**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. 3344 |  | Тукалкин В.А. |
| Преподаватель |  | Фирсов М.А. |

Санкт-Петербург

2025

**Цель работы**

Изучить и реализовать алгоритм поиска с возвратом, бэктрекинг.

**Задание**

**Вариант**: 2р. Рекурсивный бэктрекинг. Исследование времени выполнения от размера квадрата.

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*. 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

**Основные теоретические положения**

Поиск с возвратом — это метод решения задач, который заключается в систематическом переборе всех возможных вариантов решения с возвратом к предыдущим шагам при обнаружении тупиковой ситуации. Этот подход особенно эффективен для задач, где требуется найти оптимальное решение среди множества возможных, таких как разбиение квадрата на меньшие квадраты, расстановка ферзей на шахматной доске или поиск пути в лабиринте.

Основная идея:

* Начинаем с исходного состояния (пустая доска или матрица).
* На каждом шаге проверяем возможность размещения объекта (например, квадрата) в текущей позиции.
* Если размещение возможно, переходим к следующему шагу, фиксируя текущее состояние.
* Если дальнейшее продвижение невозможно (например, все варианты исчерпаны или нарушены условия задачи), возвращаемся к предыдущему состоянию и пробуем другой путь.
* Процесс повторяется до тех пор, пока не будет найдено решение или не будут перебраны все возможные варианты.

Этот метод обычно реализуется рекурсивно, где каждая ветвь рекурсии соответствует одному из возможных путей решения. Однако, поскольку алгоритм перебирает множество вариантов, его сложность, как правило, растёт экспоненциально.

**Описание алгоритма**

В этой задаче используется рекурсивный бэктрекинг для поиска минимального разбиения квадрата NхN на наименьшее количество подквадратов. Алгоритм работает так:

Сначала создаётся пустая матрица NхN, где каждая клетка изначально помечена как незаполненная. Затем проверяется, является ли число N простым или составным. Это помогает определить начальное разбиение квадрата. Если N чётное, квадрат разбивается на 4 равных подквадрата. Если N простое или составное, используется начальное разбиение на 3 квадрата.

Основная часть алгоритма — это рекурсивный бэктрекинг. Он ищет первую незаполненную клетку на доске и пытается разместить в ней квадрат максимально возможного размера. Если квадрат можно разместить, он добавляется на доску, и алгоритм рекурсивно вызывается для следующей клетки. Если все клетки заполнены, текущее количество квадратов сравнивается с минимальным найденным. Если решение лучше, оно сохраняется. Если текущее количество квадратов превышает минимальное, алгоритм откатывается назад, удаляя последний размещённый квадрат, и пробует другой вариант.

После завершения работы алгоритма выводится итоговое разбиение квадрата, показывающее, как каждый подквадрат размещён на доске. Также измеряется время выполнения алгоритма для разных размеров квадрата, чтобы исследовать его производительность.

Временная сложность алгоритма без оптимизации в худшем случае оценивается как , так как возможно покрытие всей матрицы квадратами размером 1x1. Однако за счёт начального разбиения и предпочтения крупных квадратов количество перебираемых состояний сокращается до и до наилучшего O(1),. Пространственная сложность складывается из памяти для хранения матриц , и стека рекурсивных вызовов, который в худшем случае может хранить до , состояний.

Алгоритм эффективно использует рекурсивный бэктрекинг, но из-за экспоненциальной сложности его применение ограничено для больших значений N.

**Описание функций и структур данных**

Класс содержит методы и данные, необходимые для решения задачи. Включает в себя:

Атрибуты:

* N: Размер квадрата (int).
* board: Матрица NхN, представляющая текущее состояние заполнения (list[list[bool]]).

False – клетка свободна.

True – клетка занята.

* result: Список строк, описывающих итоговое разбиение (list[str]).
* minCount: Минимальное количество квадратов, найденное на данный момент (int).
* squares: Список текущих квадратов, размещённых на доске (list[str]).
* countOperations: Счётчик операций для вывода номера итерации (int).

