CS101A(H) Homework 7

Due date: December 16, 2025, at 20:00 p.m.

Your grade for this homework will be computed as

$$\min\big(\text{Single Choices} + \text{Multiple Choices} + \text{Others},\ 50 \text{ pts}\big) + \text{Bonus}.$$

Notes: Unless otherwise stated, always express your final asymptotic bounds using $\Theta(\cdot)$ notation for tight complexity, not just $O(\cdot)$ or $\Omega(\cdot)$. All answers must be written **inside the provided answer boxes** and **in English**. When submitting, match your solutions to the problems correctly in Gradescope. No late submission will be accepted. Failure to follow these rule may result in partial or full loss of credit.

Usage of AI. The use of AI tools for searching information or obtaining assistance on this homework is strictly prohibited. All solutions must be derived from your own understanding and effort. Submissions will be reviewed carefully, and any indication of reliance on AI-generated content will result in severe penalties.

Academic Integrity. This course upholds the highest standards of academic integrity. Any form of academic dishonesty, including plagiarism, unauthorized collaboration, or the use of prohibited resources such as AI tools, will be treated as a serious violation. Such actions not only undermine your own learning but also violate university policies.

1. (10 points) Single Choices

Each question has **exactly one** correct answer. Select the best answer.

(a) (2') Let $G = (V, E)$ be a directed graph. We consider:

(1) Given two vertices u, v, test whether there is a directed edge $u \to v$.

(2) Given a vertex u, iterate over all its outgoing neighbors.

(3) Run a full DFS on the whole graph.

Which option is *impossible* for any single concrete representation of the graph?

A. Using the adjacency matrix, (1) runs in $O(1)$, (2) runs in $\Theta(|V|)$, and (3) runs in $\Theta(|V| + |E|)$.

B. Using the adjacency matrix, (1) runs in $O(1)$, (2) runs in $\Theta(|V|)$, and (3) runs in $\Theta(|V|^2)$.

C. Using the adjacency list, (1) and (2) run in $\Theta(\deg(u))$, and (3) runs in $\Theta(|V| + |E|)$.

D. Using an edge list, (1) and (2) run in $\Theta(|E|)$, and (3) runs in $\Theta(|V| + |E|)$.

Solution: Correct choice: A.

For an **adjacency matrix**:

- (1) Testing whether $u \to v$ exists is just checking one matrix entry, so it is indeed $O(1)$.
- (2) Iterating over all outgoing neighbors of u requires scanning the entire row of the matrix, which is $\Theta(|V|)$, so the bound in (2) is also consistent.
- (3) However, a full DFS must, in the worst case, consider every matrix entry (or equivalently, for each of the $|V|$ vertices, scan its whole row), which is $\Theta(|V|^2)$ time. This does *not* simplify to $\Theta(|V| + |E|)$ in general, because when the graph is sparse (e.g., $|E| = O(|V|)$), the expression $|V| + |E|$ is $O(|V|)$, which is asymptotically smaller than $|V|^2$. So claiming DFS on an adjacency matrix always runs in $\Theta(|V| + |E|)$ is incorrect.

Thus the combination in (A) cannot hold for any concrete implementation using an adjacency matrix.

The other options describe realistic implementations:

- (B) For an adjacency matrix, (1) is $O(1)$, (2) is $\Theta(|V|)$, and a full DFS is $\Theta(|V|^2)$, which is correct.
- (C) For an adjacency list, searching for a specific neighbor and iterating over all neighbors both cost $\Theta(\deg(u))$, and a full DFS runs in $\Theta(|V| + |E|)$.

- (D) For an edge list, we may need to scan all edges for (1) and (2), giving $\Theta(|E|)$, and a full DFS still can be implemented in $\Theta(|V| + |E|)$ time.

Hence only (A) is impossible.

(b) (2') Let $G = (V, E)$ be an undirected simple graph, and let c be the number of connected components. Which statement is true for *all* undirected simple graphs?

A. If G is connected and acyclic, then $|E| = |V|$.

B. If G is acyclic, then $|E| = |V| - c$.

C. If G is acyclic, then $|E| = |V| - 1$.

D. For every graph, $|E| \leq |V| - c$.

Solution: Correct choice: B.

If G is acyclic, each connected component is a tree. For a tree with n_i vertices we have $|E_i| = n_i - 1$. Summing over all components:

$$|E| = \sum_i |E_i| = \sum_i (n_i - 1) = \left(\sum_i n_i\right) - \#\text{components} = |V| - c.$$

The other statements are false:

- (A) For a connected acyclic graph (a tree), $|E| = |V| - 1$, not $|V|$.
- (C) Only holds for connected acyclic graphs (trees), not for general forests with $c > 1$.
- (D) In general we have $|E| \geq |V| - c$; equality holds iff the graph is acyclic.

(c) (2') Let G be an undirected *connected* graph. Run BFS from a source vertex s, obtaining layers $L_0, L_1, L_2, \ldots$, where L_k is the set of vertices at distance k from s. For any edge $\{u, v\} \in E$, which configuration of layers is *impossible*?

A. $u \in L_i,\ v \in L_i$ for some i

B. $u \in L_i,\ v \in L_{i+1}$ for some i

C. $u \in L_i,\ v \in L_{i-1}$ for some $i \geq 1$

D. $u \in L_i,\ v \in L_{i+2}$ for some i

Solution: Correct choice: D.

