

**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 snippet in Listing 1. In this problem, data hubs are labelled  $0, 1, \dots, n - 1$  and service providers are labelled  $n, n + 1, \dots, n + k - 1$ .

```

1 plan_city_a(num_data_hubs = 5,
2             num_service_providers = 5,
3             connections = {0: [5,7,8], 1: [5, 8], 2: [7,8,9], 3: [5, 6, 8, 9], 4: [5,6,7,8]},
4             provider_capacities = [0]*5 + [0,1,0,2,2],
5             preliminaryAssignment = {0: 8, 1: 8, 2: 9, 3: 9})

```

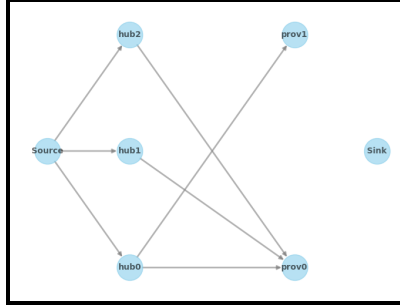
Listing 1: The python function call to be made to generate the graphs for Problems 2.a,b,c.

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.

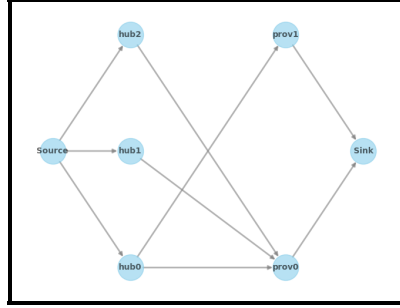
To solve this, you will make use of a graph-based approach and the Ford-Fulkerson algorithm. In this graph, the source is a vertex from which flow starts, and the sink is the one where flow ends. The source will be connected to each data hub and each service provider to the sink. If a data hub can be 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  $[0] * \text{num\_data\_hubs}$  (5 in this example) + list(service\_provider\_capacity).

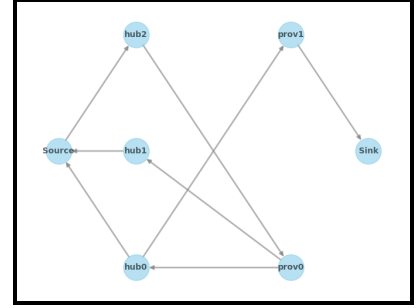
- (a) (5 points) Generate the first part of the visualization graph showing connectivity from the source to data hubs to the service providers, based on the given connections dictionary. Implement this in problem\_1/p1\_a.py. Generate the graph using the input shown in Listing 1 and provide the visualization 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  $\text{num\_data\_hubs} = 3$ ,  $\text{num\_service\_providers} = 2$ ,  $\text{connections} = \{0: [3,4], 1: [3], 2: [3]\}$ ,  $\text{provider\_capacities} = [0] * \text{num\_data\_hubs} + [2, 1]$  (naturally zero for data hubs), and  $\text{preliminary\_assignment} = \{0: 3, 1: 3\}$ , a fully satisfying assignment exists, the valid assignment is  $[4, 3, 3]$ .
- For  $\text{num\_data\_hubs} = 5$ ,  $\text{num\_service\_providers} = 5$ ,  $\text{connections} = \{0: [5, 7, 8], 1: [5, 8], 2: [7, 8, 9], 3: [5, 6, 8, 9], 4: [5, 6, 7, 8]\}$ ,  $\text{provider\_capacities} = [0] * 5 + [0, 1, 0, 2, 2]$ , and  $\text{preliminary\_assignment} = \{0: 8, 1: 8, 2: 9, 3: 9\}$ . A fully satisfying assignment exists, and the valid assignment is  $[8, 8, 9, 9, 6]$ .
- For  $\text{num\_data\_hubs} = 5$ ,  $\text{num\_service\_providers} = 5$ ,  $\text{connections} = \{0: [5, 6], 1: [5, 7, 8, 9], 2: [5, 7, 9], 3: [5, 7, 8, 9], 4: [5, 6, 9]\}$ ,  $\text{provider\_capacities} = [0] * 5 + [0, 1, 3, 0, 0]$ , and  $\text{preliminary\_assignment} = \{0: 6, 1: 7, 2: 7, 3: 7\}$ . No fully satisfying assignment exists, and thus the list of providers whose capacity should be increased should be  $[0] * \text{num\_data\_hubs} + [1, 1, 0, 0, 1] = [0, 0, 0, 0, 0, 1, 1, 0, 0, 1]$ .

