Министерство науки и высшего образования Российской Федерации
Федеральное государственное автономное образовательное учреждение
высшего образования
«СЕВЕРО-КАВКАЗСКИЙ ФЕДЕРАЛЬНЫЙ УНИВЕРСИТЕТ»

Институт цифрового развития
Кафедра инфокоммуникаций

ОТЧЕТ
ПО ПРАКТИЧЕСКОЙ РАБОТЕ №10
дисциплины «Алгоритмизация»
Вариант___

Выполнил:
Репкин Александр Павлович
2 курс, группа ИВТ-б-о-22-1,
09.03.01 «Информатика и
вычислительная техника»,
направленность (профиль)
«Программное обеспечение средств
вычислительной
техники и автоматизированных
систем», очная форма обучения

_____
(подпись)

Руководитель практики:
Воронкин Р.А., канд. техн. наук,
доцент кафедры инфокоммуникаций

_____
(подпись)

Отчет защищен с оценкой _____ Дата защиты_____

Ставрополь, 2023 г.

Порядок выполнения работы:

1. Создана программа на основе поставленных заданий – в первом задании требовалось реализовать алгоритм сортировки Heap Sort (Сортировка кучами) на любом языке (Выбран язык Java), после чего было необходимо его протестировать на разных входных данных (Отсортированный массив, Отсортированный массив в обратном порядке, Случайный массив). Помимо этого, в задании №2 требовалось дополнить полученную программу, добавив также (Для сравнения производительности) другие алгоритмы сортировки массивов – Quick Sort (Быстрая сортировка) и Merge Sort (Сортировка слиянием). В задании №3 требовалось оптимизировать изначальный алгоритм Heap Sort, после чего также сравнить с предыдущими алгоритмами сортировки. Из полученных данных видно, что при оптимизации, Heap Sort справляется лучше, чем Merge Sort и Quich Sort, из чего следует вывод, что Heap Sort может быть наилучшим выбором, если требуется эффективная сортировка больших массивов (Так как временные затраты = O(n log(n)), то при любых размерах входных данных алгоритм затратит минимальное количество времени).

```java
/* Сортировка кучей, НЕ(!)оптимизированная, затраты времени примерно = O(n * log(n)).
Тут использована стандартная реализация кучи (PriorityQueue) для сортировки полученного массива.
После сортировки элементы извлекаются из кучи и помещаются обратно в массив.
*/
static void heapSortUnoptimised(ArrayList<Integer> array) {  1 usage
    PriorityQueue<Integer> minHeap = new PriorityQueue<>();
    for (int element : array) minHeap.offer(element);
    for (int i = 0; i < arraySize; i++) array.set(i, minHeap.poll());
}
```

Рисунок 1. Полученный изначальный код Heap Sort.

```java
static int partition(ArrayList<Integer> array, int low, int high) {  1 usage
    int x = array.get(low);
    int i = low;
    for (int j = low + 1; j <= high; j++) {
        if (array.get(j) <= x) {
            i++;
            int swapping = array.get(i);
            array.set(i, array.get(j));
            array.set(j, swapping);
        }
    }
    int swapping = array.get(low);
    array.set(low, array.get(i));
    array.set(i, swapping);
    return i;
}

/* Быстрая сортировка, время работы в худшем случае колеблется от O(n * log(n)) до O(n^2).
Исходный массив разделяется в partition на две части (Подробнее я указал в комментарии
над самим partition). Полученные две части опять делятся и так до тех пор, пока не будет отсортирован весь массив.
 */
static void quickSort(ArrayList<Integer> array, int low, int high) {  2 usages
    while (low < high) {
        int partitionIndex = partition(array, low, high);
        quickSort(array, low,  high: partitionIndex - 1);
        low = partitionIndex + 1;
    }
}
```

