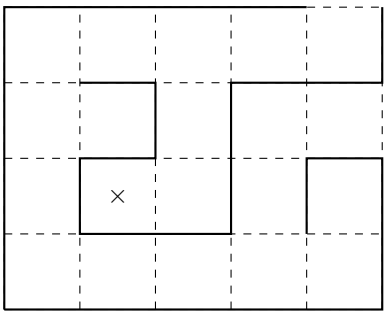
Sample Final Exam Questions

Sample Graph / Greedy Exam Questions

1 Wisconsin Jones (10 points)

1.1. (5 points) As the intrepid computer scientist/archaeologist adventurer Wisconsin Jones, you have discovered the earliest know Bucky Badger icon buried deep beneath Lake Mendota by a long forgotten student society $O\Omega\Theta$ in an elaborate labyrinth. The labyrinth is a grid, where each edge is either a wall or a door, and you must move from square to square by passing through the doors. Due to the instability of this ancient site, you want to recover the icon as fast as possible before it collapses and the waters of Lake Mendota rush in. Moreover, you know that the $O\Omega\Theta$ society have multiple different labyrinths and treasures buried under Lake Mendota so you want to come up with an algorithm that can work for any $n \times m$ labyrinth.

An example of a labyrinth, where dashed lines are doors, solid lines are walls, and $\times$ is the Bucky icon:



(a) (3 points) Give an efficient $O(nm)$ algorithm that finds the fastest route to the treasure given the input of the labyrinth as described. You can assume that the labyrinth data structure allows you to find the entrance cells in constant time, and, for some cell x, you can find cells accessible from x in constant time.

Solution:

- Treat labyrinth as a graph:
  - Assume there is a starting external node that connects to any of the entrance cells (opening in the exterior wall). Each cell will be treated as a node connected to each adjacent cell without a wall.
- Run BFS starting from the outside node.
- Return the path in the BFS traversal to the icon node.

(b) (2 points) Justify the time complexity of your algorithm from Part a.

Solution:

- BFS: $O(|E| + |V|) = O(mn)$.
- Returning the path: $O(mn)$ Once icon is found, returning the path is just traversing up the BFS tree.

Overall $O(mn)$.

1.2. (5 points) The labyrinth has m doors, labelled 1 to m, which are connected to one or more of n levers, labelled 1 to n. All the doors are closed and need to be opened by pulling all of the levers one-by-one. Your goal is to figure out the order of lever pulls to ensure that all doors are open at the end.

The behaviour of the levers is described by an $n \times m$ matrix M. When lever i is pulled, door j behaves as follows:

- If $M_{i,j} = 1$, then door j opens.
- If $M_{i,j} = -1$, then door j closes.
- If $M_{i,j} = 0$, then door j remains at its previous state (open or closed).

Since the $O\Omega\Theta$ society was able to place Bucky in the labyrinth, there is guaranteed to be a sequence of lever pulls to open all the doors. We will construct an order σ in which we pull the levers, i.e. $\sigma[i]$ is the ith lever you pull.

Suppose that there are $m = 5$ doors, and $n = 3$ levers with matrix

$$M = \begin{bmatrix} 0 & 1 & 1 & 1 & 0 \\ 1 & 1 & 1 & 0 & -1 \\ 0 & -1 & 0 & 0 & 1 \end{bmatrix}$$

The optimal order to pull the levers is $\sigma = \langle 2, 3, 1 \rangle$.

We will fill in σ in reverse order. That is, we first determine $\sigma[n]$ (the final lever to be pulled), then $\sigma[n-1]$, and so on. Let $\sigma[i+1, n] = \langle \sigma[i+1], \cdot, \sigma[n] \rangle$ be the last $n - i$ lever pulls. In other words, the $i+1$ to nth lever pulls.

Two of your computer scientist/archaeologist assistants proposed two possible heuristics for choosing the ith level to pull, i.e. how to determine $\sigma[i]$ given $\sigma[i+1, n]$. For each door $j = 1, \ldots, m$, let f_j^{i+1} be the last nonzero value in the sequence $\langle M[\sigma[i+1], j], \ldots, M[\sigma[n], j] \rangle$, or $f_j^{i+1} = 0$ if no such nonzero value exists. Let $D_{i+1} \subseteq \{1, \ldots, m\}$ be the set of doors where $f_j^{i+1} = 1$. Let $C_{i+1} = \{1, \ldots, m\} \setminus D_{i+1}$.

