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

Студентка гр. 9383 _____ Карпекина А.А.

Преподаватель _____ Фирсов М.А.

Санкт-Петербург
2021

Цель работы.

Научиться применять на практике алгоритм поиска с возвратом. Реализовать заполнение большого квадрата наименьшим количеством маленьких квадратиков.

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

Выходные данные:

Одно число K, задающее минимальное количество обрезков(квадратов), из которых можно построить столешницу(квадрат) заданного размера N. Далее должны идти K 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).

Описание алгоритма

Представим квадрат в виде матрицы, размер которой зависит от введенного параметра. Матрица в начале работы программы заполнена 0.

Сначала ставим три квадрата, которые будут занимать большую часть площади искомого квадрата, заносим их в текущий вектор. Проверяем с

помощью метода IsArrayFull класса Processor заполнен ли исходный квадрат, затем, если не заполнен, то с текущей координаты, установленной методом, проверяем, квадрат какой наибольшей величины мы можем разместить слева-направо сверху-вниз в матрице. После заполняем это пространство единицами и вводим информацию о координатах и размере в текущем векторе с сохраненным путем обхода. Если текущий размер больше размера выходного вектора, прекращаем итерацию и идем дальше.

Если после проверки матрица оказывается заполненной, тогда вызываем метод PopBackData у класса Processor, который удалит из вектора те значения квадратов, у которых размер равен 1. Извлечение происходит до тех пор, пока в векторе не останется трех исходных квадратов. Затем мы меняем значение флага, который сигнализирует об окончании поиска. Если значение больше или равно двум, то квадрат возвращается в вектор, при этом размер уменьшается на единицу, значения в матрице обновляются в соответствии с изменениями.

Когда значение флага изменилось, то останавливаем перебор и выводим текущий результат.

Оценка сложности алгоритма по времени: $o(2*(n-1)^2*n^n)$.

Оценка сложности алгоритма по памяти: $o(n^2)$.

Функции и структуры данных

класс Square:

- int size — размер стороны
- int x и int y — координаты квадрата
- Square(int s, int x, int y) — конструктор, который принимает значение размера и координат квадрата
- ~Square() - деструктор
- int GetX() - возвращает x координату квадрата
- int GetY() - возвращает y координату квадрата
- int GetSize() - возвращает размер квадрата

класс Processor:

- bool IsArrayFull(int** array, int array_size, int& x, int& y, std::vector<Square>& data, int data_size); — возвращает True/False в зависимости от того, есть ли в матрице свободные клетки
- void PrintData(std::vector<Square> data) — выводит на экран вектор с квадратами
- void PopBackData(int** array, int array_size, std::vector<Square>& data, int& data_size, int& flag) — удаляет из вектора квадраты определенного размера
- int FindNextSide(int** array, int array_size, int x, int y) — находит максимальный квадрат, от координат и возвращает размер этого квадрата
- void SquareSetting(int** array, int array_size, int x, int y, int square_side, int flag) — функция ставит нули или единицы в матрице начиная с координаты (x,y) и размера size слева-направо, сверху-вниз в матрице
- int** array — матрица с нулями и единицами
- int array_size — размер матрицы
- int& x— x координата первой пустой клетки
- int& y — y координата первой пустой клетки,
- std::vector<Square>& data — вектор с квадратами
- int data_size — длина вектора
- int& flag — флаг для проверки окончания перебора
- int color — «цвет», в который нужно закрасить квадрат

Тестирование

Таблица 1. Результаты работы программы

Входные данные	Выходные данные
7	9 0 0 4 0 4 3 4 0 3

	3 4 2 3 6 1 4 3 1 4 6 1 5 3 2 5 5 2
17	12 0 0 9 0 9 8 9 0 8 8 9 2 8 11 4 8 15 2 9 8 1 10 8 3 10 15 2 12 11 1 12 12 5 13 8 4
13	11 0 0 7 0 7 6 7 0 6 6 7 2 6 9 4 7 6 1 8 6 3 10 9 1 10 10 3 11 6 2 11 8 2

Выводы.

В ходе выполнения работы был изучен алгоритм поиска с возвратом. Также был реализован код для поиска минимального количества квадратов, заполняющих исходный квадрат.

ПРИЛОЖЕНИЕ А
ИСХОДНЫЙ КОД ПРОГРАММЫ

Название файла: lb1.cpp

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


class Square
{

private:
    int side=0;
    int x=0;
    int y=0;
public:
    Square()=default;
    Square(int s, int x, int y): side(s), x(x), y(y){}
    ~Square() = default;
    int GetX()
    {
    return this->x;
    }
    int GetY()
    {
    return this->y;
    }
    int GetSide()
    {
    return this->side;
    }
};


class Processor
{
```

```
public:
    int FindNextSide(int** array, int array_size, int x, int y);
    void SquareSetting(int** array, int array_size, int x, int y, int
square_side, int flag);
    bool IsArrayFull(int** array, int array_size, int& x, int& y,
std::vector<Square>& data, int data_size);
    void PopBackData(int** array, int array_size, std::vector<Square>&
data, int& data_size, int& flag);
    void PrintData(std::vector<Square> data);
};

