МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ

Федеральное государственное образовательное учреждение высшего образования «Санкт-петербургский государственный морской технический университет»

ФАКУЛЬТЕТ ЦИФРОВЫХ ПРОМЫШЛЕННЫХ ТЕХНОЛОГИЙ

Кафедра Киберфизических систем

ЛАБОРАТОРНАЯ РАБОТА

По дисциплине «Программирование»

Выполнил: Коткин Вадим Иванович

Проверил: Поделенюк Павел Петрович

Санкт-Петербург

2024

Оглавление

[1. Цели и формулировка задачи 3](#_Toc187676633)

[2. Результаты работы 4](#_Toc187676634)

[2.1. Реализация программы с использованием функционального программирования языка Python 4](#_Toc187676635)

[2.1.1. Ход работы 4](#_Toc187676636)

[2.1.2. Демонстрация работы программы 4](#_Toc187676637)

[2.1.3. Листинг кода 4](#_Toc187676638)

[ЗАКЛЮЧЕНИЕ 5](#_Toc187676639)

[Список использованных источников 6](#_Toc187676640)

# Цели и формулировка задачи

ЦЕЛИ РАБОТЫ  
При работе над данной лабораторной работой будут затронуты следующие вопросы:  
1. Работа с текстовыми файлами,  
2. Работа с локальными и глобальными переменными,  
3. Работа с несколькими функциями, а не только с main,   
4. Нисходящий метод разработки программ,  
5. Анализ ходов.  
ФОРМУЛИРОВКА ЗАДАНИЯ  
 Дана квадратная шахматная доска размером N x N. На доске уже размещено K фигур. Фигуры размещены так, что находятся не под боем друг друга.  
Необходимо расставить на доске еще L фигур так, чтобы никакая из фигур на доске не находилась под боем любой другой фигуры. Необходимо найти все возможные решения.  
Вы работаете с фигурой: \*(Задание по вариантам)   
Входные данные в файле input.txt. На первой строке файла записаны три числа: N L K (через пробел). Далее следует K строк, содержащих числа x и y (через пробел) - координаты уже стоящей на доске фигуры (фигуры стоят правильно). Координаты отсчитываются от 0 до N-1. 1 <= N <= 20.  
Ваш алгоритм должен быть оптимизирован, для работы с доской 20!!   
Выходные данные в файл output.txt. На каждое найденное решение необходимо записать в файл одну строку. Строка состоит из пар (x,y) - координаты фигур на доске. В решение следует вывести координаты всех фигур, находящихся на доске. Каждую фигуру необходимо записать в виде пары координат, разделенных запятой и обрамленных скобками. Координаты отсчитываются от 0 до N-1. Порядок, в котором фигуры перечислены в решении, не имеет значения. Если не было найдено ни одного решения, в файл необходимо записать no solutions.  
Выходные данные на консоль — это доска N\*N, где фигура обозначается #, ее ходы обозначаются \*, а пустые клетки обозначаются 0.

