Aaron Fox
CECS 622
Dr. Elmaghraby

Homework 3

Problem 1:

1. The "coupled tank" problem is described as a set of two differential equations as follows:

$$A_1.(dh_1/dt) = F_1 - F_2$$
$$A_2.(dh_2/dt) = F_2 - F_0$$

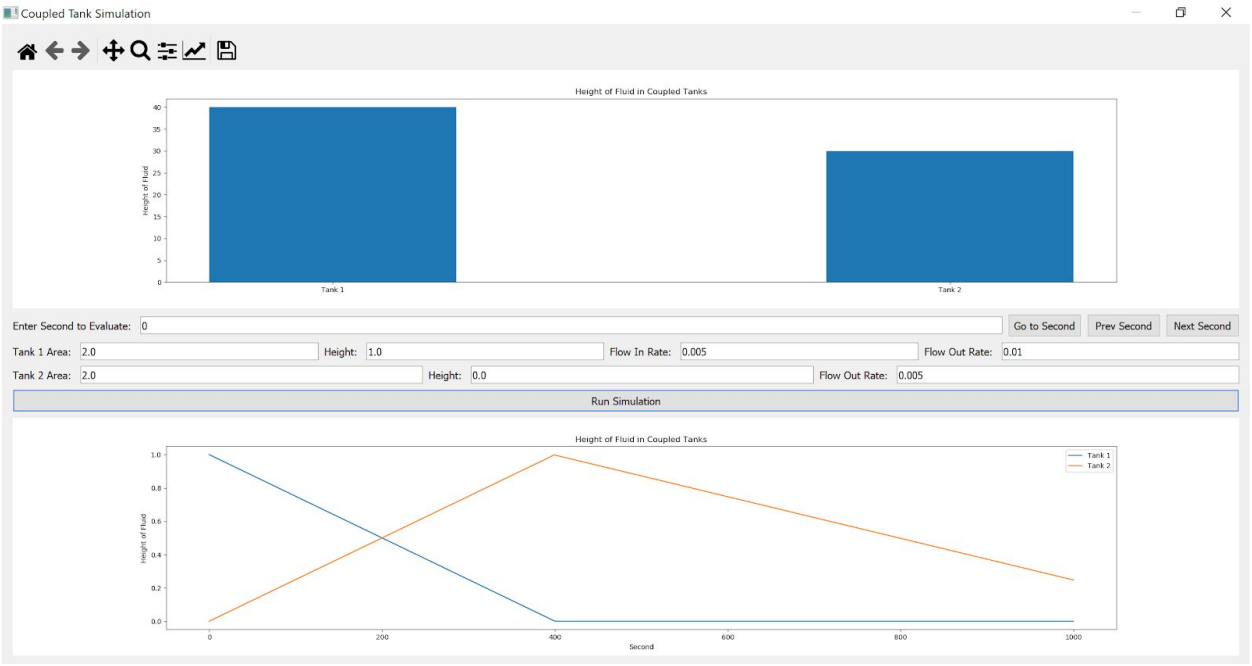
where A is the area of the tank, h is the height of the fluid, and F is the rate of flow.
a. Write a simulation program to solve the following

Introduction: A simulation program was written to simulate the coupled tank problem as described in the Assignment 3 prompt on Blackboard. The program was written in Python using the PyQt cross-platform GUI toolkit and the matplotlib plotting library. The user can begin the simulation by either clicking the "Run Simulation" button using the default input values as given in the Assignment 3 prompt or by inputting custom values for the areas, flow inputs/outputs, and the initial heights of the fluids in the tanks. The simulation then allows the user to visualize the height of fluid in each tank with a bar graph at every desired second that is run at the top half of the GUI. The user can also navigate to the previous and following seconds after the second they are currently viewing. The user can additionally visualize a line graph of the heights of the fluids in the two tanks over all the seconds over the duration of the simulation so that they can immediately see the results and general trends of the input simulation over time. The simulation aided in visualizing the effects of flow, initial heights, and the areas of a coupled tank; it could be put to practical use by allowing those who are designing the tanks to control the flow rates with a valve based on their desired behavior. The results could also provide important information regarding the ideal areas of tanks to design and what initial fluid levels to place in the tank designs.

Approach: The GUI and visualizations were set up in GUI windows using PyQt and the graphing library of matplotlib. The bar graphs were implemented so that they could display the exact values of the height of the fluid in the tanks at any input second that the user enters by pressing the "Go to Second" button; the user could also press the "Prev Second" and "Next Second" buttons to navigate either +/- one second of the current second that the user is currently viewing.

The line graphs of the tanks at the bottom half of the GUI displayed the effects of the input parameters over time on the simulation, as specified by the variable `self.seconds_to_run`, which is set in the initialization of the GUI window.

An image of the GUI can be seen below:



The GUI as implemented in this project

The user could input the variables that corresponded to the given equations in the prompt. With regard to the equation

$$A_1.(dh_1/dt) = F_1 - F_2$$
$$A_2.(dh_2/dt) = F_2 - F_0$$

where, in the GUI, 'Tank 1 Area' (`self.tank_1_area_value`) is equivalent to A_1, 'Tank 1 Height' (`self.tank_1_height_value`) is equivalent to h_1, 'Tank 1 Flow In Rate' (`self.tank_1_flow_in_value`) is equivalent to F_1, 'Tank 1 Flow Out Rate' (`self.tank_1_flow_out_value`) is equivalent to F_2, 'Tank 2 Area' (`self.tank_2_area_value`) is equivalent to A_2, 'Tank 2 Height' (`self.tank_2_height_value`) is equivalent to h_2, and 'Tank 2 Flow Out Rate' (`self.tank_2_flow_out_value`) is equivalent to F_0. Note how there is no included 'Tank 2 Flow In Rate' since that value is equivalent to the output flow rate of Tank 1 based on the coupling methodology described in the assignment prompt.

After the input variables are given to the simulation, the core of the actual simulation is run once the user clicks the 'Run Simulation' button after filling out valid inputs for the variables. This button triggers the `run_simulation` method found in the GUI class. The `run_simulation` method first sets all of the values of the inputs (e.g. `self.tank_2_flow_out_value`) as described in the previous paragraph so that the core of the simulation can run based on the given values. The results of each tank included a list of many arrays of size two. Inside each array, there is the height of the fluid and its corresponding second. These results are stored in two separate variables, with Tank 1's results being stored in the `tank_1_simulation_results` array. The initial rate of change of the heights of each tank are then placed in the dynamic variables

`tank_1_rate_of_change_of_height` and `tank_2_rate_of_change_of_height`, which are calculated initially and throughout the for loop as

```
tank_1_rate_of_change_of_height = (self.tank_1_flow_in_value - self.tank_1_flow_out_value) / self.tank_1_area_value
```

This was simply calculating the respective change of height of each variable $\frac{dh_x}{dt}$ from the given equations. The area of each tank was divided by the difference of the inflow and outflows to the tanks to yield the rate of change of the tank. After storing the times of 0 into each `tank_1_simulation_results` and their initial heights, a for loop is then run from 1 to the input `self.seconds_to_run + 1` to cover all the desired seconds for simulation. Inside the loop, the new height of Tank 1 is set in the dynamic variable tank_1_current_height, which is calculated with

```
tank_1_current_height = tank_1_current_height + tank_1_rate_of_change_of_height
```

Based on the calculated `tank_1_rate_of_change_of_height` above. Then, to help determine the new `current_tank_2_rate_of_change_of_height`, the current outflow of Tank 1 must be determined. Although the inflow into Tank 1 is always constant, the outflow of Tank 1 could possibly be less than the input to the tank if there is not enough height of the fluid in Tank 1 and the inflow of Tank 1 is less than the outflow. Thus, before determining the new change in height of Tank 2, the flow out of Tank 1 must first be determined as such:

```
current_tank_1_flow_out = 0
if tank_1_current_height < self.tank_1_flow_out_value:
    if self.tank_1_flow_out_value <= self.tank_1_flow_in_value:
        current_tank_1_flow_out = self.tank_1_flow_out_value
    else:
        current_tank_1_flow_out = self.tank_1_flow_in_value + tank_1_current_height
else:
    current_tank_1_flow_out = self.tank_1_flow_out_value
```

This nested if-else loop ensures that the correct outflow of Tank 1 is calculated based on the height of the fluid inside its tank and the inflow and outflow relationship.

Then, since Tank 2 is couple to Tank 1, the outflow of Tank 1 is effectively the same as the inflow to Tank 2, as both rely on the same F_2 in the original prompt's equation. The `current_tank_2_rate_of_change_of_height` is thus calculated based on the outflow of Tank 1 as such:

```
current_tank_2_rate_of_change_of_height = (current_tank_1_flow_out - self.tank_2_flow_out_value) / self.tank_2_area_value
```

This equation in the for loop thus ensures the coupling of the two tanks and maintains the coupling by relating the two tanks via the core variable connecting the two: `current_tank_1_flow_out`, or F_2.

The new height of Tank 2 is then calculated based on the new rate of change of the height of Tank 2 and the results are appended to the respective simulation results array.

After the for loop runs, the line line graphs are plotted and the bar graphs of the data are updated.

Results/Discussion: Results are discussed in part b and c of this question.

b. Describe how you would run the simulation for a case such as : A_1 =1.0, A_2=2.0, F_0=0.02, F_1=.01, F_2=.01 , and varying initial values for (h_{10}, h_{20}) such as: (0,0), (2,0), (0,1.5), (1,0), (0,1).

Using the GUI and the algorithm found in `run_simulation` described in part a, the simulations described here were evaluated and ran. More specifically, the `run_simulation` algorithm was chosen to only add to the height of the fluids in each tank if the current net flow (inflow - outflow rates) of each tank was positive, and to only subtract from the heights of the fluids of each tank if the current net flow was negative. If the net flow is 0, the heights of the tank remained the same.

The requested parameters to run the simulation have an area of 1.0 and 2.0 for the tanks respectively and an inflow to Tank 1 of 0.01 (F_1), a flow out of Tank 1 of 0.01 (F_2), and a flow out of Tank 2 of 0.02 (F_2). These input parameters are used consistently throughout the testing of part b.

