

**2015**

Problem Packet

|  |  |
| --- | --- |
| **Problem** | **Point Value** |
| Problem 1: Monogram | 5 |
| Problem 2: Grains of Sand | 5 |
| Problem 3: Leap Year | 10 |
| Problem 4: Is It Hot in Here? | 10 |
| Problem 5: Money, Money, Money! | 15 |
| Problem 6: Word Worm | 20 |
| Problem 7: Aerospace Intruders | 25 |
| Problem 8: Carrier Landing | 30 |
| Problem 9: Reflection | 35 |
| Problem 10: 40 – Love | 40 |
| Problem 11: Soundex Encoding | 45 |
| Problem 12: Hamming Code | 50 |
| Problem 13: Sky Scraper | 60 |
| Problem 14: CSI | 70 |
| Problem 15: RPN | 80 |
| **Total Possible Points** | **500** |

**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**

Can’t wait for high school graduation? Looking forward to all those gifts? Graduation gifts are always personalized. The monogramed towel set, blanket, ball-point pen, frame…what’s with all the monogramming? If they could only monogram cash! Your task is to write a program that will take a list of full names and output a list of monograms in the correct format.

Monogramming guidelines:

* Every name given will contain a first, middle, and last name separated by a single space. Sometimes an initial will be used instead of a spelled out name.
* Traditional monograms are presented in first, last, middle initial order.
* Monograms should be printed in all caps.

**Program Input**

The first line of the file Prob01.in.txt will contain a positive integer T denoting the number of test cases that follow. Each test case will have the following input:

* The first line of each test case will contain a positive integer N denoting the number of names that follow.
* The next N lines will contain one name per line.

**Example Input:**

2

2

Franklin Delano Roosevelt

h ross perot

4

Dwight David Eisenhower

Warren Gamaliel Harding

Harry S Truman

John Fitzgerald Kennedy

**Program Output**

Your program should print a monogram for each of the given names in the same order they were given.

**Example Output:**

FRD

HPR

DED

WHG

HTS

JKF

**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**

You work for a beach authority who’s decided to count each and every grain of sand on their beaches. Luckily for you, there are other teams doing the counting. Your responsibility is to sum up the counts from each of the teams. Unfortunately, their counts are much larger than a computer can normally handle. You might want to search the Java API for a class called BigInteger.

**Program Input**

The first line of the file Prob02.in.txt will contain a positive integer T denoting the number of test cases that follow. Each test case will have the following input:

* The first line will contain a positive integer N denoting the number of teams counting grains of sand
* The next N lines will contain each team’s count – one per line

**Example Input:**

1

3

23882732998014371873

12198640218946090114

18002450730261047954

**Program Output**

For each test case, your program should output one line containing the sum of sand grains from all the teams.

**Example Output:**

54083823947221509941

**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**

Not all years are created equal. Every now and then (approximately every four years), we get an extra day to enjoy because of a leap year. Leap years occur because we need to keep the calendar year in sync with the astronomical year, so we insert an extra day (February 29th) to correct the difference between the calendars.

Most of the world uses the Gregorian calendar to keep track of the calendar year. The Gregorian calendar was first used in the year 1582, so there were no leap years before that time. Under this calendar system, years that are not leap years are called common years. Here is the pseudo code for telling the difference between common years and leap years:

* If the year is prior to 1582, then it is a common year
* Else if the year is not divisible by 4 then it is a common year
* Else if the year is not divisible by 100 then it is a leap year
* Else if the year is not divisible by 400 then it is a common year
* Else the year is a leap year

Your task is to write a program that will tell whether or not a given year was a leap year.

**Program Input**

The first line of the file Prob03.in.txt will contain a positive integer T denoting the number of test cases that follow. Each test case will have the following input:

* The first line of each test case will contain a positive number N denoting the number of years that follow
* The next N lines will contain a single year per line

**Example Input:**

2

1

1984

3

1999

2001

4000

**Program Output**

Your program should simply print the word Yes if the year was (or will be) a leap year, and No if it was not (or will not be).

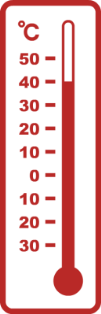
**Example Output:**

Yes

No

No

Yes

**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**

Talking about the weather has been one of the most common small-talk subjects since the beginning of time. “Whew, can you believe this heat wave? It’s been 38 degrees for days!” Wait, what? Anyone who lives in Texas should know better than to complain when the temperature gets above 38 degrees. Or should they? It all depends on the temperature scale!

