CS570 Spring2024: Analysis of Algorithms Exam I

	Points		Points
Problem 1	20	Problem 5	10
Problem 2	9	Problem 6	20
Problem 3	9	Problem 7	16
Problem 4	6	Problem 8	10
	Total	**100**	

Instructions:

1. This is a 2-hr exam. Closed book. No electronic devices or internet access. One 8.5x11 cheat sheet allowed.
2. If a description to an algorithm or a proof is required, please limit your description or proof to within 150 words, preferably not exceeding the space allotted for that question.
3. No space other than the pages in the exam booklet will be scanned for grading.
4. If you require an additional page for a question, you can use the extra page provided within this booklet. However please indicate clearly that you are continuing the solution on the additional page.
5. Do not detach any sheets from the booklet. Detached sheets will not be scanned.
6. If using a pencil to write the answers, make sure you apply enough pressure, so your answers are readable in the scanned copy of your exam.
7. Do not write your answers in cursive scripts.
8. This exam is printed double sided. Check and use the back of each page.

1) 20 pts
Mark the following statements as **TRUE** or **FALSE** by circling the correct answer. No need to provide any justification.

[TRUE/FALSE]
A bipartite graph cannot contain an odd-length cycle.

[TRUE/FALSE]
There exists an algorithm with worst-case running time complexity $O(\log n)$ for finding the smallest element of a binary max-heap with n elements.

[TRUE/FALSE]
There exists an algorithm with worst-case running time complexity $O(n)$ for merging two binomial min-heaps of size n.

[TRUE/FALSE]
Removing any element from a binary heap (not necessarily the root) and then reheapifying to maintain the heap property can be accomplished in $O(\log n)$ time, where n is the number of elements in the heap.

[TRUE/FALSE]
If a Directed Acyclic Graph G is weakly connected and has $n \geq 2$ vertices and n-1 edges, then G must have a unique topological ordering.

[TRUE/FALSE]
Dijkstra's algorithm will work correctly as long as the graph has no more than one negative weight edge.

[TRUE/FALSE]
Every tree is a bipartite graph.

[TRUE/FALSE]
The amortized cost of an operation can be lower than the worst-case cost of that operation.

[TRUE/FALSE]
The amortized cost of an operation can be higher than the worst-case cost of that operation.

[TRUE/FALSE]

If $f(n) = \Omega(g(n))$ and $g(n) = O(h(n))$, then $f(n) = O(h(n))$.

2) 9 pts (3 pts each)
Circle ALL correct answers (no partial credit when missing some of the correct answers). No need to provide any justification.

i- In a graph G with n nodes and n+5 edges, where $k \geq 7$ edges have the same weight, every minimum spanning tree of G will have at least…

a) 2 edges with the same weight.
b) 3 edges with the same weight.
c) 4 edges with the same weight.
d) None of the above

ii- What is the solution to the recurrence equation $T(n) = 2T(n/2) + 3n + \text{sqrt}(n)$?

a) $T(n) = \Theta(n \text{ sqrt}(n))$
b) $T(n) = \Theta(n \log^2 n)$
c) $T(n) = \Theta(n \log n)$
d) None of the above

iii- What is the solution to the recurrence $T(n) = 2T(n/2) + 3n \log n + 2n$?

a) $T(n) = \Theta(n \log n)$
b) $T(n) = \Theta(n)$
c) $T(n) = \Theta(n \log^2 n)$
d) $T(n) = \Theta(n^2)$

3) 9 pts

For each of the following algorithms, use the Master Method to determine its running time in terms of big-O, and if the Master Method cannot be applied explain why.

a. Algorithm *A*, which solves problems of size *n* by dividing them into five subproblems of half the size, recursively solving each subproblem, and then combining the solutions to the subproblems in linear time.
b. Algorithm *B*, which solves problems of size *n* by recursively solving two subproblems each of size *n*-1 and then combining the solutions in constant time.
c. Algorithm *C*, which solves problems of size *n* by dividing them into nine subproblems of size *n*/3, recursively solving each subproblem, and then combining the solutions in $O(n^2)$ time.

The running times T_A, T_B, T_C of the algorithms satisfy the following recurrence relations: $T_A(n) = 5T(n/2) + O(n)$

$T_B(n) = 2T(n-1) + O(1)$

$T_C(n) = 9T(n/3) + O(n^2)$.

