

**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

Дешура Д.В.

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.

Принцип работы программы заключается в 2х последовательных переборах с возвратом: сначала мы перебираем всевозможные наборы квадратов так, чтобы сумма их площадей совпадала с площадью исходного квадрата. После этого каждый подходящий набор проверяется на возможность составления квадрата исходного размера из данного набора. Отслеживание наилучшего решения позволяет значительно сокращать количество наборов, которые необходимо проверить. В ходе работы реализованы функции *main*, *findShapeSizes*, *combination* и *setShape*.

1) В функции *main* происходит считывание пользовательского ввода. Алгоритм программы строится на следующем утверждении: если мы можем представить сторону квадрата в виде произведения двух целых чисел, то мы можем разбить исходный квадрат по такому же принципу, как и любой из произведения, но стороны квадратов разбиения будут больше во второе число раз. Поэтому получив исходное число, найдём его наименьший делитель *smallest* и число на которое нужно умножить *smallest*, чтобы получить N – переменная k . После этого рассмотрим частный случай – *smallest* == 2. В таком случае мы сразу можем представить решение. После этого, чтобы получить итоговый ответ нам необходимо масштабировать найденное решение в k раз. Для этого воспользуемся формулой $(\# - 1) * k + 1$ для координат и $\# * k$ для размера квадрата. После этого ответ выводится.

2) Первым шагом алгоритма является перебор подходящих наборов разбиения. Этой задачей занимается функция *findShapeSizes*. Она получает на вход N (переменная *num*) и использует такие переменные как *biggestSquare* (наибольший квадрат в текущем разбиении), *control* (сторона квадрата

относительно которого происходит подбор набора), массив *sizes* (хранит набор квадратов *i*-я позиция отвечает за число квадратов со стороной *i* в разбиении), *squareLeft* (остаток площади, который необходимо заполнить), *thisNum* (текущее число квадратов в разбиении (быстрее, чем каждый раз считать сумму элементов списка)) и *result* (текущий рекорд). Запускается подбор первого набора (зависит от остатка свободной площади *squareLeft* и текущего элемента *control*) функцией *combination*. Далее происходит перебор всех вариантов, пока сторона наибольшего квадрата превышает половину стороны исходного. Получаемые наборы проверяются через функцию *setShape*.

3) Функция *combination* занимается составлением комбинаций, для этого она перебирает стороны квадрата от *control* до 1 и на каждом этапе ставит максимально возможное число квадратов *i*-го размера. Благодаря переменной *control* возможен перебор всех комбинаций.

4) *setShape* – рекурсивная функция, которая на каждом шаге рекурсии вычисляет первое доступное место для вставки следующего квадрата, и последовательно пытается вставлять на это место все доступные квадраты и вызывать следующий шаг. В случае, если не удалось поместить на это место ни одного из имеющихся квадратов – возвращается значение *None* и рекурсия переходит на шаг назад. Если же удалось замостить весь квадрат полностью, то рекурсия возвращает координаты текущего квадрата и его размер и идёт на уровень выше. Таким образом, на выходе мы получим полное описание локального решения. После возвращения в функцию *findShapeSizes* (в случае, если было найдено решение) обновляется рекорд и сохраняется найденное решение, перебор продолжается. Для ускорения работы функции *setShape* используются битовые маски для представления текущего заполнения квадрата.

Программа имеет высокий потенциал для дальнейшей оптимизации.

Исходный код программы указан в приложении А.

Выводы.

В ходе выполнения работы был изучен, реализован на языке программирования C++, и применён на практике метод решения задач на тему «Поиск с возвратом». Вычисления были организованы таким образом, чтобы как можно раньше выявлять неподходящие варианты и ускорить работу. Это позволило значительно уменьшить время нахождения решения и успешно пройти задание за отведенное на платформе Stepik время.

ПРИЛОЖЕНИЕ А

ИСХОДНЫЙ КОД ПРОГРАММЫ

Название файла: Deshura_Dmitriy.py

```
def findShapeSizes (num):
    biggestSquare = num - 1 #сторона наибольшего квадрата
    control = num - biggestSquare    #
    sizes = [0] * num
    sizes[biggestSquare] = 1
    squareLeft = num * num
    record = 2 * num + 1
    squareLeft -= (biggestSquare * biggestSquare)
    thisNum = 1
    mask = [0] * num
    result = None

    sizes, thisNum = combination (sizes, thisNum, control, squareLeft)

    while biggestSquare > num // 2:
        squareLeft = 0

        if thisNum < record:
            buf = sizes.copy()
            buf[biggestSquare] -= 1
            buf = setShape(mask.copy(), buf, num, biggestSquare, 0, 0)
            if buf != None:
                result = buf
                record = thisNum

        squareLeft += sizes[1]
        thisNum -= sizes[1]
        sizes[1] = 0

        i = 2
        while squareLeft // (i - 1) + thisNum > record and i < biggestSquare:
            squareLeft += (i * i * sizes[i])
            thisNum -= sizes[i]
            sizes[i] = 0
            i += 1

        for i in range(2, biggestSquare +1):
            if sizes[i]:
                if i == biggestSquare:
                    biggestSquare -= 1
                    sizes[biggestSquare] = 1
                    squareLeft -= (biggestSquare * biggestSquare)
                    control = num - biggestSquare
                    thisNum += 1
            else:
```

```

        control = i - 1
        sizes[i] -= 1
        thisNum -= 1
        squareLeft += (i * i)
        break
    sizes, thisNum = combination (sizes, thisNum, control, squareLeft)
return result

def combination (sizes, thisNum, control, squareLeft):
    while squareLeft > 0:
        if squareLeft - (control * control) >= 0:
            sizes[control] += 1
            squareLeft -= (control * control)
            thisNum += 1
        else:
            control -= 1

    return sizes, thisNum

def setShape (mask, shapes, N, size, x, y):
    square = ((1 << size) - 1) << x
    for i in range(y, y + size):
        mask[i] += square

    newX = 0
    newY = y
    fullString = (1 << N) - 1
    while newY < N and mask[newY] == fullString:
        newY += 1

    if newY == N:
        return [str(x + 1) + " " + str(y + 1) + " " + str(size)]

    currentString = mask[newY]
    while currentString % 2:
        newX += 1
        currentString = currentString >> 1

    for i in range(N-1, 0, -1):
        if shapes[i]:
            if (((1 << i) - 1) << newX) & ((1 << N) + mask[newY]) == 0 and (N -
newY >= i): #если квадрат влазит и по ширине и по длине
                shapes[i] -= 1
                res = setShape (mask, shapes, N, i, newX, newY)
                shapes[i] += 1
                if res != None:
                    return [str(x + 1) + " " + str(y + 1) + " " + str(size)] +
res
    for i in range(y, y + size):
        mask[i] -= square

```

```

return None

if __name__ == "__main__":
    N = int(input())
    smallest = N
    k = 1
    for i in range(2, N):
        if N % i == 0:
            smallest = i
            k = N // smallest
            break

    if smallest % 2:
        result = findShapeSizes(smallest)
    else:
        result = ["1 1 1", "2 1 1", "1 2 1", "2 2 1"]
    print(len(result))
    for iter in result:
        x, y, w = map(int, iter.split())
        print(str((x - 1) * k + 1) + " " + str((y - 1) * k + 1) + " " + str(w *
k))

```