Лабораторная работа №6

Выполнила работу:
Рапп Ксения Александровна
Группа: 6204-010302D

В данной лабораторной работе была разработана система для многопоточного вычисления интегралов математических функций. Основная цель работы заключалась в создании приложения, где один поток генерирует задания для интегрирования, а второй поток выполняет эти вычисления.

Задание 1: Реализация численного интегрирования методом трапеций

Требовалось разработать метод численного интегрирования в классе Functions, который бы вычислял определенный интеграл функции методом трапеций. Метод должен проверять корректность входных параметров и обрабатывать различные граничные случаи.

Реализовала метод integrate(), который представляет собой классическую реализацию метода трапеций для численного интегрирования. Основной сложностью была правильная обработка ситуации, когда область интегрирования не делится нацело на шаг дискретизации.

```
public static double integrate(Function f, double left, double right, double step) {
        if (left < f.getLeftDomainBorder() || right > f.getRightDomainBorder()) {
            throw new IllegalArgumentException("Границы интегрирования выходят за
область определения функции");
        }
        if (left >= right) {
            throw new IllegalArgumentException("Левая граница должна быть меньше
правой");
        }
        if (step <= 0) {
            throw new IllegalArgumentException("Шаг интегрирования должен быть
положительным");
        }
        double integralValue = 0.0;
        double currentX = left;
        while (currentX < right) {
            double nextX = Math.min(currentX + step, right);
            double y1 = f.getFunctionValue(currentX);
            double y2 = f.getFunctionValue(nextX);
            integralValue += (y1 + y2) * (nextX - currentX) / 2;
            currentX = nextX;
        }
        return integralValue;
    }
```

1. **Граничные проверки:** Метод тщательно проверяет, что интервал интегрирования не выходит за область определения функции, что предотвращает математически некорректные вычисления.
2. **Обработка последнего шага:** Использование Math.min(currentX + step, rightBorder)гарантирует, что последний шаг точно попадет на правую границу интегрирования, даже если оставшееся расстояние меньше шага.

Тестирование и верификация:
Для проверки корректности работы метода было проведено тестирование на функции экспоненты, поскольку для нее известно точное аналитическое решение интеграла на отрезке [0, 1]:

```
Exp exp = new Exp();
double theory = Math.E - 1;  // Точное теоретическое значение
System.out.println("Теоретическое значение интеграла exp(x) от 0 до 1: " + theory);

// Поиск оптимального шага для достижения требуемой точности
double[] stepSizes = {1.0, 0.1, 0.01, 0.001, 0.0001, 0.00001, 0.000001, 0.0000001};
System.out.println("Поиск шага для точности 1e-7:");

for (double s : stepSizes) {
    double value = Functions.integrate(exp, 0, 1, s);
    double diff = Math.abs(value - theory);
    System.out.printf("Шаг: %.8f | Значение: %.10f | Разница: %.10f%n", s, value, diff);

    if (diff < 1e-7) {
        System.out.printf("Требуемая точность достигнута при шаге: %.8f%n", s);
        break;
    }
}
```

Задание 2: Последовательная версия программы

Требовалось создать базовую последовательную версию программы, которая генерирует задания для интегрирования и последовательно их выполняет. Это служило основой для сравнения с многопоточными версиями.

Разработала класс Task, который инкапсулирует все параметры задания для интегрирования, и реализовала метод runSequential(), демонстрирующий работу программы в однопоточном режиме.
Класс Task:

```
public class Task {
    // Поля для хранения параметров задания
    private Function function;      // Интегрируемая функция
    private double leftBorder;      // Левая граница интегрирования
    private double rightBorder;     // Правая граница интегрирования
    private double step;            // Шаг дискретизации
    private int tasksCount;         // Общее количество заданий
    private int currentTask = 0;    // Текущий номер задания

    // Конструктор, принимающий общее количество заданий
    public Task(int tasksCount) {
        this.tasksCount = tasksCount;
    }

    // Синхронизированный метод для установки параметров задания
```

```
    public synchronized void setTask(Function function, double leftBorder, double
rightBorder, double step) {
        this.function = function;
        this.leftBorder = leftBorder;
        this.rightBorder = rightBorder;
        this.step = step;
    }
    public synchronized int getTasksCount() {
        return tasksCount;
    }
    // Проверка, остались ли еще задания для выполнения
    public synchronized boolean hasMoreTasks() {
        return currentTask < tasksCount;
    }

    // Увеличение счетчика выполненных заданий
    public synchronized void incrementTask() {
        currentTask++;
    }

    // Геттеры для получения параметров задания
    public synchronized Function getFunction() {
        return function;
    }

    public synchronized double getLeftBorder() {
        return leftBorder;
    }

    public synchronized double getRightBorder() {
        return rightBorder;
    }

    public synchronized double getStep() {
        return step;
    }

    // Выполнение вычисления интеграла с текущими параметрами
    public double execute() {
        return Functions.integrate(function, leftBorder, rightBorder, step);
    }
}
```

