Computer Science Competition
2017 Invitational A
Programming Problem Set

I. General Notes

1. Do the problems in any order you like. They do not have to be done in order from 1 to 12.

2. All problems have a value of 60 points.

3. There is no extraneous input. All input is exactly as specified in the problem. Unless specified by the problem, integer inputs will not have leading zeros. Unless otherwise specified, your program should read to the end of file.

4. Your program should not print extraneous output. Follow the form exactly as given in the problem.

5. A penalty of 5 points will be assessed each time that an incorrect solution is submitted. This penalty will only be assessed if a solution is ultimately judged as correct.

II. Names of Problems

Number	Name
Problem 1	Abdul
Problem 2	Daniel
Problem 3	Geming
Problem 4	Htoo
Problem 5	Isabella
Problem 6	Jean
Problem 7	Kimberley
Problem 8	Luana
Problem 9	Maria
Problem 10	Patricia
Problem 11	Rohit
Problem 12	Tisha

1. Abdul

Program Name: Abdul.java Input File: abdul.dat

In his fascination with math patterns, Abdul has discovered an interesting phenomenon when adding up odd numbered values. Help him by writing a program for his research project.

Write a program to input a positive integer N (2 < N <= 100) and output the sum of all the odd integers from 1 up to, but not including, twice that number. For example, if N is 5, you would add the values 1, 3, 5, 7 and 9, which makes 25.

Input: Several integers, each on one line.

Output: The sum of all odd values from 1 up to 2N.

Sample input:

```
5
11
16
```

Sample output:

```
25
121
256
```

2. Daniel

Program Name: Daniel.java **Input File: daniel.dat**

Daniel considers himself a word smith of sorts, and loves pattern poetry, like haikus, cinquain poems, limericks, and such. He also makes it a hobby to find really cool quotes. One day he decided to form word triangles out of the quotes he found, and since he knew of no one else who had ever done that, decided to call them Daniel Triangles.

A Daniel Triangle starts with the first word of the quote on the first line, the next two words on the next line, then the next three on the third line, and so on. If however, the next line exceeds 30 characters, he starts another triangle in the same fashion, doing this over and over until the quote has ended.

In the first example below, a fairly short quote by Steve Martin, he puts the first word "Be" on the top line of the triangle, followed by "so good" on the next line, then "they can't ignore" on the third, followed by the last word, the dash (he counts the dash as a word), and the author's name. To separate each quote, and to make sure it does not exceed his self-imposed 30-character limit, he follows each quote with a metric line of 30 characters - three sets of nine dashes, each set ending with an asterisk.

Input: Several quotes, each on one line in the data file. Each quote ends with a dash, followed by the author's name. All parts of the quote, including the dash and author's name, are separated by single spaces. Any punctuation marks are considered part of the word they follow or that contains them.

Output: A Daniel Triangle for each quote, as described above, followed by the 30-character metric line as shown.

Sample input: *(each quote will be all on one line in the data file)*

```
Be so good they can't ignore you. - Steve Martin
Ever tried? Ever failed? No matter. Try again. Fail again. Fail better. -
Samuel Beckett
Impossible is just a big word thrown around by small men who find it easier
to live in the world they've been given than to explore the power they have
to change it. Impossible is not a fact. It's an opinion. Impossible is not a
declaration. It's a dare. Impossible is potential. Impossible is temporary.
Impossible is nothing. - Muhammed Ali
```

Sample output:

```
Be
so good
they can't ignore
you. - Steve Martin
---------*---------*---------*
Ever
tried? Ever
failed? No matter.
Try again. Fail again.
Fail better. - Samuel Beckett
---------*---------*---------*
Impossible
is just
a big word
thrown around by small
men who find it easier
to live in the world they've
been given than to explore the
power
they have
to change it.
Impossible is not a
fact. It's an opinion.
Impossible
is not
a declaration. It's
a dare. Impossible is
potential. Impossible is
temporary.
Impossible is
nothing. - Muhammed
Ali
---------*---------*---------*
```

3. Geming

Program Name: Geming.java **Input File: geming.dat**

In a recent computer science class lesson on integers and their opposite values, Geming has learned a new concept called the complement. He is a bit confused with this term because he learned one definition of it in geometry, but now is being taught a new definition, which is this:

- The complement of any integer value is its opposite, minus one.

For example, the complement of zero is -1. The complement of 1 is -2. The complement of -11 is 10, and so on.

