# KU ACM
# Intramural Programming Competition

April 18, 2015

INSTRUCTIONS: The KU ACM Intramural Programming Competition is an ACM-styled competition in which 10 problems, in roughly ascending difficulty, will be administered in a period of five hours. You may complete the problems in any order you wish. The following languages are supported:

C (gcc 4.8.2)
C++ (g++ 4.8.2)
C++ 11 (g++ 4.8.2)
Haskell (ghc 7.6.3)
Java (Java 1.7.0)
JavaScript (Node 0.10.25)
Lua (5.2.3)
PHP 5
Python2.7
Python3.4
Racket 5.3.6
Ruby 1.9.3

# 1 CAESAR SALAD

DESCRIPTION: Egads! Vegetables are revolting! Although they may not be appetizing in general, vegetables all across the world are rising up against chefs, farmers, and simple backyard gardeners. In the war against all that which is neither animal nor mineral, commanders have requested that you write a program to encrypt all military transmissions. Thankfully, vegetables are none too clever, so a relatively simple cipher should be sufficient.

INPUT: Read from `stdin` an integer $n$ representing the number of lines of input. Each of the subsequent $n$ lines will represent a message to be encrypted. Each message will consist only of uppercase letters and spaces, with no punctuation.

OUTPUT: Write to `stdout` the encrypted form of each message. To encrypt a message, shift each letter in the message up the alphabet by 13 places. Should a letter in the message shift past the end of the alphabet, the shifting should continue from the front of the alphabet. For example, *A*, the first letter of the alphabet, would become *N*, the fourteenth letter, but *T*, the twentieth letter of the alphabet, would become *G*, the seventh letter of the alphabet. Each of the encrypted messages should be separated by a newline.

```
Sample Input:
3
POTATO PEOPLE HAVE STOLEN ALL RUTABAGAS STOP
CLAIM ACT AS VEGETABLE LIBERATION STOP
WE NO LONGER HAVE ANY RUTABAGAS STOP

Sample Output:
CBGNGB CRBCYR UNIR FGBYRA NYY EHGNONTNF FGBC
PYNVZ NPG NF IRTRGNOYR YVORENGVBA FGBC
JR AB YBATRE UNIR NAL EHGNONTNF FGBC
```

## 2 PAPERS, PLEASE

DESCRIPTION: You have been given the most important job of examining entry papers as a proud border guard of Annexia. There are numerous restrictions on passports that may deny one entry into the country. In order to simplify the process of inspecting documents, you decide to write a program to analyze them for you. Remember, only you can defend Annexia's glorious state of permanent nation-wide revolution.

INPUT: Read from `stdin` an integer $n$ representing the number of lines of input. Each of the subsequent $n$ lines will represent a document to inspect. Each document will be in the following format:

*Last-name First-name ID-number Home-country*

OUTPUT: For each document, write to `stdout` your reply to the document-bearer after having inspected their documents. A document-bearer will be denied entry if either of two cases are true: the digits in their identification number sum to 25, or the person's *Home-country* is any one of the following, *BYTELANDIA, LEROY, JENKINS, FLATLAND*. If either of these conditions are true, you should write `ENTRY DEINED` to `stdout`. Otherwise, if the person can be admitted, give them a friendly greeting of the form

`CAUSE NO TROUBLE` *First-name Last-name*

using their first and last name given in the document. Each reply should be separated by a newline and should be in all uppercase letters.

```
Sample Input:
3
LEE BRUCE 1234564 AMN
SAGAN CARL 3828321 LEROY
ROGERS MISTER 8277471 DALMASKA

Sample Output:
ENTRY DENIED
ENTRY DENIED
CAUSE NO TROUBLE MISTER ROGERS
```

## 3 HARMONIOUS GEOMETRY

DESCRIPTION: Out of all your peers, you have been invited to join the greatest of the great Computer Scientists in the fabled Ivory Tower. The computer scientists of the Ivory Tower spend their time on many noble pursuits, such as proving theorems, writing programs in Haskell[1], and musing on which geometric forms they find pleasing. Before you can become a full-fledged member, they want you to prove yourself by writing a simple program to determine if a geometric form as represented using ASCII characters is `Harmonious` or `Cacophonous`.

INPUT: Read from `stdin` a sequence of lines of ASCII `*` (star) characters that represent a single geometric figure. You are guaranteed that there will be no more than 101 lines of characters in a figure, and the length of each line will be no greater than 101 characters. The end of input will be denoted by a `0` character.

