Instructions on how to run the program:

The program was written with vim and debugged with pdb, even though it could be run on any Python IDE. A function call to the main function is on the last line of the source code, so running Futoshiki.py can get the program started right away. The program will print a message and ask for the name of the input file, e.g. “Input1.txt”, which should be placed under the same directory as Futoshiki.py. The next message to be printed asks for the name of the output file, e.g. “Output1.txt”, which will be written under the same directory. The program terminates after writing the output file.

Output text files:

***Output1.txt* (output for *Input1.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

***Output2.txt* (output for *Input2.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* (output for *Input3.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:

```
import io
import copy


class Cell:
    def __init__(self, coord: tuple, assign: int, domain: list,
            constr: list):
        self.coord = coord
        # The coordinates of the cell represented as a tuple
```

```
        # e.g. (2, 3) denotes Row 2, Column 3
        # Row and column numbers range from 0~4 (inclusive)

        self.assign = assign
        # The value assigned to the cell, if any
        # = None if the cell is unassigned

        self.domain = domain
        # Represented as a list of integers

        self.constr = constr
        # Constraints of inequality on the cell, represented as
        # a list of strings


class Board:
    def __init__(self, all_cells: list, all_constr: list):
        self.cells = all_cells
        # All twenty-five cells on the board, represented as
        # a five-by-five list

        self.constr = all_constr
        # All constraints of all twenty-five cells, represented as
        # a five-by-five-by-four list
        # The constraints of an individual cell is represented as
        # a list of four strings, each of which indicates the constraint
        # regarding one of the four neighbors

    """ The following are four methods that, when given one of the Cell
    objects on the board, return one of its neighbors. If moving in
    the particular direction (left, right, etc.) goes beyond the
    boundaries of the board, each of the methods raises a ValueError."""
    def go_left(self, origin: Cell) -> Cell:
        row = origin.coord[0]
        col = origin.coord[1] - 1
        if (0 <= col <= 4):
            return self.cells[row][col]
            # If the move is within the boundaries of the board,
            # return the destination as a Cell object

        else:
            raise ValueError()
            # ValueError will be caught by the function that calls
            # this method
```

```
    def go_right(self, origin: Cell) -> Cell:
        row = origin.coord[0]
        col = origin.coord[1] + 1
        if (0 <= col <= 4):
            return self.cells[row][col]
        else:
            raise ValueError()


    def go_up(self, origin: Cell) -> Cell:
        row = origin.coord[0] - 1
        col = origin.coord[1]
        if (0 <= row <= 4):
            return self.cells[row][col]
        else:
            raise ValueError()


    def go_down(self, origin: Cell) -> Cell:
        row = origin.coord[0] + 1
        col = origin.coord[1]
        if (0 <= row <= 4):
            return self.cells[row][col]
        else:
            raise ValueError()


def load_input(filename: str) -> list:
    """ Given the name of the input file, the function reads the file line by
    line and builds the data structures for the initial state as well as the
    constraints of inequality."""

    text_stream = io.open(filename, 'r', encoding='utf-8',
            errors='ignore', newline='\n')
    """ Calls Python's io function to read the file with the specified name."""

    initial_state = []
    for i in range(0,5):
        initial_state.append(list(map(int,
            text_stream.readline().rstrip().split(' '))))
        """ The rstrip method removes all trailing white space of
        the string. The split method uses the given character as the
        delimiter to break down the string and return a list of the
        substrings. The map function takes that list, converts the
        substrings into integers and returns a map object, which is
        eventually converted into a list by the exterior call to the
```

```
    list function."""

    """ A state is represented as a multi-layer list. The first
    layer contains the five rows, each of which contains a
    second layer that consists of five cells."""

blank_line = text_stream.readline()

""" In the input file, there is a blank line following the
first five lines, after which begin the next five lines
that represent the horizontal constraints."""

constr = []
""" The constraints from the input file will be
converted to a specific text format and stored in this
list."""


