

**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

\_\_\_\_\_

Пименов П.В.

Преподаватель

\_\_\_\_\_

Жангиров Т.Р.

Санкт-Петербург

2025

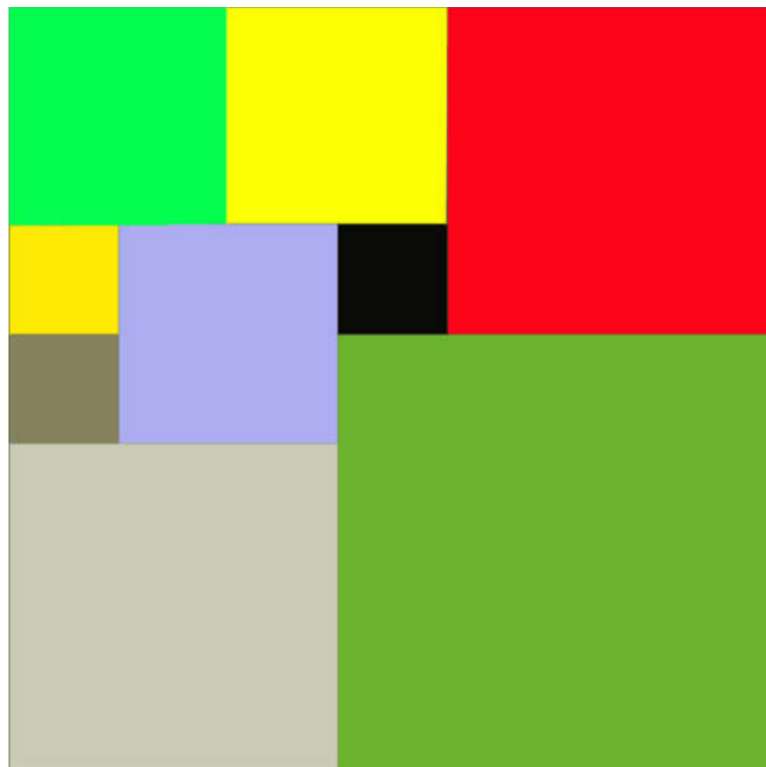
### **Цель работы.**

Решение задачи квадрирования квадрата с помощью алгоритма бэктрекинга («поиск с возвратом»)

### **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  $7 \times 7$  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  $N$  ( $2 \leq N \leq 20$ )

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

### **Выполнение работы.**

Вар. 2р. Рекурсивный бэктрекинг. Исследование времени выполнения от размера квадрата.

Для решения задачи использован алгоритм бэктрекинга.

Описание реализованных функций и классов:

- class Grid – отвечает за работу с полем, на которое ставятся квадраты. Кроме этого, хранит частичные решения.
  - `vector<vector<int>>` grid – поле, на которое ставятся квадраты
  - `vector<vector<int>>` currentPartition – вектор троек x, y, w – текущего разбиения квадрата
  - `vector<vector<int>>` bestPartition – вектор троек x, y, w – лучшего на данный момент разбиения квадрата
  - `int` bestCount – количество квадратов в лучшем разбиении

- `void fill(int x, int y, int size, int value)` – заполняет область квадрат (x, y) – (x+size, y+size) на поле значением value
- `Grid(int n)` – конструктор, инициализирует grid и bestCount
- `int getSize()` – возвращает размер поля (N)
- `int getCurrentCount()` – возвращает количество квадратов в текущем разбиении (размер currentPartition)
- `int getBestCount()` – возвращает bestCount
- `vector<vector<int>>& getBestPartition()` – возвращает bestPartition (для вывода ответа)
- `void place(int x, int y, int size)` – добавляет новый квадрат соответствующего размера по нужным координатам в текущее разбиение и на поле
- `void revert()` – убирает последний поставленный квадрат из текущего разбиения (также стирая его с поля)
- `void saveAsBest()` – сохраняет текущее разбиение как лучшее
- `bool canPlace(int x, int y, int size)` – проверяет, можно ли поставить квадрат такого размера по таким координатам на поле
- `pair<int, int> firstNotFilled()` – возвращает координаты самой первой верхней левой незанятой клетки
- `bool isFilled()` – проверяет, заполнено ли поле
- `void backtrack(Grid& grid)` – рекурсивная функция поиска с возвратом. Принимает в качестве аргумента ссылку на Grid, с которым будет работать на каждом шаге. Сначала проверяется условие выхода (количество квадратов в текущем разбиении больше либо равно количеству квадратов в лучшем разбиении) – в этом случае можно возвращаться из рекурсии, т. к. это решение либо уже лучшее, либо будет хуже. Вычисляются координаты первой незакрашенной клетки. В случае, если поле уже закрашено, то решение сохраняется как лучшее и осуществляется возврат из рекурсии (так как количество квадратов меньше, чем в лучшем разбиении). После всех проверок осуществляется сам бэктрекинг – перебор квадратов всех размеров и попытка их расположить. Берем квадрат нового размера, проверяем можно ли его

