МИНОБРНАУКИ РОССИИ
САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ
ЭЛЕКТРОТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ
«ЛЭТИ» ИМ. В.И. УЛЬЯНОВА (ЛЕНИНА)
Кафедра МО ЭВМ

ОТЧЕТ

по лабораторной работе №1
по дисциплине «Построение и Анализ Алгоритмов»
Тема: Поиск с возвратом

Студент гр. 3343 _____ Поддубный В.А.

Преподаватель _____ Жангиров Т.Р.

Санкт-Петербург
2025

Цель работы.

Изучение алгоритма поиска с возвратом, реализация с его помощью программы, решающей задачу размещения квадратов на столе.

Задание.

У Вовы много квадратных обрезков доски. Их стороны (размер) изменяются от 1 до $N-1$, и у него есть неограниченное число обрезков любого размера. Но ему очень хочется получить большую столешницу – квадрат размера N. Он может получить ее, собрав из уже имеющихся обрезков (квадратов). Например, столешница размера 7×7 может быть построена из 9 обрезков

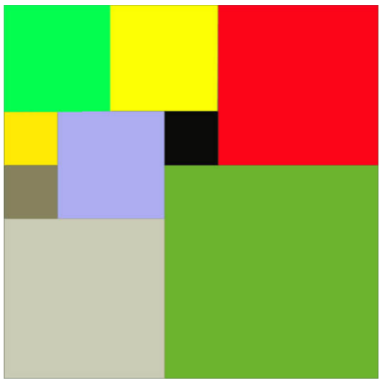

Рисунок 1 – пример размещения квадратов

Внутри столешницы не должно быть пустот, обрезки не должны выходить за пределы столешницы и не должны перекрываться. Кроме того, Вова хочет использовать минимально возможное число обрезков.

Входные данные

Размер столешницы - одно целое число N $(2 \le N \le 20)$.

Выходные данные

Одно число K, задающее минимальное количество обрезков(квадратов), из которых можно построить столешницу(квадрат) заданного размера N. Далее должны идти K строк, каждая из которых должна содержать три целых числа x,y и w, задающие координаты левого

верхнего угла ($1 \leq x,y \leq N$) и длину стороны соответствующего обрезка(квадрата).

Пример входных данных

7

Соответствующие выходные данные

9

1 1 2

1 3 2

3 1 1

4 1 1

3 2 2

5 1 3

4 4 4

1 5 3

3 4 1

Вар. 4р. Рекурсивный бэктрекинг. Расширение задачи на прямоугольные поля, рёбра квадратов меньше рёбер поля. Подсчёт количества вариантов покрытия минимальным числом квадратов.

Основные теоретические положения.

Поиск с возвратом, backtracking — общий метод нахождения решений

задачи, в которой требуется полный перебор всех возможных вариантов в некотором множестве. Решение задачи методом поиска с возвратом сводится к последовательному расширению частичного решения. Если на очередном шаге такое расширение провести не удается, то возвращаются к более короткому частичному решению и продолжают поиск дальше.

Данный алгоритм позволяет найти все решения поставленной задачи, если они существуют.

Выполнение работы.

Описание реализованного алгоритма

Для решения задачи был использован рекурсивный бэктрекинг. Поиск осуществляется перебором вариантов расстановки очередного квадрата. Данный алгоритм основывается на поиске с возвратом (backtracking). Для каждого частичного решения перебираются все возможные расширения, которые затем проверяются на возможность их размещения на поле без перекрытия и выхода за его границы. Если размещение возможно, квадрат добавляется, и поиск продолжается. Если достигнуто полное покрытие, проверяется количество использованных квадратов. В случае нахождения более оптимального решения (меньшего количества квадратов), оно запоминается.

Алгоритм работает с экспоненциальной сложностью, так как количество вариантов размещения квадратов растёт с увеличением размеров поля. Однако применённые оптимизации позволяют сократить количество перебираемых вариантов и ускорить нахождение минимального разбиения.

Описание рекурсивной функции backtrack

Сигнатура: private void backtrack(List<Square> placed, int count)

Назначение: Функция выполняет рекурсивный поиск минимального разбиения поля на квадраты. Она размещает возможные квадраты на свободные области поля, отслеживает текущее количество квадратов и находит оптимальное решение.

Аргументы:

- placed — список квадратов, которые уже размещены на поле.
- count — текущее количество размещённых квадратов.

Возвращаемое значение: Функция не возвращает значения, но обновляет переменные bestSolution (наилучшее найденное разбиение) и minSquares (минимальное количество квадратов в разбиении), если найдено более оптимальное решение.

