CSE 220: Systems Fundamentals I

Stony Brook University

Programming Assignment #4

Spring 2021

Assignment Due: Sunday, April 25th, 11:59 PM (EST)

Change Log:

04/16: Fixed typo in part12 test case
04/17: Fixed the curr_num_nodes field in the test cases of parts 7 - 11
04/18: Corrected part12 friend_of_friend definition
04/22: Part 7 test case 4, curr_num_nodes corrected from 3 to 4

Learning Outcomes

After completion of this programming project you should be able to:

- Design and implement algorithms to manipulate graph-like structures in Assembly

Getting Started

Visit the course website and download MarsSpring2021.jar. From Blackboard, download the starter code (**hw4.zip**) accompanying this document. The archive is available on the course website.

Inside **hw4.zip** you will find **hw4.asm** and a bunch of test files. The file **hw4.asm** contains several function stubs, which you will need to implement. These stubs contain only `jr $ra` instructions. Your job in this assignment is to implement all the functions as specified below. Do not change the function names since the grading scripts will be looking for functions of the given names. However, you may implement additional helper functions of your own, and add them to **hw4.asm**.

If you are having difficulty implementing these functions, write out pseudocode or implement the functions in a higher-level language first. Once you understand the algorithm and what steps to perform, then translate the logic into MIPS assembly code. It may seem like this approach takes more time. But once you have the implementation details pinned down in a higher-level language, then translating it to assembly is less error prone. This will help save time and errors.

Be sure to initialize all of your values (e.g., registers) within your functions. Never assume registers or memory will hold any particular values (e.g., zero). MARS initializes all of the

registers and bytes of main memory to zeroes. The grading scripts will fill the registers and/or main memory with random values before calling your functions.

IMPORTANT: Do not define a `.data` section in your hw4.asm file. A submission that contains a `.data` section will most likely not integrate with our grading script. This may lead to you getting no credit.

Important Information about CSE 220 Programming Projects

- Read this document carefully. If you cannot understand something, ask for clarification . The course staff will try and answer those questions proactively. However, questions whose answers are clearly stated in the documents will be given lowest priority by the course staff.
- You must use the Stony Brook version of MARS posted on the course website. Do not use the version of MARS posted on the official MARS website. The Stony Brook version has a reduced instruction set, added tools, and additional system calls you might need to complete the homework assignments.
- When writing assembly code, try to stay consistent with your formatting and to comment as much as possible. This will make your code readable, which will help us grade your code and also assist you better if you get stuck.
- Do not copy or share code. Your submissions will be checked against other submissions from this semester and from previous semesters.
- Submit to Blackboard by the due date and time. Late work will be penalized as described in the course syllabus. Code that crashes and cannot be graded will earn no credit. No changes to your submission will be permitted once the deadline has passed.

How Your CSE 220 Assignments Will Be Graded

Your programming assignments will be graded almost entirely in an automated way. Grading scripts will execute your code with input values (e.g., command-line arguments, function arguments) and will check for expected results (e.g., print-outs, return values, etc.) For this assignment, your program will be generating output and your functions will be returning values that will be checked for exact matches by the grading scripts. It is your responsibility to output/return the expected values.

Some other items you should be aware of:

- Each test case must execute in 100,000 instructions or fewer. Efficiency is an important aspect of programming. This maximum instruction count will be increased in cases where a complicated algorithm might be necessary, or a large data structure must be traversed. To find the instruction count of your code in MARS, go to the **Tools** menu and

select **Instruction Statistics**. Press the button marked **Connect to MIPS**. Then assemble and run your code as normal.

If you are using the command line to assemble and run your code, you can use the following command:

```
$ java -jar /path/to/MarsSpring2021.jar /path/to.asm --argv <args> --noGui -i -q
```

The path separators will be '\' instead of '/' if you are on a Windows machine.

- Any excess output from your program (debugging notes, etc.) will impact grading. Do not leave erroneous print-outs in your code.
- We will provide you with a small set of test cases for each assignment to give you a sense of how your work will be graded. It is your responsibility to test your code thoroughly by creating your own test cases.
- The testing framework we use for grading your work will not be released, but the test cases and expected results used for testing will be released.

Register Conventions

You must follow the register conventions discussed in lecture and recitation. You can also review the register conventions from this illustrative MIPS register conventions guide by Prof. McDonnell. Failure to follow them will result in loss of credit when we grade your work. Here is a brief summary of the register conventions and how your use of them will impact grading:

- It is the callee's responsibility to save any $s registers it overwrites by saving copies of those registers on the stack and restoring them before returning.
- If a function calls a secondary function, the caller must save $ra before calling the callee. In addition, if the caller wants a particular $a, $t or $v register's value to be preserved across the secondary function call, then place a copy of that register in an $s register before making the function call.
- A function which allocates stack space by adjusting $sp must restore $sp to its original value before returning.
- There is no reason to modify the registers $fp and $gp in this assignment. So, try not to change them. However, if a function modifies one or both, the function must restore them before returning to the caller.

Note that you will lose points, if you do not follow register conventions. Grading scripts will check to see your code follows the conventions.

Unit-Testing Functions

To test your implemented functions, you may use the files ending with `_test.asm` in the starter code (**hw4.zip**) provided to you. You should add your own test code to these files before assembling and running them. The .data section in the test files contain some initial test cases. They are by no means comprehensive. Add more test data to the .data section in the test files to gain comprehensive test coverage.

Each test file contains initial code to help you hook/call into the functions that you are supposed to implement in *hw4.asm*. You will need to add code to call the appropriate functions with appropriate arguments. The astute reader may also write a script to automatically run all the test files with the necessary test cases. This is of course not a requirement and should not be attempted unless you are comfortable with scripting. Your focus should be on first implementing the expected functionality. Your submission will be graded using the examples provided in this document and additional test cases. Do not submit your test files. They will be deleted. We will use different test files for grading.

The Social Network

In this assignment, we will learn how to build and manipulate a custom data structure using MIPS.

