

# 2011 Stanford Local ACM Programming Contest

Saturday, October 8th, 2011

## Read these guidelines carefully!

### Rules

1. You may use resource materials such as books, manuals, and program listings. You may *not* search for solutions to specific problems on the Internet, though you are permitted to use online language references (e.g., the Java API documentation) and digital versions of textbooks or course notes. However, you may not electronically copy from machine-readable versions existing code or data. That is, all programs submitted must be *manually typed in their entirety during the contest*. No cutting and pasting of code is allowed!
2. You may bring and use your own laptop to write your code for this contest. To keep the computing environments as fair as possible across contestants, we ask that you restrict your use of software during the contest to text editors, IDEs, document readers, and a browser for the sole purpose of connecting to the contest site (and perhaps a language reference). In other words, please refrain from using programs like Mathematica, Maple, or other specialized tools if you have them installed. You may also use a simple electronic calculator (or the software one that comes pre-installed on your OS), but no fancy graphing calculators please!
3. You may not collaborate in any way (verbally, electronically, in writing, using gestures, telepathically, etc.) with other contestants, students, or anyone else during the contest.
4. You are expected to adhere to the honor code. You are still expected to conduct yourself according to the rules, even if you are not participating on site in the Gates building.

### Guidelines for submitted programs

1. All programs must be written in C, C++, or Java. For judging, we will compile the programs in the following way:
  - `.c`: using `gcc -O2 -lm` (GCC version 4.1.2)
  - `.cc`: using `g++ -O2 -lm` (GCC version 4.1.2)
  - `.java`: using `javac` (Sun Java version 1.6.0)

All programs will be compiled and tested on a Leland `myth` machine. The `myth` machines are Intel Core2 Duo 3.16 GHz machines with 4 GB RAM running CentOS Linux 5.3. Compilation errors or other errors due to incompatibility between your code and the `myth` machines will result in a submission being counted incorrect.

2. Make sure you `return 0;` in your `main()`; **any non-zero return values may be interpreted by the automatic judge as a runtime error.**
3. **Java users:** Please place your `public static void main()` function in a public class with the same name as the base filename for the problem. For example, a Java solution for the `test` program should be submitted in the file `test.java` and should contain a `main()` in `public class test`.

4. All solutions must be submitted as a single file.
5. All programs should accept their input on **stdin** and produce their output on **stdout**. They should be batch programs in the sense that they do not require human input other than what is piped into **stdin**.
6. Be sure to follow the output format described in the problem exactly. We will be judging programs based on a **diff** of your output with the correct solution, so your program's output must match the judge output **exactly** for you to receive credit for a problem. As a note, each line of an output file must end in a newline character, and there should be no trailing whitespace at the ends of lines.

### How will the contest work?

1. If you chose to work remotely from a home computer, we recommend that you test out your account on the online contest system by submitting a solution for the test problem shown on the next page. We will do our best to set up the contest host to accept test problem submissions Saturday afternoon until approximately 1:45 pm.
2. For those who choose to participate onsite, from 1:00 to 1:45 pm, you should select a computer (or find a place to plug in your laptop), set up your workspace and complete a test problem. Space in Gates B02 and the PUP cluster is limited, and will be available on a first-come first-served basis. You may also choose to work directly on one of the **myth** machines in Gates B08, although technically we do not have that room reserved for the contest.
3. At 2:00 pm, the problems will be posted on the live contest page in PDF format, all registered participants will be sent an e-mail that the problems have been posted, and we will distribute paper copies of the problems to contestants competing in either Gates B02 or the PUP cluster.
4. For every run, your solution will be compiled, tested, and accepted or rejected for one of the following reasons: *compile error*, *run-time error*, *time limit exceeded*, *incorrect output*, or *presentation error*. In order to be accepted, your solution must match the judge output exactly (according to **diff**) on a set of hidden judge test cases, which will be revealed after the contest.
  - Source code for which the compiler returns errors (warnings are ok) will be judged as *compile error*.
  - A program which returns any non-zero error code will be judged as *run-time error*.
  - A program which exceeds the time allowed for any particular problem will be judged as *time-limit exceeded* (see below).
  - A program which fails a **diff -w -B** will be judged as *incorrect output*.
  - A program which passes a **diff -w -B** but fails a **diff** (i.e., output matches only when ignoring whitespace and blank lines) will be judged as *presentation error*.
  - A program which passes a **diff** and runs under the time constraints specified will be judged as *accepted*.
5. For each problem, the time allowed for a run (consisting of multiple test cases) will be 10 seconds total for all test cases. The number of test cases in a run may vary from 20 to 200 depending upon the problem, so be sure to write algorithmically efficient code!
6. You can view the status of each of your runs on the live online contest site. Please allow a few minutes for your submissions to be judged. The site also provides a live scoreboard for you to watch the progress of the contest as it unfolds.

