**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**

**Вариант 1р.**

|  |  |  |
| --- | --- | --- |
| Студент гр. 3388 |  | Кулач Д.В. |
| Преподаватель |  | Жангиров Т.Р. |

Санкт-Петербург

2025

**Цель работы.**

Разработка и реализация эффективного алгоритма поиска с возвратом для решения задачи квадрирования квадрата при условии использования минимального общего количества 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.



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

**Описание алгоритма.**

Алгоритм должен найти такое покрытие столешницы квадратами различных размеров, чтобы:

* Квадраты не пересекались между собой.
* Их объединение полностью покрывало область N \* N.
* Количество использованных квадратов было минимальным.

Для этого используется комбинация предопределенных разбиений для специальных случаев и рекурсивного поиска с возвратом для общего случая, дополненная оптимизациями, сокращающими время выполнения.

Основные этапы алгоритма:

1. Обработка специальных случаев. Для некоторых значений N, кратных 2, 3 или 5, существуют заранее известные оптимальные различения, которые позволяют избежать полного рекурсивного поиска. Если N соответствует одному из этих условий, алгоритм возвращает существующее разбиение и завершает работу.
2. Инициализация. Для общего случая (когда N не кратно 2, 3 или 5):
   * Создается двумерный массив grid размером N \* N, где изначально все клетки свободны.
   * Инициализируется переменная best\_solution, которая хранит минимальное найденное количество квадратов (начальное значение — N \* N, максимальное возможное).
   * Создается массив best\_solution для хранения координат и размеров квадратов оптимального решения.
3. Начальное размещение. Чтобы уменьшить область поиска, алгоритм начинает с размещения трех крупных квадратов:
   * Первый квадрат размером maxW = (N + 1) / 2 размещается в позиции (0, 0).
   * Второй квадрат размером bigW = N - maxW — в позиции (0, maxW).
   * Третий квадрат размером bigW — в позиции (maxW, 0).  
     Эти квадраты покрывают значительную часть столешницы, оставляя меньшую область для дальнейшего поиска.
4. Рекурсивный поиск с возвратом. Основная часть алгоритма — функция backtrack, которая рекурсивно заполняет оставшуюся площадь:
   * Входные параметры:
     + best\_solution — текущее количество размещенных квадратов.
     + squares — массив текущих квадратов (координаты и размеры).
     + remaining\_area — оставшаяся незакрытая площадь.
   * Процесс:
     1. Если squares >= best\_solution, ветвь обрезается, так как текущее решение не улучшит найденное.
     2. Находится первая свободная клетка в grid.
     3. Вычисляется максимальный размер квадрата, который можно разместить в этой клетке (не больше (N + 1) / 2 и ограниченный границами или занятыми клетками).
     4. Для каждого размера (от большего к меньшему):
        + Проверяется, можно ли разместить квадрат этого размера без перекрытия и выхода за границы.
        + Если можно, квадрат размещается: обновляется grid, squares и remaining\_area.
        + Рекурсивно вызывается backtrack для следующей свободной клетки.
        + После возврата квадрат удаляется (откат), чтобы попробовать другой размер.
     5. Если remaining\_area == 0 и squares < best\_solution, решение сохраняется как новое оптимальное.
5. Используемые оптимизации:
   * Специальные случаи: Использование предопределенных различений для N, кратных 2, 3 или 5.
   * Начальные крупные квадраты: Уменьшают оставшуюся площадь поиска.
   * Размещение от большего к меньшему: Позволяет быстрее покрывать большие области.
   * Отсечение ветвей: Прекращение поиска, если текущее количество квадратов не улучшает минимум.
   * Ограничение размера, Максимальный размер квадрата ограничен (N + 1) / 2.
   * Учет оставшейся площади: Квадрат размещается, только если его площадь не превышает remaining\_area.
   * Быстрый поиск свободной клетки: Эффективно определяет следующую позицию.

**Оценка сложности.**

Временная сложность зависит от того, как алгоритм отрабатывает входные данные:

* Специальные случаи. Если N кратно 2, 3 или 5, алгоритм использует заранее известное разбиение и выполняется за константное время: O(1).
* Общий случай. В общем случае алгоритм использует рекурсивый поиск с возвратом, что приводит к более высокой сложности. В худшем случае алгоритм перебираем все возможные комбинации размещения квадратов в сетке N \* N. На каждом шаге рекурсии:
  + Алгоритм пытается разместить квадрат максимального возможного размера в первый свободной ячейке, уменьшая размер, если размещение невозможно.
  + Число возможных размеров квадрата для каждой позиции ограничено O(N).
  + Глубина рекурсии может достигать O(N^2), если вся столешница заполняется квадратами 1 \* 1.
* На каждом уровне рекурсии количество вариантов выбора размера квадрата составляет O(N), а общее число шагов может быть экспоненциальным из-за ветвления.
* Формально, в худшем случае временная сложность достигает O(N^(N^2)), что является экспоненциальной зависимостью.