Here in the United States, we still use the Fahrenheit temperature scale, while most of the rest of the world uses the Celsius scale. Just like talking to people who speak different languages, if we want to be able to talk about the weather with someone who uses a different scale, we need a translator. That’s where you come in!

Your job is to write a temperature translator program. Here is a formula that relates a temperature measured in Fahrenheit (F) to a temperature measured in Celsius (C):

**Program Input**

The first line of the file Prob04.in.txt will contain a positive integer T denoting the number of test cases that follow. Each test case will have the following input:

* The first line of each test case will contain a positive integer N denoting the number of temperature conversions that follow.
* The next N lines will contain a single temperature value in the following format:

<Number> <Scale>

The number and the scale will be separated by a single space. The number could have a decimal point, but will have no more than one digit after the decimal. The scale will either be C or F.

**Example Input:**

2

3

0 C

212 F

50.0 C

4

98.6 F

-6 C

40.1 C

123.4 F

**Program Output**

Your program should convert the temperature to the other scale and print it out for easy reading. Your output lines should follow the following format:

<OriginalNumber> <OriginalScale> = <ConvertedNumber> <NewScale>

Your original number should appear just as it is from the input file, but your converted number should always be printed to the nearest tenth. See Appendix A for more information on how you should round.

**Example Output:**

0 C = 32.0 F

212 F = 100.0 C

50.0 C = 122.0 F

98.6 F = 37.0 C

-6 C = 21.2 F

40.1 C = 104.2 F

123.4 F = 50.8 C

**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**

If a city or a country wants to measure how much income its residents have, one such measure is called

“Per Capita Income”. Per capita income, also known as income per person, is the average income of the people in an economic unit such as a country or city. It is calculated by taking a measure of all sources of income in the aggregate (such as GDP or Gross National Income) and dividing it by the total population.

Besides just knowing the per capita income for a given point in time, it is useful to look at the trend of per capita income over a period of time to diagnose the economic health of an area. This is where you come in. You have been asked to create a program that will graph the per capita income of a region over time.

**Program Input**

The first line of the file Prob05.in.txt will contain a positive integer T denoting the number of test cases that follow. Each test case will have the following input:

* The first line of each test case will contain the name of the region
* The second line of each test case will contain a positive number N denoting the number of data points for the region
* The next N lines will contain two space separated numbers representing a single data point for the region in the following form:

<PER\_CAPITA\_INCOME> <YEAR>

Where PER\_CAPITA\_INCOME is a dollar amount that will have two decimal places and year is a four digit year. Each data point will have a distinct year.

**Example Input:**

1

USA

5

15777.0 1993

28829.0 2013

4141.0 1973

23276.0 2003

9494.0 1983

**Program Output**

Your program should output a horizontal bar graph of the data for each region. The first line of each region’s output should be the name of the region followed by a colon. The next N lines should be the data points for that region sorted ascending by their year. Each line should contain the year followed by a space followed by some number of asterisks. Each asterisk should correspond to $1000, which means you will need to round the per capita income to the nearest thousand dollars. See Appendix A for more information on how you should round.

**Example Output:**

USA:

1973 \*\*\*\*

1983 \*\*\*\*\*\*\*\*\*

1993 \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

2003 \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

2013 \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

**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**

The goal of a word search is simple: find words in a block of letters either up, down, left, right, or diagonally. We have all played this game hundreds of times. But this isn’t your run-of-the-mill word maze. Today you will search for a word worm that bends its way in any direction through a block of letters. The worm may even overlap itself. Don’t let him get away!

Here are the properties of word worms:

* Word worms can move left, right, up, down, and diagonally.
* Word worms can overlap other word worms (even itself).

Here is an example of a word worm spelling the word LOCKHEED:

A **D E** K H E Q

B X **E H K** J R

J I **L O C** K D

R P I G N A H

T E N E F H M

J U O P L N T

As another example, the following could be used to spell the word BANANA:

B A N

**Program Input**

The first line of the file Prob06.in.txt will contain a positive integer T denoting the number of test cases that follow. Each test case will have the following input:

* The first line of each test case will contain two space separated positive integers in the form R C, where R is the number of rows in the puzzle and C is the number of columns in the puzzle.
* The next R lines will each contain C space separated capital letters – this is the puzzle board.
* The next line of each test case will contain a positive integer N denoting the number of words that follow.
* The next N lines of each test case will contain a single word in all capital letters. These are the words to search for in the puzzle.

