МИНОБРНАУКИ РОССИИ
САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ
ЭЛЕКТРОТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ
«ЛЭТИ» ИМ. В.И. УЛЬЯНОВА (ЛЕНИНА)
Кафедра МО ЭВМ

ОТЧЕТ

по лабораторной работе №1

по дисциплине «Программирование и анализ алгоритмов»

Тема: Поиск с возвратом

Студентка гр. 1304 Нго Тхи Йен

Преподаватель Шевелена А.М

Санкт-Петербург

2023

Цель работы.

Изучить алгоритм поиска с возвратом и применить на практике.

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 x 7 может быть помтроена 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(см рис.1).

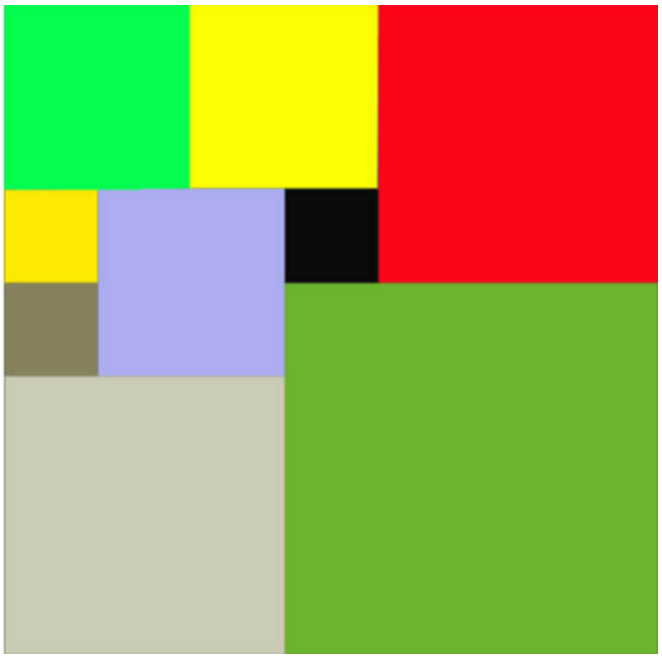

Рисунок 1- Например, столешница размера 7 x 7 может быть помтроена из 9 обрезков

Входные данные:

Размер столешницы - одно целое число N (2 <= N <= 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, w, задающие координаты левого верхнего угла (1 <= x, y <= N) и длину стороны соответствующего обрезка(квадрата).

Выполнение работы.

Алгоритм решения:

1. Найдите наименьший делитель стороны квадрата и вернитесь к решению задачи для квадрата с наименьшим делителем.
2. Если у получившегося квадрата четная сторона, сложите четыре квадрата. В противном случае расположить три квадрата: пусть на вход было подано число n = 2 * k + 1, тогда будут расположены два квадрата с шириной k и один квадрат с шириной k + 1. Больший квадрат должен располагаться в углу, лежащему между углами, в которых расположены меньшие квадраты.
3. Если в предыдущем шаге сторона была нечетной, то перебрать все варианты для непокрытой части квадрата. Учитывая особенность расположения трех квадратов из предыдущего шага, непокрытая сторона – это всегда квадрат со стороной k + 1, причем верхний левый угол которого занят.

Перечисление оптимизаций:

- Перед перебором помещать два квадрата в нижний левый и правый верхний угол с уникальными длинами во избежание повторений вследствие идентичности расстановок, связанных поворотом доски.
- Не рассматривать квадраты в предыдущем пункте с длинами меньшими или равными `n // 4`.
- Контроль глубины: не опускаться глубже уже существующего решения.

- Просматривать длины первых двух квадратов от больших к меньшим

для ранней минимизации максимальной глубины.

- Исключить квадраты со сторонами, большими `n // 2 + 1`.
- При проверке возможности постановки квадрата проверять только

правый и нижний край, т. к. остальное пространство было проверено на предыдущих шагах.

- Очередной квадрат ставится в первый свободный левый верхний угол.

Описание кода.

