CSE 101 Winter 2016
Programming Assignment (PA) One

Due: Friday January 22, 11:59 PM

This is the first of four programming assignments for CSE 101. These assignments are designed to complement the homeworks and lecture, and to give you working knowledge of how to concretely apply the algorithms and techniques covered in lecture. Our goal is for you to really understand the reasoning and logic behind these algorithm implementations, and to empower you with more tools to add to your repertoire. We don’t want you to worry about “finicky” parts of the programs as you might have seen in previous programming-focused CSE classes. To that end, we have worked hard to provide testers and clearly defined input / output so that you only have to complete the heart of what is necessary -- the algorithm. Given this, it is mandatory that you start the PA using the starter files from the class GitHub.

Some Notes:
-- PA’s are developed on ieng6, and will be graded on ieng6. This means you should also be working on ieng6. :)
-- You all have demonstrated proficient C++ skills through CSE 100, so we will not provide C++ implementation / C++ debugging help outside of the starter code and Friday (Weeks 2, 4, 6, 8) meetings.
-- Please be very careful to cite all sources of help used (Stack Overflow, a friend, etc.). If in doubt, add the citation!
-- Test cases will be graded on an all or nothing basis, so if for a particular question you receive 2/10 passing test cases, then your score is 20% for that question. DO NOT hand in code with debugging output or any other print statements, as your output will differ and you will fail the test case. No exceptions to the grading will be made unless there was an error on our side.
-- You will not be graded on style, but please keep it reasonable as we do read your code.
-- This PA has a total of FIVE questions. Each question has an assigned weight (see below).
The first three questions implement some of the algorithms discussed in lecture so that you are comfortable with their usage and with applying them to future problems. The testers are responsible for input/output, so you should not be printing anywhere in your code in the final version that you turn in. Note that all your algorithm implementations in Part One will deal with directed graphs.

**Question 1: Depth-first Search [10 Points] :: DFS.cpp/hpp**

- **Input:** directed graph, and a particular vertex ID
- **Output:** set of all reachable vertices with DFS starting at ID

Starting things off, you will implement a depth-first search algorithm using the provided headers for Graph and Vertex found in Graph.hpp. The required method signature can be found in DFS.hpp, and your implementation will go in DFS.cpp. For this question, the vertex ID’s will be represented with integers, but you must follow the template class and use a generic T so that you can re-use DFS for later problems (and any other applications you may want to pursue outside of the class). We recommend an iterative DFS solution so that your method is less likely to be memory constrained on larger graphs.

**To run the tester:**
make TestDFS
./TestDFS.out input_file

**Format for file input:**
1 The first line is always the ID of the starting vertex
1 2 Each subsequent line defines a directed graph edge where
2 3 the order is source vertex -> destination vertex.
2 4
3 4
4 5
6 5

**Sample Output:**
./TestDFS.out input_file
Result of DFS: [ 1 2 3 4 5 ]

Again, you do not have to worry about the input / output as that will be handled by the tester. Confirm that you have a working DFS implementation by making your own test cases in the format described above.
Question 2: Topological Order [20 Points] :: TopOrder.cpp/hpp

Input: directed graph
Output: list of vertex ID’s in topological order
(empty list if such an ordering does not exist)

A topological ordering of a directed acyclic graph (DAG) represents a labeling of the DAG’s vertices, beginning with a source vertex. Each vertex may be labeled only after all of its predecessors have been labeled. [As mentioned in class, you can think of a topological ordering as a feasible order in which, say, tasks can be performed, given that there are precedence constraints between tasks. If there is a directed cycle in the graph, then no topological ordering exists: it is impossible to find a vertex that can receive the lowest label among the vertices in the cycle.]

For example, let’s take the lower div CSE class requirements:

```
(12)------->(30)--------+
  ^     ^     |   
  |     |     |   
 +----+ |     |   
  |     |     v
(11)----->(15L)--------+  ---->(100)
         |     
         |     
         v
(20)-------->(21)------+
```

A valid topological ordering is: [11, 12, 15L, 30, 20, 21, 100].
Another equally valid ordering: [20, 11, 15L, 12, 21, 30, 100].

You’ll notice that as long as 11 is before 12, 20 before 21, 30 and 21 before 100, etc. then the topological ordering is valid. Your TopOrder can output any ordering as long as it is topologically valid.

The graph structure for this question also uses the graph definition from Graph.hpp, and relies on its tester for input/output. The template variable used in the tester char, and each vertex is represented by a single character ranging from a-z, A-Z, or 0-9 to keep the different permutations reasonable.
To run the tester:
   make TestTopOrder
   ./TestTopOrder.out testfile

Format for file input:
   H E    Each line represents a directed edge starting from the
   G E    first vertex and ending at the second vertex
   E D
   E F
   D C
   C B
   B A
   F A

Sample Output:
   ./TestTopOrder.out testfile
   Result of Topological Ordering: [ G H E F D C B A ]

Or if the input graph is not a directed acyclic graph:
   ./TestTopOrder.out testfile
   Result of Topological Ordering: [ ]

Again, note that any ordering of the vertices is valid as long as it
represents a correct topological ordering.
Question 3: Strongly Connected Components [20 Points] :: SCC.cpp/hpp

Input: directed graph
Output: list of sets; each set contains the vertices of an SCC

In this question, you will be implementing SCC-finding in a directed graph. As there are several excellent algorithms for finding SCCs, you may choose to implement any algorithm as long as the algorithm has been proven to run in linear time. You are encouraged to create your own helper functions and use STL data structures as suitable.

