**Java program:** Prob15.java

**Input File:** Prob15.in.txt

**Output:** Your output needs to be directed to stdout (i.e., using System.out.println())

**Introduction**

Confidentiality, Integrity, and Availability (known as the CIA triad) is a classic security principal in the evaluation or development of computer applications. To ensure sound business practices an application must be outfitted to maximize all three. In this exercise we will concentrate on confidentiality by simulating authenticating users attempting to log in to a system.

These users might include employees, customers, or malicious users whose goal is to steal your information or crash your system. At Lockheed Martin this is an issue that is taken very seriously. You can imagine what a competing company or an enemy nation could do with plans for one of the planes we manufacture. Sound security is essential to keeping our secrets safe.

Your mission here is to secure an application by detecting malicious users and authenticating valid users. You will use the MD5 hashing algorithm to accomplish this. The information in the input file is stored in this format: Username, saltString:digestString

The “salt” is a secure random string stored on the system that will be used to verify the password entered. The “digest” is what you get after using the salt and the password and running them through the hashing algorithm. Here’s how it works:

* First, you will need to create an instance of the MessageDigest class that uses the MD5 hashing algorithm:

MessageDigest md = MessageDigest.getInstance("MD5");

* Next, you will concatenate the salt for the user you are trying to authenticate with the password that they entered. You will need to turn that concatenated string into a byte array and feed it to the MessageDigest instance you just created, then use the hashing algorithm to get the hashed byte array:

byte[] byteArray; // concatenated salt and password

md.update(byteArray);

byte[] hashedBytes = md.digest(); // run the hashing algorithm

* Finally, you will convert the hashed byte array back into a string so you can compare to the digest for that user stored on the system:

StringBufferbuf = new StringBuffer();

for (inti=0; i<hashedBytes.length; i++) {

buf.append(Integer.toHexString((hashedBytes[i] >>> 4) & 0x0F));

buf.append(Integer.toHexString(0x0F &hashedBytes[i]));

}

String hashedString = buf.toString();

**Program Input**

The file Prob15.in.txt will contain:

* A positive integer number (let’s call it N) telling you how many valid username/password combinations are stored in the system
* N lines of user information that is stored on the system. Each line will be for a new user, and will contain a username followed by a comma and a space, followed by a string of seemingly random characters with a colon in the middle. The set of characters to the left of the colon is the salt, and the set of characters to the right of the colon is the digest.
* A blank line to separate user information from login attempts
* Any number of login attempts. Each attempt is on its own line, and will contain a username followed by a comma and a space, followed by the password that the user entered to try to access the system.

**Example Input:**

3

David, lskjuHYgdeBteGdvebtdJhedyCtevDea:dbf71b80ce2b8e482d1d1a4f361c1d63

Dennis, hdT2uHYgdeBteGdvebtdJhedyCtevhia:49793a3df2a5e5a0280fc507036ce522

Fred, hdT2uHYgdeBteGdvebtdJhedyCtevhdt:49793a3df2a5e5a0280fc507036ce823

David, nli2p9uin

Tiffany, ‘ or 1=1; drop tables; --

Eric, pnuaw&972

Dennis, nwliyubvcy92

**Program Output**

Your program should print out each username followed by a space and the results of their login attempt, where the results are one of the following:

* Authorized – The username is stored on the system and the password is correct
* Authorized Denied – The username is stored on the system and the password is incorrect
* Denied – The username is not stored on the system

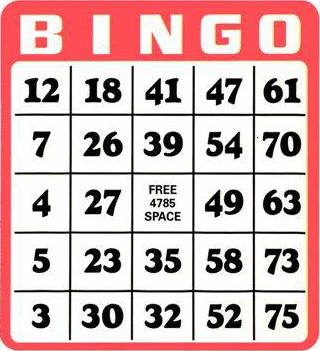
**Example Output:**

David Authorized

Tiffany Denied

Eric Denied

Dennis Authorized Denied

**Java program:** Prob16.java

**Input File:** Prob16.in.txt

**Output:** Your output needs to be directed to stdout (i.e., using System.out.println())

**Introduction**

Mabel loves to play bingo. She needs your help to be able to play multiple cards at the same time. She will provide you with the numbers for each card and the numbers called for a bingo game. You will need to tell Mabel which of her bingo cards are winners. There is no limit to the number of cards Mabel can play and no limit to the number of winning cards.