`solve(n)` – функция, которая находит наименьший делитель стороны квадрата и вызывает функцию `place_big_squares`.

`place_big_squares(n, wide)` – функция, которая помещает первые три (четыре) квадрата и вызывает функцию `solve_square_punct, n- размер стороны большого квадрата`

`solve_square_punct(bias, n, num, wide)` – функция,которая осуществляет нахождение оптимальной расстановки для квадрата с занятой точкой (см. Алгоритм решения п. 3). Осуществляет размещение двух первых квадратов и передает управление функции `back_tracking, bias - это смещение, n- это размер квадрата с выколотой точко, num- номер квадрата выводить, wide- ширина новой координатной сетки.`

`State` – класс, характеризующий текущее состояние столешницы с занятой точкой. Он хранит поле в виде булевого массива, массив квадратов, где каждый квадрат задается как [координата x, координата y, ширина], и длина квадрата с занятой точкой. В классе есть методы для размещения квадрата, функция, которая дает информацию о том, можно ли этот квадрат разместить и метод для нахождения первого свободного левого верхнего угла(`def __niti__(selt,n), def place_square(self, square), def can_be_placed(self, square), def get_free_point(self)`)

`convert(coord, wide)` – преобразовывает координату в

соответствии с делителем, который был выбран на первом шаге алгоритма.

`back_tracking(start, max_deep, n)`- итеративная функция бэктрекинга. Массив `states` хранит состояния в виде объектов класса `State`. Перебирает все состояния и из каждого из них формирует состояния, которые можно получить при размещении квадрата в левый верхний угол. Процесс продолжается до тех пор, пока не будет достигнута максимальная глубина или не найдется решение.

Вар. 2и. Итеративный бэктрекинг. Исследование времени выполнения от размера квадрата.

Для исследования был написан файл `research.py`. В нем для каждого простого размеров квадрата от 2 до 20 проводится три итерации и замеряется среднее время решения. Рассматриваются только простые значения размеров квадрата, так как случаи с составной шириной квадрата будут сведены к случаю с простым числом за время, асимптотически значительно меньшее, чем время последующей обработки. По оси x указан размер стороны квадрата, а по оси y- указано время выполнения(см Рис.2). Точки, характеризующие выполненные измерения, для наглядности сщединены сплошной линией. Динамика роста здесь очень похожа на экспоненциальную. Если подробно проследить за все шаги алгоритма, то можно заметить, что выражение, характеризующее зависимость количества случаев в переборе от длины квадрата, содержит факториалы, что не противоречит изображенному на графике, так как и факториальная зависимость, и экспоненциальная имеют очень похожие асимптотики роста. Одним из свидетельств данного сходства служит формула Стирлинга.

