

CS 211: Computer Architecture, Fall 2022

Programming Assignment 2: Graph Algorithms + Data Representation (150 points)

Instructor: Prof. Santosh Nagarakatte

Due: October 17, 2022 at 5pm Eastern Time.

Introduction

In this assignment, you will get more experience with C programming by implementing several classical graph algorithms. You will also solidify your understanding of data representation with this assignment.

In the first set of 6 programs, you will get more experience with allocating graph data structures and processing them. You will start by representing a graph data structure in C. Then, you will learn how to represent undirected and weighted directed graphs. Subsequently, you will implement two simple graph traversal algorithms, breadth-first (BFS) search, and depth-first (DFS) search. Using the implementation of DFS, you will write a program to perform single-source shortest path computation in directed acyclic graphs (DAGs). Subsequently, you will implement Dijkstra's shortest path algorithm that is not limited to DAGs and can be used in all directed graphs with no negative edge weights.

In the last 4 programs, you will improve your understanding of data representation by writing programs to represent two's complement values and floating point values.

Note that your program must follow the input-output guidelines listed in each section **exactly**, with no additional or missing output.

No cheating or copying will be tolerated in this class. Your assignments will be automatically checked with plagiarism detection tools that are pretty powerful. Hence, you should not look at your friend's code. See CS department's academic integrity policy at:
<http://nbacademicintegrity.rutgers.edu/>

First: Undirected Graph Representation (10 Points)

A graph is a way of representing relations between a set of objects, called vertices. Each pairwise connection between vertices is called an edge. In computer science, graphs have many applications in modeling maps, computer networks, neural networks, circuits, etc.

For example, consider the graph in Figure 1(a) used to model the roads between different intersections in a city. In this example, a vertex represents an intersection. An edge models a road between a pair of intersections. In graph terminology, a pair of vertices are adjacent if an edge connects them. For example, in Figure 1(a) the vertex pair (A,B) are adjacent. Further, the degree of a vertex V is defined as the number of vertices adjacent to it. For example, in Figure 1(a), vertex

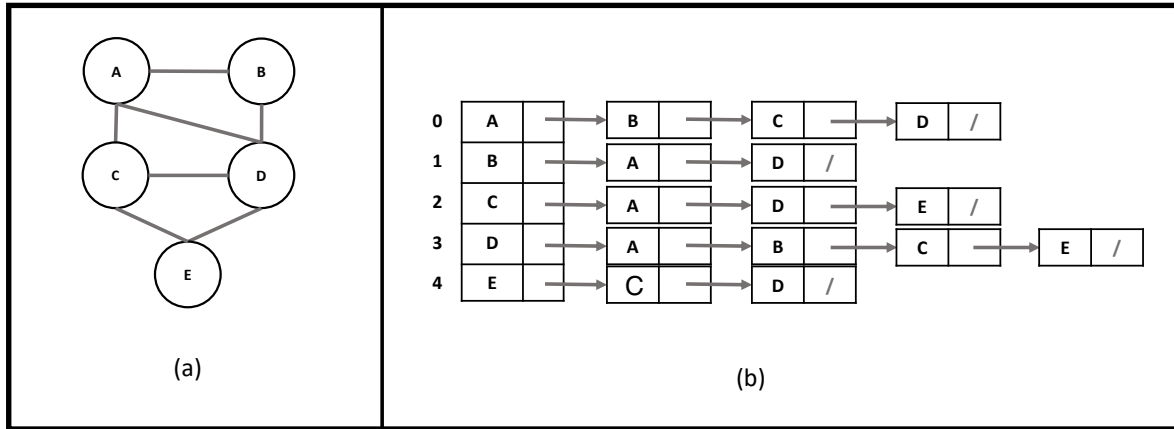


Figure 1: (a) An undirected graph G (b) Adjacency list representation of graph G.

E has a degree of 2 since its only adjacent to vertex C and D.

There exist several standard ways to represent a graph on a computer. For this programming assignment, we are going to use the adjacency list representation. In the adjacency list representation, each vertex stores a linked list of its adjacent vertices. Figure 1(b) illustrates the adjacency list representation of the undirected graph from Figure 1(a).

In this part, you write a program that will read an undirected graph from a file. Store it in an adjacency list representation and then answer simple graph queries.

Input-Output format: Your program will take two file names as its command-line input. The first file includes the undirected graph. Your program reads the contents of this file and constructs the graph data structure. The first line in this file provides the number of vertices in the graph. Next, each line contains the name of each vertex in the graph. Afterwards, each following line includes information about an edge in the graph. Each edge is described by the name of its pair of vertices, separated by a space.

