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 $\Theta(|V|+|E|)$.

(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, $|E| \leq |V| - c$.

(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

(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 $O(|E| \log |E|)$.

(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.

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.

(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.

(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.

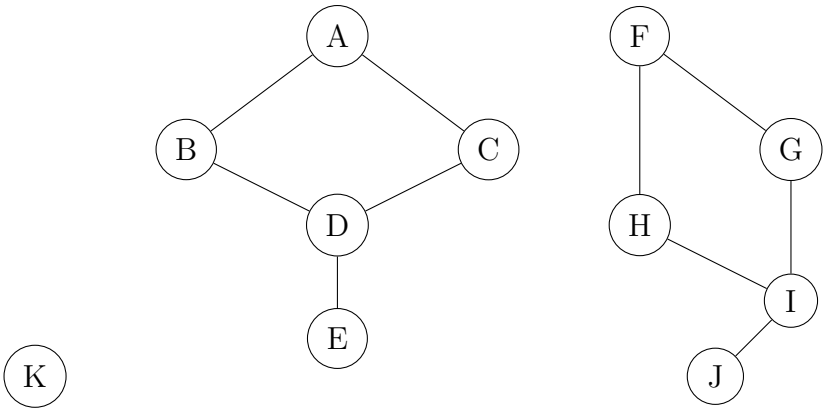
B. If G is bipartite, then for any choice of start vertex s, the above BFS process will produce exactly the same partition of V.

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.

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
```

(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.

Algorithm 2 DFS-Visit(u, `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
```

Vertex v	pre[v]	post[v]	comp[v]
A			
B			
C			
D			
E			
F			
G			
H			
I			
J			
K			

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

(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.

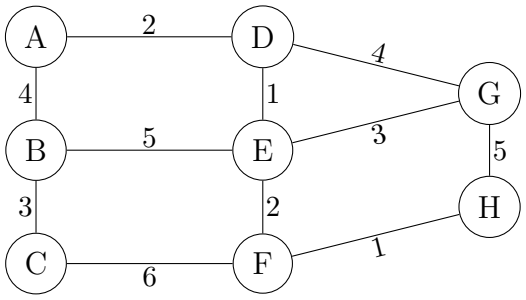
Algorithm 3 BFS-ALL with pre/post numbers and component marks

```
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
```

Vertex v	pre[v]	post[v]	comp[v]
A			
B			
C			
D			
E			
F			
G			
H			
I			
J			
K			

4. (5 points) MST Execution

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

(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			

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.

(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.

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.

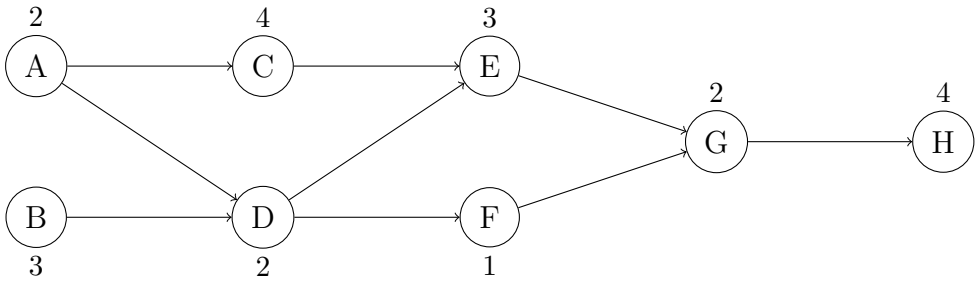
(c) (4') Design an efficient algorithm that takes the original graph $G = (V, E, w)$ and the result from the last question, as well as the edge (x, y) and its new weight $w'(x, y)$. And the algorithm should outputs an optimal set for G'.

(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.

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) $\text{dur}(v)$. 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	$\mathrm{EC}[v]$
A	
B	
C	
D	
E	
F	
G	
H	

(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: { }.

(c) (**Bonus, 4pts**) Now consider a *general* DAG $G = (V, E)$ with a nonnegative duration $\text{dur}(v)$ 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.

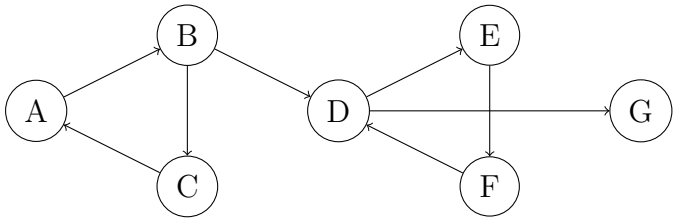
A task $v \in V$ is called **critical** if it lies on *at least one* longest path (maximum total duration) from some start task in S to t.

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

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 $\text{SCC}_i \to \text{SCC}_j$ 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.

(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?

(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.