**Министерство науки и высшего образования Российской Федерации**

**ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ**

**«НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ ИТМО»**

**(Университет ИТМО)**

Факультет **Инфокоммуникационных технологий**

Образовательная программа **Мобильные и сетевые технологии**

Направление подготовки(специальность) **09.03.03 Прикладная информатика**

**ЛАБОРАТОРНАЯ РАБОТА №2**

**По дисциплине «Программирование»**

**Тема: Решатель Судоку**

**Выполнил** Говоров П.И.; К3140

**Проверил** Терещенко В.В.

**Дата** 01.11.2024

**Санкт-Петербург 2024**

**Цель работы**

Создание и тестирования алгоритма судоку, алгоритма генерации судоку, а также реализация различных функций для этих алгоритмов

**Ход работы**

**Создание судоку**

На вход алгоритма подается матрица или файл судоку, например:

53..7....  
6..195...  
.98....6.  
8...6...3  
4..8.3..1  
7...2...6  
.6....28.  
...419..5  
....8..79

Дальше файл группируется на двумерный массив, который содержит элементы судоку.

1. Функция group разбивает одномерный массив на двумерный размерами NxN.

Код:

def group(values: tp.List[T], n: int) -> tp.List[tp.List[T]]:  
 *"""  
 Сгруппировать значения values в список, состоящий из списков по n элементов  
 >>> group([1,2,3,4], 2)  
 [[1, 2], [3, 4]]  
 >>> group([1,2,3,4,5,6,7,8,9], 3)  
 [[1, 2, 3], [4, 5, 6], [7, 8, 9]]  
 """* return [values[i:i+n] for i in range(0,len(values),n)]

1. Дальше были написаны функции получения значений из разных вариантов областей
   1. Получение значений из строки

def get\_row(grid: tp.List[tp.List[str]], pos: tp.Tuple[int, int]) -> tp.List[str]:  
 *"""Возвращает все значения для номера строки, указанной в pos  
 >>> get\_row([['1', '2', '.'], ['4', '5', '6'], ['7', '8', '9']], (0, 0))  
 ['1', '2', '.']  
 >>> get\_row([['1', '2', '3'], ['4', '.', '6'], ['7', '8', '9']], (1, 0))  
 ['4', '.', '6']  
 >>> get\_row([['1', '2', '3'], ['4', '5', '6'], ['.', '8', '9']], (2, 0))  
 ['.', '8', '9']  
 """* return grid[pos[0]]

* 1. Получение значений из столбца

def get\_col(grid: tp.List[tp.List[str]], pos: tp.Tuple[int, int]) -> tp.List[str]:  
 *"""Возвращает все значения для номера столбца, указанного в pos  
 >>> get\_col([['1', '2', '.'], ['4', '5', '6'], ['7', '8', '9']], (0, 0))  
 ['1', '4', '7']  
 >>> get\_col([['1', '2', '3'], ['4', '.', '6'], ['7', '8', '9']], (0, 1))  
 ['2', '.', '8']  
 >>> get\_col([['1', '2', '3'], ['4', '5', '6'], ['.', '8', '9']], (0, 2))  
 ['3', '6', '9']  
 """* return [grid[i][pos[1]] for i in range(len(grid))]

* 1. Получение значений из блока

def get\_block(grid: tp.List[tp.List[str]], pos: tp.Tuple[int, int]) -> tp.List[str]:  
 *"""Возвращает все значения из квадрата, в который попадает позиция pos  
 >>> grid = read\_sudoku('puzzle1.txt')  
 >>> get\_block(grid, (0, 1))  
 ['5', '3', '.', '6', '.', '.', '.', '9', '8']  
 >>> get\_block(grid, (4, 7))  
 ['.', '.', '3', '.', '.', '1', '.', '.', '6']  
 >>> get\_block(grid, (8, 8))  
 ['2', '8', '.', '.', '.', '5', '.', '7', '9']  
 """* i\_block = pos[0]//3\*3  
 j\_block = pos[1]//3\*3  
  
 return [grid[i][j] for i in range(i\_block, i\_block + 3) for j in range(j\_block, j\_block + 3) ]

