CS 5112 Fall 2024	HW4: *Randomized and Streaming Algorithms*
Name: Samy Lokanandi	**Due: December 10, 11:59pm ET**
NetID: sl3539	
Collaborators: Fill Here	

Problem 1. 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 1. 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 1: The python function call to be made to generate the graphs for Problems 2.a,b,c.

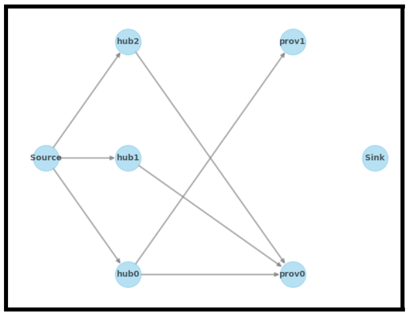
Your task is to ensure each data hub is connected to a service provider while considering capacity constraints. If the last hub ($n-1$) can't be connected, find out which service provider's capacities need to be increased by one for a feasible solution.

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_1/p1_a.py`.

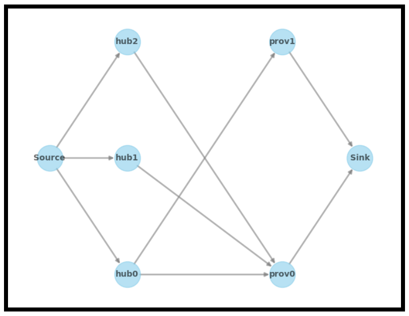
Note that in the example shown in Listing 1, since there are only 5 service providers, we assign a capacity of '0' to the 5 data hubs for convenience in implementation. Thus `provider_capacities` is to be described as [0]*`num_data_hubs` (5 in this example) + list(`service_provider_capacity`).

(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_1/p1_a.py`. Generate the graph using the input shown in Listing 1 and provide the visualization with the caption (Problem 1.a.) in the write-up.

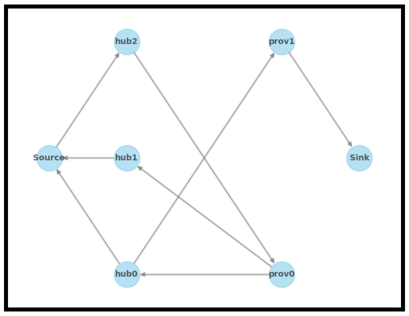
(b) (*5 points*) Extend the graph by connecting the service providers to the sink using the given provider_capacity. Implement this in `problem_1/p1_a.py`. Generate the graph using the



(a) Problem 1.a.



(b) Problem 1.b.



(c) Problem 1.c.

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

input shown in Listing 1 and provide the visualization with the figure caption (Problem 1.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_1/p1_a.py`. Generate the graph using the input displayed in Listing 1 and provide the visualization with the figure caption (Problem 1.c.) in the write-up.

(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_1/p1_d.py`. Note that you can simply copy paste the code for the residual graph generation from `problem_1/p1_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_1/p1_e.py`. Note that you can simply copy paste the code from `problem_1/p1_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].

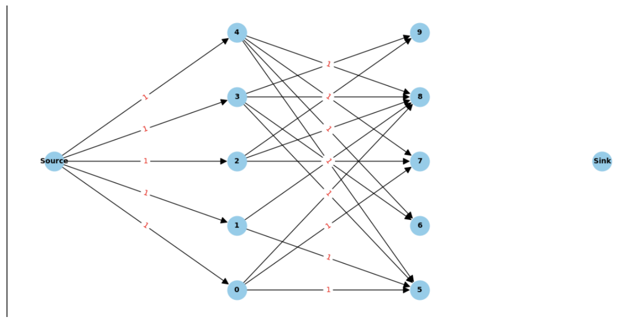


Figure 2: Graph showing the initial connectivity from the source to data hubs and service providers (Problem 1.a). This graph visualizes which data hubs can connect to which service providers based on the given connections dictionary.

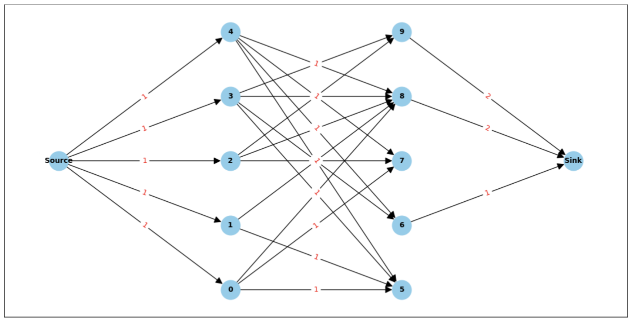


Figure 3: Graph extended with edges connecting service providers to the sink (Problem 1.b). This graph incorporates the provider capacities as edges connecting the service providers to the sink, enforcing their connection limits.