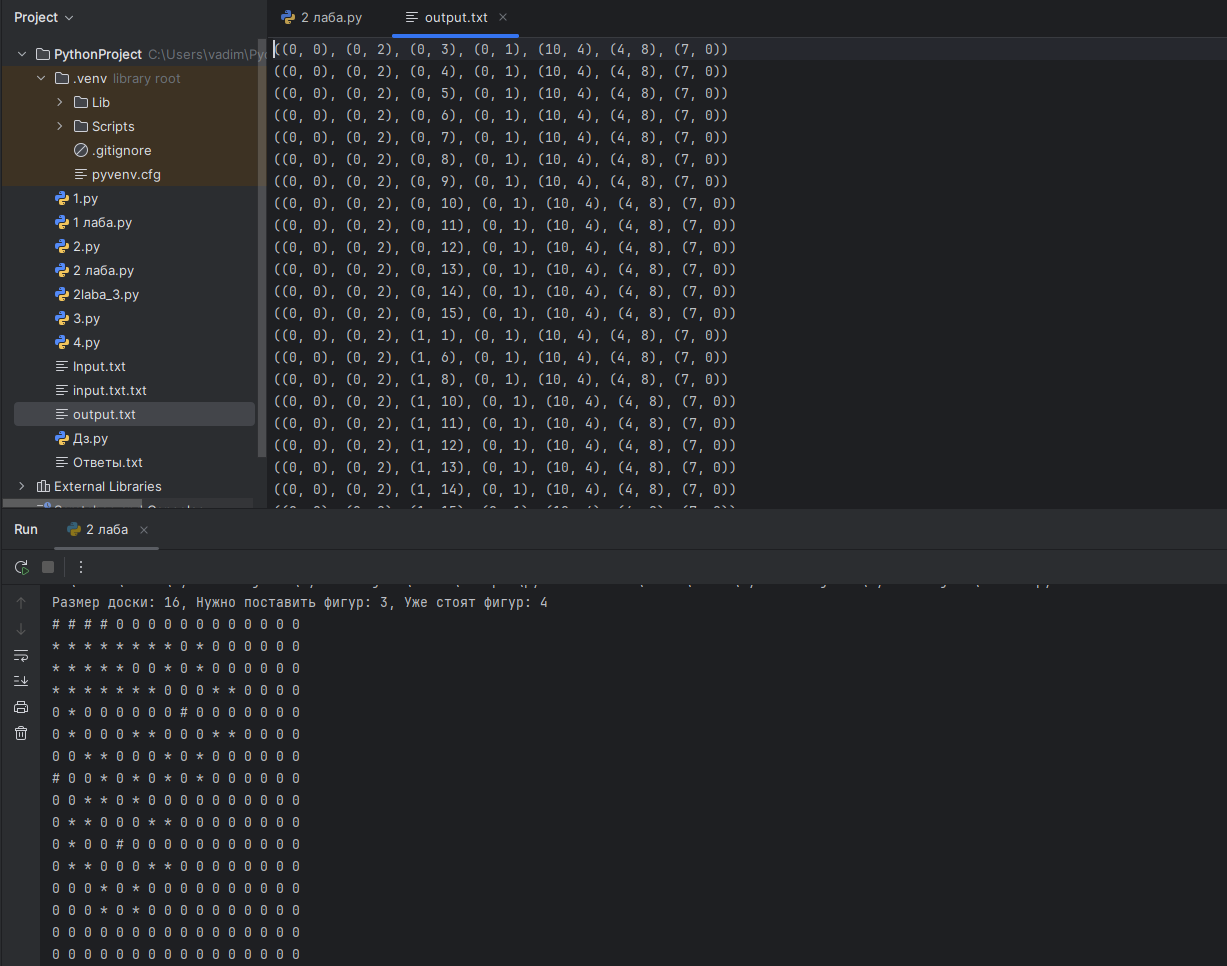
# Результаты работы

## **Реализация программы с использованием функционального программирования языка Python**

## Ход работы

Этот код используется для расстановки фигур (пони) на шахматной доске размером N x N с учетом уже поставленных фигур и выполнения требований по количеству новых фигур. Настоящий процесс включает несколько шагов:  
  
1. Функция read\_input:   
- Открывает файл и считывает значения N (размер доски), L (количество фигур для установки) и K (количество уже установленных фигур).  
- Считывает позиции уже установленных фигур (пони).  
- Возвращает N, L, K и список позиций как кортеж.  
  
2. Функция mark\_attacks:  
- Помечает клетки, на которые могут атаковать пони, из заданной позиции (x, y).  
- Обновляет клетку с позициями пони на '#', чтобы исключить ее из дальнейших перемещений.  
  
3. Функция reset\_tiles:  
- Сбрасывает состояние клетки к исходному значению '0'.  
- Обновляет доску, убирая метки атак (символы '\*') и вновь устанавливает атаки для оставшихся пони.  
  
4. Функция is\_valid\_position:  
- Проверяет, можно ли установить пони в заданной позиции (x, y).  
- Убедитесь, что клетка не находится под атакой других пони.  
  
5. Функция place\_knights:  
- Основная рекурсивная функция для размещения пони.  
- Проверяет все возможные позиции на доске, начиная с текущей.  
- Если позиция допустима, добавляет координаты пони в текущее решение, помечает клетки, куда он может нападать, и рекурсивно продолжает размещение оставшихся фигур.  
- Если удалось установить все L фигур, результат сохраняется и записывается в выходной файл.  
  
6. Функция print\_board:  
- Печатает шахматную доску в удобочитаемом формате.  
  
7. Функция main:  
- Читает входные данные из файла и создает доску.  
- Помечает уже установленные фигуры, используя mark\_attacks.  
- Вызывает place\_knights для начала размещения фигур.  
- После завершения работы читает файл "output.txt". Если решения нет, записывает "no solution". В противном случае извлекает координаты установленных фигур и обновляет доску, помечая атакующие клетки, а затем выводит доску.  
  
8. Запуск программы:  
- Срабатывает условие для запуска main.  
  
Таким образом, программа читает данные, расставляет фигуры на доске, проверяя, чтобы никто из них не находился под ударом, и выводит конечный результат на экран.

