2008 Stanford Local Programming Contest

Saturday, October 4th, 2008

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 specifics problems on the Internet, though you are permitted to use online language references (e.g., the Java API documentation) or algorithms references equivalent to what would be found in a standard algorithms textbook (e.g., CLRS). You may *not* use any machine-readable versions of software, data, or existing electronic code. That is, all programs submitted *must be typed during the contest*. No cutting and pasting of code is allowed.

2. 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.

3. 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 in Gates B02.

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 4.1.3)
   - `.cc`: using `g++ -O2 -lm` (GCC version 4.1.3)
   - `.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 dual Xeon 3.20 GHz machines with 2 GB RAM running Ubuntu Linux 7.10. 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 morning until approximately 1:45 pm.

2. For those who choose to participate onsite, from 1:00 to 1:45 pm, you will select a computer, 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. Customarily, the top six contestants in this contest will be selected to represent Stanford at the forthcoming ACM regional competition. Full results will be posted as soon as possible after the competition.

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 2008, in alphabetical order, Hristo Bojinov, Sonny Chan, Chuong Do, Jonathan Lee, and Ying Wang.

0 Test Problem (`test.{c,cc,java}`)

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 live contest page, which this year resides at:

`http://www.stanford.edu/~sonnycs/cgi-bin/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)));
        }
    }
}
```

A Apocalyptic Alignment (`alignment.{c,cc,java}`)

A.1 Description

Apples and bananas are tasty, but also dangerous. An ancient prophecy said that when you align them in a certain order, the world will be destroyed! On a cloudy day, being tired of this world, you decide to try it out. You started with a line of apples and bananas, and there is one type of allowed operation: At each step, any number of consecutive items in the line can be chosen, replaced by the same amount of fruits of one kind. You can't wait to destroy the world, so you want to know the minimum number of steps to achieve your goal.

A.2 Input

The first line of the input file contains a single number t ($t \leq 100$), the number of test cases. Each test case consists of two lines, where the first line indicates the initial pattern, and the second line indicates the evil pattern that can destroy the world. Both lines contain only characters 'A' and 'B', where 'A' stands for an apple and 'B' stands for a banana. The two lines will have the same length (no greater than 200), and there is no leading or trailing white spaces. For example,

```
2
BB
AA
BAAAB
ABBAA
```

A.3 Output

For each test case, output a single line containing the minimum number of steps to destroy the world. For example,

```
1
2
```

(In the second case of this example, you can first transform the entire row of fruits into apples, and then turn the second through third fruits into bananas, which takes 2 steps total.)

B Boundless Boxes (`boxes.{c,cc,java}`)

B.1 Description

Remember the painter Peer from the 2008 ACM ICPC World Finals?[1] Peer was one of the inventors of *monochromy*, which means that each of his paintings has a single color, but in different shades. He also believed in the use of simple geometric forms.

Several months ago, Peer was painting triangles on a canvas from the outside in. Now that triangles are out and squares are in, his newest paintings use concentric squares, and are created from the inside out! Peer starts painting on a rectangular canvas divided into a perfect square grid. He selects a number of single grid cells to act as central seeds, and paints them with the darkest shade. From each of the seed squares, Peer paints a larger square using a lighter shade to enclose it, and repeats with larger squares to enclose those, until the entire canvas is covered. Each square is exactly one grid cell larger and one shade lighter than the one it encloses. When squares overlap, the grid cell is always filled using the darker shade.

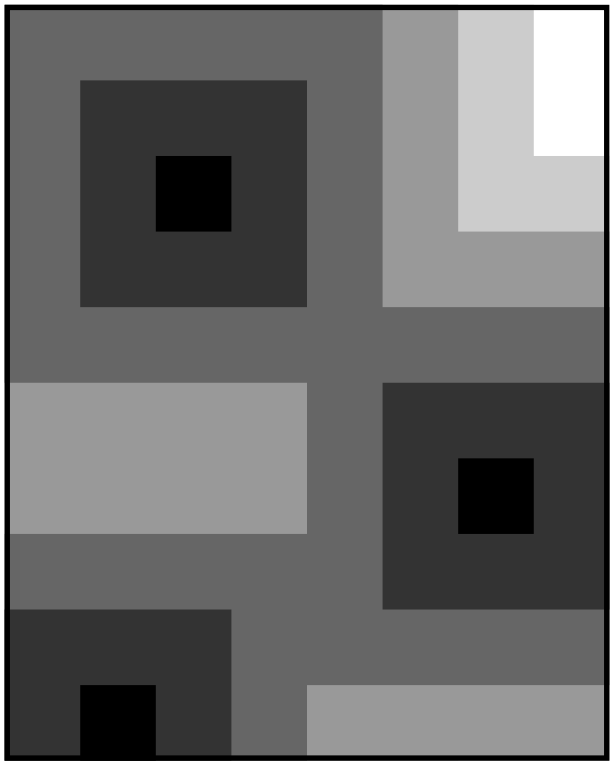

Figure 1: Example of one of Peer's most recent works using six shades of color.

After Peer decides where to place the initial squares, the only difficult part in creating these paintings is to decide how many different shades of the color he will need. To help Peer, you must write a program that computes the number of shades required for such a painting, given the size of the canvas and the locations of the seed squares.

[1] Aside from the artist Peer, this problem really does not have anything to do with the problem from the World Finals, so do not worry if you weren't there or have not read that problem before. In fact, we guarantee that any knowledge of the problem alluded to would not help you solve this one in the slightest bit!

B.2 Input

The input test file will contain multiple cases. Each test case begins with a single line containing three integers, m, n, and s, separated by spaces. The canvas contains exactly $m \times n$ grid cells ($1 \leq m, n \leq 1000$), numbered $1, \ldots, m$ vertically and $1, \ldots, n$ horizontally. Peer starts the painting with s ($1 \leq s \leq 1000$) seed cells, described on the following s lines of text, each with two integers, r_i and c_i ($1 \leq r_i \leq m$, $1 \leq c_i \leq n$), describing the respective grid row and column of each seed square. All seed squares are within the bounds of the canvas.

A blank line separates input test cases, as seen in the sample input below. A single line with the numbers "0 0 0" marks the end of input; do not process this case.

For example, the input for two paintings, including the one shown in the figure above, would look like:

