Computer Science Competition
2016 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	A_One – Loopy Lambda
Problem 2	A_Two – Twin Primes
Problem 3	A_Three – Penny Dreadful
Problem 4	A_Four – SimpleCalc
Problem 5	A_Five – Bones
Problem 6	A_Six – BubbaSpice
Problem 7	A_Seven – Mediant
Problem 8	A_Eight – Linked
Problem 9	A_Nine – 'Rithmetic
Problem 10	A_Ten – Abundance
Problem 11	A_Eleven – Block
Problem 12	A_Twelve - Recognition

1. A_One – Loopy Lambda

Program Name: A_One.java **Input File: none**

Java 8 has some really cool stuff! Below is a complete program that creates a list of integers and outputs the list in six different ways, two of which are new techniques from Java 8.

- The first three are the traditional **for, while,** and **do while loop** processes.
- Next is the **for each** loop.
- The fifth uses a lambda expression.
- The sixth uses the double colon output technique.

There is no input required for this program. Just use the List structure provided, and choose four different ways to accomplish the same output. The judge will look at your source code to make sure you used **four different ways.** ***You are welcome to copy the code shown below for your solution.***

You are encouraged to use the new Java 8 techniques shown, but only if you're running Java 8.
DO NOT simply create the output by hardcoding.

```
import java.util.*;
import static java.lang.System.*;
public class One{
	public static void main(String...args){
		List<Integer> list = Arrays.asList(1, 2, 3, 4, 5, 6, 7);
		//traditional for loop
		int x=0;
		for(x=0;x<list.size();x++)
			out.print(list.get(x));
		out.println();
		//while loop
		x=0;
		while(x<list.size()){
			out.print(list.get(x));
			x++;
		}
		out.println();
		//do while loop
		x=0;
		do{
			out.print(list.get(x));
			x++;
		}while(x<list.size());
		out.println();
		//for each loop
		for(int n:list)
		    out.print(n);
		out.println();
		//lambda version...new with Java 8
		list.forEach(n->out.print(n));
		out.println();
		//new Java 8 double colon print statement...very cool!
		list.forEach(out::print);
		out.println();
	}
}
```

Input: None

Output: Four lines of digits as shown below, each line produced by a ***different*** iterative process of your own choosing.

*****The output is NOT to be hardcoded by simply outputting the string, "1234567".*****

Sample input: None
Actual output:

```
1234567
1234567
1234567
1234567
```

2. A_Two – Twin Primes

Program Name: A_Two.java **Input File: a_two.dat**

Two consecutive odd numbers that are both prime numbers are called twin primes. For example, 5 and 7 are the largest pair of twin primes both less than 10. Likewise, 17 and 19 are the largest pair of twin primes less than 20. Given an input integer N, find the largest pair of twin primes less than N.

Input: A data file with several values of N, arranged vertically, 5<N<10,000.

Output: The largest pair of twin primes less than N, formatted as shown below.

Sample input:

```
10
20
372
```

Sample output:

```
5 7
17 19
347 349
```

3. A_Three – Penny Dreadful

Program Name: A_Three.java **Input File: a_three.dat**

A simple penny game involves a strip of several adjacent spaces, on which pennies are randomly placed. The number of squares S (2<S<100) is random, as are the number of pennies P placed, as long as P (2<=P<S) is less than S.

The rules for a complete game, should you wish to play it, are as follows:

- After the pennies are randomly placed by one player, the players take turns moving pennies to the left.
- A move can be 1, 2, 3, 4 or 5 spaces, as long as
  - a penny does not share the same space as another penny after the move, and
  - a penny can't pass another penny during the move.
- The winner of the game is the last player to move a penny.

For example, on the game board shown above, with 12 squares, and 7 pennies at positions 1, 4, 6, 8, 9, 10 and 12, 4 pennies can move 1 space (the pennies at positions 4, 6, 8 and 12), 1 penny can move 2 spaces (the one in position 4), but none can move any further.

Your job is to report the possible number of moves by the first play, itemized by possible number of spaces moved. The output for this situation would be: 4 1 0 0 0, which means there are 4 possible 1 space moves, 1 possible 2 space move, and no possible moves of 3, 4, or 5 spaces.

