Question 1:

1.1 NO_{Δ}

- Such a certificate could be a graph $c = \{(V_1, E_1), (V_2, E_2), ..., (V_n, E_n)\}$, where each V_i is an vertex and each E_i is an eage. This could be used to verify that G is in language NO_{Δ} by:
- First we check if the graph c is successfully partited into two parts by using DFS, We loop through the list and this can be done within linear time: $O(V + E)$.
- Group vertices into two parts. Within every part, try every possible combination of triple vertices and check if there exist 3 eages that connect each vertex with the other two, this can be done at most $O(n^3)$ time.
- If c is successfully partited into two parts and there is no such triple vertices within every part, we can verify that G is in language NO_{Δ}

1.2 ALL_{Δ}

- Same idea with different result
- Such a certificate could be a graph $c = \{(V_1, E_1), (V_2, E_2), ..., (V_n, E_n)\}$, where each V_i is an vertex and each E_i is an eage. This could be used to verify that G is in language ALL_{Δ} by:
- First we check if the graph c is successfully partited into two or more parts by using DFS, We loop through the list and this can be done within linear time: $O(V + E)$.
- Group vertices into two or more parts. Within every part, try every possible combination of triple vertices and check if there exist 3 eages that connect each vertex with the other two, this can be done at most $O(n^3)$ time.
- If c is successfully partited into two or more parts and we found at least one such triple vertices within every part, we can verify that G is in language ALL_{Δ}

Question 2:

2.1

The problem is the first part, alough a NFA can be converted to an equivalent DFA, that does not mean time complexity for constructing it must be within polynomial. In a DFA, any given input can only reach at most one state, however there could be multiple states for one given input in NFA, and for m states, this can go up to at most 2^m, beyond P.

2.2

- As the given algorithm suggests, we need to think about each step of transition in NFA.
- Use BFS algorithm: From the initial state Q_{init}, we look for all possible transitions labeled with current symbol and then add them to a set Q_{mid}. Then we follow all possible ε-transitions from these states and keep repeating this process until there are no more ε-transitions to follow. A new set of state Q_{next} is generated. We then repeat this process on every character in the input string w.
- Time complexity anaylsis: For input string $|w| = m$, each state can have at most n transitions leaving it on each character, and there are $O(n)$ states in Q_{init} and we need to search for $O(n)$ transitions per state. So the time to iterate over all the characters in Q_{init} to find the set Q_{mid} is $O(n^2)$. From there, we have to keep following ε-transitions until we run out of transitions to follow. The NFA can have at most $O(n^2)$ ε-transitions so runtime for each character is $O(n^2)$ and thus total time complexity is $O(mn^2)$, within P.

Question 3:

3.1

- Difine a certificate
- Let L_1 and L_2 be languages in NP. Also for $i = 1, 2$, let $V_i(x, c)$ be an algorithm s.t. for a string x and a possible certificate c, verifies whether c is actually a certificate for $x \in L_i$. If certificate c verifies $x \in L_i$ then $V_i(x, c) = 1$ else 0. Because L_1 and L_2 are in NP, we know that $V_i(x, c)$ runs in polynomial time $O(|x|^d)$ for some constant d.
- For $L_{1\cdot 2}$, we construct a verifier $V_{1\cdot 2}(x, c)$ for a string x and the possible certificate c that also runs in polynomial time. Suppose $|x| = n$. We can define $V_{1\cdot 2}(x, c) = 1$ iff $c = k * y * z$ where k is the set of all non-negative integers and $V_1(x_1 ... x_k, y) = 1$, $V_2(x_{k+1} ... x_n, z) = 1$.
- Verification
- k indicates the position where the input string x should be split into two parts, and y and z are the certificates for the two parts. So $|x_1 ... x_k| \leq |x|$ and $|x_{k+1} ... x_n| \leq |x|$. And also $V_{1\cdot 2} = 1$ if and only if $x \in L_{1\cdot 2}$. So $V_{1\cdot 2}$ will run in time $O(|x|^d)$. Thus, the language $L_{1\cdot 2}$ is also in NP.

3.2

- For any L, let M be the TM that decides it in polynomial time. We construct a TM M' that decides the complement of L in polynomial time:
- $M' =$ on input $\langle w \rangle$, run M on w. If M accepts, reject, else accept.
- M' decides the complement of L, since M runs in polynomial time, M' also runs in polynomial time.