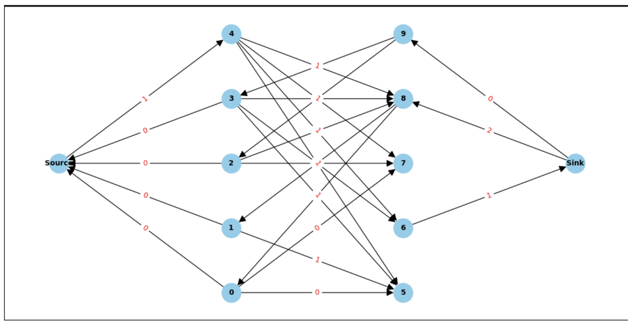


Figure 4: Residual graph for maximum flow calculation (Problem 1.c). This graph is a result of the residual capacity calculation, showing the available capacities for possible augmenting paths.

This problem involves connecting n data hubs to k service providers under specific constraints. The objective is to use a graph-based approach to model and solve the connection problem, incorporating the Ford-Fulkerson algorithm for maximum flow analysis. Below are the summaries for parts (a–e) of the problem:

Part (a)

In this step, the graph is constructed to show the initial connectivity between the source, data hubs, and service providers. Using the `connections` dictionary, edges are added between the source and each data hub with a capacity of 1, representing that each data hub can handle one connection. Additionally, edges are created between each data hub and its connected service providers based on the input data. These edges also have a capacity of 1, ensuring that a hub can connect to at most one service provider at a time. The resulting graph provides a clear representation of the allowed connections in the system. This graph is visualized to help identify bottlenecks or missing links in the initial configuration. This functionality is implemented in `plan_city_a`, and the visualization helps verify the structure of the input data.

Part (b)

The graph constructed in part (a) is extended by connecting the service providers to the sink. Edges are added between each service provider and the sink, with capacities defined by the `provider_capacities` array. This step incorporates the service provider constraints into the graph, ensuring that each provider can only serve a limited number of data hubs as specified in the input. The addition of these edges completes the network flow system, enabling us to analyze the flow of connections from the source to the sink through the data hubs and service providers. This extended graph provides a complete representation of the problem and is essential for solving the maximum flow problem. The graph is visualized in `plan_city_a`, helping to confirm that capacity constraints are correctly applied.

Part (c)

Using the connectivity graph from parts (a) and (b), a residual graph is created to represent the remaining capacities in the network after processing the preliminary assignments. The residual graph is a key tool in the Ford-Fulkerson algorithm, showing how much capacity is left on each edge for possible augmenting paths. It is updated dynamically to account for assigned flows, ensuring that capacity constraints are respected. This graph is particularly useful for detecting if there are additional augmenting paths that can increase the overall flow. In this step, edges are adjusted to reflect the preliminary assignments by reducing forward capacities and increasing reverse capacities. The visualization of the residual graph, implemented in `plan_city_c_residual`, provides insight into the state of the network after the preliminary assignments.

Part (d)

This step involves implementing an algorithm to determine whether all data hubs can be connected to service providers under the given capacity constraints. Using the residual graph from part (c), the algorithm performs a breadth-first search (BFS) to find augmenting paths from the last data hub to the sink. If a path exists, it indicates that the data hub can be successfully connected to a service provider. The function returns `True` if all hubs can be connected and `False` otherwise. This feasibility check ensures that the input constraints are satisfied and that the configuration is viable. This step is implemented in `plan_city_d`, and its output determines whether further adjustments are necessary.

Part (e)

Building on the feasibility check from part (d), this step extends the algorithm to produce a final assignment of data hubs to service providers if a feasible solution exists. If feasibility is not achievable, the algorithm identifies which service providers' capacities need to be increased to enable a valid solution. This is done by examining the residual graph for unsatisfied constraints and determining the minimum adjustments required. The function outputs either the final assignment of data hubs to service providers or a list of zeros (for data hubs) followed by indicators (0 or 1) for each service provider, where 1 indicates the need for a capacity increase. This step is implemented in `plan_city_e` and provides a comprehensive solution to the problem, either directly or with clear guidance for adjustment.

