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. 1304 _____ Кардаш Я.Е.

Преподаватель _____ Шевелева А.М.

Санкт-Петербург
2022

Цель работы.

Изучение алгоритма поиска с возвратом. Реализация задачи на применение алгоритма.

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. 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 ($2 \le N \le 20$).

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).

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

Выполнение работы.

Для выполнения задачи создан класс Board, внутри которого реализуется алгоритм.

Внутри main происходит только считывание размера поля, создание объекта класса, вызов метода solve() и вывод результата.

При создании объекта класса объявляются следующие переменные:

Size – размер поля, заданный пользователем

Square – площадь поля

Board – матрица, где 0 обозначает свободную клетку, а другие натуральные числа задают размещенные квадраты.

Result – двумерный список, хранящий данные в формате для вывода.

Record – оптимальное количество квадратов для текущего size

Simp_numb – список простых чисел.

В ходе работы определено, что задачу можно разделить на несколько подзадач, в зависимости от входных данных.

Для четного N оптимальным разбиением является 4 квадрата со стороной N/2. В этом случае задача решается за O(1), применение полного перебора не нужно.

Для нечетного составного N оптимальным решением является разбиение для наименьшего делителя N (назовем его p), пропорционально умноженное на коэффициент N/p. Задача в этом случае сводится к решению задачи для гораздо меньшего квадрата, что сильно снижает затраты времени.

Для простого N решение задачи находится с помощью полного перебора расстановок на доске всевозможных квадратов (перебор ограничен

оптимизирующими улучшениями). Вышеописанную логику реализует метод solve().

Перебор расстановок реализован в методе __recursive_find(board, counter, free_square, result). Метод принимает поле и результирующий список с текущим заполнением, текущее количество квадратов на поле и свободную на данный момент площадь. Метод рекурсивно перебирает всевозможные (с учетом оптимизационных ограничений) вариации расстановок и сравнивает их с текущей оптимальной расстановкой. В результате перебора внутри класса сохраняется оптимальная расстановка.

За проверку возможности вставки квадрата текущего размера на данную клетку отвечает метод __is_insert(board,side,x,y)

За вставку квадрата данного размера на клетку отвечает метод __insert(board,side,x,y,counter)

Для уменьшения рассматриваемых в алгоритме случаев реализовано следующее:

Перед работой рекурсивного алгоритма первые 3 квадрата вставляются вручную. Первый квадрат размера (N+1)/2 вставляется в угол, два других размера (N+1)/2 -1 в смежные углы. Данные квадраты с точностью до перестановки являются частью оптимального решения для простого числа. Также, как следствие, можно ограничить перебор строк, столбцов и размеров вставляемых квадратов при рекурсивном поиске. Вышеописанную логику реализует метод __best_begin()

Если незанятая на доске площадь = 3, это значит, что на оставшиеся места встанет 3 квадрата со стороной 1. Значит, имеет смысл не продолжать рекурсию, а вставить их вручную. Данную логику реализует метод __insert_last(border,counter,result)

Если при текущем вызове рекурсивного метода счетчик больше текущего рекорда, то данная расстановка гарантированно не является решением и ее можно прерывать, не завершая.

Если вставленный квадрат имеет сторону = 1, то есть смысл вставить квадрат, но не запускать дальнейшую рекурсию, поскольку она не имеет вариатива.

Продолжение рекурсии при вставке квадратов не в верхний левый угол – повторение уже проверенной перестановки с точностью до поворота, поэтому ее можно не вызывать.

Разработанный программный код находится в приложении А.

Результаты тестирования программы находятся в таблице 1, в приложении Б

Выводы.

Изучен алгоритм поиска с возвратом, рассмотрены способы его модификации и оптимизации затрат времени.

Реализована программа, позволяющая определить оптимальную расстановку квадратов размер N-1 на поле размера N где 2<=N<=20.

Для оптимизации алгоритма использовано следующее:

Разбиение задачи на случаи (четная, нечетная составная и простая сторона)

Начальная оптимальная вставка

Ограничение лишних вызовов рекурсии при отсутствии вариатива расстановок в них.

