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	____________________	Балаева М.О.
Преподаватель	____________________	Фирсов М.А.

Санкт-Петербург
2021

Цель работы.

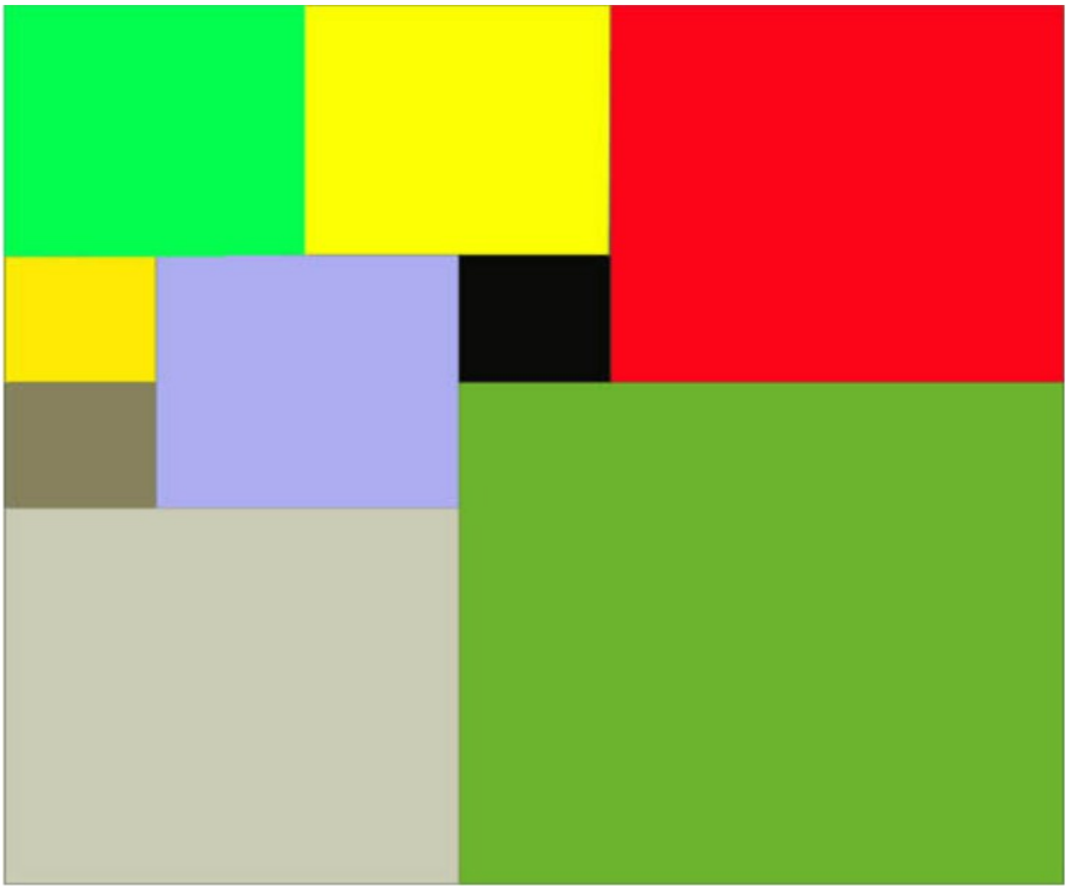
Реализовать программу, основанную на рекурсивном бэктрекинге. Исследовать время выполнения алгоритма от параметра, прописанного в задании., проследить зависимость количества операций для решения поставленной задачи от входных данных.

Задание.

Вар. 2р. Рекурсивный бэктрекинг. Исследование времени выполнения от размера квадрата.

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).

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

Описание алгоритма.

Создается матрица N на N, на которой натуральными числами отмечается где и какой по счету был поставлен квадрат. Алгоритм находит свободное место для вставки квадрата и рекурсивно ставит туда квадраты всех возможных размеров. Если карта оказывается заполненной, то количество квадратов на ней сравнивается с найденным лучшим разбиением, и если новое разбиение лучше, то заменяется.

Использованные оптимизации.

- Матрицу перебора изначально можно заполнить на 75% тремя квадратами размеров N//2 , N//2 -1 соответственно, то поиск свободной клетки, куда можно поставить квадрат, можно осуществлять только в оставшихся 25% квадрата.
- Квадрат с четной стороной имеет постоянное решение – 4 квадрата. Для квадратов наименьший простой множитель, которых равен трем, не производится перебор , а сразу выводится ответ 6. Поэтому можно не осуществлять перебор для таких квадратов, а сразу выводить ответ.
- Сжатие квадрата. Квадрат с размером N, можно сжать до размера значения наименьшего простого делителя числа N.
- Поскольку 75% квадрата заполнены, то максимальный размер квадрата, который можно поставить в матрицу перебора – N // 2.

Описание функций и структур данных.

