|  |  |
| --- | --- |
| Picture 1 | **МИНОБРНАУКИ РОССИИ**  **федеральное государственное бюджетное образовательное учреждение**  **высшего образования**  «Санкт-Петербургский государственный морской технический университет» (СПбГМТУ) |

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Факультет цифровых промышленных технологий

Направление подготовки 09.03.01.03 "Интеллектуальные технологии киберфизических систем"

**«Контрольный Проект»**

Студент 1 курса группы 20121

Очного отделения

Трапезников А. Н.

Проверил:

Поделенюк П. П.

2024

# Оглавление

[Оглавление 2](#_Toc170313153)

[Цель работы 3](#_Toc170313154)

[Разработка классов и UML диаграмма 3](#_Toc170313155)

[Результаты работы 4](#_Toc170313156)

[Реализация программы с использованием функционального программирования 4](#_Toc170313157)

[Ход работы 4](#_Toc170313158)

[Демонстрация работы программы 5](#_Toc170313159)

[Листинг кода 7](#_Toc170313160)

[Реализация программы с использованием ООП языка Python 9](#_Toc170313161)

[Ход работы 9](#_Toc170313162)

[Демонстрация работы программы 17](#_Toc170313163)

[Листинг кода 19](#_Toc170313164)

[Заключение 27](#_Toc170313165)

# Цель работы

Целью данной лабораторной работы является разработка и оптимизация алгоритма для размещения дополнительных фигур на шахматной доске таким образом, чтобы никакая фигура не находилась под боем другой. А также создание GUI приложения.

# Разработка классов и UML диаграмма

1. Чтение входных данных:

Класс MainWindow отображает главное окно приложения, где пользователь вводит размер доски N и количество фигур L.

По нажатию кнопки "Далее" происходит валидация введенных данных и открывается окно для ввода координат уже расставленных фигур с помощью класса InputDialog.

1. Инициализация доски и размещение известных фигур

Класс InputDialog предоставляет интерфейс для ввода координат фигур пользователем. Он отображает доску и позволяет пользователю добавлять или удалять фигуры.

Метод handle\_click обрабатывает клики мыши для добавления или удаления фигур.

Открывает диалоговое окно SolutionDialog для показа итогового решения.

1. Создание доски и размещение фигур

Класс Board отвечает за создание и управление состоянием доски.

Класс Piece содержит методы для получения возможных ходов фигур и их размещения на доске.

Метод ChessLogic.create\_board создает доску с указанными фигурами и возвращает её.

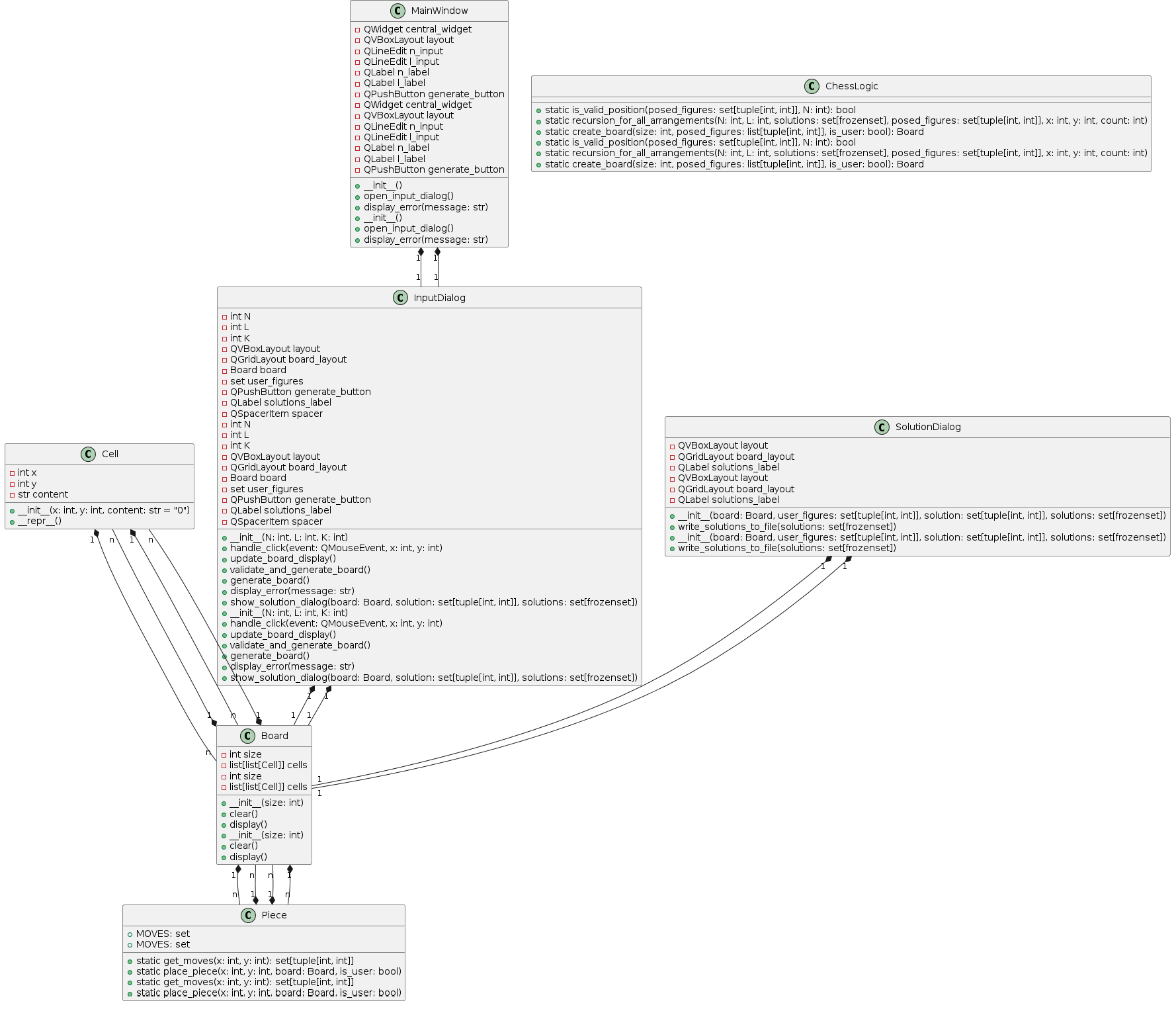
1. Оптимизированный поиск возможных решений

