**Java program:** Prob01.java

**Input File:** Prob01.in.txt

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

**Introduction**

The ability to read a file is an essential tool in every programmer’s repertoire. Your task is to demonstrate that you’ve mastered this ability by reading the contents of a file and writing the file’s contents to the screen.

**Program Input**

The file Prob01.in.txt will contain some number of lines of text. The length of the lines can vary, but no line will be more than 100,000 characters long.

**Example Input:**

CodeQuest is awesome.

Somet!im@es# I l%ik^e &to\* u(se) e1xt$ra punctuation OR CHARACTERS.

**Program Output**

Your program should print out the input file exactly as it is read.

**Example Output:**

CodeQuest is awesome.

Somet!im@es# I l%ik^e &to\* u(se) e1xt$ra punctuation OR CHARACTERS.

**Java program:** Prob02.java

**Input File:** Prob02.in.txt

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

**Introduction**

The evil queen looks in her magic mirror to see things her way. While you can’t print out text the way it would really look in a mirror, you can reflect the order of a string. Your mission, should you choose to accept it, is to reverse a set of strings.

**Program Input**

The file Prob02.in.txt will contain some number of strings, one per line.

**Example Input:**

Who is the fairest of them all?

Now is the time to come to the aid of your country.

**Program Output**

Your program should output the strings in the same order, but reversed.

**Example Output:**

?lla meht fo tseriaf eht si ohW

.yrtnuoc ruoy fo dia eht ot emoc ot emit eht si woN

**Java program:** Prob03.java

**Input File:** Prob03.in.txt

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

**Introduction**

Often, when working with files, it’s helpful to separate the extension from the filename. This is useful when wanting to perform an action on certain type of file. An example would when you want to move or rename all text files. Your task is to take a list of filenames as input and create a list of extensions along with their count.

**Program Input**

The file Prob03.in.txt will contain a list of filenames with extensions. Each filename will be on its own line.

**Example Input:**

File1.txt

File2.doc

File3.out

File4.txt

File5.so

File6.dll

**Program Output**

The output of the program should be the list of file extensions without the dot, followed by the number of times the extension was found. A single space should be between the extension and the count. The extensions will each be on separate lines and should only be listed once. List the file extensions in the order that you encounter them in the input file.

**Example Output:**

txt 2

doc 1

out 1

so 1

dll 1

**Java program:** Prob04.java

**Input File:** Prob04.in.txt

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

**Introduction**

In this exercise you will simply sort a list of integers.

**Program Input**

The file Prob04.in.txt will contain a comma delimited list of integers, one set per line. You should sort each line of the input file separately.

**Example Input:**

1,5,8,6,7,4,8,5,23,45,85,15476,3548,1,5,2,457

5,4,75,6

8,7,5,987,4815,6873,548,3651536,53

**Program Output**

Your program should print out each line of numbers from least to greatest separated by commas with no spaces.

**Example Output:**

1,1,2,4,5,5,5,6,7,8,8,23,45,85,457,3548,15476

4,5,6,75

5,7,8,53,548,987,4815,6873,3651536

**Java program:** Prob05.java

**Input File:** Prob05.in.txt

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

**Introduction**

Susan works for Widgets and Such in the IT Department. Part of Susan’s job is to remove old software from computers. Susan needs to run a program that can determine from program access logs which pieces of software have not been accessed within the last 180 days. If the software was accessed then it should remain on the system; otherwise that software is eligible to be removed. If a piece of software was accessed exactly 180 days ago, it should remain on the system.

Your task is to write a program that reads software names and access dates and decides which pieces of software to remove from a system based on the rules above.

**Program Input**

The file Prob05.in.txt will contain a reference date that the log files were collected on the first line, followed by a list of software names and access dates separated by a colon. There will be one name-date pair per line.

**Example Input:**

3/21/2013

WordZ:10/18/2012

CalculateThat:1/19/2010

DrawItWithThis:12/19/2012

Software 42:7/23/2012

**Program Output**

Your program’s output should display the software which is eligible to be removed and how many days it’s been since the software was accessed separated by a comma and a space. You should print one software-days pair per line, and your output should be in the same order as the input file.

**Example Output:**

CalculateThat, 1156

Software 42, 241

**Java program:** Prob06.java

**Input File:** Prob06.in.txt

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

**Introduction**

A golf tournament has just completed and everyone is eagerly waiting for the scores to be tabulated so they can eat the free chicken dinner. The crowd is getting restless and the waiters are starting to sweat. Write a program that can compute the scores from today’s round of golf based on the input file containing the golfers’ scores and report back the top three scores and the worst score.