Input: A data file with several sets of data. Each data set is a series of integers, all on one line, with single space separation. The first integer S indicates the number of squares on the penny strip, the second value P shows the number of pennies on the strip, followed by P positive values indicating the penny positions on the board. Each of the penny positions are guaranteed to be unique, in ascending order, and less than or equal to S. P will always be at least 2, and less than S.

Output: Five values per data set, the first indicating the number of possible 1 space moves by the pieces on the board, the second the number of 2 space moves, and so on…3 space moves, 4 space moves, and 5 space moves.

Sample input:

```
12 7 1 4 6 8 9 10 12
14 6 1 3 6 7 10 14
10 2 4 9
```

Sample output:

```
4 1 0 0 0
4 3 1 0 0
2 2 2 1 0
```

4. A_Four SimpleCalc

Program Name: A_Four.java **Input File: a_four.dat**

This problem demonstrates a very simple command line calculator using the codes listed below. Your job is to write a program to evaluate the results of the commands listed.

The codes used are listed below, where X indicates a single capital letter that indicates a storage location, and Y is an integer value within the standard range of a byte value:

DEF X Y - Assign the Y value into the X variable indicated
STO X - Store the current memory value into the X variable indicated
REC X - Place the value of X into current memory
ADD X - Add the X value into current memory
SUB X - Subtract X from current memory
MUL X - Multiply current memory by X
DIV X - Integer divide current memory by X
PRN X - Output value contained in the variable X indicated, followed by a linefeed.

Input: A series of commands, each on one line, each command representing one of the calculation codes listed above.

Output: Show the results of each PRN command during the series of commands.

Sample Input:

```
DEF A 1
DEF B 2
DEF C 3
PRN C
REC A
MUL C
STO A
PRN A
DIV B
STO C
PRN C
SUB A
STO B
PRN B
ADD A
PRN A
```

Sample output:

```
3
3
1
-2
1
```

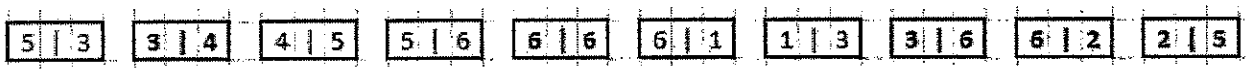
5. A_Five - Bones

Program Name: A_Five.java **Input File: a_five.dat**

A typical set of dominoes has 28 "bones", ranging in value from 0 to 6 on each half of a piece. For example the blank piece would be represented by 00, and the double six, often called the "boxcar", is 66. The pieces in ascending sequence would be 00, 01, 02, 03...32, 33, 34, 35...54, 55, 56, and 66. Keep in mind that the order of the two halves of a bone are interchangeable, so, for example, a "35" bone would be considered the same as the "53" bone.

For this problem you are given five dominoes randomly positioned in ten places along a horizontal line, followed by five more dominoes that are guaranteed to fit the five blank positions. For example, the first sample data line below represents an opening board that looks like this, where xx means a blank spot to be filled with the five remaining "bones":

with 66, 34, 25, 36, and 62 to fill the blank spots, with a final solution of:



Remembering that the dominoes can be reversed, the five remaining "bones" could be listed as in the second data example below: 66 43 52 63 26

with the same expected resulting output of:

```
53 34 45 56 66 61 13 36 62 25
```

Input: Several data sets, each line consisting of integers with single space separation, the first five indicating the positions (in the range 1-10) of the five "bones" already positioned on the board, followed by five pairs of integers representing the pieces in those positions, and then the remaining five pairs indicating the pieces to fill the blank spots.

Output: The resulting board with all pieces filled in correctly. A unique and successful solution for each data set is guaranteed.

Sample input:

```
1 3 4 6 7 5 3 4 5 5 6 6 1 1 3 6 6 3 4 2 5 3 6 6 2
1 3 4 6 7 5 3 4 5 5 6 6 1 1 3 6 6 4 3 5 2 6 3 2 6
1 3 5 7 9 2 3 4 5 6 6 1 2 6 0 3 4 6 1 5 6 5 0 2 6
```

Sample output:

```
53 34 45 56 66 61 13 36 62 25
53 34 45 56 66 61 13 36 62 25
23 34 45 56 66 61 12 26 60 05
```

6. A_Six - BubbaSpice

Program Name: A_Six.java **Input File: a_six.dat**

Bubba Gump's Spice Shop is old school. Bubba has been keeping track of price changes on a particular product all year - his secret spice packet called BubbaSpice for boiled shrimp - and needs to see a report of all of these changes, in chronological order. He's been keeping a record of all changes on note cards, and is not real sure of the accuracy, but is pretty sure if all could be compiled into one report, he would have a true picture of the price changes.

