EE471/CS471/CS573: Fall 2015-2016
Zartash A. Uzmi

Programming Assignment 3

(Due December 14, 2015 at 11:59pm; 30% penalty for max 24-hour late)

1 What this assignment is about?

You are required to implement a subset of the software that runs on bridges, i.e., you will write a simplified version of the "Spanning Tree Protocol". In an extended LAN (a.k.a. Switched Ethernet), each bridge (or ethernet switch or layer 2 switch) runs the spanning tree protocol to facilitate the formation of a spanning tree of the extended LAN. The resulting spanning tree is used by the Bridges to forward (or not forward) the frames from one LAN segment to another.

The implementation is required in C programming language. The programming assignments will be graded on a Linux system; your code must compile and run correctly on venus on which all the registered students already have an account. You should use the Makefile provided with this assignment to compile and link your code. You can do the development on other systems as well but it is your responsibility to make sure that your code correctly runs on a venus system. Details for downloading the appropriate files and programming environment help are given on the class web site.

The spanning tree protocol is a distributed protocol but we don't require you to know the issues involved in distributed systems for doing this programming assignment. The protocol mechanism is described in Section 2 and address learning is given in Section 3. Note that the formation of spanning tree is done when bridges exchange control information among themselves and it is not required to attach any station to any of the LAN segments. However, the (address) learning process, described in Section 3, requires that stations are attached to the LAN segments. Further note that learning is solely based on the information originated by the end stations and is not dependent upon the information originated by the bridges.

2 Spanning Tree Protocol

2.1 Description

The implementation of the spanning tree protocol runs on each bridge in the extended LAN. The protocol creates a spanning tree to eliminate loops that may exist in the topology. This is needed in LAN segments connected by bridges because loops in the network topology create instability in learning and endless circulation of multicast and broadcast traffic.

The spanning tree is created by exchange of messages between the bridges. These special messages are called Configuration Bridge Protocol Data Units, or Configuration BPDUs. We will use the term BPDU to refer to Configuration BPDUs throughout this document. In the spanning tree protocol, bridges exchange and process the BPDUs. Towards this end, one bridge is selected to be the root of the spanning tree. The spanning tree is then created which consists of shortest paths from each bridge to the root.

For this assignment, assume that the BPDU message has the following form:

<Root ID>.<Cost>.<Transmitting Bridge ID>.<Transmitting Port ID>

A BPDU contains all of the information given in Table 1.

Data	**Explanation of Data**
Root ID	ID of the bridge currently believed/known to be root
Transmitting Bridge ID	ID of the bridge transmitting this BPDU
Cost	Cost of the least cost path from the transmitting bridge to the currently known root
Transmitting Port ID	ID of the port out of which BPDU is transmitted

Table 1: Information Contained in BPDUs

2.2 Protocol Operation

As described above, the protocol serves to build a tree in which one bridge is determined to be the root, while the other bridges make up a spanning tree. Bridges can only forward data frames towards the root bridge or away from the root bridge.

Each bridge is assigned a unique bridge ID and the root bridge is determined to be the one with the lowest bridge ID. When a bridge is first booted up, it has no information about any other bridge in the LAN, so it believes that it is the root bridge itself. Therefore, it builds a BPDU of the form <its bridge ID>.<0>.<its bridge ID>.<Port ID> and transmits one such BPDU on each of its ports. A BPDU transmitted by a bridge on a given port is multicast to all the bridges on the LAN segment to which it is connected via that port. Likewise, a bridge receives BPDUs sent by all the other bridges on that segment. The bridge must process the received information and reevaluate which bridge it now believes to be the root bridge and its distance to the new root bridge. Depending on the BPDUs received, the bridge may block some of its ports and designate others. Finally, the bridge sends out this newly determined information on all designated ports. The process of sending, receiving, and processing BPDUs continues until a static spanning tree is formed. After the algorithm converges, one bridge is elected as the root bridge, while all other bridges have calculated their shortest path cost to the root bridge. An example of a spanning tree is shown in Figure 1.