**Example Input:**

1

6 7

A D E K H E Q

B X E H K J R

J I L O C K D

R P I G N A H

T E N E F H M

J U O P L N T

4

LOCKHEED

PLANE

JET

ENGINE

**Program Output**

Your program should print the list of words that were found in the grid, in the order they were listed in the input file. If a word could not be found, do not print it out.

**Example Output:**

LOCKHEED

JET

ENGINE

**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**

You are patrolling national air-space in an F-22 Raptor and your radar indicates incoming alien robot invaders. Your directive is to destroy each ship starting with the closest and ending with the furthest. Each time you destroy one ship, all remaining ships advance closer to you, but at differing rates. Class-A ships advance 10 X-units, Class-B ships advance 20 X-units, and Class-C ships advance 30 X-units.

For the purposes of this problem you are trying to protect the Y axis, so the closest ship is the one with the lowest X coordinate. In the event of a tie, you should destroy the ship with the largest Y coordinate first. Negative X coordinates are fine – it just means the aliens have invaded your airspace!

**Program Input**

The first line of the file Prob07.in.txt will contain a positive integer T denoting the number of test cases that follow. Each test case will have the following input:

* The first line of each test case will contain a positive integer N denoting the number of alien ships that follow.
* The next N lines will contain a description of an alien ship in the following format:

<ShipName>\_<Class>:<X>,<Y>

The ship name and the class of the ship will be separated by an underscore. The class and the X coordinate will be separated by a colon. The X and Y coordinates will be separated by a comma.

**Example Input:**

2

3

DOOM\_A:123,1444

TEST\_B:12,145

BOGEE\_C:52,345

13

SHIP1\_A:150,150

SHIP2\_B:200,150

SHIP3\_C:165,130

SHIP4\_A:205,135

SHIP5\_B:155,105

SHIP6\_C:195,120

SHIP7\_A:140,50

SHIP8\_B:175,70

SHIP9\_C:215,70

SHIP10\_A:145,10

SHIP11\_B:160,30

SHIP12\_C:185,35

SHIP13\_C:225,20

**Program Output**

Your program should output the data about the ships that it destroys in the order in which it destroys them. The format for each output line should be:

Destroyed Ship: <SHIPNAME> xLoc: <x>

**Example Output:**

Destroyed Ship: TEST xLoc: 12

Destroyed Ship: BOGEE xLoc: 22

Destroyed Ship: DOOM xLoc: 103

Destroyed Ship: SHIP7 xLoc: 140

Destroyed Ship: SHIP3 xLoc: 135

Destroyed Ship: SHIP5 xLoc: 115

Destroyed Ship: SHIP12 xLoc: 95

Destroyed Ship: SHIP6 xLoc: 75

Destroyed Ship: SHIP11 xLoc: 60

Destroyed Ship: SHIP9 xLoc: 35

Destroyed Ship: SHIP13 xLoc: 15

Destroyed Ship: SHIP8 xLoc: 15

Destroyed Ship: SHIP2 xLoc: 20

Destroyed Ship: SHIP10 xLoc: 45

Destroyed Ship: SHIP1 xLoc: 40

Destroyed Ship: SHIP4 xLoc: 85

**Java program:** Prob08.java

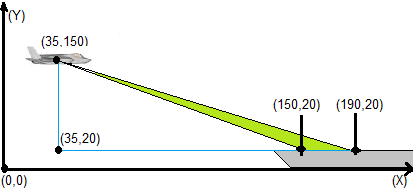
**Input File:** Prob08.in.txt

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

**Introduction**

You are in charge of directing a state-of-the-art, fifth-generation fighter jet to a safe landing aboard the USS Quest Aircraft Carrier. If the aircraft glide-slope is too steep or too shallow, you must indicate to the pilot that he or she must abort the landing and retry. Only when the glide slopes to the targeted landing zone are within tolerances are you to indicate to the pilot that he or she may commit to the landing.

You must calculate the glide slopes and determine if they are within tolerance for a safe landing. If the slope of approach is between -.8 and -1.6 (inclusive), the aircraft is clear to land – otherwise it is waved off. Because you are landing on a carrier, you must consider both the slope between the plane and the front of the landing zone as well as the slope between the plane and the end of the landing zone. Both slopes must be within tolerance for a safe landing.

