Sixth Annual
University of Central Florida

High School Programming Tournament: Online Edition

Problems

Problem Name	Filename
Baking Botchery	baking
Forest Fairies	fairies
Nanomachines, Son!	nano
Palace Walls	palace
Solid Snake	snake
Virtual Blindness	virtual
Wreck It!	wreck
Wrut Row	wrut

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

For example, if you are solving Wreck It!:
Call your program file: wreck.c, wreck.cpp or wreck.java
Call your Java class: wreck

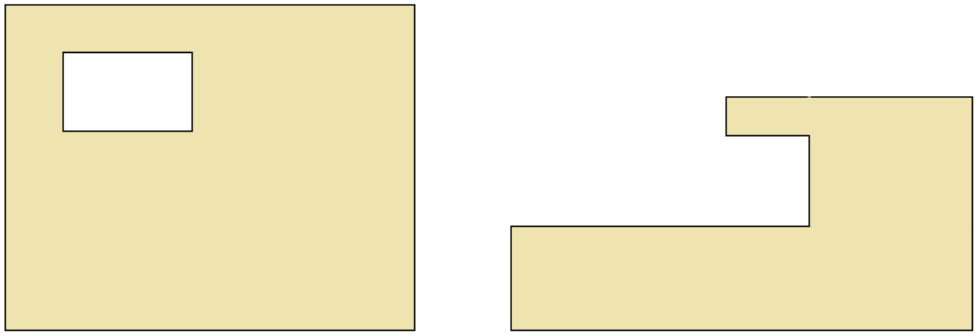
Baking Botchery

Filename: `baking`

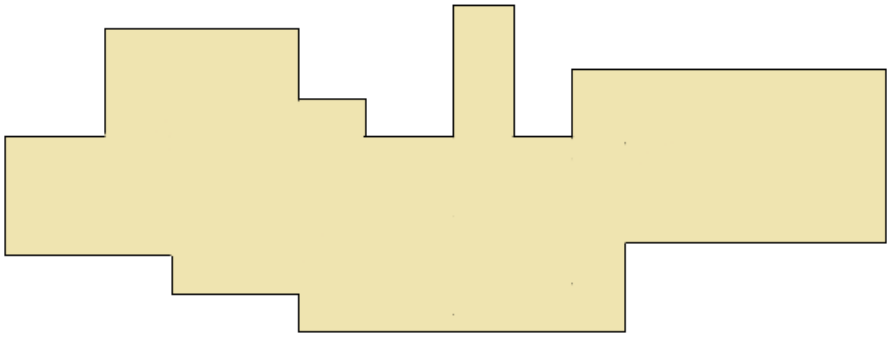
After only a semester as a UCF student, Ryan has decided that the field of computer science is rather unpleasant and he's decided to follow his dreams and open his own bakery! Ryan is also very specific in how he prepares the dough for his specialty gourmet cookies, which are his best-selling product. Before cutting the cookies, Ryan lays the cookie dough on the counter and proceeds to cut out the shapes he's using for the day. Since he always prepares the dough so that it forms a rectangle, he can easily eyeball the dough to see its area in square centimeters and tell how many cookies can be cut from it. Unfortunately, Ryan's new assistant has accidently put the perfect rectangular dough into the Corner Creator 3000™! The Corner Creator 3000™ is a machine that takes in some dough and morphs it into a new shape, with the following properties:

- All of the angles of the shape are right angles (either interior or exterior).
- All sides of the shape are parallel to either the x-axis or the y-axis.
- The left-most and right-most sides of the shape will have exactly two corners.
- Any vertical line will divide the shape into at most two pieces.
- No empty squares will be formed in the dough.
- The shape will not "wrap around" (intersect) itself.

