

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).

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 ($2 \leq N \leq 20$).

Выходные данные

Одно число K , задающее минимальное количество обрезков(квадратов), из которых можно построить столешницу(квадрат) заданного размера N . Далее должны идти KK строк, каждая из которых должна содержать три целых числа x , y , и w , задающие координаты левого верхнего угла ($1 \leq x, y \leq N$) 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

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 удастся, 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 стек.
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 чисел даже с учетом оптимизации сложность будет экспоненциальной.

Сложность по памяти $O(N*N)$, т.к. программа работает с матрицей (столешницей) размера $N*N$.

Описание функций и структур данных.

Все операции с полем подразумевают работу с матрицей размера $N*N$.

Матрица реализуется с помощью `std::vector`.

`void printAnswer(int scale = 1)` – функция вывода итогового разбиения столешницы с учетом масштаба.

`void insertBlock(std::vector<std::vector<bool>>& mainArr, int m, int x, int y)` – функция вставки блока с заданными координатами и размером в столешницу.

`void removeBlock(std::vector<std::vector<bool>>& mainArr, int m, int x, int y)` – функция удаления блока с указанными координатами и размером из столешницы.

`std::pair<int, int> findEmpty(std::vector<std::vector<bool>>& mainArr)` –

функция поиска свободной ячейки для вставки в текущем разбиении столешницы.

std::pair<int, bool> findMaxSize(std::vector<std::vector<bool>>& mainArr, int x, int y) – функция поиска максимального размера блока для вставки в текущие координаты.

void chooseBlock(std::vector<std::vector<bool>>& mainArr, std::vector<std::pair<int, std::pair<int, int>>>& tmpArr, int counter, int x, int y) – функция, выполняющая бэктрекинг. Выбирает очередной размер блока для вставки и рекурсивно вызывает себя для продолжения вставки очередного блока в столешницу.

void primeNumber(std::vector<std::vector<bool>>& mainArr) – 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 *chooseBlock*, 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*), стек (*vector<pair<int, pair<int, int>>>& tmpArr*), количество квадратиков (*int counter*) и координаты (*int x, int y*)

Демонстрация работы.

В демонстрации работы вывод отладочной информации не включался, т.к.

иначе он занимал бы очень много места, ввиду того, что функция рекурсивная.

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>
#include <Windows.h>

//#define INFO
#define FIELD

class Lab1
{
public:
    int N;
    int minCounter;
    //int depth = 0;
    std::vector<std::pair<int, std::pair<int, int>>> resArr;

    void printField(std::vector<std::vector<bool>>& mainArr)
    {
        system("cls");
        for (int i = 0; i < N; i++)
        {
            for (int j = 0; j < N; j++)
            {
                if (mainArr[i][j])
                    std::cout << "1 ";
                else
                    std::cout << "0 ";
            }
            std::cout << std::endl;
        }
        Sleep(500);
    }

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

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

```

        {
            mainArr[x + i][y + j] = false;
        }
    }
}

std::pair<int, int> findEmpty(std::vector<std::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 std::make_pair(i, j);
        }
    }
    return std::make_pair(-1, -1);
}

std::pair<int, bool> findMaxSize(std::vector<std::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 std::make_pair(N - x, true);
            return std::make_pair((N - x > i - y) ? i - y : N - x, false);
        }
    }
    if (N - x == N - y)
        return std::make_pair(N - x, true);
    return std::make_pair((N - x > N - y) ? N - y : N - x, false);
}

void chooseBlock(std::vector<std::vector<bool>>& mainArr, std::vector<std::pair<int,
std::pair<int, int>>>& tmpArr, int counter, int x, int y)
{
    #ifdef FIELD
    printField(mainArr);
    #endif
    #ifdef INFO
    //for (int i = 0; i < depth; i++)
    //    std::cout << " ";
    std::cout << "Поиск квадрата для вставки...\n";
    #endif
    std::pair<int, int> coord = findEmpty(mainArr);
    if (coord.first == -1)
    {
        if (tmpArr.size() < minCounter)
        {
            resArr = tmpArr;
            minCounter = tmpArr.size();
        }
        #ifdef INFO
        //for (int i = 0; i < depth; i++)
        //    std::cout << " ";
        std::cout << " Столешница закончилась!\n\n";
        #endif
        return;
    }
    if (counter + 1 >= minCounter)
    {

