This document gives model solutions to the assignment problems. Note that alternative solutions may exist.

Question 1

A demolition company has been tasked with taking down n buildings, whose initial heights are positive integers given in an array $H[1..n]$. The company has two tools at its disposal:

- a small explosive, which reduces the height of a single building by 1, and
- a wrecking ball, which reduces the height of all buildings by 1.

The company has an unlimited supply of the small explosives, which are free to use. However, the wrecking ball costs \$1000 each time it is used.

The company initially has \$1000 dollars. They also get \$1000 whenever they finish the demolition of a building *with the wrecking ball* (reducing its height from 1 to 0), but they receive no money if the demolition of a building is finished using an explosive.

For example, for four buildings of heights $[2, 3, 5, 1]$, the following sequence of actions demolishes all buildings.

- Use a small explosive on building 1, resulting in $[1, 3, 5, 1]$.
- Spend the \$1000 to use the wrecking ball, resulting in heights $[0, 2, 4, 0]$. Buildings 1 and 4 were demolished, so the company receives \$2000.
- Use the wrecking ball again, leaving heights $[0, 1, 3, 0]$. No buildings were demolished, so no money is gained.
- Use an explosive to demolish building 2, giving $[0, 0, 3, 0]$. No money is awarded for completing the demolition of building 2, since the wrecking ball was not used.
- Use explosives three times to demolish building 3.

A sequence of actions is *minimal* if it demolishes all the buildings *and* there is no shorter sequence of actions that does this. The sequence above is *not* minimal.

1.1 [2 marks] Find a minimal sequence of actions to demolish all four buildings in the example above, where the heights are 2, 3, 5 and 1. You must provide reasoning for why your sequence is minimal.

Explode building 3 once, then use the wrecking ball four times. Each use of the wrecking ball finishes a building, so the company always gets reimbursed. The demolition cannot be done in fewer than five actions as the third building has initial height of 5.

There are alternatives, such as using the wrecking ball three times followed by the explosive twice on building 3.

1.2 [6 marks] Show that there is always a minimal sequence of actions that results in the buildings being demolished where all uses of the explosive occur before any uses of the wrecking ball.

Consider a minimal sequence in which this is *not* true. Then the wrecking ball must be used immediately before the explosive at some point. The effect of swapping these actions is:

- For the building affected by the explosive:
  - it can't have had height 1 before these operations, because then the explosive

would do nothing and therefore the sequence was not minimal (we could omit it for a shorter sequence)

  - if it had height 2, we now gain \$1000 where we previously got nothing, and
  - if it had height > 2, there is no effect.

- For all other buildings, there is no effect.

Therefore swapping these actions yields a sequence which still demolishes all buildings and takes the same number of operations.

Swapping all instances of the wrecking ball immediately followed by an explosive will eventually leave all uses of the explosive before any uses of the wrecking ball.

1.3 **[4 marks]** Identify a criterion for whether there is a minimal sequence of actions to demolish all buildings using *only* the wrecking ball, with reasoning to support it.

Suppose that array $H[1..n]$ is sorted in increasing order to form $H'[1..n]$. Then the criterion is that for all $1 \leq i \leq n$, we must have $H'[i] \leq i$.

If this was true, the first demolition will demolish at least one building. This replenishes the funds, but also guarantees that for the new subarray $H'[2..n]$, the criterion holds again after deducting one from all entries. Continuing thus, we demolish all buildings using only the wrecking ball.

If this wasn't true, there must exist some i such that $H'[i] > i$. Then we need to swing the wrecking ball more than i times to demolish that building, costing more than i thousand dollars. However, the first i swings completely demolish at most the $i - 1$ shorter buildings, so the company receives at most $i - 1$ thousand dollars in addition to the starting \$1000. Therefore the company is unable to demolish building i, so it is impossible.

1.4 **[8 marks]** Design an algorithm that runs in $O(n \log n)$ time and determines a minimal sequence of actions to reduce each building's height to 0.

Algorithm: From 1.2, we know that there is a solution using only explosives up to some point and only wrecking balls thereafter. We will restrict ourselves to finding a solution of this form only. Furthermore, from 1.3, we know that the demolition requires only the wrecking ball once the ith shortest building has height at most i for all $1 \leq i \leq n$.

