# Отчет по задаче А3

Гуршумов Даниил Бахтиятович
БПИ 239

**Аннотация**

брбрбр 3 бзбзбз 3

## 1 Задача

Цель работы: провести эмпирическое сравнение двух реализаций алгоритмов сортировки на основе сортировки слиянием: стандартного Quick Sort и Intro Sort

Необходимо:

- Реализовать генератор тестовых данных для экспериментов.
- Замерить время работы стандартного алгоритма Quick Sort на массивах разных типов.
- Замерить время работы гибридного алгоритма Intro 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.

```cpp
#include <iostream>
#include <cmath>
#include <random>
#include <vector>
#include <fstream>
#include <iomanip>


class ArrayGenerator {
public:
    void generate_array() {
        std::random_device rd;
        std::mt19937 generator(rd());
        std::uniform_int_distribution<> dist(_min_value, _max_value);
        arr.resize(_size);
        for (int i = 0; i < _size; ++i) {
            arr[i] = dist(generator);
        }
    }
    std::vector<int> gen_1(int n) {
        return {arr.begin(), arr.begin() + n};
    }
    std::vector<int> gen_2(int n) {
        std::vector<int> temp = {arr.begin(), arr.begin() + n};
        std::sort(temp.rbegin(), temp.rend());
        return temp;
    }
```

```
28
29      std::vector<int> gen_3(int n) {
30          std::vector<int> temp = {arr.begin(), arr.begin() + n};
31          std::sort(temp.begin(), temp.end());
32          std::random_device rd;
33          std::mt19937 generator(rd());
34          std::uniform_int_distribution<> dist(0, n);
35          for (int i = 0; i < 10; ++i) {
36              std::swap(temp[dist(generator)], temp[dist(generator)]);
37          }
38          return temp;
39      }
40      ArrayGenerator() {
41          generate_array();
42      }
43
44  private:
45      int _max_value = 6000;
46      int _min_value = 0;
47      int _size = 10000;
48      std::vector<int> arr;
49  };
50
51  #include <iostream>
52  #include <vector>
53  #include <cmath>
54  #include <random>
55
56  void insertion_sort(std::vector<int>& a, int s, int e) {
57      for (int i = s + 1; i <= e; ++i) {
58          int key = a[i];
59          int j = i - 1;
60
61          while (j >= s && a[j] > key) {
62              a[j + 1] = a[j];
63              j--;
64          }
65
66          a[j + 1] = key;
67      }
68  }
69
70  void heapify(std::vector<int>& A, int n, int i) {
71      int largest = i;
72      int l = 2 * i + 1;
73      int r = 2 * i + 2;
74
75      if (l < n && A[l] > A[largest]) {
76          largest = l;
77      }
78      if (r < n && A[r] > A[largest]) {
79          largest = r;
80      }
81      if (largest != i) {
82          std::swap(A[i], A[largest]);
83          heapify(A, n, largest);
84      }
85  }
86
87  void heapSort(std::vector<int>& A) {
88      int n = A.size();
89
90      for (int i = n / 2 - 1; i >= 0; --i) {
```

```
91          heapify(A, n, i);
92      }
93
94      for (int i = n - 1; i > 0; --i) {
95          std::swap(A[0], A[i]);
96          heapify(A, i, 0);
97      }
98  }
99
100 int particion(std::vector<int>& arr, int left, int right) {
101     std::random_device rd;
102     std::mt19937 g(rd());
103     std::swap(arr[right], arr[left + g() % (right - left)]);
104
105     int pivot = arr[right];
106
107     int low = left - 1;
108     for (int j = left; j <= right - 1; ++j) {
109         if (arr[j] < pivot) {
110             low++;
111             std::swap(arr[low], arr[j]);
112         }
113     }
114     std::swap(arr[low + 1], arr[right]);
115     return low + 1;
116 }
117
118 void quick_sort(std::vector<int>& arr, int left, int right, int& counter) {
119     counter += 1;
120
121     if (counter >= 2 * std::log2(arr.size())) {
122         heapSort(arr);
123         return;
124     }
125
126     if (right - left < 16) {
127         insertion_sort(arr, left, right);
128         return;
129     }
130
131     if (left < right) {
