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. 0382	____________________	Кондратов Ю.А.
Преподаватель	____________________	Шевская Н.В.

Санкт-Петербург
2022

Цель работы.

Написать программу, реализовывающую разбиение квадрата на минимальный набор квадратов меньшего размера, используя алгоритм поиска с возвратом, а также реализовать тестирование.

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. рис. 1).

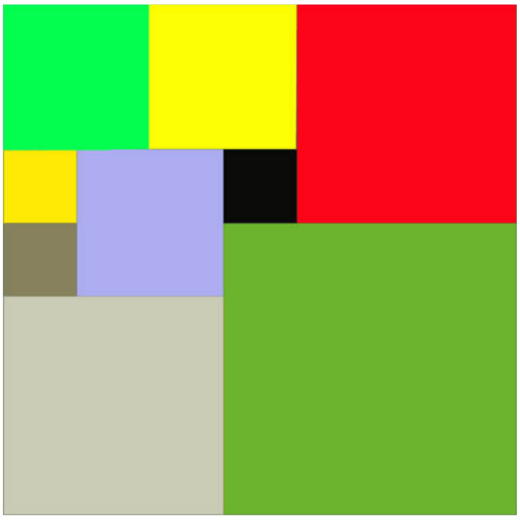

Рисунок 1 – разбиение квадрата 7×7

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 ($1 \le x,y \le N$) и длину стороны соответствующего обрезка (квадрата).

Вариант 1и:

Итеративный бэктрекинг. Выполнение двух заданий на Stepik.

Выполнение работы.

Исходный код программы представлен в приложении А.

Основная логика реализована в классе Table. Класс Square является вспомогательным дата-классом, хранящим координаты верхнего левого угла квадрата и длину его стороны. Квадрат хранится в виде матрицы N x N. Ноль в матрице означает, что клетка пока не закрашены, единица – обратное. В поле squares (тип list) хранится текущее решение (незавершённое). В поле solution хранится лучшее решение, найденное на данный момент.

Методы класса Table:

1) place_square – метод, закрашивающий клетки в соответствии с данными, хранящимися в объекте класса square, передаваемого методу;

2) del_top_square – метод, удаляющий последний добавленный в squares квадрат, также стирает его из матрицы;

3) free_cell – метод, возвращающий координаты самой верхней левой свободной клетки (не закрашенной);

4) check_square – принимает на вход объект класса Square. Возвращает true если квадрат можно разместить без наложений и выхода за пределы матрицы, false в обратном случае;

5) solve_prime – метод, возвращающий решение для простых чисел.

6)solve_other – метод, возвращающий решение для чисел кратных, наименьший делитель которых равен 2, 3 или 5.

Далее описана логика работы бэктрэкинга.

1) Инициализация. Выбирается клетка, возвращённая free_cell. На стэк кладётся квадрат с началом в этой точке, и длиной стороны равной единице.

2) Шаг работы. Достаётся верхний квадрат со стэка. Если его можно разместить, то размещаем. На стэк кладётся квадрат с тем же началом, но длины на 1 больше и квадрат единичной длины с началом в точке, возвращённой free_cell. Если квадрат нельзя разместить, то он удаляется из стека. Если свободных клеток не осталось, решение копируется из squares в solution. Количество квадратов в решение является наименьшим. Это гарантируется проверкой в начале шага цикла, которая отсекается все решение, количество квадратов в которых превосходит уже существующее.

Использованные оптимизации.

1) Всё чётные числа, кратные трём и кратные 5 заранее имеют паттерн разделения на квадраты, который является верным. Поэтому для таких чисел сразу же выводится ответ.

2) Остальные числа в диапазоне от 2 до 30 являются простыми. Для них всегда справедливо следующее: существует минимальное решение, содержащее три квадрата: (1 1 n/2+1), (1 n/2+1 n/2) (n/2+1 1 n/2). Поэтому бэктрэкинг применяется только к оставшейся части квадрата.

3) Для простых чисел не рассматриваются решения, имеющие в составе разбиения более n квадратов. Опытным путём выяснено, что такие решения не являются минимальными.

Выводы.

В ходе работы была написана программа, реализующая разбиение квадрата на минимальный набор квадратов меньших размером.

