**CSE310 Project 3: Adjacency Lists, Heaps, Stacks, Shortest Paths**
**(updated: 11/11/2023)**
**Due: 12/2/2023, 11:59pm**

**This should be your individual work.** As in the case of your first two projects, it should be written using the standard `C++` programming language, and compiled using the `g++` compiler on a Linux platform. Your project will be graded on Gradescope. If you compile your project on `general.asu.edu` using the compiler commands provided in the sample `Makefile`, you should expect the same behavior of your project on both `general.asu.edu` and Gradescope.

# 1 Functionalities of the Project

This project is the essential part of a navigation system. It reads in a directed graph or an undirected graph $G$ with $n$ vertices and $m$ edges from a file specified by the command-line (with proper arguments). It then takes corresponding actions for given instructions from `stdin`. Besides the `Stop` instruction, valid instructions include

$\alpha$: graph printing instruction (`PrintADJ`)

$\beta$: path computation instructions (`SinglePair` and `SingleSource`)

$\gamma$: length/path printing instructions (`PrintLength` and `PrintPath`)

Both `SinglePair` and `SingleSource` path computations should have worst-case time complexity $O(m \log n)$. Path printing should have worst-case time complexity $O(n)$. Length printing should have worst-case time complexity $O(1)$. Memory should be allocated when needed, and released when it is no longer needed. Memory leaks should be avoided.

# 2 Modular Design

For modular design, you need to write the following modules:

- a module for various **data structures**, with header file named `data_structures.h`
- a module for **min-heap**, with header file named `heap.h` and implementation file named `heap.cpp`
- a module for **stack**, with header file named `stack.h` and implementation file named `stack.cpp`
- a module for **graph algorithms**, with header file named `graph.h` and implementation file named `graph.cpp`
- a module for **utilities**, with header file named `util.h` and implementation file named `util.cpp`
- and **the main program**, with header file named `main.h` and implementation file named `main.cpp`

Your project should only use basic C++ libraries to compile your implementation to produce the executable PJ3. To ensure that,the following standard Makefile will be used to grade your project.

```
EXEC = PJ3
CC = g++
CFLAGS = -c -Wall

$(EXEC) :main.o util.o stack.o heap.o graph.o
        $(CC) -o $(EXEC) main.o util.o heap.o graph.o stack.o

main.o  :main.cpp main.h data_structures.h util.h stack.h heap.h graph.h
        $(CC) $(CFLAGS) main.cpp

util.o  :util.cpp util.h data_structures.h
        $(CC) $(CFLAGS) util.cpp

stack.o :stack.cpp stack.h data_structures.h
        $(CC) $(CFLAGS) stack.cpp

heap.o  :heap.cpp heap.h data_structures.h
        $(CC) $(CFLAGS) heap.cpp

graph.o :graph.cpp graph.h data_structures.h stack.h heap.h
        $(CC) $(CFLAGS) graph.cpp

clean   :
        rm *.o $(EXEC)
```

Given that your project will be compile using the above Makefile, you have to name the files exactly as specified.

# 3 Starter Code

You have been given the following files/folders to help you with the project:

1. `Makefile`: The same Makefile as above, and what will be used by Gradescope. Don't submit this file to Gradescope for grading.
2. `test_data` folder: Contains folders for each test case. Each folder in turn contains
   (a) `Execution`: Script to run the code with a specific input file, graph type, and flag.
   (b) `Instructions`: List of instructions/commands that you would normally type in the command line when testing the code in interactive mode.
   (c) `Output.txt`: The expected output file, your output file should match this file for the given test case.

3. `override` folder: Contains the provided network input files.

4. `make_executable.sh`: Script to make all the `Execution` and `Instructions` files executable. You need to run:

```
chmod +x make_executable.sh
```

   to make that file itself executable.

5. `run_test_script.sh`: Script to run all the test cases and checking if you passed or failed that test cases. You need to run:

```
chmod +x run_test_script.sh
```

   to make that file itself executable.

# 4 Data Structures

This section provides some data structures that may give you a head start with the project. You SHOULD NOT copy and paste these data structures. However, you can write similar ones after studying them. It should be your own work.

