CSC263H5 Teaching Assistant Application

Alexander He Meng

University of Toronto Mississauga

Submitted: November 29, 2025

Question 1: TA Role Preferences

Question

The main TA roles for this course are:

- Grader (grade assignments/exams, provide feedback)
- Lecture TA (answer questions during in-class worksheets)
- Tutorial TA (run tutorial sessions)

Are there any roles you are not willing to do? Grading is the only one that can be done remotely. (There will be assorted hours for other tasks like invigilation or piazza.)

Answer

I am willing and enthusiastic to perform all three TA roles. I have experience as a grader, which has strengthened my ability to provide clear, constructive feedback efficiently. I particularly enjoy the interactive aspects of being a Lecture TA and Tutorial TA, as they allow me to engage directly with students and clarify concepts in real-time. I am comfortable with both in-person and remote responsibilities, including invigilation and Piazza support.

Question 2: Big O and Big Ω Misconception

Question

Imagine a student asks: "Since big O is worst case, is big Ω best case?" Answer their question in at most two paragraphs.

Answer

This is a common misconception. Big O, big Ω, and big Θ describe the *growth rate* of functions, not specific cases of an algorithm's behavior. Big O provides an *asymptotic upper bound*, meaning $f(n) = O(g(n))$ if f grows no faster than g for sufficiently large n. Big Ω provides an *asymptotic lower bound*, meaning $f(n) = \Omega(g(n))$ if f grows at least as fast as g. Big Θ describes a tight bound when a function is both $O(g(n))$ and $\Omega(g(n))$.

When we analyze algorithms, we separately consider best-case, average-case, and worst-case scenarios, and for each scenario we can use big O, Ω, or Θ notation. For example, quicksort has worst-case runtime $\Theta(n^2)$ and best-case runtime $\Theta(n \log n)$. Both of these statements use Θ notation to describe tight bounds on the runtime for different input scenarios. The confusion arises because we often focus on worst-case analysis using big O, but the notations themselves are about growth rates, not input cases.

Question 3: Error in MST Proof

Question

Definitions: Let $G = \langle V, E, w \rangle$ be an undirected, connected, weighted graph.

A path in G is a sequence of vertices $v_1, \ldots, v_k$ where $\langle v_i, v_{i+1} \rangle \in E$ for all $1 \leq i < k$. We say the path is a path from s to t if $v_1 = s, v_k = t$. The length of the path is $\sum_{i=1}^{k-1} w(\langle v_i, v_{i+1} \rangle)$.

A spanning tree T of G is a connected, acyclic subgraph of G. The cost of T is $\sum_{e \in T} w(e)$.

A minimum spanning tree M of G is a spanning tree satisfying: for any spanning tree T, $\text{cost}(M) \leq \text{cost}(T)$.

Claim: A minimum spanning tree of a graph always contains the shortest path from s to t.

Proof: Let M be a MST of G. Let p be a shortest path from s to t in G, and let q be the path from s to t in M. Suppose $\text{length}(p) \leq \text{length}(q)$. Then $p \neq q$, so there is an edge in q that is not in p. Remove that edge. M is now disconnected.

Because p is a path from s to t, there must be at least one edge in p that will connect M, so add that edge to M to create a new minimum spanning tree M'.

Because $\text{length}(p) \leq \text{length}(q)$, $\text{cost}(M') \leq \text{cost}(M)$. This is a contradiction with the fact that M is a minimum spanning tree of G.

Answer

The error occurs in the claim that removing an edge from q and adding an edge from p creates a new spanning tree M' with $\text{cost}(M') \leq \text{cost}(M)$. The proof incorrectly assumes that the entire path p can replace the entire path q edge-by-edge, maintaining the spanning tree property.

The flaw is that we remove only *one* edge from q but may need to add only *one* edge from p to reconnect the tree, and these individual edges may have different weights. The inequality $\text{length}(p) \leq \text{length}(q)$ compares the *total* lengths of the paths, not the individual edge weights. It is entirely possible that the specific edge we add from p has greater weight than the specific edge we removed from q, resulting in $\text{cost}(M') > \text{cost}(M)$. This would not contradict M being a MST.

Furthermore, the claim itself is false. A counterexample: consider a triangle graph with vertices $\{s, t, v\}$ and edges with weights $w(s,t) = 10$, $w(s,v) = 1$, $w(v,t) = 1$. The shortest path from s to t is the direct edge with length 10, but the MST consists of edges $\{(s,v),(v,t)\}$ with total cost 2, which does not contain the shortest path.

Question 4: Amortized Analysis of Data Structure D

Question

You have a data structure D that supports three operations: create, insert and merge.

Create() returns an empty D and takes $O(1)$ time.

Insert(D, x) runs in $O(\log n)$ time, where n is the number of items in D, and it increases the size of D by 1.

Merge(D_1, D_2) runs in $O(\min(n, m))$ time, where n is the number of items in D_1 and m is the number of items in D_2. Merge(D_1, D_2) deletes D_1 and D_2 and creates a new D_3 with size $n + m$.

Give a O-bound on the amortized runtime of n of these operations. A formal proof is not necessary, just describe the worst sequence, explain why it's the worst, and analyze the runtime.

Answer

The worst-case sequence exploits the merge operation's dependence on structure size. Consider this pattern over n operations:

Worst Sequence: Start with $n/2$ Create operations, then $n/4$ Insert operations (one insert into each of $n/4$ structures, creating structures of size 2). Then repeatedly merge pairs of equal-sized structures: merge all pairs of size 2 (cost: $n/8 \cdot O(2) = O(n)$), then merge pairs of size 4 (cost: $n/16 \cdot O(4) = O(n)$), continuing through $O(\log n)$ levels.

Why it's worst: Merging equal-sized structures maximizes total cost. Each merge level processes all n elements, and we have $O(\log n)$ levels, giving total cost $O(n \log n)$ for the merges alone. Inserts contribute an additional $O(n \log n)$. This is worse than, say, merging one small structure into a large one repeatedly, which would cost $O(n)$ total.

Analysis: For n operations:

- Creates: $O(1)$ each, negligible
- Inserts: at most n inserts cost $\sum_{i=1}^{n} O(\log i) = O(n \log n)$
- Merges: balanced merge tree yields $O(n \log n)$ total

Total cost for n operations: $O(n \log n)$
Amortized bound: $O(\log n)$ per operation.