Класс Table – класс, предназначенный для выполнения поставленной задачи. Поля класса:

1. size – длина стороны квадрата.
2. std::vector<std::vector<int>> table – матрица квадрата.
3. Count – переменная , показывающая количество “вложенных” квадратов.

void constTable() - Метод , отвечающий за вставку трех “гарантированных” квадратов.

int getnumber() - метод, возвращающий количество расположенных “вложенных” квадратов.

int getsize() - метод, возвращающий длину стороны квадрата

bool isPossible(int i, int j, int n) – метод, показывающий можно ли разместить еще один квадрат.

Аргументы:

1. i – координата по y.
2. j – координата по x.
3. n – длина рассматриваемого квадрата.

Метод возвращает истину или ложь.

void insertTable(int i, int j, int n) – Метод , наносящий квадрат на карту, также считает количество “вложенных” квадратов.

Аргументы:

1. i – координата по y.
2. j – координата по x.
3. n – длина рассматриваемого квадрата.

bool checkSpace(int i) – метод , показывающий есть ли на карте еще свободные места. Возвращает истину или ложь.

Аргументы:

1. i – координата по y.

int findi(int i) - метод , возвращающий координату по y.

Аргументы:

1. i – координата по y.

int findj(int i) - метод , возвращающий координату по x.

Аргументы:

1. i – координата по y

void deleteTable(int i, int j) — метод , удаляющий(«зануляющий») матрицу.

void result() - метод , выводящий результат.

Table backTracking(Table table, int i, int j) — рекурсивная функция , находящая с помощью вышеописанных методов минимально возможное число «вложенных» квадратов. Функция возвращает экземпляр класса Table.

Аргументы:

1. Table table — экземпляр класса Table.
2. int i — координата по y.
3. int j — координата по x.

В main() производится проверка выделенных случаев(наименьшие делители 2 и 3 соответственно), а также вызов всех необходимых функций и методов.

Оценка сложности алгоритма по времени.

Поскольку используется довольно большое количество оптимизаций, посчитать точную сложность алгоритма сложно, поэтому произведем оценку алгоритма сверху.

N – длина стороны квадрата. Имеется N^2 свободных клеток, также N размеров квадрата, которые будем перебирать. Таким образом , получаем , что сложность алгоритма по времени равна O((N^2)! * N^N).

Оценка сложности алгоритма по памяти.

Матрица квадрата , хранящаяся в экземпляре класса Table , при каждом рекурсивном проходе копируется, поэтому мы возьмем максимальное количество единичных квадратов в матрице, оно равняется N*N и умножается на количество рекурсивных проходов. В процессе рекурсивного прохода, скопированные экземпляры класса удаляются , поэтому за максимум можно считать проход по матрице — N*N. Следовательно сложность алгоритма по памяти — O(N^4).

Тестирование.

Таблица 1. Результаты работы программы

№ попытки	Входные данные	Выходные данные без промежуточного вывода
1	3	6 1 1 2 1 3 1 2 3 1 3 1 1 3 2 1 3 3 1
2	5	8 1 1 3 1 4 2 3 4 1 3 5 1 4 1 2 4 3 1 4 4 2 5 3 1
3	2	4 1 1 1 2 1 1 1 2 1 2 2 1
4	9	6 1 1 6 1 7 3 4 7 3 7 1 3 7 4 3 7 7 3
5	11	11 1 1 6 1 7 5

		6 7 1 6 8 1 6 9 3 7 1 5 7 6 1 7 7 2 8 6 1 9 6 3 9 9 3
6	12	4 1 1 6 7 1 6 1 7 6 7 7 6
7	19	13 1 1 10 1 11 9 10 11 1 10 12 1 10 13 2 10 15 5 11 1 9 11 10 2 11 12 1 12 12 3 13 10 2 15 10 5 15 15 5

Исследование.

В данном варианте необходимо исследовать зависимость времени от размера квадрата, чтобы это сделать посчитаем время выполнения алгоритма для каждой длины стороны квадрата(от 2 до 20) .

Результаты времени выполнения алгоритма от размера главного квадрата представлены в Таблице 2.

Таблица 2. Зависимость времени от размера квадрата.

Длина стороны квадрата(N)	Время (c)
2	0.000157
3	0.000114
4	0.000199
5	0.000281
6	0.000123
7	0.00301
8	0.000134
9	0.000182
10	0.000118
11	0.049306
12	0.000139
13	0.100369
14	0.000153
15	0.000176
16	0.000137
17	0.911326
18	0.000103
19	3.10129
20	0.000127

Исходя из графика , можно сделать вывод, что из-за оптимизаций, время выполнения программы сокращается. Время выполнения программы при нечетных значениях растет экспоненциально, что видно из графика.

Выводы.

В ходе работы были изучены методы бэктрекинга, написана программа для поиска минимального количества квадратов для заполнения заданного с помощью рекурсивного бэктрекинга, практически освоены решения по возможным оптимизациям и исследована зависимость времени выполнения алгоритма от размера квадрата.