Рисунок 2. Полученный код Quick Sort.

```java
/* Сортировка слиянием, время работы = O (n * log(n)).
Тут происходит разделение массива на две части (От start до middle и от middle до finish), которые
сортируются по отдельности, после чего их объединяет функция merge.
Повторяется, пока весь массив не будет отсортирован.
 */
static void mergeSort(ArrayList<Integer> array, int start, int finish) {  3 usages
    if (start < finish) {
        int middle = (start + finish) / 2;
        mergeSort(array, start, middle);
        mergeSort(array,  start: middle + 1, finish);
        merge(array, start, middle, finish);
    }
}
```

Рисунок 3. Полученный код Merge Sort.

```java
private static void siftDown(ArrayList<Integer> heap, int size, int i) {  3 usages
    int largest = i;
    int start = 2 * i + 1;
    int finish = 2 * i + 2;
    // Проверка, есть ли потомок слева, и больше ли он текущего наиб. элемента. Если да - он становится наибольшим.
    if (start < size && heap.get(start) > heap.get(largest)) largest = start;
    // Проверка, есть ли потомок справа, и больше ли он текущего наиб. элемента. Если да - он становится наибольшим.
    if (finish < size && heap.get(finish) > heap.get(largest)) largest = finish;
    // Если наибольший элемент не рассматриваемый элемент (i), то меняем их местами.
    if (largest != i) {
        int temp = heap.get(i);
        heap.set(i, heap.get(largest));
        heap.set(largest, temp);
        // Проверка поддерева.
        siftDown(heap, size, largest);
    }
}

// Сортировка кучей на месте, затраты времени = O(n * log(n)).
static void heapSort(ArrayList<Integer> array) {  1 usage
    int size = arraySize;
    // Построение кучи.
    for (int i = size / 2 - 1; i >= 0; i--) siftDown(array, size, i);
    // Извлечение элементов из кучи и помещение их в конец массива.
    for (int i = size - 1; i > 0; i--) {
        int swapping = array.get(0);
        array.set(0, array.get(i));
        array.set(i, swapping);
        siftDown(array, i,  i: 0);
    }
}
```

Рисунок 4. Полученный Оптимизированный алгоритм Heap Sort.

```
Original array [34, 86, 57, 106, 74, 36, 83, 54, 111, 57, 17, 27, 46, 88, 70, 75, 63, 97, 57, 32, 56, 67, 56, 65, 12, 83, 71, 67, 0, 16, 57, 10, 61, 3, 47, 60, 65, 40, 65, 38, 78
Quick Sort [0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 2, 2, 2, 2, 2, 2, 2, 2, 3, 4, 4, 4, 4, 4, 5, 5, 5, 5, 6, 6, 7, 7, 8, 8, 8, 9, 9, 9, 9, 10, 10, 11, 12, 12, 12, 13, 16, 17, 17, 18, 19, 19,
Heap Sort [0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 2, 2, 2, 2, 2, 2, 2, 2, 3, 4, 4, 4, 4, 4, 5, 5, 5, 5, 6, 6, 7, 7, 8, 8, 8, 9, 9, 9, 9, 10, 10, 11, 12, 12, 12, 13, 16, 17, 17, 18, 19, 19, 2
Unoptimised Heap Sort [0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 2, 2, 2, 2, 2, 2, 2, 2, 3, 4, 4, 4, 4, 4, 5, 5, 5, 5, 6, 6, 7, 7, 8, 8, 8, 9, 9, 9, 9, 10, 10, 11, 12, 12, 12, 13, 16, 17, 17, 1
Merge Sort [0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 2, 2, 2, 2, 2, 2, 2, 2, 3, 4, 4, 4, 4, 4, 5, 5, 5, 5, 6, 6, 7, 7, 8, 8, 8, 9, 9, 9, 9, 10, 10, 11, 12, 12, 12, 13, 16, 17, 17, 18, 19, 19,
```

Рисунок 5. Проверка правильности работы алгоритмов.