A Bingo card utilizes the numbers 1 through 75. The five columns of the card are labeled 'B', 'I', 'N', 'G', and 'O' from left to right. The center space is usually marked "Free" or "Free Space", and is considered automatically filled. The range of printed numbers that can appear on the card is normally restricted by column, with the 'B' column only containing numbers between 1 and 15 inclusive, the 'I' column containing only 16 through 30, 'N' containing 31 through 45, 'G' containing 46 through 60, and 'O' containing 61 through 75. The object of the game is to fill 5 consecutive spaces (a whole row, a whole column, or one of the two diagonals). The game ends when a bingo is found on one or more cards.

**Program Input**

The file Prob16.in.txt will contain two sections:

1. The first section will contain the bingo card information. There will be 5 input lines per card. Each line will contain the numbers for one row of a bingo card separated by spaces. Since the free space is in the very middle of the card, the third row of input will have only 4 numbers. The first 5 rows will make up card number 1, the second 5 rows will make up card number 2, and so on.
2. The second section will be the numbers called during the game. The word PLAY will appear on a line by itself to separate the card information from the game simulation. Following the word PLAY each line will contain a value to be played on your bingo cards (i.e. B3, I27, or B5). Remember, the FREE space is always considered to be automatically played.

**Example Input:**

12 27 32 53 75

11 22 34 60 68

3 28 56 62

4 20 39 52 65

13 23 31 49 66

4 26 35 47 71

14 20 40 48 72

1 21 53 65

5 18 44 57 69

6 19 32 52 66

4 28 35 56 70

3 20 43 54 65

2 30 48 71

8 21 36 46 74

5 24 37 50 64

13 29 43 53 65

7 30 33 55 64

14 22 46 68

2 27 44 50 71

4 26 36 59 66

PLAY

B4

N40

O73

G50

B14

I26

I20

B9

O64

G59

G48

N32

G46

**Program Output**

Your program’s output should display the number of each winning card separated by spaces on a single line.

**Example Output:**

3

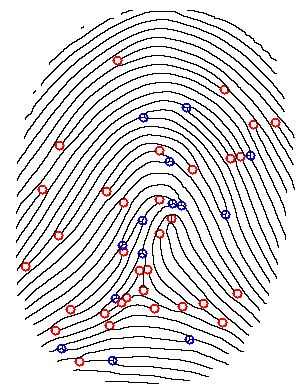
**Java program:** Prob17.java

**Input File:** Prob17.in.txt

**Output:** Your output needs to be directed to stdout (i.e., using System.out.println())

**Introduction**

Fingerprint authentication is one of the most popular and reliable of the many different methods of biometrics. It is used to tell the authenticator what you are (fingerprint) instead of what you know (password) or what you have (badge). A fingerprint consists of a pattern of ridges and valleys that are unique across individuals and do not change throughout our lives. The descriptions of these patterns are known as minutiae and are points on the fingerprint where the ridges split or end. Authentication relies on reading these minutiae and comparing it to the user’s template in a database. Minutia extraction can be done through a series of steps that include binarization, thinning, and minutia detection. Binarization converts the image of the fingerprint into a binary image where a pixel can only be black or white. Thinning is then used to reduce the width of the ridges to a length of one pixel for easier detection of minutiae. Finally, minutiae detection is done by reading each group of pixels and looking for any splitting or ending of the ridges. This problem is concerned with the thinning step of minutia extraction.