**Program Input**

The file Prob06.in.txt will contain the list of golfers and their scores for the day. No two golfers will have the same name, and you can assume that there will be no ties for first, second, third, or last place. Each line will contain the golfer’s name followed by a colon and a comma delimited list of their scores. There are 18 holes, and each golfer will play all holes.

**Example Input:**

Bob Jones:4,6,3,3,4,3,5,5,4,4,3,3,2,3,4,3,4,4

Ted Smith:3,3,3,3,4,3,5,5,4,3,3,3,2,3,4,3,4,4

Taylor Martin:4,4,3,3,4,3,5,6,4,4,4,3,2,3,5,3,5,4

Adam Lee:4,4,3,3,4,3,5,5,4,3,3,3,2,3,4,3,4,4

Ben Gonzales:4,4,4,3,4,3,5,5,4,3,3,3,2,3,4,3,4,5

Brian Foster:6,5,5,4,4,4,5,5,4,4,4,3,3,3,4,3,5,4

Julian Perez:4,4,4,4,4,4,5,7,4,4,4,4,3,3,4,3,4,4

Mike Davis:3,4,4,4,4,3,5,5,4,3,3,3,2,3,4,3,4,4

Bob Taylor:3,4,3,3,4,3,5,5,4,4,4,4,2,3,4,3,4,4

Chris Matthews:3,3,4,4,4,4,5,5,4,4,4,3,4,3,5,4,5,5

**Program Output**

Your program should output the first, second, third, and last place golfers and scores by outputting the place in all capital letters followed immediately by a colon, and then followed immediately by the player’s name for that place. Remember, in golf the lowest score wins!

**Example Output:**

FIRST:Ted Smith

SECOND:Adam Lee

THIRD:Mike Davis

LAST:Brian Foster

**Java program:** Prob07.java

**Input File:** Prob07.in.txt

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

**Introduction**

Brenda Sue just opened a new cupcake store. Brenda is more interested in baking than handling the money, and wants to make it easier for everyone to give the correct change. Write a program for Brenda Sue that determines the correct amount of change to return to her customers.

**Program Input**

The file Prob07.in.txt will contain a list of amounts customers owe and how much money the customers gave Brenda Sue. There will be one pair of numbers per line, and the numbers will be separated by a comma and a space. Both numbers will always have two decimal places.

**Example Input:**

3.54, 10.00

16.87, 20.00

1.23, 5.00

4.78, 5.03

**Program Output**

Your program’s output should display the amount of money returned to the customer, and a list of the monetary values returned all separated by a comma and a space. An equal sign with no extra space should be between the denomination and the integer number corresponding to how many of that denomination to give to the customer. The change should always be broken down by largest possible denomination; the customer doesn’t want excess change or bills. For example, a quarter should be used instead of 5 nickels. The list of valid denominations is: TWENTY, TEN, FIVE, ONE, QUARTER, DIME, NICKEL, PENNY. If no change is owed, print the word NONE.

**Example Output:**

$6.46, FIVE=1, ONE=1, QUARTER=1, DIME=2, PENNY=1

$3.13, ONE=3, DIME=1, PENNY=3

$3.77, ONE=3, QUARTER=3, PENNY=2

$.25, QUARTER=1

**Java program:** Prob08.java

**Input File:** Prob08.in.txt

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

**Introduction**

In mathematics, a combination refers to a way of selecting some members of a group of things without regard to order. For example, if there were four pieces of fruit on a table: an Apple, an Orange, a Pear, and a Banana, there would be 6 ways to choose 2 fruits: Apple-Orange, Apple-Pear, Apple-Banana, Orange-Pear, Orange-Banana, and Pear-Banana.

In a similar fashion, a fighter pilot can only choose to engage a certain number of enemy targets based on how many weapons their plane has. Your job is to write a program that will compute the number of possible combinations of targets that can be engaged given the total number of enemy combatants and the number of missiles on a plane. The mathematical formula for computing a combination is given by:

Where n is the number of things to choose from (in this case, the number of targets), and k is the number of things to choose (in this case, the number of missiles). The exclamation point denotes a factorial. In case you are unfamiliar with them, a factorial is the product of a positive integer multiplied by all lesser positive integers. For example:

**Program Input**

The file Prob08.in.txt will contain two numbers per line separated by a space. The first number is the number of enemy combatants, and the second number is the number of missiles available. Both numbers will always be integers greater than zero, and the number of combatants will always be greater than the number of missiles.

**Example Input:**

2 1

50 10

**Program Output**

Your program should output the number of possible combinations of targets to engage, one per line.