## 4.1 VERTEX

You need to have a data structure for vertices in the graph. Our algorithms also use a color system on vertices. The following is a part of the codes that I wrote. Note that the definition of `COLOR` in the following is NOT provided.

```
typedef struct TAG_VERTEX{
    int         index;
    COLOR       color;
    double      key;
    int         pi;
    int         position;
}VERTEX;
typedef VERTEX *pVERTEX;
```

Here the field `position` is used to record the **position of the vertex in the min-heap array**. You will find this very handy when `DecreaseKey` operations are needed. I use the `key` field (rather than the `d` field) to reuse my codes for the min-heap.

Here is a sample usage of the data type `VERTEX`.

```
pVERTEX *V;
// V is a pointer to pointer to VERTEX

V = (pVERTEX *) calloc(n+1, sizeof(pVERTEX));
// Now you can make reference to V[1], but not to V[1]->index

for (int i=1; i<=n; i++){
    V[i] = (VERTEX *) calloc(1, sizeof(VERTEX));
    // Now you can make reference to V[1]->index
}
```

Please note that illegal memory access is the main cause of segmentation fault.

## 4.2 EDGE

You need to have a data structure for nodes on the adjacency lists of the graph. I wrote the following in my implementation.

```
typedef struct TAG_NODE{
    int         index;
    int         u;
    int         v;
    double      w;
    TAG_NODE    *next;
}NODE;
typedef NODE *pNODE;
```

The usage of the `NODE` data type is similar to that of `VERTEX`.

## 4.3 MIN-HEAP

In Project 2, you have implemented the min-heap data structure, where the `HEAP` data structure contains the fileds `capacity` (of type `int`), `size` (of type `int`), and `H` (of type `ELEMENT **`). The data structure `ELEMENT` contains only one required field: `key` (of type `double`).

To partially re-use your implementation of Project 2, you can use the following:

```
typedef VERTEX    ELEMENT;
typedef ELEMENT *pELEMENT;
```

Because `VERTEX` has a filed named `key` (of type `double`), your earlier implementations of min-heap should work as usual. Note that you can make reference to the four other fields defined in `VERTEX`, and in `ELEMENT`.

In this project, the elements of the heap array should be pointers to objects of type `VERTEX`. To reuse codes with minimal modification, I used the following in my implementation.

```
typedef VERTEX   ELEMENT;
typedef ELEMENT *pELEMENT;
```

```
typedef struct TAG_HEAP{
    int      capacity;
    int      size;
    pELEMENT *H;
}HEAP;
typedef HEAP *pHEAP;
```

### 4.4 STACK

In order to print out the computed path from a source node `s` to a destination node `t`, we start from the destination node to trace out the path using the predecessor field. This will trace out the path in reverse order. In order to print the path in correct order, we need to use a stack. The elements of the stack should contain information (or pointers to information) about the corresponding vertices on the path.

## 5 Valid Executions

A valid execution of your project has the following form:

`./PJ3 <InputFile> <GraphType> <Flag>`

where

- `PJ3` is the executable file of your project,
- `<InputFile>` should be the exact name of the input file,
- `<GraphType>` should be substituted by either `DirectedGraph` or `UndirectedGraph`,
- `<Flag>` is either 1 or 2.

As you can see, the PJ3 executable is not taking the output file name as an input. We'll address how to write the data to a file in Section 8. Your program should check whether the execution is valid. If the execution is not valid, your program should print out the following message to `stderr` and stop.

`Usage: ./PJ3 <InputFile> <GraphType> <Flag>`

Note that your program should not crash when the execution is not valid.

## 6 Flow of the Project

### 6.1 Read in the Graph and Build the Adjacency Lists

Upon a valid execution, your program should open the input file (specified by `argv[1]`) and read in the graph. The format of the input file is the following. The first line contains two positive integers, representing the number of vertices $n$ and number of edges $m$, respectively. Each of the next $m$ lines has the following format:

`edgeIndex u v w(u, v)`

where $u$ is the start vertex of edge $(u, v)$, $v$ is the end vertex of edge $(u, v)$, $w(u, v)$ is the weight of edge $(u, v)$, and *edgeIndex* is the index of edge $(u, v)$. Your program should read in `edgeIndex`, `u`, and `v` as `int`, and read in `w(u, v)` as `double`. *For undirected graphs, each edge still has two vertices but the order is not important.*

After knowing the values of $n$ and $m$, your program should dynamically allocate memory for an array of $n$ pointers to objects of type `VERTEX`. Let us call this array `V`. Then for each $i = 1, 2, \ldots, n$, `V[i]` is a pointer to an object for vertex $i$. Your program should dynamically allocate memory for an array of $n$ adjacency lists. Let us call this array `ADJ`. Then for each $i = 1, 2, \ldots, n$, `ADJ[i]` is a pointer to the adjacency list of vertex $i$.