Since the flow in and out of Tank 1 are the same, the height of the fluid in Tank 1 is never to change. This is ensured by the algorithm

```
tank_1_rate_of_change_of_height = (self.tank_1_flow_in_value - self.tank_1_flow_out_value) / self.tank_1_area_value
```

which, since the flow into Tank 1 and the flow out of Tank 1 are equivalent, always equates to a rate of change of height of 0 of Tank 1 given the input parameters to this problem. Thus, for all cases of initial heights of *(h_{10}, h_{20})*, the heights of the fluid of Tank 1 remains the same throughout the simulation. If, however, the inflow and outflow rates into Tank 1 differ, the height of the fluid in Tank 1 must change accordingly.

The flow out of Tank 1 can only change if the outflow rate of Tank 1 is greater than the inflow rate to the respective tank. Since that is not the case in this problem, this possibly is discarded, and, in this case, the outflow rate of Tank 1 is equivalent to the inflow rate of Tank 2, illustrating how this problem is coupled together by this flow rate.

Because of the nature of these input parameters, the height of the fluid inside Tank 2 will always decrease by a rate of (F_0 - F_2) / A_2 = (0.01 - 0.02) / 2.0 = 0.005 height units per second since the flows and areas remain constant by this problem. The slope of the height of the fluid of Tank 2, regardless of if it initially starts at 1.5 or 1, will always linearly decrease by this slope until it

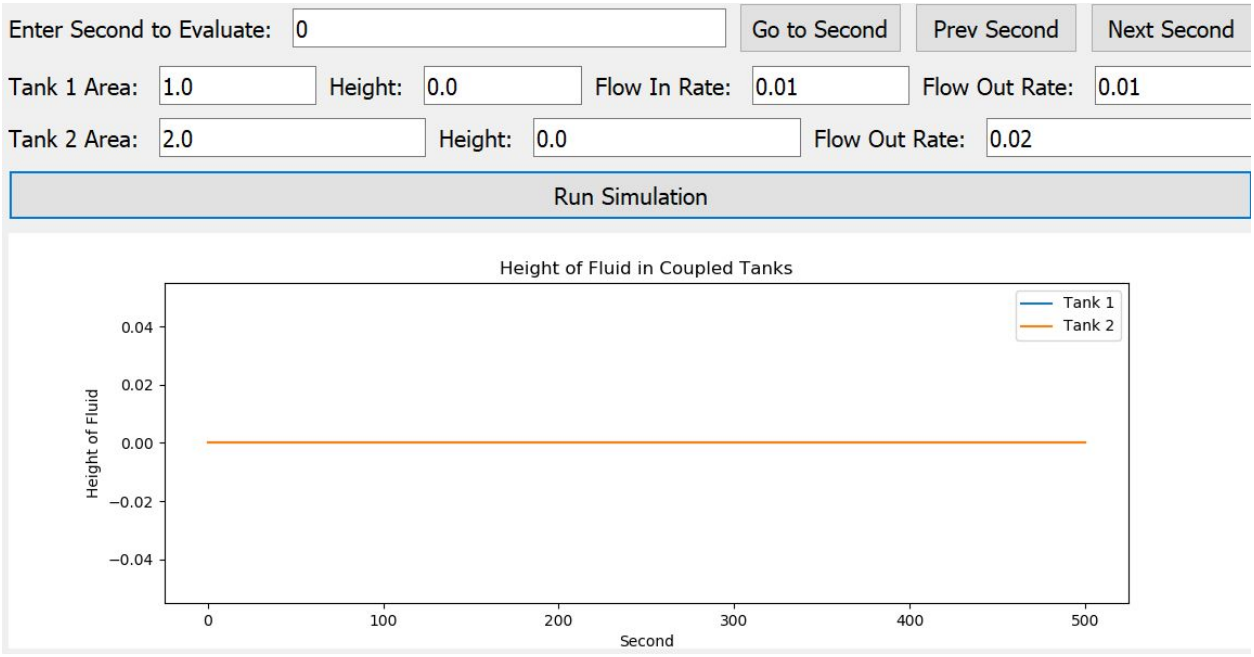
reaches 0 given these specific input parameters. While the height of the fluid in Tank 1 will always remain a horizontal slope from the given initial height of the fluid of Tank 1, the height of the fluid in Tank 2 will always decrement negatively by 0.005 height units every second until it reaches 0 given the input parameters. This can be visualized in the results in part c.

Note that this specific pattern only manifests itself given these specific input parameters of identical inflow and outflow rates to Tank 1 and a relatively higher outflow rate of Tank 2. If these parameters were to change, the results would vary more widely.

c. Analyze your results and discuss them?

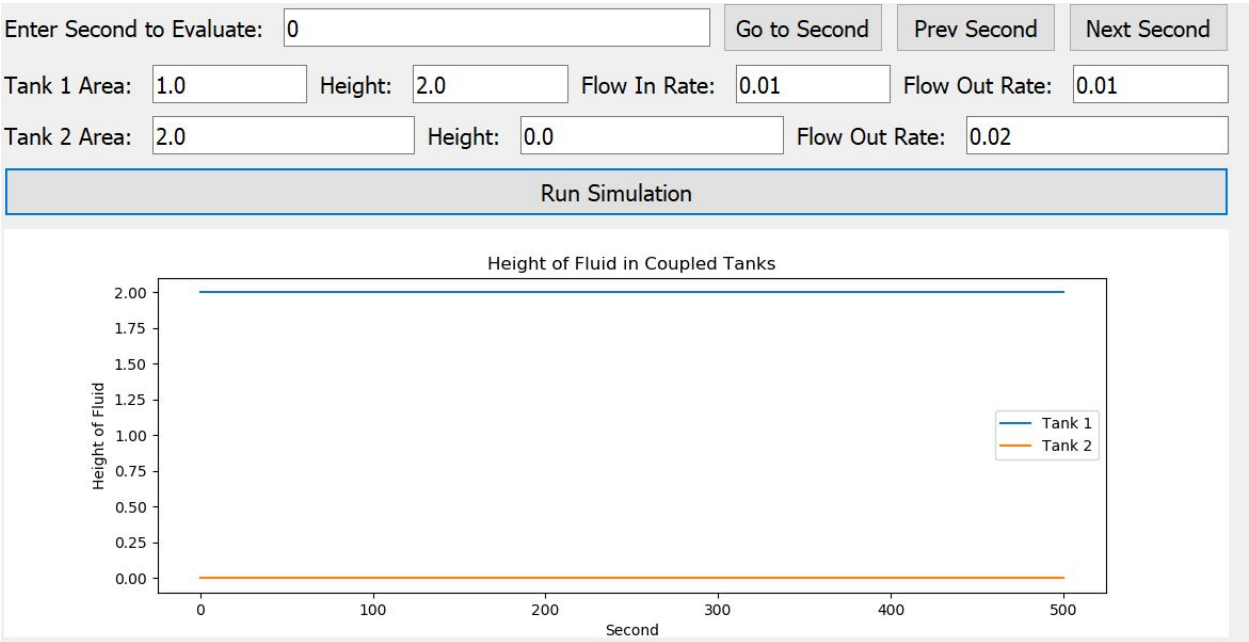
Given the analysis of part b which explains the behavior of the tanks based on the input values, the results can be visualized below:

(h_{10}, h_{20}) = (0, 0):



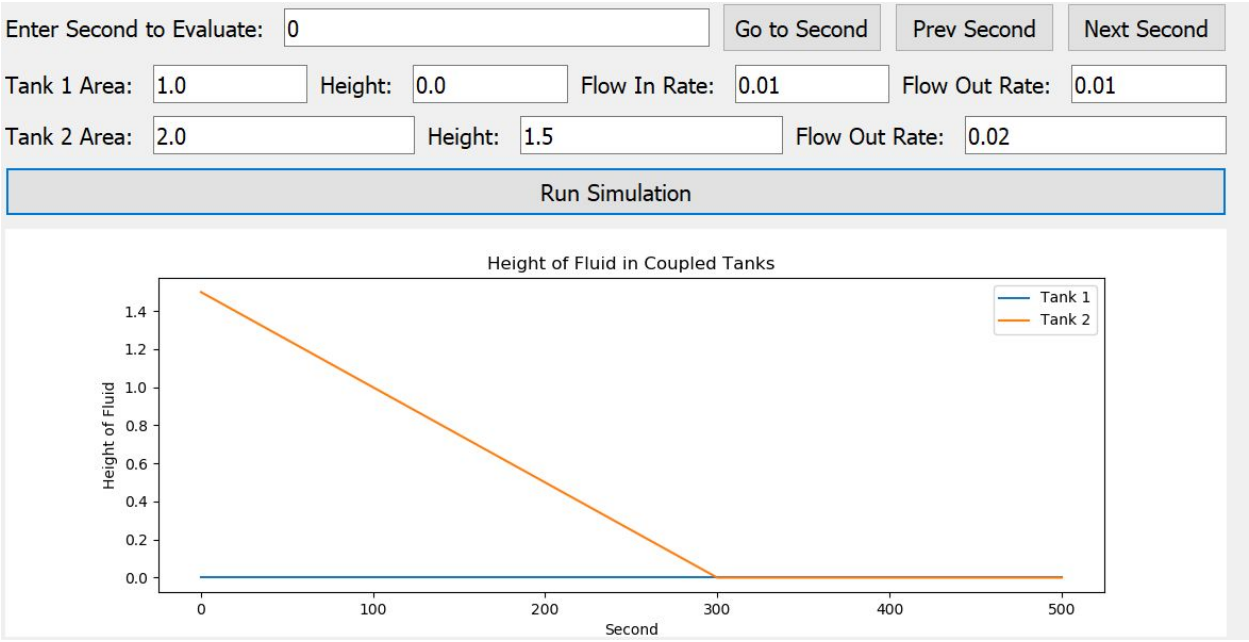
Because the initial height of Tank 1 is 0 and the net flow rate of Tank 1 is 0, the height of Tank 1 remains 0 throughout the simulation. The height of the fluid in Tank 2 remains 0 throughout the duration of the simulation as well because the initial height of Tank 2 is 0 and it has a negative net flow rate. Since Tank 2 can only lose height of the fluid and has a higher outflow than inflow, the height of Tank 2 will never increase and will remain at 0 as well throughout the simulation.