for i in range(0,5):
    a_row = []
    # A list that stores the constraints of all cells
    # in a single row
    for j in range(0,5):
        a_row.append(['U', 'D', 'L', 'R'])
        """ Each row in the list "constr" contains
        five lists, one for each of the five cells
        in a row. Each sublist stores the constraints
        in four directions that relate to the
        particular cell: its relation with the cell
        above ('U'), the cell below ('D'), the cell
        to the left ('L'), and the cell to the
        right ('R'). 'U', 'D', 'L' and 'R'
        are four strings that function as placeholders
        for now. Afterward they'll be modified to
        indicate the exact relations between each pair.
        """
        if i == 0:
            a_row[j][0] = "N/A"
            """ For all cells in the first row, the
            first element in their lists is replaced
            with "N/A", since there are no cells
            above them."""
        elif i == 4:
            a_row[j][1] = "N/A"
            """ By the same token, the second element
            in the lists of all cells in the last row
```

```
            is replaced with "N/A"."""

        if j == 0:
            a_row[j][2] = "N/A"
            """ For all cells in the first column,
            the third element in their lists is
            replaced with "N/A", since there are no
            cells to their left."""
        elif j == 4:
            a_row[j][3] = "N/A"
            """ By the same token, the fourth element
            in the lists of all cells in the last
            column is replaced with "N/A"."""

    constr.append(a_row)
    """ Now that the proper placeholders have been
    inserted into the list of a single row, append
    the row to the list "constr"."""

""" By this point "constr" has been formatted as:

1st layer: five lists, each referring to a row on
the Fukushiki board.

2nd layer: this is within each list in the 1st
layer. There are five lists, each refering to a
cell in the particular row.

3rd layer: this is within each list in 2nd layer.
There are four strings, each referring to the
relation between the current cell and the four
neighboring cells: the one above ('U'), the
one below ('D'), the one to the left ('L'),
and the one to the right ('R'). These strings
will be modified afterward to indicate the
exact relations."""


""" The following for loop reads the next five lines, which
contain the constraints between horizontally-adjacent
cells."""

for i in range(0, 5):
    line = list(map(str, text_stream.readline().rstrip().split(' ')))
    """ The functions and methods used in this line are
    identical to the ones in the previous for loop. "line"
```

```
    is a list of the four characters that represent the
    constraints in the row. """

    for j in range(len(line)):
        # len(line) is expected to be 4 (four constraints in
        # a row)
        if line[j] == '0':
            constr[i][j][3] = "None"
            constr[i][j+1][2] = "None"
            """ For instance, if j = 0, and line[j] = 0,
            that indicates there's no constraint between
            the first and second cells of the row.
            Therefore, modify the fourth element of the
            first cell, which stores the current cell's
            relation with the cell on the right; replace
            'R' with "None", since there's no constraint.
            By the same token, replace the third element
            of the second cell with "None"."""

        elif line[j] == '>':
            constr[i][j][3] = "GTR"
            constr[i][j+1][2] = "STL"
            """ For instance, if j = 0, and line[j] = '>',
            that indicates the first cell of the row has
            to be greater than the second cell. Therefore,
            modify the fourth element of the first cell,
            which stores the current cell's relation with
            the cell on the right; replace 'R' with "GTR",
            which stands for "Greater Than Right". By the
            same token, replace the third element of the
            second cell with "STL", which stands for
            "Smaller Than Left"."""

        elif line[j] == '<':
            constr[i][j][3] = "STR"
            constr[i][j+1][2] = "GTL"
            """ The same notation as above, only that the
            relation is one being smaller than the other."""

""" By the end of the double-layer for loop, all constraints
for horizontally-adjacent cells have been read and stored.
The input file contains another blank line, followed by the
last four lines, which illustrate the constraints for
vertically-adjacent cells."""

