МИНОБРНАУКИ РОССИИ
САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ
ЭЛЕКТРОТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ
«ЛЭТИ» ИМ. В.И. УЛЬЯНОВА (ЛЕНИНА)
Кафедра МОЭВМ

ОТЧЕТ

по лабораторной работе №1

по дисциплине «Построение и анализ алгоритмов»

Тема: Поиск с возвратом

Студент гр. 9382 _____ Дерюгин Д.А.

Преподаватель _____ Фирсов М.А.

Санкт-Петербург

2021

Цель работы.

Изучить работу алгоритма поиска с возвратом, найти зависимость количества операций для решения 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($2 \leq N \leq 20$).

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).

Вариант 3и.

Итеративный бэктрекинг. Исследование количества операций от размера квадрата

Описание алгоритма.

Создаем матрицу размера K*K и заполняем ее нулями. После начинает выполняться функция prepare, в которой проверяется число K. Во всех случаях сначала на доске будут находится 3 главных квадрата. Сторона одного квадрата будет равна k/2 + 1, а стороны двух других - k/2(округление в меньшую сторону).

После установки трех главных квадратов доска будет заполняться квадратами с максимально возможной длиной стороны(каждый квадрат заносится в стек). После того, как доска полностью заполнится, будем удалять с \квадраты со стека до тех пор, пока удаляемый квадрат имеет сторону равную единице. Дойдя до более крупного квадрата, мы уменьшаем ее сторону на единицу и заново заполняем доску.

Если в стеке осталось лишь три главных квадрата, то бектрекинг закончен, выводим результат.

Оптимизация алгоритма.

Для доски, сторона которой кратна 2, минимальное количество квадратов - 4. Каждый из этих квадратом имеет сторону размер k/2.

Для доски, сторона которой кратна 3, минимальное количество квадратов - 6. Самый большой квадрат имеет сторону k/3*2 и левый верхний угол его расположен на координате (0;0). Пять остальных квадрата имеют длину стороны k/3

Для доски, сторона которой кратна 5, минимальное количество квадратов - 8. Самый большой квадрат имеет сторону k/5*3 и левый верхний угол его расположен на координате (0;0). Три квадрата будут иметь сторону 2k/5, а оставшиеся 4 квадрата будут иметь сторону k/5.

Сложность алгоритма.

Оценка сложности алгоритма по памяти:

На каждом шаге алгоритма хранится матрица размером n*n, вектор с текущем решением, который не превышает n*n/2, а также вектор с оптимальным решением, который также не превышает n*n/2. Итого сложность алгоритма O(n^2)

Оценка сложности алгоритма по времени:

Для прохода по матрице потребуется n*n шагов. Чтобы вставить новый элемент в матрицу потребуется n/2*n/2 шагов. Чтобы перебрать все возможные варианты матриц потребуется n*n раз выполнить n*n шагов. Итого сложность алгоритма по времени: O(n^n)

Исследование.

Исследовалась записимость количества операций от размера доски.

За операцию считалось:

- Индексация по массиву
- Удаления квадрата из вектора
- Уменьшение стороны квадрата
- Выделение памяти для матрицы

Результаты исследования приведены ниже в таблице 1

Таблица 1. Зависимость количества операций от стороны квадрата

Сторона квадрата	Количество итераций
2	8
3	18
4	32
5	50
6	72
7	910
11	34291
13	108696
15	450
17	1210033
19	4449433
41	997791443

Как видно из таблицы при увеличении стороны квадрата, увеличивается и количества итераций(не работает со сторонами, кратными 2, 3 и 5).

Функции и структуры данных.

Class Square - класс, в котором хранится вся информация об одном квадрате.

Поля класса:

int posX - иксовая позиция левого верхнего угла квадрата

int posY - игриковая позиция левого верхнего угла квадрата

int width - длина стороны квадрата

bool isMain - булевая переменная, которая принимает значение true, если квадрат является главным

Class Desk - класс, который хранит в себе всю информацию о доске в целом

Поля класса:

int minimumSquare - минимальное количество квадратов на доске

int currentSquare - количество квадратов, которые находятся на доске в данным момент

std::stack<Square> minArrayofSquare - вектор, который содержит доску с минимальным количеством квадратов

std::stack<Square> arrayOfSquares - вектор, который содержит информацию обо всех квадратах на доске на данный момент

int countOfIterations - количество итерация, затраченных на нахождения минимального количества квдратов.

