The Problem:

A user in Khan Academy

- either coaches one or more users
- and/or is coached by one or more users

A number of attributes such as *User Id, First Name, Last Name, Email Address, Physical Address, Phone No, etc.,* may be captured and stored in their database. The **Coach to Student relationship** is also captured and stored. Each of these users use a version of the software when they login to Khan Academy site.

Khan Academy rolls out new features to their users periodically. When such roll out happens, they take an incremental approach. Generally the new version of the software is initially released to a limited set of users; the effect is observed before it is rolled out to another set of users; and so on. This process continues till all the users are on-boarded to the new version of the software.

The set of users who get the new version is selected based on the following criteria:
If a user gets a new software,

- All the users (students) that the user coaches should get the same new software version (i.e. All the classrooms for the user)
- All the coaches who coach this user should also get the same new software version (i.e. All classes that the user is attending)

The above policies make sure that only one version of the software is used in any classroom and when a user gets new version he/she uses the same version in all his/her class rooms.

Due to these policies, Khan Academy wants to know the *total_infection(**u**)* for a given user. The *total_infection(u)* is that if the user **u** is selected to receive the new version of the software, which other users should get the same version to meet the above policies.

Another form of infection is *limited_infection(n).* In this, if Khan Academy wants to select around **n** users to roll out the new version, which set of users should be selected to keep the roll out close to the desired number of users while meeting the above policies.

An application needs to be developed to model the users and calculate total_infection and limited_infection. Also the application optionally should address some new optimizations or features based on the insight gained during the development.

Solution Approach:

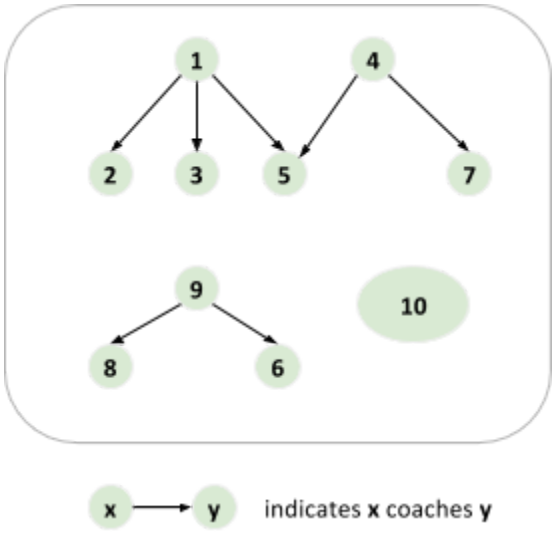
The real environment in Khan Academy may store the user information and the coach to student relationship in a database.

Assumptions:

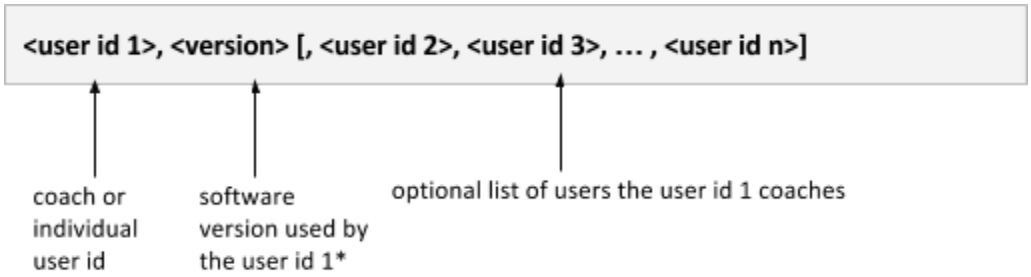
1. Due to lack of a database for the stand-alone application, it is assumed that the data elements are stored in a text file and fed to the application as input. (See File Format below.)
2. Also for efficient handling, it is assumed that the user ids are long numbers. But the same implementation approach will work with a slight adjustment of the data structures to accommodate any type of user id.

These assumptions will not hinder the flow of the application and the implementation of the algorithms.

As an example, consider the following relationship among the users:



Input File Format:



* The version information is not used by this stand-alone version as it is assumed that all the users use the same version in the beginning.

E.g.: The text file for the above graph:

1, 100, 2, 3, 5

4, 100, 5, 7

9, 100, 8, 6

10, 100

Data Structures:

In real scenario, the above information and the relationships are already stored in the database. For this stand-alone version and to implement total_infection() and limited_infection() methods, we need only the relationships to be available. So user objects are not created purposefully. Also when the input file is given it is assumed all users are using the same version. This is also not a big limitation of this application to demonstrate the working of the infection algorithms. So the following relationships are created in memory using the information given in the input file.