1. Далее напишем функцию получения координат следующей свободной ячейки (а также доп. функцию по нахождению массива всех свободных ячеек)

def find\_empty\_positions(grid: tp.List[tp.List[str]]) -> tp.Optional[tp.Tuple[int, int]]:  
 *"""Найти первую свободную позицию в пазле  
 >>> find\_empty\_positions([['1', '2', '.'], ['4', '5', '6'], ['7', '8', '9']])  
 (0, 2)  
 >>> find\_empty\_positions([['1', '2', '3'], ['4', '.', '6'], ['7', '8', '9']])  
 (1, 1)  
 >>> find\_empty\_positions([['1', '2', '3'], ['4', '5', '6'], ['.', '8', '9']])  
 (2, 0)  
 """* n = len(grid)  
 for i in range(n):  
 if '.' in set(grid[i]):  
 for j in range(n):  
 if grid[i][j] == '.':  
 return i, j  
 return -1,-1  
  
  
def find\_arr\_empty\_positions(grid: tp.List[tp.List[str]]) -> tp.List[tp.Tuple[int, int]]:  
 n = len(grid)  
 arr = []  
 for i in range(3):  
 for j in range(3):  
 temp\_arr = get\_block(grid, (i\*3,j\*3))  
 arr.extend([(i\*3+g//3,j\*3+g%3) for g in range(9) if temp\_arr[g] == '.'])  
 arr.append((-1,-1))  
 return arr

1. Напишем функцию для нахождения доступных значений ячейки по заданным координатам

def find\_possible\_values(grid: tp.List[tp.List[str]], pos: tp.Tuple[int, int]) -> tp.Set[str]:  
 *"""Вернуть множество возможных значения для указанной позиции  
 >>> grid = read\_sudoku('puzzle1.txt')  
 >>> values = find\_possible\_values(grid, (0,2))  
 >>> values == {'1', '2', '4'}  
 True  
 >>> values = find\_possible\_values(grid, (4,7))  
 >>> values == {'2', '5', '9'}  
 True  
 """* return set(str(i) for i in range(1,10) if str(i) not in set(get\_row(grid, pos)) | set(get\_col(grid, pos)) | set(get\_block(grid, pos)))

1. Алгоритм решения судоку основан на рекурсивной генерации различных комбинаций сетки, основная фишка в том, что алгоритм перебирает значения по блокам и сначала подставляет значения больше максимальных в блоке

def solve\_recursion(grid: tp.List[tp.List[str]], ind\_arr\_pos: int):  
 *'''Рекурсивная функция решения судоку'''* global solution\_grid\_global, arr\_empty\_positions, solution\_found  
 if solution\_found:  
 return  
 y, x = arr\_empty\_positions[ind\_arr\_pos]  
 if not(y == -1 and x == -1):  
 possible\_values = sorted(find\_possible\_values(grid, (y,x)))  
 if not possible\_values:  
 return  
 max\_from\_block = max(get\_block(grid, (y,x)))  
 max\_from\_block = '0' if max\_from\_block == '.' else max\_from\_block  
 possible\_values = [i for i in possible\_values if i > max\_from\_block] + [i for i in possible\_values if i <= max\_from\_block]  
 for i in possible\_values:  
 grid[y][x] = i  
 solve\_recursion(grid, ind\_arr\_pos+1)  
 grid[y][x] = '.'  
  
 else:  
 solution\_grid\_global = copy.deepcopy(grid)  
 solution\_found = True

1. Функция, которая возвращает значение, полученное в результате работы рекурсии

def solve(grid: tp.List[tp.List[str]]) -> tp.Optional[tp.List[tp.List[str]]]:  
 *""" Как решать Судоку?  
 >>> grid = read\_sudoku('puzzle1.txt')  
 >>> solve(grid)  
 [['5', '3', '4', '6', '7', '8', '9', '1', '2'], ['6', '7', '2', '1', '9', '5', '3', '4', '8'], ['1', '9', '8', '3', '4', '2', '5', '6', '7'], ['8', '5', '9', '7', '6', '1', '4', '2', '3'], ['4', '2', '6', '8', '5', '3', '7', '9', '1'], ['7', '1', '3', '9', '2', '4', '8', '5', '6'], ['9', '6', '1', '5', '3', '7', '2', '8', '4'], ['2', '8', '7', '4', '1', '9', '6', '3', '5'], ['3', '4', '5', '2', '8', '6', '1', '7', '9']]  
 """* global solution\_grid\_global, arr\_empty\_positions, solution\_found  
 arr\_empty\_positions = find\_arr\_empty\_positions(grid)  
 solution\_grid\_global = []  
 solution\_found = False  
 solve\_recursion(grid, 0)  
 return solution\_grid\_global

1. Напишем функция для проверки решенного судоку.

