1) Bipartite Graphs

To prove the equivalencies of statements (a), (b), and (c) we must show that each statement implies the other two. We go about this by showing equivalencies step by step:

- (a) $\Leftrightarrow$ (b): Given a bi-partite graph G, it vertices can be colored using only 2 colors so every edge is between vertices of opposite colors.
  - (a) $\Rightarrow$ (b): Take the bi-partite graph G with its vertex split into 2 disjoint sets L & R, and are positioned in such a way that all edges have 1 endpoint in each of these sets. L has vertices of one color and R of another. Because all edges have an endpoint in L or R, all edges connect vertices of different colors.
  - (b) $\Rightarrow$ (a): If the vertices of G can be colored only using 2 colors so every edge is in between vertices of opposite colors, we can define the sets L & R as the 2 color classes. Since every edge connects different color vertices, this means that every edge has one endpoint in L and the other in R, making G a bi-partite graph, proving (b) $\Rightarrow$ (a).
- (a) $\Leftrightarrow$ (c): If the graph G is bi-partite, it has no odd length cycles.
  - (a) $\Rightarrow$ (c): Since G is bi-partite, the cycle must have an even number of vertices to stay true to the alternating pattern of colors along the cycle. However, if it was an odd length cycle, it requires an odd amount of vertices which contradicts the assumption that G is bi-partite. G cannot have any odd length cycles, proving (a) $\Rightarrow$ (c)
  - (c) $\Rightarrow$ (a): supposing G has no odd length cycles, we can use BFS or DFS to show that G is bi-partite. Starting from a random vertex, we assign it to a color, and all of its neighbors to the other color, continuing this process. If an edge is encountered connecting 2 vertices of the same color, it is an odd length cycle, which would contradict our first given statement. Therefore if we are able to color all vertices without encountering this then G must be bi-partite, proving (c) $\Rightarrow$ (a).
- (b) $\Leftrightarrow$ (c): Transitive Property

Linear time algorithm:

1. Start with an empty set for L & R.
2. Assign a vertex v to L
3. Assign each neighbor (u) of v to R
4. Assign each neighbor (w) of u to L (if it has not already been assigned)
5. Continue using BFS or DFS, assigning the neighbors of vertices to alternating colors until there are no more vertices or we come across an inconsistency
6. If an inconsistency is found, the graph is not bi-partite. If the process is able to run until there are no more vertices, the graph is bi-partite

This algorithm visits each vertex and its neighbors exactly once, making it a linear-time algorithm and giving it a running time of $O(V+E)$.

2) Binary Tree Exploration

a. 1. Perform a DFS traversal of the binary tree starting at the root vertex
   2. During the DFS traversal, record discovery time and finish time for each vertex. (Discovery time is the time when the vertex is first encountered during the DFS traversal. Finish time is when the DFS exploration of the subtree is completed)
   3. Store the discovery and finish times in 2 separate arrays
   4. For any pair (x, y), we can answer whether x is an ancestor of y:
      - if the value at the index of x in the discovery time array is less than the value at the index of y in the discovery time array and the value at the index of x in the finish time array is greater than the value at the index of y in the finish time array, then x is an ancestor of y so answer YES
      - Otherwise, x is not an ancestor of y, so answer NO.

The preprocessing step takes linear time $O(V+E)$ as it involves a single DFS traversal of the binary tree, and answering queries takes $O(1)$ time because it only requires comparing the discovery and finish times of the 2 vertices

b. 1. Perform a DFS traversal of the binary tree starting at the root vertex
2. For each vertex during the DFS traversal, keep track of the maximum s-value among its descendants
3. Initialize an array with all elements as $-\infty$
4. When a vertex is being visited, update the value at the index of that vertex in the array to the maximum s-value of that vertex and the maximum s-value among its children
5. After the DFS traversal is complete, A[vi] = a[vi] is given by the array

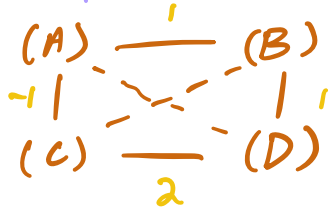
The DFS traversal takes linear time dominating at $O(V+E)$ as it visits each vertex and edge exactly once. The update step for the array takes constant time for each vertex. Hence, the time complexity of this algorithm is linear, $O(V+E)$.