As such, the following two shapes cannot be formed by the Corner Creator 3000™:



The shape below, however, meets the Corner Creator 3000's specifications:



Once the dough goes through the machine, Ryan is no longer able to eyeball the dough to see how many cookies he can make, and his customer has a very specific order to fulfill. Since Ryan is busy running the bakery, he doesn't have time to figure out the area of the new shape to know how many cookies he can make. Fortunately for him, he knows that you're a rather skilled computer programmer, and he's tasked you with finding the area of this new shape. Ryan is only able to give you the locations of the corners of the dough. Can you help him find the area?

The Problem:

Given the locations of the corners of Ryan's piece of dough, determine the area of the shape that it forms in square centimeters.

The Input:

Input will begin with a single, positive integer, n ($1 \leq n \leq 50$), representing the number of times rectangular dough has gone through the machine (Ryan's assistant has done this multiple times). Each scenario will start with a single, positive integer, c ($4 \leq c \leq 500$), representing the number of corners on the piece of dough. The next c lines contain two integers, x_i and y_i ($-10{,}000 \leq x_i \leq 10{,}000$; $-10{,}000 \leq y_i \leq 10{,}000$), which represent the x and y coordinates (in centimeters), respectively, of a unique location of a particular corner (note that these corner points may appear in any order).

The Output:

The output for each scenario should read "`Batch #`*`i`*`: `*`a`*` square centimeters`" where *i* is the number of the scenario in the input (starting with 1) and *a* is the area of the shape in square centimeters.

(Sample Input and Sample Output are on following page)

Sample Input:

```
2
4
0 0
0 10
2 10
2 0
22
-5 -1
4 -1
1 7
17 8
15 8
1 15
9 15
31 12
17 18
19 12
-5 7
15 18
9 -8
19 8
20 -8
11 10
9 -4
20 0
11 8
31 0
4 -4
9 10
```

Sample Output:

```
Batch #1: 20 square centimeters
Batch #2: 527 square centimeters
```

Forest Fairies

Filename: `fairies`

The distant Catalan Forest is one of the strangest ecosystems in the world; it has an infinite number of trees, yet the number of trees with *n* leaves is finite for every non-negative integer *n*. In fact, *C*(*n*), a function that outputs the number of trees with *n* leaves, is given by the following recurrence:

$$C(0) = 1$$
$$C(n+1) = \sum_{i=0}^{n}[C(i) * C(n-i)] \text{ for } n \geq 0$$

For example, C(1) = 1, C(2) = 2, C(3) = 5, and C(4) = 14.

Some of these trees hold forest fairies, a whimsical species that resides only in the Catalan Forest. These fairies are very picky about their habitats, and will only live in trees with between *a* and *b* leaves, inclusive. Also, forest fairies need their space to sleep, so every tree with a number of leaves in that interval houses exactly one forest fairy (we assume that this forest has reached its carrying capacity for the fairies), and of course all other trees house no forest fairies. These fairies, being rather social creatures, all feed at the same time in as few groups as possible, and to do so they fly together to Parity Pond, the nearest source of fairy water. However, like geese, they like to fly in V-shaped formations. Formally, a V-shaped formation is one fairy followed by *x* pairs of fairies for some non-negative integer x. Given that every group of fairies must be a V-shaped formation, they would like to know the minimum number of groups they can form. However, there are too many fairies to count, so all they know are the parameters *a* and *b*, the lower and upper bounds of how many leaves can be on a tree for it to house a forest fairy. Can you help them?

The Problem:

Given that forest fairies will only live in trees with between *a* and *b* leaves inclusive, compute the minimum number of V-shaped formations that can be formed by all the fairies.

The Input:

The first line of input will contain a single integer, *t*, representing the number of forests. Each of the next *t* lines will each contain two integers, the constraints *a* and *b* ($0 \leq a \leq 10^{18}$; $0 \leq b \leq 10^{18}$).

The Output:

For each forest, output a line containing a single number, the minimum number of groups the fairies can form.

Sample Input:

```
2
0 1
2 4
```

Sample Output:

```
2
1
```

Nanomachines, Son!