In BFS on an undirected graph, if u is in layer L_i, then any neighbor v of u must be discovered either:

- no later than L_{i+1} (so v is in L_i or L_{i+1}), and

- the distance difference $|\text{dist}(s,u) - \text{dist}(s,v)|$ is at most 1.

Thus the layer difference for any edge is at most 1. Same-layer edges (A) and adjacent-layer edges (B),(C) are possible, but an edge between L_i and L_{i+2} (D) would contradict the BFS distance property and cannot occur in a correct BFS layering.

(d) (2') Let $G = (V, E)$ be a connected, undirected, weighted graph stored using an adjacency list.

Kruskal's algorithm for computing a minimum spanning tree often uses a *Union-Find* (also called *Disjoint Set Union, DSU*) data structure. A Union-Find maintains a partition of the vertices into disjoint sets and supports the following operations efficiently:

- **Find**(x): return a representative of the set containing vertex x;
- **Union**(x, y): merge the (possibly different) sets containing vertices x and y.

With standard heuristics (union-by-rank and path compression), a sequence of m Union-Find operations on $|V|$ elements runs in time $O(m\,\alpha(|V|))$, where $\alpha(\cdot)$ is the inverse Ackermann function (for the purpose of this problem, you may treat $\alpha(|V|)$ as a very small constant).

Which runtime bound for MST algorithms is correct?

A. Using Kruskal's algorithm with a Union-Find (disjoint set) data structure, the running time is $O(|E| + |V| \log |V|)$.

B. Using Kruskal's algorithm with a Union-Find (disjoint set) data structure, the running time is on the order of $O(|E| \log |E|)$.

C. Using Prim's algorithm with a binary heap (priority queue), the running time is $O(|V|^2)$.

D. Using Prim's algorithm with a Fibonacci heap, the running time is $O(|E| \log |E|)$.

Solution: Correct choice: B.

For Kruskal's algorithm:

- We first sort all edges by weight, which costs $O(|E| \log |E|)$.
- Then we scan the edges in non-decreasing order, using a Union-Find data structure to test whether adding an edge would form a cycle. Each Union-Find operation (find/union) is almost constant time (amortized $O(\alpha(|V|))$), so the total cost of these operations is $O(|E|\alpha(|V|))$, which is dominated by the sorting cost.

Therefore, the overall running time is $O(|E| \log |E|)$.

The other options are inaccurate:

- (A) $O(|E| + |V| \log |V|)$ is closer to the bound for Prim's algorithm with a Fibonacci heap, not for Kruskal's algorithm.
- (C) With an adjacency list and a binary heap, Prim's algorithm runs in $O(|E| \log |V|)$, not $O(|V|^2)$. The bound $O(|V|^2)$ corresponds to Prim with an adjacency matrix and no heap.
- (D) Prim with a Fibonacci heap runs in $O(|E| + |V| \log |V|)$, not $O(|E| \log |E|)$.

Hence only (B) is correct.

(e) (2') Consider running DFS on a directed graph $G = (V, E)$. We classify each directed edge (u, v) encountered during DFS as follows:

- **Tree edge**: (u, v) where v is first discovered by exploring (u, v).
- **Back edge**: (u, v) where v is an ancestor of u in the DFS tree.
- **Forward edge**: (u, v) where v is a *proper descendant* of u in the DFS tree, but (u, v) is not a tree edge.
- **Cross edge**: Any other edge between vertices in different DFS subtrees (neither ancestor nor descendant).

Which statement is correct?

A. If in some DFS of G *no* back edge is produced, then G must be a DAG.

B. If in some DFS of G no cross edge is produced, then G must be a DAG.

C. If in some DFS of G no forward edge is produced, then G must be a DAG.

D. If in some DFS of G at least one back edge is produced, then G can still have a valid topological ordering.

Solution: Correct choice: A.

In a directed graph, a back edge (u, v) is precisely an edge from a vertex to one of its ancestors in the DFS tree. Such an edge closes a directed cycle. Therefore:

- If *any* DFS produces a back edge, the graph contains a directed cycle.
- Equivalently, if some DFS produces *no* back edges, the graph has no directed cycle, i.e., it is a DAG.

Thus (A) is true.

The other statements are incorrect:

- (B) and (C): The absence of cross edges or forward edges alone does not rule out directed cycles.

- (D): If a DFS produces a back edge, the graph has a directed cycle and therefore cannot have a valid topological ordering.

2. (9 points) Multiple Choices

Each question has **one or more** correct answers. Select **all** correct answers. You will receive **no credit** if you select any wrong choice for a question.

(a) (3') Let G be a connected undirected weighted graph, and let T be an MST of G.

A **cut** is a partition $(S, V \setminus S)$ with $S \neq \emptyset$ and $S \neq V$. An edge (u, v) *crosses* the cut if one endpoint is in S and the other in $V \setminus S$.

Which of the following statements are always true?

A. For any cut $(S, V \setminus S)$, *any* minimum-weight edge crossing this cut appears in *some* MST of G.

B. If an edge e is the *unique* minimum-weight edge crossing some cut $(S, V \setminus S)$, then e must appear in *every* MST of G.

C. In any spanning tree, adding an edge that is not in the tree creates exactly one simple cycle; removing any edge on that cycle yields another spanning tree.

D. If in some simple cycle C, an edge $e \in C$ has weight strictly larger than the weights of all other edges in C, then e may appear in *some* MST of G.

Solution: Correct choices: A, B, C.