Mark Jobs has a brilliant idea to create a network of people in which nodes represent people and edges between nodes represent relationships between people. Mark's goal, through this network, is to capture various properties about people and their relationships. Furthermore, he wants to be able to query the network

- for persons using their names and
- for friends of friends of a person also by name

We need to help Mark build and manage such a network. To this end, we will construct and maintain a data structure called *Network*.

The Network Data Structure

The *Network* data structure uses two other structures called *Node* and *Edge*. In order to understand, *Network*, we first have to understand the structure of *Node* and *Edge*. The *Node* data structure represents a person with a name and defined as follows:

```
struct Node {
   byte[N] name  // null-terminated string of N characters
}
```

The *Edge* data structure represents a relationship between two people. It is defined as follows:

```
struct Edge {
      Node* p1   // reference to a person node
      Node* p2   // reference of a person node
      int friend //attribute to identify relationship as friendship
}
```

Notice that *Edge* does not contain the actual person nodes in a relationship, but a reference to the nodes that are related. The *friend* attribute/property is a positive integer if the relationship is a friendship, and 0 otherwise.

We define *Network* as follows:

```
struct Network {
      int total_num_nodes         // max no. of nodes in the network
      int total_num_edges         // max no. of edges in the network
      int size_of_node            // The size of a node
      int size_of_edge            // The size of an edge
      int curr_num_nodes          // No. of nodes currently in Network
      int curr_num_edges          // No. of edges currently in Network
      char[] name_prop            // Name property; always set to "NAME"
      char[] frnd_prop            // Friend property; always set to
                                  //"FRIEND"
      Node[] nodes                // The set of nodes in the Network
                                  // nodes_capacity =
                                        total_num_nodes * size_of_node
      Edge[] edges                // The set of edges in the Network
                                  // edges_capacity =
                                        Total_num_edges * size_of_edge
}
```

Here is a brief description of the elements in *Network*:

1. `total_num_nodes` is a 4-byte integer that represents the maximum no. of nodes that the network can hold. If this limit is reached, the Network should not accept anymore people.
2. `total_num_edges` is a 4-byte integer that indicates the maximum no. of edges that the network can hold. If the limit is reached, then the Network should not accept anymore relationships.
3. `size_of_node` is a 4-byte integer that indicates the size of a node. Essentially, this attribute represents the maximum length of a person's name. We assume that a person's name will always be a null-terminated string.

4. `size_of_edge` is a 4-byte integer that denotes the size of an edge (i.e., a relationship between two people). This size will always be 12 as an edge is made of 3 attributes each of size 4 -- two person references and one *friend* attribute.
5. `curr_num_nodes` is a 4-byte integer that represents the no. of nodes (people) currently in the network. This number should not be greater than `total_num_nodes`.
6. `curr_num_edges` is a 4-byte integer that represents the no. of edges currently in the network. This number should not be greater than `total_num_edges`.
7. `name_prop` is a null-terminated string always set to “NAME”. This is used to add a name property to a person node. We will see its use later.
8. `frnd_prop` is a null-terminated string always set to “FRIEND”. This is used to add a friend relationship to a relationship between two people. Its use will be evident.
9. `nodes` is a set of nodes (or people) currently in the Network. It is bound by `total_num_nodes`. You can assume that the initial elements in `nodes` will be 0.
10. `edges` is a set of relationships between people currently in the Network. It is bound by `total_num_edges`. You can assume that the initial elements in `edges` will be 0.

As an example, consider the following instantiation of *Network* in MIPS:

```
Network:
  .word 5   #total_nodes (bytes 0 - 3)
  .word 10  #total_edges (bytes 4- 7)
  .word 12  #size_of_node (bytes 8 - 11)
  .word 12  #size_of_edge (bytes 12 - 15)
  .word 0   #curr_num_of_nodes (bytes 16 - 19)
  .word 0   #curr_num_of_edges (bytes 20 - 23)
  .asciiz "NAME" # Name property (bytes 24 - 28)
  .asciiz "FRIEND" # FRIEND property (bytes 29 - 35)
   # set of nodes (bytes 36 - 95)
  .byte 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0
   # set of edges (bytes 96 - 215)
  .word 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
```

In this example, *Network* has 5 nodes and 10 edges. This means that the Network can hold 5 people and 10 relationships. Further, each node can hold exactly 12 characters (including the *null character*). Every element in the set of nodes is a 11-character string indicating the name of a person. Every element in the set of edges is a collection of 3 attributes -- the first two are node (or person) addresses and the third one is the *friend* indicator. The total no. of bytes occupied by *Network* in this example is *4 + 4 + 4 + 4 + 4 + 4 + 5 + 7 + 60 + 120 = 216 bytes*.

Notice that the *set of nodes* is really an array of strings, where the strings are names of people in the Network. In a similar vein, the *set of edges* is really an array of references, where the references are addresses of person nodes in the network. Since both these sets are actually

arrays, we can access every element in them by calculating an offset from the base address of the set (array indices start at 0). The base address of *nodes* and *edges* will be at a fixed location relative to the base address of *Network.*

We will use these data structures to define operations/functions that will help us implement the social network envisioned by Mark Jobs. Your job in this assignment will be to implement each of the defined functions.

Part 1: String Length

```
int str_len(char* str)
```

The `str_len` function takes the base address of a null-terminated string and returns the no. of characters in the string (*excluding* the null character). If the string is empty, then 0 should be returned. A string is empty if it contains *only* the null character.

The function takes the following arguments:

- `str`: the base address of a null-terminated string

Returns in $v0:

- `n>=0`, no. of characters in the string

Additional requirements:

- The function should not change any part of the main memory.

Test Cases:

Function Argument	Expected Return Value ($v0)
“Jane Doe”	8
“Jill Stein”	10
“Ali Tourre”	10

Part 2: String Copy

```
int str_cpy(char* src, char* dest)
```

The function `str_cpy` takes a source address (`src`) and a destination address (`dest`) as arguments and copies all characters in a null-terminated string with base address as `src` to the destination address `dest`. After the copy, the `dest` string must be null-terminated. The function must return the no. of characters (excluding null character) copied.

