- The instructions are the same as in Homework 1 and 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:

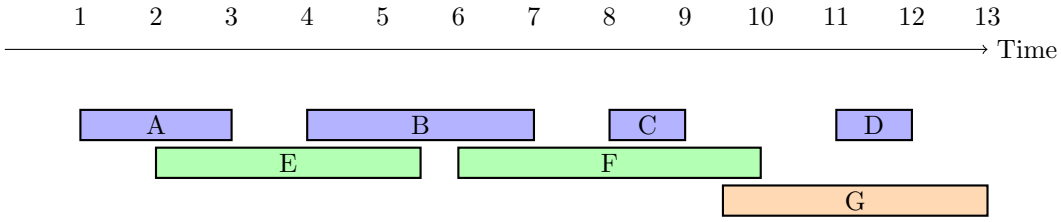
   > You have a conference to plan with n events, and you have an unlimited supply of rooms. Design an algorithm to assign events to rooms in such a way as to minimize the number of rooms.

   The following algorithm was suggested during class discussion.

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

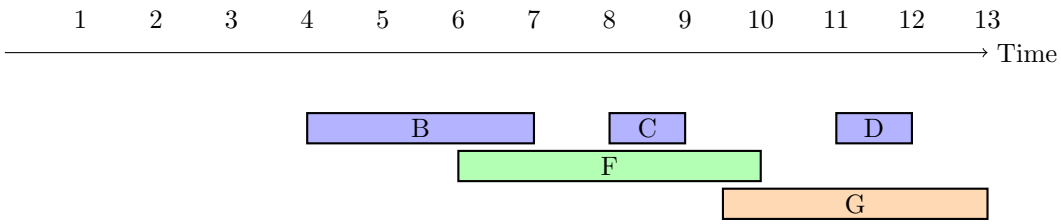
   Show that the above algorithm does not always return an optimal solution.

Solution:
We will prove that the given algorithm is not optimal by giving a counter-example as:

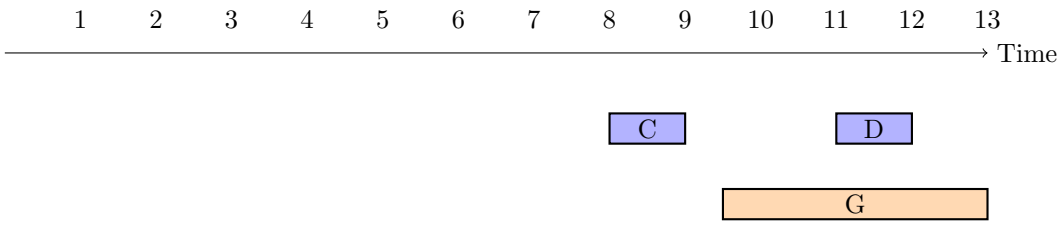


So, the start times and finish times of events are:
A $= (1, 3)$, B $= (4, 7)$, C $= (8, 9)$, D $= (11, 12)$, E $= (2, 5)$, F $= (6, 10)$ and, G $= (9.5, 13)$

Now, if we go by the algorithm given in the question. A has least finishing time so it will be scheduled in room 1 and after removing events colliding with A we will be left with:



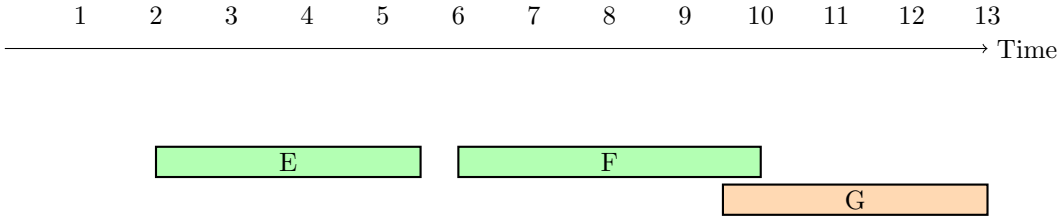
Next B has least finishing time so it will be scheduled in room 1 and after removing events colliding with B we will be left with:



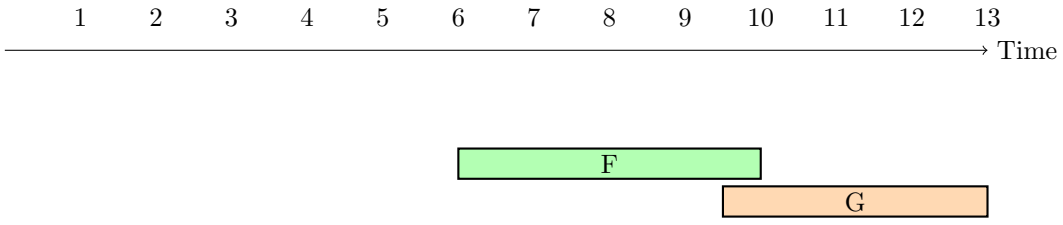
Next C has the least finishing time so it will be scheduled in room 1 and after removing events colliding with C we will be left with:

Next D has the least finishing time so it will be scheduled in room 1 and after removing events colliding with D we are left with no events that can be scheduled in room 1.

