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 _____ Гребнев Е.Д

Преподаватель _____ Жангиров Т. Р

Санкт-Петербург

2025

Цель работы.

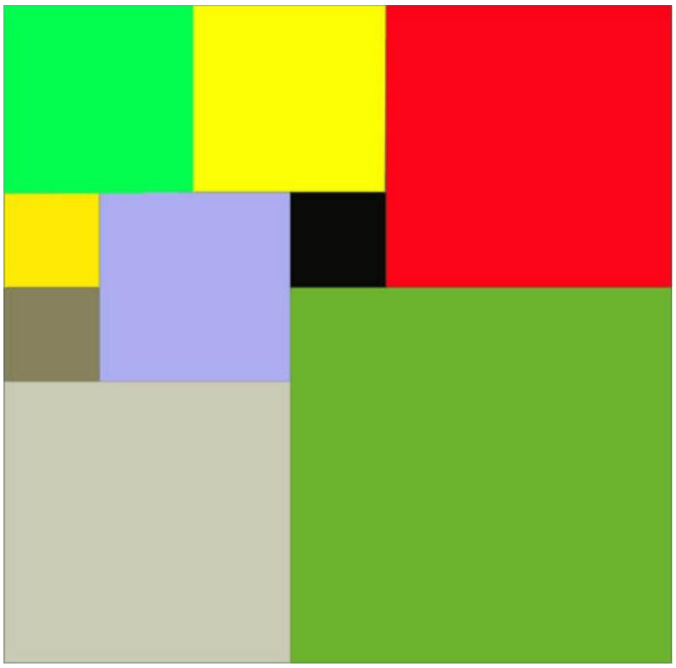
Изучить принцип работы бэктрекинга - алгоритма поиска с возвратом.

Основные теоретические положения.

Поиск с возвратом, бэктрекинг — 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. Для ускорения метода стараются вычисления организовать таким образом, чтобы как можно раньше выявлять заведомо неподходящие варианты. Зачастую это позволяет значительно уменьшить время нахождения решения.

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.



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 ≤ N ≤ 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 ≤ x, y ≤ 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. 3р. Рекурсивный бэктрекинг. Исследование кол-ва операций O(n) от размера квадрата.

Описание рекурсивной функции backtrack().

Функция backtrack() рекурсивно находит оптимальное решение для задачи расстановки квадратов на прямоугольной доске. Она перебирает все возможные варианты размещения квадратов на оставшейся части доски. Для каждого варианта она проверяет, не перекрывается ли он с уже размещенными квадратами на доске, используя функцию isOverlapping(). Если квадрат не перекрывается, он добавляется к списку размещенных квадратов, и функция вызывается рекурсивно с обновленным списком квадратов и доски.

Функция повторяет этот процесс, пока все квадраты не будут размещены на доске или пока не будет найдено лучшее решение. Если найдено решение с меньшим количеством квадратов, чем было найдено ранее, то список лучших квадратов обновляется.

Используются дополнительные параметры для отслеживания количества рекурсивных вызовов и ускорения перебор.

Выполнение работы.

Для решения задачи использован алгоритм рекурсивного бэктрекинга.

Функция *backtrack(squares: Square[], occupiedArea: number, currentCount: number, startX: number, startY: number, gridSize: number, bestCount: { value: number }, bestSolution: Square[], operationCounter: { value: number* })

Если вся поверхность заполнена *occupiedArea = gridSize × gridSizeoccupiedArea = gridSize × gridSize*), проверяется, является ли текущее решение лучше найденного ранее. Для каждой точки *(x,y)* проверяется, можно ли разместить там квадрат. Для каждой точки вычисляется максимальный размер квадрата, который можно разместить без перекрытия с уже размещенными квадратами. Рекурсивно вызывается функция *backtrack* для каждого возможного размера квадрата. Если текущее количество квадратов превышает лучшее найденное решение, ветка отсекается.

Для оптимизации в решении участвуют только квадраты с размером, кратным наибольшему делителю n. Также на столешнице сразу располагаются квадратные доски со стороной *(n+1)/2* в точку *{0, 0}* и два квадрата размером *n/2* в точки *{0, (n+1)/2} и {(n+1)/2, 0}*.

