SET3 A2 Анализ MERGE+INSERTION SORT

ID посылки: 349169926
GitHub: https://github.com/nriarovitsyna/Algorithms-and-data-structures/tree/main/SET3/A2

Файлы решения:
- **A2i.cpp** - общее решение
- **A2.cpp** - решение задачи A2
- **A2.py** - построение графиков
- **random_resulats(n).csv** - результаты эксперимента случайных массивов с параметром переключения на INSERTION SORT = n
- **reverse_results(n).csv** - результаты эксперимента обратно отсортированных массивов с параметром переключения на INSERTION SORT = n
- **almost_results(n).csv** - результаты эксперимента почти отсортированных массивов с параметром переключения на INSERTION SORT = n
- **results(n).png** - графики сравнения MERGE+INSERTION SORT и MERGE SORT для каждого параметра n
- **comparison_all_params.png** - график сравнения для всех параметров MERGE+INSERTION SORT
- **A2.pdf** - сравнительный анализ

Реализация класса ArrayGenerator

```
class ArrayGenerator {
private:
    std::mt19937 gen;
    std::uniform_int_distribution<int> dist;

public:
    ArrayGenerator() : gen(std::random_device{}()), dist(0, 10000) {}

    std::vector<int> generateRandomArray(int size) {
        std::vector<int> arr(size);
        for (int i = 0; i < size; i++) {
            arr[i] = dist(gen);
        }
        return arr;
    }

    std::vector<int> generateReverseSortedArray(int size) {
        std::vector<int> arr(size);
        for (int i = 0; i < size; i++) {
            arr[i] = size - i;
        }
        return arr;
    }

    std::vector<int> generateAlmostSortedArray(int size) {
        int swap_cnt = size / 1000;
        std::vector<int> arr(size);
        for (int i = 0; i < size; i++) {
```

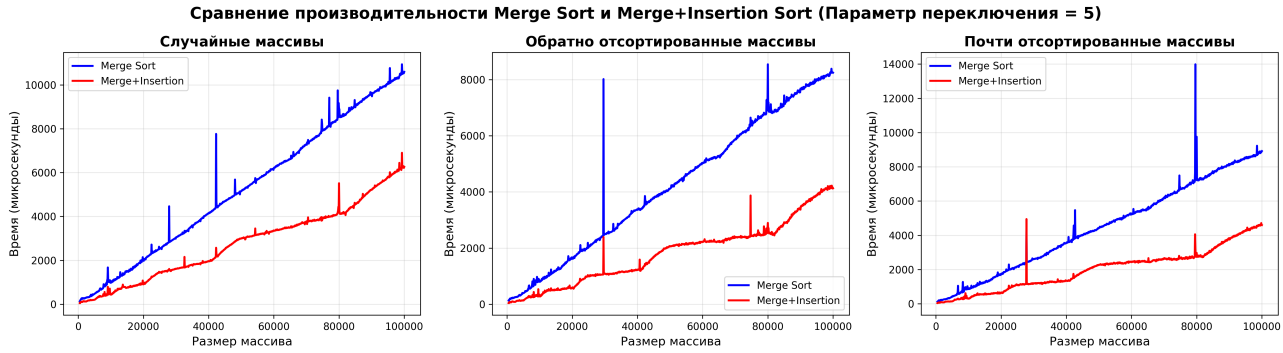
```
            arr[i] = i + 1;
        }

        for (int i = 0; i < swap_cnt; i++) {
            int idx1 = std::uniform_int_distribution<int>(0, size - 1)(gen);
            int idx2 = std::uniform_int_distribution<int>(0, size - 1)(gen);
            std::swap(arr[idx1], arr[idx2]);
        }

        return arr;
    }

    std::vector<int> getSubarray(const std::vector<int>& arr, int size) {
        if (size > arr.size()) {
            return arr;
        }
        return std::vector<int>(arr.begin(), arr.begin() + size);
    }
};
```

