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	________________	Наумкин А.Д
Преподаватель	________________	Жангиров Т. Р

Санкт-Петербург

2025

Цель работы.

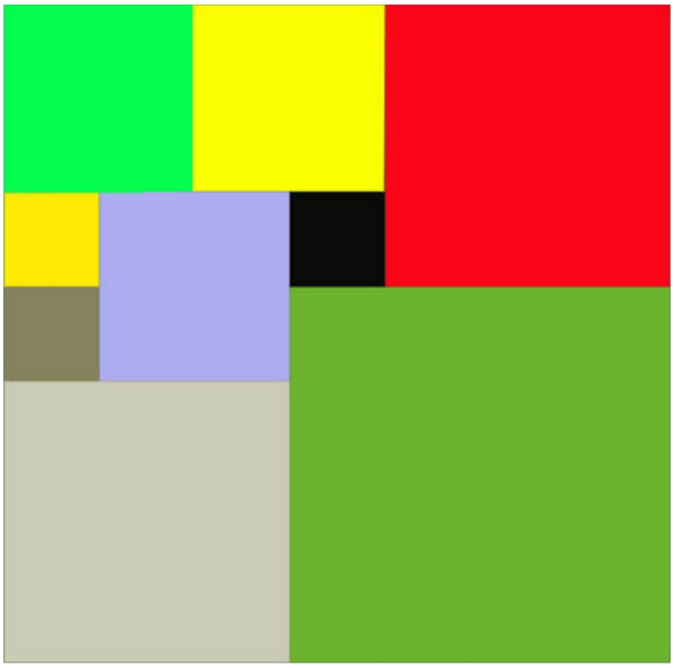
Изучить принцип работы бэктрекинга - алгоритма поиска с возвратом.

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* $(2 \leq N \leq 40)$.

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. 1р. Рекурсивный бэктрекинг. Выполнение на Stepik всех трёх заданий в разделе 2

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 isOverlap (). 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.

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

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

Класс Table используется для заполнения сетки и вывода результата

Содержит:

- Коэффициент соотношения координат для правильного вывода squareSize
- Размер сетки *gridSize;*
- Количество квадратов в наилучшем заполнении *bestCount;*
- Лучшее заполнение *bestSolution;*

Метод backtrack

Если вся поверхность заполнена $occupiedArea = gridSize \times gridSize$

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}*.

Метод isOverlap

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.

Метод findMaxSizeSquare

Функция определяет максимально возможный размер квадрата, который можно разместить в сетке, начиная с заданной точки, при этом не пересекаясь с существующими квадратами

Метод setGridRatio

Функция пытается упростить размер сетки, приводя его к наименьшему возможному целому числу

4. Метод placeSquares

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/2n/2 в правом верхнем и левом нижнем углах.

Запускает бектрекинг.

5. Метод printResult

выводит результат бектрекинга в необходимом формате

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

<table>
<tr><td>4</td><td>13</td><td>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</td></tr>
<tr><td>5</td><td>20</td><td>4
1 1 10
1 11 10
11 1 10
11 11 10</td></tr>
<tr><td>6</td><td>23</td><td>13
1 1 12
1 13 11
13 1 11
12 13 2
12 15 5</td></tr>
</table>

