Problem 1, CS38 Set 1, Matt Lim

(a) This proof is wrong because it doesn't account for the exponential build up of 2s, falsely labeling $T(n)$ polynomial instead of exponential. We will prove that this argument is false by obtaining a contradiction. So, assume

$$T(n) = 2 \cdot T(n-1) = O(T(n-1)) = O(n-1) = O(n)$$

Then we have that

$$2 \cdot T(n-1) = O(n-1)$$

This implies that, for some $n_1 \geq n_a$ and some positive constant c_1 (where n_a is a positive constant as well),

$$2 \cdot T(n_1) = O(n_1)$$

$$2 \cdot T(n_1) \leq c_1 \cdot n_1$$

We also have that

$$2 \cdot T(n_1) = O(n)$$

This implies that, for some $n_2 \geq n_b$ and some positive constant c_2 (where n_b is a positive constant as well),

$$2 \cdot T(n_2 - 1) \leq c_2 \cdot n_2$$

Note that we can pick n_1 and n_2 large enough such that, letting $n_1 = n_2 = n$,

$$2 \cdot T(n) \leq c_1 \cdot n$$

$$2 \cdot T(n-1) \leq c_2 \cdot n$$

are both true. Now let $c = max(c_1, c_2)$. Then we have that

$$2 \cdot T(n) \leq c \cdot n$$

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

Now we can proceed as follows. Multiplying the first inequality by -1 and taking the reciprocals of the second inequality gives us the following equations:

$$-2 \cdot T(n) \geq -c \cdot n$$

$$\frac{1}{2 \cdot T(n-1)} \geq \frac{1}{c \cdot n}$$

Multiplying these two equations together gives us the following (since the second of the above two equations is positive on both sides)

$$\frac{-2 \cdot T(n)}{2 \cdot T(n-1)} \geq \frac{-c \cdot n}{c \cdot n}$$

$$-2 \geq -1$$

Thus we get a contradiction, and thus the initial equality must be false.

We will now give a correct solution to the recurrence together with a correct proof. We will prove that this recurrence is in fact $O(2^n)$, which is exponential. To do this, we will prove by induction that $T(n) = 2^{n-1}$. So let us begin. The base case is when $n = 1$. Here we have that $T(1) = 1 = 2^{(1-1)}$. So we have that the base case is satisfied. Now for our inductive assumption. Let us assume that $T(n) = 2^{n-1}$. Now we must show that $T(n+1) = 2^n$. We have that

$$T(n+1) = 2 \cdot T(n)$$

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

$$T(n+1) = 2^n$$

So we have that $T(n+1) = 2^n$. Thus, by induction, we can conclude that $T(n) = 2^{n-1}$ for all $n \geq 1$. It then follows that $T(n) \leq O(2^n)$, and the proof is complete.

(b) Let $c > 0$ be some constant. Let $f(n) = O(n) \implies f(n) \leq a_1 \cdot n$, and $f(n) = O(n^c) \implies f(n) \leq a_2 \cdot n^c$, where a_1, a_2 are positive constants. We will let $a = max(a_1, a_2)$. We will prove that any recurrence of the form: $T(n) = O(T(\frac{n}{2})) + O(n)$ and $T(1) = O(1)$ satisfies $T(n) \leq O(n^c)$ (by showing $T(n) \leq a \cdot (n+1)^c$). To do this, we will use strong induction. So, let us begin. We will have our base case be when $n = 1$. Here we have that

$$T(1) = O(1)$$

$$T(1) = O(1^c)$$

$$T(1) \leq O(1^c)$$

So we have that the base case is satisfied. Now for you strong inductive assumption. Let us assume that $T(k) \leq O(k^c)$, where $1 \leq k \leq n$. Now we must show that $T(n+1) \leq O((n+1)^c)$. We will now proceed under the assumption that we are using integer division. So, consider the integer $m = \frac{n+1}{2}$. It is clear that $1 \leq m \leq n$. Due to our inductive assumption, this means that $T(m) \leq O(m^c)$. Let us now consider $T(n+1)$.