These are standard **cut** and **cycle** properties of MSTs:

- (A) For any cut, each minimum-weight edge crossing that cut is *safe*, i.e., it can be included in some MST.
- (B) If a crossing edge is the *unique* lightest (strictly smallest weight) edge across a cut, then it must appear in *every* MST.
- (C) Any spanning tree on n vertices has exactly $n - 1$ edges. Adding a non-tree edge creates exactly one simple cycle. Removing any edge on that cycle restores acyclicity while preserving connectivity, so we again get a spanning tree.

Statement (D) is **false**. The cycle property says the opposite: if an edge e is strictly heavier than all other edges on some simple cycle, then e can be safely *excluded* from *every* MST (it certainly does not have to appear in all MSTs). So (D) contradicts the cycle property.

Thus exactly A, B, C, and E are always true.

(b) (3') Let $G = (V, E)$ be a directed graph. Consider the **in-degree based** topological

sorting algorithm from lecture (repeatedly remove a vertex of in-degree 0).

Which of the following statements are correct?

A. If this algorithm outputs all $|V|$ vertices, then G has no directed cycle.

B. If at some step there is no vertex of in-degree 0 but some vertices have not been output yet, then G contains a directed cycle.

C. If during the algorithm there is a step at which there is exactly one vertex of in-degree 0, then the topological ordering is unique.

D. If G is a DAG, then no matter which in-degree-0 vertex we pick at each step, the final topological ordering is always the same.

Solution: Correct choices: A, B.

- (A) If the algorithm successfully outputs all vertices, then we never get stuck with only positive in-degrees. This implies no directed cycle remains at any point, so G is a DAG.
- (B) If at some step there is no vertex of in-degree 0 but some vertices remain unoutput, then every remaining vertex has an incoming edge from within the remaining set, which guarantees a directed cycle.

The incorrect statements:

- (C) is **false** because having exactly one in-degree-0 vertex at *one* step does not force uniqueness of the entire ordering. Uniqueness requires that at *every* step there be exactly one available choice; having a step with a unique choice is not sufficient.
- (D) is **false**: even for a DAG, if at some step there are two or more vertices of in-degree 0, choosing different vertices can lead to different valid topological orders.

Therefore exactly A, B, and E are correct.

(c) (3') Let G be an **undirected connected** graph. Run BFS from a start vertex s to obtain layers $L_0, L_1, L_2, \ldots$, and color each vertex by layer parity: even layers $L_0, L_2, L_4, \ldots$ are *red*, odd layers $L_1, L_3, L_5, \ldots$ are *blue*.

Which of the following statements are correct?

A. If for some start vertex s the above BFS coloring process finishes without finding any edge with two endpoints of the same color, then G is bipartite.

B. If G is bipartite, then for any choice of start vertex s, the above BFS process will produce exactly the same partition of V.

C. If G contains an odd-length cycle, then there exists a choice of start vertex s such that the above BFS coloring process never discovers an edge whose endpoints have the same color.

D. If for some start vertex s the BFS coloring process never produces a same-color edge, then G contains no odd-length cycle.

Solution: Correct choices: A,B, D.

- (A) If BFS coloring from s produces no same-color edge, then every edge connects a red vertex to a blue vertex. This gives a valid 2-coloring, so G is bipartite.
- (B) Since it is connected, the partition is uniquely determined.
- (D) In an undirected connected graph, "bipartite" is equivalent to "having no odd-length cycle." If for some s the BFS process produces a valid 2-coloring (no same-color edge), then G is bipartite and therefore has no odd cycle.

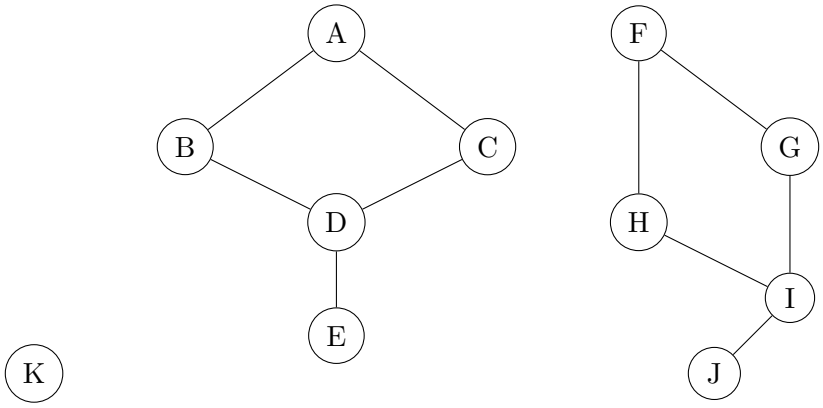
The incorrect statements:

- (C) is **false**. If G contains an odd-length cycle and is connected, then from *any* start vertex s, some edge of that odd cycle will eventually connect two vertices at the same parity level in the BFS layering, creating a same-color edge. There is no choice of s that avoids this.

Thus exactly A and D are correct.

3. (5 points) DFS, BFS, and Connectivity

We work with the following **undirected** graph $G = (V, E)$.



Throughout this question we assume that the vertices are ordered alphabetically: $A, B, \ldots, K$. When scanning a vertex's neighbors, we also consider neighbors in **alphabetical order**.

We use the following DFS and BFS procedures, both of which maintain a global counter `time`

and a **component mark** `comp`$[v]$ for every vertex.