Методы:

* isPrime(num: int) -> bool:

Проверяет, является ли число `num` простым.

- Параметры:

- `num` – число для проверки (int).

- Возвращает:

- `True`, если число простое, иначе `False`.

* findFirstPrimeDivisor(num: int) -> int:

Находит первый простой делитель числа `num`. Оптимизирован для небольших чисел.

- Параметры:

- `num` – число для разложения (int).

- Возвращает:

- Первый простой делитель (int).

- Если `num` – простое, возвращает само `num`.

* canPlace(x: int, y: int, size: int) -> bool:

Проверяет, можно ли разместить квадрат заданного размера в клетке `(x, y)` без пересечения с занятыми клетками и выхода за границы матрицы.

- Параметры:

- `x`, `y` – координаты верхнего левого угла квадрата (int).

- `size` – размер квадрата (int).

- Возвращает:

- `True`, если квадрат можно разместить, иначе `False`.

* placeSquare(x: int, y: int, size: int) -> None:

Размещает квадрат заданного размера в клетке `(x, y)` на матрице.

- Параметры:

- `x`, `y` – координаты верхнего левого угла квадрата (int).

- `size` – размер квадрата (int).

- Возвращает:

- `None` (изменяет состояние матрицы).

* removeSquare(x: int, y: int, size: int) -> None:

Удаляет квадрат заданного размера из клетки `(x, y)` на матрице.

- Параметры:

- `x`, `y` – координаты верхнего левого угла квадрата (int).

- `size` – размер квадрата (int).

- Возвращает:

- `None` (изменяет состояние матрицы).

* backtrack(count: int) -> None:

Рекурсивно ищет минимальное разбиение доски на квадраты.

- Параметры:

- `count` – текущее количество квадратов (int).

- Возвращает:

- `None` (обновляет глобальные переменные с результатами).

Основной процесс:

Ищет первую незаполненную клетку.

Перебирает возможные размеры квадратов от большего к меньшему.

Размещает квадрат, если это возможно, и рекурсивно вызывает себя.

Если все клетки заполнены, сравнивает текущее количество квадратов с минимальным найденным.

Если решение лучше, сохраняет его.

Откатывается назад, если текущее количество квадратов превышает минимальное.

* solve() -> list[str]:

Основная функция, которая запускает алгоритм и возвращает результат разбиения.

- Параметры:

- Отсутсвуют.

- Возвращает:

- Список строк, описывающих минимальное разбиение на квадраты.

- Первый элемент – количество квадратов, остальные – координаты и размеры квадратов.

- Основной процесс:

- Определяет начальное разбиение в зависимости от размера доски.

- Если размер чётный, разбивает доску на 4 равных квадрата.

- Если размер простой или составной, использует начальное разбиение на 3 квадрата.

- Запускает рекурсивный бэктрекинг для поиска оптимального решения.

* visualization(result: list[str]) -> None:

Визуализирует итоговое разбиение доски на квадраты.

- Параметры:

- `result` – список строк, описывающих разбиение (list[str]).

- Возвращает:

- `None` (выводит результат в консоль).

- Основной процесс:

- Преобразует строки с координатами и размерами квадратов в матрицу.

- Выводит матрицу, где каждому квадрату присвоен уникальный номер.

* drawSquare(field: list[list[int]]) -> None:

Визуализация квадрата с помощью библиотеки PIL.

- Параметры:

- field – двумерный список квадратов по номерам.

- Возвращает:

- `None` (выводит результат на экран).

- Основной процесс:

- Проходит по field и устанавливает квадраты разных цветов на белом квадрата.

Основной процесс программы:

1. Пользователь вводит размер квадрата N.

2. Создаётся объект класса Square, который инициализирует матрицу и необходимые переменные.

3. Запускается метод solve(), который выполняет начальное разбиение и рекурсивный бэктрекинг.

4. Результат разбиения выводится в консоль с помощью метода visualization().

5. Измеряется и выводится время выполнения алгоритма.