1. **Инкапсуляция данных:** Класс Task скрывает внутреннее представление данных и предоставляет контролируемый интерфейс доступа.
2. **Случайная генерация параметров**
3. **Обработка ошибок:** Реализована обработка исключений, которая предотвращает аварийное завершение программы при возникновении ошибок.

Задание 3: Простая многопоточная версия

Требовалось разработать многопоточную версию программы, где один поток генерирует задания, а второй - выполняет их. Основной вызов заключался в обеспечении корректного взаимодействия потоков без race conditions.

Я создала два класса, реализующих интерфейс Runnable: SimpleGenerator для генерации заданий и SimpleIntegrator для их выполнения. Для синхронизации использовались стандартные механизмы Java.

Класс SimpleGenerator (производитель):

```
public class SimpleGenerator implements Runnable {

    // Ссылка на объект для взаимодействия между потоками
    private Task task;

    // Генератор случайных чисел для создания параметров заданий
    private Random random = new Random();

    //Конструктор
    public SimpleGenerator(Task task) {
        this.task = task;
    }

    @Override
    public void run() {
        // Пока есть задания для генерации
        for (int i = 0; i < task.getTasksCount(); i++) {
            double base, leftBorder, rightBorder, step;

            // Блок синхронизации для безопасного доступа к общему объекту Task
            synchronized (task) {

                // Создание логарифмической функции со случайным основанием от 1 до 10
                base = 1 + random.nextDouble() * 9;
                Log logFunc = new Log(base);

                // Генерация параметров задания согласно
                leftBorder = random.nextDouble() * 100;          // от 0 до 100
                rightBorder = 100 + random.nextDouble() * 100;   // от 100 до 200
                step = random.nextDouble();                      // от 0 до 1

                // Установка параметров в объект задания
                task.setTask(logFunc, leftBorder, rightBorder, step);
            }

            //Сообщение о сгенерированном задании
            System.out.println("Source " + leftBorder + " " + rightBorder + " " + step);

            try {
                Thread.sleep(10);
            } catch (InterruptedException e) {
```

```
            System.out.println("Генератор прерван");
            break;
        }
    }
    System.out.println("Генератор завершил работу");
  }
}
```

Класс SimpleIntegrator (потребитель):

```
public class SimpleIntegrator implements Runnable {

  // Ссылка на объект для взаимодействия между потоками
  private Task task;

  // Конструктор класса SimpleIntegrator
  public SimpleIntegrator(Task task) {
    this.task = task;
  }

  @Override
  public void run() {
    //Цикл выполняется для всех заданий
    for (int i = 0; i < task.getTasksCount(); i++) {
      double leftBorder, rightBorder, step, result;

      // Блок синхронизации для безопасного доступа к общему объекту Task
      synchronized (task) {

        // Проверка (что задание готово для вычисления)
        if (task.getFunction() != null) {

          //Получение параметров задания
          leftBorder = task.getLeftBorder();
          rightBorder = task.getRightBorder();
          step = task.getStep();

          // Вычисление интеграла
          result = task.execute();
        } else {
          continue;
        }
      }

      System.out.println("Result " + leftBorder + " " + rightBorder + " " + step + " " + result);

      try {
        Thread.sleep(10);
      } catch (InterruptedException e) {
        System.out.println("Интегратор прерван");
        break;
      }
    }
    System.out.println("Интегратор завершил работу");   }}
```

