Comp 251: Assignment 3

Answers must be submitted online by April 1st (11:55:00 pm), 2021.
Note: we will be accepting submissions without penalty until April 8th, 11:55 pm.

General instructions (Read carefully!)

- **Important:** All of the work you submit must be done by only you, and your work must not be submitted by someone else. Plagiarism is academic fraud and is taken very seriously. For Comp251, we will use software that compares programs for evidence of similar code. This software is very effective and it is able to identify similarities in the code even if you change the name of your variables and the position of your functions. The time that you will spend modifying your code, would be better invested in creating an original solution.

  Please don't copy. We want you to succeed and are here to help. Here are a couple of general guidelines to help you avoid plagiarism:

  Never look at another assignment solution, whether it is on paper or on the computer screen. Never share your assignment solution with another student. This applies to all drafts of a solution and to incomplete solutions. If you find code on the web, or get code from a private tutor, that solves part or all of an assignment, do not use or submit any part of it! A large percentage of the academic offenses in CS involve students who have never met, and who just happened to find the same solution online, or work with the same tutor. If you find a solution, someone else will too. The easiest way to avoid plagiarism is to only discuss a piece of work with the Comp251 TAs, the CS Help Centre TAs, or the COMP 251 instructors.

- Your solution must be submitted electronically on codePost. Here is a short **tutorial** to help you understand how the platform works.

- To some extent, collaborations are allowed. These collaborations should not go as far as sharing code or giving away the answer. You must indicate on your assignments (i.e. as a comment at the beginning of your java source file) the names of the people with whom you collaborated or discussed your assignments (including members of the course staff). If you did not collaborate with anyone, you write "No collaborators". If asked, you should be able to orally explain your solution to a member of the course staff.

- This assignment is due on April 1st at 11h55:00 pm, but we will be accepting late submissions without penalty until April 8th It is your responsibility to guarantee that your assignment is submitted on time. We do not cover technical issues or unexpected difficulties you may encounter. Last minute submissions are at your own risk.

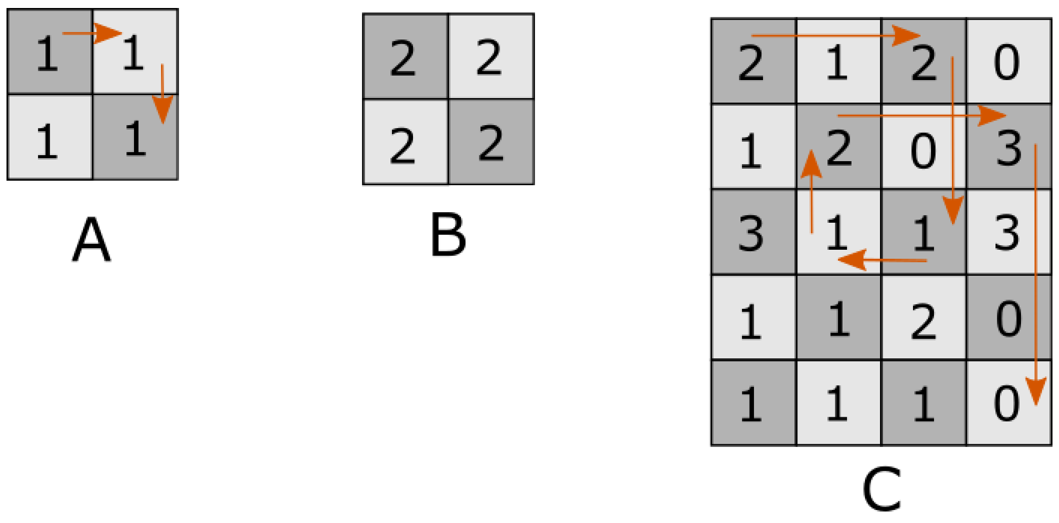
- This assignment includes a programming component, which counts for 100% of the grade, and an optional long answer component designed to prepare you for the exams. This component will not be graded, but a solution guide will be published.

