Homework Instructions: The homework contains mainly three types of tasks:

- **Coding.** You can find a template for all the coding problems inside the `problems.zip` in the module. Please use these templates, to ensure the files and functions have the same names that the autograder will check for.

  Inside the `problems` directory, you will find the `tests` directory, which contains some sample test cases for the different problems. These test cases are intended to help you test and debug your solution. However, note that Gradescope will run a more comprehensive test suite. Feel free to add additional test cases to further test your code.

  In order to run the tests for a particular problem n, navegate into `problems/tests` and run: `python3 test_problem_n.py`

- **Algorithm Description.** When a questions requires designing an algorithm, you should describe what your algorithm does in the writeup, in clear concise prose. You can cite algorithms covered in class to help your description. You should also argue for your algorithm's asymptotic run time or, in some cases when indicated in the problem, for the run time bounds of your implementation.

- **Empirical Performance Analysis.** Some questions may ask you to do an empirical performance analysis of one or more algorithms' runtime under different values of n. For these questions, you should generate at least ten test cases for the implementation, for various values of n (include both small and large instances). Then, measure the performance locally on your own system, by taking the median runtime of the implementation over ten or more iterations. Graph the resulting median run times. The x-axis should be instance size and the y-axis should be median run time. You can use any plotting library of your choice. If you have never plotted on Python before, this matplotlib turorial has some examples.

  If several subquestions in a problem ask you to provide performance analysis, please do them all in the same environment (same system and configuration), in order to have a more accurrate comparison between algorithms. You can include graphs for a same problem in the same plot, as long as the different graphs are properly labeled.

  For these types of questions, you will not need to submit the tests you generate or your code for benchmarking or plotting the algorithms. However, you should include the graphs you generate into the write-up. You should also write about what you observe from the analysis and potentially compare it to the complexity analysis. Finally, also include a description of the environment (CPU, operating system and version, amount of memory) you did the testing on.

Submission Instructions: Hand in your solutions electronically on Gradescope. There are two active assignments for this problem set on Gradescope: one with the autograder, where you submit your code, and another one where you submit the write-up.

Coding Submission: You only need to submit the code for the specific functions we ask you

to implement. While you will write additional code to generate the performance analysis graphs when required, you do not need to submit that code.

To submit to the autograder, please zip the `problems` directory we provided you with. Therefore, your zipped file should have the following structure:

```
problems
├── problem_1
│   ├── p1_a.py
│   ├── p1_b.py
│   └── ...
├── problem_2
│   └── p2.py
└── ...
```

In order to ensure your code runs correctly, do not change the names of the files or functions we have provide you. Additionally, do not import external python libraries in these files, you don't need any libraries to solve the coding assignments. Finally, if you write any helper functions for your solutions, please include them in the same files as the functions that call them.

We have some public tests in the autograder to verify that all the required files are included, so you will be able to verify on Gradescope that your work is properly formatted shortly after submitting. We strongly encourage checking well ahead of the due date that your solutions work on the autograder, and seeking out assistance from the TAs should it not. We cannot guarantee being responsive the night the assignment is due.

Write-up Submission: Your write-up should be a nicely formatted PDF prepared using the LaTeX template on Canvas, where you can type in your answer in the `main.tex` file. If you do not have previous experience using LaTeX, we recommend using Overleaf. It is an online LaTeX editor, where you can upload and edit the template we provide. For additional advice on typesetting your work, please refer to the *resources* directory on the course's website.

Academic Integrity: You may use online sources or tools (such as code generation tools), but any tools you use should be explicitly acknowledged and you must explain how you used them. You are responsible for the correctness of submitted solutions, and we expect you to understand them and be able to explain them when asked by teaching staff.

Collaboration Policy: Collaboration (in groups of up to three students) is encouraged while solving the problems, but:

1. list the netids of those in your group;

2. you may discuss ideas and approaches, but you should not write a detailed argument or code outline together;

3. notes of your discussions should be limited to drawings and a few keywords; you must **write up the solutions and write code on your own**.

Problem 1. This problem involves working with graphs, specifically focusing on bipartite properties and matching.

    (a) Test if a given graph is bipartite using the Breadth First Search (BFS) approach.

