

**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

Атоян М.А.

Преподаватель

Жангиров Т.Р.

Санкт-Петербург

2025

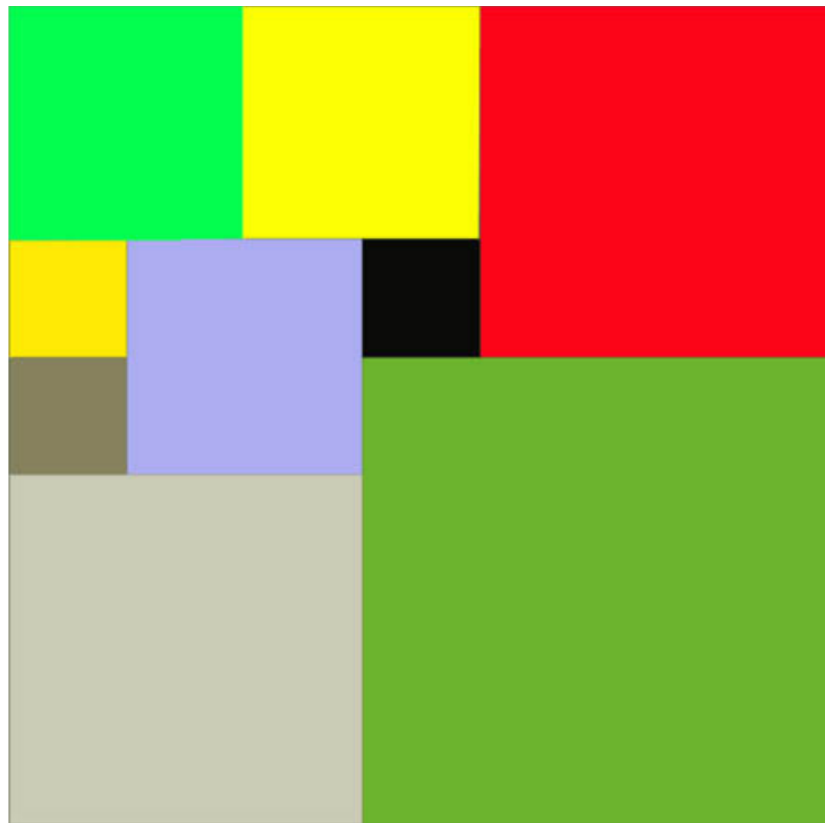
### **Цель работы.**

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(квадратов).

Например, столешница размера  $7 \times 7$  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 использовать минимально возможное число обрезков.

### **Входные данные**

Размер столешницы - одно целое число  $N$  ( $2 \leq N \leq 40$ ).

### **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).

### **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  $7 = 2 \cdot 3 - 1$ . 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	Оптимизация 2) Результат верный

	15 16 1 16 16 4 10 17 3 13 17 3	
--	--	--



### **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.

## ПРИЛОЖЕНИЕ А

### ИСХОДНЫЙ КОД ПРОГРАММЫ

Файл main.py

```
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()

```