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 №6
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. На основе псевдокода создано два варианта программы PointsCover. Данная программа покрывает полученные точки минимальным количеством отрезков, однако первый алгоритм программы (Плохой) не оптимизирован, решение в нём идёт напрямую, в то время как во втором алгоритме программы (Хорошем), код оптимизирован, что затрачивает меньше времени на выполнение. Так, первый алгоритм затрачивает $O(n^2)$, а второй $O(n * \log(n))$.

```
public static List<List<Double>> bad(List<Double> numbers) {  1 usage
    List<Double> usefull_copy = new ArrayList<>(numbers.size());
    usefull_copy.addAll(numbers);
    List<List<Double>> results = new ArrayList<>();
    long startTime = System.nanoTime();
    while (!usefull_copy.isEmpty()) {
        double xm = Collections.min(usefull_copy);
        results.add(List.of(xm, xm + 1));
        int i = 0;
        while (i < usefull_copy.size()) {
            if (results.get(results.size() - 1).get(0) <= usefull_copy.get(i) &&
                    usefull_copy.get(i) <= results.get(results.size() - 1).get(1)) {
                usefull_copy.remove(i);
            } else {
                i++;
            }
        }
    }
    long endTime = System.nanoTime();
    long spent_time = (endTime - startTime);  // divide by 1000000 to get milliseconds.
    System.out.println("Bad variant of PointsCover took " + (spent_time / 1000000) + ","
            + ((spent_time / 10000) % 100));
    return results;
}
```

Рисунок 1. Плохой алгоритм PointsCover.

```
public static List<List<Double>> good(List<Double> numbers) {  1 usage
    List<List<Double>> results = new ArrayList<>();
    long startTime = System.nanoTime();
    int i = 0;
    while (i < numbers.size()) {
        double xm = numbers.get(i);
        results.add(List.of(xm, xm + 1));
        i++;
        while (i < numbers.size() && numbers.get(i) <= xm + 1) {
            i++;
        }
    }
    long endTime = System.nanoTime();
    long spent_time = (endTime - startTime);  // divide by 1000000 to get milliseconds.
    System.out.println("Good variant of PointsCover took " + (spent_time / 1000000) +
            "," + ((spent_time / 10000) % 100));
    return results;
}
```

Рисунок 2. Хороший алгоритм PointsCover.

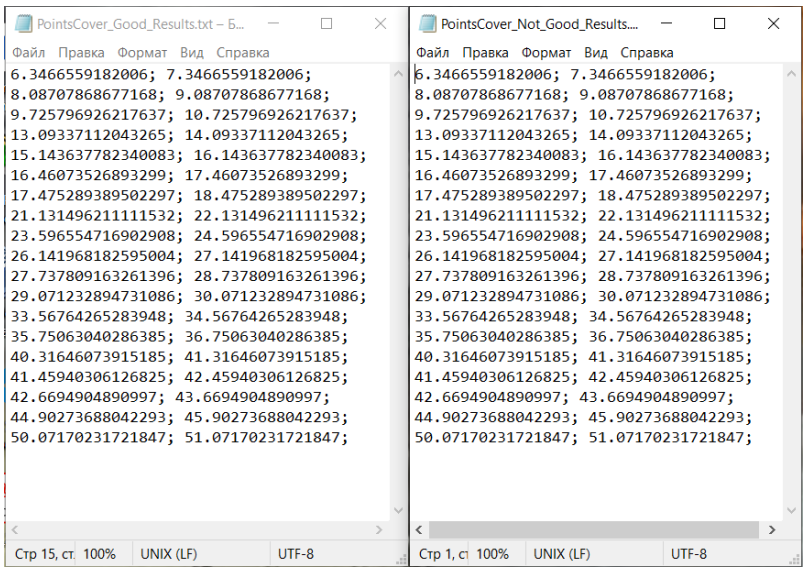

Рисунок 3. Полученные результаты выполнения идентичны.

```
Bad variant of PointsCover took 1,34
Good variant of PointsCover took 0,7
```

Рисунок 4. Сравнение времени выполнения программ, в мс.

