МИНОБРНАУКИРОССИИ
САНКТ-ПЕТЕРБУРГСКИЙ
ГОСУДАРСТВЕННЫЙЭЛЕКТРОТЕХНИЧЕСКИЙУН
ИВЕРСИТЕТ
«ЛЭТИ»ИМ.В.И.УЛЬЯНОВА(ЛЕНИНА)
КафедраМОЭВМ

ОТЧЕТ

По лабораторной работе №1

По дисциплине «Построение и анализ алгоритмов»

Тема: Поиск с возвратом

Студентгр.3343 _____ Лихацкий В. Р.

Преподаватель _____ Жангиров Т.Р.

Санкт-Петербург

2025

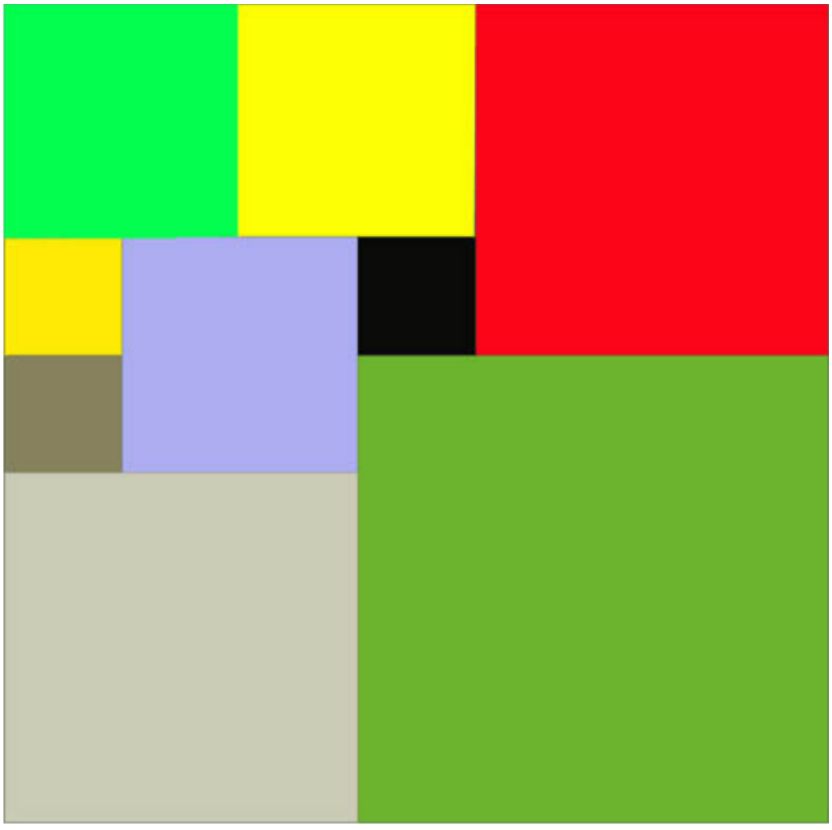
Цельработы.

Решение классической задачи квадрирования квадрата (с заданными относительно размера ограничениями) посредством программы, основанной на алгоритме поискасвозвратом (англ. backtracking).

Задание.

УВовымногоквадратныхобрезковдоски.Ихстороны(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,столешницаразмера7×7 можетбытьпостроенаиз9обрезков.



Внутристолешницынедолжнобытьпустот,обрезкинедолжнывыходитьза

пределыстолешницыинедолжныперекрываться.Крометого,Вовахочетис-пользоватьминимальновозможноечислообрезков.

Входныеданные

Размер столешницы-одно целое число N ($2 \leq N \leq 40$).

Выходныеданные

ОдночислоK,задающееминимальноеколичествообрезков(квадратов),изко-торыхможнопостроитьстолешницу(квадрат)заданногоразмера N.Далееедолжныидти K строк,каждаяизкоторыхдолжнасодержатьтрицелыхчисла x, y и w…, задающие координаты левого верхнего угла ($1 \leq x, y \leq N$) идлинустороны соответствующегообрезка(квадрата).