Now we are left with three events that have not been scheduled yet.

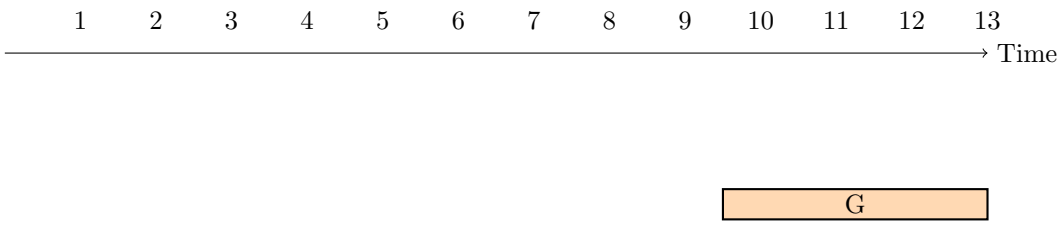


Out of these three E has the least finishing time so it will be scheduled in room 2 and after removing events colliding with E we are left with:



Now F has the least finishing time so it will be scheduled in room 2 and after removing events colliding with F we are left with no events that can be scheduled in room 2.

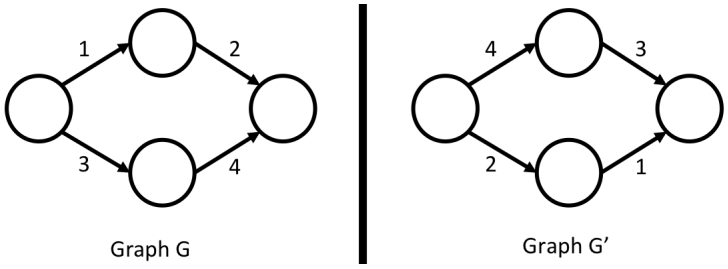
Now, we are now left with:



And this G will be scheduled in room 3.
So the given algorithm allots 3 rooms for the given events but we can observe that an optimal solution is to allot room 1 to A, B, C, G and room 2 to E, F, D. Hence, the minimum number of rooms required is 2, which is less than 3.
Therefore we conclude that the given algorithm does not always gives an optimal solution.

(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 $G = (V, E)$:

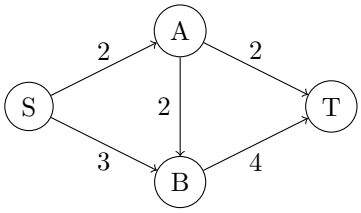
Let the weight of the edge $e \in E$ be denoted by $w(e)$ and let w_{max} be the weight of the maximum weight edge in G. Let G' be a graph that has the same vertices and edges as G, but for every edge $e \in E$, the weight of the edge is $(w_{max} + 1 - w(e))$. (*For example, consider the graph G below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

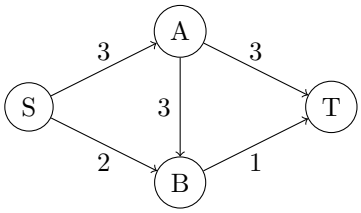
Solution:
We will prove that the given algorithm is not optimal by giving a counter-example.
Consider the graph G given below.



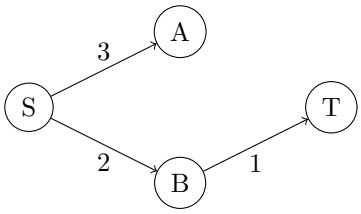
Now we will transform the edge-weights of the given graph according to the method given in the question. Since, $w_{max} = 4$, hence:

- w(AB) $= 4 + 1 - 2 = 3$
- w(SA) $= 4 + 1 - 2 = 3$
- w(SB) $= 4 + 1 - 3 = 2$
- w(AT) $= 4 + 1 - 2 = 3$
- w(BT) $= 4 + 1 - 4 = 1$

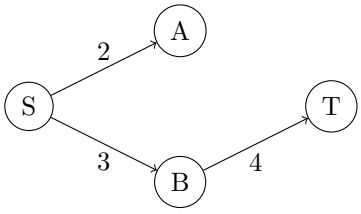
After transforming we will get the following graph G' with the modified edge weights.