Help Geming galvanize his understanding of this new concept by writing a program to input any integer value in the range, -10000...10000, and output its complement value.

Input: Several integer values, each on one line.

Output: The original value and its complement, separated by a single space, each pair of values on one line.

Sample input:

```
0
1
-11
```

Sample output:

```
0 -1
1 -2
-11 10
```

4. Htoo

Program Name: Htoo.java Input File: htoo.dat

Htoo wants to build a computer vision algorithm for identifying objects in photos. He realizes the first thing he must accomplish toward his task is to identify when two things are touching, which when realized in a two dimensional plane, means that there are two line segments that intersect. He overlaid all of his pictures for his vision algorithm with a pixel grid. Given the start and end point of each of the line segments in his pixel array, determine if the two segments in the photo intersect at any given pixel.

Htoo guarantees that there will only be two line segments, designated by two sets of ordered pairs, in any given photo.

Input: The first integer is the number of photos. Each photo will span two lines of input, the first representing the start and end pixel (in x, y coordinates) of the first line, and the second presenting the start and end pixel (in x, y coordinates) of the second line.

Output: If the two lines cross or touch at any point, print "INTERSECT" otherwise, print "NO INTERSECTION"

Sample input:

```
3
(0, 0) > (2, 2)
(2, 0) > (0, 2)
(0, 0) > (2, 2)
(0, 1) > (2, 3)
(0, 0) > (2, 2)
(4, 4) > (2, 2)
```

Sample output:

```
INTERSECT
NO INTERSECTION
INTERSECT
```

5. Isabella

Program Name: Isabella.java **Input File: isabella.dat**

Isabella is a banker at an international bank. It's the end of her workday and she is checking all the ledgers to make sure that all the accounting from the day is in agreement. However, because her work is international, so she has to deal with currency from all over the world, which means that the ledger she is checking against can be in any of the currencies of which she does business.

Luckily, Isabella is only dealing with deposits, which means she only has to verify the ledgers of the deposits. Every day she does get the latest conversion rates for all the different currencies. Since this is the world of banking, decimal points are important, so she must be sure to not lose any precision. Please help her ensure that her ledger is a good and matches the ledger she is checking against.

Each set of conversion data is of the form:

```
Currency1 Currency2 rate
```

which means that 1 unit of Currency1 is equal to the rate value of Currency2.

For example, **`USD COP 2960.843`** means that 1 USD is equal to 2960.843 COP.

Input: The first integer is the number of conversion currency rates she has to deal with in one day. Each line of the conversion data will be two 3-letter currency abbreviations, followed by a rate, in the form shown above. All currency conversions will be given, so she never has to try and calculate what a conversion should be for a country. The next line will be an integer representing the number of data sets to follow. The first integer of each data set will be the number of lines in Isabella's ledger that she needs to check. The last line of each data set will be the value from the ledger that she is checking against.

Output: The output will be either **`"GOOD LEDGER"`**, if the ledger she is checking against matches the sum in her ledger or **`"BAD LEDGER"`**, if the ledger she is checking against has a different value than the sum of her ledger.

Sample input:

```
6
MXN ISK 10
MXN USD 0.0000000000005
ISK MXN 0.1
ISK USD 0.00000000000005
USD MXN 2000000000000
USD ISK 20000000000000
3
3
1000 USD
6890 MXN
43.3 ISK
2000000000006894.33 MXN
1
1.0 USD
0.0 ISK
10
1000 USD
1000 MXN
1000 ISK
1000 ISK
1000 ISK
1000 ISK
1000 MXN
1000 USD
1000 USD
1000 USD
80000000000024000 ISK
```

Sample output:

```
GOOD LEDGER
BAD LEDGER
GOOD LEDGER
```

6. Jean

Program Name: Jean.java Input File: jean.dat

Jean is part of a bike sharing program in her town. However, she has noticed recently that some of the bike stops in her town will have an abundance of bikes, whereas other stops will be completely empty. To help address where and how the bikes are moving, she went to every location at the beginning of the day, and tagged one bike. She then monitored all the places the bike traveled that day.

She needs your help to identify where the sources and sinks are in her bike share community stops. A **source** is defined as a stop that one or more bikes left from and none came to. A **sink** is defined as a spot that one or more bikes came to but none left from. All the stops are labeled with letters for Jean to keep up with them.