		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

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 $N \times N$ 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 $O((N^2)^N)$ 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: $O((N^2)^N)$ (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*:

```
#include <algorithm>
#include <cmath>
#include <iostream>
#include <vector>

struct Square {
  public:
    const int trailing, bottom;
    int x, y, size;

    Square(int x, int y, int size) : x(x), y(y), size(size),
trailing(x + size), bottom(y + size) {}

    Square(const Square &other)
        : x(other.x), y(other.y), size(other.size),
trailing(other.trailing), bottom(other.bottom) {
    }

    Square &operator=(const Square &other) {
        if (this != &other) {
            x = other.x;
            y = other.y;
            size = other.size;
        }
        return *this;
    }
};

class Table {
  private:
    int squareSize;
    int gridSize;
    int bestCount;
    std::vector<Square> bestSolution;

  public:
    void placeSquares() {
        setGridRatio();
        int startX = gridSize / 2;
        int startY = (gridSize + 1) / 2;
        int occupiedArea = pow(startY, 2) + 2 * pow(startX, 2);
        std::vector<Square> squares = {Square(0, 0, startY),
Square(0, startY, startX),
```

```
                                          Square(startY, 0, startX)};

        backtrack(squares, occupiedArea, 3, startX, startY);
    }

    void printResult() {
        std::cout << bestCount << std::endl;
        for (const auto &square : bestSolution) {
            std::cout << square.x * squareSize + 1 << " " << square.y
* squareSize + 1 << " "
                      << square.size * squareSize << std::endl;
        }
    }

    Table(int gridSize) : gridSize(gridSize), bestCount(gridSize *
gridSize + 1) {}

  private:
    void backtrack(std::vector<Square> currentSquares, int
occupiedArea, int currentCount,
                   int startX, int startY) {
        if (occupiedArea == gridSize * gridSize) {
            if (currentCount < bestCount) {
                bestCount = currentCount;
                bestSolution = currentSquares;
            }
            return;
        }

        for (int x = startX; x < gridSize; ++x) {
            for (int y = startY; y < gridSize; ++y) {
                if (isOverlap(currentSquares, x, y))
                    continue;

                int maxSize = findMaxSizeSquare(currentSquares, x,
y);
                if (maxSize <= 0)
                    continue;

                for (int size = maxSize; size >= 1; --size) {
                    Square newSquare(x, y, size);
                    int newOccupiedArea = occupiedArea + size * size;

                    int remainingArea = gridSize * gridSize -
newOccupiedArea;
                    if (remainingArea > 0) {
                        int maxPossibleSize = std::min(gridSize - x,
gridSize - y);
                        int minSquaresNeeded =
                            (remainingArea + (maxPossibleSize *
maxPossibleSize) - 1) /
```

```
                            (maxPossibleSize * maxPossibleSize);
                        if (currentCount + 1 + minSquaresNeeded >=
bestCount) {
                            continue;
                        }
                    }

                    currentSquares.push_back(newSquare);
                    if (newOccupiedArea == gridSize * gridSize) {
                        if (currentCount + 1 < bestCount) {
                            bestCount = currentCount + 1;
                            bestSolution = currentSquares;
                        }
                        currentSquares.pop_back();
                        continue;
                    }

                    if (currentCount + 1 < bestCount) {
                        backtrack(currentSquares, newOccupiedArea,
currentCount + 1, x, y);
                    }
                    currentSquares.pop_back();
                }
                return;
            }
            startY = 0;
        }
    }

    bool isOverlap(const std::vector<Square> &squares, int x, int y)
{
        for (const auto &square : squares) {
            if (x >= square.x && x < square.trailing && y >= square.y
&& y < square.bottom) {
                return true;
            }
        }
        return false;
    }

    int findMaxSizeSquare(const std::vector<Square> &squares, int x,
int y) {
        int maxSize = std::min(gridSize - x, gridSize - y);
        for (const auto &square : squares) {
            if (square.trailing > x && square.y > y) {
                maxSize = std::min(maxSize, square.y - y);
            } else if (square.bottom > y && square.x > x) {
                maxSize = std::min(maxSize, square.x - x);
            }
        }
        return maxSize;
```

```
    }

    void setGridRatio() {
        int maxDivisor = 1;
        for (int i = gridSize / 2; i >= 1; --i) {
            if (gridSize % i == 0) {
                maxDivisor = i;
                break;
            }
        }
        squareSize = maxDivisor;
        gridSize = gridSize / maxDivisor;
    }
};

int main() {
    int gridSize;
    std::cin >> gridSize;

    Table table(gridSize);
    table.placeSquares();
    table.printResult();

    return 0;
}
```