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. 8304 _____ Мухин А. М.

Преподаватель _____ Размочаева Н.

Санкт-Петербург
2020

Цель работы.

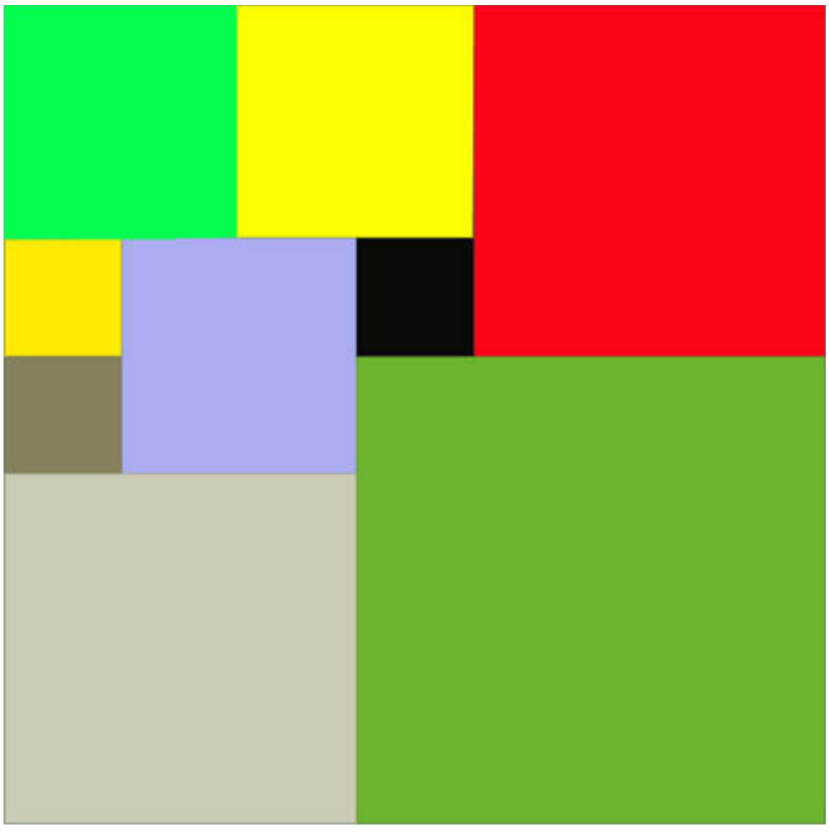
Научиться использовать алгоритм бэктрекинга, чтобы замостить квадратную столешницу заданной длины наименьшим числом квадратов.

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≤N≤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.

Алгоритм принимает длину стороны столешницы. Далее, если длина является составным числом, то замощение происходит по уже написанному шаблону для случаев кратности двум, трём, пяти. Если число простое, то используется рекурсивный алгоритм поиска с возвратом. В столешнице находится дырка, которая заполняется наименьшим квадратом и после рекурсивно вызывается эта же функция на этом же участке. Как только дырка

будет найдена в новой строке, функция будет вызываться рекурсивно с той же координатой x и новой y. Когда дырок не обнаружено, конфигурация замощения квадратами сохраняется в массиве, если она лучше предыдущей конфигурации. На обратном ходе последний вставленный квадрат удаляется и на его место, если возможно, устанавливается квадрат с большей стороной.

Сложность алгоритма по операциям:

$O(e^N)$

Сложность алгоритма по памяти:

$O(N^2)$

Описание функций и структур данных.

1.
```
struct Point {
    int size = 0;
    int number = 0;
};
```

Представляет собой точку на столешнице размера 1x1, в поле size которой хранится длина стороны квадрата, который покрывает эту точку, а в поле number которой хранится порядковый номер квадрата, который покрыл эту точку.

2.
```
class Square
    int N = 0;
    Point** matrix = nullptr;
    Point** bestMatrix = nullptr;
    int current_count = 0;
    int min_count = 0;
```

Описывает столешницу размера N*N, с текущим замощением в массиве точек matrix, лучшим замощением в массиве точек bestMatrix, текущим числом квадратов на столешнице current_count и наименьшим числом квадратов min_count в лучшей конфигурации.