Algorithm 1 DFS-ALL with pre/post numbers and component marks

```
1: time ← 1
2: compID ← 0
3: for each vertex v in alphabetical order do
4:     visited[v] ← false
5:     comp[v] ← 0
6: end for
7: for each vertex v in alphabetical order do
8:     if visited[v] = false then
9:         compID ← compID + 1
10:        DFS-Visit(v, compID)
11:    end if
12: end for
```

Algorithm 2 DFS-Visit$(u, \texttt{id})$

```
1: visited[u] ← true
2: comp[u] ← id
3: pre[u] ← time; time ← time + 1
4: for each neighbor w ∈ Γ(u) in alphabetical order do
5:     if visited[w] = false then
6:         DFS-Visit(w, id)
7:     end if
8: end for
9: post[u] ← time; time ← time + 1
```

(a) (2') Run **DFS-ALL** on the above graph and write down the pre-, post- and component numbers for each vertex in the table below.

Vertex v	pre$[v]$	post$[v]$	comp$[v]$
A	1	10	1
B	2	9	1
C	4	5	1
D	3	8	1
E	6	7	1
F	11	20	2
G	12	19	2
H	14	15	2
I	13	18	2
J	16	17	2
K	21	22	3

Solution: Using DFS-ALL with alphabetical tie-breaking, we first explore the component $\{A, B, C, D, E\}$ starting from A (marked as component 1), then the component $\{F, G, H, I, J\}$ starting from F (marked as component 2), and finally the isolated vertex K (marked as component 3). The resulting pre/post/component numbers are shown in the table above.

(b) (1') Using the DFS execution above (or otherwise), determine the number of **connected components** in the graph.

Solution: DFS-ALL starts a new DFS-Visit only when it encounters a vertex that is still unvisited in the outer loop. This happens three times: once at A, once at F, and once at K. Therefore the graph has exactly $\boxed{3}$ connected components.

(c) (2') Now run **BFS-ALL** on the same graph (with the BFS pseudocode below), again breaking ties alphabetically, and write down the pre-, post- and component numbers for each vertex in the table below.

Vertex v	pre$[v]$	post$[v]$	comp$[v]$
A	1	4	1
B	2	6	1
C	3	7	1
D	5	9	1
E	8	10	1
F	11	14	2
G	12	16	2
H	13	17	2
I	15	19	2
J	18	20	2
K	21	22	3

Solution: In BFS-ALL we again first traverse the component $\{A, B, C, D, E\}$ (marked as component 1), then $\{F, G, H, I, J\}$ (component 2), and finally the isolated vertex K (component 3). We assign `pre`$[v]$ when v is first discovered (enqueued), `post`$[u]$ when we finish scanning all neighbors of u, and `comp`$[v]$ according to the current `compID`. The resulting pre/post/component values are shown in the table above.

4. (5 points) MST Execution

In this question we work with the undirected, weighted graph G shown below. All edges are

Algorithm 3 BFS-ALL with pre/post numbers and component marks

```
1: time ← 1 ;compID ← 0
2: for each vertex v in alphabetical order do
3:     visited[v] ← false
4:     comp[v] ← 0
5: end for
6: for each vertex s in alphabetical order do
7:     if visited[s] = false then
8:         compID ← compID + 1
9:         visited[s] ← true
10:        comp[s] ← compID
11:        pre[s] ← time; time ← time + 1
12:        enqueue s into queue Q
13:        while Q not empty do
14:            dequeue u from Q
15:            for each neighbor w ∈ Γ(u) in alphabetical order do
16:                if visited[w] = false then
17:                    visited[w] ← true
18:                    comp[w] ← compID
19:                    pre[w] ← time; time ← time + 1
20:                    enqueue w into Q
21:                end if
22:            end for
23:            post[u] ← time; time ← time + 1
24:        end while
25:    end if
26: end for
```

undirected. When there is a tie between multiple edges of the same weight, we always pick the edge whose *unordered* pair (u, v) is lexicographically smallest (first compare the smaller endpoint, then the larger one).

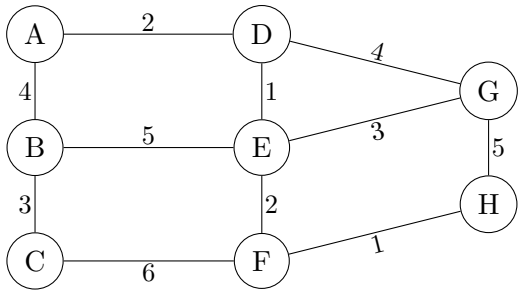
(a) (2') In the graph above, suppose we run **Kruskal's algorithm** to compute an MST. What would be the **fourth** and **fifth** edges added to the MST?

Fourth edge: _____ Fifth edge: _____

Solution: Answer: Fourth edge: $\boxed{E-F}$; Fifth edge: $\boxed{B-C}$.

Sorted by weight (then lexicographically on the unordered pair (u, v)):

$(D, E), (F, H), (A, D), (E, F), (B, C), (E, G), (A, B), (D, G), (B, E), (G, H), (C, F)$.



Kruskal adds an edge iff its endpoints are in different components. The sequence of edges *added* is

$$(D,E),\ (F,H),\ (A,D),\ (E,F),\ (B,C),\ (E,G),\ (A,B),$$

so the 4th and 5th added edges are $E-F$ and $B-C$.

(b) (3') Now run **Prim's algorithm** on the same graph, starting from vertex A. Complete the following table showing the entire execution of Prim's algorithm until all vertices are included in the tree.