**Example Output:**

2

10272278170

**Java program:** Prob09.java

**Input File:** Prob09.in.txt

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

**Introduction**

In the classroom, the three R’s are Reading, wRiting, and aRithmetic. In real estate, the three L’s are Location, Location, and Location. Today we are going to learn about the three M’s of statistics. They are the Mean, Median, and Mode. Your job is to write a program to calculate these three values for a given data set.

The Mean of a set of data is simply the average. The Median is the value that is exactly in the middle, if the data set were sorted. In the event that there is an even number of values, the Median is the Mean of the two numbers in the middle of the data set. The Mode is the number (or numbers) that occur the most often in the data set.

**Program Input**

The file Prob09.in.txt will contain sets of integers separated by commas. Each set will be on its own line.

**Example Input:**

1,5,3,4,2,5

2,1,3,1,2

**Program Output**

Your program should print the set number along with the values of the mean, median, and mode that you calculated for each set. If your answers are not integers, print only one decimal place. If there is more than one mode, sort them low to high and separate the numbers by commas.

**Example Output:**

Set 1: Mean=3.3, Median=3.5, Mode=5

Set 2: Mean=1.8, Median=2, Mode=1,2

**Java program:** Prob10.java

**Input File:** Prob10.in.txt

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

**Introduction**

An Internet Protocol address (IP address) is a numerical label assigned to each device (e.g., computer, printer) participating in a computer network that uses the Internet Protocol for communication.

IP addresses consist of 4 decimal numbers ranging from 0 to 255 separated by periods (this is called dotted decimal notation). Each number is often referred to as an octet because numbers from 0 to 255 can be represented by 8 binary digits (see the binary number system appendix if you need more information on binary numbers).

IP addresses are broken down into 5 classes using the following rules:

* Class A network addresses range from 0.0.0.0 to 127. 255. 255. 255
* Class B network addresses range from 128.0.0.0 to 191. 255. 255. 255
* Class C network addresses range from 192.0.0.0 to 223. 255. 255. 255
* Class D network addresses range from 224.0.0.0 to 239. 255. 255. 255
* Class E network addresses range from 240.0.0.0 to 255. 255. 255. 255

Your task is to create a program that will classify IP addresses and convert them from 32 bit binary notation to dotted decimal notation.

**Program Input**

The file Prob10.in.txt will contain a series of 32 bit binary numbers, one per line.

**Example Input:**

01111111000000000000000000000001

11011010110111101010101101010101

10011010110111101010101101010101

10111010110111101010101101010101

11100000111111111111111111111111

11111111111111111111111111111111

**Program Output**

Your program must print the IP address in dotted decimal notation followed by a space and the classification of the IP address in the form [CLASS X].

**Example Output:**

127.0.0.1 [CLASS A]

218.222.171.85 [CLASS C]

154.222.171.85 [CLASS B]

186.222.171.85 [CLASS B]

224.255.255.255 [CLASS D]

255.255.255.255 [CLASS E]

**Java program:** Prob11.java

**Input File:** Prob11.in.txt

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

**Introduction**

Camel Case is a naming style common in many programming languages. In Java, method and variable names typically start with a lowercase letter, with all subsequent words starting with a capital letter (example: startThread). Names of classes follow the same pattern, except that they start with a capital letter (example: BlueCar).

Your task is to write a program that creates or splits Camel Case variable, method, and class names.

**Program Input**

The file Prob11.in.txt will contain a list of operations, object types, and words that you will need to operate on. The pattern will be: Operation;ObjectType;Words. Here is an explanation of the different elements of the input file:

* The operation will be either S (for split) or C (for combine).
* The Object Type will be M (for method), C (for class), or V (for variable)
* In the case of a split operation, the words will be a camel case method, class, or variable name that you need to split into a space-delimited list of words starting with lowercase letters.
* In the case of a combine operation, the words will be a space-delimited list of words starting with lowercase letters that you need to combine into the appropriate camel case string. Methods should end with an empty set of parentheses to differentiate them from variable names.

**Example Input:**

S;M;plasticCup()

C;V;mobile phone

C;C;coffee machine

S;C;LargeSoftwareBook

C;M;white sheet of paper

S;V;pictureFrame

**Program Output**

Your program’s output should display either the space-delimited list of words (in the case of a split operation) or the appropriate camel case string (in the case of a combine operation).

**Example Output:**

plastic cup

mobilePhone

CoffeeMachine

large software book

whiteSheetOfPaper()

picture frame

**Java program:** Prob12.java

**Input File:** Prob12.in.txt

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

**Introduction**

Extensible Markup Language (XML) is a markup language that defines a set of rules for encoding documents in a format that is both human-readable and machine-readable. It is used heavily in the IT world to pass data around from system to system.