The two proposed heuristics are:

1. Choose the ith lever, i.e. set $\sigma[i]$, to be a lever whose row in M contains no -1s in the columns indexed in C_{i+1}.
2. Choose the ith lever, i.e. set $\sigma[i]$, to be a lever whose row in M contains the most 1s in the columns indexed in C_{i+1}.

Neither assistant completed CS577 so it is up to you figure out which heuristic to use. You'll only have one chance to open the doors so it is critical that the proposed algorithm works.

(a) (1 point) Which heuristic is correct?

Solution: 1

(b) (1 point) Give a counter-example for the other heuristic than indicated in Part a.

Solution:

$$M = \begin{bmatrix} 0 & 1 & 1 & 0 \\ 1 & 1 & 1 & -1 \\ 0 & -1 & 0 & 1 \end{bmatrix}$$

Heuristic 2 would set the 3rd (last) lever pull to be 2 which would close the last door.

(c) (3 points) Prove that the heuristic indicated in Part a is correct.

Solution: Consider an arbitrary door j in the ordering of Heuristic 1. Let S_j be the string of $\{0, 1, -1\}$ described by the ordering from heuristic 1 for the lever pulls 1 to n, and let k be the position of the last 1 in S_j. By definition of heuristic 1, all of the last $k-1$ lever pulls must be 0s. Therefore, j will be open at the end of all the lever pulls.

By reverse induction on the lever pulls, we will show that a sequence of lever pulls as described by Heuristic 1 exists if it is possible to open all the doors.

Base case: n-th pull. If there are no levers without a -1, it is impossible to have all the doors open. As the last lever pull would close at least one door.

Inductive step: k-th step. From the induction hypothesis, C_{k+1} is the set of doors that will not be opened from the lever pulls $k+1$ to n. If there does not exist at least one lever pull that does not contain a -1 for the set of doors C_{k+1}, then at least one door would not get closed contradicting the existence of a sequence of lever pulls that will open all doors. If there exist multiple levers with non -1 entries for C_{k+1}, any such lever would work since none of them close a door in C_{k+1} and the set $C_k \subseteq C_{k+1}$ so any such lever would be valid for earlier pulls.

2 The TP Standard, Part 1 (10 points)

It is the year 2220 and the most valuable commodity in the known universe is a roll of 2-ply toilet paper. It is lost to history exactly why Humanity moved to the TP standard... Only that it has been this way for 100s of years.

You are working in logistics for Royale Bank – a joint Irving-RBC company[1][2]. You are asked to design an order processing algorithm. Given the value of a single roll of toilet paper, shipping costs are negligible compared to a roll. You must ship rolls in bundles of fixed quantities. The goal of your algorithm is to ship the exact quantity ordered using the least number of bundles per order.

2.1. (10 points) Consider the case where Royale Bank ships n bundles sizes $b_1 < b_2 < \cdots < b_n$, where $b_1 = 1$ and $b_{i+1} = c \cdot b_i$ for some integer $c > 1$. That is b_{i+1} is a multiple of b_i.

(a) (1 point) Consider the bundle sizes: $1, 3, 6, 12$. What is the optimal number of bundles sent for an order of 42 rolls of toilet paper? Describe how many of each bundle size.

Solution: 4 – 3 of size 12 and 1 of size 6.

(b) (3 points) Provide a **greedy algorithm** to determine the optimal number of bundles and how many of each size to ship.

Solution:

- Initialize the remainder $r :=$ to the initial shipment quantity.
- Initialize each cell of an array A of size n to 0.
- For each bundle size b_i from b_n to b_1:
  - Let $A[i]$ be r integer division b_i.
  - Set $r = r - A[i] * b_i$.
- The sum of A is the total number of bundles to ship.

(c) (2 points) Prove that your algorithm terminates.

Solution: The algorithm runs through each bundle size exactly once.