****

**Program Input**

The first line of the file Prob08.in.txt will contain a positive integer T denoting the number of test cases that follow. Each test case will have the following input:

* The first line of each test case will contain a positive integer N denoting the number of aircraft sections that follow. Each aircraft section will have the following four input lines:
  + The name of the aircraft
  + The X and Y coordinates of the aircraft separated by a comma
  + The X and Y coordinates of the start of the landing zone separated by a comma
  + The X and Y coordinates of the end of the landing zone separated by a comma

**Example Input:**

2

1

ExamplePlane

35,150

150,20

190,20

2

Freebird

25,220

150,20

190,20

Lightning

75,140

110,20

150,20

**Program Output**

Your program should give instructions to the aircraft in the order they were encountered in the input file. For a given aircraft, there are two possible outputs:

<Aircraft Name>, Clear To Land!

Or

<Aircraft Name>, Abort Landing!

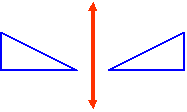
**Example Output:**

ExamplePlane, Clear To Land!

Freebird, Clear To Land!

Lightning, Abort Landing!

**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**

Reflections – you deal with them on a daily basis when you look in the mirror or drive a car. In your math classes, you may have dealt with functions that get reflected. Today, we will be dealing with pictures that will be reflected (sort of).

You will be given a picture built out of ASCII characters, and you will be asked to build the reflection of the picture you are given in one of three ways:

* Around the x axis: for this type of reflection, you will flip the picture up and down. Be careful not to add extra spaces at the ends of lines during this type of reflection.
* Around the y axis: for this type of reflection, you will flip the picture left and right. Since the starting picture will be left justified, your reflected picture should be right justified.
* Around the line y=x: for this type of reflection, the x values and y values are switched. This type of reflection is how you find the inverse of a function in mathematics. For our purposes, the origin will be the top left of the picture.

**Program Input**

The first line of the file Prob09.in.txt will contain a positive integer T denoting the number of test cases that follow. Each test case will have the following input:

* The first line of each test case will be a positive integer N denoting how many lines tall the picture is.
* The next N lines of each test case will contain the picture. The lines might vary in length, so you will need to take this into account when performing the reflection.
* The last line of each test case will contain one of these three strings:
  + X – if you should reflect the picture up and down
  + Y – if you should reflect the picture left and right
  + INVERSE – if you should switch the x and y values

**Example Input:**

3

19

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

!\\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_/!\

!! !! \

!! !! \

!! !! !

!! !! !

!! !! !

!! !! !

!! !! !

!! !! /

!!\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_!! /

!/\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\!/

\_\_\\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_/\_\_/!\_

!\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_!/

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

/oooo oooo oooo oooo /!

/ooooooooooooooooooooooo/ /

/ooooooooooooooooooooooo/ /

/C=\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_/\_/

X

6

\_\_\_\_\_ \_ \_\_\_\_ \_ \_\_\_\_\_ \_ \_

/ \_\_\_\_| | | / \_\_ \ | | | \_\_ \ | | | |

| | \_\_\_ \_\_| | \_\_\_ | | | |\_ \_ \_\_\_ \_\_\_| |\_ | |\_\_) |\_\_\_ \_\_\_| | \_\_\_\_\_| |

| | / \_ \ / \_` |/ \_ \ | | | | | | |/ \_ \/ \_\_| \_\_| | \_ // \_ \ / \_\_| |/ / \_\_| |

| |\_\_\_| (\_) | (\_| | \_\_/ | |\_\_| | |\_| | \_\_/\\_\_ \ |\_ | | \ \ (\_) | (\_\_| <\\_\_ \\_|

\\_\_\_\_\_\\_\_\_/ \\_\_,\_|\\_\_\_| \\_\_\_\\_\\\_\_,\_|\\_\_\_||\_\_\_/\\_\_| |\_| \\_\\_\_\_/ \\_\_\_|\_|\\_\\_\_\_(\_)

Y

8

\_>-. \_\_\_\_ .-<\_

>-.\.-'\_\_\_\_`-./ ,<

.','"\\_\_/"`.`<\_,-,

\_/ / \\_/ \\_/ \! (D\

\ \\_/ \\_\_/ \\_/!\_ (D/

`.`../\_\_\,.'.< `-`

\_>-'/`-.\_\_\_\_.-'\ `<\_

>-' `-<

INVERSE

**Program Output**

Your program should output the appropriate picture after the specified reflection.