132         int pi = particion(arr, left, right);
133         quick_sort(arr, left, pi - 1, counter);
134         quick_sort(arr, pi + 1, right, counter);
135     }
136 }
137
138
139
140 void standart_quick_sort(std::vector<int>& arr, int l, int r) {
141     if (l < r) {
142         int pi = particion(arr, l, r);
143         standart_quick_sort(arr, l, pi - 1);
144         standart_quick_sort(arr, pi + 1, r);
145     }
146 }
147
148
149 class SortTester{
150 public:
151     long long measureQuickSortTime(std::vector<int> array, int repetitions =
        10) {
152         long long totalTime = 0;
```

```
153         for (int i = 0; i < repetitions; ++i) {
154             auto start = std::chrono::high_resolution_clock::now();
155             standart_quick_sort(array, 0, array.size() - 1);
156             auto elapsed = std::chrono::high_resolution_clock::now() - start;
157             totalTime += std::chrono::duration_cast<std::chrono::microseconds>(
                    elapsed).count();
158         }
159         return totalTime / repetitions;
160     }
161 
162     long long measureHybridSortTime(std::vector<int> array, int repetitions =
            10) {
163         long long totalTime = 0;
164         for (int i = 0; i < repetitions; ++i) {
165             int counter = 0;
166             auto start = std::chrono::high_resolution_clock::now();
167             quick_sort(array, 0, array.size() - 1, counter);
168             auto elapsed = std::chrono::high_resolution_clock::now() - start;
169             totalTime += std::chrono::duration_cast<std::chrono::microseconds>(
                    elapsed).count();
170         }
171         return totalTime / repetitions;
172     }
173 };
174 
175 int main() {
176     ArrayGenerator generator;
177     SortTester tester;
178     std::vector<int> sizes;
179     for (int size = 500; size <= 10000; size += 100) {
180         sizes.push_back(size);
181     }
182 
183     std::ofstream csvFileStandartQuick("standart_quick_results.csv");
184     std::ofstream csvFileIntroSort("introsort_results.csv");
185 
186     csvFileStandartQuick << "Size,Random,gen_2,gen_3\n";
187     csvFileIntroSort << "Size,Random,gen_2,gen_3\n";
188     for (int size : sizes) {
189         std::vector<int> randomArray = generator.gen_1(size);
190         std::vector<int> gen_2Array = generator.gen_2(size);
191         std::vector<int> gen_3Array = generator.gen_3(size);
192 
193         long long randomTimeMerge = tester.measureQuickSortTime(randomArray);
194         long long gen_2TimeMerge = tester.measureQuickSortTime(gen_2Array);
195         long long gen_3TimeMerge = tester.measureQuickSortTime(gen_3Array);
196 
197         long long randomTimeHybrid = tester.measureHybridSortTime(randomArray);
198         long long gen_2TimeHybrid = tester.measureHybridSortTime(gen_2Array);
199         long long gen_3TimeHybrid = tester.measureHybridSortTime(gen_3Array);
200 
201         csvFileStandartQuick << size << "," << randomTimeMerge << "," <<
                gen_2TimeMerge << "," << gen_3TimeMerge << "\n";
202         csvFileIntroSort << size << "," << randomTimeHybrid << "," <<
                gen_2TimeHybrid << "," << gen_3TimeHybrid << "\n";
203 
204         std::cout << "Size: " << size
205                   << " Random Stand_Quick: " << randomTimeMerge << " microsec"
206                   << " gen_2 Stand_Quick: " << gen_2TimeMerge << " microsec"
207                   << " gen_3 Stand_Quick: " << gen_3TimeMerge << " microsec\n";
208 
209         std::cout << "Size: " << size
210                   << " Random IntroSort: " << randomTimeHybrid << " microsec"
```

```
                    << " gen_2 IntroSort: " << gen_2TimeHybrid << " microsec"
                    << " gen_3 IntroSort: " << gen_3TimeHybrid << " microsec\n";
    }

    csvFileStandartQuick.close();
    csvFileIntroSort.close();

    return 0;
}
```

