Homework 3
CSE4051 Program Verification, Fall 2026
Woosuk Lee
due: 11/18(Wed), 23:59

- You must use Python 3 for this assignment.
- Submit one python file per problem via the submission system in the course website. Make sure that your files run without errors.
- You are not allowed to use any external libraries except for Z3 for Python.

Exercise 1 Given a set of integers and a target integer T, write a program that computes the number of all subsets of the given set whose sum is equal to T. The input will be given in a file named "input.txt" in the following format:

```
[number_1] [number_2] ... [number_n]
[T]
```

For example, the following input specifies the case where the given set is $\{1, 2, 3, 4, 5\}$ and the target sum is 5.

```
1 2 3 4 5
5
```

Then, the output should be 3, since there are three subsets whose sum is equal to 5: $\{5\}$, $\{2, 3\}$, and $\{1, 4\}$.

Using Z3's `If`. `If(condition, a, b)` is an expression whose value is `a` when the condition is true and `b` when it is false. For example, let `pick = Bool('pick')` indicate whether the number 5 is selected. Then `If(pick, 5, 0)` contributes 5 to the sum if selected, and 0 otherwise. The solver will decide the value of `pick`. Z3's `If` builds this conditional expression; Python's `if` statement cannot branch on an undecided Z3 condition.

Here is the starter code:

```python
from z3 import *

def subset_sum(numbers, target):
    # fill here
    return count

if __name__ == "__main__":
    # read numbers from input.txt
    with open("input.txt", "r") as f:
        lines = f.readlines()
        numbers = list(map(int, lines[0].strip().split()))
        target = int(lines[1].strip())
        print(subset_sum(numbers, target))
```

Exercise 2 Given a directed graph where nodes are associated with weights (integers), we want to find a cut that splits the graph into two disjoint subsets such that the sum of the weights of the nodes in one subset is equal to the sum of the weights of the nodes in the other subset. One subset (called *root side*) should contain all root nodes (nodes with no incoming edges), and the other subset (called *leaf side*) should contain all leaf nodes (nodes with no outgoing edges). The input will be given in a file named "input.txt" in the following format:

```
[n] [m] (number of nodes and edges)
[w1] [w2] ... [wn] (weights of the nodes)
[u1] [v1] (edge from u1 to v1)
...
[um] [vm] (edge from um to vm)
```

For example, the following input specifies a graph with 5 nodes and 4 edges. The indices of nodes are 0-based (i.e., from 0 to n-1). The weights of the nodes are 3, 2, 4, 3, and 6. There is an edge from node 0 to node 1, from node 0 to node 2, from node 1 to node 3, and from node 2 to node 4.

```
5 4
3 2 4 3 6
0 1
0 2
1 3
2 4
```

The output should be a sequence of integers representing the nodes in the root side of the cut (sorted in ascending order). If no such cut exists, the program should output "No cut". For example, for the above input, the output should be:

```
0 1 2
```

Here is the starter code:

```python
from z3 import *

def read_graph(filename):
    nodes = []
    edges = []
    weights = {}
    with open(filename, "r") as f:
        first_line = f.readline().strip().split()
        N = int(first_line[0])
        M = int(first_line[1])

        weight_line = f.readline().strip().split()
        for i in range(N):
            node = f"{i}"
            nodes.append(node)
            weights[node] = int(weight_line[i])

        for _ in range(M):
            edge_line = f.readline().strip().split()
            u = f"{int(edge_line[0])}"
            v = f"{int(edge_line[1])}"
            edges.append((u, v))
    return nodes, edges, weights

def count_balanced_cuts(nodes, edges, weights):
    # fill here
    # return root_side or None

if __name__ == "__main__":
    nodes, edges, weights = read_graph("input.txt")
    root_side = count_balanced_cuts(nodes, edges, weights)
    if root_side == None:
        print("No cut")
    else:
        print(' '.join(sorted(root_side)))
```

Exercise 3 Recall the N-Queens problem, which is to place N queens on an N x N chessboard such that no two queens threaten each other. In this exercise, we consider a variant of the N-Queens problem where we want to place N queens on an N x N chessboard such that the maximum Manhattan distance between any two queens is at most D_{max}. The Manhattan distance between two queens located at (r_1, c_1) and (r_2, c_2) is defined as $|r_1 - r_2| + |c_1 - c_2|$.