Answer:

(a)

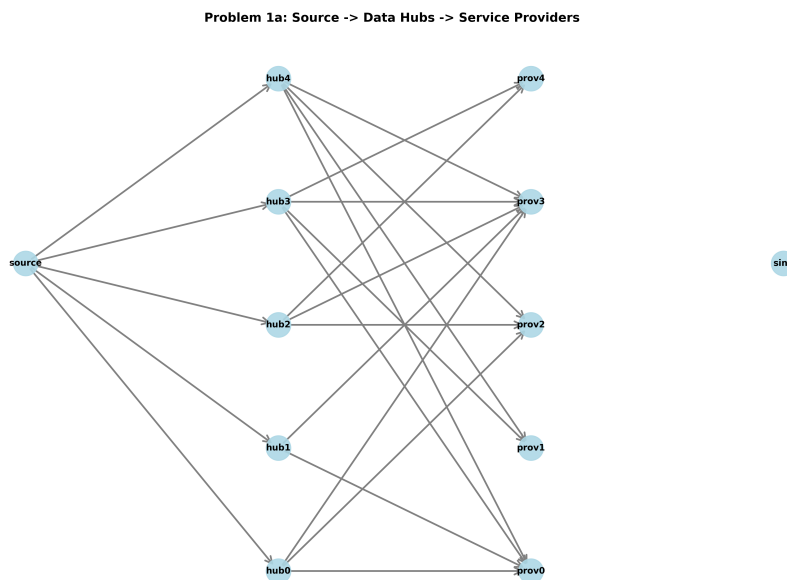


Figure 2: Problem 1.a

(b)

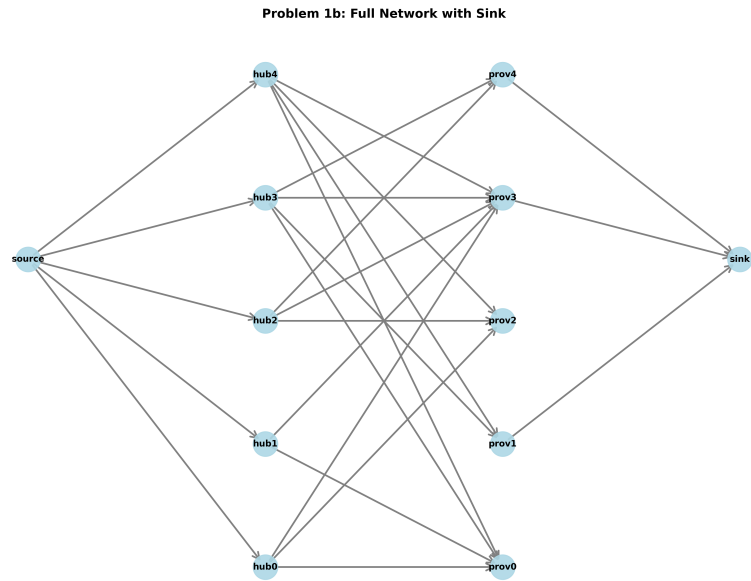


Figure 3: Problem 1.b

(c)

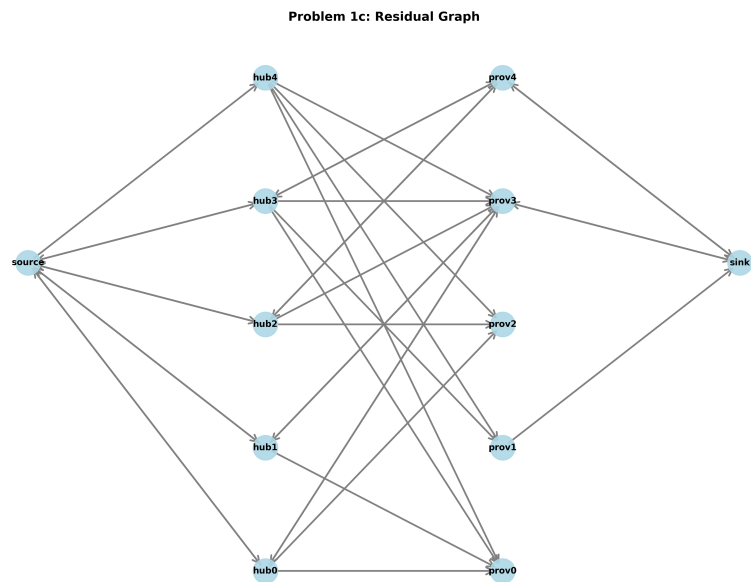


Figure 4: Problem 1.c

**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:**

$$\begin{aligned} 1 &\leq n \leq 10^5 \\ 1 &\leq m \leq 10^5 \\ 1 &\leq \text{packages}[i] \leq 10^5 \\ \text{len}(\text{boxes}[j]) &\leq 10^5 \\ \text{boxes}[j][k] &\leq 10^5 \end{aligned}$$

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: My implementation is submitted for Auto Grader, I'll explain briefly of my implementation here.