Класс ChessLogic содержит метод recursion\_for\_all\_arrangements, который рекурсивно проверяет все возможные комбинации размещения L фигур на доске размера N x N и сохраняет найденные решения.

1. Вывод доски и результатов

Класс SolutionDialog отображает доску с найденными решениями и обновляет метку с количеством решений.

Метод write\_solutions\_to\_file записывает найденные решения в файл output.txt.

UML диаграмма:

# Результаты работы

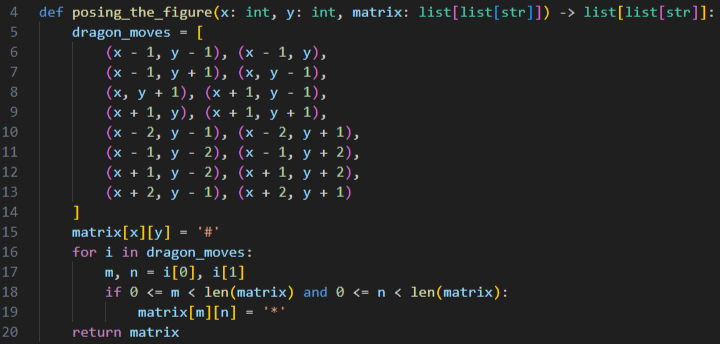
## Реализация программы с использованием функционального программирования (прошлый семестр)

### Ход работы

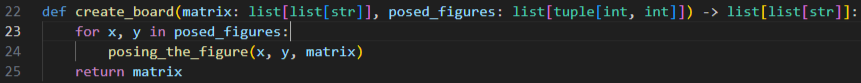
Была написана функция, которая создает доску в виде матрицы:



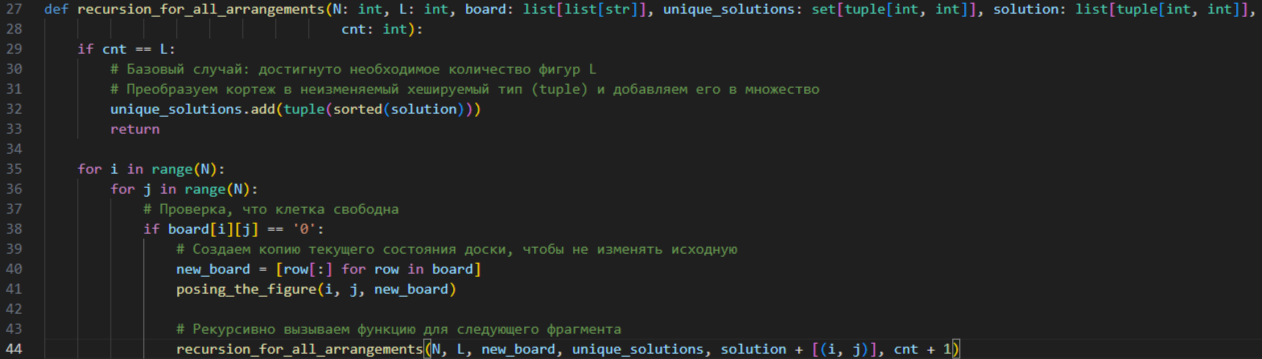
Также была создана функция, которая ставит на доску фигуру (#) и обозначает клетки, которые эта фигура бьет (\*):



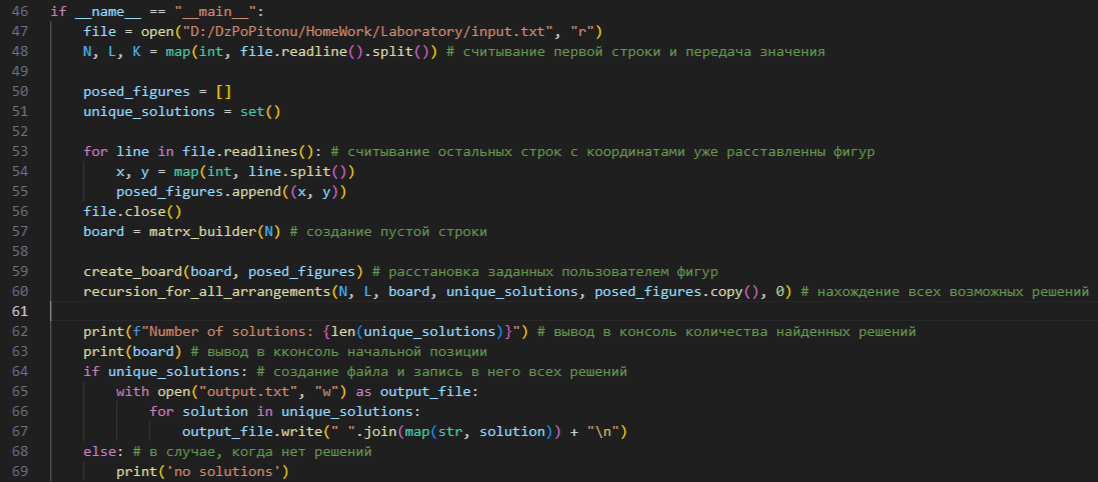
Была создана функция для создания начальной позиции, учитывая расставленные фигуры:



Далее была создана основная функция, которая расставляет фигуры на доске всеми возможными способами и заносит ответ в unique\_solutions, используя при это рекурсию:



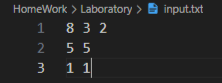
Основная часть кода с считыванием с input.txt, созданием output.txt, использованием вышеуказанных функций и выводом в консоль количества решений и начальной позиции:



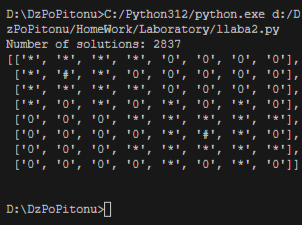
### Демонстрация работы программы

**Когда решения есть**

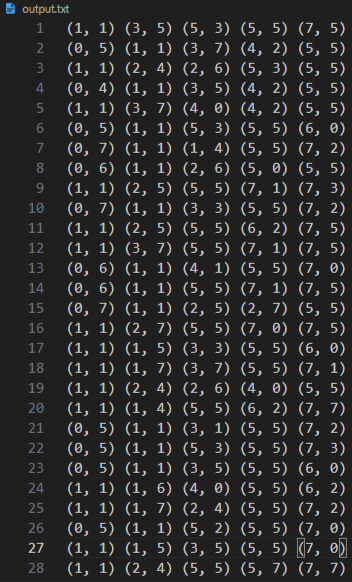
Input.txt



Консоль

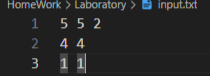


Часть файла Output.txt

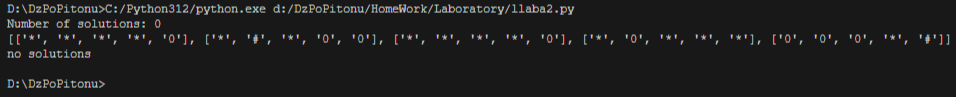


**Когда решений нет**

Input.txt



Консоль



Output.txt не создан

### Листинг кода

def matrx\_builder(N: int) -> list[list[str]]:

    return [['0' for \_ in range(N)] for \_ in range(N)]

def piece\_moves(x, y):

    moves = {

        (x - 1, y - 1), (x - 1, y),

        (x - 1, y + 1), (x, y - 1),

        (x, y + 1), (x + 1, y - 1),

        (x + 1, y), (x + 1, y + 1),

        (x - 2, y - 1), (x - 2, y + 1),

        (x - 1, y - 2), (x - 1, y + 2),

        (x + 1, y - 2), (x + 1, y + 2),

        (x + 2, y - 1), (x + 2, y + 1)

    }

    return moves

def posing\_the\_figure(x: int, y: int, matrix: list[list[str]]) -> list[list[str]]:

    dragon\_moves = [

        (x - 1, y - 1), (x - 1, y),

        (x - 1, y + 1), (x, y - 1),

        (x, y + 1), (x + 1, y - 1),

        (x + 1, y), (x + 1, y + 1),

        (x - 2, y - 1), (x - 2, y + 1),

        (x - 1, y - 2), (x - 1, y + 2),

        (x + 1, y - 2), (x + 1, y + 2),

        (x + 2, y - 1), (x + 2, y + 1)

    ]

    matrix[x][y] = '#'

    for i in dragon\_moves:

        m, n = i[0], i[1]

        if 0 <= m < len(matrix) and 0 <= n < len(matrix):

            matrix[m][n] = '\*'

    return matrix

def create\_board(matrix: list[list[str]], posed\_figures: list[tuple[int, int]]) -> list[list[str]]:

    for x, y in posed\_figures:

        posing\_the\_figure(x, y, matrix)

    return matrix

def print\_board(matrix: list[list[str]]):

    for row in matrix:

        print(" ".join(row))

def recursion\_for\_all\_arrangements(N: int, L: int, solutions: set[tuple[int, int]], solution: set[tuple[int, int]], cnt: int, last\_x: int, last\_y: int):

    if cnt == L:

        unique\_solution = tuple(solution)

        solutions.add(unique\_solution)

        # Вывод первого решения

        if len(solutions) == 1:

            print("First solution:")

            print\_board(create\_board(matrx\_builder(N), unique\_solution))

        return

    # (last\_x, last\_y) - координаты последней поставленной фигуры

    for i in range(last\_x, N):

        if i == last\_x: # если на этой строке уже поставлена фигура, то проход идет начиная с координат последней фигуры

            start\_y = last\_y

        else: # иначе будет проход всей строки с начала

            start\_y = 0

        for j in range(start\_y, N):

            if (i, j) not in solution and not piece\_moves(i, j).intersection(solution):

                solution.add((i, j))

                recursion\_for\_all\_arrangements(N, L, solutions, solution, cnt + 1, i, j)

                solution.remove((i, j))

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

    file = open("D:/DzPoPitonu/HomeWork/Laboratory/input.txt", "r")