The input will be given as two integers N and D_{max} in a file named "input.txt" in the following format:

```
[N]
[Dmax]
```

For example, the following input specifies the case where N is 6 and D_{max} is 8.

```
6
8
```

The output should be the number of all possible arrangements of N queens on the chessboard satisfying the above condition, which is

```
4
```

because there are 4 such arrangements (as shown below):

.	Q	.	.	.	.
.	.	.	Q	.	.
.	.	.	.	.	Q
Q	.	.	.	.	.
.	.	Q	.	.	.
.	.	.	.	Q	.

.	.	.	Q	.	.
Q	.	.	.	.	.
.	.	.	.	Q	.
.	Q	.	.	.	.
.	.	.	.	.	Q
.	.	Q	.	.	.

.	.	.	.	Q	.
.	.	Q	.	.	.
Q	.	.	.	.	.
.	.	.	.	.	Q
.	.	.	Q	.	.
.	Q	.	.	.	.

.	.	Q	.	.	.
.	.	.	.	.	Q
.	Q	.	.	.	.
.	.	.	.	Q	.
Q	.	.	.	.	.
.	.	.	Q	.	.

Here is the starter code:

```python
from z3 import *

def count_n_queens_solutions_with_distance(N, Dmax):
    # fill here

if __name__ == "__main__":
    with open("input.txt", "r") as f:
        lines = f.readlines()
        N = int(lines[0].strip())
        Dmax = int(lines[1].strip())
        result = count_n_queens_solutions_with_distance(N, Dmax)
        print(result)
```

Exercise 4 Approximate computing aims to improve efficiency by allowing some loss of accuracy in computations. Given a N-bit bitvector expression, we want to find an approximate expression that differs from the original expression by at most K cases in their output for all possible inputs. Because there are 2^N possible inputs for an N-bit bitvector, two expressions differ by at most K cases if they produce different outputs for at most K inputs out of the 2^N possible inputs.

Since the XOR, addition, and shift operations are cheap to compute, we will only consider expressions of the form:

$$a \times (x << b) + c \times (x >> d) + e \times (x \oplus f) + g$$

where a, b, c, d, e, f, g are N-bit bitvector constants to be synthesized, and x is the input N-bit bitvector variable, and $\oplus$ is the XOR operator.

Note that the outputs are also N-bit bitvectors, and all operations are performed in the N-bit bitvector domain. Your task is to write a program that takes as input an integer N (the bit-width of the input bitvectors), an integer K (the maximum number of differing cases allowed), and a bitvector expression using variable x (an N-bit bitvector), and outputs an approximate expression of the above form that differs from the original expression by at most K cases in their output for all possible inputs.

The input will be given in a file named "input.txt" in the following format:

```
[N]
[K]
[expression]
```

where the first line contains an integer N (the bit-width of the input bitvectors), the second line contains an integer K (the maximum number of differing cases allowed), and the third line contains a bitvector expression using variables x which is an N-bit bitvector.

The output should be an approximate expression that differs from the original expression by at most K cases in their output for all possible inputs. If no such approximate expression exists, the program should output "No solution".

For example, for the following input

```
4
4
x + 2 * (x >> 1) ^ x
```

The output could be:

```
2 0 6 1 2 3 10
```

which represents the approximate expression:

```
2 * (x << 0) + 6 * (x >> 1) + 2 * (x ^ 3) + 10
```

This approximate expression differs from the original expression by at most 4 cases in their output for all possible 4-bit inputs (6, 7, 14, and 15).