Сложность по памяти. Пространственная сложность определяется используемыми структурами данных и рекурсивным стеком:

1. Структуры данных:
   * Массив grid размером N \* N занимает O(N^2) памяти.
   * Массивы squares и optimalSolution, хранящие информацию о размещенных квадратах, в худшем случае содержат до N^2 элементов (если вся сетка заполнена квадратами 1 \* 1). Это также занимает O(N^2) памяти.
2. Рекурсивный стек:
   * Глубина рекурсии в худшем случае достигает O(N^2).
   * На каждом уровне рекурсии хранится константное количество данных, что дает дополнительную память O(N^2).

Общая память складывается из затрат на структуры данных и рекурсивный стек, что в сумме составляет O(N^2).

**Тестирование.**

Алгоритм был протестирован на различных наборах входных данных

Табл.1

|  |  |
| --- | --- |
| Входные данные | Выходные данные |
| 2 | 4  1 1 1  2 1 1  1 2 1  2 2 1 |
| 11 | 11  1 1 6  1 7 5  7 1 5  6 7 3  6 10 2  7 6 1  8 6 1  8 10 1  8 11 1  9 6 3  9 9 3 |
| 15 | 6  1 1 10  1 11 5  6 11 5  11 1 5  11 6 5  11 11 5 |
| 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 |

**Выводы.**

В ходе выполнения лабораторной работы был реализован алгоритм поиска с возвратом для решения задачи квадрирования квадрата, так же проведено тестирование реализованного алгоритма.

**Приложение А.  
исходный код программы**

Файл main.cpp

#include <iostream>

#include <vector>

#include <tuple>

#include <algorithm>

using namespace std;

bool can\_place(const vector<vector<int>>& grid, int x, int y, int size, int N) {

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 (grid[i][j] != 0) return false;

return true;

}

void place\_square(vector<vector<int>>& grid, int x, int y, int size, int label) {

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

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

grid[i][j] = label;

}

pair<int,int> find\_empty(const vector<vector<int>>& grid, int N) {

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

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

if (grid[i][j] == 0) return {i, j};

return {-1, -1};

}

void backtrack(vector<vector<int>>& grid, vector<tuple<int,int,int>>& squares, int N,

vector<tuple<int,int,int>>& best\_solution, int remaining\_area) {

if (!best\_solution.empty() && squares.size() >= best\_solution.size())

return;

auto [x, y] = find\_empty(grid, N);

if (x == -1) {

if (best\_solution.empty() || squares.size() < best\_solution.size())

best\_solution = squares;

return;

}

int max\_size = min(N - x, N - y);

int limit = N - (N + 1) / 2;

if (max\_size > limit)

max\_size = limit;

for (int size = max\_size; size >= 1; size--) {

int area = size \* size;

if (area <= remaining\_area && can\_place(grid, x, y, size, N)) {

place\_square(grid, x, y, size, squares.size() + 1);

squares.emplace\_back(x, y, size);

backtrack(grid, squares, N, best\_solution, remaining\_area - area);

squares.pop\_back();

place\_square(grid, x, y, size, 0);

}

}

}

vector<tuple<int,int,int>> squaring\_the\_square(int N) {

vector<vector<int>> grid(N, vector<int>(N, 0));

vector<tuple<int,int,int>> best\_solution;

if (N % 2 == 0) {

best\_solution = {{0, 0, N/2}, {N/2, 0, N/2}, {0, N/2, N/2}, {N/2, N/2, N/2}};

return best\_solution;

}

if (N % 3 == 0) {

int t2 = N\*2/3, t1 = N/3;

best\_solution = {

{0, 0, t2}, {0, t2, t1}, {t1, t2, t1},

{t2, 0, t1}, {t2, t1, t1}, {t2, t2, t1}

};

return best\_solution;

}

if (N % 5 == 0) {

int t3 = N\*3/5, t2 = N\*2/5, t1 = N/5;

best\_solution = {

{0, 0, t3}, {t3, 0, t2}, {t3, t2, t2}, {0, t3, t2},

{t2, t3, t1}, {t2, 4\*t1, t1}, {t3, 4\*t1, t1}, {4\*t1, 4\*t1, t1}

};

return best\_solution;

}

int maxW = (N + 1) / 2;

int bigW = N - maxW;

vector<tuple<int,int,int>> squares = {

{0, 0, maxW},

{0, maxW, bigW},

{maxW, 0, bigW}

};

place\_square(grid, 0, 0, maxW, 1);

place\_square(grid, 0, maxW, bigW, 2);

place\_square(grid, maxW, 0, bigW, 3);

backtrack(grid, squares, N, best\_solution, N\*N);

return best\_solution;

}

int main() {

int N; cin >> N;

auto solution = squaring\_the\_square(N);

cout << solution.size() << "\n";

for (auto& [x, y, size] : solution)

cout << x + 1 << " " << y + 1 << " " << size << "\n";

return 0;

}