**(a)** It sorts all packages once, then for each supplier, sorts their boxes and performs a linear search through all boxes for each package to find the smallest fitting box. The time complexity is  $O(m \times n \times b)$  because for each of the  $m$  suppliers, it iterates through  $n$  packages, and for each package it performs a linear search through  $b$  boxes.

**(b)** Like (a), it sorts all packages for binary search later. Then, it pre-computes a prefix sum array of the sorted packages. Then it processes each box in sorted order and use binary search to find the largest package index that fits in that box, then greedily assign all unassigned packages up to that index to the current box. The time complexity is  $O(m \times b \times \log n)$  because for each of the  $m$  suppliers, it iterates through  $b$  boxes, and for each box it performs a binary search on the sorted packages which takes  $O(\log n)$  time.

**Problem 3.** Let's say we want to count the number of times elements appear in a stream of data,  $x_1, \dots, x_q$ . A simple solution is to maintain a hash table that maps elements to their frequencies.

This approach does not scale: Imagine having an extremely large stream consisting of mostly unique elements. For example, consider network monitoring (either for large network flows or anomalies), large service analytics (e.g. Amazon view/buy counts, Google search popularity), database analytics, etc. Even if we are only interested in the most important ones, this method has huge space requirements. Since we do not know for which items to store counts, our hash table will grow to contain billions of elements.

The Count-Min Sketch, or CMS for short, is a data structure that solves this problem in an approximate way.

**Approximate Counting with Hashing.** Given that we only have limited space availability, it would help if we could get away with not storing elements themselves but just their counts. To this end, let's try to use only an array, with  $w$  memory cells, for storing counts as shown below in Figure 5.

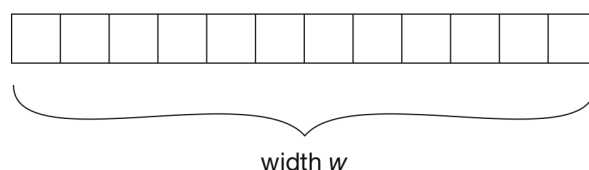


Figure 5: Counting with a single hash

With the help of a hash function  $h$ , we can implement a counting mechanism based on this array. To increment the count for element  $x$ , we hash it to get an index into the array. The cell at the respective index  $h(x)$  is then incremented by 1.

Concretely, this data structure has the following operations:

- Initialization:  $\forall i \in \{1, \dots, w\}: \text{count}[i]=0$ .
- Increment count (of element  $x$ ):  $\text{count}[h(x)]+=1$
- Retrieve count (of element  $x$ ):  $\text{count}[h(x)]$

This approach has the obvious drawback of hash conflicts, which result in over-counting. We would need a lot of space to make collisions unlikely enough to get accurate counts. However, we at least do not need to explicitly store keys anymore.

### More hash functions

Instead of just using one hash function, we could use  $d$  different ones. These hash functions should be pairwise independent. To update a count, we then hash item  $a$  with all  $d$  hash functions, and subsequently increment all indices we got this way. In case two hash functions map to the same index, we only increment its cell once.

Unless we increase the available space, of course all this does for now is to just increase the number of hash conflicts. We will deal with that in the next section. For now let's continue with this thought for a moment.

If we now want to retrieve a count, there are up to  $d$  different cells to look at. The natural solution is to take the minimum value of all of these. This is going to be the cell which had the fewest hash conflicts with other cells.

$$\min_{i=1}^d \text{count}[\mathbf{h}_i(x)]$$

While we are not fully there yet, this is the fundamental idea of the Count-Min Sketch. Its name stems from the process of retrieving counts by taking the minimum value.

### Fewer hash conflicts

We added more hash functions but it is not evident whether this helps in any way. If we use the same amount of space, we definitely increase hash conflicts. In fact, this implies an undesirable property of our solution: Adding more hash functions increases potential errors in the counts.

Instead of trying to reason about how these hash functions influence each other, we can design our data structure in a better way. To this end, we use a matrix of size  $w \times d$ . Rather than working on an array of length  $w$ , we add another dimension based on the number of hash functions as shown below in fig. 6.