Here is the starter code:

```python
from z3 import *

def read_expression(filename):
    with open(filename, "r") as f:
        lines = f.readlines()
        BV_SIZE = int(lines[0].strip())
        DIFF_BOUND = int(lines[1].strip())
        expr_str = lines[2].strip()
    return BV_SIZE, DIFF_BOUND, expr_str

def synthesize_expression(BV_SIZE, DIFF_BOUND, expr_str):
    x = BitVec('x', BV_SIZE)
    # parse str to z3 expression
    orig = eval(expr_str)
```

```python

    # Template: f(x) = a*(x << b) + c*(x >> d) + e*(x ^ f) + g
    a, b, c, d, e, f, g = BitVecs('a b c d e f g', BV_SIZE)

    # fill here
    # return None or a list of values of a, b, c, d, e, f, and g

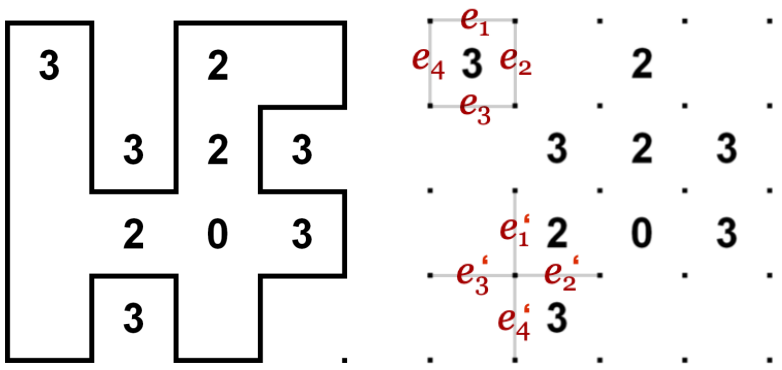
if __name__ == "__main__":
    BV_SIZE, DIFF_BOUND, expr_str = read_expression("input.txt")
    params = synthesize_expression(BV_SIZE, DIFF_BOUND, expr_str)
    if params is None:
        print("No solution")
    else:
        print(' '.join(map(str, params)))
```

You may refer to `https://z3prover.github.io/api/html/namespacez3py.html` for the Z3 Python API documentation to check how to use Z3 functions (e.g., `BitVecVal`, `substitute`, `simplify`, etc.).

Exercise 5 Slither is a puzzle played on a square grid where some cells contain clues (integers). Two rules:

1. Each number must be surrounded by that many edges.
2. All edges must form a single closed loop.

For example, the following figure shows a valid solution for a 5 x 5 grid with clues.



The first rule can be expressed: let e_1, e_2, e_3, e_4 be the four edges surrounding a clue cell with value 3 in the above figure. Then, we must have $e_1+e_2+e_3+e_4 = 3$, where each edge variable is 1 if the edge is present and 0 otherwise.

The second rule is more complex to express. We may approximate it by ensuring that every crossing should have either zero or two edges. This means that for each crossing point, if the point is used in the loop, it must have exactly two edges connected to it; otherwise, it should have no edges connected to it. In the above figure, it can be represented as $e_1' + e_2' + e_3' + e_4' = 0$ or $e_1' + e_2' + e_3' + e_4' = 2$ for each crossing point, where e_1', e_2', e_3', e_4' are the four edges connected to the crossing point in the above figure.

The problem of the above approximation is that it may produce multiple disjoint loops instead of a single closed loop because it does not ensure global connectivity; it only ensures local constraints at each crossing point.

Therefore, you can do the following: after finding a solution using the above constraints, check if the solution forms a single closed loop. If it does, output the solution. If not, add a constraint to eliminate the current solution and search for another solution. Repeat this process until a valid solution is found or there are no more solutions.

The input will be given in a file named "input.txt" as a square grid of size N x N where each cell contains either -1 (blank) or an integer clue (0 to 3). For example, the following input specifies the above 5 x 5 grid with clues.

```
3 -1 2 -1
-1 3 2 3
-1 2 0 3
-1 3 -1 -1
```

The output will be

```
+---+   +---+---+
| 3 | . | 2   . |
+   +   +   +---+
| . | 3 | 2 | 3
+   +---+   +---+
| .   2   0   3 |
+   +---+   +---+
| . | 3 | . | .
+---+   +---+   +
```

which can be generated by using the function `render_board` in the starter code.

Here is the starter code (`CLUES` is a list of lists representing the grid of clues; `n` is the number of cells per row/column; `H` and `V` are 2D lists of Z3 Boolean variables representing horizontal and vertical edges, respectively):

```python
from z3 import *

def read_clues(filename):
    with open(filename, "r") as f:
        lines = f.readlines()
        clues = []
        for line in lines:
            row = list(map(int, line.strip().split()))
            clues.append(row)

    n = len(clues)
    # Edge variables
    # H_i,j = horizontal edge below cell from (i,j) to (i,j+1)
    # V_i,j = vertical edge right of cell from (i,j) to (i+1,j)
    H = [[Bool(f"H{r}_{c}") for c in range(n)] for r in range(n+1)]
    V = [[Bool(f"V{r}_{c}") for c in range(n+1)] for r in range(n)]

