- The instructions are the same as in Homework-1, 2.

There are 6 questions for a total of 85 points.

---

1. Counterexamples are effective in ruling out certain algorithmic ideas. In this problem, we will see a few such cases.

   (a) (4 points) Recall the following event scheduling problem discussed in class:

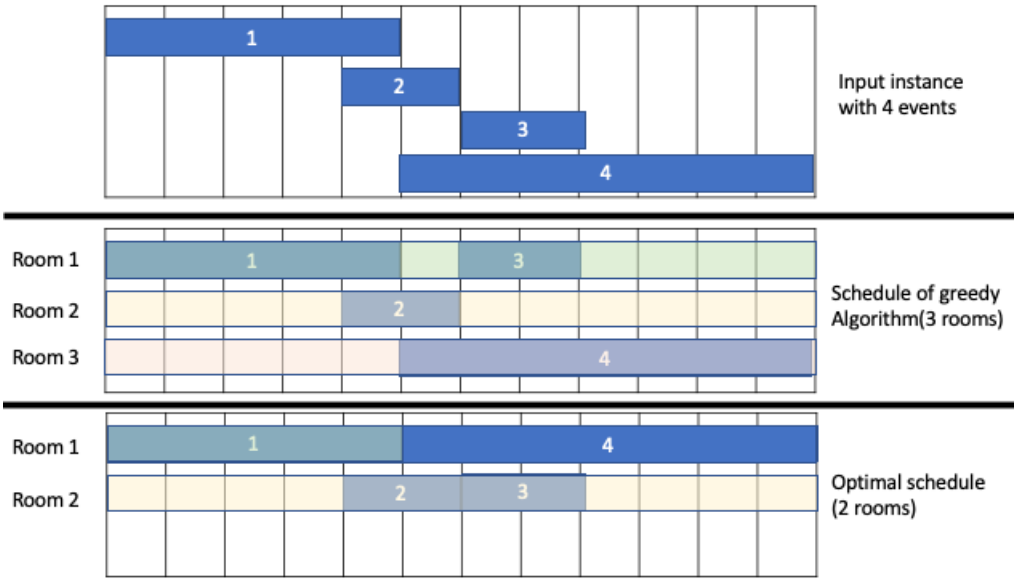
   > You have a conference to plan with n events and you have an unlimited supply of rooms. Design an algorithm to assign events to rooms in such a way as to minimize the number of rooms.

   The following algorithm was suggested during class discussion.

   `ReduceToSingleRoom`$(E_1, ..., E_n)$
   - $U \leftarrow \{E_1, ..., E_n\}$; $i \leftarrow 1$
   - While U is not empty:
     - Use Earliest Finish Time greedy algorithm on events in set U to schedule a subset $T \subseteq U$ of events in room i
     - $i \leftarrow i + 1$; $U \leftarrow U \setminus T$

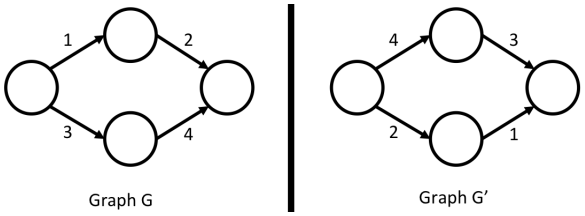
   Show that the above algorithm does not always return an optimal solution.

   Solution: Here is a counterexample.



(b) (4 points) A longest simple path from a node s to t in a weighted, directed graph is a simple path from s to t such that the sum of weights of edges in the path is maximised. Here is an idea for finding a longest path from a given node s to t in any weighted, directed graph $G = (V, E)$:

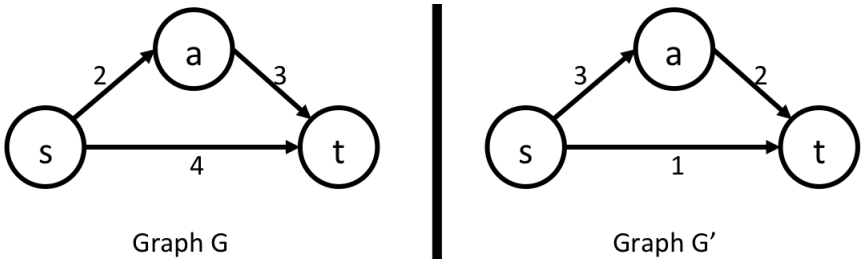
Let the weight of the edge $e \in E$ be denoted by $w(e)$ and let w_{max} be the weight of the maximum weight edge in G. Let G' be a graph that has the same vertices and edges as G but for every edge $e \in E$, the weight of the edge is $(w_{max} + 1 - w(e))$. (*For example, consider the graph G below and its corresponding graph G'.*)



Run Dijkstra's algorithm on G' with starting vertex s and return the shortest path from s to t.