ПРИЛОЖЕНИЕ А
ИСХОДНЫЙ КОД ПРОГРАММЫ

Название файла: main.cpp

```
#include <iostream>
#include <vector>
#include <ctime>

int bestNum = 0;

class Table // исходный квадрат
{
    int size;
    std::vector<std::vector<int>> table;
    int count;

public:
```

```
    Table(int size): size(size), table(size, std::vector<int>(size, 0)),
count(0) {
        if(size != 0) {
            constTable();
        }
    }
    Table(Table const &other): size(other.size), table(other.size,
std::vector<int>(other.size, 0)), count(other.count) {
        for (int i = 0; i < size; i++)
            for (int j = 0; j < size; j++)
                table[i][j] = other.table[i][j];
    }
    Table& operator=(Table const &other){
        if(&other != this){
            Table tmp(other);
            count = tmp.count;
            size = tmp.size;
            table.swap(tmp.table);
        }
        return *this;
    }
    ~Table(){}

    void constTable(){ //Вставка трех "гарантированных " квадратов
        int temp = size/2;
        insertTable(0, 0, temp + 1);
        insertTable(0, temp + 1, temp);
        insertTable(temp + 1, 0, temp);
    }

    int getnumber(){
        return count;
    }
    int getsize(){
        return size;
    }

    bool isPossible(int i, int j, int n){
        if((i + n) > size || (j + n) > size){
            return false;
        }
        for(int y = i; y < i + n; y++)
            for (int x = j; x < j + n; x++)
                if(table[y][x] != 0){
```

```
                return false;
            }
    return true;
}
void insertTable(int i, int j, int n){
    for(int y = i; y < i + n; y++){
        for (int x = j; x < j + n; x++){
            table[y][x] = n;
        }
    }
    ++count;
}
bool checkSpace(int i){
    for(int y = i; y < size; y++)
        for (int x = 0; x < size; x++)
            if(table[y][x] == 0)
                return true;
    return false;
}
int findi(int i){
    for (int y = i; y < size; y++)
        for (int x = 0; x < size; x++)
            if (table[y][x] == 0){
                return y;
            }
}
int findj(int i){
    for (int y = i; y < size; y++)
        for (int x = 0; x < size; x++)
            if (table[y][x] == 0){
                return x;
            }
}
void deleteTable(int i, int j){
    int val = table[i][j];
    for (int y = i; y < i + val; y++)
        for (int x = j; x < j + val; x++)
            table[y][x] = 0;
}
void result(){
    for(int i = 0; i < size; i++){
        for(int j = 0; j < size; j++){
            if(table[i][j] != 0){
```

```
                    std::cout << i + 1 << " " << j + 1 << " " << table[i][j] <<
std::endl;
                    deleteTable(i, j);
                }
            }
        }
    }

};

Table best(0);

Table backTracking(Table table, int i, int j){
    for(int n = table.getsize() / 2; n > 0; n--){
        if(table.getnumber() > bestNum){
            return table;
        }

        Table shape = table;
        if(shape.isPossible(i, j, n)){
            shape.insertTable(i, j, n);
            if(shape.checkSpace(i)){
                shape = backTracking(shape, shape.findi(i), shape.findj(i));
            }
            else if(bestNum >= shape.getnumber()){
                best = shape;
                bestNum = shape.getnumber();
            }
        }
    }
    return table;
}

int main(){
    int size = 0;
    std::cin >> size;
    clock_t start = clock();
    if (size%2 == 0){
        int temp = size/2;
        std::cout << "4" << std::endl;
        std::cout << "1 1 " << temp << std::endl;
        std::cout << temp+1 << " 1 " << temp << std::endl;
        std::cout << "1 " << temp+1 << " " << temp << std::endl;
        std::cout << temp+1 << " " << temp+1 << " "<< temp << std::endl;
```

```
    }
    else if (size%3 == 0){
        int temp = size/3;
        std::cout << "6" << std::endl;
        std::cout << "1" << " 1 " << temp*2 <<  std::endl;
        std::cout << "1 " << 1+temp*2 << " " << temp <<  std::endl;
        std::cout << 1+temp << " " << 1+temp*2 << " " << temp <<  std::endl;
        std::cout << 1+temp*2 << " 1 "  << temp <<  std::endl;
        std::cout << 1+temp*2 << " " << 1+temp << " "  << temp <<  std::endl;
        std::cout << 1+temp*2 << " " << 1+temp*2 << " " << temp << std::endl;

    }
    else {
        bestNum = size * size;
        Table A(size);
        A = backTracking(A, A.findi(0), A.findj(0));
        std::cout << bestNum << std::endl;
        best.result();
    }
    clock_t end = clock();
    std::cout << "Время выполнения: " << (double) (end - start) / CLOCKS_PER_SEC
<< "\n";
return 0;
}
```