3) Dijkstra's "Fix"

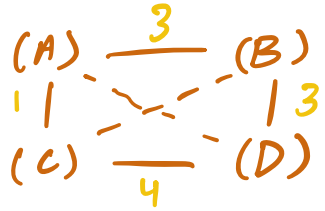
a. Using the information that a binary heap is used, the time complexity of this algorithm is $O((V+E)\log V)$:
- finding weight of largest edge takes $O(E)$ time
- adding max to all the edge weights takes $O(E)$ time, since we need to iterate through all of the edges
- Applying Dijkstra's algorithm takes $O((V+E)\log V)$ time. The decrease key operation takes $O(\log V)$ time and is performed at most E times.

Hence the overall time complexity is $O((V+E)\log V)$.

b. We can use this example to help prove/disprove:



we can apply the fix:



Now we can run Dijkstra's algorithm with modified edge weights:

initial distances: $A=0, B=\infty, C=\infty, D=\infty$

post process A: $A=0, B=3, C=1, D=\infty$

post process B: $A=0, B=3, C=1, D=4$

post process C: $A=0, B=3, C=1, D=4$

post process D: $A=0, B=3, C=1, D=4$

The original path from A to D has a length of 1, however the modified graph has this length set to 4. this shows that this fix does not guarentee preservation of shortest paths, and this modified algorithm may produce incorrect results. while it ensures non-negative edge weights and allows the use of Dijkstra's algorithm, it does not guarentee correctness of shortest paths in the modified graph.

4) Tank Capacity

a. 1. Create an adjacency list representation of cities and highways
2. Initialize a boolean array to keep track of visited cities
3. Implement a DFS traversal from the starting city s:
   - mark s as visited
   - For each neighboring city (v) of s:
     - If $l_{sv} \leq L$, or if highway is reachable within the fuel tank capacity:
       - If v is not visited, recursively call DFS on v.
4. After the DFS traversal, if the destination city t is marked as visited, then it is reachable from the city, otherwise it is not.

The DFS traversal takes O(V+E) time complexity, which is linear in the input size.

b. 1. Initialize the search interval as [low, high], where low = 0 and high = ∞ (a large value)
2. while low < high:
   - Calculate middle
   - modify the graph by removing all edges with edges greater than mid
   - use the algorithm from part a to determine reachability from s to t in the modified graph
   - if t is reachable, set high = middle, if not then set low = middle + 1

3. The minimum tank capacity needed to travel from city s to city t is low

This binary search process performs log(high), and iteration involves running the reachability algorithm from part a, which has a linear time complexity, giving the algorithm a time complexity of O((V+E)log(high)).

5) Extra Credit

We may use a BFS based algorithm:

1. initialize an array to store the distances from s to each vertex. Set the index value of s in the array to 0, and the index value of v to ∞ for all other vertices $v \neq s$.
2. Initialize another array to store the number of shortest paths from s to each vertex. Set the index value of s in this array to 1 and the index value of v to 0 for all other vertices to $v \neq s$.
3. Create a queue Q and enqueue vertex s.
4. while Q is not empty:
   - dequeue the front vertex u from Q
   - for each neighbor v of u:
     - if the index value of u in the distance array + 1 is greater than the index value of v in the distance array, then update the index value of v in the distance array to the index value of u in the distance array + 1, and the index value of v in the count array to the index value of u in the count array
     - if the index value of u in the distance array + 1 is equal to the index value of v in the distance array, then update the index value of v in the count array to increment by the value at the index of u in the count array.
     - if the index value of v in the distance array has changed, enqueue v into Q
5. The number of distinct shortest paths from s to t is given by the value at index t in the count array.

This algorithm works assuming all edge lengths are 1, and iteretively propogating the shortest paths and their counts from the source vertex s to other vertices using BFS, ensuring only vertices reachable by the shortest paths from s to t. The time complexity of this algorithm is linear O(V+E), since each vertex and edge is processed exactly during the BFS traversal.