Square findMaxSquare(int k , int x, int y, int** matrix) - функция, которая ищет квадрат с маскимальной шириной и левой верхней координатой (x, y)

Возврашает квадрат

k - сторона доски

x, y - координаты верхнего левого угла вставляемого квадрата

matrix - матрица, которая иммитирует доску

void fillMatrix(int** matrix, Square square, Desk* desk) - функция, которая вставляет в матрицу новый квадрат

matrix - матрица, которая иммитирует доску

square - квадрат, который нужно вставить

desk - класс доски

void backtracking(int k , int** matrix, Desk* desk, bool specialCase=false) - функция перебора всех вариантов досок

k - сторона доски

matrix - матрица, которая иммитирует доску

desk - класс доски

specialCase - булевая переменная, которая показывает функции, является ли доска частным случаем(кратны ли стороны 2, 3 или 5).

bool decreaseSquare(int k , int **matrix, Desk* desk) - функция, которая уменьшает ширину первого не единичного квадрата на 1. Если на доске присутствуют 3 главных квадрата, а все остальные квадраты размером 1, тогда возвращает true

k - сторона доски

matrix - матрица, которая иммитирует доску

desk - класс доски

void printOptimalDesk(int** matrix, std::stack<Square> squares)

Выводит на экран минимальное количество квадратов, а также координаты левого верхнего угла этих квадратов.

matrix - матрица, которая иммитирует доску

squares - минимальный набор квадратов

void prepare(int k, int**matrix) - главная функция, которая вызывает остальные функции в зависимости от k

k - сторона доски

matrix - матрица, которая иммитирует доску

Тестирование.

№	Входные данные	Выходные данные
1	4	4 3 3 2 3 1 2 1 3 2 1 1 2
2	5	8 5 5 1 5 4 1 5 3 1 4 3 1 4 1 2 3 4 2 1 4 2 1 1 3
3	7	9 6 6 2 6 4 2 5 7 1 5 4 1 4 7 1 4 5 2 5 1 3 1 5 3 1 1 4
4	11	11 9 9 3 9 6 3 8 11 1 8 10 1 8 6 1 7 6 1 6 10 2 6 7 3

		7 1 5 1 7 5 1 1 6
5	15	6 11 11 5 11 6 5 11 1 5 6 11 5 1 11 5 1 1 10
6	19	13 17 13 3 17 10 3 16 16 4 16 15 1 16 14 1 13 10 4 12 10 1 11 10 1 10 14 6 10 11 3 11 1 9 1 11 9 1 1 10

Вывод.

В данной лабораторной работе был изучен алгоритм поиска с возвратом, а также была найдена зависимость количества операций, требуемых для нахождения минимального количества квадратов на доске от стороны доски