blank_line = text_stream.readline()
```

```
    # Move the read cursor past the blank line.

    for i in range(0, 4):
        # range = (0,4) because this part of the input file contains
        # only four rows

        line = list(map(str, text_stream.readline().rstrip().split(' ')))
        # Contains the same methods as previously explained
        # "line" is a list that contains the four characters that
        # represent the constraints in a column

        for j in range(len(line)):
            # len(line) is expected to be 5, since there are five
            # characters in each line of this part of the input

            if line[j] == '0':
                constr[i][j][1] = "None"
                constr[i+1][j][0] = "None"
                """ 0 indicates there's no constraint, so the second
                element of the cell (i, j), which refers to its
                relation with the cell underneath it, should be
                "None". By the same token, the first element
                of the cell (i+1, j) should be "None" as well."""

            elif line[j] == '^':
                constr[i][j][1] = "STD"
                constr[i+1][j][0] = "GTU"
                # STD = Smaller Than Down
                # GTU = Greater Than Up

            elif line[j] == 'v':
                constr[i][j][1] = "GTD"
                constr[i+1][j][0] = "STU"
                # GTD = Greater Than Down
                # STU = Smaller Than Up

    text_stream.close()
    # By this point, reading the input file has concluded

    ret = [initial_state, constr]
    # Returns the two lists that represent the initial state and
    # all constraints, respectively
    return ret


def initialize_board(initial_state: list, constr: list) -> Board:
```

```
    """ The parameters are the initial state, represented as a five-by-five
    list, and the list of constraints for all twenty-five cells. The function
    instantiates twenty-five Cell objects with the given data and returns
    a Board object."""

    all_cells = []
    # Will become a five-by-five list by the end of function
    # All Cell objects to be instantiated will be appended
    # to this list, which is then used to instantiate the
    # Board object

    for i in range(0,5):
        a_row = []
        for j in range(0,5):
            assign = initial_state[i][j]
            domain = [1, 2, 3, 4, 5]
            # The initial domain of an empty cell

            if assign == 0:
                assign = None
                # Zero indicates an empty cell
            else:
                domain = None
                # Domain doesn't apply to assigned cells

            a_row.append(Cell((i, j), assign, domain, constr[i][j]))
            # Instantiates the Cell object and appends it to the
            # list of the row

        all_cells.append(a_row)

    # At this point all_cells is a five-by-five list that
    # contains all twenty-five cells

    return Board(all_cells, constr)


def forward_checking(a_board: Board, a_cell: Cell, explored: set) -> int:
    """ Conducts forward checking for the given assigned cell to
    ensure there's no other cell in the same row or column with
    the same assignment. Also reduces each neighbor's domain to
    comply with the constraints, if any. Recursive calls to
    itself are made to eventually check every cell on the board
    for arc consistency."""

    """ The four methods of the Board class--go_up, go_down, go_left
```

```
and go_right--all throw a ValueError when the move goes beyond
the board's boundaries. These errors are caught in the except
statements below, and the respective variables are set as
None. If the move is legit, the method returns the destination
as a Cell object, and the corresponding variable (left, right
etc.) is turned into a reference (shallow copy) to that Cell."""

if a_cell.coord in explored:
    return 0
else:
    explored.add(a_cell.coord)

if a_cell.domain != None:
    if len(a_cell.domain) == 0: return 1