Запуск многопоточной системы:

```
public static void runSimpleThreads() {
        // задание 3: многопоточная версия
        System.out.println("Простая многопоточная версия:");

        // общее задание для двух потоков
        Task t = new Task(100);

        // создаем и запускаем потоки
        Thread gen = new Thread(new SimpleGenerator(t));
        Thread integ = new Thread(new SimpleIntegrator(t));

        gen.start();
        integ.start();

        try {
            // ожидаем завершения потоков
            gen.join();
            integ.join();
        } catch (InterruptedException ex) {
            System.out.println("Главный поток прерван");
        }
        System.out.println();
    }
```

Решенные проблемы многопоточного программирования

1. Проблема NullPointerException (ошибка обращения к пустому объекту)

Суть проблемы:
Интегрирующий поток пытался выполнить вычисления, обращаясь к данным, которые еще не были подготовлены генератором. Это происходило потому, что поток-интегратор работал быстрее и пытался взять задание до того, как поток-генератор успевал его создать.
Решение:
Была добавлена защитная проверка - перед выполнением вычислений интегратор проверяет, готовы ли данные для обработки. Если данные отсутствуют, поток пропускает текущую итерацию и переходит к следующей, вместо того чтобы аварийно завершаться с ошибкой.

2. Проблема Race Condition

Суть проблемы:
Оба потока одновременно обращались к общему объекту задания, что приводило к конфликтам доступа. Например, в момент когда интегратор читал данные, генератор мог их изменять, что приводило к получению некорректных или "смешанных" значений.

Решение:
Для защиты общих данных были применены синхронизированные блоки. Это гарантирует, что в каждый момент времени только один поток может работать с объектом задания, исключая возможность одновременного доступа и обеспечивая целостность данных.

3. Проблема несоответствия данных

Суть проблемы:
Интегратор мог получить несогласованные данные - например, левую границу из одного задания, а правую границу и шаг из другого. Это происходило из-за того, что операции чтения не были атомарными и между чтением разных параметров генератор мог успеть обновить задание.
Решение:
Все операции чтения и записи были объединены в блоки. Теперь при чтении задания интегратор получает все параметры сразу из одного согласованного состояния, а при записи генератор устанавливает все параметры задания за одну операцию.

4. Проблема длительных блокировок

Суть проблемы:
Изначально вывод результатов в консоль выполнялся внутри синхронизированных блоков. Поскольку операции ввода-вывода являются медленными, это приводило к длительной блокировке общего ресурса. Поток-генератор мог долго ждать, пока интегратор завершит вывод, что снижало общую производительность системы.
Решение:
Операции вывода в консоль были вынесены за пределы критических секций. Теперь потоки сначала быстро считывают или записывают данные в синхронизированном блоке, а затем выполняют медленные операции ввода-вывода уже без блокировок, что значительно повысило эффективность работы системы.

Задание 4: Усложненная многопоточная версия с семафором

Требовалось разработать усовершенствованную версию с использованием семафора для более тонкого управления взаимодействием потоков, а также реализовать корректную обработку прерывания работы потоков.
Что было сделано:
Был создан класс Semaphore для координации потоков, а также
классы Generator и Integrator, расширяющие Thread.

Класс Semaphore (координация):
```
public class Semaphore {
    // флаг (данные готовы для чтения (интегрирования))
    private boolean dataReady = false;
    // флаг (данные обработаны и можно генерировать новые)
    private boolean dataProcessed = true;
    // метод для начала генерации данных
    // блокирует поток пока предыдущие данные не будут обработаны
    public synchronized void startWrite() throws InterruptedException {
        while (!dataProcessed) {
```

```
            wait();
        }
        // сбрасываем флаги для новой операции
        dataReady = false;
        dataProcessed = false;
    }
    // для завершения генерации данных
    public synchronized void endWrite() {
        dataReady = true;
        notifyAll();
    }
    // метод для начала чтения (интегрирования)
    public synchronized void startRead() throws InterruptedException {
        while (!dataReady) {
            wait();
        }
    }
    // для завершения интегрирования данных
    public synchronized void endRead() {
        dataReady = false;
        dataProcessed = true;
        notifyAll();
    }
}
```