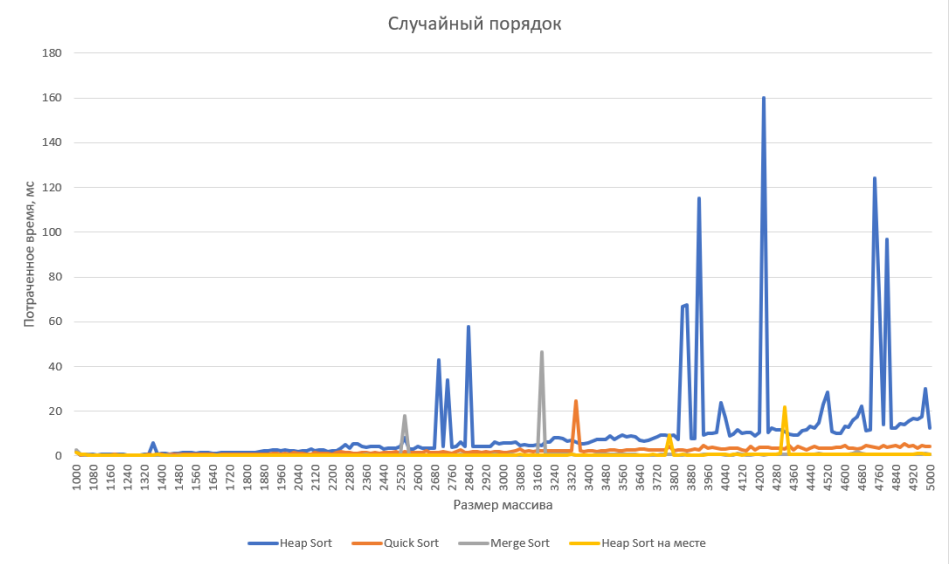

Рисунок 6. Полученный график сортировки случайного массива.

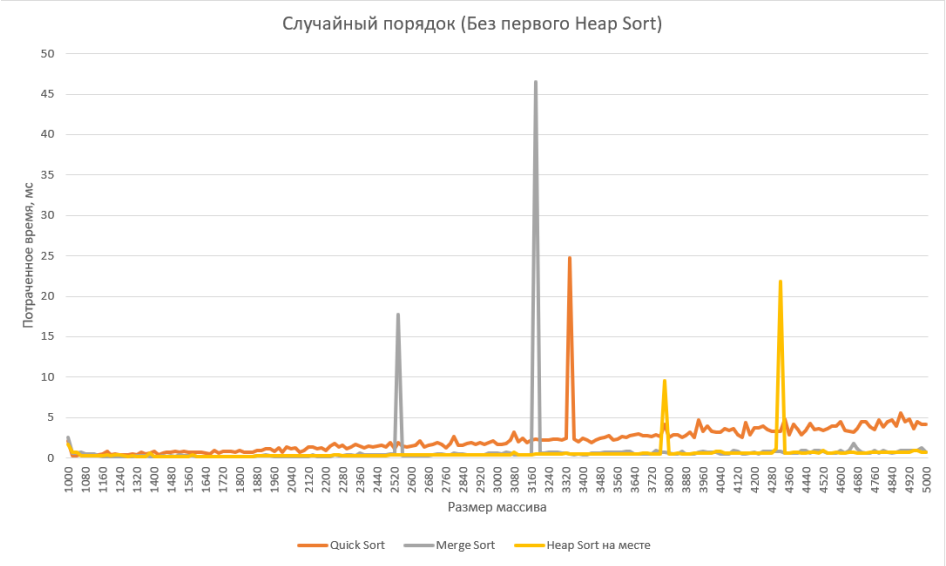

Рисунок 7. Полученный график сортировки случайного массива без первого варианта Heap Sort.

