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

Студентка гр. 1304 Чернякова А.Д.

Преподаватель Шевелева А.М.

Санкт-Петербург
2023

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 (см. Рисунок 1 - Пример столешницы 7×7).

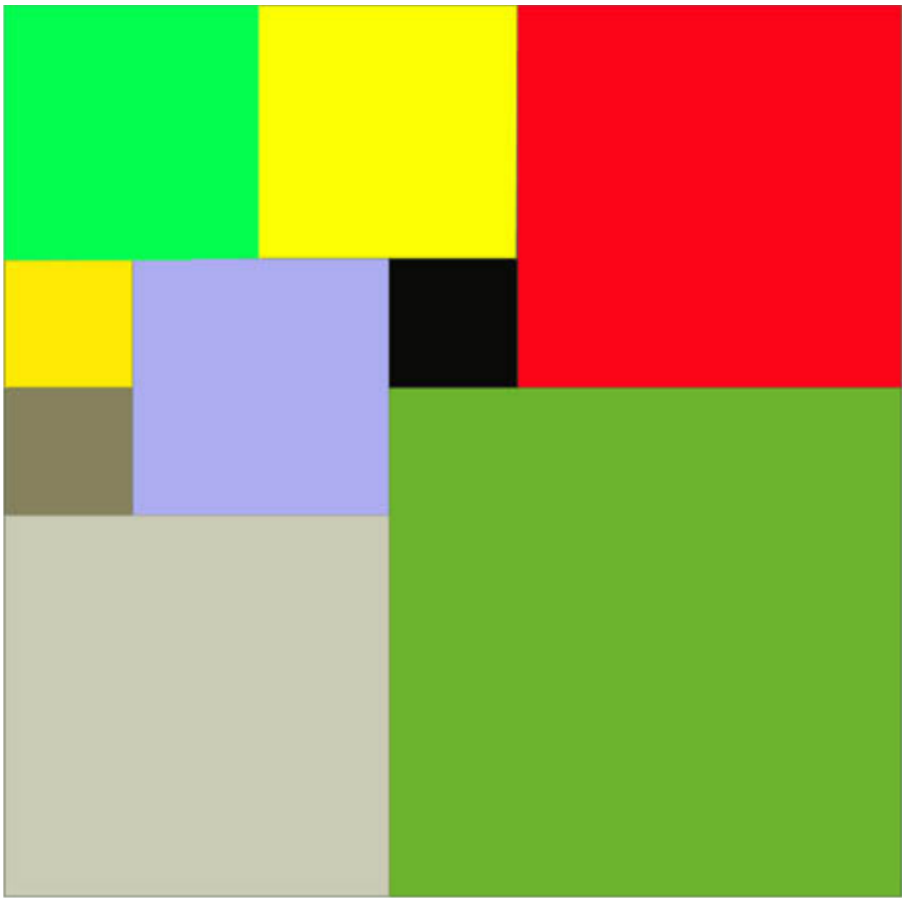

Рисунок 1 - Пример столешницы 7×7

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.

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).

Выполнение работы.

Рассматривается задача, где *N* не более 20.

Создан класс столешницы *MapSquare*, на котором базируется вся задача.

При инициализации класса методом *__init__(self, n)*, на вход принимается сторона столешницы n и создается 5 полей класса: *self.n* - сторона столешницы, *self.free_s* - свободная площадь столешницы (изначально она не заполнена, поэтому *self.free_s = n*n*), *self.map_square* - двумерный массив для хранения заполненности столешницы (0 - свободно, каждый вставленный квадрат имеет свою цифру, которой и заполняет данное поле), *self.best_map* - список для хранения координат левого угла и длины стороны вставленных квадратов (изначально пуст), *self.best_square_counter* - наименьшее количество квадратов для заполнения столешницы (так как рассматриваемая задача до 20, то наихудший случай 20x20 = 400).

Метод *best_start(self)* - оптимизация задачи, а именно лучшая начальная вставка трех квадратов в правый нижний угол и смежных с ним. С помощью двух вложенных циклов for происходит заполнение большого квадрата со стороной *(self.n+1) // 2*: в *self.map_square* заполняются позиции единицами. Аналогично происходит заполнение для двух смежных квадратов со сторонами *(self.n + 1) // 2 - 1*, только заполняют *self.map_square* соответственно цифрами 2 и 3.

