A2. Задача трех кругов

Работу выполнил Девятов Денис БПИ-238

Пункт 1.

293143032 – id посылки

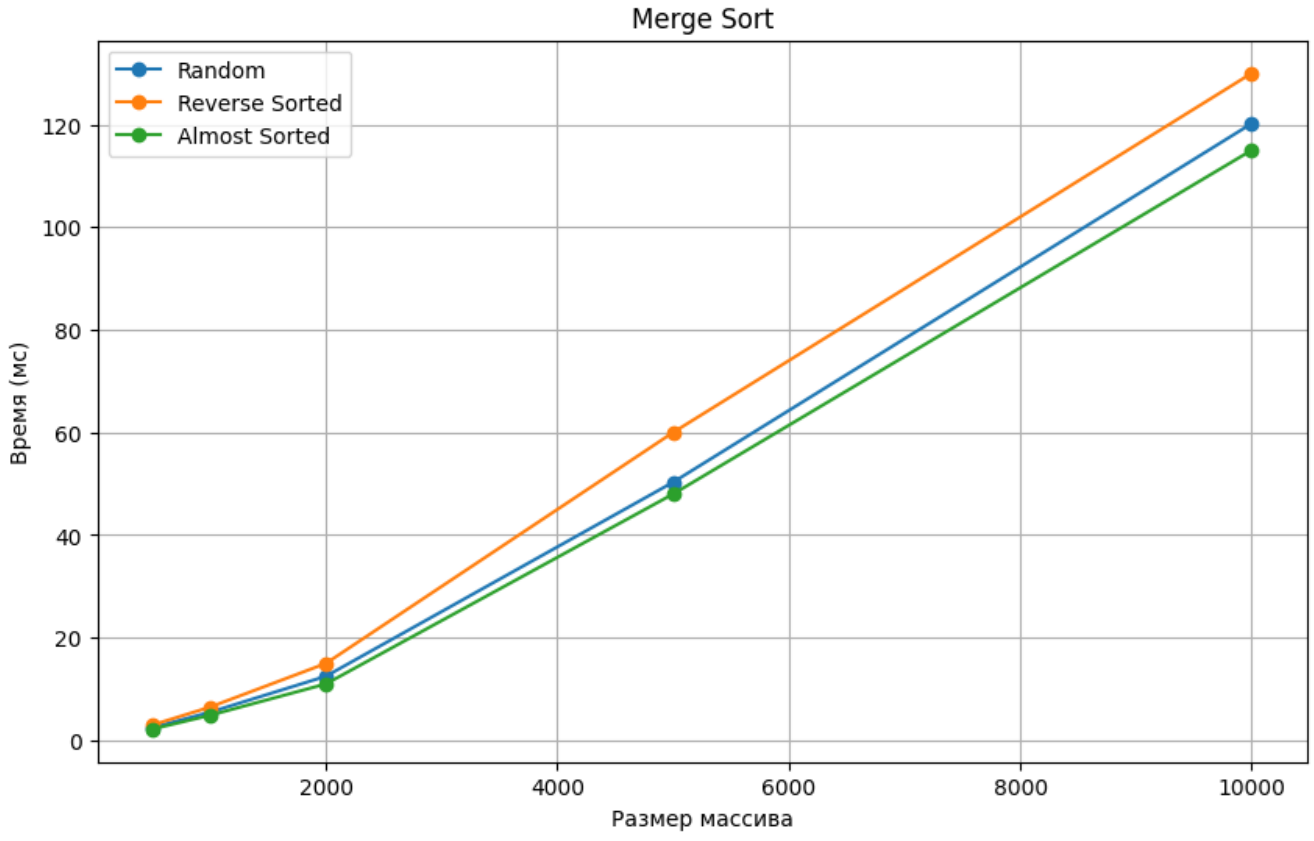
https://github.com/BelyLandy/set_3

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

class ArrayGenerator {
public:
    ArrayGenerator(int maxSize, int valueRange)
        : maxSize(maxSize), valueRange(valueRange), engine(std::random_device{}()) {
        generateBaseArray();
    }

