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: Носов Валерий

Проверил:

Санкт-Петербург

2024

Оглавление

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

[2. Результаты работы 5](#_Toc187676634)

[2.1. Реализация программы с использованием функционального программирования языка Python 5](#_Toc187676635)

[2.1.1. Ход работы 5](#_Toc187676636)

[2.1.2. Демонстрация работы программы 14](#_Toc187676637)

[2.1.3. Листинг кода 15](#_Toc187676638)

[ЗАКЛЮЧЕНИЕ 19](#_Toc187676639)

[Список использованных источников 20](#_Toc187676640)

# Цели и формулировка задачи

Цели:

1. Научиться работать с текстовыми файлами;
2. Научиться работать с локальными и глобальными переменными;
3. Научиться работать с несколькими функциям;
4. Изучить нисходящий метод разработки программ;
5. Использовать решение, анализирующее ходы.

Задача:

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: принцесса.

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

Итак, для решения данной задачи составим алгоритм работы программы:

1. Считать данные из файла **input.txt**.
2. Расставить фигуры, данные в условии.
3. Вычислить безопасные клетки (те, в которых фигура не находится под боем).
4. Поставить фигуру в первую безопасную клетку.
   1. Просчитать безопасные клетки
   2. Если безопасные клетки есть, то поставить следующую фигуру, вернуться к (4.1) и повторять, пока не будет расставлено нужное количество фигур.
   3. Если безопасных клеток нет, или расставлено нужное количество фигур, то вернуться к предыдущей фигуре и переставить её в следующую безопасную клетку. Вернуться к (4.1). Повторять, пока не исчерпаны все безопасные клетки для самой первой фигуры.
5. После расстановки всех фигур по всем возможным позициям записать результат в файл **output.txt**.
6. Отрисовать все расстановки фигур.

Первым делом требуется считать входные данные из файла **input.txt**:

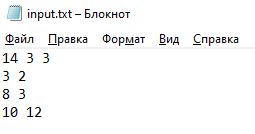


Рис. 1. Содержимое файла **input.txt**.

Для этого я использую контекстный менеджер **with:**

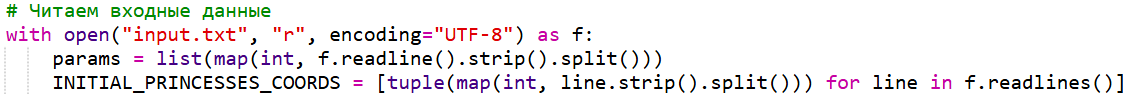


Рис. 2. Считывание входных данных.

Здесь, в переменную **params** я записываю данные с первой строчки файла, то есть **N** (размер доски), **L** (кол-во фигур, которые нужно расставить), **K** (кол-во фигур, которые уже расставлены). В переменную **INITIAL\_PRINCESSES\_COORDS** записываются координаты уже расставленных фигур. Все эти переменные являются **глобальными**, то есть видны в любой части программы, что позволяет не передавать их в качестве параметров функции.

После чего я распределяю входные данные с первой строчки:

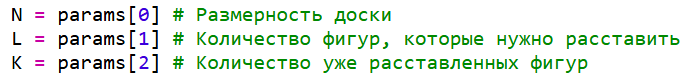


Рис. 3. Распределение входных данных.

Первый шаг завершён, теперь требуется расставить фигуры на доске, а также просчитать безопасные клетки. Для того, чтобы это сделать, нужно вычислить ходы всех этих фигур.

Чтобы вычислить ходы всех фигур, нужно вычислить ходы каждой из этих фигур, а затем объединить их. Для этого я написал функцию **get\_moves():**

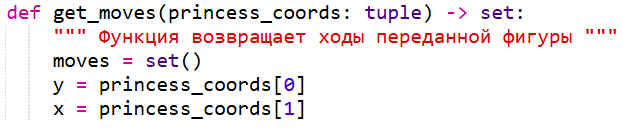


Рис. 4. Функция **get\_moves().**

Данная функция принимает на вход кортеж, содержащий координаты фигуры в формате **(y,x)**, где **y** – индекс по вертикали, а **x** – по горизонтали. И в результате возвращает множество, содержащее координаты ходов.