поставить, если нет, берем другой квадрат, а если да – ставим, и снова запускаем бэктрекинг, но уже с поставленным квадратом. Если найдется лучшее разбиение в такой конфигурации, то оно сохранится. После выхода из запущенного бэктрекинга, Этот квадрат снимается с поля и берется квадрат следующего размера, цикл повторяется.

- `vector<vector<int>> solve(int n)` – метод, выполняющий решение задачи. Создает Grid, оптимизирует по возможности, запускает бэктрекинг, возвращает лучшее разбиение.

Оптимизации:

- Для четного  $N$  – лучшее разбиение будет состоять всего из 4 квадратов размера  $N/2$ .
- Для простого  $N$  – в разбиении всегда участвуют квадраты размеров  $(N+1)/2$ ,  $(N-1)/2$ ,  $(N-1)/2$
- В остальных случаях (для нечетного составного  $N$ ) – лучшее разбиение всегда будет состоять из 6 квадратов, размер которых определяется как  $1 \times (2 \times N / \text{minfactor})$ ,  $5 \times (N / \text{minfactor})$ , где `minfactor` – минимальный делитель  $N$

Сложность алгоритма:

- По памяти:  $O(n^2)$  на создание поля  $N \times N$  и хранение лучшего и текущего разбиений
- По времени: в лучшем случае (оптимизации 1 и 2) решение получается за  $O(1)$ , в остальных случаях сложность экспоненциальная (даже в случае оптимизации 3, хотя и около 75% поля будет заполнено изначально).

Тестирование:

Входные данные	Выходные данные	Комментарий
7	9 1 1 4 5 1 3	Успех

	1 5 3 4 5 2 4 7 1 5 4 1 5 7 1 6 4 2 6 6 2	
6	4 1 1 3 4 1 3 1 4 3 4 4 3	Успех
9	6 1 1 6 7 1 3 1 7 3 7 4 3 4 7 3 7 7 3	Успех

Исследование:



Как видно на рисунке, время выполнения алгоритма сильно возрастает при увеличении размера квадрата (при больших простых  $N$ ).

### **Выводы.**

Разработан алгоритм для решения задачи квадрирования квадрата с помощью бэктрекинга. Проведено исследование времени выполнения алгоритма от размера квадрата.