2. На основе псевдокода создано два варианта программы ActSel. Данная программа находит максимальное количество не пересекающихся отрезков, однако первый алгоритм программы (Плохой) не оптимизирован, решение в нём идёт напрямую, в то время как во втором алгоритме программы (Хорошем), код оптимизирован, что затрачивает меньше времени на выполнение. Так, первый алгоритм затрачивает $O(n^2)$, а второй $O(n * \log(n))$.

```
public static List<int[]> bad(List<int[]> borders) {  1 usage
    List<int[]> results = new ArrayList<>();
    List<int[]> usefull_copy = new ArrayList<>(borders.size());
    usefull_copy.addAll(borders);
    long startTime = System.nanoTime();
    while (!usefull_copy.isEmpty()) {
        int m = Integer.MAX_VALUE;
        for (int[] x : usefull_copy) {
            m = Math.min(m, x[1]);
        }
        int minIndex = 0;
        for (int i = 0; i < usefull_copy.size(); i++) {
            if (m == usefull_copy.get(i)[1]) {
                minIndex = i;
                break;
            }
        }
        int[] d = usefull_copy.get(minIndex);
        results.add(d);
        int i = 0;
        while (i < usefull_copy.size()) {
            if (usefull_copy.get(i)[0] <= d[1]) {
                usefull_copy.remove(i);
            } else {
                i++;
            }
        }
    }
    long endTime = System.nanoTime();
    long spent_time = (endTime - startTime);  // divide by 1000000 to get milliseconds.
    System.out.println("Bad variant of ActSel took " + (spent_time / 1000000) + "," + ((spent_time / 10000) % 100));
    return results;
```

Рисунок 5. Плохой алгоритм ActSel.

```java
public static List<int[]> good(List<int[]> borders) {  1 usage
    List<int[]> usefull_copy = new ArrayList<>(borders.size());
    List<int[]> results = new ArrayList<>();
    usefull_copy.addAll(borders);
    usefull_copy.sort(Comparator.comparingInt(x -> x[1]));
    long startTime = System.nanoTime();
    results.add(usefull_copy.get(0));
    for (int[] i : usefull_copy) {
        if (i[0] > results.get(results.size() - 1)[1]) {
            results.add(i);
        }
    }
    long endTime = System.nanoTime();
    long spent_time = (endTime - startTime);  // divide by 1000000 to get milliseconds.
    System.out.println("Good variant of ActSel took " + (spent_time / 1000000) +
            "," + ((spent_time / 10000) % 100));
    return results;
}
```

Рисунок 6. Хороший алгоритм ActSel.

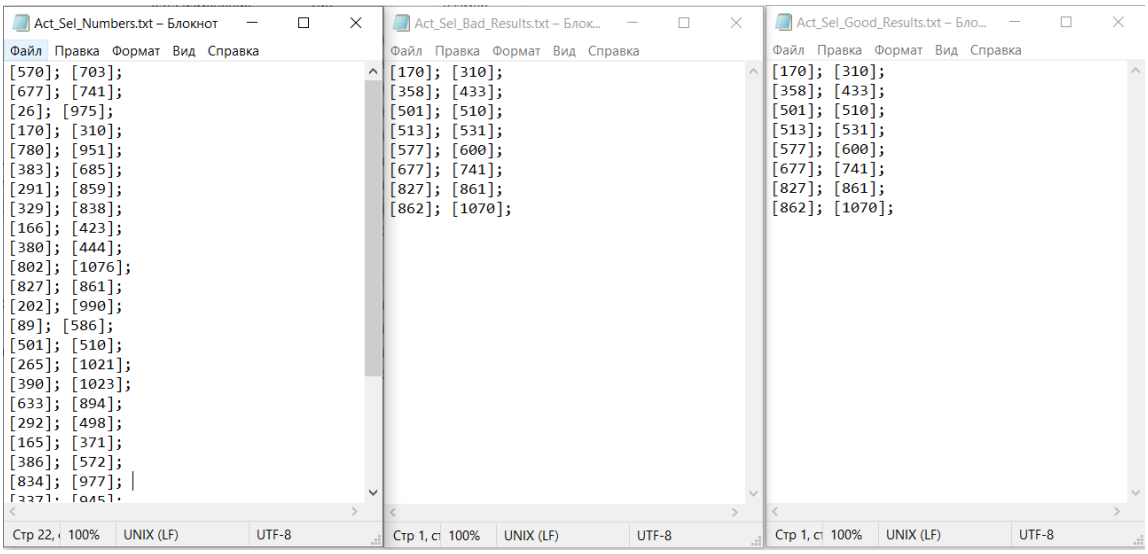

Рисунок 7. Полученные результаты выполнения идентичны.

```
Bad variant of ActSel took 0,18
Good variant of ActSel took 0,1
```

Рисунок 8. Сравнение времени выполнения программ, в мс.