For example, the first price of $5.00 was set for period from January 1, 2016 to June 30 of the same year. It shows on the card as "`1/1 6/30 5.00`". The next card shows "`2/1 3/15 5.21`", which means that during that time period, he raised the price to $5.21, but it reverted back down to $5.00 after that, from March 16 to June 30. Needless to say, he is very confused and needs your help.

Given the data as shown below, with each data set showing a starting date, ending date, and price, generate a listing after each card is entered into the mix, of the sequence of price ranges from the first date to the last date of record.

The sample output below shows the correct sequence given the sample input data.

Input: Several data sets, each on one line, consisting of two dates in "`m/d`" format representing the starting and ending date of a price, followed by the price for that date range. The starting date will always be January 1, 2016, and the ending date will be on or before December 31, 2016.

Output: A complete chronological report after each data set of the known starting date of all price changes seen up to this data set. The word END should be displayed after the final date in the date range, with a single blank line following each report, formatted as shown.

Sample Input:

```
1/1 6/30 5.00
2/1 3/15 5.21
3/16 6/30 5.75
2/22 3/16 5.16
```

Sample Output:

```
01/01/16 $5.00
06/30/16 END

01/01/16 $5.00
02/01/16 $5.21
03/16/16 $5.00
06/30/16 END

01/01/16 $5.00
02/01/16 $5.21
03/16/16 $5.75
06/30/16 END

01/01/16 $5.00
02/01/16 $5.21
02/22/16 $5.16
03/17/16 $5.75
06/30/16 END
```

7. A_Seven - Mediant

Program Name: A_Seven.java **Input File: a_seven.dat**

In music, a **mediant** is the third tone of a major scale, sung in solfege as mi. In math there is likely no such thing as a **mediant**, but for now, we'll just make up a new definition for a math based **mediant.**

A **mediant** is a number that is a variation of another, based on the median of the digits of the number, according to the following rules:

1. First find the median of the digits, then find the digit in the number which is closest to but not larger than the median value, and then change that digit according to the rules below. If more than one instance of this value exists, use the leftmost instance. For example, in the value 9999, the found digit value is clearly 9, and the one that is changed is the leftmost 9.
2. If that digit is 0, 1, or 2, replace it with the largest digit in the number.
3. If that digit is 3, 4, or 5, replace it with the smallest digit in the number.
4. If that digit is 6, 7, or 8, replace it with the ones place of the sum of the digits.
5. If that digit is 9, replace it with 0.

By definition, the median of a list of numbers is the middle value when the list is sorted. If the number of items in the list is even, and the two middle numbers are different, the median is the average of the two. For example, the median of the values 1, 2, 3 and 4 is 2.5, the average of 2 and 3, therefore the number chosen in this case would be the 2, which is closest, but not larger than 2.5.

In the first example below, the median of the value 123 is 2, which is replaced by 3, the largest digit of the number.
The value 745 becomes 744 since the median is 5, which is replaced by 4, the smallest digit.

Input: A series of integer values, N, where $100 <= N < 10000$.
Output: The **mediant** of each value, as defined in the rules listed above.

Sample Input:

```
123
745
1689
9999
```

Sample output:

```
133
744
1489
999  (note: 0999 is incorrect)
```

8. A_Eight - Linked

Program Name: A_Eight.java **Input File: a_eight.dat**

Wikipedia articles often have links to other Wikipedia articles for the objects in the article that already have Wikipedia articles. We can use this direct linking to deduce how related two Wikipedia articles could be. To say that two articles, ***a*** and ***b*** are **related** means to say that *there exists a series of links you could click that could get you to the same article c starting from article **a** and article **b***. Furthermore, the *strength of this relationship is determined by the smallest number of cumulative clicks* it takes to get from ***a*** to this mutually shared article, plus the number of clicks it took from ***b***.