OUTPUT: Write to `stdout` whether the figure is `Harmonious` or `Cacophonous`. A geometric figure is considered `Harmonious` if both the number of rows in each figure, as well as the number of characters in each row, are *prime*[2] numbers. If it is not the case that a figure is `Harmonious`, its form is considered cosmically displeasing, and your program should state that it is `Cacophonous` instead.

```
Sample Input:
*
***
*****
0

Sample Output:
Harmonious

Sample Input:
***
***
****
0

Sample Output:
Cacophonous
```

NOTE: Your program should be written to evaluate only one figure at a time. However, multiple separate test cases may be run against your program.

[1] By the Curry-Howard Correspondence, these two are apparently the same thing.
[2] Remember that a number is *prime* if and only if its only whole-number factors are 1 and the number itself.

# 4 The Count Crisis

Figure 4.1: *"One problem, two problems, three problems, four problems! AH AH AH"*

Description: You love to count, almost as much as the Count himself. But you're getting older, and your mind isn't quite as sharp as it used to be. As a result, you sometimes forget a number when you're counting, which is simply unacceptable to you. For example, suppose you began counting by 2, starting from 0. You start counting: "0, 2, 4, 8, 10, 12..." But you notice something has gone horribly wrong. Where was 6?! "6," you quickly interject. Phew. Balance has been restored to the universe.

This makes a good illustration of the problem. You always count in integer sequences such that there is a constant integer difference between terms in the sequence. However, somewhere down the line, you will forget to say exactly one term in the sequence. This term will always be somewhere in the middle of the sequence. That is, the term you omitted will never come before the first term you stated, nor can it be said you omitted a term after the last term you stated. The omitted term will always be bounded by the first and last terms of the sequence.

Input: Read from `stdin` a positive integer $n$, followed by a newline. The subsequent $n$ lines will have the following form:

$$m\ p_1\ p_2 \ldots p_m$$

where $m$ is an integer such that $3 \leq m < 1024$, and $p_1 \ldots p_m$ is a sequence of $m$ space delimited integers representing the sequence of counting numbers as described above. Be aware that any of the $p_1$ through $p_m$ may be negative, and the constant difference between the integers of a sequence may also be negative.

OUTPUT: For each of the $n$ lines, write to `stdout` the integer in that line's sequence that was omitted.

```
Sample Input:
3
3 15 10 0
4 1 2 3 5
5 -6 -3 3 6 9

Sample Output:
5
4
0
```

# 5 SCAN N' SPEAK

DESCRIPTION: Long numbers are sometimes difficult to read, so we often just read out their digits instead. A new system is made that will read long numbers to people, but it requires that the numbers be fed to it in a certain format. For the system to correctly read `1` (as in the number "one"), it must be given `11` (as in "one 1"). For it to correctly read `2222` (as in the number "two thousand two hundred twenty-two"), it must be given `42` (as in "four 2s"). Write a program that will format regular numbers so that this somewhat quirky, admittedly useless system can read them correctly.

INPUT: Read from `stdin` a positive integer $n$, followed by a newline. The subsequent $n$ lines will be strings of digits which must be translated to the program's format.

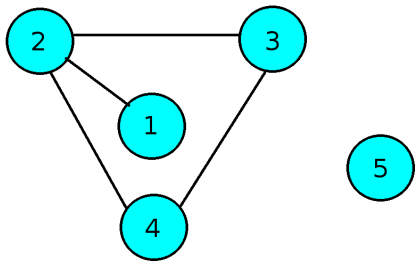
OUTPUT: For each number write to `stdout` the newly converted number, and separate each with a newline.

```
Sample input:
3
5421
5478885113
01112111122234

Sample output:
15141211
15141738152113
10311241321314
```

## 6 THE NEIGHBORS OF DINKLEBERG

DESCRIPTION: You're an upstart urban planner in the metropolis of Dinkleberg, and you have been tasked with the designing of a system for your developing cityscape. Your bosses are very concerned about traffic congestion, so you are required to deliver a report on the traffic nearby cities might experience based on a given road layout. Here is one example of a city plan, drafted using the latest state-of-the-art blueprinting software:




In this diagram, blue circles represent cities, and straight lines represent highways. Two cities are *neighbors* if they are directly connected by a single highway (e.g., cities 1 and 2 are neighbors, but cities 1 and 3 are not neighbors). A city's *metropolis number* is its number of neighbors (e.g., city 2 has a metropolis number of 3). One statistic your bosses want is a city's *total metropolis number neighbor sum (TMNNS)*, which is the sum of the metropolis numbers of each of a given city's neighbors. For example, in the diagram above, city 2 has three neighbors, cities 1, 3, and 4, which have metropolis numbers of 1, 2, and 2 respectively, giving city 2 a TMNNS of $1 + 2 + 2 = 5$. If a city has no neighbors (e.g., city 5), then its TMNNS is defined to be 0.