While reading the graph from `argv[1]` edge by edge, the adjacency lists are populated accordingly. Note that dynamic memory allocation is needed for each node (corresponding to an edge) on the adjacency lists.

Let $(u, v)$ be the edge newly read in from the file `argv[1]`. If the graph is directed and `flag` is 1, insert the corresponding node for edge $(u, v)$ at the front of `ADJ[u]`. If the graph is directed and `flag` is 2, insert the corresponding node for edge $(u, v)$ at the rear of `ADJ[u]`. If the graph is undirected and `flag` is 1, insert a corresponding node for edge $(u, v)$ at the front of `ADJ[u]`, and insert a corresponding node for edge $(v, u)$ at the front of `ADJ[v]`. If the graph is undirected and `flag` is 2, insert a corresponding node for edge $(u, v)$ at the rear of `ADJ[u]`, and insert a corresponding node for edge $(v, u)$ at the rear of `ADJ[v]`. After all $m$ edges are read in, your program should close the file `argv[1]`.

## 6.2 Initialize the Priority Queue (min-heap) and the Stack

Your program should initialize a min-heap of capacity $n$ and a stack of capacity $n$. Dynamic memory allocations are required for the min-heap and the stack. The heap will be used in the execution of Dijkstra's path finding algorithm. The stack will be used to output a path from a given source vertex to a given destination vertex.

## 6.3 Loop over the Instructions

Your program should expect the following instructions from `stdin` and act accordingly:

(a) `Stop`
On reading `Stop`, the program stops.

(b) `PrintADJ`
On reading the `PrintADJ` instruction, your program should do the following:

(b-i) Print the adjacency lists of the input graph to `stdout`. Note: Writing to `stdout` means printing the output to the screen, not writing it to a file. Refer to posted test cases for output format.

(b-ii) Wait for the next instruction from `stdin`.

(c) `SinglePair <source> <destination>`
where `<source>` and `<destination>` are two integers in the set $\{1, 2, \ldots, n\}$. **This is one of two *path computation* instructions.** On reading the `SinglePair` instruction, your program should do the following:

(c-i) Apply the variant of Dijkstra's algorithm (as taught in class and stated in the lecture slides) to compute a shortest path from `<source>` to `<destination>`. In particular, for each $i \in \{1, 2, \ldots, n\}$, the algorithm should compute the final values of `V[i]->key` and `V[i]->`$\pi$. If `<destination>` is reachable from `<source>`, `V[destination]->key` is the weight of a shortest path from `<source>` to `<destination>`, and `V[destination]->`$\pi$ is the predecessor of `<destination>` on the computed shortest path from `<source>` to `<destination>`. If `<destination>` is not reachable from `<source>`, `V[i]->`$\pi$ is `nil`, and `V[i]->key` is `DBL_MAX` (defined in the header file `<cfloat>`).

(c-ii) Wait for the next instruction from `stdin`.

(d) `SingleSource <source>`
where `<source>` is an integer in the set $\{1, 2, \ldots, n\}$. **This is the other *path computation* instruction.** On reading the `SingleSource` instruction, your program should do the following:

(d-i) Apply Dijkstra's algorithm to compute shortest paths from `<source>` to all vertices that are reachable from `<source>`. In particular, for each $i \in \{1, 2, \ldots, n\}$, the algorithm should compute the final values of `V[i]->key` and `V[i]->`$\pi$. If vertex $i$ is reachable from `<source>`, `V[i]->key` is the length of a shortest path from `<source>` to $i$, and `V[i]->`$\pi$ is the predecessor of $i$ on the computed shortest path from `<source>` to $i$. If vertex $i$ is not reachable from `<source>`, `V[i]->`$\pi$ is `nil`, and `V[i]->key` is `DBL_MAX`.

(d-ii) Wait for the next instruction from `stdin`.

(e) `PrintLength <s> <t>`
where `<s>` and `<t>` are two integers in the set $\{1, 2, \ldots, n\}$. **This is the only *length printing* instruction.** This instruction is valid if and only if `<s>` is the same as `<source>` in the most recent path computation instruction, and `<t>` is the same as `<destination>` in case the most recent path computation instruction is a `SinglePair` instruction.