**Способ хранения промежуточных решений**

Промежуточные решения хранятся в стеке (массиве строк) squares. В конец стека добавляется строка, в которой записано “x y размер”:

x – принимает значения int

y – принимает значения int

размер – принимает значения int.

Если промежуточное решение является лучше предыдущего, оно копируется в массив result.

**Оптимизации**

1. Использование начального разбиения

Вместо того чтобы начинать с пустой доски, используется готовое разбиение. Это сокращает глубину дерева перебора, уменьшая общее число состояний.

Если размер доски — простое нечетное число, то в левом верхнем углу разместить один квадрат, размер которого равен половине доски + 1, а также два квадрата, размер которых равен половине доски по бокам от него.

Если размер доски — составное нечетное число, то в левом верхнем углу разместить один квадрата, который имеет размер, сторона делённая на первый простой делитель умноженная на 2, а по бокам от него два квадрата размером в половину от первого.

Если размер доски — чётное число, то всю доску заполняем квадратами одинакового размера, равного половине доски.

1. Жадное размещение от больших квадратов к меньшим.

Сначала пытаемся разместить самый большой возможный квадрат. Это сокращает число необходимых квадратов и уменьшает количество ветвлений в поиске.

1. Отбрасывание веток, если текущее разбиение хуже найденного (if count >= minCount).

Если текущее количество квадратов уже больше, чем лучшее найденное, ветка не рассматривается. Позволяет раньше остановить поиск в неэффективных разбиениях.

**Тестирование**

Таблица 1 – Результаты тестирования

|  |  |  |  |
| --- | --- | --- | --- |
| № | Входные данные | Выходные данные | Комментарий |
| 1 | 10 | 4  1 1 5  1 6 5  6 1 5  6 6 5 | Верно |
| 2 | 7 | 9  1 1 4  1 5 3  4 5 2  4 7 1  5 1 3  5 4 1  5 7 1  6 4 2  6 6 2 | Верно |
| 3 | 13 | 11  1 1 7  1 8 6  8 1 6  7 8 2  7 10 4  8 7 1  9 7 3  11 10 1  11 11 3  12 7 2  12 9 2 | Верно |
| 4 | 17 | 12  1 1 9  1 10 8  10 1 8  9 10 2  9 12 4  9 16 2  10 9 1  11 9 3  11 16 2  13 12 1  13 13 5  14 9 4 | Верно |
| 5 | 19 | 13  1 1 10  1 11 9  11 1 9  10 11 3  10 14 6  11 10 1  12 10 1  13 10 4  16 14 1  16 15 1  16 16 4  17 10 3  17 13 3 | Верно |

Вывод с комментариями для 5:

Введите размер квадрата: 5

>>> Начало алгоритма <<<

Число 5 простое, будет поставлено 3 квадрата с параметрами (x, y, размер)

1 1 3

1 4 2

4 1 2

Итерация 1:

Текущее количество квадратов: 3

Текущий минимум: inf

Квадрат с размерами 2 установить на координаты (x,y): 2 3

Итерация 2:

Текущее количество квадратов: 4

Текущий минимум: inf

Квадрат с размерами 1 установить на координаты (x,y): 3 2

Итерация 3:

Текущее количество квадратов: 5

Текущий минимум: inf

Квадрат с размерами 1 установить на координаты (x,y): 4 2

Итерация 4:

Текущее количество квадратов: 6

Текущий минимум: inf

Квадрат с размерами 1 установить на координаты (x,y): 4 3

Итерация 5:

Текущее количество квадратов: 7

Текущий минимум: inf

Квадрат с размерами 1 установить на координаты (x,y): 4 4

Итерация 6:

Текущее количество квадратов: 8

Текущий минимум: inf

Найдено новое минимальное количество 8 (предыдущий минимум inf)

Текущее заполнение

1 1 3

1 4 2

4 1 2

3 4 2

4 3 1

5 3 1

5 4 1

5 5 1

Квадрат с размерами 1 установить на координаты (x,y): 2 3

Итерация 7:

Текущее количество квадратов: 4

Текущий минимум: 8

Квадрат с размерами 1 установить на координаты (x,y): 2 4

Итерация 8:

Текущее количество квадратов: 5

Текущий минимум: 8

Квадрат с размерами 2 установить на координаты (x,y): 3 2

Итерация 9:

Текущее количество квадратов: 6

Текущий минимум: 8

Квадрат с размерами 1 установить на координаты (x,y): 3 4

Итерация 10:

Текущее количество квадратов: 7

Текущий минимум: 8

Квадрат с размерами 1 установить на координаты (x,y): 4 4

Итерация 11:

Текущее количество квадратов: 8

Текущий минимум: 8

Результат хуже лучшего, отбрасываем ветку

Квадрат с размерами 1 установить на координаты (x,y): 3 2

Итерация 12:

Текущее количество квадратов: 6

Текущий минимум: 8

Квадрат с размерами 2 установить на координаты (x,y): 3 3

Итерация 13:

Текущее количество квадратов: 7

Текущий минимум: 8

Квадрат с размерами 1 установить на координаты (x,y): 4 2

Итерация 14:

Текущее количество квадратов: 8

Текущий минимум: 8

Результат хуже лучшего, отбрасываем ветку

Квадрат с размерами 1 установить на координаты (x,y): 3 3

Итерация 15:

Текущее количество квадратов: 7

Текущий минимум: 8

Квадрат с размерами 1 установить на координаты (x,y): 3 4

Итерация 16:

Текущее количество квадратов: 8

Текущий минимум: 8

Результат хуже лучшего, отбрасываем ветку

>>> Итоговый результат <<<

Минимальное количество квадратов: 8

Расположение квадратов (x, y, размер)

1 1 3

1 4 2

4 1 2

3 4 2

4 3 1

5 3 1

5 4 1

5 5 1

Итоговое заполнение

1 1 1 2 2

1 1 1 2 2

1 1 1 4 4

3 3 5 4 4

3 3 6 7 8

Время выполнения 0.038868 секунд

**Исследование**

Алгоритмы:

1. Базовая версия (без оптимизаций):

Последовательно перебирает все варианты размещения квадратов, начиная с максимального возможного размера.

Не использует оптимизаций, начальных разбиений или отсечений.

1. Оптимизированная версия:

Начальное разбиение

Отсечение ветвей: Игнорирование заведомо неоптимальных вариантов.

Приоритет крупных квадратов: Уменьшение глубины рекурсии.

n=2,3,4,5,6,7,8,25,29.

Замеры проводились в секундах для двух сценариев: полный перебор и оптимизированный алгоритм.

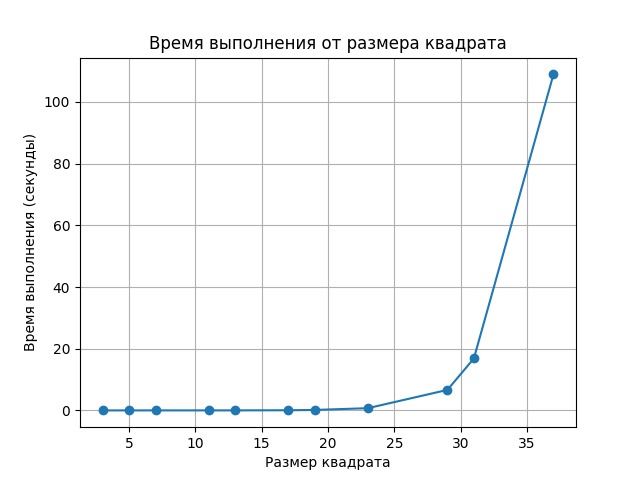
Таблица 2 – Результаты исследования

|  |  |  |
| --- | --- | --- |
| Размер | Без оптимизаций | С оптимизациями |
| 2 | 0.000000 | 0.000000 |
| 3 | 0.000000 | 0.000000 |
| 4 | 0.000000 | 0.000000 |
| 5 | 0.075999 | 0.000000 |
| 6 | 2.302904 | 0.000000 |
| 7 | 128.414531 | 0.003953 |
| 8 |  | 0.000000 |
| 25 |  | 1.391890 |
| 29 |  | 45.619610 |

График 1 - Результаты исследования



График 2 - Результаты исследования (только простые числа)



По результатам исследования можно сказать, что для простых чисел сложность растёт экспоненциально.

