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. 3342 Мохамед М.Х.

Преподаватель Виноградова Е.В.

Санкт-Петербург

2025

Цель работы.

Применить на практике алгоритм поиска с возвратом для заполнения квадрата минимальным кол-вом меньших квадратов.

Задание. Вариант – 1р(рекурсивный бэктрекинг).

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).

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 ($2 \le N \le 40$).

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 ($1 \le x,y \le N$) и длину стороны соответствующего обрезка(квадрата).

Теоретические материалы.

Бэктрекинг (поиск с возвратом) – это общий метод нахождения решений задачи, в которой требуется полный перебор всех возможных вариантов в некотором множестве. Решение задачи методом поиска с возвратом сводится к последовательному расширению частичного решения. Если на очередном шаге такое расширение провести не удается, то возвращаются к более короткому частичному решению и продолжают поиск дальше. Данный алгоритм позволяет найти все решения поставленной задачи, если они существуют. Для ускорения метода стараются вычисления организовать таким образом, чтобы как можно

раньше выявлять заведомо неподходящие варианты. Зачастую это позволяет значительно уменьшить время нахождения решения.

Описание алгоритма.

Для решения задачи был реализован итеративный алгоритм бэктрекинга, который перебирает все возможные заполнения квадрата квадратами меньшей стороны:

1. Находим первую попавшуюся свободную ячейку для вставки квадрата
2. Ищем максимальный размер квадрата для вставки
3. Запускаем цикл, в котором перебираем все возможные размеры для вставки(от максимального размера найденного в пункте 2 до единицы)
4. В цикле вставляем квадрат текущего размера в столешницу и добавляем результат вставки(координаты, куда вставляется квадрат, и его размер) в стэк.
5. Рекурсивно запускаем процесс повторно. Рекурсия будет продолжаться до тех пор, пока столешница не будет заполнена. Когда столешница заполнена, выполняется проверка того, что текущее разбиение минимально. И если это так, то запоминаем это разбиение.
6. После рекурсивного вызова происходит удаление вставленного квадрата и функция переходит на следующую итерацию цикла, уменьшая размер вставляемого квадрата на 1.

Оптимизации:

- Для столешницы с четным числом ребер минимальное разбиение всегда будет разбиение на 4 равные части.
- Если N – простое число, то в состав его минимального разбиения будут входить следующие квадраты: ○ N/2 + 1 с координатами (0;0) ○ N/2 с координатами (N/2 + 1; 0) ○ N/2 с координатами (0; N/2 +1)

- Если N – составное число, то его разбиение будет аналогично разбиению его минимального простого делителя в уменьшенном масштабе.

Сложность.

С учетом всех оптимизаций для чисел кратных 2, 3 и 5 программа будет работать за константное время. Для остальных простых числе даже с учетом оптимизации сложность будет экспоненциальной.

Описание функций и структур данных.

Все операции с полем подразумевают работу с матрицей размера N*N. Матрица реализуется с помощью std::vector.

Функция printAnswer() – функция вывода итогового разбиения столешницы с учетом масштаба.

Функция insertBlock() – функция вставки блока с заданными координатами и размером в столешницу.

Функция removeBlock() – функция удаления блока с указанными координатами и размером из столешницы.

Функция findEmpty() – функция поиска свободной ячейки для вставки в текущем разбиении столешницы.

Функция findMaxSize() – функция поиска максимального размера блока для вставки в текущее координаты.

Функция chooseBlock() – функция, выполняющая бэктрекинг. Выбирает очередной размер блока для вставки и рекурсивно вызывает себя для продолжения вставки очередного блока в столешницу.

Функция primeNumber() – функция для работы столешницы с ребрами равными простому числу. Вставляет три первых блока в столешницу и запускает функцию chooseBlock() для вставки новых блоков.

Функции division2, division3 и division5 реализуют решение частных случаев для составных чисел кратных соответственно двум, трем и пяти.
Рекурсивная функция

Рекурсивная функция chooseBlock() играет ключевую роль в алгоритме поиска с возвратом. Она выполняет разбиение столешницы на квадраты минимального количества. Основная идея заключается в последовательном добавлении квадратов в столешницу и проверке, достигнуто ли минимальное разбиение.

Алгоритм работы chooseBlock():

Найти первую свободную ячейку на столешнице.

Определить максимальный возможный размер квадрата, который можно разместить в этой ячейке.

Перебрать все возможные размеры квадратов (от максимального до минимального).

Для каждого возможного квадрата:

Вставить его в столешницу.

Добавить информацию о вставленном квадрате в текущий набор решений.

Рекурсивно вызвать chooseBlock() для поиска следующего подходящего квадрата.

После возврата из рекурсии удалить квадрат и попробовать следующий размер.

Если вся столешница заполнена, проверить, является ли текущее разбиение лучшим (минимальным) и обновить глобальное решение, если оно улучшилось.

Этот механизм позволяет эффективно искать минимальное разбиение, перебирая различные возможные варианты и отсекая заведомо неоптимальные решения.

Хранение частичных решений

Во время выполнения алгоритма хранения частичных решений осуществляется с использованием стека и массива tmpArr.