Реализация класса SortTester

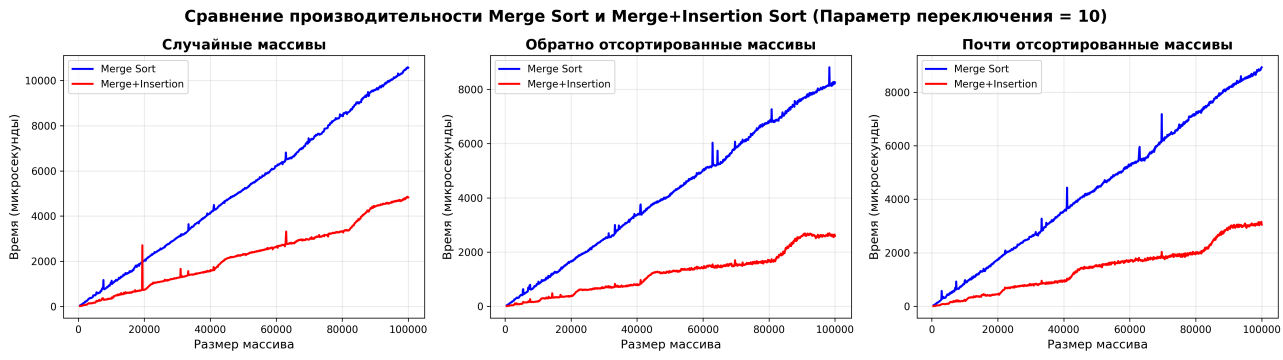
```
class SortTester {
private:
    ArrayGenerator generator;
    int num;
    int switch_param;

public:
    SortTester(int switch_param, int num = 5) : num(num), switch_param(switch_param) {}

    long long testMerge(const std::vector<int>& original) {
        std::vector<int> arr = original;
        auto start = std::chrono::high_resolution_clock::now();

        MERGE_SORT(arr, 0, arr.size() - 1);

        auto elapsed = std::chrono::high_resolution_clock::now() - start;
        return std::chrono::duration_cast<std::chrono::microseconds>(elapsed).count();
    }

    long long testMergeInsertion(const std::vector<int>& original) {
        std::vector<int> arr = original;
        auto start = std::chrono::high_resolution_clock::now();

        MERGE_INSERTION(arr, 0, arr.size() - 1, switch_param);

        auto elapsed = std::chrono::high_resolution_clock::now() - start;
        return std::chrono::duration_cast<std::chrono::microseconds>(elapsed).count();
    }

    long long getMeanRuntime(std::vector<int> arr, bool MergeInsertion = true) {
```

```
        long long total_time = 0;
        for (int i = 0; i < num; i++) {
            std::vector<int> test_arr = arr;

            if (MergeInsertion) {
                total_time += testMergeInsertion(test_arr);
            } else {
                total_time += testMerge(test_arr);
            }
        }

        return total_time / num;
    }

    bool isSorted(const std::vector<int>& arr) {
        for (size_t i = 1; i < arr.size(); i++) {
            if (arr[i] < arr[i - 1]) {
                return false;
            }
        }
        return true;
    }
};
```