Класс Generator (улучшенная версия производителя):

```
public class Generator extends Thread {
    private Task task;
    private Semaphore semaphore;
    private Random random = new Random();
    public Generator(Task task, Semaphore semaphore) {
        this.task = task;
        this.semaphore = semaphore;
    }
    @Override
    public void run() {
        System.out.println("Генератор запущен");
        try {
            for (int i = 0; i < task.getTasksCount(); i++) {
                // Проверка прерывания текущего потока напрямую
                if (Thread.currentThread().isInterrupted()) {
                    throw new InterruptedException();
                }
                semaphore.startWrite();
                // Проверка прерывания после получения разрешения
                if (Thread.currentThread().isInterrupted()) {
                    throw new InterruptedException();
                }
                // Генерация параметров задания
                double base = 1 + random.nextDouble() * 9;
                Log logFunc = new Log(base);
                double leftBorder = random.nextDouble() * 100;
                double rightBorder = 100 + random.nextDouble() * 100;
```

```
            double step = random.nextDouble();
            // Установка задания
            synchronized (task) {
                task.setTask(logFunc, leftBorder, rightBorder, step);
            }
            System.out.println("Source " + leftBorder + " " + rightBorder + " " + step);
            semaphore.endWrite();
            try {
                Thread.sleep(1);
            } catch (InterruptedException e) {
                throw new InterruptedException();
            }
        }
    } catch (InterruptedException e) {
        System.out.println("Генератор прерван");
        Thread.currentThread().interrupt();
    } finally {
        System.out.println("Генератор завершил работу");}}}
```

Класс Integrator (улучшенная версия потребителя):

```
public class Integrator extends Thread {
    private Task task;
    private Semaphore semaphore;
    public Integrator(Task task, Semaphore semaphore) {
        this.task = task;
        this.semaphore = semaphore;
    }
    @Override
    public void run() {
        System.out.println("Интегратор запущен");
        try {
            for (int i = 0; i < task.getTasksCount(); i++) {
                // Проверка прерывания текущего потока напрямую
                if (Thread.currentThread().isInterrupted()) {
                    throw new InterruptedException();
                }
                // Запрашиваем разрешение на чтение данных
                semaphore.startRead();
                // Проверка прерывания после получения разрешения
                if (Thread.currentThread().isInterrupted()) {
                    throw new InterruptedException();
                }
                double leftBorder, rightBorder, step, result;
                synchronized (task) {
                    leftBorder = task.getLeftBorder();
                    rightBorder = task.getRightBorder();
                    step = task.getStep();
                    result = task.execute();              }
                System.out.println("Result " + leftBorder + " " + rightBorder + " " + step + " " + result);
                semaphore.endRead();
                try {
                    Thread.sleep(1);
                } catch (InterruptedException e) {
                    throw new InterruptedException();}}
```

```
        } catch (InterruptedException e) {
            System.out.println("Интегратор прерван");
            Thread.currentThread().interrupt();
        } finally {
            System.out.println("Интегратор завершил работу");}}}
```

Запуск и управление усложненной версией:

```
public static void runComplexThreads() {
        // задание 4: усложненная версия с семафором
        System.out.println("Усложненная многопоточная версия:");
        // создаем общие объекты
        Task t = new Task(100);
        threads.Semaphore sem = new threads.Semaphore()
        // создаем специализированные потоки
        Generator gen = new Generator(t, sem);
        Integrator integ = new Integrator(t, sem);
        // запускаем потоки
        gen.start();
        integ.start();
        try {
            // ждем 50 мс
            Thread.sleep(50);
            // прерываем потоки
            gen.interrupt();
            integ.interrupt();
            // ожидаем завершения потоков
            gen.join();
            integ.join();
        } catch (InterruptedException ex) {
            System.out.println("Главный поток прерван");
        }
        System.out.println();
    }
```

