Ninth Annual
University of Central Florida

High School Programming Tournament: Online Edition

Problems

Problem Name	Filename
Ant Colony	colony
Hyper Even Numbers	even
Conference Expansion	expansion
Exploding Fireworks	fireworks
Flooding the Tri-State Area	flooding
Trapped Laptop	laptop
Christmas Lights	lights
Nailed It!	nails
Poker Hands	poker
Sharon the Slayer	slayer

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

For example, if you are solving Hyper Even Numbers,
Call your program file:
even.c, even.cpp, even.java, or even.py
Call your Java class: even

Ant Colony

Filename: `colony`

It's a well known fact that ants build large networks of underground tunnels for their colonies. In these colonies, there exist large open rooms, and tunnels which connect the rooms in both directions. We are interested in a specific type of ant, which are very efficient in building their network. Specifically, they build their network in such a way that any two rooms have exactly one path (direct with a single tunnel, or indirect through other rooms) connecting them.

Worker ants will be tasked with transporting things from one room to another in the colony. This is normally fine because there is always a path between any two rooms. However, recently there has been trouble in the ant colony. Some of the tunnels are collapsing! Therefore, it might not be possible to travel between certain pairs of rooms anymore. You will have to help the ants out, and tell them if it's still possible to travel between pairs of rooms. Be careful though; in between these queries more tunnels might collapse.

The Problem:

Given the tunnels and rooms of ant colonies, process events asking if a pair of rooms is connected, while receiving new information of collapsed tunnels.

The Input:

The first line of an input will contain a single, positive integer, m, which represents the number of ant colonies you will have to process. Then for each ant colony, you will receive the following set of information:First, a single line will be given consisting of an integer, n ($1 \leq n \leq 10^5$), representing the number of rooms. Each of the next n - 1 lines contains two integers, a and b ($1 \leq a \leq 10^5$; $1 \leq b \leq 10^5$; $a \neq b$), indicating there is a tunnel from room a to room b. The tunnels are numbered from 1 to n - 1 in the order they are given.

The next line will contain a single integer, q ($1 \leq q \leq 10^5$), indicating the number of events to process. Each of the next q lines will contain three integers, t, c, and d ($1 \leq t \leq 2$; $1 \leq c \leq 10^5$; $1 \leq d \leq 10^5$). If $t = 1$, the event indicates the tunnel from c to d collapsed (it is guaranteed there will be an intact tunnel between room c and d immediately previous to this point). If $t = 2$, the event is a query for whether room c can still reach room d.

The Output:

For each colony (in order from input), the first line of output should be "`Colony #i:`" where i is the number of the colony (starting from 1). Then process each event for that colony in the order given. For a type $t = 1$ event, output "`Tunnel from c to d collapsed!`" (replacing c and d with the numbers of the rooms). For a type $t = 2$ events, output a single line of either "`Room c can reach d`" or "`Room c cannot reach d`" (again replacing c and d with the numbers of the rooms). After answering all of the events for a given colony, output an extra blank line.

Sample Input:

```
2
3
2 3
1 3
3
2 1 2
1 1 3
2 1 2
5
1 4
2 5
3 1
2 1
9
2 3 5
1 1 2
2 5 3
2 3 4
2 2 5
1 2 5
2 5 3
1 1 3
2 1 4
```

Sample Output:

```
Colony #1:
Room 1 can reach 2
Tunnel from 1 to 3 collapsed!
Room 1 cannot reach 2

Colony #2:
Room 3 can reach 5
Tunnel from 1 to 2 collapsed!
Room 5 cannot reach 3
Room 3 can reach 4
Room 2 can reach 5
Tunnel from 2 to 5 collapsed!
Room 5 cannot reach 3
Tunnel from 1 to 3 collapsed!
Room 1 can reach 4
```

Hyper Even Numbers

Filename: `even`

Bash Fetchum is training his newly-caught Bulbasort. In order to beat the Elite 4 and the Pokemon Champion, Master Coleman, he needs his Bulbasort to learn the move Hyper Beam. However, as anyone who has ever tried to fetch 'em all can tell you, Bulbasort only learns the move Hyper Beam upon reaching a level that is a hyper even number.

What is a hyper even number, you ask? Well, as you know, an even number is a number that is divisible by 2. A *hyper* even number is an even number that has only even factors (not including the factor 1). Given the level of Bash's Bulbasort, how many times does he need it to level up in order for it to learn hyper beam?

The Problem:

Calculate the number of times Bash must level up his Bulbasort in order for it to reach the next level that is a hyper even number.

The Input:

The first line will consist of a single integer, t, representing the number of queries of Bulbasort's level. This will be followed by t lines, each containing a single integer, x ($1 \leq x \leq 10^9$), representing the level of Bash's Bulbasort in question.

The Output:

Output t lines in the form "`Pokemon i: y`" where `i` is the query (from the input) numbered (starting with 1) and `y` represents the number of times Bash must level up his Bulbasort for it to reach the next level that is a hyper even number given the i^{th} query.

