1a. Description of how your algorithm works ("in plain English").

The algorithm used to solve this problem is BFS(Breath-First Search) algorithm with a slight modification. Our BFS algorithm takes two inputs 'uq', 'vq'.

First, we do a simple if statement to check whether 'uq' equals 'vq', if it's equal, we print '1'. Then we initialise and declare a boolean array with the size of n to mark visited vertices. Afterwards, we declare a queue(FILO) 'q' to add vertices we are going to visit. Once the Boolean array and queue are declared, we insert 'uq' into the queue.

A while loop is created and while the queue is not empty the following steps are executed: First, we check whether the target vertex is visited, if it is we print '1' and exit the method straight away. Otherwise, we declare to variables 'current' and 'neighbour' to keep in track of the traversing. 'current' will be the last vertex removed from the queue 'q'. If the 'next' vertex is not visited before, we declare it as visited through the Boolean array. Secondly , we add the neighbour vertices of 'next' to the queue 'q'. To achieve this, we do a for loop traversing though all the vertices. In the for loop, we create the if condition to check whether the vertex we are visiting is a neighbour that has not been visited, if that is the case, we add it to our queue 'q'. Lastly, If we traversed all the paths from 'uq' and 'vq' was not discovered , we print '0'

b. Argue why your algorithm is correct.

Variables: Graph G, edge ->e $\in$ E, input : vertices uq, vq $\in$ V . Queue Q
Inductive claim: On an undirected graph G, BFS(uq, vq) reaches all vertices connected to uq through iteration of the queue Q at iteration k.
Base Case: Queue Q is initially empty.
Inductive Hypothesis: Assume kth iteration holds.
Inductive Step:

1. We add uq to the queue

Consider k + 1st iteration

2. Check whether vq has been visited through Boolean array at the beginning of each iteration.
3. Remove uq from the queue, form new front.
4. Enqueue neighbours of uq.

Return to step 2.
Corollary to Inductive Claim: At iteration n, if Boolean array[vq] is visited, a path exists. If at termination of BFS, vq is not explored, path does not exists.

c. Prove an upper bound on the time complexity of your algorithm.

We first build the graph: using an adjacency list and identifying all edges through in put takes O(m + n) time.[1]
When performing traverse through the neighbours of 'uq': we traverse O(deg(uq)), which takes O(|m|) in total to traverse all neighbours in the upper bound.

[1] Kleinberg, J & Tardos, E 2014, Algorithm Design, p.88.

Each loop we check whether 'uq' and 'uv' are connected: the size of Boolean array is n, hence this takes O(n) times.
Find path algorithm (BFS): takes O(m+n) times[2].(Kleinberg & Tardos, 2014) Moreover, the main method calls our find path function q times. Hence, the upper bound for calling and executing our find path algorithm takes O(q(m+n)) time.

Therefore, our final algorithm is O(n+m) + O(m) + O(n) + O(q(m+n)) = O(q(m+n)).

2.a. Description of how your algorithm works ("in plain English").

To solve this problem, I have implemented Kruskal's algorithm with a slight modification and union-find data structure is used.

This algorithm first sorts the edges by the weight, from minimum weight to maximum weight. (nlogn)Then we create a set for each vertex, which added to a list of sets. (n) We then select the edges provided by the input A, remove them from the array list of all edges and add them to the minimum spanning tree.
Afterwards, we create a while loop to iterate through the remaining edges. In the while loop: Firstly, we remover the first edge from the array list of edges(minimum weight) at each iteration. Secondly, we remove the edge (in sets form) from the list of sets. Lastly, we check if a cycle is formed by comparing the two sets, if cycle is not formed, we merged these two sets and ad the edge to the minimum spanning tree.
We finish it by iterating through all the edges in the minimum spanning tree and adding up the weights of all the edges of the minimum spanning tree.

Kruskal's Algorithm[3]:
Sort the edges so that: c(e1) ≤ c(e2) ≤ . . . ≤ c(em)
T ← ∅
for i : 1..m (*) if T ∪ {ei} has no cycle then
T ← T ∪ {ei}

end if

end for

b. Argue why your algorithm is correct.

The modification done in my algorithm is that the provided sets of edges A are removed from the graph represented as an array list of edges and added straight away to the minimal spanning tree.

Inductive claim: On an undirected graph G, my Kruskal algorithm produces a minimal spanning tree (MST) including the set edges A. A$\in$ E and A$\in$MST.
Base Case: MST is initially empty.
Inductive Hypothesis: According to Kabanets[4]Kruskal's algorithm outputs a minimum spanning tree for G.
Inductive step:

[2] Kleinberg, J & Tardos, E, 2014, Algorithm Design, p.91.
[3] Kabanets, V, 2011, Kruskal's Algorithm: Correctness Analysis, view from: http://tandy.cs.illinois.edu/Kruskal-analysis.pdf
[4] Kabanets, V, 2011, Kruskal's Algorithm: Correctness Analysis, view from: http://tandy.cs.illinois.edu/Kruskal-analysis.pdf

1. Add pre-selected edges A1…a to MST.
2. Remove pre-selected edges A1…a from the array list of edges. This step prevents the Kruskal algorithm to visit these edges and find absolute optimum MST.
3. Perform Kruskal's algorithm for the rest of the edges.
4. Produces MST tree, where A$\in$MST.

Corollary to Inductive Claim: At termination of the algorithm, the total weight of the MST is calculated . W(mst) = W(A) + W(Kruskal(G- A)), A represents the pre-selected edges.

c. Prove an upper bound on the time complexity of your algorithm.

- Adding edges: we create an array of edges to store the graph G, upper bound is O(m)time.
- Adding preselected edges: upper bound condition is that all edges are selected O(m)
- Collection sort in java: all edges are sorted in regards to their weight. O(mlogm), according to[5] (docs.oracle.com)
- Create a set for each vertex : loops n times , hence the time taken is O(n) +1 = O(n)
- Select preselected edges : supposed all edges are selected O(m) time
- Kruskal: is proven to be O(mlogn) [6]
- Adding weights: Supposed all edges are included in the MST, upper bound is O(m)

Hence, the final algorithm is O(m) + O(m) + O(mlogm) + O(n) +O(m) + (Omlogn) + O(m) = O(mlogn)

[5]http://docs.oracle.com/javase/7/docs/api/java/util/Collections.html#sort%28java.util.List,%20java.util.Comparator%29

[6] Kleinberg, J & Tardos, E, 2014, Algorithm Design, p.188-189