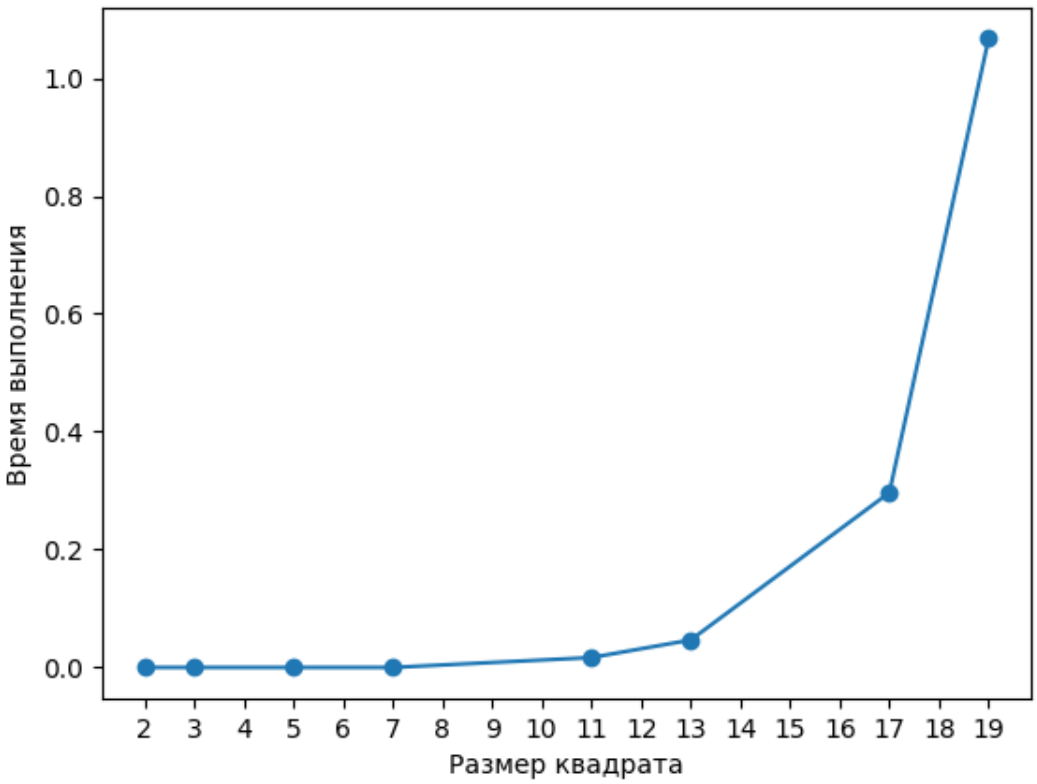

Рисунок 2 – зависимость размера квадрата от времени выполнени

Выводы.

На вход подается целочисленная таблица N(2<=N<=20), после выполнения программа выводит желаемый результат. 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 ≤ x, y ≤ N) и длину стороны соответствующего обрезка (квадрата).

Алгоритм поиска с возвратом изучен и применен на практике. Также строится график зависимости времени выполнения программы от размера квадрата. В ходе наблюдений была установлена мотивация увеличения времени выполнения и дано объяснение полученных результатов. Наглядно былопоказано, что алгоритм является гибким к различным оптимизациям, но его затраты быстровозрастают с увеличением сложности задачи.

Была выяснена сложность алгоритма по количеству операций у данного алгоритма — O(N^N), сложность по памяти — O(N^N).