Input: The first integer is the number of data sets to follow. Each data set will begin with an integer to represent the number of stops Jean is monitoring. Each line will representing the bike that Jean tagged, where it started, a "**->**" symbol, and then all places the bike traveled to (in no particular order). If a bike was not checked out, this will be noted as "**NONE**".

Output: The list of alphabetized sources, prefaced by the word "**Sources:** " and the list of alphabetized sinks, prefaced by the word "**Sinks:** ". If there are none, then say "**NONE**".

Sample input:

```
3
3
A -> A B C
B -> B
C -> NONE
4
E -> E
F -> F
G -> G
D -> NONE
5
Q -> B C V
B -> B C V
C -> B C
D -> B C V
V -> NONE
```

Sample output:

```
Sources: NONE
Sinks: C
Sources: NONE
Sinks: NONE
Sources: D Q
Sinks: V
```

7. Kimberley

Program Name: Kimberley.java **Input File: kimberley.dat**

Kimberley loves watching birds, especially the large flocks that move together in flight so gracefully. She has recently run across a theory study of how these flocks move, and needs your help in simulating a small part of that study.

Using a single row of birds that always fly from left to right, she wants to see how some simple rules will determine the position of each bird in the row after a certain time stage as the birds fly.

The rules of the simulation are:

1. At the beginning of any stage change, if the left most space in the row is empty and the right most space is occupied by a bird, that bird leaves the right most space empty and moves to the left most space for the next stage.
2. A bird stays in place if the space to the right of it is occupied.
3. A bird moves one place to the right if that space is empty.
4. A bird moves at most one time during a stage change.

For example, the diagram shown below represents a flock of birds in three stages, moving according to the rules listed above.

```
Position | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10|
-----------------------------------------------
Stage 1  | * |   | * | * |   | * |   | * | * | * |
Stage 2  |   | * | * |   | * |   | * | * | * | * |
Stage 3  | * | * |   | * |   | * | * | * |   | * |
-----------------------------------------------
```

From stage 1 to 2, the birds in positions 1, 4, and 6 move. All others remain.
From stage 2 to 3, the bird in space 10 falls back to position 1, and then the birds in 3, 5 and 9 all move.

Input: Several sets of data, each data set consisting of a row of integers with single space separation. The first integer represents the number of spaces to be used in the row. The next integer N indicates how many birds are in the row, with the next N values indicating their positions in the row. The final integer represents the number of stages to be considered for the simulation.

Output: After the final stage of each data set is completed, report the positions of all of the birds as a row of integers with single space separation.

Sample input:

```
9 3 1 3 5 2
12 6 1 3 5 7 10 11 4
10 7 1 3 4 6 8 9 10 3
```

(this data set matches the example described above)

Sample output:

```
2 4 6
2 4 6 8 10 12
1 2 4 6 7 8 10
```

8. Luana

Program Name: Luana.java **Input File: luana.dat**

Luana has just learned about a biological simulation of how cells might evolve given a certain set of rules about how they are reborn, survive, and die in several life cycle generations. In this study, a 10X10 grid is used, like the one shown below, to indicate the beginning generation. The top left cell is in position (1,1), and the one right below it is in position (2,1). Every cell has at most eight neighboring cells, one in each of the eight cardinal directions.

For this exercise, the grid shown below will be the starting generation each time a new set of evolution rules is applied. In each grid, a ***** indicates a living cell, and a **–** represents a dead cell. If the right number of living cells surrounds a dead cell, it will be reborn, or can survive to the next generation. If not enough cells surround a living cell, it will die.

Here is an example of a set of rules.

- 2 2 4: There 2 ways for a dead cell to be reborn, and that is when it is surrounded by exactly 2 or 4 living cells, otherwise it stays dead.
- 2 3 4: There are 2 ways for a living cell to survive, and that is when it has 3 or 4 living neighbors, otherwise it dies.

The research is to check the status of a particular cell, (5, 2) for example, at each point for a number of generation cycles, let's say 4.

The data set for this situation would be: 2 2 4 2 3 4 4 5 2, which shows the two rule definitions explained above, the number of generations to be evolved, and the cell to be examined.

The result will be DADAA, which means that cell (5, 2) is dead to start with, is reborn after the first generation cycle, dies after the second, is reborn after the third, and stays alive for the fourth.

Let's look at this sequence carefully, with a sample evolution of the cell in location (5,2), reborn when surrounded by 2 or 4 living cells, surviving when surrounded by 3 or 4 living cells, dying otherwise.