# Some of the recursive calls may encounter cases where the
# domain of the origin cell has been reduced to none, which
# indicates that there's no solution and that the program
# should halt. This if statement is written at the very
# beginning so that the recursive call could be termianted
# immediately if this were the case.

try:
    left = a_board.go_left(a_cell)
except ValueError:
    left = None

try:
    right = a_board.go_right(a_cell)
except ValueError:
    right = None

try:
    up = a_board.go_up(a_cell)
except ValueError:
    up = None

try:
    down = a_board.go_down(a_cell)
except ValueError:
    down = None

neighbors = [up, down, left, right]
# The Cell objects in the list are arranged in the same order
# as the constraint field of the Cell class (which stores
# the strings that represent the constraints regarding the
```

```
# cell's neighbors): the neighbor above ("up"), below ("down"),
# left and then right

up_ret = 0
down_ret = 0
left_ret = 0
right_ret = 0
# The point is to initialize these variables. If the recursive
# calls in the code below are executed, these variables
# will hold the return values of those calls

ret_vals = [up_ret, down_ret, left_ret, right_ret]
# The variables to store the return values are arranged in
# the same order as the list of Cell objects above: up,
# down, left and right

constr_strings = [["STU", "GTU"], ["STD", "GTD"],
        ["STL", "GTL"], ["STR", "GTR"]]
# A list that contains all the strings that represent
# constraints regarding neighbors.
# Arranged in the same order as the list of Cell objects
# above: the first sublist contains the two types of
# constraints for the neighbor above, the second sublist
# is for the neighbor below, followed by those for "left"
# and "right"


origin_row = a_cell.coord[0]
origin_col = a_cell.coord[1]
if a_cell.assign != None:
    # If the origin cell has been assigned a value, remove
    # this value from the domains of all other cells that
    # in the same row or column

    targets = []
    # Stores references to all cells that are in the same
    # column or row

    for i in range(0, 5):
        # Add to the list "targets" the cells that are in
        # the same column as the origin cell
        if i == origin_row:
            # Skip the origin cell itself
            continue
        targets.append(a_board.cells[i][origin_col])
```

```
        for i in range(0, 5):
            # Add to the list the cells that are in the same
            # row as the origin cell
            if i == origin_col:
                # Skip the origin cell itself
                continue
            targets.append(a_board.cells[origin_row][i])

        for i in range(len(targets)):
            current_cell = targets[i]
            if current_cell.assign == None:
                # If the current cell has yet to be assigned
                # a value, remove the origin's assigned value
                # from the current cell's domain, if applicable
                new_domain = copy.deepcopy(current_cell.domain)
                for j in range(len(current_cell.domain)):
                    if current_cell.domain[j] == a_cell.assign:
                        new_domain.remove(current_cell.domain[j])
                        break

                current_cell.domain = new_domain

    for i in range(len(neighbors)):
        if isinstance(neighbors[i], Cell):
            # If the Cell object looked for was returned by
            # the methods of the Board class
            # Python's isinstance function returns True when
            # the first parameter is an instance of the second

            if neighbors[i].assign == None:
                # If the cell is empty, the indented code below
                # will be run;
                # If the cell has been assigned a value,
                # the program will jump to the recursive call ahead

                new_domain = copy.deepcopy(neighbors[i].domain)
                # Since the following code involves a lot of removal
                # of elements in a list, the list of domain values
                # is copied into this variable "new_domain", which
                # will be used as a temporary variable from which
                # elements are removed. Once all removal is done,
                # neighbors[i].domain will be updated with the list
                # contained in new_domain.

                if a_cell.constr[i] == constr_strings[i][0]:
                # The list of Cell objects, the list of constraint
```

```
# strings and the list of return values all arrange
# their elements in the up-down-left-right order
# Therefore the same index can locate the particular
# element for the same neighboring cell

# The elements in constr_strings are ordered in such
# a way that constr_strings[i][0] is always the string
# that denotes a "smaller than" relation, whereas
# constr_strings[i][1] is the one that denotes a
# "greater than" relation

    if a_cell.assign != None:
        # If the origin cell has been assigned
        # a value

        for j in range(len(neighbors[i].domain)):
            if neighbors[i].domain[j] <= a_cell.assign:
                new_domain.remove(neighbors[i].domain[j])
                # Remove the values that are smaller
                # than or equal to the origin's
                # assigned value

elif a_cell.constr[i] == constr_strings[i][1]:
    # constr_strings[i][1] is always the string that
    # refers to a "greater than" relation

    if a_cell.assign != None:
        for j in range(len(neighbors[i].domain)):
            if neighbors[i].domain[j] >= a_cell.assign:
                new_domain.remove(neighbors[i].domain[j])

neighbors[i].domain = copy.deepcopy(new_domain)
# Updating neighbors[i].domain with the newer list
# of domain values contained in new_domain

if len(neighbors[i].domain) == 0:
    return 1
    # If an empty cell's domain has been reduced to none,
    # return 1, which indicates the puzzle has no solution
    # The return value will be caught by the function
    # that makes the call, which will then stop the program