        Given the function `is_bipartite(g_l) -> bool`, the purpose is to determine if the graph represented by the adjacency dictionary `g_l` is bipartite or not. A simple BFS algorithm can be employed where nodes are colored in a way that no two adjacent nodes share the same color. If it's possible to color the graph using two colors in this way, the graph is bipartite.

        (*10 points*) Implement the `is_bipartite` function in `problem_1/p1_a.py`.

Code input and output format. The input for **Problem 1 (a)** is a graph represented as an **adjacency dictionary** (see Listing 1), and the output should be a Boolean (True if the graph is bipartite, False otherwise).

```
{0:[1,3], 1:[0,2], 2:[1,3], 3:[0,2]}.
```

Listing 1: The adjacency dictionary representation, where the keys represent a node, and the value list represents the nodes the 'key' node is connected to.

    (b) Determine the maximal bipartite match of a given bipartite graph.

        For the function `maximal_bipartite_match(g) -> int`, the objective is to find the maximum number of matches that can be achieved in the bipartite graph represented by the adjacency matrix `g`.

        (*10 points*) Implement the `maximal_bipartite_match` function in `problem_1/p1_b.py`.

Code input and output format. The input for **Problem 1 (b)** is a bipartite graph represented as an **adjacency matrix** (see Listing 2). The output should be an integer representing the number of maximal matches.

```
[[0, 1, 0, 1], [1, 0, 1, 0], [0, 1, 0, 1], [1, 0, 1, 0]].
```

Listing 2: The adjacency matrix representation, where an entry i, j is 1 if there is an edge between nodes i and j, 0 otherwise.

Problem 2. In a futuristic digital city, there are n data hubs and k service providers that need to be interconnected. As an input, you are given the number of data hubs and service providers and a dictionary `connections` showing which data hubs *can be connected* to which service providers. You are also given `provider_capacity` showing *how many connections* each service provider can handle, and `preliminary_assignment` which has the assignment for all the other hubs except the last one. These variables are shown in the example code snipped in Listing 3. In this problem, data hubs are labelled $0, 1, ..., n-1$ and service providers are labelled $n, n+1, ..., n+k-1$.

```
plan_city_a(num_data_hubs = 5,
            num_service_providers = 5,
            connections = {0: [5,7,8], 1: [5, 8], 2: [7,8,9], 3: [5, 6, 8, 9], 4: [5,6,7,8]},
            provider_capacities = [0]*5 + [0,1,0,2,2],
            preliminaryAssignment = {0: 8, 1: 8, 2: 9, 3: 9})
```
Listing 3: The python function call to be made to generate the graphs for Problems 2.a,b,c.

Your task is to ensure each data hub is connected to a service provider while considering capacity constraints. If the last hub $(n-1)$ can't be connected, find out which service provider's capacities need to be increased by one for a feasible solution.

To solve this, you will make use of a graph-based approach and the Ford-Fulkerson algorithm. In this graph, the source is a vertex from which flow starts, and the sink is the one where flow ends. The source will be connected to each data hub and each service provider to the sink. If a data hub can be connected to a service provider, there will be a directed edge between them in the graph. The capacity of the edges from the source to each data hub, as well as the capacity of each edge from a data hub to its possible service providers, is 1. The capacity from service providers to the sink is based on `provider_capacity`. In this problem, constructing the residual graph correctly is extremely important. Therefore, part (a), (b) and (c) will focus on an incremental approach to building the correct graph. **Part (a), (b) and (c) are to be implemented in the function** `plan_city_a` **in** `problem_2/p2_a.py`.

Note that in the example shown in Listing 3, since there are only 5 service providers, we assign a capacity of '0' to the 5 data hubs for convenience in implementation. Thus `provider_capacities` is to be described as [0]*`num_data_hubs` (5 in this example) + list(`service_provider_capacity`).

    (a) (*5 points*) Generate the first part of the visualization graph showing connectivity from the source to data hubs to the service providers, based on the given connections dictionary. Implement this in `problem_2/p2_a.py`. Generate the graph using the input shown in Listing 3 and provide the visualization with the caption (Problem 2.a.) in the write-up.

    (b) (*5 points*) Extend the graph by connecting the service providers to the sink using the given provider_capacity. Implement this in `problem_2/p2_a.py`. Generate the graph using the input shown in Listing 3 and provide the visualization with the figure caption (Problem 2.b.) in the write-up.