[1] Good Canadian humour, eh? – I did it again!

[2] Royale is a toilet paper company owned by Irving (a huge Canadian oil/forestry/other company), and RBC is the Royal Bank of Canada one of the 5 big Canadian Banks.

(d) (3 points) Prove that your algorithm is optimal. Clearly state your approach: stay ahead vs exchange argument.

Solution:

Exchange Argument:

- By definition, a greedy solution is such that, for every b_i, the sum of bundles b_{i-1} to b_1 in the solution is less than b_i since they are calculated from the remainder.
- By the definition of the bundle sizes being integral multiples, the greedy solution is the unique solution with this property.
- Let S be any solution. If there is a set Q of bundles from S for b_{i-1} to b_1 that sum up to b_i. The set Q can be replaced with 1 bundle of b_i, reducing the total number of bundles by $|Q| - 1$. Hence, any solution can be transformed to the greedy solution, implying that greedy is optimal.

(e) (1 point) Prove the running time complexity of your algorithm.

Solution: For each bundle size, a constant number of operations are done. Each bundle is consider exactly once. To get the total a length n array is traversed. Hence, $O(n)$ overall.

Sample Divide and Conquer Questions

3 Political Divide (10 points)

3.1. (10 points) You are in a room with n politicians $p_1, p_2, \ldots, p_n$, where n is a power of 2. Each politician belongs to one of k parties, where $k > 1$. You want to know which politicians are in which party, but it is impolite to directly ask. However, you notice that whenever two politicians that belong to the same party are introduced to one another, they shake hands. When two politicians that belong to different parties are introduced, they stare at each other angrily.

(a) (6 points) Using divide and conquer and the $O(1)$ method `introduce(`p_i, p_j`)` to introduce two politicians p_i and p_j to each other, design an algorithm to group the politicians into their k parties.

Solution:

Base case: For 2 politicians, introduce them to each other. If they shake hands, then they're in the same party; otherwise they're in different parties.

Divide: For $n > 2$ politicians, do a recursive call for the first $n/2$ and a call for the last $n/2$ politicians.

Conquer: Each recursive call returns up to k parties. Compare each party ga from the first call to each party gb from the second call by choosing a representative from each party and introducing them. Combine ga and gb if the representatives shake hands. Return the resulting set of parties. Merging $O(k^2)$.

(b) (3 points) Give the recurrence relation for your algorithm in Part a.

Solution:

$T(n) = 2T(n/2) + O(k^2); T(2) = O(1)$ recurrence

(c) (1 point) Give the asymptotic runtime based on the recurrence given in Part b.

Solution:

$O(n \cdot k^2)$ runtime

4 Wordplay (10 points)

4.1. (10 points) The English word "below" has a curious property. Its letters are all arranged in ascending alphabetical order. We will call this an *alphabetical* word.

(a) (2 points) How quickly can you check whether a given n-letter word is alphabetical? Briefly describe an algorithm for doing so, and state its asymptotic run-time.

Solution: For each letter in the word, check whether it is in alphabetical order with the following letter. This takes $O(n)$ time.

(b) (2 points) Since there are very few alphabetical words in the English language, we are also interested in words that are almost alphabetical. A word is called k-alphabetical if removing k (or fewer) letters would make it an alphabetical word. A friend of yours proposes the following algorithm to determine whether a word is k-alphabetical. In the following algorithms, for a string w, $w[0]$ is the first letter, $w[-1]$ is the last letter:

Algorithm 1: calcAlpha

```
Input  : A word w.
Output: k, where w is k-alphabetical, and w' an ordered string.
if w has length 1 then
    return (0, w)
end
(k_1, w_1) := calcAlpha(Front-half of w)
(k_2, w_2) := calcAlpha(Back-half of w)
k_3 := k_1 + k_2
Initialize w_3 to an empty string
while w_1 or w_2 is not empty do
    if w_1 is empty then
        foreach letter ℓ ∈ w_2 do
            Append to w_3 if ℓ ≥ w_3[-1]; otherwise discard and add 1 to k_3
        end
    else if w_2 is empty then
        Append w_1 to w_3
    else
        if w_1[0] ≤ w_2[0] then
            Remove first character from w_1 and append to w_3.
        else
            Discard first character from w_2 and add 1 to k_3.
        end
    end
end
return (k_3, w_3)
```

Algorithm 2: isAlpha

Input : A word w and a integer k.
Output: true if w is k-alphabetical, false otherwise.
$(k', w') :=$ calcAlpha(w)
return $k' \leq k$

Write a recurrence relation describing the run-time of this algorithm and state the resulting asymptotic run-time.

