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. Вариант 1р.

Студент гр. 3343 _____ Иванов П. Д.

Преподаватель _____ Жангиров Т. Р.

Санкт-Петербург

2025

Цель работы

Разработать алгоритм с использованием метода поиска с возвратом (backtracking) для решения задачи оптимального размещения квадратов на столешнице, обеспечивающего минимальное количество квадратов.

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.

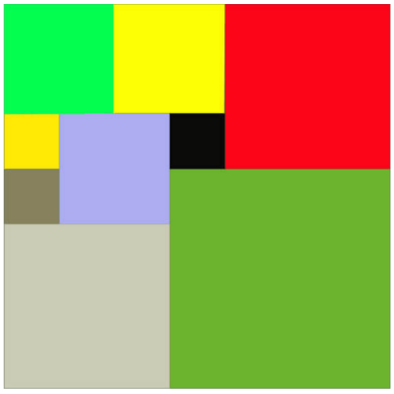

Рис. 1 - Пример

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.

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

Выполнение работы

Для решения задачи разбиения квадрата (N×N) на минимальное число меньших квадратов был использован рекурсивный алгоритм с обратным отслеживанием. Для сокращения пространства поиска применяется эвристика предварительного разбиения и отсечение невыгодных веток.

В ходе выполнения работы были разработаны следующие структуры и функции:

1 - Структура *Tile* - Каждый объект *Tile* описывает один из квадратов, которые будут размещены в решении. Он хранит:

- *posX* и *posY* – координаты верхнего левого угла.
- *sideLength* – длину стороны квадрата.
- *rightEdge* и *bottomEdge* – вычисленные константы, равные соответственно *posX* + *sideLength* и *posY* + *sideLength* - эти поля помогают быстро определить, пересекается ли новый квадрат с уже размещёнными.

2 - Функция *adjustGridRatio*() - Находит наибольший делитель для исходного размера сетки. Если *DEBUG_MODE* включён, выводится информация о найденном делителе, установленном масштабе и нормализованном размере.

3 - Функция *overlapWithTiles*() - Проверяет, находится ли точка (x, y) внутри любой из уже размещённых плиток.

4 - Функция *findMaxTileSize*() - Определяет максимальную длину стороны нового квадрата, который можно разместить в точке (x, y), не выходя за границы нормализованной сетки и не пересекаясь с уже размещёнными плитками.

5 - Функция *fillTiles*() - Запускает процесс разбиения, а именно:

- Нормализация размера - Вызывается *adjustGridRatio*() для вычисления *gUnitSize* и нормализованного размера сетки *gGridDim*.
- Эвристическое начальное разбиение - На основе нормализованного размера выбираются начальные координаты:

initX = gGridDim / 2

initY = (gGridDim + 1) / 2

Затем создаётся три стартовых плитки:

Одна от (0, 0) с размером *initY*.

Вторая от (0, *initY*) с размером *initX*.

Третья от (*initY*, 0) с размером *initX*. Затем вычисляется начальная заполненная площадь.

- Запуск рекурсии - Вызывается функция *backtrackTiles*() с начальными плитками, заполненной площадью, количеством плиток (3) и стартовыми координатами (*initX, initY*).
- Если *DEBUG_MODE* включён, выводится информация о начальных плитках и заполненной площади.

6 - Функция *void backtrackTiles(std::vector<Tile> &currentTiles, int filledArea, int tileCount, int startX, int startY)* - Рекурсивно перебирает варианты размещения плиток (квадратов) для полного покрытия нормализованной сетки с минимальным числом квадратов. Она обновляет глобальные переменные (например, *gMinTileCount* и *gOptimal*) при нахождении лучшего решения.

Аргументы функции:

- *currentTiles (std::vector<Tile>&):* Ссылка на вектор, содержащий текущую конфигурацию размещённых плиток (частичное решение).
- *filledArea (int):* Текущая заполненная площадь, вычисленная как сумма площадей всех плиток из *currentTiles*.
- *tileCount (int):* Количество плиток, уже размещённых в текущем частичном решении.
- *startX, startY (int):* Координаты, с которых начинается поиск следующей свободной ячейки в нормализованной сетке. Это позволяет избежать повторного перебора уже обработанных ячеек и ускорить поиск.

Функция не возвращает значение, но её выполнение изменяет глобальное состояние:

- При нахождении полного покрытия обновляются глобальные переменные *gMinTileCount* (минимальное число плиток) и *gOptimal* (оптимальное разбиение).
- При выполнении рекурсивных вызовов происходит изменение в векторе *currentTiles* (через добавление и удаление плиток).

Способ хранения частичных решений: В алгоритме частичные решения хранятся в виде (вектора) объектов типа *Tile*. Вектор содержит все квадраты, уже размещённые на данном этапе поиска решения. При каждом размещении нового квадрата он добавляется в вектор. Если после этого решение оказывается полным, то текущее частичное решение сравнивается с лучшим найденным решением, и, в случае улучшения, сохраняется в глобальную переменную *gOptimal*. В противном случае производится откат (удаление последнего добавленного квадрата).