ПРИЛОЖЕНИЕ А
ИСХОДНЫЙ КОД ПРОГРАММЫ

Файл main.py

```
from copy import deepcopy

class State: #класс, характеризующий текущее состояние столешницы с занятой точкой.

    def __init__(self, n): #является конструктором класса

    #self- Это ссылка на текущие объекты класса и используется для доступа к переменной n этого класса.

    #n- размер стороны большого квадрата

        self.field = [[False] * (n + 1) for i in range(n + 1)]

        self.field[1][1] = True

        self.squares = []

        self.n = n

    def place_square(self, square): #методы для размещения квадрата

        self.squares.append(square)

        y, x, width = square #x, y- координата x, координата y квадрата

        #width- ширина квадрата

        for i in range(y, y + width):

            for j in range(x, x + width):

                self.field[i][j] = True

    def can_be_placed(self, square): #метод, которая дает информацию о том, можно ли этот квадрат разместить

        y, x, width = square

        if x + width - 1 > self.n or y + width - 1 > self.n:

            return False

        for i in range(y, y + width):

            if self.field[i][x + width - 1]:
```

```
                return False
        for i in range(x, x + width):
            if self.field[y + width - 1][i]:
                return False


        return True
    def get_free_point(self): #метод для нахождения первого свободного левого верхнего угла
        for i in range(1, self.n+1):
            for j in range(1, self.n+1):
                if not self.field[i][j]:
                    return [i, j]


        return None
def convert(coord, wide): #преобразовывает координату в соответствии с делителем
    return (coord - 1) * wide + 1
def back_tracking(start, max_deep, n): #функция итеративного бэк-трекинга. Массив states хранит состояния в виде объектов класса State
    states = [start]
    while True:
        next_states = []
        for i, state in enumerate(states):
            if len(state.squares) == max_deep:
                return None
            first_point = state.get_free_point()


            if not first_point:
                return state
            for width in range(1, n // 2 + 2):
                square = [first_point[0], first_point[1], width]
```

```
                if state.field[first_point[0]][first_point[1]]:
                    break
                if state.can_be_placed(square):
                    new_state = deepcopy(state)
                    new_state.place_square(square)
                    next_states.append(new_state)
                else:
                    break
        states = deepcopy(next_states)


def create_state(first_width, second_width, n):
    '''
    помещает квадраты в верхний левый
    и правый нижний углы.
    '''
    state = State(n)
    state.place_square([n - first_width + 1, 1, first_width])
    state.place_square([1, n - second_width + 1, second_width])
    return state
def solve_square_punct(bias, n, num, wide): #функция, которая осуществляет нахождение оптимальной расстановкидля квадрата с занятой точкой
    #Осуществляет размещение двух первых квадратов и передает управление функции back_tracking, bias - это смещение
    #n- это размер квадрата с выколотой точко
    #num- номер квадрата выводить
    #wide- ширина новой координатной сетки
    max_deep = 2 * n + 1
    best_way = None
    end = n // 4
    for i in range(n // 2 + 1, end, -1):
```

```
        for j in range(i, end, -1):
            if j + i > n:
                continue
            start = create_state(i, j, n)
            result = back_tracking(start, max_deep, n)
            if result:
                if len(result.squares) < max_deep:
                    max_deep = len(result.squares)
                    best_way = result
    print(max_deep + num)
    for i in range(len(best_way.squares)):
        print("{0}    {1}    {2}".format(convert(bias    -    1    +
best_way.squares[i][1],    wide),    convert(bias    -    1    +
best_way.squares[i][0], wide), \
            best_way.squares[i][2] * wide))

def place_big_squares(n, wide): #функция, которая помещает первые три
(четыре) квадрата и вызывает функцию solve_square_punct,  n- размер
стороны большого квадрата
      if n % 2 == 0:
        width = n // 2
        print(4)
        print("{0} {1} {2}".format(convert(1, wide), convert(1, wide),
width * wide))
        print("{0}   {1}   {2}".format(convert(width   +   1,   wide),
convert(1, wide), width * wide))
        print("{0} {1} {2}".format(convert(1, wide), convert(width +
1, wide), width * wide))
        print("{0}   {1}   {2}".format(convert(width   +   1,   wide),
convert(width + 1, wide), width * wide))
    else:
        biggest_square = n // 2 + 1
        solve_square_punct(biggest_square, biggest_square, 3, wide)
        print("{0} {1} {2}".format(convert(1, wide), convert(1, wide),
biggest_square * wide))
```

```
        print("{0} {1} {2}".format(convert(biggest_square + 1, wide),
convert(1, wide), (biggest_square - 1) * wide))

        print("{0}       {1}       {2}".format(convert(1,       wide),
convert(biggest_square + 1, wide), (biggest_square - 1) * wide))

def solve(n): #функция, которая находит наименьший делитель стороны
квадрата и вызывает функцию place_big_squares

    divider = n - 1

    wide = 1

    while divider != 1:

        if n % divider == 0:

            wide *= divider

            n //= divider

            divider = n - 1

        else:

            divider -= 1

    place_big_squares(n, wide)

if __name__ == "__main__":

    N = int(input())

    solve(N)
```

Файл research.py

```
import matplotlib.pyplot as plt

import matplotlib

import time

import math

from main import solve


def check_time(epochs):

     times = []

     primes = [2, 3, 5, 7, 11, 13, 17, 19]

     for n in primes:
```

```
            sum_time = 0
            for e in range(epochs):
                  start = time.time()
                  solve(n)
                  sum_time += time.time() - start
            sum_time /= epochs
            times.append(sum_time)
      return times

xint = range(2, math.ceil(21)+1)
matplotlib.pyplot.xticks(xint)

times = check_time(3)
plt.plot([2, 3, 5, 7, 11, 13, 17, 19], times, marker='o')

plt.xlabel("Размер квадрата")
plt.ylabel("Время выполнения")
```