Алгоритм работы:

1. Проверяет, превышает ли текущее количество квадратов minSquares, и если да, прерывает выполнение.
2. Ищет первую свободную клетку.
3. Если свободных клеток нет, фиксирует текущее разбиение как лучшее, если оно оптимальнее.
4. Определяет максимальный возможный размер нового квадрата.
5. Перебирает возможные квадраты от максимального к минимальному размеру.
6. Если квадрат может быть размещён, добавляет его на поле и рекурсивно вызывает backtrack.
7. После выхода из рекурсии удаляет квадрат и продолжает перебор.

Описание методов и структур данных

Для хранения информации о поле и размещенных квадратах используется класс Field, который содержит следующие данные:

- occupied — матрица булевых значений, где true обозначает занятую клетку, a false — свободную.
- bestSolution — список, содержащий текущее наилучшее разбиение на квадраты.
- minSquares — минимальное найденное количество квадратов.
- filledArea — количество уже заполненных клеток.

Методы класса Field:

- solve() — запускает алгоритм поиска минимального разбиения и выводит лучшее найденное решение.
- backtrack(List<Square> placed, int count) — основной метод рекурсивного поиска с возвратом. Проверяет текущую расстановку и пытается разместить следующий квадрат.
- findFirstEmpty() — находит первую свободную клетку на поле, с которой начинается размещение нового квадрата.
- canPlace(int x, int y, int size) — проверяет, возможно ли разместить квадрат заданного размера в указанной позиции.
- place(int x, int y, int size, boolean state) — устанавливает или убирает квадрат с поля, обновляя соответствующую информацию.

Также используется вспомогательный класс Square, который хранит информацию о координатах и размере квадрата.

Применённые оптимизации

1. **Жадный подход к размеру квадратов.** Сначала размещаются самые большие доступные квадраты, чтобы быстрее достичь конечного решения.
2. **Ограничение на бесперспективные разбиения.** Если текущее количество использованных квадратов уже превышает найденное минимальное, дальнейший перебор прекращается.
3. **Ранний выход.** Как только найдено разбиение с минимальным количеством квадратов, дальнейшие варианты не рассматриваются.
4. **Жадный выбор стартовой позиции.** Размещение всегда начинается с первой свободной клетки, что снижает количество симметричных вариантов.

Оценка сложности алгоритма

формально сложность алгоритма в худшем случае остаётся экспоненциальной — можно записать её как O(2^N), где N — количество

клеток поля. Однако благодаря оптимизациям (жадный выбор стартовой позиции, попытка сначала размещать самые большие квадраты, отсечение неэффективных вариантов) на практике время работы меньше.

Тестирование.

Проверена корректность работы алгоритма бэктрекинга для всех возможных размеров из промежутка 2...5, 15…20.

Ввод	**Вывод**	**Ожидаемый результат**
2	4 1 1 1 1 2 1 2 1 1 2 2 1	Результат верный
3	6 1 1 2 1 3 1 2 3 1 3 1 1 3 2 1 3 3 1	Результат верный
4	4 0 0 2 0 2 2 2 0 2 2 2 2	Результат верный

5	8 1 1 3 1 4 2 3 4 2 4 1 2 4 3 1 5 3 1 5 4 1 5 5 1	Результат верный
15	6 1 1 10 1 11 5 6 11 5 11 1 5 11 6 5 11 11 5	Результат верный
16	4 1 1 8 1 9 8 9 1 8 9 9 8	Результат верный
17	12 1 1 8 1 9 9 9 1 4 9 5 3 9 8 1 10 8 2 10 10 8 12 5 1 12 6 4 13 1 5 16 6 2 16 8 2	Результат верный

18	4 1 1 9 1 10 9 10 1 9 10 10 9	Результат верный
19	13 1 1 13 1 14 6 7 14 6 13 14 2 13 16 4 14 1 6 14 7 6 14 13 1 15 13 3 17 16 1 17 17 3 18 13 2 18 15 2	Результат верный
20	4 1 1 10 1 11 10 11 1 10 11 11 10	Результат верный

Выводы.

Разработанный алгоритм позволяет находить минимальное разбиение квадратного или прямоугольного поля на квадраты. Использование рекурсивного бэктрекинга с оптимизациями позволяет значительно уменьшить время перебора возможных решений.

