CX 4010 / CSE 6010
Assignment 3
Divisor Graph
Additional requirements for graduate students only

Initial Submission Due Date: 11:59pm on Thursday, October 6
Peer Reviews Due Date: 11:59pm on Thursday, October 13
Final Submission Due Date: 11:59pm on Thursday, October 20
Submit codes as a single zipfile as described herein to Canvas
Submit peer reviews through Canvas
48-hour grace period applies to all deadlines

In this assignment, you will write a program to construct a graph that represents a divisor relation: vertices of the graph are sequential positive integers from 1 up to a specified maximum, and the graph has a directed edge from vertex i to a different vertex j whenever i is a divisor of j. You will find the length of the maximum-length path between any two vertices in the graph and also output a maximum-length path for each integer that can be reached by a path with maximum length.

When the edge weights are all 1, each maximum-length path will be a series of positive integers whose ratios form the prime factors of the final number in the path. For example, suppose we consider positive integers up to 10. The maximum-length path will proceed from 1 to 2 to 4 to 8 and thus will have length 3. The ratios $2/1$, $4/2$, and $8/4$ are the prime factors of 8: 2 * 2 * 2 = 8. Note that there may be more than one integer that can be reached by a path of maximum length. If we consider positive integers up to 7, there would be two integers that could be reached by paths of the maximum length, which is 2: both 4 and 6 can be reached with length 2. You would be expected in this case to output a maximum-length path to both values: a maximum-length path from 1 to 4 and a maximum-length path from 1 to 6. Note that the latter is not unique; you could find a path from 1 to 2 to 6 or from 1 to 3 to 6. You will be required to output only one of these and either would be acceptable.

Requirements:

- Use an adjacency list representation for the graph. You may assume that the graph will include no more than 1000 entries (and thus you may use a matrix sized accordingly). Each list should use a **singly-linked-list** data structure with a single pointer to one end of the list.
- Use a command-line argument for the maximum positive number N to include in the graph (i.e., there will be vertices from 1 to N). You are not required to do any validation/error-checking of the value, but if no value is provided, use the maximum allowable value of 1000.
- Construct the graph by adding an edge from i to a different vertex j in the graph whenever i is a divisor of j. We will consider i to be a divisor of j whenever the quotient j/i has 0 remainder for $i > 2$, and whenever $i = 1$—that is, there will be an edge from vertex 1 to every other vertex in the graph. Do not include loops in the graph (no edges from a vertex to itself). All consecutive positive integers should be vertices in the graph up to the maximum value specified by the user at run-time on the command line or 1000, as indicated above. The edge weights should be set to 1, but you should develop your graph representation as if each edge could have a different weight (so you will need to store the weights).
- Use a slight modification of Dijkstra's algorithm to find the maximum length of a path between any pair of integers and to track predecessors for each path. Note that this

approach will not work in general, but for this particular graph, you can find a maximum-length path by interchanging < and > in Dijkstra's algorithm. In addition, you should consider only paths starting from 1; because 1 is considered a divisor of every number, any supposedly maximum-length path that did not start include 1 could be lengthened by adding 1. You may use any method you like for tracking the members of the sets we described as S and Q.

- Use a slight modification of the Bellman-Ford algorithm as an alternate maximum-length path algorithm. Use a second command-line argument for the algorithm choice. Specify as "b" for modified Bellman-Ford; otherwise, use modified Dijkstra's algorithm. It is required that the first command-line argument be used if the user wishes to use the modified Bellman-Ford algorithm. In the algorithm, track paths of maximum length by checking for longer paths, rather than shorter paths, with the Relax function. For efficiency, process the edges according to the order in which they are listed in the adjacency lists, beginning with the adjacency list corresponding to 1.
- Output the maximum path length and a path to each integer that can be reached with maximum length, as described earlier.

Implementation Specifications:

- Define a typedef named `Vertex`. You should expect that the adjacency list will be maintained as an array (of size 1000) of type `Vertex*` (in other words, each element in the array will be a pointer to (which is the same as the memory address of) something of type `Vertex`. You will define the struct members as appropriate.
- Define functions with the following prototypes.
  1. `void initialize(Vertex** vertexarray);`
     Here and throughout, `vertexarray` is an array of pointers to type `Vertex`, as described above. Allocate memory for the items pointed to by the array of pointers.
  2. `void buildGraph(Vertex** vertexarray, int numvertices);`
     Use the divisor rule described above to determine where edges should be added considering each positive integer from 1 to `numvertices`. For each edge, allocate memory accordingly and store the relevant information, including the edge weight.
  3. `void findPaths(Vertex** vertexarray, int numvertices, char bf);`
     Use Dijkstra's algorithm or Bellman-Ford, according to the setting of `bf`, to find the length of the longest path from the vertex corresponding to 1 to any other vertex in the graph, then output the maximum path length and a path to each integer that can be reached with maximum length.
  4. `void finalize(Vertex** vertexarray);`
     Free memory allocated previously for the items pointed to by the array of pointers and associated with any edges added when building the graph.
- Thus, your main function should consist of only the following items:
  - Declaring and setting `numvertices` and `bf` according to the command-line arguments passed or the defaults as described earlier.
  - Declaration of `vertexarray`.
  - Calls to the four functions listed above.