Метод *can_insert(map_sq, insert_n, x, y)* принимает на вход двумерный массив столешницы *map_sq*, сторону квадрата на вставку *insert_n* и координаты левого верхнего угла *x* и *y* и с помощью двух циклов for проверяет возможно ли вставить квадрат со стороной *insert_n* на позицию *x y*. Если возможно возвращает True, в противном случае False.

Метод *insert_square(map_sq, insert_n, numb, x, y)* принимает на вход двумерный массив столешницы *map_sq*, сторону квадрата на вставку *insert_n,* номер вставляемого квадрата *numb* и координаты левого верхнего угла *x* и *y,* создает копию столешницы *map_square* с помощью функции deepcopy и с помощью двух циклов for заполняет новый массив столешницы на соответствующие позиции числом *numb*. Возвращает заполненную копию массива *new_map*.

Метод *solve(self)* - основной метод, вызываемый из main. Здесь рассматривается 3 случая. Первый частный случай - сторона столешницы является четным числом. Здесь всегда происходит разбиение столешницы на 4 равных квадрата, *self.best_square_counter* становится равным 4, *self.best_map* заполняется квадратами с параметрами *[0, 0, self.n // 2], [self.n // 2, 0, self.n // 2], [0, self.n // 2, self.n // 2], [self.n // 2, self.n // 2, self.n // 2]*. Происходит выход из функции. Второй частный случай - сторона столешницы является составным числом кратным 3. На практике в ходе исследования ясно, что столешницу 3x3 наилучшим образом можно разбить на 6 квадратов и все кратные 3 числа также разбиваются на 6 квадратов, но каждый из параметров вставляемых квадратов будет умножен на коэффициент пропорциональности. Рассчитывается коэффициент пропорциональности *k = self.n // 3, self.best_square_counter* равно 6, *self.best_map* заполняется квадратами с параметрами *[k, k, k * 2], [k * 2, 0, k], [0, k * 2, k], [0, 0, k], [0, k, k], [k, 0, k]* и происходит выход из функции. Эти два частных случаях помогают оптимизировать задачу. Третий случай для оставшихся чисел, которые не подошли в предыдущие два частных случая - перебор возможных комбинаций. Изначально вызывается метод

self.best_start(), описанный выше, создается переменная *counter* - счетчик вставленных квадратов (изначально ей присваивается значение 3, так как метод *self.best_start()* уже вставил 3 первых оптимальных квадрата), переменная *map_square*, которая является копией *self.map_square*, переменная *free_s*, равная разности свободной площади и трех вставленных квадратов, и переменная *best_map*, являющаяся копией *self.best_map.* После создания новых переменных *(map_square, free_s, counter, best_map)* они передаются в рекурсивную функцию поиска наилучшего расположения квадратов *self.find_square(map_square, free_s, counter, best_map).*

Метод *find_square(self, map_square, free_s, counter, best_map)* - рекурсивный метод поиска наилучшего расположения квадратов, принимает на вход двумерный массив столешницы *map_square*, свободную площадь *free_s*, счетчик вставленных квадратов counter и массив с квадратами для наилучшего расположения *best_map*. В самом начале функции обеспечивается выход из рекурсии, если *counter >= self.best_square_counter*. Далее реализация функции представляет собой тройной цикл for: первый осуществляет перебор строк, второй - перебор столбцов, третий - перебор размеров квадратов на вставку. Важно, что здесь перебирается только около четверти квадрата, так как три квадрата уже вставлено, что значительно уменьшает время работы программы. Происходит проверка на вставку, что нынешняя позиция равна 0 и что метод *self.can_insert(map_square, insert_n, w, h)* возвращает *True*. Если вставка возможна, то создается новая переменная столешницы *new_map_square* методом *self.insert_square(map_square, insert_n, counter + 1, w, h)*, переменная *new_result*, являющаяся копией *best_map* и в *new_result* добавляется соответствующие параметры вставляемого квадрата. Далее происходит проверка заполнен ли квадрат, если нет, то функция *self.find_square(new_map_square, free_s - insert_n * insert_n, counter + 1, new_result)* вызывается снова, а затем идет проверка на повторение уже проверенных расстановок и происходит выход из функции. Если же столешница полностью заполнилась, то происходит проверка на наилучшее

расположение и количество квадратов, в случае необходимости изменяются *self.map_square, self.best_square_counter, self.best_map;* происходит выход из функции. После проверки на заполненность столешницы происходит проверка не равна ли сторона вставляемого квадрата 1: если равна, то также происходит выход из функции.

