Задача A3b

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:

```
    int partition(std::vector<int>& arr, int left, int right) {
    std::random_device rd;
    std::mt19937 gen(rd());
    std::uniform_int_distribution<int> dis(left, right);
    int pivotIndex = dis(gen);
    std::swap(arr[pivotIndex], arr[right]);

    int pivot = arr[right];
    int i = left - 1;

    for (int j = left; j < right; j++) {
        if (arr[j] <= pivot) {
            i++;
            std::swap(arr[i], arr[j]);
        }
    }
    std::swap(arr[i + 1], arr[right]);
    return i + 1;
}

void standardQuickSort(std::vector<int>& arr, int left, int right) {
    if (left < right) {
        int pi = partition(arr, left, right);
        standardQuickSort(arr, left, pi - 1);
        standardQuickSort(arr, pi + 1, right);
    }
}

void quickSort(std::vector<int>& arr) {
    if (arr.empty()) return;
    standardQuickSort(arr, 0, arr.size() - 1);
}
```

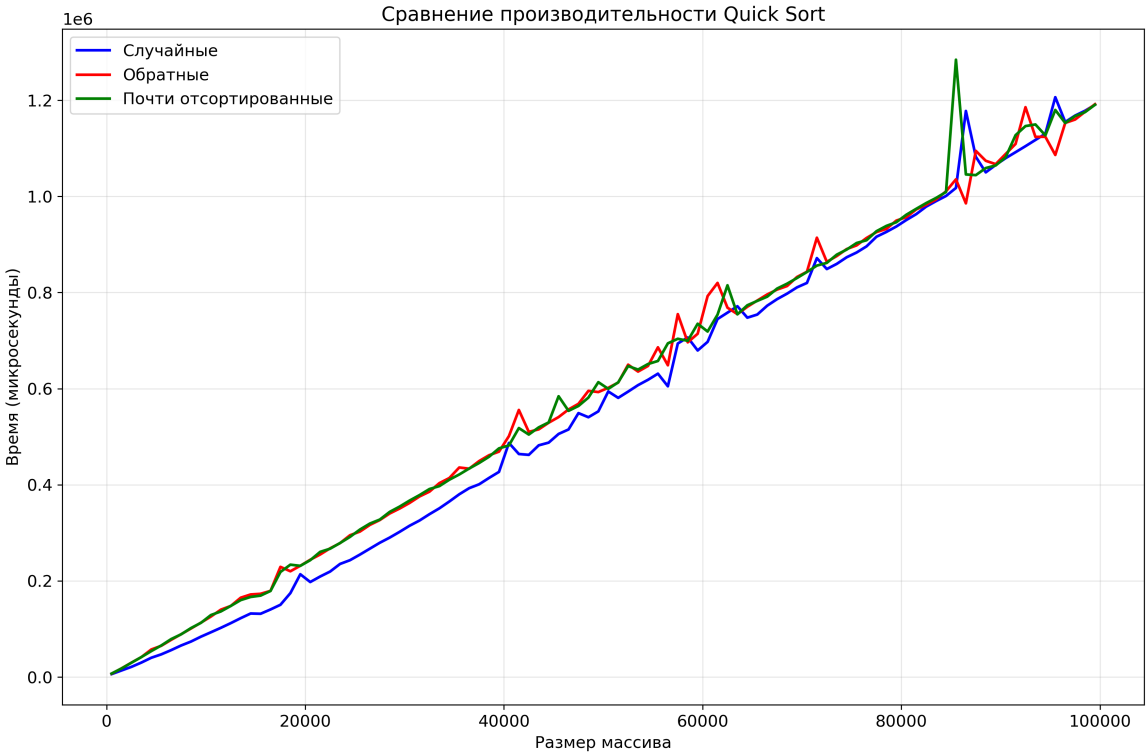
Для реализации представления графиков был реализован main() с записью результатов эксперимента с каждым видом массива в csv-файлы:

