In order to run my project simply call main and pass in the name of the file that contains the initial board along with the horizontal and vertical inequalities. In order for my code to work the naming convention of files passed in must follow the following convention: Input + # + .txt. This is just like the test input files we were given, i.e. Input1.txt, Input2.txt, Input3.txt. This is because I use the number to automatically generate an output file with the same number. Lastly, the recursion limit is sometimes hit on the Input3.txt file given so it is necessary to re-run the program to get the correct output

Output1.txt:
2 1 5 4 3
1 3 4 2 5
4 5 1 3 2
5 2 3 1 4
3 4 2 5 1

Output1.txt:
3 4 2 5 1
1 5 4 2 3
2 3 5 1 4
5 1 3 4 2
4 2 1 3 5

Output3.txt:
3 1 5 2 4
5 2 3 4 1
1 3 4 5 2
4 5 2 1 3
2 4 1 3 5

Source Code:

```
#  Created by Henry Rivera on 5/13/20.
#  File Name: futoshiki
#  Description: Futoshiki puzzle solver using backtracking and forward checking
#  Copyright © 2020 Henry Rivera. All rights reserved


from random import choice
from copy import deepcopy

''' Each location in the given board becomes a class of type Space '''
class Space:
    def __init__(self, val):
```

```
        self.val = val
        self.domain = [1, 2, 3, 4, 5]
        self.constraints = []


''' Clears the given value from the domain of the neighbors '''
def cleanDomain(board, row, col, val):
    for i in board[row]:
        if val in i.domain:
            i.domain.remove(val)
    for r in range(5):
        if int(val) in board[r][col].domain:
            board[r][col].domain.remove(val)


'''Going through board and putting in order of domain sizes'''
def smallestDomain(board):
    allDomains = []
    for row in range(5):
        for j in range(5):
            if len(board[row][j].domain) == 1:
                allDomains.append((row, j))
            elif len(board[row][j].domain) == 2:
                allDomains.append((row, j))
            elif len(board[row][j].domain) == 3:
                allDomains.append((row, j))
            elif len(board[row][j].domain) == 4:
                allDomains.append((row, j))
            elif len(board[row][j].domain) == 5:
                allDomains.append((row, j))
    return allDomains


''' Implementation of forward check'''
def forwardCheck(board):
    for row in range(5):
        for i in range(5):
            if len(board[row][i].domain) == 1:
                board[row][i].val = board[row][i].domain[0]
                board[row][i].domain = []
            if board[row][i].val != 0:
                ''' Want to clean domain of neighbors if val is not 0'''
                cleanDomain(board, row, i, board[row][i].val)
                if ">" in board[row][i].constraints:  # if constraints greater than val
```

```
                for value in board[row][i + 1].domain:
                    if value > board[row][i].val:
                        board[row][i + 1].domain.remove(value)
            if "<" in board[row][i].constraints:  # if constraints less than val
                for value in board[row][i + 1].domain:
                    if value < board[row][i].val:
                        board[row][i + 1].domain.remove(value)
            if "^" in board[row][i].constraints:  # if constraints less than val
                for value in board[row + 1][i].domain:
                    if value < board[row][i].val:
                        board[row + 1][i].domain.remove(value)
            if "v" in board[row][i].constraints:  # if constraints greater than val
                for value in board[row + 1][i].domain:
                    if value > board[row][i].val:
                        board[row + 1][i].domain.remove(value)
    return board


'''Checks to see if every location in board has a value'''
def isComplete(board):
    for i in range(5):
        for j in range(5):
            if board[i][j].val == 0:
                return False
    return True


''' Used in backtracking to go back when we've reached a dead end'''
def revert(board, tmp):
    for i in range(5):
        for j in range(5):
            board[i][j] = deepcopy(tmp[i][j])


''' Inequality check to make sure the inequalities passed in hold'''
def checkInequalities(board, constraints):
    for constraint in constraints:
        i = constraint[0]
        j = constraint[1]
        if ">" in board[i][j].constraints:
            if board[i][j].val < board[i][j + 1].val:
                return False
        if "<" in board[i][j].constraints:
            if board[i][j].val > board[i][j + 1].val:
```

```
            return False
        if "^" in board[i][j].constraints:
            if board[i][j].val > board[i + 1][j].val:
                return False
        if "v" in board[i][j].constraints:
            if board[i][j].val < board[i + 1][j].val:
                return False
    return True


''' Backtrack implementation '''
def backTrack(board, tmp, constraints):
    locations = smallestDomain(board)  # going to backtrack on smallest domains first
    prev = deepcopy(tmp)  # storing a copy of tmp passed in for recursion
    for location in locations:
        row = location[0]
        col = location[1]
        if len(board[row][col].domain) != 0 and board[row][col].val == 0:
            board[row][col].val = choice(board[row][col].domain)  # assigning random value from
piece's domain
            board[row][col].domain = []
            forwardCheck(board)  # forwardCheck to prevent future issues
        elif len(board[row][col].domain) == 0 and board[row][col].val == 0:  # otherwise we need to
go back to
            # previous iteration and try something else
            revert(board, tmp)
            backTrack(board, tmp, constraints)
    if isComplete(board) and checkInequalities(board, constraints):  # checking to see if board is
complete and
        # inequalities hold
        return board
    else:
        revert(board, prev)
        backTrack(board, prev, constraints)


def main(filename):  # open file function
    f = open(filename, "r")
    content = f.read().splitlines()
    inp = []
    for line in content:
        inp.append(line.split())
    """ Reading initial board, horizontal inequalities, and vertical inequalities from file"""
    initial = inp[0:5]
```

```
    horizontal = inp[6:11]
    vertical = inp[12:16]
    dup = deepcopy(initial)  # want to save initial board before we convert it into a matrix of class type Space
    inequalities = []  # used to store locations of inequalities in board
    for i in range(5):  # for loop to initialize board
        for j in range(5):
            if initial[i][j] != '0':  # if not 0 then the domain is empty
                initial[i][j] = Space(int(initial[i][j]))
                initial[i][j].domain = []
            else:
                initial[i][j] = Space(int(initial[i][j]))  # set each part of board as a piece of puzzle
    for row in range(5):  # for loop to read locations of horizontal inequalities
        for i in range(4):
            if horizontal[row][i] == '>':
                inequalities.append((row, i))
                if 1 in initial[row][i].domain:
                    initial[row][i].domain.remove(1)
                initial[row][i].constraints.append('>')  # [row][i] has to be > [row][i+1]
            if horizontal[row][i] == '<':
                inequalities.append((row, i))
                if 5 in initial[row][i].domain:
                    initial[row][i].domain.remove(5)
                initial[row][i].constraints.append('<')  # [row][i] has to be < [row][i+1]
    for row in range(4):  # for loop to read locations of vertical inequalities
        for i in range(5):
            if vertical[row][i] == '^':
                inequalities.append((row, i))
                if 5 in initial[row][i].domain:
                    initial[row][i].domain.remove(5)
                initial[row][i].constraints.append('^')  # [row][i] has to be < [row+1][i]
            if vertical[row][i] == 'v':
                inequalities.append((row, i))
                if 1 in initial[row][i].domain:
                    initial[row][i].domain.remove(1)
                initial[row][i].constraints.append('v')  # [row][i] has to be > [row+1][i]
    forwardCheck(initial)
    tmp = deepcopy(initial)  # creating copy of initial board
    backTrack(initial, tmp, inequalities)
    for i in range(5):  # for loop to set values of duplicate initial board = new values
        for j in range(5):
            dup[i][j] = str(initial[i][j].val)
    ''' Generating output file '''
    outputFilename = "Output" + filename[5] + ".txt"
```

```
    output = open(outputFilename, "w+")
    for r in dup:
        output.write(' '.join(r) + "\r\n")
    output.close()


main("Input1.txt")
```