Кортеж и множество я выбрал из-за их быстродействия.

Далее следуют три вложенных функции **get\_vertical\_moves()**, **get\_horizontal\_moves()**, **get\_diagonal\_moves()**:

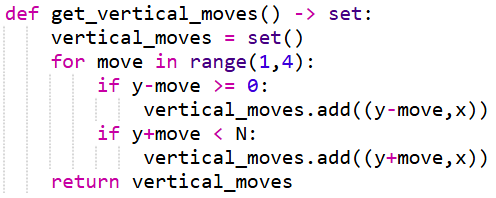


Рис. 5. Функция **get\_vertical\_moves()**.

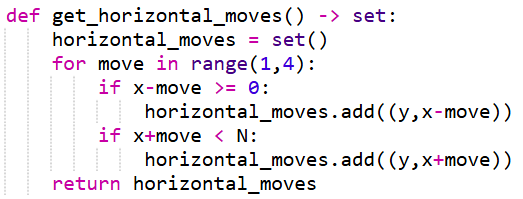


Рис. 6. Функция **get\_horizontal\_moves()**.

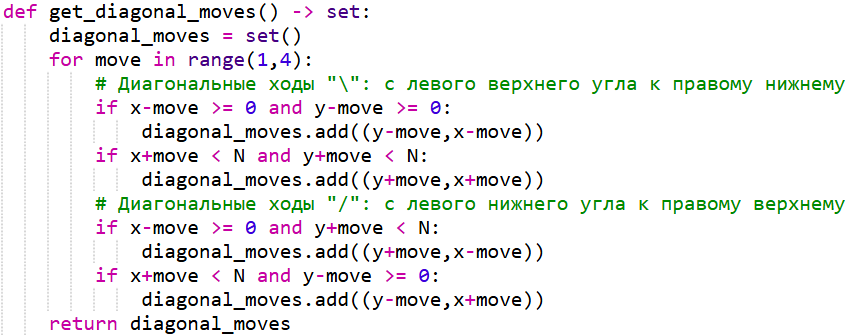


Рис. 7. Функция **get\_diagonal\_moves()**.

Так как принцесса ходит на три клетки, то эти функции создают множества координат, попадающих под ходы данной фигуры, одновременно проверяя, чтобы ходы не выходили за пределы доски.

Далее все эти ходы объединяются в одно множество **moves** и возвращаются:



Рис. 8. Объединение всех ходов и их возврат из функции.

Эта функция просчитывает ходы для одной конкретной фигуры, но мне нужно просчитать ходы для всех фигур. Для этого я написал функцию **get\_all\_moves()**, которая является родительской для функции **get\_moves()**:

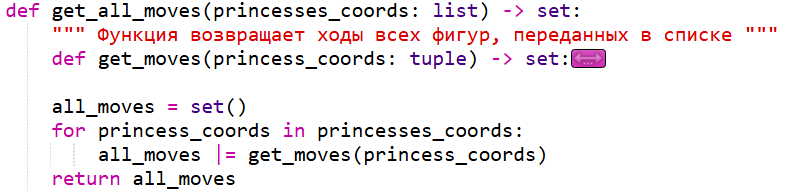


Рис. 9. Функция **get\_all\_moves()**.

Она получается на вход список, содержащий координаты нескольких фигур, для которых нужно просчитать ходы. Всё, что она делает – это проходится циклом по каждой фигуре, получает для неё все ходы с помощью дочерней функции **get\_moves()** и объединяет их в одно множество, после чего возвращает его.

