1. **Solution:** (a) A hypergraph $G = (V, \mathscr{E})$ can be modeled using an undirected graph $G' = (V', E')$, where

$$V' = V \cup \mathscr{E}$$
$$E' = \{(v, e) \in V \times \mathscr{E} \mid v \in e\}$$

In other words, the vertices of G' include one vertex to represent each hyperedge in $\mathscr{E}$ in addition to the existing vertex set V. There would be an edge from $v \in V$ to a vertex representing the hyperedge $e \in \mathscr{E}$ if and only if v is one of the endpoints of the hyperedge e.

The algorithm will return whether the hypergraph is connected.

Algorithm:
- Construct G' as described
- Perform breadth-first search on G' starting from an arbitrary vertex in V, counting the number of vertices in V that are visited
- Return true if the count is equal to $|V|$, and false otherwise

Claim 1. *For any vertices u and v, there is a path from u to v in the hypergraph G if and only if there is a path from u to v in the graph G'.*

Proof: ($\Longrightarrow$): Let $v_1, e_1, v_2, e_2, \ldots, v_n$ where $v_1 = u$ and $v_n = v$ be a path from u to v. By the definition of paths in hypergraphs, $v_i \in e_i$ and $v_{i+1} \in e_i$. The definition of E' implies that undirected edges (v_i, e_i) and (e_i, v_{i+1}) exist in G' for all $i \in \{1, \ldots, n-1\}$. Therefore, the path $v_1, (v_1, e_1), e_1, (e_1, v_2), v_2, (v_2, e_2), e_2, \ldots, v_n$ is a path from u to v in G'.

($\Longleftarrow$): Note that G' is a bipartite graph with V and $\mathscr{E}$ as the two partitions. Therefore, any path in G' must have vertices from the two sets V and $\mathscr{E}$ in an alternating fashion. Let $v_1, (v_1, e_1), e_1, (e_1, v_2), v_2, (v_2, e_2), e_2, \ldots, v_n$ where $v_1 = u$ and $v_n = v$ be a path from u to v in G'. Because (v_i, e_i) and (e_i, v_{i+1}) are edges in E', the definition of E' implies that $v_i \in e_i$ and $v_{i+1} \in e_i$ in G for all $i \in \{1, \ldots, n-1\}$. Therefore, the path $v_1, e_1, v_2, e_2, \ldots, v_n$ is a path from u to v in G. □

It follows that the number of vertices in V reachable from any vertex in V must be the same in G and G'. Thus, G is connected if and only if all vertices in V are reachable from each other in G'. In addition, note that the proof gives a one-to-one correspondence between any path from u to v in G and G' (this correspondence will be used in part (b)).

Note that $|V'| = |V| + |\mathscr{E}|$ and $|E'| = \sum_{e \in \mathscr{E}} |e|$. Construction of G' can be performed in $O(|V'| + |E'|)$ time, and BFS on G' can be performed in $O(|V'| + |E'|)$ time as well. Therefore, the total running time of the algorithm is $O(|V'| + |E'|) = O(|V| + \sum_{e \in \mathscr{E}} |e|)$ time, which is linear in the size of the input.

(b) We can model the problem by creating one vertex per person. We create one hyperedge for each group, where its endpoints are the vertices corresponding to people in that group. The algorithm will return the number of messages needed for u to reach v, which is exactly the number of hyperedges in a shortest hyperpath between u and v.

Algorithm:

- Construct G' as described in part (a)
- Perform BFS on G' starting at $u \in V'$
- $k \leftarrow$ the length from u to v in the BFS tree
- Return $k/2$

Just as in the previous part, the running time of this algorithm is $O(|V| + \sum_{e \in \mathcal{E}} |e|)$, that is, linear in the size of the input.

Brief Explanation:

The number of messages needed for u to reach v is exactly the number of hyperedges in a shortest hyperpath between u and v in G.

Performing a BFS on G' starting at u produces a shortest path of even length, say k, between u and v in G. Recall from the proof of the claim in the solution of part (a) that the elements of $\mathcal{E}$ this path visits correspond exactly to the hyperedges in a hyperpath of length $k/2$ between u and v in G. This hyperpath is a shortest hyperpath: assume there exists a hyperpath of length $\ell < k/2$ from u to v in G, then the hyperpath corresponds to a path from u to v in G' of length $2\ell < k$, a contradiction since BFS gives the shortest path.

In other words, recall the one-to-one correspondence between any path from G to G'. Note that the correspondence is between a path of length ℓ in G to a path of length 2ℓ in G'. Therefore, finding a shortest path in G is equivalent to finding a shortest path in G', taking into account the multiplicative factor of 2.