The function takes the following arguments, in this order:

- `src`: the address of null-terminated string to copy
- `dest`: the base address of the destination string

Returns in $v0:

- `n >= 0`, the no. of characters copied

Additional requirements:

- The function should change only the contents of the `dest` string and no other parts of main memory.

Test Cases:

Argument 1	Argument 2	Expected Return Value ($v0)	Side Effect
“Jane Doe\0”	<Addr1>	8	<Addr1> -> “Jane Doe\0”
“\0”	<Addr1>	0	<Addr1> -> “\0”

Part3: String Equality

```
int str_equals(char* str1, char* str2)
```

The function `str_equals` takes the base address of two strings (both null-terminated) and returns 1 if the strings are equal, i.e., contain the exact same characters. If the strings are not equal, the function returns 0.

The function takes the following arguments, in this order:

- `str1`: the base address of first string being compared
- `str2`: the base address of second string being compared

Returns in $v0:

- `1, if str1 = str2`
- `0, if str1 ≠ str2`

Additional requirements:

- The function should not change main memory.

Test Cases:

Argument 1	Argument 2	Expected Return Value ($v0)
“Jane Doe\0”	“Jane Doe\0”	1
“Jane Doe\0”	“Jane Does\0”	0

Part 4: Create Person

```
Node* create_person(Network* ntwrk)
```

The function `create_person` takes the address of *Network* (as defined above), creates a person node, adds it to the network, and returns the person node. Creating a person involves obtaining a reference (address) to the first free node in the *Network's nodes* set. Once we have the address of such a node, we increment the current no. of nodes in the *Network* by 1 and return the node address.

It is possible that the Network is at capacity, that is, no free nodes are available. In that case, the function should return -1.

The function takes the following arguments:

- `ntwrk`: the base address of *Network*

Returns in $v0:

- `Address of a node in Network or -1`

Additional requirements:

- The function should not change main memory except for what is required.

Test Cases:

Network Layout (base address `ntwrk`)	Argument 1	Expected Return Value ($v0)
Network: .word 5 #total_nodes (bytes 0 - 3) .word 10 #total_edges (bytes 4- 7) .word 12 #size_of_node (bytes 8 - 11) .word 12 #size_of_edge (bytes 12 - 15) .word 0 #curr_num_of_nodes (bytes 16 - 19)	`ntwrk`	`ntwrk + 36`

.word 0 #curr_num_of_edges (bytes 20 - 23) .asciiz "NAME" # Name property (bytes 24 - 28) .asciiz "FRIEND" # FRIEND property (bytes 29 - 35) # nodes (bytes 36 - 95) .byte 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 # set of edges (bytes 96 - 215) .word 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0		
Network: .word 5 #total_nodes (bytes 0 - 3) .word 10 #total_edges (bytes 4- 7) .word 12 #size_of_node (bytes 8 - 11) .word 12 #size_of_edge (bytes 12 - 15) .word 1 #curr_num_of_nodes (bytes 16 - 19) .word 0 #curr_num_of_edges (bytes 20 - 23) .asciiz "NAME" # Name property (bytes 24 - 28) .asciiz "FRIEND" # FRIEND property (bytes 29 - 35) # nodes (bytes 36 - 95) .byte 65 72 89 92 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 # set of edges (bytes 96 - 215) .word 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0	`ntwrk`	`ntwrk + 48`
Network: .word 5 #total_nodes (bytes 0 - 3) .word 10 #total_edges (bytes 4- 7) .word 12 #size_of_node (bytes 8 - 11) .word 12 #size_of_edge (bytes 12 - 15) .word 5 #curr_num_of_nodes (bytes 16 - 19) .word 0 #curr_num_of_edges (bytes 20 - 23) .asciiz "NAME" # Name property (bytes 24 - 28) .asciiz "FRIEND" # FRIEND property (bytes 29 - 35) # nodes (bytes 36 - 95) .byte 65 72 89 92 65 72 89 92 65 72 89 92 65 72 89 92 65 72 89 92 65 72 89 92 65 72 89 92 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 # set of edges (bytes 96 - 215) .word 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0	`ntwrk`	`-1`

Part 5: Check Person In Network

`int is_person_exists(Network* ntwrk, Node* person)`

The function `is_person_exists` verifies if a person exists in the *Network*. It takes a reference to the *Network* structure and a reference to a person node as arguments. It returns 1 if the person node is in *Network's* set of person nodes; otherwise it returns 0.

We say that a person node is in the *Network's* set of nodes if the person node's address is also an address of one of the current nodes in the Network's set of nodes.

The function takes the following arguments, in this order:

- `ntwrk`: the address of *Network*
- `person`: the base of a person node

Returns in $v0:

- `0 or 1`

Additional requirements:

- The function should not change main memory.

Test Cases:

Network Layout (base address `ntwrk`)	Argument 1	Argument 2	Expected Return Value ($v0)
Network: .word 5 #total_nodes (bytes 0 - 3) .word 10 #total_edges (bytes 4- 7) .word 12 #size_of_node (bytes 8 - 11) .word 12 #size_of_edge (bytes 12 - 15) .word 0 #curr_num_of_nodes (bytes 16 - 19) .word 0 #curr_num_of_edges (bytes 20 - 23) .asciiz "NAME" # Name property (bytes 24 - 28) .asciiz "FRIEND" # FRIEND property (bytes 29 - 35) # nodes (bytes 36 - 95) .byte 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 # set of edges (bytes 96 - 215) .word 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0	`ntwrk`	`ntwrk + 36`	0

