

Задача А3б

Зарина Рамазанова

SET 3, ноябрь 2025

Этап 1

Реализация класса ArrayGenerator осталась прежней, подробнее в репозитории(ссылка в конце)

Этап 2

При замере была использована следующая реализация Quick Sort:

```
1 int partition(std::vector<int>& arr, int left, int right) {
2     std::random_device rd;
3     std::mt19937 gen(rd());
4     std::uniform_int_distribution<int> dis(left, right);
5     int pivotIndex = dis(gen);
6     std::swap(arr[pivotIndex], arr[right]);
7
8     int pivot = arr[right];
9     int i = left - 1;
10
11    for (int j = left; j < right; j++) {
12        if (arr[j] <= pivot) {
13            i++;
14            std::swap(arr[i], arr[j]);
15        }
16    }
17    std::swap(arr[i + 1], arr[right]);
18    return i + 1;
19}
20
21 void standardQuickSort(std::vector<int>& arr, int left, int right) {
22     if (left < right) {
23         int pi = partition(arr, left, right);
24         standardQuickSort(arr, left, pi - 1);
25         standardQuickSort(arr, pi + 1, right);
26     }
27 }
28
29 void quickSort(std::vector<int>& arr) {
30     if (arr.empty()) return;
31     standardQuickSort(arr, 0, arr.size() - 1);
32 }
```

Для реализации представления графиков был реализован main() с записью результатов эксперимента с каждым видом массива в csv-файлы:

```
1 int main() {
2     ArrayGenerator generator;
3
4     const size_t MAX_SIZE = 100000;
5     const size_t MIN_SIZE = 500;
6     const size_t STEP = 1000;
7
8     auto randomBase = generator.generateRandomArray(MAX_SIZE);
9     auto reverseBase = generator.generateReverseSortedArray(MAX_SIZE);
10    auto almostBase = generator.generateAlmostSortedArray(MAX_SIZE, 50);
```