Rubric: Out of 10 points:
- 6 points for part (a)
  - 3 points for a correct algorithm
  - 3 points for proof of correctness and correct runtime analysis
- 4 points for part (b)
  - 3 points for a correct algorithm
  - 1 point for correct runtime analysis

2. **Solution:**

- Changing the length of e can only change the length of paths including e. Since a shortest path tree must contain a shortest path to each vertex, changing the length of e can change the shortest path tree only if it changes the shortest path to some vertex to a path using e. If e is in the shortest path to some vertex, it must be in a shortest path to q (since if there was a shorter path to q not using e than any path to q using e, every path using e could be made shorter by using this alternate path to q).

  The shortest path to q which uses e is clearly the shortest path to p followed by the edge e, so decreasing $\ell(e)$ will cause T to become invalid if and only if it causes $d(s,p) + \ell(e)$ to become less than $d(s,q)$, that is if we decrease $\ell(e)$ by at least $(d(s,p) + \ell(e) - d(s,q) + 1$

  Since we assumed that we can access $d(s,v)$ in constant time for any v, computing $(d(s,p) + \ell(e) - d(s,q) + 1$ takes $O(1)$ steps.

- Increasing the length of e only makes paths using e worse, so it cannot change the shortest path to any vertex whose shortest path does not currently use e. Thus, if increasing the length of e causes T to become invalid, it must be because it causes a different path to be shorter than the one in T for reaching q or some descendant of q in T, that is some vertex in the subtree T' rooted at q.

For any vertex v in the subtree of T', when we increase the length of e by k, we increase the length of the path to v in T by k as well. Thus, increasing the length of e by k will cause T to be invalid if and only if some such v can be reached by a path of length less than $d(s,v)+k$ which does not use e.

Let $d'(s,v)$ be the length of the shortest path from s to v which does not use the edge e (or ∞ if no such path exists). Then the maximum amount that we can increase the weight of e by without causing T to become invalid is $k = \min_{v\in T'} d'(s,v) - d(s,v)$.

Fix some vertex v in T' such that $d'(s,v)-d(s,v)=k$, the minimum value, and the number of edges in the shortest path from s to v with the fewest edges is less than or equal to the number of edges in the shortest path from s to t with the fewest edges for any other $t \in T'$ with $d'(s,t)-d(s,t)=k$. Let (u,v) be the last edge in a shortest path from s to v with the minimal number of edges.

We prove that u must be in $T \setminus T'$. If u was in T', then we have that $d'(s,v) = d'(s,u)+\ell(u,v)$ and $d(s,v) \le d(s,u)+\ell(u,v)$ (since the shortest path to v using edge e may or may not go through u), and thus, $d'(s,u) = d'(s,v)-\ell(u,v)$ and $d(s,u) \ge d(s,v)-\ell(u,v)$, and so $d'(s,u)-d(s,u) \le d'(s,v)-d(s,v) = k$. Thus, if u was in T', then it would also have the minimum difference between the length of the shortest path to it not using e and the length of the shortest path using e, but the number of edges in the path to u with the fewest would be less than the number of edges in the shortest path to v with the fewest edges, since the path to u is a subpath of the path to v. This would contradict our assumption above, and thus u cannot be in T'.

Since $u \in T \setminus T'$, the shortest path to u in the original graph already didn't use e, and increasing the weight of e cannot possibly change this. Thus, we find that $d'(s,v) = d(s,u)+\ell(u,v)$. We can compute this value without knowing u by simply taking the minimum of this value across all valid choices of u, since we can create a path to v by taking the shortest path to some other vertex with an edge to v and then following that edge. Thus $d'(s,v) = \min_{u\in T\setminus T'} d(s,u)+\ell(u,v)$.

As noted above, the maximum amount that we can increase $\ell(e)$ by without rendering T invalid is $\min_{v\in T'} d'(s,v)-d(s,v)$ which we now know is equal to $\min_{v\in T'}(\min_{u\in T\setminus T'} d(s,u)+\ell(u,v)) - d(s,v)$. The minimum integer amount that we can increase by to render T invalid is obviously just this value plus one.