Network: .word 5 #total_nodes (bytes 0 - 3) .word 10 #total_edges (bytes 4- 7) .word 12 #size_of_node (bytes 8 - 11) .word 12 #size_of_edge (bytes 12 - 15) .word 1 #curr_num_of_nodes (bytes 16 - 19) .word 0 #curr_num_of_edges (bytes 20 - 23) .asciiz "NAME" # Name property (bytes 24 - 28) .asciiz "FRIEND" # FRIEND property (bytes 29 - 35) # nodes (bytes 36 - 95) .byte 65 72 89 92 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 # set of edges (bytes 96 - 215) .word 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0	`ntwrk`	`ntwrk + 36`	1
Network: .word 5 #total_nodes (bytes 0 - 3) .word 10 #total_edges (bytes 4- 7) .word 12 #size_of_node (bytes 8 - 11) .word 12 #size_of_edge (bytes 12 - 15) .word 5 #curr_num_of_nodes (bytes 16 - 19) .word 0 #curr_num_of_edges (bytes 20 - 23) .asciiz "NAME" # Name property (bytes 24 - 28) .asciiz "FRIEND" # FRIEND property (bytes 29 - 35) # nodes (bytes 36 - 95) .byte 65 72 89 92 65 72 89 92 65 72 89 92 65 72 89 92 65 72 89 92 65 72 89 92 65 72 89 92 65 72 89 92 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 # set of edges (bytes 96 - 215) .word 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0	`ntwrk`	`ntwrk + 48`	1

Part 6: Check Person Name In Network

```
int is_person_name_exists(Network* ntwrk, char* name)
```

The function `is_person_name_exists` verifies if a name exists in *Network*. It takes a reference to the *Network* structure and a (null-terminated) string as arguments. It returns 1 if the

any person in Network's *nodes* set has the same name; otherwise it returns 0. Additionally, if a name exists in the Network, then the function also returns a reference to the person with the name.

The function takes the following arguments, in this order:

- `ntwrk`: the address of *Network*
- `name`: base address of a (null-terminated) string

Returns in $v0:

- `0 or 1`

Returns in $v1:

- Reference to a person in *Network* if `$v0 = 1`
- Nothing if `$v0 = 0`.

Additional requirements:

- The function should not change main memory.

Test Cases:

Network Layout (base address `ntwrk`)	Argument 1	Argument 2	Expected Return Value ($v0)	Expected Return Value ($v1)
Network: .word 5 #total_nodes (bytes 0 - 3) .word 10 #total_edges (bytes 4- 7) .word 12 #size_of_node (bytes 8 - 11) .word 12 #size_of_edge (bytes 12 - 15) .word 3 #curr_num_of_nodes (bytes 16 - 19) .word 0 #curr_num_of_edges (bytes 20 - 23) .asciiz "NAME" # Name property (bytes 24 - 28) .asciiz "FRIEND" # FRIEND property (bytes 29 - 35) # nodes (bytes 36 - 95) .byte "Jane Doe\0" ..."John Doe\0"... "Ali Tourre\0" ... 0 0 # set of edges (bytes 96 - 215) .word 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0	`ntwrk`	"John Doe\0"	`1`	`ntwrk + 48`

...				
Network: .word 5 #total_nodes (bytes 0 - 3) .word 10 #total_edges (bytes 4- 7) .word 12 #size_of_node (bytes 8 - 11) .word 12 #size_of_edge (bytes 12 - 15) .word 3 #curr_num_of_nodes (bytes 16 - 19) .word 0 #curr_num_of_edges (bytes 20 - 23) .asciiz "NAME" # Name property (bytes 24 - 28) .asciiz "FRIEND" # FRIEND property (bytes 29 - 35) # nodes (bytes 36 - 95) .byte 65 72 89 92 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 # set of edges (bytes 96 - 215) .word 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0	`ntwrk`	“John Doe\0”	0	`Don’t care`
Network: .word 5 #total_nodes (bytes 0 - 3) .word 10 #total_edges (bytes 4- 7) .word 12 #size_of_node (bytes 8 - 11) .word 12 #size_of_edge (bytes 12 - 15) .word 3 #curr_num_of_nodes (bytes 16 - 19) .word 0 #curr_num_of_edges (bytes 20 - 23) .asciiz "NAME" # Name property (bytes 24 - 28) .asciiz "FRIEND" # FRIEND property (bytes 29 - 35) # nodes (bytes 36 - 95) .byte “Jane Doe\0” ... “John Doe\0” ... “Ali Tourre\0” 0 0 ... 0 0 # set of edges (bytes 96 - 215) .word 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0	`ntwrk`	“Joe\0”	0	`Don’t care`

Part 7: Add A Property To A Person

```
int add_person_property(Network* ntwrk, Node* person, char* prop_name,
char* prop_val)
```

A person may have various properties. One common property is a person's name. This function `add_person_property` sets the name property of an existing person in the *Network*. It takes as input the address of *Network,* the address of the person whose property needs to be set, the property name (null-terminated string), and the property value (null-terminated string) to be added. Since we are primarily interested in a person's name, we will use this function to add the name property of a person. All other properties should be ignored. So, the function should add/set the name of an existing person in the *Network* to string *prop_val* only if

1. `prop_name` is equal to the string "NAME",
2. `person` exists in Network
3. The no. of characters in `prop_val`(excluding null character) < `Network.size_of_node`
4. `prop_val` is unique in the Network.

The function should return `1` if the name property is added successfully to the person. It should return `0` if condition 1 is violated, `-1` if condition 2 is violated, `-2` if condition 3 is violated, and `-3` if condition 4 is violated.

In case of multiple violations, give priority to condition 1, followed by condition 2, then condition 3, and finally condition 4.

The function takes the following arguments, in this order:

- `ntwrk`: the address of *Network*
- `person`: base address of a Node
- `prop_name`: Null-terminated string that indicates the name of the property to be added
- `prop_val`: null-terminated string that indicates the value of the property to be added

Returns in $v0:

- `1,` if property was added successfully
- `0,` if property being added is not "NAME"
- `-1,` if person does not exist
- `-2,` if the property's value is too large
- `-3,` if name being added is not unique

Additional requirements:

- The function should not change main memory except for what is required.

