

# Отчет по задаче А2

Гуршумов Даниил Бахтиянович  
БПИ 239

## Аннотация

брбрбр 3 бзбзбз 2

В данном отчете представлено сравнение стандартного алгоритма сортировки слиянием (MERGE SORT) и гибридного алгоритма MERGE+INSERTION SORT. Выполнены эмпирические замеры временных затрат для различных типов массивов и проведен сравнительный анализ эффективности алгоритмов.

## 1 Задача

Цель работы: провести эмпирическое сравнение двух реализаций алгоритмов сортировки на основе сортировки слиянием: стандартного MERGE SORT и гибридного MERGE+INSERTION SORT.

Необходимо:

- Реализовать генератор тестовых данных для экспериментов.
- Замерить время работы стандартного алгоритма MERGE SORT на массивах разных типов.
- Замерить время работы гибридного алгоритма MERGE+INSERTION SORT на тех же массивах.
- Выполнить сравнительный анализ результатов.

## 2 Решение

### 2.1 Реализация алгоритмов

В данном разделе представлен код, используемый для генерации тестовых данных, реализации алгоритмов и замера времени.

```
1 #include <iostream>
2 #include <cmath>
3 #include <random>
4 #include <vector>
5 #include <fstream>
6 #include <iomanip>
7
8 //
9 class ArrayGenerator {
10 public:
11     void generate_array() {
12         std::random_device rd;
13         std::mt19937 generator(rd());
14         std::uniform_int_distribution<> dist(_min_value, _max_value);
15         arr.resize(_size);
16         for (int i = 0; i < _size; ++i) {
17             arr[i] = dist(generator);
18         }
19     }
20     std::vector<int> gen_1(int n) {
21         return {arr.begin(), arr.begin() + n};
22     }
23 }
```

```

23     std::vector<int> gen_2(int n) {
24         std::vector<int> temp = {arr.begin(), arr.begin() + n};
25         std::sort(temp.rbegin(), temp.rend());
26         return temp;
27     }
28     std::vector<int> gen_3(int n) {
29         std::vector<int> temp = {arr.begin(), arr.begin() + n};
30         std::sort(temp.begin(), temp.end());
31         std::random_device rd;
32         std::mt19937 generator(rd());
33         std::uniform_int_distribution<> dist(0, n);
34         for (int i = 0; i < 10; ++i) {
35             std::swap(temp[dist(generator)], temp[dist(generator)]);
36         }
37         return temp;
38     }
39     ArrayGenerator() {
40         generate_array();
41     }
42
43 private:
44     int _max_value = 6000;
45     int _min_value = 0;
46     int _size = 10000;
47     std::vector<int> arr;
48 };
49
50 //                               Insertion Sort
51 void insertion_sort(std::vector<int>& a, int s, int e) {
52     for (int i = s + 1; i <= e; ++i) {
53         int key = a[i];
54         int j = i - 1;
55
56         while (j >= s && a[j] > key) {
57             a[j + 1] = a[j];
58             j--;
59         }
60
61         a[j + 1] = key;
62     }
63 }
64
65 //                               Merge Sort
66 void merge_sort(std::vector<int>& a, int s, int m, int e) {
67     int n1 = m - s + 1;
68     int n2 = e - m;
69     std::vector<int> left(n1), right(n2);
70
71     for (int i = 0; i < n1; i++) {
72         left[i] = a[s + i];
73     }
74     for (int i = 0; i < n2; i++) {
75         right[i] = a[m + 1 + i];
76     }
77
78     int l = 0, r = 0, k = s;
79     while (l < n1 && r < n2) {
80         if (left[l] <= right[r]) {
81             a[k] = left[l];
82             l++;
83         } else {
84             a[k] = right[r];
85             r++;

