CS 38 Introduction to Algorithms **Spring 2014**

Solution Set 1

Posted: April 15

If you have not yet turned in the Problem Set, you should not consult these solutions.

1. (a) To prove that $T(n) = O(n)$ we need to look at the definition of big-oh and we see that the precise statement that is needed is that for sufficiently large n, $T(n) \leq cn$, for some constant c. For any particular value of c, the main step of the faulty proof fails: namely, when we apply the induction hypothesis to $T(n-1)$ we find that

$$T(n) = 2 \cdot T(n-1) \leq 2c(n-1)$$

which is not bounded above by cn.

The correct claim is that $T(n) \leq 2^n$, and the proof by induction is as follows: the base case of $n = 1$ is trivial, and

$$T(n) = 2 \cdot T(n-1) \leq 2 \cdot 2^{n-1} = 2^n,$$

as required, where the inequality uses the induction hypothesis.

(b) Let us be precise about what we are given. We have $T(1) = \gamma$ and $T(n) \leq \alpha \cdot T(n/2) + \beta \cdot n$ for some constants α, β, γ.

We claim that $T(n) \leq dn^c$ for $d = \max\{\gamma, 2\beta\}$ and $c = \max\{\log_2(2\alpha), 1\}$. The proof is by induction on n. For the base case of $n = 1$ we have $T(1) = \gamma \leq d \cdot 1^c$.

For the induction case, we have

$$T(n) \leq \alpha \cdot T(n/2) + \beta \cdot n \leq \alpha \cdot d(n/2)^c + \beta \cdot n \leq (\alpha d/2^c + \beta)n^c$$

where the second inequality used the induction hypothesis and the last used the fact that $c \geq 1$. By our choice of d and c, we have

$$\alpha d/2^c + \beta \leq d/2 + \beta \leq d,$$

so $T(n) \leq dn^c$ as required.

2. (a) We consider two cases, depending on which component the DFS discovers first (we say the DFS discovers a component when it first visits a vertex in that component). If the DFS discovers C_2 first, then it must visits all of C_2 before any vertex in C_1, because both are SCCs and there is an edge from C_1 to C_2 (so there can be no path from C_2 to C_1 without merging components C_1 and C_2). Thus $f(C_2)$ is in fact smaller than the *first* discovery time of C_1 which is certainly smaller than $f(C_1)$.

If the DFS discovers C_1 first, via vertex $v \in C_1$, then vertex v is not finished until all of its descendants in its DFS tree are finished. Since there is a path of unvisited vertices from v to every vertex in C_2 (if (x, y) is the edge from C_1 to C_2, then there is a path from u to x in C_1 since it is strongly connected, and a path from y to v in C_2 because it is strongly connected), each of the vertices in C_2 become descendants of v in the DFS tree. Thus the finishing time of every vertex in C_2 is earlier than the finishing time of v, from which we conclude $f(C_1) > f(C_2)$.

(b) By the previous part, the latest finishing time must occur in a source SCC C in G_{SCC}. In G^T_{SCC}, C is a sink SCC (note that the SCCs of G and G^T are the same). Since it is a sink SCC, the DFS will visit exactly the vertices in C and then return to "line 3" to find the vertex with the next largest finishing time. Again by the previous part, the next highest finishing time must occur in a source SCC of G after the removal of C. This is because if it didn't, then there would be an edge from another component to that component, and the other component would have the later finishing time by the previous part. In G^T this is a sink component after the removal of C, and so the DFS traversal visits all of it, and then returns to "line 3". Continuing in this fashion we find that the components of G^T_{SCC} are indeed visited in reverse topological order.

(c) Consider the graph $G = (v, E)$ with $V = \{a, b, c, d, e, f\}$ and

$$E = \{(a, b), (b, c), (c, a), (b, d), (d, e), (e, f), (f, d)\}.$$

There are two SCCs: $\{a, b, c\}$ and $\{d, e, f\}$. A DFS traversal starting at a could explore the vertices in the following order a, b, c, d, e, f, with discovery times $1, 2, 3, 5, 6, 7$, respectively, and finishing times $12, 11, 4, 10, 9, 8$, respectively. Thus the first finishing time occurs in the the connected component $\{a, b, c\}$ and the DFS starting there visits all of the graph (since there is a path from c to every other vertex in the graph) producing only one DFS tree.