Итак, ходы получены. Следующим шагом будет получить безопасные клетки. Для этого я использую функцию **get\_childs()**:

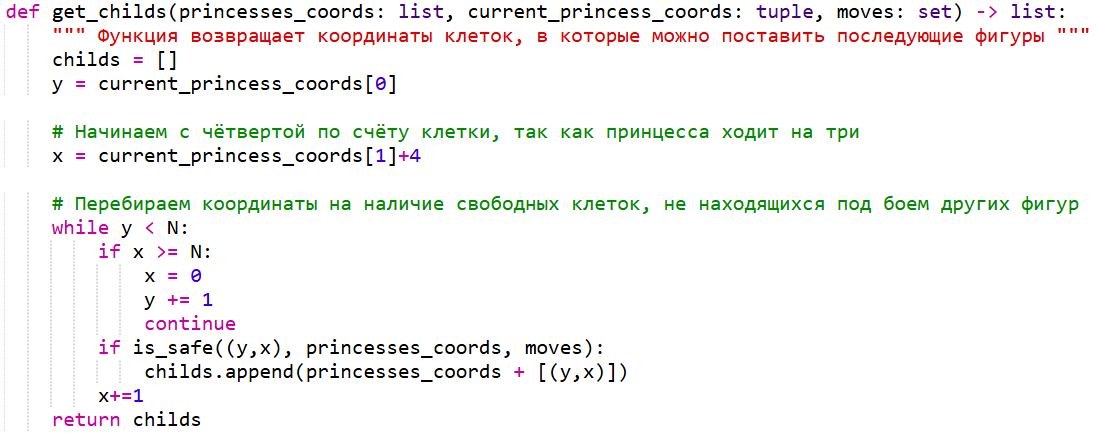


Рис. 10. Функция **get\_childs()**.

Данная функция получает на вход список координат всех расставленных фигур, кортеж с координатами последней поставленной фигуры и множество уже просчитанных ходов, и возвращает список безопасных клеток в формате:

[(3,2),(8,3),(10,12),(0,0),(0,4),(0,8)], где последняя клетка является безопасной, а остальные – клетки, занятые фигурами.

Работа функции заключается в том, что она перебирает все клетки, следующие после поставленной фигуры, причём начинает с четвёртой по счёту клетки, то есть:

Если фигура стоит на координате (3,2), то функция начнёт перебор с клетки (3,6), так как принцесса ходит на три клетки, следовательно клетки (3,3), (3,4), (3,5) будут под боем и проверять их смысла нет.

Далее циклом **while** до конца доски (в нашем случае до клетки с координатами (13,13)), функция перебирает все клетки, проверяя их на безопасность функцией **is\_safe()**, после чего возвращает список с клетками, не находящимися под боем.

Функция **is\_safe()** по сути представляет из себя проверку на принадлежность клетки другой фигуре или ходам:

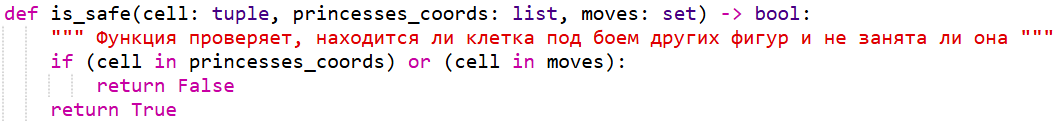


Рис. 11. Функция **is\_safe()**.

Получает на вход координаты клетки, список координат расставленных фигур и множество ходов. Возвращает **True**, если клетка безопасна, иначе – **False**.

Получаем список первоначальных безопасных клеток:



Рис. 12. Получение первоначальных безопасных клеток.

Так как первоначальные безопасные клетки – это потенциальные места, куда будут расставлены первые фигуры, то переменную я использую c именем **all\_first\_princesses**.

Следующий пункт алгоритма под цифрой 4: поставить фигуру в первую безопасную клетку, после чего снова просчитать безопасные клетки, поставить следующую фигуру, и так по кругу, пока не будет достигнуто нужное количество расставленных фигур. Вернуться к предыдущей фигуре и повторять всё вышеперечисленное, пока не дойдём до конца доски.