**Example Output:**

/C=\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_/\_/

/ooooooooooooooooooooooo/ /

/ooooooooooooooooooooooo/ /

/oooo oooo oooo oooo /!

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

!\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_!/

\_\_\\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_/\_\_/!\_

!/\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\!/

!!\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_!! /

!! !! /

!! !! !

!! !! !

!! !! !

!! !! !

!! !! !

!! !! \

!! !! \

!\\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_/!\

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

\_ \_ \_\_\_\_\_ \_ \_\_\_\_ \_ \_\_\_\_\_

| | | | \ \_\_ | | | \ \_\_ / | | |\_\_\_\_ /

| |\_\_\_\_\_ | |\_\_\_ \_\_\_| )\_\_| | \_| |\_\_\_ \_\_\_ \_ \_| | | | \_\_\_ | |\_\_ \_\_\_ | |

| |\_\_ / /| |\_\_ / \ \_ // \_ | |\_\_ |\_\_ /\ \_ /| | | | | | | \ \_ /| `\_ / \ \_ / | |

|\_\ \_\_\< |\_\_( | )\_( \ \ | | \_| \ \_\_\/\_\_ | |\_| | |\_\_| | /\_\_ | |\_( | )\_( |\_\_\_| |

)\_(\_\_\_\\_\|\_|\_\_\_\ /\_\_\_\\_\ |\_| |\_\_\/\_\_\_||\_\_\_\|\_,\_\_\\\_\\_\_\_\ |\_\_\_\|\_,\_\_\ /\_\_\_\\_\_\_\_\_\

\_ \_

>> \_ >>

-- /\ --

... `''

\'/\./

., \_``

-'\/.-

'"\_ ..

\_\_\/\/\_

\_\_\_ \_\_\_

\_\_\_ \_\_\_

\_\_/\/\\_

`"\_ ,.

-`/\.-

.. \_''

/`\/.\

. <!!< `

-,\_ \_ `-

<<, `<<

\_ -((-\_

,DD`

\/

**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**

Tennis scoring is similar to other games, but it has been cleverly encoded to sound strange to non-tennis fans. Today, you will break this code and write a program to keep track of the score of a tennis game.

**Game, Set, Match**

When two players play each other in tennis, they are playing a tennis match. The overall objective is to win the match. A tennis match is made up of sets, and a set is made up of games. We will not concern ourselves with the match or set scoring today – only a single game.

A tennis game is made up of points. A player wins a tennis game by being the first player to win four points, but you must win by two. There are no tie-breakers in games.

**What’s love got to do with it?**

So far, tennis sounds easy, right? The clever encoding of tennis scores comes at the point level. The following table describes the different names for tennis points:

|  |  |
| --- | --- |
| Number of Points Won | Name of Score |
| 0 | love |
| 1 | 15 |
| 2 | 30 |
| 3 | 40 |

Table 1: Tennis point names

Tennis scores are usually separated by a dash with the server’s score first, and in our games player 1 will always be serving. Some example scores are 15-30 and 40-love.

When the score is tied, tennis does things a little differently. Tie scores of 15-15 and 30-30 are called “15-all” and “30-all” respectively. After this point, when the score is tied at 40-40 or beyond, the score is referred to as “deuce”.

Once a deuce situation is encountered, the score is called out according to which player has the “advantage”, meaning the player that needs to win the next point to win the game. So, after a deuce the only two possible scores would be “Advantage Player 1” or “Advantage Player 2”. Depending on who wins the point after that, either the game is over or the score is tied at deuce again.

**Program Input**

The file Prob10.in.txt will contain an unknown number of lines. Each line will either contain a 1 or a 2, signifying which player won the point. Hint: if you’re using a BufferedReader to get your input (like the Hello World problem does), the readLine method will return null if the end of a file is encountered. So, a statement like while ((inLine = br.readLine()) != null) { might be useful to you.

**Example Input:**

1

1

2

1

1

1

2

1

2

1

2

1

2

2

2

**Program Output**

Your program should print out the score of each game as it progresses. At the beginning of each game, you should print the text “Game start”, and when a game is won you should print “Game Player x”, where x is the number of the player that won. Your program should play as many games as it can until the input runs out.