- We first sort the buildings in increasing order with merge sort to yield an array H', while also tracking the original indices.[a]
- Next, we iterate through array H' and reduce the ith shortest building with an explosive $max(0, H'[i] - i)$ times, so that its height becomes less than or equal to i.
- Finally, we can use the wrecking ball to demolish all buildings. The number of swings of the wrecking ball is just the height of the tallest remaining building: $H'[n]$.

Correctness: We first note that the total height reduction is fixed at $H[1] + \ldots + H[n]$. A swing of the wrecking ball achieves height reduction equal to the number of buildings still standing, while an explosive only yields 1 unit of height reduction. Therefore, a sequence with more uses of the wrecking ball must use fewer or the same number of actions as a sequence with fewer of them. It follows that there exists a minimal sequence with the largest possible number of uses of the wrecking ball. Our greedy approach uses the wrecking ball $\max(n, H'[n])$ times, which is maximal because:

- the wrecking ball cannot be used more than n times due to the money constraint, and

- the wrecking ball also cannot be used more times than the height of the largest building, as there would be nothing left to demolish in the final swing.

All that remains to check is that our solution minimises the use of explosives. In this first stage, we had to reduce the height array H in order to satisfy the criterion from 1.3. We achieved this by reducing the sorted height array H' to be elementwise at most $[1, \ldots, n]$. Suppose instead we reduced each $H'[i]$ to be at most $\sigma(i)$, where σ is another permutation of $1..n$. Then σ must have an adjacent inversion, i.e. an index j such that $\sigma(j) > \sigma(j+1)$. Reducing the jth and $(j+1)$th shortest buildings to these heights requires

$$\max(0, H'[j] - \sigma(j)) + \max(0, H'[j+1] - \sigma(j+1)) \tag{0.1}$$

explosives, whereas swapping the final heights requires

$$\max(0, H'[j] - \sigma(j+1)) + \max(0, H'[j+1] - \sigma(j)) \tag{0.2}$$

explosives. This is equal if $H'[j] \geq \sigma(j)$ or $H'[j+1] \leq \sigma(j+1)$, but otherwise the swap saves explosives.[b] Therefore, we use the fewest explosives possible, completing the proof that the greedy is optimal.

Time Complexity:

- Applying merge sort to array H takes $O(n \log n)$ time.
- Determining the number of explosives used on each building takes constant time, as we simply calculate $H'[i] - i$ and associate it to the original index of the building. Therefore the first stage takes a total of $O(n)$ time.
- The second stage involves using the wrecking ball only. The number of wrecking ball uses is the value of $H'[n]$ after explosives (or equivalently the maximum of the original $H'[n]$ value and n), which is accessed in $O(1)$.

Hence the total time complexity is $O(n \log n)$.

[a] This can be done by sorting an array of pairs $(H[i], i)$ by first coordinate, which only adds a constant factor to the time complexity.

[b] This can be confirmed by taking all cases of the $\max(0, \cdot)$ expressions in (0.2).

Question 2

You run a jewellery shop and have recently been inundated with orders. In an attempt to satisfy everyone, you have rummaged through your supplies, and have found m pieces of jewellery, each with an associated price. To make things easier, you have ordered them in increasing order of their price. You now need to find a way to allocate these items to $n \leq m$ customers, while minimising the number of customers who walk away with nothing.

2.1 **[8 marks]** Each customer has a minimum price they will pay, since they all want to impress their significant others. Of course, you need to decide who gets what quickly, or else everyone will just leave to find another store.

Given an array $P[1..m]$ of jewellery prices, sorted in ascending order, and an array $M[1..n]$ of customers' minimum prices, design an algorithm which runs in $O(n \log n + m)$ time and allocates items to as many customers as possible, such that the price of the item given to customer i is at least $M[i]$, if they are given an item at all.

For example, suppose the store has $m = 5$ pieces of jewellery with prices $P = [5, 10, 15, 20, 25]$, and the minimum prices of the $n = 3$ customers are $M = [21, 15, 31]$. In this case, the first customer can get the \$25 item, and the second customer can get the \$15 item or the \$20 item.

However, the third customer will walk away as the store does not have any jewellery that is priced at \$31 or higher.

Algorithm: We begin by sorting M into ascending order. We will loop through the customers in order, and assign each customer the cheapest available item that they will accept, this can be done with a two pointer approach:

- Initialise i and j to 1.
- While $j \leq n$ and $i \leq m$
  - If customer j will accept item i ($M[j] \geq P[i]$), then allocate item i to customer j and increment both i and j.
  - If customer j will not accept the item, then increment i.

