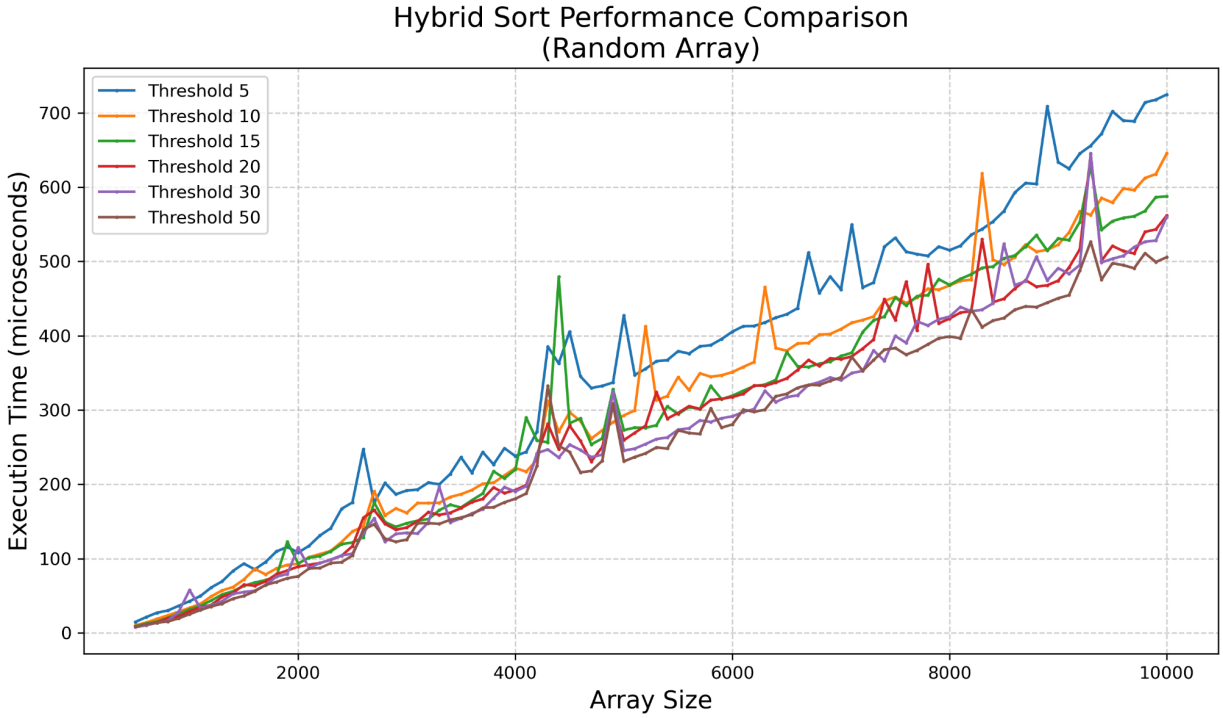

Merge sort Performance - Random Array
Execution Time (microseconds)
0
200
400
600
800
2000
4000
6000
8000
10000
Array Size (n)