    N, L, K = map(int, file.readline().split())

    posed\_figures = set()

    solutions = set()

    for line in file.readlines():

        x, y = map(int, line.split())

        posed\_figures.add((x, y))

    file.close()

    recursion\_for\_all\_arrangements(N, L, solutions, posed\_figures, 0, 0, 0)

    print(f"Number of solutions: {len(solutions)}")

    if solutions:

        solutions\_str = [" ".join(map(str, solution)) + "\n" for solution in solutions]

        with open("D:/DzPoPitonu/HomeWork/Laboratory/output.txt", "w") as output\_file:

            output\_file.writelines(solutions\_str)

    else:

        print('no solutions')

## Реализация программы с использованием ООП языка Python

### Ход работы

Был написан class Cell, который отвечает за координаты клетки и ее содержание.

class Cell: # класс клетки

    def \_\_init\_\_(self, x: int, y: int, content: str = "0"):

        self.x = x

        self.y = y

        self.content = content

    def \_\_repr\_\_(self):

        return self.content

class Piece: отвечает за фигуру, то есть, за ее возможные ходы и размещение

    class Piece: # класс фигуры и ее ходов

    MOVES = {

        (-1, -1),(-1, 0),(-1, 1),(0, -1),

        (0, 1),(1, -1),(1, 0),(1, 1),

        (-2, -1),(-2, 1),(-1, -2),(-1, 2),

        (1, -2),(1, 2),(2, -1),(2, 1),

    }

    @staticmethod

    def get\_moves(

        x: int, y: int

    ) -> set[tuple[int, int]]:  # находит возможные ходы для фигуры

        moves = {

            (x - 1, y - 1),(x - 1, y),

            (x - 1, y + 1),

            (x, y - 1),(x, y + 1),

            (x + 1, y - 1),(x + 1, y),

            (x + 1, y + 1),(x - 2, y - 1),

            (x - 2, y + 1),(x - 1, y - 2),

            (x - 1, y + 2),(x + 1, y - 2),

            (x + 1, y + 2),

            (x + 2, y - 1),(x + 2, y + 1),

        }

        return moves

    @staticmethod

    def place\_piece(x: int, y: int, board: "Board", is\_user: bool): # размещает фигуру

        content = "U" if is\_user else "#"

        board.cells[x][y].content = content

        for dx, dy in Piece.MOVES:

            nx, ny = x + dx, y + dy

            if 0 <= nx < board.size and 0 <= ny < board.size:

                if board.cells[nx][ny].content == "0":

                    board.cells[nx][ny].content = "\*"

class Board: отвечает за доску. Имеет метод сбрасывания всех клеток, метод для показа текущего состояния.

class Board: # класс для доски

    def \_\_init\_\_(self, size: int):

        self.size = size

        self.cells = [[Cell(x, y) for y in range(size)] for x in range(size)]

    def clear(self): # сбрасывает все клетки

        self.cells = [[Cell(x, y) for y in range(self.size)] for x in range(self.size)]

    def display(self): # показывает текущее состояние

        for row in self.cells:

            print(" ".join(str(cell) for cell in row))

class ChessLogic отвечает за логику программы(проверка позиций и генерация решений

class ChessLogic: # класс с логикой и проверкой позиций и генерации решений

    @staticmethod

    def is\_valid\_position(posed\_figures: set[tuple[int, int]], N: int) -> bool: # проверяет допустимость расположения фигуры

        for x, y in posed\_figures:

            for move in Piece.get\_moves(x, y):

                if move in posed\_figures:

                    return False

        return True

    @staticmethod

    def recursion\_for\_all\_arrangements(

        N: int,

        L: int,

        solutions: set[frozenset],

        posed\_figures: set[tuple[int, int]],

        x: int,

        y: int,

        count: int,

    ): # находит все решения расстановки фигур

        if count == L:

            solutions.add(frozenset(posed\_figures))

            return

        for i in range(x, N):

            for j in range(y if i == x else 0, N):

                if (i, j) not in posed\_figures and ChessLogic.is\_valid\_position(

                    posed\_figures | {(i, j)}, N

                ):

                    new\_posed\_figures = posed\_figures | {(i, j)}

                    ChessLogic.recursion\_for\_all\_arrangements(

                        N, L, solutions, new\_posed\_figures, i, j + 1, count + 1

                    )

    @staticmethod

    def create\_board(

        size: int, posed\_figures: list[tuple[int, int]], is\_user: bool

    ) -> Board: # создает доску с указанными фигурами

        board = Board(size)

        for x, y in posed\_figures:

            Piece.place\_piece(x, y, board, is\_user)

        return board

create\_board(matrix, posed\_figures): Создает доску matrix с уже размещенными фигурами из posed\_figures.

@staticmethod

    def create\_board(

        matrix: list[list[str]], posed\_figures: list[tuple[int, int]]

    ) -> list[list[str]]: # создание доски с расстановкой фигур

        board = matrix

        for x, y in posed\_figures:

            board = ChessLogic.posing\_the\_figure(x, y, board)

        return board

class InputDialog(QDialog) отвечает за ввод данных пользователем. Также открывает SolutionDialog.