Filename: nano

Undergoing a full-body cybernetic conversion has its perks, most notably becoming a wicked-sick cyborg ninja with lightning powers. This transformation comes with a price, however: the eternal thirst for enemy cyborgs' electrolytes! I mean nanomachines! Whatever! The cyber-assassin Raiden is a master of combat, capable of bringing entire armies to their knees. To maintain optimal performance levels, the man-machine must perform a maneuver termed "Zandatsu," the ol' cut-and-take! The sunder-and-plunder! Shank-and-yank! Slit-and-ask-for-it! I've got a million of 'em![1] Through this technique, Raiden slices and dices his way to the electrolyte-infused core of a cybernetic enemy, exposing its fuel cells, and absorbs the sweet, sweet nanomachines contained within.

Raiden's body is mantained by the German doctor Wilhelm Voigt, who after combat missions downloads and analyzes data from Raiden's memory banks for use in later virtual reality simulations. An important aspect of Raiden's combat data is the number of sword swings necessary to expose an enemy cyborg's fuel cells. Each time Raiden swings his sword, a piece of the enemy combatant is sliced in two. Wilhelm needs to know the number of pieces each enemy has been carved into, so he can further enhance Raiden's cybernetic capabilities!

The Problem:

Given the dump from Raiden's memory banks for a single encounter, determine the number of pieces the enemy cyborg was sliced into, assuming that Raiden bisects a piece of his enemy with each slash. The combat log will be given in the form of a string, where each character in the string represents a piece of data. A sword swing in the combat log is denoted by one of the characters from the set {`-`, `|`, `\`, `/`}. Raiden's cybercortex is constantly analyzing angles of attack and performing calculations for totally radical backflips during combat, so there may be a bit of noise in the data, which Wilhelm suggests you ignore.

---

[1] *Rend and lend!*
Slash and stash!
Maim and claim!
Gore and store!
Stab and grab!
Slag and snag!
Hack and sack!
Rip and grip!
Gash and cash!
Bisect and collect!
Claw and withdraw!
Shank and yank!
Sunder and plunder!
Divide and inbibe!
Scythe and tithe!
Cleave and thieve!
Break and take!
Mash and cache!
Chop and shop!
Mince and pinch!
Reap and keep!
Saw and paw!
Slit and ask-for-it!

The Input:

The first line of the input file will contain a single, positive integer, *c*, the number of cyborgs Raiden encountered on his last mission. On each of the next *c* lines will be a string composed of alphanumeric characters and characters from the set {`-`, `|`, `\`, `/`}, the combat log for this encounter. Each combat log will be between 1 and 1000 characters in length.

The Output:

For each cyborg Raiden encounters, output "`Cyborg #`*i*`: `", where *i* is the encounter number (starting at 1). On the same line, output "*x* `pieces!`", where *x* is the number of pieces the cyborg was sliced into, if Raiden cut the cyborg at least once. If Raiden failed to cut the cyborg at all, output "`1 piece? You're supposed to be stronger than this!`". Output encounter reports on separate lines.

Sample Input:

```
3
/RU\L-E|SOF|-N/AT\URE
///////\\\\\\\
LEFTHANDRULE
```

Sample Output:

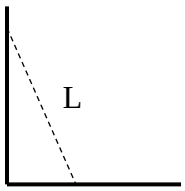
```
Cyborg #1: 9 pieces!
Cyborg #2: 15 pieces!
Cyborg #3: 1 piece? You're supposed to be stronger than this!
```

Palace Walls

Filename: `palace`

Princess Jasmine lives in a wonderful palace, but sometimes she wishes she could spend more time outside the palace walls. People tell her where to go and how to dress, she's not free to make her own choices, and sometimes she feels so… trapped. She thought maybe if she had more space to wander around she would be happier, so she used her authority as princess to get the palace walls rebuilt.

The construction workers have been hard at work making a new courtyard in the shape of a right triangle. They have just one more piece of wall to place, but they just received a command from Jasmine that the area of the courtyard must be at least x square meters so there's enough space for her and her pet tiger to be comfortable. The diagram below shows the present state of construction:



The bold solid lines are walls that have already been placed, and the diagonal dashed line is a possible position for the last piece of wall. However, any placement of the new wall is allowed as long as it forms a straight line and has one endpoint on each existing wall. The last piece of wall has length L, and the entire piece must be used. It cannot be cut into multiple pieces, and it is guaranteed that both existing walls are longer than L in length (so their exact length is not relevant to the area of the courtyard that will be formed).

The Problem:

Given L, the length of the last piece of wall to be placed, calculate whether or not it is possible to satisfy Jasmine's command that the courtyard have area greater than or equal to x.

The Input:

The first line contains a single, positive integer, w, the number of possible walls to assess. Each of the next w lines contains two integers, L ($0 \leq L \leq 1{,}000$) and x ($1 \leq x \leq 1{,}000{,}000$), representing the length of the last piece of wall in meters, and the desired size of the courtyard in square meters, respectively.

The Output:

For each wall, output a line containing "`Wall #w: `" where w is the wall number (starting from 1) followed by "`YES`" if it is possible to satisfy Jasmine's command, or "`NO`" otherwise.

Sample Input:

```
4
5 6
13 40
10 30
200 50000
```

Sample Output:

```
Wall #1: YES
Wall #2: YES
Wall #3: NO
Wall #4: NO
```

Solid Snake

Filename: snake

Travis's roommates love playing the game Metal Gear Solid. Travis has heard them talk about this game many times and hence is an expert on the game.

The game involves hiding the main character Solid Snake in his signature cardboard box. Sometimes the box moves around by exposing its metal gears which act as wheels for the box. Clearly, since snakes bend, the main challenge of the game is getting the snake in the box!

Travis would like to impress his roommates by playing this Metal Gear Solid game. He would like you to write a program to do the most difficult part (fitting the snake in the box). That way his character, Solid Snake, can relax and enjoy his signature Solid Snake Box!

Since all games involving snakes are similar to the classic arcade game Snake, Metal Gear Solid is realized on the 2D grid. Snake's tail is at (0,1) and his head is at (0, L) where L is the length of Snake. You must put Snake in the box in no more than 1,000,000 moves. None of the moves should allow Snake to cross his body.

At each step, you can move Snake in one of four directions: left, right, up, or down. When Snake's head moves, his body will follow and will use the same path. In other words, each segment from head to tail will move to the spot the previous segment (the one closer to the head). For example, if the segments of Snake are numbered 1 to L, segment i+1 will move to the location of segment i. The head will move to the adjacent location in the direction specified.

The Problem:

Given the start location of Snake, find a sequence of moves that will put him fully within the box.

The Input:

The first line of the input contains a single, integer, *t*, representing the number of times Solid Snake needs to hide. The next *t* lines each contain 5 integers, L ($1 \leq L \leq 100$), x_0, y_0, x_1 and y_1 ($-100 \leq x_0 \leq x_1 \leq 100$; $-100 \leq y_0 \leq y_1 \leq 100$), representing the length of the snake and the box coordinates of the (inclusive) lower left corner and the upper right corner of the box, respectively.

The Output:

For each time Snake needs to hide, output the header "`Mission #`*d*`:`", where *d* is the mission number, starting from 1. Follow this by a string composed of only the letters {`U`, `D`, `L`, `R`}. It represents the moves of Snake in order. The length of the string is the number of moves Snake makes and should be no longer than 1,000,000 letters in length. At the end of these moves, Snake should be in the box. It will always be possible to put Snake in the box. In fact, there could be multiple correct answers for a mission; you can output any one of them.

Each command in the strong is one of the following:

- `U` – move up
- `D` – move down
- `L` – move left
- `R` – move right

Sample Input:

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

Sample Output:

```
Mission #1: RDDDRUL
Mission #2: LLLDDDD
```

Virtual Blindness

Filename: `virtual`

Virtual Boy game systems are fun to play. However, as anyone who has played one knows, they can hurt your eyes if you play for an extended period of time. As such, after every *p* minutes of playing, it is important to take a break for *b* minutes.

However, these numbers may be different for different players. So let's calculate how long it takes to play the game to the finish! Note: you do not count any break you need to take after the game is finished.

The Problem:

Given the values of *p* and *b* for a player, and given the amount of time spent playing (not counting breaks) needed to finish a game, calculate the total amount of time it will take from start to finish before the game is finished.

The Input:

The first line will contain a single, positive integer, *n*, representing the number of people for which to answer this question. Each of the following lines will contain a description for one person. Each person will consist of a line with three integers, *b* ($0 \leq b \leq 100$), *p* ($1 \leq p \leq 100$) and *t* ($0 \leq t \leq 10{,}000$), representing the time for each break, the play time before each break, and the total play time to finish the game, respectively.

The Output:

Output one line for each person, containing the total time from start to finish to play through a game, taking breaks as needed.

Sample Input:

```
2
5 15 100
2 10 20
```

Sample Output:

```
130
22
```

Wreck It!

Filename: `wreck`

Ralph's favorite pastime is wrecking buildings. If it were possible, he would wreck a building every day. He lives in a city with *n* buildings and *m* two-way roads, where each road connects two buildings to each other. The buildings are numbered from 1 to *n*, and Ralph is currently located in the building numbered 1. Every day, Ralph wrecks the building he is located at and then chooses a road to get from his current building to a different building (or stays put if there are no roads connected to his current building). After Ralph wrecks a building, he cannot wreck it again anytime during the next *k* days because he must wait for his friend Felix to fix it. Because of this restriction, it may be impossible for him to destroy a building every day (i.e., there may be some point in time when there is no road he can take that leads him to a building he can wreck the next day). Ralph has asked for your help in determining whether or not it is possible for him to prevent this from happening and wreck a building every single day.

The Problem:

Given the values of *n* and *k*, as well as the configurations of roads for different cities, Ralph would like to know whether or not it is possible for him to destroy a building every day for any number of days he wants.

The Input:

The first line contains a single, positive integer, *t*, representing the number of cities Ralph wants you to analyze. The input for each city starts with a line containing three integers, *n* ($1 \leq n \leq 5$), *m* ($0 \leq m \leq n * (n - 1) / 2$), and *k* ($1 \leq k \leq 100$), representing the number of buildings in the city, the number of roads in the city, and the number of days Felix needs to fix buildings in that city, respectively. Each of the next *m* lines contains two integers, *a* and *b* ($1 \leq a \leq n$; $1 \leq b \leq n$; $a \neq b$), indicating that there is a two-way road connecting the buildings numbered *a* and *b*. There will be at most one road between any pair of buildings.

The Output:

For each city, output a line containing "`City #c: `" where *c* is the number of the city (starting from 1) followed by either "`YES`" if Ralph can destroy a building every day, or "`NO`" if he cannot.

Sample Input:

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

Sample Output:

```
City #1: NO
City #2: YES
```

Wrut Row

Filename: `wrut`

Scooby-Doo is learning to do math with his best friend Shaggy. Unfortunately, Scooby is having trouble grasping the concept of addition. Can you help Scooby figure out if his calculations are correct? If he does he a good job, Shaggy may give him a Scooby Snack!

The Problem:

Given the values for the variables *a*, *b*, and *c* in the equation $a + b = c$, determine if the values for *a*, *b*, and *c* satisfy the equation.

The Input:

The first line of the input file will contain a single, positive integer, *e*, representing the number of equations to examine. This will be followed by *e* lines of 3 integers each, the values of *a*, *b*, and *c* ($1 \leq a \leq 1{,}000$; $1 \leq b \leq 1{,}000$; $1 \leq c \leq 1{,}000$).

The Output:

For each equation that is correct, output the message "`Correct!`" on a line by itself. If the equation is incorrect, output the phrase "`Wrut Row!`" instead.

Sample Input:

```
4
1 2 3
1 1 11
47 74 1
5 5 10
```

Sample Output:

```
Correct!
Wrut Row!
Wrut Row!
Correct!
```