Solution:

$$T(n) = 2T(n/2) + n$$

$$O(n \log n)$$

(c) (3 points) Unfortunately, your friend's algorithm does not always work. Provide an example of a word and a number k such that the isAlpha reports the word is NOT k-alphabetical when it actually is. (Your word does not need to be a real English word.) Draw the recursion tree to demonstrate your example.

Solution: "zaps" is 1-alphabetical (just remove the 'z').

The algorithm runs:

```
z (0) | a (0) | p (0) | s (0)
    z (1)     |     ps (0)
            z (3)
```

and reports that "zaps" is 3-alphabetical.

(d) (3 points) In the previous section, you showed that your friend's algorithm does not always return the optimal answer. Prove that it only errs in one direction. It never reports a word is k-alphabetical when that word in fact is not k-alphabetical.

Solution: The proof will be by strong induction on the length of the word w.

Base case: w has length 1. calcAlpha correctly returns $(0, w)$, i.e., w is 0-alphabetical.

Induction step: Consider a word w of length n. By the induction hypothesis, (k_1, w_1) and (k_2, w_2) only err in one direction. Namely, that the front half of w is $\leq k_1$-alphabetical, and the back half of w is $\leq k_2$-alphabetical. Moreover both w_1 and w_2 are ordered subsequences of the front half of w and back half of w, respectively.

k_3 is initialized to $k_1 + k_2$. Throughout the execution, k_3 is increased only when a character from w_1 or w_2 is discarded. In addition, the merge process ensures that w_3 is an ordered subsequence by maintaining the relative order. That is, all letters in w_1 are added or dropped prior to the remaining non-dropped letters (if there are any) in w_2 are added to w_3.

Therefore, k_3 counts the number of letters to remove from w to produce an ordered subsequence w_3.

Sample Dynamic Programming Questions

5 World Traveller (10 points)

You're planning on flying around the world. Your plane has a fuel tank of capacity T and can fly 1 kilometre per litre of fuel. You've mapped out your flight plan and it consists of n cities where you can stop to refuel. For each city i, you have two values f_i, the price per litre of fuel in city i, and c_i the cost to stop and refuel. That is, the cost to refuel in city i is $f_i \cdot \ell + c_i$, where ℓ is the number of litres purchased.

For any two sequential cities, i and $i+1$, you know the distance $d_{i,i+1}$ in kilometres which is no more than T by design and, as chance would have it, all $d_{i,i+1}$ are integral.

Before heading out, you decide to calculate the optimal refuelling schedule. That is, calculate how much fuel to purchase at each city given that you start at city 1, have no gas in the tank, and will visit all the cities in order from 1 to n.

5.1. (2 points) Assume that $c_i = 0$ for all cities. Give an optimal greedy algorithm that runs in $O(n \log n)$ time. Note: you do not have to prove that the algorithm is optimal.

Solution: Order the cites from cheapest to most expensive gas. Starting from the cheapest city c, fill the tank up to full or to the remaining distance whichever is smaller. This gas will cover the portion of the trip starting from c. Consider the next cheapest city d. Fill the tank up to the minimum of the tanks capacity and the uncovered distance to the end, or the next cheaper city. The amount of gas purchased p at d will cover the next p uncovered distance after d. Repeat until all the trip is covered.

5.2. (8 points) Assume that $c_i \geq 0$ for all cities, design a dynamic program to calculate the optimal refuelling with a worst-case runtime of $O(nT^2)$.

(a) (2 points) Describe the array/matrix M used in your dynamic program. Be sure to give the dimensions and any cells that can be immediately initialized.

