Introduction to Algorithms: 6.006
Massachusetts Institute of Technology — March 20, 2014
Professors Srini Devadas, Nancy Lynch and Vinod Vaikuntanathan — Problem Set 4

Problem Set 4

All parts are due Thursday, April 10 at 11:59PM. Please download the .zip archive for this problem set, and refer to the `README.txt` file for instructions on preparing your solutions. Remember, your goal is to communicate. Full credit will be given only to a correct solution which is described clearly. Convoluted and obtuse descriptions might receive low marks, even when they are correct. Also, aim for concise solutions, as it will save you time spent on write-ups, and also help you conceptualize the key idea of the problem.

Part A

Problem 4-1. [20 points] **Using Breadth-First Search for Model-Checking an Algorithm**

Model-checking is a method wherein an algorithm or system is modeled abstractly, as a directed graph, where the nodes of the graph represent the possible *states* of the algorithm or system, and the edges represent possible transitions from one state to another. The models are typically *nondeterministic*, in that there might be many transitions from the same state s to other states, represented by many edges leaving the node labeled s. Model-checkers use Breadth First Search (BFS) and other searching techniques to explore the states of the graph, in order to determine whether or not all the states that are *reachable* from the start state satisfy some desirable safety property. The inventors of model-checking won the Turing award in 2007 for this work.

In this problem you will consider how a model-checker might use BFS to check a fundamental safety property of a basic concurrent algorithm. The algorithm in question is based on a simplified version of Peterson's 2-process mutual exclusion algorithm.[1] We have two agents, agent 1 and agent 2. At any point, each agent has a *color*, which can be white, pink, or red. White represents the idea that the agent is uninvolved, pink means that it is trying to obtain a resource, and red means that it has been granted the resource. The point of the algorithm is to ensure that we never have both agents colored red (granted the resource) at the same time.

In addition to the colors, the algorithm records somewhere who was the last agent to try to obtain the resource, that is, to turn from white to pink. Thus, a state of the system is modeled as a triple (COLOR(1), COLOR(2), LAST), where COLOR(i) may be WHITE, PINK, or RED, and LAST is either 1 or 2. The initial state is described by the triple (WHITE, WHITE, 1).

The behavior of the algorithm is described by a set of six simple formal *rules* that describe allowable transitions:

1. Starting from any state where COLOR(1) = WHITE, change COLOR(1) to PINK, and set LAST to 1.

[1]Peterson's 2-process mutual exclusion algorithm, "Myths about the Mutual Exclusion Problem"

2. Starting from any state where COLOR(2) = WHITE, change COLOR(2) to PINK and set LAST to 2.
3. Starting from any state where COLOR(1) = PINK and either COLOR(2) = WHITE or LAST = 2, change COLOR(1) to RED.
4. Starting from any state where COLOR(2) = PINK and either COLOR(1) = WHITE or LAST = 1, change COLOR(2) to RED.
5. Starting from any state where COLOR(1) = RED, change COLOR(1) to WHITE.
6. Starting from any state where COLOR(2) = RED, change COLOR(2) to WHITE.

These rules don't say that these transitions *must* occur—rather, they simply constrain what *may* occur during the execution of the algorithm.

(a) [2 points] How many total states does this algorithm have?

(b) [3 points] Draw the directed graph representing this algorithm; indicate the initial state.

(c) [3 points] Give its adjacency list representation.

(d) [3 points] Now we will see that not all of the states of the algorithm are actually reachable from the initial state. Describe an algorithm that outputs the reachable states of the Peterson algorithm, together with the length of the shortest path to each reachable state. You may assume the adjacency list representation of the graph.

(e) [3 points] What output is produced by your algorithm when applied to the Peterson algorithm graph?

(f) [3 points] Now recall that the entire point of the mutual exclusion algorithm is to ensure the "safety" property that we never have both agents colored red at the same time. Explain how we know that is true.

(g) [3 points] Your algorithm above used the adjacency list representation of the algorithm's model graph. Now describe a space-efficient version of your algorithm that verifies safety of the Peterson algorithm (that is, we never have both agents colored red in a reachable state), without ever writing down the entire adjacency list representation at any one time.