Show that the above algorithm does not necessarily output the longest simple path.

Solution: Consider the counterexample given below.



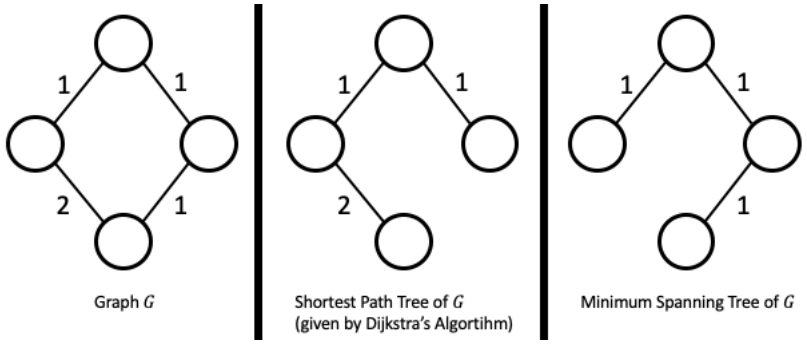
The shortest path from s to t in G' is $s \to t$. However, this is not a longest simple path from s to t in G.

(c) (4 points) Recall that a *Spanning Tree* of a given connected, weighted, undirected graph $G = (V, E)$ is a graph $G' = (V, E')$ with $E' \subseteq E$ such that G' is a tree. The cost of a spanning tree is defined to be the sum of weight of its edges. A *Minimum Spanning Tree (MST)* of a given connected, weighted, undirected graph is a spanning tree with minimum cost. The following idea was suggested for finding an MST for a given graph in the class.

> Dijkstra's algorithm gives a shortest path tree rooted at a starting node s. Note that a shortest path tree is also a spanning tree. So, simply use Dijkstra's algorithm and return the shortest path tree.

Show that the above algorithm does not necessarily output a MST. In other words, a shortest path tree may not necessarily be a MST. (*For this question, you may consider only graphs with positive edge weights.*)

Solution: Here is a counterexample.



2. (15 points) You are given a directed graph $G = (V, E)$ where nodes represent cities and directed edges represent road connections. There is a positive weight $w(e) > 0$ associated with every directed edge $e \in E$ that denotes the cost of traveling along that road (this could be tolls, gas cost etc.). There is also a positive weight $c(v) > 0$ associated with every node $v \in V$ that denotes the cost of visiting the city v (this could be food, lodging etc.). You are planning a trip from a city $s \in V$ to a city $t \in V$ and you want to find a route that will cost you the least amount of money. Note that this is a path that starts with s and ends at t and the sum of weight of edges plus the sum of weight of *intermediate nodes* (i.e., nodes except s and t) in the path is minimised.

   Design an algorithm for this problem. Give running time analysis and proof of correctness.

Solution: We will solve this problem by reducing it to the shortest path problem. Indeed, this problem looks very similar to the shortest path problem, the main difference being that the vertices also have weights in this problem and the length/weight of a path adds up the weight of edges in the path AND the weight of intermediate nodes. For the reduction, we need to create an instance of the shortest path problem (that does not have vertex weights) for any given instance of this problem (that has vertex weights in addition to edge weights).

Given a directed graph $G = (V, E)$ with edge weight function w and vertex weight function c, we consider the graph G' which is the same as G but with edge weight function w' defined as:

$$\begin{aligned}&\text{Let } c'(s) = c'(t) = 0 \text{ and for every vertex } v \in V \setminus \{s, t\}, c'(v) = c(v),\\ &\text{for every edge } (u, v) \in E, w'((u, v)) = w((u, v)) + c'(v)\end{aligned}$$

In other words, we "transfer" the weight of the endpoint of an edge to the edge itself. The following lemma shows that the weight of s-t paths in G and G' are the same.