    // Генерация массива случайных значений
    std::vector<int> getRandomArray(int size) {
        if (size > maxSize) {
            throw std::invalid_argument("Size exceeds maximum allowed size.");
        }

        std::vector<int> randomArray(baseArray.begin(), baseArray.begin() + size);
        std::shuffle(randomArray.begin(), randomArray.end(), engine);
        return randomArray;
    }

    // Генерация массива, отсортированного в обратном порядке
```

```
    std::vector<int> getReversedArray(int size) {
        if (size > maxSize) {
            throw std::invalid_argument("Size exceeds maximum allowed size.");
        }

        std::vector<int> reversedArray(baseArray.begin(), baseArray.begin() + size);
        std::sort(reversedArray.begin(), reversedArray.end(), std::greater<int>());
        return reversedArray;
    }

    // Генерация "почти" отсортированного массива
    std::vector<int> getNearlySortedArray(int size, int swaps) {
        if (size > maxSize) {
            throw std::invalid_argument("Size exceeds maximum allowed size.");
        }

        std::vector<int> nearlySortedArray(baseArray.begin(), baseArray.begin() + size);
        std::sort(nearlySortedArray.begin(), nearlySortedArray.end());

        for (int i = 0; i < swaps; ++i) {
            int index1 = getRandomIndex(size);
            int index2 = getRandomIndex(size);
            std::swap(nearlySortedArray[index1], nearlySortedArray[index2]);
        }

        return nearlySortedArray;
    }

private:
    int maxSize;
    int valueRange;
    std::vector<int> baseArray;
```

```
    std::mt19937 engine;

    void generateBaseArray() {
        std::uniform_int_distribution<int> dist(0, valueRange);
        baseArray.resize(maxSize);
        for (int i = 0; i < maxSize; ++i) {
            baseArray[i] = dist(engine);
        }
    }