В начале работы chooseBlock() используется массив tmpArr, который хранит текущее разбиение столешницы.

При добавлении нового квадрата его данные (размер и координаты) добавляются в tmpArr.

Если найдено разбиение, минимизирующее количество квадратов, tmpArr копируется в resArr, который хранит лучшее найденное решение.

После возврата из рекурсии добавленный квадрат удаляется из tmpArr, что позволяет исследовать альтернативные разбиения.

Демонстрация работы.

Ввод	Вывод

2	4 1 1 1 2 1 1 1 2 1 2 2 1
3	6 1 1 2 3 1 1 1 3 1 2 3 1 3 2 1 3 3 1
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

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

Выводы.

Применен на практике алгоритм поиска с возвратом для заполнения квадрата минимальным кол-вом меньших квадратов. В результате работы было придумано несколько оптимизаций, которые позволили уменьшить основание в экспоненциальной сложности, сократив время работы алгоритма.

ПРИЛОЖЕНИЕ А

ИСХОДНЫЙ КОД ПРОГРАММЫ

Файл main.cpp:

```
#include <iostream>
#include <vector>

// Class representing a table that can be divided into blocks
class Table
{
    int N; // Size of the table (N x N)
    int minCounter; // Minimum number of blocks needed to cover the table
    std::vector<std::pair<int, std::pair<int, int>>> resArr; // Stores the result
blocks (size, (x, y))
    std::vector<std::vector<bool>> mainArr; // 2D array representing the table
(true = occupied, false = empty)

public:
    // Constructor to initialize the table
    Table(int N) : N(N), minCounter(N * N), mainArr(N)
    {
        // Resize each row of the table to N columns
        for (int i = 0; i < N; i++)
```

```
            mainArr[i].resize(N);
    }

    // Function to insert a block of size m x m at position (x, y)
    void insertBlock(int, int, int);

    // Function to remove a block of size m x m at position (x, y)
    void removeBlock(int, int, int);

    // Function to find the first empty cell in the table
    std::pair<int, int> findEmpty();

    // Function to find the maximum possible size of a block that can be placed at
(x, y)
    std::pair<int, bool> findMaxSize(int, int);

    // Recursive function to choose blocks and try to cover the table
    void chooseBlock(std::vector<std::pair<int, std::pair<int, int>>>&, int, int,
int);

    // Function to handle the case when N is a prime number
    void primeNumber();

    // Function to print the result (minimum number of blocks and their positions)
    void printAnswer(int scale = 1);

    // Function to handle the case when N is divisible by 2
    void division2();

    // Function to handle the case when N is divisible by 3
    void division3();

    // Function to handle the case when N is divisible by 5
    void division5();
};

// Function to print the result (minimum number of blocks and their positions)
void Table::printAnswer(int scale)
{
    std::cout << minCounter << '\n'; // Print the minimum number of blocks
    for (int i = 0; i < minCounter; i++)
    {
        // Print the position and size of each block, scaled by the given scale
        std::cout << resArr[i].second.first * scale + 1 << ' ' <<
resArr[i].second.second * scale + 1 << ' ' << resArr[i].first * scale << '\n';
    }
}
```

```cpp
// Function to insert a block of size m x m at position (x, y)
void Table::insertBlock(int m, int x, int y)
{
    for (int i = 0; i < m; i++)
    {
        for (int j = 0; j < m; j++)
        {
            mainArr[x + i][y + j] = true; // Mark the cells as occupied
        }
    }
}

// Function to remove a block of size m x m at position (x, y)
void Table::removeBlock(int m, int x, int y)
{
    for (int i = 0; i < m; i++)
    {
        for (int j = 0; j < m; j++)
        {
            mainArr[x + i][y + j] = false; // Mark the cells as empty
        }
    }
}

// Function to find the first empty cell in the table
std::pair<int, int> Table::findEmpty()
{
    // Start searching from the middle of the table to the end
    for (int i = N / 2; i < N; i++)
    {
        for (int j = N / 2; j < N; j++)
        {
            if (!mainArr[i][j]) // If the cell is empty
                return std::make_pair(i, j); // Return its coordinates
        }
    }
    return std::make_pair(-1, -1); // Return (-1, -1) if no empty cell is found
}

// Function to find the maximum possible size of a block that can be placed at (x,
y)
std::pair<int, bool> Table::findMaxSize(int x, int y)
{
    // Check the maximum possible size in the row
    for (int i = y + 1; i < N; i++)
    {
        if (mainArr[x][i]) // If an occupied cell is found
        {
            if (N - x == i - y) // If the block can be a square
```

```
                return std::make_pair(N - x, true); // Return the size and true
(square)
            return std::make_pair((N - x > i - y) ? i - y : N - x, false); //
Return the size and false (not square)
        }
    }
    // If no occupied cell is found in the row
    if (N - x == N - y) // If the block can be a square
        return std::make_pair(N - x, true); // Return the size and true (square)
    return std::make_pair((N - x > N - y) ? N - y : N - x, false); // Return the
size and false (not square)
}