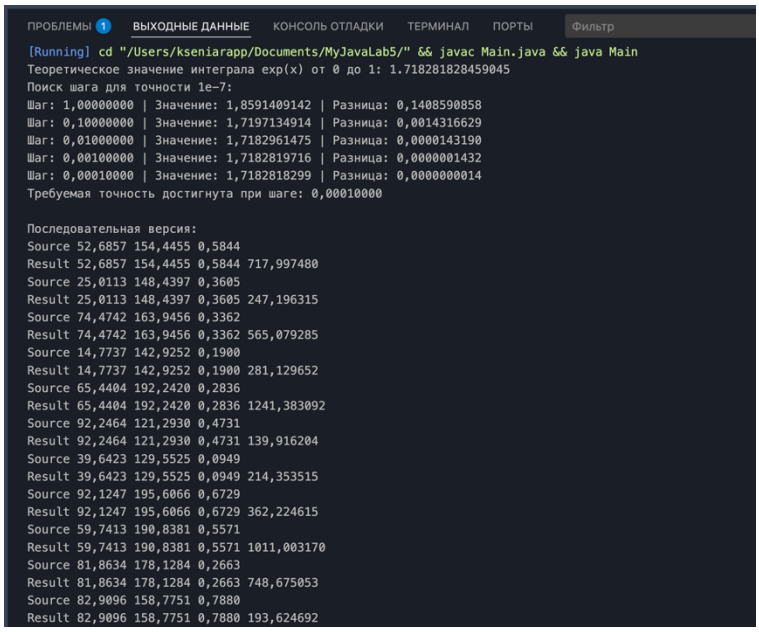
Сравнительный анализ и выводы

Сравнение подходов к синхронизации:

1. **Synchronized блоки (Задание 3):**
    - Простота реализации
    - Высокая производительность для простых сценариев
    - Ограниченная гибкость управления

2. **Семафоры (Задание 4):**
    - Более тонкое управление потоками
    - Явное разделение операций чтения/записи
    - Лучшая масштабируемость

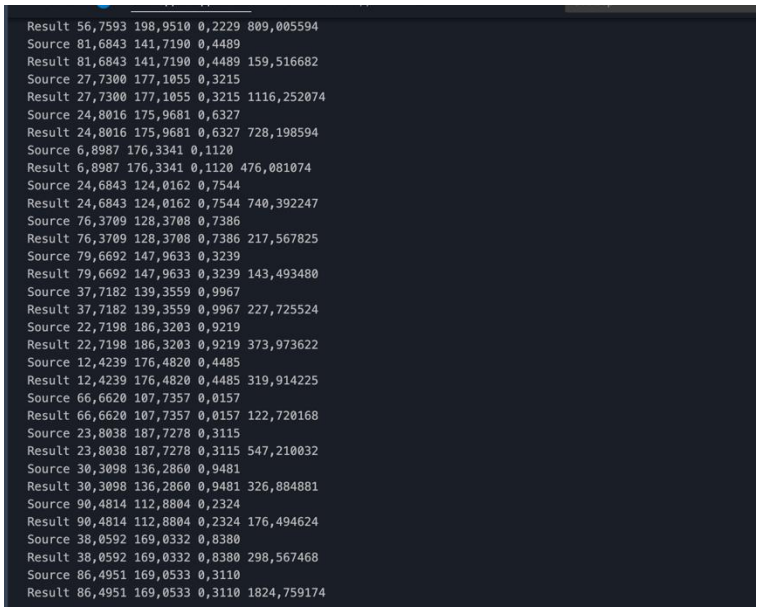
Тестирование в консоли:



```
ПРОБЛЕМЫ 1  ВЫХОДНЫЕ ДАННЫЕ  КОНСОЛЬ ОТЛАДКИ  ТЕРМИНАЛ  ПОРТЫ  Фильтр
[Running] cd "/Users/kseniarapp/Documents/MyJavaLab5/" && javac Main.java && java Main
Теоретическое значение интеграла exp(x) от 0 до 1: 1.718281828459045
Поиск шага для точности 1e-7:
Шаг: 1,00000000 | Значение: 1,8591409142 | Разница: 0,1408590858
Шаг: 0,10000000 | Значение: 1,7197134914 | Разница: 0,0014316629
Шаг: 0,01000000 | Значение: 1,7182961475 | Разница: 0,0000143190
Шаг: 0,00100000 | Значение: 1,7182819716 | Разница: 0,0000001432
Шаг: 0,00010000 | Значение: 1,7182818299 | Разница: 0,0000000014
Требуемая точность достигнута при шаге: 0,00010000

Последовательная версия:
Source 52,6857 154,4455 0,5844
Result 52,6857 154,4455 0,5844 717,997480
Source 25,0113 148,4397 0,3605
Result 25,0113 148,4397 0,3605 247,196315
Source 74,4742 163,9456 0,3362
Result 74,4742 163,9456 0,3362 565,079285
Source 14,7737 142,9252 0,1900
Result 14,7737 142,9252 0,1900 281,129652
Source 65,4404 192,2420 0,2836
Result 65,4404 192,2420 0,2836 1241,383092
Source 92,2464 121,2930 0,4731
Result 92,2464 121,2930 0,4731 139,916204
Source 39,6423 129,5525 0,0949
Result 39,6423 129,5525 0,0949 214,353515
Source 92,1247 195,6066 0,6729
Result 92,1247 195,6066 0,6729 362,224615
Source 59,7413 190,8381 0,5571
Result 59,7413 190,8381 0,5571 1011,003170
Source 81,8634 178,1284 0,2663
Result 81,8634 178,1284 0,2663 748,675053
Source 82,9096 158,7751 0,7880
Result 82,9096 158,7751 0,7880 193,624692
```

