# 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. 3343 |  | Силяев Р.А. |
| Преподаватель |  | Жангиров Т.Р. |

Санкт-Петербург 2025

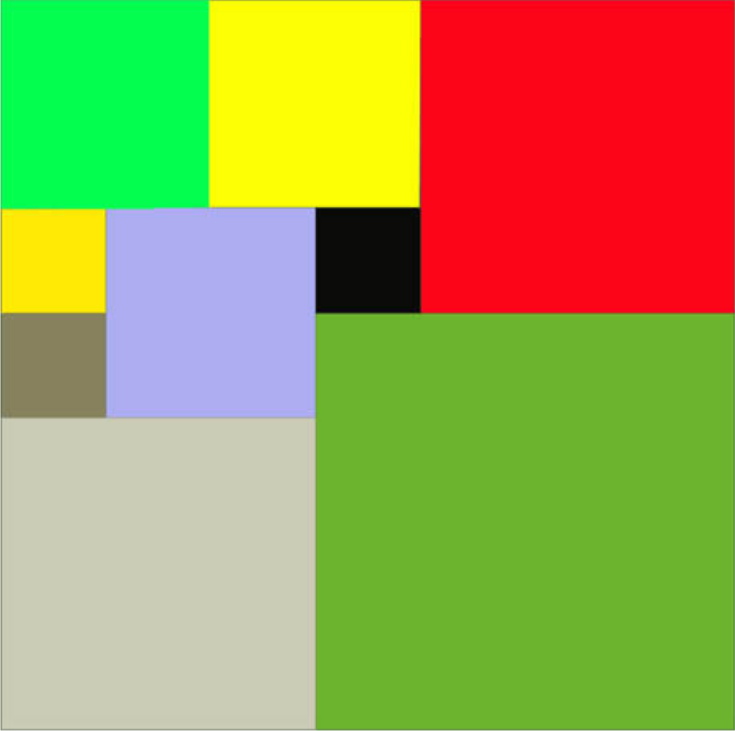
# Цель работы.

Решение классической задачи квадрирования квадрата (с заданными относительно размера ограничениями) посредством программы, основанной на алгоритме поиска с возвратом (англ. backtracking).

# 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(квадра- тов).

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 ис- пользовать минимально возможное число обрезков.

# Входные данные

Размер столешницы - одно целое число *N* (2 ≤ *N* ≤ 40).

# Выходные данные

Одно число *K*, задающее минимальное количество обрезков(квадратов), из ко- торых 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 ≤ *x*, *y* ≤ *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

**Вариант 2р.** Рекурсивный бэктрекинг. Исследование времени выполнения от

размера квадрата

# Описание функций и структур данных.

### **Класс *Square***

Структура, описывающая квадрат:

* ***x***, ***y*** - координаты левого верхнего угла квадрата
* ***size***- длина стороны квадрата
* ***right***,***bottom***- вычисляемые координаты правой и нижней границ квадрата

### **Класс BacktrackState**

Основной класс, реализующий алгоритм поиска с возвратом:

* ***squares***- список размещенных квадратов
* ***occupied\_area*** - занятая площадь
* ***current\_count*** - текущее количество квадратов
* ***start\_x***, ***start\_y*** - начальные координаты для поиска места
* ***grid\_size*** - размер основного квадрата
* ***best\_count***, ***best\_solution*** - лучшее найденное решение

**Методы:**

* ***is\_overlapping(x, y)*** - проверяет, пересекается ли точка с существующими квадратами
* ***backtrack()*** - основной рекурсивный метод поиска с возвратом
* ***calculate\_max\_size(x, y)*** - вычисляет максимальный размер квадрата для данной точки
* ***try\_place\_squares(x, y, max\_size)*** - пробует разместить квадраты разных размеров
* ***should\_skip()*** - проверяет, стоит ли пропускать текущую ветвь поиска
* ***update\_best\_solution()***- обновляет лучшее решение

### **Вспомогательные функции**

1. ***initialize\_initial\_squares(grid\_size)*** - создает начальное разбиение для оптимизации
2. ***find\_max\_square\_size(grid\_size)*** - находит максимальный делитель для оптимизации

### **Основная функция**

***main()***- считывает входные данные, инициализирует поиск и выводит результат

## **Алгоритм работы**

****Инициализация****:

* Считывается размер квадрата N
* Находится оптимальный размер для разбиения
* Создается начальное разбиение (оптимизация)

****Поиск с возвратом****:

* Рекурсивно перебираются возможные размещения квадратов
* На каждом шаге проверяется возможность улучшения текущего решения
* При нахождении полного покрытия проверяется его оптимальность

****Оптимизации****:

* Для четных N используется разбиение на 4 равных квадрата
* Для чисел вида N = 2^r-1 применяется специальный шаблон разбиения
* Используется отсечение ветвей, которые заведомо не приведут к улучшению

## Сложность алгоритма

* ****Пространственная сложность****: O(N^2) для хранения информации о размещенных квадратах
* ****Временная сложность****:
* O(1) для четных N и специальных случаев (например, N=7)
* Экспоненциальная в худшем случае для произвольных N

# Исследование.

# С помощью функции *BenchmarkSolve(b \*testing.B)* замерено время выполнения программы для каждого размера ребра квадрата в диапазоне от 2 до 20.

# Благодаря оптимизациям алгоритм крайне эффективно справляется с чётными числами и числом 7 = 2\*\*3-1. В остальных случаях наблюдается повышенное время исполнения, так как приходится делать полный перебор по площади квадрата.