XML looks very much like HTML in that XML elements are encased in a set of less than and greater than signs. End tags are similar but have a slash at the beginning of the element name. Here is an example:

<MyElement>This is the text inside of a MyElement element!</MyElement>

XML tags can be named whatever you want, and they can be nested (which is what starts to unleash the true power of XML). Your job is to look through an XML file and list the element names that you find in order and the number of times that element name exists in the XML file. Remember - do not count the ending tags, only the starting ones!

**Program Input**

The file Prob12.in.txt will contain a snippet of XML. There will be no self-ending elements (i.e. <ThisIsSelfEnding />). Elements may or may not have any content.

**Example Input:**

<XML>

<MoviesILike>

<Movie>

<Title>Back to the Future</Title>

<Year>1982</Year>

</Movie>

<Movie>

<Title>Iron Man</Title>

</Movie>

</MoviesILike>

<MoviesIDoNotLike>

<Movie>

<Title>Steel Magnolias</Title>

</Movie>

<Movie>

<Title>Grease</Title>

</Movie>

</MoviesIDoNotLike>

</XML>

**Program Output**

Your program should print out the names of every XML element in the file in the order that you come across them, along with the number of times that element exists in the file separated by a space.

**Example Output:**

XML 1

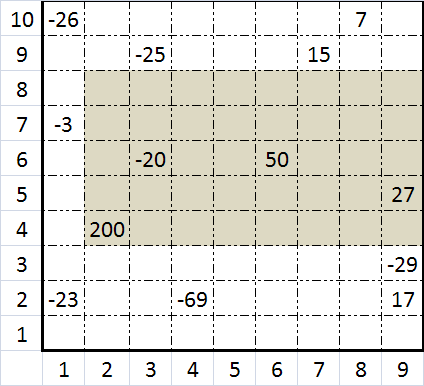
MoviesILike 1

Movie 4

Title 4

Year 1

MoviesIDoNotLike 1

**Java program:** Prob13.java

**Input File:** Prob13.in.txt

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

**Introduction**

Two friends have been using instant messenger and have also been watching too many spy movies. They are afraid that people will be able to listen in on their conversation, so they use steganography to hide what they are talking about. The friends send each other YouTube links that follow this pattern:

https://www.youtube.com/watch?v=UYU8Q3

This looks like any normal YouTube link to the untrained eye, but the text after the “v=” is actually an encoded message. Your job is to write a program that will decode a conversation encoded using these YouTube links. The encoding is done using the following table:

|  |  |  |  |
| --- | --- | --- | --- |
| Value | Encoding | Value | Encoding |
| 0 | A | **8** | Y |
| 1 | B | **9** | 3 |
| 2 | X | **10** | R |
| 3 | 1 | **11** | G |
| 4 | 5 | **12** | K |
| 5 | 8 | **13** | 4 |
| 6 | U | **14** | E |
| 7 | Q | **15** | 7 |

Every character in normal text is split into two encoded characters as follows:

1. Find the ASCII value of the character (for example, the letter h has an ASCII value of 104, or 01101000).
2. Use the integer value of the first 4 bits to get the first encoded character (for h, the first 4 bits are 0110, or 6, which maps to U from the table).
3. Do the same for the second 4 bits (for h, the second 4 bits are 1000, or 8, which maps to Y from the table).
4. Put these encoded characters together and repeat the process for each character in the string you are encoding (so h would map to UY).

Your program should reverse this process to find the actual text from the encoded link parameters.

**Program Input**

The file Prob13.in.txt will contain a conversation between two or more people using the fake links containing the encodings. Their conversation will only use alphanumeric characters, and special characters such as ‘=’ will not be used in the conversation.

**Example Input:**

Alice: https://www.youtube.com/watch?v=UYU8Q3

Bob: https://www.youtube.com/watch?v=QQUYUBQ5XAU3Q1XAU3Q5

Alice: https://www.youtube.com/watch?v=U1UBUEXAQQU8XAU4U8U8Q5

Bob: https://www.youtube.com/watch?v=5E57

Bob: https://www.youtube.com/watch?v=53XAU5U7UEQ5XAUGUEU7QQXAQ3U7Q8

**Program Output**

Your program’s output should display the conversation in the same format as the input but with the decoded values instead of the links.