Step	Edge added	New vertex	Tree vertices after this step
1	$A-D$	D	$\{A,D\}$
2			
3			
4			
5			
6			
7			

Solution: Execution of Prim's algorithm from A:

Step	Edge added	New vertex	Tree vertices after this step
1	$A-D$	D	$\{A,D\}$
2	$D-E$	E	$\{A,D,E\}$
3	$E-F$	F	$\{A,D,E,F\}$
4	$F-H$	H	$\{A,D,E,F,H\}$
5	$E-G$	G	$\{A,D,E,F,H,G\}$
6	$A-B$	B	$\{A,B,D,E,F,H,G\}$
7	$B-C$	C	$\{A,B,C,D,E,F,H,G\}$

Thus the MST constructed by Prim has the same set of edges as in part (a).

5. (11 points) Campus Network

The CS101A(H) teaching team wants to build a wired network across campus so that every building can reach at least one *CS101A lab server*. We model the campus as a connected,

undirected, weighted graph $G = (V, E, w)$, where each vertex $v \in V$ is a building, and each edge $(u, v) \in E$ is a possible cable between two buildings with installation cost $w(u, v) \geq 0$.

There are only k buildings that can host CS101A lab servers, represented by a subset $L \subseteq V$. We must choose a set of cables $R \subseteq E$ to actually install, so that:

- For every vertex $v \in V$, there exists some lab $\ell \in L$ such that v and ℓ are in the same connected component of the subgraph (V, R); that is, every building can reach at least one CS101A lab.
- The total installation cost $\sum_{(u,v)\in R} w(u, v)$ is minimized.

Note that the final network does *not* need to connect different lab buildings to each other: different labs may end up in different connected components.

(a) (4') Give a precise and succinct description of an **efficient algorithm** that outputs an optimal set R. You may use any MST algorithm (e.g. Kruskal or Prim) as a subroutine. Your description should clearly explain:

- what graph you run MST on,
- how you post-process the MST to obtain the final set R.

Solution: One convenient algorithm is to add a *virtual node* that connects to all labs with zero-cost edges, and then compute an MST on this augmented graph.

1. Construct a new graph
$$G^* = (V^*, E^*, w^*),$$
where:
   - $V^* = V \cup \{s\}$, adding a new virtual node s;

- $E^* = E \cup \{(s, \ell) : \ell \in L\}$;
- the weights are

$$w^*(e) = \begin{cases} w(e) & \text{if } e \in E, \\ 0 & \text{if } e = (s, \ell) \text{ for some } \ell \in L. \end{cases}$$

2. Run any MST algorithm on G^* to obtain a minimum spanning tree T^* of G^*.
3. Delete the virtual node s and all edges incident to s from T^*. Let R be the remaining set of edges:

$$R \;=\; E(T^*) \setminus \{(s, \ell) : \ell \in L\}.$$

4. Output R.

Because every connected component of $T^* \setminus \{s\}$ must contain at least one lab (it was attached to s by a zero-cost edge), the graph (V, R) is a forest in which every component contains at least one lab. Moreover, any feasible solution for the original problem can be turned into a spanning tree of G^* by adding zero-cost edges from s to one lab in each component, so the MST T^* has minimum possible total cost. Therefore the induced forest R has minimum total installation cost in the original graph.

(b) (2') Let $n = |V|$, $m = |E|$, and $k = |L|$. What is the runtime of your algorithm from part (a) in big-O notation (in terms of n, m, k)? Please give a brief explanation.

Solution:

- Constructing the augmented graph G^* adds one new vertex and k new edges, which takes $O(n + m + k)$ time to set up adjacency lists.
- Running an MST algorithm (e.g. Kruskal or Prim) on G^* with $|V^*| = n + 1$ vertices and $|E^*| = m + k$ edges takes

$$O\big((m + k) \log(n + 1)\big) = O\big((m + k) \log n\big)$$

time.

- Deleting the virtual node s and its incident edges from T^* takes $O(k)$ time.

Thus the total runtime is

$$O\big((m+k)\log n\big),$$

which is asymptotically dominated by the MST computation. The dependence on k comes only from the additional k zero-cost edges incident to the virtual node.

We have already built a minimum-cost CS101A campus network as in the previous question. Now the network provider makes the following change:

- For one particular cable $(x, y) \in E$, its installation cost decreases from $w(x, y)$ to a strictly smaller value $w'(x, y) < w(x, y)$. The costs of all other edges remain unchanged.

Let $G' = (V, E, w')$ be the new weighted graph after this price change. We would like to update our routing plan to reflect the new prices, but we do *not* want to run an MST algorithm on G' from scratch.

(c) (4') Design an efficient algorithm that takes the original graph $G = (V, E, w)$ and the result from the last question, as well as the edge (x, y) and its new weight $w'(x, y)$. And the algorithm should outputs an optimal set for G'.

Solution: High-level algorithm:

Assume we still keep an MST T of the original graph (for example, the tree T^* from the augmented graph with the virtual node, or an MST of G itself); then we can update it without recomputing from scratch.

1. Update the weight of (x, y) in the graph from $w(x, y)$ to $w'(x, y)$.
2. **Case 1:** $(x, y) \in T$.
   - Simply change the stored weight of this edge in T from $w(x, y)$ to $w'(x, y)$.
   - Return the same tree $T' = T$ (with the new edge weight). Since we only made an MST edge cheaper, T is still an MST of G'.

3. **Case 2:** $(x, y) \notin T$.

   (a) Temporarily add edge (x, y) to T. This creates a unique simple cycle C in the graph $T \cup \{(x, y)\}$.

