UIL Computer Science

February 18, 2017

Team Packet

#	Problem Name
1	The Oracle of Delphi
2	AstroNot
3	Space Units
4	Red Rover
5	Optical Constellation Recognition
6	Telemetry
7	Sexagesimal
8	Ring Theory
9	Space Junk
10	Mnemonic
11	Rocket Fuel
12	Hello Moon

Contest Instructions

1. For each problem, read the problem statement thoroughly, making careful note of the problem's required file names and input/output specifications.
2. Write a Java class with a **`main()`** method that meets the stated requirements of the problem.
3. You can test your solution using the sample input supplied with the problem statement. However, be aware that your solution will be judged using more extensive test cases. Design your solution to meet all possible input conditions described by the problem specifications.
4. In the PC^2 application, use the "**`Submit Run`**" tab to upload your Java file to the PC^2 server for judging.
   - Select the problem for which you wish to submit a solution.
   - Select "**`Java`**" as the programming language.
   - Click "**`Select`**" and browse to the Java source code file (i.e., "**`.java`**") that you wish to submit.
   - Click "**`Submit`**" and then "**`Yes`**" to confirm your submission.
   - In a few minutes, you will receive a response from the judges confirming whether your solution was judged to be correct or incorrect. If incorrect, you may correct your code and resubmit.
5. If you are uncertain about the wording or specific requirements of a particular problem, you can use the "**`Request Clarification`**" tab in PC^2 to relay your question to the judges. Responses will be shared with all teams and will be made available using the "**`View Clarifications`**" tab.

Copyright © 2016-2017 Jeff C. Mickel

The Oracle of Delphi — TEAM

Program Name: `Oracle.java` **Input File:** `oracle.dat`

> *"The Pythia"*:
> – Greek mythological figure who served as the oracle of the Temple of Apollo at Delphi.

Periodically, multiple planets within a given solar system will align with one another and their host star. The Pythia star system is quite unique in that many of the alignments between its planets happen to fall along a line directly between Earth and the system's star, Pythia.

Since this odd phenomenon, as astronomically unlikely as it is, complicates the process of identifying exoplanets during their transits across Pythia's surface when viewed from Earth, astronomers would like to know when and how often such planetary alignments occur so that they can better plan and interpret their future observations.

Write a program that, given the time it takes for each exoplanet to complete one orbit around its star, will determine the minimum number of days between one alignment along the Earth/Pythia axis and the next.

By convention, the naming of exoplanets combines the host star name and the order in which its exoplanets are discovered. For example, the first exoplanet discovered around Pythia is called *Pythia b* while the second planet is called *Pythia c*. Each successively discovered exoplanet around Pythia would be given designations *'d'*, *'e'*, *'f'*, etc.

Input

- The first line of input will contain a single integer, `p`, indicating the number of planets to follow.
- Each of the next `p` lines of input contains a positive integer representing the number of Earth days in an exoplanet's orbit. The exoplanets are listed in order of their discovery (i.e., '`b`', '`c`', '`d`', '`e`', etc.).
- The next line of input will contain a single integer, `n`, indicating the number of alignment cases to follow.
- Each alignment case consists of 2 or more space-separated letters ('`b`', '`c`', '`d`', '`e`', etc.) denoting which exoplanets must align for that test case. The exoplanets within a test case may be listed in any order and no exoplanet will be listed twice in the same test case.

Constraints

- `2 <= p <= 10`
- `1 <= n <= 10`

Output

- Print the minimum number of days between alignments of all planets within each test case along the Earth/Pythia axis.

Example Input File

oracle.dat

```
3
128
512
192
3
b c
d b
c b d
```

Example Output To Screen

{System.out}

```
512
384
1536
```

Copyright © 2016-2017 Jeff C. Mickel

AstroNot TEAM

Program Name: `AstroNot.java` **Input File:** `astronot.dat`

Write a program to identify those who did *not* serve as an astronaut aboard a NASA space shuttle flight.

Input

- The first line of input will contain a single integer, `n`, indicating the number of candidate names to follow.
- Each of the next `n` lines consists of a single last name (possibly hyphenated, but containing no spaces).
- The next line of input will contain a single integer, `m`, indicating the number of shuttle missions to follow.
- Each of the next `m` lines consists of a space-separated list of the date, mission designation, and last names of all astronauts aboard the shuttle for the mission.