```

```

86         }
87         k++;
88     }
89
90     while (l < n1) {
91         a[k] = left[l];
92         l++;
93         k++;
94     }
95
96     while (r < n2) {
97         a[k] = right[r];
98         r++;
99         k++;
100     }
101 }
102
103 //                                     MERGE+INSERTION SORT
104 void merge(std::vector<int>& a, int s, int e) {
105     if (e - s <= 15) {
106         insertion_sort(a, s, e);
107         return;
108     }
109
110     int mid = s + (e - s) / 2;
111     merge(a, s, mid);
112     merge(a, mid + 1, e);
113     merge_sort(a, s, mid, e);
114 }
115
116 //                                     MERGE SORT
117 void standart_merge(std::vector<int>& a, int l, int r) {
118     if (l >= r) {
119         return;
120     }
121     int mid = l + (r - l) / 2;
122     standart_merge(a, l, mid);
123     standart_merge(a, mid + 1, r);
124     merge_sort(a, l, mid, r);
125 }
126
127 //
128
129 class Test {
130 public:
131     long long measureMergeSortTime(std::vector<int> array, int repetitions =
132     10) {
133         long long totalTime = 0;
134         for (int i = 0; i < repetitions; ++i) {
135             auto start = std::chrono::high_resolution_clock::now();
136             standart_merge(array, 0, array.size() - 1);
137             auto elapsed = std::chrono::high_resolution_clock::now() - start;
138             totalTime += std::chrono::duration_cast<std::chrono::microseconds>(
139                 elapsed).count();
140         }
141         return totalTime / repetitions;
142     }
143
144     long long measureHybridSortTime(std::vector<int> array, int repetitions =
145     10) {
146         long long totalTime = 0;
147         for (int i = 0; i < repetitions; ++i) {
148             auto start = std::chrono::high_resolution_clock::now();

```

```

145         merge(array, 0, array.size() - 1);
146         auto elapsed = std::chrono::high_resolution_clock::now() - start;
147         totalTime += std::chrono::duration_cast<std::chrono::microseconds>(
            elapsed).count();
148     }
149     return totalTime / repetitions;
150 }
151 };
152
153 int main() {
154     ArrayGenerator generator;
155     Test tester;
156     std::vector<int> sizes;
157     for (int size = 500; size <= 10000; size += 100) {
158         sizes.push_back(size);
159     }
160
161     std::ofstream csvFileMerge("merge_results.csv");
162     std::ofstream csvFileHybrid("hybrid_results.csv");
163
164     csvFileMerge << "Size,Random,gen_2,gen_3\n";
165     csvFileHybrid << "Size,Random,gen_2,gen_3\n";
166     for (int size : sizes) {
167         std::vector<int> randomArray = generator.gen_1(size);
168         std::vector<int> gen_2Array = generator.gen_2(size);
169         std::vector<int> gen_3Array = generator.gen_3(size);
170
171         long long randomTimeMerge = tester.measureMergeSortTime(randomArray);
172         long long gen_2TimeMerge = tester.measureMergeSortTime(gen_2Array);
173         long long gen_3TimeMerge = tester.measureMergeSortTime(gen_3Array);
174
175         long long randomTimeHybrid = tester.measureHybridSortTime(randomArray);
176         long long gen_2TimeHybrid = tester.measureHybridSortTime(gen_2Array);
177         long long gen_3TimeHybrid = tester.measureHybridSortTime(gen_3Array);
178
179         csvFileMerge << size << "," << randomTimeMerge << "," << gen_2TimeMerge
            << "," << gen_3TimeMerge << "\n";
180         csvFileHybrid << size << "," << randomTimeHybrid << "," <<
            gen_2TimeHybrid << "," << gen_3TimeHybrid << "\n";
181
182         std::cout << "Size: " << size
183             << " Random Merge: " << randomTimeMerge << " microsec"
184             << " gen_2 Merge: " << gen_2TimeMerge << " microsec"
185             << " gen_3 Merge: " << gen_3TimeMerge << " microsec"
186             << "\n";
187     }
188     csvFileMerge.close();
189     csvFileHybrid.close();
190 }

```

## 2.2 Результаты экспериментов

На основании проведенных экспериментов были построены следующие графики, отображающие зависимость времени выполнения алгоритмов от размера массива  $N$ .