**Вывод**

Был реализован алгоритм, использующий рекурсивный бэктрекинг для решения задачи о заполнении квадрата размером N наименьшим количеством квадратов размеров от 1 до n - 1.

**Приложение А**

**ИСХОДНЫЙ КОД ПРОГРАММЫ**

Имя файла: код с комментариями.py

import time

import matplotlib.pyplot as plt

from PIL import Image, ImageDraw

class Square:

def \_\_init\_\_(self, N):

"""

Инициализация класса

:type N: int

"""

self.N = N # Размер квадрата

self.result = [] # Массив для записи в него лучшего решения

self.board = [[False] \* N for i in range(N)] # Двухмерный массив, который используется для нахождения места

self.minCount = float("inf") # Количество квадратов, лучшее решение

self.squares = [] # Массив для размещения промежуточных результатов, потом копируются в result

self.countOperations = 0 # Счётчик итераций для вывода номера в консоль

def isPrime(self, num):

"""

Проверка на простоту

:type num: int

:rtype: bool

"""

for i in range(3, int(num \*\* 0.5) + 1, 2):

if num % i == 0:

return False

return True

def findFirstPrimeDivisor(self, num):

"""

Поиск наименьшего простого делителя

:type num: int

:rtype int

"""

# Проверка распространенных простых делителей для оптимизации

for i in [3, 5, 7, 11, 13, 17, 19, 23]:

if num % i == 0:

return i

return num

def placeSquare(self, x, y, size):

"""

Поставить квадрат по заданным координатам

:type x: int

:type y: int

:type size: int

"""

for i in range(x, x + size):

for j in range(y, y + size):

self.board[i][j] = True

def removeSquare(self, x, y, size):

"""

Удалить квадрат по заданным координатам

:type x: int

:type y: int

:type size: int

"""

for i in range(x, x + size):

for j in range(y, y + size):

self.board[i][j] = False

def canPlace(self, x, y, size):

"""

Проверка места для квадрата

:type x: int

:type y: int

:type size: int

:rtype: bool

"""

if x + size > self.N or y + size > self.N:

return False

# Проверка левой и верхней границы

for i in range(x, x + size):

if self.board[i][y]:

return False

for j in range(y, y + size):

if self.board[x][j]:

return False

return True

def backtrack(self, count):

"""

Алгоритм бэтрекинга (основной цикл обработки)

:type count: int

"""

self.countOperations += 1

print(f"\nИтерация {self.countOperations}:")

print(f"Текущее количество квадратов: {count}")

print(f"Текущий минимум: {self.minCount}")

# Отсечение ветвей, если текущее решение хуже найденного

if count >= self.minCount:

print("Результат хуже лучшего, отбрасываем ветку")

return

# Поиск места для нового квадрата

x, y = -1, -1

for i in range(self.N):

for j in range(self.N):

if not self.board[i][j]:

x, y = i, j

break

if x != -1:

break

# Запись нового минимума

if x == -1:

if count < self.minCount:

# Вывод промежуточных результатов

print(f"Найдено новое минимальное количество {count} (предыдущий минимум {self.minCount})")

print(f"Текущее заполнение")

for i in self.squares:

print(i)

# Копирование нового минимума

self.minCount = count

self.result = self.squares.copy()

return

# Подбор квадрата, подходящего на это место

for size in range(min(self.N - x, self.N - y), 0, -1):

if self.canPlace(x, y, size):

print(f"Квадрат с размерами {size}", end=' ')

print(f"установить на координаты (x,y): {x} {y}")

self.placeSquare(x, y, size)

self.squares.append(f'{x + 1} {y + 1} {size}') # +1 для индексации с 1

self.backtrack(count + 1)