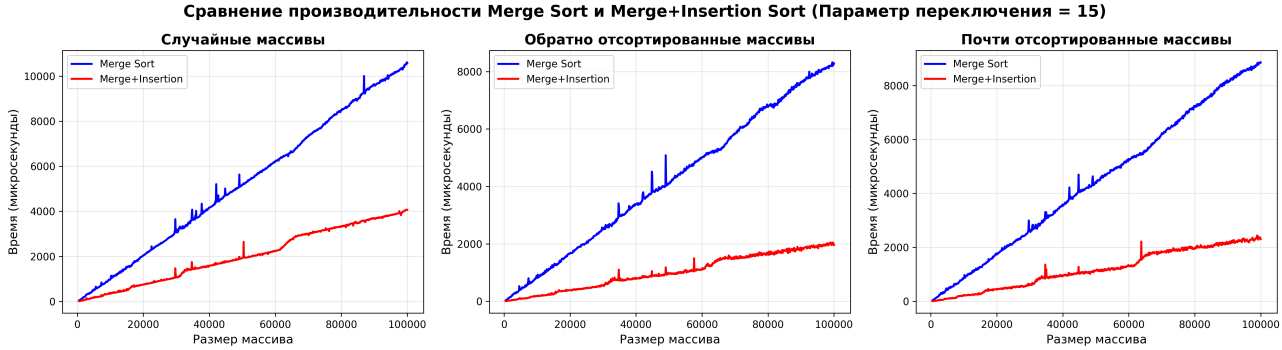
Описание результатов эксперимента



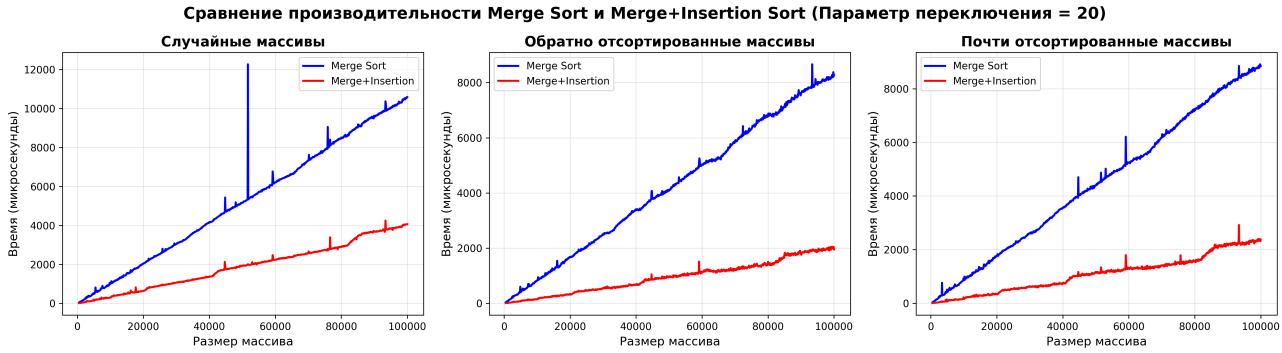
Сравнительный анализ трех типов массивов выявил различные показатели производительности гибридного алгоритма. На случайных массивах наблюдается стабильное и возрастающее преимущество Merge+Insertion Sort, особенно выраженное при N > 20000. Для обратно отсортированных массивов характерна нестабильная динамика, в которой Merge+Insertion Sort выравнивается по эффективности со средним показателем стандартной реализацией Merge Sort, но не превосходит ее. На почти отсортированных массивах при N равным приблизительно 30000 Merge Sort показывает наименьшие затраты по времени, далее гибридный алгоритм снова показывает наилучшую производительность.



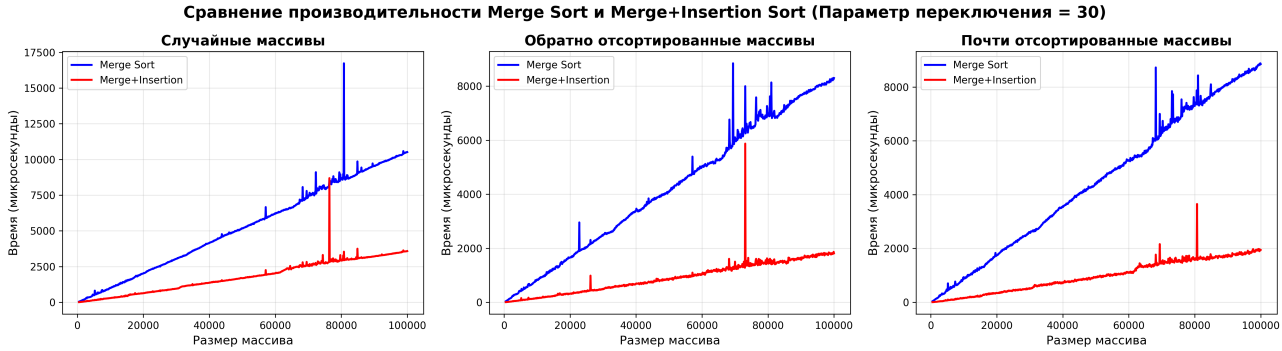
Общая тенденция для параметра 10 такова: гибридный алгоритм Merge+Insertion Sort демонстрирует преимущество над стандартным Merge Sort, особенно это заметно с ростом N. Однако, на графики случайного массива видно, что при N приблизительно 20000 гибридный алгоритм резко демонстрирует худшую производительность, но позже общая тенденция сохраняется.