Network Layout (base address `ntwrk`)	Arguments	Expected Return Value ($v0)
Network: .word 5 #total_nodes (bytes 0 - 3)	`ntwrk,` `ntwrk + 36,`	`-1`

.word 10 #total_edges (bytes 4- 7) .word 12 #size_of_node (bytes 8 - 11) .word 12 #size_of_edge (bytes 12 - 15) .word 0 #curr_num_of_nodes (bytes 16 - 19) .word 0 #curr_num_of_edges (bytes 20 - 23) .asciiz "NAME" # Name property (bytes 24 - 28) .asciiz "FRIEND" # FRIEND property (bytes 29 - 35) # nodes (bytes 36 - 95) .byte 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 # set of edges (bytes 96 - 215) .word 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 ...	"NAME\0", "Timmy\0"	
Network: .word 5 #total_nodes (bytes 0 - 3) .word 10 #total_edges (bytes 4- 7) .word 12 #size_of_node (bytes 8 - 11) .word 12 #size_of_edge (bytes 12 - 15) .word 3 #curr_num_of_nodes (bytes 16 - 19) .word 0 #curr_num_of_edges (bytes 20 - 23) .asciiz "NAME" # Name property (bytes 24 - 28) .asciiz "FRIEND" # FRIEND property (bytes 29 - 35) # nodes (bytes 36 - 95) .byte 65 72 89 92 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 # set of edges (bytes 96 - 215) .word 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0	ntwrk, ntwrk + 36, "DOB\0", "Timmy\0"	0
Network: .word 5 #total_nodes (bytes 0 - 3) .word 10 #total_edges (bytes 4- 7) .word 12 #size_of_node (bytes 8 - 11) .word 12 #size_of_edge (bytes 12 - 15) .word 3 #curr_num_of_nodes (bytes 16 - 19) .word 0 #curr_num_of_edges (bytes 20 - 23) .asciiz "NAME" # Name property (bytes 24 - 28) .asciiz "FRIEND" # FRIEND property (bytes 29 - 35) # nodes (bytes 36 - 95) .byte "Jane Doe\0" … "John Doe\0" … "Ali Tourre\0" … 0 0 … 0 0 # set of edges (bytes 96 - 215) .word 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0	ntwrk, ntwrk + 36, "NAME\0", "Timmy Washington\0"	-2

Network:
.word 5 #total_nodes (bytes 0 - 3)
.word 10 #total_edges (bytes 4- 7)
.word 12 #size_of_node (bytes 8 - 11)
.word 12 #size_of_edge (bytes 12 - 15)
.word 4 #curr_num_of_nodes (bytes 16 - 19)
.word 0 #curr_num_of_edges (bytes 20 - 23)
.asciiz "NAME" # Name property (bytes 24 - 28)
.asciiz "FRIEND" # FRIEND property (bytes 29 - 35)
# nodes (bytes 36 - 95) 	
.byte "Timmy\0" … "John Doe\0" … "Ali Tourre\0" … 0 0 … 0 0
# set of edges (bytes 96 - 215) 	
.word 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
	`ntwrk,`
	`ntwrk + 72,`
	`"NAME\0",`
	`"Timmy\0"`
		`-3`

Network:
.word 5 #total_nodes (bytes 0 - 3)
.word 10 #total_edges (bytes 4- 7)
.word 12 #size_of_node (bytes 8 - 11)
.word 12 #size_of_edge (bytes 12 - 15)
.word 4 #curr_num_of_nodes (bytes 16 - 19)
.word 0 #curr_num_of_edges (bytes 20 - 23)
.asciiz "NAME" # Name property (bytes 24 - 28)
.asciiz "FRIEND" # FRIEND property (bytes 29 - 35)
# nodes (bytes 36 - 95) 	
.byte "Timmy\0" … "John Doe\0" … "Ali Tourre\0" 0 0 … 0 0
# set of edges (bytes 96 - 215) 	
.word 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
	`ntwrk,`
	`ntwrk + 72,`
	`"NAME\0",`
	`"Jimmy\0"`
		`1`

The last test case will have a side effect, that is, Network.nodes[3] = "Jimmy\0"

Part 8: Query Network By Person Name

```
Node* get_person(Network* network, char* name)
```

The function `get_person` takes two arguments -- a reference to *Network* and a string indicating a person's name. The function should return a reference (or address) to the person node in *Network* that has its name property set to `name`. If no such person is found, then the function should return 0.

The function takes the following arguments, in this order:

- `ntwrk`: the address of *Network*

- `name`: base address of a null-terminated string

Returns in $v0:

- Address of a person node in *Network*
- `0` if no person found

Additional requirements:

- The function should not change main memory.

Test Cases:

Network Layout (base address `ntwrk`)	Arguments	Expected Return Value ($v0)
Network: .word 5 #total_nodes (bytes 0 - 3) .word 10 #total_edges (bytes 4- 7) .word 12 #size_of_node (bytes 8 - 11) .word 12 #size_of_edge (bytes 12 - 15) .word 0 #curr_num_of_nodes (bytes 16 - 19) .word 0 #curr_num_of_edges (bytes 20 - 23) .asciiz "NAME" # Name property (bytes 24 - 28) .asciiz "FRIEND" # FRIEND property (bytes 29 - 35) # nodes (bytes 36 - 95) .byte 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 # set of edges (bytes 96 - 215) .word 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 ...	`ntwrk,` `"Timmy\0"`	0
Network: .word 5 #total_nodes (bytes 0 - 3) .word 10 #total_edges (bytes 4- 7) .word 12 #size_of_node (bytes 8 - 11) .word 12 #size_of_edge (bytes 12 - 15) .word 3 #curr_num_of_nodes (bytes 16 - 19) .word 0 #curr_num_of_edges (bytes 20 - 23) .asciiz "NAME" # Name property (bytes 24 - 28) .asciiz "FRIEND" # FRIEND property (bytes 29 - 35) # nodes (bytes 36 - 95) .byte "Jane Doe\0" ... "John Doe\0" ... "Ali Tourre\0" 0 0 ... 0 0 # set of edges (bytes 96 - 215) .word 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0	`ntwrk,` `"Timmy Washington\0"`	0