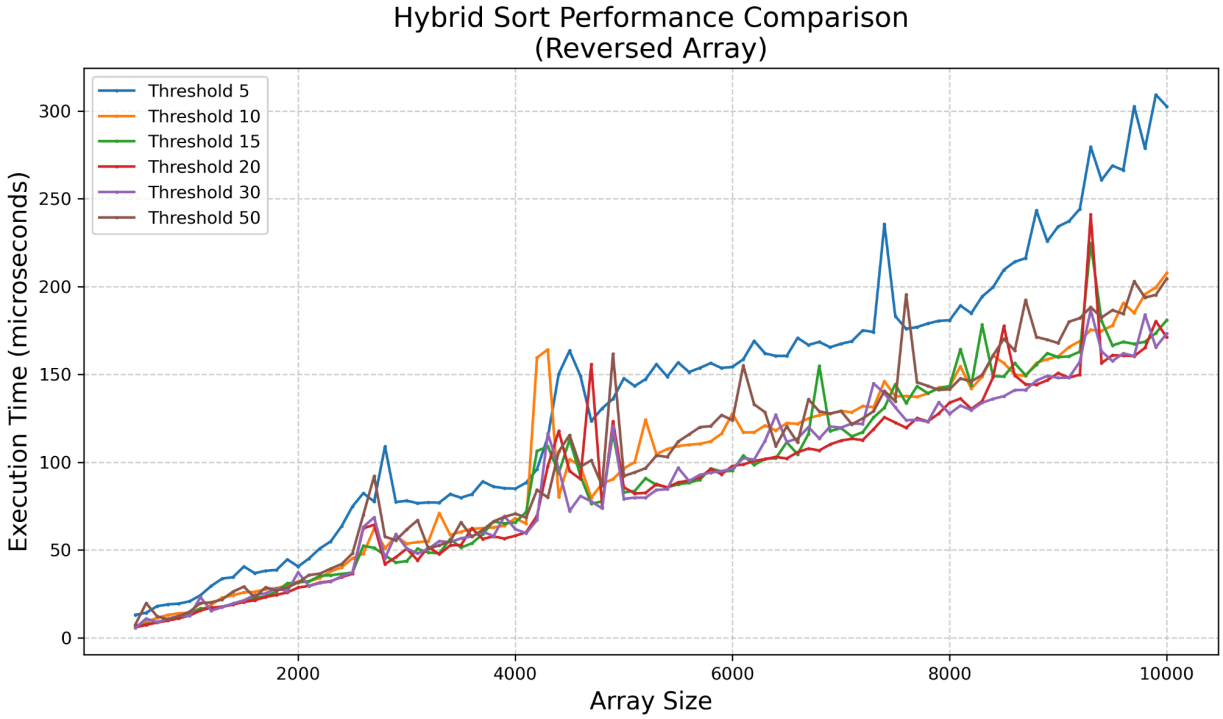

Merge sort Performance - Reversed Array
Execution Time (microseconds)
0
100
200
300
400
500
2000
4000
6000
8000
10000
Array Size (n)

Эмпирический анализ гибридного алгоритма MERGE+INSERTION SORT:

- hybrid_threshold_comparison_random.png, hybrid_threshold_comparison_reversed.png, hybrid_threshold_comparison_nearly_sorted.png - три отдельных графика, показывающих сравнение времени выполнения merge+insertion sort при разных значениях порога threshold на разных типах массивов (случайный, обратно отсортированный, почти отсортированный) в зависимости от его размера n