Functions T_A and T_C can be found by the Master Theorem :

$T_A(n) = O(n^{\log_2(5)})$

$T_C(n) = O(n^{\log_3(9)}) = O(n^2 \log n)$.

For T_B we have $T_B(n) = 2T_B(n-1) + O(1) = 2^2 TB(n-2) + 2 \cdot O(1) = 2^3 T(n-3) + 3 \cdot O(1) = \cdots = 2^n T(1) + n \cdot O(1)$.

Thus, $T_B(n) = O(2^n)$.

4) 6 pts (1 pt per correct answer)
Select all choices that correctly describe the worst-case running time complexity of the following code.

```
int a = 0;
for (i = 0; i < n; i++) {
     for (j = n; j > i; j--) {
          a = a + i + j;
     }
}
```

a) $O(n^2)$
b) $\Theta(n^2 \log n)$
c) $\Omega(n^2 \log n)$
d) $O(n^2 \log n)$
e) $\Omega(n^2)$
f) $\Theta(n^2)$

5) 10 pts

Consider a collection of n integer-valued variables $x_1, ..., x_n$ and a collection of m constraints, where in each constraint, two variables are constrained to be equal (e.g. $x_i=x_j$) or unequal (e.g. $x_i \neq x_j$).

Depending on the constraints, it may be impossible to assign values to the variables without violating at least one constraint. For example, the following collection of constraints cannot all be satisfied simultaneously: $x_1=x_2$, $x_2=x_3$, $x_3=x_4$, $x_4 \neq x_2$.

Design an algorithm (hint: graph based) that takes as input m constraints over n variables and decides in $O(m+n)$ time whether the given set of constraints can be satisfied. Analyze the worst-case running time complexity of your algorithm.

5 points for solutions that run in $O(m^2)$

Let G = (V, E) be the following graph: G has a node corresponding to every variable x_i, and there is an edge (x_i, x_j) if $x_i = x_j$ is an equality constraint.

Full Points: We do a Depth First Search of G to find all its connected components. For each node X_i, if it is in the k-th connected component of G, we assign it a mark k.

Now for each inequality constraint $x_i \neq x_j$, we check to see if x_i and x_j occur in different connected components of *G*. If there is some inequality constraint $x_i \neq x_j$ such that x_i and x_j occur in the same connected component of G, then we report that the constraints are unsatisfiable. If there is no such constraint, then we report satisfiable.

Finding all connected components takes O(m+n). checking all inequality takes O(m).

Overall complexity is O(m+n)

Common major errors:

1) Including all constraints as edges and not just for equality constraints. These attempts typically follow up with 2) below.
2) Finding cycles (any cycles or odd-length cycles) - cycles containing zero or two or more inequality constraints does not force insatisfiability, only the ones with exactly one inequality do force it. Finding just any cycle is easy with BFS/DFS but finding one with certain property (e.g. Number of edges of certain type) is not easy, certainly not with brute force.
3) "Processing" all constraints in whichever order. This will not work, ALL equality constraints need to be taken into account first and THEN check for satisfiability of inequalities.
4) BFS/DFS on weighted graphs
5) Missing the fact that we may get multiple connected components from BFS/DFS and not just one tree.
6) Use of directed edges - unnecessary and often incorrect

5 Points: For each inequality $x_i \neq x_j$ run BFS or DFS from x_j to see if x_i is reachable. If it is then the set of constraints are not satisfiable. Run time = O(m*(m+n)) = $O(m^2)$

2 Points: Construct a graph containing N nodes, the i-th node is corresponding to the integer x_i. Only for each equality constraint $x_i = x_j$, draw an undirected edge from x_i to x_j.

Before applying for regrading Q5, please check for common incorrect approaches listed above - it would help to check if your algorithm can pass the following examples correctly.
[1] x1 noeq x2, x2 noeq x3, x3 noeq x1 (this graph is not bipartie)
[2] x1 eq x2, x3 eq x4, x1 noeq x3, x2 noeq x4 (there is a cycle, but the graph is valid)
[3] x1 eq x3, x2 eq x3, x2 eq x4, x4 noeq x1 (no loop in directed graph)
[4] x1 eq x2, x3 eq x4, x2 eq x4, x4 noeq x1 (sequentially coloring will cause problem when handling x2 eq x4)

6) 20 pts
Consider the Minimum Leaf-Constrained Spanning Tree (MLCST) Problem, defined below:

Given an undirected weighted graph $G = (V, E, w)$ and a subset of vertices $U \subseteq V$, find the least-weight spanning tree T in G in which each vertex in U is a leaf of T. If no such tree exists, your algorithm should indicate so.

a. Design an algorithm for this problem with worst-case running time $O(|E|\log|V|)$. (10 pts)
b. Analyze the worst-case running time complexity of your algorithm. (5 pts)
c. Give an example of a graph G and a set of vertices $U \subseteq V$ for which such an MLCST does not exist. (5 pts)

Solution:
A MLCST T of G consists of a least-weight edge incident to u and another vertex v $\in$ V - U for each vertex u $\in$ U, and the edges of a MST of G - U.

