This is an individual exam. Please do not discuss this with your fellow students or seek help from anyone. You have 48 hours to complete it after you have taken the exam.

Your goal is to develop an Auto Grader system. This Auto Grader will have a limited set of features. Students will have to write a solution to a problem presented to them. The problem and the test cases for the problem will be defined by the instructor. The students will have to write their solution to the problem. In our case, there will be only 1 problem that is presented to the students. The students will have to write their solution and their solution will automatically be tested against the test cases and they are graded for correctness.

Digging a little deeper, the Auto Grader should have the following features.

Auto Grader will present the problem to the student and the student will have to load the solution (can either type in a window or upload the solution). The Auto Grader will start the grading process after either the student uploads the file or clicks on a submit button.

Access to Auto Grader should be restricted to only SJSU students. Please create an account for narayan.balasubramanian@sjsu.edu so I can test the system.

The students will have to type in their solution to the problem on a window or load a file that contains the solution. Their responses will be graded. If 4 out of 8 test cases pass, the students will get a 50% grade. If  8 out of 8 test cases pass, the student will get a 100% grade.

Store the results of the problem in a datastore. Store the students id and their best score. The students are allowed to submit a solution as many times as they want.

Consider an example:

The problem is defined as:

# Knight Attack

A knight and a pawn are on a chess board. Can you figure out the minimum number of moves required for the knight to travel to the same position of the pawn? On a single move, the knight can move in an "L" shape; two spaces in any direction, then one space in a perpendicular direction. This means that on a single move, a knight has eight possible positions it can move to. (see end of document for a picture)

Write a function, knight_attack, that takes in 5 arguments:

n, kr, kc, pr, pc

n = the length of the chess board

kr = the starting row of the knight

kc = the starting column of the knight

pr = the row of the pawn

pc = the column of the pawn

The function should return a number representing the minimum number of moves required for the knight to land on top of the pawn. The knight cannot move out of bounds of the board. You can assume that rows and columns are 0-indexed. This means that if n = 8, there are 8 rows and 8 columns numbered 0 to 7. If it is not possible for the knight to attack the pawn, then return None.

For this problem, the students will be graded against test cases which will look like:

```
def test_01():
    assert knight_attack(8, 1, 1, 2, 2)) == 2
def test_02():
    assert knight_attack(8, 1, 1, 2, 3) == 1
def test_03():
    assert knight_attack(8, 0, 3, 4, 2) == 3
def test_04():
    assert knight_attack(8, 0, 3, 5, 2) == 4
def test_05():
    assert knight_attack(24, 4, 7, 19, 20) == 10
def test_06():
    assert knight_attack(100, 21, 10, 0, 0) == 11
def test_07():
    assert knight_attack(3, 0, 0, 1, 2) == 1
def test_08():
    assert knight_attack(3, 0, 0, 1, 1) is None
```

The students will be provided a template file like:

```
def knight_attack(n, kr, kc, pr, pc):
```

The student has to fill in their solution and then receive a grade.

A couple of things to note:
The individual test cases should take less than 2 seconds to run.

If any individual test takes more than 2 seconds to run, abort and award the student a 0.

If the code submitted by the student does not compile, award a 0.

Students are allowed to submit the code only in Python.

You are allowed to write this code in any language on any cloud platform.

For reference, a sample solution to the above problem can be:

```
from collections import deque
# given the position of the knight on the board, this helper
# function returns  all the possible locations for the knight
# kr and kc can run from 0 - n-1
# there are 8 positions that a knight can move
def knight_positions(n, kr, kc):
    possible_positions = []
    # knight moves in a L shape, 2 steps in either direction and
then 1 step in either direction
    for r in [-2, 2]:
        for c in [-1, 1]:
           row_pos = kr + r
           col_pos = kc + c
           possible_positions.append((row_pos, col_pos))
    for r in [-1, 1]:
        for c in [-2, 2]:
           row_pos = kr + r
           col_pos = kc + c
           possible_positions.append((row_pos, col_pos))
    # now these positions can be out of bounds, if so dont
include in final list
    possible_positions = is_bounded(n, possible_positions)
    return possible_positions
def is_bounded(n, positions):
    valid_positions = []
    for position in positions:
        row, col = position
        row_bound = 0 <= row < n
        col_bound = 0 <= col < n
        if row_bound and col_bound:
            valid_positions.append((row, col))
    return valid_positions
```

```
def knight_attack(n, kr, kc, pr, pc):
    visited = set()
    n = bfs(n, kr, kc, pr, pc, visited)
    return n
# return shortest path from knight position to pawn
def bfs(n, kr, kc, pr, pc, visited):
    q = deque()
    moves = 0
    q.append((kr, kc, moves))
    pawn_location = (pr, pc)
    while q:
        cur_row, cur_col, move  = q.popleft()
        cur_pos = (cur_row, cur_col)
        #print("current position ", cur_pos, "moves = ", move)
        if cur_pos in visited:
            continue
        visited.add(cur_pos)
        if cur_pos == pawn_location:
            return move
        for position in knight_positions(n, cur_row, cur_col):
               r, c = position
               q.append((r, c, move+1))

    return None
```

Some points to help you get started:

- Start with a high-level architecture in mind.
- Implement a small feature and then test it.
- Integrate code only after it has been tested and after it works

Please turn in the following:

- An architecture diagram
- An explanation about your choice of frameworks, tools
- Limitations and strengths of your implementation
- The output should content the result of each test and how long each test took to run
  Eg:

```
test knight-attack.py
> building source...
> executing 8 tests...
test_00 [PASS] 50ms
test_01 [PASS] 48ms
```

```
test_02 [PASS] 49ms
test_03 [PASS] 48ms
test_04 [PASS] 52ms
test_05 [PASS] 55ms
test_06 [PASS] 46ms
test_07 [PASS] 48ms
> 8/8 tests passed
```

- If you happen to find existing code that helped you achieve this, include a pointer to the code and steps you took to tweak it for your implementation

If you do not get the complete project to work, do not fret. There are a lot of things like architecture, approach, ideas to break down the problem etc that can contribute to your final score.

When you do get the project working, please let me know. I would love to poke around and test it with some inputs.

Good luck!!!

Example movement of a knight on a chess board