Correctness: We will prove by induction that the greedy algorithm produces an allocation of optimal size with as many high cost items unallocated as possible.

Our base case is $n = 1$. In this case, the greedy algorithm will check every item and allocate the first possible one to the customer. Clearly the maximum allocation here is 1 and the algorithm leaves the most expensive possible items left over.

Now, we assume that the algorithm produces an optimal allocation for $M[1..n-1]$ and $P[1..m]$ and leaves the most expensive possible items unallocated, where M has already been sorted (so that $M[n] > M[i]$ for all $i < n$). We have two cases to consider:

Case 1: Only c of the first $n-1$ customers have been allocated items, for some $c < n-1$. As the greedy algorithm allocates items to customers in order, this means that there were no available items expensive enough for customers $c+1, \ldots, n-1$. Since $M[n] > M[n-1]$, this means there are also no unallocated items that are expensive enough for customer n, and so the previous allocation is still optimal.

Case 2: All of the first $n-1$ customers have been allocated items. In this case, the greedy algorithm will proceed to allocate the cheapest possible acceptable item to customer n if such an allocation is possible. If customer n will not accept any unallocated items, then there exists no other allocation for the first $n-1$ customers in which there will be a left over item that they will accept, as the most expensive possible items are still available by the inductive assumption. If the customer does accept an unallocated item, then the greedy algorithm assigns the cheapest possible item, maintaining the inductive property as required.

It follows by induction that the algorithm produces an optimal solution for all integers n, m.

Time complexity: Sorting M takes $O(n \log n)$ time, and the two pointer approach takes $O(n+m)$ time in the worst case for a total time complexity of $O(n \log n + m)$.

As per the announcement, questions 2.2 and 2.3 are collectively worth **[12 marks]**

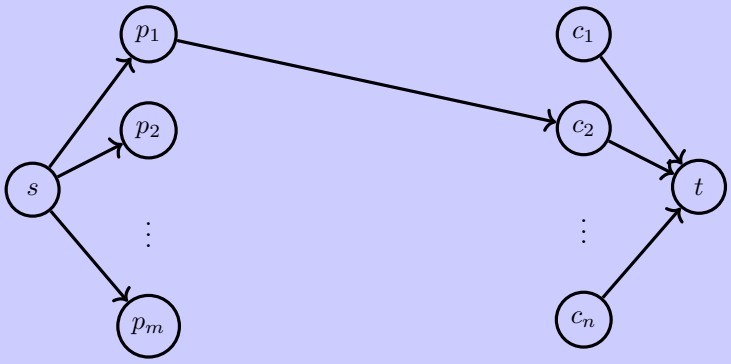
2.2 With a sigh of relief, and an allocation set up, you go to ring up the first customer's item, just to find out that they can't afford it! In a panic, you get everyone to give you their budgets, and go back to the drawing board.

Given an array $P[1..m]$ of jewellery prices, sorted in ascending order, and array $M[1..n]$ and $B[1..n]$ of customers' minimum prices and budgets, design an algorithm which runs in $O(n^2 m)$ time and allocates items to as many customers as possible, such that the price of the item given to customer i is at least $M[i]$ and doesn't exceed $B[i]$, if they are given an item at all.

For example, the store has $m = 5$ pieces of jewellery with prices $P = [5, 10, 15, 20, 25]$. The minimum prices of the $n = 3$ customers are $M = [21, 16, 31]$, and their budgets are $B = [26, 19, 38]$. In this case, the first customer can get the \$25 item, but the second customer has to walk away as he refuses the \$5, \$10 and \$15 items but cannot afford the \$20 and \$25 items. The third customer will also walk away as the store does not have any jewellery that is priced at \$31 or higher.

This is a standard maximum bipartite matching problem, where we match items to customers. We construct the flow network as follows:

- For each item i, construct a vertex p_i.
- For each customer j, construct a vertex c_j.
- For each item i and customer j, construct an edge of capacity 1 from p_i to c_j if $P[i] \geq M[j]$ and $P[i] \leq B[j]$.
- Construct a vertex s and a vertex t to represent the source and sink respectively.
- For each item i, construct an edge of capacity 1 from s to p_i.
- For each customer j, construct an edge of capacity 1 from c_j to t.