Всё это осуществляется в рекурсивной функции **solve\_childs()**:

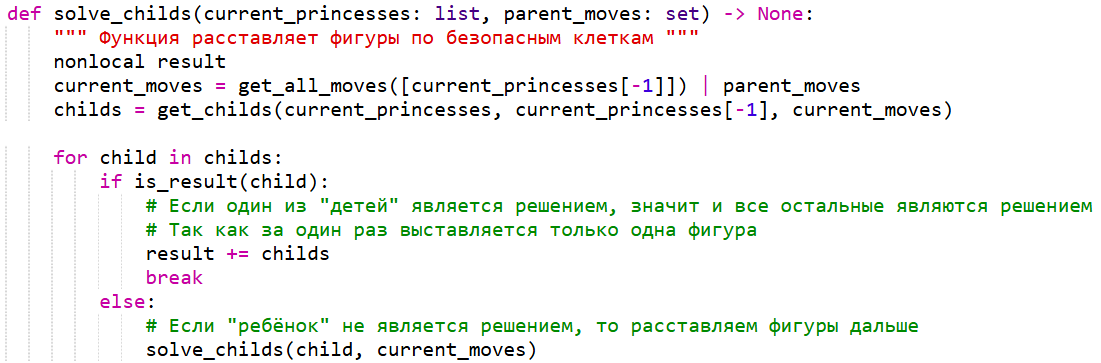


Рис. 13. Функция **solve\_childs()**.

Она получает на вход список расставленных фигур и множество ходов, после чего, для экономии времени, просчитывает ходы только для текущей фигуры и объединяет их с предыдущими. Затем просчитывает безопасные клетки для текущей принцессы и сохраняет их в переменную **childs**.

Затем, в цикле проходится по каждой безопасной клетке и проверяет их функцией **is\_result()**.

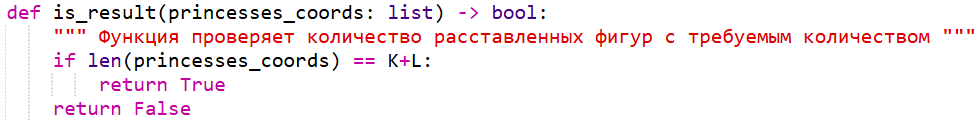


Рис. 14. Функция **is\_result()**.

Так как безопасные клетки записываются в виде списка с координатами всех расставленных ранее фигур, то длина этого списка равна количеству **K** изначальных фигур, заданных по условию (в нашем случае 3) **+** количество **L** фигур, которые нужно расставить. Следовательно, если список содержит в себе **K+L** фигур, то мы расставили нужное количество фигур и он является ответом.

Возвращаясь к условию функции **solve\_childs()**:

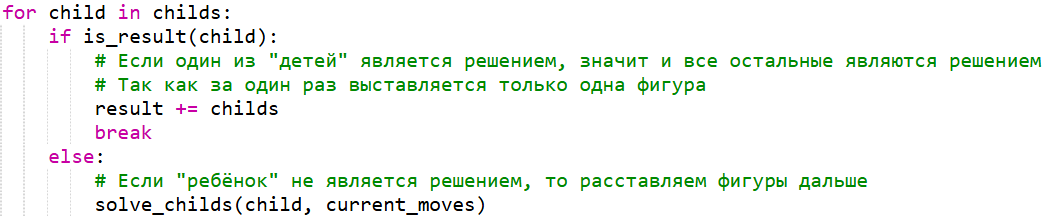


Рис. 15. Цикл функции **solve\_childs()**.

Как и сказано в комментарии, если один детей является решением, то и остальные являются решением. В таком случае я добавляю во внешний список **result** список, содержащий «детей», после чего выходим из цикла и завершаем функцию.

В противном случае, рекурсивно вызываю функцию для данного ребёнка, передавая уже известные ходы, чтобы избежать лишних расчётов.

Эта функция мы выполняется для всех первоначальных безопасных клеток, которые содержаться в **all\_first\_princesses**:

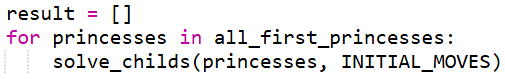


Рис. 16. Применение функции **solve\_childs()** в цикле.