Solution:

- A 2 dimensional array M with the city id on one axis and the other runs from 0 to T, representing amount of gas purchased.
- Cell $M[i][j]$ is the cost to travel to city n starting with j units of fuel at city i.
- $M[n][i] = 0$ for $i = 1$ to T.
- $M[i][j] = \infty$ for $i \leq 0$ or $j < 0$.

(b) (1 point) In which cell of M will the optimal fuel cost located?

Solution: The solution will be found in cell $M[1][0]$.

(c) (2 points) Give the Bellman equation for your dynamic program.

Solution:

$$M[i][j] = \min_{\max\{j, d_{i,i+1}\} \le t \le T} \begin{cases} M[i+1][t - d_{i,i+1}] & \text{if } t = j \\ M[i+1][t - d_{i,i+1}] + c_i + f_i(t-j) & \text{otherwise} \end{cases}$$

(d) (2 points) Give an efficient algorithm to recover the amount of fuel purchased in each city from the array/matrix M populated based on your dynamic program.

Solution:

- Initialize each cell of an array A of size n to 0.
- Initialize $i := 1$ and $j = 0$.
- While $i \le n$:
  - if $M[i][j] = M[i+1][j - d_{i,i+1}]$: set $i = i + 1$.
  - else find $M[i+1][t] = M[i][j] - c_i - f_i(t + d_{i,i+1} - j)$:
    Set $i = i + 1$, $A[i] = t + d_{i,i+1} - j$, and $j = t$.

The amount of fuel purchased at city k is found in $A[k]$.

(e) (1 point) We would say that this dynamic program is:

A. pseudo-polynomial

B. polynomial

6 The TP Standard, Part 2 (10 points)

6.1. (10 points) Consider the more general case where Royale Bank ships n bundles sizes $b_1 < b_2 < \cdots < b_n$, where $b_1 = 1$.

(a) (1 point) Consider the bundle sizes: $1, 2, 6, 9$. What is the optimal number of bundles sent for an order of 12 rolls of toilet paper? Describe how many of each bundle size.

Solution: 2 – 2 of size 6.

(b) (1 point) Again, consider the bundle sizes: $1, 2, 6, 9$. What is the greedy solution for an order of 12 rolls of toilet paper? Describe how many of each bundle size. (Note it should not be optimal!)

Solution: 3 – 1 of size 9, 1 of size 2, 1 of size 1.

(c) (8 points) Let $T \geq 1$ be the total number of rolls of toilet paper ordered. Design a dynamic program to calculate the optimal number of bundles to send with a run-time complexity of $O(nT)$.

i. (2 points) Describe the array/matrix used in your dynamic program. Be sure to give the dimensions and any cells that can be immediately initialized.

Solution:

- A 2 dimensional array M with the bundle size id on one axis and the other runs from 0 to T, representing roll order sizes up to T.
- Cell $M[i][j]$ is the minimum number of bundles to send j rolls when considering the first i bundles.
- $M[i][0] = 0$ for $i = 1$ to n.
- $M[1][j]$ for $j = 1$ to T can be set to j since $b_1 = 1$.

ii. (1 point) In which cell will the optimal solution be located?

Solution: The solution will be found in cell $M[n][T]$.

iii. (2 points) Give the Bellman equation for your dynamic program.

Solution:

$$M[i][j] = \min(M[i-1][j], f(i,j) \cdot (M[i][j - b_i] + 1)) \ ,$$

where $f(i,j) = \begin{cases} 1 & \text{if } j - b_i \geq 0, \text{ and} \\ \infty & \text{otherwise.} \end{cases}$

iv. (2 points) Describe how to recover the number of each bundle size to send for the optimal solution.

Solution:

- Initialize each cell of an array A of size n to 0.
- Initialize $i := n$ and $j = T$.
- While $i \geq 1$ and $j \geq 1$:
  - if $M[i][j] = M[i-1][j]$: set $i = i - 1$.
  - else if $M[i][j] = f(i,j) \cdot (M[i][j - b_i] + 1)$: Increment $A[i]$ by 1 and set $j := j - b_i$.