The Zhan-Suen algorithm is a popular method for thinning a binarized image. It consists of a series of steps that looks at a pixel’s surrounding pixels and marking it for deletion based on the values of these neighboring pixels. For example, pixel P which is a 1 (black pixel) has 8 surrounding neighbors as seen below.

|  |  |  |
| --- | --- | --- |
| P8 = 0 | P1 = 1 | P2 = 1 |
| P7 = 0 | P = 1 | P3 = 1 |
| P6 = 0 | P5 = 1 | P4 = 1 |

* N(P) equals the number of neighbors for P that are set to 1 (P1-P8).

Ex: P1, P2, P3, P4, P5 = 1. So N(P) = 5

* S(P) equals the number of transitions from 0 to 1 (P1->P2, P2->P3, P3->P4,…, P8->P1).

Ex: P8(0)->P1(1). So S(P) = 1

The algorithm is as follows:

1. For each pixel P set to 1, mark P for deletion if:
   1. 2 ≤ N(P) ≤ 6 AND
   2. S(P) = 1 AND
   3. P1 \* P3 \* P5 = 0 AND
   4. P3 \* P5 \* P7 = 0
2. After all pixels in the input have been checked, delete all pixels marked for deletion by setting them to 0
3. For each pixel P set to 1, mark P for deletion if:
   1. 2 ≤ N(P) ≤ 6 AND
   2. S(P) = 1 AND
   3. P1 \* P3 \* P7 = 0 AND
   4. P1 \* P5 \* P7 = 0
4. After all pixels in the input have been checked, delete all pixels marked for deletion by setting them to 0
5. If any pixels in steps 2 or 4 were deleted, go back to step 1 and read through all pixels again

Hint: Do not delete pixels when iterating through the input pixels in steps 1 and 3. Only mark them for deletion.

Hint: If any neighboring pixels are out of bounds of the array, then those neighboring pixels are set to 0.

Ex: In pixel P atinput\_pixel[0,0], the gray indexes will be 0

|  |  |  |
| --- | --- | --- |
| (-1,-1) | (-1,0) | (-1,1) |
| (0,-1) | (0,0) | (0,1) |
| (1,-1) | (1,0) | (1,1) |

**Program Input**

The file Prob17.in.txt will containa series of 1’s and 0’s representing an image of something. Each 1 or 0 represents a pixel and the value 1 is the black pixel that will be thinned if it meets the above requirements.

**Example Input:**

111111111100000000001111111111

111111111100000000001111111111

111111111100000000001111111111

111111111100000000001111111111

111111111100000000001111111111

111111111100000000001111111111

111111111100000000001111111111

111111111100000000001111111111

111111111100000000001111111111

111111111100000000001111111111

111111111111111111111111111111

111111111111111111111111111111

111111111111111111111111111111

111111111111111111111111111111

111111111111111111111111111111

111111111100000000001111111111

111111111100000000001111111111

111111111100000000001111111111

111111111100000000001111111111

111111111100000000001111111111

111111111100000000001111111111

111111111100000000001111111111

111111111100000000001111111111

111111111100000000001111111111

111111111100000000001111111111

**Program Output**

Your output should be in the same 0 and 1 format as the input with the 1’s thinned from the algorithm.

**Example Output:**

000000000000000000000000000000

000000000000000000000000000000

000000000000000000000000000000

000000000000000000000000000000

000000000000000000000000000000

000010000000000000000000100000

000010000000000000000000100000

000010000000000000000000100000

000010000000000000000000100000

000010000000000000000000100000

000010000000000000000000100000

000010000000000000000000100000

000011111111111111111111100000

000010000000000000000000100000

000010000000000000000000100000

000010000000000000000000100000

000010000000000000000000100000

000010000000000000000000100000

000010000000000000000000100000

000010000000000000000000100000

000000000000000000000000000000

000000000000000000000000000000

000000000000000000000000000000

000000000000000000000000000000

000000000000000000000000000000