Problem 1

(a) Interpreting $\log n$ as $\log_2 n$, we can write $2^{(\log n)^c}$ as

$$\left(2^{\log_2 n}\right)^{(\log_2 n)^{c-1}} = n^{(\log_2 n)^{c-1}}.$$

Now note that the exponent $(\log_2 n)^{c-1}$ is a growing function in n and therefore $n^{(\log_2 n)^{c-1}}$ grows asymptotically faster than any polynomial function, which has the form n^a, where a is a positive integer constant. Also note that since $(\log n)^c$ grows more slowly than n, we know that $2^{(\log n)^c}$ grows asymptotically more slowly than 2^n. So the $2^{(\log n)^c}$ is "sandwiched" between polynomial and exponential functions.
Note: Functions of this form are called *quasipolynomial* functions and Babai's now famous paper is titled "Graph Isomorphism in Quasipolynomial Time."

(b) To understand f_2, let us study the slightly simplified function $2^{2^{\sqrt{\log_2 n}}}$. We will first show that $2^{\sqrt{\log_2 n}}$ grows faster than $(\log_2 n)^c$ for any constant c. Rewrite $(\log_2 n)^c$ as $\left(2^{\log_2 \log_2 n}\right)^c = \left(2^{c \log_2 \log_2 n}\right)$. Now, if we replace $\log_2 n$ by m, we see that we are comparing $2^{\sqrt{m}}$ with $\left(2^{c \log_2 m}\right)$. Since $\sqrt{m}$ grows faster than $c \log_2 m$ for any constant c, we get that $2^{\sqrt{\log_2 n}}$ grows faster than $(\log_2 n)^c$ for any constant c. This fact then implies that f_2 grows asymptotically faster than f_1 (and therefore, using (a), we see that f_2 grows faster than any polynomial function).

To compare $2^{2^{\sqrt{\log_2 n}}}$ to 2^n, we compare the exponents $2^{\sqrt{\log_2 n}}$ and n. We then write n as $2^{\log_2 n}$ and this means that we need to compare $\sqrt{\log_2 n}$ with $\log_2 n$. Since, $\sqrt{\log_2 n}$ grows more slowly relative to $\log_2 n$, we get that $2^{2^{\sqrt{\log_2 n}}}$ grows asymptotically more slowly than 2^n. Thus, we get

$$\text{polynomial functions} \ll f_1 \ll f_2 \ll \text{ exponential functions.}$$

Note: On Jan 4, 2017, Babai announced on his webpage that an error was discovered in his paper and that his solution to Graph Isomorphism took f_2 time, and not f_1 time, as his original paper had claimed. His announcement had the title "quasipolynomial claim withdrawn." Happily, just a few days later, Babai announced that he'd been able to fix his error and he announced "quasipolynomial claim restored."

(c) No one knows how to solve Graph Isomorphism in polynomial time. Currently, the fastest algorithm for the problem is Babai's quasipolynomial time algorithm. From (a) we know that quasipolynomial functions grow faster than polynomial time algorithms. From (b) we know that quasipolynomial functions grow more slowly compared to exponential functions. So we know that it does not take an exponential function to solve Graph Isomorphism.
Note: This is different from the situation for MVC. For MVC, we don't know how to solve it in less than exponential time and the general feeling among algorithms researchers is that an exponential algorithm is necessary to solve MVC.

Problem 2

(b) Due to the three nested **for**-loops with indices traveling from 1 through n, the running time of the algorithm is $\Theta(n^3)$. The input size is $s = \Theta(n^2)$ and we can rewrite $\Theta(n^3)$ as $\Theta\left((n^2)^{3/2}\right) = \Theta(s^{3/2})$. Thus the running time of this algorithm is super-linear, but sub-quadratic.

Algorithm 1 Multiply(A, B)

(a)
```
1: for i ← 1 to n do
2:     for j ← 1 to n do
3:         C[i, j] ← 0
4:         for k ← 1 to n do
5:             C[i, j] ← C[i, j] + A[i, k] · B[k, j]
6:         end for
7:     end for
8: end for
9: return C
```

Problem 3

(a) Initially, all vertices are white. After each iteration of the **while** loop, the results are as below:
1) white: E, F, G; grey: B, C, D; black: A
2) white: G; grey: C, D, E, F; black: A, B
3) white: None; grey: C, D, E, F, G; black: A, B, D
The final dominating set is A, B, D.