3. `explicit Square(int N) : N(N), current_count(0), min_count(1600), matrix(new Point* [N]), bestMatrix(new Point* [N]);`

Конструктор класса столешницы. Предназначен для инициализации полей экземпляра. Принимает размер столешницы, записывая его в поле N. Текущее число вставленных квадратов current_count равно нулю, в лучшей конфигурации число квадратов min_count условно принимается как количество единичных квадратов в поле 40x40, для двумерного массива точек matrix выделяется память и он инициализируется нулями, для bestMatrix тоже выделяется память.

4. `void fill();`

Основной метод класса, вызывающийся в main. Не принимает аргументов. Предназначен для заполнения столешницы квадратами. Если длина столешницы – составное число, то происходит шаблонное замощение квадратами, если нет, то в верхний левый угол размещается квадрат со стороной N/2+1, а в правый верхний и левый нижний угол квадраты со сторонами N/2. Далее вызывается функция бэктрекинга для замощения оставшихся дырок.

5. `bool isFreePointOnMatrix(int &x, int &y, int xLimit, int yLimit);`

Функция принимает ссылки на координаты для записи в них координаты дырки в столешнице, поиск которой осуществляется в прямоугольнике с левым верхним углом в xLimit, yLimit и нижним правым углом в нижнем правом углу столешницы. Если дырка найдена возвращает истину, иначе ложь.

8. `void setSquare(int x, int y, int size);`

Функция принимает координаты левого верхнего угла квадрата и длину его стороны. Предназначена для замощения столешницы квадратом с указанными атрибутами.

9. `void deleteSquare(int x, int y, int size);`

Функция принимает координаты левого верхнего угла квадрата и длину его стороны. Предназначена для удаления со столешницы квадрата с указанными атрибутами.

10. `bool isHasFreeSpace(int x, int y, int size);`

Функция принимает координаты левого верхнего угла квадрата и длину его стороны. Предназначена для проверки, уместится ли квадрат с указанными атрибутами в столешницу. Если квадрат можно вставить, функция возвращает истину и вставляет квадрат, если нет, возвращает ложь.

11. `void setBestConfiguration();`

Функция проверяет, является ли существующая на данном этапе бэктрекинга конфигурация лучше, чем сохранённая. Если это так, точки Point матрицы bestMatrix перебиваются точками матрицы matrix.

12. `void squaring(int xLimit, int yLimit, int size);`

Рекурсивная функция принимает верхний левый угол квадрата длины size для замощения его квадратами всеми возможными вариантами. Является функцией поиска с возвратом. Если в заданном квадрате есть пустая точка, в неё удаётся вписать квадрат минимальной длины, то рекурсивно вызывается эта же функция с теми же аргументами. Как только поиск дырки в столешнице переходит на новую строку, эта функция начинает рекурсивно вызываться с тем же постоянным аргументом x, но увеличенной переменной y. При обратном ходе происходят попытки вставить в дырки квадраты размером побольше. Когда дырок уже нет, конфигурация проверяется на оптимальную.

13. `int main();`

Функция возвращает код ошибки 0. В теле в цикле от 2 до 40 создаётся экземпляр столешницы размера N, введённого пользователем. Вызывается метод замощения и замеряется время работы функции.

Способ хранения частичных решений.

Частичные решения, т.е. конфигурации замощения столешницы, которые ещё не покрывают её всю, хранятся в двумерном массиве matrix типа Point**.

Использованные оптимизации алгоритма.