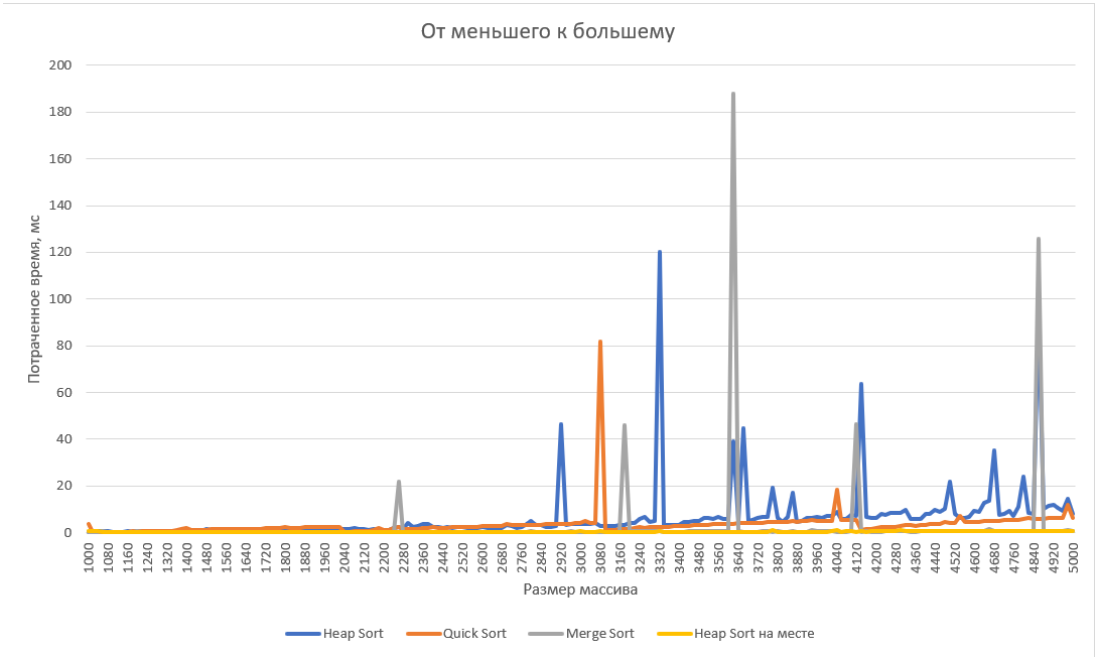

Рисунок 8. Полученный график сортировки отсортированного массива.

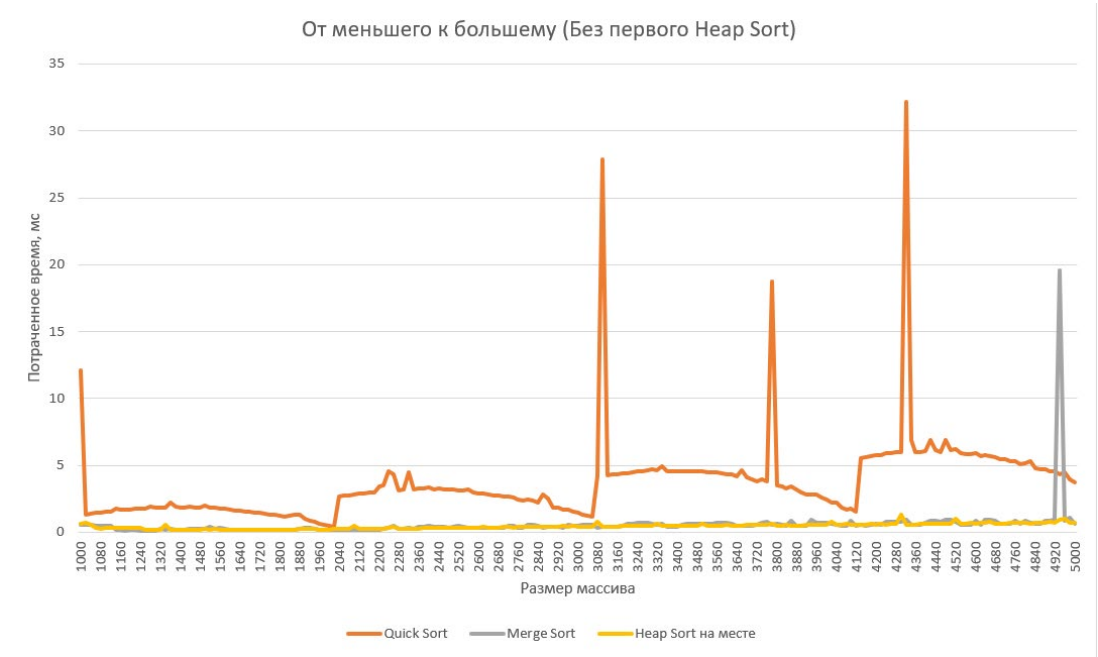

Рисунок 9. Полученный график сортировки отсортированного массива без первого варианта Heap Sort.