    (c) (*10 points*) Using the graphs from the previous parts, create the residual graph. This graph will help in determining the feasible flow of connections. Implement this in `problem_2/p2_a.py`. Generate the graph using the input displayed in Listing 3 and provide the visualization with the figure caption (Problem 2.c.) in the write-up.
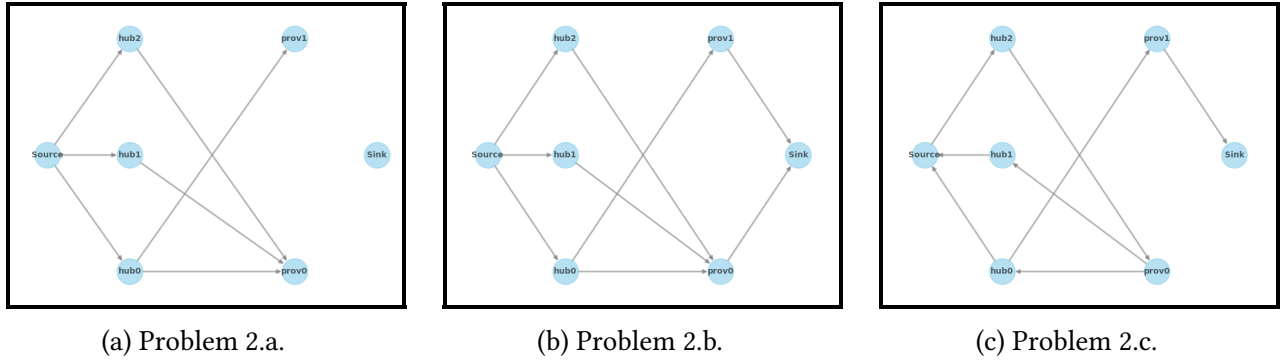
(a) Problem 2.a.	(b) Problem 2.b.	(c) Problem 2.c.

Figure 1: Example generated graphs for the problems 2.a, 2.b and 2.c.

(d) (*10 points*) Implement the algorithm to determine if each data hub can be connected to a service provider. Return 'True' if each data-hub can be connected to a service provider, otherwise return 'False'. Implement this in `problem_2/p2_d.py`. Note that you can simply copy paste the code for the residual graph generation from `problem_2/p2_a.py` and focus on the rest of the algorithmic implementation.

(e) (*20 points*) Extend the implementation to now provide the final connectivity map if it is feasible. If it is not feasible, the output should be a list of zeros for each data hub, followed by a list of 0s and 1s indicating which service providers capacities should be increased by one for a feasible solution. Implement this in `problem_2/p2_e.py`. Note that you can simply copy paste the code from `problem_2/p2_d.py` and focus on the rest of the algorithmic implementation.

Code input and output format. You are to design a function called plan_city with the following inputs:

- `num_data_hubs`(n): An integer representing the number of data hubs.

- `num_service_providers`(k): An integer representing the number of service providers.

- `connections`: A dictionary representing potential connections between data hubs and service providers.

- `provider_capacities`: A list indicating the capacities of each service provider.

- `preliminary_assignment`: A dictionary where the keys are the data hubs and the values are the service providers they are initially assigned to. The last data hub ($n - 1$) will not have a connection.

The output should be a list of integers representing the final connectivity of each data hub to its assigned service provider if a feasible assignment exists. If not feasible, the output should be **a list of zeros for each data hub followed by indicators (1 or 0) for each service provider**, where a 1 indicates the provider whose capacity should be increased.

Examples:

- For num_data_hubs = 3, num_service_providers = 2, connections = {0: [3,4], 1: [3], 2:

[3]}, provider_capacities = [0]*num_data_hubs + [2, 1] (naturally zero for data hubs), and preliminary_assignment = {0: 3, 1: 3}, a fully satisfying assignment exists, the valid assignment is [4, 3, 3].

- For num_data_hubs = 5, num_service_providers = 5, connections = {0: [5, 7, 8], 1: [5, 8], 2: [7, 8, 9], 3: [5, 6, 8, 9], 4: [5, 6, 7, 8]}, provider_capacities = [0]*5 + [0, 1, 0, 2, 2], and preliminary_assignment = {0: 8, 1: 8, 2: 9, 3: 9}. A fully satisfying assignment exists, and the valid assignment is [8, 8, 9, 9, 6].