```
10 8 3
3 3
7 7
10 2

2 2 1
1 2

0 0 0
```

B.3 Output

For each test case, your program should print one integer on a single line: the number of different shades required for the painting described. Output corresponding to the sample input would appear as such:

```
6
2
```

C Cargo Carriage (`cargo.{c,cc,java}`)

C.1 Description

A seemingly straightforward task such as driving cargo from point A to point B can sometimes be very tricky! GPS systems can be very helpful for finding routes, but usually don't take all things into consideration. For instance, when finding the quickest route, do they consider waiting at traffic lights? What about congestion? Maybe you can write superior software?

In this problem, you will be given a street map containing roads, numbered intersections (with traffic lights), and two warehouses. Your task is to find the fastest route to drive a truck from warehouse A to warehouse B.

Street maps are always drawn on a grid, and look like this:

```
#A###0##1#
.#..#..#.
.#..#..#.
.###2#.B.
```

Figure 2: A street map with warehouses, roads, and intersections.

Adjacency on this map is defined as neighboring north, south, east, or west. The only symbols that appear on the map are the following:

- A `#` character indicates a road cell that trucks can drive on. Roads are adjacent to at most two other road, intersection, or warehouse cells.
- A single number `[0-9]` marks an intersection controlled by a traffic light. Intersections are adjacent to at least three road cells. Intersections are uniquely numbered sequentially: no number will appear unless all nonnegative integers less than it also appear on the map. The behavior of traffic lights will be described below.
- Exactly one character `A` marks the location of the warehouse where your trucks start.
- Exactly one character `B` marks the location of the warehouse where you'd like to ship your cargo to.
- A `.` character is just grass. You cannot drive on these.

You have one cargo truck that starts at warehouse A, and you are trying to drive it to warehouse B according to the rules below. For simplicity, we can also discretize time into atomic units, or turns.

- On a single turn, you may move the truck onto an adjacent road, intersection, or warehouse cell, or simply remain in the same cell.
- A truck may only move into an intersection cell if the traffic light for the intersection is green in the direction the truck is entering from during that turn. However, a truck on an intersection cell can exit in any direction at any time.

Intersections with traffic lights periodically allow either east-west or north-south traffic flow, but not both at the same time. They are described by an initial direction and two numbers indicating the east-west and north-south periods, respectively. For example, an intersection initially green on the north-south direction described by "2 3" will have a green light facing north and south on turns 1-3 inclusive, facing east and west on turns 4-5, and again north and south on turns 6-8, etc.

C.2 Input

The input test file will contain multiple cases. Each test case begins with a single line containing two integers, m and n, separated by spaces. The street map consists of m rows (east-west) and n columns (north-south) of grid cells ($2 \leq m, n \leq 20$).

The next m lines contain n characters each, which describe the map using the symbols defined in the problem statement above. For each numbered intersection that appears in the map, in ascending order beginning with 0, there will be a line of text with the intersection number followed by either a '-' or '|' character and two integers, a_i and b_i ($1 \leq a_i, b_i \leq 100$), the duration (in turns) of the east-west and north-south periods of the light, respectively. A '-' indicates that the traffic light is initially green in the east-west direction, while a '|' indicates that it is initially green in the north-south direction.

A blank line separates input test cases, as seen in the sample input below. A single line with the numbers "0 0" marks the end of input; do not process this case.