The number of bundle size k is found in $A[k]$.

v. (1 point) We would say that this dynamic program is:

A. pseudo-polynomial

B. polynomial

Sample Network Flow Questions

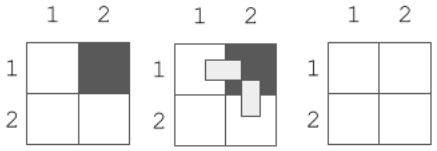
7 Garden Addition (10 points)

7.1. (10 points) The University of Wisconsin Botanical Garden is hiring you to help plan their new addition. It will contain n^2 individual garden tiles, arranged in a square. Each of these gardens will be connected to the horizontally and vertically adjacent gardens, so that visitors can easily traverse them. However, the ground that they want to build on has many hills, so some gardens are at low height and some are at high height. The heights are provided in an array $H[i, j]$, where $H[i, j] = 1$ if garden (i, j) is at high height, and 0 if it is at low height. If a garden is at a different height than an adjacent garden, they will need to add a ramp between those two gardens to ensure that they are connected. Each ramp costs R dollars. They are also capable of raising or lowering individual gardens (changing the height from low to high or from high to low), at a cost of W dollars each, in order to reduce the number of ramps needed. Your job is to decide which gardens to raise/lower (if any) in order to minimize the cost of building the new addition.

In the following example:

$$n = 2,\ R = 2,\ W = 3, \text{ and } H = \begin{bmatrix} 0 & 1 \\ 0 & 0 \end{bmatrix}.$$

If we do not change the heights of any gardens, then we will need to build one ramp from garden $(1, 1)$ to garden $(1, 2)$ and another ramp from garden $(2, 2)$ to garden $(1, 2)$, costing $2R = 4$. Otherwise, if we lower garden $(1, 2)$, as shown on the right, then we will not need to build any ramps, so the total cost will be $W = 3$, which is the optimal cost for this instance.



(a) (3 points) We will solve this problem using network flow. For this part, you only need to describe a graph that has a maximum flow value equal to the optimal cost. Give a precise description of all nodes, edges, and capacities. Do not use lower bounds or node demands.

Solution:
Create one node $v_{i,j}$ for each garden (i, j), as well as a source node s and a sink node t. Connect all adjacent garden nodes via two directed edges (one going each direction) each with capacity R. Create edges $s \to v_{i,j}$ for all gardens (i, j) that are at low height ($H[i, j] = 0$) with capacity W and edges $v_{i,j} \to t$ for all gardens (i, j) that are at high height ($H[i, j] = 1$) also with capacity W.

(b) (4 points) Justify why the maximum flow in the graph you created has value equal to the minimum cost.

Solution:
The maximum flow is equal in value to the minimum (s,t) cut. (s,t)-cuts $(S, V \setminus S)$ are in one-to-one correspondence with decisions of garden heights, by letting S be the low height gardens and $V \setminus S$ be the high height gardens. For a cut $(S, V \setminus S)$, $(s, v_{i,j})$ crosses the cut if and only if $v_{i,j} \in V \setminus S$, and likewise $(v_{i,j}, t)$ crosses the cut if and only if $v_{i,j} \in S$, and by construction this is if and only if the height of garden i, j has been changed (contributing W per each such edge to the value of the cut). Edges of the form $(v_{i,j}, v_{i',j'})$ cross the cut if and only if $v_{i,j} \in S$ and $v_{i',j'} \in V \setminus S$, and this is if and only if the (adjacent) gardens $(i,j), (i',j')$ are at different heights (contributing R per such edge to the value of the cut). Thus a cut has value equal to the cost of the associated garden height decision.

(c) (3 points) Give an algorithm (using the graph you constructed in part (a)) which returns which gardens should have their heights changed in order to minimize the total construction cost. You should return an $n \times n$ array A where $A[i,j] = 1$ if garden (i,j) needs to have its height changed and 0 otherwise.

Solution:
Find the s,t min-cut $(S, V \setminus S)$ in the graph. After removing the nodes s and t, the low height nodes are in S and the high height nodes are in $V \setminus S$. Thus, for each node, $A[i,j] = 1$ if and only if we have changed the height of garden (i,j), meaning $H[i,j] = 0$ and $v_{i,j} \in V \setminus S$ or $H[i,j] = 1$ and $v_{i,j} \in S$.