Network: .word 5 #total_nodes (bytes 0 - 3) .word 10 #total_edges (bytes 4- 7) .word 12 #size_of_node (bytes 8 - 11) .word 12 #size_of_edge (bytes 12 - 15) .word 3 #curr_num_of_nodes (bytes 16 - 19) .word 0 #curr_num_of_edges (bytes 20 - 23) .asciiz "NAME" # Name property (bytes 24 - 28) .asciiz "FRIEND" # FRIEND property (bytes 29 - 35) # nodes (bytes 36 - 95) .byte "Timmy\0" ... "John Doe\0" ... "Ali Tourre\0" ... 0 0 ... 0 0 # set of edges (bytes 96 - 215) .word 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0	`ntwrk, "Timmy\0"`	`ntwrk + 36`

Part 9: Verify if People Are Related

```
int is_relation_exists(Network* ntwrk, Node* person1, Node* person2)
```

Two people in the *Network* are related if *Network.edges* contains an edge between *person1* and *person2*. An edge in *Network.edges* is undirected, that is, if there is an edge between *person1* and *person2*, then there is also an edge between *person2* and *person1*.

The function `is_relation_exists` takes a reference to Network (`ntwrk`), and two references to two person nodes (`person1` and `person2`) . If *Network.edges* contains an edge between `person1` and *`person2`*, the function returns `1`; otherwise the function returns `0`. Recall that as per the structure of *Network* defined earlier, *Network.edges is really an array of addresses*.

The function takes the following arguments, in this order:

- `ntwrk`: the address of *Network*
- `person1`: address of a person node
- `Person2`: address of a person node

Returns in $v0:

- 0 or 1

Additional requirements:

- The function should not change main memory.

Test Cases:

Network Layout (base address `ntwrk`)	Arguments	Expected Return Value ($v0)

Network:
.word 5 #total_nodes (bytes 0 - 3)
.word 10 #total_edges (bytes 4- 7)
.word 12 #size_of_node (bytes 8 - 11)
.word 12 #size_of_edge (bytes 12 - 15)
.word 3 #curr_num_of_nodes (bytes 16 - 19)
.word 2 #curr_num_of_edges (bytes 20 - 23)
.asciiz "NAME" # Name property (bytes 24 - 28)
.asciiz "FRIEND" # FRIEND property (bytes 29 - 35)
# nodes (bytes 36 - 95) 	
.byte "Jane Doe\0" … "John Doe\0" … "Ali Tourre\0" …0 0 … 0 0
# set of edges (bytes 96 - 215) 	
.word ntwrk+36 ntwrk+72 0 ntwrk+48 ntwrk+72 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
	`ntwrk,`
	`ntwrk + 84,`
	`ntwrk + 48`
		0

Network:
.word 5 #total_nodes (bytes 0 - 3)
.word 10 #total_edges (bytes 4- 7)
.word 12 #size_of_node (bytes 8 - 11)
.word 12 #size_of_edge (bytes 12 - 15)
.word 4 #curr_num_of_nodes (bytes 16 - 19)
.word 2 #curr_num_of_edges (bytes 20 - 23)
.asciiz "NAME" # Name property (bytes 24 - 28)
.asciiz "FRIEND" # FRIEND property (bytes 29 - 35)
# nodes (bytes 36 - 95) 	
.byte "Timmy\0" … "John Doe\0" … "Ali Tourre\0" … 0 0 … 0 0
# set of edges (bytes 96 - 215) 	
.word ntwrk+36 ntwrk+72 0 ntwrk+48 ntwrk+72 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
	`ntwrk,`
	`ntwrk + 72,`
	`ntwrk + 48`
		1

Part 10: Add Relationship

```
int add_relation(Network* ntwrk, Node* person1, Node* person2)
```

The function `add_relation` takes a reference to *Network* and two references to two person nodes. If both persons exist in the *Network*, then an edge between `person1` and `person2` should be added to *Network.edges*. At this moment, the *friend* property of the edge should be 0. The function returns `1` if the relation was added successfully. It fails to add the relation if:

1. Neither `person1` nor `person2` exists in *Network*
2. The *Network* is at c apacity, that is, it already contains the maximum no. of edges possible
3. A relation between `person1` and `person2` already exists in *Network*. Relations must be unique.
4. `person1` == `person2`. A person cannot be related to herself.

The function should return `0` if failure to add the relation is due to condition 1, `-1` if the failure is due to condition 2, and `-2` if the failure is due to condition 3, and `-3` if the failure is due to condition 4.

In case of multiple failures, condition 1 has highest priority, followed by condition 2, then condition 3, and then condition 4.

The function takes the following arguments, in this order:

- `ntwrk`: the address of *Network*
- `person1`: base address of a Node
- `person2`: base address of a Node

Returns in $v0:

- `1`, if relation was added successfully
- `0`, if either person does not exist in *Network*
- `-1`, if *Network* is at edge capacity
- `-2`, if *Network* contains relation being added
- `-3`, if `person1` and `person2` are the same people

Additional requirements:

- The function should not change main memory except for what is required.

