Министерство образования и науки Российской Федерации Федеральное государственное бюджетное образовательное учреждение высшего образования   
Санкт-Петербургский Государственный Морской Технический Университет

Лабораторная работа № 2

По дисциплине: «Программирование»

|  |  |
| --- | --- |
| Выполнил: | Расщупкин Дмитрий Антонович, студент СПБГМТУ, группа 20121 |
| Принял: | Ассистент СпБГМТУ Поделенюк Павел Петрович |

Санкт-Петербург

2023

Оглавление

[Цели работы 3](#_Toc124168814)

[Ход работы 4](#_Toc124168815)

[Результат работы 6](#_Toc124168816)

[Вывод 10](#_Toc124168817)

[Листинг кода 11](#_Toc124168818)

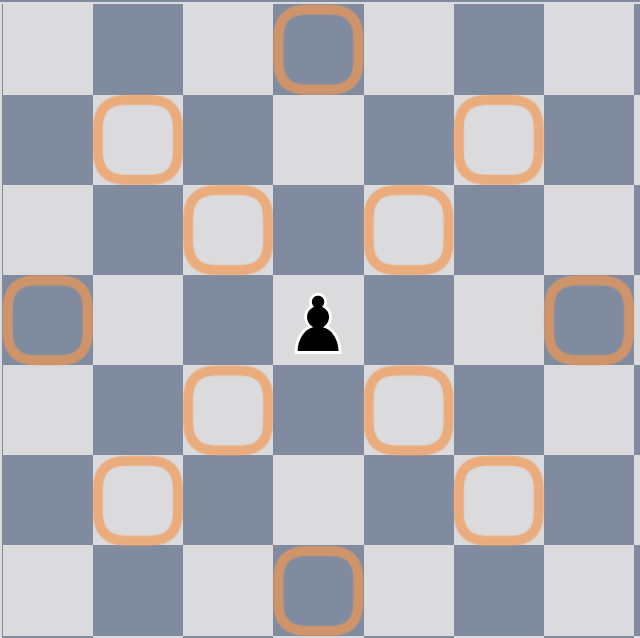
# Цели работы

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.

# Ход работы

Фигура Алфин + Рыбка (через две клетки по диагонали и вертикале)



На вход поступает 3 числа: N, L, K, где N -размерность квадратной доски, L - количество фигур, которые необходимо поставить самостоятельно и K – количество фигур, координаты которых даны. Далее идет K строчек, содержащих координаты фигур.

Объявляем глобальные переменные: MAX\_N (отвечает за максимальный размер доски), FREE (значение, при котором поле свободно), BUSY (значение, при котором поле занято самой фигурой), ATTACKED (значение, при котором поле находиться под атакой), IS\_FINDED (флаг, отвечающий за то, что существует хотя бы одна расстановка, а так же равняется количеству расстановок

Создаем двумерный массив chessboard длиной N, заполненный 0, считываем с файла input N, L, K.

Устанавливаем первые фигуры по координатам из файла input. С помощью глобальной переменной BUSY и функции set\_attacked устанавливаем значения на поля 1 и 2 соответственно из этого, значит на эти поля нельзя устанавливать какие-либо фигуры.

Функция set\_attacked расставляет 2 на поля, которые будут находиться под атакой с помощью двух циклов, которые ставят 2 на те поля, которые находятся по диагонали и на прямых по нашему условию.

Далее создается файл output и в последствии начинает выполняться рекурсивная функция search\_solution по параметрам: доски, размера доски, количества фигур, которые надо расставить, временной переменной и координат x и y.

Функция search\_solution создает дополнительную доску, в которую в последствии мы будем копировать нашу основную доску. Далее происходит перебор x, y при котором мы проверяем пустое ли поле (параметр FREE), если поле свободно, то копируем основную доску в дополнительную доску, после занимаем поле (параметр BUSY) и отмечаем поля под атакой.

Далее если количество фигур (k), которые мы уже расставили сравнялось с L (количество фигур, которые надо расставить) записываем расстановки в файл с помощью созданной функции output (просто записывает координаты наших фигур в файл) и устанавливаем флаг, что мы нашли хотя бы одну расстановку, дополнительно выводим в консоль дополнительную доску с помощью функции print\_board (функция выводит доску заменяя пустые поля (FREE) на 0, занятые фигурами поля (BUSY) на «#» и поля под атакой (ATTACKED) на «\*»).

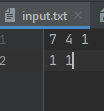
Если количество фигур (k), которые мы расставили, не сравнялось с L (количество фигур, которые надо расставить), то мы к параметру k добавляем единицу и передаем дополнительную доску, n (размер доски), l (количество фигур, которые надо расставить) и передаем координаты x, y и повторяем итерацию.