Осталось заключить всё в общую функцию **solve()** и вернуть **result**:

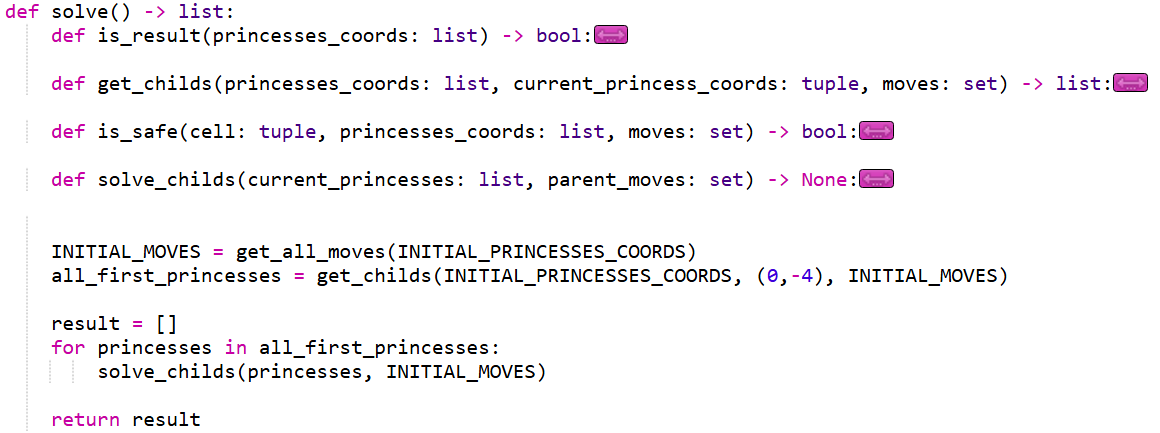


Рис. 17. Функция **solve()**.

После чего записать всё в глобальную **result** и вывести решение в файл **output.txt**, если они есть, иначе записать **no solutions**:

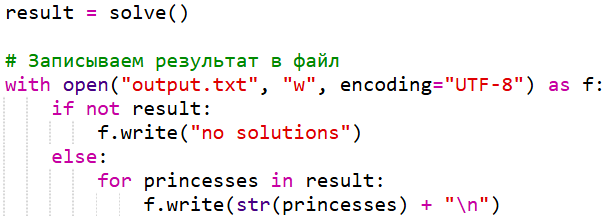


Рис. 18. Запись результата в **output.txt**.

Последним шагом будет отрисовка решения каждой доски с помощью функции **render\_result()**:



Рис. 19. Отрисовка результатов.

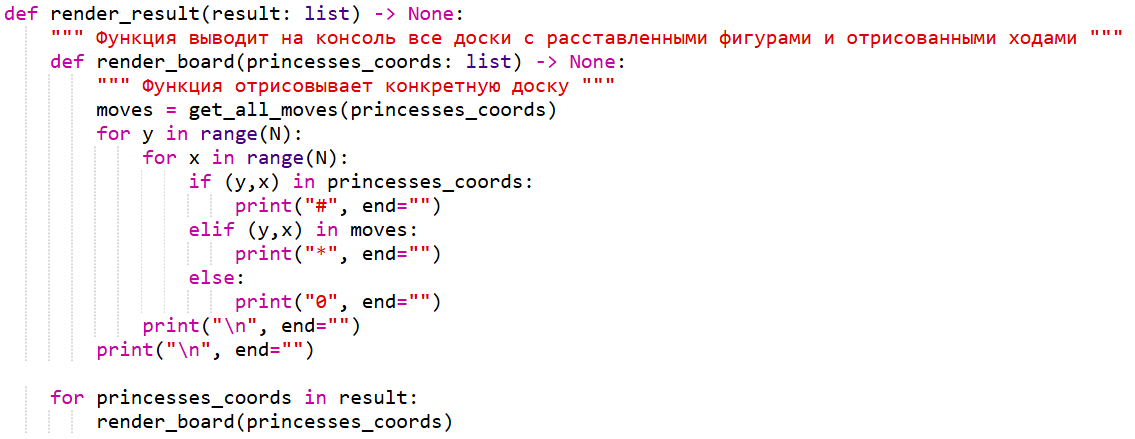


Рис. 20. Функции **render\_result()** и **render\_board()**.