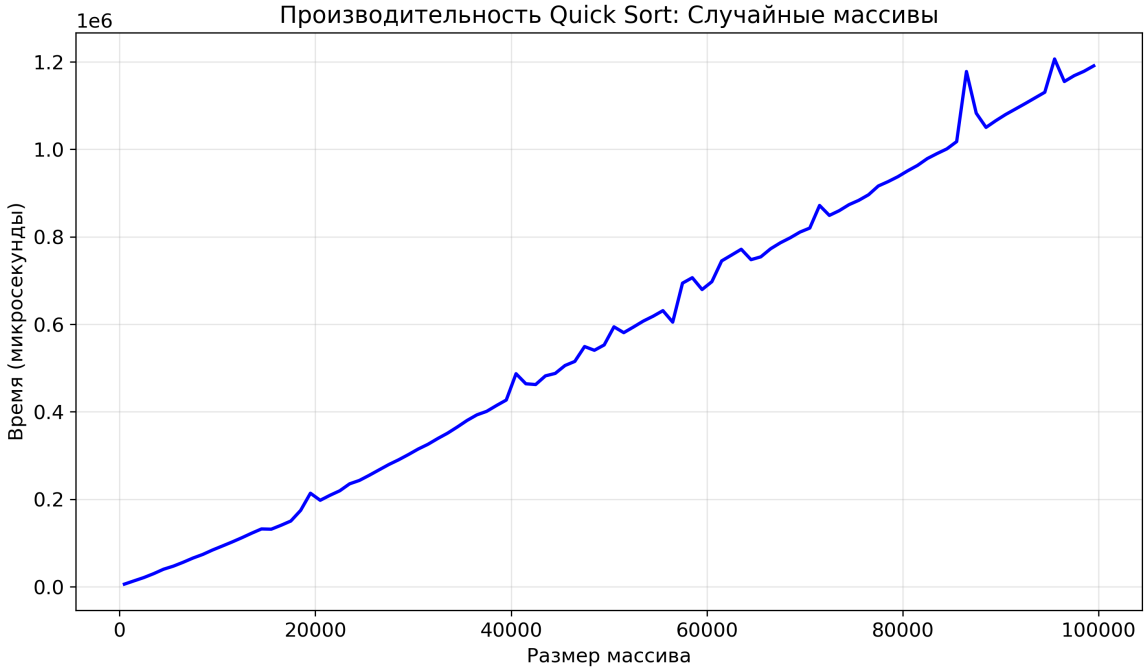
```
    int main() {
    ArrayGenerator generator;

    const size_t MAX_SIZE = 100000;
    const size_t MIN_SIZE = 500;
    const size_t STEP = 1000;

    auto randomBase = generator.generateRandomArray(MAX_SIZE);
    auto reverseBase = generator.generateReverseSortedArray(MAX_SIZE);
    auto almostBase = generator.generateAlmostSortedArray(MAX_SIZE, 50);
```