Putting all of the above observations together, we obtain the following algorithm:

```
FindCostIncrease?($G=(V,E), L, D, T, e=(p,q)$):
  //G is the graph, L the edge lengths, D[j] contains the shortest path from $s$ to $j$
  //T is the shortest path tree and $e$ is the edge we're reweighting
  Run DFS from $q$ in $T$ to find $T'$, the subtree rooted at $T$
  Create an array $A$ from this DFS where $A[i]$ is 1 if $i \in T'$ and 0 otherwise.
  $k = \infty$
  For $(u,v) \in E \setminus \{e\}$
      if $A[u] = 0 \wedge A[v] = 1$
          $k = \min(k, D[u]+L[u,v]-D[v])$
  return $k+1$
```

Finding T' takes $O(n)$ time since we are running DFS on a tree. The remainder of the algorithm is iterating over the edge list of G and doing constant work for each edge, so it runs in $O(m)$ time, and thus our total runtime is $O(m+n)$.

■

Rubric: Out of 10 points:

- part (a) 3pts:
  - 2pts Correct algorithms
  - 1pts Brief explanation
- part (b) 7pts:
  - 4.5pts Correct algorithm
  - 1.5pts Brief correctness justification
  - 1 pt Runtime analysis

3. **Solution:** (a) Let $n := |V|$, $m := |E|$. If the graph is a DAG, then we can compute the topological sorting of the vertices and use it to renumber the vertices by $1..n$ such that $i < j$ for any edge $(i, j) \in E$. Then, we can use a standard dynamic programming algorithm to solve the problem. Let $\text{OPT}[i]$ denote the maximum amount of candy one can collect starting from vertex i. Therefore, we have the following recurrence:

$$\text{OPT}[i] = \begin{cases} w(n) & \text{if } i = n \\ w(i) + \max_{j:(i,j)\in E} \text{OPT}[j] & \text{if } 1 \le i < n \end{cases}.$$

We can use an array to memorize OPT and fill the array in decreasing order of the indices. Then $\text{OPT}[s]$ is the answer we return, where we also use s to denote the number that vertex s is assigned to in the topological ordering.

If the graph is strongly connected, then one can visit all the vertices of the graph starting from any vertex in the graph. Therefore, if one can reach any vertex of any strongly connected components, then one can collect candies at all the vertices in this strongly connected component. More formally, we claim that if the walk P in G starting from s that collects the maximum amount of candies visits any vertex u, then there exists a walk that collects no less amount of candy and visits all vertices in the strongly connected component C containing u. Since C is a strongly connected component, we can find a walk $W(u)$ that starts from u, visits every vertex in C, and ends with u, by fixing an ordering $u = v_1, v_2, \dots, v_n$ of the vertices in C and concatenating the path between consecutive nodes and the path from v_n to u. Consider the walk P' obtained by replacing u in P with $W(u)$. P' is a walk that collects no less candies than P since P' visits all vertices that P visits and potentially more vertices each of which contains non-negative amount of candies.

Therefore, from the above observation, we can compute the meta-graph $G^{SCC} = (V', E')$ of G and for each $v' \in V'$ that corresponds to a strongly connected components C in G, we set $w(v') = \sum_{v\in C} w(v)$. Since G^{SCC} is a DAG, we can apply the dynamic programming algorithm above on the graph. Let s' be the vertex in V' that corresponds to the strongly connected components containing s in G. Then we return $\text{OPT}[s']$ as the maximum amount of candies one can collect starting from s.

The running time of the algorithm is $O(m+n)$ since computing G^{SCC} takes $O(m+n)$ time, and computing the topological sorting and running the DP on G^{SCC} each takes $O(|E'|+|V'|) = O(m+n)$ time.

(b) The algorithm is roughly the same as in part(a), the only difference is that instead of returning $\text{OPT}[s']$, we return $\max_{v' \in V'} \text{OPT}[v']$ since we can start the collecting walk from any vertex. The running time of the algorithm does not change as the comparison in the end takes $O(|V'|) = O(n)$ time, which is domnated by the other parts of the running time.

■

Rubric: Out of 10 points:

- part (a) 8pts:
  - 3pts for the DP on DAG (1pt for topological sorting, 1pt for recurrence, 1pt for evaluation order and return value)
  - 1pt for the observation on SCC, i.e, some form of the statement "one can reach any vertex of any strongly connected components, then one can collect candies at all the vertices in this strongly connected component", which serve as the brief explanation why we can create meta-graph and assign weights as we did. A formal proof as given in the solution above is not needed.
  - 2pts for correctly using the observation to create meta-graph and assign weights to vertices in meta-graph
  - 1pt for using the DP on DAG on the meta-graph with correct return value
  - 1pt for running time analysis

  If a pseudo-code or a clear description of the whole algorithm (including creating meta-graph, assigning weight, finding topological ordering of the vertices, and running DP) is present with brief explanation on why this graph modeling procedure is correct, then it suffices for all bullet points besides the time analysis.
- part (b) 2pts:
  - 1.5pts for correct return value
  - 0.5pt for running time