1. Coach to Students Relationship
2. Student to Coaches Relationship
3. Independent Users (in the project description it says we do not have to handle this, but I added it anyway)
4. A mapping of idToUser. This is just so that the anyone running the algorithm can see what version the user is on.

Coach to Students Relationship:

Hashmap<Long><Set<Long>> -- Hashmap of User (coach) to set of users (students)

User id (Coach)	**Set of User ids (Student Ids)**	**Remarks**
1	{2, 3, 5}	1 coaches 2 and 3
4	{5, 7}	4 coaches 5 and 7
9	{8, 6}	9 coaches 8 and 6

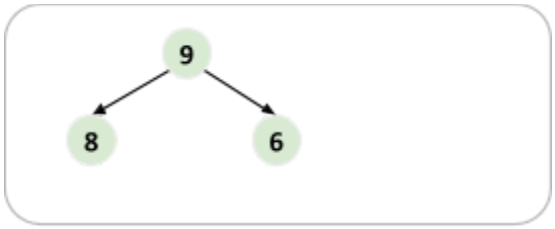
Student to Coaches Relationship:

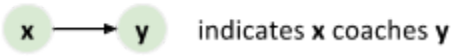
Hashmap<Long><Set<Long>> -- Hashmap of User (student) to set of users (coaches)

User id (Student)	**Set of User ids (Coach Ids)**	**Remarks**
2	{1}	2 is coached by 1
3	{1}	2 is coached by 1
5	{1, 4}	5 is coached by 1, 4, and 9
8	{9}	
6	{9}	
10	--	10 is not coached by anyone

Algorithm: *total_infection(userId, version)*

Total infection for a given user id is nothing but finding the connected subgraph that contains the user id. For example, total_infection(9) in the above diagram results in:





Algorithm Steps: *toatl_infection(userId, version)*

1. Set Inf_list = {*userId*} // inf_list is a set. So no duplicates are added to it.
2. Set visited_set = {}
3. For every **x** in inf_list:

   If **x** is not in visited_set

   a. Add all the students that **x** teaches to the inf_list (Use Coach to Student Hashmap)

   b. Add all coaches that teach **x** (Use Student to Coaches Hashmap)

   c. Add x to the visited_set i.e. Visited_set = visited_set U {x}

   d. Loop through Step 3

4. Return inf_list // This set contains all the user ids that are in the infection list surrounding *userId*

Algorithm: *limited_infection(noOfUsers, tolerance, version)*

Limited Infection for a given number of users with a tolerance limit is to find a connected subgraph that contains the desired number of users. In this case, the connected subgraph should have k users such that

(noOfUsers - tolerance) <= k <= (noOfUsers + tolerance)

When tolerance = 0, it is limited_infection with exactly the required noOfUsers.

1. Find all partitions in the graph and sort them according to the number of users in the partition (descending order)
2. allPartitions ← sorted list of all partitions
3. solutionStack ← new stack to hold partitions
4. Partitions ← findPartitions(noOfUsers, tolerance, noOfUsers, allPartitions, solutionStack)

findPartitions finds the set of partitions recursively:

Boolean findPartitions(totalRequired, tolerance, remaingRequired, partitions, solutionStack)

1. If sum of the users in *partitions* does not have enough number of users to meet the remainingRequired users, return FALSE // Not enough users to satisfy the remaining number of users needed
2. For i = 0 to numberOfPartitions in partitions
    a. Add i-th partition to the solutionStack
    b. noOfUsersAdded ← size of the i-th partition
    c. If the sum of the users in the partitions in the solutionStack meets the criteria, return TRUE
    d. Else reduce the 'partitions' list starting from (i+1) th position to have only the partitions that has (remainingRequired - noOfUsersAdded + tolerance) or less users
    e. Let reducedPartitions ← the reduced partitions
    f. Now find the the partitions for the new remaining number of users as follows: findPartitions(totalRequired, tolerance, remainingRequired-noOfUsersAdded, reducedPartitions, solutionStack)
    g. If the above call returns TRUE, then return TRUE
    h. Otherwise remove pop() the partition from the solutionStack and continue with the next iteration
3. The solutionStack contains the necessary partitions to meet the number of users for limited_infection with the tolerance limit

Algorithm: *exact_infection(noOfUsers, version)*

Exact Infection is the same as Limited Infection with the tolerance set equal to 0