

**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. 1304

Мамин Р.А.

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

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

Выполнение работы.

Для начала был создан класс *Square*, хранящий координаты и размер квадратного обрезка. В файле *main.py* для него объявлены следующие поля и методы:

- *def __init__(self, x, y, w)* – конструктор класса.
- *self.x* – координата левого верхнего угла обрезка по оси Ox
- *self.y* – координата левого верхнего угла обрезка по оси Oy .
- *self.w* – длина стороны обрезка.

Переменная *bestcount* хранит наименьшее количество разбиений.

В *bestarray* хранится массив длины *bestcount*, в котором хранятся объекты класса *Square*.

В переменную *table* записывается начальное состояние доски, ставится квадрат размера $(n+1)/2$ в левый верхний угол доски и два квадрата размером $n/2$ правее и ниже.

Функция *Collision(one, x, y)* проверяет, не занята ли клетка, которую ей передали. Она принимает на вход массив-столешницу *one* и координаты x и y данной точки типа *int*. Функция возвращает *True*, если клетка занята, и *False* в обратном случае.

Основной этап решения задачи происходит в функции *Backtracking(one, summ, boardlen, minx, miny)*, где *one* – массив-столешница, *summ* – величина занятой обрезками площади столешницы типа *int*, *boardlen* – количество обрезков типа *int*, а *minx*, *miny* – координаты типа *int*, с которых начинается сам

перебор. Внутри вложенного перебора координат столешницы в случае, если клетка свободна, размер стороны вставляемого обрезка циклом *for* перебирается от максимально до минимально возможного. При этом на каждой итерации цикла квадрат текущего размера вставляется в столешницу и в случае, если текущая площадь меньше заданной площади столешницы – функция вызывается и повторяет всё вышеперечисленное рекурсивно, в противном случае, очевидно, столешница собрана, поэтому текущее разбиение сравнивается с наилучшим и если первое меньше второго, то оно становится наилучшим. Функция ничего не возвращает, а ответ записывает в переменную *bestarray*.

В самой процедуре нахождения решений также были использованы следующие оптимизации:

1. Если стол не является заполненным и количество квадратов в нём равняется количеству квадратов в прошлом решении, уменьшенному на единицу, нахождение ответа можно прекратить, так как при следующем рекурсивном вызове функции необходимо будет поставить на столешницу ещё один квадрат, после чего их количество будет не менее количества квадратов в решении.

2. Размер столешницы делится на его наибольший делитель, а при выводе ответа обратно умножается на него. Это сделано для уменьшения количества рекурсивных вызовов функции из-за уменьшения площади столешницы.

3. В переменную *table* записывается начальное состояние доски, ставится квадрат размера $(n+1)//2$ в левый верхний угол доски и два квадрата размером $n//2$ правее и ниже, так как это всегда является оптимальным началом решения.

4. Перебираются координаты не всей площади столешницы, а только правой нижней её четверти, так как остальные четверти изначально заполнены в предыдущем пункте и их рассматривать не имеет смысла.

Также данное решение подходит для усложнённой (второй) версии задачи на платформе Stepik.

Разработанный программный код см. в приложении А.

Выводы.

В ходе выполнения работы был изучен, реализован на языке программирования *Python 3*, и применён на практике метод решения задач на тему «Поиск с возвратом». Как правило, он позволяет решать задачи, в которых ставятся вопросы типа: «Перечислите все возможные варианты ...», «Сколько существует способов ...», «Есть ли способ ...», «Существует ли объект...» и т. п. Решение задачи методом поиска с возвратом сводится к последовательному расширению частичного решения. Если на очередном шаге такое расширение провести не удастся, то он возвращается к более короткому частичному решению и продолжает поиск дальше. Данный алгоритм позволил найти все решения поставленной задачи, если они существуют. Для ускорения метода вычисления были организованы таким образом, чтобы как можно раньше выявлять заведомо неподходящие варианты. Это позволило значительно уменьшить время нахождения решения.

ПРИЛОЖЕНИЕ А

ИСХОДНЫЙ КОД

Название файла: *main.py*

```
class Square:

    """Класс квадратного обрезка столешницы."""

    def __init__(self, x, y, w):

        """Конструктор, принимает координаты левого верхнего угла
        столешницы по осям оХ и оУ, длину стороны обрезка типа int"""

        self.x = x

        self.y = y

        self.w = w

n = int(input())

bestcount = 2 * n + 1

bestarray = []

boardsize = 0

for i in range(1, n):

    if (n % i == 0):

        boardsize = i

n = n // boardsize

table = []
```

```

table.append(Square(0, 0, int((n + 1) // 2)))

table.append(Square(0, int((n + 1) // 2), int(n // 2)))

table.append(Square(int((n + 1) // 2), 0, int(n // 2)))

def Collision(one, x, y):

    for square in one:

        if (x >= square.x and x < square.x + square.w and y >=
square.y

        and y < square.y + square.w):

            return True

    return False

def Backtracking(one, summ, boardlen, minx, miny):

    global bestcount

    for x in range(minx, n):

        for y in range(miny, n):

            if (not Collision(one, x, y)):

                right = min(n - x, n - y)

                for square in one:

                    if (square.x + square.w > x and square.y > y):

                        right = min(right, square.y - y)

                for r in range(right, 0, -1):

                    square = Square(x, y, r)

                    b = one.copy()

                    b.append(square)

```

```

if (summ + square.w * square.w == n * n):

    # обновление ответа

    if (boardlen + 1 < bestcount):

        bestcount = boardlen + 1

        bestarray[:] = b.copy()

    else:

        if (boardlen + 1 < bestcount):

            Backtracking(b, summ + square.w * square.w, boardlen +
1, x,y+r)

        else:

            return

    return

miny = n // 2

Backtracking(

    table,

    (n + 1) // 2 * n + 1) // ) + 2 * n // 2 * n // 2, 3,

    n // 2, (n + 1) // 2)

print(len(bestarray))

for i in bestarray:

    print(

        str(i.x * boardsize) + ' ' + str(i.y * boardsize) + ' ' +

        str(i.w * boardsize))

```