7. At 6:00 pm, the contest will end. No more submissions will be accepted. Contestants will be ranked by the number of solved problems. Ties will be broken based on total time, which is the sum of the times for correct solutions; the time for a correct solution is equal to the number of minutes elapsed since 2:00 pm plus 20 penalty minutes per rejected solution. No penalty minutes are charged for a problem unless a correct solution is submitted. After a correct submission for a problem is received, all subsequent incorrect submissions for that problem do not count towards the total time.
8. The results of this contest will be used in part to select team members for representing Stanford at the forthcoming ACM regional competition. Six or more contestants have been customarily invited to the Stanford ACM ICPC teams in previous years.

### Helpful hints

1. **Make sure your programs compile and run properly on the myth machines.** If you choose not to develop on the Leland systems, you are responsible for making sure that your code is portable.
2. **Read (or skim) through all of the problems at the beginning to find the ones that you can code quickly.** Finishing easy problems at the beginning of the contest is especially important as the time for each solved problem is measured from the beginning of the contest. Also, check the leaderboard frequently in order to see what problems other people have successfully solved in order to get an idea of which problems might be easy and which ones are likely hard.
3. If you are using C++ and unable to get your programs to compile/run properly, try adding the following line to your `.cshrc` file

```
setenv LD_LIBRARY_PATH /usr/pubsw/lib
```

and re-login.

4. The **myth** machines in Gates B08 are not officially reserved for the contest, but these will be the machines used for judging/testing of all programs. You may find it helpful to work on these machines in order to ensure compatibility of your code with the judging system.
5. If you are a CS major and have a working **xenon** account, please work in the **PUP cluster** rather than Gates B02; the PUP cluster has UNIX machines, which may be a more convenient programming environment if you intend to use Emacs, etc. If you don't know where the PUP cluster is, just ask!
6. If you are working on a PC in Gates B02, it may be helpful to run a VNC session if you don't like coding from a terminal. Check out the IT services page on using VNC, which can be found at <http://unixdocs.stanford.edu/moreX.html>. If you wish to use an IDE (e.g., Visual Studio or Eclipse), please make sure that you know how to set this up yourself beforehand. We will not be able to provide technical support related to setting up IDEs during the contest.
7. If you need a clarification on a problem or have any other questions, post an clarification request to the live contest page, or just come talk to us in **Gates B02** or the **PUP cluster**.

*The directions given here are originally based on those taken from Brian Cooper's 2001 Stanford Local Programming Contest problem set, and have been updated year after year to the best of our ability. The contest organizers would like to thank the problem authors of 2011, in alphabetical order, Jaehyun Park, Jeffrey Wang, Jonathan Lee, Ralph Furmaniak, and Sonny Chan.*

## 0 Test Problem

### 0.1 Description

This is a test problem to make sure you can compile and submit programs. Your program for this section will take as input a single number  $N$  and return the average of all integers from 1 to  $N$ , inclusive.

To submit a solution, navigate your browser to the test contest page, which this year resides at:

<http://cs.stanford.edu/group/acm/slpctest/>

Note that this URL is different from the live contest URL used for the real contest. To submit problems during the actual contest, go to <http://cs.stanford.edu/group/acm/slpclive/>.

You must log in using your assigned user ID and password for the contest. If you have registered for the contest, but have not received an email message containing your login information, then it's time to contact one of the contest organizers in panic! Students in Gates B02 or the PUP cluster will also be given their login information on paper before the contest.

Use the "Submissions" tab on the contest page to submit your solution to the problem. Detailed instructions can be found in the "Help" section online if necessary.

After submitting your solution, please allow a few minutes for it to be judged. You can resubmit rejected solutions as many times as you like (though incurring a 20 minute penalty for each rejected run of a problem you eventually get right). Once you have submitted a correct solution, future submissions of that problem will still be graded but will not count towards your final score or total time.

