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. 9382 _____ Голубева В.П.

Преподаватель _____ Фирсов М.А.

Санкт-Петербург

2021

Цель работы.

Получить представление о решении NP — полных задач, изучить такой метод решения, как поиск с возвратом, проследить зависимость количества операций для решения поставленной задачи от входных данных.

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

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

Вариант 3и.

Итеративный бэктрекинг. Исследование количества операций от размера квадрата

Описание алгоритма

Для удобной работы с квадратом представим его в качестве матрицы. Размер будет зависеть от введённого размера. Изначально она будет заполнена ноликами, мы, в ходе работы алгоритма будем ставить туда единички. При этом текущим решением будет вектор, содержащий объекты класса Square, хранящие информацию о координатах и размерах поставленных квадратиков на матрице, также текущим решением будет количество этих квдратиков.

Сначала ставим три квадрата, которые будут занимать 75% площади искомого квадрата, заносим их в текущий вектор. Проверяем с помощью метода CheckAllMatrix класса Handler 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, затем заполняем это пространство единицами, также заносим информацию о координатах и размере в текущий вектор с хранимым путём обхода. Если текущий размер больше размера для выходного вектора, прекращаем перебор на этой итерации, переходим к следующей.

Если после проверки матрица оказывается заполненной, тогда вызываем метод PopBackVector у класса Handler, который удалит из вектора те значения квадратов, у которых размер равен 1. Извлечение происходит, до тех пор, пока в векторе не останется три первоначальных квадрата. Тогда мы ставим меняем значение флага, которое сигнализирует об окончании перебора. Если же значение больше или равно двум, тогда квадрат возвращается в вектор, с уменьшенным на единицу размером, значения в матрице обновляются в соответсвии с изменениями.

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

Оценка сложности алгоритма по времени: в NP-полных задачах оценка затруднена, так как достигает очень больших значений. Для прохода по матрице и поиску пустой клетки требуется $o(n^2)$, в каждом из них таких проходов может понадобиться $o(n^2)$ и тд, то есть это получается $o(n^n)$, чтобы поставить и удалить квадрат с поля трубуется $o(2*(n-1)^2)$, т.е. По времени выходит $o(2*(n-1)^2*n^n)$.

Оценка сложности алгоритма по памяти: на каждом шаге храним матрицу размера n. Храним вектор с текущими решениями, который не может превышать размера 2n(заполнить квадрат квдратом n-1, а оставшуюся часть размером 1). И храним переменные для количества, которые не увеличивают сложность. Таким образом, сложность по памяти равна $o(n^2)$.

Функции и структуры данных

Был реализован класс Square с полями

int size — размер квадрата, int x_coordinate и int y_coordinate — координаты квадрата, Square() - конструктор, Square(int s, int x, int y) — конструктор, принимает значение размера и координат квадрата, ~Square() - деструктор, int GetXCoordinate() - возвращает x-вую координату квадрата, int GetYCoordinate() - возвращает y-ковую координату квадрата, int GetSize() - возвращает размер квадрата

Также был реализован класса-помощник Handler. В нём были реализованы функции bool CheckAllMatrix(int** matrix, int matrix_size, int& x_coordinate, int& y_coordinate, std::vector<Square>& vector, int vector_size) - int** matrix — матрица с нулями и единицами, int matrix_size — размер этой матрицы, int& x_coordinate — здесь будет храниться x-вая координата первой пустой клетки, int& y_coordinate - здесь будет храниться y-ковая координата первой пустой клетки, std::vector<Square>& vector — вектор с квадратами, int vector_size — длина этого вектора, функции возвращает True/False в зависимости от того, есть ли в матрице свободные клетки(заполненные

