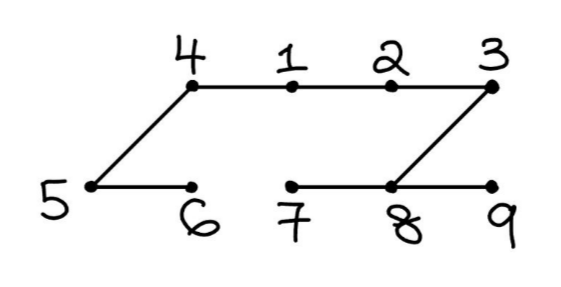
# 1a.

* 1 can go in four positions (1, 3, 6, or 8)
* Fix 1. Then 2 and 3 are fixed.
* 4 can go in two positions (4 or 5).
* Fix 4. Then 5 and 7 are fixed.
* 6 can go in two positions.
* Fix 6. Then 8 is fixed.

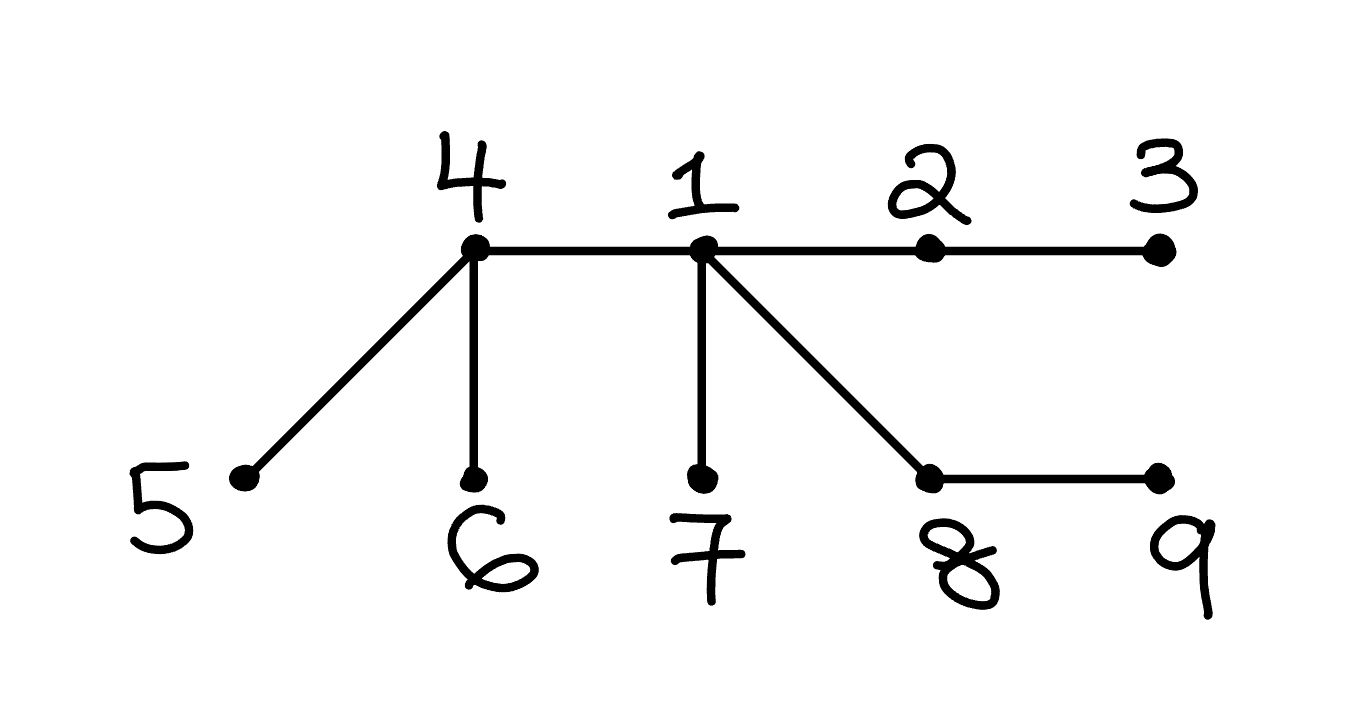
So there are 4 \* 2 \* 2 = 16 isomorphisms.

# 1bi)



1, 2, 3, 8, 7, 9, 4, 5, 6

# ii)



1, 2, 4, 7, 8, 3, 5, 6, 9

# C

Early returning would be nice but the template doesn’t let us early return so we’ll have to add the ugly if clause:

procedure findmin(i)

entered[i] = true

min[i] = i

for j in adj[i]:

if not entered[j]:

findmin(j)

min[i] = min(min[i], min[j])

for i = 1 to n:

if not entered[i]:

findmin(i)

# d

<https://leetcode.com/problems/longest-increasing-subsequence/>

for i = 0 to n – 1:

L[i] = 1

for i = 1 to n – 1:

for j = 0 to I – 1:

if A[i] > A[j]:

L[i] = max(L[i], 1 + L[j])

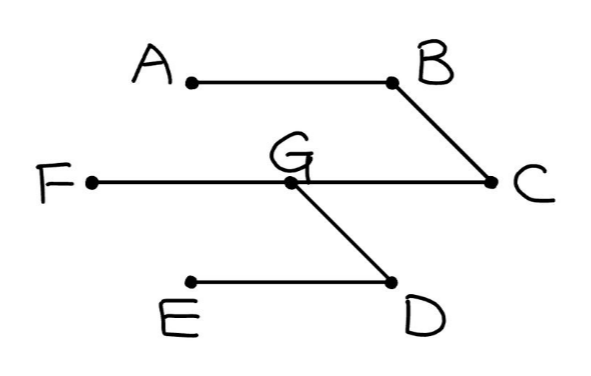
maxlen = 0

for i = 0 to n – 1:

maxlen = max(maxlen, L[i])

return maxlen

# 2ai)



# ii)

Yes

## Explanation 1:

The minimum spanning tree in i) uses all arcs with weight <= 4. Any spanning tree that does not use all of these arcs must have weight >= the weight of the MST in i), so all MSTs must include all arcs of weight <= 4. Now we have two connected components, one containing AB and another containing CDEFG. Since there is only one arc of minimum weight that joins these two connected components, namely, BC, any other spanning tree that uses a different arc must have weight >= weight of MST in i). So MST unique.

## Explanation 2:

The heaviest arc in the MST in i) is BC which has weight 5, and all other arcs in the MST have weight < 5. There is only one unused arc that has weight <= 5, namely, EG. In order to attempt to construct a distinct MST, we must take one of the following two steps:

* If we replace any arc in the MST with any unused arc with weight > 5, or replace any arc in the MST other than BC with EG, then the resulting subgraph (it may not be a spanning subgraph, nor a tree) must have weight > weight of MST in i).
* If we replace BC with EG, then e.g. A and G will not be connected and there will be a cycle including G.

Since both steps will not result in an MST (either violates minimum, spanning, or tree), the MST in i) is unique.

# iii)

A black line drawing of a structure

Description automatically generated

# 2b

n^2 comparisons in total.

First half of list takes 0 + 1 + … + (n – 1) = n(n-1)/2 comparisons

Second half of list takes 1 + 2 + … + n = n(n+1)/2 comparisons. Same as first half, but now the elements in the second half also have to compare with the last element in the first half.

# ci) (please, please study the tutorial sheets if you don’t know how to do this, these are free marks)

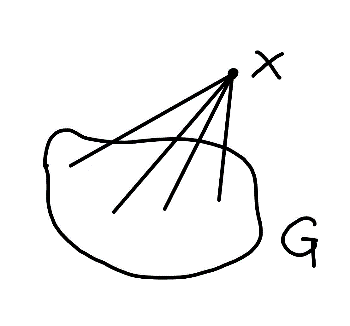
Given a clique C,

* |C| <= p(|G|) for some polynomial p, since number of nodes in C <= number of nodes in G by definition of clique
* Define Ver-CliqueN(G, x, k, C) iff C is a clique of size >= k containing x. Then it trivially follows by the definition of CliqueN that CliqueN(G, x, k) iff there exists C: Ver-CliqueN(G, x, k, C)
* Ver-CliqueN is in P, since we only need to check the following:
  + Size >= k, trivially polynomial time
  + C contains x, trivially polynomial time
  + All O(n^2) pairs of nodes are adjacent. Checking if one pair of nodes is adjacent takes p-time so checking p-many nodes takes p-time too
  + All nodes of C are in G, trivially polynomial time

## cii)

1. Show that Clique reduces to CliqueN

Define f(G) by adding a node x to G and then connecting every node in G to x.



f runs in p-time since we are adding one node (so p-many nodes) and p-many arcs.

We claim that Clique(G, k) iff CliqueN(f(G), k + 1, x), where x is the node added to G by f.

If: Suppose we have a clique C’ in f(G) of size >= k + 1 containing x. Since all pairs of nodes in C’ are adjacent, all pairs of nodes in C = C’ \ {x} are also adjacent. C has size >= k, and C is a subset of nodes(G) since x is the only node added to G to give f(G).

Only if: Suppose we have a clique C in G of size >= k. Then C’ = C union {x} is a clique in f(G) of size >= k + 1 containing x, since any pair of nodes in C that does not include x are adjacent to each other (since C is a clique in G) and any pair of nodes in C that include x are adjacent (by construction of f). Since x is not in G, x is not in C, so C’ is of size >= k + 1.

So Clique <= CliqueN.

1. Since Clique is NP-complete, Clique is NP-hard. Take any D in NP. By definition of NP-hard, D <= Clique. By transitivity, D <= CliqueN. This holds for arbitrary D in NP, so CliqueN is NP-hard. Since CliqueN in NP and CliqueN is NP-hard, CliqueN is NP-complete.