Министерство образования Республики Беларусь
Учреждение образования
«Брестский государственный технический университет»
Кафедра ИИТ

Лабораторная работа №2
По дисциплине «Надежность программного обеспечения»
Тема: ***«Нагрузочное тестирование программного модуля»***

Выполнил:
Студент 3 курса
Группы ПО-11
Головач И.А.
Проверил:
Козик И.Д.

Брест 2025

Цель работы: проведение нагрузочного тестирования и анализ устойчивости программы.

Ход работы:

Задание. Реализуйте нагрузочное тестирование, например, с использованием библиотек (на Python — locust, на Java — JMeter и т.д.). Проверьте, как программа ведет себя при увеличении нагрузки (например, количество запросов, объем данных и т.д.). Определите "узкие места" в программе и предложите оптимизации для повышения надежности.

Вариант задания. Поиск дубликатов в массиве: Реализуйте программу для поиска дубликатов в массиве. Проведите тестирование на массивах разного размера.

Код программы на языке Java:

Main.java:

```
package org.asgardtime;

import java.sql.SQLOutput;
import java.util.ArrayList;
import java.util.Arrays;
import java.util.List;

public class Main {
    public static void main(String[] args) {
        DuplicateFinder finder = new DuplicateFinder();

        List<Integer> data_1000 = new ArrayList<>();
        List<Integer> data_10000 = new ArrayList<>();
        List<Integer> data_500000 = new ArrayList<>();
        List<Integer> data_5000000 = new ArrayList<>();

        for (int i = 0; i < 1_000; i++) {
            data_1000.add(i);
            data_1000.add(1);
            data_1000.add(2);
            data_1000.add(3);
        }

        for (int i = 0; i < 10_000; i++) {
            data_10000.add(i);
            data_10000.add(4);
            data_10000.add(5);
            data_10000.add(6);
        }

        for (int i = 0; i < 500_000; i++) {
            data_500000.add(i);
            data_500000.add(100_000);
            data_500000.add(300_000);
```

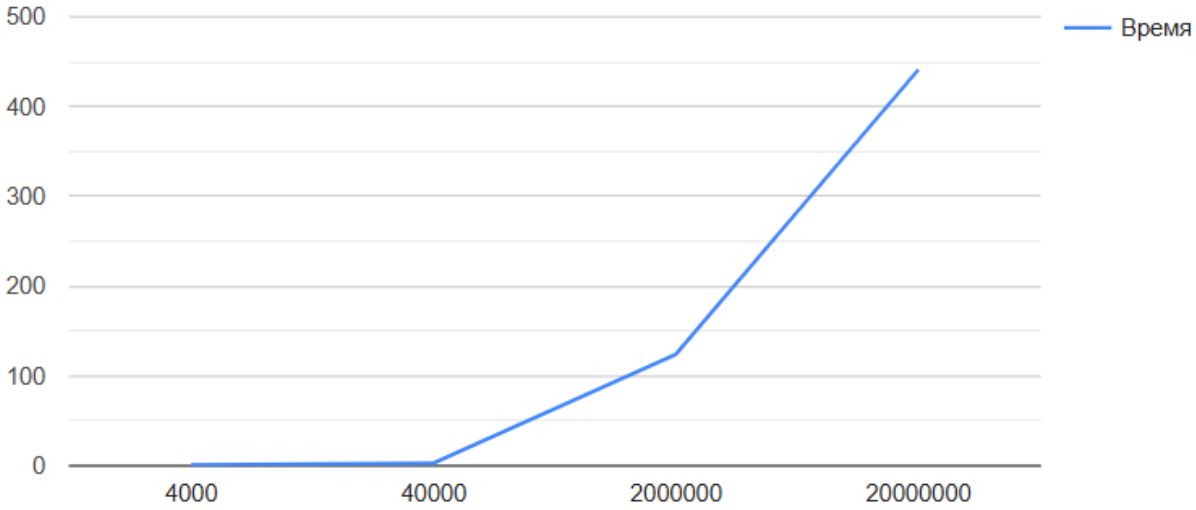
```
            data_500000.add(500_000);
        }

        for (int i = 0; i < 5_000_000; i++) {
            data_5000000.add(i);
            data_5000000.add(1_000_001);
            data_5000000.add(1_000_200);
            data_5000000.add(1_320_200);
        }



        double startTime_1 = System.currentTimeMillis();
        List<Integer> duplicates1 = finder.findDuplicates(data_1000);
        double endTime_1 = System.currentTimeMillis();
        double timeResult_1 = endTime_1 - startTime_1;

        double startTime_2 = System.currentTimeMillis();
        List<Integer> duplicates2 = finder.findDuplicates(data_10000);
        double endTime_2 = System.currentTimeMillis();
        double timeResult_2 = endTime_2 - startTime_2;


        double startTime_3 = System.currentTimeMillis();
        List<Integer> duplicates3 = finder.findDuplicates(data_500000);
        double endTime_3 = System.currentTimeMillis();
        double timeResult_3 = endTime_3 - startTime_3;

        double startTime_4 = System.currentTimeMillis();
        List<Integer> duplicates4 = finder.findDuplicates(data_5000000);
        double endTime_4 = System.currentTimeMillis();
        double timeResult_4 = endTime_4 - startTime_4;


        System.out.println("Размер списка 1: " + data_1000.size());
        System.out.println("Время работы программы: " + timeResult_1);
        System.out.println("Дубликаты: " + duplicates1);

        System.out.println("Размер списка 2: " + data_10000.size());
        System.out.println("Время работы программы: " + timeResult_2);
        System.out.println("Дубликаты: " + duplicates2);

        System.out.println("Размер списка 3: " + data_500000.size());
        System.out.println("Время работы программы: " + timeResult_3);
        System.out.println("Дубликаты: " + duplicates3);

        System.out.println("Размер списка 4: " + data_5000000.size());
        System.out.println("Время работы программы: " + timeResult_4);
        System.out.println("Дубликаты: " + duplicates4);

    }
}
```