Класс Square используется для хранения информации о каждом квадрате. Он содержит:

- Координаты верхнего левого угла *(x,y)*.
- Длину стороны квадрата *size*.
- Вычисляемые свойства *right* и *bottom*, которые определяют правую и нижнюю границы квадрата.

Функция isOverlapping

Функция проверяет, перекрывает ли точка (x,y) уже размещенные квадраты. Она работает следующим образом:

- Итерируется по массиву квадратов и проверяет, попадает ли точка в границы какого-либо квадрата.
- Возвращает true, если перекрытие обнаружено, и false в противном случае.

4. Функция initializeInitialSquares

Функция инициализирует начальное состояние поверхности, размещая три квадрата:

Один квадрат в левом верхнем углу размером (n+1)/2(*n*+1)/2.

Два квадрата размером n/2*n*/2 в правом верхнем и левом нижнем углах.

5. Функция findMaxSquareSize

Функция находит максимальный размер квадрата, который можно разместить на поверхности, и вычисляет наибольший делитель n*n*.

Исследование сложности используемой памяти.

Пространственная сложность функции *backtrack()* может быть *O(n^2)* в худшем случае, где *n* - размер доски. Однако на практике пространственная сложность может быть меньше благодаря использованию *isOverlapping()*.

Тестирование.

Таблица тестирования:

Номер Теста.	Входные данные	Выходные данные
1	4	4 1 1 2 1 3 2 3 1 2 3 3 2
	7	9 1 1 4 1 5 3 5 1 3 4 5 2 4 7 1 5 4 1 5 7 1 6 4 2 6 6 2

3	9	6 1 1 6 1 7 3 7 1 3 4 7 3 7 4 3 7 7 3

4	13	11 1 1 7 1 8 6 8 1 6 7 8 2 7 10 4 8 7 1 9 7 3 11 10 1 11 11 3 12 7 2 12 9 2

5	20	4 1 1 10 1 11 10 11 1 10 11 11 10
6	23	13 1 1 12 1 13 11 13 1 11 12 13 2 12 15 5

		12 20 4 13 12 1 14 12 3 16 20 1 16 21 3 17 12 7 17 19 2 19 19 5

7	25	1 1 15 1 16 10 16 1 10 11 16 10 16 11 5 21 11 5 21 16 5 21 21 5
8	29	14 1 1 15 1 16 14 16 1 14 15 16 2 15 18 5 15 23 7 16 15 1

		17 15 3 20 15 3 20 18 3 20 21 2 22 21 1 22 22 8 23 15 7

9	31	15 1 1 16 1 17 15 17 1 15 16 17 3 16 20 6 16 26 6 17 16 1 18 16 1 19 16 4 22 20 1 22 21 1 22 22 10 23 16 6 29 16 3 29 19 3
10	37	15

		1 1 19 1 20 18 20 1 18 19 20 2 19 22 5 19 27 11 20 19 1 21 19 3 24 19 8 30 27 3 30 30 8 32 19 6 32 25 1 32 26 1 33 25 5
11	40	1 1 20 1 21 20 21 1 20 21 21 20

Исследование кол-ва операций от размера квадрата.

В данной работе под кол-вом операций подразумевается кол-во вызова функции *backtrack()*.

Был написан код, который в был расположен в файле *perfomance_tests.ts* который итеративно перебрал стороны квадрата от 4 до 60. В итоге получили следующие данные:

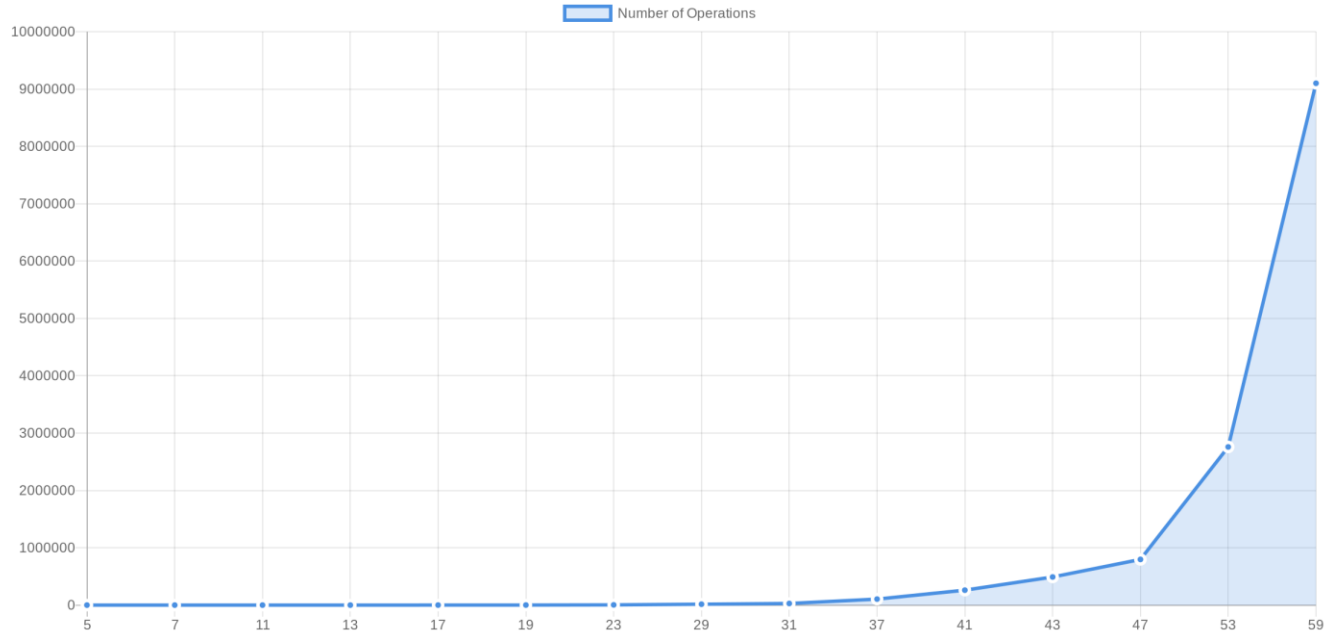
n	Avg Time (ms)	Best Count	Avg Operations
4	0.0936	4	1
5	0.0326	8	6
6	0.0106	4	1
7	0.0711	9	15
8	0.0062	4	1
9	0.0118	6	3
10	0.0056	4	1
11	0.4989	11	76
12	0.0068	4	1
13	0.4745	11	141
14	0.0062	4	1
15	0.0087	6	3
16	0.0057	4	1
17	1.5823	12	553
18	0.0058	4	1
19	0.7294	13	1022
20	0.0044	4	1

21	0.0046	6	3
22	0.0043	4	1
23	1.7234	13	3943
24	0.0037	4	1
25	0.0063	8	6
26	0.0040	4	1
27	0.0033	6	3
28	0.0038	4	1
29	6.6873	14	15921
30	0.0043	4	1
31	16.5269	15	28498
32	0.0052	4	1
33	0.0043	6	3
34	0.0034	4	1
35	0.0066	8	6
36	0.0038	4	1
37	52.5757	15	105015
38	0.0049	4	1
39	0.0044	6	3
40	0.0039	4	1

40	0.0039	4	1
41	129.7486	16	260749
42	0.0052	4	1
43	280.6072	16	490712
44	0.0042	4	1
45	0.0033	6	3
46	0.0030	4	1
47	453.2683	16	796805
48	0.0051	4	1
49	0.0078	9	15
50	0.0040	4	1
51	0.0063	6	3
52	0.0043	4	1
53	1624.6834	17	2758460
54	0.0165	4	1
55	0.0053	8	6
56	0.0044	4	1
57	0.0043	6	3
58	0.0039	4	1
59	5618.9923	17	9100830
60	0.0052	4	1

Как можно заметить, что кол-во итераций растет с каждым последующим простым числом, так как эти числа нельзя разложить, то есть нельзя упростить задачу для вычисления алгоритмом backtracking.

График зафисимости количества операции от простого числа



Анализ временной сложности алгоритма Backtracking

Этот алгоритм использует поиск с возвратом (backtracking) для разбиения квадратного поля (gridSize × gridSize) на минимальное количество квадратов.

Анализ временной сложности
1. Количество возможных размещений квадратов:
- В худшем случае каждый квадрат может начинаться в любой клетке сетки gridSize × gridSize.
- Следовательно, начальных позиций порядка $O(N^2)$, где N = gridSize.

2. Размер квадратов:
- В худшем случае размер квадрата может быть от 1 до gridSize, что даёт O(N) возможных размеров.