3. На основе псевдокода создана программа MaxIndependentSet. Данная программа находит максимальное количество не соединённых (Родитель и ребёнок) вершин. Время работы программы = $O(T)$.

```java
public static void max_independent_control() {  1 usage
    List<Map<Node, List<Node>>> tree = Tree.filling_tree( levels: 5);
    Tree.printing_tree(tree);
    Set<Node> maxIndependentSet = max_Independent_Set(tree);
    System.out.println("\nMax Independent Set:");
    for (Node node : maxIndependentSet) {
        System.out.println("Узел: " + node.value);
    }
}

public static Set<Node> max_Independent_Set(List<Map<Node, List<Node>>> wholeTree) {  1 usage
    Set<Node> result = new HashSet<>();
    Set<Node> visited = new HashSet<>();
    for (Map<Node, List<Node>> level : wholeTree) {
        for (Map.Entry<Node, List<Node>> entry : level.entrySet()) {
            Node currentNode = entry.getKey();
            if (!visited.contains(currentNode) && isIndependent(currentNode, visited)) {
                result.add(currentNode);
            }
        }
    }
    return result;
}

private static boolean isIndependent(Node node, Set<Node> visited) {  1 usage
    if (node == null || visited.contains(node)) {
        return false;
    }
    for (Node child : node.children) {
        if (visited.contains(child) || visited.contains(node) || visited.containsAll(child.children)) {
            return false;
```

Рисунок 9. Алгоритм MaxIndependentSet.

```
Уровень 1:
Узел: 25
Узел: 252
Узел: 265
Узел: 431
Узел: 458

Уровень 2:
Узел: 298
Узел: 483
Узел: 240

Уровень 3:
Узел: 461
Узел: 102
Узел: 403
Узел: 90

Уровень 4:
Узел: 20
Узел: 352
Узел: 117
Узел: 163
Узел: 52
Узел: 359

Уровень 5:
Узел: 283
Узел: 458
Узел: 137


Max Independent Set:
Узел: 352
Узел: 252
Узел: 137
Узел: 52
Узел: 458
Узел: 403
Узел: 90
Узел: 283
Узел: 298
```

Рисунок 10. Результат программы MaxIndependentSet.

4. На основе псевдокода создана программа KnapSack. Данная программа находит максимальную цену за вещи относительно разрешённого веса. Время работы программы = $O(n * \log(n))$.

```
static void calculating(int[][] items, int capacity){  2 usages
    /* Сортировка двумерного массива цен\веса относительно специального компаратора.
        cost - цена, weight - вес. Сортировка происходит относительно сравнения цены/вес.
     */
    Arrays.sort(items, (cost, weight) -> Integer.compare(weight[1] / weight[2], cost[1] / cost[2]));
    Map<Integer, Integer> price_and_amount = new HashMap<>();
    int max_money = 0;
    int max_weight = 0;
    for (int[] item : items) {
        int remaining_capacity = capacity - max_weight;
        if (remaining_capacity / item[2] >= 1) {
            price_and_amount.put(item[0], item[2]);
            max_weight += item[2];
            max_money += item[1];
        } else {
            price_and_amount.put(item[0], remaining_capacity);
            max_money += (double) item[1] / item[2] * remaining_capacity;
            break;
        }
    }
    for (Map.Entry<Integer, Integer> element : price_and_amount.entrySet()) {
        System.out.println("Количество элемента " + element.getKey() + " = " + element.getValue());
    }
    System.out.println("Потраченные деньги - " + max_money);
}
```

Рисунок 11. Алгоритм KnapSack.

```
Когда в магазине три предмета, как в примере, то:
Количество элемента 1 = 2
Количество элемента 2 = 3
Количество элемента 3 = 2
Потраченные деньги - 42
Рандомное количество предметов, рандомные цены и веса:
Продукт 1 стоит 29 и весит 23
Продукт 2 стоит 96 и весит 30
Продукт 3 стоит 90 и весит 28
Продукт 4 стоит 43 и весит 13
Продукт 5 стоит 80 и весит 4
Продукт 6 стоит 17 и весит 13
Продукт 7 стоит 44 и весит 18
Продукт 8 стоит 55 и весит 11
Продукт 9 стоит 5 и весит 6
Продукт 10 стоит 35 и весит 27
Количество элемента 2 = 21
Количество элемента 5 = 4
Количество элемента 8 = 11
Потраченные деньги - 202
```

Рисунок 12. Полученные результаты выполнения KnapSack.

Вывод: в ходе выполнения практической работы были рассмотрены жадные алгоритмы. Из полученных данных выявлена важность оптимизации кода. Изучены алгоритмы для покрытия всех точек минимальным

количеством отрезков; использования максимального количества непересекающихся отрезков; поиска максимального количества несоединённых друг с другом вершин; поиска максимальной цены за вещи относительно разрешённого веса.