**Example Output:**

Alice: hey

Bob: what is it

Alice: can we meet

Bob: NO

Bob: I dont know you

**Java program:** Prob14.java

**Input File:** Prob14.in.txt

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

**Introduction**

Word search puzzles have provided a challenging game for all ages since they were first introduced over 40 years ago. They require a keen eye and plenty of patience to complete. Seemingly random letters are aligned in a square or rectangular grid. Generally, they are kept to 30x30 or less but they can be of any size. Given a list of words, the solver must find every listed word in the grid. The words can be found in the following different directions: across, up, down, diagonally, and reversed in any direction. Your task is to create a program that will find the words in the grid.

**Program Input**

The file Prob14.in.txt will contain a grid of uppercase letters. The grid can be up to any size and will not necessarily be square. The line above the grid will have the string “#PUZZLE” without the quotes. Following the grid there will be a list of words, each on a separate line. The line above the beginning word in the list will have the string “#WORDS” without the quotes. Remember that the first row and first column of the grid start at 0, not 1. In the following example, the first letter “J” is at location [0,0], not [1,1].

**Example Input:**

#PUZZLE

JOYQFT

KIRELO

BHJASP

MITEQA

UJHGJI

GTKRUO

#WORDS

THE

JOY

TOP

OUR

**Program Output**

The output of your program should list the words, sorted alphabetically, each followed by the beginning letter’s location in the grid in [row,column] coordinate format. Make sure to place a single space between the word and its location. Each word / location pair should be on its own line. All words should be found. If there are two or more instances of a word found, report the one with the lowest column value coordinate. If two or more instances have the same x value, report the one with the lowest row value coordinate.

**Example Output:**

JOY [0,0]

OUR [5,5]

THE [5,1]

TOP [0,5]

**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:

StringBuffer buf = new StringBuffer();

for (int i=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

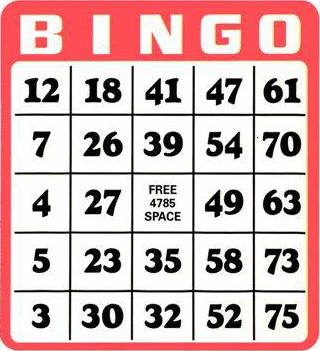
**Example Output:**

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

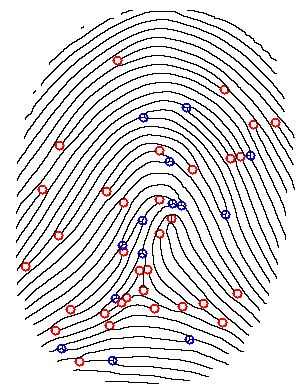
**Java program:** Prob17.java

**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.



Minutiae Detection

Binarization

Thinning

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 at input\_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 contain a 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

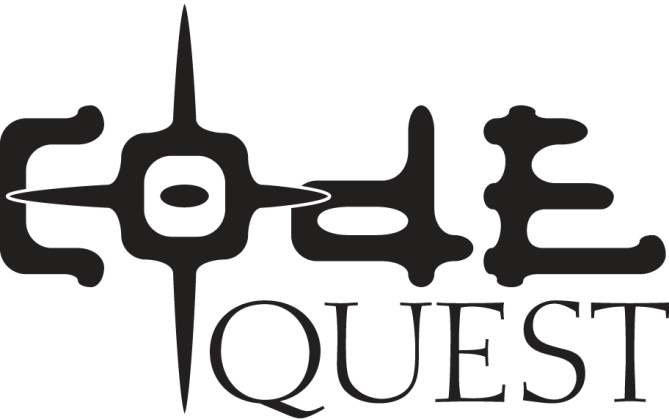
000000000000000000000000000000

000000000000000000000000000000

000000000000000000000000000000

000000000000000000000000000000

000000000000000000000000000000



**2013**

Problem Packet

|  |  |
| --- | --- |
| **Problem** | **Point Value** |
| Problem 1: File I/O | 1 |
| Problem 2: Mirror, Mirror | 2 |
| Problem 3: File Extensions | 3 |
| Problem 4: Sort Numbers | 3 |
| Problem 5: Software Removal | 4 |
| Problem 6: Golf Scores | 4 |
| Problem 7: Making Change | 5 |
| Problem 8: n choose k | 6 |
| Problem 9: The Three M's | 6 |
| Problem 10: IP Addresses | 7 |
| Problem 11: Camel Case | 8 |
| Problem 12: XML | 8 |
| Problem 13: Encode / Decode | 10 |
| Problem 14: Word Search | 10 |
| Problem 15: Authenticate Logins | 11 |
| Problem 16: Bingo! | 14 |
| Problem 17: Thinning | 15 |
| **Total Possible Points** | **117** |