# класс диалогового окна для ввода координат

class InputDialog(QDialog):

    def \_\_init\_\_(self, N: int, L: int, K: int):

        super().\_\_init\_\_()

        self.setWindowTitle("Ввод координат")

        self.setSizePolicy(QSizePolicy.Expanding, QSizePolicy.Expanding)

        self.layout = QVBoxLayout()

        self.N = N

        self.L = L

        self.K = K

        self.board\_layout = QGridLayout()

        self.board = Board(N)

        self.user\_figures = set()

        for i in range(N):

            for j in range(N):

                label = QLabel()

                label.setStyleSheet(

                    "background-color: white; border: 1px solid black; min-width: 20px; min-height: 20px;"

                )

                label.setAlignment(Qt.AlignCenter)

                label.setSizePolicy(QSizePolicy.Expanding, QSizePolicy.Expanding)

                label.mousePressEvent = lambda event, x=i, y=j: self.handle\_click(

                    event, x, y

                )

                self.board\_layout.addWidget(label, i, j)

        self.layout.addLayout(self.board\_layout)

        self.generate\_button = QPushButton("Сгенерировать доску")

        self.generate\_button.clicked.connect(self.validate\_and\_generate\_board)

        self.layout.addWidget(self.generate\_button)

        self.solutions\_label = QLabel("")

        self.layout.addWidget(self.solutions\_label)

        spacer = QSpacerItem(20, 40, QSizePolicy.Minimum, QSizePolicy.Expanding)

        self.layout.addItem(spacer)

        self.setLayout(self.layout)

    # метод для обработки клика по ячейке

    def handle\_click(self, event: QMouseEvent, x: int, y: int):

        if event.button() == Qt.LeftButton:

            if (x, y) not in self.user\_figures and ChessLogic.is\_valid\_position(

                self.user\_figures | {(x, y)}, self.N

            ):

                self.user\_figures.add((x, y))

                self.update\_board\_display()

        elif event.button() == Qt.RightButton and (x, y) in self.user\_figures:

            self.user\_figures.remove((x, y))

            self.update\_board\_display()

    # метод для обновления отображения доски

    def update\_board\_display(self):

        self.board.clear()

        for x, y in self.user\_figures:

            Piece.place\_piece(x, y, self.board, True)

        for i in range(self.N):

            for j in range(self.N):

                label = self.board\_layout.itemAtPosition(i, j).widget()

                cell\_content = self.board.cells[i][j].content

                if cell\_content == "U":

                    label.setStyleSheet(

                        "background-color: yellow; border: 1px solid black; min-width: 20px; min-height: 20px;"

                    )

                elif cell\_content == "\*":

                    label.setStyleSheet(

                        "background-color: red; border: 1px solid black; min-width: 20px; min-height: 20px;"

                    )

                else:

                    label.setStyleSheet(

                        "background-color: white; border: 1px solid black; min-width: 20px; min-height: 20px;"

                    )

    # метод для проверки корректности ввода и генерации доски

    def validate\_and\_generate\_board(self):

        if not ChessLogic.is\_valid\_position(self.user\_figures, self.N):

            self.display\_error("Некорректное расположение фигур")

            return False

        self.generate\_board()

        return True

    # метод для генерации доски

    def generate\_board(self):

        solutions = set()

        ChessLogic.recursion\_for\_all\_arrangements(

            self.N, self.L, solutions, self.user\_figures, 0, 0, 0

        )

        if solutions:

            solution = next(iter(solutions))

            self.show\_solution\_dialog(Board(self.N), solution, solutions)

        else:

            self.display\_error("Решений не найдено")

    # метод для отображения сообщения об ошибке

    def display\_error(self, message: str):

        QMessageBox.critical(self, "Ошибка", message)

    # метод для отображения итоговой доски

    def show\_solution\_dialog(

        self, board: Board, solution: set[tuple[int, int]], solutions: set[frozenset]

    ):

        dialog = SolutionDialog(board, self.user\_figures, solution, solutions)

        dialog.exec()

class SolutionDialog отвечает за отображение итога и записи всех решений в output.txt

# класс диалогового окна для отображения итоговой доски

class SolutionDialog(QDialog):

    def \_\_init\_\_(

        self,

        board: Board,

        user\_figures: set[tuple[int, int]],

        solution: set[tuple[int, int]],

        solutions: set[frozenset],

    ):

        super().\_\_init\_\_()

        self.setWindowTitle("Итоговая доска")

        self.setSizePolicy(QSizePolicy.Expanding, QSizePolicy.Expanding)

        self.layout = QVBoxLayout()

        self.board\_layout = QGridLayout()

        # размещаем фигуры пользователя на доске

        for x, y in user\_figures:

            Piece.place\_piece(x, y, board, True)

        # размещаем фигуры алгоритма на доске

        for x, y in solution:

            Piece.place\_piece(x, y, board, False)

        # отображаем доску с выделением отличий

        for i in range(board.size):

            for j in range(board.size):

                label = QLabel()

                cell\_content = board.cells[i][j].content

                if (i, j) in user\_figures:

                    label.setStyleSheet(

                        "background-color: yellow; border: 1px solid black; min-width: 20px; min-height: 20px;"

                    )

                elif cell\_content == "#":

                    label.setStyleSheet(

                        "background-color: blue; border: 1px solid black; min-width: 20px; min-height: 20px;"

                    )

                elif cell\_content == "\*":

                    label.setStyleSheet(

                        "background-color: red; border: 1px solid black; min-width: 20px; min-height: 20px;"

                    )

                else:

                    label.setStyleSheet(

                        "background-color: white; border: 1px solid black; min-width: 20px; min-height: 20px;"