## 2.2 Результаты экспериментов

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

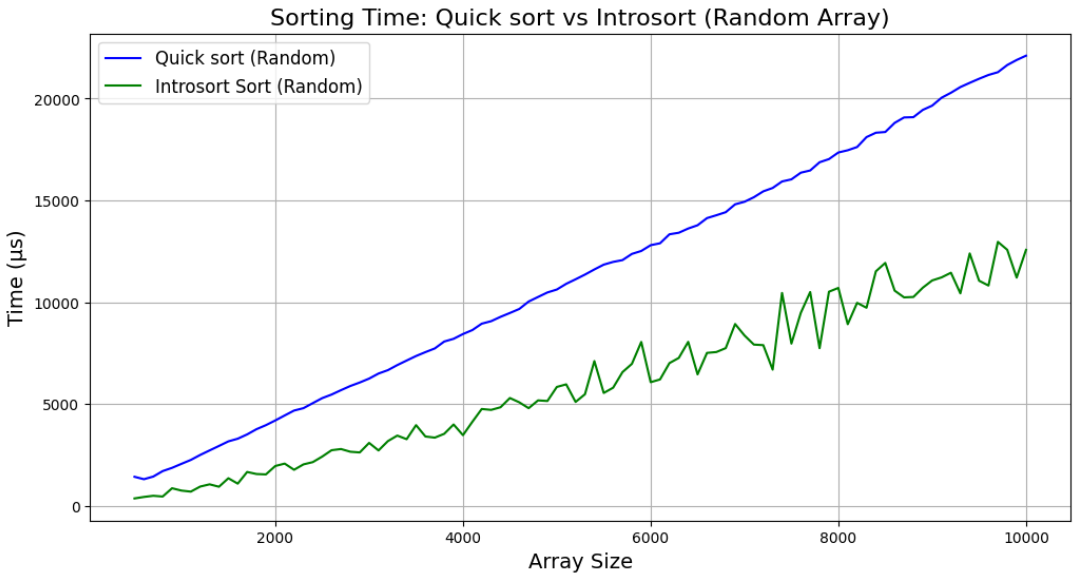


Рис. 1: Сравнение Quick sort и Intro SORT на случайном массиве.