    # The domain field of a Cell is "None" only if the Cell
    # has not been assigned a value, in which case the
    # entire code block will have been skipped due to the
    # "if neighbors[i].assign == None" statement above.
```

```
            # Therefore this line does not incur a runtime error.

        ret_vals[i] = forward_checking(a_board, neighbors[i],
            explored)
        """ This line is executed if the neighbor has been assigned
        a value or the neighbor's domain is not empty after the
        reduction."""

    return max(ret_vals)
    """ If any of the recursive calls returns one, there's no solution
    to the puzzle. If any of the four elements of "ret_vals" equals
    one, the function will return one. The preceding function that made
    the first call to forward_checking will stop the program if the
    return value is one and continue if it's zero."""


def identical_boards(prev_board: Board, curr_board: Board) -> bool:
    """ The function works in tandem with start_fc. It takes two Board
    objects and verifies whether they are identical, i.e. whether the
    assigned values and domains of each cell are identical between the
    two boards. It returns False as soon as a difference is spotted;
    if no difference is found after comparing all twenty-five cells,
    it returns True."""

    for i in range(0, 5):
        for j in range(0, 5):
            prev_cell = prev_board.cells[i][j]
            curr_cell = curr_board.cells[i][j]

            if prev_cell.assign != curr_cell.assign:
                return False

            if prev_cell.domain != curr_cell.domain:
                return False

            """ The other two fields of the Cell class, the coordinate
            and the list of constraints, are not compared because they
            are not altered over the course of forward checking. Only
            the assigned value and the list of domain values need to
            be compared."""

    return True


def calc_degree(a_board: Board, origin: Cell) -> int:
    """ The function takes a Board object and a Cell object as its
```

```
    parameters and returns the degree of the given cell, which
    equals the number of constraints the cell has regarding its
    **unassigned** neighbors."""

    degree = 0
    for i in range(len(origin.constr)):
        if (origin.constr[i] == "N/A") or (origin.constr[i] == "None"):
            continue

        """ The strings that represent constraints are all in a form
        similar to "STU", "GTD" and so on, as explained previously.
        Therefore the third letter of the string indicates which
        neighbor the constraint applies to."""
        if origin.constr[i][2] == 'U':
            neighbor = a_board.go_up(origin)
        elif origin.constr[i][2] == 'D':
            neighbor = a_board.go_down(origin)
        elif origin.constr[i][2] == 'L':
            neighbor = a_board.go_left(origin)
        elif origin.constr[i][2] == 'R':
            neighbor = a_board.go_right(origin)

        if neighbor.assign == None: degree += 1

    return degree


def start_fc(a_board: Board, a_cell: Cell) -> int:
    """ This is the overarching function for forward checking. It calls
    forward_checking on the given Cell object, located on the given Board,
    and returns 1 when there's no solution to the puzzle (same as how
    forward_checking behaves). If forward_checking returns 0, meaning that
    function ran without error, start_fc calls identical_boards to verify
    whether the board has been modified. If yes, start_fc repeatedly calls
    forward_checking on the same cell of the same board until the board is
    no longer modified, after which start_fc returns 0. The point is to
    ensure every other cell's domain is updated once a cell has been
    modified."""

    identical = False
    explored = set()
    while not identical:
        explored.clear()
        prev_board = copy.deepcopy(a_board)
        # Saves a copy of the original board
        failure = forward_checking(a_board, a_cell, explored)
```

```
        if failure:
            # forward_checking returns 1 when there's no solution to
            # the puzzle and returns 0 when it has run without error
            return 1
        identical = identical_boards(prev_board, a_board)

    return 0


def select_unassigned_cell(a_board: Board) -> Cell:
    """ The function takes a Board object as its parameter and returns
    a cell on the board, selected by the minimum-remaining-values (MRV)
    heuristic and, in case there's a tie, the degree heuristic as well."""