                    )

                label.setAlignment(Qt.AlignCenter)

                label.setSizePolicy(QSizePolicy.Expanding, QSizePolicy.Expanding)

                self.board\_layout.addWidget(label, i, j)

        self.layout.addLayout(self.board\_layout)

        self.solutions\_label = QLabel(f"Количество решений: {len(solutions)}")

        self.layout.addWidget(self.solutions\_label)

        self.setLayout(self.layout)

        self.write\_solutions\_to\_file(solutions)

    # запись решений в output.txt

    def write\_solutions\_to\_file(self, solutions):

        with open("output.txt", "w") as file:

            for solution in solutions:

                line = " ".join(f"({x},{y})" for x, y in solution)

                file.write(line + "\n")

class MainWindow(QMainWindow): отвечает за основное окно приложения.

class MainWindow(QMainWindow): # основное окно приложения

    def \_\_init\_\_(self):

        super().\_\_init\_\_()

        self.setWindowTitle("Расстановка фигур")

        self.setSizePolicy(QSizePolicy.Expanding, QSizePolicy.Expanding)

        self.central\_widget = QWidget()

        self.layout = QVBoxLayout()

        self.n\_input = QLineEdit()

        self.l\_input = QLineEdit()

        self.n\_label = QLabel("Размер доски:")

        self.l\_label = QLabel("Количество фигур:")

        self.layout.addWidget(self.n\_label)

        self.layout.addWidget(self.n\_input)

        self.layout.addWidget(self.l\_label)

        self.layout.addWidget(self.l\_input)

        self.generate\_button = QPushButton("Далее")

        self.generate\_button.clicked.connect(self.open\_input\_dialog)

        self.layout.addWidget(self.generate\_button)

        self.central\_widget.setLayout(self.layout)

        self.setCentralWidget(self.central\_widget)

    def open\_input\_dialog(self): # открывает окно для ввода размера доски и кол-ва фигур

        try:

            N = int(self.n\_input.text())

            L = int(self.l\_input.text())

            K = 0

            if N <= 0 or L <= 0 or K < 0 or K >= N\*\*2 or L + K > N\*\*2:

                raise ValueError

            dialog = InputDialog(N, L, K)

            dialog.exec()

        except ValueError:

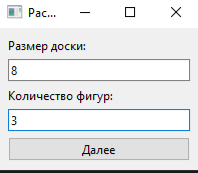
            self.display\_error("Некорректный ввод")

    def display\_error(self, message: str): # отображение ошибки

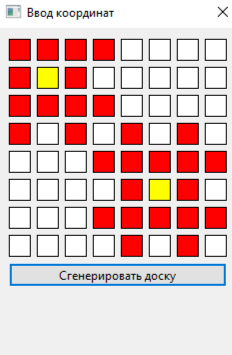
        QMessageBox.critical(self, "Ошибка", message)

### Демонстрация работы программы

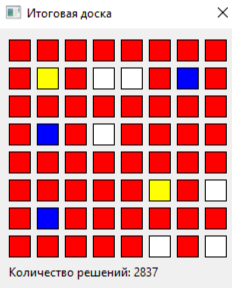
Ввод данных пользователя (размер доски, количество фигур)



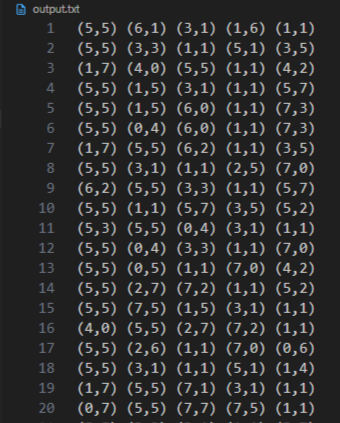
Ввод координат уже расставленных фигур



Получение результата



Часть файла output.txt



### Листинг кода

import sys

from PySide6.QtWidgets import (

    QApplication,

    QMainWindow,

    QWidget,

    QVBoxLayout,

    QPushButton,

    QLineEdit,

    QLabel,

    QGridLayout,

    QDialog,

    QSizePolicy,

    QSpacerItem,

    QMessageBox,

)

from PySide6.QtCore import Qt

from PySide6.QtGui import QMouseEvent

# класс ячейки, содержащий координаты и содержимое ячейки

class Cell:

    def \_\_init\_\_(self, x: int, y: int, content: str = "0"):

        self.x = x

        self.y = y

        self.content = content

    def \_\_repr\_\_(self):

        return self.content

# класс фигуры, содержащий возможные ходы и методы для размещения фигур

class Piece:

    MOVES = {

        (-1, -1),

        (-1, 0),

        (-1, 1),

        (0, -1),

        (0, 1),

        (1, -1),

        (1, 0),

        (1, 1),

        (-2, -1),

        (-2, 1),

        (-1, -2),

        (-1, 2),

        (1, -2),

        (1, 2),

        (2, -1),

        (2, 1),

    }

    @staticmethod

    def get\_moves(x: int, y: int) -> set[tuple[int, int]]:

        moves = {

            (x - 1, y - 1),

            (x - 1, y),

            (x - 1, y + 1),

            (x, y - 1),

            (x, y + 1),

            (x + 1, y - 1),

            (x + 1, y),

            (x + 1, y + 1),

            (x - 2, y - 1),

            (x - 2, y + 1),

            (x - 1, y - 2),

            (x - 1, y + 2),

            (x + 1, y - 2),

            (x + 1, y + 2),

            (x + 2, y - 1),

            (x + 2, y + 1),

        }

        return moves

    # метод для размещения фигуры на доске

    @staticmethod

    def place\_piece(x: int, y: int, board: "Board", is\_user: bool):

        content = "U" if is\_user else "#"

        board.cells[x][y].content = content

        for dx, dy in Piece.MOVES:

            nx, ny = x + dx, y + dy

            if 0 <= nx < board.size and 0 <= ny < board.size:

                if board.cells[nx][ny].content == "0":

                    board.cells[nx][ny].content = "\*"

