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 _____ Старков С.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

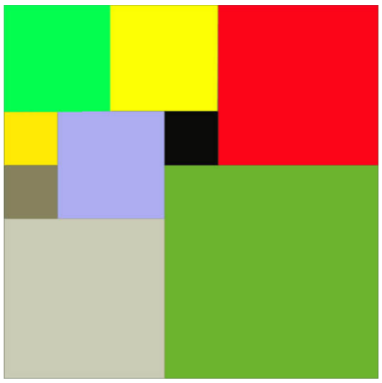

Рисунок 1 – пример размещения квадратов

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

Основу решения составляет метод поиска с возвратом (backtracking), реализованный через рекурсию. На каждом шаге алгоритм последовательно исследует все возможные позиции для размещения очередной плитки, проверяя два условия: плитка не должна выходить за границы поля и не должна пересекаться с уже размещенными. Если условия выполняются, система фиксирует плитку и переходит к поиску позиции для следующей. При обнаружении полного покрытия поля текущее количество плиток сравнивается с ранее найденным минимумом. Если решение оказывается более оптимальным, оно сохраняется как эталонное.

Алгоритм обладает экспоненциальной сложностью, что типично для задач полного перебора. Например, для поля 15×15 анализируются миллионы вариантов. Для сокращения времени работы используются оптимизации, такие как раннее прекращение ветвей перебора (pruning). Например, если текущее количество плиток уже превышает ранее найденный минимум, поиск в этой ветви останавливается.

Описание рекурсивной функции backtrack

Сигнатура: private void backtrack (List<Square> alreadyPlaced, int alreadyUsed, int depthOfRecursion)

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:

- alreadyPlaced — список квадратов, которые уже размещены на поле.
- alreadyUsed — текущее количество размещённых квадратов.

Возвращаемое значение: Функция не возвращает значения, но обновляет переменные bestSolution (наилучшее найденное разбиение) и squaresMin (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 первый свободный квадрат.
3. Если свободных квадратов нет, считает текущее замощение 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 замощен, 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:

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

Методы класса Table:

- solve() — запускает алгоритм поиска минимального замощения и выводит лучшее найденное решение.
- backtrack(List<Square> alreadyPlaced, int alreadyUsed) — основной метод рекурсивного поиска с возвратом. Проверяет текущую расстановку и пытается разместить следующий квадрат.
- findEmpty() — ищет свободную клетку на поле где происходит попытка размещения.
- canPlace(int x, int y, int size) — проверка, можно ли разместить квадрат на указанной позиции.
- place(int x, int y, int size, boolean state) — фиксирует либо убирает квадрат с поля.

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

Сложность Алгоритма при таких оптимизациях остается экспоненциальной — 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 org.piaa;

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

import java.util.List;
import java.util.Scanner;


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

    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);
        logger.info("Enter length:");
        String[] inputValues = scanner.nextLine().split(" ");
        int length = Integer.parseInt(inputValues[0]);
        int width = (inputValues.length > 1) ?
  Integer.parseInt(inputValues[1]) : length;
        logger.info("Created field of size {}", length);

        Table table = new Table(length,width);
        long startTime = System.currentTimeMillis();
        table.solve();
        printBestSolution(table.getBestSolution());
        System.out.println( "  ---------- The best solution! -----------
  \n"+ table.getBestSolution());


        long endTime = System.currentTimeMillis();
        logger.info("Execution time: {} ms", endTime - startTime);
    }
public static void printBestSolution(List<Square> bestSolution) {

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


}
```

Название файла: Table.java

```
package org.piaa;

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

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


public class Table {
    private static final Logger log = LogManager.getLogger(Table.class);
```

```
ArrayList<ArrayList<Square>> placed = new ArrayList<>();
private List<Square> bestSolution = new ArrayList<>();
private final int width;
private final int length;
private int filledSquares;
private boolean engagedPlaces[][];
private int squaresMin = Integer.MAX_VALUE;

public int getSquaresMin() {
    return squaresMin;
}

public List<Square> getBestSolution() {
    return bestSolution;
}

public int getLength() {
    return length;
}


public Table(int length, int width) {
    this.length = length;
    this.width = width;
    this.engagedPlaces = new boolean[length][width];
    this.filledSquares = 0;
}




private boolean placeOppotunity(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 (engagedPlaces[x + dx][y + dy]) {
                return false;
            }
        }
    }
    return true;
}

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

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++) {
            engagedPlaces[x + dx][y + dy] = state;
        }
    }
    filledSquares += (state ? size * size : -size * size);
```

```
  }

  public void solve() {
      log.info("Starting field solution...");
      backtrack(new ArrayList<>(), 0,0);
      log.info("Min square count: {}", squaresMin);

  }


  private void backtrack(List<Square> alreadyPlaced, int alreadyUsed, int
depthOfRecursion) {
      if (alreadyUsed >= squaresMin) {
          log.debug(">>> {}{} The way is not optimal, should go back.",
depthOfRecursion, "-".repeat(depthOfRecursion));
          return;
      }

      int[] pos = findEmpty();
      if (pos == null) {
          if (alreadyUsed < squaresMin) {
              log.info(">>> {}{} Finded new bestway with {} squares.",
depthOfRecursion, "-".repeat(depthOfRecursion), alreadyUsed);
              squaresMin = alreadyUsed;
              bestSolution = new ArrayList<>(alreadyPlaced);
          }
          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 areaThatRemaining = length * width - filledSquares;
      int possibleSizeMax = maxSize;
      int possibleMinSize = (int) Math.ceil((double) areaThatRemaining /
(possibleSizeMax * possibleSizeMax));


      //predict
      if (alreadyUsed + possibleMinSize >= squaresMin) {
          return;
      }

      log.debug(">>> {}{} Attempt to place square on coordinates ({}, {})
*****", depthOfRecursion, "-".repeat(depthOfRecursion), x + 1, y + 1);

      for (int size = maxSize; size >= 1; size--) {
          if (placeOppotunity(x, y, size)) {
              log.debug(">>> {}{} Place a square of size {}x{} at position
+ ({}, {})", depthOfRecursion, "-".repeat(depthOfRecursion), size, size, x
+ 1, y + 1);
              place(x, y, size, true);
              alreadyPlaced.add(new Square(x + 1, y + 1, size));
              backtrack(alreadyPlaced, alreadyUsed + 1, depthOfRecursion +
1);
              alreadyPlaced.remove(alreadyPlaced.size() - 1);
              place(x, y, size, false);
              log.debug(">>> {}{} Remove a square of size {}x{} from
position ({}, {})", depthOfRecursion, "-".repeat(depthOfRecursion), size,
size, x + 1, y + 1);
          } else {
```

```
                log.debug(">>> {}{} A square of size {}x{} cannot be placed
  in position ({}, {})", depthOfRecursion, "-".repeat(depthOfRecursion),
  size, size, x + 1, y + 1);
            }
        }
    }
}
```

Название файла: Square.java

```
  package org.piaa;

  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;
      }

  @Override
      public String toString() {
          return ">>> Coords : (" + x + ", " + y + "), Size: " + length + "x"
  + length + " \n";
      }


      public int getX() {
          return x;
      }

      public int getY() {
          return y;
      }

      public int getLength() {
          return length;
      }
  }
```