We have included a tester through which you can run tests in a similar manner to that of the DFS tester, so make sure to use this to ensure correct, well-formatted output.

To run the tester:
make TestSCC
./TestSCC.out input_file

Format for file input:
1 2 Each line defines a directed graph edge where 2 3 the order is source vertex -> destination vertex.
2 5
2 6
3 4
3 7
4 3
4 8
5 1
5 6
6 7
7 6
8 7
8 4

Sample Output:
./TestSCC.out input_file
SCC groupings:
6 7
3 4 8
1 2 5
The next two questions will have you apply various algorithms and techniques to answer “deeper” problems. A solid implementation of DFS will definitely help for Question 4 below, so it is recommended that you complete the questions in order.

**Question 4: Worm [25 Points] :: Worm.cpp/hpp**

Let’s play a game! Much like the classic arcade game Snake, our game is played on a low-resolution n x n grid with a worm moving from position to position on the grid and pieces of food occupying single tiles on the grid. In our case, our worm occupies two horizontally or vertically adjacent (not diagonally adjacent) tiles on the grid.

Unlike a snake, however, the worm has no interest in eating the food and cannot move into a tile occupied by food. The worm simply wants to move from its start position to a specified end point. To make things harder, the worm cannot move in a straight line and can only move by turning. To illustrate, say we have a 4 x 4 board with tiles labeled as follows:

```
  12    13    14    15
  ---------------
  8     9    10    11
  ---------------
  4     5     6     7
  ---------------
  0     1     2     3
  ---------------
```

If a worm begins on tiles 5-6, it has exactly four potential moves:
- turn to 6-10
- turn to 6-2
- turn to 5-9
- turn to 5-1
(Notice that after making a move, the worm still occupies one of the two tiles that it previously occupied.)

Given this beginning position of 5-6, note that only the turn to position 5-9 will be blocked if a food obstacle exists on tile 9. If a food obstacle exists on tile 2, the worm cannot turn to 6-2, but it can still turn to 5-1.
Other notes:
- The worm cannot wriggle forward or backwards to 4-5 or 6-7.
- The worm cannot roll sideways to 9-10 or 1-2.
- The worm cannot move off the grid.

You will be given inputs:
- int n : the size of the board
- int s1 : id of tile of one end of worm’s initial position
- int s2 : id of tile of other end of worm’s initial position
- int d1 : id of tile of one end of worm’s end position
- int d2 : id of tile of other end of worm’s end position
- list<int> o : list of tile ids containing food obstacles

Your mission, should you choose to accept it, involves determining (in linear time with respect to grid area) whether the worm can reach the specified target (d1, d2) given start position (s1, s2), obstacles [o1, o2, ...], and grid dimensions n x n. Note that all tiles of any n x n grid will have labels 0, 1, ..., (n^2 - 1) as in the example above -- that is, left to right in each row, starting with the bottom row of tiles. Note also that a reasonable approach to this problem might be to set up a graph of worm positions, eliminate vertices that are “illegal” due to food obstacles, and then apply code from previous problems.

You may:
- Use STL data structures as appropriate. Skeleton code has been provided for your benefit. Choose your data structures wisely!

To run the tester:
make TestWorm
./TestWorm.out input_file

Format for file input:
4     // n
0 1   // s1 s2
11 15 // d1 d2
4 5   // [o1, o2, ...]

Sample Output:
./TestWorm.out input_file
Result of Worm: 0     // 0 for unreachable, 1 for reachable
Question 5: Analysis of Search in Rotated Sorted Arrays [25 points]

A rotated sorted array is a sorted array which has had a single rotation applied to it.

For example: [1, 2, 3, 4, 5, 6] -> [3, 4, 5, 6, 1, 2]

For this problem, you will design an algorithm to find an element in a rotated sorted array within O(log n) time.

You will also compare your algorithm’s runtime to that of naive search through the array in O(n) time. You must give a detailed description of your algorithm (hint, use Divide-and-Conquer), and quantitatively (experimentally) analyze its merits over the naive approach (comparing average runtime over varying array sizes, multiple trials, etc.). Please put this analysis along with your supporting graphs and tables in a PDF and include it along with your .cpp implementation in the same folder as your other files. Again, you have to back up your claims with data that you collect (on ieng6) for the runtimes of naive vs. DQ for varying array sizes (10^3, 10^4, 10^5, 10^6, 10^9, etc.).

You will notice that we are stressing your analysis and writeup of the algorithm more than your implementation for this question. (The starter code already implements timer functionality and generates the rotated sorted array of size n, the element to search for, etc. for you!) Please keep in mind that just an implementation without analysis will not earn very many points for this question.

To use the provided timing starter code:

```
make RotatedArray
./RotatedArray.out
```

To change parameters for size of array, edit the definition at the top of RotatedArray.cpp.