# класс доски, содержащий ячейки и методы для их отображения

class Board:

    def \_\_init\_\_(self, size: int):

        self.size = size

        self.cells = [[Cell(x, y) for y in range(size)] for x in range(size)]

    # метод для очистки доски

    def clear(self):

        self.cells = [[Cell(x, y) for y in range(self.size)] for x in range(self.size)]

    # метод для отображения доски в консоли

    def display(self):

        for row in self.cells:

            print(" ".join(str(cell) for cell in row))

# класс для логики игры

class ChessLogic:

    # метод для проверки корректности позиции

    @staticmethod

    def is\_valid\_position(posed\_figures: set[tuple[int, int]], N: int) -> bool:

        for x, y in posed\_figures:

            for move in Piece.get\_moves(x, y):

                if move in posed\_figures:

                    return False

        return True

    # рекурсивный метод для поиска всех возможных расстановок фигур

    @staticmethod

    def recursion\_for\_all\_arrangements(

        N: int,

        L: int,

        solutions: set[frozenset],

        posed\_figures: set[tuple[int, int]],

        x: int,

        y: int,

        count: int,

    ):

        if count == L:

            solutions.add(frozenset(posed\_figures))

            return

        for i in range(x, N):

            for j in range(y if i == x else 0, N):

                if (i, j) not in posed\_figures and ChessLogic.is\_valid\_position(

                    posed\_figures | {(i, j)}, N

                ):

                    new\_posed\_figures = posed\_figures | {(i, j)}

                    ChessLogic.recursion\_for\_all\_arrangements(

                        N, L, solutions, new\_posed\_figures, i, j + 1, count + 1

                    )

    # метод для создания доски с фигурами

    @staticmethod

    def create\_board(

        size: int, posed\_figures: list[tuple[int, int]], is\_user: bool

    ) -> Board:

        board = Board(size)

        for x, y in posed\_figures:

            Piece.place\_piece(x, y, board, is\_user)

        return board

# класс диалогового окна для ввода координат

class InputDialog(QDialog):

    def \_\_init\_\_(self, N: int, L: int, K: int):

        super().\_\_init\_\_()

        self.setWindowTitle("Ввод координат")

        self.setSizePolicy(QSizePolicy.Expanding, QSizePolicy.Expanding)

        self.layout = QVBoxLayout()

        self.N = N

        self.L = L

        self.K = K

        self.board\_layout = QGridLayout()

        self.board = Board(N)

        self.user\_figures = set()

        for i in range(N):

            for j in range(N):

                label = QLabel()

                label.setStyleSheet(

                    "background-color: white; border: 1px solid black; min-width: 20px; min-height: 20px;"

                )

                label.setAlignment(Qt.AlignCenter)

                label.setSizePolicy(QSizePolicy.Expanding, QSizePolicy.Expanding)

                label.mousePressEvent = lambda event, x=i, y=j: self.handle\_click(

                    event, x, y

                )

                self.board\_layout.addWidget(label, i, j)

        self.layout.addLayout(self.board\_layout)

        self.generate\_button = QPushButton("Сгенерировать доску")

        self.generate\_button.clicked.connect(self.validate\_and\_generate\_board)

        self.layout.addWidget(self.generate\_button)

        self.solutions\_label = QLabel("")

        self.layout.addWidget(self.solutions\_label)

        spacer = QSpacerItem(20, 40, QSizePolicy.Minimum, QSizePolicy.Expanding)

        self.layout.addItem(spacer)

        self.setLayout(self.layout)

    # метод для обработки клика по ячейке

    def handle\_click(self, event: QMouseEvent, x: int, y: int):

        if event.button() == Qt.LeftButton:

            if (x, y) not in self.user\_figures and ChessLogic.is\_valid\_position(

                self.user\_figures | {(x, y)}, self.N

            ):

                self.user\_figures.add((x, y))

                self.update\_board\_display()

        elif event.button() == Qt.RightButton and (x, y) in self.user\_figures:

            self.user\_figures.remove((x, y))

            self.update\_board\_display()

    # метод для обновления отображения доски

    def update\_board\_display(self):

        self.board.clear()

        for x, y in self.user\_figures:

            Piece.place\_piece(x, y, self.board, True)

        for i in range(self.N):

            for j in range(self.N):

                label = self.board\_layout.itemAtPosition(i, j).widget()

                cell\_content = self.board.cells[i][j].content

                if cell\_content == "U":

                    label.setStyleSheet(

                        "background-color: yellow; border: 1px solid black; min-width: 20px; min-height: 20px;"

                    )

                elif cell\_content == "\*":

                    label.setStyleSheet(

                        "background-color: red; border: 1px solid black; min-width: 20px; min-height: 20px;"

                    )

                else:

                    label.setStyleSheet(

                        "background-color: white; border: 1px solid black; min-width: 20px; min-height: 20px;"

                    )

    # метод для проверки корректности ввода и генерации доски

    def validate\_and\_generate\_board(self):

        if not ChessLogic.is\_valid\_position(self.user\_figures, self.N):

            self.display\_error("Некорректное расположение фигур")

            return False

        self.generate\_board()

        return True

    # метод для генерации доски

    def generate\_board(self):

        solutions = set()