Note that you do not need to submit this problem during the actual contest!

### 0.2 Input

The input test file will contain multiple test cases. Each test case is specified on a single line containing an integer  $N$ , where  $-100 \leq N \leq 100$ . The end-of-file is marked by a test case with  $N = -999$  and should not be processed. For example:

```
5
-5
-999
```

### 0.3 Output

The program should output a single line for each test case containing the average of the integers from 1 to  $N$ , inclusive. You should print numbers with exactly two decimal places. For example:

```
3.00
-2.00
```

## 0.4 Sample C Solution

```
#include <stdio.h>

int main() {
    int n;
    while (1) {
        scanf("%d", &n);
        if (n == -999) break;
        if (n > 0) printf("%.2lf\n", (double)(n * (n + 1) / 2) / n);
        else printf("%.2lf\n", (double)(1 + n * (1 - n) / 2) / (2 - n));
    }
    return 0;
}
```

## 0.5 Sample C++ Solution

```
#include <iostream>
#include <iomanip>
using namespace std;

int main() {
    cout << setprecision(2) << setiosflags(ios::fixed | ios::showpoint);
    while (true) {
        int n;
        cin >> n;
        if (n == -999) break;
        if (n > 0) cout << double(n * (n + 1) / 2) / n << endl;
        else cout << double(1 + n * (1 - n) / 2) / (2 - n) << endl;
    }
    return 0;
}
```

## 0.6 Sample Java Solution

```
import java.util.*;
import java.text.DecimalFormat;

public class test {
    public static void main(String[] args) {
        Scanner s = new Scanner(System.in);
        DecimalFormat fmt = new DecimalFormat("0.00");
        while (true) {
            int n = s.nextInt();
            if (n == -999) break;
            if (n > 0) System.out.println(fmt.format((double)(n * (n + 1) / 2) / n));
            else System.out.println(fmt.format((double)(1 + n * (1 - n) / 2) / (2 - n)));
        }
    }
}
```

# 1 Translation

The alien language of Palinnilap revolves around the use of palindromes as their words. You are working on translating a Palinnilap text but have discovered that the text contains some errors, i.e. words that are not palindromes. You decide to correct these by replacing the words with the longest substring (contiguous set of characters) which is a palindrome, i.e. for the word “EINXLLXNABA” you would replace it with “NXLLXN”. If multiple substrings are tied for the longest, choose the one that occurs first in the original word. Since this is quite a tedious task, write a program to do it for you.

## 1.1 Input

The input consists of multiple test cases. Each test case is a single line containing a word that is made up of only uppercase letters of the alphabet and has at most 100 characters. The input terminates with a line containing “END”. For example:

```
EINXLLXNABA
ABCBDABE
C
END
```

## 1.2 Output

For each test case, print a single line that contains the longest substring of the word which is a palindrome (breaking ties by choosing the one which occurs first in the original word). For example, the correct output for the sample input above would be:

```
NXLLXN
BCB
C
```

## A Another Rock-Paper-Scissors Problem

Sonny uses a very peculiar pattern when it comes to playing rock-paper-scissors. He likes to vary his moves so that his opponent can't beat him with his own strategy.

Sonny will play rock (R) on his first game, followed by paper (P) and scissors (S) for his second and third games, respectively. But what if someone else is using the same strategy? To thwart those opponents, he'll then play paper to beat rock, scissors to beat paper, and rock to beat scissors, in that order, for his 4th through 6th games. After that, he'll play scissors, rock, and paper for games 7–9 to beat anyone copying his last set of moves. Now we're back to the original order—rock, paper, scissors—but instead of being predictable and using the same moves, do you know what would be better? You guessed it! Sonny then plays the sequence of moves that would beat anyone trying to copy his whole strategy from his first move, and on it goes...

To recap, in symbolic form, Sonny's rock-paper-scissors moves look like this:

R P S PSR SRP PSRSRPRPS SRPRPSPSR PSRSRPRPSSRPRPSPSRRPSPSRSRP ...

The spaces are present only to help show Sonny's playing pattern and do not alter what move he'll play on a certain game.

Naturally, your job is to beat Sonny at his own game! If you know the number of the game that you'll be playing against Sonny, can you figure out what move you would need to play in order to beat him?

