UNIVERSITY OF MINNESOTA
DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING
4041: ALGORITHMS AND DATA STRUCTURES
FALL 2017

PROGRAMMING ASSIGNMENT 2, PART 1 :
Assigned: 11/11/17 Due: 12/03/17 at 11:55pm (submit via moodle)

Make sure you apply your code to different inputs to test the results. Copying code from others or the internet constitutes cheating and the University policies will be followed. For each problem hand in these three things (after putting them in a single .zip file **without any folders**):

1. The code you create.
2. A "readme.txt" file explaining any how to run your code and any assumptions that you made. If you were not able to complete a problem, describe your approach here.
3. A "run.sh" file that will both compile (if necessary) and run your program on the input text file. **You must ensure your code works with the run.sh on a cselabs machine**.

We will use files to both send input into your program and to read the output of your program. We should be able to run your program by saying "./run.sh input.txt" for some input text file. Your run.sh has to open the string after it as the input file (i.e. not always "input.txt"). In each part, your code should create a text file named "output.txt" with your solution. When printing multiple things, separate them by spaces (except after the last thing). To get your run.sh to work, you might need to type: chmod 700 run.sh

Please ensure your code is easily readable and well structured. If the code is obfuscated, has poorly named variables/functions, not well documented, etc., then points will be taken off. You may choose to code in any of these languages: C/C++, Java, or Python. For each problem the point breakdown will be roughly: 65% correct output, 15% readme.txt and .sh file sufficient, and 20% coding style. Your code must work on a caselabs machine. We will test it on the machine: csel-kh1260-13.cselabs.umn.edu

Problem 1. (20 points)
Make a string match program using a finite automata. However, you need to modify it so it finds the patter either forwards or backwards. The first word in the input file is the pattern (on its own line). The second word will be the string you are trying to match. Output all indexes where a match starts happening.

You may assume we will only put the normal 26 English letters in lower case for both the pattern and string. It is possible for a match to happen twice (if the pattern is symmetric). You should output the index of this match twice then. Assume indexes start from 0.

Sample input file (all characters on a single line with no spaces between):
```
abcc
abccbabcc
```

Corresponding sample output.txt file (spaces separating indexes (no space after final index)):
```
0 2 5
```

Problem 2. (20 points)
In this problem the input file will have classes listed with their prerequisites. Your output.txt should list a valid order for taking all of these classes while meeting all the prerequisites.

You may assume there are no circular prerequisites (i.e. class A requires class B. Class B requires class C. Class C requires class A). You may assume all classes that are prerequisites of another are listed. The courses can be any string (not necessarily numbers)

Sample input file (course identifier before ":". Prerequisites after ":" with spaces between):

```
1001:
1133:
2011:
1933:1133
4041:1933 2011
```

Sample output.txt (valid class ordering with spaces between (no space after final class)):

```
1133 2011 1933 4041 1001
```

Problem 3. (20 points)
Implement Dijkstra's algorithm to find the shortest path to every other vertex. The input text file will contain the graph represented as an adjacency matrix. Values of 2 million will represent "infinity" edge weight (when there is no edge between the verticies). You can assume all shortest paths will be less than 2 million.

The first line in the input file is the source vertex, one space, then the destination vertex (with the first row of the adjacency matrix being vertex "0", the second row being vertex "1", and so on). Afterwards is the adjacency matrix (which will be square). Your output.txt should contain the shortest path length from this source to destination vertex followed by a ":". After the ":" should be the actual list of vertexes you should travel through for this shortest path.

Sample input file (fire line always source and destination vertex, afterwards is adjacency matrix):

```
0 1
0 2000000 4
2 0 7
2000000 3 0
```

Sample output.txt (7 is the shortest path length (before ":"). 0 2 1 is the actual path (after ":")):

```
7: 0 2 1
```