Constraints

- `1 <= n <= 500`
- `1 <= m <= 135`

Output

- Print the names of each candidate (1 per line) who were not listed as an astronaut aboard any of the shuttle missions contained within the input file. Names should be printed in the order given in the input file.

Example Input File

astronot.dat

```
5
Armstrong
Garriott
Ride
Gagarin
Crippen
23
04/12/1981 STS-1 Young Crippen
11/12/1981 STS-2 Engle Truly
03/22/1982 STS-3 Lousma Fullerton
06/27/1982 STS-4 Mattingly Hartsfield
11/11/1982 STS-5 Brand Overmyer Allen Lenoir
04/04/1983 STS-6 Weitz Bobko Peterson Musgrave
06/18/1983 STS-7 Crippen Hauck Fabian Ride Thagard
08/30/1983 STS-8 Truly Brandenstein Gardner Bluford Thornton
11/28/1983 STS-9 Young Shaw Garriott Parker Merbold Lichtenberg
02/03/1984 STS-41-B Brand Gibson McCandless McNair Stewart
04/06/1984 STS-41-C Crippen Scobee Nelson VanHoften Hart
08/30/1984 STS-41-D Hartsfield Coats Mullane Hawley Resnik Walker
10/05/1984 STS-41-G Crippen McBride Sullivan Ride Leestma Garneau Scully-Power
11/08/1984 STS-51-A Hauck Walker Fisher Gardner Allen
01/24/1985 STS-51-C Mattingly Shriver Onizuka Buchli Payton
04/12/1985 STS-51-D Bobko Williams Seddon Griggs Hoffman Walker Garn
04/29/1985 STS-51-B Overmyer Gregory Lind Thagard Thornton Vandenberg Wang
06/17/1985 STS-51-G Brandenstein Creighton Lucid Fabian Nagel Baudry AlSaud
07/29/1985 STS-51-F Fullerton Bridges Musgrave England Henize Acton Bartoe
08/27/1985 STS-51-I Engle Covey VanHoften Lounge Fisher
10/03/1985 STS-51-J Bobko Grabe Hilmers Stewart Pailes
10/30/1985 STS-61-A Hartsfield Nagel Dunbar Buchli Bluford Furrer Messerschmid Ockels
11/26/1985 STS-61-B Shaw O'Connor Ross Cleave Spring Neri Walker
```

Example Output To Screen

{System.out}

```
Armstrong
Gagarin
```

Copyright © 2016-2017 Jeff C. Mickel

Space Units TEAM

Program Name: `SpaceUnits.java` **Input File:** `spaceunits.dat`

> *"People never do that, you know. They never put the word 'space' in front of something just because everything's all sort of hi-tech and future-y. It's never space restaurant or space champagne or space, you know, hats. It's just restaurants, champagne or hats."* – The Doctor

Following the 1999 loss of the Mars Climate Orbiter, in which the ground-based software provided data in different units than those expected by the orbiter's onboard software, it has been decided that future missions shall standardize all measurements in *Space Units*. Using the conversion table below, write a program that converts SI units into Space Units.

Space Unit Conversion Table

1 meter (m)	=	1.00 space-meters
1 gram (g)	=	1.00 space-grams
1 second (s)	=	1.00 space-seconds
1 newton (N)	=	1.00 space-newtons
1 hertz (Hz)	=	1.00 space-hertz

Input

- The first line of input will contain a single integer, `n`, that indicates the number of measurements to follow.
- Each measurement consists of an integer value, `i`, and an abbreviation (case-sensitive) for one of the SI units shown in the conversion table above.

Constraints

- `1 <= n <= 100`
- `-100,000,000 < i < 100,000,000`

Output

- For each measurement, print the converted values in the appropriate Space Units. All conversions should be displayed to 2 digits of precision.

