**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

Рекурсивный бэктрекинг. Исследование времени выполнения от размера квадрата

**Вариант – 2р**

|  |  |  |
| --- | --- | --- |
| Студент гр. 3388 |  | Березовский М.А. |
| Преподаватель |  | Жангиров Т.Р. |

Санкт-Петербург

2025

## Цель работы

Изучить бэктрекинг (рекурсивный), поиск решений с возвратом. Применить полученные знания на практике, реализовав программный код.

## 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≤N≤20).  
**Выходные данные**  
   Одно число *K*, задающее минимальное количество обрезков(квадратов), из которых можно построить  
столешницу(квадрат) заданного размера N*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).  
  
**﻿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

**Описание алгоритм для решения задачи.**

Предварительная обработка: если размер столешницы N делится на 2 или 3, для таких случаев применяется специализированное разбиение, позволяющее сразу получить корректное решение без полного перебора. Это существенно снижает время вычисления.

Общее решение: для него используем метод полного перебора с возвратом (backtracking), идея такая:

- Используем двумерный массив grid размером N\*N – столешница, где каждая ячейка будет отмечена как заполненная или свободная.

- При нахождении первой свободной ячейки, алгоритм пытается разместить в ней квадрат максимально возможного размера (учитывая условия, что квадрат не выходит за границы и не пересекается с уже размещёнными квадратами).

После успешного размещения алгоритм рекурсивно переходит к следующей свободной ячейке и при полном покрытии столешницы, решение сохраняется в переменную best\_placement.

**Оптимизация алгоритма.**

- Специальные разбиения для N кратных 2 или 3 (вручную прописываем фиксированное наилучшее размещение 4 или 6 квадратов)

- Выбор наибольшего возможного квадрата (при каждом рекурсивном шаге функция backtrack ищет первую незаполненную ячейку и пытается разместить в ней квадрат максимального возможного размера, с условием, что он не выходит за границы не перекрывает другие квадраты)

- Остановка (если количество квадратов в текущем частичном решении уже не может быть меньше или равно, чем оптимальное решение до этого, дальнейшее углубление рекурсии прекращается)

- Фиксированное начальное разбиение (перед запуском рекурсивного бэктрекинга для общего случая на простых числах используются три фиксированных квадрата, которые заранее размещаются в матрице, это размещение максимально полезно заполняет большую часть поля, уменьшая площадь, которую необходимо покрыть рекурсивным поиском, улучшая общее время выполнения)

**Оценка сложности алгоритма во времени и памяти в нотации O.**

По времени: **в теории** в худшем случае алгоритм будет работать, перебирать все клетки на столешнице, их N², а также для каждой перебирать все возможные квадраты заполнения, то есть будет O((N²)!), но на практике мы сильно сокращаем эти переборы оптимизациями.

Во всей программе вс существенную для задачи работы выполняет функция backtrack, поэтому проведёт исследование по ней. Представим в таблице количество вызовов backtrack в зависимости от поданного просто числа

Таблица 1 – Количество вызовов *backtrack* в зависимости от поданного простого числа.

|  |  |  |
| --- | --- | --- |
| № п/п | Входные данные | Выходные данные (количество вызовов backtrack) |
|  | 3 | 4 |
|  | 5 | 17 |
|  | 7 | 63 |
|  | 11 | 970 |
|  | 13 | 2384 |
|  | 17 | 16606 |
|  | 19 | 65527 |

Задача является NP-трудной (EXPTIME)*,* так что сложность не полиномиальная. Исходя из исследования, алгоритм имеет сложность

По памяти: затраты приходятся на хранение матрицы grid размером N\*N, сложность для неё O(N²)

- Рекурсивный стек, глубина определяется количеством размещённых квадратов, в худшем случае если 1\*1, тогда O(N²)

- Списки хранения частичных и оптимального решений O(N²)

Итого: O(N²)

**Способ хранения частичных решений.**

- Матрица grid – представляет собой двумерный список размера N\*N, где каждый элемент имеет значение 0 (свободная ячейка) или 1 (ячейка занята квадратом).

- Список current\_placement – список, где каждое частичное решение представлено в виде кортежа (x, y, s) , где x и y – координаты левого верхнего угла размещённого квадрата (индексация с 1), а s – длина стороны квадрата. Список отражает текущую последовательность размещённых квадратов.

- Список best\_placement – используется для хранения оптимального решения, обновляется, когда в процессе рекурсии находится полное покрытие столешницы с меньшим количеством квадратов, чем ранее.

**Рекурсивный бэктрекинг.**

Сигнатура: def backtrack(count)

Назначение: функция выполняет рекурсивный поиск с возвратом для размещения квадратов на столешнице таким образом, чтобы полностью заполнить её с минимальным количеством элементов. Функция ищет все возможные варианты заполнения, используя стратегию жадного выбора наибольшего квадрата и раннюю остановку для сокращения числа переборов.