Sample Input:

```
4
1
2
3
268435439
```

Sample Output:

```
Pokemon 1: 1
Pokemon 2: 2
Pokemon 3: 1
Pokemon 4: 17
```

Conference Expansion

Filename: `expansion`

The Big 2048 is expanding! As commissioner, your job is to add as many teams as possible to the conference. However, there is just one problem: the current teams in the conference. Whenever you suggest a team to be added, a vote must be taken on whether that team will be added. If the majority of teams vote in favor, the team will be added (tied votes will fail).

Fortunately, you have already figured out the voting strategy used by every team. When a team is suggested for expansion, every team better than that team (as indicated by their respective skill levels) will vote against that team, in order to avoid diluting their brand. Every other team will vote in favor. As the commissioner, you get to determine the order in which teams are suggested to be added to the conference. In addition, teams are added one at a time, meaning every team you successfully add will now vote on all future additions, according to the same rules as all other teams.

The Problem:

Given a list of current team skill levels and the skill levels of potential new teams, determine the maximum number of teams you may add to the conference.

The Input:

The first line will be a single integer, t, representing the number of expansion scenarios to consider.

Following this will be t scenarios. In each scenario the first line will consist of two integers, o and n ($1 \leq o \leq 2{,}048$; $1 \leq n \leq 2{,}048$), representing the number of original teams and the number of potential new teams, respectively. This will be followed by a line of o integers representing the skills of the current conference teams. This, in turn, will be followed by a line of n integers representing the skills of potential new teams to add. All skills are bounded by 1 and 10,000 inclusive.

The Output:

For each scenario, output "`Expansion #`*i*:" where *i* represents the scenario number (starting with 1), followed by the maximum number of teams that may be added.

(Sample Input and Sample Output follow on next page)

Sample Input:

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

Sample Output:

```
Expansion #1: 1
Expansion #2: 3
```

Exploding Fireworks

Filename: `fireworks`

The Ultimate Collection of Fireworks (UCF) has the largest collection of fireworks in Central Florida. UCF has large, sparkling ones; small, popping ones; and tons more.

UCF wanted to celebrate its 55th year in business this past summer, so they decided to launch their entire fireworks inventory at once! Peter (an employee at UCF) was tasked with collecting statistics about all of the fireworks.

It has been months since Peter's task was due, but he still can't get it done! Peter doesn't have the programming skills to process his data into cool figures and statistics; he needs you, a programmer, to help him figure out the following: the first rocket that exploded, and the rocket that exploded the highest.

Because UCF's fireworks are the best in the world, each firework explodes at its maximum height. Additionally, each explodes the instant its vertical velocity is zero. The night of the celebration had negligible wind. All fireworks were launched from the roof of UCF's headquarters, a perfectly horizontal plane, at *exactly* the same time. The acceleration due to gravity is constant. And because UCF will celebrate 60 years one day soon (hopefully!), make sure you can handle multiple scenarios.

The Problem:

Given the initial velocity for each firework, output which of the fireworks explodes: (1) the earliest after launch, and (2) the highest above the launch point.

Recall the equation $\frac{1}{2}at^2 + vt + x_0 = x(t)$, where x is the position (at some time t, with $x(0)$ being the initial position of a particle) of a particle with initial velocity v and constant acceleration a. Also recall that $v(t) = at + v_0$, where v is the velocity at time t of a particle with initial velocity v_0 and constant acceleration a. The acceleration of gravity is $9.81\ m/s^2$ towards Earth.

The Input:

The first line of input contains a single integer, *t*, representing the number of scenarios to follow. For each scenario, the first line contains an integer, *n* ($1 \le n \le 2{,}000$), denoting the number of fireworks to process. The next line has *n distinct* whole numbers where the *i*th number, v_i ($1 \le v_i \le 5{,}000$), represents the initial velocity of the *i*th rocket.

The Output:

For each scenario, first output "`Scenario #`j`:`" on its own line (starting with 1). Then, on its own line, print out "`Highest Firework:` i" denoting that the i^{th} firework exploded the highest. Then, on the next line, print "`Earliest Firework:` k" denoting that the k^{th} firework exploded the earliest. Finally, output a blank line after the output for each scenario.

Sample Input:

```
2
2
4 3
5
8 7 6 5 9
```

Sample Output:

```
Scenario #1:
Highest Firework: 1
Earliest Firework: 2

Scenario #2:
Highest Firework: 5
Earliest Firework: 4
```

Flooding the Tri-State Area

Filename: `flooding`

Doofenshmirtz M.D. is at it again! This time, he is building a tide-raising-inator to flood the tri-state area so that everyone will have to buy his Buoyancy Operated Aquatic Transport vehicles to drive around on water. Luckily, two middle-school kids who had nothing better to do with their summer vacation have conveniently just finished building a triangular wall around the entire triangular tri-state area in case of any unpredicted floods.

However, Secret Agent Perry has n secret tunnels in the tri-state area, and in order for the wall to be effective, any tunnels that go from one side of the area to the other (i. e. tunnels that start inside and end outside, or ones that start outside and end inside) need to be closed. Given the bird's-eye view coordinates of the three corners of the tri-state area wall, and the start and end points of all of Perry's secret tunnels, help Perry figure out how many tunnels he has to close off to stop Doofenshmirtz' evil plans.