int Processor::FindNextSide(int** array, int array_size, int x, int y)
{
    int right_side = 1, down_side = 1;
    int i = x, j = y;
    int max_side = array_size-1;

    while (right_side <= max_side && j+right_side<array_size &&
array[i][j+right_side]!=1 )
        right_side++;

    while (down_side <= max_side && i+down_side<array_size &&
array[i+down_side][j]!=1 )
        down_side++;
    int side=std::min(right_side, down_side);
    return side;
}

void Processor::SquareSetting(int** array, int array_size, int x, int
y, int square_side, int flag)
{
    for (int i = x; i < x + square_side; i++)
    {
        for (int j = y; j < y + square_side;j++)
            array[i][j]=flag;
    }
}
```

```
bool Processor::IsArrayFull(int** array, int array_size, int& x, int&
y, std::vector<Square>& data, int data_size)
{
for (int i=0;i<array_size;i++)
{
        for (int j=0;j<array_size;j++)
        {
            if (array[i][j]==0)
            {
                x = i;
                y = j;
                return true;
            }
        }
    }
    return false;
}

void Processor::PopBackData(int** array, int array_size,
std::vector<Square>& data, int& data_size, int& flag)
{
   while (true)
   {
       if (data_size<=3)
       {
           flag=1;
           break;
        }
        else
        {
            Square square = data[data_size-1];
            if (data[data_size-1].GetSide()>=2)
            {
                data.pop_back();
                data_size-=1;
                this->SquareSetting(array, array_size, square.GetX(),
square.GetY(), square.GetSide(), 0);
                this->SquareSetting(array, array_size, square.GetX(),
square.GetY(), square.GetSide()-1, 1);
```

```
                data.push_back(Square(square.GetSide()-1,
square.GetX(), square.GetY()));
                data_size+=1;
                break;
            }
            else
            {
                data.pop_back();
                data_size-=1;
                this->SquareSetting(array, array_size, square.GetX(),
square.GetY(), square.GetSide(), 0);
                continue;
            }
        }
    }
}

void Processor::PrintData(std::vector<Square> data)
{
    for (int i=0;i<data.size();i++)
        std::cout<<data[i].GetX()<<' '<<data[i].GetY()<<'
'<<data[i].GetSide()<<"\n";
}

int main()
{
    std::vector <Square> result;
    std::vector <Square> actual;
    Processor processor;
    int n;
    std::cin>>n;
    int desired_side = n*n;

    if (n%2==0)
    {
        desired_side = 4;
        result.push_back(Square(n/2, 0, 0));
        result.push_back(Square(n/2, 0, n/2));
        result.push_back(Square(n/2, n/2, 0));
```

```
    result.push_back(Square(n/2, n/2, n/2));
}
else
{

    int** array = new int*[n];
    for (int i=0; i<n; i++) array[i]= new int[n];

    for (int i=0;i<n;i++)
    {
        for (int j=0;j<n;j++)
            array[i][j]=0;
    }

    if (n%3==0)
    {
        actual.push_back(Square(n*2/3, 0, 0));
        actual.push_back(Square(n/3, 0, n*2/3));
        actual.push_back(Square(n/3, n*2/3,0));

        processor.SquareSetting(array, n, 0, 0, n*2/3, 1);
        processor.SquareSetting(array, n, 0, n*2/3, n/3, 1);
        processor.SquareSetting(array, n, n*2/3, 0, n/3, 1);

    }
    else
    {
        actual.push_back(Square(n/2 + 1, 0, 0));
        actual.push_back(Square(n/2, 0, n/2+1));
        actual.push_back(Square(n/2, n/2+1,0));

        processor.SquareSetting(array, n, 0, 0, n/2+1, 1);
        processor.SquareSetting(array, n, 0, n/2+1, n/2, 1);
        processor.SquareSetting(array, n, n/2+1, 0, n/2, 1);
    }

    int actual_x = 0;
    int actual_y = 0;
    int actual_max_side = 3;
```

```
        int new_square_side = 0;
        int flag=0;

        while (!actual.empty() && flag!=1)
        {

            if (n>11 && actual_max_side>=n)
                processor.PopBackData(array, n, actual,
actual_max_side, flag);

            if (actual_max_side>=desired_side)
                processor.PopBackData(array, n, actual,
actual_max_side, flag);

            if (processor.IsArrayFull(array, n, actual_x, actual_y,
actual, actual_max_side))
            {
                int new_side = processor.FindNextSide(array, n,
actual_x, actual_y);
                actual.push_back(Square(new_side, actual_x, actual_y));
                actual_max_side+=1;
                processor.SquareSetting(array, n, actual_x, actual_y,
new_side, 1);
            }
            else
            {
                result = actual;
                desired_side = actual_max_side;
                processor.PopBackData(array, n, actual,
actual_max_side, flag);
            }
        }
        for (int i=0; i<n; i++)
            delete [] array[i];
        delete [] array;
    }

    std::cout << desired_side << std::endl;
    processor.PrintData(result);
```

```
    return 0;
}
```