Example Input File

spaceunits.dat

```
4
-12000 m
30 s
42 N
24 Hz
```

Example Output To Screen

{System.out}

```
-12000.00 space-meters
30.00 space-seconds
42.00 space-newtons
24.00 space-hertz
```

Copyright © 2016-2017 Jeff C. Mickel

Red Rover TEAM

Program Name: `Rover.java` **Input File:** `rover.dat`

The China Aerospace Science and Technology Corporation (CASC) is currently developing an unmanned mission to Mars for 2021 in which they plan to deploy a rover on the Martian surface. In preparation for the mission, the team must first evaluate a number of landing sites for their suitability. Due to the rover's design parameters, the range of terrain that it can explore will be limited by the topography surrounding its landing site. The rover is expected to be able to safely ascend slopes of up to a 33˚ inclination and descend steeper slopes of up to 42˚.

The rover can only move in the 4 cardinal directions (i.e., north, south, east, and west) and only if the gradient between locations does not exceed its safety limits. Write a program than can analyze a topographical region to determine a map of all locations that can be reached from its landing site.

Input

- The first line of input will contain a two integers, `r` and `c`, that indicate the dimensions of the topographical map of the rover's surrounding region. The example input below is 6 rows tall by `10` columns wide.
- The second line of input will contain a two integers, `y` and `x`, that indicate the row (`y`) and column (`x`) number of the rover's landing site. In the example input below, the rover starts at [row: 3, column: 2].
    - Rows and columns are numbered starting with location [row: 0, column: 0] in the northwest (upper-left) corner of the topographical map.
- The next `r` lines of input consist of `c` space-separated, floating-point values, `e`, representing the elevation of the Martian surface (measured in meters) at each location within the topographical map.
    - Rows and columns are spaced at 10-meter intervals across the Martian surface.

Constraints

- `2 <= r <= 100`
- `2 <= c <= 100`
- `-9,000 <= e <= 22,000`

Output

- Display a 2-dimensional representation of the topographical map that indicates each location that the rover can reach with a hash symbol (#) and each location that it cannot reach with a dash (-).

Example Input File

rover.dat

```
6 10
3 2
5123.2 5131.3 5130.2 5125.1 5120.8 5130.9 5133.1 5138.0 5140.2 5146.7
5132.8 5140.2 5129.7 5119.5 5130.2 5133.7 5111.2 5150.7 5144.3 5154.3
5124.3 5137.4 5140.8 5130.3 5133.8 5138.4 5140.7 5157.2 5152.5 5151.5
5127.6 5130.8 5133.6 5138.8 5140.5 5144.2 5137.3 5132.9 5124.7 5122.7
5118.5 5138.1 5140.4 5144.7 5137.7 5132.5 5143.8 5122.2 5127.8 5112.9
5131.7 5144.6 5137.3 5132.0 5122.9 5123.3 5126.5 5113.1 5118.6 5115.2
```

Example Output To Screen

{System.out}

```
-----####-
----##-##-
#--####---
##########
-#####-##-
--##------
```

Copyright © 2016-2017 Jeff C. Mickel

Optical Constellation Recognition TEAM

Program Name: `OCR.java` **Input File:** `ocr.dat`

Given a set of templates describing the compositions of a number of constellations, write a program that can scan a star field to identify these known constellations. Each complete constellation may appear at any location within the star field and may appear rotated 90˚, 180˚, or 270˚ relative to its template. For example, the *Arrow* constellation may appear to point in any of the following directions:

```
Template: ...*.    90˚: .*.    180˚: .*...    270˚: .*.
          *****         .*.          *****          ***
          ...*.         .*.          .*...          .*.
                        ***                         .*.
                        .*.                         .*.
```

While every constellation contains *all* of the stars described by its template, other stars that do not belong to the constellation may also appear within the empty space of a its `h` × `w` region of the star field.

Input

- The first line of input will contain three integers, `n`, `h`, and `w`, indicating the number of constellation templates (`n`) as well has the standard height (`h`) and width (`w`) of each template.
- The next `h` lines of input describe `n` constellations, numbered left-to-right, 1 through `n`.
- Each template consists of an `h` (height) by `w` (width) matrix of individual stars ('`*`') or empty space ('`.`').
- The next line of input will contain 2 integers, `r` and `c`, representing the dimensions of a star field.
- Each star field consists of an `r` (rows) by `c` (columns) matrix of individual stars ('`*`') or empty space ('`.`').