```
3 4
A##B
#..#
####

4 9
#A##0##1#
.#..#..#.
.#..#..#.
.###2#.B.
0 - 1 17
1 | 3 5
2 - 2 4

2 2
A.
.B

0 0
```

C.3 Output

For each test case, your program should print one integer on a single line: the minimum number of turns it takes to drive your truck from warehouse A to warehouse B. If it is not possible to get to warehouse B, print a single word "impossible". Output corresponding to the sample input would appear as such:

```
3
17
impossible
```

(In the second example, it is actually quicker to take the bottom route than the top. However, your truck must wait west at intersection 2 until it can enter on turn 7, when the light is green, and then reaches intersection 0 on turn 10, wait again west of intersection 1 until turn 14, and finally enters warehouse B at the end of turn 17).

D Dreadful Deadlines (`deadlines.{c,cc,java}`)

D.1 Description

Contrary to popular belief, diligence does not always pay off! Over the course of his years as an earnest Stanford undergraduate, David found that despite his best efforts, work would always expand to fill the time available. In order to improve his day-to-day efficiency, David has decided to learn the art of procrastination.

David has n assignments due next week. The ith assignment takes x_i units of time and must be finished by time t_i. David can only work on one assignment at a time, and once David begins an assignment, he must work until it is finished. What is the latest time that David can start in order to ensure that all his deadlines are met?

D.2 Input

The input file will contain multiple test cases. Each test case consists of three lines. The first line of each test case contains a single integer n ($1 \leq n \leq 1000$). The second line of each test case contains n integers, $x_1\ x_2\ \ldots\ x_n$ ($1 \leq x_i \leq 10$) separated by single spaces. The third line of each test case contains n integers, $t_1\ t_2\ \ldots\ t_n$ ($1 \leq t_i \leq 1000$) separated by single spaces.

A blank line separates input test cases, as seen in the sample input below. A single line containing "0" marks the end of input; do not process this case.

```
3
1 2 1
9 9 7

2
2 2
3 3

0
```

D.3 Output

For each input test case, print a single line containing an integer indicating the latest time that Jim can start yet still manage to finish all his assignments on time. If the latest time would require Jim to start before time 0, print "impossible". For example,

```
5
impossible
```

E Earthquake Emendations (`earthquake.{c,cc,java}`)

E.1 Description

A major earthquake has turned a beautiful stained glass window at the Farm Hill College chapel into shards of glass lying on the floor. The chapel manager is calling upon you to help him quickly put the window back together. Luckily the window only broke along its lead joints, and the individual colored pieces are all intact. Additionally, as if by divine intervention, all of the pieces landed on the floor either in their original orientation, or rotated by a multiple of 90 degrees ($\frac{\pi}{2}$ radians), and no pieces have been flipped on their face.

After much searching, a schematic of the original stained glass window as it stood before the earthquake was found in the library. You learn from the schematic that no two colored pieces of the window are identical in shape and size. With this schematic in hand, your mission is to write a program that will identify the strewn shards to help reassemble the window.

E.2 Input

Your program will be given several test cases, each comprising a description of a broken window for you to reassemble. Every test case begins with a single line containing an integer n ($1 \leq n \leq 20$), the number of individual glass pieces in the window. The next n lines each contain a description of a unique colored glass piece in the form of a simple polygon with nonzero area. The coordinates of the k ($3 \leq k \leq 100$) vertices of each polygon will be given in counter-clockwise order in the following format: $x_1\ y_1\ x_2\ y_2\ \ldots\ x_k\ y_k\ x_1\ y_1$. You may assume that the vertices of each polygon are distinct, i.e., $(x_i, y_i) \neq (x_j, y_j)$ whenever $i \neq j$, and consecutive vertices are not collinear. All coordinates are integers restricted to the range $0 \leq x_i, y_i \leq 100$.

The final line of each test case is the schematic of the original window, written as a concatenation of n non-overlapping polygons in the same format described above. Each of the polygons in the schematic is guaranteed to correspond to some rotation plus translation of one of the glass pieces described above. Test cases will be separated by blank lines, and the final line of the input will contain a single integer "0", marking the end of input. For example,

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

4
0 0 4 0 4 2 2 2 0 0
0 0 4 0 2 2 0 2 0 0
4 4 0 4 2 3 4 4
0 4 4 4 2 6 0 4
0 0 4 0 2 2 0 0 0 0 2 2 2 4 0 4 0 0 2 2 4 0 4 4 2 4 2 2 0 4 4 4 2 5 0 4

0
```