## 3 Анализ и выводы

На основании графиков можно сделать следующие выводы:

1. **Эффективность гибридного алгоритма на случайных данных:** 1. Случайные массивы График (для случайного массива): Гибридный алгоритм Merge+Insertion Sort показывает

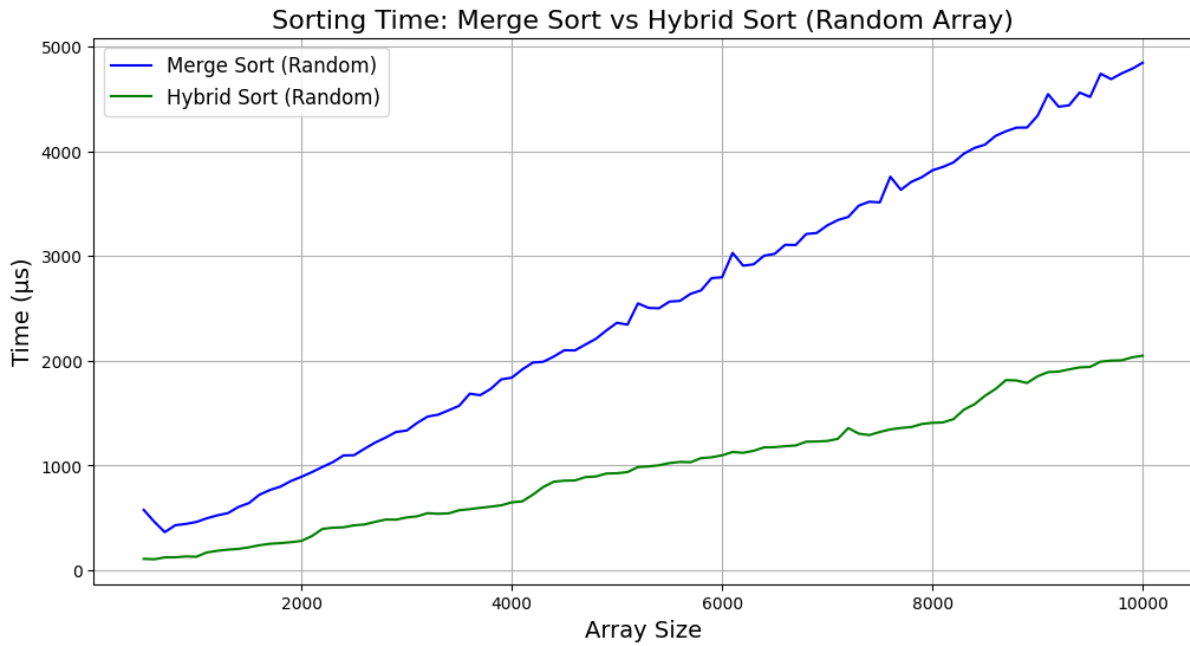


Рис. 1: Сравнение MERGE SORT и MERGE+INSERTION SORT на случайном массиве.