Consider an arbitrary MLCST T of G. For each vertex u $\in$ U, u is a leaf of T, so T contains exactly one edge incident to u. If this edge is not an edge of least-weight among those incident to u and some other vertex v $\in$ V - U, then removing this edge from T and adding a least-weight edge incident to u and a vertex v $\in$ V - U yields a LCST of G with

less weight than T, a contradiction. Therefore, if T is a MLCST of G, then for each vertex $u \in U$, T contains exactly one edge incident to u, and that edge is a least-weight edge among those incident to u and a vertex $v \in V - U$. If for some $u \in U$, there does not exist an edge that is incident to u and another vertex $v \in V - U$, then G does not have any MLCST.

Let EU be the set of edges in T that are incident to some vertex $u \in U$. Since each $u \in U$ is a leaf of T, we must have that T - EU is a MST of G - U, since if there exists a different spanning tree T' of G with less weight, then $T' \cup EU$ is a LCST of G with less weight than T, a contradiction. If G - U is not connected and so does not have any MST, then G does not have any MLCST.

Algorithms:
Approach I (removing U from G)
If U = V, then a LCST is possible iff V (or U) contains at most 2 vertices and G is connected. The MLCST is the graph G itself. Otherwise, a LCST is not possible.
If $U \neq V$, then remove all vertices $u \in U$ from G. If G - U is disconnected, then a LCST is not possible.
Otherwise, find a MST T of G - U using Prim's or Kruskal's algorithm.
For each vertex $u \in U$, let Eu be the set of edges $(u, v) \in E$ such that $v \in V - U$. If $Eu = \emptyset$ for any u, then a LCST is not possible.
For each vertex u, pick a least-weight edge from Eu and add it to T. The final tree is a MLCST.
Approach II (modified Kruskal)
Same as Approach I
If $U \neq V$, then we use Kruskal's algorithm to find the MLCST. Sort all edges in ascending order of weight, breaking ties arbitrarily.
For each vertex $u \in U$, set added[u] = False.
For each edge $e = (a, b) \in E$, if find(a) ≠ find(b), do the following. If both a and $b \in V - U$, add e to the tree. If both a and $b \in U$, skip the edge e. If $a \in U$ and $b \in V - U$, add e to the tree only if added[u] = False. After adding the edge, set added[u] = True. Do similar for $a \in V - U$ and $b \in U$.
If the final tree contains less than |V| - 1 edges, then a LCST is not possible. Otherwise, the final tree is the MLCST.
Approach III (modified Prim's) = does not work
Same as Approach 1
Pick a vertex r from V - U and initiate Prim's algorithm
If we select a vertex $u \in U$ from the priority queue, then for all vertices v that are adjacent to u and not yet included in the tree, we set $w(\{u, v\}) = \infty$. This ensures that all vertices $u \in U$ are leaves in the tree.
This approach does not work because the selection of the root r can determine for a vertex $u \in U$, which edge is added to include u in the tree.
For example, consider a graph with vertices A, B, C, and D, and the edges are AB, AC, BD, and CD. The weights of all edges is 10 except for BD whose weight is 1. Let U = {B}. Therefore, the MLCST is {BD, CD, AC}. It excludes the AB edge. However, if you

run Prim's algorithm from A as root, it can include the AB edge, which produces the wrong MLCST.

Checking if U = V can be done in O(|V|) time. Finding the cardinality of V also takes O(|V|). We can check if G - U is connected in O(|E| + |V|) using either BFS or DFS. We can find the MST T of G - U in O(|E| log |V|) using either Prim's or Kruskal's algorithm. We can find Eu and the minimum edge in Eu for all u $\in$ U in O(|E|) time. Therefore, the worst-case time complexity of our algorithm is O(|E| log |V|)

Give an example of a disconnected graph.
Or, if U = V, give an example of a graph containing at least 3 vertices.
Or, if U $\neq$ V, give an example of a graph G which becomes disconnected when we remove the vertices in U from G.

