**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**

|  |  |  |
| --- | --- | --- |
| Студентка гр. 3343 |  | Лобова Е. И. |
| Преподаватель |  | Жангиров Т. Р. |

Санкт-Петербург

2025

## Цель работы

Целью работы является изучение алгоритма поиска с возвратом и практическое его применение на задаче о квадрировании квадрата минимальным количеством.

## 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

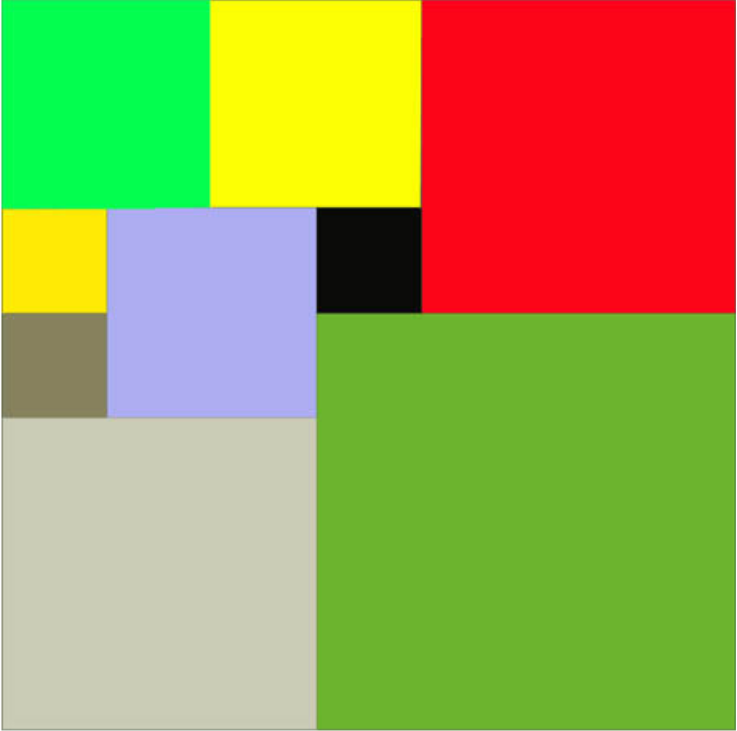


Рисунок 1 – пример размещения квадратов

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 if (currentResult.size() >= minSquares) break;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

Вариант 1и. Итеративный бэктрекинг. Выполнение на Stepik двух заданий в разделе 2

## Выполнение работы

Для решения задания лабораторной работы были разработаны классы Square, Desk, State.

Класс Square используется для хранения координат левого верхнего угла квадрата и его стороны, имеет конструктор и переопределенный оператор вывода в поток.

Класс Desk используется для работы со столешницей и заполнением ее досками. Имеет поля:

int N - длина столешницы.

int num\_occupied - количество занятых единичных квадратов.

bool\*\* occupied - двумерная сетка для представления столешницы.

Методы класса:

* Конструктор Desk(int n): Инициализирует объект класса Desk с размером n, создавая двумерный массив occupied для хранения состояния ячеек (занята или свободна). Все ячейки изначально устанавливаются как свободные (false).
* Деструктор ~Desk(): Освобождает память, выделенную под двумерный массив occupied, чтобы избежать утечек памяти.
* void placeSquare(const Square& square): Устанавливает квадрат на доске, обновляя состояние ячеек в массиве occupied на true (занята).
* void removeSquare(const Square& square): Убирает квадрат с доски, обновляя состояние ячеек в массиве occupied на false (свободна).
* bool canPlace(int x, int y, int size) const: Проверяет, можно ли разместить квадрат размером size с верхним левым углом в координатах (x, y). Возвращает true, если квадрат можно разместить, иначе false.
* bool isFull() const: Проверяет, заполнена ли доска (т.е. все ячейки заняты). Возвращает true, если количество занятых ячеек равно N \* N, иначе false.
* Square findEmptySquare(): Находит первую свободную ячейку на доске и определяет максимальный размер квадрата, который можно разместить с этой ячейкой в качестве верхнего левого угла. Возвращает объект Square, представляющий координаты и размер максимального пустого квадрата.
* friend std::ostream& operator<<(std::ostream& os, const Desk& desk): Реализует вывод состояния доски в поток os. Каждая ячейка выводится как 1 (занята) или 0 (свободна).