После выхода из функции мы смотрим на параметр IS\_FINDED и если он 0, то выводим в файл no solutions

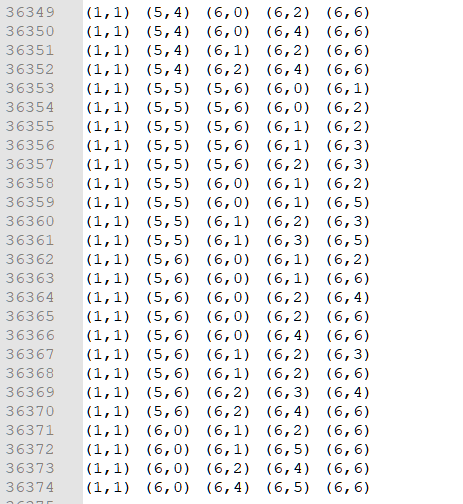
# Результат работы

В результате лабораторной работы был получен алгоритм, который решает поставленную задачу.

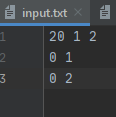
При входных данных:



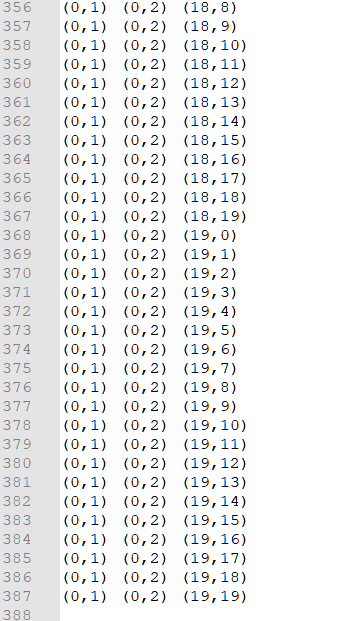
Получаем вывод:



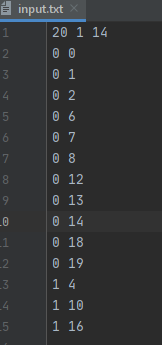
При входных данных:



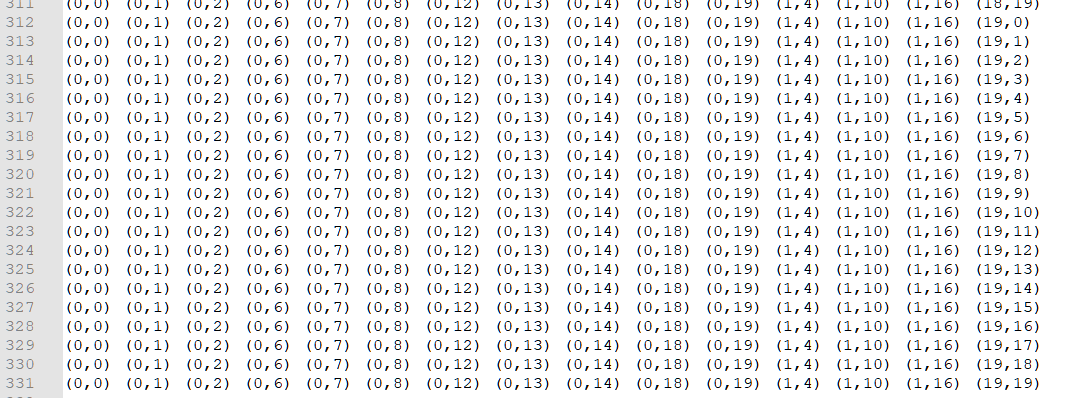
Получаем вывод:



При входных данных:



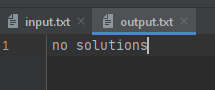
Получаем вывод:



При входных данных:

20 10 131  
0 0  
0 1  
0 2  
0 6  
0 7  
0 8  
0 12  
0 13  
0 14  
0 18  
0 19  
1 4  
1 10  
1 16  
2 1  
2 7  
2 13  
2 18  
2 19  
3 3  
3 4  
3 5  
3 9  
3 10  
3 11  
3 15  
3 16  
4 0  
4 1  
4 7  
4 13  
4 18  
4 19  
5 4  
5 10  
5 15  
5 16  
6 0  
6 1  
6 6  
6 7  
6 8  
6 12  
6 13  
6 18  
6 19  
7 3  
7 4  
7 10  
7 15  
7 16  
8 0  
8 1  
8 7  
8 12  
8 13  
8 18  
8 19  
9 3  
9 4  
9 9  
9 10  
9 15  
9 16  
10 0  
10 1  
10 6  
10 7  
10 12  
10 13  
10 18  
10 19  
11 3  
11 4  
11 9  
11 10  
11 15  
11 16  
12 0  
12 1  
12 6  
12 7  
12 12  
12 13  
12 18  
12 19  
13 3  
13 4  
13 9  
13 10  
13 15  
13 16  
14 0  
14 1  
14 6  
14 7  
14 12  
14 13  
14 18  
14 19  
15 3  
15 4  
15 9  
15 10  
15 15  
15 16  
16 0  
16 1  
16 6  
16 7  
16 12  
16 13  
16 18  
16 19  
17 3  
17 4  
17 9  
17 10  
17 15  
17 16  
18 0  
18 1  
18 6  
18 7  
18 12  
18 13  
18 18  
18 19  
19 3  
19 4  
19 9

Получаем вывод:



# Вывод

По итогу выполнения лабораторной работы был получен алгоритм решения заданной задачи через рекурсию. Также укрепились навыки работы с

1. текстовыми файлами,
2. локальными и глобальными переменными,
3. несколькими функциями, а не только с «main»,
4. Нисходящим методом разработки программ,
5. Анализом ходов.

