**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. 3344 |  | Коняева М.В. |
| Преподаватель |  | Фирсов М.А. |

Санкт-Петербург

2025

**Цель работы.**

Изучить и реализовать алгоритм поиска с возвратом, бэктрекинг.

**Задание.**

**Вариант**: 2и. Итеративный бэктрекинг. Исследование времени выполнения от размера квадрата.

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

**Основные теоретические положения.**

Алгоритм backtracking (возврат к исходным данным) — это метод решения задачи перебора всех возможных вариантов с последующим выбором оптимального решения.

Основная идея backtracking:

1. Первым шагом в алгоритме backtracking является выбор, который заключается в выборе подходящего элемента или состояния, чтобы продолжить решение задачи. Этот шаг определяет потенциальные варианты на следующем этапе алгоритма.
2. После выбора происходит проверка, где текущий выбранный элемент или состояние проверяется на соответствие критериям или ограничениям задачи. Если проверка проходит успешно, то мы переходим к следующему шагу. В противном случае, если элемент не подходит, алгоритм возвращает его, чтобы выбрать другой элемент и повторить проверку.
3. Третий шаг — откат, представляет собой возврат к предыдущему выбору и проверке, когда все возможные варианты были исчерпаны. Откат может произойти в случаях, когда текущий выбор не приводит к решению задачи или к моменту, когда мы исчерпали все возможные варианты.
4. Четвертый шаг происходит после успешной проверки и выбора. На этом шаге мы продолжаем алгоритм, используя текущий выбор и двигаясь вперед к нахождению следующего возможного варианта или решения задачи.

**Описание алгоритма.**

Этот алгоритм решает задачу минимального разбиения квадрата размером n×n на наименьшее возможное количество меньших квадратов. Сначала создаётся начальное разбиение в зависимости от размера квадрата: если размер чётный, он делится на четыре равные части, если простое число — используется специфическое разбиение, а если составное — разбиение основывается на первом простом делителе. Затем алгоритм использует стек для перебора возможных вариантов разбиения. Он начинает с начального состояния, представляющего собой матрицу, где отмечены уже размещённые квадраты. Далее в каждой итерации выбирается первая свободная ячейка, находится максимальный возможный размер нового квадрата, который можно разместить, и проверяются все возможные разбиения. Если найденное разбиение имеет меньше квадратов, чем текущее лучшее, оно сохраняется. Перебор продолжается, пока не будет найдено оптимальное разбиение.

Оценка сложности алгоритма по памяти и по операциям: O(nn^2). Идея алгоритма такова, что мы начинаем добавлять решения от левого верхнего угла квадратами от 1 до n-1. После точка начала смещается вправо и так же добавляем квадраты от 1 до n-1. Таким образом можно представить все операции в виде дерева. В худшем случаи всё дерево будет заполнено. Определим глубину дерева – n2 т.к самым простым решением будет заполнение всей зоны квадратами 1x1, всего их будет n2. Низ дерева можно определить как nn^2 – столько операций совершит код и столько решений нам необходимо хранить.

**Описание функций и структур данных.**

1. Структура State: хранит текущее состояние алгоритма разбиения квадрата. Поля: std::vector<std::vector<int>> matrix – текущая матрица разбиения, где 0 означает пустую ячейку, а другие числа – ID квадратов; int count – количество использованных квадратов; std::vector<std::tuple<int, int, int>> squares – список размещенных квадратов в формате (x, y, size), где x, y – координаты верхнего левого угла квадрата, size – его размер.
2. findStart(const std::vector<std::vector<int>>& matrix) -> std::pair<int, int>: ищет первую свободную (нулевую) ячейку в матрице. Принимает: matrix – текущая матрица разбиения. Возвращает: (y, x) – координаты первой свободной клетки, либо (-1, -1), если пустых клеток нет.
3. findLargestPossibleSquare(const std::vector<std::vector<int>>& matrix, int x, int y) -> int: определяет максимальный размер квадрата, который можно разместить в позиции (x, y). Возвращает**:** Максимально возможный размер квадрата.
4. isPrime(int num) -> bool: проверяет, является ли число num простым.Возвращает: true, если число простое, иначе false.
5. firstPrimeDivisor(int num) -> int: Находит первый простой делитель числа num среди {3, 5, 7, 11, 13, 17, 19, 23}. Возвращает: первый найденный простой делитель или само число, если делителей в списке нет.
6. startSquares(int size) -> std::vector<std::tuple<int, int, int>>: генерирует начальное разбиение квадрата размера size. Принимает**:** size – размер квадрата. Возвращает: вектор квадратов (x, y, size), задающих начальное разбиение.
7. findMinSquarePartition(int n) -> std::vector<std::tuple<int, int, int>>: основной алгоритм поиска минимального разбиения квадрата n × n. Создает начальную матрицу разбиения (initialMatrix) и стек (stack). Заполняет initialMatrix стартовыми квадратами. Перебирает возможные разбиения с помощью стека, добавляя новые состояния. Если достигнуто лучшее разбиение, оно сохраняется. Возвращает: Вектор (x, y, size) – минимальное разбиение квадрата.
8. benchmarkPartition(int maxSize) -> void: измеряет время выполнения алгоритма для различных размеров квадрата. Выводит: таблицу времени работы алгоритма для четных и нечетных значений n.