Аргументы: count – целое число – текущее количество размещённых квадратов в частичном решении (список current\_placement), оно используется для сравнения с текущим оптимальным решением и для контроля глубины рекурсии.

Возвращаемое значение: функция ничего не возвращает напрямую, но внутри себя, при нахождении полного покрытия столешницы копирует текущее частичное решение в переменную best\_placement, таким образом обновляя текущее оптимальное решение на протяжении рекурсии.

**Тестирование. Демонстрация граничных случаев алгоритма.**

Результаты тестирования представлены в табл. 1.

Таблица 1 – Результаты тестирования

|  |  |  |  |
| --- | --- | --- | --- |
| № п/п | Входные данные | Выходные данные | Комментарии |
|  | 2 | 4  1 1 1  2 1 1  1 2 1  2 2 1 | Верный вывод |
|  | 20 | 4  1 1 10  11 1 10  1 11 10  11 11 10 | Верный вывод |
|  | 7 | 9  4 4 4  5 1 3  1 5 3  1 1 2  3 1 2  1 3 2  3 3 1  4 3 1  3 4 1 | Верный вывод |
| 4. | 9 | 6  1 1 6  7 1 3  1 7 3  7 4 3  4 7 3  7 7 3 | Верный вывод |
| 5 | 19 | 13  10 10 10  11 1 9  1 11 9  1 1 6  7 1 4  7 5 4  1 7 4  5 7 2  5 9 2  7 9 2  9 9 1  10 9 1  9 10 1 | Верный вывод |

**Исследование. Результаты. Выводы.**

Проведено исследование, написан дополнительный файл, который представляет график зависимости времени от размера квадратов (Рис.1)