Класс State используется для хранения частичных решений, а также лучшего решения и минимального количества квадратов.

Методы класса:

* Конструктор State(int N): Инициализирует объект класса State с размером N, создавая объект Desk для управления состоянием доски. Устанавливает начальные значения для minSquares, currentResult и bestResult.
* std::vector<Square> find\_solution(): Основной метод для поиска оптимального размещения квадратов на доске.
* void addSquare(): Метод добавляющий квадрат на доску и в вектор текущих решений, а также выводящий информацию о производимом действии.
* void removeSquare(): Метод удаляющий квадрат с доски и из вектора текущих решений, а также выводящий информацию о производимом действии.
* friend std::ostream& operator<<(std::ostream& os, const State& state): Реализует вывод информации о текущем состоянии объекта State в поток os, включая минимальное количество квадратов, текущие рассматриваемые квадраты.

Используемые оптимизации:

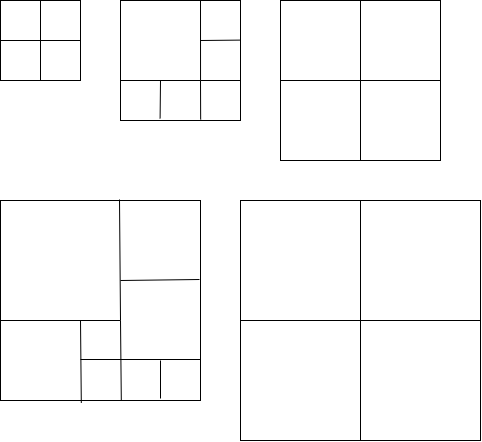
1. Рассмотрим несколько начальных размеров квадратов и минимальных расстановок для них.

Рисунок 2 - Варианты размещения минимального количества квадратов для сторон 2-6

Отсюда наглядно видно, что для сторон, длина которых кратна 2, минимальное разбиение - это 4 одинаковых квадрата, а для других простых чисел - обязательно три квадрата, где один квадрат со стороной (n+1)/2, а другие 2 квадрата с n/2, при этом под эти же формулы попадают и “чётные” квадраты. Также можно заметить, что если сторона квадрата - составное число, то можно решить задачу разбиения квадрата для делителя числа, а затем просто отмасштабировать квадрат до изначального размера.

1. Вторая оптимизация заключается в том, что мы не просто пытаемся бездумно разместить квадраты размером от N-1 до 1, а так как мы начинаем размещение с левого верхнего угла, то с помощью самой верхней левой свободной клетки можем понять, что размер вставляемого квадрата не может превышать min(N-x, N-y). Поэтому начинаем перебор вниз с этого числа.
2. Третья оптимизация состоит в том, что если у нас на каком-то шаге количество квадратов в текущей расстановке больше или равно существующего наилучшего, то дальше эту расстановку мы не рассматриваем, так как это просто не имеет смысла.

Оценка сложности полученного алгоритма:

* По памяти: O(n^2) для хранения сетки столешницы.
* По времени:

Отдельные операции: раскраска квадрата O(N^2), поиск пустого места О(N^2), удаление квадрата O(N^2).

В оставшемся после установки 3х квадратов месте максимум можно поставить N/4 – 1 квадрат, то есть второй цикла while + его содержимое – О(N^3).

Сложность с учётом первого цикла while определить сложно, но на заданном наборе данных с учетом внутреннего содержимого цикла временная сложность О(N^4).

Разработанный программный код см. в приложении А.

## Тестирование

Результаты тестирования представлены в табл. 1.