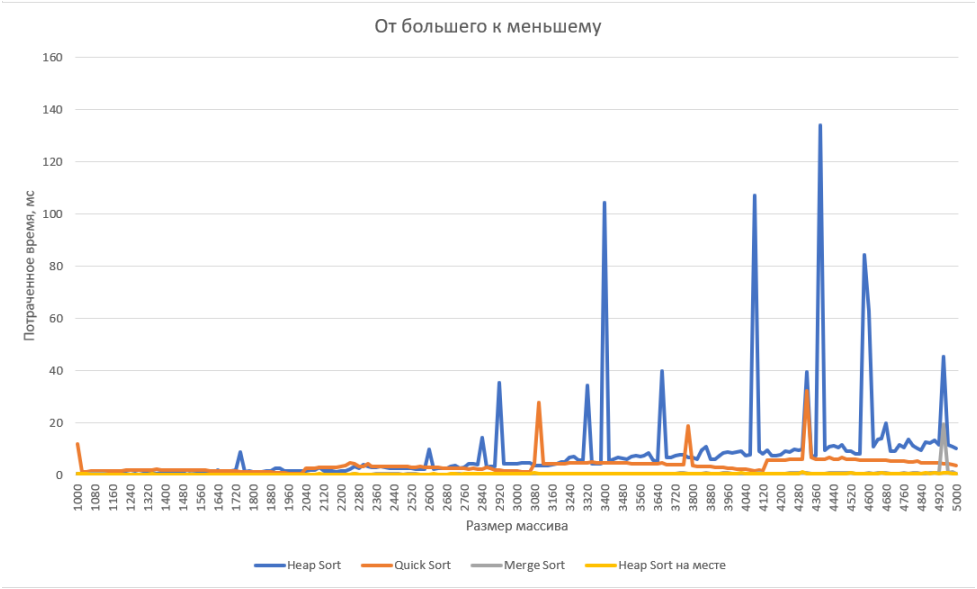

Рисунок 10. Полученный график сортировки отсортированного наоборот массива.

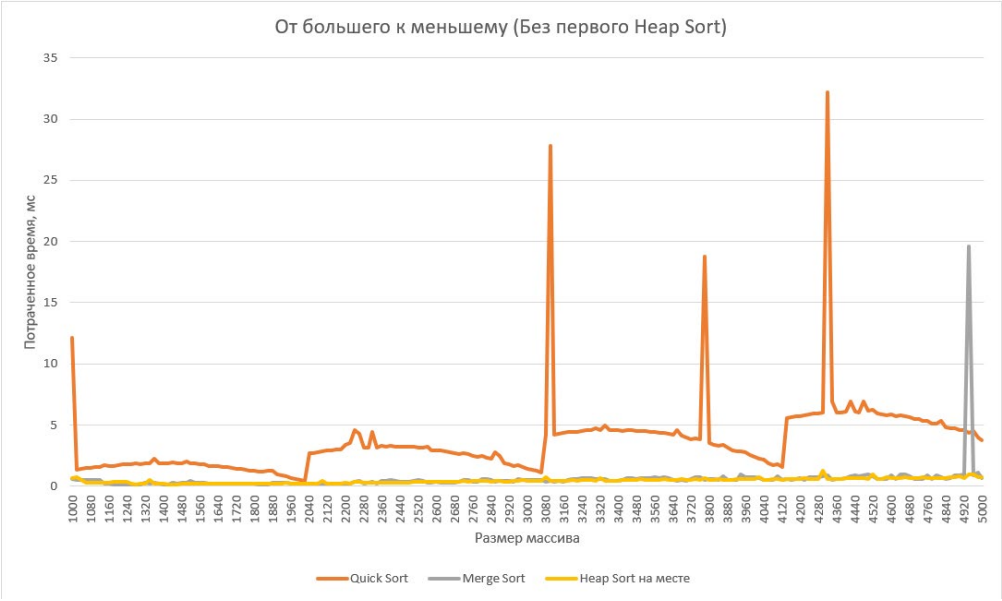

Рисунок 11. Полученный график сортировки отсортированного наоборот массива без первого варианта Heap Sort