...



```
Result 56,7593 198,9510 0,2229 809,005594
Source 81,6843 141,7190 0,4489
Result 81,6843 141,7190 0,4489 159,516682
Source 27,7300 177,1055 0,3215
Result 27,7300 177,1055 0,3215 1116,252074
Source 24,8016 175,9681 0,6327
Result 24,8016 175,9681 0,6327 728,198594
Source 6,8987 176,3341 0,1120
Result 6,8987 176,3341 0,1120 476,081074
Source 24,6843 124,0162 0,7544
Result 24,6843 124,0162 0,7544 740,392247
Source 76,3709 128,3708 0,7386
Result 76,3709 128,3708 0,7386 217,567825
Source 79,6692 147,9633 0,3239
Result 79,6692 147,9633 0,3239 143,493480
Source 37,7182 139,3559 0,9967
Result 37,7182 139,3559 0,9967 227,725524
Source 22,7198 186,3203 0,9219
Result 22,7198 186,3203 0,9219 373,973622
Source 12,4239 176,4820 0,4485
Result 12,4239 176,4820 0,4485 319,914225
Source 66,6620 107,7357 0,0157
Result 66,6620 107,7357 0,0157 122,720168
Source 23,8038 187,7278 0,3115
Result 23,8038 187,7278 0,3115 547,210032
Source 30,3098 136,2860 0,9481
Result 30,3098 136,2860 0,9481 326,884881
Source 90,4814 112,8804 0,2324
Result 90,4814 112,8804 0,2324 176,494624
Source 38,0592 169,0332 0,8380
Result 38,0592 169,0332 0,8380 298,567468
Source 86,4951 169,0533 0,3110
Result 86,4951 169,0533 0,3110 1824,759174
```

```
Простая многопоточная версия:
Source 23.467247152259652 103.77262653388144 0.12390644102340598
Result 23.467247152259652 103.77262653388144 0.12390644102340598 162.56736085115503
Source 71.7178637705408 155.3754382633645 0.7722263200203169
Result 71.7178637705408 155.3754382633645 0.7722263200203169 347.4039528661451
Source 93.24031234517818 174.7980454278906 0.8304611129862085
Result 93.24031234517818 174.7980454278906 0.8304611129862085 212.71385388165854
Result 93.24031234517818 174.7980454278906 0.8304611129862085 212.71385388165854
Source 46.08049577712558 182.87849373036184 0.4112512455397125
Result 46.08049577712558 182.87849373036184 0.4112512455397125 452.9365524422332
Source 86.72749215034173 115.38491562940723 0.5746634355075281
Result 86.72749215034173 115.38491562940723 0.5746634355075281 57.775743572772306
Source 43.31197912446505 190.57815404200107 0.30948314017671685
Result 43.31197912446505 190.57815404200107 0.30948314017671685 504.0752471544224
Source 42.381050139590215 157.8112347396941 0.1466603335968566
Source 82.6728375068226 100.69168432124764 0.8728350787327677
Result 82.6728375068226 100.69168432124764 0.8728350787327677 40.18498884790039
Result 82.6728375068226 100.69168432124764 0.8728350787327677 40.18498884790039
Source 47.45899223788682 123.508507123193 0.24643877573434336
Result 47.45899223788682 123.508507123193 0.24643877573434336 156.65873076090332
Source 60.792495186860215 145.42634850965266 0.7096880161986472
Result 60.792495186860215 145.42634850965266 0.7096880161986472 217.43038212358067
Source 22.39289851902231 163.3751937700296 0.8368579271383868
Result 22.39289851902231 163.3751937700296 0.8368579271383868 273.3780007625051
Source 0.9205421963634275 121.61475915837624 0.6625422524574138
Result 0.9205421963634275 121.61475915837624 0.6625422524574138 936.2718904118966
Source 26.035904273535195 143.62845439593417 0.09317779173667651
Result 26.035904273535195 143.62845439593417 0.09317779173667651 293.13665253100305
Source 95.90969850640721 140.3025981412664 0.3209311513495987
Source 8.743929091372015 107.27238841582229 0.531115793133248
Result 8.743929091372015 107.27238841582229 0.531115793133248 195.44580793857315
Source 34.38358950679648 163.86685550785776 0.22262831293613128
Result 34.38358950679648 163.86685550785776 0.22262831293613128 409.868068822665
```