(h_{10}, h_{20}) = (2, 0):



Similar to the explanation for (h_{10}, h_{20}) = (0, 0), the height of the fluid of Tank 1 of 2.0 will remain constant throughout the duration of the simulation given its net zero flow rate. Similarly, Tank 2 remains at 0 because it has a negative net flow rate, so it cannot increase the height of the fluid in the tank from 0; it can only decrease the height of the fluid in the tank.

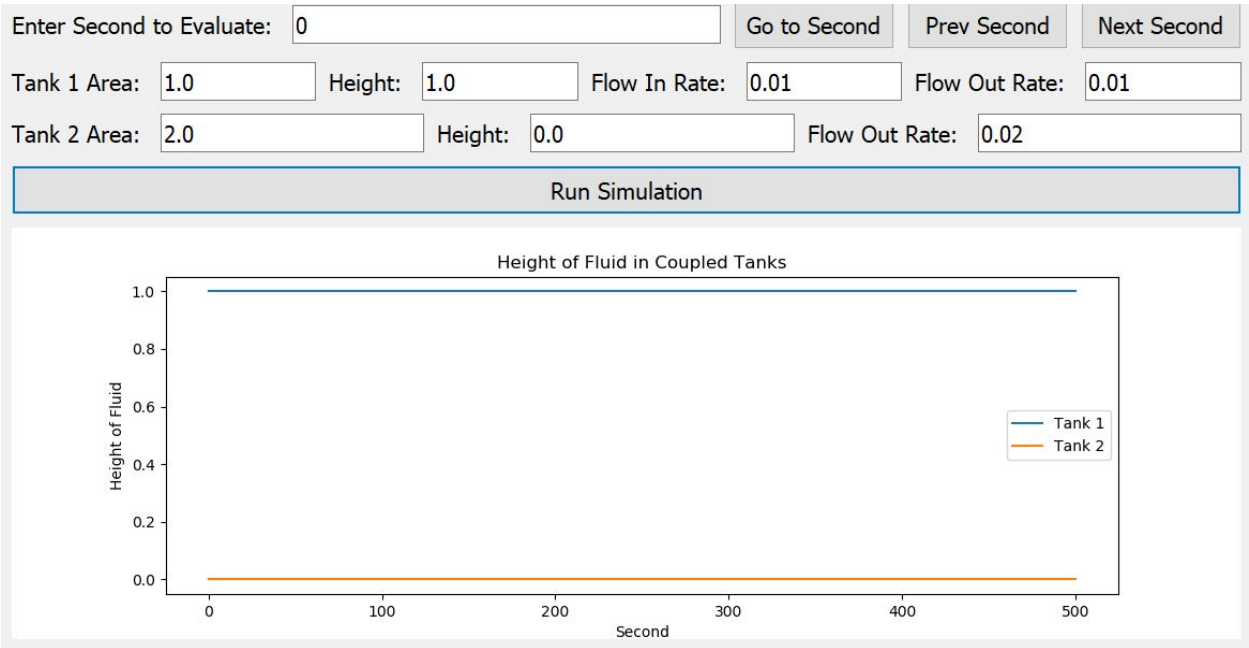
(h_{10}, h_{20}) = (0, 1.5):



Again, the height of the fluid in Tank 1 remains constant because it has a net flow rate of 0. The slope of -0.005 can be visualized in Tank 2's height of the fluid here from 0 to 300, where it eventually hits a height of 0 and cannot decrease anymore. Note how the slope of Tank 2 ends

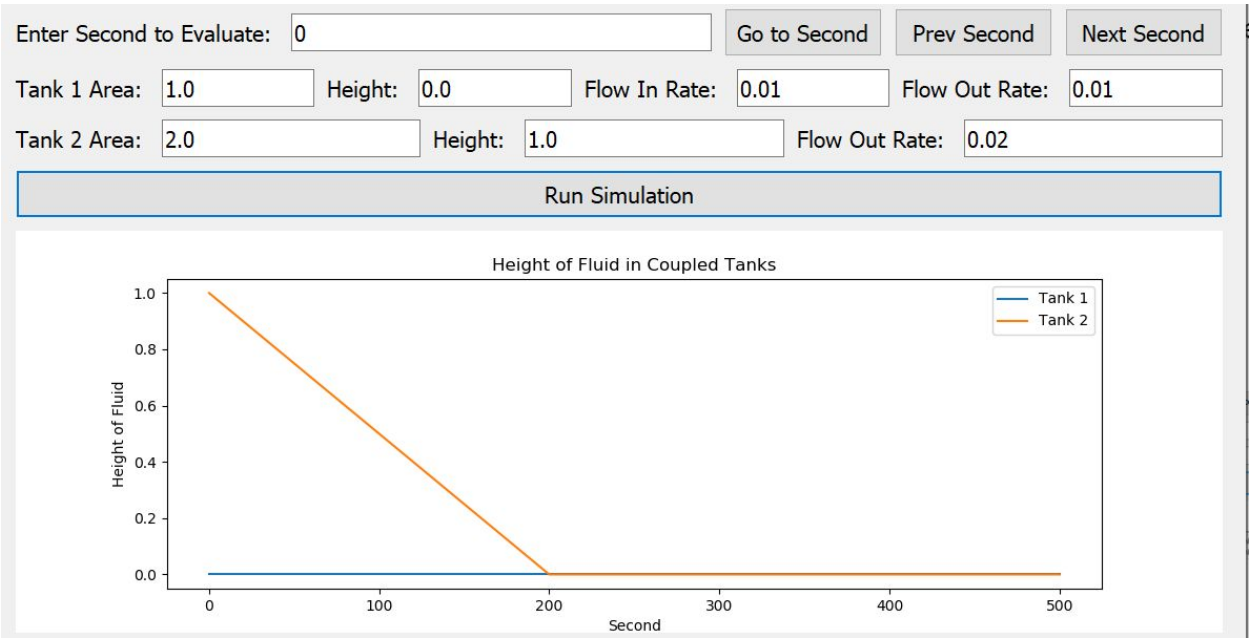
at t=300 seconds because the equation to determine the end of the slope is y = -0.005x + 1.5, and solving for x when y is 0 leads to 1.5 / 0.005 = 300 seconds.

(h_{10}, h_{20}) = (1, 0):



The results of this experiment are effectively the same as the explanation found above in the section labeled (h_{10}, h_{20}) = (2, 0).

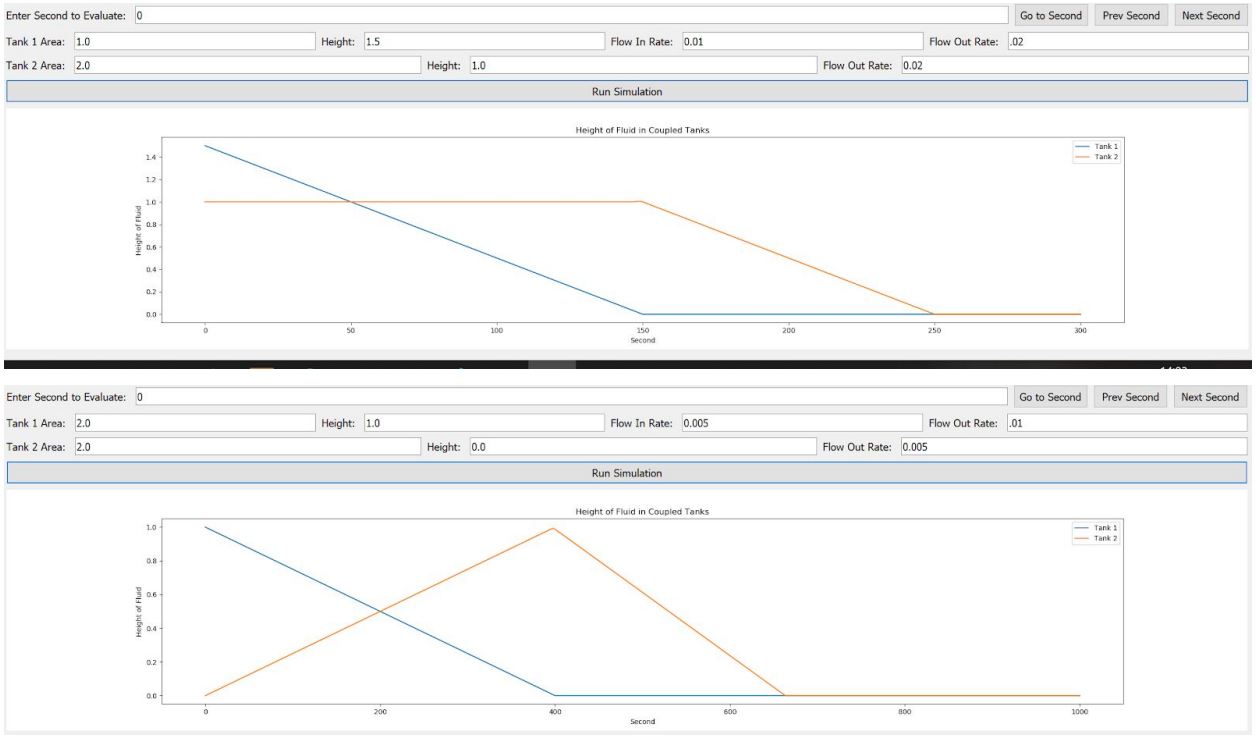
(h_{10}, h_{20}) = (0, 1):



The explanation for this experiment can be found in the section labeled (h_{10}, h_{20}) = (0, 1.5), although note how the slope of Tank 2 ends at t=200 seconds here, because the equation to

determine the end of the slope is y = -0.005x + 1.0, and solving for x when y is 0 leads to 1.0 / 0.005 = 200 seconds.

More interesting results occur when the flow rate in and out of the upper tank (tank 1) are different, as seen below:



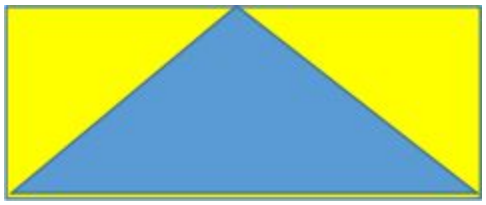
These results lead to fluctuations in the flow rates out of Tank 1 based on if enough fluid can come out of Tank 1 relative to the inflow rate of the fluid of Tank 1, making for more interesting results and analysis. The different inflow and outflow rates into Tank 1 make the coupling between Tank 2 and Tank 1 even more noticeable because the different Tank 1 flow rate influences the outflow rate of Tank 1, which can impact Tank 2 in more interesting ways.