On reading a valid `PrintLength` instruction, your program should do the following:

(e-i) If your program has computed a shortest `<s>` to `<t>` path in the most recent path computation, print the length of the computed path to `stdout`. Note: Writing to `stdout` means printing the output to the screen, not writing it to a file. Refer to posted test cases for output format.

If your program has not computed a shortest `<s>` to `<t>` path in the most recent path computation, print the following to `stdout`:

`There is no path from <s> to <t>.`

In the above, `<s>` and `<t>` should be replaced by their given values. Refer to posted test cases for output format.

(e-ii) Wait for the next instruction from `stdin`.

(f) `PrintPath <s> <t>`
where `<s>` and `<t>` are two integers in the set $\{1, 2, \ldots, n\}$. **This is the only *path printing* instruction.** This instruction is valid if and only if `<s>` is the same as `<source>` in the most recent path computation instruction, and `<t>` is the same as `<destination>` in case the most recent path computation instruction is a `SinglePair` instruction.

On reading a valid `PrintPath` instruction, your program should do the following:

(f-i) If your program has computed a shortest `<s>` to `<t>` path in the most recent path computation, print the computed path to `stdout`. Note: Writing to `stdout` means printing the output to the screen, not writing it to a file. Refer to posted test cases for output format.

If your program has not computed a shortest `<s>` to `<t>` path in the most recent path computation, print the following to `stdout`:

`There is no path from <s> to <t>.`

In the above, `<s>` and `<t>` should be replaced by their given values. Note: Writing to `stdout` means printing the output to the screen, not writing it to a file. Refer to posted test cases for output format.

(f-ii) Wait for the next instruction from `stdin`.

(g) `Invalid instruction`
On reading an invalid instruction, your program should do the following:

(g-i) Write the following to `stderr`:

`Invalid instruction.`

(g-ii) Wait for the next instruction from `stdin`.

# 7 Format of the Input File

Refer to Section 6.1 and the posted test cases for the format of the input file. In the provided starter code, you can find the input files in the folder `override`. While `w(u, v)` may look like an `int` in the input file, it should be read in as `double`.

# 8 Format of the Output

Because the `PJ3` executable does not take the output file name as the input, our program will technically be writing the output to the `stdout`. However, you can redirect the content from `stdout` to a text file from the terminal.

```
./PJ3 <InputFile> <GraphType> <Flag>  >  <OutputFile>
```

For example:

```
./PJ3 override/network01.txt DirectedGraph 1 > myOutputFile.txt
```

This will write your output to `myOutputFile.txt`. Now if you have a file for instructions such as `Instructions.txt`, you can specify that as well. It should look like:

```
./PJ3 override/network01.txt DirectedGraph 1 <Instructions> myOutputFile.txt
```

We have given you a `test_data` folder, which has subfolders for each test cases. If you want to run test case 1 from the root of your directory, your command would be like:

```
./test_data/1/Execution < test_data/1/Instructions > test_data/1/myOutput.txt
```

Go through the `./test_data/1/Execution` and `./test_data/1/Instruction` files to see how they work. You can then compare your output with the expected output by running:

```
diff test_data/1/myOutput.txt test_data/1/Output.txt
```

Since the autograder on Gradescope uses the Linux `diff` function to compare your output with the expected output, your output has to match the expected output exactly. It is your responsibility to make sure that your output matches the expected output, with the aid of the posted test cases. The following are some general guidelines to help you.

## 8.1 Output Integers to `stdout`

All integers (to `stdout`) should be printed using the `%d` format. Refer to the posted test cases.

## 8.2 Output Floating Point Numbers to `stdout`

Edge weights (to `stdout`) should be printed using the `%4.2lf` format when printing the adjacency lists (for easy debugging). In all other places, doubles (to `stdout`) should be printed using the `%8.2lf` format. Refer to the posted test cases.

## 8.3 Output Strings (possibly with numerical values) to `stdout`

Refer to the posted test cases for strings printed to `stdout`. Keep in mind the rules in Section 8.1 (for `int`) and Section 8.2 (for `double`).

### 8.4 Output to `stderr`

While the autograder does not test the messages written to `stderr`, your program should output meaningful messages with sufficient information when a warning or error message should be written to `stderr`.