INPUT: Read from `stdin` a positive integer $n$, followed by a newline. The subsequent $n$ lines will have the following format:

$$c\ h\ e_1 \ldots e_{2h}$$

where $c$ is the number of cities, $h$ is the number of highways, and $e_1 \ldots e_{2h}$ is a sequence of $2h$ cities, separated by spaces, where each group of two cities in the sequence is connected by a highway. For example, the above blueprint would be specified by the following:

```
5 4 1 2 2 3 3 4 2 4
```

since it has 5 cities and 4 highways, where highways connect cities 1 and 2, cities 2 and 3, cities 3 and 4, and cities 2 and 4. You may assume that the city numbers are always $1, 2, \ldots, c$.

OUTPUT: For each of the $n$ lines, print the TMNNS for each city in ascending numeric order (i.e., starting with city 1, then city 2, up to city $c$), putting a space after each TMNNS. Separate each of the $n$ cases by a newline.

```
Sample Input:
2
3 3 1 2 2 3 1 3
5 4 1 2 2 3 3 4 2 4

Sample Output:
4 4 4
3 5 5 5 0
```

# 7 LETTER CYCLES

DESCRIPTION: Consider a code in which words are strings consisting entirely of A's and B's. Codes have a fixed word length, and every pair of *consecutive* words differ by exactly one character (including the first and last words, making them consecutive as well). For example, a code of length two:

*AA, AB, BB, BA*

It is easy to verify that this satisfies the code properties, since

- AA and AB differ only by the second character
- AB and BB differ only by the first character
- BB and BA differ only by the second character
- BA and AA differ only by the first character

Note that codes may be shifted cyclically, or even reversed and still maintain the previously stated property (e.g. [AA, AB, BB, BA] can be cyclically shifted to [AB, BB, BA, AA] or even reversed to [BA, BB, AB, AA]). But other orderings will not necessarily satisfy the property (e.g. [AA, BB, AB, BA], because AA and BB do not differ by exactly one character). Because of this property, for any length of A's and B's, allowing for shifting or reversals, there is one way and only one way to represent the code stream. As another example, here is a code of length three:

*AAA, AAB, ABB, ABA, BBA, BBB, BAB, BAA*

Your task is to write a program that will determine if two arbitrary words in the code are consecutive.

INPUT: Read from `stdin` a positive integer $n$ followed by a newline. The subsequent $n$ lines will have the following format:

$$m\ w_1\ w_2$$

followed by a newline, where $m$ is the character length of the code, $w_1$ is a code word of length $m$, and $w_2$ is another code word of length $m$. You are guaranteed that $3 \leq m \leq 18$.

OUTPUT: For each of the $n$ lines, write to `stdout` the word `Consecutive` if $w_1$ and $w_2$ are consecutive code words in the code stream, otherwise write `Not Consecutive`. Separate each case with a newline.

```
Sample input:
4
2 AA AB
2 AA BB
3 BAA AAA
3 AAA ABA

Sample output:
Consecutive
Not Consecutive
Consecutive
Not Consecutive
```

## 8 THAT ONE LEVEL BEFORE THE WATER TEMPLE

DESCRIPTION: Link needs those dang Iron Boots to get to the bottom of Lake Hylia, so he's going through that honestly pretty forgettable ice cavern level[3] to get them. Before he can have another frustrating reunion with Sheik, he has to navigate an ice maze. The ice is so slick however, he can't just take one step. If he does, he simply slides until he hits an object, and his movement is too fast to grab any object to stabilize as he passes by. To help him get through the maze, and one step closer to defeating the evil Ganondorf, write a program to find the shortest path through the icy maze.

INPUT: Read from `stdin` a line of two space separated integers, $N$ and $M$, followed by a newline, which represents a cavern of dimensions $N$ *x* $M$. The following $N$ lines, each of length $M$ will consist of the characters *X*, *L*, *G*, and the `space` character, ' '. *X* denotes an barrier that will stop Link if he is currently moving, and that he cannot pass through. *L* denotes Link's starting position, *G* denotes the end of the maze, and a `space` character denotes open space over which Link can slide. You may assume that the outer edges of the rectangle are walls which Link cannot traverse.