Next we will run dijkstra on this graph G' and we get the following spanning tree :



Now replacing edge weights in G' with the original edges edge weights in G, we get:



Now, according to this algorithm the longest path from S to T is S-B-T i.e., of weight 7.
But,

we can see in the original graph G that the longest path from S to T is S-A-B-T ,i.e., of weight 8.

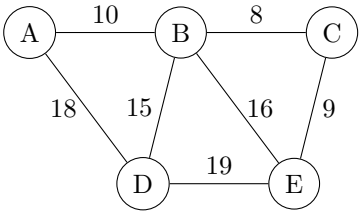
Thus, we can conclude that our algorithm is incorrect as it does not always gives a correct output.

(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 as the sum of the 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.

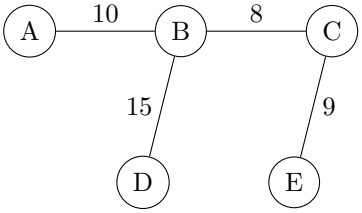
> Dijkstra's algorithm gives a shortest path tree rooted at a starting node s. Note that a shortest path tree is also a spanning tree. So, simply use Dijkstra's algorithm and return the shortest path tree.

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

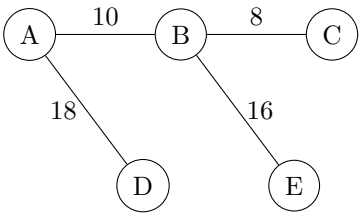
Solution:
We will prove that the given algorithm is not optimal by giving a counter-example.
Let us take the graph given below.



The minimum spanning tree MST for the above graph will be :



⇒ Sum of weight of edges = 10+ 8+ 15+ 9 = 42
But,
When we run Dijkstra's Algorithm from node A we will get the spanning tree MST' as:



⇒ Sum of weight of edges = 10+ 8+ 18+ 16 = 52

We notice that this spanning tree MST' is very different from the MST above and for a graph with distinct positive edge weights, the minimum spanning tree is unique, which means if our approach was correct, MST should have been same as MST', but it is not so.
Hence, the above algorithm does not necessarily output a Minimum Spanning Tree.

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 toll, 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 want to find a route that will cost you the least. Note that this is a path that starts with s and ends at t and the sum of the weight of edges plus the sum of the weight of intermediate nodes (i.e., nodes except s and t) in the path is minimized.

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

Solution:

Instance: Directed Graph $G = (V, E)$ with positive edge-weights $w(e)$ and node-weights $c(v)$, $\forall v \in V$ and, $\forall e \in E$; nodes $s, t \in V$

Solution Format: A list of edges $e_1, e_2, e_3, \cdots, e_k$

Constraint: Must from a path from s to t

Objective: Minimize $\sum_{i=1,j=1}^{k,k+1}(w(e_i) + c(v_j))$

Algorithm: Procedure solve(G, s, t)

1. Create a new graph $G' = (V', E')$ from $G = (V, E)$, by adding the cost-weight of a node v to all the edge-weights of edges incident on it, for all v in V, i.e, $w(e') = w(e) + c(v)$, $\forall$ e incident on v and $\forall e' \in E'$.
2. Initialize two arrays - dist and prev, of size $|V'|$, each.
3. for all u in V':
   - dist[u] = ∞
   - prev[u] = -1
4. dist[s] = 0.
5. Initialize a Min Heap H as a priority queue.
6. for all v in V':
   - Insert {dist[v], v} into H.
7. while H is not empty:
   - u = deleteMin(H)
   - for each edge (u, v) in E':
     - if (dist[v] >(dist[u] + $w'(u, v)$))
       - dist[v] = dist[u] + $w'(u, v)$
       - prev[v] = u
       - decreaseKey(H, {dist(v), v})
8. Create a Tree $T = (V, E'')$
   b = t
   while b != s:
   create a edge from prev[b] to b
   b = prev[b]
9. Return tree T which will give path p from s to t and dist[t] - c[t].

Note: In the above algorithm:

→ *= means assignment.*

→ *Indentation, along with hyphen (-), denotes the scope.*

Running Time Analysis:

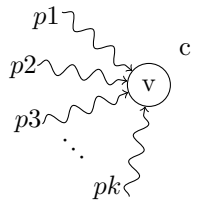
1. Creating G' requires a simple traversal of adjacency list of G that takes $O(V + E)$ time.
2. Initializing dist and prev array takes $O(V')$ time.
3. Inserting V' keys into heap H takes $O(V'logV')$ time.
4. Performing delete operations from a min Heap H for all V' vertices will take $O(V'logV')$ time.
5. Finding neighbours for all vertices requires one traversal of adjacency list of G' that takes $O(V' + E')$ time.
6. Performing decrease-key for all E' edges will take in worst case $O(E'logV')$ time.
7. Creating Spanning Tree T will at max take $O(V)$ time.

