CSE 101: Homework 1
(Due Thursday, April 10th in Lecture)

(a) $f = \Theta(g)$. Here $n$ dominates the constant factors.

(c) $f = \Theta(g)$. $n$ dominates in both $f$ and $g$ since polynomials dominate logarithms (Rule 4). Also, constant factors can be omitted, hence the 100 in $100n$ means nothing to big-Oh.

(e) $f = \Theta(g)$. $\log an = \log a + \log n$, so each term is dominated by $\log n$.

(g) $f = \Omega(g)$. We can consider the more fundamental question here of comparing $n^{0.01}$ to $\log^2 n$, and by Rule 4 any polynomial dominates any log.

(i) $f = \Omega(g)$. Any polynomial dominates any log.

(k) $f = \Theta(g)$. $\sqrt{n} = n^{0.5}$ and any polynomial dominates any log.

(m) $f = O(g)$. Taking logs of both sides gives $\log n + n \log 2 \leq n \log 3$. Rearranging gives $\log n \leq n \log 3/2$, which is true since any polynomial dominates any log.

(n) $f = \Theta(g)$. $2^n \leq 2^{n+1}$ for all $n > 0$ ints, and $2^{n+1} \leq 2 \cdot 2^n$ for all $n > 0$ ints.

2. Prove that if $f_1(n) = O(g_1(n))$ and $f_2(n) = O(g_2(n))$, then $f_1(n) + f_2(n) = O(g_1(n) + g_2(n))$. Ans: Let $f_1(n) = O(g_1(n))$ and $f_2(n) = O(g_2(n))$. This means that there exist constants $c_1, c_2 > 0$ such that $f_1(n) \leq c_1 g_1(n)$ and $f_2(n) \leq c_2 g_2(n)$ for all $n > 0$ integers. To prove the claim, we must find some constant $c_3$ that causes $f_1(n) + f_2(n) \leq c_3 [g_1(n) + g_2(n)]$ for all $n > 0$ integers.

$$f_1(n) + f_2(n) \leq c_1 g_1(n) + c_2 g_2(n)$$

$$\leq \max(c_1, c_2) g_1(n) + \max(c_1, c_2) g_2(n)$$

$$\leq \max(c_1, c_2) [g_1(n) + g_2(n)]$$

$$= c_3 [g_1(n) + g_2(n)]$$

We’ve found a $c_3 = \max(c_1, c_2)$ that satisfies the definition of big-Oh, proving the claim.
As a base case, notice that $F_0 = 0 \geq 2^{0.5 \cdot 0} = 0$, and $F_1 = 1 \geq 2^{0.5 \cdot 1} \approx 1.1316$. Now assume the induction hypothesis that $F_k \geq 2^{0.5k}$ for all $6 \leq k < n$. We would now like to show that the inequality holds for $k = n$. We know $F_n = F_{n-1} + F_{n-2} \geq 2^{0.5(n-1)} + 2^{0.5(n-2)} = 2^{0.5n - 0.5} + 2^{0.5(n-1)} = 2^{0.5n} [1/\sqrt{5} + 1/2] \geq 2^{0.5n}$. The last step is true since $(1/\sqrt{5} + 1/2) > 1$. This proves (a).

Let $F_n \leq 2^c$ for all $n \geq 0$, where $c < 1$. Then,

$$F_n = F_{n-1} + F_{n-2} \leq 2^c(n-1) + 2^c(n-2) \leq 2^c(n-1) + 2^c2^{-2c} \leq 2^c \left[ 2^{-c} + (2^{-c})^2 \right]$$

Since we want $F_n \leq 2^c$, we must choose $c$ so that $2^{-c} + (2^{-c})^2 \leq 1$. The smallest $c$ that we can choose is when $2^{-c} + (2^{-c})^2 = 1$. We can solve for $c$ algebraically, by substituting $x = 2^{-c}$, which gives $x^2 + x - 1 = 0$. Using the quadratic formula gives $x = \frac{-1 \pm \sqrt{5}}{2}$, and $c = -\lg x$. The only real $c$ of those two choices is $c \approx 0.6942$. So any $c$ in the range $0.6942 \leq c \leq 1$ will do the trick. You can use induction that has logic identical to the above to show that any of these $c$ values would work.

Before we start, notice that if $F_n = \Omega(2^{cn})$, that means we can find some $k > 0$ where $F_n \geq k2^c$ for all pos. ints. This is the same as rewriting $k = 2^t$, and say that $F_n \geq 2^{cn+t}$ for all pos ints. If we could choose some $c$ where $F_n \geq 2^{cn+t}$ for all pos. ints, that would mean

$$F_n = F_{n-1} + F_{n-2} \geq 2^c(n-1) + 1 + 2^c(n-2) + t \geq 2^c [(2^{-c} + (2^{-c})^2) + t]$$

We need the largest $c$ possible such that $2^{-c} + (2^{-c})^2 \geq 1$. As $c$ gets larger the right hand side gets smaller. So this is the same as the solution steps we did in part (b), and $c \approx 0.6942$. Notice that this solution is related to the golden ratio which appears many times with the Fibonacci numbers in a variety of ways.

We run DFS starting at node A. Here is the pre-array in alphabetical order:

$\text{pre}(v) = [1, 2, 3, 13, 5, 4, 14, 15, 7]$. And, $\text{post}(v) = [12, 11, 10, 18, 6, 9, 17, 16, 8]$. There are only two back edges: $(D, H)$ and $(B, E)$. Every other edge is a tree-edge.

The main idea that we will exploit is that in the DFS-tree, cycles are revealed by back edges. So, if we want to test whether a particular edge $e = (u, v)$ is part of a cycle, we must simply run DFS starting at $u$, and see if a back edge ever comes back to $u$. Here is the algorithm:
EdgeCycle(G, e=(u, v)):

1. Run DFS(u), but only exploring out on the v branch (no other branches of the DFS-tree).
2. For each back edge encountered, if it points to u then output “YES.”
3. If no back edge points to u, then output “NO.”

Let’s first analyze the runtime of this algorithm. The DFS takes worst case $O(|V| + |E|)$, step 2 takes at worst $O(|E|)$ time, and therefore the entire algorithm is dominated by the DFS call and the runtime of EdgeCycle(G, e=(u, v)) is $O(|V| + |E|)$.

Let’s prove the correctness claim: EdgeCycle(G, e=(u, v)) outputs “YES” if and only if $e$ is contained in some cycle in $G$. For the first direction, assume that EdgeCycle(G, e=(u, v)) outputs “YES”. That means there is some back-edge, $(w, u)$, that is on the $v$ branch of the DFS-tree that points back to $u$. The cycle that contains $(u, v)$ is then from $u$ to $v$ down the tree to $w$ and then back to $u$. For the other direction, assume that $e$ is contained in some cycle in $G$. So, if we start DFS at $u$, the $v$ branch must contain a back edge that points back to $u$, which is exactly what our algorithm tests for.