### A.1 Input

Each line of the input contains a single integer  $N$ ,  $1 \leq N \leq 10^{12}$ , the number of the game you'll be playing against Sonny. An integer  $N = 1$  indicates it would be Sonny's first game,  $N = 7$  indicates it would be the 7th game, and so forth. The input terminates with a line with  $N = 0$ . For example:

```
1
7
33
0
```

**Warning:**  $N$  may be large enough to overflow a 32-bit integer, so be sure to use a larger data type (i.e. `long` in Java or `long long` in C/C++) in your program.

### A.2 Output

For each test case, output a single line which contains the letter corresponding to the move you would need to play to beat Sonny on that game. For example, the correct output for the sample input above would be:

```
P
R
S
```

## B Ball Painting

There are  $2N$  white balls on a table in two rows, making a nice 2-by- $N$  rectangle. Jon has a big paint bucket full of black paint. (Don't ask why.) He wants to paint all the balls black, but he would like to have some math fun while doing it. (Again, don't ask why.) First, he computed the number of different ways to paint all the balls black. In no time, he figured out that the answer is  $(2N)!$  and thought it was too easy. So, he introduced some rules to make the problem more interesting.

- The first ball that Jon paints can be any one of the  $2N$  balls.
- After that, each subsequent ball he paints must be adjacent to some black ball (that was already painted). Two balls are assumed to be adjacent if they are next to each other horizontally, vertically, or diagonally.

Jon was quite satisfied with the rules, so he started counting the number of ways to paint all the balls according to them. Can you write a program to find the answer faster than Jon?

### B.1 Input

The input consists of multiple test cases. Each test case consists of a single line containing an integer  $N$ , where  $1 \leq N \leq 1,000$ . The input terminates with a line with  $N = 0$ . For example:

```
1
2
3
0
```

### B.2 Output

For each test case, print out a single line that contains the number of possible ways that Jon can paint all the  $2N$  balls according to his rules. The number can become very big, so print out the number modulo 1,000,000,007. For example, the correct output for the sample input above would be:

```
2
24
480
```

## C City Driving

You recently started frequenting San Francisco in your free time and realized that driving in the city is a huge pain. There are only  $N$  locations in the city that interest you, though, and you have decided to try to improve your driving experience. Since you lack a GPS and cannot remember too many different routes, you wrote down the directions and how long it takes to get between  $N$  different pairs of locations (the same in both directions), ensuring that using only these paths you can get from any location to any other one.

Now you are planning your trip for the weekend and you need to figure out the fastest way to get between  $Q$  pairs of locations in the city using only the routes you have written down.

### C.1 Input

The input consists of multiple test cases. The first line contains a single integer  $N$ ,  $3 \leq N \leq 100,000$ , the number of locations of interest and the number of routes you wrote down. The next  $N$  lines each contain three integers  $u$ ,  $v$ , and  $w$  ( $1 \leq w \leq 1,000$ ), indicating that you have directions from location  $u$  to location  $v$  and vice-versa (0-indexed) which take  $w$  time. The following line contains a single integer  $Q$ ,  $1 \leq Q \leq 10,000$ , the number of pairs of locations you need to compute the travel time for. The next  $Q$  lines each contain two integers  $u$  and  $v$ , indicating that you should find the minimum time to get from location  $u$  to location  $v$ . The input terminates with a line with  $N = 0$ . For example:

```
7
0 1 2
0 2 3
1 3 2
2 3 8
2 4 3
3 5 1
1 6 7
3
4 5
0 6
1 2
0
```

### C.2 Output

For each test case, print out  $Q$  lines, one for each pair of locations  $u$  and  $v$  you are finding the fastest routes for. Each line should simply contain the minimum time it takes to travel from location  $u$  to location  $v$ . For example, the correct output for the sample input above would be:

```
11
9
5
```

## D Drunken Walk

After having a bit too much to drink in the evening, you find yourself going on a long walk on a directed graph—but not too long, as there are no cycles. You start at vertex 0, and whenever you are at a vertex, you will randomly leave the vertex along one of its outgoing edges with probability proportional to the weight of the edge.

You continue walking until you reach a vertex that has no edges leaving it, after which you fall asleep. The length of your walk is the number of edges you traverse. Moreover, before leaving vertex 0, you may choose one edge from anywhere in the graph that you do not like and ignore it during your walk (or you may choose to not ignore any of them). Determine the longest possible expected length of your walk.