Network Layout (base address `ntwrk`)	Arguments	Expected Return Value ($v0)

Network:
.word 5 #total_nodes (bytes 0 - 3)
.word 10 #total_edges (bytes 4- 7)
.word 12 #size_of_node (bytes 8 - 11)
.word 12 #size_of_edge (bytes 12 - 15)
.word 3 #curr_num_of_nodes (bytes 16 - 19)
.word 2 #curr_num_of_edges (bytes 20 - 23)
.asciiz "NAME" # Name property (bytes 24 - 28)
.asciiz "FRIEND" # FRIEND property (bytes 29 - 35)
	`ntwrk,`
	`ntwrk + 84,`
	`ntwrk + 48`
		0

# nodes (bytes 36 - 95) .byte "Jane Doe\0" … "John Doe\0" … "Ali Tourre\0" … 0 0 … 0 0 # set of edges (bytes 96 - 215) .word ntwrk+36 ntwrk+72 0 ntwrk+48 ntwrk+72 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0		

Network:
.word 5 #total_nodes (bytes 0 - 3)
.word 10 #total_edges (bytes 4- 7)
.word 12 #size_of_node (bytes 8 - 11)
.word 12 #size_of_edge (bytes 12 - 15)
.word 4 #curr_num_of_nodes (bytes 16 - 19)
.word 2 #curr_num_of_edges (bytes 20 - 23)
.asciiz "NAME" # Name property (bytes 24 - 28)
.asciiz "FRIEND" # FRIEND property (bytes 29 - 35)
# nodes (bytes 36 - 95) 	
.byte "Timmy\0" … "John Doe\0" … "Ali Tourre\0" … 0 0 … 0 0
# set of edges (bytes 96 - 215) 	
.word ntwrk+36 ntwrk+72 0 ntwrk+48 ntwrk+72 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
	`ntwrk,`
	`ntwrk + 72,`
	`ntwrk + 48`
		`-2`

Network:
.word 5 #total_nodes (bytes 0 - 3)
.word 10 #total_edges (bytes 4- 7)
.word 12 #size_of_node (bytes 8 - 11)
.word 12 #size_of_edge (bytes 12 - 15)
.word 4 #curr_num_of_nodes (bytes 16 - 19)
.word 2 #curr_num_of_edges (bytes 20 - 23)
.asciiz "NAME" # Name property (bytes 24 - 28)
.asciiz "FRIEND" # FRIEND property (bytes 29 - 35)
# nodes (bytes 36 - 95) 	
.byte "Jane Doe\0" … "John Doe\0" … "Ali Tourre\0" 0 0 … 0 0
# set of edges (bytes 96 - 215) 	
.word ntwrk+36 ntwrk+72 0 ntwrk+48 ntwrk+72 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
	`ntwrk,`
	`ntwrk + 36,`
	`ntwrk + 48`
		`1`

Note: The last test case will insert the relation (`ntwrk + 36,ntwrk + 48`) in `Network.edges`

Part 11: Add Friendship Property

```
int add_relation_property(Network* ntwrk, Node* person1, Node*
person2, char* prop_name, int prop_value)
```

The function `add_relation_property` takes as arguments a reference to *Network*, two references to two person nodes (`person1` and `person2`), a null-terminated string (`prop_name`), and a non-negative integer (`prop_value`). It sets the *friend* property of an existing relation in *Network* to `prop_value`. A positive integer in *friend* property of the relation `person1`, `person2` indicates that `person1` and `person2` are friends. The function returns `1` if the property was added successfully to the relation. It fails to add the property if:

1. A relation between `person1` and `person2` does not exist in *Network*, or
2. `prop_name` is not the string “FRIEND”, or
3. `prop_value < 0`

The function should return `0` if failure to add the property is due to condition 1, `-1` if the failure is due to condition 2, and `-2` if the failure is due to condition 3.

In case of multiple failures, condition 1 has highest priority, followed by condition 2, and then condition 3.

The function takes the following arguments, in this order:

- `ntwrk`: the address of *Network*
- `person1`: base address of a Node
- `person2`: base address of a Node
- `prop_name`: null-terminated string indicating name of property to be added
- `prop_val`: integer indicating value of property to be added

Returns in $v0:

- `1`, if friend property was added successfully
- `0`, if relation does not exist in *Network*
- `-1`, if *prop_name* is not “FRIEND”
- `-2`, if *prop_val < 0*

Additional requirements:

- The function should not change main memory except for what is required.

Test Cases:

Network Layout (base address `ntwrk`)	Arguments	Expected Return Value ($v0)
Network:	`ntwrk, ntwrk + 84,`	`0`

.word 5 #total_nodes (bytes 0 - 3)
.word 10 #total_edges (bytes 4- 7)
.word 12 #size_of_node (bytes 8 - 11)
.word 12 #size_of_edge (bytes 12 - 15)
.word 3 #curr_num_of_nodes (bytes 16 - 19)
.word 2 #curr_num_of_edges (bytes 20 - 23)
.asciiz "NAME" # Name property (bytes 24 - 28)
.asciiz "FRIEND" # FRIEND property (bytes 29 - 35)
# nodes (bytes 36 - 95) 	
.byte "Jane Doe\0" .. "John Doe\0" ... "Ali Tourre\0" 0 0 … 0 0
# set of edges (bytes 96 - 215) 	
.word ntwrk+36 ntwrk+72 0 ntwrk+48 ntwrk+72 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
	`ntwrk + 48,`
	`"FRIEND\0",`
	`1`	

Network:
.word 5 #total_nodes (bytes 0 - 3)
.word 10 #total_edges (bytes 4- 7)
.word 12 #size_of_node (bytes 8 - 11)
.word 12 #size_of_edge (bytes 12 - 15)
.word 3 #curr_num_of_nodes (bytes 16 - 19)
.word 2 #curr_num_of_edges (bytes 20 - 23)
.asciiz "NAME" # Name property (bytes 24 - 28)
.asciiz "FRIEND" # FRIEND property (bytes 29 - 35)
# nodes (bytes 36 - 95) 	
.byte "Timmy\0" ... "John Doe\0" ... "Ali Tourre\0" 0 0 … 0 0
# set of edges (bytes 96 - 215) 	
.word ntwrk+36 ntwrk+72 0 ntwrk+48 ntwrk+72 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
	`ntwrk,`
	`ntwrk + 72,`
	`ntwrk + 48,`
	`"FRIENDS\0",`
	`1`
		`-1`