This is especially seen in the case above, where the height of the fluid of Tank 2 initially increases from t=0 to t=400 seconds and then decreases from t=400 to t=650 seconds. This illustrates the coupled relationship of the two tanks especially at t=400 seconds, where Tank 1 runs out of the overstock of fluid as it now has a height of 0. Since Tank 1 has a height of 0 at t=400, it's outflow rate becomes equal to its inflow rate, which is half of the outflow rate. This change at t=400 seconds thus illustrates how Tank 2 also starts decreasing its height due to its coupled relation to the outflow of Tank 1.

This example illustrates the coupled relationship between the two tanks better than the extreme cases that are discussed in part b.

The code for this problem is given below in the Appendix as “Problem 4 code, *fluid_in_tanks_sim_fox.py.”*

Problem 2: *Show the pseudo-code for computing random variates that follow the triangle distribution (with probability density function f(x) defined below) using the inverse distribution function method.*
Assume there is a function rand() that returns random numbers that are uniformly distributed between 0 and 1. Show all work in deriving your solution.
f(x) = x for 0 ≤ x < 1
f(x) = 2-x for 1 ≤ x < 2
f(x) = 0 otherwise
Report and explain some sample results.

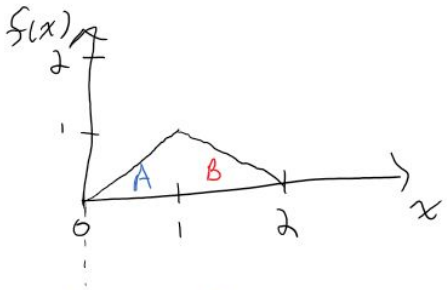


First, the cumulative distribution function (CDF) of the probability function needs to be found. Once the CDF of the given triangle distribution is found, random variates can be computed from that distribution. The inverse CDF method for computing this random variate realizes that a continuous distribution function is 1-1 mapping of the domain of the CDF onto the interval (0,1). Thus, a random uniform method such as rand() as described in the problem 2 prompt leads to a distribution $Y = F^{-1}(rand())$ with Y having the distribution F [1].

The process of finding the cumulative distribution function based on the probability function is performed below:

A random variable X has probability density function f(x) given by

$$f(x) = \begin{cases} x, & 0 \le x < 1 \\ 2 - x, & 1 \le x < 2 \\ 0, & \text{otherwise} \end{cases}$$



$$A: \int_0^x x\,dx = \frac{x^2}{2}\Big|_0^x = \frac{x^2}{2} - 0 = \frac{x^2}{2}$$

$$B: \int_1^x (2-x)\,dx + \frac{1}{2} = \left[2x - \frac{x^2}{2}\right]_1^x + \frac{1}{2} = \left(2x - \frac{x^2}{2}\right) - \left(2(1) - \frac{1}{2}\right) + \frac{1}{2}$$

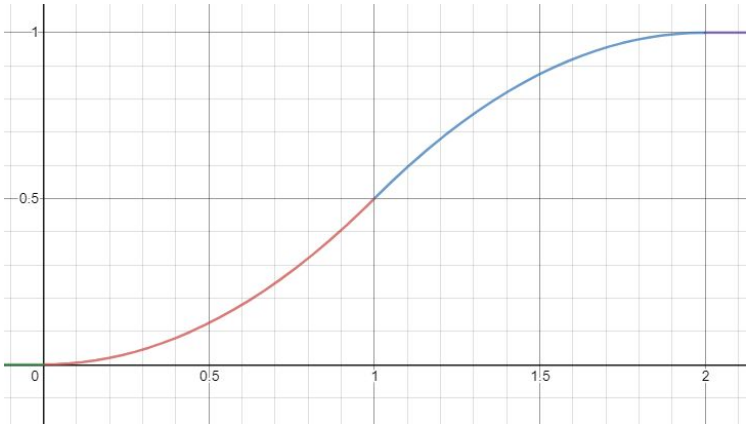
$$= 2x - \frac{x^2}{2} - \frac{3}{2} + \frac{1}{2} = 2x - \frac{x^2}{2} - 1$$

Check: Area of $\left[2x - \frac{x^2}{2} - 1\right]_{x=2}$ should be 1

$$\Rightarrow 2(2) - \frac{(2)^2}{2} - 1 = 4 - 2 - 1 = 1$$

$$\therefore \text{CDF} = F(x) = \begin{cases} 0, & x < 0 \\ \frac{x^2}{2}, & 0 \le x < 1 \\ 2x - \frac{x^2}{2} - 1, & 1 \le x \le 2 \\ 1, & x > 2 \end{cases}$$

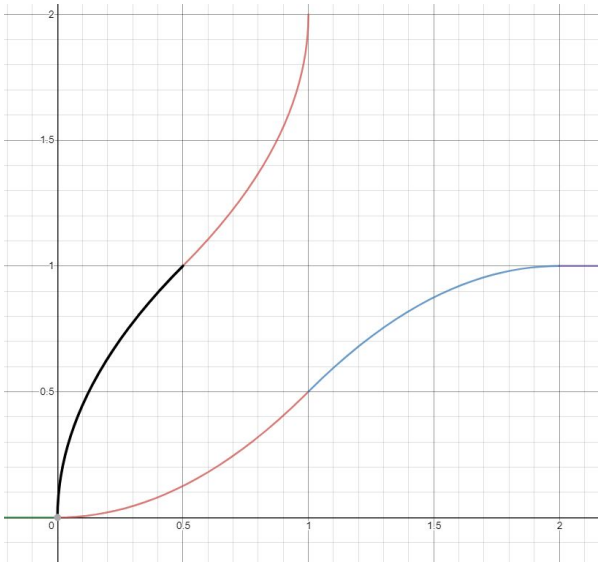
The CDF of F(x) above thus looks like



The inverse CDF of the function can thus be calculated as

$$F^{-1}(x) = \begin{cases} 0, & x < 0 \\ \sqrt{2x}, & 0 \le y < 1 \\ \frac{-4 + \sqrt{-8x + 8}}{2}, & 1 \le y \le 2 \\ 1, & x > 2 \end{cases}$$

with an inverse function appearing like so (the left curve is the inverse, the right is the orginal CDF.)



The inverse CDF(left) and the original CDF (right)

The inverse CDF calculated above can thus be used along with the given rand() function to generate random variates by plugging in the result of rand() into the inverse of the calculated CDF.

After deriving these functions, those variables can be plugged in and hardcoded into this problem for this specific triangle distribution so that the correct variates can be returned.

For example, following the graph of the inverse CDF found above, a random *U* value returned by rand() of 0.75 would return a variate of about 1.25 above based on the function above.

The pseudocode for computing the variates that follow the triangle distribution is below:

```
// This function returns the cumulative distribution function of the triangle
distribution
function get_cdf_of_triangle_dist(probability_function)
       return F(x) = {0 if x <0,
                        (x^2 / 2) if x >= 0 and if x < 1,
                        (2*x - (x^2/2) - 1) if x >= 1 and if x <= 2,
                        1 if x > 2
                         }

// This function returns the inverse of the CDF of the triangle distribution
function get_inverse_of_triangle_cdf(cdf)
       return F(x) = {sqrt(2*x) if y >= 0 and if y < 1,
                       -((-4 + sqrt(-8*x + 8)) / 2) if y >= 1 and if y <= 2
                      }
// This is the main function which returns the random variate based on the functions
// calculated by hand
function compute_random_variate(probability_function)
       // First, get CDF of the given probability function f(x)
       let probability_function = input probability function f(x)
       cdf = null
       if probability_function == triangle_dist
              cdf= get_cdf_of_triangle_dist(probability_function)

              // Second, generate a random number from interval (0,1) using given
              rand() method
              u = rand()

              // Third, find inverse of desired CDF
              inverse_of_cdf = get_inverse_of_triangle_cdf(cdf)

              // Third, compute random variable X by plugging in u to the inverse of
              the CDF
              // X now has distribution found of F_X(x) (the triangle distribution)
              X = inverse_of_cdf(u)
```

```
        // Return X
        return X
```

The pseudocode uses the functions derived by hand to ultimately use the inverse CDF of the given triangle distribution function so that random variates can be computed that follow said triangle distribution.

Some results from running `compute_random_variate` with the probability function as the triangle distribution could yield any number between 0 and 2, depending on what the random variable *U* that is returned by rand() on the interval (0,1) is. For example, if rand() were to return 0.25, the output of the `compute_random_variate` function would be 0.54772 based on the inverse CDF of the triangle distribution. Other results would be:

based on rand() = 0.05, result = 0.31622
based on rand() = 0.2, result = 0.63245
based on rand() = 0.48, result = 0.97979
based on rand() = 0.75, result = 1.29289
based on rand() = 0.95, result = 1.68377

These results are consistent with sampling from the given distribution in the prompt.

Problem 3: *In an interview, a human resource (HR) specialist asks candidates 20 questions with 5 possible multiple choice answers to each question. What is the probability of the following candidates for success given the following assumptions:*

a. HR has the last answer always correct, candidates select answers at random.
b. HR's solution is in a random location and student selects an answer at random

Note: assume the interview has only 3 questions, trace some (approx. 10) random candidates for each scenario but run a larger sample on the computer and do not respond philosophically without support.

Introduction: A script to simulate the probabilities of a candidate answering twenty multiple choice questions with five possible choices and one correct answer was simulated. The script was written in Python to compute the probabilities many times. The script is written so that it can simulate the results of the correct answer of the five multiple choices always being the last answer while the candidate answers randomly; the script can also simulate the correct answer being randomly distributed throughout the five questions while the candidate again answers randomly. The results of running these probabilistic simulations were analyzed and evaluated. Running these simulations provided intuition and feedback regarding questions about randomness and provided insights for potential students that, so long as the candidates select answers randomly, the placement of the actual correct answer doesn't appear to matter.