Метод *__str__(self)* реализует требуемый вывод: количество квадратов и параметры каждого из них в требуемом формате. Координаты при выводе увеличиваются на 1, так как нумерация массива идет с 0, а в задаче с 1.

При вызове *main()* считывается длина стороны столешницы *n*, создается объект *map_square* класса *MapSquare()* и вызывается функция *solve()* у данного объекта, а после объект печатается на экран.

Выводы.

В ходе выполнения работы был изучен и реализован на практике алгоритм «Поиск с возвратом». Он позволяет решать задачи, где необходимо перебирать все возможные варианты. Решение задачи методом поиска с возвратом сводится к последовательному расширению частичного решения. Если на очередном шаге такое расширение провести не удается, то он возвращается к более короткому частичному решению и продолжает поиск дальше. Этот алгоритм успешно использовался в данной задачи, но без оптимизаций программа работала бы бесконечно долго. Для ускорения были рассмотрены частные случаи: четная сторона столешница или кратная 3, оптимальная начальная вставка первых трех квадратов, а также контроль текущего количества вставленных квадратов. Таким образом программа прошла тест на платформе Степик и уложилась в 3 секунды.

ПРИЛОЖЕНИЕ А
ИСХОДНЫЙ КОД

Название файла: *lab1.py*

```
FROM COPY IMPORT DEEPCOPY


CLASS MAPSQUARE:  # ОСНОВНОЙ КЛАСС СТОЛЕШНИЦЫ, НА КОТОРОМ БАЗИРУЕТСЯ ВСЯ ЗАДАЧА
    DEF __INIT__(SELF, N):
        SELF.N = N  # СТОРОНА СТОЛЕШНИЦЫ
        SELF.FREE_S = SELF.N ** 2  # СВОБОДНАЯ ПЛОЩАДЬ СТОЛЕШНИЦЫ, ИЗНАЧАЛЬНО СВОБОДНА
ВСЯ
        SELF.MAP_SQUARE = [[0] * N FOR _ IN RANGE(N)]  # ДВУМЕРНЫЙ МАССИВ ДЛЯ ХРАНЕНИЯ
ЗАПОЛНЕННОСТИ СТОЛЕШНИЦЫ (0 - СВОБОДНО)
        SELF.BEST_MAP = []  # СПИСОК ДЛЯ ХРАНЕНИЯ КООРДИНАТ ЛЕВОГО УГЛА И ДЛИНЫ СТОРОНЫ
ВСТАВЛЕННЫХ КВАДРАТОВ
        SELF.BEST_SQUARE_COUNTER = 400  # НАИМЕНЬШЕЕ КОЛИЧЕСТВО КВАДРАТОВ ДЛЯ
ЗАПОЛНЕНИЯ(ТАК КАК РАССМАТРИВАЕТСЯ ЗАДАЧА ДО 20, ТО НАИХУДШИЙ ВАРИАНТ 20*20=400)


    DEF BEST_START(SELF):  # ОПТИМИЗАЦИЯ: ЛУЧШАЯ НАЧАЛЬНАЯ ВСТАВКА ТРЕХ КВАДРАТОВ В ПРАВЫЙ
НИЖНИЙ УГОЛ И СМЕЖНЫХ С НИМ
        FOR H IN RANGE(SELF.N // 2, SELF.N):  # ВСТАВКА БОЛЬШОГО КВАДРАТА СО СТОРОНОЙ
(N + 1) // 2
            FOR W IN RANGE(SELF.N // 2, SELF.N):
                SELF.MAP_SQUARE[H][W] = 1
        SELF.BEST_MAP.APPEND([SELF.N // 2, SELF.N // 2, (SELF.N + 1) // 2])


        FOR H IN RANGE(SELF.N // 2):  # ВСТАВКА СМЕЖНОГО КВАДРАТА СО СТОРОНОЙ (N + 1)
// 2 - 1
            FOR W IN RANGE(SELF.N // 2 + 1, SELF.N):
                SELF.MAP_SQUARE[H][W] = 2