2. Согласно полученным данным, оптимизированная версия Heap Sort всегда затрачивает O (n*log(n)) времени: первый цикл (для построения кучи) имеет сложность O(n), так как каждый элемент рассматривается ровно один раз, и для каждого элемента выполняется siftDown (затрачивающий O(log(n)) времени) => на весь цикл уходит O(n*log(n)) времени; второй цикл (извлечение элементов), на каждой итерации обменивает корень кучи с последним элементом (O(n)) и вызывает siftDown (O(log(n)) для уменьшенной

кучи, на всё это также уходит O(n*log n) времени. Так, получаемая сложность Heap Sort = O(n * log(n)) + O(n * log(n)) = O(2 * n * log(n)) = O(n * log(n)), так как 2 не влияет на сам рост функции.

3. Выполнено задание №6. Согласно заданию, даны массивы A[1…n] и B[1…n], необходимо вывести все n^2 сумм вида A[i]+B[j] в возрастающем порядке. Было также указано, что наивный способ решения – создать массив, содержащий все такие суммы, и отсортировать его, однако такой алгоритм имеет время работы O(n^2 *log(n)) и использует O(n^2) памяти. Так, было необходимо привести алгоритм с таким же временем работы, который использует линейную память.

```java
public static void calculatingSums(ArrayList<Integer> first, ArrayList<Integer> second) {  1 usage
    if (first.isEmpty() || second.isEmpty()) return;
    // Сортировка обоих списков по возрастанию
    first.sort(Comparator.naturalOrder());
    second.sort(Comparator.naturalOrder());
    // Приоритетная очередь для отслеживания минимальных сумм с их индексами
    PriorityQueue<int[]> heap = new PriorityQueue<>(Comparator.comparingInt(a -> a[0]));
    // Множество для отслеживания уже рассмотренных индексов
    HashSet<int[]> used = new HashSet<>();
    // Добавление первой суммы в очередь и множество
    heap.add(new int[]{first.get(0) + second.get(0), 0, 0});
    used.add(new int[]{0, 0});
    for (int count = 0; count < first.size() * second.size(); count++) {
        // Извлечение минимальной суммы из очереди
        int[] current = heap.poll();
        int sum = current[0];
        int i = current[1];
        int j = current[2];
        // Удаление уже полученных только что индексов из множества
        used.remove(new int[]{i, j});
        // Вывод текущей суммы
        System.out.print(sum + ", ");
        // Проверка и добавление следующих сумм, если индексы не рассмотрены
        if (i + 1 < first.size() && !used.contains(new int[]{i + 1, j})) {
            heap.add(new int[]{first.get(i + 1) + second.get(j), i + 1, j});
            used.add(new int[]{i + 1, j});
        }
        if (j + 1 < second.size() && !used.contains(new int[]{i, j + 1})) {
            heap.add(new int[]{first.get(i) + second.get(j + 1), i, j + 1});
            used.add(new int[]{i, j + 1});
        }
    }
```

Рисунок 12. Полученный код задания №6.

```
Good day! Lists are:
First: [12, 71, 32, 43, 100, 51]
Second: [41, 66, 50, 78, 7, 11]
Summs are:
19, 23, 39, 43, 43, 50, 53, 54, 54, 54, 58, 62, 62, 62, 62, 62, 73, 73, 73, 78, 78, 82, 82, 82, 82, 82, 82, 82, 82, 82, 84, 84, 84, 84, 84, 84,
```

Рисунок 13. Полученный результат задания №6.

Вывод: в ходе выполнения практической работы были рассмотрены способы сортировки массивов (Heap Sort, Heap Sort оптимизированный, Quick Sort, Merge Sort). Из полученных данных выявлена значимость оптимизации (В то время как изначальный Heap Sort справлялся хуже всех, при

оптимизации, полученные значения было всегда лучше, чем Quick Sort и Merge Sort).