Approach: The main method of the Python script ran the the core simulation method of `get_answers_correct_percent`. This method took in the variables:

- `num_questions`, which determined the number of questions on each test (set to 20 by default in the prompt)
- `num_multiple_choice`, which determined the number of choices (set to 5 by default in the prompt)
- `last_answer_always_correct`, which determined if the simulation should have the final choice always be correct or if the correct answer should be randomly placed among the choices
- `iterations_to_run`, which determined the number of iterations to be run so that larger patterns of the probabilities could be revealed (e.g. the average of 50 iterations could be more telling than just running one iteration of the test which could be inherently skewed.)

`get_answers_correct_percent` stored an array of all the probabilities for running each iteration. A for loop then ran over the input number of iterations to run. The for loop then forked to either evaluate the probability of running the simulation with the last answer always having the sole correct answer or with the correct answer being randomly shuffled in the correct choices each time with a simple if loop.

If case a from the prompt is to be evaluated (i.e. `last_answer_always_correct` is true), then an ordered list of choices is created, and the first `num_multiple_choice - 1` choices are set to be a false boolean in the ordered list. The final choice is then set to be a true boolean. The true boolean is thus considered the “correct” choice in this simulation for simplicity. After assigning equal weights to every choice, a for loop then runs over the variable `num_questions`. Python’s uniform random.choices function is then used to randomly select a choice from the ordered list of choices inside the for loop. If the answer chosen happens to be True, then the variable `num_correct_choices` is incremented. After the for loop for each question is complete, the probability of selecting a correct answer is calculated by dividing `num_correct_choices` by `num_questions`.

The logic for finding the probability of selecting a correct answer for the randomly distributed correct answer among the multiple choices was very similar, except the ordered list of choices are shuffled before using pythons `random.shuffle` method, which shuffles the choices uniformly and randomly for each question. The results of running these methods several times was then observed and analyzed in the following section.

Results/Discussion:
After running the averages of both cases, with Case A being the case where the candidate always randomly chooses the answer and the correct answer always being last and with Case B being the case where the candidate also always chooses the answer randomly but the correct answer is randomly shuffled in the possible multiple choice questions, it became evident that the

overall average of both simulation turns out to be the same. Here is a sample of averaging 100 iterations for both Case A and Case B several times:

Average probability of correct answers for Case A in 100 iterations: **0.19350**
Average probability of correct answers for Case B in 100 iterations: **0.19900**
Average probability of correct answers for Case A in 100 iterations: **0.19950**
Average probability of correct answers for Case B in 100 iterations: **0.19500**
Average probability of correct answers for Case A in 100 iterations: **0.19300**
Average probability of correct answers for Case B in 100 iterations: **0.19100**
Average probability of correct answers for Case A in 100 iterations: **0.20900**
Average probability of correct answers for Case B in 100 iterations: **0.19600**
Average probability of correct answers for Case A in 100 iterations: **0.20250**
Average probability of correct answers for Case B in 100 iterations: **0.21150**
Average probability of correct answers for Case A in 100 iterations: **0.19550**
Average probability of correct answers for Case B in 100 iterations: **0.20900**
Average probability of correct answers for Case A in 100 iterations: **0.20700**
Average probability of correct answers for Case B in 100 iterations: **0.19800**
Average probability of correct answers for Case A in 100 iterations: **0.19600**
Average probability of correct answers for Case B in 100 iterations: **0.21800**

Running an average *on top* of those averages to make doubly sure there is no clear evidence of one case having a better probability per this simulation yields the the averages of all those averages of:

Average probability of averages of Case A iterations: **0.19968**
Average probability of averages of Case B iterations: **0.20022**
The cases where the averages of Case A was higher was 46 out of 100 overall averages of 100 iterations.

Based on this one case, it could be said that Case B is times more likely to occur than Case A. However, if this simulation were to continue to be run, the trend does not continue.

Running the same simulation 10 more times, the results are:
1. Case A average of averages higher: **50**, Case B average of averages higher: **50**
2. Case A average of averages higher: **60**, Case B average of averages higher: **40**
3. Case A average of averages higher: **45**, Case B average of averages higher: **55**
4. Case A average of averages higher: **54**, Case B average of averages higher: **46**
5. Case A average of averages higher: **51**, Case B average of averages higher: **49**
6. Case A average of averages higher: **48**, Case B average of averages higher: **52**
7. Case A average of averages higher: **52**, Case B average of averages higher: **48**
8. Case A average of averages higher: **50**, Case B average of averages higher: **50**
9. Case A average of averages higher: **48**, Case B average of averages higher: **52**
10. Case A average of averages higher: **43**, Case B average of averages higher: **57**

While it is evident that sometimes Case A appears with a higher probability than Case B during some runs, the opposite appears true just as frequently.

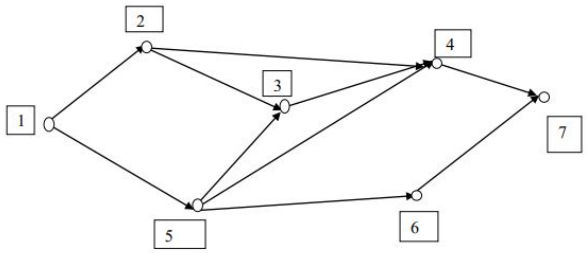
It can thus be determined that, so long as the candidate is randomly and uniformly choosing a multiple choice answer, there is no effect on the probability of selecting a correct answer regardless of if the correct answer to the questions are randomly shuffled in the choices or if they are always the very last choice.

This was surprising to me, at least, for intuitively I assumed more that randomly choosing a random answer from a randomly shuffled list would somehow have more randomness than simply randomly selecting a choice amongst a constantly ordered list of answers where the correct answer was always at the end. However, I now realize that multiplying randomness does not quite work this way, and that, since the candidate always randomly chooses an answer, it does not matter which answer is correct, for there will *always* be a 20% chance of the candidate selecting a correct answer regardless of the consistency of the position of the correct answer. This intuition makes sense: there is always a 20% chance of the correct answer being randomly chosen; the position of the correct answer has no effect of the chances of choosing the correct answer. That is why the averages of the probabilities of choosing the correct answers always nears .20 for both cases.

The code for this problem is given below in the Appendix as "Problem 3 code, *hr_question_prob_fox.py.*"

Problem 4: A software process is represented as a network with some process time between nodes having Uniform distribution (U) and others with Deterministic values.

a. Analyze the performance of the system
b. After adequate samples, show how you quantify the criticality of each path.
c. Briefly explain your redesign perspective of such a system
(1, 2): U (4,6)
(1, 5): 6
(2, 3): 6
(2, 4): U (6,8)
(3, 4): U (4,8)
(4, 7): 4
(5, 3): 8
(5, 4): 11
(5, 6): U (8,10)
(6, 7): U (9, 10)



Answer here PERT, Critical Path Method (CPM), Petri,

Introduction: The length of each given path was evaluated and analyzed after assigning both deterministic and uniformly random values to the lengths of each connecting path as given in the Homework 3 prompt using the Python programming language. The performance of the system was then evaluated and adequate samples were taken to help determine the criticality of each path, and an explanation of how such a system could be redesigned was included, especially concerning which processes in the software should be optimized so that the criticality of the path is reduced. The similarities to the Critical Path Methods (CPM) and program evaluation and review technique (PERT) are analyzed and compared to the techniques used in the code written in this problem.

Approach: A `Node` class was created to hold information about all the nodes and their respective connections and connection lengths, initialized as such:

```
class Node:
    def __init__(self, name):
        self.node_connections = []
        self.name = name
```

The `node_connections` list contained connections that were added using the `add_connections` method, which appended lists of size two containing the connecting node and the length of the connection to that node. It worked like so:

```
def add_connection(self, connecting_node, length):
        self.node_connections.append([connecting_node, length])
```

The name of the function that was added to each `Node` class was simply used for representing the name of each `Node` in the interpreter and debugging stages and for evaluating the results of the overall path lengths. In addition to the “setter” function `add_connection`, there were two getter functions that allowed for retrieving all the connections that each node has (which was later used for recursively iterating through the paths) and one for retrieving the length of the connection to its respective connecting node.

Each `Node` and its connections were initialized and their length connections were set following this strategy:

```
## Node 1 ##
node_1.add_connection(node_2, random.uniform(4, 6))  # (1, 2): U (4,6)
node_1.add_connection(node_5, 6)  # (1, 5): 6
```

Since the project prompt specified that the connection between Node 1 and Node 5 was a deterministic length of always 6, the connection length parameter of `add_connection` was simply set to 6. Since the length of the connection between Node 1 and Node 2 was to be uniformly distributed between 4 and 6, the value of the length for that connection was uniformly set using Python's `random.uniform` method, which sets the value to be uniformly between 4 and 6 each time. This approach was then used for each Node and their connections, depending on if they were to be deterministic of uniform connection lengths or not.

The function `analyze_performance` was then used to analyze the performance of each node. This function worked by recursively iterating through every possible path to make sure the critical path could be found. Every possible list of the nodes and their connections were stored in the global array `list_results`. The function `get_length_of_connections` was then used to get the length of the connections of every node and their connections. This allowed for evaluating the overall path length of all connection lists.

A simple if loop was then run on list_results and the maximum path length paths and minimum path length paths were determined for all the results. This was run several times to help determine which lengths and nodes were longest and which could be optimized for future improvements on the critical path.

Results/Discussion: The result of each run appeared like:

> [Node 1, Node 2, Node 3, Node 4, Node 7]: **21.511046520343598**
> [Node 1, Node 2, Node 4, Node 7]: **14.889294635562656**
> [Node 1, Node 5, Node 3, Node 4, Node 7]: **24.791496723805228**
> [Node 1, Node 5, Node 4, Node 7]: **21**
> [Node 1, Node 5, Node 6, Node 7]: **23.634236463912863**
> Max Value list: [Node 1, Node 5, Node 3, Node 4, Node 7], Length of **24.791496723805228**
> Min Value list: [Node 1, Node 2, Node 4, Node 7], Length of **14.889294635562656**

The program thus determined that there were 5 possible paths, which are all listed above in brackets (e.g. [Node 1, Node 2, Node 3, Node 4, Node 7] symbolizes the path from Node 1 to Node 2 to Node 3 to Node 4 and ending at Node 7, and so on.) The maximum and minimum length of the paths were determined in each list as well. Below are different results of running the same algorithm, illustrating how the uniformly distributed lengths vary in length and can lead to different critical paths depending on how large the uniformly distributed connections happen to be.