        SELF.BEST_MAP.APPEND([0, SELF.N // 2 + 1, (SELF.N + 1) // 2 - 1])


        FOR H IN RANGE(SELF.N // 2 + 1, SELF.N):  # ВСТАВКА ЕЩЕ ОДНОГО СМЕЖНОГО
КВАДРАТА СО СТОРОНОЙ (N + 1) // 2 - 1
            FOR W IN RANGE(SELF.N//2):
                SELF.MAP_SQUARE[H][W] = 3
        SELF.BEST_MAP.APPEND([SELF.N // 2 + 1, 0, (SELF.N + 1) // 2 - 1])


    @STATICMETHOD
```

```
    def can_insert(map_sq, insert_n, x, y):  # проверка на вставку квадрата со стороной insert_n на позицию x y
        for h in range(y, y + insert_n):
            for w in range(x, x + insert_n):
                if map_sq[h][w] != 0:
                    return False
        return True

    @staticmethod
    def insert_square(map_sq, insert_n, numb, x, y):  # вставка квадрата со стороной insert_n и номером numb на позицию x y и возврат новой карты столешницы
        new_map = deepcopy(map_sq)
        for h in range(y, y + insert_n):
            for w in range(x, x + insert_n):
                new_map[h][w] = numb
        return new_map

    def solve(self):  # основной метод, вызываемый из main
        if self.n % 2 == 0:  # оптимизация: если размер стороны столешницы четный, то поле разбивается на 4 квадрата со сторонами n/2
            self.best_square_counter = 4
            self.best_map.append([0, 0, self.n // 2])
            self.best_map.append([self.n // 2, 0, self.n // 2])
            self.best_map.append([0, self.n // 2, self.n // 2])
            self.best_map.append([self.n // 2, self.n // 2, self.n // 2])
            return

        if self.n % 3 == 0:  # оптимизация: является ли кратным 3
            k = self.n // 3  # коэффициент пропорциональности
            self.best_square_counter = 6  # на практике доказано, что квадрат 3x3 и квадраты со стороной кратной 3 можно минимально разбить на 6 квадратов
            # ниже представлены параметры квадратов на вставку, умноженные на коэффициент пропорциональности
            self.best_map.append([k, k, k * 2])
            self.best_map.append([k * 2, 0, k])
            self.best_map.append([0, k * 2, k])
            self.best_map.append([0, 0, k])
            self.best_map.append([0, k, k])
```

```
                SELF.BEST_MAP.APPEND([K, 0, K])
                RETURN

        # ДЛЯ ОСТАВШИХСЯ ЧИСЕЛ ИСПОЛЬЗУЕТСЯ ПЕРЕБОР
        SELF.BEST_START()
        COUNTER = 3  # СЧЕТЧИК ВСТАВЛЕННЫХ КВАДРАТОВ (3, ТАК КАК 3 КВАДРАТА УЖЕ ВСТАВЛЕНЫ ФУНКЦИЕЙ BEST_START)
        MAP_SQUARE = DEEPCOPY(SELF.MAP_SQUARE)
        FREE_S = N ** 2 - ((SELF.N + 1) // 2) ** 2 - 2 * (((SELF.N + 1) // 2 - 1) ** 2)  # СВОБОДНАЯ ПЛОЩАДЬ УМЕНЬШАЕТСЯ НА ПЛОЩАДЬ ВСТАВЛЕННЫХ КВАДРАТОВ