The second file includes queries on the constructed graph. Each line contains a separate query that starts with the query type and a vertex, separated by a space. There are two query types. Degree queries start with the letter 'd', followed by a space and the vertex's name, which is a string. Upon processing a degree query, your program must print out the queried vertex's degree, followed by a newline character. Adjacency queries start with the letter 'a', followed by a space and the vertex's name. Upon processing an adjacency query, your program must print out the vertices adjacent to the queried vertex, each vertex separated by a space and finally, a newline character. When you print the results of the adjacency query, the results have to be sorted lexicographically.

Example Execution:

Let's assume we have the following graph and query file:

```
graph.txt
5
A
B
C
D
```

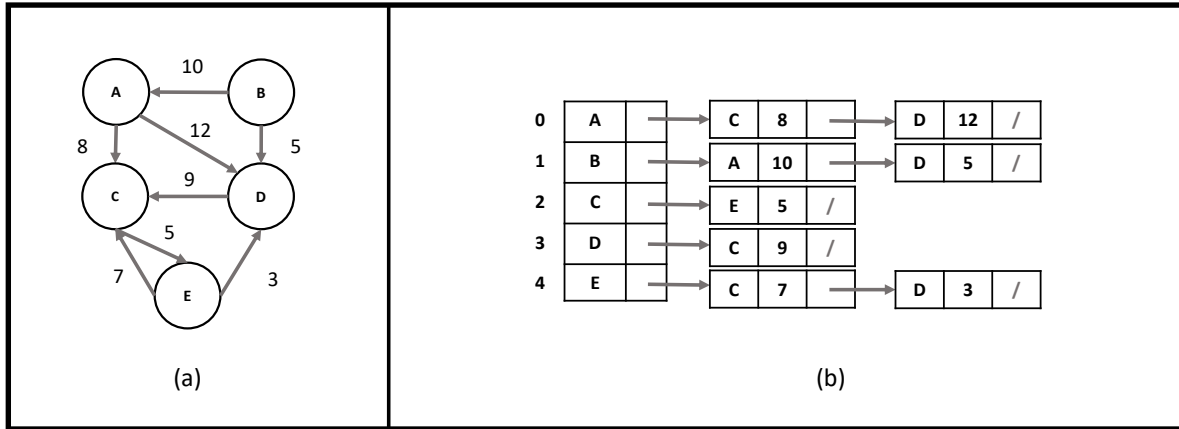


Figure 2: (a) A weighted directed graph G (b) Adjacency list representation of graph G.

E
A B
A C
A D
B D
C D
C E
D E

query.txt:
d E
a C
d A
a A

Then the result will be:

```

$./first graph.txt query.txt
2
A D E
3
B C D

```

We will not give you improperly formatted files. You can assume all your input files will be in proper format, as stated above.

Note: A vertex can be any valid identifier (*i.e.*, a string consisting of characters [a-z] and [0-9] that is terminated by a space).

Second: Weighted Directed Graph Representation (10 Points)

Consider the example graph in Figure 1(a). While this graph models the existence of a road between a pair of intersections, it doesn't capture the distance between them. Further, some roads

may not be bidirectional, which cannot be modeled using an undirected graph. In this part, we write a program to store and query a weighted directed graph. In a weighted graph, we attribute a numeric value to each edge. Figure 2(a) shows a weighted directed graph. As shown in Figure 2(a), note that in a directed graph, the inclusion of edge (A,C) does not imply the existence of edge (C,A). Figure 2(b) visualizes the adjacency list representation of the weighted directed graph in Figure 2(a).

In this part, you write a program that will read a weighted directed graph from a file. Store it in an adjacency list representation and then answer simple graph queries.

Input-Output format: Your program will take two file names as its command-line input. The first file includes the weighted directed graph. Your program reads the contents of this file and constructs the graph data structure. The first line in this file provides the number of vertices in the graph. Next, each line contains the name of each vertex in the graph. Afterwards, each following line includes information about a weighted directed edge in the graph. Each weighted edge is described by the name of its pair of vertices, followed by the edge weight, separated by a space. For example, B A 10 defines a directed edge from vertex B to vertex A with an edge weight of 10.

The second file includes queries on the constructed graph. Each line contains a separate query that starts with the query type and a vertex, separated by a space. There are three query types. Out-degree queries start with the letter 'o', followed by a space and the vertex's name. Upon processing an out-degree query, your program must print out the queried vertex's out-degree ¹, followed by a newline character. In degree queries start with the letter 'i', followed by a space and the vertex's name. Upon processing an in degree query, your program must print out the queried vertex's in degree ², followed by a newline character. Adjacency queries start with the letter 'a', followed by a space and the vertex's name. Upon processing an adjacency query, your program must print out the vertices adjacent to the queried vertex, each vertex separated by a space and finally, a newline character. When you print the results of the adjacency query, the results have to be sorted lexicographically.