ПРИЛОЖЕНИЕ А
ИСХОДНЫЙ КОД ПРОГРАММЫ

main.cpp

```
#include <iostream>
#include <stack>

bool IS_SHOW_INTERMEDIATE_RESULTS = true;
//array of names of squares
const char namesOfSquares[] = {'0', '1', '2', '3', '4', '5', '6', '7', '8','9',
'a', 'b', 'c', 'd', 'e', 'i', 'f'};

//class of square
class Square {
public:
    int posX, posY, width;
    bool isMain;
    Square(int x, int y, int w, bool isMain=false) {
        posX = x;
        posY = y;
        width = w;
        this->isMain = isMain;
    }
};

//class of desk
class Desk {
public:
    int minSquares = -1;
    int currentSquare = 1;
    std::stack<Square> minArrayOfSquare;
    std::stack<Square> arrayOfSquares;
    int countOfOperation = 0;
};
```

```
Square findMaxSquare(int k , int x, int y, int** matrix) {
    int width = 0;
    int startWith;
    // set max coords
    if (x > y) startWith = x;
    else startWith = y;
    //looking for maximum width of square
    for (int i = startWith; i < k; i++) {
        if (matrix[x + width][y + width] == 0 && matrix[x + width][y] == 0 &&
matrix[x][y + width] == 0) {
            width++;
        } else {
            break;
        }
    }
    return {x, y, width};
}

void fillMatrix(int** matrix, Square square, Desk* desk) {
    //update desk by new square
    for (int i = square.posX; i < square.posX + square.width; i++) {
        for (int j = square.posY; j < square.posY + square.width; j++) {
            matrix[i][j] = desk->currentSquare;
        }
    }
    desk->currentSquare++;
}

void backtracking(int k , int** matrix, Desk* desk, bool specialCase=false) {
        if (IS_SHOW_INTERMEDIATE_RESULTS) {
            //print desk
            std::cout<<"New desk\n";
            for (int i = 0; i < k; i++) {
                for (int j = 0; j < k; j++) {
                    std::cout<<matrix[i][j]<<" ";
                }
                std::cout<<"\n";
            }
            std::cout<<"\n";
        }
        for (int i = 0; i < k; i++) {
            for (int j = 0; j < k; j++) {
```

```
                // increment count of operations
                desk->countOfOperation++;
                // if ceil isn't empty than skip it
                if (matrix[i][j] != 0) {
                    continue;
                }
                // if current count of squares more than previous count
                if (desk->currentSquare >= desk->minSquares && desk->minSquares
!= -1) {
                    if (IS_SHOW_INTERMEDIATE_RESULTS) {
                        std::cout<<"Current count of square greater than
previous, that's why this desk will be update\n\n";
                    }
                    return;
                }
                //create new max square which can fit at position (i,j)
                Square square = findMaxSquare(k, i, j, matrix);
                // if k multiple of 2, 3 of 5
                if (specialCase) {
                    desk->minArrayOfSquare.push(square);
                }
                else {
                    desk->arrayOfSquares.push(square);
                }
                // fill new square
                fillMatrix(matrix, square, desk);
                if (IS_SHOW_INTERMEDIATE_RESULTS) {
                    std::cout<<"Square with coords ("<<i+1<<";"<<j+1<<") has
width "<<square.width<<" was set in desk\n";
                    //print desk
                    for (int i = 0; i < k; i++) {
                        for (int j = 0; j < k; j++) {
                            std::cout<<matrix[i][j]<<" ";
                        }
                        std::cout<<"\n";
                    }
                    std::cout<<"\n";
                }
            }
        }
```

```
}

bool decreaseSquare(int k , int **matrix, Desk* desk) {
    if (IS_SHOW_INTERMEDIATE_RESULTS) {
        std::cout<<"Start remove squares\n";
    }
    // change minimum count of squares
    if (desk->currentSquare < desk->minSquares || desk->minSquares == -1) {
        desk->minArrayOfSquare = desk->arrayOfSquares;
        desk->minSquares = desk->currentSquare;
    }

    //remove squares with side equal 1
    while (desk->arrayOfSquares.top().width == 1 ) {
        desk->countOfOperation++;
        Square top = desk->arrayOfSquares.top();
        matrix[top.posX][top.posY] = 0;
        if (IS_SHOW_INTERMEDIATE_RESULTS) {
            std::cout<<"Square  with  coords  ("<<top.posX+1<<";"<<top.posY+1<<")
has width "<<top.width<<" was removed\n";
            //print desk
            std::cout<<"Current desk\n";
            for (int i = 0; i < k; i++) {
                for (int j = 0; j < k; j++) {
                    std::cout<<matrix[i][j]<<" ";
                }
                std::cout<<"\n";
            }