def check\_solution(solution: tp.List[tp.List[str]]) -> bool:  
 *""" Если решение solution верно, то вернуть True, в противном случае False  
 >>> grid = read\_sudoku('puzzle1.txt')  
 >>> solution = solve(grid)  
 >>> check\_solution(solution)  
 True  
 >>> grid = read\_sudoku('puzzle\_false.txt')  
 >>> check\_solution(grid)  
 False  
 """* if not solution:  
 return False  
 for i in range(len(solution)):  
 row\_nums = set(get\_row(solution, (i, 0)))  
 col\_nums = set(get\_col(solution, (0, i)))  
 block\_nums = set(get\_block(solution, (i//3\*3, i%3\*3)))  
 if '.' in row\_nums:  
 return False  
 if not (len(row\_nums) == len(col\_nums) == len(block\_nums) == len(solution)):  
 return False  
 return True

1. Далее для функции генерации судоку напишем следующие доп. функции, они выполняют тождественные преобразования матрицы судоку для ее перемешивания (пояснения к функциям описаны в doc-strings).

def transpon(grid: tp.List[tp.List[str]]) -> tp.List[tp.List[str]]:  
 *'''Транспонирование матрицы'''* grid\_temp = [['.']\*len(grid) for \_ in range(len(grid))]  
 for i in range(len(grid)):  
 for j in range(len(grid)):  
 grid\_temp[i][j] = grid[j][i]  
 return grid\_temp  
  
def swap\_row(grid: tp.List[tp.List[str]], n\_row\_1, n\_row\_2) -> tp.List[tp.List[str]]:

*'''Смена двух строк из 1 области'''*  
 grid\_temp = []  
 for i in range(len(grid)):  
 if i == n\_row\_2:  
 grid\_temp.append(grid[n\_row\_1][:])  
 elif i == n\_row\_1:  
 grid\_temp.append(grid[n\_row\_2][:])  
 else:  
 grid\_temp.append(grid[i][:])  
 return grid\_temp  
  
def swap\_col(grid: tp.List[tp.List[str]], n\_col\_1, n\_col\_2) -> tp.List[tp.List[str]]:

*'''Смена двух столбцов из 1 области'''*  
 grid\_temp = []  
 for i in range(len(grid)):  
 arr\_temp = grid[i][:]  
 arr\_temp[n\_col\_1], arr\_temp[n\_col\_2] = arr\_temp[n\_col\_2], arr\_temp[n\_col\_1]  
 grid\_temp.append(arr\_temp)  
 return grid\_temp  
  
def swap\_cols\_area(grid: tp.List[tp.List[str]], n\_col\_area\_1, n\_col\_area\_2) -> tp.List[tp.List[str]]:

*'''Смена двух областей(столбцы)'''*  
 grid\_temp = []  
 for i in range(len(grid)):  
 arr\_temp = grid[i][:]  
 (arr\_temp[n\_col\_area\_1\*3], arr\_temp[n\_col\_area\_1\*3+1], arr\_temp[n\_col\_area\_1\*3+2],  
 arr\_temp[n\_col\_area\_2\*3], arr\_temp[n\_col\_area\_2\*3+1], arr\_temp[n\_col\_area\_2\*3+2]) \  
 =\  
 (arr\_temp[n\_col\_area\_2\*3], arr\_temp[n\_col\_area\_2\*3+1], arr\_temp[n\_col\_area\_2\*3+2],  
 arr\_temp[n\_col\_area\_1\*3], arr\_temp[n\_col\_area\_1\*3+1], arr\_temp[n\_col\_area\_1\*3+2])  
 grid\_temp.append(arr\_temp)  
 return grid\_temp  
  
def swap\_rows\_area(grid: tp.List[tp.List[str]], n\_row\_area\_1, n\_row\_area\_2) -> tp.List[tp.List[str]]:

*'''Смена двух областей(строки)'''*  
 grid\_temp = []  
 for i in range(0, len(grid), 3):  
 if i // 3 == n\_row\_area\_1:  
 grid\_temp.append(grid[n\_row\_area\_2 \* 3][:])  
 grid\_temp.append(grid[n\_row\_area\_2 \* 3 + 1][:])  
 grid\_temp.append(grid[n\_row\_area\_2 \* 3 + 2][:])  
 elif i // 3 == n\_row\_area\_2:  
 grid\_temp.append(grid[n\_row\_area\_1 \* 3][:])  
 grid\_temp.append(grid[n\_row\_area\_1 \* 3 + 1][:])  
 grid\_temp.append(grid[n\_row\_area\_1 \* 3 + 2][:])  
 else:  
 grid\_temp.append(grid[i][:])  
 grid\_temp.append(grid[i + 1][:])  
 grid\_temp.append(grid[i + 2][:])  
 return grid\_temp  
  
def two\_random\_nums(fr: int,to: int) -> tp.Tuple[int,int]:

*'''Получить рандомные 2 номера для переставления'''*  
 num\_1 = random.randint(fr, to)  
 num\_2 = random.randint(fr, to)  
 while num\_1 == num\_2:  
 num\_2 = random.randint(fr, to)  
 return num\_1, num\_2  
  
def two\_random\_nums\_row\_or\_col(num\_row\_or\_col: int) -> tp.Tuple[int,int]:

*'''Получить рандомные 2 номера областей для переставления'''*  
 num\_1 = random.randint(num\_row\_or\_col\*3, num\_row\_or\_col\*3+2)  
 num\_2 = random.randint(num\_row\_or\_col\*3, num\_row\_or\_col\*3+2)  
 while num\_1 == num\_2:  
 num\_2 = random.randint(num\_row\_or\_col\*3, num\_row\_or\_col\*3+2)  
 return num\_1, num\_2

1. Функция для генерации судоку, берет за основу решенное судоку и применяет множество раз тождественные преобразования, а после для рандомных позиций убирает нужное количество элементов

def generate\_sudoku(N: int) -> tp.List[tp.List[str]]:  
 *"""Генерация судоку заполненного на N элементов  
 >>> grid = generate\_sudoku(40)  
 >>> sum(1 for row in grid for e in row if e == '.')  
 41  
 >>> solution = solve(grid)  
 >>> check\_solution(solution)  
 True  
 >>> grid = generate\_sudoku(1000)  
 >>> sum(1 for row in grid for e in row if e == '.')  
 0  
 >>> solution = solve(grid)  
 >>> check\_solution(solution)  
 True  
 >>> grid = generate\_sudoku(0)  
 >>> sum(1 for row in grid for e in row if e == '.')  
 81  
 >>> solution = solve(grid)  
 >>> check\_solution(solution)  
 True  
 """* arr\_pos\_to\_gen = [(i,j) for i in range(9) for j in range(9)]  
 random.shuffle(arr\_pos\_to\_gen)  
  
 gen\_sudoku\_grid = create\_grid('123456789 456789123 789123456'  
 '234567891 567891234 891234567'  
 '345678912 678912345 912345678')  
 for i in range(100):  
 op\_num = random.randint(0,4)  
 if op\_num == 0:  
 gen\_sudoku\_grid = transpon(gen\_sudoku\_grid)  
 #print(gen\_sudoku\_grid)  
 elif op\_num == 1:  
 num\_row\_1, num\_row\_2 = two\_random\_nums\_row\_or\_col(random.randint(0, 2))  
 gen\_sudoku\_grid = swap\_row(gen\_sudoku\_grid, num\_row\_1, num\_row\_2)  
 elif op\_num == 2:  
 num\_col\_1, num\_col\_2 = two\_random\_nums\_row\_or\_col(random.randint(0, 2))  
 gen\_sudoku\_grid = swap\_col(gen\_sudoku\_grid, num\_col\_1, num\_col\_2)  
 elif op\_num == 3:  
 num\_row\_area\_1, num\_row\_area\_2 = two\_random\_nums(0, 2)  
 gen\_sudoku\_grid = swap\_rows\_area(gen\_sudoku\_grid, num\_row\_area\_1, num\_row\_area\_2)  
 elif op\_num == 4:  
 num\_col\_area\_1, num\_col\_area\_2 = two\_random\_nums(0, 2)  
 gen\_sudoku\_grid = swap\_cols\_area(gen\_sudoku\_grid, num\_col\_area\_1, num\_col\_area\_2)  
  
 for i in range(81-N):  
 y, x = arr\_pos\_to\_gen[i]  
 gen\_sudoku\_grid[y][x] = '.'  
  