   (b) Find the edge $e_{\max}$ on this cycle with the maximum weight (w.r.t. the *new* weights w'):

   $$e_{\max} = \arg\max_{e \in C} w'(e).$$

   (c) If $w'(x, y) < w'(e_{\max})$, then set

   $$T' \leftarrow \big(T \cup \{(x, y)\}\big) \setminus \{e_{\max}\}.$$

   That is, insert (x, y) and delete the heaviest edge on the cycle.

   (d) Otherwise (if $w'(x, y)$ is not smaller than the heaviest edge on the cycle), keep the original tree: $T' \leftarrow T$.

4. Output T'.

Intuitively, when (x, y) is not in T, adding it forms a cycle; if (x, y) is lighter than the heaviest edge on that cycle, we can swap them to obtain a cheaper spanning tree; otherwise no improvement is possible.

(d) (1') Let $n = |V|$ and $m = |E|$. What is the runtime of your update algorithm (which means you don't need to take previous MST algorithm into consider) from part (a)? Please give a brief explanation and answer in big-O notation.

Solution:

- In Case 1, where $(x, y) \in T$, we simply change the weight stored on that edge: $O(1)$ time.
- In Case 2, where $(x, y) \notin T$, we must:
  - find the unique path between x and y in the tree T;
  - scan this path to find the maximum-weight edge $e_{\max}$.

  Using a simple parent-pointer representation or adjacency lists for T, the path

length is at most $n-1$, so we can walk along the path and track the maximum weight in $O(n)$ time. Deleting and inserting a single edge is $O(1)$.

Thus the worst-case runtime of the update algorithm is

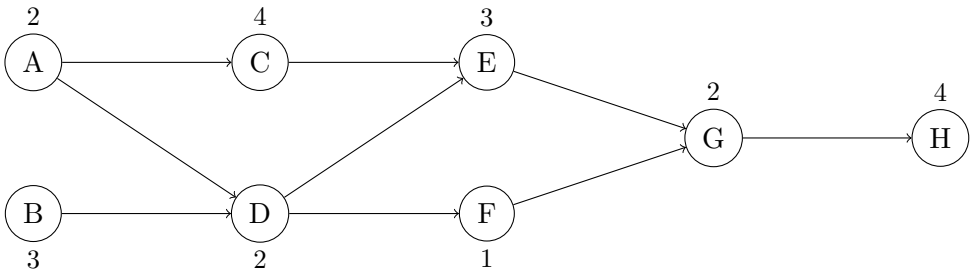
$$\boxed{O(n)},$$

which is asymptotically faster than recomputing an MST from scratch in $O(m \log n)$ time.

6. (5 points) Multi-Stage Project Pipeline

The CS101A(H) teaching team is planning a multi-stage *project pipeline* for the course. Each task depends on some previous tasks, and each task v has a processing time (duration) $\mathrm{dur}(v)$. We model this as a directed acyclic graph $G = (V, E)$, where an edge $(u, v) \in E$ means: *task v can start only after task u has finished*. The duration is attached to the *vertex*.

Below is the DAG for one such plan:



Assume that tasks A and B have no prerequisites and can start at time 0. All other tasks must respect the edge constraints.

For each vertex v, define:

$$\mathrm{EC}[v] := \text{earliest completion time of } v \text{ if we start at time 0,}$$

and let the duration of the whole project be $\mathrm{EC}[H]$, since H is the final deliverable.

(a) (2') Using **topological order** on the DAG, compute the earliest completion time $\mathrm{EC}[v]$ for each task $v \in \{A, B, C, D, E, F, G, H\}$.

You may use the recurrence

$$\mathrm{EC}[v] = \begin{cases} \mathrm{dur}(v), & \text{if } v \text{ has no predecessors,} \\ \mathrm{dur}(v) + \max\limits_{(u,v)\in E} \mathrm{EC}[u], & \text{otherwise.} \end{cases}$$

Fill in the table:

Task v	EC[v]
A	
B	
C	
D	
E	
F	
G	
H	

Solution: Compute in a topological order, e.g. A, B, C, D, E, F, G, H:

$\mathrm{EC}[A] = 2$ (no predecessors)

$\mathrm{EC}[B] = 3$ (no predecessors)

$\mathrm{EC}[C] = \mathrm{dur}(C) + \mathrm{EC}[A] = 4 + 2 = 6$

$\mathrm{EC}[D] = \mathrm{dur}(D) + \max(\mathrm{EC}[A], \mathrm{EC}[B]) = 2 + \max(2, 3) = 5$

$\mathrm{EC}[E] = \mathrm{dur}(E) + \max(\mathrm{EC}[C], \mathrm{EC}[D]) = 3 + \max(6, 5) = 9$

$\mathrm{EC}[F] = \mathrm{dur}(F) + \mathrm{EC}[D] = 1 + 5 = 6$

$\mathrm{EC}[G] = \mathrm{dur}(G) + \max(\mathrm{EC}[E], \mathrm{EC}[F]) = 2 + \max(9, 6) = 11$

$\mathrm{EC}[H] = \mathrm{dur}(H) + \mathrm{EC}[G] = 4 + 11 = 15.$

So the table is:

Task v	EC[v]
A	2
B	3
C	6
D	5
E	9
F	6
G	11
H	15

Total project duration is $\boxed{15}$.

(b) (3') We now want to know which tasks are *critical*, i.e. they lie on at least one longest path from a start task to H. For each task v, define $\mathrm{LC}[v]$ to be the *latest* completion time of v that still allows the project to finish by time $\mathrm{EC}[H]$.