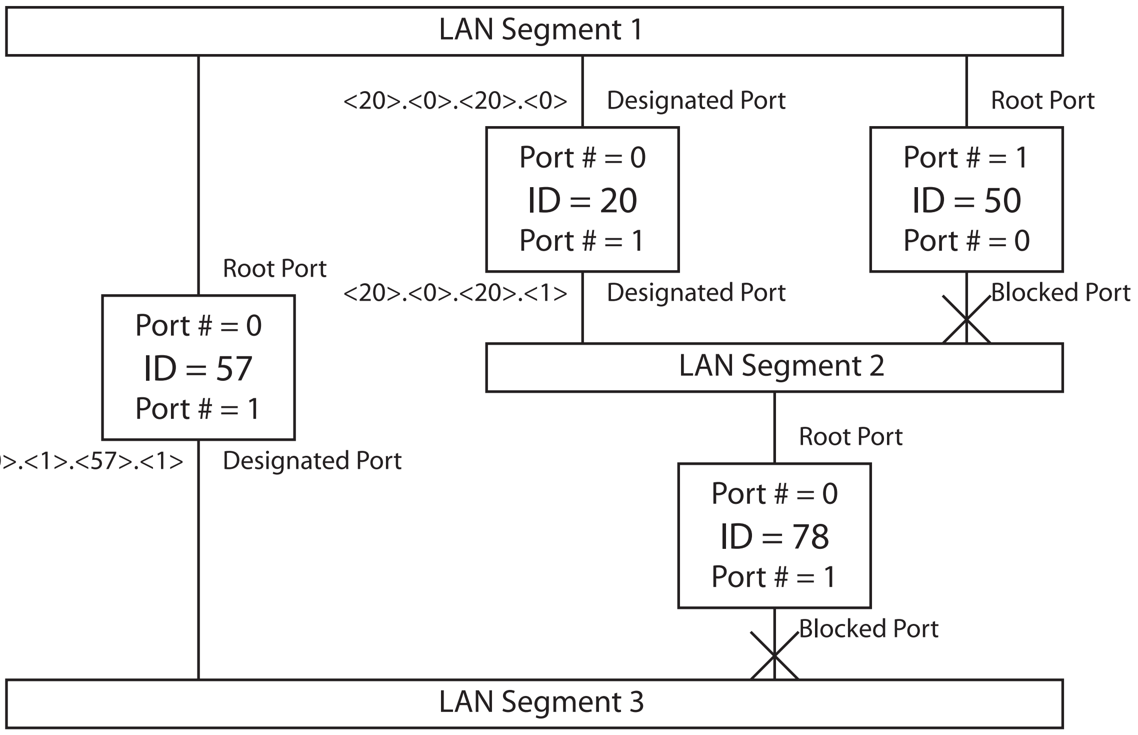

Figure 1: Spanning Tree Example

Bridge with bridge ID 20 (or simply bridge 20) is determined to be the root bridge since it has the lowest bridge ID. All ports of bridge 20 are designated. Every other bridge has found its root port, and either designated or blocked all other ports. A root port is the port that leads to the root bridge along the shortest path, and a designated port is an unblocked port connected to a segment. A blocked port is not part of the spanning tree, and no data traffic flows through it.

In above figure, bridge 50 has two ports that lead to the root bridge with the same cost. In this case, the transmitting port ID of bridge 20 is used to determine that port 1 of bridge 50 should be the root port (details later in Section 2.3).

There can only be one designated bridge per LAN segment, and it is the bridge with lowest cost to the root. Thus, root bridge is designated on all the LAN segments it is connected to, and all ports of the root bridge are designted. In case of a tie in determining the designated bridge for a LAN segment, the bridge with the lowest bridge ID is chosen. Thus for segment 3, bridge 57 (rather than bridge 78) is chosen as the designated bridge.

2.3 Protocol Algorithm

The protocol algorithm is described below:

1. Initially each bridge assumes itself to be the root bridge and transmits the following BPDU on each of its ports:

   <Transmitting Bridge ID>.<0>.<Transmitting Bridge ID>.<Transmitting Port ID>

2. During a period of time (one simulation cycle for this assignment), each bridge receives BPDUs on each of its ports. For each port, the bridge saves the "best" among all BPDUs received and the BPDU that it would transmit on that port. When comparing the BPDUs, we use the following rules:

   - A BPDU with lower root bridge ID is "better" than any other BPDU with higher root bridge ID.
   - If above rule is a tie (i.e., root bridge ID is same), then a BPDU with lower cost is "better" than any other BPDU with higher cost.
   - If above two rules result in a tie (i.e., root bridge ID and cost are the same), then a BPDU with (numerically) lower transmitting bridge ID is better than any other BPDU with higher transmitting bridge ID.
   - If all three rules above result in a tie (i.e., root bridge ID, cost, and transmitting bridge ID are the same), then a BPDU with lower transmitting port ID is "better" than any other BPDU with higher transmitting port ID.