Example Execution:

Let's assume we have the following graph and query file:

```
graph.txt
5
A
B
C
D
E
B A 10
A C 8
A D 12
B D 5
C E 5
```

¹The number of edges directed out of a vertex in a directed graph.

²The number of edges directed towards a vertex in a directed graph.

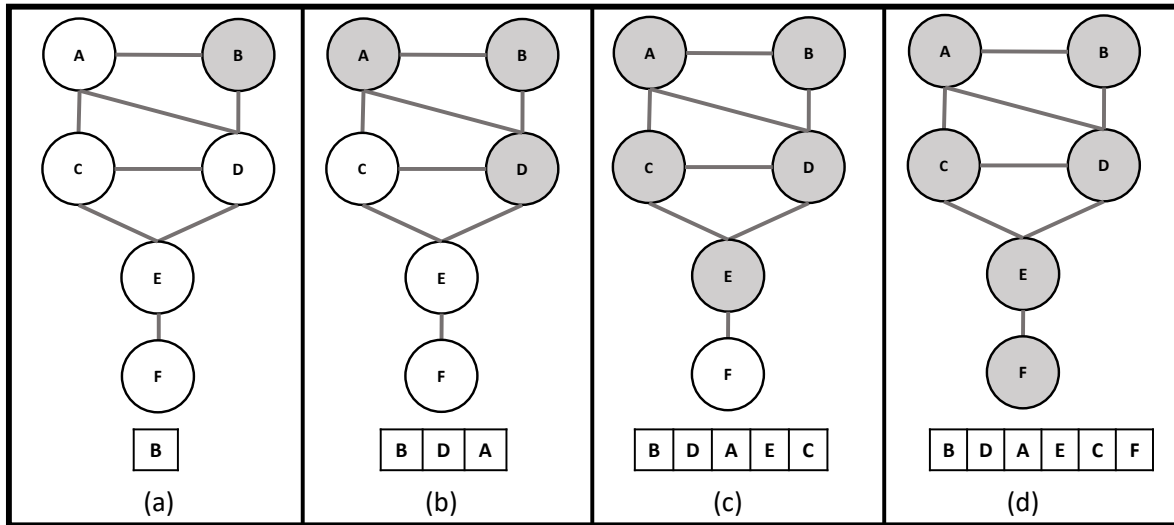


Figure 3: This figure illustrates an undirected graph and the BFS traversal steps of the graph starting from source vertex B.

```
D C 9
E C 7
E D 3
```

query.txt:

```
o E
a C
i E
a A
a E
```

Then the result will be:

```
./second graph.txt query.txt
2
E
1
C D
C D
```

We will not give you improperly formatted files. You can assume all your input files will be in proper format, as stated above.

Third: Breadth-first Search (BFS) (15 points)

In this part, you will implement the breadth-first search (BFS) graph traversal algorithm. For a given input graph $G=(V,E)$ and a source vertex s , BFS starts exploring the edges of G until it discovers all vertices reachable from the source vertex. During a BFS search, we start by visiting the adjacent

vertices to the source vertex, processing them, and subsequently exploring vertices in order of edge distance (*i.e.*, the smallest number of edges) from it. Figure 3 shows an example graph and its BFS traversal starting from source vertex B. Note that the vertices are processed in order of their distance from the source³.

You will write a program that will read an undirected graph from a file using your implementation from part 1 and perform BFS starting from different source vertices.

Input-Output format: Your program will take two file names as its command-line input. The first file includes the undirected graph. This file is similar to the graph file from part 1. Your program reads the contents of this file and constructs the graph data structure. The first line in this file provides the number of vertices in the graph. Next, each line contains the name of each vertex in the graph. Afterwards, each following line includes information about an edge in the graph. Each edge is described by the name of its pair of vertices, separated by a space.

The second file includes BFS queries on the constructed graph. Each line contains a different BFS query specifying a source vertex for the BFS. Your program must read the source vertex, perform a BFS traversal on the constructed graph using the chosen source vertex, and print out the graph vertices in order of BFS processing. Each vertex is separated by a space and finally, a newline character.