Problem 4-2. [20 points] **Depth-First Lasagna**

Marcella makes great lasagna, but she sometimes has trouble keeping track of the order in which to perform the various steps of the recipe. She knows that making lasagna involves the following steps:

1. Layer sauce
2. Boil noodles
3. Fry ground beef

4. Add tomatoes and herbs to fried beef and garlic
5. Grate cheese
6. Preheat oven
7. Bake lasagna
8. Fry garlic
9. Layer noodles
10. Layer cheese
11. Simmer sauce

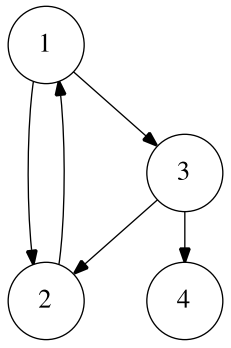
But she doesn't remember the order in which to do these. She does know that certain steps must be done before other steps:

- 6 should precede 7 and 9
- 2 should precede 9
- 8 should precede 4
- 3 should precede 4
- 4 should precede 11
- 11 should precede 1, and also should precede 6 (to avoid wasting energy)
- 5 should precede 10
- 9 should precede 1
- 1 should precede 10
- 10 should precede 7

(a) [3 points]

Draw a directed graph with states corresponding to the eleven steps and a directed edge from any step to any other step that must follow it. A topological sort of this graph will give a feasible order for performing the steps in order to produce a lasagna.

(b) [3 points] Emulate DFS on the graph and draw a picture of the resulting forest, plus a list of the order of events in which you discover and finish each node. Start with node 1 and when you must make a choice, choose the lowest-numbered unvisited node. For example, if the graph were:



Then your output would be:
Discover 1, Discover 2, Finish 2, Discover 3, Discover 4, Finish 4, Finish 3, Finish 1

(c) [3 points] Demonstrate that the order of *discover* events in your list in part (b) does not yield a correct order for the lasagna recipe.

(d) [3 points] Now consider the *reverse* of the order of *finish* events (topological sort). List the steps. Does this satisfy all the ordering constraints? Does this yield a sensible lasagna recipe?

(e) [4 points] The key to why this works is that, in performing the DFS of an acyclic directed graph, there are no situations where there is an edge in the graph from a node u to another node v, but u finishes before v finishes. Explain why no such situation arises, in your own words.

(f) [4 points] Now consider DFS for not-necessarily-acyclic directed graphs. Give a (very small, no more than four or five nodes) example of a graph that contains a path from a node u to a node v but v appears above u in a tree of the DFS forest, and another (very small) example where v appears to the right of u.

Problem 4-3. [20 points] **Social Network Structure**

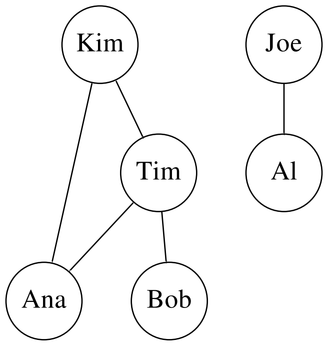
A social network, like Facebook, can be modeled as a very large undirected graph $G = (V, E)$. Here the vertices V are the people who are members of the network, and the edge set E consists of one edge for each pair of *friends*. We assume that the *friends* relation is symmetric: if X is friends with Y then Y is friends with X.

The network has some interesting features, which have some significance for social organization. First, a definition: Let's say that person X *knows* person Y if there is a finite sequence of friendship edges from X to Y in the network graph. That is, X is a friend of someone who is a friend of someone else who...who is a friend of Y. As special cases, we say that X knows himself/herself, and also knows all of his/her friends.

Also, define a *community* to be any set of people in the network such that everyone in the community knows everyone else in the community (according to the way we just defined *knows*) and no one in the community knows anyone outside the community.

Furthermove, a community may have one or more people who are classified as *hubs*. We define a *hub* of a community to be an individual H whose removal disconnects the community, that is, there are two other members of the community, say X and Y, for which the only paths between X and Y run through H. So with the removal of H, there is no remaining path in the graph between X and Y.

Consider the social network below consisting of 6 people. There are two communities, the first consisting of Kim, Ana, Tim, and Bob, and the second consisting of Joe and Al. The only hub in the social network is Tim, because if he were removed then Bob would no longer know Kim or Ana.