**Example Output:**

Game start

15-love

30-love

30-15

40-15

Game Player 1

Game start

15-love

15-all

30-15

30-all

40-30

deuce

Advantage Player 1

deuce

Advantage Player 2

Game Player 2

**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**

Soundex is a phonetic hashing algorithm that groups together names that sound similar yet have minor differences in spelling. It can be useful for genealogical studies by identifying variations for a given surname. The hashing part of the algorithm uses the following character groups:

* Group 1: b, f, p, v
* Group 2: c, g, j, k, q, s, x, z
* Group 3: d, t
* Group 4: l
* Group 5: m, n
* Group 6: r
* Wild: h, w
* Vowels: a, e, i, o, u, y

An American Soundex code for a name consists of a letter followed by a three digit number. The code can be determined by the following steps:

1. Find the first letter in the name that belongs to one of the numbered groups. Starting from that letter and working left to right, if two or more letters from the same numbered group are adjacent to one another, remove all but the first letter. Note that h and w are “wild” meaning that they will match letters from any group 1-6.
2. Retain the first letter of the name and remove all vowels and wild letters.
3. Retain the first letter of the name and replace all other letters with their group number.
4. If there are less than three numbers, add zeroes until there are three. If there are more than three numbers, just keep the first three. Make sure the letter at the beginning is capitalized.

Examples:

Ashcroft --> Asroft --> Asrft --> A2613 --> A261

Pfister --> Pister --> Pstr --> P236

Williams --> Wiliams --> Wlms --> W452

**Program Input**

The first line of the file Prob11.in.txt will contain a positive integer T denoting the number of test cases that follow. Each test case will have the following input:

* The first line of each test case will contain a positive integer N denoting the number of names that follow.
* The next N lines will contain one name per line.

**Example Input:**

2

3

williams

ashcroft

pfister

10

Gary

Clare

Jane

Gore

Geier

June

Claire

George

John

Jenny

**Program Output**

Your program should print out the list of Soundex codes generated by the list of names (ordered alphabetically) and the number of times that code was generated separated by a space for each test case. The first line of each test case’s output should be the word OUTPUT.

**Example Output:**

OUTPUT

A261 1

P236 1

W452 1

OUTPUT

C460 2

G600 3

G620 1

J500 4

**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**

When computer data disruption and corruption cannot be tolerated, such as in scientific or financial computing applications, Error-Correcting Code (ECC) memory can be used to limit the impact of electrical and magnetic interference inside of a computer. One of the most popular ECC techniques is implemented through a hamming code. A hamming code includes a number of extra parity bits in precise locations that allow for the detection and correction of errors. The number of parity bits that you need to add depends on the length of the data you are trying to encode. The general algorithm for creating a hamming code is as follows:

1. Number the bits from left to right starting with the number 1.
2. Label the bit positions with their numbers converted to binary:
   1. Bits whose position numbers are powers of 2 (1, 2, 4, etc.) are the parity bits. Notice that these are the positions where there is a single 1 in the binary representation of the bit position number. This is the key to calculating the value of the parity bits.
   2. All other bits are the data bits.
3. Fill in the data bits with the original data.
4. Calculate the value of the parity bits. For this problem, we will be using even parity.
   1. Parity is calculated by adding the value of each checked bit. If the value is even, the parity is 0; if the value is odd, the parity is 1.
   2. Parity bit 1 checks all bit positions which have the least significant bit set to 1 (1, 3, 5…)
   3. Parity bit 2 checks all bit positions which have the second least significant bit set to 1 (2, 3, 6…)
   4. Parity bit 4 checks all bit positions which have the third least significant bit set to 1 (4-7, 12-15…)

**Example hamming code:**

Suppose we had an original input of 1101, and we wanted to create the hamming code for this set of bits. We would do the following:

1. Calculate the number of parity bits we need. Remember that the parity bits are in the bit positions that are a power of two. You can notice from the table below that adding a parity bit at position M gives us an additional M-1 data bits that we can encode. For this example, we have 4 data bits. Parity bit 1 does not give us any data bits. Parity bit 2 gives us one data bit, and parity bit 4 gives us three more. Thus, we need 7 bits total: 3 for parity, and 4 for data.
2. Calculate the values of the parity bits using the algorithm above. The table below shows the bits considered when calculating each parity bit.