нулями) или нет. void PrintMatrix(int** matrix, int marix_size) — принимает int** matrix — матрица с нулями и единицами, int matrix_size — размер этой матрицы, выводит матрицу на экран. void PrintVector(std::vector<Square> vector) — принимает std::vector<Square> vector — вектор с квадратами и выводит его на экран, void PopBackVector(int** matrix, int matrix_size, std::vector<Square>& vector, int& vector_size, int& flag) — принимает int** matrix — матрица с нулями и единицами, int matrix_size — размер этой матрицы, std::vector<Square>& vector — вектор с квадратами, int& vector_size — длина этого вектора, int& flag — флаг для проверки окончания перебора, функция удаляет из вектора кадраты определённого размера. int FindNewSizeRightDown(int** matrix, int matrix_size, int x_coordinate, int y_coordinate) — принимает int** matrix — матрица с нулями и единицами, int matrix_size — размер этой матрицы, int x_coordinate и int y_coordinate — координаты квадрата — координаты в матрице, начиная от них направо и вниз нужно найти максимальный квадрат, функция возвращает размер этого квадрата. void SetSquare(int** matrix, int matrix_size, int x_coordinate, int y_coordinate, int square_size, int color) - int** matrix — матрица с нулями и единицами, int matrix_size — размер этой матрицы, int x_coordinate и int y_coordinate — координаты в матрице, int square_size — размер квадрата, int color — «цвет», в который нужно закрасить квадрат, функция ставить нули(если color==0) или единицы(если color==1) в матрице начиная с координаты (x,y) и размера size слева-направо, сверху-вниз в матрице

Тестирование

Протестировано с помощью тестировочной системы на Stepik. Также результаты можно посмотреть в Таблице 1.

Таблица 1. Результаты работы программы

Входные данные	Выходные данные(без

	промежуточного вывода работы программы)
3	Minimal result size: 6 Coordinates & Sizes of result squares: 2 0 0, 1 0 2, 1 2 0, 1 1 2, 1 2 1, 1 2 2, All number of operations for square size 3 = 268
5	Minimal result size: 8 Coordinates & Sizes of result squares: 3 0 0, 2 0 3, 2 3 0, 2 2 3, 1 3 2, 1 4 2, 1 4 3, 1 4 4, All number of operations for square size 5 = 1482
7	Minimal result size: 9 Coordinates & Sizes of result squares: 4 0 0, 3 0 4, 3 4 0, 2 3 4, 1 3 6, 1 4 3, 1 4 6, 2 5 3, 2 5 5, All number of operations for square size 7 = 6924
9	Minimal result size: 6 Coordinates & Sizes of result squares: 6 0 0, 3 0 6, 3 6 0, 3 3 6, 3 6 3, 3 6 6, All number of operations for square size 9 = 3628
11	Minimal result size: 11 Coordinates & Sizes of result squares: 6 0 0, 5 0 6, 5 6 0, 3 5 6, 2 5 9, 1 6 5, 1 7 5, 1 7 9, 1 7 10, 3 8 5, 3 8 8,

	All number of operations for square size 11 = 165231
12	Minimal result size: 4 Coordinates & Sizes of result squares: 6 0 0, 6 0 6, 6 6 0, 6 6 6, All number of operations for square size 12 = 20
15	Minimal result size: 6 Coordinates & Sizes of result squares: 10 0 0, 5 0 10, 5 10 0, 5 5 10, 5 10 5, 5 10 10, All number of operations for square size 15 = 20833

Исследование

Было проведено исследование количества операций от размера квадрата

Что считалось в качестве операции:

1) любая индексация по массиву|вектору

2) присваивание переменной нового значения

3) проверка условия (не сама конструкция if, а именно условия, проверяемые операторами !, ==, >, < и похожими конструкциями)

4) динамическое выделение памяти/её очищение считалось за одну операцию

5) вызов функции, реализованной в стандартной библиотеке

Что не считалось в качестве операции:

1) сложение, вычитание и прочие операции, без присваивания(т.е. a=1+1 - операция, 1+1 - нет)

2) проверка условия, если она находится в цикле for

3) увеличение счётчика количества операций

4) вызов функции, реализованной в программе

5) создание переменной, без инициализации ей значения

6) вывод информации на экран

Если что-то было недостаточно качественно оценено, то, в любом случае, нужно проследить общую тенденцию изменения количества операций, а не его точное число.