# 9 Submission

You should submit your project to Gradescope via the link on Canvas. Don't submit your Makefile, only submit all header files and implementation files. You should put your name and ASU ID at the top of each of the header files and the implementation files, as a comment.

Submissions are always due before `11:59pm` on the deadline date. Do not expect the clock on your machine to be synchronized with the one on Canvas/Gradescope. **This project is due on 12/2/2023**. It is your responsibility to submit your project well before the deadline. **Since you have about 3 weeks to work on this project, no extension request (too busy, other business, sick, need more accommodations) is a valid one.**

The instructor and the TAs will offer more help to this project early on, and will not answer emails/questions near the project due date that are clearly in the very early stage of the project. So, please **manage your time, and start working on this project today**.

You should not use Gradescope and the autograder on Gradescope as a debugger. The main purpose of the test cases are for you to get familiar with the output format. It is suggested to use a similar shell script to test your program on `general.asu.edu`. You also need to submit to Gradescope early enough to avoid last-minute surprises.

# 10 Grading

**All programs will be compiled and graded on Gradescope**. **If your program does not compile and work on Gradescope, you will receive** 0 **on this project**. If your program works well on `general.asu.edu`, there should not be much problems. The maximum possible points for this project is 100. The following shows how you can have points deducted.

1. **Non-working program**: If your program does not compile or does not execute on Gradescope, you will receive a 0 on this project. Do not claim "my program works perfectly on my PC, but I do not know how to use Gradescope."

2. **Program requires non-standard libraries**: If your program does not compile/working using the compiler and flags as stated in the sample Makefile, 20 points will be marked off.

3. **Posted test cases**: For each of the posted test cases that your program fails, 4 points will be marked off.

4. **Hidden test cases**: For each of the Hidden test cases that your program fails, 4 points will be marked off.

# 11 Examples and Test Cases

Test cases will be posted on Canvas shortly. The following is the content of the input file `network01.txt`, which is a directed graph with $n = 8$ and $m = 14$.

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

This graph is illustrated in Figure 1. You are highly encouraged to debug your program using this directed graph.

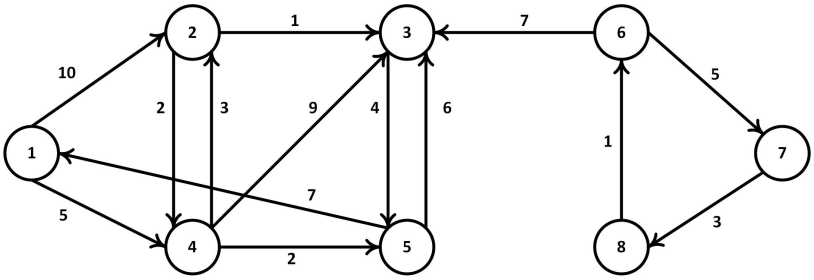


Figure 1: Directed graph $G_1$ contained in the file `network01.txt`

The following is the content of the input file `network02.txt`, which is an undirected graph with $n = 8$ and $m = 11$.

```
 8 11
 1 1 2 10
 2 1 4 5
 3 1 5 7
 4 2 3 1
 5 2 4 3
 6 3 4 9
 7 3 5 4
 8 3 6 7
```

```
 9 4 5 2
10 5 6 6
11 7 8 3
```

This graph is illustrated in Figure 2. You are highly encouraged to debug your program using this undirected graph.

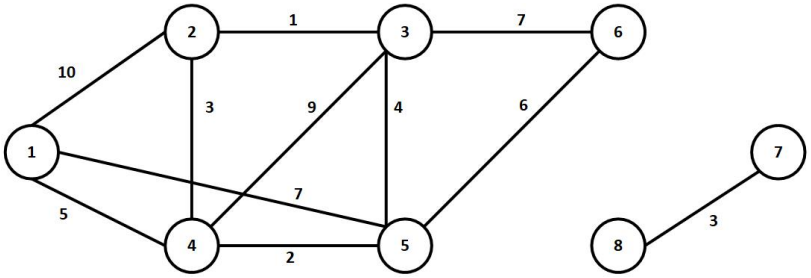


Figure 2: Undirected graph $G_2$ contained in the file `network02.txt`

Some examples are presented in the following subsections. I ran the following command to generate the output file from the root directory:

```
./test_data/1/Execution < test_data/1/Instructions > test_data/1/Output.txt
```

For different examples the test case number changes, but again these are just examples.