Таблица 1 – Результаты тестирования

|  |  |  |  |
| --- | --- | --- | --- |
| № п/п | Входные данные | Выходные данные | Комментарии |
|  | 1 | Wrong value of N. | Для N меньших заданного диапазона работает корректно. |
|  | 52 | Wrong value of N. | Для N больших заданного диапазона работает корректно. |
|  | 2 | 4  1 1 1  2 1 1  1 2 1  2 2 1 | Для чётного числа работает корректно. |
|  | 3 | 6  1 1 2  3 1 1  1 3 1  2 3 1  3 2 1  3 3 1 | Для простого числа работает корректно. |
|  | 6 | 4  1 1 3  4 1 3  1 4 3  4 4 3 | Для составного числа работает корректно. |
|  | 29 | 14  1 1 15  16 1 14  1 16 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 | Для большого простого числа корректно. |

## Выводы

В ходе работы был успешно изучен алгоритм поиска с возвратом, а также на его основе разработана программа, решающая задачу о квадрировании квадрата.

# Приложение А Исходный код программы

Название файла: main.cpp

#include <iostream>

#include <vector>

#include <time.h>

class Square{

public:

int x;

int y;

int size;

Square(int x, int y, int size) : x(x), y(y), size(size) {};

friend std::ostream& operator<<(std::ostream& os, const Square& square){

os << "Квадрат с координатами верхнего левого угла x = " << square.x << ", y = " << square.y << " и размером стороны " << square.size;

return os;

}

};

class Desk {

public:

int N; // Размер сетки

int num\_occupied; // количество занятых ячеек

bool\*\* occupied; // Динамический массив для хранения состояния ячеек

// Конструктор

Desk(int n) : N(n), num\_occupied(0) {

// Инициализация двумерного массива

occupied = new bool\*[N];

for (int i = 0; i < N; ++i) {

occupied[i] = new bool[N]{false}; // Инициализируем все ячейки как свободные

}

}

// Деструктор

~Desk() {

for (int i = 0; i < N; ++i) {

delete[] occupied[i]; // Освобождаем память

}

delete[] occupied;

}

void placeSquare(const Square& square) {

if (square.x + square.size > N || square.y + square.size > N) {

throw std::out\_of\_range("Square exceeds desk boundaries");

}

for (int i = square.x; i < square.x + square.size; ++i) {

for (int j = square.y; j < square.y + square.size; ++j) {

occupied[i][j] = true;

++num\_occupied;

}

}

}

void removeSquare(const Square& square) {

if (square.x + square.size > N || square.y + square.size > N) {

throw std::out\_of\_range("Square exceeds desk boundaries");

}

for (int i = square.x; i < square.x + square.size; ++i) {

for (int j = square.y; j < square.y + square.size; ++j) {

occupied[i][j] = false;

--num\_occupied;

}

}

}

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

if (x + size > N || y + size > N) {

return false;

}

for (int i = x; i < x + size; ++i) {

for (int j = y; j < y + size; ++j) {

if (occupied[i][j])

return false;

}

}

return true;

}

// Проверка на заполненность (все ячейки)

bool isFull() const {

return num\_occupied == N \* N;

}

// Нахождение максимального пустого квадрата, который можно вставить

Square findEmptySquare() {

// Найдем первую(наиболее верхнюю и левую) свободную ячейку

int firstEmptyX = -1, firstEmptyY = -1;

for (int i = 0; i < N; ++i){

for (int j = 0; j < N; ++j){

if (!occupied[i][j]){

firstEmptyX = i;

firstEmptyY = j;

break;

}

}

if (firstEmptyX != -1) break;

}

// Если есть хотя бы одна пустая клетка, то пытаемся вставить квадрат наибольшего размера

int maxCoord, possibleSide;

if (firstEmptyX != -1 && firstEmptyY != -1){

maxCoord = std::max(firstEmptyX, firstEmptyY);

possibleSide = N - maxCoord;

while (!canPlace(firstEmptyX, firstEmptyY, possibleSide)){

--possibleSide;

}

return {firstEmptyX, firstEmptyY,possibleSide};

}

return {0, 0, 0};

}

friend std::ostream& operator<<(std::ostream& os, const Desk& desk){

for (int i = 0; i < desk.N; ++i){

for (int j = 0; j < desk.N; ++j){

os << desk.occupied[i][j] << ' ';

}

os << std::endl;

}

return os;

}

};

class State{

public:

std::vector<Square> currentResult; //уже добавленные квадраты

Desk currentDesk; // текущее состояние занятости доски

int minSquares; // текущее минимальное количество квадратов

std::vector<Square> bestResult; //квадраты,участвующие в расстановке, соответствующей minSquares

