**Q1: Conference Reviewing (20 pts, 5 pts each)**

Say that we are organizing a conference and researchers can submit papers (manuscripts) to our conference. We also have reviewers who evaluate the submitted papers and score them. Each score is an integer between 1 and 5, where:

* 1 corresponds to "Strong Reject"
* 2 to "Reject"
* 3 to "Borderline"
* 4 to "Weak Accept"
* 5 to "Strong Accept"

We store papers' scores as a matrix in list of lists representation. Let's call this matrix M. Each row of M corresponds to one paper, and each column corresponds to one reviewer. M[i][j] stores the score of paper i according to reviewer j (between 1-5). If reviewer j did not review paper i, then M[i][j] = -1.

For example,

M = [[1, 2, -1, -1],

[-1, 3, -1, 3],

[2, 5, 3, 4],

[-1, 2, -1, 2],

[1, 1, 1, -1]]

**(a)** Implement the function paper\_stats(M) such that it calculates the min score, max score, and average score of each paper. It should store this information in a list of lists and return it. For the example above, it should return:

[[1, 2, 1.5],

[3, 3, 3.0],

[2, 5, 3.5],

[2, 2, 2.0],

[1, 1, 1.0]]

You should ensure that you do NOT consider -1 scores (indicating paper i was NOT reviewed by reviewer j) in the min, max and average calculations.

You can assume that each paper will have at least one review.

**(b)** Implement the function find\_best(S, n) which takes as input the statistics S computed by paper\_stats (function you wrote in part a) and an integer n. This function should find the best n papers according to papers' average scores. Return the indices of these papers in a list in decreasing order of scores.

For example, if n=4, you need to find the best 4 papers according to their average score. For the example above, find\_best(S, 4) should return [2, 1, 3, 0].

**(c)** Implement a function called tough\_reviewers(M) which takes as input the matrix M containing papers \* scores from (a). This function should return the reviewers who never gave a score above 3 for any paper.

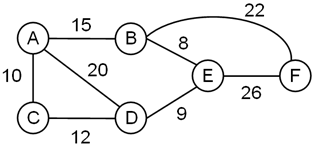
In the example above, reviewers 0 and 2 never gave scores above 3, then the return value of tough\_reviewers(M) should be: [0, 2].

**(d)** If a certain paper is among the best 15 papers and it has been reviewed by *at least one tough reviewer*, we call this paper an "award nominee".

Implement a function called award\_nominees(M) where M is the papers \* scores matrix. This function should return a list of award nominee papers. In the example above, the return value of award\_nominees is: [2, 0, 4]. Make sure to order the nominees according to decreasing average score.

**Q2: Graph Calculations (20 pts, 5 pts each)**

A **graph** is a data structure that represents relationships between vertices. For example, consider the following graph which contains 6 vertices named A, B, C, D, E, F. The numbers next to the edges represent edge weights.



A graph such as this can be stored in many different ways. In graph.txt, you are given one of those ways. Inspect its format and contents. You will see that:

* The first line contains the names of the vertices.
* After that, each line represents an edge between two vertices.

**(a)** Implement the function read\_graph(filename) such that:

* filename is the name of the input file, e.g., graph.txt.
* Your function should return the graph stored as a dictionary, in the following format.
* There should exist one key-value pair for each vertex in the graph. Its key should be the name of the vertex. Its value should be a list of tuples, where each tuple contains one connected vertex and the weight of the edge between this vertex and the connected vertex.

For example, here is how the return value should look like for the above example. Make sure that data types are correct in your dictionary (vertex names are strings, edge weights are integers):

{'A': [('B', 15), ('C', 10), ('D', 20)], 'B': [('A', 15), ('E', 8), ('F', 22)], 'C': [('A', 10), ('D', 12)], 'D': [('A', 20), ('C', 12), ('E', 9)], 'E': [('B', 8), ('D', 9), ('F', 26)], 'F': [('B', 22), ('E', 26)]}

Order of edges in the lists does not matter.

**Hint:** When you are adding an edge between vertex X and Y, you need to make two additions to the dictionary. First one is for X's key, second one is for Y's key.

**(b)** Implement the function largest\_weight(graph\_dict) which takes as input the dictionary representation of the graph (graph\_dict) as constructed in part (a) and behaves as follows.

largest\_weight should find the edge with the largest weight in the whole graph (call this edge x) and return which two vertices edge x is connecting. For example, if x is between vertices v1 and v2, then largest\_weight should return the tuple: (v1, v2). It is also OK to return (v2, v1).

**(c)** Implement the function remove\_edge(graph\_dict, vX, vY) which takes as input the dictionary representation of the graph (graph\_dict) and two vertices vX, vY as inputs. If an edge exists between vX and vY, this function should remove that edge and return the resulting dictionary.

**Hint:** When you are removing edge B-F, you need to make two deletions from the dictionary. One from key B, one from key F.