```

```

        return;
    }
    int tmpBestCounter = minCounter;
    //поиск места для вставки блока
    std::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
    {
        #ifdef INFO
            //for (int i = 0; i < depth; i++)
            //    std::cout << " ";
            std::cout << " Вставляем квадрат размера " << maxSize.first << " по координатам (" << coord.first+1 << ', ' << coord.second+1 << ")\n\n";
        #endif
        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);
        #ifdef FIELD
            printField(mainArr);
        #endif
        #ifdef INFO
            std::cout << " Удаляем квадрат размера " << maxSize.first << " по координатам (" << coord.first + 1 << ', ' << coord.second + 1 << ")\n";
        #endif
        //depth--;
        tmpArr.pop_back();
    }
    else
    {
        for (int i = maxSize.first; i >= 1; i--)
        {
            if (tmpBestCounter > minCounter && i == 1)
                continue;
            #ifdef INFO
                //for (int i = 0; i < depth; i++)
                //    std::cout << " ";
                std::cout << " Вставляем квадрат размера " << i << " по координатам (" << coord.first+1 << ', ' << coord.second+1 << ")\n\n";
            #endif
            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);
            #ifdef FIELD
                printField(mainArr);
            #endif
            #ifdef INFO
                std::cout << " Удаляем квадрат размера " << i << " по координатам (" << coord.first + 1 << ', ' << coord.second + 1 << ")\n";
            #endif
            tmpArr.pop_back();
        }
    }
}

void primeNumber(std::vector<std::vector<bool>>& mainArr) //вставка начальных блоков
и начало работы бэктрекинга
{
    #ifdef INFO
        std::cout << "Заполнение столешницы можно начать со вставки 3-х квадратов.\n";
        std::cout << "Делается это для сокращения времени работы алгоритма.\n";
        std::cout << "Вставляем квадрат размера " << N/2+1 << " на координаты (1,1)\n";
    #endif
}

```

```

        std::cout << "и 2 других квадрата размера " << N/2 << " на координаты (" << N/2+2
<< ",1) и (1," << N/2+2 << ").\n\n";
    #endif
    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;
    std::vector<std::pair<int, std::pair<int, int>>> tmpArr;
    tmpArr.push_back(std::make_pair(N / 2 + 1, std::make_pair(0, 0)));
    tmpArr.push_back(std::make_pair(N / 2, std::make_pair(N / 2 + 1, 0)));
    tmpArr.push_back(std::make_pair(N / 2, std::make_pair(0, N / 2 + 1)));
    chooseBlock(mainArr, tmpArr, counter, N / 2, N / 2);
}

void division2()
{
    #ifdef INFO
    std::cout << "Тривиальный случай, когда размер квадрата - чётный.\n";
    std::cout << "Минимальное количество квадратов всегда будет 4, а\n";
    std::cout << "их расположение - по углам столешницы.\n";
    #endif
    if (N % 2 == 0)
    {
        int N_div = N / 2;
        std::cout << 4 << '\n';
        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';
    }
}

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

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

};

int main()
{
    setlocale(LC_ALL, "Russian");
    Lab1 a;
    std::cin >> a.N;
    a.minCounter = a.N * a.N;
    if (a.N % 2 == 0)
    {
        a.division2();
        return 0;
    }
    //создание "карты" столешницы
    std::vector<std::vector<bool>> mainArr(a.N);

```

```
    for (int i = 0; i < a.N; i++)
        mainArr[i].resize(a.N);

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

    return 0;
}
```