ПРИЛОЖЕНИЕ А
ИСХОДНЫЙ КОД ПРОГРАММЫ

Название файла: Main.java

```
package com.rect;

import org.apache.logging.log4j.LogManager;
import org.apache.logging.log4j.Logger;

import java.util.Scanner;

public class Main {
    private static final Logger logger = LogManager.getLogger(Main.class);

    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        logger.info("Введите длину и ширину поля через пробел:");
        String[] input = sc.nextLine().split(" ");
        int length = Integer.parseInt(input[0]);
        int width = (input.length > 1) ? Integer.parseInt(input[1]) :
length;
        logger.info("Создано поле размером {}x{}", length, width);

        Field field = new Field(length, width);
        long startTime = System.currentTimeMillis();
        field.solve();
        long endTime = System.currentTimeMillis();
        logger.info("Время выполнения: {} ms", endTime - startTime);
    }
}
```

Название файла: Field.java

```
package com.rect;

import org.apache.logging.log4j.LogManager;
import org.apache.logging.log4j.Logger;

import java.util.ArrayList;
import java.util.List;

public class Field {
    private static final Logger logger = LogManager.getLogger(Field.class);

    private final int length;
    private int filledArea;
    private final int width;
    private final boolean[][] occupied;
    private List<Square> bestSolution = new ArrayList<>();
    private int minSquares = Integer.MAX_VALUE;

    public Field(int length, int width) {
        this.length = length;
        this.width = width;
        this.occupied = new boolean[length][width];
        this.filledArea = 0;
    }

    public void solve() {
```

```
        logger.info("Начинаем поиск минимального количества квадратов...");
        backtrack(new ArrayList<>(), 0);
        logger.info("Минимальное количество квадратов: {}", minSquares);
        logger.info("Лучшее решение:");
        for (Square s : bestSolution) {
            logger.info(s.toString());

        }
        System.out.println(minSquares);
        for (Square s : bestSolution) {
            System.out.println(s.getX() + " " + s.getY() + " " +
s.getLength());
        }
    }

    private void backtrack(List<Square> placed, int count) {
        if (count >= minSquares) {
            logger.debug("Текущий путь не оптимален, возвращаемся.");
            return;
        }

        int[] pos = findFirstEmpty();
        if (pos == null) {
            if (count < minSquares) {
                logger.info("Найдено новое лучшее решение с {}
квадратами.", count);
                minSquares = count;
                bestSolution = new ArrayList<>(placed);
            }
            return;
        }

        int x = pos[0], y = pos[1];
        int maxSize = Math.min(length - x, width - y);
        maxSize = Math.min(maxSize, Math.min(length, width) - 1);
        int remainingArea = length * width - filledArea;

        int maxPossibleSize = maxSize;
        int minRemaining = (int) Math.ceil((double) remainingArea /
(maxPossibleSize * maxPossibleSize));
        if (count + minRemaining >= minSquares) {
            return;
        }

        logger.debug("Попытка разместить квадраты в позиции ({}, {})...", x
+ 1, y + 1);

        for (int size = maxSize; size >= 1; size--) {
            if (canPlace(x, y, size)) {
                logger.debug("Размещаем квадрат размером {}x{} в позиции
({}, {})", size, size, x + 1, y + 1);
                place(x, y, size, true);
                placed.add(new Square(x + 1, y + 1, size));
                backtrack(placed, count + 1);
                placed.remove(placed.size() - 1);
                place(x, y, size, false);
                logger.debug("Убираем квадрат размером {}x{} из позиции
({}, {})", size, size, x + 1, y + 1);
            } else {
                logger.debug("Квадрат размером {}x{} нельзя разместить в
позиции ({}, {})", size, size, x + 1, y + 1);
```

```
            }
        }
    }

    private int[] findFirstEmpty() {
        for (int i = 0; i < length; i++) {
            for (int j = 0; j < width; j++) {
                if (!occupied[i][j]) return new int[]{i, j};
            }
        }
        return null;
    }

    private boolean canPlace(int x, int y, int size) {
        if (x + size > length || y + size > width) return false;
        for (int dx = 0; dx < size; dx++) {
            for (int dy = 0; dy < size; dy++) {
                if (occupied[x + dx][y + dy]) return false;
            }
        }
        return true;
    }

    private void place(int x, int y, int size, boolean state) {
        for (int dx = 0; dx < size; dx++) {
            for (int dy = 0; dy < size; dy++) {
                occupied[x + dx][y + dy] = state;
            }
        }
        filledArea += (state ? size * size : -size * size);
    }
}
```

Название файла: Square.java

```
package com.rect;

public class Square {
    private final int x;
    private final int y;
    private final int length;

    public Square(int x, int y, int length) {
        this.x = x;
        this.y = y;
        this.length = length;
    }

    public int getX() {
        return x;
    }

    public int getY() {
        return y;
    }

    public int getLength() {
        return length;
    }

    @Override
```

```
    public String toString() {
        return "Квадрат: (" + x + ", " + y + "), Размер: " + length + "x" +
length;
    }
}
```