- Multiple submissions are allowed before the deadline. We will only grade the last submitted file. Therefore, we encourage you to submit as early as possible a preliminary version of your solution to avoid any last minute issue.
- Late submissions can be submitted for 24 hours after the extended deadline, and will receive a flat penalty of 20%. We will not accept any submission more than 24 hours after the extended deadline. The submission site will be closed, and there will be no exceptions, except medical.
- In exceptional circumstances, we can grant a small extension of the deadline (e.g. 24h) for medical reasons only. However, such request must be submitted before the deadline.
- Violation of any of the rules above may result in penalties or even absence of grading. If anything is unclear, it is up to you to clarify it by asking either directly the course staff during office hours, by email at (`cs251-winter@cs.mcgill.ca`) or on the discussion board on Piazza (recommended). Please, note that we reserve the right to make specific/targeted announcements affecting/extending these rules in class and/or on Piazza. It is your responsibility to monitor Piazza for announcements.
- The course staff will answer questions about the assignment during office hours or in the online forum. We urge you to ask your questions as early as possible. We cannot guarantee that questions asked less than 24h before the submission deadline will be answered in time. In particular, we will not answer individual emails about the assignment that are sent sent the day of the deadline.

Exercise 1 (Honors (30 points)) Please remember (from our second assignment) that The University and the School of Computer Science approved the law "Sudoku". This law establishes that no computer science student can graduate without implementing (at least once) a Sudoku solving program. This law also establishes the conditions to apply to an "honor" degree. Particularly, in order to graduate with honors you will need to 1) Implement a Sudoku solving program (that you already did in the previous assignment) and to 2) Solve another puzzle. During this exercise, we will help you to complete the requirements to apply for an honor degree.

The board of this exercise has size nXm $(1 \leq n, m \leq 500)$. It is guaranteed that at least one of n and m is greater than 1. Each cell on the board has a digit on it (a number between 0 and 9, inclusive). Your task is to develop an algorithm to compute the minimum number of `moves` required to go from the top-left corner of the board to the bottom-right corner. If it is not possible to go from corner to corner, your algorithm must return the integer -1. From a given cell in the board that has digit j on it, a `move` consist of jumping exactly j cells in one of the four cardinal directions.

Let see some examples to be sure that the exercise is clear. The figure below shows three examples of boards. For the first example (i.e., sub-figure with label 'A'), the value returned by your algorithm must be 2, given that the minimum number of `moves` to go from the the top-left corner of the board to the bottom-right corner is equal to two (the orange arrows depict one of those minimum paths). For the second example (i.e., sub-figure with label 'B'), your algorithm must return the integer '-1', given that there is no path that connects both corners. Note that in this example, any move will locate you outside of the board. For the last example (i.e., sub-figure with label 'C'), your algorithm must return the integer '6'. The orange arrows show the minimum required moves to go from one corner to the other.



Your job for this part of the assignment is to complete the function `min_moves`, which takes an `int[][]` 2d-array that represents the board and returns the minimum number of `moves` needed to go from the the top-left corner of the board to the bottom-right corner. This function is located in the java file `Honors`.

Note: main function

We have already implemented a `main` function and a `test` function to read the data from the files, and to initialize the board that the function `min_moves` is getting as a parameter. Please note that this function will not be graded, and it is there only to make sure that all of the Comp251 students understand the input of the problem and to test their own code.

Exercise 2 (Vaccines (35 points)) After more than one year of research and development, different COVID-19 vaccines are finally being distributed around the world. However, there is a huge difference between the number of vaccines accessed by rich and poor countries. As rich countries ramped their vaccination efforts up, they bought a significant number of vaccines (sometimes more vaccines than their population). Some reports suggest that Canada, for example, has ordered enough vaccines to protect each Canadian five times. Some experts claim that rich nations are 'hoarding' a billion doses of excess COVID vaccine, and that that surplus of vaccine doses would be sufficient to vaccinate entire adult populations in poor countries. G7 leaders have indicated that they will be willing to share their excess doses, but only to their allies poor countries.

The current scenario creates a network of countries that can share their surplus of vaccines; however given the current COVID-19 variants it is still unknown if the rich countries will be willing to do that. For this exercise, we will model the scenario where one (rich) country stops sharing their surplus doses and we will investigate the consequences of that decision in the world. Particularly, we will want to count how many countries were able to achieve herd immunity after the decision.