8 The Grocery Store (10 points)

8.1. (10 points) With the social distancing guidelines, some grocery stores are labelling aisle with a direction only allowing shoppers to "flow" down the aisle one way. Signs throughout the store also remind shoppers to stay 6 feet apart at all times.

A grocery chain would like your help figuring out their stores' capacities during the crisis. For each store, aisles end at an entrance, an exit, or an intersection (with other aisles). The chain can provide you with an estimate of how many people can pass down each aisle while observing social distancing, but they're not sure how to calculate the overall store capacity from this data.

(a) (4 points) Describe a flow network that can be solved using the Ford-Fulkerson method to determine how many shoppers can pass through the store with a single entrance, and a single exit in one hour (while observing social distancing requirements).

Solution: Create a node for each aisle intersection. Each aisle can be represented by an edge indicating the direction of the aisle and the number of people that can move through it in an hour.

(b) (3 points) Of course, it is not enough for shoppers to simply pass from the entrance to the exit. At least one unique shopper must pass through each aisle **intersection** during an hour. Otherwise, the featured products can't be sold.

How would you alter the graph from Part 8.1a to calculate whether the suggested number of shoppers is feasible given these additional requirements?

Solution: Transform each aisle intersection node i in the graph into a pair of nodes i_1, i_2. All incoming edges to i are redirected to i_1. All outgoing edges from i come from i_2 instead. Add an edge $i_1 \to i_2$ with capacity ∞ and lower bound 1. Add an edge from the store exit node to the store entrance node with infinite capacity and lower bound equal to the number of aisle intersection nodes. The demand on all the nodes is 0. Then ask whether a circulation is feasible.

(c) (3 points) The store manager thinks they understand the Ford-Fulkerson method, but only in a graph with a single source, single sink, capacity values for edges, and no node properties at all. Describe how to modify your solution from Part 8.1b so that an instance can be solved using only the basic Ford-Fulkerson method.

Solution: Augment the graph with a super-sink node t and a super-source node s. For each edge $i_1 \to i_2$ in the graph with a minimum flow value of x, remove the minimum flow value from the edge. Add an edge $i_1 \to t$ with capacity x, and add an edge $s \to i_2$ with capacity x.

Sample Intractability Questions

9 Speed Cams (10 points)

In your city, there has been a rash of traffic accidents due to speeding. You have been tasked by the city to help distribute speed cameras. The city would like to purchase high-tech speed cameras that, when place at an intersection, can monitor all the incident roads. The city council would like to know what is the minimum number of cameras needed to monitor all the roads, where the city has q intersections and r distinct road segments, a portion of a road between two intersections.

9.1. (1 point) Give a brute-force algorithm to determine the minimum number of cameras?

Solution: Check every subset of intersections from smallest to largest. Stop at the first subset that covers all road segments.

9.2. (1 point) What is the worst-case runtime of your brute-force algorithm?

Solution: $O(r2^q)$

9.3. (1 point) Re-state the camera placement optimization problem as a decision problem.

Solution: Given q intersections and r road segments, is it possible to monitor all road segments with k or fewer cameras?

9.4. (7 points) Prove that the decision version of the camera placement problem is in NP-Complete.

Solution:

In NP:

- Certificate: A subset of intersections.
- For each road segment, check if one end is in the subset of intersections.

NP-Complete:

- Reduce from vertex cover (VC).
- Consider an arbitrary instance of VC. Let $G = (V, E)$ be the graph.
  - For each node v, add an intersection v.
  - For each edge (u, v), add a road segment between u and v.
  - k in VC is k in speed cams.
- $\Rightarrow$: If there is a vertex cover of size at most k in G, this corresponds to a set of at most k intersections in the speed cam instance that cover all the intersections.
- $\Leftarrow$: Any a set of at most k intersections that cover all the road segments, then this directly corresponds to vertex cover of at most k in the original graph G.