        ChessLogic.recursion\_for\_all\_arrangements(

            self.N, self.L, solutions, self.user\_figures, 0, 0, 0

        )

        if solutions:

            solution = next(iter(solutions))

            self.show\_solution\_dialog(Board(self.N), solution, solutions)

        else:

            self.display\_error("Решений не найдено")

    # метод для отображения сообщения об ошибке

    def display\_error(self, message: str):

        QMessageBox.critical(self, "Ошибка", message)

    # метод для отображения итоговой доски

    def show\_solution\_dialog(

        self, board: Board, solution: set[tuple[int, int]], solutions: set[frozenset]

    ):

        dialog = SolutionDialog(board, self.user\_figures, solution, solutions)

        dialog.exec()

# класс диалогового окна для отображения итоговой доски

class SolutionDialog(QDialog):

    def \_\_init\_\_(

        self,

        board: Board,

        user\_figures: set[tuple[int, int]],

        solution: set[tuple[int, int]],

        solutions: set[frozenset],

    ):

        super().\_\_init\_\_()

        self.setWindowTitle("Итоговая доска")

        self.setSizePolicy(QSizePolicy.Expanding, QSizePolicy.Expanding)

        self.layout = QVBoxLayout()

        self.board\_layout = QGridLayout()

        # размещаем фигуры пользователя на доске

        for x, y in user\_figures:

            Piece.place\_piece(x, y, board, True)

        # размещаем фигуры алгоритма на доске

        for x, y in solution:

            Piece.place\_piece(x, y, board, False)

        # отображаем доску с выделением отличий

        for i in range(board.size):

            for j in range(board.size):

                label = QLabel()

                cell\_content = board.cells[i][j].content

                if (i, j) in user\_figures:

                    label.setStyleSheet(

                        "background-color: yellow; border: 1px solid black; min-width: 20px; min-height: 20px;"

                    )

                elif cell\_content == "#":

                    label.setStyleSheet(

                        "background-color: blue; border: 1px solid black; min-width: 20px; min-height: 20px;"

                    )

                elif cell\_content == "\*":

                    label.setStyleSheet(

                        "background-color: red; border: 1px solid black; min-width: 20px; min-height: 20px;"

                    )

                else:

                    label.setStyleSheet(

                        "background-color: white; border: 1px solid black; min-width: 20px; min-height: 20px;"

                    )

                label.setAlignment(Qt.AlignCenter)

                label.setSizePolicy(QSizePolicy.Expanding, QSizePolicy.Expanding)

                self.board\_layout.addWidget(label, i, j)

        self.layout.addLayout(self.board\_layout)

        self.solutions\_label = QLabel(f"Количество решений: {len(solutions)}")

        self.layout.addWidget(self.solutions\_label)

        self.setLayout(self.layout)

        self.write\_solutions\_to\_file(solutions)

    # запись решений в output.txt

    def write\_solutions\_to\_file(self, solutions):

        with open("output.txt", "w") as file:

            for solution in solutions:

                line = " ".join(f"({x},{y})" for x, y in solution)

                file.write(line + "\n")

# класс основного окна приложения

class MainWindow(QMainWindow):

    def \_\_init\_\_(self):

        super().\_\_init\_\_()

        self.setWindowTitle("Расстановка фигур")

        self.setSizePolicy(QSizePolicy.Expanding, QSizePolicy.Expanding)

        self.central\_widget = QWidget()

        self.layout = QVBoxLayout()

        self.n\_input = QLineEdit()

        self.l\_input = QLineEdit()

        self.n\_label = QLabel("Размер доски:")

        self.l\_label = QLabel("Количество фигур:")

        self.layout.addWidget(self.n\_label)

        self.layout.addWidget(self.n\_input)

        self.layout.addWidget(self.l\_label)

        self.layout.addWidget(self.l\_input)

        self.generate\_button = QPushButton("Далее")

        self.generate\_button.clicked.connect(self.open\_input\_dialog)

        self.layout.addWidget(self.generate\_button)

        self.central\_widget.setLayout(self.layout)

        self.setCentralWidget(self.central\_widget)

    # метод для открытия диалогового окна ввода координат

    def open\_input\_dialog(self):

        try:

            N = int(self.n\_input.text())

            L = int(self.l\_input.text())

            K = 0

            if N <= 0 or L <= 0 or K < 0 or K >= N\*\*2 or L + K > N\*\*2:

                raise ValueError

            dialog = InputDialog(N, L, K)

            dialog.exec()

        except ValueError:

            self.display\_error("Некорректный ввод")

    # метод для отображения сообщения об ошибке

    def display\_error(self, message: str):

        QMessageBox.critical(self, "Ошибка", message)

# запуск приложения

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

    app = QApplication(sys.argv)

    window = MainWindow()

    window.show()

    sys.exit(app.exec())

# Заключение

В ходе выполнения лабораторной работы был разработан и оптимизирован алгоритм размещения дополнительных фигур на шахматной доске. Полученный алгоритм работает эффективно и находит все возможные решения задачи.

Были проведены тесты с различными входными данными, включая доски размером до 20х20. Программа успешно справляется с поставленной задачей, обеспечивая оптимальное распределение фигур и выводя доску. Количество найденных решений выводится под доской.

Таким образом, цель работы достигнута, и разработанный алгоритм может быть использован для эффективного решения аналогичных задач размещения фигур на шахматной доске, а также разработано приложение на основе этого алгоритма.