## 11.1 Example 1

The content of `Execution` is

```
#!/bin/bash
./PJ3 network01.txt DirectedGraph 1
```

The content of `Instructions` is

```
PrintADJ
Stop
```

The content of the produced `Output.txt` is

```
ADJ[1]:-->[1 4: 5.00]-->[1 2: 10.00]
ADJ[2]:-->[2 4: 2.00]-->[2 3: 1.00]
ADJ[3]:-->[3 5: 4.00]
ADJ[4]:-->[4 5: 2.00]-->[4 3: 9.00]-->[4 2: 3.00]
ADJ[5]:-->[5 3: 6.00]-->[5 1: 7.00]
ADJ[6]:-->[6 7: 5.00]-->[6 3: 7.00]
ADJ[7]:-->[7 8: 3.00]
ADJ[8]:-->[8 6: 1.00]
```

Please note that the nodes are inserted **at the front** of the adjacency list.

## 11.2 Example 2

The content of `Execution` is

```
#!/bin/bash
./PJ3 network01.txt DirectedGraph 2
```

The content of `Instructions` is

```
PrintADJ
Stop
```

The content of the produced `Output.txt` is

```
ADJ[1]:-->[1 2: 10.00]-->[1 4: 5.00]
ADJ[2]:-->[2 3: 1.00]-->[2 4: 2.00]
ADJ[3]:-->[3 5: 4.00]
ADJ[4]:-->[4 2: 3.00]-->[4 3: 9.00]-->[4 5: 2.00]
ADJ[5]:-->[5 1: 7.00]-->[5 3: 6.00]
ADJ[6]:-->[6 3: 7.00]-->[6 7: 5.00]
ADJ[7]:-->[7 8: 3.00]
ADJ[8]:-->[8 6: 1.00]
```

Please note that the nodes are inserted **at the rear** of the adjacency list.

## 11.3 Example 3

The content of `Execution` is

```
#!/bin/bash
./PJ3 network02.txt UndirectedGraph 1
```

The content of `Instructions` is

```
PrintADJ
Stop
```

The content of produced `Output.txt` is

```
ADJ[1]:-->[1 5: 7.00]-->[1 4: 5.00]-->[1 2: 10.00]
ADJ[2]:-->[2 4: 3.00]-->[2 3: 1.00]-->[2 1: 10.00]
ADJ[3]:-->[3 6: 7.00]-->[3 5: 4.00]-->[3 4: 9.00]-->[3 2: 1.00]
ADJ[4]:-->[4 5: 2.00]-->[4 3: 9.00]-->[4 2: 3.00]-->[4 1: 5.00]
ADJ[5]:-->[5 6: 6.00]-->[5 4: 2.00]-->[5 3: 4.00]-->[5 1: 7.00]
ADJ[6]:-->[6 5: 6.00]-->[6 3: 7.00]
ADJ[7]:-->[7 8: 3.00]
ADJ[8]:-->[8 7: 3.00]
```

## 11.4 Example 4

The content of `Execution` is

```
#!/bin/bash
./PJ3 network02.txt UndirectedGraph 2
```

The content of `Instructions` is

```
PrintADJ
Stop
```

The content of produced `Output.txt` is

```
ADJ[1]:-->[1 2: 10.00]-->[1 4: 5.00]-->[1 5: 7.00]
ADJ[2]:-->[2 1: 10.00]-->[2 3: 1.00]-->[2 4: 3.00]
ADJ[3]:-->[3 2: 1.00]-->[3 4: 9.00]-->[3 5: 4.00]-->[3 6: 7.00]
ADJ[4]:-->[4 1: 5.00]-->[4 2: 3.00]-->[4 3: 9.00]-->[4 5: 2.00]
ADJ[5]:-->[5 1: 7.00]-->[5 3: 4.00]-->[5 4: 2.00]-->[5 6: 6.00]
ADJ[6]:-->[6 3: 7.00]-->[6 5: 6.00]
ADJ[7]:-->[7 8: 3.00]
ADJ[8]:-->[8 7: 3.00]
```

## 11.5 Example 5

The content of `Execution` is

```
#!/bin/bash
./PJ3 network01.txt DirectedGraph 1
```

The content of `Instructions` is

```
SinglePair 1 3
PrintPath 1 3
Stop
```

The content of the produced `Output.txt` is

```
The shortest path from 1 to 3 is:
[1:    0.00]-->[4:    5.00]-->[2:    8.00]-->[3:    9.00].
```