Использованные оптимизации алгоритма:

- Нормализация входного размера - Путём нахождения наибольшего делителя входного размера, алгоритм преобразует исходный квадрат в более компактную нормализованную сетку. Это уменьшает размер обрабатываемой области.
- Эвристическое начальное разбиение - На основе нормализованной сетки задаётся стартовое разбиение с тремя плитками. Это покрывает большую часть площади сразу, сокращая число оставшихся ячеек для перебора.
- Отсечение невыгодных веток - Перед рекурсивным вызовом алгоритм оценивает минимальное число плиток, которое потребуется для покрытия оставшейся площади. Если сумма уже использованных плиток и этого минимального числа не может превзойти текущее лучшее решение, ветка перебора отсекается.
- Оптимизация поиска свободной ячейки - При переборе координат для размещения нового квадрата поиск начинается с заданных

стартовых координат (*startX*, *startY*), что помогает ускорить нахождение следующей свободной ячейки.

- Вычисление максимально допустимого размера плитки - Функция *findMaxTileSize*() быстро определяет максимально возможный размер нового квадрата в выбранной точке с учётом уже размещённых плиток и границ нормализованной сетки.

Основной компонент алгоритма (рекурсивный перебор) в общем случае имеет экспоненциальную сложность. При полном переборе вариантов число возможных конфигураций может расти экспоненциально относительно количества ячеек (или нормализованного размера сетки) и в худшем случае время работы можно оценить как $O\left(2^{N^2}\right)$, где N – размер нормализованной сетки.

Глубина рекурсии определяется количеством поставленных плиток. В худшем случае она может достигать $O(N^2)$ (при N^2 маленьких плитках). Дополнительные структуры (например, вектор текущих плиток) используют память пропорционально числу плиток, что тоже оценивается как $O(N^2)$ в худшем случае. Это значит, что память, требуемая для алгоритма – $O(N^2)$, где N – размер нормализованной сетки.

Тестирование

Результаты тестирования представлены в таблице 1.

Табл. 1. – Результаты тестирования

№ п/п	Входные данные	Выходные данные	Комментарии
1.	2	4 1 1 1 1 2 1 2 1 1 2 2 1	Минимальный размер доски. Результат соответствует ожиданиям.
2.	40	4 1 1 20 1 21 20 21 1 20 21 21 20	Максимальный размер доски. Результат соответствует ожиданиям.
3.	39	6 1 1 26 1 27 13 27 1 13 14 27 13 27 14 13 27 27 13	Максимальный нечетный размер доски. Результат соответствует ожиданиям.
4.	19	13 1 1 10 1 11 9 11 1 9 10 11 3 10 14 6 11 10 1 12 10 1 13 10 4 16 14 1 16 15 1 16 16 4 17 10 3 17 13 3	Размер доски представлен простым числом. Результат соответствует ожиданиям.

Выводы

В процессе выполнения лабораторной работы был разработан алгоритм, который заполняет столешницу минимальным количеством квадратов, используя рекурсивный бэктрекинг. Для улучшения эффективности работы алгоритма были разработаны оптимизации.

ПРИЛОЖЕНИЕ А

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

// Структура описывающая плитку
struct Tile {
    const int rightEdge, bottomEdge;
    int posX, posY, sideLength;

    Tile(int x, int y, int len)
            : posX(x), posY(y), sideLength(len), rightEdge(x + len),
bottomEdge(y + len) {}

    Tile(const Tile &other)
            : posX(other.posX), posY(other.posY),
sideLength(other.sideLength),
              rightEdge(other.rightEdge), bottomEdge(other.bottomEdge) {}

    Tile &operator=(const Tile &other) {
        if (this != &other) {
            posX = other.posX;
            posY = other.posY;
            sideLength = other.sideLength;
        }
        return *this;
    }
};


int gGridDim;               // нормированная размерность сетки
int gUnitSize;              // единичный размер плитки
int gMinTileCount;          // минимальное количество плиток
std::vector<Tile> gOptimal; // оптимальное расположение плиток

bool DEBUG_MODE = false;

// Функция нормализации сетки
void adjustGridRatio() {
    int bestDiv = 1;
    for (int d = gGridDim / 2; d >= 1; --d) {
```

```
            if (gGridDim % d == 0) {
                bestDiv = d;
                break;
            }
        }
        gUnitSize = bestDiv;
        gGridDim /= bestDiv;
        if (DEBUG_MODE) {
            std::cout << "[DEBUG] adjustGridRatio: bestDiv = " << bestDiv
                      << ", gUnitSize = " << gUnitSize
                      << ", normalized grid dimension = " << gGridDim <<
std::endl;
        }
    }

    // Проверка накладывается ли новая плитка на уже размещённые
    bool overlapWithTiles(const std::vector<Tile> &tiles, int x, int y) {
        for (const auto &tile : tiles) {
            if (x >= tile.posX && x < tile.rightEdge &&
                y >= tile.posY && y < tile.bottomEdge)
                return true;
        }
        return false;
    }