Constraints

- `2 <= n <= 20`
- Template dimensions: `2 <= h <= 10`; `2 <= w <= 10`
- Star field dimensions: `2 <= r <= 100`; `2 <= c <= 100`

Output

- Print the list of constellations that appear completely within the star field (in ascending order by number).

Example Input File *(the matching constellations are shown highlighted here for easier identification)*

ocr.dat

```
6 5 5
*...* *.... ....* .*.*. ..... ..*..
..... *.... ....* *...* *.*.* .*.*.
..*.. ..*.. ..*.. ..... ..... *...*
..... ....* *.... *...* *.*.* .*.*.
*...* ....* *.... .*.*. ..*.. ..*..
8 75
.....*.*.....*.......*................................*....................
.........................................**...........................*....
.....*.**..................*......*................*.......................
...........................*...............*.............*......*..........
.....*.*........*............*.................................*.*.......*.
...........................*...*............**................*.*.*........
......................*........*...............................*.*.........
..........*...........................*.........................*.......*..
```

Example Output To Screen

{System.out}

```
2 3 5 6
```

Copyright © 2016-2017 Jeff C. Mickel

Telemetry TEAM

Program Name: `Telemetry.java` **Input File:** `telemetry.dat`

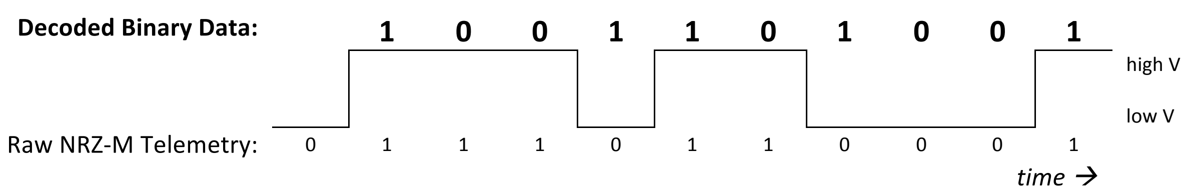
Pulse Code Modulation (PCM) is one of the most common radio telemetry techniques used for the recovery of test data from aerospace vehicles. Its use is complicated by the fact that the transmission of data from a satellite occurs asynchronously with respect to the corresponding ground-base receivers. That is, signals are usually transmitted continuously without a clear beginning or end for receiving stations to lock onto. To account for this, PCM transmissions are packaged as discrete bundles of well-formatted information known as *frames*.

Each frame consists of a *frame sync pulse* followed by a *key-value payload*. The frame sync pulse consists of a fixed and well-known 32-bit pseudorandom sequence that the ground receiver can use to scan through a data stream and identify the start of an individual frame of telemetry data. The remainder of the frame consists of the actual telemetry payload – in this case, a 24-bit key (representing a 3-letter ASCII string) and a 32-bit integer value.

Previous Frame	*Current Frame*			*Next Frame*
	Frame Sync Pulse 32-bits (0xFE6B2840)	*Payload Key 24-bits*	*Payload Value 32-bits*	

time →

The telemetry data stream itself uses *Non-Return-to-Zero Mark (NRZ-M)* encoding on the transmitted signal. With NRZ-M encoding, a "one" bit is represented by a *change* in signal level (either low-to-high or high-to-low) while a "zero" bit is represented by *no change* in signal level. In the example below, the NRZ-M telemetry waveform received by a ground station would decode to a binary data sequence of `1001101001`.



Input

- The input will contain a single line of `'0'` and `'1'` characters representing high (1) and low (0) signals in the NRZ-M telemetry data stream received from an incoming satellite transmission.
- The received input stream contains exactly 1 valid frame surrounded by incomplete frames on either end.
- The 32-bit frame sync pulses for this telemetry transmission are always `0xFE6B2840`.

Constraints

- The NRZ-M encoded input stream will always start with a `'0'` character.

Output

- Print the telemetry payload's key and value encoded within the input stream, separated by an equals sign (=) as shown in the example output below.