We can compute $\mathrm{LC}[v]$ by processing the DAG *backwards* in a reverse topological order:

$$\mathrm{LC}[v] = \begin{cases} \mathrm{EC}[H], & \text{if } v = H, \\ \min\limits_{(v,u)\in E} \big(\mathrm{LC}[u] - \mathrm{dur}(u)\big), & \text{otherwise.} \end{cases}$$

(i) Compute $\mathrm{LC}[v]$ for all vertices $v \in \{A, B, C, D, E, F, G, H\}$ using the recurrence above.

(ii) List all critical tasks.

Fill in the table:

Task v	EC[v]	LC[v]
A		
B		
C		
D		
E		
F		
G		
H		

Set of critical tasks: { }.

Solution: We already have EC[v] from part (a). Work backwards in a reverse topological order, e.g. H, G, E, F, C, D, A, B.

$$\begin{aligned}
\mathrm{LC}[H] &= \mathrm{EC}[H] = 15. \\
\mathrm{LC}[G] &= \mathrm{LC}[H] - \mathrm{dur}(H) = 15 - 4 = 11. \\
\mathrm{LC}[E] &= \mathrm{LC}[G] - \mathrm{dur}(G) = 11 - 2 = 9. \\
\mathrm{LC}[F] &= \mathrm{LC}[G] - \mathrm{dur}(G) = 11 - 2 = 9. \\
\mathrm{LC}[C] &= \mathrm{LC}[E] - \mathrm{dur}(E) = 9 - 3 = 6. \\
\mathrm{LC}[D] &= \min\big(\mathrm{LC}[E] - \mathrm{dur}(E),\ \mathrm{LC}[F] - \mathrm{dur}(F)\big) \\
&= \min(9-3,\ 9-1) = \min(6, 8) = 6. \\
\mathrm{LC}[A] &= \min\big(\mathrm{LC}[C] - \mathrm{dur}(C),\ \mathrm{LC}[D] - \mathrm{dur}(D)\big) \\
&= \min(6-4,\ 6-2) = \min(2, 4) = 2. \\
\mathrm{LC}[B] &= \mathrm{LC}[D] - \mathrm{dur}(D) = 6 - 2 = 4.
\end{aligned}$$

So the table is:

Task v	EC[v]	LC[v]
A	2	2
B	3	4
C	6	6
D	5	6
E	9	9
F	6	9
G	11	11
H	15	15

A task is critical if $\mathrm{LC}[v] = \mathrm{EC}[v]$, so the critical tasks are

$$\{A, C, E, G, H\}.$$

One critical path is $A \to C \to E \to G \to H$ with total duration $2+4+3+2+4 = 15$.

(c) (**Bonus, 4pts**) Now consider a *general* DAG $G = (V, E)$ with a nonnegative duration $\mathrm{dur}(v)$ on each vertex v. Assume you are given:

- a designated start set $S \subseteq V$ of tasks that can begin at time 0;
- a designated final task $t \in V$;
- a topological ordering of all vertices of G.

A task $v \in V$ is called **critical** if it lies on *at least one* longest path (maximum total duration) from some start task in S to t.

Design an $O(|V| + |E|)$-time algorithm that marks exactly the *critical* tasks in G. (You do *not* need to prove correctness formally.)

Solution: We reuse the ideas of EC[·] (earliest completion / longest prefix) and LC[·] (latest completion) from parts (a) and (b), but now on a general DAG with multiple start tasks S and a single final task t.

Step 1: Forward pass (EC in a general DAG).

Process vertices in the given topological order. We define $\mathrm{EC}[v]$ to be the length (total duration) of a longest path from some $s \in S$ to v.

Initialization:

$$\mathrm{EC}[v] = \begin{cases} \mathrm{dur}(v), & v \in S, \\ -\infty, & v \notin S. \end{cases}$$

Transition (when we process v in topological order): for each incoming edge $(u, v) \in E$, we update

$$\mathrm{EC}[v] \leftarrow \max\big(\mathrm{EC}[v],\ \mathrm{EC}[u] + \mathrm{dur}(v)\big).$$

At the end, the length of a longest path from S to t is

$$L^* := \mathrm{EC}[t].$$

(Vertices that are not reachable from S keep $\mathrm{EC}[v] = -\infty$.)

Step 2: Backward pass (LC in a general DAG).

We now define $\mathrm{LC}[v]$ to be the latest completion time of v, under the requirement that the project finishes by time L^* at t. This is exactly analogous to part (b).

Process vertices in *reverse* topological order.

Initialization:

$$\mathrm{LC}[v] = \begin{cases} L^*, & v = t, \\ +\infty, & v \neq t. \end{cases}$$

Transition (when we process v in reverse topological order): for each outgoing edge $(v, u) \in E$, we update

$$\mathrm{LC}[v] \leftarrow \min\big(\mathrm{LC}[v],\ \mathrm{LC}[u] - \mathrm{dur}(u)\big).$$

Vertices that cannot reach t keep $\mathrm{LC}[v] = +\infty$.

Step 3: Mark critical vertices.

