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

Студентка гр. 2383 Миненок А.М.

Преподаватель Шевелева А.М.

Санкт-Петербург
2024

ЦЕЛЬ РАБОТЫ

Изучить алгоритм поиска с возвратом, который является переборным алгоритмом, а также способы его оптимизации.

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).

ОСНОВНЫЕ ТЕОРЕТИЧЕСКИЕ ПОЛОЖЕНИЯ

Поиск с возвратом - это метод поиска информации, который позволяет пользователям искать нужную информацию, получать результаты поиска и сохранять их для последующего возврата и дальнейшего изучения. Этот подход предполагает, что пользователь может начать поиск, получить некоторые результаты, а затем вернуться к ним позже, чтобы углубить свое понимание или принять более обоснованное решение.

Важными аспектами поиска с возвратом являются открытость результатов поиска (то есть доступность результатов для просмотра и анализа), возможность постепенного уточнения запроса (пользователь может изменить параметры поиска для получения более точных результатов) и обеспечение релевантности результатов (поиск должен быть нацелен на предоставление наиболее актуальной и полезной информации).

Основная цель метода поиска с возвратом - обеспечить удовлетворение потребностей пользователя, предоставляя ему возможность эффективно находить нужную информацию и легко возвращаться к ней для дальнейшего использования или анализа.

ХОД ВЫПОЛНЕНИЯ РАБОТЫ

Обычный путь решения этой задачи с перебором всех вариантов решений с помощью поиска с возвратом достаточно неэффективен, поэтому были приняты некоторые решения для реализации, чтобы оптимизировать алгоритм.

Можно заметить, что если размер стола – число раскладывающееся на простые множители (N), то самый эффективный способ для его заполнения, это взять его наименьший множитель (k), и запустить алгоритм поиска для этого множителя, после чего чтобы получить заполнение исходного квадрата

следует домножить координаты и длины сторон маленьких квадратов на значение равное N/k.

Для простых чисел описанный выше алгоритм не подойдет, поэтому для уменьшения перебора не стоит рассматривать заведомо плохие решения. Так, если найдено более короткое решение, а текущее решение больше его, следует его прекратить и перейти к следующему. Также во всех минимальных расстановках квадратов на поле всегда существуют квадраты с длиной N и N-1, поэтому для эффективности они заполняются сразу же.

Для начала работы была создана функция `divisor(n)`, которая принимает на вход длину набольшего квадрата, и возвращает его наименьший делитель, в случае если число простое оно вернет – 1.

Для поиска свободного места для нового квадрата была создана функция `find_empty(square)`, которая сначала проходится построчно в поисках свободного месте, и как только находит клетку, высчитывает максимальную длину для нового квадрата. Происходит это с помощью прохождения по координате x и координате y до первого заполненного квадрата, после чего берется наименьше из значений найденных длин. Так функция возвращает координаты для нового квадрата, а также его максимальную длину. В случае если места для расстановки квадрата нет – вернет False.

Функция `fill_square(square, area)` – заполняет область новым квадратом по заданным координатам, записывая в массив значение длины квадрата.

Основной функция программы является функция `solve(square, squares)`, она реализует алгоритм поиска с возвратом. В начале проверяется наличие пустого места для расстановки нового квадрата, после чего сравнивается количество квадратов в существующей минимальной расстановки, и в текущей, в случае если какое-то из условий ложно то эта ветка решения обрывается. После чего начинается цикл, который расставляет на поле квадрат с максимально возможной длиной, после чего проверяется, не был ли этот квадрат последним на поле, если да – то сохраняется текущая

расстановка квадратов в глобальную переменную min_count, и происходит очищение этого квадрата, для попытки совершить другую расстановку, иначе – происходит углубление в рекурсию, для расстановки следующего квадрата. Таким образом функция уходит в рекурсию пока не найдет первую расстановку квадратов, которая задаст ограничение количества квадратов, после чего будет каждый раз возвращаться на шаг назад в попытке поставить другой квадрат, в случае если же новая расстановка будет иметь больше квадратов чем минимальная, то она не будет произведена.

В начале программы также прописано условие, при котором при простых числах сразу происходит заполнение трех самых больших квадратов на поле, и уже с этими квадратами запускается функция бэктрекинга.

Разработанный программный код смотрите в Приложении А.

ИССЛЕДОВАНИЕ

Простой алгоритм бектрекинга без какой-либо оптимизации достаточно трудоемкий и его асимптотика для этой задачи равна O(n!), но учитывая добавленную оптимизацию в код программы она будет почти всегда меньше этого значения.

ТЕСТИРОВАНИЕ

Результаты тестирования предоставлены в таблице 2.

Таблица 2.

№	Входные данные	Выходные данные	Комментарии
1	5	8 5 5 1 4 5 1 3 5 1 3 4 1 1 4 2 4 3 2 4 1 2	5 – простое число

		1 1 3	
2	27	6 19 19 9 10 19 9 1 19 9 19 10 9 19 1 9 1 1 18	27 – число с наименьшим делителем - 3
3	11	11 7 1 5 1 7 5 1 1 6 9 9 3 6 9 3 11 8 1 10 8 1 6 8 1 6 7 1 10 6 2 7 6 3	11 – простое число

ВЫВОД

В ходе данной лабораторной работы был исследован и реализован алгоритм поиска с возвратом. Он является достаточно неэффективным хоть и рассматривает все варианты. В данной задаче с заполнением поле квадратами были добавлены некоторые шаги, которые оптимизируют алгоритм:

1. Если длина поля – это составное число, то можно взять его наименьший делитель за длину поля, и заполнять более маленький квадрат.
2. Если длина поля – простое число, то заранее заполним это поле 3 большими квадратами, для упрощения дальнейшего решения.
3. Если во время поиска расстановки, было найдено наименьшее решении, то следующие решения, превышающие его будут обрываться, таким образом уменьшая пространство для поиска.

Исходя их этих оптимизаций можно сказать, что заполнение составных чисел будет происходить почти мгновенно, а вот простые числа будут требовать достаточно много итераций. Сложность алгоритма без оптимизации будет составлять O(n!).

ПРИЛОЖЕНИЕ А.

Код программы

Файл main.py:

```
import copy

def find_empty(square):
    y = x = 0
    while y < len(square) and square[y][x] != 0:
        x += square[y][x]
        if x == len(square):
            y += 1
            x = 0
    if y == len(square):
```

```
            return False
        r_y = 0 if y != 0 else -1
        r_x = 0 if x != 0 else -1
        for i in range(y, len(square)):
            if (square[i][x] == 0):
                r_y += 1
            else:
                break
        for i in range(x, len(square)):
            if (square[y][i] == 0):
                r_x += 1
            else:
                break
        return [x,y, min(r_x, r_y)]

    def fill_square(square, area): # area = [x, y, lenght]
        x = area[0]
        y = area[1]
        for i in range(area[2]):
            for j in range(area[2]):
                square[y + i][x + j] = area[2]

    min_count = []
    def solve(square, squares):
        global min_count
        max_square = find_empty(square)
        if not max_square:
            return
        if len(squares) > len(min_count) and len(min_count) != 0:
            return
        max_len = max_square.copy()
        for i in range(max_len[2], 0, -1):
            max_len[2] = i
            fill_square(square, max_len)
            squares.append(max_len)
            if all([all(i) for i in square]) and ((len(min_count) == 0) or
    (len(min_count) > len(squares))):
                min_count = copy.deepcopy(squares)
            else:
                solve(square, squares)

            for j in range(max_len[2]):
                for k in range(max_len[2]):
                    square[max_len[1] + j][max_len[0] + k] = 0
            squares.pop()
        return

    def divisor(n):
        if not n % 2:
            return 2
        if not n % 3:
            return 3
        if not n % 5:
            return 5
        return 1


    n = int(input())
```

```
squares = []

div = divisor(n)

if div != 1:
    square = [[0 for _ in range(div)] for _ in range(div)]
    solve(square, squares)
else:
    square = [[0 for _ in range(n)] for _ in range(n)]
    half = n - n//2
    fill_square(square, [0,0,half])
    fill_square(square, [0, half, half-1])
    fill_square(square, [half,0,half-1])
    solve(square, squares)
    min_count.append([0,0,half])
    min_count.append([0, half, half-1])
    min_count.append([half,0,half-1])

print(len(min_count))
scale = n//div if div != 1 else 1
j = len(min_count) - 1
while j >= 0:
    print(*[min_count[j][i] * scale + 1 if i != 2 else min_count[j][i]
* scale for i in range(3)])
    j -= 1
    min_count.pop()
```