Lemma: For any simple path $P = v_1, v_2, ..., v_k$ where $s = v_1$ and $t = v_k$,

$$\sum_{i=1}^{k-1} w((v_i, v_{i+1})) + \sum_{i=2}^{k-1} c(v_i) = \sum_{i=1}^{k-1} w'((v_i, v_{i+1}))$$

Proof. The proof follows from the following sequence of equalities:

$$\begin{aligned}\sum_{i=1}^{k-1} w((v_i, v_{i+1})) + \sum_{i=2}^{k-1} c(v_i) &= \sum_{i=1}^{k-2} (w((v_i, v_{i+1})) + c(v_{i+1})) + w((v_{k-1}, v_k))\\ &= \sum_{i=1}^{k-2} (w((v_i, v_{i+1})) + c'(v_{i+1})) + (w((v_{k-1}, v_k)) + c'(t))\\ &\quad \text{(since } c'(t) = 0 \text{ and } c'(v) = c(v) \text{ for } v \in V \setminus \{s, t\})\\ &= \sum_{i=1}^{k-1} (w((v_i, v_{i+1})) + c'(v_{i+1}))\\ &= \sum_{i=1}^{k-1} w'((v_i, v_{i+1}))\end{aligned}$$

The algorithm based on the above reduction is summarised in the following pseudocode:

```
OptimalRoute(G, s, t)
  - Construct weighted graph G' as described above.
  - Run DijkstraAlgorithm on G' with starting vertex s
  - Let P denote the path from s to t in the shortest path tree given by the algorithm
  - return(P)
```

Running time: Constructing graph G' involves defining weight w' that can be done while making a single pass over the adjacency list of G which takes $O(|V| + |E|)$ time. After this, we can run Dijkstra's algorithm using Binary Heaps which costs $O(|E| \cdot \log |V|)$ time. So, the overall running time of the algorithm is $O(|E| \cdot \log |V|)$.

3. You are staying in another city for n days. Unfortunately, not every hotel is available for every day. You are given a list of hotels, $h_1, ..h_k$ and a $k \times n$ array of booleans $Avail[i, j]$ that tells you whether hotel i is available on day j of your trip. You wish to pick which hotel to stay in each day of your trip, in order to minimize the number of times you have to switch hotels. You can assume that at least one hotel is available every day.

   For example, say $n = 7$ and there are three hotels, A, B, C. Hotel A is available days 1-3 and days 6-7, Hotel B is available days 1-5, and Hotel C is available days $3 - 7$. Then one solution is to stay at Hotel B days 1-5, then switch to Hotel C for days 6-7. (We could equally well switch to Hotel A; either way is the optimal 1 switch).

   Here is a greedy strategy that finds the schedule with the fewest switches.

   > **Greedy Strategy:** Stay at the hotel available on the first day with the most consecutive days of availability starting at that day. Stay there until it becomes unavailable, and then repeat with the remaining days.

   (a) Fill in the steps of the proof of the following modify-the-solution lemma:

   Lemma: *Suppose that the greedy algorithm stays at hotel g on the first day and stays there until day I. Let $OStays$ be any scheduling of available hotel rooms. Then there exists a schedule of available hotel rooms assignment $OStays'$ that stays at hotel g until day I, and switches hotel rooms no more than the number of switches for $OStays$.*

   proof: Let $OStays$ be as above.

   i. (1 point) Define $OStays'$.

   > **Solution:** Let $OStays'$ stay in hotel g for the first I days, then be as in $OStays$ for all remaining days.

   ii. (1 point) $OStays'$ is a valid solution because... (Justify why in $OStays'$, we never try to stay at an unavailable hotel.)

   > **Solution:** Since the greedy solution was valid, hotel g has a room available for the first I days. Since $OStays$ is valid, the other hotels have rooms available the days we stay in them.

   iii. (1 point) The number of switches in $OStays'$ is less than or equal to that in $OStays$ because... (justify).

   > **Solution:** In $Ostays$, we have at most one switch in the first I days, that on day I. Since no hotel had a room available for $I+1$ days by the definition of the greedy strategy, $OStays$ must also have at least one switch within those days. In the remaining days, we have the same schedule as $OStays$ so the exact same number of switches. Thus, the total number of switches in $OStays'$ is at most that of $OStays$.