## 11.6 Example 6

The content of `Execution` is

```
#!/bin/bash
./PJ3 network01.txt DirectedGraph 2
```

The content of `Instructions` is

```
SinglePair 1 3
PrintPath 1 3
Stop
```

The content of the produced `Output.txt` is

```
The shortest path from 1 to 3 is:
[1:    0.00]-->[4:    5.00]-->[2:    8.00]-->[3:    9.00].
```

## 11.7 Example 7

The content of `Execution` is

```
#!/bin/bash
./PJ3 network01.txt DirectedGraph 1
```

The content of `Instructions` is

```
SinglePair 1 6
PrintPath 1 6
Stop
```

The content of the produced `Output.txt` is

```
There is no path from 1 to 6.
```

## 11.8 Example 8

The content of `Execution` is

```
#!/bin/bash
./PJ3 network02.txt UndirectedGraph 1
```

The content of `Instructions` is

```
SinglePair 1 3
PrintPath 1 3
Stop
SingleSource 1
SinglePair 1 3
```

The content of the produced `Output.txt` is

```
The shortest path from 1 to 3 is:
[1:    0.00]-->[4:    5.00]-->[2:    8.00]-->[3:    9.00].
```

## 11.9 Example 9

The content of `Execution` is

```
#!/bin/bash
./PJ3 network02.txt UndirectedGraph 1
```

The content of `Instructions` is

```
SinglePair 1 8
PrintPath 1 8
Stop
SingleSource 1
SinglePair 1 3
```

The content of the produced `Output.txt` is

```
There is no path from 1 to 8.
```

## 11.10 Example 10

The content of `Execution` is

```
#!/bin/bash
./PJ3 network01.txt DirectedGraph 1
```

The content of `Instructions` is

```
SingleSource 1
PrintPath 1 5
PrintPath 1 8
Stop
SingleSource 1
SinglePair 1 3
```

The content of the produced `Output.txt` is

```
The shortest path from 1 to 5 is:
[1:    0.00]-->[4:    5.00]-->[5:    7.00].
There is no path from 1 to 8.
```

## 11.11 Example 11

The content of `Execution` is

```
#!/bin/bash
./PJ3 network01.txt DirectedGraph 1
```

The content of `Instructions` is

```
SingleSource 1
PrintLength 1 5
PrintLength 1 8
Stop
SingleSource 1
SinglePair 1 3
```

The content of the produced `Output.txt` is

```
The length of the shortest path from 1 to 5 is:     7.00
There is no path from 1 to 8.
```

## 11.12 Example 12

The content of `Execution` is

```
#!/bin/bash
./PJ3 network02.txt UndirectedGraph 1
```

The content of `Instructions` is

```
SingleSource 1
PrintPath 1 5
PrintPath 1 8
Stop
SingleSource 1
SinglePair 1 3
```

The content of the produced `Output.txt` is

```
The shortest path from 1 to 5 is:
[1:    0.00]-->[5:    7.00].
There is no path from 1 to 8.
```

## 11.13 Example 13

The content of `Execution` is

```
#!/bin/bash
./PJ3 network02.txt UndirectedGraph 1
```

The content of `Instructions` is

```
SingleSource 1
PrintLength 1 5
PrintLength 1 8
Stop
SingleSource 1
SinglePair 1 3
```

The content of the produced `Output.txt` is

```
The length of the shortest path from 1 to 5 is:     7.00
There is no path from 1 to 8.
```

## 11.14 Example 14

The content of `Execution` is

```
#!/bin/bash
./PJ3 network03.txt DirectedGraph 1
```

The content of `Instructions` is

```
SingleSource 1
PrintPath 1 2
PrintPath 1 3
PrintPath 1 4
PrintPath 1 5
PrintPath 1 6
PrintPath 1 7
PrintPath 1 8
PrintPath 1 9
Stop
```