    ranking = []
    """ After the following for loop has completed, the list "ranking"
    will contain all cells on the board, ranked by the number of values
    in each cell's domain in ascending order."""

    for i in range(0, 5):
        for j in range(0, 5):
            current = a_board.cells[i][j]
            if current.assign == None:
                inserted = False
                for k in range(len(ranking)):
                    if len(ranking[k].domain) >= len(current.domain):
                        ranking.insert(k, current)
                        # Insert the current cell into the list,
                        # ahead of the first element that has more
                        # remaining values or the same number of remaining
                        # values
                        inserted = True
                        break
                if not inserted:
                    # Indicates the current cell has more remaining values
                    # than any element in the list
                    ranking.append(current)

    if len(ranking) == 1: return ranking[0]
    tied = [ranking[0]]
    # At this point the first element in the list "ranking" has the
    # least remaining values in its domain
    ind = 1
    while len(ranking[ind].domain) == len(tied[0].domain):
        # Loop through the list to find out if any other cell has the
        # same number of remaining values, i.e. whether there's a tie
```

```
        tied.append(ranking[ind])
        ind += 1
        if ind == len(ranking):
            break

    if len(tied) == 1: return tied[0]
    # Indicates there's no tie. Return the only element in the
    # list "tied"

    """ If there is a tie, call the calc_degree function on each
    cell in the list "tied" to rank these cells by their degrees in
    descending order."""
    degree_ranking = []
    for i in range(len(tied)):
        degree_ranking.append((i, calc_degree(a_board, tied[i])))
        # Appends a tuple whose first element is the cell's index in
        # the list "tied" and second element is the degree

    degree_ranking = sorted(degree_ranking, key=lambda pair: pair[1],
            reverse=True)
    """ In Python, lambda is an anonymous function. Here the function
    contains the expression that returns the second element of each
    tuple as the key for sorting, i.e. the degree of each cell. The
    parameter "reverse=True" instructs the function to sort in
    descending order; without this parameter, the function sorts in
    ascending order by default.

    After sorting, the first tuple in degree_ranking refers to the cell
    with the lowest degree. Since the first element of the tuple is the
    cell's index in "tied", subscript "tied" with this index to return
    the cell."""

    return tied[degree_ranking[0][0]]


def order_domain_values(a_cell: Cell) -> list:
    """ The function takes a Cell object, sorts its list of domain
    values in ascending order and returns the sorted list."""

    """ Python's sorted() function does NOT alter the original list;
    it simply creates a new list during sorting and returns the new,
    sorted list."""
    return(sorted(a_cell.domain))


def is_complete(a_board: Board) -> bool:
```

```
    """ The function takes a Board object as its parameter and loops
    through all cells on the board to verify whether each has been
    assigned a value. It returns False as soon as an unassigned
    cell is spotted. Otherwise it returns True after the for loop
    has completed."""

    for i in range(0, 5):
        for j in range(0, 5):
            if a_board.cells[i][j].assign == None:
                return False
    return True


def is_consistent(a_board: Board, a_cell: Cell, value: int) -> bool:
    """ The function takes a Board object, a Cell object that's part of
    the board, and a candidate value as its parameters. It first verifies
    whether the candidate value has been assigned to any other cell
    in the same row or column as the given cell. It then verifies whether
    the candidate value violates any of the constraints on the given cell.
    It returns False as soon as the candidate violates a condition;
    otherwise it returns True when all conditions have been met."""