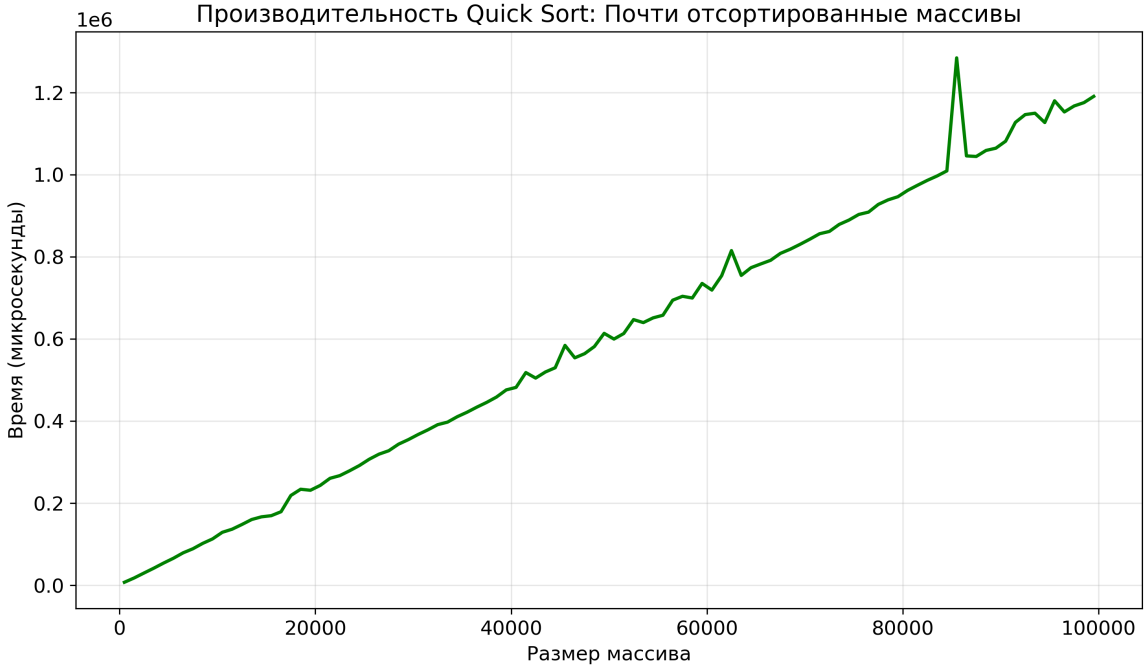
```
11
12      std::ofstream randomFile("quick_sort_random.csv");
13      std::ofstream reverseFile("quick_sort_reverse.csv");
14      std::ofstream almostFile("quick_sort_almost.csv");
15
16      randomFile << "size,time(ms)\n";
17      reverseFile << "size,time(ms)\n";
18      almostFile << "size,time(ms)\n";
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          reverseFile << size << "," << reverseTime << "\n";
34
35          auto almostArray = generator.getSubarray(almostBase, size);
36          start = std::chrono::high_resolution_clock::now();
37          quickSort(almostArray);
38          elapsed = std::chrono::high_resolution_clock::now() - start;
39          long long almostTime = std::chrono::duration_cast<std::chrono::microseconds>(elapsed).count();
40          almostFile << size << "," << almostTime << "\n";
41      }
42
43      randomFile.close();
44      reverseFile.close();
45      almostFile.close();
46
47      return 0;
48  }
```

Графики

Были получены следующие графики:







Этап 3

Реализация класса SortTester:

```
class SortTester {
private:
    ArrayGenerator generator;

    // Insertion Sort для небольших массивов
    void insertionSort(std::vector<int>& arr, int left, int right) {
        for (int i = left + 1; i <= right; i++) {
            int key = arr[i];
            int j = i - 1;
            while (j >= left && arr[j] > key) {
                arr[j + 1] = arr[j];
                j--;
```

```
            }
            arr[j + 1] = key;
        }
    }

    void heapify(std::vector<int>& arr, int heap_size, int i, int offset) {
        int largest = i;
        int left = 2 * (i - offset) + 1 + offset;
        int right = 2 * (i - offset) + 2 + offset;

        if (left < heap_size && arr[left] > arr[largest])
            largest = left;

        if (right < heap_size && arr[right] > arr[largest])
            largest = right;

        if (largest != i) {
            std::swap(arr[i], arr[largest]);
            heapify(arr, heap_size, largest, offset);
        }
    }

    // Heap Sort
    void heapSort(std::vector<int>& arr, int left, int right) {
        int n = right - left + 1;

        // Строим кучу (max-heap)
        for (int i = left + n/2 - 1; i >= left; i--) {
            heapify(arr, right + 1, i, left);
        }

        // Извлекаем элементы из кучи
        for (int i = right; i > left; i--) {
            std::swap(arr[left], arr[i]);
            heapify(arr, i, left, left);
        }
    }

    // Quick Sort со случайным опорным элементом
    int partition(std::vector<int>& arr, int left, int right) {
        // Случайный выбор опорного элемента
        std::random_device rd;
        std::mt19937 gen(rd());
        std::uniform_int_distribution<int> dis(left, right);
        int pivotIndex = dis(gen);
        std::swap(arr[pivotIndex], arr[right]);

        int pivot = arr[right];
        int i = left - 1;

        for (int j = left; j < right; j++) {
            if (arr[j] <= pivot) {
                i++;
                std::swap(arr[i], arr[j]);
            }
        }
        std::swap(arr[i + 1], arr[right]);
        return i + 1;
    }

    // Introsort
    void introSort(std::vector<int>& arr, int left, int right, int depthLimit) {
        // Если элементов мало, используем Insertion Sort
        if (right - left < 16) {
            insertionSort(arr, left, right);
            return;
```

```
        }

        // Если глубина рекурсии превысила лимит, используем Heap Sort
        if (depthLimit == 0) {
            heapSort(arr, left, right);
            return;
        }

        // Иначе продолжаем Quick Sort
        int pi = partition(arr, left, right);
        introSort(arr, left, pi - 1, depthLimit - 1);
        introSort(arr, pi + 1, right, depthLimit - 1);
    }

public:
    SortTester() {}

    // Тестирование Introsort
    long long testHybridSort(const std::vector<int>& originalArray) {
        std::vector<int> arr = originalArray;
        auto start = std::chrono::high_resolution_clock::now();

        if (!arr.empty()) {
            int depthLimit = 2 * log2(arr.size());
            introSort(arr, 0, arr.size() - 1, depthLimit);
        }

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

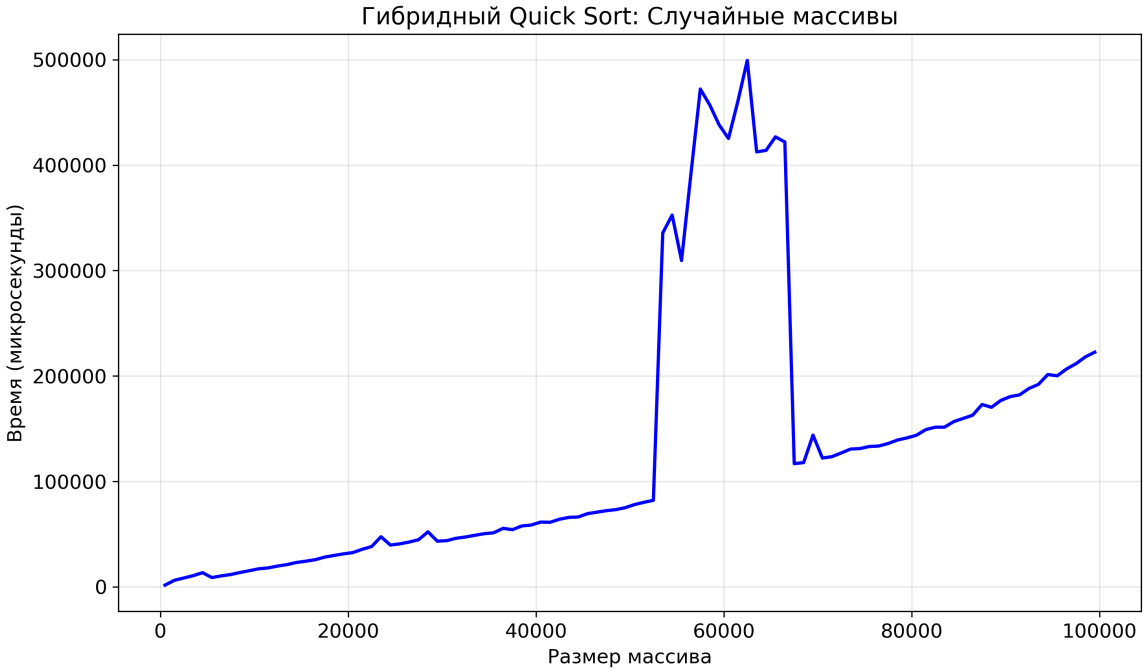
Для реализации представления графиков был реализован main() с записью результатов эксперимента с каждым видом массива в csv-файлы:

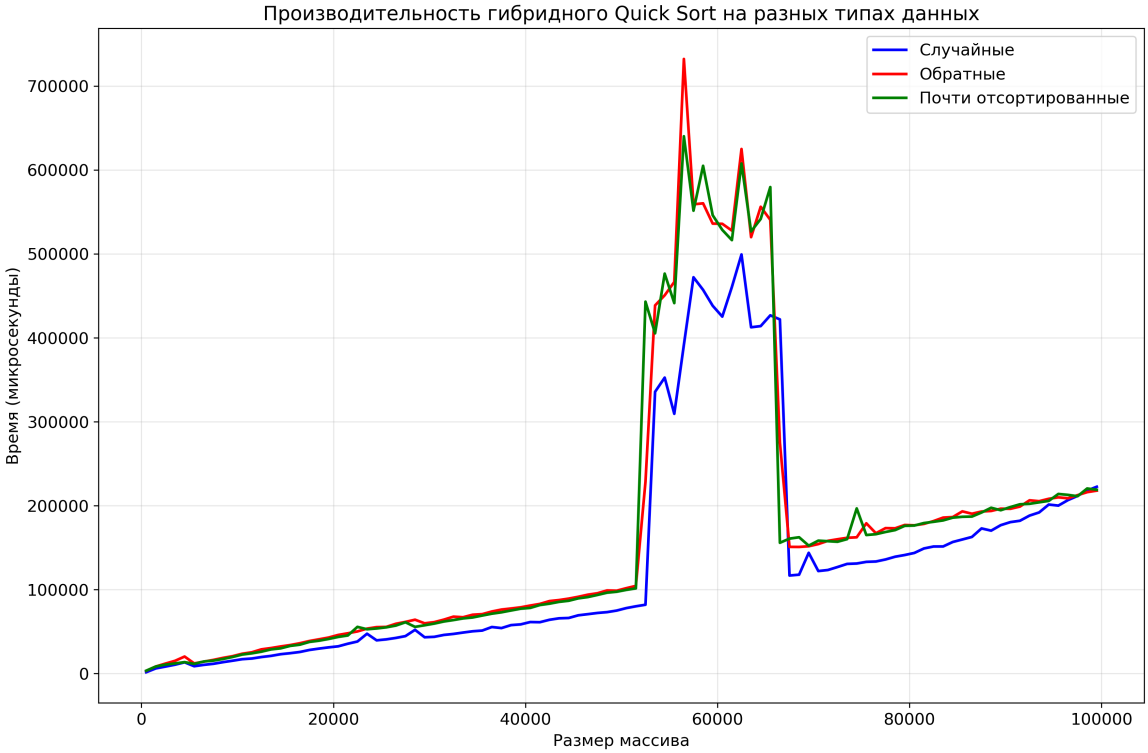
```
    int main() {
    SortTester tester;
    ArrayGenerator generator;