Example Execution:

Let's assume we have the following graph and query file:

graph.txt

```
6
A
B
C
D
E
F
A B
A C
A D
B D
C D
C E
D E
E F
```

query.txt:

```
B
E
```

Then the result will be:

³For more information on BFS and example pseudocode, see https://en.wikipedia.org/wiki/Breadth-first_search

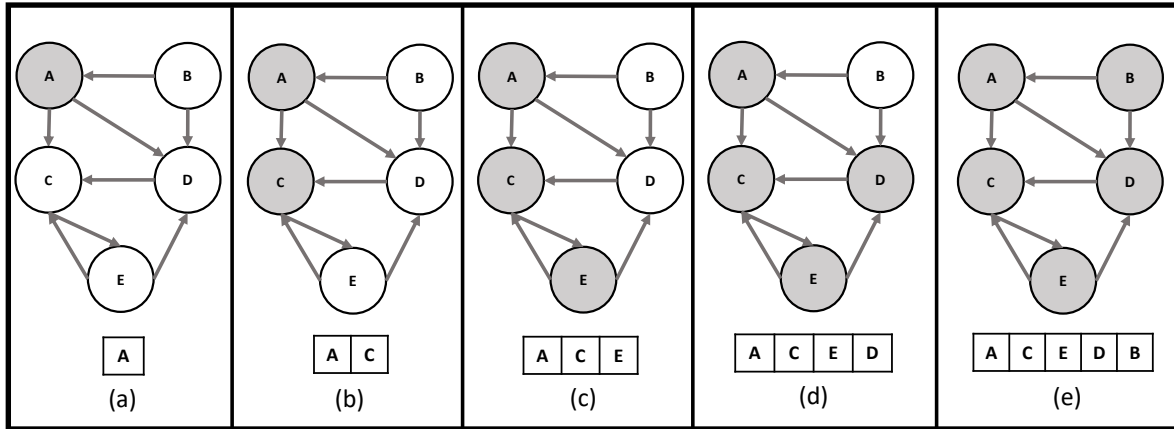


Figure 4: This figure illustrates a directed graph and its DFS traversal steps starting from vertex A.

```

$./third graph.txt query.txt
B A D C E F
E C D F A B

```

We will not give you improperly formatted files. You can assume all your input files will be in proper format, as stated above.

You have to traverse the immediate children of a node in a lexicographic order. For example, if A node has three edges to nodes B, C, and D, then the BFS will process B first, then C, and finally D. This requirement ensures that every graph has a single unique BFS traversal for a given source vertex. You can easily satisfy this requirement by maintaining the adjacency list in a lexicographically sorted order.

Fourth: Depth-first Search (DFS) (15 points)

In this part, you will implement the depth-first search (DFS) graph traversal algorithm. For a given input graph $G=(V,E)$, DFS visits an unvisited vertex v . At each step in DFS, we choose an unvisited vertex adjacent to the most recently discovered vertex. We continue this process until all vertices reachable from v are discovered. If any undiscovered vertices remain, we choose one of them and repeat the above process until all vertices are discovered. For example, Figure 4 illustrates a DFS traversal of the example graph in Figure 2(a) ⁴.

In this part, you write a program that will read a directed graph from a file using your implementation from part 2 and perform a DFS traversal, printing out the graph vertices in order of DFS vertex visit. When you are choosing a vertex to visit next among the adjacent children, you have to pick the vertex that is not visited yet and occurs first in the lexicographic order.

Input-Output format: Your program will take a file name as its command-line input. This file includes a directed graph, and it follows the same format from part 2. Your program reads the

⁴For more information on DFS and example pseudocode, see https://en.wikipedia.org/wiki/Depth-first_search

contents of this file and constructs the graph data structure. The first line in this file provides the number of vertices in the graph. Each following line includes information about a weighted directed edge in the graph. Each weighted edge is described by the name of its pair of vertices, followed by the edge weight, separated by a space. Your program must read and construct this graph, perform a DFS traversal, and print out the graph vertices in order of DFS visitation. Each vertex is separated by a space and, finally, a newline character.

Example Execution:

Let's assume we have the following graph input file:

```
graph.txt
5
A
B
C
D
E
B A 10
A C 8
A D 12
B D 5
C E 5
D C 9
E C 7
E D 3
```

Then the result will be:

```
$/fourth graph.txt
A C E D B
```

We will not give you improperly formatted files. You can assume all your input files will be in proper format, as stated above.

Hints and Suggestions

- In a DFS traversal, each vertex is processed at most once. A DFS traversal will visit all graph vertices even when the graph is disconnected. Make sure your program works correctly in such cases.
- You may have noticed that your program can safely ignore the graph edge weights in this part. However, this part's solution is going to be used in part 5, which requires reading graph weights. Hence reusing part 2's solution for weighted directed graphs is recommended in programming this part of the assignment.
- When visiting graph vertices, visit them based on lexicographical ordering.

Fifth: Single-source Shortest Path in a Directed Acyclic Graph (DAG) (20 points)

Given a weighted directed graph $G=(V,E)$ and the source vertex s , the single-source shortest path problem's goal is to identify the shortest path from the source vertex to all vertices in the graph. For example, finding the shortest path from our home to different adjacent cities can be modeled as a single-source shortest path problem with our home being the source vertex.

