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

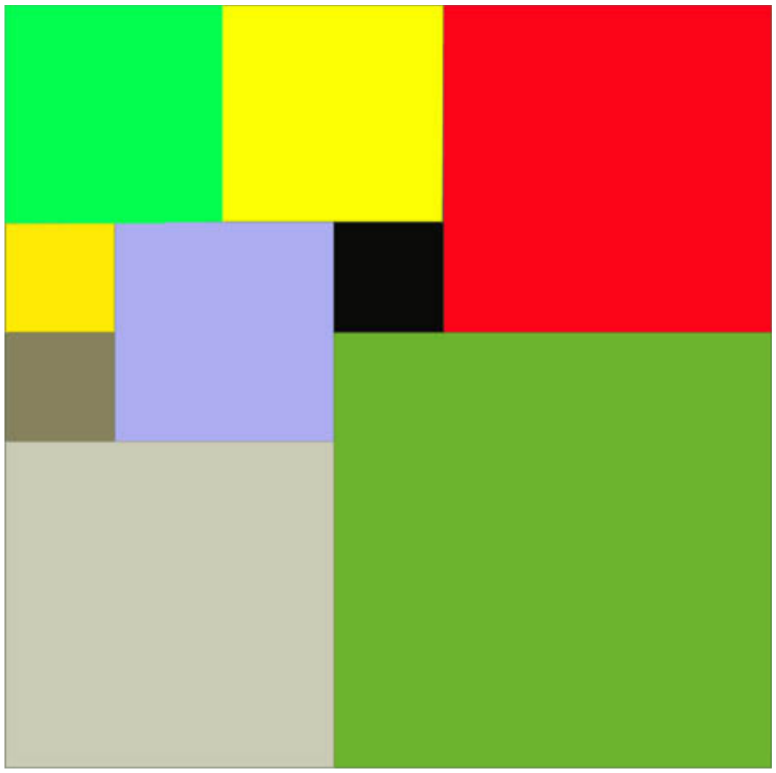
Цель работы.

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 x 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

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) – в разбиении всегда участвуют квадраты размеров lscale * n / mf, sscale * n / mf, sscale * n / mf, где mf – минимальный делитель N, lscale = mf / 2 + 1, a sscale = mf - lscale.

Сложность алгоритма:

- По памяти: O(n^2) на создание поля N x N и хранение лучшего и текущего разбиений
- По времени: в лучшем случае решение получается за O(1), остальных случаях сложность экспоненциальная – O(n^n) (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 40

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) {
        cout << "Place x: " << x + 1 << ", y: " << y + 1 << ", size: " << size
             << endl;
        fill(x, y, size, currentPartition.size() + 1);
        currentPartition.push_back({x, y, size});
    }

    void revert() {
        if (!currentPartition.empty()) {
            auto lastSquare = currentPartition.back();
            int x = lastSquare[0], y = lastSquare[1], size = lastSquare[2];
            cout << "Revert x: " << x + 1 << ", y: " << y + 1
                 << ", size: " << size << endl;
            fill(x, y, size, 0);
            currentPartition.pop_back();
        }
    }

    void saveAsBest() {
        cout << "Save as best" << 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()) {
        cout << "Leave" << endl;
        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();
        cout << "Leave" << endl;
        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();
```

```
            }
        }
    }

    int minFactor(int n) {
        if (n % 2 == 0) {
            return 1;
        }
        for (int i = 3; i * i <= n; ++i) {
            if (n % i == 0) {
                return i;
            }
        }
        return 1;
    }

    vector<vector<int>> solve(int n) {
        Grid grid = Grid(n);

        cout << "Primary fill" << endl;
        // Оптимизации
        if (n % 2 == 0) {  // N - четное
            cout << "N is even" << endl;
            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();
        }
        int mf = minFactor(n);
        if (mf == 1) {  // N - простое
            cout << "N is prime" << endl;
            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
            cout << "N is odd composite" << endl;
            int lscale = mf / 2 + 1, sscale = mf - lscale;
            int lsize = lscale * n / mf, ssize = sscale * n / mf;
            grid.place(0, 0, lsize);
            grid.place(lsize, 0, ssize);
            grid.place(0, lsize, ssize);
        }

        if (grid.isFilled()) {
            return grid.getBestPartition();
        }

        cout << "Backtracking" << endl;
        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;
}
```