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. 9382 _____ Рыжих Р.В.

Преподаватель _____ Фирсов М.А.

Санкт-Петербург
2021

Цель работы.

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

Задание.

Вариант 1р.

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 сведения.

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 рекурсивный 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.

В данной программе была реализована рекурсивная функция choose-Block, которая «прокручивает» все возможные варианты заполнения столешницы квадратиками и выбирает вариант, при котором количество квадратиков будет минимальным. Работа рекурсивной функции описана в работе алгоритма.

Аргументы, которые передаются в функцию: сама столешница (vector<vector<bool>>& mainArr), стэк (vector<pair<int, pair<int, int>>>& tmpArr), количество квадратиков (int counter) и координаты (int x, int y)

Демонстрация работы.

Ввод	Вывод

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
10	4 1 1 5 6 1 5 1 6 5 6 6 5
23	13 1 1 12 13 1 11 1 13 11 12 13 2 12 15 5 12 20 4 13 12 1 14 12 3 16 20 1 16 21 3 17 12 7

	17 19 2 19 19 5
39	6 1 1 26 27 1 13 1 27 13 14 27 13 27 14 13 27 27 13

25	8 1 1 15 16 1 10 1 16 10 11 16 10 16 11 5 21 11 5 21 16 5 21 21 5

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

```
#include <iostream>
#include <vector>
using namespace std;

int N;
int minCounter;
vector<pair<int, pair<int, int>>> resArr;

void printAnswer(int scale = 1)
{
    cout << minCounter << '\n';
    for (int i = 0; i < minCounter; i++)
    {
        cout << resArr[i].second.first * scale + 1 << ' ' << resArr[i].second.second *
scale + 1 << ' ' << resArr[i].first * scale << '\n';
    }
}

void insertBlock(vector<vector<bool>>& mainArr, int m, int x, int y) //Вставка квадрата
размером m * m с левым верхним углов в точке (x, y)
{
    for (int i = 0; i < m; i++)
    {
        for (int j = 0; j < m; j++)
        {
            mainArr[x + i][y + j] = true;
        }
    }
}

void removeBlock(vector<vector<bool>>& mainArr, int m, int x, int y) //Вставка квадрата
размером m * m с левым верхним углов в точке (x, y)
{
    for (int i = 0; i < m; i++)
    {
        for (int j = 0; j < m; j++)
        {
            mainArr[x + i][y + j] = false;
        }
    }
}

pair<int, int> findEmpty(vector<vector<bool>>& mainArr) //Поиск первой свободной ячейки
для вставки
{
    for (int i = N / 2; i < N; i++)
    {
        for (int j = N / 2; j < N; j++)
        {
            if (!mainArr[i][j])
                return make_pair(i, j);
        }
    }
    return make_pair(-1, -1);
}

pair<int, bool> findMaxSize(vector<vector<bool>>& mainArr, int x, int y)
{
    for (int i = y + 1; i < N; i++)
    {
```

```
        if (mainArr[x][i])
        {
            if (N - x == i - y)
                return make_pair(N - x, true);
            return make_pair((N - x > i - y) ? i - y : N - x, false);
        }
    }
    if (N - x == N - y)
        return make_pair(N - x, true);
    return make_pair((N - x > N - y) ? N - y : N - x, false);
}

void chooseBlock(vector<vector<bool>>& mainArr, vector<pair<int, pair<int, int>>>&
tmpArr, int counter, int x, int y)
{
    pair<int, int> coord = findEmpty(mainArr);
    if (coord.first == -1)
    {
        if (tmpArr.size() < minCounter)
        {
            resArr = tmpArr;
            minCounter = tmpArr.size();
        }
        return;
    }
    if (counter + 1 >= minCounter)
    {
        return;
    }
    int tmpBestCounter = minCounter;
    //поиск места для вставки блока
    pair<int, bool> maxSize = findMaxSize(mainArr, coord.first, coord.second); //поиск
максимального размера блока для вставки на найденное пустое место
    if (maxSize.second)                                                        //Если
можно вставить сразу блок максимального размера
    {
        tmpArr.push_back(make_pair(maxSize.first, coord));
        insertBlock(mainArr, maxSize.first, coord.first, coord.second);
        chooseBlock(mainArr, tmpArr, counter + 1, x, y); //вставляем очередной блок
        removeBlock(mainArr, maxSize.first, coord.first, coord.second);
        tmpArr.pop_back();
    }
    else
    {
        for (int i = maxSize.first; i >= 1; i--)
        {
            if (tmpBestCounter > minCounter && i == 1)
                continue;
            tmpArr.push_back(make_pair(i, coord));
            insertBlock(mainArr, i, coord.first, coord.second);
            chooseBlock(mainArr, tmpArr, counter + 1, x, y); //вставляем очередной блок
            removeBlock(mainArr, i, coord.first, coord.second);
            tmpArr.pop_back();
        }
    }
}

void primeNumber(vector<vector<bool>>& mainArr) //вставка начальных блоков и начало ра-
боты бэктрекинга
{
    insertBlock(mainArr, N / 2 + 1, 0, 0);
    insertBlock(mainArr, N / 2, N / 2 + 1, 0);
    insertBlock(mainArr, N / 2, 0, N / 2 + 1);
    int counter = 3;
    int minCounter = N * N;
```

```
    vector<pair<int, pair<int, int>>> tmpArr;
    tmpArr.push_back(make_pair(N / 2 + 1, make_pair(0, 0)));
    tmpArr.push_back(make_pair(N / 2, make_pair(N / 2 + 1, 0)));
    tmpArr.push_back(make_pair(N / 2, make_pair(0, N / 2 + 1)));
    chooseBlock(mainArr, tmpArr, counter, N / 2, N / 2);
}

void division2()
{
    if (N % 2 == 0)
    {
        int N_div = N / 2;
        cout << 4 << '\n';
        cout << 1 << ' ' << 1 << ' ' << N_div << '\n';
        cout << N_div + 1 << ' ' << 1 << ' ' << N_div << '\n';
        cout << 1 << ' ' << N_div + 1 << ' ' << N_div << '\n';
        cout << N_div + 1 << ' ' << N_div + 1 << ' ' << N_div << '\n';
    }
}

void division3(vector<vector<bool>>& mainArr)
{
    int realN = N;
    int scale = N / 3;
    N = 3;
    primeNumber(mainArr);
    printAnswer(scale);
}

void division5(vector<vector<bool>>& mainArr)
{
    int realN = N;
    int scale = N / 5;
    N = 5;
    primeNumber(mainArr);
    printAnswer(scale);
}

int main()
{
    cin >> N;
    minCounter = N * N;
    if (N % 2 == 0)
    {
        division2();
        return 0;
    }
    //создание "карты" дял стола
    vector<vector<bool>> mainArr(N);
    for (int i = 0; i < N; i++)
        mainArr[i].resize(N);

    if (N % 3 == 0)
    {
        division3(mainArr);
    }
    else if (N % 5 == 0)
    {
        division5(mainArr);
    }
    else
    {
        primeNumber(mainArr);
        printAnswer();
    }
```

```
return 0;
   }
```