> [Node 1, Node 2, Node 3, Node 4, Node 7]: **21.58993583390757**
> [Node 1, Node 2, Node 4, Node 7]: **16.13200276894913**
> [Node 1, Node 5, Node 3, Node 4, Node 7]: **24.405316854422246**

[Node 1, Node 5, Node 4, Node 7]: **21**
[Node 1, Node 5, Node 6, Node 7]: **24.610395391729533**
Max Value list: [Node 1, Node 5, Node 6, Node 7], Length of **24.610395391729533**
Min Value list: [Node 1, Node 2, Node 4, Node 7], Length of **16.13200276894913**

It can also be seen here how the longest path changed from [Node 1, Node 5, Node 3, Node 4, Node 7] to [Node 1, Node 5, Node 6, Node 7] because of the randomness involved in the simulation. It can also be noted how the path [Node 1, Node 5, Node 4, Node 7] *only* has deterministic values, and so will always be 21 regardless in this simulation.

Utilizing the law of large numbers, all of the paths were then run 500 times and the average value of their paths were taken to help establish a useful average path with 500 samples to help establish more conceretely which paths were longest. The results were:

Average values for each path over 5000 samples:
Average path_12347_values: 22.118791197211838
Average path_1247_values: 16.048479207694072
Average path_15347_values: 25.511074767592824
Average path_1547_values: 21.0
Average path_1567_values: 23.5291063659135

The order of the longest paths, with the backing of 5000 samples, is *still* subject to minimal reordering due to the nature of the inherent randomness involved in the simulation of the lengths. However, after running this simulation and averaging over 5000 samples ten times, the order of the paths remained the same. The law of large numbers thus helps to support the overarching trend of these paths.

After running this analysis, it can be seen that, of all the paths, the [Node 1, Node 5, Node 3, Node 4, Node 7] path remains the critical path most consistently amongst the five paths. The processes involved in that path would most definitely have to be optimized if the software process were to run faster. The worst offender in this path is the deterministic length of 8 of the connection between Nodes 5 and 3. Since this process time is deterministic, the designer of this system should most definitely be able to optimize the time required to run from Node 5 to Node 3 down from the time-consuming 8 time units it currently takes. *If this connection could be optimized, the overall software process could possibly run much faster.*

If that connection were to be fixed, the overall connection time of [Node 1, Node 5, Node 3, Node 4, Node 7] could be reduced to be below that of the [Node 1, Node 5, Node 6, Node 7] path. The uniform connections of (5, 6): U(8, 10) and (6, 7): U(9, 10) are the worst offenders in the entire path lengths and would thus need to be fixed to improve the design of the software process even more. redesigning this process through optimization of these three connections would greatly improve the runtime speed and efficiency of the software process.

To be sure, if a complete redesign that could simply not have 5 nodes relying on each other like is found in the [Node 1, Node 5, Node 3, Node 4, Node 7] path could be created, the program would be much more efficient as well since so many Nodes would not have to be dependent on the others to complete. This would also lower the critical length of the system. This could be accomplished by potentially removing the connection between Node 5 and Node 3 and passing the necessary information directly from Node 5 to Node 4. This would require a hands-on look of the software process to see if such a design were possible, of course.

a. Analyze the performance of the system

As seen above, the measures of the longest path, i.e. the critical path, and the shortest path were the main performance measures for evaluate the overall performance of the system.

b. After adequate samples, show how you quantify the criticality of each path.
As mentioned above, the criticality of each path is quantified by first recursively finding every single possible path. Every path is then assigned its values -either uniformly or deterministically- and then the length of all the connections is found. This process was repeated and sampled 5000 times several times to help determine the average criticality of each path to help avoid random bias in the pseudorandomness.

c. Briefly explain your redesign perspective of such a system

An explanation of how the software process could be optimized is given in the text above before part a. In summary: the nodes inside the most critical path of [Node 1, Node 5, Node 3, Node 4, Node 7] would need to be optimized, especially the most time-consuming node connection from Node 5 to Node 3. If this part of the process were optimized, in effect the logjam that is preventing the entire path from completing more quickly would be removed. This process in discovering and attempting to optimize the longest nodes is no different than those followed in project management using the CPM and PERT methods together.

The code for this problem is given below in the Appendix as “Problem 4 code, *critical_path_simulation.py*.”