$$\begin{aligned}
\text{T.C.} &= O\left(|V| + |E| + |V'| + |V'|log|V'| + |V'|log|V'| + |V'| + |E'| + |E'|log|V'| + |V|\right) \\
&= O\left(4.|V| + 2.|E| + 2.|V|log|V| + |E|.log|V|\right) \quad (as|V| = |V'|and|E| = |E'|) \\
&= O\left(c + (|V| + |E|)\, log|V|\right), \text{ where } c \text{ is some constant} \\
&= O\left((|V| + |E|)\, log|V|\right)
\end{aligned}$$

Proof of Correctness:

Termination Check: The algorithm will execute on each vertex exactly once, i.e., until the min-Heap H becomes empty. In each iteration, it will delete a vertex u from H and perform edge relaxation and decrease key operation, if needed. Once all vertices have been processed and H becomes empty, the algorithm will eventually terminate.

Proof: Consider an intermediate node v with cost $c(c > 0)$. Suppose k paths, with costs - $p_1, p_2, p_3, \cdots, p_k$ are incident on v as shown below:



Let us say p_i is a path with least cost where i ranges from 1 to k. Now, the total optimal cost after processing v should be $c + min(p_1, p_2, p_3, \ldots, p_k) = min(c + p_1, c + p_2, c + p_3, \ldots, c + p_k)$ as adding a positive constant to each value doesn't change the fact that path with minimum cost is still p_i.
Thus, if we add cost of node *(a positive constant)* to each edge incident on it, the fact that path with minimum cost is still p_i won't change. As a result the graph G' will become a directed graph with positive edge-weights and zero node-weights. Thus, the minimum cost/shortest path from source to destination in G' can be found via Dijkstra's Algorithm and so correctness of our algo entirely depends on correctness of Dijkstra's Algorithm.

And we know, that the Dijsktra's Algorithm is correct as the Alarm Clock Algorithm was correct and the Alarm Clock Algorithm was correct because BFS was correct.
$\therefore$ Our Algorithm is correct.

Q.E.D.

3. (*Example for modify-the-solution*) You are staying in another city for n days. Unfortunately, not every hotel is available 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 the day j of your trip. You wish to pick which hotel to stay in each day of your trip 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 on 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 on days 1-5, then switch to Hotel C for days 6-7. (We could equally well switch to Hotel A; either way the optimal is 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 on that day. Stay there until it becomes unavailable, and then repeat with the remaining days.

   We will show that the above greedy strategy gives an optimal solution using modify-the-solution. For this, we will first need to prove the following exchange lemma.

   Exchange 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 $OStays$.*

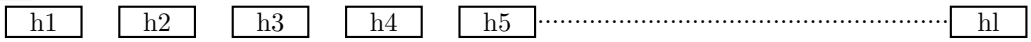
   proof: Let $OStays$ be as above. We ask you to complete the proof of the exchange lemma below.

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

Solution:

GS :

$OStays$:



$OStays'$:

Let $OStays = h_1, h_2, h_3, \ldots, h_l$ be any arbitrary feasible solution that does not include g. Let $OStays'$ be another solution that does include g and $cost(OS') \leq cost(OS)$, where, *cost* refers to the number of switches.

Now,

$$OStays' = g \oplus SS(I')$$

where, SS' denotes some solution on remainder instance I' and, $\oplus$ means "followed by".

or,

$$OStays' = OStays - \{h_1, \ldots, h_m\} + g$$

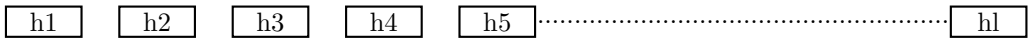
where, $1 \leq m \leq l$.

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

Solution:

$\underline{GS:}$

$\underline{OStays:}$



$\underline{OStays':}$

$OStays'$ is the same as $OStays$, except that the first choice in $OStays'$ is the greedy choice g. So, in $OStays'$, we stay at the hotel g for the first I days. After the I^{th} day we can switch to corresponding hotels in $OStays'$ and since at least one hotel is available every day *(given)*. Hence, we never stay at an unavailable hotel in $OStays'$ as $OStays$ is a feasible solution and at least one hotel is available every day.
$\therefore$ $OStays'$ is a valid solution.

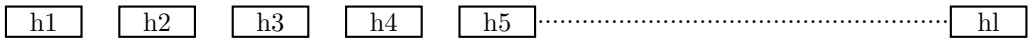
Q.E.D.