ПРИЛОЖЕНИЕ А
ИСХОДНЫЙ КОД ПРОГРАММЫ

Название файла: main.cpp

```
    #для копирования матриц импортирован метод deepcopy
from copy import deepcopy

#класс, реализующий решение задачи
class Board:
    def __init__(self,size):
        self.size = size #размер поля
        self.square = self.size*self.size #площадь поля (используется как
метрика)
        self.board = [[0 for _ in range(size)]for _ in range(size)] #матрица,
где 0 - незанятые квадратами клетки
        self.result = [] #координаты и размеры вставленных кусков
        self.record = 50 #количество кусков (итоговое)
        self.simp_numb = [2,3,5,7,11,13,17,19,23,29,31,37] #простые числа до 40

    #метод решения задачи, вызываемый из main
    def solve(self):
        # если размер стороны четный, оптимальным решением будет разбиение поля
на 4 квадрата со сторонами size/2
        if self.size %2 == 0:
            self.record = 4
            self.result.append([0,0,self.size//2])
            self.result.append([0,self.size//2,self.size//2])
            self.result.append([self.size//2,0,self.size//2])
            self.result.append([self.size//2,self.size//2,self.size//2])
            return
        #для составных чисел оптимальным заполнением будет оптимальное
заполнение для квадрата со стороной наименьшего делителя, пропорционально
увеличенное
        if self.size not in self.simp_numb:
            for elem in self.simp_numb:
                if self.size%elem == 0:
                    coef = self.size//elem
                    small_board = Board(elem)
                    break
            small_board.solve()
            self.record = small_board.record
```

```
            for i in range(self.record):
                self.result.append([])
                for j in range(3):
                    app_elem = small_board.result[i][j] * coef
                    self.result[i].append(app_elem)
            return
        #для простых чисел необходимо найти заполнение с помощью перебора
        self.__best_begin()
        counter = 3 #счетчик для количества вставленных квадратов
        board = deepcopy(self.board)
        free_square = self.square -  ((self.size+1)//2)**2 - 2*((self.size+1)//2
-1)**2
        result = deepcopy(self.result) #массив с форматным выводом
        self.__recursive_find(board,counter,free_square, result)
        self.result.sort()

    #рекурсивный поиск оптимальной вставки
    #принимает текущее заполнение поля и списка результатов, счетчик вставленных
квадратов и свободную площадь
    def __recursive_find(self,board,counter,free_square,result):
        if counter>=self.record:
            return
      for i in range(self.size//2+1): #перебор строк
            for j in range(self.size//2+1): #перебор столбцов
                if board[i][j] != 0:
                    continue
                for k in range(self.size//2, 0, -1): #перебор возможных размеров
квадратов (ограничен первоначальной вставкой)
                    if self.__is_insert(board,k,j,i):
                            new_board = self.__insert(board,k,j,i,counter+1)
                            new_result = deepcopy(result)
                            new_result.append([i,j,k])
                            if free_square-k*k >0:
                                if free_square - k*k == 3: #если оставшаяся
площадь == 3 вариатива в расстановке кубиков нет
                                    new_board =
self.__insert_last(new_board,counter,new_result)
                                    continue

self.__recursive_find(new_board,counter+1,free_square-k*k,new_result)
                            if (i!=0 and j !=0): #перебор вставок не в
верхний левый угол - повторение уже проверенных расстановок с точностью до
поворота
```

```
                                return
                            else:
                                if counter < self.record: #сравнение с текущим
рекордом
                                    self.board = deepcopy(new_board)
                                    self.record = counter+1
                                    self.result = deepcopy(new_result)
                                return
                            if k == 1:
                                return

    # Оптимизация. Лучшая начальная вставка
    def __best_begin(self):
        for i in range(self.size//2,self.size):
            for j in range(self.size//2,self.size):
                self.board[i][j] = 1
        self.result.append([self.size//2,self.size//2,(self.size+1)//2])

        for i in range(self.size//2):
            for j in range(self.size//2+1,self.size):
                self.board[i][j] = 2
        self.result.append([0,self.size//2+1,(self.size+1)//2-1])

        for i in range(self.size//2+1,self.size):
            for j in range (self.size//2):
                self.board[i][j] = 3
        self.result.append([self.size // 2 + 1,0, (self.size + 1) // 2 - 1])
        pass

    #проверка на возможность вставки кубика со стороной side на позицию x y
    def __is_insert(self,board,side,x,y):
        for i in range(y,y+side):
            for j in range(x,x+side):
                if board[i][j] !=0:
                    return False
        return True

    #вставка кубика размера side с номером numb на позицию x y
    def __insert(self,board,side,x,y,numb):
        new_board = deepcopy(board)
        for i in range(y,y+side):
            for j in range(x,x+side):
```

```
                    new_board[i][j] = numb
        return new_board

    #вставка последних трех кубиков размером 1*1
    def __insert_last(self,board,counter,result):
        c = 1
        for i in range(self.size//2+1):
            for j in range(self.size//2+1):
                if board[i][j] == 0:
                    board[i][j] = counter+c
                    result.append([i,j,1])
                    c+=1
        return board
      # формат для вывода
    def __str__(self):
        if self.record == 50 or self.result == []:
            return ""
        result = str(self.record)+'\n'
        for i in range(self.record):
            result = result + str(self.result[i][0]+1)+ '
'+str(self.result[i][1]+1)+' ' +str(self.result[i][2])+'\n'
        return result
if __name__=="__main__":
    n = int(input())
    board = Board(n)
    board.solve()
    print(board)
```

ПРИЛОЖЕНИЕ Б

ТЕСТИРОВАНИЕ ПРОГРАММЫ

Тестирование.

Результаты тестирования представлены в табл. 1.

Таблица 1 – Результаты тестирования

№ п/п	Входные данные	Выходные данные	Комментарии
1.	10	4 1 1 5 1 6 5 6 1 5 6 6 5	Работа программы на четном числе
2.	7	9 1 1 2 1 3 1 1 4 1 1 5 3 2 3 2 3 1 2 4 3 1 4 4 4 5 1 3	Работа программы на небольшом простом числе (число < 10)
3.	15	6 1 1 5 1 6 5 1 11 5 6 1 5 6 6 10 11 1 5	Работа программы на нечетном составном числе

4	19	13 1 1 2 1 3 2 1 5 6 1 11 9 3 1 4 7 1 4 7 5 4 7 9 1 7 10 1 8 9 2 10 9 1 10 10 10 11 1 9	Работа программы на большом простом числе (число > 10)