More concretely, imagine we had these Wikipedia articles, where the @ sign indicates a link from one article to another, and before the : represents the title of the article.

```
black_lab: A dog_breed@ that has black@ fur.
Jack_Russell: A hyper dog_breed@ that is often used to assist when hunting.
black: A color.
dog_breed: A phenotype of a dog@.
cat: A small mammal@ that the internet loves.
dog: A larger mammal@ that is a common household pet.
mammal: An animal@ with fur and births its young.
animal: A living organism that eats organic matter.
carrot: A yummy orange plant.
```

In this example, **black_lab** and **Jack_Russell** articles are related because they both link to the **dog_breed** article with one click from **black_lab** and one click from **Jack_Russell**, so 2 cumulative clicks apart. In a similar situation are **cat** and **dog** related, both linking to **mammal** with one click again making 2 cumulative clicks between. **black_lab** and **cat** are related because they both link to mammal, 3 clicks from **black_lab** (once on **dog_breed**, once on **dog**, then to **mammal**) and then one click from **cat**, so 4 cumulative clicks.

There is a notion of an article being linked to itself which implies that **black_lab** is related to **dog_breed** by 1 cumulative click (1 click from **black_lab** and 0 clicks from **dog_breed**). Furthermore, this means that **Jack_Russell** is related to animal by 4 cumulative clicks (4 clicks - **Jack_Russell** to **dog_breed** to **dog** to **mammal** to **animal** + 0 clicks from **animal**).

Lastly, carrot and cat are not related because there are no links that can be clicked starting from these two articles to end up at the same article (because carrot is connected to nothing and nothing is connected to carrot).

Of the article pairs we just analyzed, **black_lab <=> dog_breed** is the strongest relationship because it has the fewest cumulative clicks. **carrot <=> cat** is the weakest because it has no relationship at all.

Now, using these definitions of related and cumulative clicks, given a list of article name pairs, put them in order from strongest relationship to weakest relationship.

Input: The first line will have an initial integer N indicating the N data sets to follow. In each data set the first line will contain an integer indicating the number of Wikipedia articles in this data set. Each article will take up one line, starting with a single string as its title, followed by a colon, followed by a brief article. All words in the article description ending with @ are considered linked to another article as denoted by the title. The @ only designates a link.

For example, **dog_breed@** appearing in a sentence means that there exists an article with the title, **dog_breed**, as you can see in the sample data below. The two strings for each article listing include the title and then a brief description, separated by **": "**. The description will have zero or more references as described above. For instance, the first data listing below has two references, **dog_breed@** and **black@**. The listing for **black** has no references at all.

After the listing of the articles will be another integer M indicating the M article pairs to be analyzed. Each article pair will have two strings with the titles of the two articles to be analyzed.

Output: A sorted list of the M article pairs, ordered from strongest relationship (least link count) to weakest relationship in the given configuration. If more than one pair have the same strength relationship, order them alphabetically. Output a blank line between each output set.

(A_Eight – cont)

Assumptions: All links will exist in the data set. All article sets will have at least 2 articles. In an article pair, the two articles will be different. All article titles will be unique. All articles will have a title and content, with zero or more links to other articles.

Sample Input

```
2
9
black_lab: A dog_breed@ that has black@ fur.
Jack_Russell: A hyper dog_breed@ that is often used to assist when hunting.
black: A color.
dog_breed: A phenotype of a dog@.
cat: A small mammal@ that the internet loves.
dog: A larger mammal@ that is a common household pet.
mammal: An animal@ with fur and births its young.
animal: A living organism that eats organic matter.
carrot: A yummy orange plant.
6
black_lab  Jack_Russell
cat  dog
black_lab  cat
black_lab  dog_breed
Jack_Russell  animal
carrot cat
10
a: b@
b: c@
c: d@
d: e@
e: f@
f: g@
g: h@
h: i@
i: j@
j: nothing
5
a b
a c
a d
a j
i j
```

Sample output:

```
black_lab dog_breed
black_lab Jack_Russell
cat dog
Jack_Russell animal
black_lab cat
carrot cat

a b
i j
a c
a d
a j
```

9. A_Nine – 'Rithmetic

Program Name: A_Nine.java **Input File: a_nine.dat**

This is an easy one. Just write a program that does some simple **'rithmetic!**

Input: An initial value N, followed by N pairs of positive integers A and B, with 0 < A <=100,000 and 0 < B <=100,000.