Depending on the input graphs type, a single-source shortest path problem can be solved using different algorithms that vary in asymptotic running time and complexity. For example, the BFS algorithm from part 3 is sufficient to solve the single-source shortest path problem for unweighted graphs. However, we need other algorithms to solve the single-source shortest path problem in weighted graphs⁵.

In this part, your task is to write a program to solve the single-source shortest path problem for a type of directed graph called directed acyclic graphs (DAG). A DAG is a directed graph with no cycles⁶.

The single-source shortest problem in DAGs can be solved by visiting the DAG's vertices in a topologically sorted order and updating the shortest path of the visited vertex's adjacent vertices. You must use the DFS traversal from part 4 to topologically sort the DAG.

Algorithm 1 shows the steps to compute the single source path for the graph G and source vertex **src**. The algorithm maintains a distance array that is initially set to infinity for all vertices except the source vertex. `distance[u]` contains the shortest path from the source vertex to vertex u at the end of the algorithm or infinity if no path between **src** and u exists.

A topological sorting of the DAG $G(V,E)$ is an ordering of its vertices, T such that for every directed (u,v) , vertex u appears before vertex v in its topological ordering. For example, Figure 5(a) shows an example DAG and its topological sort. Figure 5(b-f) illustrates the steps taken by the DFS inspired algorithm⁷ to compute the topological ordering of the DAG by using a stack. Lastly, Figure 5(g) shows updates to the distance array by using Algorithm 1 on the DAG from Figure 5(a).

Input-Output format: Your program will take two file names as its command-line input. This first file includes a DAG, and it follows the same format from parts 2 and 4. Your program reads the contents of this file and constructs the graph data structure. The first line in this file provides the number of vertices in the DAG. Each following line includes information about a weighted directed edge in the DAG. Each weighted edge is described by the name of its pair of vertices, followed by the edge weight, separated by a space. Your program must read and construct this DAG,

The second file includes single source shortest path queries on the constructed DAG. Each line contains a different single-source shortest path query by specifying a source vertex. Your program must read the source vertex, perform the single source shortest path algorithm using the provided source vertex, and print out each of vertex in the DAG in lexicographic ordering, followed by the length of the shortest path to that vertex, and a newline character. Note that an additional newline character follows the last vertex in DAG. Further, your program must detect if the input graph is not a DAG. In such cases, your program simply prints out CYCLE, followed by a newline character.

⁵For more information on graph shortest path problems, see https://en.wikipedia.org/wiki/Shortest_path_problem

⁶For more information on DAGs, see https://en.wikipedia.org/wiki/Directed_acyclic_graph

