EECS 336: Algorithms **Homework 1 Solutions**

1 Asymptotics

a. $f(n) = \Theta(g(n))$

b. $f(n) = O(g(n))$

c. $f(n) = \Theta(g(n))$

d. $f(n) = \Theta(g(n))$

e. $f(n) = O(g(n))$

f. $f(n) = \Omega(g(n))$

g. $f(n) = \Omega(g(n))$

h. $f(n) = O(g(n))$

i. $f(n) = O(g(n))$

j. $f(n) = \Theta(g(n))$

2 More Asymptotics

a. The claim is true. If $f(n)$ is $\Omega(g(n))$, then there exists a $n_0, c > 0$ such that for all $n > n_0$, $f(n) > cg(n)$. For this choice of n_0 and c, it follows that for all $n > n_0$,

$$\begin{aligned} \log f(n) &> \log cg(n) \\ &= \log g(n) + \log c. \end{aligned}$$

It follows that $f(n)$ is $\Omega(\log g(n) + \log c)$. Since $\log g(n) + \log c$ is $\Omega(\log g(n))$, the claim follows.

b. This claim is false. As an example, consider $f(n) = \log_2 n^2$ and $g(n) = \log_2 n^3$. We have $2^{f(n)} = n^2$ and $2^{g(n)} = n^3$. It is not the case that $n^2 = \Theta(n^3)$.

c. We will first show that if $f(n)$ is $O(g(n))$, then $\sqrt{f(n)}$ is $\sqrt{g(n)}$. If $f(n)$ is $O(g(n))$, then there exist $n_0, c > 0$ such that $f(n) < cg(n)$ for all $n > n_0$. For such n, we also have $\sqrt{f(n)} < \sqrt{c}\sqrt{g(n)}$. It follows that we can choose the same n_0 and $\sqrt{c}$ to satisfy the definition of big-Oh for $\sqrt{f(n)}$ and $\sqrt{g(n)}$.

Similarly, if $f(n)$ is $\Omega(g(n))$, then there exist $n_0, c > 0$ such that $f(n) > cg(n)$ for all $n > n_0$. For such n, we also have $\sqrt{f(n)} > \sqrt{c}\sqrt{g(n)}$. It follows that we can choose the same n_0 and $\sqrt{c}$ to satisfy the definition of big-Omega for $\sqrt{f(n)}$ and $\sqrt{g(n)}$.

d. This claim is false. Consider $f(n) = \sqrt{\log n} + \log n$ and $g(n) = 1 + \log n$. Both functions are $\Theta(\log n)$. Subtracting $\log n$ from both functions leaves $\sqrt{\log n}$ and 1. It is not the case that $\sqrt{\log n} = \Theta(1)$.

3 ARM WRESTLING!

Claim 1. *For any outcome graph $G = (V, E)$ (with n vertices), a total ordering exists.*

Proof by induction on n.
Base Case: $n = 1$. There's only one ordering, and it is trivially total.

Inductive Hypothesis: Assume there exists a total ordering in any outcome graph with k vertices.

Inductive Case: Consider an outcome graph $G = (V, E)$ with $k + 1$ vertices, and consider an arbitrary vertex $v_1 \in V$. The graph $(V \setminus \{v\}, E)$ has a total ordering $v_1, \ldots, v_k$ (where v_1 beats v_2, etc), by the inductive hypothesis. Now let j be the index of the first student in the ordering who loses to v_0. Since v_{j-1} beats v_0, we may insert v_0 into the ordering between v_{j-1} and v_j and get a total ordering, proving the claim. □

4 Modern Tourism

Algorithm: Run BFS to compute the distance to each vertex x from the start vertex, s. Then partition vertices into layers L_k for $k = 0, 1, 2, \ldots$, where L_k is the vertices distance k from s.

We now record the total number of shortest paths from s to x, $num(x)$, for every vertex. We start by setting $num(s) = 1$. Then we proceed layer by layer in the following way: for each $x \in L_k$, let S_x be the nodes in L_{k-1} adjacent to x. Set $num(x) = \sum_{y \in S_x} num(y)$.

Return $num(t)$.

Runtime: Let $m = |E|$ and $n = |V|$.

- BFS is $O(m + n)$.
- Partitioning vertices into layers is $O(n)$.
- Computing $num()$ requires that the algorithm check neighbors to every vertex. This checks each edge at most twice (once in each direction), and therefore takes $O(m)$ time.

The total is therefore $O(n + m)$.

Correctness: We have proved that BFS correctly computes distances. All that remains is to verify that the computation of $num()$ is correct. We argue that this is true for all $x \in L_k$ by induction on k. (This is a loop invariant argument.)

> **Base Case:** $k = 0$. There is exactly one path from t to t.
>
> **Inductive Hypothesis:** Assume that for all in $x \in L_k$, $num(x)$ is the number of shortest paths from u to x.

Inductive Case: Consider some vertex $x \in L_{k+1}$. Any shortest path from s to x must go through a vertex in S_x immediately before arriving at x. (Why? You can argue by contradiction.) Note that for any $y \in S_x$, the number of shortest paths from s to x which visit y immediately before x is precisely the number of shortest paths from s to y. Since $S_x \subseteq L_k$, the IH implies that $num(y)$ is exactly this quantity as well. Since each shortest path from s to x must go through some such y, it follows that $\sum_{y \in S_x} num(y)$ correctly counts the number of shortest paths from s to x.

Note: it is possible to modify BFS to compute $num()$ as it goes. The above is slightly clearer to present.

5 The Maize Runner

There are two ways to solve this problem. Both ways first involve getting BFS trees for every component of the graph. To do this, we maintain a "visited" variable for each vertex, which is initialized to 0. For each vertex, if that vertex is unvisted, we run BFS starting at that vertex, and mark every vertex in the resultant BFS tree as visited. This will end with a BFS tree for every component of the graph.

Next, there are two ways forward, either of which works:

1. Compute the number of edges m in G, and the total number of edges m' in the BFS trees. If $m > m'$, return "True." Else, return "False."
2. Iterate through G, checking if each edge is in its components' corresponding BFS tree. (This can be done in linear time by first iterating through each BFS tree and marking the edges you see, then going through once more for unmarked edges.)

Proof of correctness: assume G contains an edge e not in any of the BFS trees, and let C be the component containing e. Assume the BFS tree for C contains m'_C edges. Then C contains at least $m'_C + 1$ edges. But by basic graph theory, C contains $m'_C + 1$ vertices. Since a graph is acyclic if and only if its number of edges is no more than the number of vertices minus one, it follows that C and thus G has a cycle.

Conversely, assume that $m = m'$. Then every component of G must be a tree, which implies that G is acyclic.

Runtime: For each component C, let m_C and n_C be the number of edges and vertices in C, respectively, and let m and n be the number of edges and vertices in G. Running BFS on each component yields a runtime which is $O(\sum_C m_C + n_C)$, which is $O(m + n)$. Either of the two next steps of the algorithm involve simply iterating over the edges of the G and/or the search trees, which is an $O(m)$ operation. Hence, the total runtime is $O(m + n)$.