### D.1 Input

The input consists of multiple test cases. Each test case begins with a line containing two integers  $N$ ,  $2 \leq N \leq 10,000$ , and  $M$ ,  $1 \leq M \leq 100,000$ , the number of vertices and edges in the graph, respectively. The next  $M$  lines each contain three integers  $u$ ,  $v$ , and  $w$  ( $1 \leq w \leq 1,000$ ), indicating that there is a directed edge from vertex  $u$  to vertex  $v$  (numbered from 0 to  $N - 1$ ) with weight  $w$ . The input terminates with a line with  $N = M = 0$ . For example:

```
4 5
0 1 2
0 2 1
0 3 3
1 3 1
2 3 4
9 8
0 1 1
1 2 1
2 3 1
0 4 2
4 5 10
4 6 1
6 7 1
7 8 1
0 0
```

### D.2 Output

For each test case, print out a single line that contains the longest possible expected length of your walk. Your answer will be considered correct if it is within  $10^{-6}$  absolute or relative precision. In the first sample case, ignoring the edge from vertex 0 to vertex 3 gives the maximum possible expected length of 2. (Without ignoring it, the expected length is 1.5.) The correct output for the sample input above would be:

```
2.00000000
3.66666667
```

## E Empty Triangles

Do you know how easy it is to make a very simple problem into a brutally hard one? Here is an example. How many triangles can you make with  $N$  straight lines in the plane? As long as they have different slopes and no three of them meet at a single point, there will be  $\binom{N}{3}$  triangles, which is the maximum possible you can get.

Okay, that wasn't too bad. But let's see what happens if we only count triangles that are empty (that is, none of the lines pass through the interior of the triangle). Then, the number of triangles suddenly becomes very small. For example, with 4 straight lines, we can only make 2 empty triangles, whereas the total number of triangles can be as big as 4. Refer to the diagram.

In fact, a general formula for the maximum number of empty triangles that can be drawn with  $N$  lines is not known. The hard part, however, is to find the right configuration of the lines. Your job is much easier; given  $N$  straight lines on the plane, count the number of empty triangles.

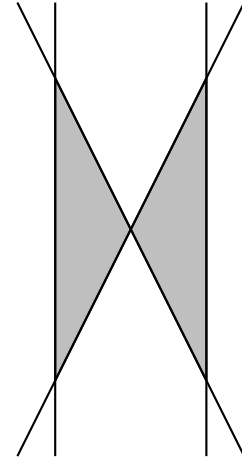


Figure 1: Four lines making two empty triangles (shaded).

### E.1 Input

The input consists of multiple test cases. Each test case begins with a line containing an integer  $N$ ,  $1 \leq N \leq 500$ , which indicates the number of lines on the plane. The next  $N$  lines each contain four integers  $x_1, y_1, x_2,$  and  $y_2$  (between  $-1,000$  and  $1,000$ ), representing a straight line that goes through  $(x_1, y_1)$  and  $(x_2, y_2)$ . It is guaranteed that no three lines meet at the same point, and all the lines are distinct. The input terminates with a line with  $N = 0$ . For example:

```
4
0 0 1 2
0 0 -1 2
1 0 1 1
-1 0 -1 -1
5
0 0 1 0
0 1 1 1
0 2 1 2
0 3 1 3
0 4 1 4
0
```

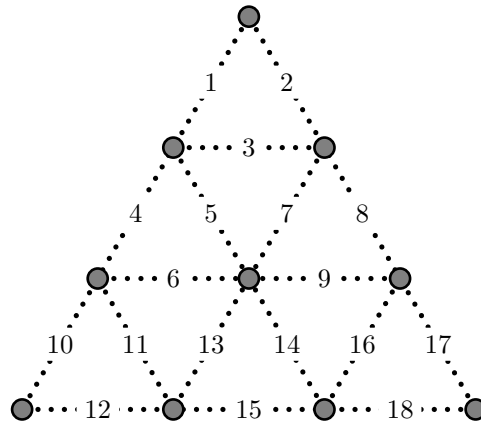
### E.2 Output

For each test case, print out a single line that contains the number of empty triangles formed by the given lines. For example, the correct output for the sample input above would be:

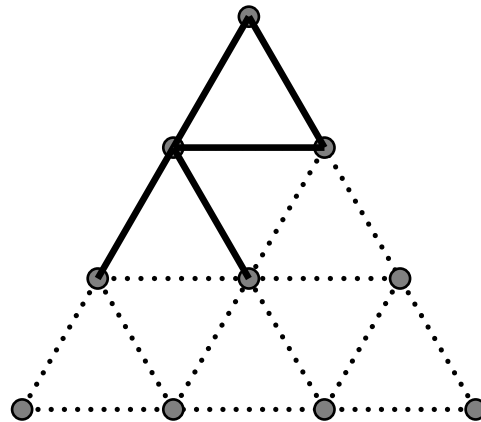
```
2
0
```

## F Fighting for Triangles

Andy and Ralph are playing a two-player game on a triangular board that looks like the following:



At each turn, a player must choose two adjacent vertices and draw a line segment that connects them. If the newly drawn edge results in a triangle on the board (only the smallest ones count), then the player claims the triangle and draws another edge. Otherwise, the turn ends and the other player plays. The objective of the game is to claim as many triangles as possible. For example, assume that it is Andy's turn, where the board has five edges as shown in the picture below. If Andy draws edge 6, then he will claim the triangle formed by edge 4, 5, and 6, and continue playing.



Given a board that already has some edges drawn on it, decide the winner of the game assuming that both Andy and Ralph play optimally. Andy always goes first. Note that if a triangle exists on the board before the first move, neither player claims it.

## F.1 Input

The input consists of multiple test cases. Each test case begins with a line containing an integer  $N$ ,  $5 \leq N \leq 10$ , which indicates the number of edges that are already present on the board before the game begins. The next line contains  $N$  integers, indicating the indices of these edges. The input terminates with a line with  $N = 0$ . For example:

```
6
1 2 3 4 5 6
5
4 5 6 7 8
0
```

## F.2 Output

For each test case, print out a single line that contains the result of the game. If Andy wins, then print out “Andy wins”. If Ralph wins, then print out “Ralph wins”. If both players get the same number of triangles, then print out “Draw”. Quotation marks are used for clarity and should not be printed. For example, the correct output for the sample input above would be:

```
Andy wins
Ralph wins
```

## G Guessing Game

Jaehyun has two lists of integers, namely  $a_1, \dots, a_N$  and  $b_1, \dots, b_M$ . Jeffrey wants to know what these numbers are, but Jaehyun won't tell him the numbers directly. So, Jeffrey asks Jaehyun a series of questions of the form "How big is  $a_i + b_j$ ?" Jaehyun won't even tell him that, though; instead, he answers either "It's at least  $c$ ," or "It's at most  $c$ ." (Right, Jaehyun simply doesn't want to give his numbers for whatever reason.) After getting Jaehyun's responses, Jeffrey tries to guess the numbers, but he cannot figure them out no matter how hard he tries. He starts to wonder if Jaehyun has lied while answering some of the questions. Write a program to help Jeffrey.

### G.1 Input

The input consists of multiple test cases. Each test case begins with a line containing three positive integers  $N$ ,  $M$ , and  $Q$ , which denote the lengths of the Jaehyun's lists and the number of questions that Jeffrey asked. These numbers satisfy  $2 \leq N + M \leq 1,000$  and  $1 \leq Q \leq 10,000$ . Each of the next  $Q$  lines is of the form  $i \ j \ \leq \ c$  or  $i \ j \ \geq \ c$ . The former represents  $a_i + b_j \leq c$ , and the latter represents  $a_i + b_j \geq c$ . It is guaranteed that  $-1,000 \leq c \leq 1,000$ . The input terminates with a line with  $N = M = Q = 0$ . For example:

```
2 1 3
1 1 <= 3
2 1 <= 5
1 1 >= 4
2 2 4
1 1 <= 3
2 1 <= 4
1 2 >= 5
2 2 >= 7
0 0 0
```

### G.2 Output

For each test case, print a single line that contains "Possible" if there exist integers  $a_1, \dots, a_N$  and  $b_1, \dots, b_M$  that are consistent with Jaehyun's answers, or "Impossible" if it can be proven that Jaehyun has definitely lied (quotes added for clarity). The correct output for the sample input above would be:

```
Impossible
Possible
```

## H Hidden Code

It's time to put your hacking skills to the test! You've been called upon to help crack enemy codes in the current war on... something or another. Anyway, the point is that you have discovered the encryption technique used by the enemy; it is quite simple, and proceeds as follows. Note that all strings contain only uppercase letters of the alphabet.