    const size_t MAX_SIZE = 100000;
    const size_t MIN_SIZE = 500;
    const size_t STEP = 1000;

    auto randomBase = generator.generateRandomArray(MAX_SIZE);
    auto reverseBase = generator.generateReverseSortedArray(MAX_SIZE);
    auto almostBase = generator.generateAlmostSortedArray(MAX_SIZE, 50);

    std::ofstream randomFile("hybrid_quick_sort_random.csv");
    std::ofstream reverseFile("hybrid_quick_sort_reverse.csv");
    std::ofstream almostFile("hybrid_quick_sort_almost.csv");

    randomFile << "size,time(ms)\n";
    reverseFile << "size,time(ms)\n";
    almostFile << "size,time(ms)\n";

    for (size_t size = MIN_SIZE; size <= MAX_SIZE; size += STEP) {
        auto randomArray = generator.getSubarray(randomBase, size);
        long long hybridTime = tester.testHybridSort(randomArray);

        randomFile << size << "," << hybridTime << "\n";

        auto reverseArray = generator.getSubarray(reverseBase, size);
        hybridTime = tester.testHybridSort(reverseArray);

        reverseFile << size << "," << hybridTime << "\n";

        auto almostArray = generator.getSubarray(almostBase, size);
```

```
        hybridTime = tester.testHybridSort(almostArray);

        almostFile << size << "," << hybridTime << "\n";
    }

    randomFile.close();
    reverseFile.close();
    almostFile.close();

    return 0;
}
```

Графики

Были получены следующие графики:





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`