// Recursive function to choose blocks and try to cover the table
void Table::chooseBlock(std::vector<std::pair<int, std::pair<int, int>>>& tmpArr,
int counter, int x, int y)
{
    std::pair<int, int> coord = findEmpty(); // Find the first empty cell
    if (coord.first == -1) // If no empty cell is found (table is fully covered)
    {
        if (tmpArr.size() < minCounter) // If the current solution is better than
the previous best
        {
            resArr = tmpArr; // Update the result
            minCounter = tmpArr.size(); // Update the minimum counter
        }
        return;
    }
    if (counter + 1 >= minCounter) // If the current solution is already worse than
the best, stop
    {
        return;
    }
    int tmpBestCounter = minCounter; // Store the current best counter

    std::pair<int, bool> maxSize = findMaxSize(coord.first, coord.second); // Find
the maximum possible block size at the empty cell
    if (maxSize.second) // If the block can be a square
    {
        tmpArr.push_back(std::make_pair(maxSize.first, coord)); // Add the block to
the temporary solution
        insertBlock(maxSize.first, coord.first, coord.second); // Insert the block
into the table
        chooseBlock(tmpArr, counter + 1, x, y); // Recursively try to cover the
rest of the table
        removeBlock(maxSize.first, coord.first, coord.second); // Remove the block
(backtracking)
        tmpArr.pop_back(); // Remove the block from the temporary solution
    }
```

```
    else // If the block cannot be a square
    {
        for (int i = maxSize.first; i >= 1; i--) // Try all possible block sizes
from max to 1
        {
            if (tmpBestCounter > minCounter && i == 1) // If the current solution
is already worse, skip
                continue;
            tmpArr.push_back(std::make_pair(i, coord)); // Add the block to the
temporary solution
            insertBlock(i, coord.first, coord.second); // Insert the block into the
table
            chooseBlock(tmpArr, counter + 1, x, y); // Recursively try to cover the
rest of the table
            removeBlock(i, coord.first, coord.second); // Remove the block
(backtracking)
            tmpArr.pop_back(); // Remove the block from the temporary solution
        }
    }
}

// Function to handle the case when N is a prime number
void Table::primeNumber()
{
    // Insert three initial blocks to cover the table
    insertBlock(N / 2 + 1, 0, 0);
    insertBlock(N / 2, N / 2 + 1, 0);
    insertBlock(N / 2, 0, N / 2 + 1);
    int counter = 3; // Counter for the number of blocks used
    int minCounter = N * N; // Initialize the minimum counter
    std::vector<std::pair<int, std::pair<int, int>>> tmpArr; // Temporary array to
store the blocks
    tmpArr.push_back(std::make_pair(N / 2 + 1, std::make_pair(0, 0))); // Add the
first block
    tmpArr.push_back(std::make_pair(N / 2, std::make_pair(N / 2 + 1, 0))); // Add
the second block
    tmpArr.push_back(std::make_pair(N / 2, std::make_pair(0, N / 2 + 1))); // Add
the third block
    chooseBlock(tmpArr, counter, N / 2, N / 2); // Start the recursive process to
cover the rest of the table
}

// Function to handle the case when N is divisible by 2
void Table::division2()
{
    if (N % 2 == 0) // Check if N is divisible by 2
    {
        int N_div = N / 2; // Calculate the size of each sub-block
```

```
        std::cout << 4 << '\n'; // Print the number of blocks (always 4 for N
divisible by 2)
        // Print the positions and sizes of the 4 blocks
        std::cout << 1 << ' ' << 1 << ' ' << N_div << '\n';
        std::cout << N_div + 1 << ' ' << 1 << ' ' << N_div << '\n';
        std::cout << 1 << ' ' << N_div + 1 << ' ' << N_div << '\n';
        std::cout << N_div + 1 << ' ' << N_div + 1 << ' ' << N_div << '\n';
    }
}

// Function to handle the case when N is divisible by 3
void Table::division3()
{
    int realN = N; // Store the original size of the table
    int scale = N / 3; // Calculate the scaling factor
    N = 3; // Set N to 3 (since the table is divided into 3x3 sub-blocks)
    primeNumber(); // Solve the problem for the 3x3 table
    printAnswer(scale); // Print the result, scaled by the scaling factor
}

// Function to handle the case when N is divisible by 5
void Table::division5()
{
    int realN = N; // Store the original size of the table
    int scale = N / 5; // Calculate the scaling factor
    N = 5; // Set N to 5 (since the table is divided into 5x5 sub-blocks)
    primeNumber(); // Solve the problem for the 5x5 table
    printAnswer(scale); // Print the result, scaled by the scaling factor
}

int main()
{
    int N;
    std::cin >> N; // Input the size of the table
    Table table(N); // Create a Table object

    if (N % 2 == 0) // If N is divisible by 2
    {
        table.division2(); // Handle the case
        return 0;
    }

    if (N % 3 == 0) // If N is divisible by 3
    {
        table.division3(); // Handle the case
    }
    else if (N % 5 == 0) // If N is divisible by 5
    {
        table.division5(); // Handle the case
```

```
    }
    else // If N is a prime number
    {
        table.primeNumber(); // Handle the case
        table.printAnswer(); // Print the result
    }

    return 0;
}
```