   We will now use the above exchange lemma to argue that the greedy algorithm outputs an optimal solution for any input instance. We will show this using mathematical induction on the input size (i.e., number of days). The base case for the argument is trivial since, for $n = 1$, there is no switch in the greedy algorithm which is optimal.

   (a) (4 points) Show the inductive step of the argument.

   > **Solution:** Inductive step: Here, we assume that the greedy algorithm outputs an optimal solution for any input with k trip days, where $1 \leq k \leq n - 1$. We will show that the greedy algorithm outputs an optimal solution for any input with n days. Let J denote any input where

the number of days is n. So, J is basically an $k \times n$ availability matrix. We break the analysis into points as we did in the lecture.

- Let $GS(J)$ denote the greedy solution in input J.
- Let g be the hotel in which one stays for the first I days as per the greedy solution.
- So, we can write $GS(J) = (g, g, ..., g, GS(J'))$, where J' is the $k \times (n-I)$ matrix $J[1...k, I+1...n]$. That is, $GS(J)$ assigns the first I days to hotel g and then applies the greedy method on the remaining days.
- Let $OS(J)$ denote an arbitrary solution for input J.
- Then by the exchange lemma, there is another solution $OS'(J)$ as per which one stays in hotel g for the first I days and $|OS'(J)| \leq |OS(J)|$, where $|.|$ denotes the number of switches in the assignment.
- So, we have $OS'(J) = (g, g, ..., g, SomeSolution(J'))$.
- Then we have:

  $$|GS(J)| = 1 + |GS(J')| \leq 1 + |SomeSolution(J')| = |OS'(J)| \leq |OS(J)|$$

  The second inequality follows from the induction hypothesis since the input J' has less than n days.
- So, the greedy solution is as good as any other solution.

This completes the inductive step and the exchange argument.

Having proved the correctness, we now need to give an efficient implementation of the greedy strategy and give time analysis.

(a) (6 points) Give pseudocode of an efficient algorithm implementing the above strategy and give a time analysis for your algorithm.

```
GreedyHotels(Avail[1..k, 1..n])
  - Day = 0.
  - While Day < n do:
      - MaxEnd = Day
      - MaxHotel = 0
      - FOR J = 1 TO k do:
          - ThisEnd = Day
          - While ThisEnd < n and Avail[J, ThisEnd + 1] do: ThisEnd++
          - IF ThisEnd > MaxEnd THEN do: MaxEnd = ThisEnd, MaxHotel = J.
      - FOR d = Day + 1 to MaxEnd do: Stay[d] = MaxHotel
      - Day = MaxEnd
  - Return Stay[1..n]
```

Solution: Consider the two inside nested loops, the FOR and the While. For a fixed run of the outside While, for each of k hotels, we increment $ThisEnd$ at most $MaxEnd - Day$ times. So the total time is $O(k(MaxEnd - Day))$, and we sum up these periods over the outside while since we keep the new Day at $MaxEnd$. Note that the periods $[Day, MaxEnd]$ in each of the iterations of the outside loop are disjoint, so the total of all the differences is n, the number of

days. Thus, the total time is $O(nk)$.

4. You are going cross country, and want to visit different national parks on your way. The parks on your trip are, in order, $Park_1, Park_2, ..Park_k$. You have $1 \leq n \leq k$ days for your trip, and you would like to visit a different park each day. You don't want to do unnecessary driving so you will only visit parks in order from smallest number to largest.

   Your input is a $k \times n$ array of booleans $Avail[i, j]$ that tells you whether park i is available on day j of your trip. You wish to pick which park to stay in each day of your trip. Your strategy should either find a way to visit n different parks, or tell you that this is impossible.

   For example, say $n = 3$ and there are four parks, $1, 2, 3, 4$ (i.e., $k = 4$). Park 1 is available all three days. Park 2 is available day 3 only. Park 3 is available days 2 and 3. Park 4 is available all three days. Then we could visit Park 1 the first day, skip Park 2, visit Park 3 the second day, and visit Park 4 the last day.

   Here is a greedy strategy that finds a schedule visiting n parks if one exists:

   > **Greedy Strategy:** On the first day, visit the first park available on the first day. Each other day, visit the first park available on that day which is after your current park.