$$T(n+1) = O(T(\frac{n+1}{2}) + O(n+1)$$

$$T(n+1) = O(T(m)) + O(n+1)$$

$$T(n+1) \leq O(O(m^c)) + O(n+1)$$

$$T(n+1) \leq O(m^c) + O(n+1)$$

$$T(n+1) \leq O((\frac{n+1}{2})^c) + O(n+1)$$

$$T(n+1) \leq a_2 \cdot (\frac{1}{2})^c \cdot (n+1)^c + a_1 \cdot (n+1)$$

We will choose $c \geq 2$, which implies $n \leq \frac{n^c}{2}$ for $n \geq 2$. So for large n, we have

$$T(n+1) \leq a \cdot \frac{1}{4} \cdot (n+1)^c + a \cdot \frac{1}{2} \cdot (n+1)^c$$

$$T(n+1) \leq \frac{3}{4} \cdot a \cdot (n+1)^c$$

Then we have that

$$T(n+1) \leq O((n+1)^c)$$

Thus, by induction, we can conclude that $T(n) \leq O(n^c)$ for all $n \geq 1$.

Problem 2, CS38 Set 1, Matt Lim

(a) To prove this, we will consider two cases. First, we will set some things up. Let $(u, v) \in E$ be the edge from C_1 from C_2; that is, $u \in C_1$ and $v \in C_2$.

The first case is when we start with a vertex in C_1. That is, $d(C_1) < d(C_2)$. Let us call the vertex first discovered in C_1 x. At the time when x is discovered, $x.d$, all vertices in C_1 and C_2 are white. It is also true that, at this time, G contains a path from x to each vertex in C_1 consisting of only white vertices (since C_1 is a SCC). Further, it is true that, at this time, there exists a path in G from x to any vertex $w \in C_2$ that consists of only white vertices: $x \rightsquigarrow u \rightarrow v \rightsquigarrow w$. This is because we have that there is some edge from a vertex to C_1 to C_2. Because C_1 is a SCC, that vertex can be reached from x, and because C_2 is a SCC, once you reach a vertex in C_2 you can reach any other vertex in C_2. Now we have that, by the white-path theorem in CLRS (page 608), all vertices in C_1 and C_2 become descendants of x in the depth-first tree. Then, by Corollary 22.8 in CLRS (page 608), x has the latest finishing time of any of its descendants, and thus we can conclude that $x.f = f(C_1) > f(C_2)$.

The second case is when we start with a vertex in C_2. That is, $d(C_2) < d(C_1)$. Let us call the vertex first discovered in C_2 y. At the time when y is discovered, $y.d$, all vertices in C_2 are white and G contains a path from y to each vertex in C_2 that consists of only white vertices (since C_2 is a SCC). The white-path theorem tells us that all vertices in C_2 become descendants of y in the depth-first tree. Then, by Corollary 22.8, $y.f = f(C')$. Now, let us again consider time $y.d$. At this time, all vertices in C_1 are white. Since there is an edge (u, v) from C_1 to C_2, Lemma 22.13 tells us that there cannot be a path from C_2 to C_1. Then we have that no vertex in C_1 can be reached from y. Therefore, at time $y.f$, all the vertices in C_1 are white. Thus we can conclude that, for any vertex $w \in C_1$, we have that $w.f > y.f$ (since y finishes before w even starts), which tne gives us that $f(C_1) > f(C_2)$.

(b) Consider G. The vertex with the latest finishing time will be a source node for its SCC. Let us call this vertex v and the SCC it is in S. There will be no edges that go from another SCC to S, since then it would not have the latest finishing time. There will only be edges that go from S to another SCC. We know this from $2(a)$. Now consider a DFS of G^T that selects vertices in line 3 by decreasing order of finishing times. S now becomes a sink in G^T_{SCC}. This is because in G, the only edges connecting S to other SCCs were directed out towards other SCCs. Thus, when we reverse all the edges to make G^T, the only edges connecting it to other SCCs are directed in towards S. So, we have that the first component visited is a sink in G^T_{SCC}. Now consider what happens after we remove this component. This means we will consider all the graphs with the first component removed. So, let us consider the vertex with the now-latest finishing time (after the removal of the first component), called v_2, along with its SCC, S_2. We will first consider it in G. The vertex we are considering has the latest finishing time of the graph. Again from $2(a)$, we have that the only edges that connect S_2 to other SCCs are directed out towards other SCCs. Thus, when we consider G^T, the only edges that connect S_2 with other SCCs are directed in towards S_2. Thus we have that S_2 is a sink in G^T_{SCC}. Continuing on, it is clear to see that we keep on visiting sinks in G^T_{SCC}. This is true because of $2(a)$. We showed above that the first component visited is a sink. Then, when we remove it, we get a new vertex in a new SCC, which must be a sink in G^T_{SCC} for the same reason the first component had to be a sink. This logic continues until all the components have been visited. Thus, since the SCCs of G^T_{SCC} are all sinks when they are visited, we have that the SCCs of G^T_{SCC} are visited in reverse topological order.