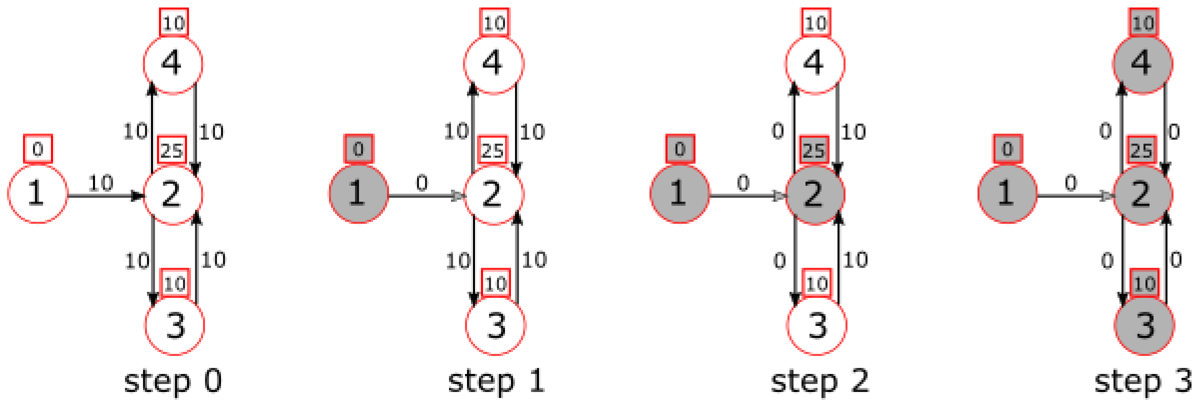
We model the above described situation as follows. Each country has a threshold T_i that represents the number of vaccines needed by country i to guarantee the immunity herd of its population. If the number of vaccines drops below that threshold, the country will immediately make the decision of stopping the sharing of their surplus doses (because they will need the excess to vaccinate its population). This decision means that that country will no longer provide vaccines to its allies countries, thereby potentially causing them to do not be able to pass their own thresholds (having as consequence the stopping of their sharing too). If a country does not have enough vaccines to pass its threshold to guarantee the immunity herd, we assume that vaccines that would have been delivered to that country are effectively lost (the country is locked down); it does not get redistributed and delivered to other country instead. We will start the simulation by always making the first country to stop the sharing of its surplus doses.

Let see some examples to be sure that the exercise is clear. The figure below shows two examples. The scenarios (as expected) are modeled using graphs. The countries are modeled as nodes and they are named using an integer identifier. Each node (i.e., a circle with an ID integer inside) has a square (with an integer value inside) on the top of it. This square (and the integer value inside) represents the threshold T_i for the node with identifier i. Two countries (i.e., nodes) will share vaccine surplus doses if there is an edge connecting them. The edges are directional and they have a weight indicating the amount of vaccines that one country is willing to send to its ally. If a country i is unable to reach the immunity herd threshold, the node will be colored grey and the out-coming edges are set to zero (to represent the fact that the country will stop sending vaccines).

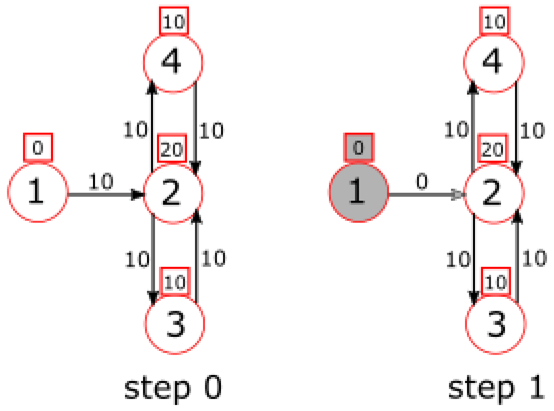
For the first example, we have four countries (labeled from 1 to 4). "step 0" shows the starting scenario. "step 1" shows the scenario just after the first country stops supplying vaccines. Please notice that the simulation will always start by stopping the supply of the first country. Please also notice that the node 1 is colored grey and the edges connecting its neighbors are set to 0. In "step 2", it is clear that the second country (node 2) is unable to guarantee the immunity herd of its population ($T_2 = 25$, but it is only getting 20 vaccines from its neighbors). Please note that the node 2 is colored grey and the edges connecting its neighbors are set to 0. As a consequence of the decision made by country 2 of stopping the supply of vaccines, in "step 3" the countries 3 and 4 are now unable to guarantee the herd immunity of its population ($T_3 = T_4 = 10$, but each country is only getting 0 vaccines from its neighbors). At this point, it is clear that no country was able to achieve the herd immunity and the algorithm must return 0 as the answer.

"Example 2" set up is really similar than that of the first example. The only difference is that now $T_2 = 20$. "step 1" shows the scenario just after the first country stops supplying vaccines. In "step 2", it is clear that the second country (node 2) is able to guarantee the herd immunity of its population, regardless the decision made by the first country ($T_2 = 20$, and it is getting 20 vaccines from its neighbors 3 and 4). At this point, it is clear that three countries (i.e., country 2, 3 and 4) were able to achieve the herd immunity and the algorithm must return 3 as the answer.

Example 1



Example 2



You can safely assume the following limits for the input scenarios.