При использовании параметра 15 Merge+Insertion Sort демонстрирует стабильное преимущество над стандартным Merge Sort для любых массивов, с ростом N преимущество становиться более выраженным.

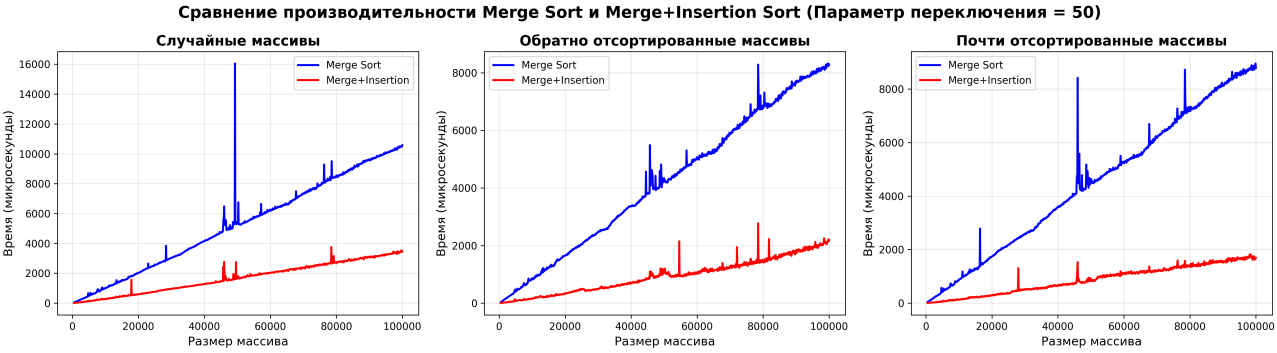


При использовании параметра 20 Merge+Insertion Sort демонстрирует стабильное преимущество над стандартным Merge Sort для любых массивов, с ростом N преимущество становиться выраженным.