Alternatively, we can prove this by contradiction. Assume a visited SCC S isn't a sink in G^T_{SCC} (with the previously visited components having been removed). Then there is some other SCC T that S points to in G^T_{SCC}. So in G_{SCC}, T points to S. That is, there is an edge from T to S in G_{SCC}. But this means $f(T) > f(S)$. This is a contradiction, since we select vertices by decreasing order of finishing times, and we selected a vertex in S. So all SCCs visited must be sinks (with the previously visited components having been removed).

(c) See the diagram below. The format x/y is discovery time/finishing time. As we can see, starting at the node with the earliest finishing time (the node on the far left) gives us a DFS forest with only one tree (instead of two, one covering each SCC).

(d) The algorithm is basically the same as the algorithm for DFS given in the slides. There is only one major change: when DFS-Visit is performed on a starting vertex u, in the for-loop that visits all the adjacent vertices, if $v.color$ is grey and the parity of its discovery time is the same as u, then the graph has an odd length cycle. So, the pseudo-code looks as follows:

```
DFS(directed graph G)

  1. for each vertex $v$, $v.color = white$, $v.pred = nil$
  2. $time = 0$
  3. $oddCycle = false$
  4. for each vertex $u$, IF $u.color = white$ THEN DFS-VISIT$(G, u)$

DFS-VISIT(directed graph $G$, starting vertex $u$)

  1. $time = time + 1, u.discovered = time, u.color = grey$
  2. for each $v$ adjacent to $u$
       3. IF $v.color = white$ THEN
            4. $v.pred = u$, DFS-VISIT$(G, v)$
       5. ELIF $v.color = grey$ AND $u.discovered$ % 2 = $v.discovered$ % 2 THEN
            6. $oddCycle = true$
  7. $u.color = black$; $time = time + 1$; $u.finished = time$
```

The boolean value *oddCycle* tells us whether or not the graph has an odd length simple directed cycle. This algorithm works because, if we are considering a vertex u, if it has an edge to an adjacent vertex v, that means u can reach v. If v is grey, it also means that v can reach u, since if v is grey that means that v and u are in the same DFS tree, with v having already been discovered. So, we have a cycle if u has an edge to a grey vertex v. To determine if it is an odd cycle, we can simply check if the parity of the discovery times of u and v are equal. If they are, then two adjacent vertices in the cycle have the same parity. And since the parity of the discovery time alternates for successively visited vertices, this can only occur if the cycle is of odd length.

Now, to show it is $O(m + n)$. Clearly, we did not add anything to this algorithm that changes its running time. We are still iterating through the same number of vertices and edges; all that changes is that we check gray vertices and compare discovery time parities (line 5 DFS-Visit) which are both constant time operations. Thus, we have a $O(m + n)$ time algorithm to determine if a directed graph G contains an odd length (simple, directed) cycle.

Problem 3, CS38 Set 1, Matt Lim

Let us assume, to the contrary, that $f(n) + g(n) < \Omega(\text{log}n)$. The time to sort n numbers using a heap involves building a heap, which takes time $n \cdot f(n)$ (n insertions), and extracting the minimum n times, which takes time $n \cdot g(n)$ (n **extract-min** operations). So we get that the time to sort a heap is $n \cdot f(n) + n \cdot g(n) < n \cdot \Omega(\text{log}n) = \Omega(n\text{log}n)$. Then we have that the time to sort a heap is faster than $\Omega(n\text{log}n)$. But this is a contradiction, since we proved in lecture that comparison-based sorts can never do better than $\Omega(n\text{log}n)$; that is, we proved that sorting time $\geq \Omega(n\text{log}n)$. Thus we have a contradiction, and can conclude that $f(n) + g(n) \geq \Omega(\text{log}n)$.