- The number of countries N is greater or equal than 1 and less or equal than 100.000
- The IDs used to name the countries are distinct and between 1 and N inclusive.
- All the herd immunity threshold are in the following limits ($0 \leq T_i \leq 50.000$)
- The number of allies countries K_i from which the country i can receive vaccines are in the following limits $0 \leq K_i \leq N - 1$
- The amount of vaccines V_{ij} that country i receives from country j satisfy $1 \leq V_{ij} \leq 1000$, and their sum is at least T_i (i.e., the total number of vaccines that the country i receives is at least T_i).
- The sum of all the K_i's for all the N countries is at most 500.000

Your job for this part of the assignment is to complete the function `vaccines`, which takes a graph (i.e., an array of objects of type `Country`) as input and returns and integer representing the number of countries that were able to achieve herd immunity after the first country made the decision to stop sharing its vaccine surplus doses. Each country is modeled using the class `Country`, which has the following signature.

```
public class Country{
  private int ID;
  private int vaccine_threshold;
  private int vaccine_to_receive;
  private ArrayList<Integer> allies_ID;
  private ArrayList<Integer> allies_vaccine;
    ......
    ......
```

The attribute `ID` represents the ID for each country. The attribute `vaccine_threshold` represents the threshold T_i for country with `ID = i`. The attribute `vaccine_to_receive` represent the total number of vaccines that the country will receive from its allies. The attribute `allies_ID` represent the list of IDs of its allies. Finally the attribute `allies_vaccine` represent the amounts of vaccines that the country is willing to share with each of its allies (stored in `allies_ID`). Please notice that `allies_ID` and `allies_vaccine` are parallel list (i.e., the country is willing to share a total of `allies_vaccine[i]` vaccines with the country `allies_ID[i]`).

Note: main function

We have already implemented a `main` function and a `test` to read the data from the files, to create all the initial information and finally to create the graph. Please note that this function will not be graded, and it is there only to make sure that all of the Comp251 students understand the input of the problem and to test their own code.

Exercise 3 (Ford-Fulkerson (35 points)) We will implement the Ford-Fulkerson algorithm to calculate the Maximum Flow of a directed weighted graph. Here, you will use the files WGraph.java and Ford-Fulkerson.java. Your role will be to complete two methods in the template FordFulkerson.java.
The file `WGraph.java` implements two classes `WGraph` and `Edge`. An `Edge` object stores all informations about edges (i.e. the two vertices and the weight of the edge), which are used to build graphs.
The class `WGraph` has two constructors `WGraph()` and `WGraph(String file)`. The first one creates an empty graph and the second uses a file to initialize a graph. Graphs are encoded using the following format: the first line corresponds to two integers, separated by one space, that represent the "source" and the "destination" nodes.The second line of this file is a single integer n that indicates the number of nodes in the graph. Each vertex is labelled with an number in $[0, \cdots, n-1]$, and each integer in $[0, \cdots, n-1]$ represents one and only one vertex. The following lines respect the syntax "n_1 n_2 w", where n_1 and n_2 are integers representing the nodes connected by an edge, and w the weight of this edge. n_1, n_2, and w must be separated by space(s). It includes setter and getter methods for the Edges and the parameters "source" and "destination". There is also a constructor that will allow the creation of a graph cloning a WGraph object. An example of such file can be found in the file `ff2.txt`. These files will be used as an input in the program `FordFulkerson.java` to initialize the graphs. This graph corresponds to the same graph depicted in [CLRS2009] page 727.
Your task will be to complete the two static methods `(fordfulkerson WGraph graph)` and `pathDFS(Integer source, Integer destination, WGraph graph)`. The second method `pathDFS` finds a path via Depth First Search (DFS) between the nodes "source" and "destination" in the "graph". You must return an ArrayList of Integers with the list of unique nodes belonging to the path found by the DFS. The first element in the list must correspond to the "source" node, the second element in the list must be the second node in the path, and so on until the last element (i.e., the "destination" node) is stored. If the path does not exist, return an empty path. The method `fordfulkerson` must compute an integer corresponding to the max flow of the "graph", as well as the graph encoding the assignment associated with this max flow.
Once completed, compile all the java files and run the command line `java FordFulkerson ff2.txt`. Your program will output a String containing the relevant information. An example of the expected output is available in the file `ff2testout.txt`. This output keeps the same format than the file used to build the graph; the only difference is that the first line now represents the maximum flow (instead of the "source" and "destination" nodes). The other lines represent the same graph with the weights updated to the values that allow the maximum flow. There are a few other open test cases you can access on codePost. You are invited to run other examples of your own to verify that your program is correct. There are namely some examples in the textbook.