State(int N) : currentDesk(Desk(N)), minSquares(N \* N + 1), currentResult({}), bestResult({}) {}

void addSquare(Square square){

currentDesk.placeSquare(square);

currentResult.push\_back(square);

std::cout << "Добавлено:" << square << std::endl;

}

void removeLastSquare(){

std::cout << "Удалено:" << currentResult.back() << std::endl;

currentDesk.removeSquare(currentResult.back());

currentResult.pop\_back();

}

std::vector<Square> find\_solution(){

//Оптимизация 1 - для любого квадрата со стороной - простым числом(он в мктод всегда таким уже подаётся) заранее ставим оптимальные 3 квадрата

int lsize = (currentDesk.N + 1) / 2, ssize = (currentDesk.N) / 2;

addSquare({0, 0, lsize});

addSquare({lsize, 0, ssize});

addSquare({0, lsize, ssize});

bool flag = true;

while (currentResult.size() > 3 || flag){

flag = false;

while (!currentDesk.isFull()){

// Оптимизация - отсечение заведомо проигрышного решения

if (currentResult.size() >= minSquares){

break;

std::cout << "Текущее решение хуже имеющегося лучшего\n";

}

// Оптимизация 2 - нахождение наибольшего свободного квадрата

Square finding\_square = currentDesk.findEmptySquare();

addSquare(finding\_square);

}

// Оставляем лучший результат из текущего и имеющегося лучшего

if (currentResult.size() < minSquares) {

std::cout << "Текущий результат оказался лучше - было " << minSquares << " квадратов в расстановке,стало - " << currentResult.size() << "\n";

minSquares = currentResult.size();

bestResult = currentResult;

}

std::cout << \*this;

removeLastSquare();

// Убираем все квадраты со стороной 1, потому что их никак не уменьшить

while (!currentResult.empty() && currentResult[currentResult.size()-1].size == 1){

removeLastSquare();

}

//Если возможно, то у последнего не единичного квадрата у меньшаем сторону на 1 и дальшем рассматриваем такой вариант

if (currentResult.size() > 3){

Square last\_square = currentResult.back();

currentDesk.removeSquare(last\_square);

currentDesk.placeSquare({last\_square.x, last\_square.y, last\_square.size - 1});

currentResult[currentResult.size()-1].size -= 1;

std::cout << "У квадрата с координатами x = " << currentResult.back().x << ", y = " << currentResult.back().y << " уменьшаем сторону на 1 и она становится равна: " << currentResult.back().size << std::endl;

}

}

return bestResult;

}

friend std::ostream& operator<<(std::ostream& os, const State& state){

os << "\nОдно из решений:\n";

os << "Минимальное количество квадратов: " << state.minSquares << "\n";

os << "Текущие рассматриваемые квадраты:\n";

int N = 1;

for (const auto& square : state.currentResult) {

os << N << ") " << square << "\n"; // Использует оператор вывода для класса Square

++N;

}

os << "\n";

return os;

}

};

//Функция нахождения первого простого делителя,кроме 1

int firstDivisor(int N) {

for (int d = 2; d \* d <= N; ++d) {

if (N % d == 0) return d;

}

return N;

}

int main() {

int N;

std::cin >> N;

if (N < 2 || N > 30){

std::cout << "Wrong value of N.\n";

}

else{

clock\_t start = clock();

int d = firstDivisor(N);

//Коэффициент масштабирования

int scale = N / d;

State state(d);

std::vector<Square> smallResult = state.find\_solution();

std::vector<Square> finalResult;

//Полученный результат умножаем на коэфициент масштабирования и координаты сдвигаем на 1

for (const auto& sq : smallResult) {

finalResult.push\_back({sq.x \* scale + 1, sq.y \* scale + 1, sq.size \* scale});

}

std::cout << finalResult.size() << std::endl;

for (const auto& sq : finalResult) {

std::cout << sq.x << " " << sq.y << " " << sq.size << std::endl;

}

clock\_t end = clock();

double seconds = (double)(end - start) / CLOCKS\_PER\_SEC;

//printf("The time: %f seconds\n", seconds);

}

return 0;

}