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 _____ Сулименко М.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).

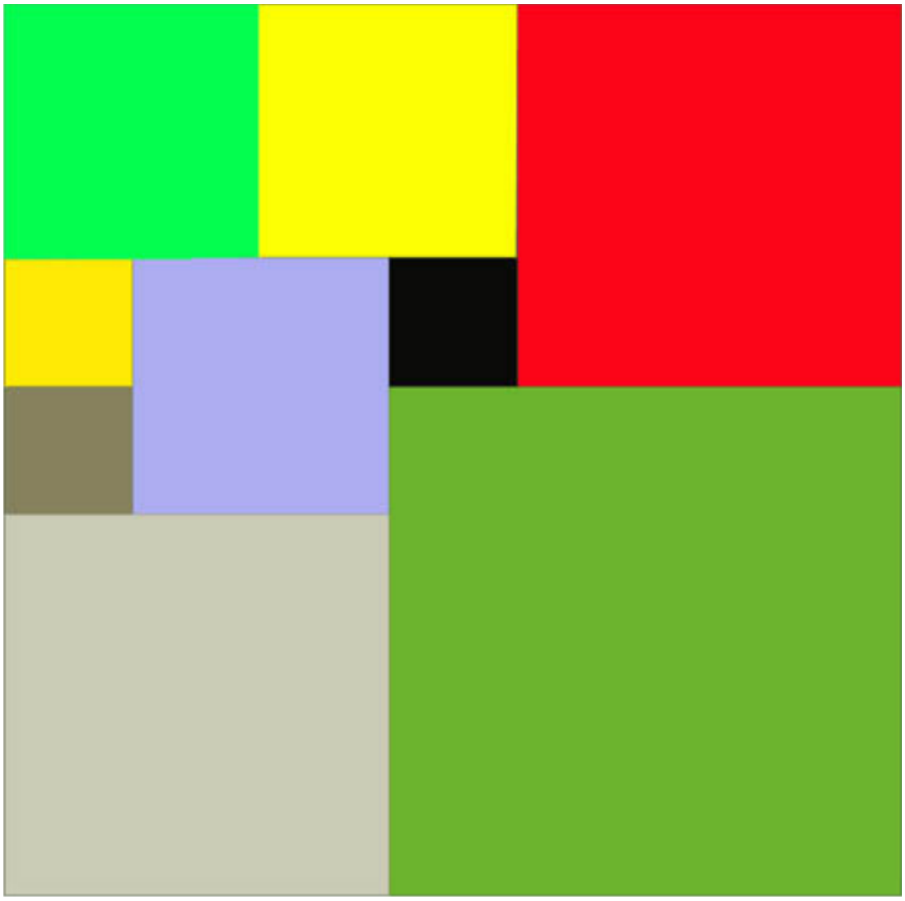

Рисунок 1 - Пример столешницы 7×7

Внутри столешницы не должно быть пустот, обрезки не должны выходитьза 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), изкоторых 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 сторонысоответствующего обрезка(квадрата).

Выполнение работы.

Класс *Square* - хранит координаты и размер квадрата, которыми будет заполнятся вся столешница. Был объявлен конструктор класса с полями координаты квадрата и его ширины:

def __init__(self, x, y, size) – конструктор класса.

Помимо этого были разработаны следующие функции:

is_taken(table, x, y) – функция, которая проверяет клетку на то, занята она, или свободна, и возвращает true/false.

Backtracking(table, currentVolume, squareCount, minx, minY) – основная функция для решения задачи . Она проверяет 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 столешница собрана, поэтому текущее разбиение сравнивается с наилучшим и если первое меньше второго, то оно становится наилучшим. Функция ничего не возвращает.

Переменная *bestScore* хранит наименьшее количество разбиений.

В *bestTable* хранится массив длины *bestScore,* в котором хранятся объекты класса *Square*.

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

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, уменьшенном 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 startT*able* 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 удается, 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 ПРОГРАММЫ

Название файла: *main.py*

```
# Класс маленьких квадртатов, из которых будет заполняться весь стол
class Square:
    # Метод для инициализзации переменных для размера и координат квадрата
    def __init__(self, x, y, size):
        self.x = x
        self.y = y
        self.size = size


# Метод, возвращающий true, если данное место занято, и false в противном
случае
def is_taken(table, x, y):
    for square in table:
        if square.x <= x < square.x + square.size and square.y <= y < square.y
+ square.size:
            return True
    return False


# Основная рекурсивная функция, последовательно пытающаяся расставить
квардаты с помощью метода бектрекинга
def Backtracking(table, currentVolume, squareCount, minX, minY):
    global bestScore
    for x in range(minX, n):
        for y in range(minY, n):
            if not is_taken(table, x, y):
                right = min(n - x, n - y)
                for square in table:
                    if square.x + square.size > x and square.y > y:
                        right = min(right, square.y - y)

                for size in range(right, 0, -1):
                    square = Square(x, y, size)
                    currentTable = table.copy()
                    currentTable.append(square)
                    if currentVolume + square.size * square.size == n * n:
                        if squareCount + 1 < bestScore:
                            bestScore = squareCount + 1
                            bestTable[:] = currentTable.copy()
                    else:
                        if squareCount + 1 < bestScore:
                            Backtracking(currentTable,    currentVolume    +
square.size * square.size, squareCount + 1, x,
                                         y + size)
                        else:
                            return
                return
        minY = int(n // 2)


n = int(input())
bestScore = 2 * n + 1
bestTable = []
sizeOfTable = 1
```

```
for i in range(2, n):
    if n % i == 0:
        sizeOfTable = i
n = n // sizeOfTable

startTable = [Square(0, 0, int((n + 1) // 2)), Square(0, int((n + 1) // 2),
int(n // 2)),
              Square(int((n + 1) // 2), 0, int(n // 2))]

Backtracking(startTable, int((n + 1) // 2) * int((n + 1) // 2) + 2 * int(n
// 2) * int(n // 2), 3,
             int(n // 2), int((n + 1) // 2))

print(len(bestTable))
for i in bestTable:
    print(str(i.x * sizeOfTable) + ' ' + str(i.y * sizeOfTable) + ' ' +
str(i.size * sizeOfTable))
```