For example:

After remove\_edge(graph\_dict, 'B', 'F'):

{'A': [('B', 15), ('C', 10), ('D', 20)], 'B': [('A', 15), ('E', 8)], 'C': [('A', 10), ('D', 12)], 'D': [('A', 20), ('C', 12), ('E', 9)], 'E': [('B', 8), ('D', 9), ('F', 26)], 'F': [('E', 26)]}

**(d)** A list of vertices is said to form a **clique** if and only if there is an edge between EVERY pair of vertices in the list. For example, the list ['A', 'C', 'D'] forms a clique because the graph contains edges A-C, A-D, C-D. However, the list ['B', 'D', 'E'] does not form a clique since there is no edge between B-D.

Implement the function check\_if\_clique(graph\_dict, v\_list) which returns True (boolean) if v\_list forms a clique according to graph\_dict, and False (boolean) otherwise.

Sample executions:

check\_if\_clique(graph\_dict, [‘A’, ‘C’, ‘D’]) → True

check\_if\_clique(graph\_dict, [‘B’, ‘D’, ‘E’]) → False

check\_if\_clique(graph\_dict, [‘A’, ‘B’, ‘C’, ‘D’]) → False

**Q3: Exception Log (15 pts)**

In software systems, it is common to maintain a "log file" which records events, errors or exceptions that occur while the software is running. The log file is reviewed by system admins to check for abnormal behavior, cyberattacks, etc. In this question, you will write code to maintain a simple exception log file to record exceptions that occur in a Python function.

Open q3.py and find the function divide\_elementwise(a,b). This function:

* Takes as input two matrices a and b in list of lists representation
* Performs elementwise division: divide a[i][j] by b[i][j]
* A caveat: a and b originally contain string elements, therefore the strings are converted to floats by divide\_elementwise before division

You may assume a and b are square matrices (number of rows = number of columns).

There are multiple exceptions that may occur in divide\_elementwise. You should modify this function so that these exceptions are handled. Furthermore, you should ensure that the appropriate exception log messages are appended to exceptionlog.txt.

Here are the potential exceptions and how you should handle them:

1. A string element cannot be converted to float since it contains text. Determine which exception will occur and catch it. When you catch it, append a one-line message to exceptionlog.txt: "Exception! Cannot convert string to float."
2. Division by zero. Determine which exception will occur and catch it. When you catch it, append a one-line message to exceptionlog.txt: "Exception! Cannot divide by zero."
3. The sizes of a and b are different. To address this case, **you need to implement your own type of exception**. Raise your exception and catch it where appropriate. When you catch it, append a one-line message to exceptionlog.txt: "Exception! Matrix sizes are different."

If divide\_elementwise runs without any exceptions, append a one-line message to exceptionlog.txt: "divide\_elementwise() ran without any exceptions."

After every execution of divide\_elementwise, regardless of whether an exception occurred or not, always append a one-line message to exceptionlog.txt: "Exiting divide\_elementwise() now."

**Your task:** Rewrite the divide\_elementwise function to achieve the exception handling behavior above.

* You must handle all exceptions within divide\_elementwise. Do not handle them within a main() function or elsewhere.

**Q4: Santa's Elves (15 pts)**

Santa Claus has many elves (numbered 0, 1, 2, ..., N) who act as his helpers. Some of these elves are great friends with each other, so they gossip frequently. However, some of them dislike each other, and they won't gossip.

Information regarding which elves gossip with which elves is given to you in a matrix denoted by G. This is an (N+1)\*(N+1) matrix.

* G[i][j] = 1 indicates that elf i gossips (tells secrets) to elf j.
* G[i][j] = 0 indicates that elf i does not gossip (does not tell secrets) to elf j.

When an elf learns a secret, s/he will quickly tell it to all other elves s/he gossips to. For example, when elf 2 learns a secret, she can tell the elves [9, 24, 27]. Then, each of these elves who learned the secret will go and tell the other elves they gossip to. For example, 9 may tell the secret to [0, 3, 5] and 24 may tell the secret to [25, 33], and 27 may tell the secret to [30, 35]. Then, each of these elves will tell the elves they gossip to, and so on and so forth... The secret disseminates quickly!

**Task**: Santa Claus has a big secret that he can no longer hold inside. He will tell the secret to exactly one elf but his secret will disseminate quickly. He would like to figure out: If he tells the secret to elf a, will it reach elf b?

Help Santa Claus by implementing the function is\_reachable(G, a, b) which returns True (boolean) if Santa's secret will reach b (starting from a), and False otherwise.

**Important**:

* You must solve this question using RECURSION.
* Do not add new parameters (arguments) to is\_reachable(G, a, b).

**Q5: Agents in Environments (20 pts)**

We are going to implement a simple robotics environment using Object Oriented Programming. We will have an agent interacting with this environment and receiving some signals as a result of this interaction. The environment can be something like a maze or a street where the agent can act, for example by walking or driving.

1. The current state of the environment can be observed by the agent via its sensors, for example, a camera. Using the camera, the agent can take a picture of the environment at any time, which is the current *state* of the environment.
2. Given the current state, the agent will perform an action, for example, by walking one unit in the environment.
3. The state of the environment will change according to the action of the agent. The *next state* of the environment as well as a feedback signal will be returned to the agent by the environment. The feedback signal, which is called the *reward*, will tell the agent how well the agent did as a result of the action taken by the agent.