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

Solution:

$\underline{GS:}$

$\underline{OStays:}$



$\underline{OStays':}$

Now, there are two possible scenarios:

<u>case 1:</u> Hotel g is available simultaneously with the first m hotels of OS, $1 < m \leq l$.
In this case:

$$cost(OS) = l$$

and,

$$cost(OS') = l - m + 1.$$

Since,

$$1 < m \leq l$$

$$\Rightarrow cost(OS') < cost(OS)$$

<u>case 2:</u> Hotel g is available till one day before the first availability or the last availability of hotel H_2, $m = 1$
In this case:

$\Rightarrow$ After I^{th} day, switch to Hotel H_2 from g.

So,

$$H'_2 = H_2, H'_3 = H_3, \ldots H'_l = H_l$$

$$\Rightarrow cost(OS') = cost(OS)$$

$\therefore \text{cost}(OS') \leq \text{cost}(OS)$

Q.E.D.

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: Let $P(n)$ denote the correct output for n number of days.
Assume for $n = 1, 2, 3, \ldots, k$ days, greedy strategy outputs an optimal solution.
Now,
For $n = k + 1$: Using the induction hypothesis:

$$\begin{aligned}
&\text{cost}(GS(k \text{ days})) \leq \text{cost}(OS'(k \text{ days})) \\
\Rightarrow\ &1 + \text{cost}(GS(k \text{ days})) \leq 1 + \text{cost}(OS'(k \text{ days})) \\
\Rightarrow\ &|g| + \text{cost}(GS(k \text{ days})) \leq |k| + \text{cost}(OS'(k \text{ days})) \\
\Rightarrow\ &\text{cost}(GS(k+1 \text{ days})) \leq \text{cost}(OS'(k+1 \text{ days})) \\
\Rightarrow\ &\text{cost}(GS(k+1 \text{ days})) \leq \text{cost}(OS'(k+1 \text{ days})) \leq \text{cost}(OS(k+1 \text{ days}))
\end{aligned}$$

(Using Exchange Lemma)

$\Rightarrow P(k+1)$ holds for $n = k + 1$.
$\because [P(1) \wedge P(2) \wedge \ldots \wedge P(k)] \Rightarrow P(k+1)$ holds.
$\therefore$ P(n) is true, $\forall$ n ≥ 1 (induction step).

Q.E.D.

Having proved the correctness, we now need to efficiently implement the greedy strategy and give time analysis.

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

Solution:

Instance: A $k \times n$ boolean array $Avail[i, j]$, $1 \leq n \leq k$

Solution Format: A list of hotels $I \subseteq \{h_1, h_2, h_3, \ldots, h_k\}$

Constraint: Hotel h_i must be available on Day d_j, $\forall h_i \in I, j =$ index of h_i in I

Objective: Minimize $|I|$

Algorithm: Procedure solve$(Avail, k, n)$

1. Initialize a dynamic data structure *(e.g. a vector in C++ or, an ArrayList in Java)* called I to store our answer.
2. Initialize a variable $j = 1$.
3. while $j \leq n$:
   - Initialize a variable $choice = -1$
   - Initialize a variable $maxdiff = 0$
   - For $i = 1, 2, 3, \ldots, k$:
     - if $(Avail[i][j] == true)$
       - Initialize a variable $start = j$
       - Initialize a variable $end = j$
       - while $(Avail[i][end] == true)$
         - $end = end + 1$
       - Initialize a variable $currdiff = end - start$
       - if $(currdiff > maxdiff)$
         - $maxdiff = currdiff$
         - $choice = i$
   - Add $choice$ to I
   - $j = maxdiff + j$
4. Output the optimal choice of hotels using I and minimum number of switches required is equal to size of I.

Note: In the above algorithm:

→ *= means assignment.*

→ *== means equality comparison operator.*

→ *Indentation, along with hyphen (-), denotes the scope.*

Running Time Analysis:

1. Initializing our dynamic data structure and variable j takes $O(1)$ time.
2. In worst case, each day, hotel will be available for at most one day. So, the next loops will run overall for $O(n.k)$ time.
3. The returning of the order of hotels will take $O(n)$ time.

$$\begin{aligned}\text{T.C.} &= \mathrm{O}(1 + n.k + n) \\ &= O\left(c + n + n.k\right) \\ &= O\left(n.k\right)\end{aligned}$$

4. (*Example of greedy-stays-ahead*) 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 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 the smallest number to the largest.

   Your input is a $k \times n$ array of booleans $Avail[i, j]$ that tells you whether park i is available on the 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 on day 3 only. Park 3 is available on days 2 and 3. Park 4 is available all three days. Then we could visit Park 1 on the first day, skip Park 2, visit Park 3 on the second day, and visit Park 4 on the last day.