Example Input File

telemetry.dat

```
01001001101111010110000101010111011001000110000011111111001111110011000011000
10000000000000000000000000001010000101100001101001000001101010111100000001100
10100000101111100101100100111010101110111100111110111110101011100011100111011
11100011100101011000101001110101101001110111110110110111000100100110001100011
```

Example Output To Screen

{System.out}

```
PTS=60
```

Copyright © 2016-2017 Jeff C. Mickel

Sexagesimal TEAM

Program Name: `Sexagesimal.java` **Input File:** `sexagesimal.dat`

Sexagesimal (literally, base-60) notation is often used to describe angular measurements, such as the *right ascension* (RA) and *declination* (Dec) of a star's position on the celestial sphere. The sexagesimal format uses hours (or degrees), minutes, and seconds to denote increasingly finer levels of precision, with 60 seconds making up a minute, 60 minutes making up an hour or degree, and 24 hours making up 360˚. Each of these base-60 measurements can also be expressed in decimal (base-10) degrees. For example, the following table shows the sexagesimal and decimal equivalents for the celestial coordinates of several stars:

Star	Right Ascension (RA)	Declination (Dec)
Polaris	2h 31m 49s = 37.9542˚	+89˚ 15' 51" = +89.2642˚
Betelgeuse	5h 55m 10s = 88.7917˚	+7˚ 24' 25" = +74069˚
Mintaka	5h 32m 0s = 83.0000˚	–0˚ 17' 57" = –0.2992˚

Write a program to convert a star's sexagesimal coordinates into decimal degrees. *Right ascension* (RA) is measured in hours-minutes-seconds while *declination* (Dec) is measured in degrees-minutes-seconds.

Input

- The first line of input will contain a single integer, `n`, indicating the number of star coordinates to follow.
- Each star consists of a right ascension (RA) and declination (Dec) separated by a space-pipe-space (" | ").
  - The right ascension is given in the form "`RA hh:mm:ss`", where `hh`, `mm`, and `ss` are two-digit, zero-padded integers representing *hours*, *minutes*, and *seconds*, respectively.
  - The declination is given in the form "`Dec ±dd:mm:ss`", where `dd`, `mm`, and `ss` are two-digit, zero-padded integers representing *degrees*, *minutes*, and *seconds*, respectively. The sign is always included.

Constraints

- `1 <= n <= 100`
- Hours: `00 <= hh < 24`
- Degrees: `–90 < dd < 90` (always expressed with a leading *positive* (+) or *negative* (–) sign)
- Minutes: `00 <= mm < 60`
- Seconds: `00 <= ss < 60`

Output

- Print the right ascension (RA) and declination (Dec) for each star, measured in decimal degrees and rounded to 3 decimal places of precision.
- Format the output as shown in the example output below with a space-pipe-space (" | ") separating the right ascension from the declination.

Example Input File

sexagesimal.dat

```
3
RA 02:31:49 | Dec +89:15:51
RA 05:55:10 | Dec +07:24:25
RA 05:32:00 | Dec -00:17:57
```

Example Output To Screen

{System.out}

```
RA 37.954 | Dec +89.264
RA 88.792 | Dec +7.407
RA 83.000 | Dec -0.299
```

Copyright © 2016-2017 Jeff C. Mickel

Ring Theory TEAM

Program Name: `Rings.java` **Input File:** `rings.dat`

Planetary ring systems are derived from the varying density levels of fine ice particles orbiting a planet. Each disc of ice can be subdivided into a series of individual rings and gaps. A density variation of more than 2:1 between any two adjacent measurements is enough to distinguish a boundary between one region and the next. Sparser regions with an average density of less than 10 g/cm^2 are seen as gaps between rings. Each ring is designated by a letter, with the first distinct band being referred to as "*A* Ring", the second as "*B* Ring", etc.

Input

- The first line of input will consist of a single integer, `n`, indicating the number of measurements to follow.
- The following `n` lines represent a series of individual density measurements extending through a cross-section of the planet's ring system.
    - Each individual measurement consists of a floating-point value, `d`, representing the density of ice particles (measured in g/cm^2) at a given elevation above the planet's surface.