Appendix

Problem 1 code, *fluid_in_tanks_sim_fox.py*:

```
# Aaron Fox
# CECS 622-01
# Dr. Elmaghraby
# Assignment 3 Problem 1
# The "coupled tank" tank problem has a tank with two differential equations
# as follows:
# A_1 * (dh_1/dt) = F_1 - F_2
# A_2 * (dh_2/dt) = F_2 - F_0
```

```
import sys

from PyQt5.QtWidgets import QDialog, QApplication, QPushButton, QVBoxLayout, QLineEdit, QLabel, QHBoxLayout  # for GUI
from PyQt5.QtCore import Qt  # To enable maximizing of the GUI

# Use Matplotlib for the embedded graphing display of the dots inside the circle/square
from matplotlib.backends.backend_qt5agg import FigureCanvasQTAgg as FigureCanvas
from matplotlib.backends.backend_qt5agg import NavigationToolbar2QT as NavigationToolbar
import matplotlib.pyplot as plt

# For basic math like absolute value
import math


# The tank class
class tank:
    def __init__(self, height, area, flow_in, flow_out):
        self.height = height
        self.area = area
        self.flow_in = flow_in
        self.flow_out = flow_out

    def print(self):
        print("Height == " + str(self.height))
        print("Area == " + str(self.area))
        print("Flow in == " + str(self.flow_in))
        print("Flow out == " + str(self.flow_out))


###### GUI for visualizing the tank values ######
# All of the gui is based out of this Window class using Python PyQt5 GUI framework
class Window(QDialog):
    def __init__(self, parent=None):
        super(Window, self).__init__(parent)
        # Allow for easy maximizing of the GUI
        self.setWindowFlag(Qt.WindowMaximizeButtonHint, True)
        self.setWindowTitle("Coupled Tank Simulation")

        # Figure instance for plotting
        self.figure = plt.figure()

        # self.canvas is the widget that displays the figure,
        # taking the figure instance as a parameter to __init__
        self.canvas = FigureCanvas(self.figure)

        # Adds toolbar to top of GUI for saving pictures, zooming in graph, etc.
        self.toolbar = NavigationToolbar(self.canvas, self)

        # For keeping track of day evaluation of bar charts
        self.current_second = 0

        # For making sure they don't try to evaluate without having run sim
        self.has_run_sim = False

        # set the layout (vertical)
        layout = QVBoxLayout()
        layout.addWidget(self.toolbar)
```

```
layout.addWidget(self.canvas)

self.go_to_second_label = QLabel()
self.go_to_second_label.setText("Enter Second to Evaluate: ")
self.go_to_second_value = QLineEdit()
self.go_to_second_value.setText("0")
self.go_to_second_button = QPushButton("Go to Second")
self.go_to_second_button.clicked.connect(self.go_to_second_button_clicked)

self.next_second_button = QPushButton("Next Second")
self.next_second_button.clicked.connect(self.next_second_clicked)

self.prev_second_button = QPushButton("Prev Second")
self.prev_second_button.clicked.connect(self.prev_second_clicked)

horizontal_layout1 = QHBoxLayout()
horizontal_layout1.addWidget(self.go_to_second_label)
horizontal_layout1.addWidget(self.go_to_second_value)
horizontal_layout1.addWidget(self.go_to_second_button)
horizontal_layout1.addWidget(self.prev_second_button)
horizontal_layout1.addWidget(self.next_second_button)
layout.addLayout(horizontal_layout1)

# Second horizontal layout: All input info for Tank 1
self.tank_1_area_label = QLabel()
self.tank_1_area_label.setText("Tank 1 Area: ")
self.tank_1_area = QLineEdit()
self.tank_1_area.setText("1.0")

self.tank_1_height_label = QLabel()
self.tank_1_height_label.setText("Height: ")
self.tank_1_height = QLineEdit()
self.tank_1_height.setText("1.5")

self.tank_1_flow_in_label = QLabel()
self.tank_1_flow_in_label.setText("Flow In Rate: ")
self.tank_1_flow_in = QLineEdit()
self.tank_1_flow_in.setText("0.01")

self.tank_1_flow_out_label = QLabel()
self.tank_1_flow_out_label.setText("Flow Out Rate: ")
self.tank_1_flow_out = QLineEdit()
self.tank_1_flow_out.setText("0.01")

horizontal_layout2 = QHBoxLayout()
horizontal_layout2.addWidget(self.tank_1_area_label)
horizontal_layout2.addWidget(self.tank_1_area)
horizontal_layout2.addWidget(self.tank_1_height_label)
horizontal_layout2.addWidget(self.tank_1_height)
horizontal_layout2.addWidget(self.tank_1_flow_in_label)
horizontal_layout2.addWidget(self.tank_1_flow_in)
horizontal_layout2.addWidget(self.tank_1_flow_out_label)
horizontal_layout2.addWidget(self.tank_1_flow_out)

layout.addLayout(horizontal_layout2)

# third horizontal layout: All input info for Tank 1
```

```
        self.tank_2_area_label = QLabel()
        self.tank_2_area_label.setText("Tank 2 Area: ")
        self.tank_2_area = QLineEdit()
        self.tank_2_area.setText("2.0")

        self.tank_2_height_label = QLabel()
        self.tank_2_height_label.setText("Height: ")
        self.tank_2_height = QLineEdit()
        self.tank_2_height.setText("1.0")

        self.tank_2_flow_out_label = QLabel()
        self.tank_2_flow_out_label.setText("Flow Out Rate: ")
        self.tank_2_flow_out = QLineEdit()
        self.tank_2_flow_out.setText("0.02")

        horizontal_layout3 = QHBoxLayout()
        horizontal_layout3.addWidget(self.tank_2_area_label)
        horizontal_layout3.addWidget(self.tank_2_area)
        horizontal_layout3.addWidget(self.tank_2_height_label)
        horizontal_layout3.addWidget(self.tank_2_height)
        horizontal_layout3.addWidget(self.tank_2_flow_out_label)
        horizontal_layout3.addWidget(self.tank_2_flow_out)

        layout.addLayout(horizontal_layout3)

        # Fourth horizontal layout is to run the simulation!
        # Button connected to generation of points on figure to visually
        # display the uniform random distribution of points inside/outside the circle
        self.run_sim_button = QPushButton('Run Simulation')
        self.run_sim_button.clicked.connect(self.run_simulation)
        horizontal_layout4 = QHBoxLayout()
        horizontal_layout4.addWidget(self.run_sim_button)

        layout.addLayout(horizontal_layout4)

        # Add parent layout as layout
        self.setLayout(layout)

        # Create initial empty bar plot for aesthetic purposes
        self.initial_plot()

        # Second Figure instance for line plot
        self.figure2 = plt.figure()

        # self.canvas is the widget that displays the figure,
        # taking the figure instance as a parameter to __init__
        self.canvas2 = FigureCanvas(self.figure2)
        layout.addWidget(self.canvas2)
        self.lineplot_init()

    # Plot previous second of graph
    def prev_second_clicked(self):
        if not self.has_run_sim:
            print("Please run at least one simulation first")
            return
        if self.current_second <= 0:
            print("Cannot go before 0 seconds in this simulation")
```

```
            return
        self.current_second = int(self.go_to_second_value.text())
        self.current_second = self.current_second - 1
        self.go_to_second_value.setText(str(self.current_second))
        self.go_to_second_button_clicked()

    # Plot next second of graph
    def next_second_clicked(self):
        if not self.has_run_sim:
            print("Please run at least one simulation first")
            return
        if self.current_second >= self.seconds_to_run:
            print("Cannot go after " + self.seconds_to_run + " seconds in this simulation.")
            print("Please adjust self.seconds_to_run to raise the limit if needed")
            return
        self.current_second = int(self.go_to_second_value.text())
        self.current_second = self.current_second + 1
        self.go_to_second_value.setText(str(self.current_second))
        self.go_to_second_button_clicked()

    # Go to second as specified by user
    def go_to_second_button_clicked(self):
        if not self.has_run_sim:
            print("Please run at least one simulation first")
            return
        if not self.go_to_second_value.text().isnumeric():
            print("Please enter an integer second value")
            return
        if int(self.go_to_second_value.text()) > self.seconds_to_run:
            print("Please enter an integer value less than or equal to " + str(self.seconds_to_run))
            return

        self.current_second = int(self.go_to_second_value.text())

        self.figure.clear()

        # create an axis
        ax = self.figure.add_subplot(1, 1, 1)

        tank_1_val = self.tank_1_simulation_results[self.current_second][1]
        tank_2_val = self.tank_2_simulation_results[self.current_second][1]
        labels = ['Tank 1', 'Tank 2']
        label_locs = [1, 2]
        width = .40
        ax.bar(label_locs, [tank_1_val,
                            tank_2_val], width, label='Tanks')

        # Set Graph labels, title, x-axis
        ax.set_ylabel('Height of Fluid')
        ax.set_title('Height of Fluid in Coupled Tanks')
        ax.set_xticks(label_locs)
        ax.set_xticklabels(labels)

        # refresh canvas
        self.canvas.draw()

    # Initial plot to make the GUI look initially pretty for aesthetic purposes
```

```
def initial_plot(self):
    self.figure.clear()

    # create an axis
    ax = self.figure.add_subplot(1, 1, 1)
    # plt.gca().set_aspect('equal', adjustable='box')

    labels = ['Tank 1', 'Tank 2']
    label_locs = [1, 2]
    width = .40
    ax.bar(label_locs, [40,
                        30], width, label='Tanks')

    # Set Graph labels, title, x-axis
    ax.set_ylabel('Height of Fluid')
    ax.set_title('Height of Fluid in Coupled Tanks')
    ax.set_xticks(label_locs)
    ax.set_xticklabels(labels)

    # refresh canvas
    self.canvas.draw()

# Initial Line plot to make GUI look initially pretty for aesthetic purpises
def lineplot_init(self):
    self.figure2.clear()
    # create an axis
    ax = self.figure2.add_subplot(1, 1, 1)

    # Set Graph labels, title, x-axis
    ax.set_ylabel('Inventory Available')
    ax.set_xlabel('Day')
    ax.set_title('Inventory for Store')
    self.canvas2.draw()

# Plot the line graph illustrating changes in each inventory
def plot_line_graph(self):
    self.figure2.clear()
    # create an axis
    ax = self.figure2.add_subplot(1, 1, 1)

    # Set Graph labels, title, x-axis
    ax.set_ylabel('Height of Fluid')
    ax.set_xlabel('Second')
    ax.set_title('Height of Fluid in Coupled Tanks')

    tank_1_x = []
    tank_1_y = []
    tank_2_x = []
    tank_2_y = []
    for i in range(len(self.tank_1_simulation_results)):
        tank_1_x.append(self.tank_1_simulation_results[i][0])
        tank_1_y.append(self.tank_1_simulation_results[i][1])
        tank_2_x.append(self.tank_2_simulation_results[i][0])
        tank_2_y.append(self.tank_2_simulation_results[i][1])

    ax.plot(tank_1_x, tank_1_y, label='Tank 1')
    ax.plot(tank_2_x, tank_2_y, label='Tank 2')
```

```
        ax.legend()

        # refresh canvas
        self.canvas2.draw()

    # Makes sure a value is float
    def is_float(self, string):
        try:
            float(string)
            return True
        except ValueError:
            return False
    def run_simulation(self):
        print("Running simulation")

        # # Upper tank of the two coupled tanks
        # upper_tank = tank(height=2, area=10, flow_in=1, flow_out=2)
        # # Lower tank of the two coupled tanks
        # lower_tank = tank(height=3, area=40, flow_in=1, flow_out=2)

        # Get input numbers from input boxes to run sim
        if (self.is_float(self.tank_1_area.text()) and self.is_float(self.tank_1_area.text())
            and self.is_float(self.tank_1_height.text()) and self.is_float(self.tank_1_flow_in.text())
              and self.is_float(self.tank_1_flow_out.text()) and self.is_float(self.tank_2_area.text())
              and self.is_float(self.tank_2_height.text()) and self.is_float(self.tank_2_flow_out.text())):
            # Inside if loop, set all respective values
            self.tank_1_area_value = float(self.tank_1_area.text())
            self.tank_1_height_value = float(self.tank_1_height.text())
            self.tank_1_flow_in_value = float(self.tank_1_flow_in.text())
            self.tank_1_flow_out_value = float(self.tank_1_flow_out.text())
            self.tank_2_area_value = float(self.tank_2_area.text())
            self.tank_2_height_value = float(self.tank_2_height.text())
            self.tank_2_flow_out_value = float(self.tank_2_flow_out.text())
            self.has_run_sim = True
        else:
            print("Invalid input variables. Please enter float values for all inputs.")
            return

        self.seconds_to_run = 500

        self.tank_1_simulation_results = []
        self.tank_2_simulation_results = []

        # Since flow rates and areas are assumed constant, we only need to calculate rate of change of heights
        # (Although these flow rates can change based on the height of the liquid and the input flows to each tank)
        tank_1_rate_of_change_of_height = (self.tank_1_flow_in_value - self.tank_1_flow_out_value) / self.tank_1_area_value
        tank_2_rate_of_change_of_height = (self.tank_1_flow_out_value - self.tank_2_flow_out_value) / self.tank_2_area_value

        tank_1_current_height = self.tank_1_height_value
        tank_2_current_height = self.tank_2_height_value
        self.tank_1_simulation_results.append([0, tank_1_current_height])
        self.tank_2_simulation_results.append([0, tank_2_current_height])
        current_tank_2_rate_of_change_of_height = 0

        # Core of sim runs here
        for i in range(1, self.seconds_to_run + 1):
            # Since the rate of change of tank 1 is always constant bc its flow is effectively
```

```
            # decoupled from the flow of tank 2, just add the current rate of change of height per second of tank
            # to the current height of tank 1
            tank_1_current_height = tank_1_current_height + tank_1_rate_of_change_of_height

            # Make sure tank_1_rate_of_change_of_height is correct based on tank 1's height also
            # solve problem where, although tank 1's current height doesn't change if the flow coming in and
            # the flow coming out are the same, the flow is accounted for in the output
            # This means the flow out of tank 1 must at least be equal to the flow in to tank 1 if the
            # input flow into the tank is less than or greater than the output flow in to the tank
            # BUT, since Tank 2's flow is couple with Tank 1, we must make sure that the current rate of
            # change of tank 2 is correct
            # Basically, tank 1's rate of flow out can be only be at the given input maximum if and only if
            # the current height of tank 1 is at least the given input rate of flow out of tank 1

            current_tank_1_flow_out = 0
            if tank_1_current_height < self.tank_1_flow_out_value:
                # current_tank_1_flow_out = tank_1_current_height
                # If output of tank 1 flow is less than or equal to input of tank 1 flow, then always use max outflow
                if self.tank_1_flow_out_value <= self.tank_1_flow_in_value:
                    current_tank_1_flow_out = self.tank_1_flow_out_value
                else:  # If outflow of tank 1 is greater than inflow of tank 1, then set outflow to be equal to inflow level plus whatever height
is left in tank
                    current_tank_1_flow_out = self.tank_1_flow_in_value + tank_1_current_height
            else:
                current_tank_1_flow_out = self.tank_1_flow_out_value


            current_tank_2_rate_of_change_of_height = (current_tank_1_flow_out - self.tank_2_flow_out_value) /
self.tank_2_area_value

            if tank_1_current_height < 0:
                tank_1_current_height = 0.0

            tank_2_current_height = tank_2_current_height + current_tank_2_rate_of_change_of_height
#tank_2_rate_of_change_of_height

            if tank_2_current_height < 0:
                tank_2_current_height = 0.0

            self.tank_1_simulation_results.append([i, tank_1_current_height])
            self.tank_2_simulation_results.append([i, tank_2_current_height])

        self.plot_line_graph()

###### END GUI #####

if __name__ == "__main__":
    print("Running Assignment 3 Problem 1")
    app = QApplication(sys.argv)

    main = Window()
    main.show()

    sys.exit(app.exec_())
```