Эта функция принимает на вход список с расставленными фигурами, после чего в цикле вызывает функцию **render\_board()**. Та, свою очередь просчитывает ходы для переданных координат фигуры, и отрисовывает каждую клетку, причём: если клетка принадлежит фигуре, то функция выводит на экран **#**, если клетка принадлежит ходу - **\***, иначе **0**.

После этого программа завершается.

## Демонстрация работы программы

В файле **input.txt** имеем следующие данные:

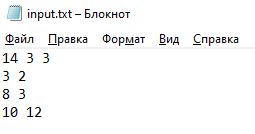


Рис. 21. Входные данные.

Запускаем программу и через секунду получаем решение в **output.txt**:

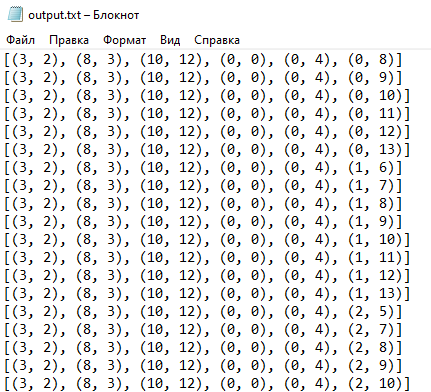


Рис. 22. Выходные данные.

Далее следуют ещё 266+ тысяч строчек.

## Листинг кода

def get\_all\_moves(princesses\_coords: list) -> set:

""" Функция возвращает ходы всех фигур, переданных в списке """

def get\_moves(princess\_coords: tuple) -> set:

""" Функция возвращает ходы переданной фигуры """

moves = set()

y = princess\_coords[0]

x = princess\_coords[1]

def get\_vertical\_moves() -> set:

vertical\_moves = set()

for move in range(1,4):

if y-move >= 0:

vertical\_moves.add((y-move,x))

if y+move < N:

vertical\_moves.add((y+move,x))

return vertical\_moves

def get\_horizontal\_moves() -> set:

horizontal\_moves = set()

for move in range(1,4):

if x-move >= 0:

horizontal\_moves.add((y,x-move))

if x+move < N:

horizontal\_moves.add((y,x+move))

return horizontal\_moves

def get\_diagonal\_moves() -> set:

diagonal\_moves = set()

for move in range(1,4):

# Диагональные ходы "\": с левого верхнего угла к правому нижнему

if x-move >= 0 and y-move >= 0:

diagonal\_moves.add((y-move,x-move))

if x+move < N and y+move < N:

diagonal\_moves.add((y+move,x+move))

# Диагональные ходы "/": с левого нижнего угла к правому верхнему

if x-move >= 0 and y+move < N:

diagonal\_moves.add((y+move,x-move))

if x+move < N and y-move >= 0:

diagonal\_moves.add((y-move,x+move))

return diagonal\_moves

moves = get\_vertical\_moves() | get\_horizontal\_moves() | get\_diagonal\_moves()

return moves

all\_moves = set()

for princess\_coords in princesses\_coords:

all\_moves |= get\_moves(princess\_coords)

return all\_moves

def solve() -> list:

def is\_result(princesses\_coords: list) -> bool:

""" Функция проверяет количество расставленных фигур с требуемым количеством """

if len(princesses\_coords) == K+L:

return True

return False

def get\_childs(princesses\_coords: list, current\_princess\_coords: tuple, moves: set) -> list:

""" Функция возвращает координаты клеток, в которые можно поставить последующие фигуры """

childs = []

y = current\_princess\_coords[0]

# Начинаем с чётвертой по счёту клетки, так как принцесса ходит на три

x = current\_princess\_coords[1]+4

# Перебираем координаты на наличие свободных клеток, не находящихся под боем других фигур

while y < N:

if x >= N:

x = 0

y += 1

continue

if is\_safe((y,x), princesses\_coords, moves):

childs.append(princesses\_coords + [(y,x)])

x+=1

return childs

def is\_safe(cell: tuple, princesses\_coords: list, moves: set) -> bool:

""" Функция проверяет, находится ли клетка под боем других фигур и не занята ли она """

if (cell in princesses\_coords) or (cell in moves):

return False

return True

def solve\_childs(current\_princesses: list, parent\_moves: set) -> None:

""" Функция расставляет фигуры по безопасным клеткам """

nonlocal result

current\_moves = get\_all\_moves([current\_princesses[-1]]) | parent\_moves

childs = get\_childs(current\_princesses, current\_princesses[-1], current\_moves)

for child in childs:

if is\_result(child):

# Если один из "детей" является решением, значит и все остальные являются решением

# Так как за один раз выставляется только одна фигура

result += childs

break

else:

# Если "ребёнок" не является решением, то расставляем фигуры дальше

solve\_childs(child, current\_moves)

INITIAL\_MOVES = get\_all\_moves(INITIAL\_PRINCESSES\_COORDS)

all\_first\_princesses = get\_childs(INITIAL\_PRINCESSES\_COORDS, (0,-4), INITIAL\_MOVES)

result = []

for princesses in all\_first\_princesses:

solve\_childs(princesses, INITIAL\_MOVES)

return result

def render\_result(result: list) -> None:

""" Функция выводит на консоль все доски с расставленными фигурами и отрисованными ходами """

def render\_board(princesses\_coords: list) -> None:

""" Функция отрисовывает конкретную доску """

moves = get\_all\_moves(princesses\_coords)

for y in range(N):

for x in range(N):

if (y,x) in princesses\_coords:

print("#", end="")

elif (y,x) in moves:

print("\*", end="")

else:

print("0", end="")

print("\n", end="")

print("\n", end="")

for princesses\_coords in result:

render\_board(princesses\_coords)

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

# Читаем входные данные

with open("input.txt", "r", encoding="UTF-8") as f:

params = list(map(int, f.readline().strip().split()))

INITIAL\_PRINCESSES\_COORDS = [tuple(map(int, line.strip().split())) for line in f.readlines()]

N = params[0] # Размерность доски

L = params[1] # Количество фигур, которые нужно расставить

K = params[2] # Количество уже расставленных фигур

result = solve()

# Записываем результат в файл

with open("output.txt", "w", encoding="UTF-8") as f:

if not result:

f.write("no solutions")

else:

for princesses in result:

f.write(str(princesses) + "\n")

#render\_result(result) # Отрисовать все результаты

# ЗАКЛЮЧЕНИЕ

# Для выполнения этой работы использовались следуюшие методы: 1) Работа с файлами. Входные данные читались из файла, после чего ответ так же записывался в файл. Такой подход позволяет не взаимодествовать с программой через терминал и не менять значение внутри самой программы, чтобы получить ответ. Вместо этого нужно просто задать входные данные в файл, после чего получить ответ в выходном файле, что очень удобно.

2) **Работа с глобальными и локальными переменными.** Использование глобальных переменных может быть выгодным в программе со множеством функций, так как в таком случае не нужно будет передавать эти значения в качестве параметров функции, что делает код более читабельным. В свою очередь локальные переменные позволяют экономить память, так как они существуют только тогда, когда исполняется функция. Как только функция завершает своё выполнение, локальная перменная удаляется из памяти, освобождая её.

3) **Работа с несколькими функциями.** Разбиение кода на отдельные функции во много раз увеличивает читабельность и понимание кода. Кроме того, это позволяет выполнять одни и те же действия, просто вызвав функцию вместо того, чтобы прописывать код целиком, что так же способствует укорачиванию кода.

4) **Использование нисходящего метода разработки программ.** Данный метод подразумевает разбиение одного высокого уровня на более низкие, что позволяет упростить разработку. Это помогает лучше понять работу программе и изменять отдельные её части.

5) **Анализ ходов.** Такой подход позволяет очень сильно сократить время на перебор всех возможных вариантов. К тому же такое решение выглядит более понятным.

Таким образом, объединив все эти методы, мне удалось написать программу, которая решает поставленную задачу.

# Список использованных источников