Примервходныхданных

7

Соответствующиевыходныеданные

9

112

132

311

411

322

513

444

153

341

Вариант 4и. Итеративный бэктрекинг. Расширение задачи на прямоугольные поля, рёбра квадратов меньше рёбер поля. Подсчёт количества вариантов покрытия минимальным числом квадратов.

Описаниефункцийиструктурданных.

Type Square– тип, задающий квадрат. Полями являются целочисленные значения. Поля структуры:

- *X,Y*– координаты левого верхнего угла квадрата.
- *Side* – длина стороны квадрата.

Class Board - основная структура, представляющая таблицу и состояние решения. Поля структуры:

- *width* – ширина заполняемого прямоугольника
- *height* – высота заполняемого прямоугольника.
- *filled* – Битовая карта, аналогичная двумерному массиву состояний 0-пустая ячейка, 1 - полная.
- *best* – массив квадратов, представляющий лучшее найденное решение.
- *variants*– массив лучших решений для задания по варианту.

Методы структуры Table:

1. *Конструктор* создает поле width x height и инициализирует поля класса.
2. *Search* итеративно ищет лучшее решение, используя стек.
3. *Solve() Result:*Основная функция, которая запускает процесс решения задачи. Пытается применить оптимизации, если возможно. Использует backtracking для поиска минимального количества квадратов. Возвращает результат в виде набора квадратов.

Оптимизации:

1. Нахождение GCD для высоты и ширины поля, последующее масштабирование на этот самый GCD.
2. Заполнение начиная с бОльщих квадратов позволяет быстрее найти хорошее решение
3. Отсечение решения если оно не лучше чем best

Описаниепрограммы.

Происходитсчитывание длины и ширины поля. Далее вызывается вышеописанная функция*Solve()*,проверяющая условия оптимизаций.*Solve()* влечет за собой вызов функции *search ()*.

search () –итеративная функция поиска с возвратом (backtracking). Она пушит в стек начальное состояние, затем пока стек не пуст проверяет заполнено ли поле, если заполнено сравнивает текущее решение с лучшим, иначе переходит на первую пустую клетку и пытается заполнить ее квадратом размеров от 1 до maxSide.

Шаги работы функции:

1. **Проверка завершения:**
   - Если текущий столбец превышает ширину таблицы, переходим на уровень ниже
   - Если текущая высота y превышает высоту таблицы, таблица заполнена. Текущее решение становится лучшим
2. **Поиск свободной ячейки:**
   - Если ячейка по текущим x, y не пуста, переходим на следующую.
3. **Проверка целесообразности продолжения:**
   - Если текущее количество квадратов *squares.length* уже превышает лучшее найденное решение *best.length*, дальнейший поиск прекращается. Это позволяет отсечь заведомо неоптимальные ветви.
4. **Определение максимального размера квадрата:**
   - Вычисляется максимальный размер квадрата, который можно разместить в текущей ячейке. Этот размер ограничен:
     - Размером таблицы width - *1* и *height - 1*.
     - Оставшимся местом по ширине и высоте
     - Проверкой что квадрат не будет перекрывать соседние

- Переменная *size* принимает значение максимального допустимого размера.

5. **Перебор возможных размеров квадратов:**
   - В цикле перебираются все возможные размеры квадратов от 1 до size (т.к стек LIFO, в реальности перебираться квадраты будут от большего к меньшему):

   for (let size=1; size <= maxSideSize; size++) {
   - Для каждого размера:
     - Размещаем правую и нижнюю стороны, т.к размер квадрата начинается от 1, в итоге все квадраты вплоть до maxSideSize будут корректными
     - Пушим текущее состояние в стек
6. **Возврат (Backtracking):**
   - Возврат осуществляется stack.pop(), состояние откатывается до предыдущего
7. **Сложность алгоритма** по памяти составляет O(n)+O(m * n) –где O(n) количество строк (используется один number на всю строку), O(m * n) – размер стека в худшем случае.