3. Рекурсивная глубина:
- Максимальное количество квадратов, которыми можно покрыть поле, соответствует разбиению на наименьшие возможные квадраты (например, разбиение N × N на квадраты 1 × 1).
- Это даёт потенциально экспоненциальную глубину рекурсии: $O(N^2)$.

4. Фильтрация путей:
- Используются эвристики для обрезки невыгодных веток (например, minSquaresNeeded, проверка bestCount).

- Это снижает среднее время работы, но в худшем случае ветки всё равно исследуются.

Итоговая сложность

Если рассматривать худший случай, то алгоритм может генерировать экспоненциальное количество решений, что даёт сложность O((N²)^N) в наивном случае. Однако, благодаря отсечению неэффективных путей (эвристикам), реальная сложность намного меньше, но всё равно экспоненциальная в худшем случае.

Итог:
- В среднем случае: O(2^N) (благодаря отсечениям).
- В худшем случае: O((N²)^N) (если эвристики не помогут).

Выводы.

В результате работы была написана программа, решающая поставленную задачу при помощи рекурсивного бэктрекинга. По результатам исследования зависимость времени работы алгоритма от размера квадрата экспоненциальная.

ПРИЛОЖЕНИЕ А

ИСХОДНЫЙКОД ПРОГРАММЫ

файл *main.ts*:

```
import * as readline from 'readline';
import { Square, backtrack, initializeInitialSquares,
findMaxSquareSize } from './alghoritm';
import { visualizeGrid } from './visualization';

async function main() {
    const rl = readline.createInterface({
        input: process.stdin,
        output: process.stdout,
    });

    const operationCounter = { value: 0 };
    const gridSize: number = await new Promise((resolve) => {
        rl.question("Enter grid size: ", (answer: string) => {
            resolve(parseInt(answer, 10));
        });
    });

    const squareSize = { value: 0 };
    const newGridSize = findMaxSquareSize(gridSize, squareSize);
    const bestCount = { value: 2 * newGridSize + 1 };
    let squares = initializeInitialSquares(newGridSize);
    const bestSolution: Square[] = [];
    const initialOccupiedArea = squares[0].size ** 2 + 2 *
squares[1].size ** 2;
    const startX = squares[2].bottom, startY = squares[2].x;
    backtrack(squares, initialOccupiedArea, 3, startX, startY,
newGridSize, bestCount, bestSolution, operationCounter);

    console.log(`Grid size: ${gridSize}`);
    console.log(`Operation count: ${operationCounter.value}`);
    console.log(`Best count: ${bestCount.value}`);
    for (const square of bestSolution) {
        console.log(`${1 + square.x * squareSize.value} ${1 +
square.y * squareSize.value} ${square.size * squareSize.value}`);
    }
    visualizeGrid(newGridSize, bestSolution);
    rl.close();
}

main();
```