   Examples:

   $< 29 > . < 15 > . < 35 > . < 0 >$ is better than $< 31 > . < 12 > . < 32 > . < 1 >$

   $< 35 > . < 80 > . < 39 > . < 1 >$ is better than $< 35 > . < 80 > . < 40 > . < 0 >$

   $< 35 > . < 15 > . < 80 > . < 2 >$ is better than $< 35 > . < 18 > . < 38 > . < 1 >$

   $< 35 > . < 15 > . < 80 > . < 0 >$ is better than $< 35 > . < 15 > . < 80 > . < 1 >$

3. If a bridge receives a BPDU on a port that is better than the one that it would transmit on that port, it no longer transmits BPDUs on that port (that port will be named as either root port or blocked port). Therefore, when the algorithm stabilizes, only one

bridge on each segment (the designated bridge for that segment) transmits BPDUs on that segment.

Based on the received BPDUs from all the interfaces, each bridge independently determines the identity of the root bridge (which will eventually converge to a single identity). After receiving BPDUs, the bridge ID of the "newly determined" root bridge is, therefore, the minimum of a bridge's own bridge ID and the root ID contained in all other received BPDUs. Each bridge is able to determine its lowest cost to the root, its root port, and its designated ports, if any.

As an example, assume that bridge B has a bridge ID of 18. Suppose that the "best" BPDU received by bridge B on each of its ports is as given in Table 2:

Port Number	**Best BPDU Received on the Port**
Port 1	$< 12 > . < 85 > . < 51 > . < 0 >$
Port 2	$< 12 > . < 85 > . < 32 > . < 2 >$
Port 3	$< 81 > . < 0 > . < 81 > . < 1 >$
Port 4	$< 15 > . < 31 > . < 27 > . < 1 >$

Table 2: Example: Best BPDUs received on each port

From the best BDPUs received by bridge B on its ports, it is clear that both port 1 and port 2 received BPDUs containing the lowest root ID. Bridge B has bridge ID 18 which is greater than 12. Therefore, bridge B is not the root bridge, and after B receives the BPDUs listed in Table 2, bridge B believes that bridge 12 is the root bridge. Bridge B must also determine the cost to the root bridge; this cost is calculated by adding 1 to the lowest cost reported to bridge 12 by any of the best BPDUs given in Table 2. Likewise, bridge B must also select its root port—the port through which the root bridge can be reached via the smallest cost. Since the BPDUs received at port 1 and port 2 report the same cost to root (i.e., 85), the cost to root for bridge B is $1+85 = 86$ and since the transmitting bridge ID is lower in the BPDU received at port 2, bridge B selects port 2 as its root port. Bridge B no longer transmits BPDUs on port 2.

Bridge B now has a BPDU of $< 12 > . < 86 > . < 18 >$. <possible port number on bridge B>. This BPDU is better than the ones received at ports 3 and 4, so bridge B becomes designated bridge for the two LAN segments, one connected to port 3 and the other to port 4. Port 3 and port 4 are the designated ports, and bridge B continues to transmit BPDUs on these ports.

Finally, the BPDU bridge B would send out is not better than the BPDU received at port 1. Port 1 is, therefore, blocked and bridge B no longer transmits BPDUs out of this port.

Thus bridge B has determined the following:

- the root bridge and bridge B's root port (bridge 12 and port 2)
- its own distance (cost) to the root (86)
- ports for which it is designated (3, 4)
- blocked ports (1)

4. At each step, the bridge must decide which ports to include in the spanning tree and which ports to exclude. The root port and all designated ports are included, whereas all blocked ports are excluded. Thus, in the above example, ports 2, 3, and 4 are included in the spanning tree and port 1 is excluded.

   The bridges place all the ports included in the spanning tree in the "Forwarding" state. Data frames are transmitted to and received from ports in the forwarding state. The bridges do not, however, accept data from the blocked ports nor do they transmit data on the blocked ports (this doesn't apply to BPDUs)

   By having all bridges running the same algorithm, a spanning tree will be formed that connects every LAN segment into a loop-free tree. Data frames can, therefore, be transported along the branches of the tree.