Сложность алгоритма по операциям зависит от входных данных и в худшем случае составляет O((n + 1)^m * min(m, n)). Однако в реальности алгоритм работает сильно быстрее, т.к хорошие решения находятся довольно быстро

Выводы.

В соответствии с заданным условиям была написана программа, осуществляющая покрытие квадрата меньшими квадратами посредством поиска с возвратом. В ходе изучения поставленной задачи были выявлены и применены оптимизации, обеспечивающие значительное сокращение перебираемых решений.

ПРИЛОЖЕНИЕА

ИСХОДНЫЙКОДПРОГРАММЫ

Файл index.ts

```
import { IO } from "./io";

type Square = { x: number, y: number, side: number }

class Board {
    private height: number;
    private width: number;
    private filled: number[] = [];
    private best: Square[];
    private variants: Square[][] = [];

    constructor(width: number, height: number) {
        this.height = height;
        this.width = width;
        this.filled = new Array(height).fill(0);
        this.best = Array.from({ length: width * height + 1 });
    }

    search() {
        const stack: { x: number, y: number, squares: Square[], filled:
number[] }[] = [
            { x: 0, y: 0, squares: [], filled: [...this.filled] }
        ]

        main: while (stack.length != 0) {
            let { x, y, squares, filled: tmp } = stack.pop()!;
            const filled = [...tmp]

            if (x === this.width) {
                y++;
                x = 0;
            }

            if (y === this.height) {
                if(squares.length === this.best.length) {
                    this.variants.push(squares)
                } else {
                    this.variants = [squares]
```

```
                    this.best = squares;
                }
                continue main;
            }

            if ((filled[y] >> x) & 1) {
                stack.push({ x: x + 1, y, squares, filled })
            } else if (squares.length + 1 <= this.best.length) {
                let maxRows: number = 0;
                let maxCols: number = 0;

                check: for (let checkRow: number = y; checkRow <
this.height; checkRow++) {
                    if ((filled[checkRow] >> x) & 1) break check;
                    maxRows++;
                }

                check: for (let checkCol: number = x; checkCol <
this.width; checkCol++) {
                    if ((filled[y] >> checkCol) & 1) break check;
                    maxCols++;
                }

                let maxSquareSize: number = Math.min(maxRows, maxCols,
this.width - 1, this.height - 1)

                for (let side: number = 1; side <= maxSquareSize;
side++) {
                    for (let offset: number = 0; offset < side;
offset++) {
                        filled[y + side - 1] |= 1 << (x + offset);
                        filled[y + offset] |= 1 << (x + side - 1);
                    }

                    stack.push({ y: y, x: x + side, squares:
[...squares, { x: x, y: y, side }], filled: [...filled]});
                }
            }
        }
    }

    private gcd(a: number, b: number): number {
        a = Math.abs(a);
```

```
        b = Math.abs(b);

        if (a === b) {
            if (a === 0) return 0;

            for (let i = Math.min(a - 1, Math.abs(a)); i > 0; i--) {
                if (a % i === 0) {
                    return i;
                }
            }
            return 1;
        }

        if (b === 0) {
            return a;
        }

        return this.gcd(b, a % b);
    }

    solve() {
        let gcd = this.gcd(this.width, this.height);

        this.height /= gcd;
        this.width /= gcd;

        this.search();

        return this.variants.map(variant => variant.map(({ x, y, side
}) => ({ x: x * gcd, y: y * gcd, side: side * gcd })));
    }
}

(async () => {
    const io = new IO({ input: process.stdin, output: process.stdout })
    const { width, height } = await io.read({ width: "number", height:
"number" })

    console.log(
        new Board(width, height)
        .solve()
```

```
        .at(0)!
        .map(({ x, y, side }) => `${x} ${y} ${side}`)
        .join("\n")
    )
})()
```