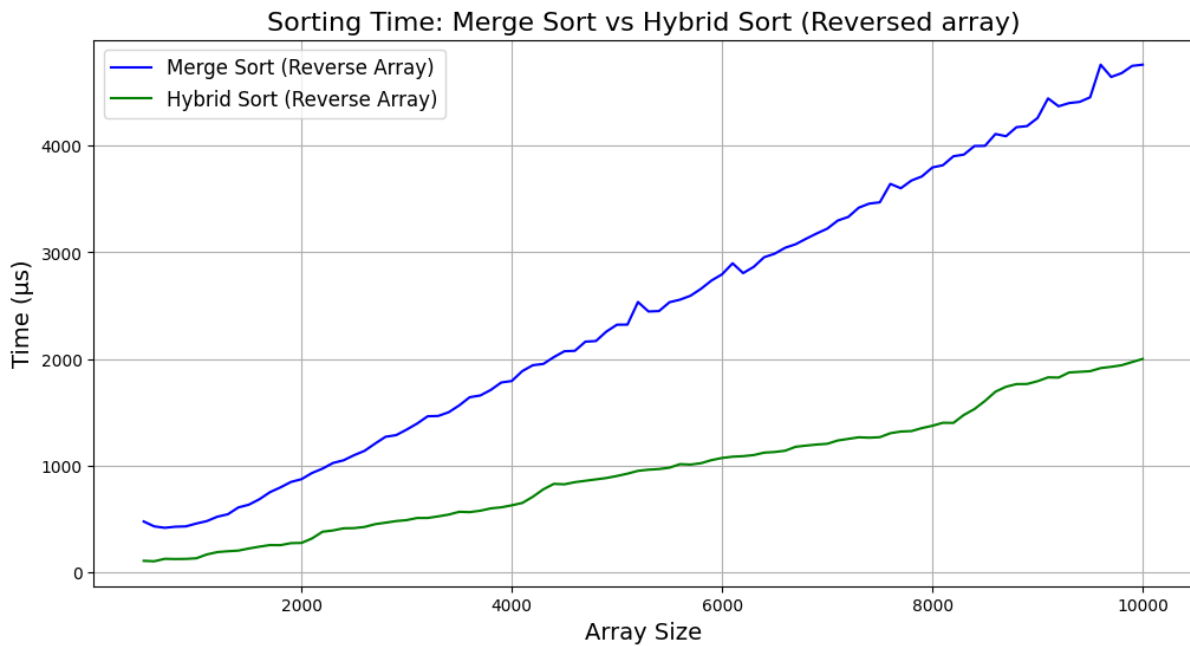


Рис. 2: Сравнение MERGE SORT и MERGE+INSERTION SORT на массиве с обратным порядком элементов.

преимущество над стандартным Merge Sort. Разрыв в производительности не так сильно выражен, как в случае почти отсортированных массивов, но с увеличением размера массива гибридный подход стабильно опережает.

2. **Эффективность на массиве с обратным порядком:** Для обратных массивов разница между Merge Sort и гибридным алгоритмом также очевидна. Гибридный алгоритм быстрее во всех тестах, с преимуществом, которое растёт с увеличением размера массива. Insertion Sort в гибридной версии обрабатывает небольшие подмассивы, эффективно исправляя порядок значений.
3. **Эффективность на почти отсортированных данных:** Гибридный Merge+Insertion Sort

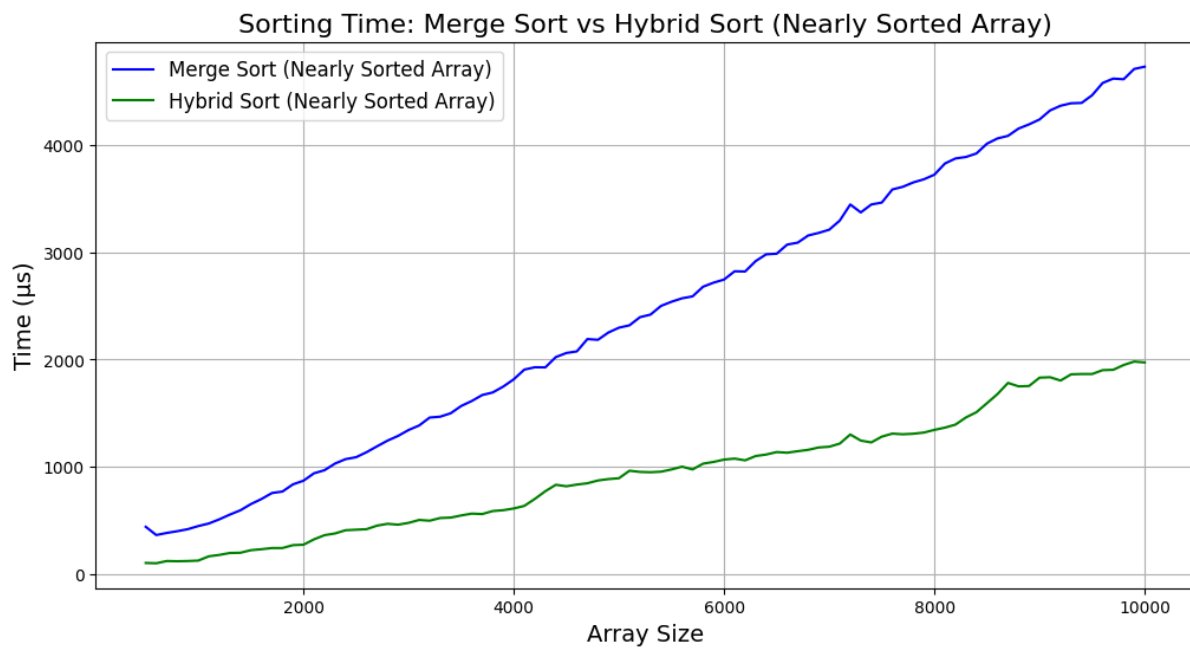


Рис. 3: Сравнение MERGE SORT и MERGE+INSERTION SORT на почти отсортированном массиве.