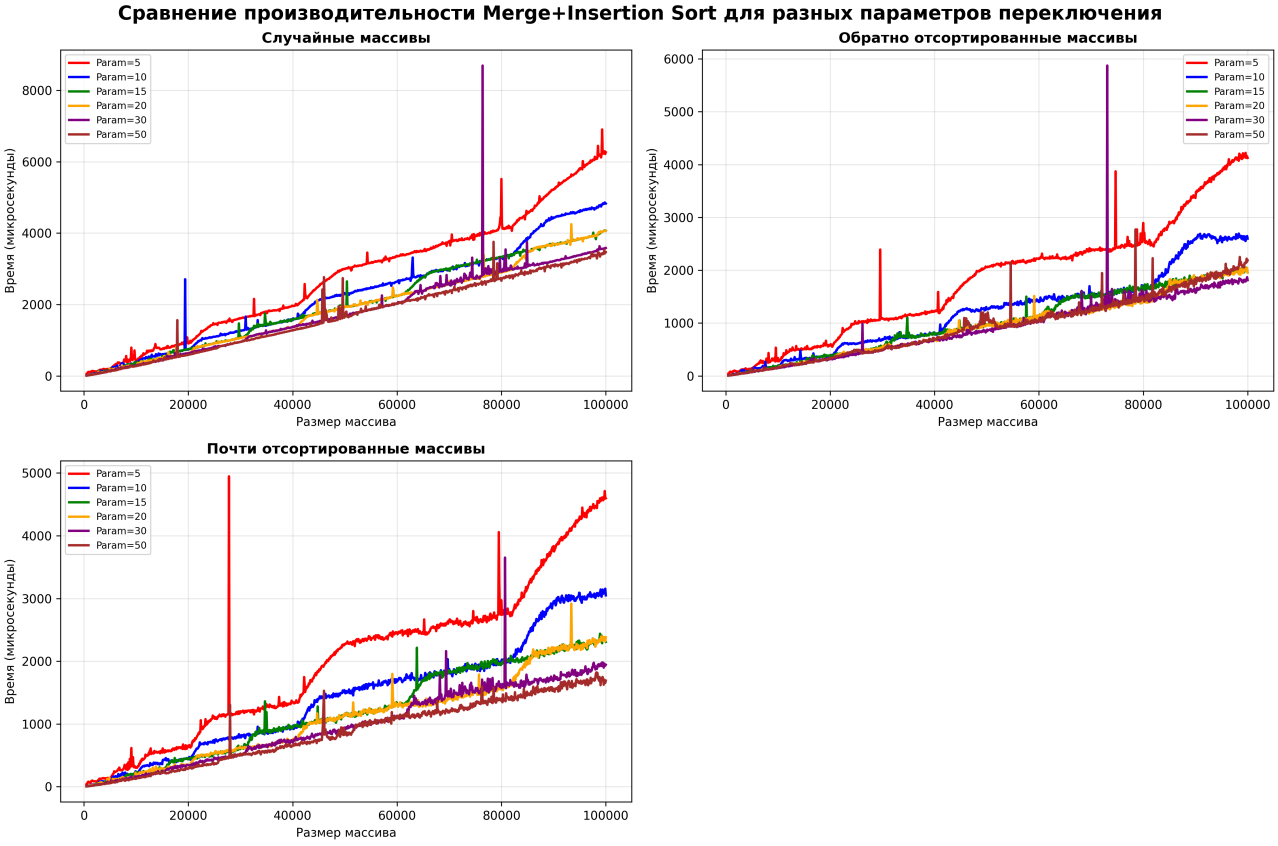


Общая тенденция для параметра 30 такова: гибридный алгоритм Merge+Insertion Sort демонстрирует преимущество над стандартным Merge Sort, особенно это заметно с ростом N. Однако, на графики случайного массива видно, что при N приблизительно 70000 гибридный алгоритм демонстрирует худшую производительность, затраты по времени хуже стандартной реализации Merge Sort, но позже общая тенденция сохраняется. Для обратно отсортированных массивов видно, что приблизительно при N = 70000 производительность алгоритма

Merge+Insertion Sort резко возрастает, но затраты по времени не превосходят Merge Sort. Аналогично при N приблизительно 80000 для массива почти отсортированных массивов.



При использовании параметра 50 Merge+Insertion Sort демонстрирует преимущество над стандартным Merge Sort для любых массивов, с ростом N преимущество становиться выраженным.



Анализ графиков позволяет выявить оптимальные параметры переключения для различных типов массивов. Для случайных массивов наилучшая производительность достигается при параметре 50, который также демонстрирует высокую стабильность с минимальными колебаниями временных показателей. Наихудшие результаты наблюдаются при параметре 30 для N приблизительно 80000. В случае обратно отсортированных массивов максимальная эффективность отмечается при параметре 30, особенно выраженная на больших объемах данных. Однако при N = 700000 фиксируется резкий скачок производительности, превосходящий все остальные параметры. Наиболее стабильное поведение характерно для значений 15, 20 и 50. Для почти отсортированных массивов

наблюдается схожая с случайными массивами тенденция: параметр 50 обеспечивает оптимальную производительность, а значение 5 демонстрирует наименее эффективную работу алгоритма.

Общий вывод

В ходе эксперимента гибридный алгоритм Merge+Insertion Sort показал себя как более эффективное решение по сравнению с базовой версией Merge Sort. Наибольшая эффективность гибридного алгоритма достигается при параметре 50, которые обеспечивает стабильное превосходство с выраженной положительной динамикой при увеличении объема данных.