файл algoritm.ts

```
// ANSI color codes
const colors = {
    reset: "\x1b[0m",
    bright: "\x1b[1m",
    dim: "\x1b[2m",
    underscore: "\x1b[4m",
    blink: "\x1b[5m",
    reverse: "\x1b[7m",
    hidden: "\x1b[8m",
    // Foreground (text) colors
    fg: {
        black: "\x1b[30m",
        red: "\x1b[31m",
        green: "\x1b[32m",
        yellow: "\x1b[33m",
        blue: "\x1b[34m",
        magenta: "\x1b[35m",
        cyan: "\x1b[36m",
        white: "\x1b[37m",
    },
    // Background colors
    bg: {
        black: "\x1b[40m",
        red: "\x1b[41m",
        green: "\x1b[42m",
        yellow: "\x1b[43m",
        blue: "\x1b[44m",
        magenta: "\x1b[45m",
        cyan: "\x1b[46m",
        white: "\x1b[47m",
    },
};

export class Square {
    public readonly right: number;
    public readonly bottom: number;

    constructor(
        public readonly x: number,
        public readonly y: number,
        public readonly size: number
    ) {
        this.right = x + size;
        this.bottom = y + size;
        console.log(`${colors.fg.cyan}[Square]${colors.reset} Создан
квадрат: x=${x}, y=${y}, size=${size}`);
    }
}
```

```
export function isOverlapping(squares: Square[], x: number, y:
number): boolean {
    for (const square of squares) {
        if (x >= square.x && x < square.right && y >= square.y && y <
square.bottom) {
            console.log(`${colors.fg.yellow}[Overlap]${colors.reset}
Точка (${x}, ${y}) пересекается с квадратом (x=${square.x},
y=${square.y}, size=${square.size})`);
            return true;
        }
    }
    return false;
}

export function backtrack(
    squares: Square[],
    occupiedArea: number,
    currentCount: number,
    startX: number,
    startY: number,
    gridSize: number,
    bestCount: { value: number },
    bestSolution: Square[],
    operationCounter: { value: number }
) {
    operationCounter.value++;
    console.log(`\n${colors.fg.blue}[Step
${operationCounter.value}]${colors.reset} Занято ${occupiedArea} из
${gridSize * gridSize}, текущий счетчик квадратов: ${currentCount}`);

    if (occupiedArea === gridSize * gridSize) {
        if (currentCount < bestCount.value) {
            bestCount.value = currentCount;
            bestSolution.length = 0;
            bestSolution.push(...squares);
            console.log(`${colors.fg.green}[Best
Solution]${colors.reset} Найдено лучшее решение с ${currentCount}
квадратами`);
        }
        return;
    }

    for (let x = startX; x < gridSize; x++) {
        for (let y = startY; y < gridSize; y++) {
            if (isOverlapping(squares, x, y)) continue;

            let maxSize = Math.min(gridSize - x, gridSize - y);

            for (const square of squares) {
                if (square.right > x && square.y > y) {
                    maxSize = Math.min(maxSize, square.y - y);
```

```
                } else if (square.bottom > y && square.x > x) {
                    maxSize = Math.min(maxSize, square.x - x);
                }
            }

            if (maxSize <= 0) continue;

            for (let size = maxSize; size >= 1; size--) {
                console.log(`${colors.fg.magenta}[Try]${colors.reset}
Пробуем разместить квадрат размером ${size} в (${x}, ${y})`);
                const newSquare = new Square(x, y, size);
                const newOccupiedArea = occupiedArea + size * size;

                const remainingArea = gridSize * gridSize -
newOccupiedArea;
                if (remainingArea > 0) {
                    const maxPossibleSize = Math.min(gridSize - x,
gridSize - y);
                    const minSquaresNeeded = Math.ceil(remainingArea
/ (maxPossibleSize * maxPossibleSize));
                    if (currentCount + 1 + minSquaresNeeded >=
bestCount.value) {
                        console.log(`${colors.fg.red}[Skip]${colors.r
eset} Пропускаем из-за неэффективности: ${currentCount + 1 +
minSquaresNeeded} >= ${bestCount.value}`);
                        continue;
                    }
                }

                squares.push(newSquare);
                if (newOccupiedArea === gridSize * gridSize) {
                    if (currentCount + 1 < bestCount.value) {
                        bestCount.value = currentCount + 1;
                        bestSolution.length = 0;
                        bestSolution.push(...squares);
                        console.log(`${colors.fg.green}[Update]${colo
rs.reset} Обновлен лучший результат: ${currentCount + 1} квадратов`);
                    }
                    squares.pop();
                    continue;
                }

                if (currentCount + 1 < bestCount.value) {
                    backtrack(
                        squares,
                        newOccupiedArea,
                        currentCount + 1,
                        x,
                        y,
                        gridSize,
                        bestCount,
```

```
                        bestSolution,
                        operationCounter
                    );
                }
                squares.pop();
            }
            return;
        }
        startY = 0;
    }
}

export function initializeInitialSquares(gridSize: number): Square[]
{
    const halfSize = Math.floor((gridSize + 1) / 2);
    const smallSize = Math.floor(gridSize / 2);
    return [
        new Square(0, 0, halfSize),
        new Square(0, halfSize, smallSize),
        new Square(halfSize, 0, smallSize)
    ];
}

export function findMaxSquareSize(gridSize: number, squareSize: {
value: number }): number {
    let maxDivisor = 1;
    for (let i = Math.floor(gridSize / 2); i >= 1; i--) {
        if (gridSize % i === 0) {
            maxDivisor = i;
            break;
        }
    }
    squareSize.value = maxDivisor;
    return gridSize / maxDivisor;
}
```