**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 |  | Лексин М.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 minSquareCount, которая хранит минимальное найденное количество квадратов (начальное значение — N \* N, максимальное возможное).
   * Создается массив optimalSolution 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 maxWidth = (N + 1) / 2 размещается в позиции (0, 0).
   * Второй квадрат размером largerWidth = N - maxWidth — в позиции (0, maxWidth).
   * Третий квадрат размером largerWidth — в позиции (maxWidth, 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:
     + currentCount — текущее количество размещенных квадратов.
     + squares — массив текущих квадратов (координаты и размеры).
     + remainingArea — оставшаяся незакрытая площадь.
   * Процесс:
     1. Если currentCount >= minSquareCount, 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 remainingArea.
        + Рекурсивно вызывается backtrack для следующей свободной клетки.
        + После возврата квадрат удаляется (откат), чтобы попробовать другой размер.
     5. Если remainingArea == 0 и currentCount < minSquareCount, 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 remainingArea.
   * 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 |
| N = 2 | Итоговое решение:  4  1 1 1  2 1 1  1 2 1  2 2 1 |
| N = 11 | Итоговое решение:  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 |
| N = 15 | Итоговое решение:  6  1 1 10  1 11 5  6 11 5  11 1 5  11 6 5  11 11 5 |
| N = 37 | Итоговое решение:  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**

**Код.**

Файл main.c

#include <stdio.h>

#include <stdlib.h>

#include <stdbool.h>

typedef struct {

int row;

int col;

int size;

} SquareInfo;

int gridSize;

int minSquareCount;

SquareInfo\* optimalSolution;

int\*\* grid;

int backtrackCalls = 0;

int squaresPlaced = 0;

int squaresRemoved = 0;

void printGrid() {

printf("\n═════ Текущее состояние сетки %d×%d ═════\n", gridSize, gridSize);

printf(" ");

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

printf("%2d ", j + 1);

}

printf("\n ");

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

printf("───");

}

printf("\n");

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

printf("%2d │ ", i + 1);

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

if (grid[i][j] == 0) {

printf(" · ");

} else {

printf("%2d ", grid[i][j]);

}

}

printf("\n");

}

printf("════════════════════════════════════\n");

}

bool canPlaceSquare(int row, int col, int size) {

printf("Проверка возможности размещения квадрата размера %d в позиции (%d, %d)\n",

size, row + 1, col + 1);

if (row + size > gridSize || col + size > gridSize) {

printf(" Квадрат выходит за границы сетки\n");

return false;

}

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

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

if (grid[i][j] != 0) {

printf(" Ячейка (%d, %d) уже занята квадратом #%d\n", i + 1, j + 1, grid[i][j]);

return false;

}

}

}

printf(" Квадрат можно разместить\n");

return true;

}

void markArea(int row, int col, int size, int value) {

if (value > 0) {

printf("Размещение квадрата #%d размера %d в позиции (%d, %d)\n",

value, size, row + 1, col + 1);

squaresPlaced++;

} else {

printf("Удаление квадрата из позиции (%d, %d) размера %d\n",

row + 1, col + 1, size);

squaresRemoved++;

}

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

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

grid[i][j] = value;

}

}

}

void saveSolution(int count, SquareInfo\* squares) {

printf("\nНАЙДЕНО НОВОЕ ОПТИМАЛЬНОЕ РЕШЕНИЕ: %d квадратов (ранее было: %d)\n",

count, minSquareCount);

minSquareCount = count;

printf("Решение содержит следующие квадраты:\n");

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

optimalSolution[i].row = squares[i].row + 1;

optimalSolution[i].col = squares[i].col + 1;

optimalSolution[i].size = squares[i].size;

printf(" Квадрат #%d: позиция (%d, %d), размер %d\n",

i + 1, optimalSolution[i].row, optimalSolution[i].col, optimalSolution[i].size);

}

printf("\n");

}

bool findFirstEmptyCell(int\* outRow, int\* outCol) {

for (int row = 0; row < gridSize; row++) {

for (int col = 0; col < gridSize; col++) {

if (grid[row][col] == 0) {

\*outRow = row;

\*outCol = col;

printf("Найдена первая пустая ячейка в позиции (%d, %d)\n", row + 1, col + 1);

return true;

}

}

}

printf("Пустых ячеек не найдено. Все заполнено!\n");

return false;

}

bool handleSpecialCases() {

printf("\nПроверка особых случаев для gridSize = %d\n", gridSize);

if (gridSize % 2 == 0) {

int half = gridSize / 2;

printf("Особый случай: gridSize делится на 2. Можно покрыть 4 квадратами размера %d\n", half);

SquareInfo squares[4] = {

{0, 0, half},

{half, 0, half},

{0, half, half},

{half, half, half}

};

saveSolution(4, squares);

return true;

} else if (gridSize % 3 == 0) {

int third = gridSize / 3;

printf("Особый случай: gridSize делится на 3. Можно покрыть 6 квадратами\n");

SquareInfo squares[6] = {

{0, 0, 2 \* third},

{0, 2 \* third, third},

{third, 2 \* third, third},

{2 \* third, 0, third},

{2 \* third, third, third},

{2 \* third, 2 \* third, third}

};

saveSolution(6, squares);

return true;

} else if (gridSize % 5 == 0) {

int fifth = gridSize / 5;

printf("Особый случай: gridSize делится на 5. Можно покрыть 8 квадратами\n");