3 Address Learning and Frame Forwarding

When a data frame arrives at a port on a bridge, the bridge must determine on which outgoing port the frame should be transmitted. The decision is made using a forwarding database (FDB). The FDB is simply a table containing destination MAC addresses and the appropriate output port that should be used to reach those destinations.

To maintain the FDB and forward frames correctly to the destinations, a bridge will use the following procedure:

- When a bridge receives a frame, the bridge stores the frame's source address in the forwarding database along with the port on which it was received.
- The bridge then looks up the frame's destination address in the forwarding database:

  – If the destination address is found, the bridge forwards the frame on the outgoing port specified in the forwarding database (except when the frame is received from the same port).
  – If the destination address is not found, the bridge forwards the frame on all the outgoing ports except for the one on which the frame was received.

4 Problem Definition

You are required to implement the algorithm running on each bridge; the algorithm is assumed to perform the following three functions:

- Update the state of the ports on a bridge according to the spanning tree algorithm.
- Maintain the forwarding database.
- Forward MAC frames.

You are provided with a header file *bridge.h*, that contains the definition for each of the functions to be implemented, and *bridge.c*, the skeleton code in which to implement the functions. An explanation of the functions is contained within *bridge.h*.

4.1 Functions

The following function definitions and explanations are contained within *bridge.h*:

```
1. void *AllocateBridge(...);
2. void InitializeBridge(...);
3. int Execute(...);
4. PortStateType GetPortState(...);
5. int GetRootPort(...);
6. MACFrame *GetConfigBPDUatPort(...)
7. int *GetFDBEntry(...);
8. void BridgeCleanup(...);
```

The following function, used for sending frames, is implemented in the network simulator:

```
void SendMACFrame(...);
```

5 Data Structures

The following data structures are specified for the interface between your code and the simulator:

MACFrame

This structure defines the encoding of a MAC frame. For the purpose of this assignment, this same structure is used to send data frames and BPDUs. This data structure should NOT be modified as it is used by the network simulator.

The `MACFrame` structure is as follows:

```
typedef struct
{
    int dstAddress;
    int srcAddress;
    int length;
    int DSAP;
    int SSAP;
    int data[FIELDS];
}MACFrame;
```

For the data frames, the `dstAddress` field can either be station address or the `BROADCAST` address. For the BPDUs, however, the `dstAddress` field must be `ALLBRIDGES`.

The `length` field is set by a bridge for BPDUs, and a sending station for data frames. This field has to be a valid number in an ethernet LAN, and indicates the size in bytes of the frame, as it would be in reality.

`DSAP` and `SSAP` are the destination and source service access points respectively, and should both be set to the `BRIDGEPROTOCOL` value in BPDUs.

The `data` field contains the appropriate information, depending on the frame type. See *bridge.h* for a complete description.

MACQueueElement

This structure defines the encoding of an element in the receive queue of a bridge. It contains a pointer to a received frame, the global Port ID of the port where the frame was received, and a pointer to the next element in the queue. When traversing the queue, follow the pointer to the `next` element; the last element in the queue points to a `NULL` `next` element. An empty queue has the value `NULL`.

Note: The simulator frees the memory for `MACQueueElement` structure; do not free it when you are done processing it.

The `MACQueueElement` structure is as follows:

```
typedef struct MACqueue_element
{
    MACFrame *frame;
    struct MACqueue_element *next;
    int globalPortNo;
}MACQueueElement;
```

6 Constants

The following constants are defined. You are to use them where needed.

Addresses

`BROADCAST`	destination address to be placed in a MAC broadcast frame.
`ALLBRIDGES`	destination address to be placed in a BPDU.

Protocol Numbers

`BRIDGEPROTOCOL`	DSAP and SSAP value for the BPDU.
`IP`	DSAP and SSAP value for the IP protocol.

Port States

`Blocking`	port state for blocked port.
`Forwarding`	port state for Designated Port (DP) or Root port (RP).

Other

`FIELDS`	Number of fields in a data frame. This is set to 250 as the simulator uses a static array for the data fields.

7 The Network Simulator

You are provided with a network simulator to which your bridge code interfaces. The simulator uses two input text files: one specifies the network topology and the other specifies a scenario of events (e.g., a station sending a frame) that occur over time.

7.1 Simulator Description

The network simulator performs the following functions.

1. It allows you to define the network topology in terms of bridges, stations, and segments that interconnect them. It implements the station code, simulates LAN transmissions, and interfaces to your bridge code.