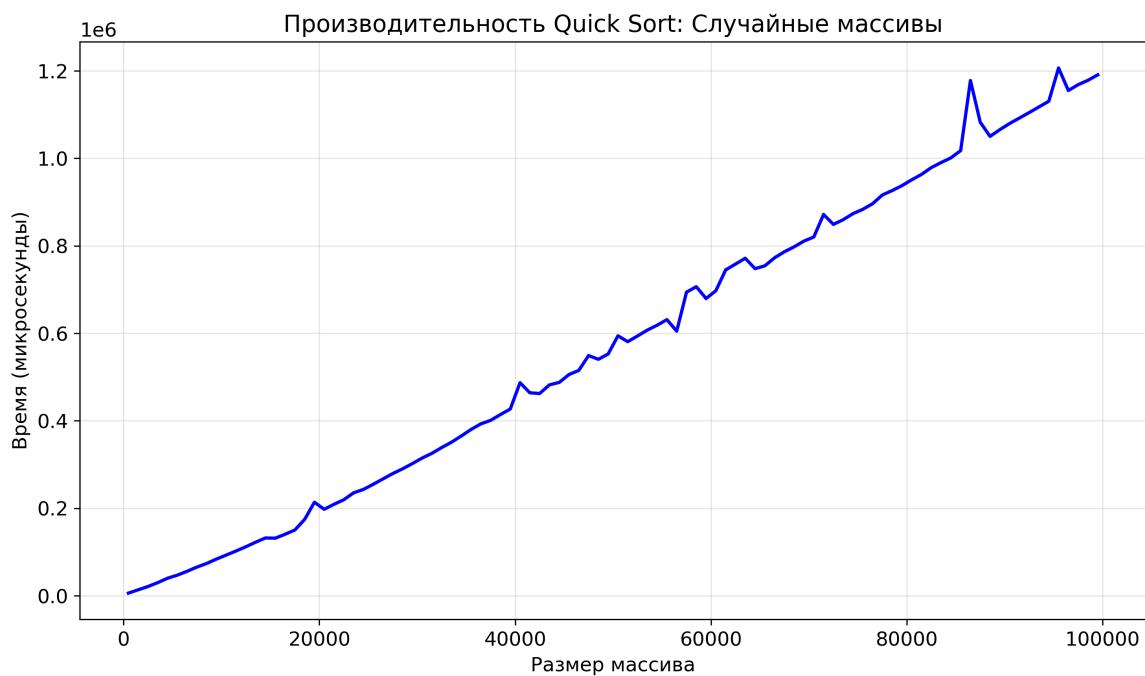
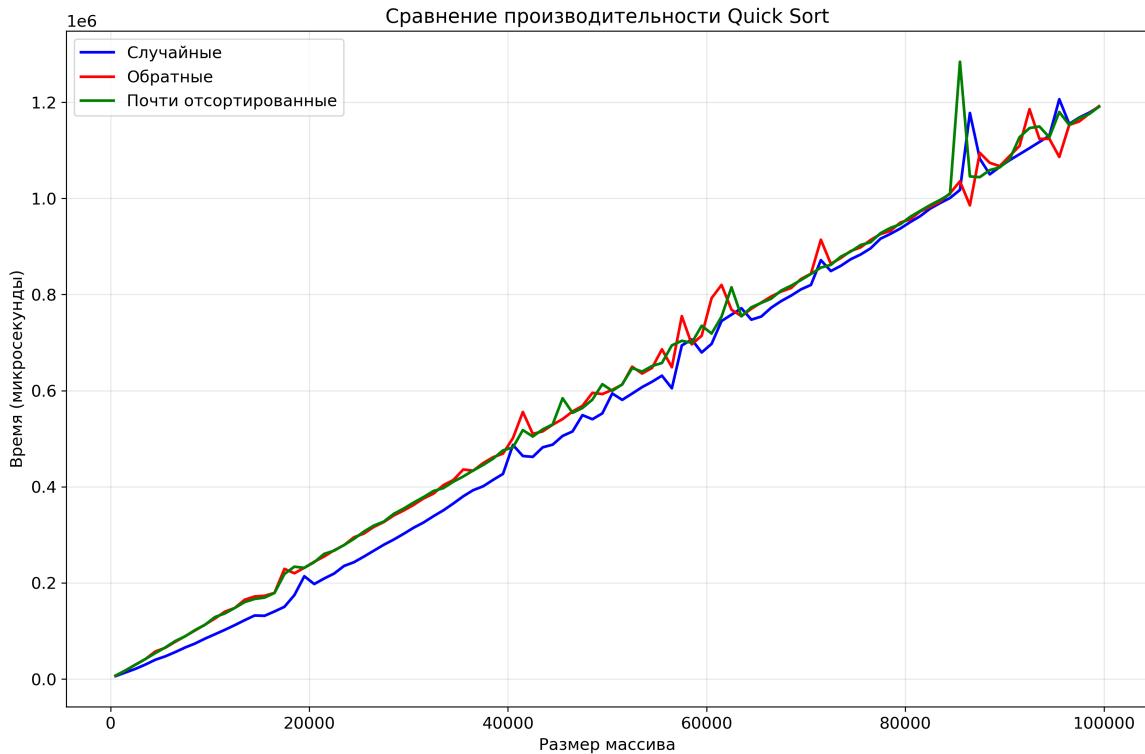
```

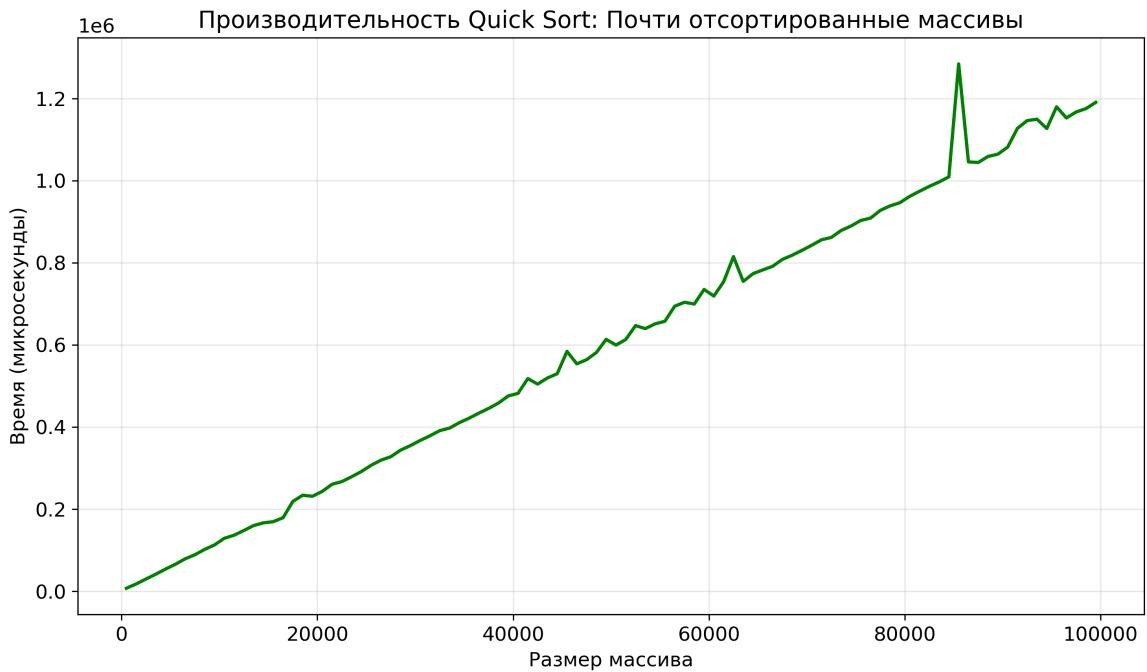
11
12     std::ofstream randomFile("quick_sort_random.csv");
13     std::ofstream reverseFile("quick_sort_reverse.csv");
14     std::ofstream almostFile("quick_sort_almost.csv");
15
16     randomFile << "size,time(ms)\n";
17     reverseFile << "size,time(ms)\n";
18     almostFile << "size,time(ms)\n";
19
20     for (size_t size = MIN_SIZE; size <= MAX_SIZE; size += STEP) {
21         auto randomArray = generator.getSubarray(randomBase, size);
22         auto start = std::chrono::high_resolution_clock::now();
23         quickSort(randomArray);
24         auto elapsed = std::chrono::high_resolution_clock::now() - start;
25         long long randomTime = std::chrono::duration_cast<std::chrono::microseconds>(elapsed).count();
26         randomFile << size << "," << randomTime << "\n";
27
28         auto reverseArray = generator.getSubarray(reverseBase, size);
29         start = std::chrono::high_resolution_clock::now();
30         quickSort(reverseArray);
31         elapsed = std::chrono::high_resolution_clock::now() - start;
32         long long reverseTime = std::chrono::duration_cast<std::chrono::microseconds>(elapsed).count()
33         ;
34         reverseFile << size << "," << reverseTime << "\n";
35
36         auto almostArray = generator.getSubarray(almostBase, size);
37         start = std::chrono::high_resolution_clock::now();
38         quickSort(almostArray);
39         elapsed = std::chrono::high_resolution_clock::now() - start;
40         long long almostTime = std::chrono::duration_cast<std::chrono::microseconds>(elapsed).count();
41         almostFile << size << "," << almostTime << "\n";
42     }
43
44     randomFile.close();
45     reverseFile.close();
46     almostFile.close();
47
48     return 0;
}