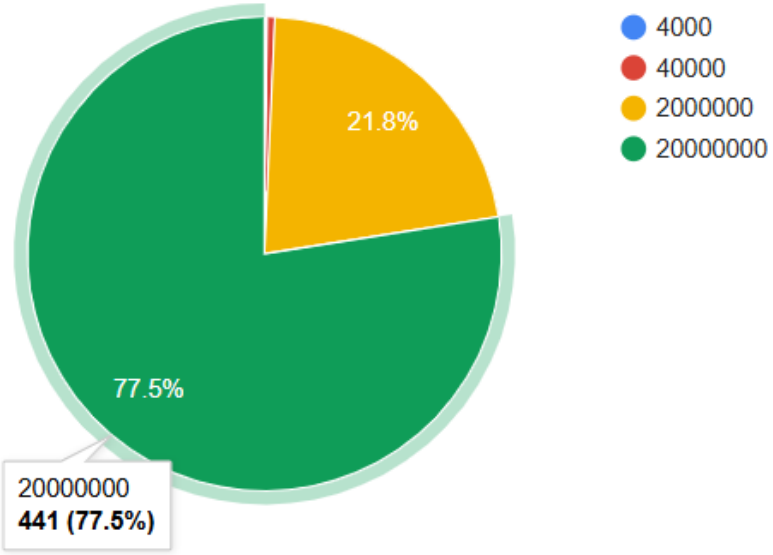
DuplicateFinder.java:

```
package org.asgardtime;

import java.util.ArrayList;
import java.util.HashSet;
import java.util.List;
import java.util.Set;

public class DuplicateFinder {

    /**
     * Метод для поиска дубликатов в массиве.
     *
     * @param data массив чисел
     * @return список дубликатов
     */
    public List<Integer> findDuplicates(List<Integer> data) {
        if (data == null || data.isEmpty()) {
            throw new IllegalArgumentException("Массив не может быть
пустым.");
        }

        Set<Integer> seen = new HashSet<>();
        Set<Integer> duplicates = new HashSet<>();

        for (Integer num : data) {
            if (!seen.add(num)) {
                duplicates.add(num);
            }
        }

        return new ArrayList<>(duplicates);
    }
}
```

Время работы программы в мс:

```
Размер списка 1: 4000
Время работы программы: 1.0
Дубликаты: [1, 2, 3]
Размер списка 2: 40000
Время работы программы: 3.0
Дубликаты: [4, 5, 6]
Размер списка 3: 2000000
Время работы программы: 124.0
Дубликаты: [100000, 300000, 500000]
Размер списка 4: 20000000
Время работы программы: 441.0
Дубликаты: [1000200, 1320200, 1000001]

Process finished with exit code 0
```





Определение "узких мест" и оптимизация:

4.1. Узкие места

1. Большой размер входных данных:
    - При работе с массивами размером 5_000_000 и более, основное время уходит на обработку большого количества элементов.
    - Метод findDuplicates выполняет итерацию по всему списку, что приводит к линейной сложности $O(n)$.
2. Использование HashSet:

- Хотя HashSet обеспечивает быстрое добавление и проверку элементов (в среднем $O(1)$), при больших объемах данных может возникать накладная работа из-за хеширования и возможных коллизий.

3. Частые дубликаты:

    - Если в массиве много дубликатов, они будут многократно добавляться в duplicates. Это может замедлить работу программы, особенно при преобразовании Set в List.

4. Создание множества объектов:

    - Каждый вызов new ArrayList<>(duplicates) создает новый объект, что увеличивает нагрузку на память и сборщик мусора.

4.2. Оптимизация

1. Уменьшение числа операций с памятью:

    - Вместо создания нового ArrayList из duplicates, можно сразу работать с Set. Это позволит избежать лишних операций копирования данных.

2. Использование примитивных типов:

    - Java использует объекты типа Integer для хранения чисел в коллекциях. Это создает накладные расходы на упаковку/распаковку (boxing/unboxing). Для оптимизации можно использовать сторонние библиотеки, такие как Eclipse Collections или Trove , которые поддерживают примитивные типы напрямую.

3. Параллельная обработка:

    - Если данные очень большие, можно разделить массив на части и обрабатывать их параллельно с использованием потоков (ForkJoinPool или parallelStream).

4. Кэширование результатов:

    - Если поиск дубликатов выполняется для одних и тех же данных несколько раз, можно сохранить результаты предыдущих вычислений в кэше.

5. Алгоритмическая оптимизация:

    - Если заранее известно, что диапазон значений ограничен (например, числа от 0 до 1_000_000), можно использовать массив для подсчета частоты встречаемости каждого числа. Это позволит снизить сложность до $O(n)$ без использования хеш-таблиц.

Вывод: Программа показывает хорошую производительность для массивов размером до нескольких миллионов элементов. Однако при работе с очень большими данными (>5_000_000) наблюдаются задержки из-за большого объема данных и использования **HashSet**. Предложенные оптимизации (использование массива для подсчета частот, параллельная обработка и кэширование) позволят значительно ускорить выполнение программы и снизить нагрузку на память.