Далее приведены результаты подсчёта количества операций в зависимости от размера квадрата., их можно посмотреть в Таблице 2. Также можно проследить зависимость, посмотрев на График 1.

Таблица 2. Количество операций при заданных размерах квадрата

Размер квадрата	Количество операций
2	2
3	268
4	20
5	1482
6	20
7	6924
8	20
9	36070
10	20
11	165231
12	20
13	442752

14	20
15	20833
16	20
17	4582872
18	20
19	14992721
20	20

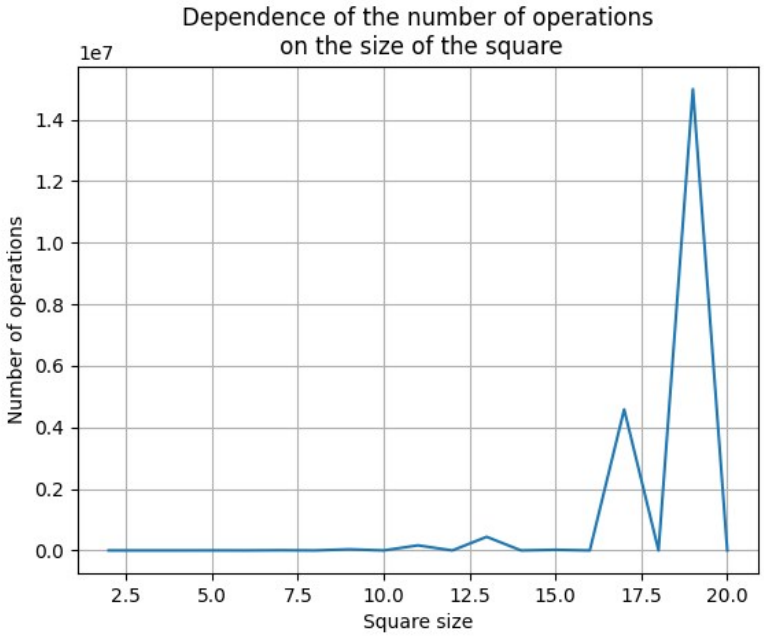

График 1. Зависимость количества операций от размера квадрата

Как видно, прослеживается тенденция увеличения количества операций от размера квадрата, если он нечётный. При чётных значения размера количество константно(кроме первого случая, когда оно ещё меньше).

Выводы.

Был реализован поиск с возвратом для поиска минимального колисетва непересекающихся квадратов, заполняющих исходный квадрат, проанализирована зависимость количества операций от размера, выявлена закономерность роста количества операций от размера.

ПРИЛОЖЕНИЕ А
ИСХОДНЫЙ КОД ПРОГРАММЫ

Название файла: Makefile

```
all:
	g++ all.cpp -o lab1
```

Название файла: all.cpp

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

int count = 0;

class Square{

private:
    int size=0;
    int x_coordinate=0;//top down
    int y_coordinate=0;//from left to right
public:
    Square()=default;
    Square(int s, int x, int y): size(s), x_coordinate(x),
y_coordinate(y){}
    ~Square() = default;
    int GetXCoordinate();
    int GetYCoordinate();
    int GetSize();
};

class Handler{

public:
```

```
    bool CheckAllMatrix(int** matrix, int matrix_size, int&
x_coordinate, int& y_coordinate, std::vector<Square>& vector, int
vector_size);
    void PrintMatrix(int** matrix, int marix_size);
    void PrintVector(std::vector<Square> vector);
    void PopBackVector(int** matrix, int matrix_size,
std::vector<Square>& vector, int& vector_size, int& flag);
    int FindNewSizeRightDown(int** matrix, int matrix_size, int
x_coordinate, int y_coordinate);
    void SetSquare(int** matrix, int matrix_size, int x_coordinate,
int y_coordinate, int square_size, int color);