```

Графики

Были получены следующие графики:





Этап 3

Реализация класса SortTester:

```

1 class SortTester {
2     private:
3         ArrayGenerator generator;
4
5     // Insertion Sort для небольших массивов
6     void insertionSort(std::vector<int>& arr, int left, int right) {
7         for (int i = left + 1; i <= right; i++) {
8             int key = arr[i];
9             int j = i - 1;
10            while (j >= left && arr[j] > key) {
11                arr[j + 1] = arr[j];
12                j--;
13            }
14            arr[j + 1] = key;
15        }
16    }
17
18    void quickSort(std::vector<int>& arr, int left, int right) {
19        if (left < right) {
20            int pivotIndex = partition(arr, left, right);
21            quickSort(arr, left, pivotIndex - 1);
22            quickSort(arr, pivotIndex + 1, right);
23        }
24    }
25
26    int partition(std::vector<int>& arr, int left, int right) {
27        int pivot = arr[right];
28        int i = left - 1;
29
30        for (int j = left; j < right; j++) {
31            if (arr[j] < pivot) {
32                i++;
33                std::swap(arr[i], arr[j]);
34            }
35        }
36        std::swap(arr[i + 1], arr[right]);
37        return i + 1;
38    }
39
40    void checkSort() {
41        std::vector<int> arr = generator.generate(100000);
42        quickSort(arr, 0, arr.size() - 1);
43        for (int i = 0; i < arr.size() - 1; i++) {
44            if (arr[i] > arr[i + 1]) {
45                std::cout << "Error: array is not sorted" << std::endl;
46                return;
47            }
48        }
49        std::cout << "Array is sorted" << std::endl;
50    }
51
52    void testInsertionSort() {
53        std::vector<int> arr = generator.generate(10000);
54        insertionSort(arr, 0, arr.size() - 1);
55        for (int i = 0; i < arr.size() - 1; i++) {
56            if (arr[i] > arr[i + 1]) {
57                std::cout << "Error: insertion sort failed" << std::endl;
58                return;
59            }
60        }
61        std::cout << "Insertion sort passed" << std::endl;
62    }
63
64    void testQuickSort() {
65        std::vector<int> arr = generator.generate(100000);
66        quickSort(arr, 0, arr.size() - 1);
67        for (int i = 0; i < arr.size() - 1; i++) {
68            if (arr[i] > arr[i + 1]) {
69                std::cout << "Error: quick sort failed" << std::endl;
70                return;
71            }
72        }
73        std::cout << "Quick sort passed" << std::endl;
74    }
75
76    void runTests() {
77        checkSort();
78        testInsertionSort();
79        testQuickSort();
80    }
81
82    void generateAndSort() {
83        std::vector<int> arr = generator.generate(100000);
84        quickSort(arr, 0, arr.size() - 1);
85        std::cout << "Sorted array: ";
86        for (int i = 0; i < arr.size(); i++) {
87            std::cout << arr[i] << " ";
88        }
89        std::cout << std::endl;
90    }
91
92    void generateAndPrint() {
93        std::vector<int> arr = generator.generate(100000);
94        std::cout << "Generated array: ";
95        for (int i = 0; i < arr.size(); i++) {
96            std::cout << arr[i] << " ";
97        }
98        std::cout << std::endl;
99    }
100}
```

```

13         }
14         arr[j + 1] = key;
15     }
16 }
17
18 void heapify(std::vector<int>& arr, int heap_size, int i, int offset) {
19     int largest = i;
20     int left = 2 * (i - offset) + 1 + offset;
21     int right = 2 * (i - offset) + 2 + offset;
22
23     if (left < heap_size && arr[left] > arr[largest])
24         largest = left;
25
26     if (right < heap_size && arr[right] > arr[largest])
27         largest = right;
28
29     if (largest != i) {
30         std::swap(arr[i], arr[largest]);
31         heapify(arr, heap_size, largest, offset);
32     }
33 }
34
35 // Heap Sort
36 void heapSort(std::vector<int>& arr, int left, int right) {
37     int n = right - left + 1;
38
39     // Строим кучу (max-heap)
40     for (int i = left + n/2 - 1; i >= left; i--) {
41         heapify(arr, right + 1, i, left);
42     }
43
44     // Извлекаем элементы из кучи
45     for (int i = right; i > left; i--) {
46         std::swap(arr[left], arr[i]);
47         heapify(arr, i, left, left);
48     }
49 }
50
51 // Quick Sort со случайным опорным элементом
52 int partition(std::vector<int>& arr, int left, int right) {
53     // Случайный выбор опорного элемента
54     std::random_device rd;
55     std::mt19937 gen(rd());
56     std::uniform_int_distribution<int> dis(left, right);
57     int pivotIndex = dis(gen);
58     std::swap(arr[pivotIndex], arr[right]);
59
60     int pivot = arr[right];
61     int i = left - 1;
62
63     for (int j = left; j < right; j++) {
64         if (arr[j] <= pivot) {
65             i++;
66             std::swap(arr[i], arr[j]);
67         }
68     }
69     std::swap(arr[i + 1], arr[right]);
70     return i + 1;
71 }
72
73 // Introsort
74 void introSort(std::vector<int>& arr, int left, int right, int depthLimit) {
75     // Если элементов мало, используем Insertion Sort
76     if (right - left < 16) {
77         insertionSort(arr, left, right);
78         return;

```

```

79     }
80
81     // Если глубина рекурсии превысила лимит, используем Heap Sort
82     if (depthLimit == 0) {
83         heapSort(arr, left, right);
84         return;
85     }
86
87     // Иначе продолжаем Quick Sort
88     int pi = partition(arr, left, right);
89     introSort(arr, left, pi - 1, depthLimit - 1);
90     introSort(arr, pi + 1, right, depthLimit - 1);
91 }
92
93 public:
94     SortTester() {}
95
96     // Тестирование Introsort
97     long long testHybridSort(const std::vector<int>& originalArray) {
98         std::vector<int> arr = originalArray;
99         auto start = std::chrono::high_resolution_clock::now();
100
101        if (!arr.empty()) {
102            int depthLimit = 2 * log2(arr.size());
103            introSort(arr, 0, arr.size() - 1, depthLimit);
104        }
105
106        auto elapsed = std::chrono::high_resolution_clock::now() - start;
107        return std::chrono::duration_cast<std::chrono::microseconds>(elapsed).count();
108    }
109};