Network:
.word 5 #total_nodes (bytes 0 - 3)
.word 10 #total_edges (bytes 4- 7)
.word 12 #size_of_node (bytes 8 - 11)
.word 12 #size_of_edge (bytes 12 - 15)
.word 3 #curr_num_of_nodes (bytes 16 - 19)
.word 2 #curr_num_of_edges (bytes 20 - 23)
.asciiz "NAME" # Name property (bytes 24 - 28)
.asciiz "FRIEND" # FRIEND property (bytes 29 - 35)
# nodes (bytes 36 - 95) 	
.byte "Jane Doe\0" ... "John Doe\0" ... "Ali Tourre\0" 0 0 … 0 0
# set of edges (bytes 96 - 215) 	
	`ntwrk,`
	`ntwrk + 72,`
	`ntwrk + 48,`
	`"FRIEND\0",`
	`-1`
		`-2`

.word ntwrk+36 ntwrk+72 0 ntwrk+48 ntwrk+72 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0		

Network:
.word 5 #total_nodes (bytes 0 - 3)
.word 10 #total_edges (bytes 4- 7)
.word 12 #size_of_node (bytes 8 - 11)
.word 12 #size_of_edge (bytes 12 - 15)
.word 4 #curr_num_of_nodes (bytes 16 - 19)
.word 2 #curr_num_of_edges (bytes 20 - 23)
.asciiz "NAME" # Name property (bytes 24 - 28)
.asciiz "FRIEND" # FRIEND property (bytes 29 - 35)
# nodes (bytes 36 - 95) 	
.byte "Jane Doe\0" ... "John Doe\0" ... "Ali Tourre\0" ... 0 0 ... 0 0
# set of edges (bytes 96 - 215) 	
.word ntwrk+36 ntwrk+72 0 ntwrk+48 ntwrk+72 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
	`ntwrk,`
	`ntwrk + 72,`
	`ntwrk + 48,`
	`"FRIEND\0",`
	`1`
		`1`

Note: The last test case will change ntwrk+48 ntwrk+72 0 ***to*** *ntwrk+48 ntwrk+72 1* ***in Network.edges***

Part 12: Verify Friends of Friends

```
int is_friend_of_friend(Network* ntwrk, char* name1, char* name2)
```

We say that **person1** is a *friend-of-friend* of **person3** if **person2** is friend of **person3** and ***person2*** is also friend of **person1** but **person1** is not directly a friend of **person3**.

The function `is_friend_of_friend` takes a reference to *Network* and two null-terminated person names, `name1` and `name2`. The function returns `1` if a person with `name1` is a *friend-of-friend* of a person with `name2` or vice-versa. It returns 0 if a person with `name1` is not friend-of-friend of a person with `name2`. Further, it returns `-1` if neither a person with `name1` nor a person with `name2` exists in Network.

The function takes the following arguments, in this order:

- `ntwrk`: the address of *Network*
- `name1`: null-terminated person name
- `name2`: null-terminated person name

Returns in $v0:

- `0`,`1`, or, `-1`

Additional requirements:

- The function should not change main memory.

Test Cases:

Network Layout (base address `ntwrk`)	Arguments	Expected Return Value ($v0)
Network: .word 5 #total_nodes (bytes 0 - 3) .word 10 #total_edges (bytes 4- 7) .word 12 #size_of_node (bytes 8 - 11) .word 12 #size_of_edge (bytes 12 - 15) .word 3 #curr_num_of_nodes (bytes 16 - 19) .word 2 #curr_num_of_edges (bytes 20 - 23) .asciiz "NAME" # Name property (bytes 24 - 28) .asciiz "FRIEND" # FRIEND property (bytes 29 - 35) # nodes (bytes 36 - 95) .byte "Jane Doe\0" … "John Doe\0" … "Ali Tourre\0" … 0 0 … 0 0 # set of edges (bytes 96 - 215) .word ntwrk+36 ntwrk+60 1 ntwrk+48 ntwrk+60 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0	`ntwrk, "Jasmine\0", "Joel\0"`	`-1`
Network: .word 5 #total_nodes (bytes 0 - 3) .word 10 #total_edges (bytes 4- 7) .word 12 #size_of_node (bytes 8 - 11) .word 12 #size_of_edge (bytes 12 - 15) .word 3 #curr_num_of_nodes (bytes 16 - 19) .word 0 #curr_num_of_edges (bytes 20 - 23) .asciiz "NAME" # Name property (bytes 24 - 28) .asciiz "FRIEND" # FRIEND property (bytes 29 - 35) # nodes (bytes 36 - 95) .byte "Timmy\0" … "John Doe\0" … "Ali Tourre\0" … 0 0 … 0 0 # set of edges (bytes 96 - 215) .word ntwrk+36 ntwrk+60 1 ntwrk+48 ntwrk+60 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0	`ntwrk, "Timmy\0", "Ali Tourre\0"`	`0`
Network: .word 5 #total_nodes (bytes 0 - 3) .word 10 #total_edges (bytes 4- 7) .word 12 #size_of_node (bytes 8 - 11)	`ntwrk, "Timmy\0", "John Doe\0"`	`1`

.word 12 #size_of_edge (bytes 12 - 15) .word 3 #curr_num_of_nodes (bytes 16 - 19) .word 2 #curr_num_of_edges (bytes 20 - 23) .asciiz "NAME" # Name property (bytes 24 - 28) .asciiz "FRIEND" # FRIEND property (bytes 29 - 35) # nodes (bytes 36 - 95) .byte “Timmy\0” … “John Doe\0” ... “Ali Tourre\0” ... 0 0 … 0 0 # set of edges (bytes 96 - 215) .word ntwrk+36 ntwrk+60 1 ntwrk+48 ntwrk+60 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0		

How to Submit Your Work for Grading

Go to **Assignments->Assignment 4** on **Blackboard** and submit **hw4.asm**.

You can submit multiple times. We will grade your most recent submission before the due date.