};

int Square::GetXCoordinate(){
    return this->x_coordinate;
}

int Square::GetYCoordinate(){
    return this->y_coordinate;
}

int Square::GetSize(){
    return this->size;
}

int Handler::FindNewSizeRightDown(int** matrix, int matrix_size, int
x_coordinate, int y_coordinate){
    int right_size = 1, down_size = 1;
    int i = x_coordinate, j = y_coordinate;

    int max_size = matrix_size-1;

    count+=5;

    //find a square with maximum size in this coordinates
```

```
    while (right_size <= max_size && j+right_size<matrix_size &&
matrix[i][j+right_size]!=1 ){
        right_size++;
        count+=4;//increase a right_size and check values in condition
    }

    while (down_size <= max_size && i+down_size<matrix_size &&
matrix[i+down_size][j]!=1 ){
        down_size++;
        count+=4;//increase a right_size and check value in matrix
    }
    int size=std::min(right_size, down_size);
    std::cout<<"Find new square at coordinates: ("<<x_coordinate<<",
"<<y_coordinate<<")... Find a square size: "<<size<<"\n";
    return size;
}

void Handler::SetSquare(int** matrix, int matrix_size, int
x_coordinate, int y_coordinate, int square_size, int color){
    //set square of ones with given size and coordinates in matrix
    for (int i = x_coordinate; i < x_coordinate + square_size; i++){
        for (int j = y_coordinate; j < y_coordinate + square_size;j++)
{
            matrix[i][j]=color;
        }
    }
    count+=square_size*square_size;

}

bool Handler::CheckAllMatrix(int** matrix, int matrix_size, int&
x_coordinate, int& y_coordinate, std::vector<Square>& vector, int
vector_size){
    std::cout<<"Cheking free space in matrix...\n";
    for (int i=0;i<matrix_size;i++){
        for (int j=0;j<matrix_size;j++){
            count+=1;
```

```
            if (matrix[i][j]==0){//if find a free cell
                x_coordinate = i;
                y_coordinate = j;
                count+=2;
                std::cout<<"Find free space at coordinates:
("<<x_coordinate<<", "<<y_coordinate<<")\n";
                return true;
            }
        }
    }
    std::cout<<"Free space didn't find\n";
    return false;
}

void Handler::PopBackVector(int** matrix, int matrix_size,
std::vector<Square>& vector, int& vector_size, int& flag){

   while (true){
       if (vector_size<=3){
           flag=1;
           count+=2;
           std::cout<<"All optimal candidate for filling square was
find, stoped cheking\n";
           break;
        } else{
            Square square = vector[vector_size-1];
            count+=1;
            if (vector[vector_size-1].GetSize()>=2){//delete the last
value in vector, push value with decreasing size

                vector.pop_back();
                vector_size-=1;
                this->SetSquare(matrix, matrix_size,
square.GetXCoordinate(), square.GetYCoordinate(), square.GetSize(),
0);
```

```
                this->SetSquare(matrix, matrix_size,
square.GetXCoordinate(), square.GetYCoordinate(), square.GetSize()-1,
1);
                vector.push_back(Square(square.GetSize()-1,
square.GetXCoordinate(), square.GetYCoordinate()));
                vector_size+=1;
                count+=7;
                break;
            }
            else{//deletethe last value in vector
                vector.pop_back();
                vector_size-=1;
                this->SetSquare(matrix, matrix_size,
square.GetXCoordinate(), square.GetYCoordinate(), square.GetSize(),
1);
                count+=4;
                continue;
            }
        }
    }
}

void Handler::PrintMatrix(int** matrix, int matrix_size){
    for (int i=0;i<matrix_size;i++){
        for (int j=0;j<matrix_size;j++){
            std::cout<<matrix[i][j]<<' ';
        }
        std::cout<<'\n';
    }
    count+=matrix_size*matrix_size;
}

void Handler::PrintVector(std::vector<Square> vector){

    for (int i=0;i<vector.size();i++){
        std::cout<<vector[i].GetSize()<<'
'<<vector[i].GetXCoordinate()<<' '<<vector[i].GetYCoordinate()<<", ";
```