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| Bit Position (int): | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
| Bit Position (binary): | 0001 | 0010 | 0011 | 0100 | 0101 | 0110 | 0111 |
| Bit Label: | P1 | P2 | D1 | P4 | D2 | D3 | D4 |
| Original Data: |  |  | 1 |  | 1 | 0 | 1 |
| P1: | 1 |  | 1 |  | 1 | 0 | 1 |
| P2: |  | 0 | 1 |  | 1 | 0 | 1 |
| P4: |  |  | 1 | 0 | 1 | 0 | 1 |
| Encoded Data: | 1 | 0 | 1 | 0 | 1 | 0 | 1 |

Table 1: Hamming Code Example

For each parity bit calculation, the parity bit’s position is highlighted black, and the bits used to calculate it are highlighted grey. Looking at parity bit 2, its position number in binary is 0010, so it only cares about positions that fit the pattern xx1x, where the x could be either a 0 or a 1. Positions 3, 6, and 7 fit this pattern. Two of the three of those data bits are 1, so the parity bit’s value is 0 because there are an even number of 1 values in the data for that parity bit.

**Program Input**

The first line of the file Prob12.in.txt will contain a positive integer T denoting the number of test cases that follow. Each test case will have the following input:

* The first line of each test case will contain a positive integer N denoting the number of binary numbers that follow.
* The next N lines will contain binary numbers, one per line. You will not know the length of each number beforehand.

**Example Input:**

2

1

1101

3

01001101

11011101

10011010

**Program Output**

For each binary number read, your program should output the hamming code for that number.

**Example Output:**

1010101

010010011101

011110111101

011100101010

**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**

Sky scrapers are the tallest man-made structures in the world; the tallest being Burj Khalifa in Dubai. They are expertly designed to stretch high into the sky and resist wind and seismic activity. Your task, should you choose to accept it, is to build the tallest sky scraper possible from a set of bricks. But like any good construction project there are design requirements:

* You will be given a set of rectangular bricks of varying sizes. You can use each brick only once.
* You must stack them on top of one another to build the tallest structure possible.
* You can only stack a brick on top of another brick if the dimensions of the lower brick’s top are greater than or equal to the base of the brick you are stacking on top.
* You can rotate the brick so that any of the sides can function as its base.
* Brick surfaces must stay parallel to the x, y, and z axes, so only 90 degree rotation is possible in any direction.

**NP-Complete**

This problem is an example of an NP-complete type problem. What that means is that there is no slick algorithm that will cut your program’s running time down. It also means that we promise to keep the judging input set small. In our testing, even 50 blocks made us go over the 2 minute run time allotted for problem submissions. The judging data will have a maximum of 25 blocks, and our solution will run for a maximum of 30 seconds. The judging software will allow your program two minutes to run before timing out. Good luck!



**Program Input**

The first line of the file Prob13.in.txt will contain a positive integer T denoting the number of test cases that follow. Each test case will have the following input:

* The first line of each test case will contain a positive integer N denoting the number of blocks.
* The next N lines will contain the description of a single block in the form LxWxH, where L is the block’s length, W is the block’s width, and H is the block’s height.

**Example Input:**

2

4

1x2x3

2x2x4

5x4x1

2x1x2

5

7x7x7

6x6x6

40x40x1

5x5x5

4x4x4

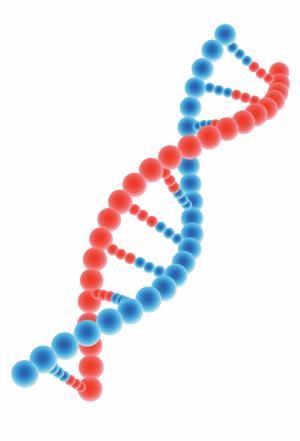
**Program Output**

Your program should print out the height of the tallest tower you can make with the blocks you were given.

**Example Output:**

12

40

**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**

Last month Bobby reported to the police that someone broke into his house and stole his favorite pair of jeans. He was extremely distraught and asked the police to do whatever it takes to find the burglar. The police looked for DNA samples from around the house. Luckily they found only one, and they believe it belongs to the burglar. Unfortunately, the police’s database searching program is on the fritz so they want you to create a program to search their database of all the people in the town and find the closest DNA match.

DNA samples are rarely perfect – parts of a sample could be missing or contaminated. Therefore, a direct match is not usually found. The police use the “longest common subsequence” method to find the most likely suspect. Two strings share a common subsequence if they have the same set of letters in the same order, but the letters in the subsequence do not necessarily have to be adjacent to one another.

For example:

* The string ABCDEFGHIJKLMNOPQRSTUVWXYZ has subsequences KNOT and MOW, but not PAT.
* The longest common subsequence between the strings ABAFCDEF and BCFEDE is BCDE.