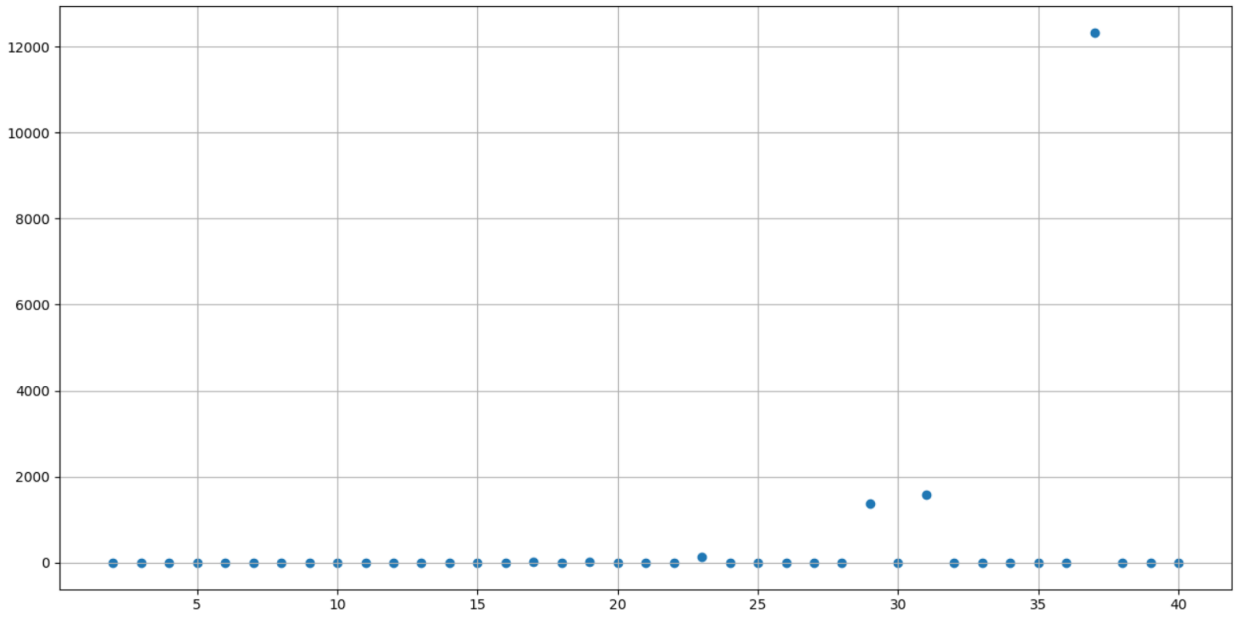
Если на данном шаге поиска с возвратом в столешнице квадратов больше или равно числу квадратов в лучшей конфигурации, то такая конфигурация не рассматривается.

```
if (current_count >= min_count) {
        return;
}
```

Исходная столешница изначально заполняется квадратом со стороной N/2+1 в левом верхнем углу и квадратами со стороной N/2 в левом нижнем и правом верхнем углах.

Тестирование.

На языке программирования Python 3 была написана программа, считывающая из файлов вида i.txt, где i = [2, 40], значения времени выполнения нашей программы и рисующая график зависимости (времени выполнения программы, 10^{-3} с от размера стороны квадрата) представленный ниже.



Выводы.

Были получены навыки по использованию бэктрекинга. Написана программа по замещению квадратной столешницы заданной длины наименьшим числом квадратов. А также благодаря полученной визуально зависимости между длинной стороны квадрата и временем выполнения программы, можно сделать вывод о том, что применённые к алгоритму оптимизации корректно отработали и практически свели время выполнения программы на сторонах больших 5, к времени выполнения задач в частных случаях.