   We will show the optimality of the greedy strategy using the greedy-stays-ahead method. Suppose that the greedy algorithm stays at parks number $g_1, ..g_n$, in order, and $OStays$ is another solution staying at parks number $p_1, ..p_n$. We claim that at all days i, $g_i \leq p_i$. We prove this by induction on i.

   (a) (7 points) Prove the above claim using induction on i. Show base case and induction step.

   Solution:
   Base case: On the first day, the greedy strategy stays at the first available park, g_1. $OStays$ stays at some available park, p_1. Thus, $g_1 \leq p_1$.
   Inductive step: Assume that $g_i \leq p_i$ for $1 \leq i \leq n-1$. In the induction step, we need to show $g_{i+1} \leq p_{i+1}$. g_{i+1} is the first available park on day $i+1$ which comes after g_i. p_{i+1} is some park that is available on day $i+1$ which comes after p_i. Since $g_i \leq p_i$ by the induction hypothesis, p_{i+1} comes after g_i, so it is at least g_{i+1} (since g_{i+1} was the first such park.)

   (b) (3 points) Use the claim to argue that the greedy algorithm outputs a valid travel plan in case such a plan exists and outputs "impossible" otherwise.

   Solution: Suppose there exists a valid travel plan $p_1, ..., p_n$, then by the previous part we know that the greedy solution satisfies $g_1 \leq p_1, ..., g_n \leq p_n$. This implies that the greedy solution is also a valid travel plan. The other direction is simpler since a valid greedy solution implies that a valid travel plan exists.

   (c) (5 points) Give an efficient algorithm implementing the above strategy and give a time analysis for your algorithm.

   `GreedyParks(`$Avail[1..k, 1..n]$`)`
   - $Day = 1$, $Park = 1$.
   - While $Day \leq n$ and $Park \leq k$ do:
     - While ($Avail[Park, Day] == False$ and $Park \leq k$) do: $Park{+}{+}$;
     - IF $Park > k$ THEN return "Impossible" ELSE $Stay[Day] = Park, Day{+}{+}, Park{+}{+}$
   - IF $Days > n$ return $Stays[1..n]$ ELSE return "Impossible".

Solution: In every step within the outer While loop, the value of $Park$ is incremented by 1 (the only other case is when the program terminates and returns "impossible") and the loop terminates when the value of $Park$ exceeds k. So the running time is $O(k)$.

5. (15 points) You are a party organiser and you are asked to organise a party for a company. The CEO of the company wants this party to be fun for everyone who is invited and has asked you to invite the maximum number of people from the company with the constraint that for no two people who are invited, one is the immediate boss of the other. The company has a typical hierarchical tree structure, where every person except the CEO has exactly one immediate boss.

   Design an algorithm for this problem. You are given as input an integer array $B[1...n]$, where $B[i]$ is the immediate boss of the i^{th} employee of the company. The CEO is employee number 1 and $B[1] = 0$. The output of your algorithm is a subset $S \subseteq \{1, ..., n\}$ of invited employees. Give running time analysis and proof of correctness.

Solution: Given an undirected graph $G = (V, E)$, an independent set of the graph is a subset $I \subseteq V$ of vertices such that for no two nodes $u, v \in I$ we have $(u, v) \in E$. The given problem is the problem of finding the largest independent set of a given tree. We will design a greedy algorithm for finding the largest independent set of any given forest (a set of trees). Here is the greedy strategy that we will use for this problem:

> **Greedy algorithm**: Pick a node v of degree ≤ 1 from the forest, remove this node and its neighbors, and recurse on the remaining forest (obtained by removing node v and its neighbor in case one exists).

Let us prove that the above algorithm indeed outputs a largest independent set of any given forest. We show this using an exchange argument. Let us first obtain the exchange lemma.

Lemma: *For any input forest $G = (V, E)$, let $g \in V$ be the first vertex picked by the greedy algorithm. Let I be any independent set of G that does not contain g. Then there is another independent set I' that contains g and $|I'| \geq |I|$.*

Proof. Note that g is a vertex in G with degree 0 or 1. If g is a degree 0 vertex, then $I' = I \cup \{g\}$ is also an independent set of larger size. If g is of degree 1, then there are two cases to consider:

- The neighbor u of g is in I: In this case, $I' = I - \{u\} \cup \{g\}$ is also an independent set of same size as I.
- The neighbor u of g is not in I: In this case, $I' = I \cup \{g\}$ is also an independent set of larger size.