## ПРИЛОЖЕНИЕ А

### Файл main.cpp:

```
#INCLUDE <Iostream>
#include <unordered_set>
#include <vector>

#define MIN_N 2
#define MAX_N 20

using namespace std;

class Grid {
private:
    vector<vector<int>>> grid;
    vector<vector<int>>> bestPartition;
    vector<vector<int>>> currentPartition;
    int bestCount;

    void fill(int x, int y, int size, int value) {
        for (int i = x; i < x + size; ++i) {
            for (int j = y; j < y + size; ++j) {
                grid[i][j] = value;
            }
        }
    }

public:
    Grid(int n) {
        grid = vector<vector<int>>>(n, vector<int>(n, 0));
        bestCount = n * n;
    }

    int getSize() { return grid.size(); }

    int getCurrentCount() { return currentPartition.size(); }

    int getBestCount() { return bestCount; }

    vector<vector<int>>>& getBestPartition() { return bestPartition; }

    void place(int x, int y, int size) {
        fill(x, y, size, currentPartition.size() + 1);
        currentPartition.push_back({x, y, size});
    }

    void revert() {
        if (!currentPartition.empty()) {
            auto lastSquare = currentPartition.back();
            fill(lastSquare[0], lastSquare[1], lastSquare[2], 0);
            currentPartition.pop_back();
        }
    }

    void saveAsBest() {
        print();
        cout << endl;

        bestPartition = currentPartition;
        bestCount = currentPartition.size();
    }

    bool canPlace(int x, int y, int size) {
```



```

        IF (X + SIZE > GRID.SIZE() || Y + SIZE > GRID.SIZE()) {
            RETURN FALSE;
        }
        FOR (INT I = X; I < X + SIZE; ++I) {
            FOR (INT J = Y; J < Y + SIZE; ++J) {
                IF (GRID[I][J] != 0) {
                    RETURN FALSE;
                }
            }
        }
        RETURN TRUE;
    }

    PAIR<INT, INT> FIRSTNOTFILLED() {
        FOR (INT I = 0; I < GRID.SIZE(); ++I) {
            FOR (INT J = 0; J < GRID[I].SIZE(); ++J) {
                IF (GRID[I][J] == 0) {
                    RETURN {I, J};
                }
            }
        }
        RETURN {-1, -1};
    }

    BOOL ISFILLED() { RETURN FIRSTNOTFILLED() == MAKE_PAIR(-1, -1); }

    VOID PRINT() {
        FOR (AUTO& ROW : GRID) {
            FOR (INT CELL : ROW) {
                COUT << CELL << " ";
            }
            COUT << ENDL;
        }
    }
};

VOID BACKTRACK(Grid& GRID) {
    IF (GRID.GETCURRENTCOUNT() >= GRID.GETBESTCOUNT()) {
        RETURN;
    }

    INT N = GRID.GETSIZE();

    PAIR<INT, INT> FIRSTNOTFILLED = GRID.FIRSTNOTFILLED();
    INT X = FIRSTNOTFILLED.FIRST, Y = FIRSTNOTFILLED.SECOND;

    // ЕСЛИ СВОБОДНЫХ КЛЕТОК НЕТ - РЕШЕНИЕ НАЙДЕНО
    IF (X == -1) {
        GRID.SAVEASBEST();
        RETURN;
    }

    // ПРОБУЕМ ПОСТАВИТЬ КВАДРАТЫ РАЗНЫХ РАЗМЕРОВ
    FOR (INT SIZE = MIN(MIN(N - X, N - Y), N - 1); SIZE >= 1; --SIZE) {
        IF (GRID.CANPLACE(X, Y, SIZE)) {
            GRID.PLACE(X, Y, SIZE);
            BACKTRACK(GRID);
            GRID.REVERT();
        }
    }
}

VECTOR<VECTOR<INT>> SOLVE(INT N) {

```

```

GRID GRID = GRID(N);
UNORDERED_SET<INT> PRIMES = {3, 5, 7, 11, 13, 17, 19};

// ОПТИМИЗАЦИИ
IF (N % 2 == 0) { // N - четное
    INT SIZE = N / 2;
    GRID.PLACE(0, 0, SIZE);
    GRID.PLACE(SIZE, 0, SIZE);
    GRID.PLACE(0, SIZE, SIZE);
    GRID.PLACE(SIZE, SIZE, SIZE);
    GRID.SAVEASBEST();
} ELSE IF (PRIMES.COUNT(N) == 1) { // N - простое
    INT LSIZE = (N + 1) / 2, SSIZE = (N - 1) / 2;
    GRID.PLACE(0, 0, LSIZE);
    GRID.PLACE(LSIZE, 0, SSIZE);
    GRID.PLACE(0, LSIZE, SSIZE);
} ELSE { // N - нечетное составное
    INT MINFACTOR = N;
    FOR (INT I = 3; I * I <= N; ++I) {
        IF (N % I == 0) {
            MINFACTOR = I;
            BREAK;
        }
    }

    INT SSIZE = N / MINFACTOR, LSIZE = 2 * SSIZE;
    GRID.PLACE(0, 0, LSIZE);
    GRID.PLACE(LSIZE, 0, SSIZE);
    GRID.PLACE(0, LSIZE, SSIZE);
    GRID.PLACE(LSIZE, SSIZE, SSIZE);
    GRID.PLACE(SSIZE, LSIZE, SSIZE);
    GRID.PLACE(LSIZE, LSIZE, SSIZE);
    GRID.SAVEASBEST();
}

IF (GRID.ISFILLED()) {
    RETURN GRID.GETBESTPARTITION();
}

BACKTRACK(GRID);

RETURN GRID.GETBESTPARTITION();
}

INT MAIN() {
    INT N;
    CIN >> N;

    IF (N < MIN_N || N > MAX_N) {
        RETURN 0;
    }

    VECTOR<VECTOR<INT>> SOLUTION = SOLVE(N);

    COUT << SOLUTION.SIZE() << ENDL;
    FOR (AUTO& SQUARE : SOLUTION) {
        COUT << SQUARE[0] + 1 << " " << SQUARE[1] + 1 << " " << SQUARE[2]
            << ENDL;
    }
    RETURN 0;
}

```