1. We are given a key  $K$  and a plaintext  $P$  which is encrypted character-by-character to produce a ciphertext  $C$  of the same length.
2. If  $|K|$  is the length of the key  $K$ , then the first  $|K|$  characters of  $C$  are obtained by adding the first  $|K|$  characters of  $P$  to the characters of  $K$ , where adding two letters means interpreting them as numbers ( $A = 0, B = 1$ , and so on) and taking the sum modulo 26. That is,  $C_i = (P_i + K_i) \bmod 26$  for  $i = 1, \dots, |K|$ . If  $|K| > |P|$ , then the extra characters in  $K$  are ignored.
3. The remaining characters of  $P$ , i.e.  $P_i$  for  $i > |K|$ , are encrypted using the previous ciphertext characters by  $C_i = (P_i + C_{i-|K|}) \bmod 26$  for  $i = |K| + 1, \dots, |P|$ .

As an example, consider the encryption of the string "STANFORD" using the key "ACM":

```
STA  NFORD
+ ACM SVMFA
-----
SVM  FAAWD
```

Knowing this, you are well on your way to being able to read the enemy's communications. Luckily, you also have several pairs of plaintexts and ciphertexts which your team recovered, all of which are known to be encrypted with the same key. Help find the key that the enemy is using.

### H.1 Input

The input consists of multiple test cases. Each test case begins with a line containing a single integer  $N$ ,  $1 \leq N \leq 100$ , the number of plaintext and ciphertext pairs you will receive. The next  $N$  lines each contain two strings  $P$  and  $C$ , the plaintext and ciphertext, respectively.  $P$  and  $C$  will contain only uppercase letters (A-Z) and have the same length (at most 100 characters). The input terminates with a line with  $N = 0$ . For example:

```
1
A B
3
STANFORD SVMFAAWD
AVOWIENR AXAWFEJW
VAMRI VCYMK
3
ABCDEFGHIJKLMNPOQRSTUVWXYZ AAAAAAAAAAAAAAAAAAAAAAAAAA
Y Y
Z Z
2
A B
B A
0
```

## H.2 Output

For each test case, print a single line that contains the shortest possible key or “Impossible” (quotes added for clarity) if no possible key could have produced all of the encryptions. For example, the correct output for the sample input above would be:

```
B  
ACM  
AZYXWVUTSRQPONMLKJIHGFEDCB  
Impossible
```

# I Identity Checker

You likely have seen that  $x(\sin^2 x + \cos^2 x) - x = 0$ , and you may have seen that  $\sin(2x) - 2 \sin x \cos x = 0$ . But did you know that  $\tan(2x)(x - x \tan^2 x) - 2x \tan x = 0$ ? Would you believe that  $\sin(2x) - 2 \cos x = 0$ ? That last one is false, but don't just take our word for it; you should write a program that determines whether an algebraic expression simplifies to zero (whenever it is defined).

## I.1 Input

The input consists of multiple test cases, each on one line. Each test case starts with an integer  $N$ , the number of tokens that describes a formula. The next  $N$  tokens describe a formula in reverse polish notation. The notation works as follows. There is a stack that begins empty, and the following commands manipulate the contents of the stack:

- “**x**” pushes the variable  $x$  to the stack.
- “**sin**”, “**cos**”, and “**tan**” replace the top element of the stack with its sin, cos, and tan, respectively.
- “**+**”, “**-**”, and “**\***” replace the top two elements of the stack ( $a$  on top, followed by  $b$ ) with their sum ( $b + a$ ), difference ( $b - a$ ), and product ( $b * a$ ), respectively.

You may assume that the input is valid, and results in a single item on the stack, which is the desired expression. The length of a line will be at most 300 characters. Function arguments can contain functions, so **x sin sin** is valid, but the recursion will not go any deeper than this. The input terminates with a line with  $N = 0$ . For example:

```
15 x sin x sin * x cos x cos * + x * x -
16 x sin x cos * x sin x cos * + x x + sin -
24 x x + tan x x tan x tan * x * - * x tan x * - x tan x * -
10 x x + sin x cos - x cos -
0
```

## I.2 Output

For each test case, print out a single line that contains “**Identity**” if the expression is always zero, and “**Not an identity**” otherwise (quotes added for clarity). For example, the correct output for the sample input above would be:

```
Identity
Identity
Identity
Not an identity
```