⁷see https://en.wikipedia.org/wiki/Topological_sorting for pseudocode

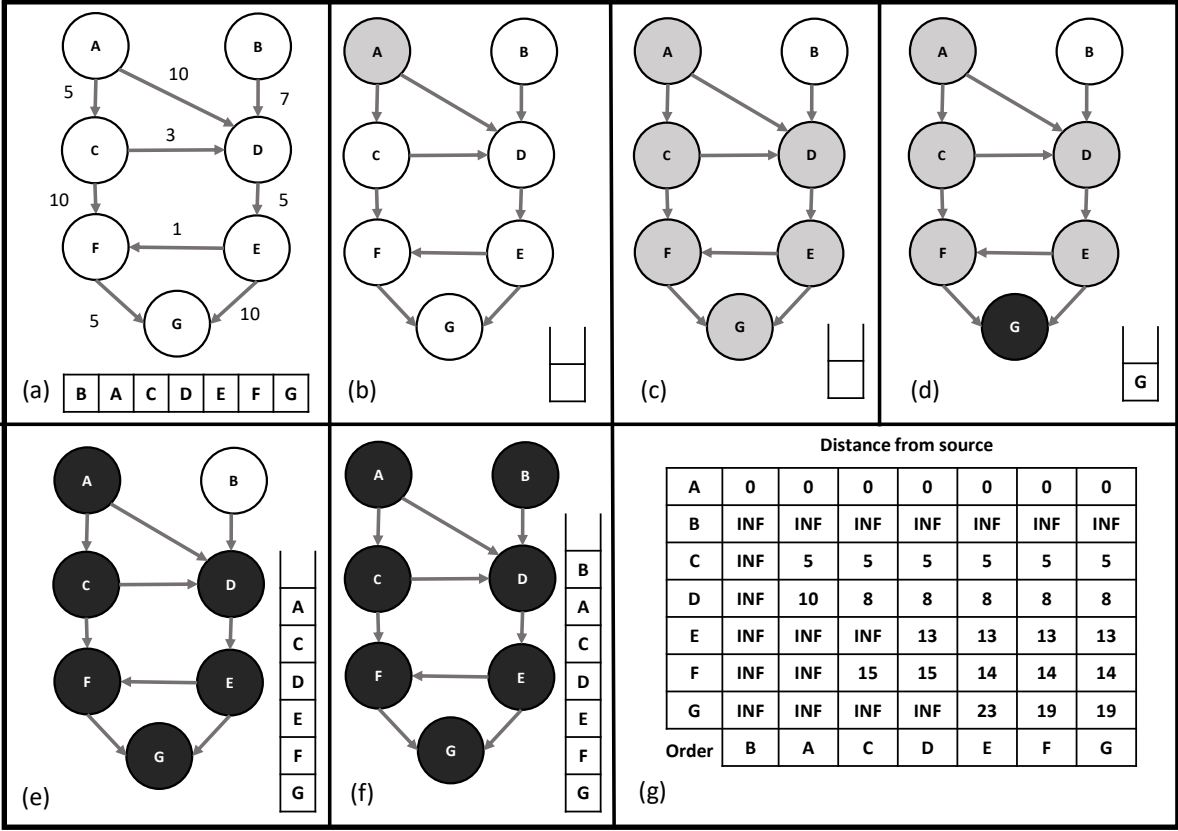


Figure 5: This figure illustrates a directed graph and the steps involved in identifying its topological sort

Algorithm 1: Single-source shortest path algorithm for DAGs

```
1 procedure DAG-SSP( $G, src$ )
2    $T \leftarrow TopologicalSort(G)$ 
3   foreach vertex  $v$  in Graph  $G$  do
4      $distance[v] \leftarrow inf$ 
5   end
6    $distance[src] \leftarrow 0$ 
7   foreach vertex  $u$  in topologically sorted order  $T$  do
8     foreach vertex  $v \in u.Adjacent$  do
9       if  $distance[v] > distance[u] + weight(u, v)$  then
10         $distance[v] \leftarrow distance[u] + weight(u, v)$ 
11      end
12   end
```

Example Execution:

Let's assume we have the following graph input file:

graph.txt

```
7
A
B
C
D
E
F
G
A D 10
A C 5
B D 7
C D 3
D E 5
E F 1
C F 10
E G 10
F G 5
```

query.txt

```
A
G
```

Then the result will be:

```
$/fifth graph.txt query.txt
```

```
A 0
B INF
C 5
```

```
D 8
E 13
F 14
G 19
```

```
A INF
B INF
C INF
D INF
E INF
F INF
G 0
```

For the scenario when the input graph is not a DAG:

```
not_dag.txt
2
CA
NJ
NJ CA 3000
CA NJ 3100
```

```
query.txt
CA
```

Then the result will be:

```
$/fifth not_dag.txt query.txt
CYCLE
```

We will not give you improperly formatted files. You can assume all your input files will be in proper format, as stated above.

Hints and Suggestions

- The shortest path from the source vertex to itself is 0.
- The shortest path from the source vertex to any unreachable vertex is infinity. In the output, we represent infinity with INF, as shown in the example execution.
- You can use MAX_INT from <limits.h> to represent infinity in your program. The input DAGs will not contain edges with weights larger than MAX_INT. Further, you may safely assume that the shortest paths do not overflow.
- Edge weights can be negative numbers.

Sixth: Dijkstra's Shortest Path Algorithm (20 points)

In this part, we implement Dijkstra's Algorithm. This algorithm solves the single-source shortest path problem for graphs with nonnegative edge weights. The key idea behind Dijkstra's algorithm

is to maintain a set of vertices whose final shortest path from the source vertex has been determined. This set starts empty, and at each iteration of the algorithm, we add the next vertex by extracting it from a min-priority queue of vertices⁸.

Input-Output format: Your program will take two file names as its command-line input. This first file includes a weighted directed graph, and it follows the same format from parts 2 and 4. Your program reads the contents of this file and constructs the graph data structure. The first line in this file provides the number of vertices in the graph. Each following line includes information about a weighted directed edge in the graph. Each weighted edge is described by the name of its pair of vertices, followed by the edge weight, separated by a space. Your program must read and construct this graph,

The second file includes single source shortest path queries on the constructed graph. Each line contains a different single-source shortest path query by specifying a source vertex. Your program must read the source vertex, perform Dijkstra's single-source shortest path algorithm using the provided source vertex, and print out each of vertex in the graph in lexicographical ordering, followed by the length of the shortest path to that vertex, and a newline character. Note that an additional newline character follows the last vertex in DAG.