10 Office Hour Assignment Problem (10 points)

As you are passing by the room CS 1304, you may have noticed that sometimes, there are too many TAs in the room. To make sure that there are not too many TAs in the room at each time, but also that there is always someone, we want to divide the TAs into two groups: ones that hold office hours and ones that do not. But not everyone is available at every moment, so each TAs get to sign up on a timeslot, to indicate that they are available to hold office hour during that time.

Formally, we define the problem OfficeHourAssignmentProblem (OHAP): Suppose we are given n TAs and m timeslots, in which each timeslot has a list of TAs that can hold office hour during that time. Can we choose a subset of TAs to assign to office hour duty, such that there is **exactly one** TA per timeslot?

Example input:
TAs: t_1, t_2, t_3, t_4, t_5
Timeslot 1: (t_1, t_2, t_4)
Timeslot 2: (t_2, t_3)
Timeslot 3: (t_1, t_2, t_4, t_5)

10.1. (2 points) Provide a valid solution for the above example.

Solution: We can choose t_1, t_3.

10.2. (8 points) After reflecting on the problem, Marc thinks that OHAP is NP-complete. Show that OHAP is NP-complete.

(a) (2 points) Show that OHAP is in NP.

Solution: Let the certificate be a subset S of TAs. We can verify by going through each timeslot c_i and verifying that $|c_i \cap S| = 1$. This runs in time $O(nm|S|)$, since for each of the $O(n)$ TAs in each of the m clauses, we can naïvely compare this TA with all TAs in S.

(b) (6 points) Show that OHAP is NP-Hard.

Solution: We can reduce from 3Coloring. Suppose we have a graph $G = (V, E)$. For each vertex v, we create three TAs, v_r, v_b, v_g and a timeslot $c_v = (v_r, v_b, v_g)$. For each edge $e = (u, v)$ we create three TAs, e_r, e_b, e_g and three timeslots, $c_{e,r} = (v_r, u_r, e_r)$, $c_{e,b} = (v_b, u_b, e_b)$, and $c_{e,g} = (v_g, u_g, e_g)$. Let (T, C) be the resulting TAs and timeslots.

$\Rightarrow$: Suppose $G \in$ 3Coloring. Then, there is a color assignment $f : V \to \{r, b, g\}$, such that $f(u) \neq f(v)$ if $(u, v) \in E$. Then, we can first choose a subset of TAs by choosing $v_{f(v)}$ for each vertex v. Then, we can see that c_v has exactly one TA assigned to office hours. For each edge $e = (u, v)$, since $f(u) \neq f(v)$, we can see that $c_{e,i}$ has at most one TA assigned. Since there will still be one color i such that $c_{e,i}$ has no TA assigned, we can add the TA e_i additionally in our subset. This resulting set will satisfy the requirements, so $(T, C) \in$ OHAP.

$\Leftarrow$: Suppose $(T, C) \in$ OHAP. So, there is $S \subset T$ such that $S \cap c = 1$ for each $c \in C$. This gives us a coloring, since for each vertex v, we have $c_v \cap S = 1$, so exactly one $v_i \in S$ for $i \in \{r, g, b\}$. Let f be the coloring assigning v to the unique color i such that $v_i \in S$. This is a valid coloring, since we cannot have $f(u) = f(v)$ for $e = (u, v) \in E$. If $f(u) = f(v) = i$, then $c_{e,i} = (v_i, u_i, e_i) \cap S$ must contain both v_i and u_i, contrary to our assumption. Therefore, $G \in$ 3Coloring.

10.3. (3 points (bonus)) After reflecting more, Marc has an idea that maybe there is a way to make this problem tractable. What can you say if there is a restriction that all timeslots have exactly 2 TAs signed up? What about 3?

Solution: If the timeslots are all of length 3, the above reduction still works, so it is still NP-Hard.

If all the timeslots have length 2, then we can solve it in polynomial time. We can view the problem as a graph 2 coloring, in which the TAs are vertices, and timeslots are the edges. Choosing a subset of TAs such that exactly one is chosen per timeslot is same as assigning a color to the vertices such that each neighboring vertices have distinct color. Since graph 2 coloring can be solved in polyomial time by greedily choosing the color of each vertex, the restriction of OHAP with timeslot length 2 is in P.