    int getRandomIndex(int size) {
        std::uniform_int_distribution<int> dist(0, size - 1);
        return dist(engine);
    }
};
```

Пункт 2.

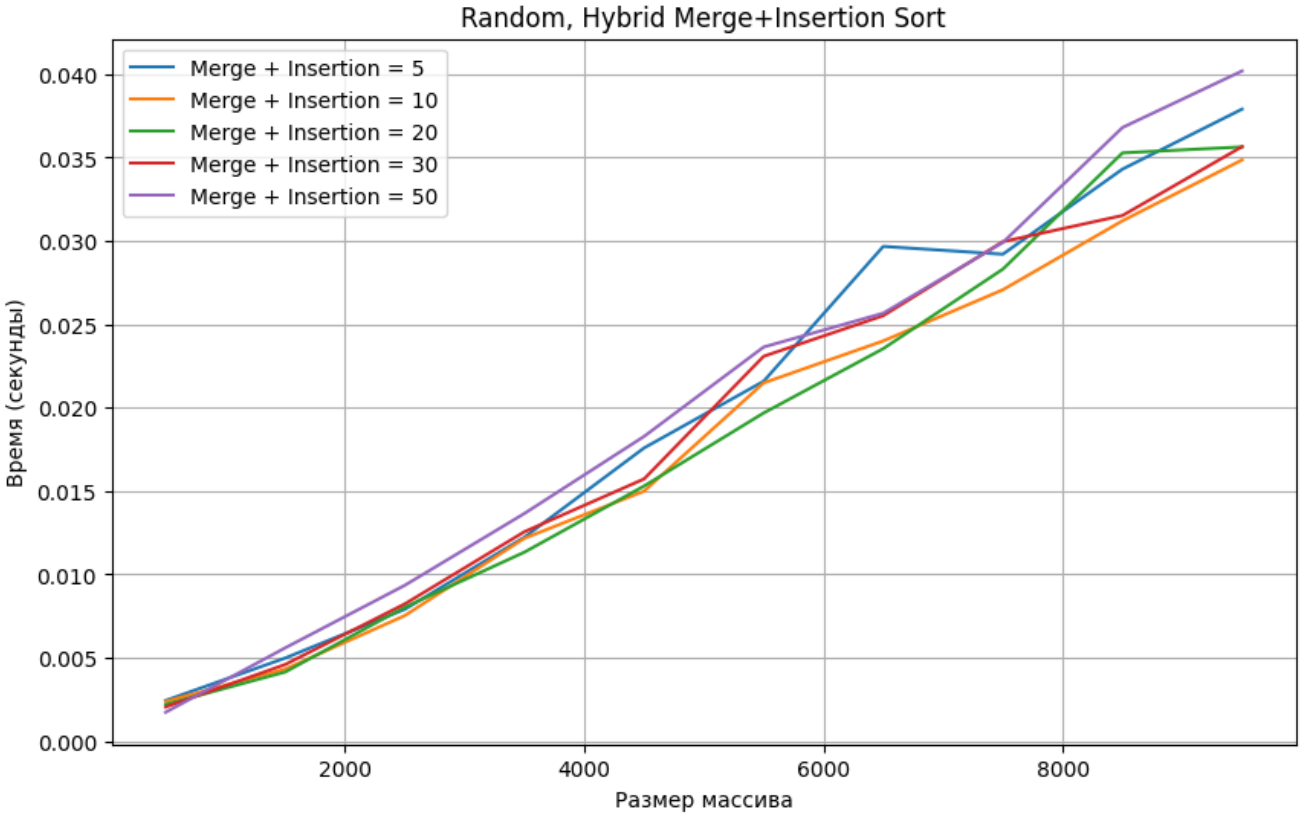