E.3 Output

Your program should produce a single line of output for each test case. For every glass piece in the input for a test case, write a single integer indicating the position within the schematic that the piece appears. Separate the numbers with a single space in the output. For example,

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

F Folded Fixtures (`fixtures.{c,cc,java}`)

F.1 Description

You may know of a certain popular and coveted construction toy that Francis got for her birthday this week. This toy consists of a number of metal spheres that can be connected to each other using rigid links (all of the same length) with a magnet on each end. The ends of the links stick to the metal spheres, and the links can freely rotate and extend from the sphere in any direction (essentially forming a spherical joint), allowing you to create a variety of interesting structures.

Francis has assembled several such structures, and now wishes to store her creations by hanging them up in a corner of her room. She notices that, for some of her structures, when she picks it up and holds it by a single sphere, all the links collapse into a single, thin vertical line (see Figure 3) due to the pull of gravity and the spherical joints. Francis can hang up her creations by affixing one sphere on the structure to her ceiling, and wishes to save space by hanging each one up by the sphere that results in the shortest collapsed line. She deems those fixtures which do not hang as a single thin line to take up too much space, and discards them.

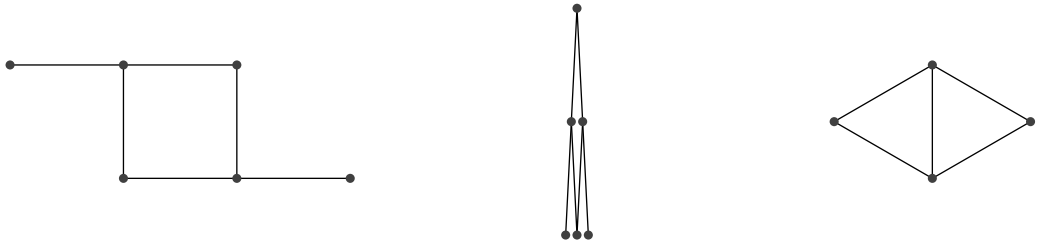

Figure 3: One of Francis's constructions (left), the same collapsed into a straight-line fixture of length 2 (middle), and a structure that will not collapse into a single line when hung up by any sphere (right). Note that in the middle diagram, the horizontal gaps shown between the spheres are for illustrative purposes only! Mathematically, the fixture would be a single, infinitely thin vertical line.

For simplicity, we can treat the metal spheres as infinitely small points and the links as line segments of unit length. Can you write a program to help Francis figure out how much space her fixtures will take, and which ones to discard? Your task is to find the shortest length possible for the collapsed fixture if you were to hang it up by a single sphere, or report that there is no way to hang up the fixture by so that it collapses into an infinitely thin straight line.

F.2 Input

The input will contain multiple test cases for you to analyze. Each test case describes a fully connected fixture (i.e. there are no loose, unattached components). The first line of a test case consists of two integers, n and m, separated by a space, indicating the number of spheres ($1 \leq n \leq 100$) and links ($0 \leq m \leq 1000$) used in the structure, respectively. The spheres are numbered uniquely from 1 to n. The following m lines of input each contain two integers, a_i and b_i ($1 \leq a_i, b_i \leq n$), indicating that Francis has attached sphere a to sphere b in her fixture. Note that both ends of a link cannot be attached to a single sphere, and no two links will attach the same two spheres.