Constraints

- `1 <= n <= 10,000`
- `0.000 < d <= 250.000`

Output

- Print a list of each distinct region of the ring system described by the cross-sectional data.
    - For gaps, print "`-------`" (7 dashes).
    - For rings, print the name of the ring and its average density (rounded to 3 decimal places), formatted as shown in the example output below.

Example Input File

rings.dat

```
13
0.203
132.378
140.014
2.876
4.931
88.725
9.582
90.821
89.752
92.529
20.774
55.126
64.937
```

Example Output To Screen

{System.out}

```
-------
A Ring: 136.196
-------
B Ring: 88.725
-------
C Ring: 91.034
D Ring: 20.774
E Ring: 60.031
```

Space Junk TEAM

Program Name: `Junk.java` **Input File:** `junk.dat`

There are currently 1,419 active satellites in orbit around the Earth. When these devices reach their end-of-life, they will join over 16,000 other pieces of man-made debris still littering the space surrounding our planet. Write a program to identify all of the potential space junk that each spacefaring nation is responsible for.

Input

- The first line of input will contain a single integer, `n`, that indicates the number of satellites to follow.
- Each of the next `n` lines will contain the following tab-separated information about a satellite:
  - The name of the satellite
  - The nation that owns and/or operates the satellite
  - The mass of the satellite at launch (in kilograms)
  - The date of its launch, formatted as "month/day/year" (all months and days are 2 characters each)
- The next line of input will contain a single integer, `m`, that indicates the number of test cases to follow.
- Each of the next `m` lines will specify a nation and either the word "DATE" or "MASS", separated by a colon.

Constraints

- `1 <= n <= 1,419`
- `1 <= m <= 10`
- Input data will contain at least 1 satellite entry for each test case nation.

Output

- For each test case, print a comma-separated list of the specified nation's satellites listed in *descending* order of either the date of their launches or their masses, depending on the specified criteria for the test case.
- Enclose each printed list in a pair of square brackets as shown in the example output below.

Example Input File

junk.dat

```
13
Akebono         Japan           294             02/21/1989
Amsat-Oscar 7   USA             29              11/15/1974
Aqua            Japan           2934            05/04/2002
Cubesat XI-V    Japan           1               10/27/2005
Firefly         USA             3               11/19/2013
Hodoyoshi-1     Japan           65              11/06/2014
Leasat 5        Australia       3400            01/09/1990
NOAA-15         USA             2223            05/13/1998
Optus 10        Australia       3270            09/11/2014
Optus B3        Australia       2858            08/28/1994
Optus D2        Australia       2400            10/05/2007
Sirius-1        USA             3727            06/30/2000
Superbird-C     Japan           3130            07/28/1997
3
Japan:DATE
Australia:MASS
Australia:DATE
```

Example Output To Screen

{System.out}

```
[Hodoyoshi-1, Cubesat XI-V, Aqua, Superbird-C, Akebono]
[Leasat 5, Optus 10, Optus B3, Optus D2]
[Optus 10, Optus D2, Optus B3, Leasat 5]
```

Copyright © 2016-2017 Jeff C. Mickel

Mnemonic TEAM

Program Name: `Mnemonic.java` **Input File:** `mnemonic.dat`

After demoting Pluto to a dwarf planet, a new movement is now underway to restore its rightful status as a planet while simultaneously *demoting* Mercury! The primary rationale behind this anti-Mercury sentiment is that its presence in the list of planets makes the standard mnemonics for remembering the order of the planet names more difficult. Because Mars and Mercury both start with the letter 'M', the familiar phrase "*my very educated mother just served us nine pizzas*" has been deemed to be too ambiguous. With the removal of Mercury, the newly proposed mnemonic for the planets of our solar system would be "*very egocentric martians just solved unpleasant naming problem*".

Write a program to convert a mnemonic string into an unambiguous list of its corresponding planets.

Input

- The first line contains a space-separated list of planet names within a star system. All planet names consist of a single, capitalized word. No two planets have names starting with the same letter.
- The second line of input will contain a single integer, `n`, that indicates the number of mnemonics to follow.
- Each mnemonic consists of a space-separated list of lowercase words that describes a desired ordering of planetary names.
    - All words in the mnemonic will match one of the listed planets
    - The mnemonics may or may not contain all of the listed planets
    - Some planets may be included multiple times in the same mnemonic

Constraints

- `1 <= n <= 10`

Output

- For each mnemonic, print its corresponding list of planet names with spaces separating each name.