    current_row = a_cell.coord[0]
    current_column = a_cell.coord[1]
    for i in range(0, 5):
        if i == current_column: continue
        if a_board.cells[current_row][i].assign != None:
            if a_board.cells[current_row][i].assign == value:
                return False

    for i in range(0, 5):
        if i == current_row: continue
        if a_board.cells[i][current_column].assign != None:
            if a_board.cells[i][current_column].assign == value:
                return False

    """ This part verifies whether the candidate value complies with
    all the constraints on the given cell."""
    constr = a_cell.constr
    for i in range(len(constr)):
        if (constr[i] == "N/A") or (constr[i] == "None"):
            continue
        if constr[i][2] == 'U':
            neighbor = a_board.go_up(a_cell)
        elif constr[i][2] == 'D':
```

```
            neighbor = a_board.go_down(a_cell)
        elif constr[i][2] == 'L':
            neighbor = a_board.go_left(a_cell)
        elif constr[i][2] == 'R':
            neighbor = a_board.go_right(a_cell)

        if neighbor.assign == None: continue
        if (constr[i][0] == 'S') and (value >= neighbor.assign):
            return False
        if (constr[i][0] == 'G') and (value <= neighbor.assign):
            return False

    return True


def backtrack(a_board: Board) -> bool:
    """ The function takes a Board object as its parameter and runs
    backtracking on the board. If a solution can be obtained, it
    calls generate_output and then returns True."""

    if is_complete(a_board):
        # If the assignment is complete, generate the output file
        # and then return True
        generate_output(a_board)
        return True

    selected = select_unassigned_cell(a_board)

    # The two lines below keep track of the row & column numbers
    # of the selected cell
    """ The reason is as deep copies of the board will be created in
    the code below, and the variable "selected" will need to be
    reassigned the Cell object on the newly-created board. """
    cell_row = selected.coord[0]
    cell_col = selected.coord[1]

    sorted_domain = order_domain_values(selected)
    for i in range(len(sorted_domain)):
        old_board = copy.deepcopy(a_board)
        # Created a deep copy of the board before the recursive calls
        # are made so that if the algorithm backtracks, the original
        # state of the board can be restored

        selected = a_board.cells[cell_row][cell_col]
        # As stated previously, "selected" needs to be re-assigned
        # the Cell object on the newly-created board
```

```
        if is_consistent(a_board, selected, sorted_domain[i]):
            selected.assign = sorted_domain[i]
            selected.domain = None
            if not start_fc(a_board, selected):
                # Run forward checking after the cell has been assigned
                # a value. Only make the recursive call to backtrack
                # if start_fc returns 0, which indicates forward checking
                # was completed without spotting any cell with an empty
                # domain
                if backtrack(a_board): return True

        a_board = copy.deepcopy(old_board)
        # The function only reaches this point when the candidate value,
        # sorted_domain[i], made the algorithm backtrack. In that case,
        # use the deep copy previously made to restore the board and
        # move onto the next iteration of the for loop

    return False


def generate_output(a_board: Board) -> int:
    """ This function is called when backtrack has obtained a solution.
    It takes the Board object passed by backtrack, asks the user for
    the output filename and writes the solution into a plain text
    file."""

    out_filename = input("""Now please enter below the output filename, e.g. "Output1.txt". The
filename is case-sensitive.\n""")
    with open(out_filename, 'w') as out_file:
        for i in range(0, 5):
            for j in range(0, 5):
                out_file.write(str(a_board.cells[i][j].assign))
                if j == 4:
                    out_file.write('\n')
                    # Insert the newline character at the end of each line
                else:
                    out_file.write(' ')
                    # Insert a space between each number on the line

    out_file.close()
    return 0


def main() -> int:
```

```
    in_filename = input("""Please enter below the input filename, e.g. "Input1.txt". The filename is case-sensitive.\n""")
    input_return = load_input(in_filename)
    # load_input returns a list whose first element is the list of
    # all cells on the board and second element is the list of constraints
    # of all twenty-five cells

    a_board = initialize_board(input_return[0], input_return[1])
    start_fc(a_board, a_board.cells[0][0])
    # Once the board has been initialied, apply forward checking once
    # before running backtracking

    backtrack(a_board)
    return 0


main()
```