- **(a)** [4 points] Describe an algorithm that, given an undirected graph in adjacency list format, produces a list of of all the communities in the network represented by the graph. Each entry in the list should be a list of the members of one community, in any order. Your algorithm should run in time $O(|V| + |E|)$. Analyze its complexity.
- **(b)** [4 points] Consider a DFS tree for a graph representing a community. In terms of the information in the tree, we want to identify which community members are the hubs. First, under what conditions is the root of the DFS tree a hub? Prove your answer.
- **(c)** [5 points] Again consider a DFS tree of a graph representing a community. In terms of the information in the tree, describe which non-root nodes are hubs. Prove your answer.
- **(d)** [7 points] Describe an algorithm that, given an undirected graph in adjacency list form, produces a list of the hubs. Try to achieve $O(|V| + |E|)$ time complexity.

Part B

Problem 4-4. [40 points] **High-Precision Root-Finding Using Newton's Method**

Newton's method produces a sequence of successive approximations $x_1, x_2, x_3, \ldots$ for a root of a real-valued, differentiable function f, given an initial guess x_0. The method doesn't always converge (e.g., if the derivative is not continuous at the root), but it does converge for commonly-encountered functions, and for choices of x_0 that are reasonably close to the root. In fact, in those cases, it generally converges quadratically, i.e., the error bound is approximately squared at each step, which means that the number of significant digits in the answer approximately doubles at each step.

In this problem, you will implement a high-precision calculation of cube roots using Newton's method. Given a positive integer d representing the precision, and a positive integer y with d digits of precision, your program should produce a representation of the cube root of y, again with d digits of precision.

(a) [5 points] As a warm-up, let's use Python floating-point numbers, which have 15 digits of precision. Use Newton's method to calculate $7^{1/3}$.

As you do this, try to figure out how good your initial guess needs to be for the method to converge quickly. (You do not need to turn in an answer to this.)

Test this experimentally. First, try starting with an initial guess of 2. Print out the results for 10 iterations. Do the numbers appear to be converging quadratically? Now, see what happens if you replace 2 by a terrible guess like 1,000,000. Does the method seem to be converging quadratically for this guess?

(b) [10 points] Generalize your work in Part (a) to calculate $y^{1/3}$ for an arbitrary integer y having 15 digits of precision.

As you saw in the previous part, for this to work well, you will need a way to choose an initial guess that is reasonably close to the right answer. It should be sufficient for your initial guess to be an integer.

As you work on this part, try to figure out how much the guesses can continue to change once the method stabilizes (at the "best" answer). (You also do not need to turn in an answer for this, but it will be helpful to understand what is going on before working on the next parts of the problem.)

Test this experimentally. How much do the numbers change when you continue to perform Newton iterations to compute the cube root of 2 after 20 or more iterations? How much do they change when you do the same for the cube root of 26? How about 4204? Can you guess a general bound in terms of y?

(Note: You may not see this behavior with these particular numbers depending on several factors including the particular system you are running on. Here are some more numbers you can try: 1268, 9885, or 77714. Alternatively, you can write a loop (trying every number up to, say, a million) to find the numbers that exhibit the most change on your system.)

(c) [15 points] Now let's generalize to arbitrary precision d, not just 15.

In this part, you will implement part of a `Decimal` class that implements arbitrary precision decimal numbers. Much of this has been provided for you, but you will need to implement the routine for computing the reciprocal of a `Decimal` number.

Each `Decimal` number is represented as a pair (z, e). The first part, z, is an integer in the range $[10^d, 10^{d+1})$. In other words, it is an integer with exactly d decimal digits. The number e indicates the position of the decimal point. Specifically, the pair (z, e) represents the number $z \times 10^e$.

For example, the number 1.025 (with 4 digits of precision) would be stored as the pair $(1025, -3)$ since $1.025 = 1025 \times 10^{-3}$.

Implement the `reciprocal` method of `Decimal` using Newton's method. You should use the Newton-derived iterative calculation $x_{i+1} = x_i(2 - yx_i)$. As in the previous part, you will need (1) a way of making an initial guess that is close to the right answer and (2) a way of determining when to stop applying Newton iterations.

(d) [10 points] Finally, write a Python program that uses Newton's method as in Part (a), but this time working with `Decimal` numbers. Your routine should take in a parameter indicating how many digits of precision to use.

Your routine should be able to quickly compute cube roots to hundreds or even thousands of digits of precision.