This sets up the corresponding flow network. We can now run Edmonds-Karp on the flow network and the corresponding maximum flow L is the maximum number of customer-item pairs. Flow through an edge from p_i to c_j indicates that customer j should purchase item i. Since edges from the source to each item and from the customers to the sink all have capacity 1, each item can only be purchased once and each customer can only purchase a single item.

To get the actual assignment, we can retrieve final residual graph from running Edmonds-Karp. Within the final residual graph, we assign item i to customer j if and only if the edge from p_i to c_j carries flow.

To show that such an algorithm runs in $O(n^2m)$, first note that Edmonds-Karp runs in $O\left(\min\{E \cdot L, V \cdot E^2\}\right)$ time. Now, $L \leq n$, as the total flow exiting the customer nodes is at most n. We also see that the number of edges $|E| \leq n + m + nm$ since at most every vertex p_i can be connected to every vertex c_j. Thus, $E \cdot L \leq n\,(n + m + nm) = O(n^2m)$. On the other hand, $V \cdot E^2 = (n + m + 2) \cdot (n + m + nm)^2$. Clearly, the time complexity is $O(E \cdot L) = O(n^2m)$.

If we use *Ford-Fulkerson* instead of Edmonds-Karp, we will obtain the same time complexity because the breadth-first search implementation of the augmenting path does not yield a better time complexity since the flow is bounded by n. Thus, using Ford-Fulkerson is just as good.

There is also a greedy approach, that can run in $O(mn)$. For each piece of jewellery i from lowest to highest price, and for each customer j, find a customer who:

- has yet to be given a piece of jewellery;
- has an allowable minimum price (that is, $M[j] \leq P[i]$); and
- has the smallest possible budget that can afford the item.

If there is such a customer, allocate them the piece of jewellery. Otherwise, no customer will get that piece.

For each of the n piece of jewellery, we run an $O(m)$ search for a suitable customer, so this comes to $O(nm)$ time overall.

To prove this is correct, suppose an alternative solution differs from our solution. Then at some point, our algorithm must differ. Let's suppose our solution matches the alternative solution up until item i. There are 3 cases:

Case 1: Both solutions have allocated the item, but to different customers. Suppose our allocation gave item i to customer j and the alternative gave it to customer k. This means that $M[j] \leq P[i] \leq B[j]$ **and** $M[k] \leq P[i] \leq B[k]$. Further, since our solution allocates items to the customer with the least budget, and the solutions match until this point, we must have that $B[j] \leq B[k]$. Now, we have two cases:

[A] If the alternative solution does not allocate any item to customer j, then we can allocate that customer item i instead of customer k, so the solutions match to i, and the number of allocated items has not decreased.

[B] Otherwise, customer j has been allocated some other item, say i'. Since the solutions match to item i, this means $i' > i$, so that $P[i] \leq P[i']$. Further, since we allocated it to customer j, we have $M[j] \leq P[i'] \leq B[j]$. Now, we can use these inequalities to establish that $M[k] \leq P[i] \leq P[i']$, and that $P[i'] \leq B[j] \leq B[k]$. Thus, we can swap the allocation of item i and i' so that the two solutions match to i, and the number of allocated items has not decreased.

Case 2: Our solution has allocated item i to customer j, but the alternative has allocated it to no customer. We can simply allocate it to customer j in the alternative solution, replacing whatever item (if any) customer j was allocated. The number of allocated items cannot decrease from this.

Case 3: Our solution does not allocate item i to any customer, but the alternative solution allocates it to a customer. This is impossible, since we would only not allocate an item if no customer is able to take it, and since the two solutions match until item i, the "pool" of available customers must contain customer j.

So, we have a way of exchanging items around in the alternative solution to match our solution closer without decreasing the number of allocated items. With this measure, we can conclude that our solution is no worse than any alternative solution, and is thus globally optimal.

2.3 There are only $k < m$ days until Valentine's day, and all of your customers need to get their orders before then. Unfortunately, you are only able to process a total of five orders per day. To make matters worse, each customer is only available to pick up their orders on some of the days.