3.3

- If $P = NP$, then since P is closed under complement, so is NP.
- So if we can find a TM to decide a language in NP, we can find a P machine to decide that language.
- Then the complement of NP is also decideable by switching the accpet and reject state of P machine.
- However in reality switching the states does not necessarily give a new machine to decide the complement.
- $\overline{NP} \neq NP$ and so $P \neq NP$.

Question 4:

4.1

- Use BFS:
- Choose one vertex s to start with and give it one color
- Move on to all its neighbors and color them with same color
- Move to all neighbors of the neighbors and color them and repeat this process until all vertices are given color
- Time complexity for this is linear

4.2

- Use BFS:
- Choose one vertex s to start with and give it one color
- Move on to all its neighbors and color them with another color
- Move on to all neighbors of the neighbors and color them with different color from former vertices. Repeat until all vertices are given color
- If after this process we find every two vertices pair with an edge connecting them have different colors, we can verify that the graph is in the language.
- Time complexity for this is also linear

Question 5:

- Demonstrate $COLOR_k$ is polynomial time reducible to $COLOR_{k+1}$:
- Given a graph G_k s.t. it is painted with k colors, and we create a new vertex with the $k+1$ th color that is not present in graph G.
- Connect the new vertex with other vertices with eages and create a new graph G_{k+1}, so in G_{k+1}, there are total $k+1$ colors and no vertices pair connected by an eage have the same color.
- So if we use it in a reversed way: Given a graph G_{k+1} with $k+1$ colors and we find the vertex that has the unique color, we can remove that vertex and get a graph G_k with k colors, we know it is k-colorable. So $COLOR_k \leq_P COLOR_{k+1}$.
- Given the fact that $COLOR_3$ is NP-complete, so for all $k >= 3$, $COLOR_k \leq_P COLOR_{k+1}$, all are NP-complte, so the rule applies to all with $k >= 3$.

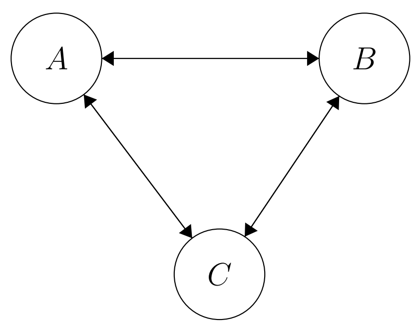
Question 6:

6.1

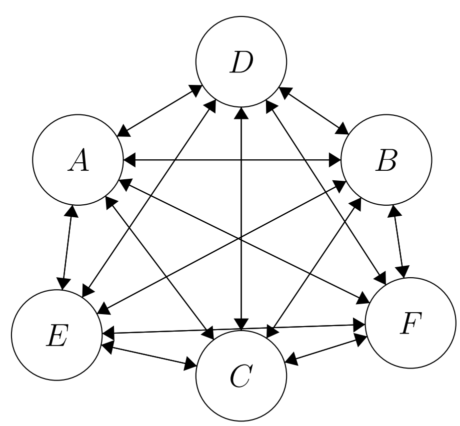
c would be 3 sets of vertices along with all the eages that still exist after cutting.

6.2

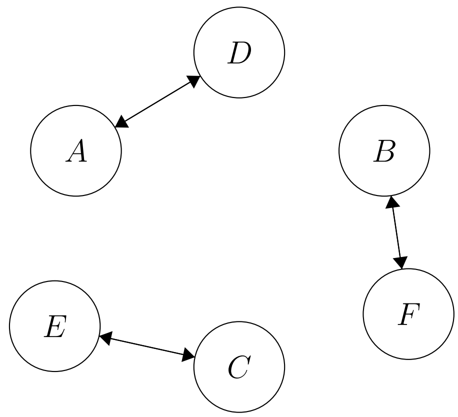
- Demonstrate $G \in COLOR_3 \iff G' \in FOREST_3$:
- Create a graph G that is 3-colorable. Since there are in total 3 colors, we know for G to be 3-colorable, every vertex can have at most 2 eages connecting to other vertices.
- For example:



- Then we add three more vertices and make sure each vertex is connected to every other vertex.



- In total, 6 vertices with 15 eages.
- For successful division, we split the graph into 3 sets:



- Notice that if we treat one set, for example $\{A, D\}$ as a whole, we can continue to add new vertex to it without creating a cycle in it, and this applies to the other two sets as well.
- So it is possible to add infinite vertices to the graph and still find a way to divide it. So we know that if G is 3-colorable, then G' is 3-forestable.
- In reverse, if we know G' is 3-forestable, then we can certainly find the vertices that connect to a vertex with same color with an edge, and by deleting these vertices, we are able to get a graph G that is 3-colorable.