**Способ хранения частичных решений.**

Способ хранения частичных решений основан на структуре State, которая содержит текущую матрицу разбиения (matrix), количество использованных квадратов (count) и список размещенных квадратов (squares). Основной алгоритм использует стек (stack), куда помещаются состояния при переборе вариантов разбиения. Каждый новый шаг создает копию текущего состояния, добавляя в него новый квадрат, после чего новое состояние помещается в стек. Когда алгоритм достигает полностью заполненной матрицы, он сравнивает текущее количество квадратов с лучшим найденным решением (minSquares). Если текущее разбиение лучше, обновляем bestSolution и minSquares. Таким образом, стек позволяет перебирать возможные разбиения квадрата и отбрасывая неэффективные пути.

**Применённые оптимизации.**

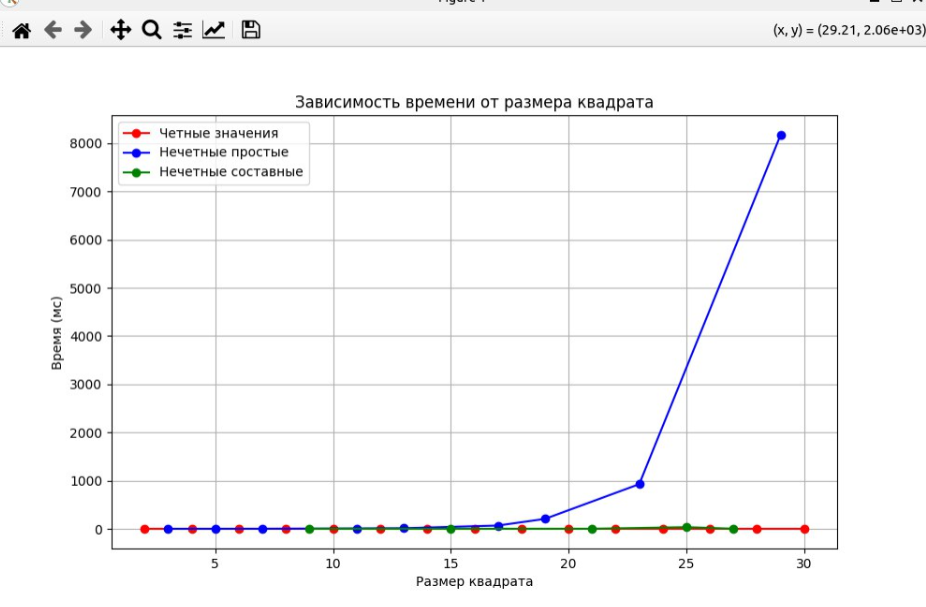
1. Жадный выбор крупных квадратов: предпочтение отдаётся наибольшему возможному квадрату, который может быть размещён в текущей незаполненной позиции. Это сокращает число перебираемых вариантов.
2. Начальное разбиение: перед основным перебором выполняется разбиение на основе свойств числа N (чётность, простота, делители). Это позволяет уменьшить глубину дерева решений. Если размер доски — простое нечетное число, то в левом верхнем углу разместить один квадрат, размер которого равен половине доски + 1, а также два квадрата, размер которых равен половине доски по бокам от него. Если размер доски — составное нечетное число, то в левом верхнем углу разместить один квадрата, который имеет размер, сторона деленная на первый простой делитель умноженная на 2, а по бокам от него два квадрата размером в половину от первого. Если размер доски — чётное число, то всю доску заполняем квадратами одинакового размера, равного половине доски.
3. Отсечение заведомо неэффективных путей: если текущее разбиение уже хуже найденного оптимального (имеет больше квадратов), дальнейший перебор прекращается.

**Тестирование.**

Результаты тестирования приведены в Приложении Б.