Given an array $P[1..m]$ of jewellery prices, sorted in ascending order, arrays $M[1..n]$ and $B[1..n]$ of customers' minimum prices and budgets, and an array $F[1..n][1..k]$ where

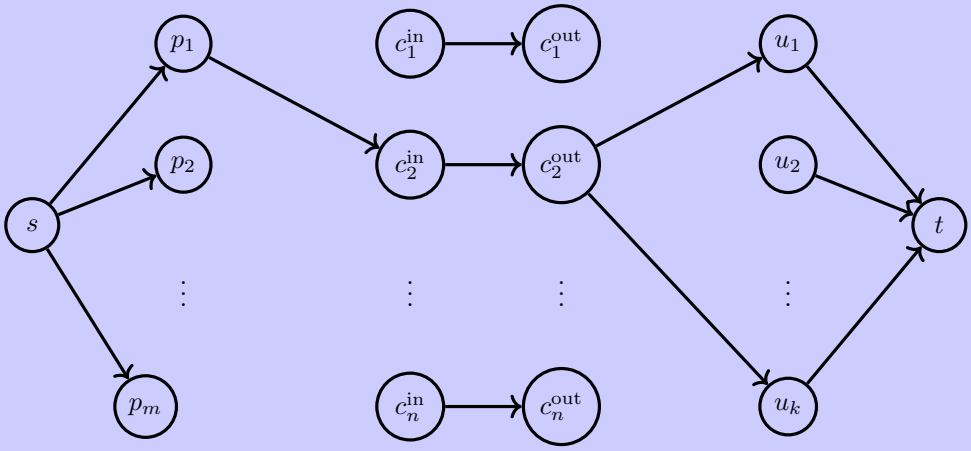
$$F[i][j] = \begin{cases} 1 & \text{if customer } i \text{ is free on day j} \\ 0 & \text{otherwise} \end{cases}$$

design an algorithm which runs in $O(n^2m)$ and allocates items and assigns collection days to as many customers as possible, such that the price of the item given to customer i is at least $M[i]$ and doesn't exceed $B[i]$ if they are given an item at all, and they are free on their assigned collection day.

For example, the store has $m = 5$ pieces of jewellery with prices $P = [5, 10, 15, 20, 25]$. The minimum prices of the $n = 3$ customers are $M = [21, 16, 31]$ and their budgets are $B = [26, 19, 38]$. The array $F[1..n][1..k]$ has entries $F[1][1], F[2][1], F[2][2]$ and $F[3][2]$ equal to 1, and everything else set to 0, representing customers 1 and 2 free on day 1, and customers 2 and 3 free on day 2. In this case, the number of purchases is less than 5 on both days, so for the same reasons described in 2.2, only the first customer can get jewellery.

We slightly modify the bipartite graph from 2.2 by adding an extra column to form a tripartite graph. Construct the flow network as follows:

- Construct a source s and sink t.
- For each item i, construct a vertex p_i.
- For each customer j, construct a pair of vertices c_j^{in} and c_j^{out}.
- For each day d, construct a vertex u_d.
- For each item i, construct an edge of capacity 1 from s to p_i.
- For each item i and customer j pair, construct an edge of capacity 1 from p_i to c_j^{in} if customer j can buy item i, i.e. if $M[j] \leq P[i] \leq B[j]$.
- For each customer j, construct an edge of capacity 1 from c_j^{in} to c_j^{out}.
- For each customer j and day d pair, construct an edge of capacity 1 from c_j^{out} to u_d if customer j is free on day d, i.e. if $F[j][d] = 1$.
- For each day d, construct an edge of capacity 5 from u_d to t.



We can now run Edmonds-Karp (or Ford-Fulkerson) on the flow network and the corresponding maximum flow L' is the maximum number of item-customer-time triplets, where a flow path from p_i to c_j to u_d indicates that customer j should purchase item i and collect it on day d.

To get the actual assignment we can retrieve the final residual graph from running Edmonds-Karp. With the final residual graph, we assign item i to customer j if and only if the edge connected p_i to c_j exists in the graph. We also tell customer j to collect the item on day d if and only if the edge connecting c_j to u_d exists in the graph.

As with the previous question, the algorithm runs in $O(n^2m)$. The total edges in the flow network $|E| \leq n + m + mn + nk + k$ as there is at most 1 additional edge connecting each customer to each day. We also know that the flow $L' \leq n$ as each of the n customer vertices has a capacity of 1. Since $k < m$, it follows that $E \cdot L \leq n\,(n + m + k + nm + nk) \leq n\,(n + 2m + 2nm) = O(n^2m)$, and so the time complexity of our algorithm is $O(n^2m)$.