## Демонстрация работы программы



## Листинг кода

1. def read\_input(filename: str) -> tuple:  
    # Считывает данные из файла  
    with open(filename) as f:  
    # Читает размеры доски и количество фигур  
    N, L, K = map(int, f.readline().strip().split())  
    # Читает позиции уже установленных фигур  
    pieces = [tuple(map(int, f.readline().strip().split())) for \_ in range(K)]  
    return N, L, K, pieces  
     
     
   def mark\_attacks(x: int, y: int, board: list):  
    # Определяет возможные движения пони  
    possible\_moves = [  
    (2, 1), (2, -1), (-2, 1), (-2, -1),  
    (1, 2), (1, -2), (-1, 2), (-1, -2),  
    (3, 1), (3, -1), (-3, 1), (-3, -1),  
    (1, 3), (1, -3), (-1, 3), (-1, -3)  
    ]  
     
    # Помечает клетки, которые могут быть атакованы  
    for dx, dy in possible\_moves:  
    if 0 <= x + dx < n and 0 <= y + dy < n:  
    board[x + dx][y + dy] = '\*'  
    board[x][y] = '#' # Помечает позицию пони  
     
     
   def reset\_tiles(x: int, y: int, board: list):  
    # Сбрасывает позицию пони  
    board[x][y] = '0' # Возвращает клетку к исходному состоянию  
     
    # Убирает метки атак  
    for i in range(n):  
    for j in range(n):  
    if board[i][j] == '\*':  
    board[i][j] = '0'  
     
    # Восстанавливает атаки для оставшихся пони  
    for i in range(n):  
    for j in range(n):  
    if board[i][j] == '#':  
    mark\_attacks(i, j, board)  
     
     
   def is\_valid\_position(x: int, y: int, board) -> bool:  
    # Проверяет, можно ли установить пони в заданной позиции  
    possible\_moves = [  
    (2, 1), (2, -1), (-2, 1), (-2, -1),  
    (1, 2), (1, -2), (-1, 2), (-1, -2),  
    (3, 1), (3, -1), (-3, 1), (-3, -1),  
    (1, 3), (1, -3), (-1, 3), (-1, -3)  
    ]  
     
    # Проверяет, не находится ли позиция под атакой  
    for dx, dy in possible\_moves:  
    if 0 <= x + dx < n and 0 <= y + dy < n:  
    if board[x + dx][y + dy] == '#' or board[x][y] == '#':  
    return False  
    return True # Позиция допустима  
     
     
   def place\_knights(x: int, y: int, remaining: int, current\_solution: list, existing\_pieces: list, board: list,  
    cache: set):  
    # Если все пони установлены, сохраняем решение  
    if remaining == 0:  
    sorted\_solution = tuple(sorted(current\_solution))  
    if sorted\_solution not in cache: # Проверка на уникальность решения  
    cache.add(sorted\_solution)  
    f.write(f"({', '.join(map(str, current\_solution + existing\_pieces))})\n")  
    return  
     
    # Цикл по всем возможным позициям для размещения пони  
    for i in range(x, n):  
    for j in range(y if i == x else 0, n):  
    if is\_valid\_position(i, j, board): # Проверка на допустимость позиции  
    current\_solution.append((i, j)) # Добавляем текущую позицию в решение  
    mark\_attacks(i, j, board) # Помечаем атакуемые клетки  
    place\_knights(i, j, remaining - 1, current\_solution, existing\_pieces, board, cache) # Рекурсивный вызов  
    reset\_tiles(i, j, board) # Сброс состояния клеток  
    current\_solution.pop() # Удаление последней позиции из текущего решения  
     
     
   def print\_board(board: list):  
    # Печатает доску на экран  
    for row in board:  
    print(' '.join(row))  
     
     
   def main():  
    global n, l, f, const\_pieces  
    n, l, k, const\_pieces = read\_input('Input.txt') # Чтение данных из файла  
    board = [['0'] \* n for \_ in range(n)] # Инициализация доски  
     
    # Помечаем уже установленные фигуры  
    for cx, cy in const\_pieces:  
    mark\_attacks(cx, cy, board)  
     
    with open('output.txt', 'w') as f:  
    cache = set() # Множество для хранения уникальных решений  
    print(f'Размер доски: {n}, Нужно поставить фигур: {l}, Уже стоят фигур: {k}') # Вывод информации  
    place\_knights(0, 0, l, [], const\_pieces, board, cache) # Начинаем размещение пони  
     
    # Чтение результатов из выходного файла  
    with open('output.txt', 'r+') as f:  
    content = f.readline()  
    if not content:  
    f.write('no solution') # Если решений нет  
    else:  
    coordinates = list(map(int, content.replace('(', '').replace(')', '').strip().split(', ')))  
    placements = [(coordinates[i], coordinates[i + 1]) for i in range(0, len(coordinates), 2)]  
    placements.sort() # Сортировка размещений  
     
    # Помечаем атакующие клетки на доске  
    for x, y in placements:  
    mark\_attacks(x, y, board)  
     
    print\_board(board) # Выводим финальную доску  
     
   # Условие для запуска программы  
   if \_\_name\_\_ == '\_\_main\_\_':  
    main()

# 

# ЗАКЛЮЧЕНИЕ

В ходе работы мы научились работать с текстовыми файлами. Научились работать с локальными и глобальными переменными. Научились работать с несколькими функциями, а не только с main. Научились нисходящему методу разработки программ. Написали программу, которая анализирует ходы.

# Список использованных источников