**Исследование.**

Исследование времени выполнения от размера квадрата. Результат исследования наглядно показан на графике. Для чётных n решение находится за 1 итерацию благодаря начальному разбиению на 4 квадрата. Для составных нечётных: сложность походит на полиномиальную (близкая к O(n^2)). Для простых нечётных: сложность походит на полиномиальную (близкая к O(n^2))



**Выводы.**

В результате выполнения работы реализован алгоритм, решающий поставленные задачи с использованием бэктрекинга. Программа была протестирована. Выводятся промежуточные результаты.

**Приложение А**

**ИСХОДНЫЙ КОД ПРОГРАММЫ**

Имя файла: lab1\_comments.cpp

#include <iostream>

#include <vector>

#include <stack>

#include <cmath>

#include <limits>

#include <tuple>

#include <chrono>

// Структура для хранения состояния алгоритма

struct State {

    std::vector<std::vector<int>> matrix; // Текущая матрица разбиения

    int count; // Количество использованных квадратов

    std::vector<std::tuple<int, int, int>> squares; // Список размещенных квадратов (x, y, size)

};

// ------------------------------------------------------------------

// 1. Вспомогательные функции

// ------------------------------------------------------------------

// Функция поиска первой свободной ячейки в матрице

std::pair<int, int> findStart(const std::vector<std::vector<int>>& matrix) {

    int n = matrix.size();

    for (int y = 0; y < n; ++y) {

        for (int x = 0; x < n; ++x) {

            if (matrix[y][x] == 0) {

                return {y, x}; // Возвращаем координаты первой найденной пустой ячейки

            }

        }

    }

    return {-1, -1}; // Если пустых ячеек нет

}

// Определяет максимальный размер квадрата, который можно разместить в (x, y)

int findLargestPossibleSquare(const std::vector<std::vector<int>>& matrix, int x, int y) {

    int n = matrix.size();

    int size = 1;

    while (x + size < n && y + size < n && matrix[y][x + size] == 0 && matrix[y + size][x] == 0) {

        ++size;

    }

    return size;

}

// Проверка числа на простоту

bool isPrime(int num) {

    if (num < 2) return false;

    for (int i = 2; i \* i <= num; ++i) {

        if (num % i == 0) return false;

    }

    return true;

}

// Находит первый простой делитель числа

int firstPrimeDivisor(int num) {

    int primes[] = {3, 5, 7, 11, 13, 17, 19, 23};

    for (int p : primes) {

        if (num % p == 0) return p;

    }

    return num;

}

// ------------------------------------------------------------------

// 2. Разбиение доски

// ------------------------------------------------------------------

// Генерирует начальное разбиение доски

std::vector<std::tuple<int, int, int>> startSquares(int size) {

    std::vector<std::tuple<int, int, int>> partition;

    if (size % 2 == 0) {

        for (int i = 0; i < 2; ++i)

            for (int j = 0; j < 2; ++j)

                partition.emplace\_back(i \* (size / 2), j \* (size / 2), size / 2);

    } else if (isPrime(size)) {

        int half = size / 2;

        partition.emplace\_back(0, 0, half + 1);

        partition.emplace\_back(half + 1, 0, half);

        partition.emplace\_back(0, half + 1, half);

    } else {

        int p = firstPrimeDivisor(size);

        partition.emplace\_back(0, 0, size / p \* 2);

        partition.emplace\_back(size / p \* 2, size / p, size / p);

        partition.emplace\_back(size / p, size / p \* 2, size / p);

    }

    return partition;

}

// Основной алгоритм поиска минимального разбиения на квадраты

std::vector<std::tuple<int, int, int>> findMinSquarePartition(int n) {

    // Минимальное количество квадратов, найденное на данный момент (изначально максимально возможное)

    int minSquares = std::numeric\_limits<int>::max();

    std::vector<std::tuple<int, int, int>> bestSolution; // Лучшее разбиение квадрата

    int newSquareID;

    std::stack<State> stack; // Стек для хранения состояний при переборе вариантов

    std::vector<std::vector<int>> initialMatrix(n, std::vector<int>(n, 0)); // Исходная пустая матрица

    std::vector<std::tuple<int, int, int>> initialPartition = startSquares(n); // Начальное разбиение квадрата

    int squareID = 1; // ID первого квадрата

    // Заполняем матрицу начальными квадратами

    for (const auto& partition : initialPartition) {

        int x, y, size;

        std::tie(x, y, size) = partition;

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

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

                initialMatrix[y + i][x + j] = squareID; // Заполняем клетки значением squareID

        squareID++; // Увеличиваем ID для следующего квадрата

    }

    // Добавляем начальное состояние в стек

    stack.push({initialMatrix, (int)initialPartition.size(), initialPartition});