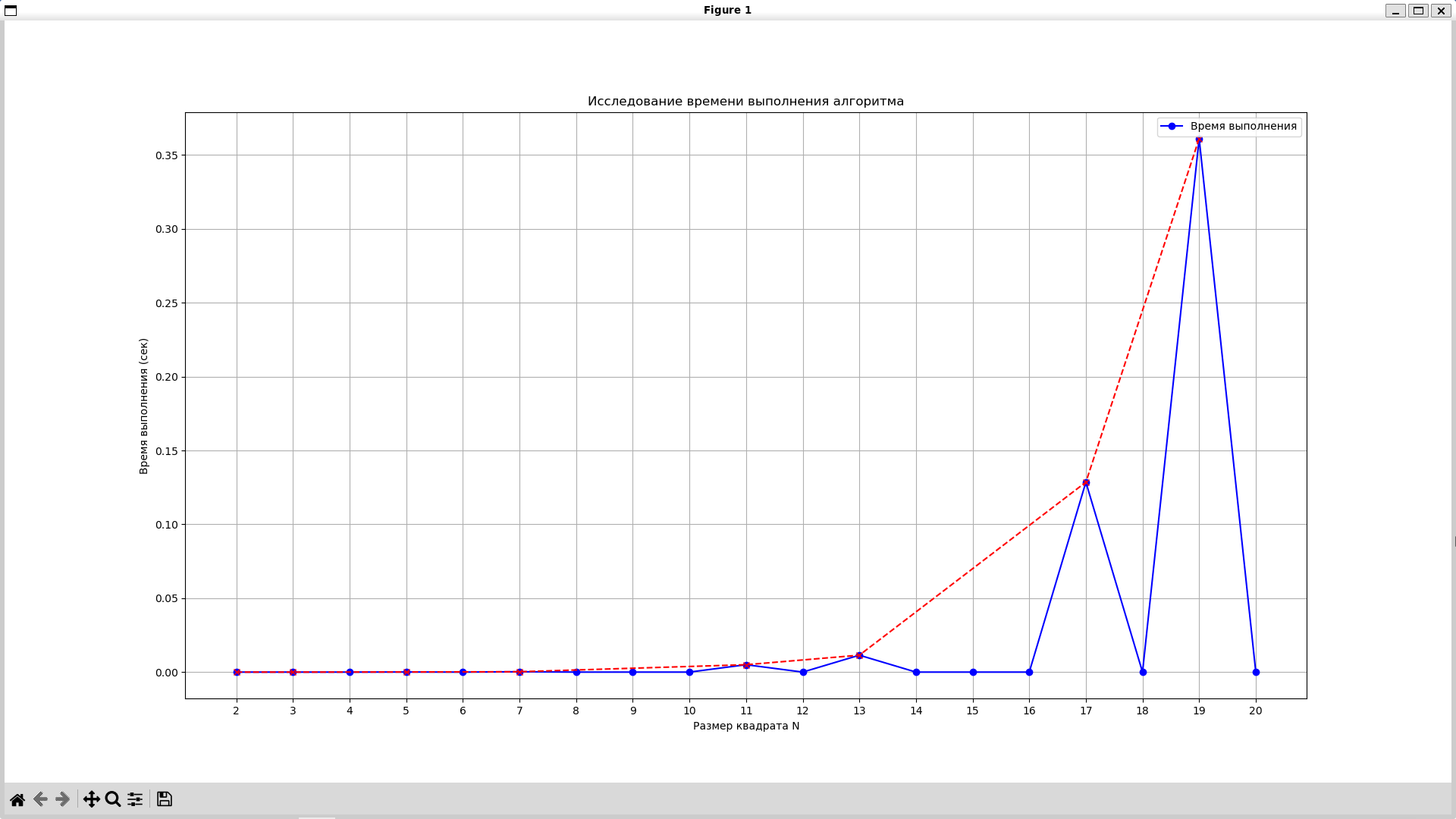


Рис.1

Как видим, случаи, где N кратно 2 или 3 обрабатываются моментально, остальные, простые, числа занимают некоторое время вычислений, но среди них можно наблюдать тенденцию пропорциональности увеличения N и времени выполнения.

Выводы: применение специального разбиения для N кратных 2 или 3 позволяет эффективно обрабатывать многие входные данные. Жадная стратегия выбора размещение в первую очередь наибольших квадратов позволяет уменьшить глубину рекурсии и число перебираемых вариантов. Прерывание рекурсии тоже сокращает число вычислений.

Разработанный программный код см. в приложении А

# Приложение А Исходный код программы

Название файла: lab1.py

def square\_splitting(N):

    if N % 2 == 0:

        half = N // 2

        return [

            (1, 1, half),

            (half + 1, 1, half),

            (1, half + 1, half),

            (half + 1, half + 1, half)

        ]

    elif N % 3 == 0:

        third = N // 3

        return [

            (1, 1, 2 \* third),

            (2 \* third + 1, 1, third),

            (1, 2 \* third + 1, third),

            (2 \* third + 1, third + 1, third),

            (third + 1, 2 \* third + 1, third),

            (2 \* third + 1, 2 \* third + 1, third)

        ]

    grid = [[0] \* N for \_ in range(N)]

    best\_placement = None

    def find\_empty():

        for y in range(N):

            for x in range(N):

                if grid[y][x] == 0:

                    return x, y

        return None

    def can\_place(x, y, s):

        if x + s > N or y + s > N:

            return False

        for dy in range(s):

            for dx in range(s):

                if grid[y + dy][x + dx] != 0:

                    return False

        return True

    def place(x, y, s, value):

        for dy in range(s):

            for dx in range(s):

                grid[y + dy][x + dx] = value

    def backtrack(count):

        nonlocal best\_placement

        if best\_placement is not None and count >= len(best\_placement):

            return

        first\_empty\_position = find\_empty()

        if first\_empty\_position is None:

            best\_placement = current\_placement.copy()

            return

        x, y = first\_empty\_position

        for s in range(min(N - x, N - y), 0, -1):

            if can\_place(x, y, s):

                place(x, y, s, 1)

                current\_placement.append((x + 1, y + 1, s))

                backtrack(count + 1)

                current\_placement.pop()

                place(x, y, s, 0)

    current\_placement = []

    a = (N + 1) // 2

    b = N - a

    fixed = [(a, a, a), (N - b + 1, 1, b), (1, N - b + 1, b)]

    for x, y, s in fixed:

        place(x - 1, y - 1, s, 1)

    backtrack(0)

    return fixed + best\_placement

if \_\_name\_\_ == "\_\_main\_\_":

    N = int(input())

    result = square\_splitting(N)

    print(len(result))

    for square in result:

        print(\*square)

Файл experiment.py

import time

import matplotlib.pyplot as plt

from lab1 import square\_splitting

def measure\_execution\_time():

    sizes = list(range(2, 21))

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

    times = []

    times\_simple = []

    for N in sizes:

        start\_time = time.time()

        square\_splitting(N)

        end\_time = time.time()

        times.append(end\_time - start\_time)

    for i in simple:

        times\_simple.append(times[i-2])

    plt.figure(figsize=(10, 5))

    plt.plot(sizes, times, marker='o', linestyle='-', color='b', label='Время выполнения')

    plt.plot(simple, times\_simple, marker = 'x', linestyle='dashed', color='r')

    plt.xlabel('Размер квадрата N')

    plt.ylabel('Время выполнения (сек)')

    plt.title('Исследование времени выполнения алгоритма')

    plt.xticks(sizes)

    plt.legend()

    plt.grid()

    plt.show()

measure\_execution\_time()