First generation

```
*---------
*---------
---*------
----*-----
-[-]*--*---*  Cell in (5,2) is currently dead, but is reborn in the
-*-**--*--    next generation since it is surrounded by 2 living cells
--*-----**
-*-------*
*--------*
---------*
```

Second generation

```
-*--------
-*--------
----*-----
--*--*----
-[*]-**-*-*-  Cell in (5,2) is now alive, but dies since it is only surrounded
--**-**-*-    by 2 living cells (needs 3 or 4 living neighbors to survive)
*-*-*--**-
*-*-----**
-*--------
--------*-
```

Third generation

```
*-*-------
*-*-------
-***-*----
-*-*****--
```

```
-□-**-**-*  Cell in (5,2) is dead, but is reborn in the next generation
***--**-*-   since it is surrounded by 4 living cells
--*----***
--------*-
*-*----*--
----------
```

Fourth generation

```
-*-*------
--***-----
-*-*-***--
*------*--
-*-*----*-  Cell in (5,2) is alive, and stays alive in the next generation,
-***-**--*   since it is surrounded by 3 living cells.
**-*-*-***
--**--***-
-*------*-
-*--------
```

Input: The initial 10X10 grid as described above, with a value N on the second line, followed on the next N lines with simulation data sets. Each data set consists of an integer B, followed by B integers indicating the number of neighbors required for a dead cell to be reborn, then an integer S, followed by S integers indicating the number of neighbors required for survival. After that is an integer G, which indicates how many generations to examine, and finally an ordered pair of integers R and C, indicating the cell location to study.

Output: A string of letters D or A indicating the status of the indicated cell, first showing the initial state, and the state after each generation according to the rules specified for that data set.

Sample input:

```
*---------
*---------
---*------
----*-----
--*--*---*
-*-**--*--
--*-----**
-*-------*
*--------*
---------*
2
2 2 4 2 3 4 4 5 2
3 1 2 3 3 1 2 3 2 5 10
```

Sample output:

```
DADAA
ADD
```

9. Maria

Program Name: Maria.java Input File: maria.dat

Maria is writing a game program and needs your help displaying the different faces of a common six-sided die.

Write a program to input an integer in the range of 1-6 and output the corresponding 3X3 pattern that represents the value, with "*"s for the dots and "-"s for the spaces.

For example, if the input is 5, the output would be:

```
*-*
-*-
*-*
```

For the values 2 and 3, make the diagonal go from top left to bottom right. The middle dot will only be used for odd dice values.

Input: Several integers, all on one line, separated by single spaces.

Output: The corresponding 3X3 pattern representing the die face for that value.
Follow each die face output with a blank line.

Sample input:
```
5 1 4
```

Sample output:
```
*-*
-*-
*-*

---
-*-
---

*-*
---
*-*
```

10. Patricia

Program Name: Patricia.java **Input File: patricia.dat**

Chance and probability have always interested Patricia, and she has been experimenting with drawing red and green pebbles out of a sack, in different combinations. She has learned the theory behind it all, and knows that the chances of drawing a single red pebble out of a sack of both red and green marbles is simply a fraction with the numerator being the number of red pebbles in the sack, and the bottom number being the total number of red and green pebbles. For example, if there are 8 red pebbles and 2 green pebbles in the sack, the chances of removing a red pebble is 8 out of 10. The chances of drawing a green pebble would be 2 out of 10.

She also knows that there are several combinations of two draws that can be calculated. For example, if the same 8 red and 2 green marbles are in the bag, the chance of drawing first a red, and then a green, depends on whether or not you put back the first pebble. If you do put it back, the chances of this two pebble combination would be the product of the two fractions, which would be 8/10 times 2/10, or 16/100. However, if you did not replace the first pebble drawn, the chances of the same combination increase to 16/90, the product of 8/10 and 2/9. She understands that the number pebbles in the bag has been reduced by 1, which is why the second fraction has the denominator of 9.

She needs your help in researching all possibilities with different number of red and green pebbles, including drawing two of the same color, with or without replacing the first pebble, and so on.

Input: An initial integer N, followed by N data sets. Each data set starts with the value P (1 or 2), indicating how many pebbles are to be drawn out of the bag for the experiment. Following that are two integers, indicating how many red and green pebbles are in the bag. If 2 draws are to be made, a letter Y or N indicates whether or not the first pebble will be replaced, followed by P instances of the letters 'R' and/or 'G', indicating the colors to be drawn.