Problem 4, CS38 Set 1, Matt Lim

Before we dive into the proof, we will first establish a few things. We will let n be the number of vertices in our graph and m be the number of edges. Let us consider a min-d-ary-heap with n items. The **decrease-key** operation (involves swaps upward since we are dealing with a min-d-ary-heap; i.e. we **heapify-up** on the item that is decreased) takes $O(\log_d n) = O(\frac{\log n}{\log d})$ time. We can see this is true because $\log_d n$ is just the height of the heap, and the height of the heap bounds how many swaps can occur. Here is a quick proof that gets the height of the tree. Let n be the number of nodes, h the height of the tree, and d be the maximum number of children of each node. Then

$$n = 1 + d + d^2 + \cdots + d^h$$

$$n \cdot d = d + d^2 + d^3 + \cdots + d^{h+1}$$

$$n(1-d) = 1 - d^{h+1}$$

$$n = \frac{n(1-d)}{(1-d)} = \frac{(1-d^{h+1})}{(1-d)} \approx d^h$$

$$\implies \log_d n \approx h$$

There is no constant in front of the $\log_d n$ because each swap upward only requires one comparison, a comparison between the item that is being decreased and its parent. This is constant time. Note that this also shows that **heapify-up** is $O(\log_d n)$. The **extract-min** operation (involves swaps downward since we put the last element at the top and **heapify-down** on it) takes $O(d \cdot \log_d n) = O(\frac{d \cdot \log n}{\log d})$. We can see this is true because $\log_d n$ is just the height of the heap, and the height of the heap bounds how many swaps can occur. Each swap requires at most d comparisons, since we must find the minimum of the children ($d-1$ comparisons) and compare it to their parent (1 comparison). Now, here is the pseudo code for the algorithm.

```
1. Create an empty min-$d$-ary-heap
2. FOR EACH $v \neq s$ : $d(v) \leftarrow \infty$ ; $d(s) \leftarrow 0$.
3. FOR EACH $v \in V$ : insert $v$ into min-$d$-ary-heap
4. WHILE (the min-$d$-ary-heap is not empty)
    5. $u \leftarrow$ extract-min from min-$d$-ary heap.
    6. FOR EACH edge $(u, v) \in E$ leaving $u$ :
        7. IF $d(v) > d(u) + \ell(u, v)$
            8. decrease-key of $v$ to $d(u) + \ell(u, v)$ in min-$d$-ary-heap.
            9. $d(v) \leftarrow d(u) + \ell(u, v)$.
            10. $v.pred = u$
```

The data structure we will use with this algorithm is a min-d-ary-heap, where $d = m/n$. We will now show that this algorithm runs in time $O(\frac{m \cdot \log n}{\log(m/n)})$. In line 2, we initialize all the distances, of which there are n, making this operation $O(n)$. In line 3, we make a min-d-ary-heap of n items. This can be done by the **build-heap** operation showed in class, which takes an array and re-orders it so that it satisfies the heap property. We showed in class that this is $O(n)$ for a max-heap; it is also clearly $O(n)$ for a min-heap (same algorithm, except when **heapify-down** is done it swaps when a parent is larger than a child instead of when it is smaller). Further, it is still $O(n)$ for a min-d-ary-heap, since the changes we would make to the proof in class (use $\log_d n$ and d instead of $\log n$ and 2) just make the summation smaller. In line 5, we extract n min elements elements from the heap (since we are in the while-loop that goes through the whole heap. We showed above that the **extract-min** operation is $O(\frac{d \cdot \log n}{\log d})$. Since we have that $d = m/n$, this gives us time $O(\frac{m \cdot \log n}{n \log(m/n)})$ for each **extract-min** operation. And since we do this operation n times, this gives us time $O(\frac{m \cdot \log n}{\log(m/n)})$. The for-loop in line 6 goes through order $O(m)$ iterations (since m is the number of edges). And worst case, each time through the for-loop, we perform the **decrease-key** operation, which takes $O(\frac{\log n}{\log d}) = O(\frac{\log n}{\log(m/n)})$. And since we do this operation $O(m)$

times, this gives us time $O(\frac{m \cdot \log n}{\log(m/n)})$. So, the largest order operation is $O(\frac{m \cdot \log n}{\log(m/n)})$. So we can conclude that the algorithm runs in time $O(\frac{m \cdot \log n}{\log(m/n)})$.