Example Execution:

Let's assume we have the following graph input file:

graph.txt

```
5
A
B
C
D
E
B A 10
A C 8
A D 12
B D 5
C E 5
D C 9
E C 7
E D 3
```

query.txt:

```
A
E
```

Then the result will be:

```
$/sixth graph.txt query.txt
A 0
```

⁸See Dijkstra's shortest path algorithm at https://en.wikipedia.org/wiki/Dijkstra%27s_algorithm

B INF
C 8
D 12
E 13

A INF
B INF
C 7
D 3
E 0

We will not give you improperly formatted files. You can assume all your input files will be in proper format, as stated above. Further, all input graphs edges are nonnegative for this part.

Hints and Suggestions

- Depending on the priority queue's implementation, the asymptotic running time of Dijkstra's algorithm varies. For the this programming assignment, using a simple array-based priority queue is acceptable.

Seventh: Conversion to Two's Complement Representation (10 points)

In this part, your task is to write a C program that prints the two's complement binary representation of a number with a specific number of bits. The argument to the program is an input file, whose format is described in the input format. If a given number is not representable with a given number of bits because the number is greater than largest positive value possible with the given number of bits, then you should print the representation for the largest positive value with the given number of bits in the two's complement representation. If a given number is not representable with a given number of bits because is smaller than smallest negative value with the given number of bits, then you should print the representation for the smallest negative with the given number of bits in the two's complement representation.

Input-Output format: Your program will take one file name as its command-line input. Each line in the input file will have two integers separated by a space: an integer that you want to represent in binary and the number of bits to use for the representation. For each line in the input, you should print out the binary representation of the number followed by a newline character.

Example Execution:

Let's assume we have the following input file:

```
input.txt
42 7
16 4
-9 4
```

When you execute the program, the result should be:

```
$/seventh input.text
0101010
0111
1000
```

We will not give you improperly formatted files. You can assume all your input files will be in proper format, as stated above.

Eighth: Decimal Fraction Input to a Canonical Binary Fraction (15 points)

You will write a program to convert a decimal fraction to a binary fraction in the canonical representation (*i.e.*, $(-1)^s \times M \times 2^E$). For this program, M lies between $[1, 2)$. You do not have to perform any rounding for this part. You are required to print as many digits after the decimal point as specified by the input.

Input-Output format: Your program will take one file name as its command-line input. Each line in the input file will have a decimal fraction (use a double type to read it) and the number of bits to show in the canonical binary representation separate by space. For each line in the input, you should print the M value and E value in the canonical representation separated by space. Add a newline character after printing the output for each input.

Example Execution:

Let's assume we have the following input file:

```
input.txt
6.25 6
12.5 3
```

The result should be:

```
$/eighth input.text
1.100100 2
1.100 3
```

We will not give you improperly formatted files. You can assume all your input files will be in proper format, as stated above. Further, we will provide only positive fractions for this part of the assignment (*i.e.*, no negative numbers).

Ninth: Decimal to IEEE-754 FP with Rounding (20 points)

Your task is to write a program to convert a decimal fraction to the IEEE-754 FP representation in a given configuration with the rounding to nearest with ties-to-even rounding mode.

Input-Output format: Your program will take one file name as its command-line input. Each line in the input file will have a decimal fraction (use a double type to read it), the number of

the bits (n) in the IEEE-754 FP representation, number of bits for the exponent, and number of fraction bits. These numbers on a given line are separated by a space. For each line in the input, you should the IEEE-754 representation with n-bits followed by a new line.

Example Execution:

Let's assume we have the following input file:

```
6.5 8 4 3
.0546875 8 4 3
.013671875 8 4 3
6.375 8 4 3
8.5 8 4 3
9.5 8 4 3
```

Then the result will be:

```
$/ninth input.text
01001101
00010110
00000111
01001101
01010000
01010010
```

We will not give you improperly formatted files. You can assume all your input files will be in proper format, as stated above. You can assume that input will not have NaNs and any value will not round up or down to infinities.

Tenth: Hexadecimal Bit-pattern in the IEEE-FP Format to Decimal Fraction (15 points)

Your task is to write a program that takes a hexadecimal bit-pattern and prints the decimal fractional value of the number.

Input-Output format: Your program will take one file name as its command-line input. Each line in the input file will have the total number of bits, the number of bits for the exponent, number of bits for the fraction, the hexadecimal bit-pattern, and the number of precision bits after the decimal point in the decimal fraction. These numbers on a given line are separated by a space. For each line in the input, you should print out the decimal fraction value with the specified number of precision bits followed by a new line.