```

Для реализации представления графиков был реализован main() с записью результатов эксперимента с каждым видом массива в csv-файлы:

```

1 int main() {
2     SortTester tester;
3     ArrayGenerator generator;
4
5     const size_t MAX_SIZE = 100000;
6     const size_t MIN_SIZE = 500;
7     const size_t STEP = 1000;
8
9     auto randomBase = generator.generateRandomArray(MAX_SIZE);
10    auto reverseBase = generator.generateReverseSortedArray(MAX_SIZE);
11    auto almostBase = generator.generateAlmostSortedArray(MAX_SIZE, 50);
12
13    std::ofstream randomFile("hybrid_quick_sort_random.csv");
14    std::ofstream reverseFile("hybrid_quick_sort_reverse.csv");
15    std::ofstream almostFile("hybrid_quick_sort_almost.csv");
16
17    randomFile << "size,time(ms)\n";
18    reverseFile << "size,time(ms)\n";
19    almostFile << "size,time(ms)\n";
20
21    for (size_t size = MIN_SIZE; size <= MAX_SIZE; size += STEP) {
22        auto randomArray = generator.getSubarray(randomBase, size);
23        long long hybridTime = tester.testHybridSort(randomArray);
24
25        randomFile << size << "," << hybridTime << "\n";
26
27        auto reverseArray = generator.getSubarray(reverseBase, size);
28        hybridTime = tester.testHybridSort(reverseArray);
29
30        reverseFile << size << "," << hybridTime << "\n";
31
32        auto almostArray = generator.getSubarray(almostBase, size);

```

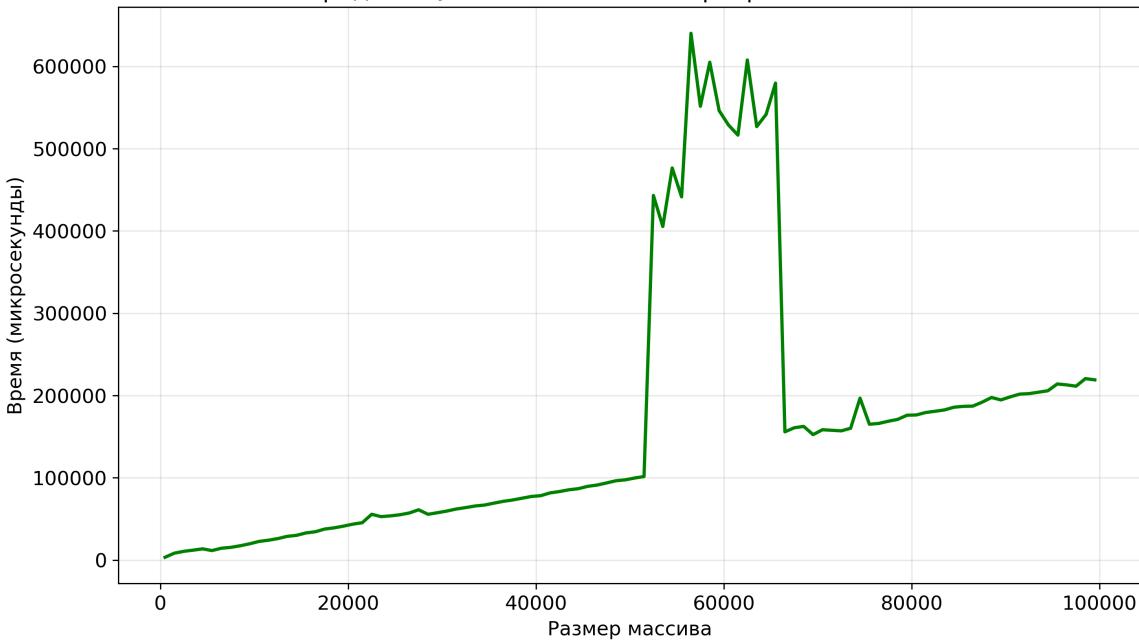
```
33     hybridTime = tester.testHybridSort(almostArray);
34
35     almostFile << size << "," << hybridTime << "\n";
36 }
37
38 randomFile.close();
39 reverseFile.close();
40 almostFile.close();
41
42 return 0;
43 }
```