Example Input File

mnemonic.dat
```
Venus Earth Mars Jupiter Saturn Uranus Neptune Pluto
4
very egocentric martians just solved unpleasant naming problem
every planet spins very nicely
multiple mnemonic meanings make me mental
victoriously each magnanimous judge serves us no penalties
```

Example Output To Screen

{System.out}
```
Venus Earth Mars Jupiter Saturn Uranus Neptune Pluto
Earth Pluto Saturn Venus Neptune
Mars Mars Mars Mars Mars Mars
Venus Earth Mars Jupiter Saturn Uranus Neptune Pluto
```

Rocket Fuel TEAM

Program Name: `Fuel.java` **Input File:** `fuel.dat`

A fundamental problem with rockets is that they have to lift their own fuel. That is, not only do they need enough fuel to lift the mass of the rocket, but then they need enough additional fuel to lift that fuel, which itself requires even more fuel, etc. The *Tsiolkovsky rocket equation* (shown below) describes the relationship between a rocket's "wet mass" (vehicle and contents plus propellant) and its "dry mass" (vehicle and contents without propellant).

$$orbital\ velocity = v_{exhaust} * ln\left(\frac{m_{vehicle} + m_{fuel}}{m_{vehicle}}\right)$$

Here, *orbital velocity* represents the minimum velocity needed for any object to achieve a stable orbit. This orbital velocity can also be calculated as a function of the planetary body's *mass* (M) and *radius* (r) as follows:

$$orbital\ velocity = \sqrt{\frac{G*M}{r}}, \text{ where } \ G = 6.67x10^{-11}\frac{Nm^2}{kg^2}$$

Rockets can achieve an *exhaust velocity* ($v_{exhaust}$) that is proportional to their *specific impulse* ($I_{sp} = \ 340.0s$, for liquid propellants) and the *gravitational acceleration* (g) of the planetary body from which they are launched according to the following formula:

$$v_{exhaust} = I_{sp} * g$$

In aerospace engineering, *mass ratio* is a measurement describing how many kg of propellant are needed for each kg of the vehicle's mass; that is, the ratio of the rocket's *wet mass* ($m_{vehicle} + m_{fuel}$) to its *dry mass* ($m_{vehicle}$). Given the *mass* (M), *radius* (r), and *gravitational acceleration* (g) for a planetary body, write a program that determines the mass ratio of a rocket system capable of launching from the surface into an orbit 200km above.

Input

- The first line of input will contain a single integer, `n`, that indicates the number of planetary bodies to follow.
- Each planetary body consists of 1 line of data containing 3 tab-separated numerical values:
  - The mass of the planetary body, measured in kilograms (kg)
  - The radius of the planetary body, measured in kilometers (km)
  - The gravitational acceleration of the planetary body, measured in meters-per-second-per-second (m/s^2)

Constraints

- `1 <= n <= 10`

Output

- Print the mass ratio of the vehicle's "wet mass" (i.e., vehicle plus fuel) to its "dry mass" (i.e., vehicle only), rounded to 2 digits of precision. For example, in the first test case (i.e., launching from Earth), for every 1 kg of the vehicle's mass, an additional 9.33 kg of fuel are required, yielding a `10.33:1` wet-to-dry ratio.

Example Input File

fuel.dat

```
4
5.972e24    6371     9.807
7.348e22    1737     1.622
6.390e23    3390     3.711
1.898e27    69911    24.79
```

Example Output To Screen

{System.out}

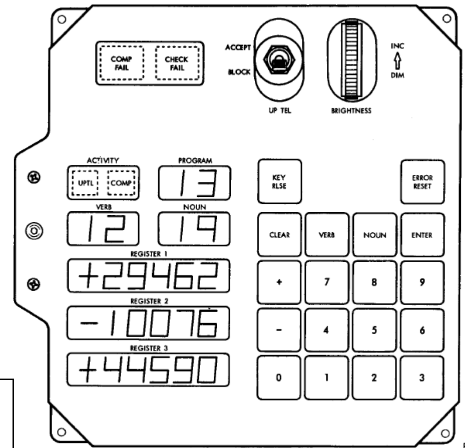
```
10.33:1
17.89:1
15.35:1
154.71:1
```