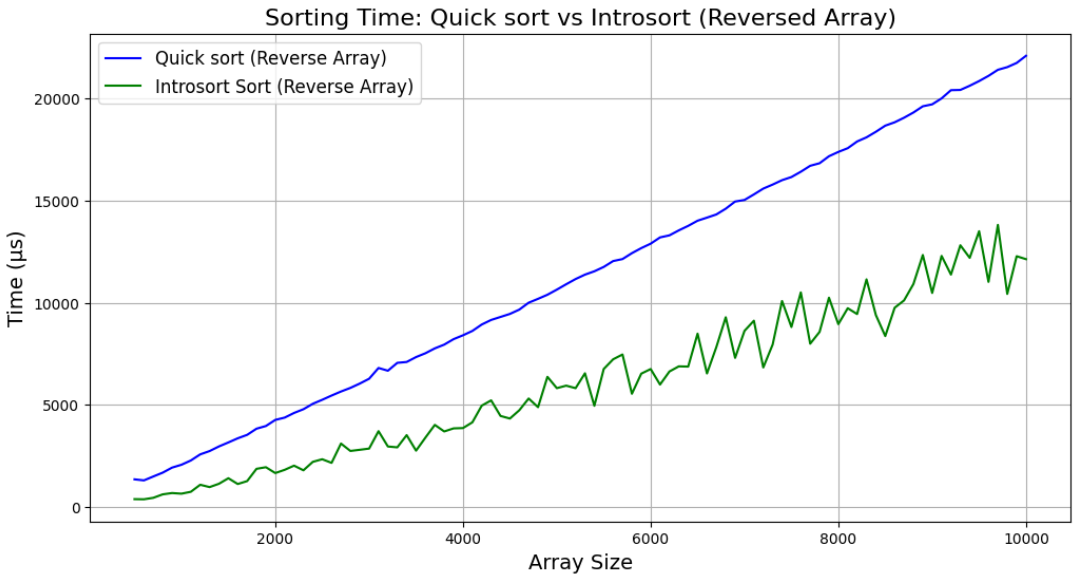


Рис. 2: Сравнение Quick sort и Intro SORT на массиве с обратным порядком элементов.

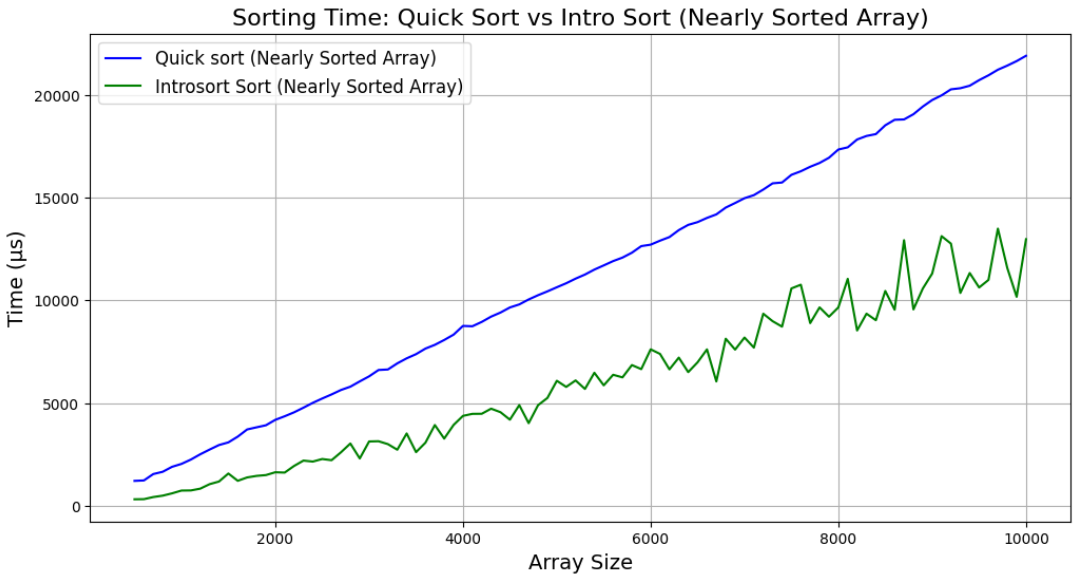


Рис. 3: Сравнение Quick sort и Intro SORT на почти отсортированном массиве.

# 3 Анализ и выводы

## 3.1 Random array

По графику становится очевидна разница между предложенными алгоритмами, однако стоит обратить внимание, что если у стандартного Quick Sort прослеживается относительная линейная зависимость, то у Intro Sort сильно бросается в глаза не равномерность данных. Конечно здорово, что скорость в некоторых промежутках меньше даже более чем в 2 раза, но стоит задуматься и нестабильности такого алгоритма.

## 3.2 Reversed array

В целом мы видим повторении ситуации как и на графике Random array. Intro Sort работает быстрее, однако опять видим разброс значений и непредсказуемость по мере увелечения количества элементов.

## 3.3 Nearly Sorted array

Единственным визуальным отличием является то, что график Intro sort до $n \approx 2900$ время относительно растёт линейно, однако опять виден разброс.

# 4 Общий вывод

Мы видим, что Intro Sort превосходит Quick Sort на трёх видах массивов, однако смежным ощущением является неопределённость поведения алгоритма на графике.