Rubric

Algorithms:
Using Approach I
+1 point for correctly handling the U = V case.
+1 point for correctly identifying the case when U $\neq$ V and a LCST is not possible if G - U is disconnected.
+1 point for correctly identifying the case when U $\neq$ V and a LCST is not possible if Eu = Ø for any vertex u $\in$ U.
+4 points for finding MST on G - U.
+3 points for correctly adding the minimum edge (u, v), u $\in$ U and v $\in$ V - U, for each vertex u $\in$ U.
Using Approach II
+1 point for correctly handling the U = V case
+1 point for correctly identifying the case when U $\neq$ V and a LCST is not possible if the final tree contains less than |V| - 1 edges.
+4 points for correctly handling a (u, v) edge when u $\in$ U and v $\in$ V - U.
+2 points for correctly handling a (u, v) edge when both u and v $\in$ U.
+2 points for correctly handling a (u, v) edge when both u and v $\in$ V - U.

Worst-case running time analysis
We give full points even if the algorithm in a) is wrong.
+2 points for showing some working
+3 points for writing the correct runtime

+5 points for correct example.

7) 16 pts
Consider two lists, L_1 of length n and L_2 of length m. We say that L_2 is a subsequence of L_1 if we can delete certain elements from L_1 so that the remaining sequence is equal to L_2. This means that there exists m indices $i_1 < \ldots < i_m$ such that $L_1[i_j] = L_2[j]$ for each j. Design an efficient algorithm that detects if L_2 is a subsequence of L_1 and outputs the indices $i_1, \ldots, i_m$ if L_2 is a subsequence of L_1 (8 pts), analyze your algorithm's worst-case running time (3 pts), and prove that your algorithm is correct (5 pts).

Solution:

The algorithm is a greedy algorithm that makes one pass over both lists. It starts with pointers at the first element of each list, repeatedly incrementing the pointer on L_1 until it finds an element i such that $L_1[i] = L_2[1]$. This value i is the output i_1, and at this point, the algorithm increments the pointer on L_2 to point to the second element while also incrementing the pointer on the first list to point to the $i+1$st element. The process repeats until both lists have been traversed. Here is the pseudocode:

$j = 1, k=1$

for $k = 1, \ldots, m$ ***do***

while $L_1[j] \mathrel{!}= L_2[k]$ ***do***

$j \leftarrow j + 1$

end while

$i_k = j$ *(in the solution)*

$j \leftarrow j + 1, k \leftarrow k + 1$

end for

This algorithm stays ahead of any other possible solution in the sense that among all possible subsequences, it outputs the one with the lowest indices $i_1 < \ldots < i_m$. More formally, for any other partial subsequence M' (consisting of indices

$i'_1 < \ldots < i'_m$, where some could be said larger than n if elements were not matched) if we define $\Phi(j; M')$ to be the number of elements of L_2 that have been matched in M used the first j indices of L_1, then we'll soon prove that our algorithm satisfies $\Phi(j; M) \geq \Phi(j; M')$ for all j where M is the subsequence produced by our algorithm.

Assuming this claim is true momentarily, we see that $\Phi(n; I) \geq \Phi(n; M')$ for all other potential partial subsequences, which means that if L_2 is a subsequence of L_1, then M must be one of them. The proof of the claim involving Φ is based on the fact that our algorithm always chooses the first match it finds. Since $j = 0$, all subsequences have zero matches, and since we always choose the first valid match, our algorithm always stays ahead.

The running time is clearly $O(n + m)$. We just make one pass through both lists.

8) 10 pts

You are assigned n tasks, numbered from 1 to n. There are m dependence relations between the tasks. For the i-th dependence relation, (u_i, v_i) indicates that task u_i should be finished before starting task v_i. You can only process one task at a time. Design an efficient algorithm for deciding whether it is possible to finish all tasks without violating any dependence relations. Your solution should run in linear time with respect to n and m. (7pts)

Analyze the worst-case running time complexity of your solution. (3 pts)

Brief solution to the problem

a) Construct the directed graph for the dependence relations.
b) Using topological sort to decide whether this graph is acyclic.
c) These tasks can be finished if and only if the graph is acyclic.

Additional Space

Additional Space

Additional Space