Output: The sum, positive difference, product, and quotient (formatted to two places, always >= 1.0) of each integer pair.

Sample Input:

```
4
10 4
14 20
25 15
100 34
```

Sample Output:

```
14 6 40 2.50
34 6 280 1.43
40 10 375 1.67
134 66 3400 2.94
```

10. A_Ten - Abundance

Program Name: A_Ten.java **Input File: a_ten.dat**

An abundant number is one whose sum of all proper divisors is greater than the number itself. By definition, a proper divisor is a positive divisor of a number, excluding itself.

The first such number is 12, where 1+2+3+4+6 equals 16, which is greater than 12. Others include 18, 20, 24, 30, and so on. Given an input value N, find and output the largest abundant number less than N.

Input: A data file with several values of N, each on its own line, where `13 <= N <= 100000`.

Output: The largest abundant number less than N.

Sample input:

```
15
20
50
107
4050
```

Sample output:

```
12
18
48
104
4048
```

11. A_Eleven - Block

Program Name: A_Eleven.java **Input File: a_eleven.dat**

This two-player game is played on a standard 8X8 grid, as in checkers or chess. It involves rectangular blocks that span two squares, placed either vertically or horizontally. Each player takes a turn placing a piece, always in the same direction for each player. The first player chooses the desired direction, and the second player uses the other direction.

The idea of the complete game is to be the last player to place a block on the board, preventing the other player from any more possible moves. The (8,1) block is located at the top left corner of the board, at row 8, column 1, and each position in the data represents a (row, column) location in the grid.

Below is a sample board after 5 moves by each player. The **H** player (the one with horizontal pieces), owns the blocks indicated by "**" at positions (3,3), (4,5), (5,1), (6,2) and (8,1), and has the next move.
The **V** blocks, shown as two vertically stacked "#" signs, are indicated by the lower of the two positions, which means there are

*	*		#				
			#				
	*	*	#		#		#
*	*		#		#		#
				*	*	#	
		*	*			#	

V blocks at positions (3,7), (5,4), (5,6), (5,8) and (7,4). The "#" signs at (7,4) and (8,4) indicate the topmost **V** block.

In the example shown, based on the first data sample, your job is to calculate onto how many places of the board an "**H**" block can be placed next. The first letter of the data sample indicates the player whose move is next.

For example, on the top row, an "H" block could be placed at position (8,5), (8,6), or (8,7), each position indicating the left-most of the two spaces required. In other words, an H piece placed at position (8,7) would also cover (8,8). There are 27 possible placements in this situation for the next move for an "H" block as the next move - 3 on the top row, 5 on the next, none on row 6 or 5, 3 on row 4, 2 on row 3, and 7 each on the bottom two rows.

Input: A data file with several sets of data. Each data set begins with the letter V or H, indicating which player went first AND whose turn it is next, followed by an integer N indicating how many pieces each player has played, followed by N pairs of numbers indicating marker positions for the first player, and then N more marker positions for the second player. The first line of sample data below indicates the board shown above.

Output: An integer M indicating the number of possible next moves for the first player.

Sample Input:

```
H 5 3 3 4 5 5 1 6 2 8 1 3 7 5 4 5 6 5 8 7 4
V 8 6 1 6 3 6 5 6 7 3 1 3 3 3 5 3 7 2 1 5 1 5 3 5 5 5 7 8 2 8 4 8 6
H 8 2 1 5 1 5 3 5 5 5 7 8 2 8 4 8 6 6 1 6 3 6 5 6 7 3 1 3 3 3 5 3 7
```

Sample Output:

```
27
18
12
```

12. A_Twelve – Recognition

Program Name: A_Twelve.java **Input File: a_twelve.dat**

Rodney has been given a bunch of scanned images. He is tasked with determining if these scanned images are pictures of the same thing or location. To do this, he has assigned you to write a program that, given all of these scanned images, finds the largest sub-image that appears in all of the given images. To complicate matters, Rodney knows that these images weren't all taken under the exact same conditions. Unfortunately, this means that the positions and lighting of each sub-image may change from image to image, so you'll need to account for those variations.