Question 3

You are studying the ancient Antonise language, and are trying to create some kind of alphabet for it. You have scoured the texts and figured out that there are g many glyphs in this language, but you don't know the order of the alphabet, and want to figure out what it might be. Thankfully, you have found a tome with n unique names, all conveniently k glyphs long, which you suspect are in alphabetical order, and hope to find a way to rearrange the glyphs into an alphabet consistent with the order of the names. For example, suppose your tome contained the following $n = 3$ names, consisting of $k = 3$ glyphs each, from $g = 5$ possible glyphs:

Then both ▼✤❁✯✺ and ▼✤❁✺✯ would be possible alphabets, as the names are in alphabetical order with respect to either of these. However, ✤▼❁✯✺ would not, as the names ✤▼❁ and ✤✤▼ would be out of order with respect to any alphabet where ✤ comes before ▼.

3.1 **[4 marks]** Provide a small example of a tome of at most five names, each of length at most four glyphs, which are not in alphabetical order regardless of how you rearrange the glyphs to form an alphabet.

You must list the names in the order they are found in the tome, and briefly explain why they cannot be in alphabetical order, no matter what order the glyphs are arranged into an alphabet. You may use any symbols for the glyphs in this example, such as the dingbats used in the question, English letters, or numbers.

A small example would be three names in the order ✤❁, ❁✤, ✤✤. The first two names require ✤ to come before ❁ in the alphabet, but the last two names force ❁ to come first, so no alphabet can satisfy both at once.

3.2 **[16 marks]** You are given the number of glyphs g, the number of names n, the number of glyphs in each name k, and a two-dimensional array $S[1..n][1..k]$ containing the numbers 1 through g, where $S[i]$ is an array containing the i^{th} name, and $S[i][j]$ is the index of the j^{th} glyph in that name. The original indexing is arbitary, and the names may not be in alphabetical order with respect to this indexing.

Design an algorithm which runs in $O(nk + g)$ time and finds an alphabet (i.e. reindexes the glyphs) so that the names in S are in alphabetical order if this is possible, or otherwise determines that no such alphabet exists.

An algorithm which runs in $\Theta(n^2k + g)$ time will be eligible for up to 12 marks.

Notation: For each pair $(S[i][1..k], S[i+1][1..k])$ of adjacent names, let the first column index in which they differ be denoted by d_i. Note that these are the positions in the names that determine alphabetical ordering (the first letter in which two adjacent words differ).

Algorithm: Begin by creating a directed graph G with g vertices numbered from 1 to g. We iterate through the $n-1$ pairs of names in S, and for each pair add an edge to G from $S[i][d_i]$ to $S[i+1][d_i]$. This edge indicates that glyph $S[i][d_i]$ must come before glyph $S[i+1][d_i]$ for the names to be in alphabetical order. Using Kahn's algorithm (topological sort covered in lectures), find a topological ordering for the graph G. If the algorithm finds a cycle in G, then there is no possible alphabet ordering that puts the tome in alphabetical order. Otherwise, the topological sort is a valid alphabet.

Correctness: Each pair of names in the tome imposes a single condition on the ordering of the alphabet based on the first column in which they differ. A valid ordering of the alphabet must satisfy all of these conditions (of which there are at most $n-1$). It is not necessary to consider non-adjacent pairs due to the transitive property; if we find an ordering in which $S[i] < S[j]$ and $S[j] < S[k]$, then we are already guaranteed that $S[i] < S[k]$.

To see why this is equivalent to a topological sort on the graph G, consider the definition. A topological sort in G is any order of the vertices in G for which u comes before v whenever there is an edge from u to v. Since each vertex represents a glyph, and there is a directed edge for every condition given by the tome, any topological sort on G must also satisfy all of these conditions, and hence be an ordering of the alphabet that puts the tome in alphabetical order. Additionally, if there is a cycle in G, this means there is a cycle in the implications imposed by the tome and therefore no valid ordering of the alphabet exists, as we require both $u > v$ and $v > u$ for all glyphs u, v that are part of the cycle.

Time complexity:

- Creating the graph with g vertices and 0 edges takes $O(g)$ time.
- For each i, finding d_i takes $O(k)$ time with a linear search, so adding all of the edges to the graph takes a total of $O(nk)$ time.
- Topological sorting with Kahn's algorithm takes $O(|V| + |E|)$ time, where $|V| = g$ and $|E| \leq n - 1$.

Hence the total time complexity is $O(nk + g)$.