    // Определение максимально возможной стороны плитки
    int findMaxTileSize(const std::vector<Tile> &tiles, int x, int y) {
        int maxSide = std::min(gGridDim - x, gGridDim - y);
        for (const auto &tile : tiles) {
            if (tile.rightEdge > x && tile.posY > y) {
                maxSide = std::min(maxSide, tile.posY - y);
            } else if (tile.bottomEdge > y && tile.posX > x) {
                maxSide = std::min(maxSide, tile.posX - x);
            }
        }
        return maxSide;
    }

    void backtrackTiles(std::vector<Tile> &currentTiles, int filledArea, int
tileCount, int startX, int startY) {
        if (DEBUG_MODE) {
            std::cout << "[DEBUG] backtrackTiles: tileCount = " << tileCount
                      << ", filledArea = " << filledArea
```

```
                        << ", startX = " << startX << ", startY = " << startY <<
std::endl;
            }
            if (filledArea == gGridDim * gGridDim) {
                if (tileCount < gMinTileCount) {
                    gMinTileCount = tileCount;
                    gOptimal = currentTiles;
                    if (DEBUG_MODE) {
                        std::cout << "[DEBUG] Found complete tiling with tileCount
= " << tileCount << std::endl;
                    }
                }
                return;
            }

            for (int x = startX; x < gGridDim; ++x) {
                for (int y = startY; y < gGridDim; ++y) {
                    if (overlapWithTiles(currentTiles, x, y))
                        continue;

                    int possibleSide = findMaxTileSize(currentTiles, x, y);
                    if (possibleSide <= 0)
                        continue;

                    for (int len = possibleSide; len >= 1; --len) {
                        Tile newTile(x, y, len);
                        int newFilled = filledArea + len * len;

                        int remaining = gGridDim * gGridDim - newFilled;
                        if (remaining > 0) {
                            int maxPossible = std::min(gGridDim - x, gGridDim - y);
                            int   minNeeded   =   (remaining   +   (maxPossible   *
maxPossible) - 1) / (maxPossible * maxPossible);
                            if (tileCount + 1 + minNeeded >= gMinTileCount)
                                continue;
                        }

                        if (DEBUG_MODE) {
                            std::cout << "[DEBUG] Placing tile at (" << x << ", "
<< y
                                      << ") with side = " << len
                                      << ", newFilled = " << newFilled
```

```
                                    << ", tileCount = " << tileCount + 1 <<
std::endl;
                        }

                        currentTiles.push_back(newTile);
                        if (newFilled == gGridDim * gGridDim) {
                            if (tileCount + 1 < gMinTileCount) {
                                gMinTileCount = tileCount + 1;
                                gOptimal = currentTiles;
                                if (DEBUG_MODE) {
                                    std::cout << "[DEBUG] Complete tiling reached
after placing tile at ("
                                              << x << ", " << y << ") with side =
" << len << std::endl;
                                }
                            }
                            currentTiles.pop_back();
                            continue;
                        }

                        if (tileCount + 1 < gMinTileCount)
                            backtrackTiles(currentTiles, newFilled, tileCount + 1,
x, y);
                        if (DEBUG_MODE) {
                            std::cout << "[DEBUG] Removing tile at (" << x << ", "
<< y
                                      << ") with side = " << len << std::endl;
                        }
                        currentTiles.pop_back();
                    }
                    return;
                }
                startY = 0;
            }
        }

        // Функция начальной установки плиток с использованием эвристики
        void fillTiles() {
            adjustGridRatio();
            int initX = gGridDim / 2;
            int initY = (gGridDim + 1) / 2;
            int areaFilled = initY * initY + 2 * initX * initX;
            std::vector<Tile> initTiles = {
```

```
                Tile(0, 0, initY),
                Tile(0, initY, initX),
                Tile(initY, 0, initX)
        };
        if (DEBUG_MODE) {
            std::cout << "[DEBUG] Initial placement:" << std::endl;
            for (const auto &tile : initTiles) {
                std::cout << "[DEBUG] Tile at (" << tile.posX << ", " <<
tile.posY
                          << ") with side = " << tile.sideLength << std::endl;
            }
            std::cout << "[DEBUG] Initial filled area = " << areaFilled <<
std::endl;
        }
        backtrackTiles(initTiles, areaFilled, 3, initX, initY);
    }

    // Вывод оптимального расположения плиток с учетом масштабирования
    void printArrangement() {
        std::cout << gMinTileCount << std::endl;
        for (const auto &tile : gOptimal) {
            std::cout << tile.posX * gUnitSize + 1 << " "
                      << tile.posY * gUnitSize + 1 << " "
                      << tile.sideLength * gUnitSize << std::endl;
        }
    }

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

        gGridDim = inputSize;
        gMinTileCount = inputSize * inputSize + 1;

        fillTiles();
        printArrangement();
        return 0;
    }
```