SquareInfo squares[8] = {

{0, 0, 3 \* fifth},

{3 \* fifth, 0, 2 \* fifth},

{3 \* fifth, 2 \* fifth, 2 \* fifth},

{0, 3 \* fifth, 2 \* fifth},

{2 \* fifth, 3 \* fifth, fifth},

{2 \* fifth, 4 \* fifth, fifth},

{3 \* fifth, 4 \* fifth, fifth},

{4 \* fifth, 4 \* fifth, fifth}

};

saveSolution(8, squares);

return true;

}

printf("Для N = %d не найдено особых случаев. Применяем общий алгоритм\n", gridSize);

return false;

}

int min3(int a, int b, int c) {

int min = a;

if (b < min) min = b;

if (c < min) min = c;

return min;

}

void backtrack(int currentCount, SquareInfo\* squares, int remainingArea) {

backtrackCalls++;

printf("\n----- Вызов backtrack #%d -----\n", backtrackCalls);

printf("Текущее количество квадратов: %d, Оптимальное: %d, Оставшаяся площадь: %d\n",

currentCount, minSquareCount, remainingArea);

if (currentCount >= minSquareCount) {

printf("Отсечение ветви: текущее количество квадратов >= минимальное (%d >= %d)\n",

currentCount, minSquareCount);

return;

}

int row, col;

if (findFirstEmptyCell(&row, &col)) {

int maxSize = min3(gridSize - row, gridSize - col, (gridSize + 1) / 2);

printf("Максимально возможный размер квадрата для позиции (%d, %d): %d\n",

row + 1, col + 1, maxSize);

for (int size = maxSize; size >= 1; size--) {

printf("\nПробуем поставить квадрат размера %d в позицию (%d, %d)...\n",

size, row + 1, col + 1);

if (size \* size <= remainingArea && canPlaceSquare(row, col, size)) {

markArea(row, col, size, currentCount + 1);

printGrid();

squares[currentCount].row = row;

squares[currentCount].col = col;

squares[currentCount].size = size;

printf("Рекурсивный вызов с добавленным квадратом #%d (размер %d, позиция (%d, %d))\n",

currentCount + 1, size, row + 1, col + 1);

backtrack(currentCount + 1, squares, remainingArea - size \* size);

printf("Возврат из рекурсии, удаление квадрата из позиции (%d, %d) размера %d\n",

row + 1, col + 1, size);

markArea(row, col, size, 0);

printGrid();

} else {

if (size \* size > remainingArea) {

printf(" Квадрат слишком большой для оставшейся площади: %d > %d\n",

size \* size, remainingArea);

}

}

}

} else if (currentCount < minSquareCount) {

saveSolution(currentCount, squares);

}

}

void solve() {

printf("\n=== Начало решения для N = %d ===\n", gridSize);

optimalSolution = (SquareInfo\*)malloc(gridSize \* gridSize \* sizeof(SquareInfo));

SquareInfo\* squares = (SquareInfo\*)malloc(gridSize \* gridSize \* sizeof(SquareInfo));

if (!handleSpecialCases()) {

printf("\nПрименяем общий алгоритм для N = %d\n", gridSize);

printf("Начальное размещение трех больших квадратов для разделения области\n");

int maxWidth = (gridSize + 1) / 2;

int largerWidth = gridSize - maxWidth;

printf("Максимальная ширина: %d, Оставшаяся ширина: %d\n", maxWidth, largerWidth);

squares[0].row = 0;

squares[0].col = 0;

squares[0].size = maxWidth;

squares[1].row = 0;

squares[1].col = maxWidth;

squares[1].size = largerWidth;

squares[2].row = maxWidth;

squares[2].col = 0;

squares[2].size = largerWidth;

markArea(0, 0, maxWidth, 1);

markArea(0, maxWidth, largerWidth, 2);

markArea(maxWidth, 0, largerWidth, 3);

printGrid();

int remainingArea = gridSize \* gridSize - maxWidth \* maxWidth - 2 \* largerWidth \* largerWidth;

printf("Размещены три начальных квадрата. Оставшаяся площадь: %d\n", remainingArea);

backtrack(3, squares, remainingArea);

}

printf("\n=== ИТОГОВАЯ СТАТИСТИКА ===\n");

printf("Всего вызовов backtrack: %d\n", backtrackCalls);

printf("Размещено квадратов: %d\n", squaresPlaced);

printf("Удалено квадратов: %d\n", squaresRemoved);

printf("\n=== ФИНАЛЬНОЕ РЕШЕНИЕ ===\n");

printf("%d\n", minSquareCount);

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

printf("%d %d %d\n", optimalSolution[i].row, optimalSolution[i].col, optimalSolution[i].size);

}

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

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

grid[i][j] = 0;

}

}

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

int row = optimalSolution[i].row - 1;

int col = optimalSolution[i].col - 1;

int size = optimalSolution[i].size;

markArea(row, col, size, i + 1);

}

printf("\nФинальное расположение квадратов:\n");

printGrid();

free(squares);

free(optimalSolution);

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

free(grid[i]);

}

free(grid);

}

int main() {

int n;

if (scanf("%d", &n) != 1 || n < 2 || n > 40) {

printf("Ошибка: N должно быть в диапазоне [2, 40]\n");

return 1;

}

gridSize = n;

minSquareCount = gridSize \* gridSize;

printf("Задача о разбиении квадрата %d×%d на минимальное количество квадратов\n", n, n);

grid = (int\*\*)malloc(gridSize \* sizeof(int\*));

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

grid[i] = (int\*)calloc(gridSize, sizeof(int));

}

solve();

return 0;

}