МИНОБРНАУКИ РОССИИ

САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ

ЭЛЕКТРОТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ

"ЛЭТИ" ИМ. В.И.УЛЬЯНОВА(ЛЕНИНА)

Кафедра МО ЭВМ

ОТЧЁТ

по лабораторной работе

по дисциплине «Построение и анализ алгоритмов»

Тема: Перебор с возвратом.

Студент гр. 1304 _____ Мусаев А.И.

Преподаватель _____ Шевелева А.М.

Санкт-Петербург

2023

Цель работы

Написать алгоритм, решающий задачу, используя перебор с возвратом.

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 (рис.1).

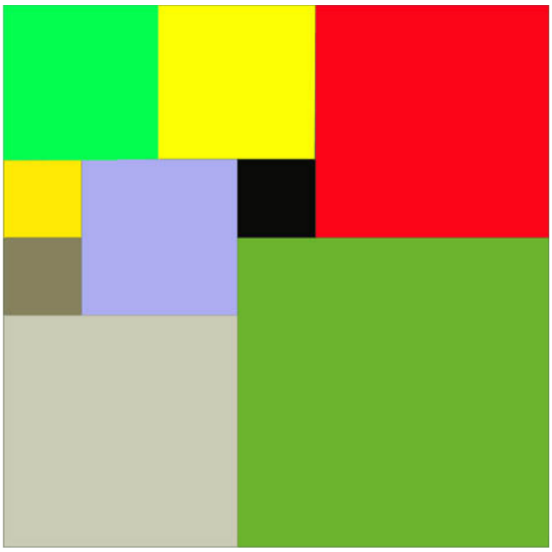

Рисунок 1 – Расстановка для $n = 7$

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 $(2 \le N \le 0)$.

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 ($1 \leq x, y \leq N$) 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

Основные теоретические положения

Поиск с возвратом - общий метод нахождения решений задачи, в которой требуется полный перебор всех возможных вариантов в некотором множестве. Однако, мы будем пользоваться методом ветвей и границ, в отличие от полного перебора - с отсевом подмножеств допустимых решений, заведомо не содержащих оптимальных решений.

Выполнение рабоы

Сначала разберемся с хранением квадрата и результатов. Квадрата $n * n$ - это

матрицы $n * n$, поэтому для начала составим все подматрицы данной матрицы:

```
submatrices = {}
for k in reversed(range(n - (n + 1) // 2)):
    submatrices[k + 1] = [
        (i, j)
        for i in reversed(range(n // 2, n))
        for j in reversed(range(n // 2, n))
        if k + j < n and k + i < n
    ]
```

Подматрицы хранятся с помощью словаря, где ключи - это размеры сторон подматриц, а значения координаты левого верхнего угла, хранящиеся по этим значения. То есть:

`submatrices[7] = [(1,1)]` - это значит, что существует подматрица размером 7, у которой левый верхний угол находится в точке с координатами (1,1).

В ответ мы будем добавлять кортежи размером 3, где первый элемент - координата x левого верхнего угла, второй - координата y, третий - длина стороны, то есть элемент из прошлого абзаца, можем записать (1,1,7).

Теперь разберемся с поиском квадратов. Рекурсивно мы будем перебирать всевозможные подматрицы, пока не найдем ответ. То есть: выбрали одну из подматриц, вызываем функцию еще раз, с добавлением выбранной подматрицы. Если квадрат оказался заполненным и количество подматриц оказалось меньше, чем нынешняя лучшая расстановка, мы заменяем нынешнюю лучшую расстановку на найденную.

Разберемся с оптимизациями:

1. Для четных n всегда можно добиться расстановки, содержащей 4 квадрата, кроме того, координаты и длины этих квадратов мы знаем, поэтому для четных

можно сразу выводить ответ без вызова рекурсивной функции (рис.1).

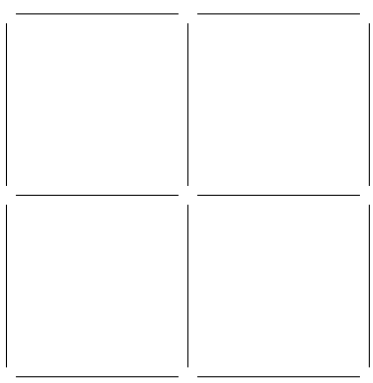

Рисунок 2 – Гарантированная расстановка для четных n

2. Для n, кратных 3, мы можем воспользоваться расстановкой для 3 (рис.2), увеличенной в $\frac{n}{3}$ раз. Как следствие, мы можем просто посчитать расстановку для $n = 3$ и увеличить ее с нужным коэффициентом.

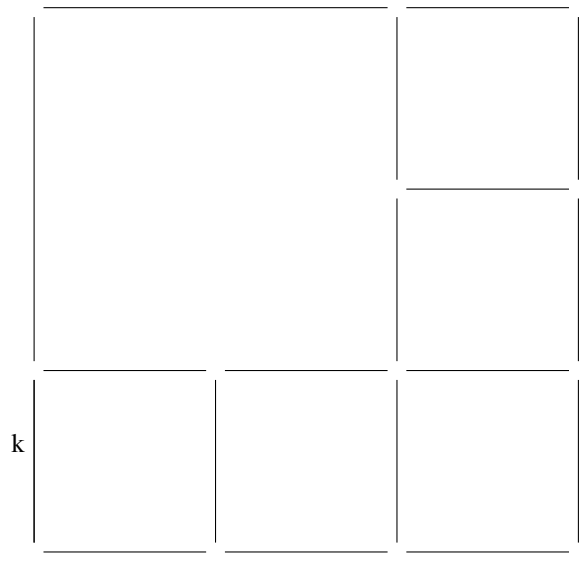

Рисунок 3 – Гарантированная расстановка для n, кратных 3

3. Было выяснено, что для простых n, в одном из углов будет стоять квадрат размером $\frac{n+1}{2}$, кроме того, два квадрата размера $\frac{n-1}{2}$, смежных с ним (рис. 3).