...

```
Source 7.098668448671Z3 113.27399300110102 0.25715542006011516
Result 7.09866844867123 113.27399300110102 0.25715542006011516 283.44819436224725
Source 32.4415834326194 104.39231521768617 0.5818003803165492
Result 32.4415834326194 104.39231521768617 0.5818003803165492 176.35746669874328
Source 44.863214200097815 169.48800594339014 0.646885524323199
Result 44.863214200097815 169.48800594339014 0.646885524323199 506.26034280948693
Source 39.83333617320744 141.36525956838722 0.6738434553750026
Result 39.83333617320744 141.36525956838722 0.6738434553750026 328.8569944164312
Source 9.509290561939455 164.065359815829 0.1631762166801115
Result 9.509290561939455 164.065359815829 0.1631762166801115 429.36863115585777
Source 47.17869667554019 126.43005288502755 0.795673540016148
Result 47.17869667554019 126.43005288502755 0.795673540016148 156.92496398164852
Source 97.18527028546336 108.15928654885674 0.3086797990787884
Result 97.18527028546336 108.15928654885674 0.3086797990787884 23.454897094367027
Source 56.65183289273864 165.4049375829274 0.006742454791924368
Result 56.65183289273864 165.4049375829274 0.006742454791924368 233.23010947928583
Source 45.95916093793549 150.61797774495872 0.12554441424395124
Result 45.95916093793549 150.61797774495872 0.12554441424395124 584.4356838240788
Source 91.20877795522429 169.66719585019 0.4838429930750534
Result 91.20877795522429 169.66719585019 0.4838429930750534 381.2867155973504
Source 3.792407614466653 128.17351306481663 0.73020899855987
Result 3.792407614466653 128.17351306481663 0.73020899855987 441.39388548007753
Source 99.71028796392852 108.55460415562709 0.01967936209369825
Result 99.71028796392852 108.55460415562709 0.01967936209369825 21.45503521841831
Source 74.425074304921 149.45036099080096 0.1400070267982677
Result 74.425074304921 149.45036099080096 0.1400070267982677 315.39039102358225
Source 37.8076359588989 170.0005257801505 0.9480124087127847
Result 37.8076359588989 170.0005257801505 0.9480124087127847 331.72971480967976
Source 18.12552426509484 192.87158186307042 0.4736107895565719
Result 18.12552426509484 192.87158186307042 0.4736107895565719 360.16244706768293
Source 64.04446259877508 171.98076251246295 0.2024105978294527
Интегратор завершил работу
Генератор завершил работу

Усложненная многопоточная версия:
Интегратор запущен
Генератор запущен
Source 74.76628299626252 189.09275145107773 0.4075935560617667
Result 74.76628299626252 189.09275145107773 0.4075935560617667 320.0427508344569
Source 89.00139513502047 159.1907994374578 0.5838907922688369
Result 89.00139513502047 159.1907994374578 0.5838907922688369 248.66074146098683
Source 14.31478920093453 145.82659180374492 0.6798667305046155
Result 14.31478920093453 145.82659180374492 0.6798667305046155 350.6230154825079
Source 8.576967564659432 103.07819879102746 0.8726604821449921
Result 8.576967564659432 103.07819879102746 0.8726604821449921 178.18981930265707
Source 29.943681727329807 155.96418922648863 0.1829793110498179
Result 29.943681727329807 155.96418922648863 0.1829793110498179 1074.7981472810893
Source 51.34756101479424 114.36269094822899 0.8555060811046481
Result 51.34756101479424 114.36269094822899 0.8555060811046481 143.80623386992508
Source 1.1567352598160552 122.01545437790053 0.7548373478775731
Result 1.1567352598160552 122.01545437790053 0.7548373478775731 213.44693935629277
Source 75.62121210822474 119.85582906648789 0.493413146818487
Result 75.62121210822474 119.85582906648789 0.493413146818487 131.3522181127684
Source 74.68474718063538 178.38780671116092 0.7705951652967186
Result 74.68474718063538 178.38780671116092 0.7705951652967186 1017.5362854309792
Source 84.32198927372025 119.95659454297505 0.32099486114797104
Result 84.32198927372025 119.95659454297505 0.32099486114797104 108.87607385075034
Source 65.76776031156218 108.48945580818156 0.9453488156083838
Result 65.76776031156218 108.48945580818156 0.9453488156083838 87.70071490138213
Source 14.837263534525913 131.64884146409423 0.32868375304076913
Result 14.837263534525913 131.64884146409423 0.32868375304076913 587.5947305390205
Source 22.993175969379863 138.32806616815128 0.7382088048509861
Result 22.993175969379863 138.32806616815128 0.7382088048509861 317.66340398177886
Source 5.8974614502593 123.0331890844756 0.23731835139858215
Result 5.8974614502593 123.0331890844756 0.23731835139858215 439.24443423993887
Source 64.89292099088414 167.9328780694106 0.7522834894963885
Result 64.89292099088414 167.9328780694106 0.7522834894963885 253.7038598574372
```