# Откат изменений

self.squares.pop()

self.removeSquare(x, y, size)

def solve(self):

"""

Расстановка базовых квадратов и запуск бэктрекинга для нечётных чисел

:rtype: List[str]

"""

print(">>> Начало алгоритма <<<")

if self.N < 2:

return ['0']

# Алгоритм для чётных чисел

if self.N % 2 == 0:

print(f"Число {self.N} чётное, будет разбито на 4 квадрат.")

half = self.N // 2

self.minCount = 4

self.result.append("1 1 " + str(half))

self.result.append(f"{1 + half} 1 {half}")

self.result.append(f"1 {1 + half} {half}")

self.result.append(f"{1 + half} {1 + half} {half}")

else:

# Алгоритм для простых чисел

if self.isPrime(self.N):

print(f"Число {self.N} простое, будет поставлено 3 квадрата с параметрами (x, y, размер)")

half = self.N // 2

# Верхний левый квадрат

self.placeSquare(0, 0, half + 1)

self.squares.append(f"1 1 {half + 1}")

# Левый нижний квадрат

self.placeSquare(0, half + 1, half)

self.squares.append(f"1 {half + 2} {half}")

# Правый верхний квадрат

self.placeSquare(half + 1, 0, half)

self.squares.append(f"{half + 2} 1 {half}")

for i in self.squares: print(i)

self.backtrack(3)

else:

# Алгоритм для составных чисел

print(f"Число {self.N} составное, будет поставлено 3 квадрата с параметрами (x, y, размер)")

divisor = self.findFirstPrimeDivisor(self.N)

newSize = self.N // divisor

# Верхний левый квадрат

self.placeSquare(0, 0, newSize \* 2)

self.squares.append(f"1 1 {newSize \* 2}")

# Левый нижний квадрат

self.placeSquare(0, newSize \* 2, newSize)

self.squares.append(f"1 {newSize \* 2 + 1} {newSize}")

# Правый верхний квадрат

self.placeSquare(newSize \* 2, 0, newSize)

self.squares.append(f"{newSize \* 2 + 1} 1 {newSize}")

for i in self.squares: print(i)

self.backtrack(3)

return [str(len(self.result))] + self.result

def visualization(self, result):

"""

Визуализация итогового квадрата, вывод количества квадратов и расположения

:type result: List[str]

"""

print("\n>>> Итоговый результат <<<")

print(f'Минимальное количество квадратов: {result[0]}')

print("Расположение квадратов (x, y, размер)")

for i in result[1:]:

print(i)

print("\nИтоговое заполнение")

arr = [[int(x) for x in i.split()] for i in self.result]

field = [[0] \* self.N for i in range(self.N)]

for k in range(len(arr)):

for i in range(arr[k][0] - 1, arr[k][0] - 1 + arr[k][2]):

for j in range(arr[k][1] - 1, arr[k][1] - 1 + arr[k][2]):

field[i][j] = k + 1

for i in field:

print(\*i, sep=' ')

self.drawSquare(field)

def drawSquare(self, field):

'''

Рисует квадраты в квадрате

:type field: List[List[int]]

'''

colors = [

"green", "gray", "red", "cyan", "navy", "black", "orange", "yellow", "lime", "blue", "purple", "magenta",

"pink", "teal", "lavender", "brown", "beige", "maroon"

]

pixel = 50

size = self.N \* pixel

img = Image.new("RGB", (size, size), "white")

draw = ImageDraw.Draw(img)

for i in range(self.N):

for j in range(self.N):

x1 = j \* pixel

y1 = i \* pixel

x2 = x1 + pixel

y2 = y1 + pixel

# Закрашиваем квадрат

color = colors[(field[i][j] - 1) % len(colors)]

draw.rectangle((x1, y1, x2, y2), fill=color)

img.show() # Открываем изображение

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

width = int(input("Введите размер квадрата: "))

start = time.time()

s = Square(width)

result = s.solve()

s.visualization(result)

print(f"\nВремя выполнения {time.time() - start:.06f} секунд")