**Program Input**

The first line of the file Prob14.in.txt will contain a positive integer T denoting the number of test cases that follow. Each test case will have the following input:

* The first line of each test case will contain a string representing the DNA sequence found by the police.
* The second line of each test case will contain a positive integer N denoting the number of items in the town’s database.
* The next N lines of each test case will contain one item in the town’s DNS database. Each item will be in the form NAME=DNA\_STRING. The DNA sequence letters is made up of the letters A, C, G, and T (A: Ademine, C: Cytosine, G: Guanine, T: Thymine).

**Example Input:**

1

TTTCAGTCTTCGAAACGT

5

David=ATGGCCATCGGGGTCGGCCGTCGCTGGC

Bobby=TTTCAGTCTTCGACGT

Brian=TTGAATGGCGTCTGGCAAACTGGCTT

Jose=TTGACCATGACGTGCCCACTGGC

Kyle=TTGACCAGGGGAATAAACTTTCT

**Program Output**

Your program’s output should display the name of the person whose DNA was the closest match to the sample according to the length of the longest subsequence algorithm. If the length of the longest subsequence is shared by two or more people, output the names in alphabetical order separated by commas.

**Example Output:**

Bobby

**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**

Reverse Polish Notation (RPN, also called postfix notation) is a mathematical notation where the operator (+, -, \*, /, etc.) follows the operands. For example, 2 + 3 in postfix notation is 2 3 +. This notation has the advantage that it’s easy to evaluate in a computer program and it doesn’t require parenthesis to handle order of operations.

Your task is to write a program that will convert expressions from algebraic (also called infix) notation to RPN.

**Some Notes on RPN**

* Operators use the two operands to their left if you’re looking at the final RPN equation. Once used, the result becomes available as an operand for another operation.
  + A + B – C \* D --> A B + C D \* -

In this example, A and B are added together and saved for later. C and D are multiplied together and saved for later. The minus sign at the end acts on both saved values and subtracts the two.

* Operators are used from left to right. Note the difference between these two lines because we know multiplication has to happen before addition. The order of the operators is switched in the second example to account for the order in which the operations must happen.
  + A \* B + C --> A B \* C +
  + A + B \* C --> A B C \* +
* Exponentiation is right-associative (meaning if two exponent operations are next to each other, they happen from right to left instead of left to right):
  + 2 ^ 2 ^ 3 is the same as 2 ^ (2 ^ 3), not (2 ^ 2) ^ 3
* The operands should stay in the same order in your output. For example, if you were given the expression A + B, the answers A B + and B A + are mathematically equivalent because of the commutative property of addition. However, for the purposes of this problem, the only acceptable answer would be A B +.

**Program Input**

The first line of the file Prob15.in.txt will contain a positive integer T denoting the number of test cases that follow. Each test case will have the following input:

* The first line of each test case will contain a positive integer N denoting the number of input lines that follow.
* The next N lines will each contain an expression in regular algebraic notation for you to convert. Each character in the expression will be separated by a space.
* The only non-operand characters you will encounter are PEMDAS characters:
  + ( and ) for grouping
  + ^ for exponentiation
  + \* and / for multiplication and division
  + + and – for addition and subtraction
* If you are unfamiliar with PEMDAS, it is a pneumonic device for remembering the order in which operations are performed:
  + Parentheses first (innermost to outermost)
  + Exponentiation second
  + Multiplication and Division next (they have the same precedence)
  + Addition and Subtraction last (they have the same precedence)

**Example Input:**

2

2

a \* b + c

2 ^ 2 ^ 3

3

( a - b ) \* c / d + e / f

( ( a + b ) - ( c - d ) ^ x ) + q \* w

A + B \* C - D

**Program Output**

Your program should convert the expressions in the order they are read to RPN. All your output characters should be separated by a single space.

**Example Output:**

a b \* c +

2 2 3 ^ ^

a b – c \* d / e f / +

a b + c d - x ^ - q w \* +

A B C \* + D -

**Rounding**

For all Code Quest problems that ask you to round, we will be using the “round to nearest” method, which is what most people consider to be normal rounding. If we were asked to round to the nearest integer, the following results would occur:

* 1.49 would round down to 1
* 1.51 would round up to 2

Because we are rounding to the nearest item, what happens when a number is exactly in the middle? In that case, we will use the “round away from zero” tie breaker, which is also what is generally considered to be normal rounding. Again, if we were rounding to the nearest integer:

* 1.5 would round up to 2
* -1.5 would round down to -2