```

```
    return clues, H, V, n

def solve_slitherlink(CLUES, H, V, n):
    # fill here!
    # return a model m

# -----------------------------
# Render a simple ASCII board:
# -----------------------------
def render_board(m, CLUES, H, V, n):
    def cell_char(i,j):
        c = CLUES[i][j]
        return "." if (c is None or c == -1) else str(c)

    lines = []
    for i in range(n+1):
        # top border line with horizontals
        t = []
        for j in range(n):
            t.append("+")
            t.append("---" if is_true(m[H[i][j]]) else "   ")
        t.append("+")
        lines.append("".join(t))
        if i < n:
            # middle line with verticals and clues
            mline = []
            for j in range(n+1):
                mline.append("|" if is_true(m[V[i][j]]) else " ")
                if j < n:
                    mline.append(f" {cell_char(i,j)} ")
            lines.append("".join(mline))
    print("\n".join(lines))

if __name__ == "__main__":
    CLUES, H, V, n = read_clues("input.txt")
    m = solve_slitherlink(CLUES, H, V, n)
    render_board(m, CLUES, H, V, n)
```

Exercise 6 Given an R by C Minesweeper board and a total mine count M, classify every unrevealed cell. An entry is either -1 (unrevealed) or a clue from 0 to 8 counting mines in horizontal, vertical, and diagonal neighbors within the board. Revealed cells never contain mines.

Consider *all* placements satisfying every clue and the total mine count. Output `M` for a cell that always contains a mine, `S` for one that is always safe, and `?` if both are possible. If no placement exists, `classify_cells` must return `None`; the starter code then prints "No solution". For example, a 1 by 1 board with clue 1 and $M = 0$ is impossible: it has no neighboring cells.

Assume $R, C \geq 1$ and $0 \leq M \leq RC$. The input will be given in a file named "input.txt" in the following format:

```
[R] [C] [M]
[cell_00] ... [cell_0(C-1)]
... (R rows of C integers each)
```

Example input:

```
2 5 2
1 -1 1 -1 -1
1 1 1 -1 -1
```

Example output:

```
1 M 1 S ?
1 1 1 S ?
```

Column 1 contains a forced mine; column 3 is safe; exactly one cell in column 4 contains the other mine. Column indices are 0-based. Keep revealed clues unchanged and separate output entries with spaces.

HINT: Create a Z3 solver object `s` with `s = Solver()` and add the board constraints. After checking that a placement exists, test each unrevealed cell with and without a mine. For each test, `s.push()` saves the current constraints, `s.add(...)` adds the assumption, and `s.check()` tests it. Then `s.pop()` removes constraints added since `push()`, restoring the board constraints.
Here is the starter code:

```python
from z3 import *
def classify_cells(R, C, M, board):
    # Return None if no placement satisfies all clues and M.
    # Otherwise return R rows of strings: clues, "M", "S", "?".
    # fill here
    pass
if __name__ == "__main__":
    with open("input.txt", "r") as f:
        R, C, M = map(int, f.readline().split())
        board = [list(map(int, f.readline().split()))
                 for _ in range(R)]
    result = classify_cells(R, C, M, board)
    if result is None:
        print("No solution")
    else:
        for row in result:
            print(" ".join(row))
```

Exercise 7 A binary image is an N by N grid whose entries are 0 or 1. Instead of the image itself, you are given the sum of each row, the sum of each column, and the sums of the two main diagonals. Write a program that reconstructs an image consistent with these sums and determines whether the reconstruction is unique.

Let $x_{r,c}$ denote the entry in row r and column c, using 0-based indices. The main diagonal sum is $\sum_{i=0}^{N-1} x_{i,i}$, and the other diagonal sum is $\sum_{i=0}^{N-1} x_{i,N-1-i}$. When N is odd, the center cell contributes to both sums. Assume $N \geq 1$ and every given sum is between 0 and N. The input will be given in a file named "input.txt" in the following format:

```
[N]
[row_sum_0] ... [row_sum_(N-1)]
[column_sum_0] ... [column_sum_(N-1)]
[main_diagonal_sum] [other_diagonal_sum]
```

If the sums are inconsistent, output "No solution". Otherwise, print "Unique" or "Multiple" on the first line, followed by any satisfying image as N rows of space-separated 0s and 1s. Two images are different if at least one cell differs; rotations and reflections count as different images.

Example input:

```
3
2 1 1
1 2 1
2 2
```

Example output:

```
Unique
1 0 1
0 1 0
0 1 0
```

For comparison, if $N = 2$, all row and column sums are 1, and both diagonal sums are 0, the output should be "No solution".