```
    }
    count+=3*vector.size();
    std::cout<<"\n";
}

int main(){

    std::vector <Square> finish;//candidate of result suquence
    std::vector <Square> current;//current path
    Handler handler;

    int n;
    std::cin>>n;

    int result_size = n*n;//max size, if squares has sizes 1

    count+=2;//for following and previous operations

    if (n==2){
        std::cout<<'4'<<"\n";
        std::cout<<"1 1 1\n1 2 1\n2 1 1\n2 2 1\n";
        std::cout<<"\nAll number of operations for square size "<< n
<<" = "<<count<<"\n";

        return 0;
    }

    count+=1;
    if (n%2==0){//especial case
        result_size = 4;
        finish.push_back(Square(n/2, 0, 0));
        finish.push_back(Square(n/2, 0, n/2));
        finish.push_back(Square(n/2, n/2, 0));
        finish.push_back(Square(n/2, n/2, n/2));
        count+=5;
    }
    else{
```

```
int** matrix = new int*[n];

for (int i=0; i<n; i++){
    matrix[i]= new int[n];
}

count+=2*n+1;

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


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

    handler.SetSquare(matrix, n, 0, 0, n*2/3, 1);
    handler.SetSquare(matrix, n, 0, n*2/3, n/3, 1);
    handler.SetSquare(matrix, n, n*2/3, 0, n/3, 1);

    count+=4;
}
else{
    current.push_back(Square(n/2 + 1, 0, 0));
    current.push_back(Square(n/2, 0, n/2+1));
    current.push_back(Square(n/2, n/2+1,0));

    handler.SetSquare(matrix, n, 0, 0, n/2+1, 1);
    handler.SetSquare(matrix, n, 0, n/2+1, n/2, 1);
    handler.SetSquare(matrix, n, n/2+1, 0, n/2, 1);
    count+=4;
```

```
        }

        int current_x = 0;
        int current_y = 0;
        int current_max_size = 3;
        int new_square_size = 0;
        int flag=0;

        count+=5;


        while (!current.empty() && flag!=1){//while not checked all
optimal decisions
            count+=2;

            std::cout<<"\nCurrent path: ";
            handler.PrintVector(current);

            std::cout<<"\nCurrent matrix:\n";
            handler.PrintMatrix(matrix, n);

            count+=3;
            if (n>11 && current_max_size>=n){
                handler.PopBackVector(matrix, n, current,
current_max_size, flag);
                continue;
            }

            count+=1;
            if (current_max_size>=result_size){
                std::cout<<"Current path size is longer then finish
path size, stop cheking further\n";
                handler.PopBackVector(matrix, n, current,
current_max_size, flag);
                continue;
            }
```

```
            //set coordinates in free cell or delete values in vector
if free cell din't find
            if (handler.CheckAllMatrix(matrix, n, current_x,
current_y, current, current_max_size)){
                int new_size = handler.FindNewSizeRightDown(matrix, n,
current_x, current_y);
                current.push_back(Square(new_size, current_x,
current_y));
                current_max_size+=1;
                handler.SetSquare(matrix, n, current_x, current_y,
new_size, 1);
                count+=3;
            }
            else{
                finish = current;
                result_size = current_max_size;
                handler.PopBackVector(matrix, n, current,
current_max_size, flag);
                count+=2;
            }
        }
        for (int i=0; i<n; i++){
            delete [] matrix[i];
        }
        delete [] matrix;
        count+=2*n+1;
    }


    std::cout<<"\nMinimal result size: "<< result_size<<"\nCoordinates
& Sizes of result squares: \n";
    handler.PrintVector(finish);

    std::cout<<"\nAll number of operations for square size "<< n <<" =
"<<count<<"\n";
    return 0;
}
```