2. It is responsible for the transfer of frames between a network entity (bridge or station) to the other entities that are connected to the **same** segment. When a bridge needs to send a BPDU or forward a data frame on one of its ports, it calls the `SendMACFrame(...)` function provided by the network simulator. This function takes care of delivering the frame to the appropriate ports of the recipients on the segment connected to the sending port.

3. It allows you to specify the occurrence of events such as a station sending a data frame, display a bridge's port states or filtering database, etc. in a scenario file. It is responsible for the processing of these events at the specified times. For this assignment, the simulator only allows you to send data frames from the stations at specified times.

4. It calls the `InitializeBridge(...)` function in each bridge to set its Bridge ID, number of ports, etc. Although the standard specifies that configuration BPDUs have to be sent at initialize time, this is not possible in the simulator (`SendMACFrame(...)` cannot be called before all bridges are initialized). Instead, you should send these BPDUs at the first `Execute(...)` call.

5. It performs a round robin call of each bridge's `Execute( )` function, providing it with a queue of frames that it received since last executed. When all bridges have been called, the time is advanced one STEP (a constant defined internally within the simulator). The process is repeated until the time for the end of the simulation that is indicated in the scenario is reached.

To run the simulator, you can use the sample topology and scenario files; these files along with other files can be downloaded from class web site. After compiling your code into the executable bridge (type `make` at the command prompt of a unix-like system), you may run your program using the simulator. To run your code from command line, type *bridge topology.in scenario.in*, where *topology.in* and *scenario.in* are the input topology and scenario files, respectively. If this causes a command not found error then you might need to add the current directory "." to your path; this can be done in the *.bash_profile* or similar file (on UNIX-based systems). At the end of a simulation, the simulator will output the following information:

1. The state of each port in every active bridge at the time the simulation ended.
2. The contents of the filtering database in every bridge.

7.2 Network Topology File

The network topology is described in a text file, (e.g., *topology.in*), that is passed as a command line argument to the simulator. The contents of the input file are given below. Note that the first column shows the commands, while the rest of the columns indicate the appropriate numerical values as obtained from the extended LAN topology.

```
num_of_bridges      number
num_of_segments     number
num_of_stations     number
bridge              bridge#     bridgeID    numPorts
bridge              bridge#     bridgeID    numPorts
...
bridge              bridge#     bridgeID    numPorts
port                MACaddress  portID      bridge#     segment#
port                MACaddress  portID      bridge#     segment#
...
port                MACaddress  portID      bridge#     segment#
station             MACaddress  stationID   segment#
station             MACaddress  stationID   segment#
```

For defining the network topology file, the following rules must be satisfied:

1. The entries on a line are separated by a space character.
2. `num_of_bridges`, `num_of_segments`, and `num_of_stations` are positive integers.
3. `bridge#` lies between 0 and (`num_of_bridges - 1`), both inclusive. Therefore, it is a unique number in the simulation. It is used in simulator function calls and in specifying the attached ports for the bridge.

4. `bridgeID` is a unique non-negative integer that is used as bridge ID in the bridge protocol.

5. `MACaddress` is an integer value that uniquely identifies a bridge port or a station in the topology. Because this value is used in the simulator to index into data structures, we require that the range be [0 to total number of ports on all bridges plus total number of stations minus 1]. In other words, number the ports and stations sequentially starting at 0, without skipping any number. (see example file!)

6. `portID` lies between 0 and (`numPorts`-1), both inclusive, where `numPorts` is the number of ports on that bridge. Port IDs are thus unique within a bridge.

7. `numPorts` is an integer greater than 1.

8. `segment#` lies between 0 and (`num_of_segments`-1), both inclusive. Therefore it is a unique number in the topology. `stationID` lies between 0 and the number of stations you want to specify minus 1. It is used for internal bookkeeping.

As an example of input topology file, consider the extended LAN shown in Figure 2.