This means that the sub-image can be at any location within the given picture, and that their color values might differ by small amounts. Rodney has further given you a "quality compensation" factor. This quality compensation determines how closely the colors need to match. A higher quality compensation value means that the images are not as good of a match and require more leeway in considering them.

Each of the images Rodney has given you is represented by a square, 2D array of integers. These integers are between `0` and `255` and are grouped together in triplets representing the *red, green, blue* values for each pixel in the image. Each group of 3 integers in a row represents one addressable pixel. So a 4-pixel image with 2 diagonal black pixels `(0 0 0)` and 2 diagonal white pixels `(255 255 255)` would be represented by:

0 0 0	255 255 255
255 255 255	0 0 0

Consider the following three 2X2 pixel images:

Image A

0 0 0	0 0 0
130 130 130	1 5 8

Image B

2 2 2	200 200 2
1 5 8	3 3 3

Image C

157 88 193	1 5 8
45 190 203	150 40 17

Suppose Rodney has set the quality compensation value for this batch of images to be 0.0. That would mean that any sub-image with all pixels whose corresponding red, green, and blue values differ by no more than 0.0 (i.e., they are an exact match) are considered to be identical. In this case, all 3 images contain a matching sub-image (1, 5, 8):

Image A	`(1, 5, 8)`	← 1x1 sub-image in lower-right
Image B	`(1, 5, 8)`	← 1x1 sub-image in lower-left
Image C	`(1, 5, 8)`	← 1x1 sub-image in upper-right

In this next example, the quality compensation value is set to 2.0, indicating that the corresponding red, green, and blue values for each pixel of a sub-image may differ by no more than 2.0. The qualifying sub-image match for each example is shown to the right. In the closest sub-image match, indicated by the grey boxes, notice how the red values **[1, 3 and 1]** differ by no more than 2.0, and the green values **[5, 5 and 5]** are a perfect match. However, the blue values **[8, 6, and 9]** DO NOT all fall within the stated quality compensation of 2.0, since the greatest span is 3.0 between the 6 and 9, therefore the output is **NONE.**

Image A

0 0 0	5 5 5
130 130 130	1 5 8

Image B

3 5 6	200 200 2
10 10 10	3 3 3

NONE

Image C

157 88 193	1 5 9
45 190 203	150 40 17

To make it easier on you, Rodney has cropped and scaled all the given images such that all the given images are always square and the common sub image is also guaranteed to be square. Help Rodney find the largest sub-image that appears in all the given images within the quality compensation.

Input: The first integer, `i`, will be the number of data sets to follow. Each data set will begin with a floating-point number, `t`, that represents the quality compensation value given by Rodney. The next integer, `q`, will be the dimension in the image in pixels (representing a `q × q` image). The next integer, `j`, will be the number of images that Rodney has given you. The next `q` rows will be an image. Each row will contain `q` groups of 3 images, representing the *red*, *green*, and *blue* values of each pixel. Each image will be followed by a blank line. There will be no more than 4 images considered per data set, and the maximum image square matrix size will be 8 × 8.

Output: The largest sub-image found as it appears in each image. Each pixel in the sub-image should be represented with `(r, g, b)` values with parentheses as shown in the sample sutput below. There should be `j` sub-images output. If no match is found among all of the sub-images being considered within the quality compensation given by Rodney, output NONE. Each sub-image output is to be followed by a blank line, and each complete output will also be followed by the string "-----".

Sample Input:

```
3
0.0
2
3
0 0 0 0 0 0
130 130 130 1 1 1

2 2 2 200 200 2
1 1 1 3 3 3

157 88 193 1 1 1
45 190 203 150 40 17

1.0
2
3
0 0 0 5 5 5
130 130 130 1 1 1

2 2 2 200 200 2
10 10 10 3 3 3

157 88 193 1 1 1
45 190 203 150 40 17

2.0
2
3
0 0 0 5 5 5
130 130 130 1 5 8

3 5 6 200 200 2
10 10 10 3 3 3

157 88 193 1 5 9
45 190 203 150 40 17
```

Sample Output:

```
(1, 1, 1)

(1, 1, 1)

(1, 1, 1)

-----
(1, 1, 1)

(2, 2, 2)

(1, 1, 1)

-----
NONE

-----
```