As for correctness, the same proof as we saw in the Princeton lecture slides can be applied, since all we are doing is changing the data structure it uses.

Problem 5, CS38 Set 1, Matt Lim

Let S be the set of vertices, s be the starting vertex, $b(u)$ be the current smallest bottleneck weight to a vertex u (note that this definition slightly changes for the proof of correctness), $w(x, y)$ be the weight of the edge between x and y, $B(x, y, \dots, z)$ be the bottleneck weight of the path $x \rightsquigarrow y \rightsquigarrow \cdots \rightsquigarrow z$, and $\phi(v)$ be the max of the bottleneck weight of the smallest bottleneck-weighted path to some node u in explored part and the weight of the single edge (u, v)

```
1. Create an empty min-heap
2. FOR EACH $v \neq s$ : $b(v) \leftarrow \infty$ ; $b(s) \leftarrow 0$.
3. FOR EACH $v \in V$ : **insert** $v$ into min-heap
4. WHILE (the min-heap is not empty)
    5. $u \leftarrow$ **extract-min** from min heap.
    6. FOR EACH edge $(u, v) \in E$ leaving $u$ :
        7. IF $b(v) > max(b(u), w(u, v))$
            8. **decrease-key** of $v$ to $max(b(u), w(u, v))$ in min-heap.
            9. $b(v) \leftarrow max(b(u), w(u, v))$.
            10. $v.pred = u$
```

We will now show that this algorithm runs in time $O(n + m)$ plus the time for n **insert** operations, m **decrease-key** operations, and n **extract-min** operations. In line 2, we initialize all the distances, of which there are n, making this operation $O(n)$ (which goes into the $O(n + m)$ running time). In line 3, we make a min-d-ary-heap of n items. This gives us the time for n **insert** operations, since we can build a min-heap by just inserting all n elements into the heap. Line 5 gives us the n **extract-min** operations, since, because of the while-loop, we extract n min elements from the min-heap. Line 8 gives us the m **decrease-key** operations, since the for-loop and while-loop combined loop through all the edges, and worst case for every edge we will have to run the **decrease-key** operation. The general $O(n + m)$ part comes from the fact that we loop through all n vertices with the while loop and all m edges with the for-loop (the distance initialization time is built into this).

Now we will show that this algorithm is correct. That is, we will show that it computes the smallest bottleneck-weighted paths from a source vertex s to all other vertices. The proof is given below:

Invariant For each node $u \in S$, $b(u)$ is the smallest bottleneck weight of the $s \rightsquigarrow u$ path.

Proof by induction on $|S|$.

Base case: $|S| = 1$ is easy since $S = \{s\}$ and $b(s) = 0$.

Inductive hypothesis: Assume true for $|S| = k \geq 1$.

- Let v be the next node added to S, and let (u, v) be the final edge.
- $\phi(v) = max(b(u), w(u, v))$.
- Consider any $s \rightsquigarrow v$ path P. We will show that its bottleneck weight is no smaller than $\phi(v)$.
- Let (x, y) be the first edge in P that leaves S, and let P' be the subpath to x.
- The bottleneck weight of $s \rightsquigarrow y$ is too long as soon as it reaches y.
- This is because

$$B(s, x, y, v) \geq B(s, x) + B(x, y) \geq b(x) + B(x, y) \geq \phi(y) \geq \phi(v)$$

The first inequality is true because we have non-negative weights, the second equality is because of the inductive hypothesis, the third inequality is because of the definition of $\phi(y)$, and the fourth inequality is because Dijkstra chose v instead of y.