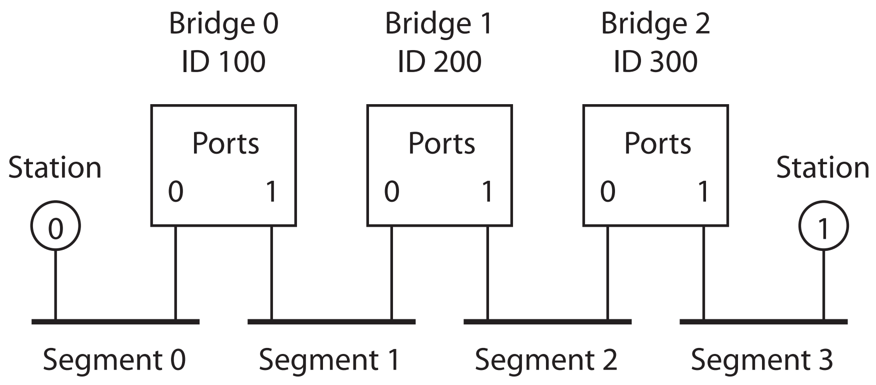

Figure 2: Network topology Example

The network topology file for the network of Figure 2 is given below:

```
num_of_bridges      3
num_of_segments     4
num_of_stations     2
bridge              0   100 2
bridge              1   200 2
bridge              2   300 2
port                0   0   0   0
port            1  1  0  1
port            2  0  1  1
port            3  1  1  2
port            4  0  2  2
port            5  1  2  3
station         6  0  0  0
station         7  1  3  3
```

7.3 Scenario File

The scenario file, which is also passed as a command line argument to the network simulator, contains the possible scenario of events that could happen in the LAN over the length of time you specify. The following list describes the events that can occur during your simulation.

```
end_time time
num_station_sends number
station_send sendingMACaddress sourceMACaddress destinationMACaddress
                                                length time
```

The description of the entries in scenario file is as given below:

`end_time`

The simulation will stop at the time specified by `end_time`. The simulator works in small steps (finite time increments), and stops when the last increment resulted in a time value greater or equal to than `end_time`. This number will specify the number of times Execute(. . .) function is called for each bridge. Inside the simulator, a maximum limit of 1000 is set on this number.

`num_station_sends`

The learning process is facilitated by making stations send a frame at a desired time before the simulation ends. This number specifies the total number of data frames that will be sent by the stations in the simulation. The source and destination addresses and the data are defined subsequently in `station_send` statements.

`station_send`

The source station in this statement attempts to send a data MAC frame, of the `length` specified, to the destination station, originating at the location segment specified at the `time` indicated. In the *scenario.in* file, the `sendingMACaddress` represents the physical station from which the frames are being transmitted, while the `sourceMACaddress` represents the source address that will be placed in the frames. Since in this programming assignment you are not required to handle the situation of stations moving around the LAN, `sendingMACaddress` and `sourceMACaddress` will always be the same. In the simulator, a limit of 100 is set on maximum value of `num_station_sends`.

An example scenario file is shown below:

```
end_time 800
num_station_sends 1
station_send 6 6 7 207 600
```

Finally, given that the processing is performed in finite time steps, you cannot specify arbitrary small times between events that depend on each other. All time values are given in integer units.

8 Deliverables

You are required to submit your source files and a README file that contains:
First Name:
Middle Name:
Last Name:
Student ID:

If you wish, you can also make BRIEF comments about your program in the README file. Your comments, if any, should not exceed a few sentences.

Submission

You have to turn in your *bridge.h*, *bridge.c* and README files. We will require submission of a soft copy only before the deadline. The soft copy must be submitted through the submission process (LMS System).

Comments

Please make sure your submission follows the additional guidelines below.

1. Make sure you remove any debugging `printf` statements in your code.

2. Make sure your code is efficient enough to finish in reasonable time (less than 1 min with the example topology and scenario files provided). A reasonable solution should finish in less than a minute, even if it were to use large statically allocated arrays and brute-force search algorithms, etc.

3. Create new scenarios and topologies, simple and complex, to test your code. The fact that your code correctly works with the example topology and scenario files does not guarantee its correctness. Feel free to use example from class slides.

9 Important Note

This project is meant to be done alone. Absolutely no cooperation is allowed. If there are questions, ask the course staff; they are there to help you. For this project, you must do all thinking, all coding by yourself. Make it a point that you will not discuss the project with anyone else other than the course staff. Do not even discuss with anyone how far you are with this project.

ANY HELP (EXCEPT FROM COURSE STAFF) IS PROHIBITED!!!

Warning: We will use software similarity techniques during grading.

10 More Important Note

We are also giving you the responsibility to **REPORT** all incidences of cooperation. You **MUST** report to the instructor (or the TAs) such incidences. If you do not, we will consider you as guilty as those who you witnessed cooperating. All coding, debugging, web search, etc. must be done by individual students!