HINT: After finding one image, add a constraint requiring at least one cell to differ from it, then check for another image.

Here is the starter code:

```python
from z3 import *

def reconstruct_image(N, rows, cols, main, other):
    # Return ("No solution", None), ("Unique", image), or
    # ("Multiple", image). image is a list of N lists of ints.
    # fill here
    pass

if __name__ == "__main__":
    with open("input.txt", "r") as f:
        N = int(f.readline())
        rows = list(map(int, f.readline().split()))
        cols = list(map(int, f.readline().split()))
        main, other = map(int, f.readline().split())
    status, image = reconstruct_image(N, rows, cols, main, other)
    print(status)
    if image is not None:
        for row in image:
            print(" ".join(map(str, row)))
```

Exercise 8 There are $T \times P$ golfers who want to play for W weeks. Each week, they must be divided into T teams of exactly P golfers each. Every golfer must play on exactly one team each week, and no pair of golfers may belong to the same team in more than one week.

Write a program that finds any schedule satisfying these conditions, or outputs "No schedule" if none exists. Golfers are numbered from 0 to $T \times P - 1$, and teams are numbered from 0 to $T - 1$ independently each week. Assume $T, P, W \geq 1$. The input will be given in a file named "input.txt" in the following format:

```
[T] [P] [W]
```

Output W lines, one per week. Each line must contain $T \times P$ space-separated integers, where the integer in position i is the team assigned to golfer i that week. Any valid schedule is accepted.

Example input:

```
2 2 3
```

Possible output:

```
0 0 1 1
0 1 0 1
0 1 1 0
```

The pairs are $(0, 1)$ and $(2, 3)$ in week 0, $(0, 2)$ and $(1, 3)$ in week 1, and $(0, 3)$ and $(1, 2)$ in week 2. Each pair occurs exactly once. For input `2 2 4`, the output should be "No schedule": each golfer has only three possible partners, but would need four distinct partners.

HINT: Use an integer variable for each golfer's team in each week, with a value from 0 to $T - 1$. Require exactly P golfers on each team each week. For every pair of golfers, count the weeks in which their team numbers are equal; this count must be at most 1.

Here is the starter code:

```python
from z3 import *

def schedule_golfers(T, P, W):
    # Return None if impossible; otherwise return W lists,
    # each containing T * P integer team assignments.
    # fill here
    pass

if __name__ == "__main__":
    with open("input.txt", "r") as f:
        T, P, W = map(int, f.readline().split())
    schedule = schedule_golfers(T, P, W)
    if schedule is None:
        print("No schedule")
    else:
        for week in schedule:
            print(" ".join(map(str, week)))
```

Exercise 9 Use Z3 to cover an R by C board with K rectangular tiles. Tile i has height h_i and width w_i in grid cells. Use every tile exactly once, with no gaps, overlaps, or parts outside the board. Tile edges must follow the grid lines.

Each tile may be rotated by 90 degrees, which simply swaps its height and width. Tiles may not be cut or resized. Tiles with identical sizes still count as separate tiles.

Assume $R, C, K \geq 1$ and all tile dimensions are positive integers. Tiles are numbered from 1 to K in input order. The input will be given in a file named "input.txt" in the following format:

```
[R] [C] [K]
[h_1] [w_1]              (height and width of tile 1)
... (K lines, one per tile)
```

Print R rows of C tile IDs (1 to K), separated by spaces, showing which tile covers each cell. Any valid arrangement is accepted. If none exists, print "No tiling".

Example input:	Possible output:
3 4 2	1 1 2 2
2 3	1 1 2 2
2 3	1 1 2 2

Each tile is originally 2 cells high and 3 cells wide. Rotating both makes them 3 cells high and 2 cells wide, so they fit side by side on the board.
Here is the starter code:

```python
from z3 import *
def tile_board(R, C, tiles):
    # tiles contains (height, width) pairs, in input order.
    # Return None if no tiling exists; otherwise return
    # R rows of C tile IDs, numbered from 1 to len(tiles).
    # fill here
    pass
if __name__ == "__main__":
    with open("input.txt", "r") as f:
        R, C, K = map(int, f.readline().split())
        tiles = [tuple(map(int, f.readline().split()))
                 for _ in range(K)]
    result = tile_board(R, C, tiles)
    if result is None:
        print("No tiling")
    else:
        for row in result:
            print(" ".join(map(str, row)))
```