The content of the produced `Output.txt` is

```
The shortest path from 1 to 2 is:
[1:    0.00]-->[2:    9.93].
The shortest path from 1 to 3 is:
[1:    0.00]-->[3:    1.36].
The shortest path from 1 to 4 is:
[1:    0.00]-->[3:    1.36]-->[4:    1.76].
The shortest path from 1 to 5 is:
[1:    0.00]-->[3:    1.36]-->[4:    1.76]-->[5:    4.18].
The shortest path from 1 to 6 is:
[1:    0.00]-->[3:    1.36]-->[4:    1.76]-->[5:    4.18]-->[6:    9.10].
The shortest path from 1 to 7 is:
[1:    0.00]-->[3:    1.36]-->[4:    1.76]-->[5:    4.18]-->[6:    9.10]-->[7:   23.51].
The shortest path from 1 to 8 is:
[1:    0.00]-->[2:    9.93]-->[8:   12.69].
The shortest path from 1 to 9 is:
[1:    0.00]-->[2:    9.93]-->[8:   12.69]-->[9:   19.88].
```

## 11.15 Example 15

The content of `Execution` is

```
#!/bin/bash
./PJ3 network01.txt DirectedGraph 1
```

The content of `Instructions` is

```
SingleSource 1000
PrintPath 1000 1010
Stop
```

The content of produced `Output.txt` is

```
The shortest path from 1000 to 1010 is:
[1000:    0.00]-->[1004:    1.24]-->[1006:    3.54]-->[1010:    7.15].
```

## 11.16 Example 16

The content of `Execution` is

```
#!/bin/bash
./PJ3 network01.txt DirectedGraph 3
```

The content of `Instructions` is

```
PrintADJ
Stop
```

The file `Output.txt` is empty. The following message is written to `stderr`.

```
Usage: ./PJ3 <InputFile> <GraphType> <Flag>
```

## 11.17 Example 17

The content of `Execution` is

```
#!/bin/bash
./PJ3 network01.txt DirectedGraph 1
```

The content of `Instructions` is

```
PrintPath 1 3
Stop
```

The file `Output.txt` is empty. The following is writrten to `stderr`.

```
Error: Invalid instruction.
```

### 11.18 Example 18

The content of `Execution` is

```
#!/bin/bash
./PJ3 network01.txt DirectedGraph 1
```

The content of `Instructions` is

```
SingleSource 1
PrintPath 1 2
PrintPath 2 3
PrintPath 1 8
SingleSource 2
PrintPath 1 2
PrintPath 2 5
Stop
```

The content of produced `Output.txt` is

```
The shortest path from 1 to 2 is:
[1:    0.00]-->[4:    5.00]-->[2:    8.00].
There is no path from 1 to 8.
The shortest path from 2 to 5 is:
[2:    0.00]-->[4:    2.00]-->[5:    4.00].
```

## 12 Suggested Schedule

**The workload for this project is not light. It is recommended that you start working on it immediately.**

I suggest that you try the following schedule, starting immediately.

1. **Day 1:** Get the data structures implemented and compiled on `general.asu.edu`. You can have each of the functions having an empty shell for now (just to have the correct syntax).
2. **Day 2:** Read in the graph and build the adjacency lists for directed graphs, with `flag`=1, using `network01.txt`. Make sure it compiles and works correctly on `general.asu.edu`. You can use Example 1 as an aid.
3. **Day 3:** Read in the graph and build the adjacency lists for directed graphs, with `flag`=2, using `network01.txt`. Make sure it compiles and works correctly on `general.asu.edu`. You can use Example 2 as an aid.
4. **Day 4:** Read in the graph and build the adjacency lists for undirected graphs, with `flag`=1 and 2, using `network02.txt`. Make sure it compiles and works correctly on `general.asu.edu`. You can use Examples 3 and 4 as an aid.

5. **Day 5:** Read in all valid instructions, and print the instructions to `stderr`. Make sure it compiles and works correctly on `general.asu.edu`. You should be able to get this done easily after the first project.

6. **Day 6:** Write the function for `SinglePair` and `PrintPath`. Make sure it compiles and works correctly on `general.asu.edu`.

7. **Day 7:** Write the function for `SingleSource` and `PrintLength`. Make sure it compiles and works correctly on `general.asu.edu`.

8. **Day 8:** Work on all possible cases you can think of (including invalid instructions). Work on output format (use the `diff` utility to compare files).

9. **Day 9:** Submit to Gradescope and see how many test cases your code passes.

If you can keep up with the proposed schedule or get one day's work in two days, you should be on target to get a perfect score. If you cannot keep up with the proposed schedule, you need to work harder. Please note that the first few test cases are easy, but some of the later ones are much harder. Start early and manage your time wisely.

**Good Luck!**