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

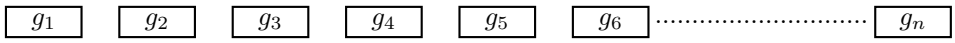
   > **Greedy Strategy:** On the first day, visit the first park available on the first day. Each other day, visit the first park available on that day, which is after your current park. If there is no park to pick on a day, say that the trip is impossible.

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

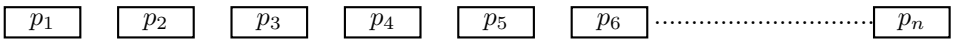
   (a) (7 points) Prove the above claim using induction on i. Show base case and induction step.

Solution:

$\underline{GS:}$



$\underline{OStays:}$



Let $P(i)$ = correct solution for the first i days.

For $i = 1$: Any strategy is optimal as greedy will select the first available day *(if any)*.
Assume for $1, 2, 3, \ldots, k$ days, $g_i \leq p_i, 1 \leq i \leq k$.

For $n = k + 1$: Using the induction hypothesis:

$$g_k \leq p_k$$
$$\Rightarrow g_{k+1} \leq p_{k+1}$$

Now, Assume that greedy wasn't optimal i.e., $|GS| < |OStays|$.
So, suppose that GS only chose till g_k but $OStays$ chose till p_{k+1}.

$$\Rightarrow g_k \leq p_k \leq p_{k+1}$$

But, our greedy algo. wouldn't have stopped and would've picked p_{k+1} as it was available on day $k + 1$.

$$\Rightarrow \text{It is a contradiction.}$$
$$\therefore \text{Our assumption was incorrect.}$$

$\Rightarrow |GS| \geq |OStays|$ or, $g_{k+1} \leq p_{k+1}$
$\Rightarrow P(k + 1)$ holds for $n = k + 1$.
$\because [P(1) \wedge P(2) \wedge \ldots \wedge P(k)] \Rightarrow P(k + 1)$ holds.
$\therefore$ P(n) is true, $\forall$ n ≥ 1 (induction step).

Q.E.D.

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

Solution: We know,

$$g_i \leq pi, \forall i$$

case 1: No park is available on the i^{th} day
So, $g_i = p_i$ as GS outputs impossible and $OStays$ also outputs impossible.

case 2: k parks are available on the i^{th} day
Suppose $OStays$ chose the k^{th} day.
So, $g_i \leq p_i$ as GS will choose the least park number available (k or, lesser) and continue further.

$\therefore$ Greedy algorithm always outputs a valid travel plan if such a plan exists and it outputs "impossible" otherwise.

Q.E.D.

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

Solution:

Instance: A $k \times n$ boolean array $Avail[i, j]$, $1 \leq n \leq k$

Solution Format: A list of parks $P = \{p_1, p_2, p_3, \ldots, p_n\}$

Constraint: Park p_i must be available on Day d_j, $\forall p_i \in P, j$ = index of p_i in P. Also, $p_i \neq p_j, \forall p_i, p_j \in P$

Objective: $i < j$, $\forall$ consecutive $p_i, p_j \in P$

Algorithm: Procedure solve($Avail, k, n$)

1. Initialize a dynamic data structure *(e.g. a vector in C++ or, an ArrayList in Java)* called P to store our answer.
2. Initialize a variable $start = 1$.
3. For $j = 1, 2, 3, \ldots, n$:
   - Initialize a boolean variable $flag = false$
   - For $i = start, start + 1, start + 2, \ldots, k$:
     - if ($Avail[i][j] == true$)
       - Add i to P.
       - $start = i + 1$
       - $flag = true$
       - Break out of inner for loop
   - if ($flag == false$)
     - Ouput "It is impossible".
4. Output "It is possible" and output the required path using P.

Note: In the above algorithm:

→ *= means assignment.*

→ *== means equality comparison operator.*

→ *Indentation, along with hyphen (-), denotes the scope.*

Running Time Analysis:

1. Initializing our dynamic data structure and start variable takes $O(1)$ time.
2. Even though there are two nested loops of size n and k, yet they will take $O(k)$ time in the worst case as the start variable will force the inner for loop, to run for at most k iterations, throughout. This is because once a park is picked, the value of the start will be set to the next row number, and the inner for loop will start iterating from that row in successive iteration.
3. The returning of order of parks *(if exists)*, will take $O(1)$.

$$\begin{aligned} \text{T.C.} &= O(1 + k + 1) \\ &= O(c + k) \\ &= O(k) \end{aligned}$$

5. (15 points) You are a party organizer, and you are asked to organize a party for a company. The company's CEO wants this party to be fun for everyone who is invited and has asked you to invite the maximum number of employees of the company with the constraint that for no two employees who are invited, one is the immediate boss of the other. The company has a typical hierarchical tree structure, where every employee 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:

Instance: An integer array $B[1...n]$, denoting $B[i]$ is the immediate boss of i^{th} employee

Solution Format: A list of employees $S \subseteq \{1, 2, 3, ..., n\}$, denoting employees invited to the party.