# Тестирование.

|  |  |  |
| --- | --- | --- |
| Входные данные | Выходные данные | Комментарий |
| 7 | 9  1 1 4  1 5 3  5 1 3  6 4 2  4 6 2  6 6 2  5 4 1  4 5 1  5 5 1 | Оптимизация 3)  Результат верный |
| 15 | 12  1 1 8  1 9 7  9 1 7  12 8 4  8 12 4  12 12 4  10 8 2  8 10 2  10 10 2  9 8 1  8 9 1  9 9 1 | Оптимизация 4)  Результат верный |
| 16 | 4  1 1 8  1 9 8  9 1 8  9 9 8 | Оптимизация 1)  Результат верный |
| 19 | 13  1 1 10  1 11 9  11 1 9  11 10 3  14 10 6  10 11 1  10 12 1  10 13 4  14 16 1  15 16 1  16 16 4  10 17 3  13 17 3 | Оптимизация 2)  Результат верный |

# Выводы.

В соответствии с заданным условиям была написана программа, осуществляющая покрытие квадрата меньшими квадратами посредством поиска с возвратом. В ходе изучения поставленной задачи были выявлены и применены оптимизации, обеспечивающие значительное сокращение перебираемых решений.

Файл main.py

# ПРИЛОЖЕНИЕ А

**ИСХОДНЫЙ КОД ПРОГРАММЫ**

class Square:

def \_\_init\_\_(self, x, y, size):

self.x = x

self.y = y

self.size = size

self.right = x + size

self.bottom = y + size

class BacktrackState:

def \_\_init\_\_(self, squares, occupied\_area, current\_count, start\_x, start\_y, grid\_size, best\_count, best\_solution):

self.squares = squares.copy()

self.occupied\_area = occupied\_area

self.current\_count = current\_count

self.start\_x = start\_x

self.start\_y = start\_y

self.grid\_size = grid\_size

self.best\_count = best\_count

self.best\_solution = best\_solution

def is\_overlapping(self, x, y):

for square in self.squares:

if (x >= square.x and x < square.right) and (y >= square.y and y < square.bottom):

return True

return False

def backtrack(self):

if self.occupied\_area == self.grid\_size \* self.grid\_size:

if self.current\_count < self.best\_count[0]:

self.best\_count[0] = self.current\_count

self.best\_solution[:] = self.squares.copy()

return

for x in range(self.start\_x, self.grid\_size):

for y in range(self.start\_y, self.grid\_size):

if self.is\_overlapping(x, y):

continue

max\_size = self.calculate\_max\_size(x, y)

if max\_size <= 0:

continue

self.try\_place\_squares(x, y, max\_size)

return

self.start\_y = 0

def calculate\_max\_size(self, x, y):

max\_size = min(self.grid\_size - x, self.grid\_size - y)

for square in self.squares:

if square.right > x and square.y > y:

max\_size = min(max\_size, square.y - y)

elif square.bottom > y and square.x > x:

max\_size = min(max\_size, square.x - x)

return max\_size

def try\_place\_squares(self, x, y, max\_size):

for size in range(max\_size, 0, -1):

new\_square = Square(x, y, size)

new\_occupied\_area = self.occupied\_area + size \* size

if self.should\_skip(new\_occupied\_area, x, y, size):

continue

self.squares.append(new\_square)

if new\_occupied\_area == self.grid\_size \* self.grid\_size:

self.update\_best\_solution()

self.squares.pop()

continue

if self.current\_count + 1 < self.best\_count[0]:

new\_state = BacktrackState(

self.squares,

new\_occupied\_area,

self.current\_count + 1,

x,

y,

self.grid\_size,

self.best\_count,

self.best\_solution

)

new\_state.backtrack()

self.squares.pop()

def should\_skip(self, new\_occupied\_area, x, y, size):

remaining\_area = self.grid\_size \* self.grid\_size - new\_occupied\_area

if remaining\_area > 0:

max\_possible\_size = min(self.grid\_size - x, self.grid\_size - y)

if max\_possible\_size == 0:

return True

min\_squares\_needed = (remaining\_area + (max\_possible\_size \*\* 2 - 1)) // (max\_possible\_size \*\* 2)

if (self.current\_count + 1 + min\_squares\_needed) >= self.best\_count[0]:

return True

return False

def update\_best\_solution(self):

if self.current\_count + 1 < self.best\_count[0]:

self.best\_count[0] = self.current\_count + 1

self.best\_solution[:] = self.squares.copy()

def initialize\_initial\_squares(grid\_size):

half\_size = (grid\_size + 1) // 2

small\_size = grid\_size // 2

return [

Square(0, 0, half\_size),

Square(0, half\_size, small\_size),

Square(half\_size, 0, small\_size)

]

def find\_max\_square\_size(grid\_size):

max\_divisor = 1

for i in range(grid\_size // 2, 0, -1):

if grid\_size % i == 0:

max\_divisor = i

break

return max\_divisor, grid\_size // max\_divisor

def main():

grid\_size = int(input().strip())

square\_size, new\_grid\_size = find\_max\_square\_size(grid\_size)

best\_count = [2 \* new\_grid\_size + 1]

initial\_squares = initialize\_initial\_squares(new\_grid\_size)

best\_solution = []

initial\_occupied\_area = initial\_squares[0].size \*\* 2 + 2 \* initial\_squares[1].size \*\* 2

start\_x = initial\_squares[2].bottom

start\_y = initial\_squares[2].x

state = BacktrackState(

initial\_squares,

initial\_occupied\_area,

3,

start\_x,

start\_y,

new\_grid\_size,

best\_count,

best\_solution

)

state.backtrack()

print(best\_count[0])

for square in best\_solution:

print(f"{1 + square.x \* square\_size} {1 + square.y \* square\_size} {square.size \* square\_size}")

if \_\_name\_\_ == "\_\_main\_\_":

main()