 return gen\_sudoku\_grid

1. Напишем тесты для различных функций судоку

def test\_create\_grid(self):  
 string\_sudoku = '8..4.6..7\n......4..\n.1....65.\n5.9.3.78.\n....7....\n.48.2.1.3\n.52....9.\n..1......\n3..9.2..5'  
 alr\_created\_grid = [['8', '.', '.', '4', '.', '6', '.', '.', '7'], ['.', '.', '.', '.', '.', '.', '4', '.', '.'], ['.', '1', '.', '.', '.', '.', '6', '5', '.'], ['5', '.', '9', '.', '3', '.', '7', '8', '.'], ['.', '.', '.', '.', '7', '.', '.', '.', '.'], ['.', '4', '8', '.', '2', '.', '1', '.', '3'], ['.', '5', '2', '.', '.', '.', '.', '9', '.'], ['.', '.', '1', '.', '.', '.', '.', '.', '.'], ['3', '.', '.', '9', '.', '2', '.', '.', '5']]  
 self.assertEqual(sud.create\_grid(string\_sudoku), alr\_created\_grid)  
  
def test\_group(self):  
 test1 = [1,2,3,4]  
 n1 = 2  
 self.assertEqual(sud.group(test1,n1),[[1, 2], [3, 4]])  
 test2 = [1,2,3,4,5,6,7,8,9]  
 n2 = 3  
 self.assertEqual(sud.group(test2, n2), [[1, 2, 3], [4, 5, 6], [7, 8, 9]])  
  
def test\_get\_row(self):  
 test1 = [['1', '2', '.'], ['4', '5', '6'], ['7', '8', '9']]  
 pos1 = (0,2)  
 self.assertEqual(sud.get\_row(test1, pos1), ['1', '2', '.'])  
 test2 = [['4', '7', '2'], ['3', '1', '.'], ['6', '9', '5']]  
 pos2 = (1, 1)  
 self.assertEqual(sud.get\_row(test2, pos2), ['3', '1', '.'])  
  
def test\_get\_col(self):  
 test1 = [['4', '7', '2'], ['3', '1', '.'], ['6', '9', '5']]  
 pos1 = (1, 2)  
 self.assertEqual(sud.get\_col(test1, pos1), ['2', '.', '5'])  
 test2 = [['1', '2', '3'], ['4', '.', '6'], ['7', '8', '9']]  
 pos2 = (0, 1)  
 self.assertEqual(sud.get\_col(test2, pos2), ['2', '.', '8'])  
  
def test\_get\_block(self):  
 test1 = sud.read\_sudoku('src/lab3/puzzle1.txt')  
 pos1 = (1, 2)  
 pos2 = (7, 7)  
 self.assertEqual(sud.get\_block(test1, pos1), ['5', '3', '.', '6', '.', '.', '.', '9', '8'])  
 self.assertEqual(sud.get\_block(test1, pos2), ['2', '8', '.', '.', '.', '5', '.', '7', '9'])  
  
def test\_find\_empty\_position(self):  
 test1 = [['1', '2', '.'], ['4', '5', '6'], ['7', '8', '9']]  
 self.assertEqual(sud.find\_empty\_positions(test1), (0, 2))  
 test2 = [['1', '2', '3'], ['4', '5', '6'], ['.', '8', '9']]  
 self.assertEqual(sud.find\_empty\_positions(test2), (2, 0))  
  
def test\_possible\_values(self):  
 test1 = sud.read\_sudoku('src/lab3/puzzle1.txt')  
 pos1 = (0,2)  
 self.assertEqual(sud.find\_possible\_values(test1, pos1), {'1', '2', '4'})  
 pos2 = (4,7)  
 self.assertEqual(sud.find\_possible\_values(test1, pos2), {'2', '5', '9'})  
  
def test\_solve(self):  
 test1 = sud.solve(sud.read\_sudoku('src/lab3/puzzle1.txt'))  
 self.assertEqual(test1, [['5', '3', '4', '6', '7', '8', '9', '1', '2'], ['6', '7', '2', '1', '9', '5', '3', '4', '8'], ['1', '9', '8', '3', '4', '2', '5', '6', '7'], ['8', '5', '9', '7', '6', '1', '4', '2', '3'], ['4', '2', '6', '8', '5', '3', '7', '9', '1'], ['7', '1', '3', '9', '2', '4', '8', '5', '6'], ['9', '6', '1', '5', '3', '7', '2', '8', '4'], ['2', '8', '7', '4', '1', '9', '6', '3', '5'], ['3', '4', '5', '2', '8', '6', '1', '7', '9']])  
  
def test\_check\_solution(self):  
 test1 = sud.solve(sud.read\_sudoku('src/lab3/puzzle1.txt'))  
 self.assertEqual(sud.check\_solution(test1), True)  
 test2 = sud.solve(sud.read\_sudoku('src/lab3/puzzle\_false.txt'))  
 self.assertEqual(sud.check\_solution(test2), False)

**Вывод**

В результате проделанной работы я научился писать рекурсивный алгоритм решения судоку, алгоритм генерации судоку и второстепенные функции для них, а также написал тесты для всех этих функций