# Листинг кода

MAX\_N = **20**

FREE = **0**

BUSY = **1**

ATTACKED = **2**

IS\_FINDED = **0** # флаг, что хоть одна расстановка найдена (= кол-во расстановок)

**def** **print\_board**(chessboard: list, n: int) -> None: # вывод доски в консоль

**for** x **in** range(**0**, n):

**for** y **in** range(**0**, n):

**if** chessboard[x][y] == FREE:

**print**("0 ", end="")

**if** chessboard[x][y] == BUSY:

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

**if** chessboard[x][y] == ATTACKED:

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

**print**()

**print**()

**def** **set\_attacked**(chessboard: list, x: int, y: int, n: int) -> None: # установка атаки фигуры, везде проверяя, что не выходим за границы доски

**for** i **in** range(-**2**, **3**, **1**): # устанавливаем атаку на диагоналях

**if** i == **0**:

**continue**

**if** x + i >= **0** **and** y + i >= **0** **and** x + i < n **and** y + i < n:

chessboard[x + i][y + i] = ATTACKED

**if** x + i >= **0** **and** y - i >= **0** **and** x + i < n **and** y - i < n:

chessboard[x + i][y - i] = ATTACKED

**for** i **in** range(-**3**, **4**, **6**): # устанавливаем атаку на прямых

**if** x + i >= **0** **and** x + i < n:

chessboard[x + i][y] = ATTACKED

**if** y + i >= **0** **and** y + i < n:

chessboard[x][y + i] = ATTACKED

**print**(type(set\_attacked([**0**], **0**, **0**, **0**)))

**def** **output**(chessboard: list, n: int) -> None: # вывод расстановки в файл

**with** open("output.txt", mode="a") **as** output\_file:

**for** x **in** range(**0**, n):

**for** y **in** range(**0**, n):

**if** chessboard[x][y] == BUSY:

output\_file.write("({0},{1}) ".format(x, y))

output\_file.write("**\n**")

**def** **search\_solutions**(chessboard: list, n: int, l: int, k: int, x\_prev: int, y\_prev: int) -> None:

**global** IS\_FINDED

board = [[**0**] \* MAX\_N **for** i **in** range(MAX\_N)] # доп. доска

**for** x **in** range(x\_prev,

n): # начинаем проходить по полям с прошлой итерации (или с начальной 0 0), ищем свободное поле

y1 = y\_prev **if** x == x\_prev **else** **0**

**for** y **in** range(y1, n):

**if** chessboard[x][y] == FREE: # если поле свободно, то

**for** x1 **in** range(**0**, n): # копируем доску в доп. доску

**for** y1 **in** range(**0**, n):

board[x1][y1] = chessboard[x1][y1]

board[x][y] = BUSY # занимаем поле

set\_attacked(board, x, y, n) # отмечаем поля, находящиеся под ударом

**if** k + **1** == l: # если кол-во расставленных фигур равно L, то

output(board, n) # вывод расстановки в файл

IS\_FINDED += **1** # флаг, что хоть одна расстановка найдена

**if** IS\_FINDED == **1**:

print\_board(board, n) # вывод доски в консоль

**else**:

search\_solutions(board, n, l, k + **1**, x, y) # вызов ф-ции для следующей установки фигуры

**def** **main**():

N: int # размер доски

L: int # сколько фигур нужно еще расставить

K: int # сколько фигур уже расставлено

chessboard = [[**0**] \* MAX\_N **for** i **in** range(MAX\_N)]

**with** open("input.txt", "r") **as** input\_file:

strings = input\_file.readlines(**1**)

N = int(strings[**0**].split(" ")[**0**])

L = int(strings[**0**].split(" ")[**1**])

K = int(strings[**0**].split(" ")[**2**])

strings = input\_file.readlines()

**for** i **in** range(**0**, K): # ввод начальной расстановки из файла

x = int(strings[i].replace("**\n**", '').split(" ")[**0**])

y = int(strings[i].replace("**\n**", '').split(" ")[**1**])

chessboard[x][y] = BUSY # занимаем поле

set\_attacked(chessboard, x, y, N) # отмечаем поля, находящиеся под ударом

output\_file = open("output.txt", "w")

output\_file.close()

**if** L == **0**:

print\_board(chessboard, N)

**else**:

search\_solutions(chessboard, N, L, **0**, **0**, **0**) #поиск всех возможных расстановок

**if** IS\_FINDED == **0**: #если расстановки не найдены, вывод no solutions

**with** open("output.txt", "w") **as** output\_file:

output\_file.write("no solutions")

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

main()