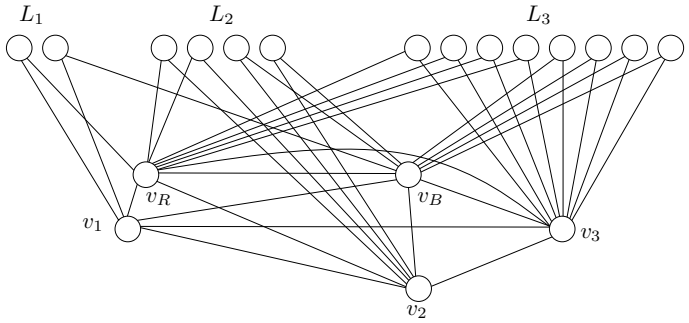

Figure 1: A bad example for greedy algorithm for Minimum Dominating Set.

(b) The execution of the greedy algorithm will repeatedly pick the vertex with the maximum number of white neighbors. In the beginning, v_3 has 8 white neighbors from L_3 plus v_1, v_2, v_R, v_B, which are white as well. Thus v_3 has a white neighborhood size of 13 (including itself). The remaining vertices have the following white neighborhood sizes: v_2: 9, v_1: 7, v_R: 12, v_B: 12, and every vertex $x \in L_1 \cup L_2 \cup L_3$ has white neighborhood size equal to 3. Thus v_3 will be picked first and colored black. Once v_3 is colored black, the white neighborhood sizes become: v_2: 4, v_1: 2, v_R: 3, v_B: 3 and every vertex $x \in L_1 \cup L_2$ has white neighborhood size equal to 1. Thus v_2 will be picked next. Now the white neighborhood sizes are v_1: 2, v_R: 1, v_B: 1 and every vertex $x \in L_1$ has white neighborhood size equal to 1. So v_1 is picked in the last iteration. In this way, the dominating set created by the algorithm is the set $\{v_1, v_2, v_3\}$, but the minimum dominating set is easily seen as $\{v_R, v_B\}$.

(c) The minimum dominating set in G_n has size 2. The greedy algorithm returns a dominating set $\{v_1, v_2, \ldots, v_n\}$ of size n in G_n.

(d) The graph G_{21} serves as a counterexample to the claim that the greedy algorithm is a 10-approximation. This is because the greedy algorithm produces a solution of size 21 which is *strictly more* than 10 times the size of a minimum dominating set.

Problem 4

(a) The greedy algorithm in Problem 2 with input adjacency list can be implemented in the following way:

Algorithm 2 Dominate(L)

```
1: Set nonblack be an empty object to host non-black vertices
2: Let ds be an empty set for hosting the dominating set
3: Let color be a length-n array, all of whose slots are initialized to white
4: for each vertex i in the graph do
5:     nonblack.insert(i, L[i].length+1)
6: end for
7: (v, whiteDeg) ← nonblack.getmax()
8: while whiteDeg > 0 do
9:     Save v to ds
10:    if color[v] == white then
11:        for each neighbor j of vertex v do
12:            nonblack.decreaseValue(j, 1)
13:        end for
14:    end if
15:    for each neighbor j of vertex v do
16:        if color[j] == white then
17:            for each neighbor k of vertex j do
18:                nonblack.decreaseValue(k, 1)
19:            end for
20:            color[j] ← gray
21:        end if
22:    end for
23:    color[v] ← black
24:    (v, whiteDeg) ← nonblack.getmax()
25: end while
26: return ds
```

(b) Given the running time of the 3 methods, `getMax`, `insert`, and `decreaseValue`, we can analyze the algorithm's running time complexity as follows. The for-loop (Lines 4-6) executes `insert`(k, v) n times, taking $O(\log n)$ time for each insertion. Thus, this for-loop will run in $O(n \log n)$ time. The while-loop is executed n times because with each execution, one vertex is deleted from **nonblack**. Each execution of `getmax`, takes $O(\log n)$ time and therefore extracting vertices with largest white neighborhood from **nonblack** take $O(n \log n)$ time. After a vertex v is extracted from **nonblack** and added to **ds**, we have to update white neighborhood sizes associated with vertices in `nonblack`. Now note that for each vertex that changes from white to gray or black, we update its neighbors' values in **nonblack**. A vertex changes from white to some other color only once and therefore for each edge we perform this update at most twice. Updates of these values (via `decreaseValue`) take $O(\log n)$ time. Thus the total time to update sizes of white neighborhood sizes is $O(m \log n)$. Thus the total running time of this algorithm is $O((m + n) \log n)$.

(c) The data structure that can fulfill the runtime specifications is a *max-heap* implementation of a priority queue data type..