Графики

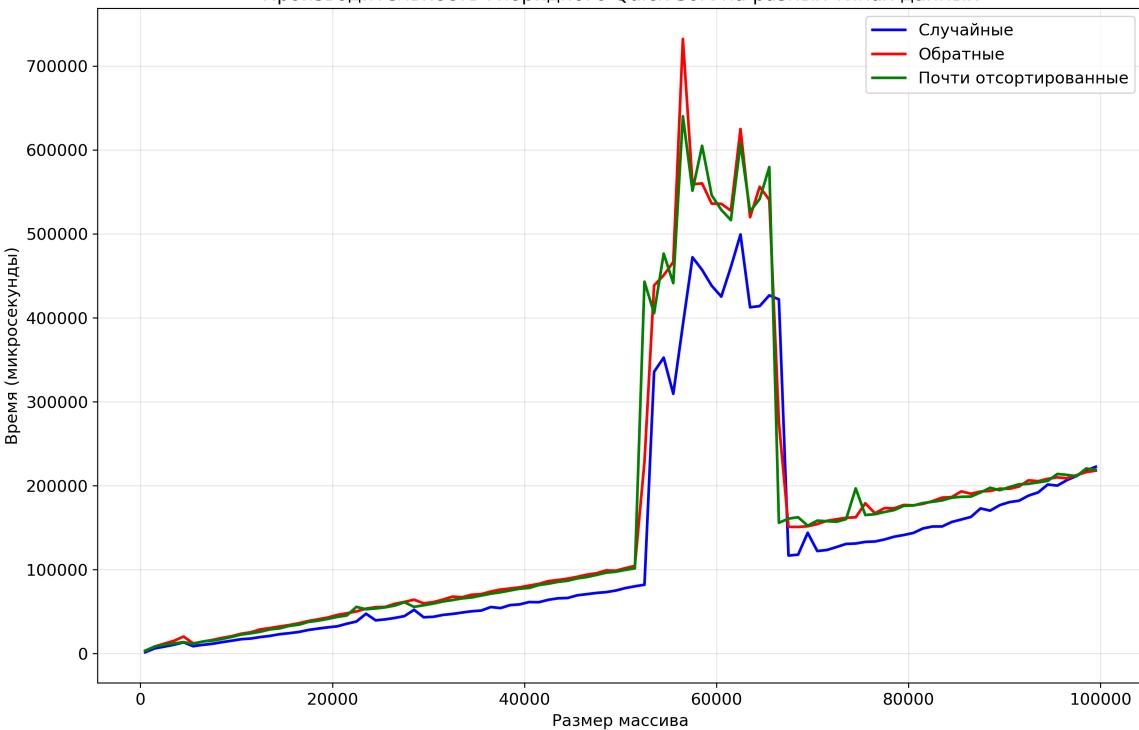
Были получены следующие графики:



Гибридный Quick Sort: Почти отсортированные массивы



Производительность гибридного Quick Sort на разных типах данных



Этап 4

На основе проведенного экспериментального исследования двух реализаций алгоритма быстрой сортировки можно сделать следующие выводы:

1. Стандартный Quick Sort продемонстрировал хорошую производительность на случайных массивах, но показал уязвимость на специфических наборах данных. На обратно отсортированных массивах наблюдалось значительное замедление работы, что соответствует теоретической оценке худшего случая $O(n^2)$.

2. Гибридный алгоритм (Introsort) показал существенное преимущество на сложных наборах данных. На обратно отсортированных массивах гибридная версия работала значительно стабильнее благодаря автоматическому переключению на Heap Sort при достижении критической глубины рекурсии.

3. На почти отсортированных массивах оба алгоритма показали хорошие результаты, но гибридная версия демонстрировала более предсказуемое поведение за счет использования Insertion Sort для неболь-

ших подмассивов.

4. На больших массивах (80000-100000 элементов) гибридный алгоритм показал лучшую масштабируемость, особенно на неблагоприятных наборах данных, где стандартный Quick Sort мог демонстрировать квадратичную сложность.

Прочие данные

Номер посылки на codeforces: 348800334

Ссылка на публичный репозиторий с материалами(в том числе с реализациями классов ArrayGenerator и SortTester): github.com/sunflowerthu/ADS_A3b_SET3