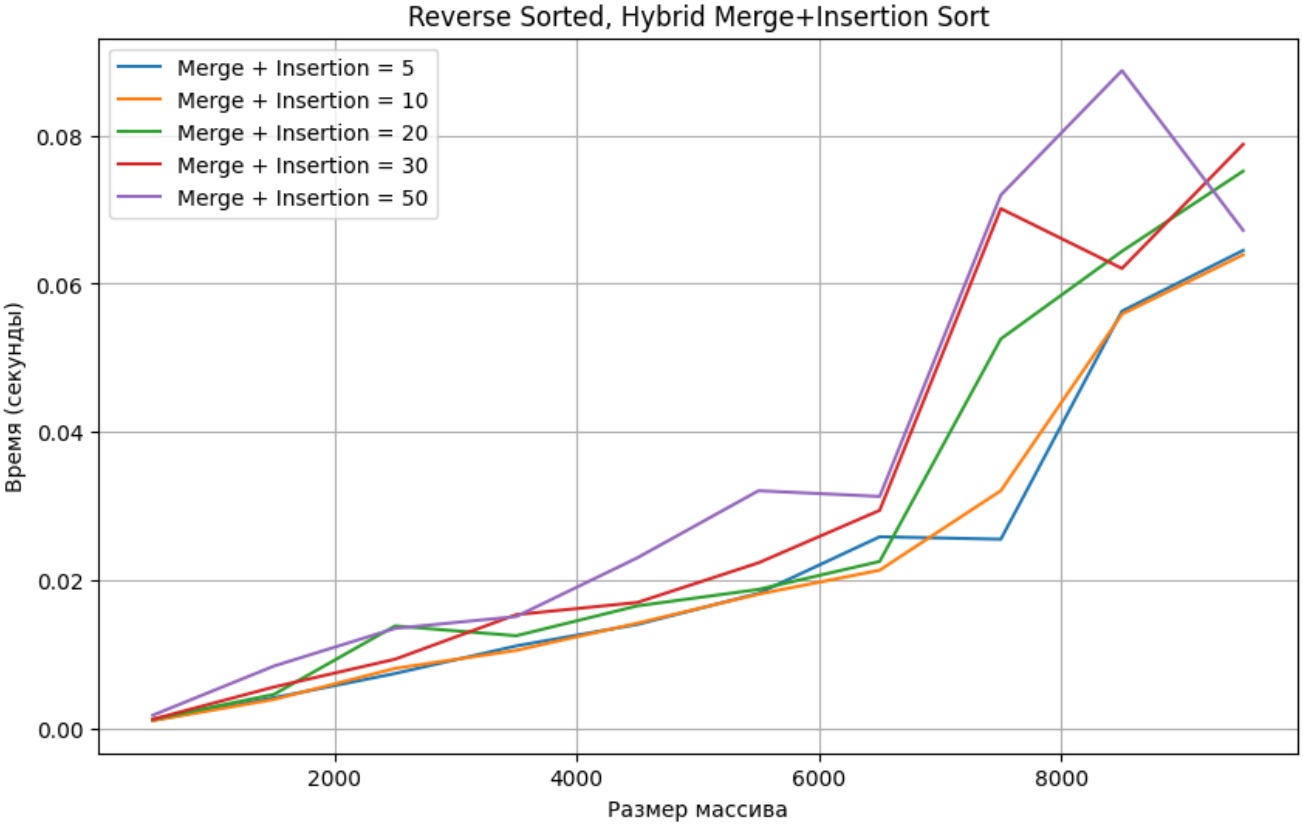
Пункт 3.

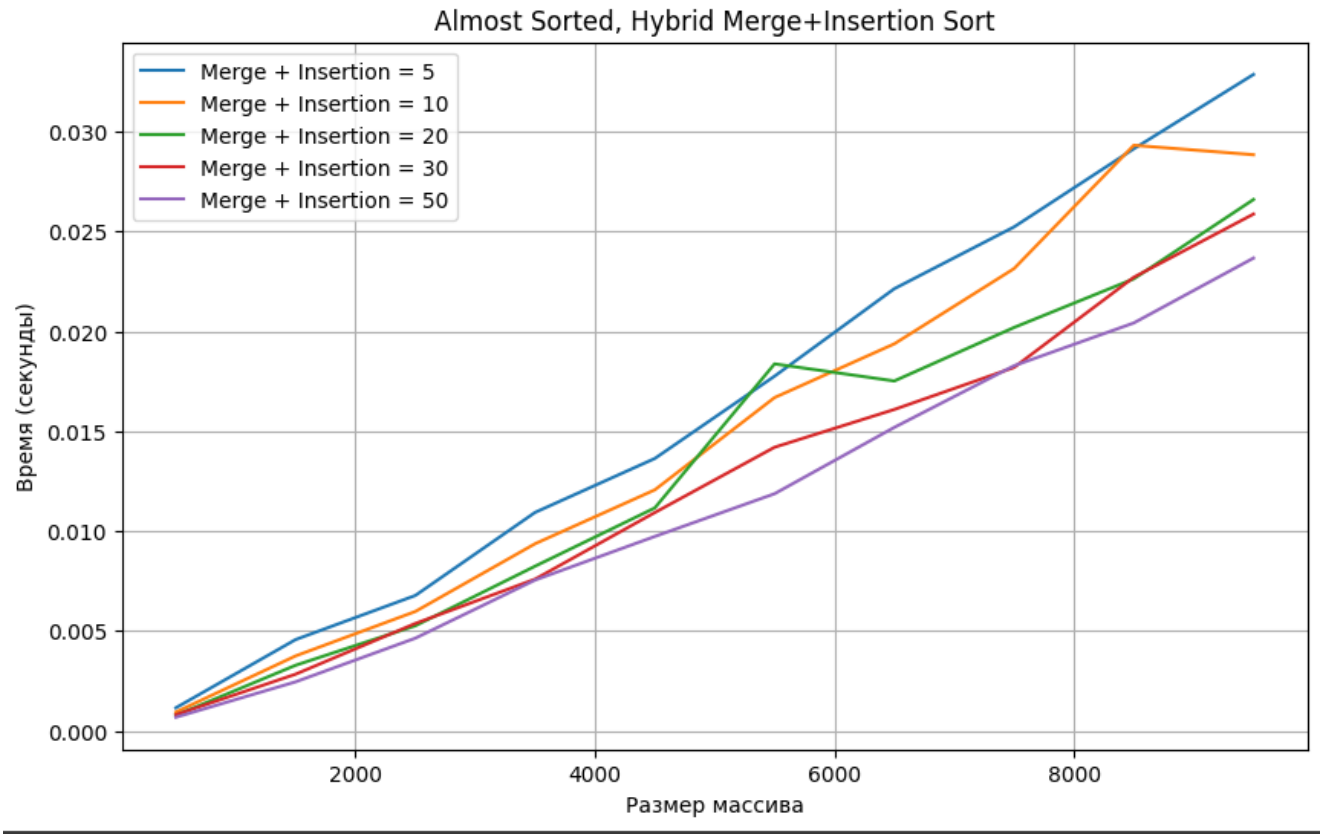
```
#include <iostream>
#include <vector>
#include <chrono>
#include <algorithm>
#include <random>
#include <iomanip>