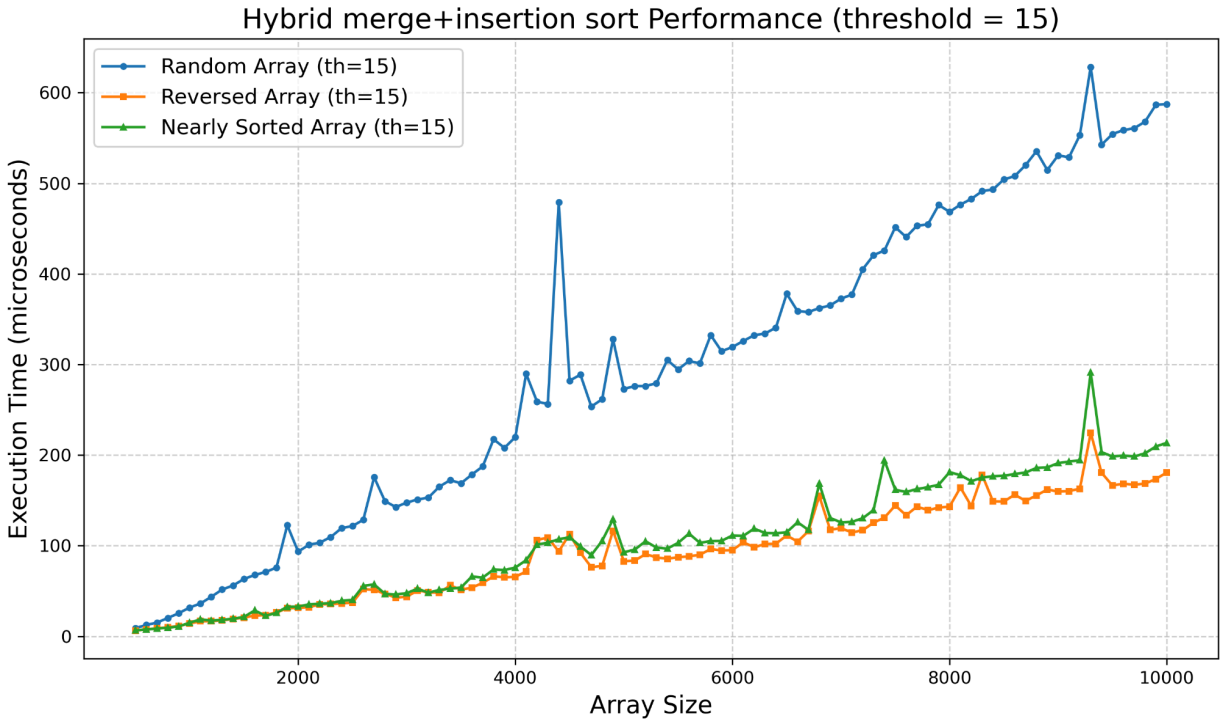






- hybrid_threshold_comparison_all.png - то же самое, просто объединенный график из 3 подграфов для наглядности

- hybrid_sort_comparison_th_15.png - сравнение времени выполнения merge+insertion sort при фиксированном threshold = 15 на массивах разных типов в зависимости от их размера n



Сравнительный анализ:

1) Стандартный merge sort:

- По монотонному росту времени выполнения с увеличением размера массива n видно стабильную производительность, близкую к теоретической оценке O(n log n)
- Время выполнения алгоритма сильно зависит от исходной упорядоченности массива:
    - Наиболее быстрый случай - почти отсортированные массивы, тк при слиянии подмассивов операция merge может быть оптимизирована за счет уже частичной упорядоченности
    - Средний случай - обратно отсортированные массивы, тк после первого разделения массива возникают ситуации, когда один из подмассивов состоит из больших чисел, а другой из маленьких, что позволяет эффективно выполнять слияние

- Наиболее медленный случай - случайные массивы, тк очевидно, отсутствие какой-либо упорядоченности приводит к максимальному количеству сравнений и перестановок на каждом уровне слияния

2) Merge+insertion sort:

- При использовании любого порога (от 5 до 50) merge+insertion выполняется значительно быстрее, чем просто merge sort, особенно на массивах среднего и небольшого размера, тк на уровнях рекурсии, где размер подмассива становится меньше порога, insertion sort заменяет merge
- Влияние типа массива:
    - Наиболее быстрый случай - почти отсортированные массивы, тк на маленьких подмассивах используется insertion sort
    - Средний случай - обратно отсортированные массивы
    - Наиболее медленный случай - случайные массивы - здесь гибридный алгоритм работает медленнее, чем на других типах данных, но все равно значительно быстрее, чем стандартный merge sort

3) Оптимальное пороговое значение:

- Можно предположить, что оптимальное значение threshold находится в районе 20, кривые для threshold=20 и threshold=15 часто находятся внизу, что говорит о наименьшем времени выполнения
- Гибридный алгоритм начинает работать медленнее стандартного merge при больших значениях threshold (например, 50) на небольших и средних массивах (слишком большой порог приводит к использованию insertion sort на слишком больших подмассивах)