You may define any additional functions and/or typedefs as needed.

Keep in mind that your arrays will be indexed beginning with 0 but your graph will include only positive integers. There are several ways to handle this situation appropriately.

You should define all these items in files you write named **divisorgraph.h** and **divisorgraph.c**. Use these auxiliary files to implement your main function in a file you write named **main.c**.

Because your program will be written as described, you should expect that your divisorgraph.h and divisorgraph.c files could be used with a main.c file written by someone else.

Your code (final submission) should successfully compile (using gcc) and run on the COC-ICE cluster. This does not mean we necessarily will try to run your code on the cluster, but if we have any difficulties compiling and running it, we will move it to that environment for further testing. If you are compiling and running your codes under linux, most likely you will not need to make any changes as a result of this requirement.

You should submit to Canvas a single zipfile that is named according to your Georgia Tech login—the part that precedes @gatech.edu in your GT email address. To receive full credit, your code must be well structured and documented so that it is easy to understand. Be sure to include comments that explain your code statements and structure.

The zipfile should include the following files:

(1) your code (all .c and .h files). If you are using linux or Mac OS, you may use a makefile to compile and run your program, and you should include it if so.

(2) a README text file (not formatted in a word processor, for example) that includes the compiler and operating system you used for compiling and running your code along with instructions on how to compile and run your program.

(3) a series of 3 slides composed in PowerPoint or similar software, saved either in PowerPoint or as a PDF and named slides.pptx or slides.pdf, structured as follows:

- Slide 1: your name and a brief explanation of how you developed/structured your program. This should not be a recitation of material included in this assignment document but should focus on the main structural and functional elements of your program (e.g., how you defined any structs, how you handled the sets S and Q in Dijkstra's algorithm, etc.). You are limited to one slide.
- Slides 2: a justification of why you believe your program is performing correctly. Include sample output produced by your program. Focus on what you think is most important.
- Slide 3: a brief exposition of what you found (or did not find) useful about the peer review process.

Initial submission

Your initial submission does not need to include the slides. It will not be graded for correctness or even tested (including for compatibility with gcc on the COC-ICE cluster) but rather will be graded based on the appearance of a good-faith effort to complete the majority of the assignment. However, you should be aware that **your peers will review your initial submission**. Thus, it is recommended you complete as much of the main functionality of the assignment as possible, even if

you are continuing to adjust certain aspects (e.g., command-line arguments, Bellman-Ford, separating your code into the functions as described, etc.).

Peer review assignment

In this assignment, you will have two weeks between the initial and final submission to allow you the opportunity to review others' submissions and provide feedback. Although the feedback you receive from your peers should be useful, we also expect that having the opportunity to view others' code will be a valuable experience for you as well. You will submit your final submission after the peer reviews have been completed so that you can benefit from the process. However, while you may find yourself inspired by what others have done, you should **be careful not to plagiarize from others' code**. **You should write and submit your assigned peer reviews while taking a break from looking at your code and you should not look at others' code again afterward.**

Peer review assignments will be made in Canvas after all initial assignments have been submitted. (peer review assignments should be made by Monday, October 10). **If you do not submit an initial assignment on time, you may not be able to complete the peer review assignment either.**

You will be assigned **three peer reviews**. Your peer review does not have to be long but it should address the following 5 topics. **Use the Comment box to submit your peer review.**

- Program execution: does the program compile and execute without error, and is the output clear?
- Program structure and implementation: is the structure of the code clear and easy to follow and are the implementation choices good?
- Program style and documentation: are the comments in the program useful and at the appropriate level, is the program easy to follow visually through indenting and blank lines, and does the program avoid copying the same code in multiple places (redundancy)?
- Overall strengths of the code.
- Any constructive suggestions for improvement.

Because of Canvas limitations, grades for the peer reviews you write will be reported in a separate peer review assignment. (You will not need to do anything extra for the official Canvas assignment, but that is how the grades will appear.) Peer reviews you receive will not affect your grade in any way. For information on how to use Peer Reviews in Canvas, including where to submit your peer reviews and where to find peer reviews of your submission, please see the following videos (or other similar ones you may find).

https://www.youtube.com/watch?v=cKyvKKq0mtc
https://community.canvaslms.com/t5/Video-Guide/Feedback-Overview-Students/ta-p/383514

Some hints:

- Start early!
- There are two main parts to this assignment: implementing a graph as an adjacency list and implementing Dijkstra's algorithm. Identify a logical flow for trying to progressively incorporate this main functionality piece by piece, then add the remaining functionality. Grads should implement either Dijkstra's or Bellman-Ford fully before implementing the other.