This completes the proof of the exchange lemma. □

We now prove using induction that the greedy algorithm outputs a largest independent set for any given forest. We consider the following proposition:

$P(n)$: The greedy algorithm produces a largest independent set for any forest with n nodes.

We will show that $P(n)$ holds for all n using induction.

Base case: $P(1)$ holds since for any graph with 1 node, the greedy algorithm outputs one node which is optimal.

Induction Hypothesis: Assume that $P(1), P(2), ..., P(n-1)$ hold for an arbitrary $n > 1$.

Induction step: Here, we will show that $P(n)$ holds. For any forest $G = (V, E)$ with $|V| = n$,

- Let g be the first vertex picked by the greedy algorithm.
- Let G' be the forest obtained by deleting vertex g and its neighbor (if there exists one). Note that G' is a forest that has less than n nodes.

- Let $GS(.)$ denote the greedy solution for an input graph.
- Then, $GS(G) = \{g\} \cup GS(G')$.
- Let I be any independent set of forest G. Then from the exchange lemma, there exists an independent set I' such that $I' = \{g\} \cup IS(G')$ where $IS(G')$ denotes some solution for forest G' and $|I'| \geq |I|$.
- Now we can conclude the argument using the following sequence of inequalities:

$$\begin{aligned} |GS(G)| &= 1 + |GS(G')| \\ &\geq 1 + |IS(G')| \quad \text{(using induction hypothesis)} \\ &\geq |I'| \quad \text{(using the definition of } I') \\ &\geq |I| \quad \text{(using the exchange lemma)} \end{aligned}$$

This implies that the greedy solution is as good as any other solution which means that the greedy algorithm outputs a largest independent set for G. This completes the inductive argument and proof of optimality of our greedy algorithm.

Let us now focus on the implementation of the algorithm. Note that what we are given is not the tree in adjacency list representation but the parent of every node in the array B. However, we can obtain the adjacency list using the following pseudocode:

```
ConvertToTree(B, n)
  - Initialize an adjacency list L with n nodes and no edges
  - For i = 2 to n:
      - Append B[i] to the linked list for node i in L
      - Append i to the linked list for node B[i] in L
  - return(L)
```

The running time of the above procedure is clearly $O(n)$. We can use the above as a subroutine in our algorithm.

```
GreedyAlgorithm(B, n)
  - L ← ConvertToTree(B, n)
  - Go over adjacency list L to find the degree D[1...n] of all nodes
  - Initialise a list S of all degree 0 and degree 1 nodes
  - While there is at least one node that is not marked deleted:
      - Let v be any node in S that is not marked deleted
      - Remove v from S; I ← I ∪ {v}
      - Mark v as deleted
      - If (D[v] = 1)
          - Let u be the only node in the linked list of v that has not been marked deleted
          - Mark u as deleted
          - For every neighbor w of u that has not been marked deleted:
              - D[w] ← D[w] − 1
              - If (D[w] ≤ 1) append w to list S
  - return(I)
```

Running time: The first step takes $O(n)$ time. Going over the adjacency list and initialising the list takes $O(n)$ time. Within the while loop the time spent per node before it is marked deleted is some constant times the degree of the node. So, the time in the while loop is $O(|V| + |E|)$ where V and E are the vertex and edge sets respectively. Since the given graph is a tree, we have $|E| = O(|V|) = O(n)$. So, the overall running time is $O(n)$.

6. (15 points) Given positive integers $(d_{out}[1], d_{out}[2], ..., d_{out}[n])$ and $(d_{in}[1], d_{in}[2], ..., d_{in}[n])$, design a greedy algorithm that determines whether there exists a simple directed graph where the i^{th} node has indegree $d_{in}[i]$ and outdegree $d_{out}[i]$. Your algorithm should output "yes" if such a graph exists and "no" if no such graph exists. Give running time analysis and proof of correctness for your greedy algorithm.
(*Recall that a simple directed graph is a graph that does not have self-loops or multi-edges.*)