A blank line separates input test cases, as seen in the sample input below. A single line containing "0 0" marks the end of input; do not process this case.

```
6 6
1 2
5 6
3 2
3 5
4 2
4 5

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

0 0
```

F.3 Output

For each input test case, print a single line containing the shortest length possible of the collapsed fixture. If it is not possible to hang the described fixture up by a single sphere so that it collapses into a line, print "impossible". For example,

```
2
impossible
```

G Globulous Gumdrops (`gumdrops.{c,cc,java}`)

G.1 Description

Gwen just bought a bag of gumdrops! However, she does not like carrying gumdrops in plastic bags; instead, she wants to pack her gumdrops in a cylindrical tube of diameter d. Given that each of her gumdrops are perfect spheres of radii $r_1, r_2, \ldots, r_n$, find the shortest length tube Gwen can use to store her gumdrops.

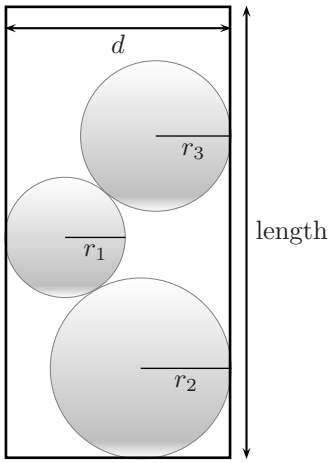

Figure 4: Gumdrops packed into a cylindrical tube.

You should assume that the gumdrop radii are sufficiently large that no three gumdrops can be simultaneously in contact with each other while fitting in the tube. Given this restriction, it may be helpful to realize that the gumdrops will always be packed in such a way that their centers lie on a single two-dimensional plane containing the axis of rotation of the tube.

G.2 Input

The input file will contain multiple test cases. Each test case will consist of two lines. The first line of each test case contains an integer n ($1 \leq n \leq 15$) indicating the number of gumdrops Gloria has, and a floating point value d ($2.0 \leq d \leq 1000.0$) indicating the diameter of the cylindrical tube, separated by a space. The second line of each test case contains a sequence of n space-separated floating point numbers, $r_1\ r_2\ \ldots\ r_n$ ($1.0 \leq r_i \leq d/2$) are the radii of the gum drops in Gloria's bag. A blank line separates input test cases. A single line with the numbers "0 0" marks the end of input; do not process this case.

```
2 98.1789
42.8602 28.7622

3 747.702
339.687 191.953 330.811

0 0
```

G.3 Output

For each input test case, print the length of the shortest tube, rounded to the nearest integer.

```
138
1628
```

H Honed Hops (`hops.{c,cc,java}`)

H.1 Description

In the Olympics, appearances do matter!

The trajectory of a long jumper is given by $h(x) = \max(0, p(x))$, where $p(x) = a(x-h)^2 + k$ is a quadratic polynomial describing a parabola opening downward whose vertex (h, k) lies in the upper half-plane. (That is, $a < 0$ and $k > 0$.) Due to rigorous training, each jumper always jumps with the same trajectory, and due to corporate sponsorship and branding requirements, no two jumpers have the same trajectory.

Adoring fans who wish to preserve the moment occasionally sample their favorite athlete's coordinates at various times and write them down, such as: $(0,0), (1,3), (2,4), (3,3), (4,0), (7,0)$. Given two sample sets, your job is to determine whether they were taken from the same athlete or not, assuming there is enough information to do so.

H.2 Input

The input test file will contain multiple cases, each separated by a blank line. Each test case consists of three lines of text. The first line contains two integers, n_1 and n_2 ($1 \le n_1, n_2 \le 10$) separated by a space, indicating the number of sample points for the first and second sample sets, respectively. The second and third lines contain the sample points for the two sets, in the format $x_1\ y_1\ x_2\ y_2 \cdots x_n\ y_n$. You may assume that $x_1 < x_2 < \cdots < x_n$; moreover, $0 \le x_i \le 100,000$ and $0 \le y_i \le 1000$ for each i. (Be careful that your calculations have sufficient precision for all input conforming to the stated bounds.)

Input is terminated by a single line containing "0 0"; do not process this case. For example:

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

6 1
0 0 1 3 2 4 3 3 4 0 7 0
0 0

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

0 0
```

H.3 Output

For each test case, your program should output a single line containing "same" if the two sample sets are indeed from the same athlete, "different" if they are not, and "unsure" if there is not enough information to tell. For example, the output corresponding to the input above would be:

```
same
unsure
different
```