Your task is to check whether a given node within a trust graph is considered trusted or not, and return true of false accordingly.

A node is considered trusted if it has a trust distance of trustedDistance or less from any pre-trusted peer in the graph. Trust distance is calculated by finding a shortest path between two nodes, measured by summing up all of the edge weights specified in trustGraph. Pre-trusted peers are specified using an array of pre-trusted peer indices in prestrustedPeers.

Notes:

* The trustGraph is represented by an NxN symmetric adjecency matrix where edge weights are represented by positive integers. The lack of an edge is represented with a 0 value.
* The trust graph can contain cycles.
* If node itself is listed in the pretrustedPeers array, then it is trusted.

Inputs and Outputs:

* **[execution time limit] 4 seconds (js)**
* **[input] integer node**

The index of the node that you want to check whether it's trusted or not.

* **[input] array.array.integer trustGraph**

The graph of trust relationships between nodes, represented as an adjacency matrix. This graph is symmetric and can contain loops. A value of 0 indicates no edge between nodes, and a positive value represents the trust edge weight.

* **[input] array.integer pretrustedPeers**

List of pre-trusted peers

* **[input] integer trustThreshold**

The max trust distance from a pre-trusted peer to be considered trusted.

* **[output] boolean**
  + Whether node is considered trusted or not (ie whether it has a trust distance that is less than or equal to trustThreshold, from a node in pretrustedPeers.

Given arrays representing startBalances and pendingTransactions and the integer blockSize, create a blockchain[1] that includes all **valid** pending transactions in the order in which they are given and return the last block.

**Blocks**

Blocks are encoded as strings of the form:  
"blockHash, prevBlockHash, nonce, blockTransactions"

* blockHash: The value returned by sha1(“prevBlockHash, nonce, transactions”)[2], e.g. sha1("0000000000000000000000000000000000000000, 28427, [[0, 1, 5], [1, 2, 5]]").
* prevBlockHash: The blockHash of the previous block. Should be 0000000000000000000000000000000000000000 for the first block.
* nonce: The lowest integer for which the first four characters of blockHash are equal to 0000
* blockTransactions: A string encoded representation of the transactions included in this block. Each individual transaction takes the form [fromAddress, toAddress, value], where fromAddress, toAddress, and value are each integers, e.g. [0, 1, 5].

Each block should have blockSize transactions if there are >= blockSize transactions that have yet to be included in a block. If there are fewer than blockSize transactions remaining, all remaining transactions should be included in the final block.

**Transactions**

A transaction **ti** is valid if the address at from has a balance >= value after processing all transactions **tj** for which j < i. Some transactions in pendingTransactions may be invalid. These transactions should be omitted from all blocks. You can assume that from and to will have entries in startBalances.

**Example**

getLastBlock([5, 0, 0], [[0, 1, 5], [1, 2, 5]], 2) = "00000d03a1ce56a06bfdbceb0249bbb2204a6f22, 0000000000000000000000000000000000000000, 28427, [[0, 1, 5], [1, 2, 5]]"

**Notes**

**[1]** A blockchain is an immutable linked list of ‘blocks’, each containing up to 5 valid transactions. Each block is linked to the previous block via a cryptographic hash rather than a pointer. The global state of each account can be derived by examining the entire chain. More information about the structure and content of a block can be found in the 'Blocks' section.  
**[2]** Below are some examples of how to run sha1 in popular languages, we recommend that you copy paste this code into your solution.

**python**:

import hashlib

def sha1(text):

s = hashlib.sha1()

s.update(text.encode('utf-8'))

return s.hexdigest()

**C++**

#include <openssl/sha.h>

std::string sha1(std::string text) {

unsigned char obuf[20];

SHA1((unsigned char\*)text.c\_str(), strlen((char\*)text.c\_str()), obuf);

char strbuf[40];

for(int j = 0; j < 20; j++) {

sprintf(&strbuf[2\*j], "%02x", obuf[j]);

}

return std::string(strbuf);

}

**Javascript**

var CryptoJS = require("crypto-js");

function sha1(text) {

const hash = CryptoJS.SHA1(text)

return CryptoJS.enc.Hex.stringify(hash);

}

**Java**

String sha1(String text) {

String sha1 = "";

try

{

java.security.MessageDigest crypt = java.security.MessageDigest.getInstance("SHA-1");

crypt.update(text.getBytes("UTF-8"));

Formatter formatter = new Formatter();

for (byte b : crypt.digest()) {

formatter.format("%02x", b);

}

sha1 = formatter.toString();

}

catch(Exception e)

{

e.printStackTrace();

}

return sha1;

}

**Go**

import "crypto/sha1"

import "encoding/hex"

func sha1(text string) string {

h := sha1.New()

io.WriteString(h, text)

return hex.EncodeToString(h.Sum(nil))

}

* **[execution time limit] 4 seconds (js)**
* **[input] array.integer startBalances**

An array representing starting balances. The element with index i and value x initializes the balance of the node with address i to x.

* **[input] array.array.integer pendingTransactions**

A two dimensional array of integers, where each subarray is of the form [fromAddress, toAddress, value]

* **[input] integer blockSize**

An integer specifying the maximum number of transactions that can be included in a block

* **[output] string**
  + A string representing the encoded block, e.g.  
    "00000d03a1ce56a06bfdbceb0249bbb2204a6f22, 0000000000000000000000000000000000000000, 28427, [[0, 1, 5], [1, 2, 5]]"