ПРИЛОЖЕНИЕ А.
ИСХОДНЫЙ КОД

```
#include <iostream>
#include "chrono"
#include "fstream"

struct Point {
    int size = 0;
    int number = 0;
};

class Square {
    int N = 0;
    Point** matrix = nullptr;
    Point** bestMatrix = nullptr;
    int current_count = 0;
    int min_count = 0;

public:
    explicit Square(int N) : N(N), current_count(0), min_count(1600),
matrix(new Point* [N]), bestMatrix(new Point* [N]) {

        for (int i = 0; i < N; i++) {
            matrix[i] = new Point [N];
            bestMatrix[i] = new Point [N];

            for (int j = 0; j < N; j++){
                matrix[i][j].size = 0;
                matrix[i][j].number = 0;
            }
        }
    }

    void fill() {
        std::cout << "N: " << N << std::endl;
        if(!(N % 2) || !(N % 3) || !(N % 5)) {
            if (!(N % 2)) {
                int N1 = N / 2;
                setSquare(0, 0, N1);
                setSquare(N1, 0, N1);
                setSquare(0, N1, N1);
                setSquare(N1, N1, N1);
            }
            else if (!(N % 3)) {
                int N1 = N / 3;
                setSquare(0, 0, 2 * N1);
                setSquare(2 * N1, 0, N1);
                setSquare(0, 2 * N1, N1);
                setSquare(2 * N1, N1, N1);
                setSquare(N1, 2 * N1, N1);
                setSquare(2 * N1, 2 * N1, N1);
            }
            else if (!(N % 5)) {
                int N1 = N / 5;
                setSquare(0, 0, 3 * N1);
                setSquare(3 * N1, 0, 2 * N1);
```

```
                setSquare(3 * N1, 3 * N1, 2 * N1);
                setSquare(0, 3 * N1, 2 * N1);
                setSquare(2 * N1, 3 * N1, N1);
                setSquare(2 * N1, 4 * N1, N1);
                setSquare(3 * N1, 2 * N1, N1);
                setSquare(4 * N1, 2 * N1, N1);
            }
            setBestConfiguration();
            return;
        }
        setSquare(0, 0, N / 2 + 1);
        setSquare(N / 2 + 1, 0, N / 2);
        setSquare(0, N / 2 + 1, N / 2);

        squaring(N / 2, N / 2, N / 2);
    }

private:

    bool isFreePointOnMatrix(int &x, int &y, int xLimit, int yLimit) {
        for (int i = yLimit; i < N; i++) {
            for (int j = xLimit; j < N; j++) {
                if (matrix[i][j].size == 0) {
                    x = j;
                    y = i;
                    return true;
                }
            }
        }
        return false;
    }

    void setSquare(int x, int y, int size) {
        current_count++;
        for (int i = y; i < y + size; i++)
            for (int j = x; j < x + size; j++) {
                matrix[i][j].size = size;
                matrix[i][j].number = current_count;

            }
    }

    void deleteSquare(int x, int y, int size) {
        for (int i = y; i < y + size; i++)
            for (int j = x; j < x + size; j++) {
                matrix[i][j].size = 0;
                matrix[i][j].number = 0;
            }
        current_count--;
    }

    bool isHasFreeSpace(int x, int y, int size) {
        if ((x + size > N) || (y + size > N))
            return false;
        for (int i = y; i < y + size; i++) {
            for (int j = x; j < x + size; j++) {
                if (matrix[i][j].size != 0) {
```

```
                    return false;
                }
            }
        }
        setSquare(x, y, size);
        return true;
    }

    void setBestConfiguration() {
        if (current_count < min_count) {
            for (int i = 0; i < N; i++){
                for (int j = 0; j < N; j++){
                    bestMatrix[i][j] = matrix[i][j];
                }
            }
            min_count = current_count;
        }
    }

    void squaring(int xLimit, int yLimit, int size) {
        if (current_count >= min_count) {
            return;
        }
        int x = 0;
        int y = 0;

        if (isFreePointOnMatrix(x, y, xLimit, yLimit)) {
            for (int possibleSize = 1; possibleSize < size;
possibleSize++) {
                if (isHasFreeSpace(x, y, possibleSize)) {
                    squaring(xLimit, y, size);
                    deleteSquare(x, y, possibleSize);
                }
            }
            return;
        }
        setBestConfiguration();
    }
};

int main() {
    for (int i = 2; i < 41; i++) {
        std::ofstream output_data(std::to_string(i) + ".txt",
std::ios::trunc | std::ios::out);
        auto start = std::chrono::high_resolution_clock::now();
        Square square(i);
        square.fill();
        auto end = std::chrono::high_resolution_clock::now();
        auto duration =
std::chrono::duration_cast<std::chrono::microseconds>(end - start);
        output_data << duration.count();
        output_data.close();
    }
    return 0;
}
```

ПРИЛОЖЕНИЕ Б.
КОД ПРОГРАММЫ ДЛЯ ПОСТРОЕНИЯ ГРАФИКА ЗАВИСИМОСТИ НА ЯЗЫКЕ PYTHON 3

```
import matplotlib.pyplot as plt

points_x = []
points_y = []

for i in range(2, 41):
    with open(f"{i}.txt") as input_file:
        points_x.append(i)
        points_y.append(int(input_file.readline()) / 1000)

points_x, points_y = zip(*sorted(zip(points_x, points_y)))

fig, axs = plt.subplots()
axs.scatter(points_x, points_y)
plt.grid()
plt.show()
```