        BEST_MAP = DEEPCOPY(SELF.BEST_MAP)
        SELF.FIND_SQUARE(MAP_SQUARE, FREE_S, COUNTER, BEST_MAP)  # РЕКУРСИВНАЯ ФУНКЦИЯ ПОИСКА НАИЛУЧШЕГО РАСПОЛОЖЕНИЯ КВАДРАТОВ

    DEF FIND_SQUARE(SELF, MAP_SQUARE, FREE_S, COUNTER, BEST_MAP):  # ПОИСК ВСТАВКИ КВАДРАТА (РЕКУРСИВНАЯ РЕАЛИЗАЦИЯ)
        IF COUNTER >= SELF.BEST_SQUARE_COUNTER:  # ВЫХОД ИЗ РЕКУРСИИ ЕСЛИ СЧЕТЧИК ДОСТИГАЕТ ИЛИ ПРЕВЫШАЕТ ЛУЧШЕЕ КОЛИЧЕСТВО КВАДРАТОВ
            RETURN
        FOR H IN RANGE(SELF.N // 2 + 1):  # ПЕРЕБОР СТРОК
            FOR W IN RANGE(SELF.N // 2 + 1):  # ПЕРЕБОР СТОЛБЦОВ
                FOR INSERT_N IN RANGE(SELF.N // 2, 0, -1):  # ПЕРЕБОР РАЗМЕРОВ КВАДРАТОВ НА ВСТАВКУ
                    IF MAP_SQUARE[H][W] == 0 AND SELF.CAN_INSERT(MAP_SQUARE, INSERT_N, W, H):  # ПРОВЕРКА НА ВСТАВКУ
                        NEW_MAP_SQUARE = SELF.INSERT_SQUARE(MAP_SQUARE, INSERT_N, COUNTER + 1, W, H)
                        NEW_RESULT = DEEPCOPY(BEST_MAP)
                        NEW_RESULT.APPEND([H, W, INSERT_N])
                        IF FREE_S - INSERT_N * INSERT_N > 0:  # ПРОВЕРКА ЗАПОЛНЕН ЛИ КВАДРАТ
                            SELF.FIND_SQUARE(NEW_MAP_SQUARE, FREE_S - INSERT_N * INSERT_N, COUNTER + 1, NEW_RESULT)
                            IF H != 0 AND W != 0:  # ПЕРЕБОР ВСТАВОК НЕ В ВЕРХНИЙ ЛЕВЫЙ УГОЛ - ПОВТОРЕНИЕ УЖЕ ПРОВЕРЕННЫХ РАССТАНОВОК
                                RETURN
                        ELSE:
```

```
                                    IF COUNTER < SELF.BEST_SQUARE_COUNTER:   # ПРОВЕРКА ДЛЯ
ОБНОВЛЕНИЯ ТЕКУЩЕГО НАИЛУЧШЕГО РАСПОЛОЖЕНИЯ
                                        SELF.MAP_SQUARE = DEEPCOPY(NEW_MAP_SQUARE)
                                        SELF.BEST_SQUARE_COUNTER = COUNTER + 1
                                        SELF.BEST_MAP = DEEPCOPY(NEW_RESULT)
                                    RETURN
                                IF INSERT_N == 1:
                                    RETURN


    DEF __STR__(SELF):   # РЕАЛИЗУЕТ ТРЕБУЕМЫЙ ВЫВОД: КОЛИЧЕСТВО КВАДРАТОВ И ПАРАМЕТРЫ
КАЖДОГО ИЗ НИХ
        RES = STR(SELF.BEST_SQUARE_COUNTER) + '\N'
        FOR I IN RANGE(SELF.BEST_SQUARE_COUNTER):
            # +1 К КООРДИНАТАМ ТАК КАК В МАССИВЕ ИДЕТ НУМЕРАЦИЯ С НУЛЯ, А В ЗАДАЧЕ С
ЕДИНИЦЫ
            RES = RES + STR(SELF.BEST_MAP[I][0] + 1) + ' ' +
STR(SELF.BEST_MAP[I][1] + 1) + ' ' + STR(SELF.BEST_MAP[I][2]) + '\N'
        RETURN RES


IF __NAME__ == "__MAIN__":
    N = INT(INPUT())
    MAP_SQUARE = MAPSQUARE(N)
    MAP_SQUARE.SOLVE()
    PRINT(MAP_SQUARE)
```