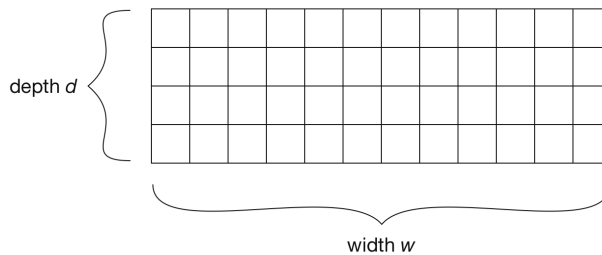


Figure 6: Counting with multiple hashes.

Next, we change our update logic so that each function operates on its own row. This way, hash functions cannot conflict with another anymore. To increment the count of element  $a$ , we now hash it with a different function once for each row. The count is then incremented in exactly one cell per row.

- Initialization:  $\forall i \in \{1, \dots, d\}, j \in \{1, \dots, w\} : \text{count}[i, j] = 0$
- Increment count (of element  $x$ ):  $\forall i \in \{1, \dots, d\} : \text{count}[i, \mathbf{h}_i(x)] += 1$
- Retrieve count (of element  $x$ ):  $\min_{i=1}^d \text{count}[i, \mathbf{h}_i(x)]$

This is the full CMS data structure. We call the underlying matrix a sketch.

- (a) **(10 points)** Your friend implemented the following hash functions for each row in the sketch:  $\mathbf{h}_i(x) = (x + a_i) \bmod w$  for  $0 < i < d$  where  $a_i$  is chosen randomly for each row, and mod is the mod operation (sometimes written as %). Is the choice of hash functions good or bad? Please justify your answer in 1 – 2 sentences or provide a counter example.



Answer: This is a **BAD** choice of hash functions.

**Justification:** If two elements  $x_1$  and  $x_2$  collide under one hash function, i.e  $x_1 \equiv x_2 \pmod{w}$ , they will collide under *all* hash functions  $h_i$  regardless of the value of  $a_i$ . This defeats the purpose of Count-Min Sketch, which relies on independent hash functions to reduce overcounting from collisions.

**Counter-example:** Let  $w = 10$ ,  $d = 3$ , and  $\mathbf{a} = [1, 2, 3]$ . Consider elements  $x_1 = 5$  and  $x_2 = 15$ . Then:

$$\begin{aligned}h_1(5) &= (5 + 1) \bmod 10 = 6, & h_1(15) &= (15 + 1) \bmod 10 = 6 \\h_2(5) &= (5 + 2) \bmod 10 = 7, & h_2(15) &= (15 + 2) \bmod 10 = 7 \\h_3(5) &= (5 + 3) \bmod 10 = 8, & h_3(15) &= (15 + 3) \bmod 10 = 8\end{aligned}$$

The elements collide in every row, making the min operation useless since all rows have the same inflated count.

- (b) **(10 points)** Implement CMS in the function `count_min_sketch` in `problem_3/p3_b.py`. Given the vectors  $\mathbf{a} = [a_1, \dots, a_d]$ ,  $\mathbf{b} = [b_1, \dots, b_d]$ , a scalar  $p$  implement the hash function  $h_i(x) = ((a_i x + b_i) \bmod p) \bmod w$ . Then, use this hash function on a stream of data to create the sketch matrix.

**Code input and output format.** The function `count_min_sketch` takes in the following arguments:  $\mathbf{a} = [a_1, \dots, a_d]$ ,  $\mathbf{b} = [b_1, \dots, b_d]$  as vectors with positive entries,  $w$  and  $p$  as scalars, and a python generator function, `stream`, that produces a stream of data. The output of the function is the sketch matrix, of size  $d \times w$ .

### Example

For  $d = 2, w = 3$ , given the vectors  $\mathbf{a} = [1, 2]$ ,  $\mathbf{b} = [3, 5]$ ,  $p = 100$ ,  $w = 3$ , we get the following hash functions:  $h_1(x) = ((x+3) \% 100) \% 3$ ,  $h_2(x) = ((2x+5) \% 100) \% 3$ . For the stream of data `[10, 11, 10]` the function `count_min_sketch` should return the following: `[[0, 2, 1], [1, 2, 0]]` which corresponds to the following sketch matrix:

1	0, 2, 1
2	1, 2, 0

Answer: My implementation is submitted to Auto-Grader.

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  $x \in \{1, \dots, m\}$  is its numeric value, and  $c \in \{1, \dots, k\}$  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  $(i + 1) \bmod n$ ).
- 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:

```
1 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:

```
1 cards_game(m=3, k=2, n=3, counts = {
2     1: [(1,2), (2,2), (2,2), (3,2)],
3     2: [(1,1), (2,1)],
4     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.