- For num_data_hubs = 5, num_service_providers = 5, connections = {0: [5, 6], 1: [5, 7, 8, 9], 2: [5, 7, 9], 3: [5, 7, 8, 9], 4: [5, 6, 9]}, provider_capacities = [0]*5 + [0, 1, 3, 0, 0], and preliminary_assignment = {0: 6, 1: 7, 2: 7, 3: 7}. No fully satisfying assignment exists, and thus the list of providers whose capacity should be increased should be [0] * num_data_hubs + [1, 1, 0, 0, 1] = [0, 0, 0, 0, 0, 1, 1, 0, 0, 1].
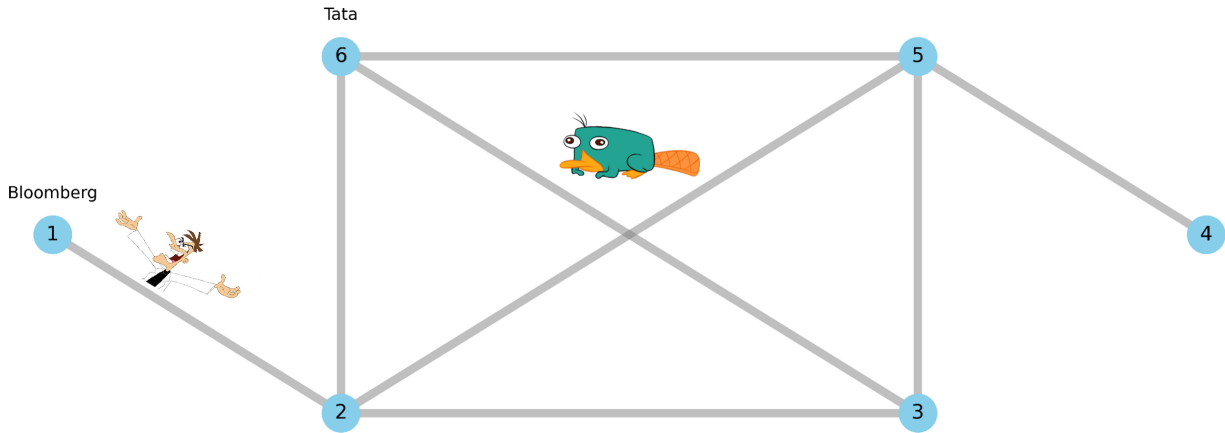
Tata

Bloomberg

Problem 3. Dr. Heinz Doofenshmirtz has just discovered a secret product studio idea in the Bloomberg Masters Studio and is making his way to the Tata Innovation Center to present it. Perry the Platypus from a rival start-up aims to intercept Dr. Heinz Doofenshmirtz by blocking the paths to the Tata Innovation Center. Determine the fewest number of paths that Perry the Platypus should block so that there's no path between Masters Studio and Tata.

(a) (*30 points*) Compute the minimal number of paths to be blocked by Perry the Platypus.

Code input and output format. You are to design a function called `find_paths` with the following inputs:

- n, which correspond to the total number of junctions. The junctions are numbered from 1 to n, Dr. Heinz Doofenshmirtz is at the masters studio, which is at junction 1. Tata Innovation Center is at junction n.

- paths: A list of tuples, where each tuple contains two integers—x and y—indicating a passage between junctions x and y. All paths are accessible in both directions, and there is at most one passage between two junctions.

Examples:

- For $n = 5$, paths = [(1,2), (3,4), (1,5), (2,5)], the paths (1,2) and (1,5) or (2,5) and (1,5) need to be blocked off to ensure product studio failure. The output should be 2.

- For $n = 12$, paths = [(1,4), (1,8), (1,9), (8, 9), (9, 12), (8, 11), (8, 12), (3,4), (3, 12), (1,5), (2,5)], three paths, such as (1, 4), (1, 8), (1, 9), need to be blocked off to ensure product studio failure. The output should be 3.

Instructions: Challenge problems are, as the term indicates, *challenging*. They do not count for the homework score (90% of your course grade); instead, they are considered separately as extra credit over your course grade (additional 15% in total, 3.75% per assignment).