A vertex v lies on *some* longest path from S to t if and only if

- it is reachable from S and can reach t: $\mathrm{EC}[v] \neq -\infty$ and $\mathrm{LC}[v] \neq +\infty$; and
- its earliest and latest completion times coincide:

$$\mathrm{EC}[v] = \mathrm{LC}[v].$$

Indeed, if $\mathrm{EC}[v] = \mathrm{LC}[v]$, then v must sit at a fixed time on any schedule that finishes at t by time L^*, and the prefix (from S to v) plus suffix (from v to t) has total length

$$\mathrm{EC}[v] + \big(L^* - \mathrm{LC}[v]\big) = L^*,$$

so there is at least one longest path going through v.

Thus, our algorithm is:

1. Run the forward DP to compute all $\mathrm{EC}[v]$.
2. Run the backward DP to compute all $\mathrm{LC}[v]$.
3. Mark v as **critical** if $\mathrm{EC}[v] \neq -\infty$, $\mathrm{LC}[v] \neq +\infty$, and $\mathrm{EC}[v] = \mathrm{LC}[v]$.

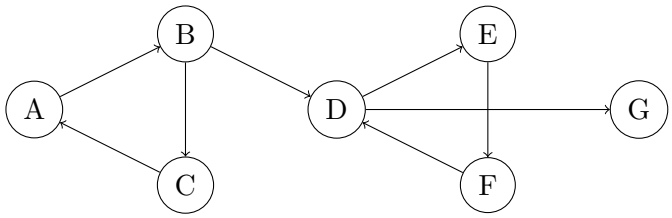
Each edge and each vertex is processed a constant number of times in the two passes and the final scan, so the total running time is

$$O(|V| + |E|).$$

7. (9 points) Strongly Connected Study Groups at ShanghaiTech

Whenever group u figures out a new homework trick, they immediately share it with group v in WeChat.

The directed graph $G = (V, E)$ of study groups is:



We say two groups u and v are **strongly connected** if u can eventually reach v *and* v can eventually reach u by following directed edges. A **strongly connected component (SCC)** is a maximal set of groups where every pair of groups in the set is strongly connected to each other.

A beautiful high-level picture is:

Every directed graph can be decomposed as a DAG of strongly connected components.

In this problem, you will see how this works on the study-group graph and then generalize it into an algorithm.

(a) (3') **SCCs and the SCC-DAG on the study-group graph.**

Using reachability (via DFS or BFS) on the graph above:

- List all SCCs of the graph, each as a set of vertices (study groups).
- Contract each SCC to a single "super-group", draw (or describe) the new graph whose vertices are SCCs and where there is an edge $\mathrm{SCC}_i \to \mathrm{SCC}_j$ if at least one original edge goes from a vertex in SCC_i to a vertex in SCC_j.

- Give one valid topological order of this SCC-graph.

Solution: The SCCs are

$$\{A, B, C\}, \quad \{D, E, F\}, \quad \{G\}.$$

Contracting each SCC gives a 3-vertex graph

$$\{A, B, C\} \longrightarrow \{D, E, F\} \longrightarrow \{G\},$$

which is a DAG. One topological order is

$$\{A, B, C\},\ \{D, E, F\},\ \{G\}.$$

Interpretation: if the rumor starts in the first SCC, it can spread inside that SCC, then move to the second SCC, then to the third, but never in the reverse direction.

(b) (2') **A two-pass DFS experiment on the same graph.**

Consider the following two-step procedure on a directed graph $G = (V, E)$:

Step 1: Run `DFS-ALL` on G (as in earlier questions), and record the finishing time post$[v]$ for every vertex.

Step 2: Build the reversed graph G^R by reversing every edge. Run `DFS-ALL` again on G^R, but in the outer loop visit vertices in *decreasing* order of post$[v]$ from Step 1 (instead of alphabetical order). Every time a new DFS tree starts in Step 2, give a new component ID to all vertices in that tree.

Apply this two-step procedure to the study-group graph above.

- Write the vertices in decreasing order of post$[v]$ after Step 1. (Any order consistent with one valid DFS run is fine.)
- After Step 2 finishes, which vertices share the same component ID? Group them as sets. Compare with the SCCs from part (a): what happens?

Solution: One valid DFS run on G yields decreasing post order

$$G,\ D,\ F,\ E,\ A,\ C,\ B.$$

Running DFS on G^R in that order gives three DFS trees:

$$\{G\}, \quad \{D, E, F\}, \quad \{A, B, C\}.$$

These sets are exactly the SCCs from part (a): each DFS tree in Step 2 matches one strongly connected component, and the components appear in reverse topological order of the SCC-DAG.

(c) (4') **From ShanghaiTech study groups to a general algorithm.**

Now imagine a much larger directed graph built from *all* CS-related group chats at ShanghaiTech. You want an $O(|V| + |E|)$-time program that:

- outputs every SCC of the graph, and
- outputs a topological order of the DAG whose nodes are SCCs.

Describe, at a high level, how you would do this. You may refer to the two-step procedure from part (b) as a building block, and you may also use BFS/DFS/topological sort as black boxes.

Solution: A standard approach is:

1. Run DFS-ALL on G; record all post times post$[v]$.
2. Build the reversed graph G^R.
3. Run DFS-ALL on G^R, visiting vertices in decreasing order of post$[v]$. Each DFS tree from this pass is one SCC; label vertices with SCC IDs.
4. To build the SCC-DAG, create one node for each SCC ID and scan all edges $u \to v$ of G. Whenever the SCC IDs differ, add an edge from SCC(u) to SCC(v) (ignoring duplicates).
5. Finally, run a topological sort on this SCC-DAG to get a topological order of components.

Each step runs in $O(|V| + |E|)$ time, so the whole algorithm is linear.