Copyright © 2016-2017 Jeff C. Mickel

Hello Moon TEAM

Program Name: `HelloMoon.java` **Input File:** `hellomoon.dat`

When travelling to the Moon, Apollo astronauts interacted with the onboard computer via the DSKY (display/keyboard unit), a simple panel consisting of a number pad an a handful of program status lights and numeric register displays. The Apollo Guidance Computer (AGC) could programmatically output signed, 5-digit decimal values to any of the panel's 3 register displays (*Register 1*, *Register 2*, and *Register 3*).



A simple "Hello World" exercise for the AGC programmers was to output the value "+07734" to one of the registers. When viewed upside-down, the display appears to read as the word "hELLO+".

Each register consists of a pos./neg. sign and five 7-segment digits:

Register 1	+	0	7	7	3	4
	R1Sign	*R1Digit1*	*R1Digit2*	*R1Digit3*	*R1Digit4*	*R1Digit5*

To change the displayed digits for any register, up to three 16-bit words must be written to the `OUT0` output channel. Each 16-bit word consists of the following bit pattern, where `WWWW` selects from 1 of 16 possible relays that control different panel components, `S` indicates whether to show a particular sign (specified by a `1` bit) or not (specified by a `0` bit), and `AAAAA` and `BBBBB` indicate two of the digits to be displayed.

16-bit word:	0	W	W	W	W	S	A	A	A	A	A	B	B	B	B	B

The following table shows the binary encodings for each of the relays that control the sign and digit components of the 3 register displays.

	0	WWWW	S	AAAAA	BBBBB
Relay 8	0	1000	0	00000	*R1Digit1*
Relay 7	0	0111	*R1Sign (+)*	*R1Digit2*	*R1Digit3*
Relay 6	0	0110	*R1Sign (–)*	*R1Digit4*	*R1Digit5*
Relay 5	0	0101	*R2Sign (+)*	*R2Digit1*	*R2Digit2*
Relay 4	0	0100	*R2Sign (–)*	*R2Digit3*	*R2Digit4*
Relay 3	0	0011	0	*R2Digit5*	*R3Digit1*
Relay 2	0	0010	*R3Sign (+)*	*R3Digit2*	*R3Digit3*
Relay 1	0	0001	*R3Sign (–)*	*R3Digit4*	*R3Digit5*

Each digit to be displayed is encoded using one of the 5-bit patterns shown in the table below.

Display Digit	AAAAA or BBBBB
unchanged	00000
0	10101
1	00011
2	11001
3	11011
4	10111
5	11110
6	11100
7	10011
8	11101
9	11111

For example, to display "+07734" in Register 2, the following 16-bit words would need to be written to `OUT0`:

`0 0101 1 10101 10011 = 0x2eb3`	Displays "+07" in *R2Sign*, *R2Digit1*, and *R2Digit2*
`0 0100 0 10011 11011 = 0x227b`	Displays "73" in *R2Digit3* and *R2Digit4*
`0 0011 0 10111 00000 = 0x1ae0`	Displays "4" in *R2Digit5*, leaving *R3Digit1* unchanged

Copyright © 2016-2017 Jeff C. Mickel

Input

- The only line of input will contain a register number, `r`, (either `R1`, `R2`, or `R3`) followed by an equals sign (=) and the signed, 5-digit decimal value, `n`, to be displayed in the specified register, `r`.

Constraints

- `r ∈ { R1, R2, R3 }`
- `-99999 <= n <= +99999`

Output

- List the hexadecimal representations for each of the 16-bit values that must be written to `OUT0` in order to display the specified value in the specified register of the DSKY interface console.
- List the hexadecimal values in descending order.

Example Input File

hellomoon.dat

```
R2=+07734
```

Example Output To Screen

{System.out}

```
0x2eb3
0x227b
0x1ae0
```