            std::cout<<"\n";
        }
        desk->arrayOfSquares.pop();
        desk->currentSquare--;
    }


    Square top = desk->arrayOfSquares.top();
    // end of backtracking
    if (top.isMain) {
        return true;
    }
    // decrease square side by 1
    for (int i = 0; i < top.width; i++) {
```

```
        desk->countOfOperation++;
        matrix[top.posX + i][top.posY + top.width - 1] = 0;
        matrix[top.posX + top.width - 1][top.posY + i] = 0;


    }
    if (IS_SHOW_INTERMEDIATE_RESULTS) {
        std::cout<<"Square with coords ("<<top.posX+1<<";"<<top.posY+1<<") has
width "<<top.width<<" was decrement width\n";
        //print desk
        std::cout<<"Current desk\n";
        for (int i = 0; i < k; i++) {
            for (int j = 0; j < k; j++) {
                std::cout<<matrix[i][j]<<" ";
            }
            std::cout<<"\n";
        }
        std::cout<<"\n";
    }
    //decrement width of square
    desk->arrayOfSquares.top().width--;
    return false;

}


void printOptimalDesk(int** matrix, std::stack<Square> squares) {
    // print intermediate results
    if (IS_SHOW_INTERMEDIATE_RESULTS) {
        int numberOfSquare = 1;
        //print desk
        while (!squares.empty()) {
            Square top = squares.top();
            for (int i = top.posX; i < top.posX + top.width; i++) {
                for (int j = top.posY; j < top.posY + top.width; j++) {
                    matrix[i][j] = numberOfSquare;
                }
            }
            numberOfSquare++;
            squares.pop();
        }
    }
```

```
    //print answer
    else {
        int count = squares.size();
        std::cout<<count<<std::endl;
        //print coords of squares
        while (!squares.empty()) {
            std::cout<<squares.top().posX + 1<<" "<<squares.top().posY + 1<<"
"<<squares.top().width<<std::endl;
            squares.pop();
        }
    }

}

void prepare(int k, int**matrix) {
    Desk desk = Desk();

    // even side
    if (k % 2 == 0) {
        std::cout<<"k % 2 == 0 => min count of square = 4 and side of this
squares = k / 2\n";
        Square square(0, 0, k / 2);
        desk.minArrayOfSquare.push(square);
        fillMatrix(matrix, square, &desk);
        backtracking(k, matrix, &desk, true);
    }
    // side multiple of 3
    else if (k % 3 == 0) {
        std::cout<<"k % 3 == 0 => min count of square = 3 and side of the
largest square = k / 3 * 2\n";
        Square square(0, 0, k / 3 * 2);
        desk.minArrayOfSquare.push(square);
        fillMatrix(matrix, square, &desk);
        backtracking(k, matrix, &desk, true);
    }
    //side multiple of 5
    else if (k % 5 == 0) {
        std::cout<<"k % 5 == 0 => min count of square = 5 and side of the
largest square = k / 5 * 3\n";
        Square square(0, 0, k / 5 * 3);
        desk.minArrayOfSquare.push(square);
        fillMatrix(matrix, square, &desk);
```

```
        backtracking(k, matrix, &desk, true);
    }
    //otherwise
    else {
        //create 3 main squares
        Square squareMax(0,0, k / 2 + 1, true);
        Square squareMin1(0, k / 2 + 1, k / 2, true);
        Square squareMin2(k / 2 + 1, 0, k / 2, true);
        desk.arrayOfSquares.push(squareMax);
        fillMatrix(matrix, squareMax, &desk);
        desk.arrayOfSquares.push(squareMin1);
        fillMatrix(matrix, squareMin1, &desk);
        desk.arrayOfSquares.push(squareMin2);
        fillMatrix(matrix, squareMin2, &desk);
        //enumeration of all options
        while (true) {
            backtracking(k, matrix, &desk);
            if(decreaseSquare(k, matrix, &desk)) {
                break;
            }
        }
    }


    //print result
    printOptimalDesk(matrix, desk.minArrayOfSquare);
    // print desk if IS_SHOW_INTERMEDIATE_RESULTS = true
    if (IS_SHOW_INTERMEDIATE_RESULTS) {
        std::cout<<"Optimal desk:\n";
        for (int i = 0; i < k; i++) {
            for (int j = 0; j < k; j++) {
                std::cout<<namesOfSquares[matrix[i][j]]<<" ";
            }
            std::cout<<"\n";
        }

    }

    std::cout<<"Count of operation with side square equal "<<k<<" -
"<<desk.countOfOperation + k * k<<std::endl;
    desk.countOfOperation = 0;
}
```

```
int main() {
    int k;
    std::cin>>k;
    //create matrix
    int **matrix = new int*[k];
    for (int i = 0; i < k; i++) {
        matrix[i] = new int[k];
    }

    //fill matrix
    for (int i = 0; i < k; i++) {
        for (int j = 0; j < k; j++) {
            matrix[i][j] = 0;
        }
    }
    //start backtracking
    prepare(k, matrix);

    //free mem
    for (int i = 0; i < k; i++) {
        delete matrix[i];
    }
    delete[] matrix;

    return 0;
}
```