For example, the first data set below indicates that one red pebble will be drawn from a bag containing 8 red and 2 green pebbles.

For the next data set, two pebbles (first a red, then a green) will be drawn from a bag initially containing 8 red and 2 green pebbles. The N indicates that the first pebble will not be replaced.

Output: The non-reduced fraction representing the chances of drawing the pebble(s) indicated by the data set.

Sample input:

```
3
1 8 2 R
2 8 2 N R G
2 8 2 Y R G
```

Sample output:

```
8/10
16/90
16/100
```

11. Rohit

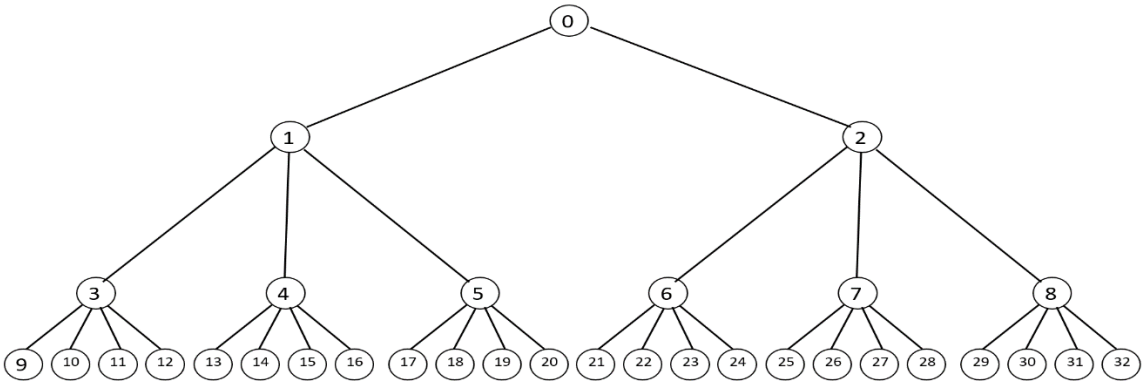
Program Name: Rohit.java Input File: rohit.dat

Rohit just learned about binary trees and is so excited that he has decided to invent a new kind of tree data structure!

Binary trees are trees where each node has 2 or fewer children. However, trees can have any number of children, even a variable number of children. Each level of Rohit's trees may have any number of children. There will always be a single node root for his trees.

He would like your help implementing the **getChildren** method of this new tree. To prove that your implementation works, Rohit has asked that given the number of children at each level of the tree beyond the root, you build a tree filled incrementally by level. Then using that tree test your **getChildren** method.

For example, if Rohit gives you "2, 3, 4" then this corresponds to a tree where the root has 2 children, each of those 2 children have 3 children, and each of those have 4 children. The resulting tree would look like this:



If Rohit calls getChildren(4), the result would be "13 14 15 16", all of the children of the node in position 4. If he asked for children of "0" the answer would be "1 2". If he asks for the children of a node that has no children, like any of the nodes on the last level, such as 19, then the answer would be "NO CHILDREN". If Rohit asks for the children of a node that isn't in the tree, like 33, then answer would be "NOT IN TREE".

Input: The first integer is the number of data sets to follow. Each data set will have two lines, the first line is a list of integers representing the number of children at each level of the tree. The second integer is the node being used in the getChildren method call.

Output: The list of children of the node in question, "NO CHILDREN", or "NOT IN TREE".

Sample input:

```
4
2 3 4
4
5 2 2
23
2 2 2 2 2 2 2
119
1 1 1 1 1
12
```

Sample output:

```
13 14 15 16
NO CHILDREN
239 240
NOT IN TREE
```

12. Tisha

Program Name: Tisha.java Input File: none

Making ASCII character face patterns is so much fun for Tisha that she wants you to join her in making a face pattern of your own.

Create an 8X8 character pattern using characters that can be generated from the keyboard, or if you can, other characters you might know of, like box characters.

The entire pattern must span exactly eight rows and eight columns, with at least one character on each row, and at least one in each column. There must be at least one \ (backslash) character and one " (double quote) character in the pattern.

Input: None

Output: A unique face pattern, of your own creation, as described above.

Here is an 8X8 dash pattern to help you sketch out your creation.

```
- - - - - - - -
- - - - - - - -
- - - - - - - -
- - - - - - - -
- - - - - - - -
- - - - - - - -
- - - - - - - -
- - - - - - - -
```