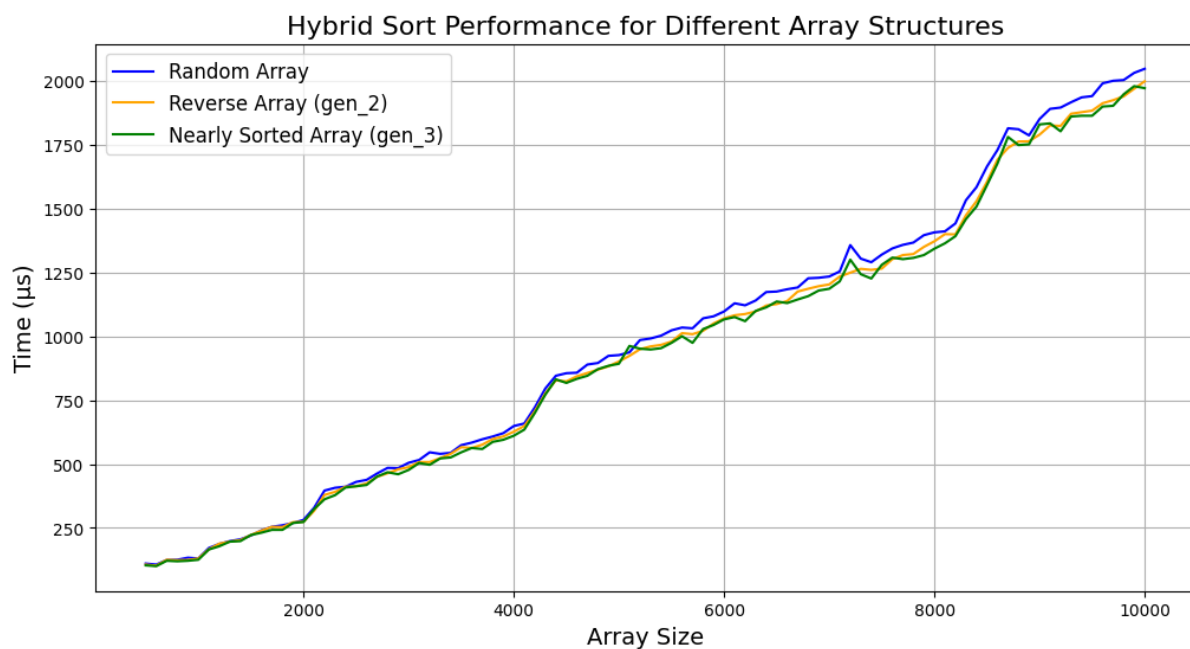


Рис. 4: Сравнение Hybrid sort для различных векторов.

выигрывает особенно заметно на почти отсортированных массивах, где наблюдается наибольший прирост скорости. В случае почти отсортированных массивов Insertion Sort работает практически за линейное время на малых сегментах, что делает гибридный подход оптимальным.

#### 4. Рекомендации по выбору алгоритма:

- Гибридный алгоритм всегда быстрее стандартного Merge Sort. Его преимущество наибольшее в случаях с упорядоченными или частично упорядоченными данными (почти отсортированные массивы). На случайных массивах прирост производительности также заметен, но чуть меньше.

- Оптимизация за счёт переключения: Использование Insertion Sort на небольших подмассивах снижает накладные расходы, связанные с рекурсией Merge Sort. Подбор порога переключения (например, 20–50 элементов) позволяет достичь максимальной эффективности.
- Практическое применение: Гибридный подход предпочтителен для массивов любой структуры, особенно если данные часто бывают частично отсортированными или имеют небольшую длину.