Questions about challenge problems will have lowest priority in office hours, and we do not provide assistance beyond a few hints to help you know whether you are on the right track.

Submission Instructions: You can choose not to hand a submission, but we encourage everyone to attempt the challenge problem. If you solve it, please hand in your answers (both, the coding and the write-up) through Gradescope, along with the rest of your assignment.

Academic Integrity and Collaboration Policy: The same guidelines apply to challenge problems as for the regular homework problems.

You and your n friends have a collection of cards from the hit card game One! Each card has a numeric value, ranging from 1 to m, and each card belongs to one of k colors, which we represent as numbers between 1 and k for convenience. So a card is a tuple (x, c) where $x \in \{1, \ldots, m\}$ is its numeric value, and $c \in \{1, \ldots, k\}$ is its color.

We can represent a regular 52-card deck by taking $m = 13$ and $k = 4$, with Ace representing the numeric value 1, Jack being 11, Queen being 12, and King being 13, and clubs being suit 1, diamonds being 2, hearts being 3 and spades being 4.

To keep track of the number of cards you have over multiple decks, you and your friends decide to make stacks out of the cards. The stacks have to meet the following constraints:

- Every stack has to start with the lowest card (value 1) and end with the highest (value m).

- Within each stack, each card (x, c) can be followed by card (x', c') in the following cases:

    - if the next card is of the same color and the next higher value (that is, $c' = c$ and $x' = x + 1$),

    - if the next card is of any different color and the same value (that is, c' can be anything, and $x' = x$).

Each of you will get some cards (not necessarily the same number of cards each). You are seated around a table and you label the people clockwise from $i = 1$ to n. Turns are done in the following way:

- Any person can start the next stack, by putting one or more cards (according to the rules described above).

- Once the stack has been started by person i, you take turns in increasing circular order

placing cards on the stack following the rules described above (so the next person to place a card will be $(i + 1) \mod n$).

- You and your friends continue taking turns this way until the stack is complete.

- Each person can play multiple cards, but at least one card must be played each turn.

Naturally, each card can only be part of one stack.

Given the cards held by you and each of your friends (where not all possible cards are necessarily present, and there may also be duplicates), design a polynomial-time algorithm to find out the maximum number of stacks you can create from your cards. Implement it in `challenge_1/cards_a.py`.

Code input and output format. You are to design a function called cards_game with the following inputs:

- m, n, and k correspond to the range of the cards numeric value $(1, \ldots, m)$, the number of friends (n) and the number of colors (k).

- counts is a dictionary where for each 'friend' as the dictionary key, we have a list of tuples (a, b) indicating the cards numeric value and color respectively.

Example:

Let $m = 3$, $k = 2$ and $n = 3$, and suppose you and your friends have the following cards:

Person 1: $(1, 2), (3, 2)$

Person 2: $(1, 1), (2, 1), (2, 2)$

Person 3: $(2, 2), (3, 2)$

Thus, the function call would be:

```
cards_game(m=3, k=2, n=3, counts = {1: [(1,2),(3,2)], 2: [(1,1), (2,1), (2,2)], 3: [(2,2), (3,2)]})
```

Then we can make two stacks:

- Person 1 places $(1, 2)$; person 2 places $(2, 2)$; person 3 places $(3, 2)$

- Person 2 places $(1, 1), (2, 1)$; person 3 places $(2, 2)$; person 1 places $(3, 2)$.

For the following input:

Person 1: $(1, 2), (2, 2), (2, 2), (3, 2)$

Person 2: $(1, 1), (2, 1)$

Person 3: $(3, 2)$

The function call would be:

```
cards_game(m=3, k=2, n=3, counts = {
                1: [(1,2), (2,2), (2,2), (3,2)],
                2: [(1,1), (2,1)],
                3: [(3,2)]})
```

Only one stack is possible.

A little help to get you started:

- Our current topic is network flows, so think about whether you can reduce the stated problem to the maximum flow problem.

- Setting up good notation to express what you want to achieve is a good first step to help you think about this! Because of the rules of the game, it seems sensible to represent a card as (i, x, c) where i is the person holding the card, x is the card's value, and c is the card's suit. What can you say about a card (i', x', c') if it is possible to place it on top of (i, x, c)?