    // Основной цикл перебора возможных разбиений

    int iteration = 0;

    while (!stack.empty()) {

        ++iteration;

        State state = stack.top(); // Получаем текущее состояние

        stack.pop(); // Убираем его из стека

        // Вывод текущего состояния

        std::cout << "\nИтерация " << iteration << ":\n";

        std::cout << "Текущее количество квадратов: " << state.count << "\n";

        std::cout << "Лучший результат: " << (minSquares != std::numeric\_limits<int>::max() ? std::to\_string(minSquares) : "не найден") << "\n";

        // Если текущее разбиение уже хуже найденного лучшего, пропускаем его

        if (state.count >= minSquares) {

            std::cout << "Отбрасываем ветку - результат хуже текущего лучшего\n";

            continue;

        }

        // Ищем первую свободную клетку в текущей матрице

        auto [y, x] = findStart(state.matrix);

        if (y == -1 && x == -1) { // Если пустых клеток нет, значит, матрица полностью заполнена

            std::cout << "!!! Найдено новое решение !!!\n";

            std::cout << "Количество квадратов: " << state.count << " (предыдущий минимум: " << minSquares << ")\n";

            std::cout << "Визуализация заполнения:\n";

            for (const auto& row : state.matrix) {

                for (int cell : row) {

                    std::cout << cell << "\t";

                }

                std::cout << "\n";

            }

            minSquares = state.count; // Обновляем минимальное количество квадратов

            bestSolution = state.squares; // Запоминаем разбиение

            continue;

        }

        // Определяем максимальный возможный размер квадрата, который можно разместить в (x, y)

        int max\_size = findLargestPossibleSquare(state.matrix, x, y);

        std::cout << "Пустая клетка: (" << x + 1 << ", " << y + 1 << "), максимальный размер: " << max\_size << "\n";

        // Перебираем все возможные размеры квадратов от 1 до max\_size

        for (int size = 1; size <= max\_size; ++size) {

            std::vector<std::vector<int>> newMatrix = state.matrix; // Копируем текущее состояние матрицы

            newSquareID = state.squares.size() + 1; // ID нового квадрата

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

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

                    newMatrix[y + i][x + j] = newSquareID; // Заполняем клетки нового квадрата

            std::vector<std::tuple<int, int, int>> newSquares = state.squares; // Копируем список квадратов

            newSquares.emplace\_back(x, y, size); // Добавляем новый квадрат

            // Добавляем новое состояние в стек для дальнейшего рассмотрения

            stack.push({newMatrix, state.count + 1, newSquares});

            std::cout << "Добавлено состояние в стек. Новое количество: " << state.count + 1 << "\n";

        }

    }

    // Вывод итогового результата

    std::cout << "\n=== Итоговый результат ===\n";

    std::cout << "Минимальное количество квадратов: " << minSquares << "\n";

    std::cout << "Расположение квадратов (x, y, size):\n";

    for (const auto& square : bestSolution) {

        std::cout << std::get<0>(square) + 1 << " " << std::get<1>(square) + 1 << " " << std::get<2>(square) << "\n";

    }

    // Визуализация матрицы

    std::cout << "\nВизуализация заполнения:\n";

    for (const auto& row : bestSolution) {

        std::cout << std::get<0>(row) << "\t" << std::get<1>(row) << "\t" << std::get<2>(row) << "\n";

    }

}

int main() {

    int n;

    std::cin >> n;

    findMinSquarePartition(n);

    return 0;

}

**ПРИЛОЖЕНИЕ Б**

**ТЕСТИРОВАНИЕ**

Таблица 1 – Результаты тестирования

|  |  |  |  |
| --- | --- | --- | --- |
| № | Входные данные | Выходные данные | Комментарий |
| 1 | 2 | 4  1 1 1  2 1 1  1 2 1  2 2 1 | Верно |
| 2 | 4 | 4  1 1 2  3 1 2  1 3 2  3 3 2 | Верно |
| 3 | 5 | 8  1 1 3  4 1 2  1 4 2  3 4 2  4 3 1  5 3 1  5 4 1  5 5 1 | Верно |
| 4 | 7 | 9  1 1 4  5 1 3  1 5 3  4 5 2  4 7 1  5 4 1  5 7 1  6 4 2  6 6 2 | Верно |
| 5 | 9 | 6  1 1 3  7 1 3  1 7 3  1 4 3  4 1 3  4 4 6 | Верно |

Вывод в режиме отладки: