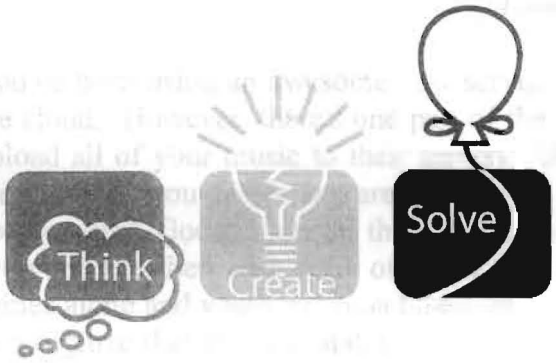
University of Central Florida



2011 (Fall)

Local Programming Contest

Problems

Problem#	Filename	Problem Name
1	music	An (Almost) Perfect Match
2	coin	Good Coin Denomination
3	circles	Cameron's Crazy Circles
4	matrix	Matrix Transformation
5	polycake	Cut the Cake!
6	campout	Camp Out
7	smart	Sorry About That, Chief!
8	goldknight	Knight Moves – Gold Edition
9	blackknight	Knight Moves – Black Edition
10	goldcactus	A Prickly Problem – Gold Edition
11	blackcactus	A Prickly Problem – Black Edition

Call your program file: filename.c, filename.cpp, or filename.java
Call your input file: filename.in

For example, if you are solving An (Almost) Perfect Match:

Call your program file: music.c, music.cpp, or music.java
Call your input file: music.in

UCF Local Contest — September 3, 2011

An (Almost) Perfect Match

filename: `music`

You've been using an awesome new service from Google that lets you stream your music from the cloud. However, there's one part of the service that you really don't like: it takes forever to upload all of your music to their servers. Being a smart computer scientist and avid computer programmer, you decide you are going to fix this problem for them. You realize that with all the people using Google Music, there are probably a lot of duplicate tracks (for instance, nearly everyone has their own copy of Poker Face by Lady Gaga in their locker). When a new user comes along and wants to upload their own copy of Poker Face, it would be much more efficient to recognize that this is a match to an existing track and give the new user access to an existing copy instead of making them upload their own copy.

Let's assume that you've already figured out a nice way to create a fingerprint of a track as a sequence of integers. Assuming F is a function that takes as input a track and returns the fingerprint of that track, it's trivially true that tracks x and y are the same if F(x) = F(y). Unfortunately, your life is not this easy! Often the same track encoded by two different users will have close, but not identical, fingerprints. Because you want your service to be robust and not too picky about slight differences in encoding formats, your matching algorithm should allow tracks that are *close enough* to still be considered a match.

In order to define *close enough*, we need to define some terms. Recall that the fingerprint of a track is a sequence of integers. A single integer in this sequence is called a block, and between 1 and 5 contiguous blocks are called a section. Formally, we define a new track x to match an existing track y if:

1. At most K sections of track y are missing from track x.
2. Each non-deleted block of track y is within a tolerance T of the matching block in track x.

In other words, the absolute value of the difference between each matched block is at most T. K and T are parameters that you set before running your matching algorithm against a set of tracks.

Consider the following example:

Existing track y: 6 12 11 6 7 14 25
New track x: 6 7 7 22

If K=3 and T=5, then x matches y by deleting two sections (12 11, 14) and the aligned blocks being within the required tolerance (6:6 6:7 7:7 25:22).
If K=1 and T=7, then x matches y by deleting one section (12 11 6) and the aligned blocks being within the required tolerance (6:6 7:7 14:7 25:22).
If K=1 and T=2, then x does not match y.

The Problem:

Write a program that reads the fingerprints of all existing tracks and the fingerprints of a set of new tracks to add and determines for each whether the new track matches any existing track.

The Input:

Input will begin with a positive integer C denoting the number of test cases to process. Following this will be C test cases. Each test case will begin with a line containing two non-negative integers $K \leq 20$ and $T \leq 255$ denoting the number of sections that may be deleted and the tolerance, respectively. Following this will be a line containing a single non-negative integer $E \leq 100$ denoting the number of existing tracks on the server. This will be followed by E lines, each containing the fingerprint of an existing track. Each fingerprint will begin with a non-negative integer $N \leq 100$ denoting the length of the fingerprint, and will be followed by N space-separated integers (each integer will be between 0 and 255 inclusive). This will be followed by a line containing a single non-negative integer $U \leq 100$ denoting the number of new tracks the user wants to upload. This will be followed by U lines, each containing the fingerprint of a new track to upload. Each fingerprint will be of the same format described above.

The Output:

For each test case, start with a line "Case #x:" where x is the test case number, starting with 1. Follow this with U lines of output, one for each new track the user wants to upload. For each track, output either the line "Track #i: Match found!" or "Track #i: Need to upload this track." where i is the track number starting with 1. Follow the output of each test case with a blank line.

Sample Input:

```
3
3 5
1
7 6 12 11 6 7 14 25
1
4 6 7 7 22
1 2
1
7 6 12 11 6 7 14 25
1
4 6 7 7 22
0 1
2
5 1 2 3 4 5
5 5 4 3 2 1
3
5 0 2 3 4 5
5 4 4 3 2 1
4 1 2 3 5
```

Sample Output:

```
Case #1:
Track #1: Match found!

Case #2:
Track #1: Need to upload this track.

Case #3:
Track #1: Match found!
Track #2: Match found!
Track #3: Need to upload this track.
```

UCF Local Contest — September 3, 2011

Good Coin Denomination

filename: `coin`

Different countries use different coin denominations. For example, the USA uses 1, 5, 10, and 25. A desirable property of coin denominations is to have each coin at least twice the amount of its previous coin in sorted order. For example, the USA denominations have this property, but the coin denominations {1, 5, 6} do not (6 is not at least twice 5).

The Problem:

Given the coin denominations, you are to determine if the set has the above property.

The Input:

The first input line contains a positive integer, *n*, indicating the number of denomination sets to check. The sets are on the following *n* input lines, one set per line. Each set starts with an integer d $(1 \le d \le 10)$, which is a count of various coin amounts in the set; this is followed by d distinct positive integers (each less than 1,000) giving each coin amount (assume the coin amounts are given in increasing order).

The Output:

At the beginning of each test case, output "`Denominations: v`" where `v` is the input values. Then, on the next output line, print a message indicating whether or not the set has the above property. Leave a blank line after the output for each test case. Follow the format illustrated in Sample Output.

Sample Input:

```
2
4 1 5 10 25
3 1 5 6
```

Sample Output:

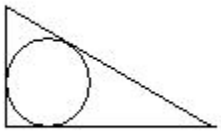
```
Denominations: 1 5 10 25
Good coin denominations!

Denominations: 1 5 6
Bad coin denominations!
```

UCF Local Contest — September 3, 2011

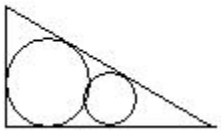
Cameron's Crazy Circles

filename: `circles`

Your cousin Cameron loves circles and has created a rather tantalizing geometry problem, involving an infinite number of circles. He also loves triangles, but he loves circles even more. His problem starts with inscribing a circle in a right triangle as shown below:



Most circle and triangle lovers would stop here, but Cameron is curious what happens if you inscribe a second circle in between the longer leg of the triangle, the hypotenuse and the previous circle (without circles overlapping each other):



In fact, it turns out that you could continue this process an infinitum, inscribing a new circle in between the longer leg, the hypotenuse and the previous circle.

The Problem:

Given the lengths of the two legs of the right triangle, determine the ratio of the area of all the (infinite number of) circles along the longer leg to the area of the original right triangle.

The Input:

The first input line contains a single positive integer, *n*, indicating the number of triangles to solve. Each of the following *n* input lines will contain one test case. Each input case will be two positive integers, L_1 and L_2, separated by spaces, where $L_1 < L_2 < 10000$; these are the two legs of a right triangle.

The Output:

For each test case, first output "`Case #i:`" where `i` is the test case number, starting with 1. Then, output the correct ratio for the case, printed to 4 decimal places, rounded to the nearest ten-thousandth (e.g., 0.00113 should round to 0.0011, 0.00115 should round to 0.0012, and 0.00117 should round to 0.0012). Answers will be judged as correct if they are within 0.0001 of the judge's answer.

Leave a blank line after the output for each test case. Follow the format illustrated in Sample Output.

Sample Input:

```
2
3 4
12 16
```

Sample Output:

```
Case #1: 0.7171

Case #2: 0.7171
```

UCF Local Contest — September 3, 2011

Matrix Transformation

filename: `matrix`

You have an integer matrix A, with R rows and C columns. That means it has R rows with each row containing C integers. Two integers are adjacent if their container cells share an edge. For example, in the following grid

0	1	2
3	4	5
6	7	8

(0, 1), (4, 5), (1, 4), (5, 2) are adjacent but (0, 4), (2, 6), (5, 7) are not adjacent.

You are allowed to do only one kind of operation in the matrix. In each step you will select two adjacent cells and increase or decrease those two adjacent values by 1, i.e., both values are increased by 1 or both values are decreased by 1.

The Problem:

Given a matrix, determine whether it is possible to transform it to a zero matrix by applying the allowed operations. A zero matrix is the one where each of its entries is zero.

The Input:

The first input line contains a positive integer, *n*, indicating the number of matrices. Each matrix starts with a line containing R ($2 \le R \le 30$) and C ($2 \le C \le 30$) separated by a single space. Each of the next R lines contains C integers. Each of these integers is between -20 and +20 inclusive. Assume that each input matrix will have at least one non-zero value.

The Output:

For each matrix (test case), first output "`Case #i:`" where *`i`* is the test case number, starting with 1. Then output "YES" if you can transform it to a zero matrix or "NO" otherwise. Leave a blank line after the output for each test case. Follow the format illustrated in Sample Output.

(Sample Input/Output on the next page)

Sample Input:

```
6
3 3
-2 2 2
1 1 0
2 -2 -2
3 3
-1 0 1
-2 -1 1
0 1 2
3 3
-1 0 1
0 2 -1
-1 1 2
3 3
-1 2 1
-1 -1 -3
1 1 -1
2 3
0 -2 3
1 3 1
2 3
3 1 1
2 0 1
```

Sample Output:

```
Case #1: YES

Case #2: NO

Case #3: NO

Case #4: YES

Case #5: NO

Case #6: YES
```

UCF Local Contest — September 3, 2011

Cut the Cake!

filename: `polycake`

On the faraway planet of Gastronomia, the native Gastronomes consider polygonal pancakes to be the highest form of art (as well as the tastiest). Every year, they celebrate the Polycake Festival, in which thousands of ninja chef monasteries perform the Thousand Slices, a series of rituals of tremendous importance to Gastronome society. In each of these rituals, a polycake is carefully placed on a ceremonial altar, and a ninja chef will solemnly cut the cake with a single, perfectly straight slice, dividing it into two pieces, one on the North part of the altar (representing the past) and the other on the South part (representing the future). The two pieces are carefully separated, and their perimeters measured. The results will determine the proportions of the ingredients used in polycakes planetwide, until the next Polycake Festival.

As the foremost Novice at the Temple of the Promised Cosmic Polycake, you are entrusted the task of measuring the perimeters of the two slices. Take care, for any mistakes will be mercilessly mocked by the Brotherhood of the Illusory Polycake[1]. It would be best if you wrote a program to do this.

The Problem:

Given a polygon representing the polycake in the Cartesian plane and a line parallel to the X-axis indicating the position of the slice, you are to compute the perimeter of each of the two pieces thus formed. The perimeter of a polygon is defined as the sum of the lengths of all its sides. Assume that the input polygons will be convex (i.e., not concave) and simple (i.e., not complex, not intersecting).

The Input:

The first input line contains a positive integer, n, indicating the number of polycakes used in the ritual. This is followed by n data sets, each representing a single slicing of a polycake.

The first line of each set consists of two integers, V $(3 \le V \le 10)$ and Y $(-1000 \le Y \le 1000)$, representing the number of vertices in the polycake and the y-coordinate of the horizontal cut, respectively. (Recall that the equation of a line parallel to the X-axis is of the form $y=k$.)

The next V lines each contain two integers, x and y $(-1000 \le x, y \le 1000)$, the Cartesian coordinates of the vertices of the polycake, in counter-clockwise order.

To minimize complications (and following the rules of the ritual), it is guaranteed that the cut will always go through the cake and will never pass through a vertex.

---

[1] Reviled heretics who claim that the Cosmic Polycake is a lie.

The Output:

For each test case, first output "`Case #i:`" where *i* is the test case number, starting with 1. Then, output "*a b*", the perimeters of the two slices, in increasing order. Output the results to 3 decimal places, rounded to the nearest thousandth (e.g., 77.0113 should round to 77.011, 88.0115 should round to 88.012, and 99.0117 should round to 99.012). Leave a blank line after the output for each test case. Follow the format illustrated in Sample Output.

Sample Input:

```
2
4 2
0 0
4 0
4 4
0 4
6 10
3 15
10 1
12 5
11 19
9 23
6 20
```

Sample Output:

```
Case #1: 12.000 12.000

Case #2: 25.690 35.302
```

UCF Local Contest — September 3, 2011

Camp Out

filename: `campout`

Duke basketball is coming to Central Florida. It's the biggest ticket available on campus this year. UCF has immense school pride and wants to show other schools that whatever they do, UCF can do it better. Duke has a long standing tradition of students camping out for important basketball tickets. Students get in groups of 10 and man a tent for up to 168 hours straight before the tickets are released for sale at midnight. At all times, at least two people must be at the tent in order to secure the group's place in line.

UCF has decided to utilize the same policy for selling tickets to its Duke game, except that it will require three people to be present at each tent at all times for the whole week (168 hours). Also, no individual may man a tent for more than 80 hours during the week, since that would take away too much time from academic pursuits. You have gotten your 9 friends together to form your group of 10 and have collected their schedules. You plan on manning the tent in four hour shifts (12am-4am, 4am-8am, 8am-12pm, 12pm-4pm, 4pm-8pm, and 8pm-12am). Your goal will be to figure out whether or not your group will be able to man the tent. *Note that if a student in the group is busy for part of a shift, they are not allowed to man the tent AT ALL for that particular shift.*

The Problem:

Given ten students' schedules, determine if they can successfully man the tent with at least three individuals for the 168 hours before the tickets for the Duke game go on sale.

The Input:

The first input line contains a single positive integer, n $(1 < n \leq 20)$, indicating the number of groups of students to check. Each of the n input sets follows. Each set contains 10 lines of input, one line per student in the group. Each line of input will contain a list of times that particular student is busy. The format for each of these input lines is as follows:

The first positive integer, I $(I < 20)$, on each line represents the number of intervals the student is busy. (Thus, each student will be busy at least once during the week! There are no complete slackers.) Each of the intervals follow and each piece of data on the line is separated by a single space. Each interval is described with three integers, d, s and e, where d represents the day in the week, s represents the start time, and e represents the end time of the interval. The days are numbered 1 through 7, inclusive, and the start and end times are in military time in hours, in between 0 and 24, inclusive. All intervals are guaranteed to be contained within a single day, no intervals will end with 0 and no intervals will begin with 24. For example, the interval 13 to 17 on day 2 represents a four hour interval starting at 1pm and ending at 5pm on the second day of the 168-hour week. All intervals will be at least one hour in duration.

The Output:

For each test case, first output "`Case #i:`" where *`i`* is the test case number, starting with 1. Then, output the string "YES" or "NO", depending on whether or not the group in question can properly man the tent. Leave a blank line after the output for each test case. Follow the format illustrated in Sample Output.

Sample Input:

```
2
7 1 0 24 2 0 24 3 0 24 4 0 24 5 0 24 6 0 24 7 0 24
6 1 0 24 2 0 24 3 0 24 4 0 24 5 0 24 6 0 24
5 1 0 24 2 0 24 3 0 24 4 0 24 5 0 24
4 1 0 24 2 0 24 3 0 24 4 0 24
4 1 0 24 2 0 24 3 0 24 4 0 23
1 1 0 24
2 1 10 12 2 3 7
7 1 0 24 2 0 24 3 0 24 4 0 24 5 0 24 6 0 24 7 0 24
7 1 0 24 2 0 24 3 0 24 4 0 24 5 0 24 6 0 24 7 0 24
7 1 0 24 2 0 24 3 0 24 4 0 24 5 0 24 6 0 24 7 0 24
7 1 0 1 2 0 1 3 0 1 4 0 1 5 0 1 6 0 1 7 0 1
7 1 1 2 2 1 2 3 1 2 4 1 2 5 1 2 6 1 2 7 1 2
1 2 1 2
1 3 1 2
1 4 1 2
1 5 1 2
1 6 1 2
1 7 1 2
1 2 2 3
1 2 3 4
```

Sample Output:

```
Case #1: NO

Case #2: YES
```

UCF Local Contest — September 3, 2011

Sorry About That, Chief!

filename: `smart`

When Dr. Orooji was your age, one of the popular TV shows was "Get Smart!" The main character in this show (Maxwell Smart, a secret agent) had a few phrases; we used one such phrase for the title of this problem and we'll use couple more in the output!

The Problem:

A "prime" number is an integer greater than 1 with only two divisors: 1 and itself; examples include 5, 11 and 23. Given a positive integer, you are to print a message indicating whether the number is a prime or how close it is to a prime.

The Input:

The first input line contains a positive integer, *n (n ≤ 100)*, indicating the number of values to check. The values are on the following *n* input lines, one per line. Each value will be an integer between 2 and 10,000 (inclusive).

The Output:

At the beginning of each test case, output "`Input value: v`" where `v` is the input value. Then, on the next output line, print one of the following two messages:

- If the number is a prime, print "`Would you believe it; it is a prime!`"
- If the number is not a prime, print "`Missed it by that much (d)!`" where `d` shows how close the number is to a prime number (note that the closest prime number may be smaller or larger than the given number).

Leave a blank line after the output for each test case. Follow the format illustrated in Sample Output.

(Sample Input/Output on the next page)

Sample Input:

```
4
23
25
22
10000
```

Sample Output:

```
Input value: 23
Would you believe it; it is a prime!

Input value: 25
Missed it by that much (2)!

Input value: 22
Missed it by that much (1)!

Input value: 10000
Missed it by that much (7)!
```

UCF Local Contest — September 3, 2011

Knight Moves – Gold Edition

filename: `goldknight`

You have a chessboard of size N x N. The rows and columns are numbered from 1 to N. In a cell located at row R1 and Column C1, a knight is starting his journey. The knight wants to go to the cell located at row R2 and Column C2. Move the knight from the starting cell to this destination cell with minimum number of moves.
As a reminder, a knight's jump moves him 2 cells along one of the axes, and 1 cell along the other one. In other words, if a knight is at (A,B), it may move to (A-2,B -1), (A-2, B+1), (A+2, B-1), (A+2, B+1), (A-1, B-2), (A+1,B-2), (A-1, B+2) or (A+1, B+2). Of course, the knight cannot leave the board.

The Problem:

Given N, R1, C1, R2 and C2, determine the minimum number of steps necessary to move the knight from (R1, C1) to (R2, C2).

The Input:

The first input line contains a positive integer, *T*, indicating the number of test cases. Each case consists of a line containing five integers N ($3 \leq N \leq 20$), R1, C1, R2 and C2 ($1 \leq R1, C1, R2, C2 \leq N$).

The Output:

For each test case, first output "`Case #i:`" where *`i`* is the test case number, starting with 1. Then, output the minimum number of steps needed to move the knight from (R1, C1) to (R2, C2). Assume that there will always be a solution, i.e., it's possible to move the knight from its starting cell to its destination cell. Leave a blank line after the output for each test case. Follow the format illustrated in Sample Output.

Sample Input:

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

Sample Output:

```
Case #1: 1

Case #2: 4
```

UCF Local Contest — September 3, 2011

Knight Moves – Black Edition

filename: `blackknight`

The Black Edition of Knight Move is identical to the Gold Edition except that it deals with a much larger input range (limit). More specifically, the new range for the board size is:

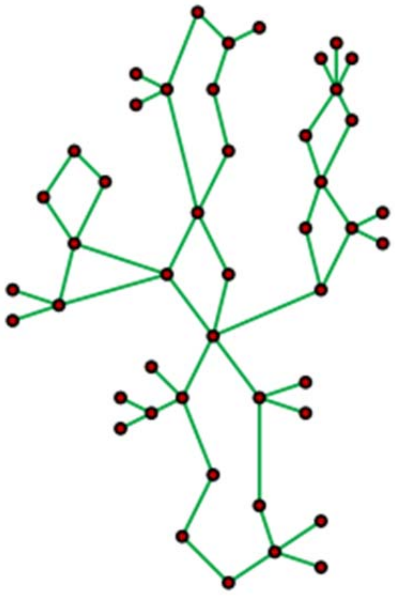
$$N\ (3 \leq N \leq 10^{15})$$

Please note that all the specs from the Gold Edition (except the board size) should be followed for this Black Edition.

UCF Local Contest — September 3, 2011

A Prickly Problem – Gold Edition

filename: `goldcactus`



A tree is a connected graph in which any two vertices have exactly one path between them.

A cactus (sometimes called a cactus tree) is a connected graph in which any two simple cycles have at most one vertex in common. Equivalently, every edge in such a graph belongs to at most one simple cycle. The graph pictured above is an example of a cactus graph.[1]

A spanning tree can be created from a connected graph by removing a set of edges such that if there are V vertices in the graph, then there are V – 1 edges remaining and every pair of vertices has exactly one path between them. Depending on which edges you choose to remove you will end up with different spanning trees. The cactus graph pictured above contains 36,864 spanning trees.

The Problem:

In this problem, your task is to count the number of spanning trees that a given cactus has. Since this result may be quite large, you should report your result modulo 1,007.

The Input:

Input will begin with an integer T denoting the number of test cases. Each test case will begin with two positive integers $V \leq 100$ and $E \leq (3*V)/2$ denoting the number of vertices and the number of edges, respectively. This will be followed by E lines, each containing an edge in the graph. Each edge is represented by its two vertices and each vertex listed will be between 1 and

[1] Picture of graph and definition of a cactus graph taken from www.wikipedia.org

V (assume that there is at most one edge in the input between any two vertices). It is guaranteed that the graph described in the input will be a cactus.

The Output:

For each test case, output a single line "Case #x: y" where x is the case number starting with 1 and y is the number of spanning trees modulo 1,007. Leave a blank line after the output for each test case. Follow the format illustrated in Sample Output.

Sample Input:

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

Sample Output:

```
Case #1: 3

Case #2: 9

Case #3: 1
```

UCF Local Contest — September 3, 2011

A Prickly Problem – Black Edition

filename: `blackcactus`

The Black Edition of Prickly Problem is identical to the Gold Edition except that it deals with a much larger input range (limit). More specifically, the new range for the number of vertices is:

$$V \leq 50{,}000$$

Please note that all the specs from the Gold Edition (except the number of vertices) should be followed for this Black Edition.