Constraint: $i \neq B[j]$ and, $j \neq B[i]$, $\forall i, j \in S$

Objective: Maximize $|S|$

Algorithm: Procedure solve(B, n)

1. Initialize a dynamic data structure *(e.g. a vector in C++ or, an ArrayList in Java)* called S to store our answer.

2. First we will convert the given array B into its corresponding tree and store it in an adjacency list. To do this:
   - Initialize a dynamic data structure *(e.g. a vector¡int¿ adj[n] in C++)* called *adj* to store adjacency list of the tree corresponding to array B.
   -For $i = 1, 2, 3, \ldots, n$:
   -Initialize a variable $u = B[i]$
   -Initialize a variable $v = i$
   -if (v is not equal to 0):
   -Add v to $adj[u]$

3. Initialize a queue Q and a stack STK.

4. Add 1 to Q.

5. while Q is not empty:
   -Initialize a variable $size$ = size of Q
   -while ($size$ is not equal to 0)
   -Initalize a variable $temp$ = Dequeue(Q)
   -for all neighbors u of $temp$:
   -Push u into Q
   -Push $temp$ in STK
   -$size = size - 1$

6. Initialize a boolean array called vis, of size n, to false

7. while STK is not empty:
   - Initialize a variable emp = Pop(STK)
   - If emp has no neighbour:
   - Add emp to S
   - Set $vis[emp]$ to $true$
   - else:

- Initialize a boolean variable $flag = false$
- for all neighbors u of emp:
  - if $(vis[u] == true)$
    - set $flag = true$
    - break out of the current for loop
- if $(flag == false)$
  - Add emp to S
  - set $vis[emp] = true$

8. Return array S

Note: In the above algorithm:

→ *= means assignment.*

→ *== means equality comparison operator.*

→ *Indentation, along with hyphen (-), denotes the scope.*

Running Time Analysis:

1. Initializing our dynamic data structure and adj array takes $O(1)$ time.
2. Creating adjacency list adj from given array B takes $O(n)$ time.
3. Initializing queue and stack and pushing 1 into queue takes $O(1)$ time.
4. The while loop will execute $O(n)$ as it is a tree, and each time constant time operations are performed in it, that takes $O(1)$ time.
5. Initializing a boolean array of size n to $false$ takes $O(n)$ time,
6. Inviting employees after popping out from stack takes $O(n)$ time.
7. Returning the answers takes $O(1)$ time.

$$\begin{aligned} \text{T.C.} &= O\left(1 + n + 1 + n + n + n + 1\right) \\ &= O\left(c + 4.n\right) \\ &= O\left(n\right) \end{aligned}$$

Proof Of Correctness:

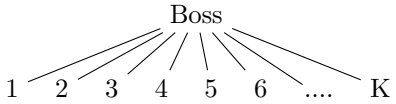
Termination Check: Each node is pushed exactly once in the queue and exactly once in the stack, followed by some finite time processing. Hence, after processing all the nodes, our algorithm will eventually terminate.

Proof:

Claim: Inviting employees who are not boss of anyone first, will lead to maximum number of invites.

- We have created a Tree using the given array B such that the parent of node i is the immediate boss of i.

- Consider a node *Boss* with k children, $1, 2, 3, ..., k$ as below:



Here there will be two cases:

  - **Case1: There is exactly one immediate employee of a boss.**
    In this case we will invite the employee as if we invite the boss we will not be able to invite boss of the boss.
  - **Case2: There is more than one immediate employee of a boss.**

    * **Case1 : We invite boss first**
      If we do this then we will not be able to invite 1 to k employees. So we are inviting 1 boss and not inviting k employees. So this is not a good choice.
    * **Case2 : We invite any employee of Boss first**
      If we do this then we will not be able to invite 1 boss. So we are inviting k employees and not inviting 1 boss . So this is a good choice.

- So in our tree leaves are the employees who are not boss of anyone. So we are inviting them first and inviting others then by traversing tree in reverse order by checking the condition that any of their immediate employee is not invited.

Now let us prove by induction that our algorithm is correct:

P(n): We have invited maximum number of employee for tree of depth n.

Base Case:
For n = 1, we will always invite the employee because of the reason given above. Hence, P(n) holds true for n = 1.

Inductive Hypothesis:
Using weak induction let us say P(k) holds true for $k \leq n$.