class SortTester {
public:
    static double measureHybridMergeSort(std::vector<int> array, int threshold) {
        auto start = std::chrono::high_resolution_clock::now();
        hybridMergeSort(0, array.size(), array, threshold);
        auto end = std::chrono::high_resolution_clock::now();
        std::chrono::duration<double> elapsed = end - start;
        return elapsed.count();
    }

private:
    static void insertionSort(int left, int right, std::vector<int> &array) {
        for (int i = left; i < right; ++i) {
            int current = array[i];
            int j = i - 1;
            while (j >= left && array[j] > current) {
                array[j + 1] = array[j];
                --j;
            }
            array[j + 1] = current;
        }
    }

    static void mergeSegments(int left, int right, std::vector<int> &array) {
```

```
int mid = (left + right) / 2;
int leftSize = mid - left;
int rightSize = right - mid;
std::vector<int> leftSegment(leftSize), rightSegment(rightSize);

for (int i = 0; i < leftSize; ++i) {
   leftSegment[i] = array[left + i];
}

for (int j = 0; j < rightSize; ++j) {
   rightSegment[j] = array[mid + j];
}

int i = 0, j = 0, k = left;

while (i < leftSize && j < rightSize) {
   if (leftSegment[i] <= rightSegment[j]) {
      array[k] = leftSegment[i];
      ++i;
   } else {
      array[k] = rightSegment[j];
      ++j;
   }
   ++k;
}

while (i < leftSize) {
   array[k] = leftSegment[i];
   ++i;
   ++k;
}
```

```
        while (j < rightSize) {
            array[k] = rightSegment[j];
            ++j;
            ++k;
        }
    }

    static void hybridMergeSort(int left, int right, std::vector<int> &array, int threshold) {
        if (left + 1 >= right) {
            return;
        }
        if (right - left <= threshold) {
            insertionSort(left, right, array);
            return;
        }

        int mid = (left + right) / 2;

        hybridMergeSort(left, mid, array, threshold);
        hybridMergeSort(mid, right, array, threshold);

        mergeSegments(left, right, array);
    }
};
```


Random, Hybrid Merge+Insertion Sort
Merge + Insertion = 5
Merge + Insertion = 10
Merge + Insertion = 20
Merge + Insertion = 30
Merge + Insertion = 50
Время (секунды)
0.000
0.005
0.010
0.015
0.020
0.025
0.030
0.035
0.040
2000
4000
6000
8000
Размер массива


Reverse Sorted, Hybrid Merge+Insertion Sort
Merge + Insertion = 5
Merge + Insertion = 10
Merge + Insertion = 20
Merge + Insertion = 30
Merge + Insertion = 50
Время (секунды)
0.00
0.02
0.04
0.06
0.08
2000
4000
6000
8000
Размер массива



Пункт 4.

Сравнительный анализ алгоритмов сортировки.

Гибридный алгоритм MERGE+INSERTION SORT эффективен на малых и средних массивах (до 1000-2000 элементов), где сортировка вставками ускоряет выполнение на маленьких подмассивах.

С увеличением размера массива и порога переключения на сортировку вставками его эффективность снижается, и он может стать медленнее стандартного Merge Sort.

Тем временем Merge Sort стабильный и предсказуемый для всех размеров массивов с временем работы O(n log n). Идеален для больших массивов (свыше 1000-2000 элементов), где его производительность не зависит от переключения на другую сортировку.

Выводы.

Гибридный алгоритм быстрее для малых и почти отсортированных массивов, но для больших массивов его производительность хуже.

Стандартный Merge Sort эффективен на больших данных и обеспечивает стабильные результаты.