4. Для $2 \leq n \leq 20$ итогове количество квадратов не превышает 13, поэтому мы можем поставить изначальный ответ равным 14.

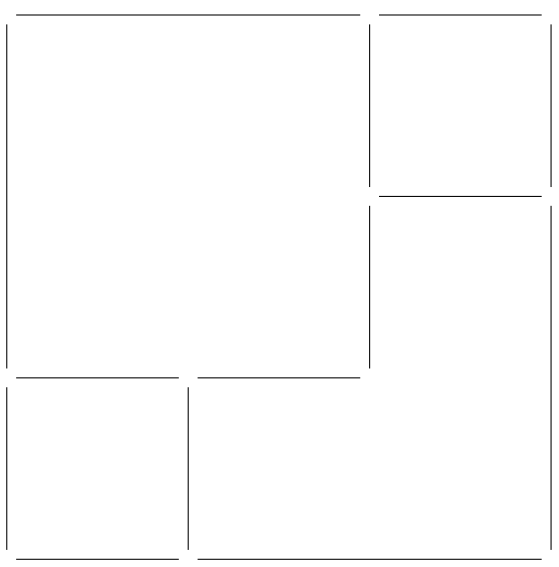

Рисунок 4 – Гарантированная начальная расстановка для простых n

5. Если квадрат еще не заполнен и количество квадратов + 1≥ ans, обрываем данную ветвь, так как даже, если мы, добавив 1 квадрат, дойдем до ответа, он будет не меньше, чем нынешний.

В коде реализованы следующие функции

- is_empty_area - функция принимает уже добавленные подматрицы и подматрицу, которую мы хотим добавить. Функция возвращает булоево значение, которое символизирует, можем ли мы добавить эту подматрицу в нынешнюю расстановку.
- get_optimal_start - расстановка первые трех квадратов в соответствии с 3 пунктом оптимизации.
- backtracking - рекурсивная функция поиска ответа.
- sol_for_even - вывод решения для четных.
- sol_for_multiple3 - вывод решения для кратных 3.
- sol_for_else - вывод решения для остальных чисел.

Вывод

В ходе выполнения данной работы был изучен метод ветвей и границ. Для получения ответа пришлось анализировать задачу, чтобы найти случаи выхода из рекурсии. Программа была ускорена с изначального несчетного времени выполнения до 3 секунд при $2 \geq n \geq 20$.

Каждая функция была ускорена для получения ответа, в реализациях отсутствует полный перебор без отбрасываний, чтобы уложиться во время тестирующей системы.

Кроме того, было изучено, как ускорить программу на Python с помощью средств языка, чтобы ускорить программу.

Приложение А

```
def is_empty_area(added, x, y, k):  # проверка на возможность вставки квадрата
    for i in added:
        if x < i[0] + i[2] and x + k > i[0] and y < i[1] + i[2] and y + k > i[1]:
            return False
    return True


def get_optimal_start():  # установка первых трех квадратов для оптимального старта
    backtracking(
        {
            (0, 0, (n + 1) // 2),
            (0, (n + 1) // 2, (n - 1) // 2),
            ((n + 1) // 2, 0, (n - 1) // 2),
        },
        ((n + 1) // 2) ** 2 + 2 * (n - (n + 1) // 2) ** 2,
    )


def backtracking(
        added, ocup_area
):  # функция, которая перебирает все варианты и возвращает наилучший
    global ans
    if ocup_area < n ** 2 and len(added) + 1 >= ans:
        return
    if len(added) < ans and ocup_area == n * n:
        global ans_mass
        ans, ans_mass = len(added), added
        return

    for size_of_input in submatrices.keys():
        if size_of_input ** 2 <= (n * n - ocup_area):
            for elem in submatrices[size_of_input]:
                if is_empty_area(added, elem[0], elem[1], size_of_input):
                    if not ((elem[0], elem[1], size_of_input) in added):
                        backtracking(
                            added.union({(elem[0], elem[1], size_of_input)}),
                            size_of_input * size_of_input + ocup_area,
```

```
                        )
                        if size_of_input == 1 or size_of_input == 2:
                            return
                        break


def sol_for_even():  # вывод решения для четных
    print(4)
    print(0, 0, n // 2)
    print(0, n // 2, n // 2)
    print(n // 2, 0, n // 2)
    print(n // 2, n // 2, n // 2)
    exit(0)


def sol_for_multiple3():  # вывод решения для кратных 3
    get_optimal_start()
    print(ans)
    for i in ans_mass:
        print(i[0] * koeff, i[1] * koeff, i[2] * koeff)


def sol_for_else():  # вывод решения для остальных
    get_optimal_start()
    print(ans)
    for i in ans_mass:
        print(*i)


n = int(input())
if n % 3 == 0 and n % 2 != 0:
    koeff, n = n // 3, 3
ans, ans_mass, submatrices = 14, {}, {}
for k in reversed(range(n - (n + 1) // 2)):
    submatrices[k + 1] = [
        (i, j)
        for i in reversed(range(n // 2, n - k))
        for j in reversed(range(n // 2, n - k))
```

```
    ]
if n % 2 == 0:
    sol_for_even()
elif n % 3 == 0:
    sol_for_multiple3()
else:
    sol_for_else()
```