These three steps will be performed in the given order in a loop for the agent to interact with the environment and collect data in the process. The data that the agent collects is a tuple composed of (state, action, next\_state, reward) which denotes the current state at that time, action taken by the agent according to the current state, next\_state of the environment after taking the action, and the reward that the agent will receive as a result of the action.

**Part 1 (10 pts)**

In q51.py, implement Environment and Agent classes to simulate the interactions of the agent with the environment as formulated above. The interaction loop below should work without a problem and return the data as a list of tuples collected by the agent.

1. Note that both Environment and Agent are template classes, which means that they act as a placeholder. Initially, you can return an empty list [] for the state and the next state, None for the action, and 0 for the reward. These functionalities will be implemented by specific environment and agent examples as you will see next.

**Important:** The code snippet given above should not yield any errors when tested on your Environment and Agent classes, which means that you need to implement every method called there in the correct class.

1. To prevent an infinite loop, kill the agent with 0.01 probability in is\_alive method of Agent class. Similarly, terminate the environment with 0.001 probability in is\_terminal method of Environment class. For that purpose, generate a random number between 0 and 1 using the random module. In is\_alive(), kill the agent by returning False if the generated number is less than 0.01. Otherwise return True to signal that the agent is alive. Terminate the environment by doing the same in is\_terminal() but with 0.001 probability.

**Part 2 (10 pts)**

In q52.py, implement the Maze and Rat classes as described below using INHERITANCE. Maze class should inherit from Environment and the Rat class should inherit from Agent.

1. The state of the maze will be a 4x4 grid, which you can represent as a list of lists. Each element represents the occupancy of that location in the maze. In Maze.\_\_init\_\_(), initialize Maze.state with all zeros except for (0,0) which is 1 to denote the starting location of the agent: [[1,0,0,0], [0,0,0,0], [0,0,0,0], [0,0,0,0]].
2. The agent initially starts at (0,0) and can move up, down, left, right depending on where it is currently in the maze. If there is no place to move, it stays in the current location. For example at (0,0), it cannot go up or left but only to right (0,1) or down (1,0).
3. You need an attribute to represent the position of the rat in the maze. You can initialize Rat.position as (0,0) in Rat.\_\_init\_\_(). Recognize how this is consistent with the initial Maze.state. The rat randomly chooses one of the four actions (up, down, left, right). Use the random class to choose the action randomly. In act method, update the position of the Rat and return the action to the environment. For your convience in the next step while updating the environment, in addition to the action, return the current\_position and the next\_position of rat as well, as a triple, in that order.
4. Given the current state and the action triple, update the state of the environment in update() method of Maze class. Update the environment by simply setting the current location of the rat in the Maze.state to 0 and the next location to 1. Note that you have access to both the current and the next position from the previous step.

**Important Warning**: Remember that lists are just pointers. If you write "list\_a = list\_b", you are just creating another pointer to the same memory location. Without fixing this problem, all the states will be identical in the resulting data because they are all pointing to the same memory location. Remember aliasing and how to fix it? **Hint:** You can use the copy() method to copy a list. For a list of lists, you need to loop over the outer list and copy each inner list with copy().

1. As long as the agent moves, it gets a reward of 1, otherwise 0. For example, if the action is up at (0,0), the agent cannot change its location due to running into the maze border, therefore, it gets a reward of 0. You can simply compare the current and the next location of the rat to see if it moved.

A sample execution is provided in "q52\_sample\_output.py".

**Q6: Bubblin' with Coordinates (15 pts)**

You are given a list L of d-dimensional points, where each point is represented as a tuple. For example, when d=2, L can be: [(3,4), (8,8), (0,0), ..., (9,1)]. When d=3, L can be: [(1,4,0), (2,2,5), (0,0,0), ..., (9,1,6)].

You are also given a separate d-dimensional point called the **"reference point"**, denoted by pt.

**(a)** [10 pts] Your task is to sort L in increasing order of the points' Euclidean distances to pt, using **Bubble Sort**. The Euclidean distance between two points x = (x1, x2, ..., xd) and y = (y1, y2, ..., yd) is defined as:

euc\_dist(x,y) = sqrt((x1-y1)^2 + (x2-y2)^2 + ... + (xd-yd)^2)

Implement your code in a function called bubble\_coords(L, pt). You can define and use additional functions if you want. bubble\_coords should have two return values:

1. The sorted list of points
2. A list containing how many pairwise swap operations were made in each pass of bubble sort. For example, [8, 3, 2, 1, 0] tells us that bubble sort made 5 total passes; 8 swaps were made in the first pass, 3 swaps were made in the second pass, 2 swaps were made in the third pass, and so on.

Notes and remarks:

* You MUST use bubble sort; otherwise you will receive 0.
* Your function should work for any number of dimensions d >= 1.
* If two points in L have equal distance to pt, what should you do? Behave the same way as the original bubble sort algorithm would behave.

**(b)** [5 pts] What is the time complexity of your solution in big-O notation? Write it in your Python file. Explain how you arrive at your answer.