Problem 2. In a charming town renowned for unique collectibles, a shopkeeper Samuel faced a challenge. He had received n packages, each with distinct sizes, and needed to find boxes to match. Samuel had multiple suppliers denoted by m, each offering boxes of varying dimensions. His goal was to choose a supplier whose boxes would minimize the total wasted space, defined as the difference between the box and package sizes. Note that **one box can contain only one package**. Your task is to help Samuel choose a single supplier and use boxes from them such that the total wasted space is minimized. Let the size of the i^{th} box be b_i and the size of the j^{th} package be p_j. For each package in a box, we define the space wasted to be $b_i - p_j$. The total wasted space is the sum of the space wasted in all the boxes.

For example, if you have to fit packages with sizes `[2,3,5]` and the supplier offers boxes of sizes `[4, 8]`, you can fit the packages of size `2` and size `3` into two boxes of size `4` and the package with size `5` into a box of size `8`. This would result in a waste of `(4-2) + (4-3) + (8-5) = 6`.

(10 points) Implement the function `linear_search` in `problem_2/p2_a.py` that has a time-complexity of $\Theta(m \times n \times b)$ where b denotes the average number of box types offered by each supplier.

(30 points) Implement the function `binary_search` in `problem_2/p2_b.py` that has a time-complexity of $\Theta(m \times \log(n) \times b)$ where b denotes the average number of box types offered by each supplier.

Code input and output format. The package sizes are given as an integer array `packages`, **each with distinct sizes**, where `packages[i]` is the size of the i^{th} package. The suppliers are given as 2D integer array boxes, where `boxes[j]` is an array of box sizes that the j^{th} supplier produces. Return the **minimum total wasted space** by choosing the box supplier **optimally**, or -1 if it is **impossible** to fit all the packages inside boxes.

Constraints:

$$1 \le n \le 10^5$$
$$1 \le m \le 10^5$$
$$1 \le \texttt{packages}[i] \le 10^5$$
$$len(\texttt{boxes}[j]) \le 10^5$$
$$\texttt{boxes}[j][k] \le 10^5$$

The elements in `boxes`$[j]$ are **distinct**.

Example 1:

- Input: `packages = [2,3,5]`, `boxes = [[4,8],[2,8]]`
- Output: 6
- Explanation: It is optimal to choose the first supplier, using two size-4 boxes and one size-8 box. The total waste is (4-2) + (4-3) + (8-5) = 6.

Example 2:

- `packages = [2,3,5]`, `boxes = [[1,4],[2,3],[3,4]]`

- Output: -1
- Explanation: There is no box that the package of size 5 can fit in.

Example 3:

- Input: `packages = [3,5,8,10,11,12]`, `boxes = [[12],[11,9],[10,5,14]]`
- Output: 9
- It is optimal to choose the third supplier, using two size-5 boxes, two size-10 boxes, and two size-14 boxes. The total waste is (5-3) + (5-5) + (10-8) + (10-10) + (14-11) + (14-12) = 9.

Answer:

The goal is to minimize the total wasted space when fitting packages into boxes provided by multiple suppliers. Wasted space is defined as the difference between the box size and the package size, with the constraint that one box can hold only one package. Two approaches were implemented: a brute-force `linear_search` algorithm and an optimized `binary_search` algorithm.

The `linear_search` function iterates through each supplier's sorted boxes and attempts to fit all packages in the order of their sizes. For each box, it calculates the wasted space for all packages it can fit and tracks the supplier with the minimum wasted space. This approach has a time complexity of $O(m \cdot n \cdot b)$, where m is the number of suppliers, n is the number of packages, and b is the average number of boxes per supplier. While effective, it can be computationally expensive for larger inputs.

The `binary_search` function improves efficiency by using a binary search helper to group packages that can fit into each box, reducing the complexity of calculating wasted space. The total wasted space is adjusted by subtracting the total package size from the cumulative box sizes. This reduces the time complexity to $O(m \cdot \log n \cdot b)$. Both approaches ensure correctness by checking if a supplier can fit all packages, and return -1 if no valid configuration exists.

CS 5112 Fall 2023	HW4: Challenge Problem
Name: Samy Lokanandi	**Due: December 10, 11:59pm ET**
NetID: sl3539	
Collaborators: Fill Here	

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.

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,..., 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 tuple (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)
    ]})
```

Listing 2: The python function call to be made to generate the graphs for Problems 2.a,b,c.

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)]})
```

Listing 3: The python function call to be made to generate the graphs for Problems 2.a,b,c.

Only one stack is possible.

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), implement 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`.

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)?

Answer:

Please submit functional code. No written component for this problem.