...

```
Source 51.98370928780503 144.67245366712876 0.4576046989732364
Result 51.98370928780503 144.67245366712876 0.4576046989732364 472.58114513466865
Source 38.89872138675079 195.60146269108057 0.7248655398789887
Result 38.89872138675079 195.60146269108057 0.7248655398789887 795.2278438074007
Source 89.46182520035975 139.0734736703038 0.07879605200631734
Result 89.46182520035975 139.0734736703038 0.07879605200631734 125.63660034303417
Source 5.463251546744418 171.54823511426844 0.27168183341788177
Result 5.463251546744418 171.54823511426844 0.27168183341788177 373.66753282658294
Source 57.649690194918705 100.42313790902465 0.6358050914964237
Result 57.649690194918705 100.42313790902465 0.6358050914964237 96.16106982078425
Source 62.66697795339326 190.0711682084467 0.4323417297356805
Result 62.66697795339326 190.0711682084467 0.4323417297356805 365.4218400142978
Source 61.972423156199696 156.92565534846926 0.2975457112663863
Result 61.972423156199696 156.92565534846926 0.2975457112663863 197.85978876667383
Source 43.55231771160836 140.40932482506798 0.9392050859443908
Result 43.55231771160836 140.40932482506798 0.9392050859443908 191.6036103862029
Source 25.112253903515647 198.86838485704837 0.21779995368793548
Result 25.112253903515647 198.86838485704837 0.21779995368793548 435.112510542051
Source 60.864867366936416 192.66072731951004 0.8844871223676498
Result 60.864867366936416 192.66072731951004 0.8844871223676498 691.0087752625298
Source 43.12167887571916 120.16055805351725 0.5419825970824831
Result 43.12167887571916 120.16055805351725 0.5419825970824831 1661.2120759373508
Source 44.60271786565825 166.68007101631906 0.8446071367724409
Result 44.60271786565825 166.68007101631906 0.8446071367724409 644.3589255016144
Source 87.70969321843032 111.17204857687206 0.560206717097388
Result 87.70969321843032 111.17204857687206 0.560206717097388 86.34345509701058
Source 32.965408233816284 111.06557055842087 0.8793538683624783
Result 32.965408233816284 111.06557055842087 0.8793538683624783 157.0585274701326
Source 50.40426646828112 151.70029522942053 0.2761819396398477
Интегратор прерван
Интегратор завершил работу
Генератор прерван
Генератор завершил работу
```