Inductive Step:
Now let us say that at level k+1 there are E employees and we have invited all the employees and set their invited as true .
P(k+1) = E + P(k)
Thus we are left with k levels and by induction hypothesis we know that P(k) is true. Thus P(k+1) holds true.
Hence we can say that P(n) holds true.

$\therefore$ Our algorithm is correct.

Q.E.D.

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.*)

Solution:

Instance: Two integer array $(d_{out}[1], d_{out}[2], ..., d_{out}[n])$ and $(d_{in}[1], d_{in}[2], ..., d_{in}[n])$, where the i^{th} node has indegree $d_{in}[i]$ and outdegree $d_{out}[i]$

Solution Format: Yes or No.

Constraint:

Objective: Check if there exists a directed graph G which has indegree array as d_{in} and outdegree array as d_{out}.

Algorithm: Procedure solve($n, d_{out}[], d_{out}[]$)

1. Read an integer "n"
2. Create arrays $d_{out}[]$ and $d_{in}[]$ of size "n"
3. Loop i=0 to n:
   - Read $d_{out}[i]$ from standard input
4. Loop i=0 to n:
   - Read $d_{in}[i]$ from standard input
5. Initialize sumoutdeg and sumoutdeg with 0
   Loop i=0 to n
   - Increment "sumoutdeg" by $d_{out}[i]$
   - Increment "sumindeg" by "$d_{in}[i]$"
6. If "sumoutdeg" != "sumindeg":
   - Print No
   - Exit
7. If for any index i, $d_{out}[i] >= n$ or $d_{in}[j] >= n$:
   - Print No
   - Exit
8. Apply Count sort on $d_{out}[]$ in non-increasing order and store original index of all in $pos_{out}[]$
9. For i = 1 to n:
   - Apply Count sort on $d_{in}[]$ in non-increasing order and store original index of all in $pos_{in}[]$
   -Initialize j = n-1
   - while $d_{out}[i]$! = 0:
     - if ($pos_{out}[i]! = pos_{in}[j]$):
       - Decrement $d_{out}[i]$
       - Decrement $d_{in}[j]$
       - Decrement j by 1

10. If all elements of $d_{in}[]$ and $d_{out}[]$ are 0:
    - Print Yes

    Else:
    - Print No

Note: *In the above algorithm:*

→ *= means assignment.*

→ *== means equality comparison operator.*

→ *Indentation, along with hyphen (-), denotes the scope.*

Running Time Analysis:

1. Reading input for both arrays $d_{in}[]$ and $d_{out}[]$ will take time $O(n)$.
2. Calculating sumindeg for $d_{in}[]$ and sumoutdeg for $d_{out}[]$ will take time $O(n)$.
3. Applying count sort on $deg_{out}[]$ array and creating pos_{out} will take time $O(n)$.
4. Now loop traversing each node in $deg_{out}[]$ will take $O(n^2)$ time as for each node we are again applying count sort on the deg_{in} array and doing some additional work.

$$\begin{aligned} \text{T.C.} &= O\left(n + n + n^2\right) \\ &= O\left(cn + n^2\right) \\ &= O\left(n^2\right) \end{aligned}$$

Proof Of Correctness:

Termination Check: Every time we remove a node with outdegree, say k, we reduce k largest indegrees by -1, and move on to the next node in outdegree array. Once we have processed all the outdegrees, our algorithm eventually terminates.

Proof: To prove our algorithm, we will first prove the following claim:

Claim: A graph can be constructed for G with out-degree sequence $= \{o_1, o_2, o_3, ..., o_n\}$ and in-degree sequence $= \{i_1, i_2, i_3, ..., i_n\}$ iff a graph can be constructed for G' with in-degree sequence $= \{o_2, o_3, ..., o_n\}$ and in-degree sequence $= \{i_1 - 1, i_2 - 1, i_3 - 1, ..., i_{0_1} - 1..., i_n\}$

Now,
Assume that a graph can be constructed with degree sequences of G.
$\Rightarrow$ If we remove a vertex of out-degree o_1 from G, a graph with expected degree sequences of G' is obtained *(after rearranging some edges, if needed).*
$\Rightarrow$ A graph can be constructed with degree sequences of G'.
Again,
Assume that a graph can be constructed with degree sequences of G'.
$\Rightarrow$ If we add a vertex of out-degree o_1 (i.e, adjacent to all nodes with indegrees $\{i_1 - 1, i_2 - 1, i_3 - 1, ..., i_{0_1} - 1\}$)into G', a graph with the expected degree sequences of G is obtained.
$\Rightarrow$ A graph can be constructed with degree sequences of G.
And, this is what our algorithm is trying to simulate.
$\therefore$ Our Algorithm is correct.

Q.E.D.