(d) As suggested, we consider a strongly connected graph first. We perform a DFS, labeling vertices with "even" or "odd" according to its parent in the DFS tree (i.e., it gets the opposite label of its parent). We claim that the graph contains an odd cycle iff we try to label an odd vertex even or vice versa. In the forward direction, suppose there is an odd cycle. Since a DFS traverses every edge in the graph, it must traverse all the edges along the cycle at some point, and if in the course of exploring these edges it never tries to label an even vertex odd or vice versa, then it must have labeled the vertices around the cycle with alternating "even" and "odd" labels, which implies that the cycle is even, a contradiction.

In the backward direction, consider the first time the DFS explores an edge from an "even" vertex u to an already-labeled "even" vertex v (or the same for "odd"). Since this is the first such edge, and G is strongly connected, there is an even length path from v to u (always going from even to odd to even to odd, etc...). Together with the edge (u, v) this constitutes an odd cycle.

Since we can update the labels at the same time we fill in the predecessor pointer in an ordinary DFS, the overall algorithm can be made to run in the same time, $O(m + n)$.

For a general graph G, we first decompose it into strongly connected components in $O(m+n)$ time, and then run the above algorithm in each strongly connected component (and note that any cycle must be entirely contained within a single strongly connected component).

3. Implementing heapsort via a comparison-based heap implementation amounts to a comparison-based sorting algorithm, which invokes n INSERT operations and n EXTRACT-MIN operations. Thus the running time is at least $n(f(n) + g(n))$, and we know from the lower bound on comparison-based sorting algorithms that this quantity must be at least $\Omega(n \log n)$; thus $f(n) + g(n) \geq \Omega(\log n)$.

4. We consider implementing heaps via d-ary trees, for a parameter d. The main operation of HEAPIFY-DOWN now take $O(d \log_d n)$ since each time an element moves down, we need to compare its value to each of its d children, and there may be as many as $\log_d n$ moves as that that is the height of the tree. On the other hand HEAPIFY-UP now takes only $O(\log_d n)$ since each move only requires comparing with the parent. As a result, we find that EXTRACT-MIN (which invokes HEAPIFY-DOWN) has a cost of $O(d \log_d n)$, while DECREASE-KEY and INSERT (which invoke HEAPIFY-UP) have a cost of $O(\log_d n)$.

   Using such a heap, we can implement Dijkstra in time $O(nd \log_d n + m \log_d n)$ since it requires n INSERT operations (to build the heap), n EXTRACT-MIN operations to choose each of the tree edges, and m DECREASE-KEY operations as it processes each edge.

   Balancing, we find that choosing $d = m/n$ yields the desired running time.

5. We simply modify Dijkstra's algorithm from class so that the "dist" attribute of each vertex contains an estimate of the bottleneck weight of the best path from s to v. We have to modify only one step: originally, when a new vertex u is added to S, we go through each of its neighbors v possibly calling DECREASE-KEY if $v.dist$ is larger than $u.dist$ plus the weight of edge (u, v). Now, we call decrease key if $v.dist$ is larger than the *maximum* of $u.dist$ and the weight of edge (u, v).

   Similar to the original proof of correctness, we find can prove an invariant of the new algorithm: for all $v \in S$, $v.dist$ equals the bottleneck weight of the best path from s to v.

   The proof is by induction on $|S|$; again the base case is trivial. When $|S| = k$, referring to the figure on slide ? in Lecture 4, we find that the s-to-y path must have a larger maximum weight edge, than the entire s-to-v path, because otherwise the algorithm wouldn't have chosen v. We conclude that for any other s-to-v path (exiting at y in the figure), the bottleneck weight is larger than the bottleneck weight of the chosen path, and thus the lemma follows.