Problem 2 code, *hr_question_prob_fox.py*:

```python
# Aaron Fox
# CECS 622-01
# Dr. Elmaghraby
# Assignment 3 Problem 3

import random # For randomly selecting choices from answers and randomly selecting where answer is

# get_answers_correct_percent gets the percentage of questions the candidate answered correctly
# INPUT:
# num_questions (int): determines the number of questions on each test (set to 20 by default in the prompt)
# num_multiple_choice (int): determines the number of choices(set to 5 by default in the prompt)
# last_answer_always_correct (bool): determines if the simulation should have the final choice always be correct or if the correct answer should be randomly placed among the choices
# iterations_to_run (int): determines the number of iterations to be run so that larger patterns of the probabilities could be revealed(e.g. the average of 50 iterations could be more telling than just running one iteration of the test which could be inherently skewed.)
# OUTPUT:
# answer_correct_percentages (list): list of the probabilities of choosing a correct answer (e.g. [0.21, 0.19])
def get_answers_correct_percent(num_questions, num_multiple_choice, last_answer_always_correct, iterations_to_run):
    num_correct_choices = 0

    answer_correct_percentages = []
    for i in range(iterations_to_run):
        num_correct_choices = 0
        if last_answer_always_correct:
            # Create ordered list of choices
            choices = []
            # First append all incorrect choices (False) to be first num_multiple_choice - 1 answer
            for i in range(num_multiple_choice - 1):
                choices.append(False)

            # Append correct answer (True) to be last choice
            choices.append(True)

            # Weights should be 1 for every choice for equal weighting
            weights = [1] * len(choices)

            # Run simulation by randomly choosing from one of the choices equally
            for i in range(num_questions):
                candidate_answer = (random.choices(choices, weights))[0]
                if candidate_answer == True:
                    num_correct_choices = num_correct_choices + 1

            answer_correct_percentages.append(num_correct_choices / num_questions)
        else:   # Otherwise, place correct answer randomly in list by shuffling around choices
            # Create ordered list of choices
            choices = []
            # First append all incorrect choices (False) to be first num_multiple_choice - 1 answer
            for i in range(num_multiple_choice - 1):
                choices.append(False)

            # Append correct answer (True) to be last choice
            choices.append(True)

            # Shuffle choices list so that correct answer is uniformly randomly inside list
            random.shuffle(choices)
```

```
            # Weights should be 1 for every choice for equal weighting
            weights = [1] * len(choices)

            # Run simulation by randomly choosing from one of the choices equally
            for i in range(num_questions):
                # Randomly and uniformly shuffle choices list each time
                random.shuffle(choices)

                candidate_answer = (random.choices(choices, weights))[0]
                if candidate_answer == True:
                    num_correct_choices = num_correct_choices + 1

            answer_correct_percentages.append(num_correct_choices / num_questions)
    return answer_correct_percentages

def average(nums):
    return sum(nums) / len(nums)

if __name__ == "__main__":
    print("Running HR Interview Question Probability Simulator...")

    # Test overall iterations
    average_of_averages_case_a = []
    average_of_averages_case_b = []
    number_of_averages_to_average = 100
    a_average_higher = 0
    for i in range(number_of_averages_to_average):
        # Case a: last answer is always correct
        last_answer_always_correct = True
        iterations_to_run = 100
        num_questions = 20
        a_results = get_answers_correct_percent(num_questions=num_questions, num_multiple_choice=5,
last_answer_always_correct=last_answer_always_correct, iterations_to_run=iterations_to_run)
        # print("a_results == " + str(a_results))
        print("Average probability of correct answers for Case A in " + str(iterations_to_run) + " iterations: " +
"{:07.5f}".format((average(a_results))))
        average_of_averages_case_a.append(average(a_results))

        # Case b: correct answer is randomly chosen
        last_answer_always_correct = False
        iterations_to_run = 100
        num_questions = 20
        b_results = get_answers_correct_percent(num_questions=num_questions, num_multiple_choice=5,
last_answer_always_correct=last_answer_always_correct, iterations_to_run=iterations_to_run)
        # print("b_results == " + str(b_results))
        print("Average probability of correct answers for Case B in " +
            str(iterations_to_run) + " iterations: " + "{:07.5f}".format((average(b_results))))
        average_of_averages_case_b.append(average(b_results))

        if average(a_results) > average(b_results):
            a_average_higher = a_average_higher + 1

    print("average_of_averages_case_a == " + str(average(average_of_averages_case_a)))
    print("average_of_averages_case_b == " + str(average(average_of_averages_case_b)))
    print("a_average_higher == " + str(a_average_higher))
```

Problem 4 code, *critical_path_simulation.py*:

```python
# Aaron Fox
# CECS 622-01
# Dr. Elmaghraby
# Assignment 3 Problem 4

import random # for uniform random distributions of length of process between nodes

# Node class is used to contain all info abotu the nodes and their connections and connection lengths
class Node:
    def __init__(self, name):
        self.node_connections = []
        self.name = name
        print("Creating Node...")

    # For printing purposes
    def __repr__(self):
        return self.name

    def __str__(self):
        return str(self.name + ". Connections: " + str(self.node_connections))

    def add_connection(self, connecting_node, length):
        self.node_connections.append([connecting_node, length])

    def get_connecting_nodes(self):
        return self.node_connections

    def get_connection_length(self, other_node):
        for node, length in self.node_connections:
            if node == other_node:
                return length

# list_results stores all connections globally for use in main function
list_results = []
# analyze_performance recursively iterates through every node connection
# and attaches every completed path to global variable list_results
def analyze_performance(start_node, end_node, curr_list):
    if start_node == end_node:
        curr_list.append(start_node)
        list_results.append(curr_list)
        return
    for node, node_length in start_node.get_connecting_nodes():
        # Ensure duplicates aren't added to list
        if start_node not in curr_list:
            curr_list.append(start_node)
        copy_list = curr_list.copy()
        analyze_performance(node, end_node, copy_list)

# get_length_of_connections returns the length of all the connections of the path
# INPUT: node_list (list): list of all nodes and their connections, e.g. [Node 1, Node 2, Node 3, Node 4, Node 7]
# OUTPUT: length_sum (float): float sum of all the lengths
def get_length_of_connections(node_list):
    length_sum = 0
    for i in range(len(node_list) - 1):
        length = node_list[i].get_connection_length(node_list[i + 1])
        length_sum = length_sum + length
    # print(node_list)
    return length_sum

if __name__ == "__main__":
    print("Running critical path simulation...")
```

```
node_1 = Node("Node 1")
node_2 = Node("Node 2")
node_3 = Node("Node 3")
node_4 = Node("Node 4")
node_5 = Node("Node 5")
node_6 = Node("Node 6")
node_7 = Node("Node 7")

# Add respective deterministic/uniformly distributed connections per prompt
## Node 1 ##
node_1.add_connection(node_2, random.uniform(4, 6))  # (1, 2): U (4,6)
node_1.add_connection(node_5, 6)  # (1, 5): 6

## Node 2 ##
node_2.add_connection(node_3, 6)  # (2, 3): 6
node_2.add_connection(node_4, random.uniform(6, 8))  # (2, 4): U (6,8)

## Node 3 ##
node_3.add_connection(node_4, random.uniform(4, 8))  # (3, 4): U (4,8)

## Node 4 ##
node_4.add_connection(node_7, 4)  # (4, 7): 4

## Node 5 ##
node_5.add_connection(node_3, 8)  # (5, 3): 8
node_5.add_connection(node_4, 11)  # (5, 4): 11
node_5.add_connection(node_6, random.uniform(8, 10))  # (5, 6): U (8,10)

## Node 6 ##
node_6.add_connection(node_7, random.uniform(9, 10))  # (6, 7): U (9, 10)

# Run over each path to get average values for each path
path_12347_values = []
path_1247_values = []
path_15347_values = []
path_1547_values = []
path_1567_values = []

iterations_to_run = 5000
for i in range(iterations_to_run):
    list_results = []
    analyze_performance(start_node=node_1, end_node=node_7, curr_list=[])

    max_value = 0
    min_value = float('inf')
    max_connection = []
    min_connection = []

    for connection_list in list_results:
        length = get_length_of_connections(connection_list)
        print(connection_list, end='')
        print(": ", end=' ')
        print(length)

        # Assign length values to each path to get average values
        if str(connection_list) == "[Node 1, Node 2, Node 3, Node 4, Node 7]":
            path_12347_values.append(length)
        elif str(connection_list) == "[Node 1, Node 2, Node 4, Node 7]":
            path_1247_values.append(length)
        elif str(connection_list) == "[Node 1, Node 5, Node 3, Node 4, Node 7]":
            path_15347_values.append(length)
```

```
        elif str(connection_list) == "[Node 1, Node 5, Node 4, Node 7]":
            path_1547_values.append(length)
        elif str(connection_list) == "[Node 1, Node 5, Node 6, Node 7]":
            path_1567_values.append(length)


        if max_value < length:
            max_value = length
            max_connection = connection_list
        if min_value > length:
            min_value = length
            min_connection = connection_list

    print("Max Value list: ", end='')
    print(max_connection, end=', Length of ')
    print(max_value, end='\n')
    print("Min Value list: ", end='')
    print(min_connection, end=', Length of ')
    print(min_value, end='\n')
    print("Finished iteration " + str(i + 1), end='\n\n')

print("Average values for each path over " + str(iterations_to_run) + " samples: ")
print("Average path_12347_values: " + str(sum(path_12347_values)/len(path_12347_values)))
print("Average path_1247_values: " + str(sum(path_1247_values)/len(path_1247_values)))
print("Average path_15347_values: " + str(sum(path_15347_values)/len(path_15347_values)))
print("Average path_1547_values: " + str(sum(path_1547_values)/len(path_1547_values)))
print("Average path_1567_values: " + str(sum(path_1567_values)/len(path_1567_values)))
```

References

[1] R. Wicklin, “The inverse CDF method for simulating from a distribution,” *The DO Loop*, 22-Jul-2013. [Online]. Available: https://blogs.sas.com/content/iml/2013/07/22/the-inverse-cdf-method.html. [Accessed: 28-Feb-2020].