ПРИЛОЖЕНИЕ А
ИСХОДНЫЙ КОД ПРОГРАММЫ

Название файла: main.cpp

```
from dataclasses import dataclass, astuple
from itertools import product
from copy import deepcopy


@dataclass
class Square:
    x: int
    y: int
    size: int


class Table:
    def __init__(self, n: int):
        self.n = n
        self.matrix = [[0 for _ in range(n + 1)] for _ in range(n +
1)]
        self.squares = []
        self.solution = []

    def __str__(self) -> str:
        return '\n'.join([str(x) for x in self.matrix])

    def place_square(self, square: Square) -> None:
        x, y, size = astuple(square)
        for i, j in product(range(size), range(size)):
            self.matrix[y + j][x + i] = 1
        self.squares.append(square)

    def del_top_square(self) -> None:
        square = self.squares.pop()
        for i, j in product(range(square.size), range(square.size)):
            self.matrix[square.y + j][square.x + i] = 0

    def free_cell(self) -> None | tuple[int, int]:
        for y, x in product(range(self.n // 2 + 1, self.n + 1),
range(self.n // 2 + 1, self.n + 1)):
            if self.matrix[y][x] == 0:
                return y, x

    def check_square(self, square: Square) -> bool:
        x, y, size = astuple(square)
        if y + size - 1 > self.n or x + size - 1 > self.n:
            return False
        for i, j in product(range(size), range(size)):
            if self.matrix[y + j][x + i] == 1:
                return False
        return True

    def solve_prime(self) -> list[Square]:
        z = self.n // 2 + 1
        self.place_square(Square(1, 1, z))
```

```
            self.place_square(Square(1, z + 1, z - 1))
            self.place_square(Square(z + 1, 1, z - 1))
            stack = []
            y, x = self.free_cell()
            stack.append(Square(x, y, 1))
            while stack:
                if    self.solution    and    len(self.squares)    >=
len(self.solution) - 1 or \
                        not self.solution and len(self.squares) > self.n:
                    self.del_top_square()
                    stack.pop()
                    continue
                square = stack[len(stack) - 1]
                x, y, size = astuple(square)
                if self.check_square(square):
                    self.place_square(square)
                    try:
                        new_y, new_x = self.free_cell()
                    except TypeError:
                        self.solution = deepcopy(self.squares)
                        self.del_top_square()
                        self.del_top_square()
                        stack.pop()
                        continue
                    stack.pop()
                    stack.append(Square(x, y, size + 1))
                    stack.append(Square(new_x, new_y, 1))
                    continue
                self.del_top_square()
                stack.pop()
            return self.solution

        def solve_other(self) -> list[Square]:
            if self.n % 2 == 0:
                self.place_square(Square(1, 1, n // 2))
                self.place_square(Square(1, n // 2 + 1, n // 2))
                self.place_square(Square(n // 2 + 1, 1, n // 2))
                self.place_square(Square(n // 2 + 1, n // 2 + 1, n // 2))
                return self.squares

            if self.n % 3 == 0:
                self.place_square(Square(1, 1, (n * 2) // 3))
                self.place_square(Square(1, (n * 2) // 3 + 1, n // 3))
                self.place_square(Square(n // 3 + 1, (n * 2) // 3 + 1, n
// 3))
                self.place_square(Square((n * 2) // 3 + 1, (n * 2) // 3
+ 1, n // 3))
                self.place_square(Square((n * 2) // 3 + 1, 1, n // 3))
                self.place_square(Square((n * 2) // 3 + 1, n // 3 + 1, n
// 3))
                return self.squares

            if self.n % 5 == 0:
                self.place_square(Square(1, 1, (n * 3) // 5))
                self.place_square(Square(1, (n * 3) // 5 + 1, (n * 2) //
5))
                self.place_square(Square((n * 3) // 5 + 1, 1, (n * 2) //
5))
```

```
                    self.place_square(Square((n * 3) // 5 + 1, (n * 3) // 5
+ 1, (n * 2) // 5))
                    self.place_square(Square((n * 2) // 5 + 1, (n * 3) // 5
+ 1, n // 5))
                    self.place_square(Square((n * 2) // 5 + 1, (n * 4) // 5
+ 1, n // 5))
                    self.place_square(Square((n * 3) // 5 + 1, (n * 2) // 5
+ 1, n // 5))
                    self.place_square(Square((n * 4) // 5 + 1, (n * 2) // 5
+ 1, n // 5))
                    return self.squares


if __name__ == "__main__":
    n = int(input())
    table = Table(n)
    if n % 2 == 0 or n % 3 == 0 or n % 5 == 0:
        solution = table.solve_other()
    else:
        solution = table.solve_prime()
    print(len(solution))
    for x in solution:
        print(x.y, x.x, x.size)
```