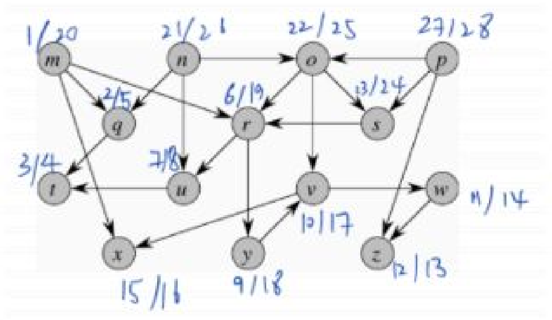
Solution - 8

1.O(n^2)

With an adjacency matrix, the time required to find all outgoing edges is O(n) because all n columns in the row for a node must be inspected. Summing up across all n nodes, this works out to O(n^2).

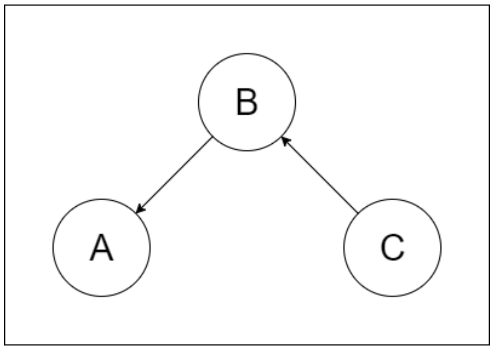
2.

The produced graph after running DFS:



The ordering of vertices produced by TOPOLOGICAL-SORT is $p, n, o, s, m, r, y, v, x, w, z, u, q, t$.

3. Consider the following graph,



If we perform depth first traversal, on this graph with alphabetical ordering, we find that all 3 nodes will lie in 3 different depth-first trees containing only themselves. Vertex B in particular satisfies the requirement that the vertex should have both incoming and outgoing edges. This situation is only possible when all the outgoing edges of the vertex have already been visited and none of vertices that the incoming edges are connected to have been visited.

4.

This problem can be solved using DFS. We do a dfs traverse on the tree and store the dfs start and end times for each node. The data corresponding to these times can be stored in 2 arrays start_time[] and end_time[] with the data in the data for node i and index i. If the node contain string labels we can use a HashMap/ordered_map/Dictionary mapping the node -> start_time, end_time. Once this is done pre-processing is complete. This operation takes O(n) [dfs + insert for each node O(1)*n]

Now, for any two nodes u and v:

```
isAncestor(u,v):
    If start_time[u] < start_time[v] && end_time[u] > end_time[v]:
        Print(u is an ancestor of v)
        Return
    If start_tim[u] > start_time[v] && end_time[u] < end_time[v]:
        Print (v is an ancestor of u)
        Return
    Print (No relationship)
    Return
```

5.

A directed graph is acyclic if and only if a DFS yields no back edge. Therefore, we use a modified version of DFS to solve this problem.

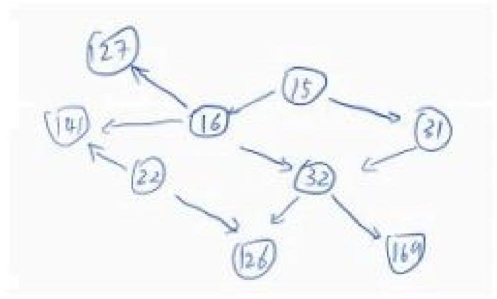
We first run DFS, and if there is an edge goes back to a visited vertex, the algorithm terminates. The way to check whether an edge is a back edge is to check whether v.d < u.d AND v.f = NIL.

The running time of DFS is O(E + V), so the running time of this algorithm is also O(E + V).

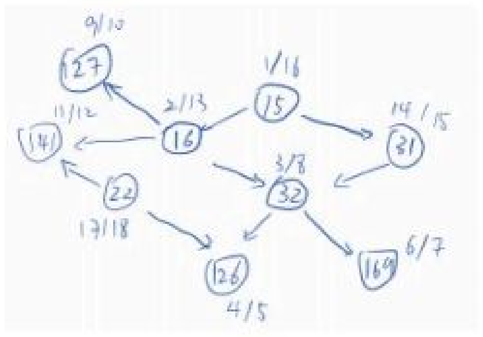
You can learn more here: http://www.geeksforgeeks.org/detect-cycle-in-a-graph/

6.

This is a typical topological sort problem. We make a directed graph with courses as 3 vertices and each vertex has incoming edges from its prerequisites.We obtain the graph below:



We call TOPOLOGICAL − SORT(G) on the graph. After first step calling DFS, we obtain:



And the topological sorted list: {LA22, LA15, LA31, LA16, LA141, LA127, LA32, LA169, LA126}. This is the order in which the student can take all courses with the prerequisites.

7.

There are 2 possible solutions for this question depending on whether you treat the graph as a DAG (directed acyclic graph). If you treat the graph as a DAG, then you can follow solution 1.

Solution 1: We can think of each intersection as a node, and each corridor as a directed

edge. Additionally, we a set start point, we can think of this as a tree, with the start point as the root. This reduces to what is known as the single source longest path problem. At this point, we can use a greedy algorithm to solve this problem.

We begin with a topological ordering of all nodes between the start and the end, and mark each node with a MAXDIST field and a MAXPRED field, set to negative infinity. We will also mark our start node with a MAXDIST of 0.

Now we consider our nodes in the aforementioned topological order. Starting with the start node we consider all edges outgoing from our node, and check their MAXDIST value. If node.MAXDIST < cnode.MAXDIST + edge_weight, then we update node.MAXDIST with cnode.MAXDIST + edge_weight, and node.MAXPRED with cnode. Once we have considered all of the nodes in the ordering (note that we ignore nodes inaccessible from ENTER), we just recursively trace back from end to start using the MAXPRED values.

This algorithm runs in O(V+E) time.

If you don't think the graph has to be a DAG, Solution 2 can handle more general cases.

Solution 2:

Main idea is use DFS to traverse all possible path and calculate sum of weight of each path.

```
class Node:
    weight = 0
    neighs = [] # list of Node

weight = -1
path = []

def find_path(entrance, end):
   DFS(entrance, end, [], 0, set())
   return path

def DFS(entrance, end, prev_path, prev_weight, visited):
   if entrance == end:
       if prev_weight + end.weight > weight:
           weight = prev_weight + end.weight
           path = prev_path + [end]
       return

   for neigh in entrance.neighs:
       if neigh in visited:
           continue

       visited.add(neigh)
       DFS(
           neigh,
           end,
           prev_path + [neigh],
           prev_weight + neigh.prev_weight,
           visited
           )
       visited.remove(neigh)

    return
```

8.

a. Using Induction to prove this.

Basic Step: If there are two martians, 1<-2 , P(2) is True

Induction Hypothesis: Assume P(k) is True, so 1<-2<-3<- … <- k

Induction Step: K+1 is new martian. We first put it in the first place. Check whether K+1 kiss right one(let it be Q). If yes, put it to right of Q. Else we swap K+1 with Q and do next check.It will finally find the right place.

So P(k+1) is also true.

Therefore, the theory is always true.

b. Two kinds of solutions are acceptable:

One is slightly change topological sort. But topological sort only works on DAG. You should mention how to process circle. (You can extend the topological sorting algorithm to deal with cycles by first finding the cycles of the set, then creating a set where all members of a cycle are replaced by a single placeholder. Next, topologically sort this smaller set. Finally, replace each placeholder with all the members of the corresponding cycle.)

Other is everytime insert one martian. Follow the step of induction proof.

The first one is better.