Example Execution:

Let's assume we have the following input file:

```
input.txt
8 4 3 0x4d 2
8 4 3 0x16 7
```


Then the output should be:

```
$/tenth input.text
6.50
.0546875
```

Structure of your submission folder

All files must be included in the **pa2** folder. The **pa2** directory in your tar file must contain 10 subdirectories, one each for each of the parts. The name of the directories should be named first through tenth (in lower case). Each directory should contain a **c** source file, a header file (if you use it) and a Makefile. For example, the subdirectory **first** will contain, **first.c**, and any additional **.c** or **.h** (if you create one) and Makefile (the names are case sensitive).

```
pa2
|- first
|  |-- first.c
|  |-- <additional .c .h files> (if used)
|  |-- Makefile
|- second
|  |-- second.c
|  |-- <additional .c .h files> (if used)
|  |-- Makefile
|- third
|  |-- third.c
|  |-- <additional .c .h files> (if used)
|  |-- Makefile
|- fourth
|  |-- fourth.c
|  |-- <additional .c .h files> (if used)
|  |-- Makefile
|- fifth
|  |-- fifth.c
|  |-- <additional .c .h files> (if used)
|  |-- Makefile
|- sixth
|  |-- sixth.c
|  |-- <additional .c .h files> (if used)
|  |-- Makefile
|- seventh
|  |-- seventh.c
|  |-- <additional .c .h files> (if used)
|  |-- Makefile
|- eighth
|  |-- eighth.c
|  |-- <additional .c .h files> (if used)
|  |-- Makefile
|- ninth
|  |-- ninth.c
|  |-- <additional .c .h files> (if used)
|  |-- Makefile
|- tenth
```

```
|-- tenth.c  
|-- <additional .c .h files> (if used)  
|-- Makefile
```

Submission

You have to e-submit the assignment using Canvas. Your submission should be a tar file named **pa2.tar**. To create this file, put everything that you are submitting into a directory (folder) named **pa2**. Then, **cd** into the directory containing **pa2** (that is, **pa2**'s parent directory) and run the following command:

```
tar cvf pa2.tar pa2
```

To check that you have correctly created the tar file, you should copy it (**pa2.tar**) into an empty directory and run the following command:

```
tar xvf pa2.tar
```

This should create a directory named **pa2** in the (previously) empty directory.

The **pa2** directory in your tar file must contain 10 subdirectories, one each for each of the parts. The name of the directories should be named first through ninth (in lower case). Each directory should contain a c source file, a header file and a make file. For example, the subdirectory first will contain, **first.c**, **first.h** and **Makefile** (the names are case sensitive).

AutoGrader

We provide the auto grader (**pa2_autograder.tar.gz**) to test your assignment. Executing the following command will extract the **pa2_autograder**.

```
tar -zxvf pa2_autograder.tar.gz
```

There are two modes available for testing your assignment with the autograder

First mode

Testing when you are writing code with a **pa2** folder

- (1) Lets say you have a **pa2** folder with the directory structure as described in the assignment.
- (2) Copy the folder to the directory of the **pa2_autograder** folder
- (3) Run the auto grader with the following command

```
python pa2_auto_grader.py
```

It will run your programs and print your scores.

Second mode

This mode is to test your final submission (i.e, pa2.tar)

- (1) Copy pa2.tar to the pa2_autograder folder
- (2) Run the auto grader with pa2.tar as the argument.

The command line is

```
python pa2_auto_grader.py pa2.tar
```

Scoring

The autograder will print out information about the compilation and the testing process. At the end, if your assignment is completely correct, the score will something similar to what is given below.

You scored

7.5 in tenth

5.0 in second

7.5 in fourth

7.5 in third

10.0 in sixth

10.0 in ninth

5.0 in seventh

7.5 in eighth

10.0 in fifth

5.0 in first

Your TOTAL SCORE = 75.0 /75

Your assignment will be graded for another 75 points with test cases not given to you

Grading Guidelines

This is a large class so that necessarily the most significant part of your grade will be based on programmatic checking of your program. That is, we will build the binary using the Makefile and source code that you submitted, and then test the binary for correct functionality against a set of inputs. Thus:

- **You should not see or use your friend's code either partially or fully. We will run state of the art plagiarism detectors. We will report everything caught by the tool to Office of Student Conduct.**
- You should make sure that we can build your program by just running `make`.
- Your compilation command with gcc should include the following flags: **-Wall -Werror -fsanitize=address -std=c11**
- You should test your code as thoroughly as you can. For example, programs should *not* crash with memory errors.

- Your program should produce the output following the example format shown in previous sections. Any variation in the output format can result **in up to 100% penalty**. Be especially careful to not add extra whitespace or newlines. That means you will probably not get any credit if you forgot to comment out some debugging message.
- **Your folder names in the path should have not have any spaces. Autograder will not work if any of the folder names have spaces.**

Be careful to follow all instructions. If something doesn't seem right, ask on Canvas discussion forums or contact the TAs during office hours.