OUTPUT: Write to `stdout` the minimum number of moves necessary for Link to reach the exit. Note that turning is not considered a move, only a slide over one or more spaces is considered a move. He can also only move `up`, `down`, `left`, and `right`.

```
Sample input:
4 4
L
 X X

  XG

Sample output:
5
```

[3] You know the one.

# 9 THE ANNIHILATOR OF AMBIGUITY

DESCRIPTION: Being social is hard. One contributor to that reality is the inherent ambiguity in language. There can be multiple ways to say the same thing[4], and oftentimes, one thing can mean many things. Traveling to a more familiar topic, algebraic infix expressions can also sometimes be ambiguous. Thankfully, parentheses can remove ambiguity in an algebraic expression. Since we can't parenthesize human beings[5], we'll parenthesize Boolean expressions instead. Consider the two Boolean values (`T` for *True* and `F` for *False*), and three logical operators, `AND`, `OR`, and `XOR`, represented by `&`, `|`, and `^` respectively. The truth tables for these operators are given below:

| AND (&) | | |
|---|---|---|
| *Inputs* | | *Outputs* |
| T | T | T |
| T | F | F |
| F | T | F |
| F | F | F |

| OR (\|) | | |
|---|---|---|
| *Inputs* | | *Outputs* |
| T | T | T |
| T | F | T |
| F | T | T |
| F | F | F |

| XOR (^) | | |
|---|---|---|
| *Inputs* | | *Outputs* |
| T | T | F |
| T | F | T |
| F | T | T |
| F | F | F |

For this problem, you should not assume anything about the precedence of &, |, or ^. This means that parenthesizing the expression is extremely important. For example, the expression *T ^T & F* can be evaluated two ways: *(T ^T) & F*, which evaluates to F, but *T ^(T & F)* evaluates to T. We are only interested in the number of distinct ways you can parenthesize an expression such that it evaluates to true.

INPUT: Read from `stdin` a positive integer *n*, followed by a newline. The subsequent *n* lines will consist of input in the following format:

*x b o*

where *x* is the number of Boolean values, *b* is a sequence of *x* Boolean values, and *o* is a sequence of *(x - 1)* logical operators, followed by a newline. Logical operators will always be inserted between Boolean values in the order they appear. For example, the line

```
3 TTF ^&
```

would represent the logical expression *T ^T & F* previously examined. It can always be assumed that $2 < x \leq 40$.

[4] Unlike Python, in which I've been told "There's only one way to do it."
[5] If only…

OUTPUT: For each logical expression, write to `stdout` the number of distinct ways the expression can be parenthesized such that it evaluates to *T*.

```
Sample input:
2
3 TTF ^&
4 TTFT |&^

Sample output:
1
4
```

## 10 THE GREATEST THING SINCE LEADED CRIBS

DESCRIPTION: Noah Webster helped popularize conventional dictionaries, but now it's your turn to help popularize an entirely new type of dictionary. Specifically, you are going to help anagram dictionaries take the world by storm. Anagrams are equal-length words composed of the same characters, but in different orders. For example, "bat" and "tab", "lake" and "kale", and "rise" and "sire" are all pairs of anagrams[6]. However, why restrict yourself to conventional words? Your dictionary will include all possible anagrams using a set of characters, regardless of whether or not it is a so-called "real" word. For example, using the characters in "bat", we can produce 6 different anagrams. In alphabetical order, they are "abt", "atb", "bat", "bta", "tab", and "tba".

This may prove to be a difficult task to perform by hand, since there are a *lot* of anagrams out there (From "mississippi", one can make 34,650 distinct anagrams alone). To make it easier, you decide to adopt a numbering system to track how anagrams are sorted alphabetically with respect to other anagrams that use the same characters. Using the "bat" example above, we would give "abt" number 1, "atb" number 2, and so on until "tba", which gets number 6.

INPUT: Read a positive integer $n$, followed by a newline. The next $n$ lines will consist of all-lowercase strings, of length 1 to 20 (inclusive), with no spaces, each followed by a newline.

OUTPUT: For each of the $n$ words, output the number of the anagram if it were put in alphabetical order with respect to its anagram siblings. Always index your anagrams starting with 1.

```
Sample input:
3
lee
bruce
banner

Sample output:
3
17
66
```

[6] "leaded cribs" is apparently an anagram of "sliced bread," but the administrators of this competition in no way mean to endorse the making of cribs from lead.