The Problem:

Determine the number of secret tunnels that must be clogged for the city to not be flooded.

The Input:

The first line will contain an integer, t, representing the number of scenarios to consider. For each scenario, the first line will contain six integers, a_x, a_y, b_x, b_y, c_x, and c_y, representing the coordinates of the three points of the tri-state area, a, b, and c. These points will be unique and form a triangle with non-zero area. The next line will contain a single integer, n ($1 \le n \le 10^4$), representing the number of secret tunnels. This will be followed by n lines, each with 4 space-separated integers, s_x, s_y, e_x, e_y, representing the coordinates of a secret tunnel starting at s and ending at e. Neither s or e will fall directly on the triangle representing the tri-state area. The absolute value of all coordinates will be no greater than 10^4.

The Output:

For each scenario, output a single line containing "`Scenario i: x`" where i is the scenario number in the input (starting with 1), and x is the number of tunnels that must be closed.

(Sample Input and Sample Output follow on next page)

Sample Input:

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

Sample Output:

```
Scenario 1: 0
Scenario 2: 1
```

Trapped Laptop

Filename: `laptop`

Sharon and Charles are on a field trip. Sharon, being the smart, naughty boy he is, decides to place Charles' laptop in a mechanical cage surrounded by n buttons, each a distance of r away from the center. Each button lies on the vertex of a regular n-gon made by placing the caged laptop in the center. Each button also has one of the following colors: Red (R), Green (G), or Blue (B).

In order for Charles to retrieve his laptop, he must press all the buttons. Sharon, however, also added this twist: all of the red buttons must be pressed before all of the green and blue buttons, and all of the green buttons must be pressed before all of the blue buttons. If no red buttons exist, Charles can move onto the green buttons, and likewise with green and blue buttons. Charles can also walk over (on top of) the caged laptop if needed as the cage is not very tall and will also protect the laptop.

Being an energy conservationist, Charles wants to walk as little as possible to press all of the buttons in the proper order. Given the distance r and the n colored buttons in clockwise order, find the minimum distance required for Charles to travel to press all of the buttons in a valid order.

Charles may start at any of the buttons, and you are to find the minimum distance he travels from the first button he presses to the last button he presses. He starts measuring distance walked as soon as he touches a button. Also, Charles does not need to measure the distance from the last button to his laptop.

The Problem:

Given the distance, r, which represents the distance between each button and Charles' laptop and the colors of n buttons, output the minimum distance required for Charles to walk in order to press all of the buttons in a valid order.

The Input:

The first line contains a single, positive integer, t, which represents the number of scenarios.

For each scenario, the first line contains two integers, n ($3 \leq n \leq 15$) and r ($1 \leq r \leq 2{,}000$), representing the number of buttons that surround the laptop and the distance between each button and the laptop, respectively. The next line contains a string s of n characters all consisting of the letters R, G, or B which represents the colors of the buttons in clockwise order.

The Output:

For each scenario, output "`Scenario #`*i*`:` *d*" where i is the scenario number in the input (starting with 1) and d is the minimum distance walked (rounded to 3 decimal places; note that 0.0014 should round to 0.001 and 0.0015 should round to 0.002) required for Charles to press all of the buttons in a proper order.

Sample Input:

```
2
5 10
RGRBG
15 8
RGBRGBRGBRGBRGB
```

Sample Output:

```
Scenario #1: 61.554
Scenario #2: 119.508
```

Christmas Lights

Filename: `lights`

Christmas is coming and it's time to put up some lights! Unfortunately, your local homeowners association requires that you use a new color pattern for your lights each year. That way the neighborhood stays interesting from year to year. However, the local homeowners association also requires that the lights contain a repeating pattern. Specifically, they require that the set of lights can be divided into two or more consecutive groups where the order of the lights in each group is exactly the same. Additionally, all groups must be the same size except for the last group, which can be the same or shorter in size. For example:

> "**RBG**RBG**RBG**" contains a repeating pattern of length 3 (all groups of same size)
> "**RBOWY**RBOW" contains a repeating pattern of length 5 (last group is shorter)
> "RBGRG" does not contain a repeating patttern

You have created several potential patterns for your lights, but before you start putting them up you want to make sure that you're not going to get in trouble and be forced to change them.

The Problem:

Given a string of Christmas lights, determine if they contain a repeating pattern.

The Input:

The input will begin with a single, positive integer, n, representing the number of strings of Christmas lights you want to check. Each of the following n lines will contain l ($2 \leq l \leq 10^6$) characters with no spaces between them. The i^{th} character represents the color of the i^{th} Christmas light in the pattern. All characters will be uppercase letters and each unique letter will represent the same color (for example, an "R" will always be the same color – perhaps red).

The Output:

For each pattern output a single line with "`OK`" if it contains at least 1 repeating pattern, or "`MESSY`" if it does not.

Sample Input:

```
3
RBGRBGRBG
RBOWYRBOW
RBGRG
```

Sample Output:

```
OK
OK
MESSY
```

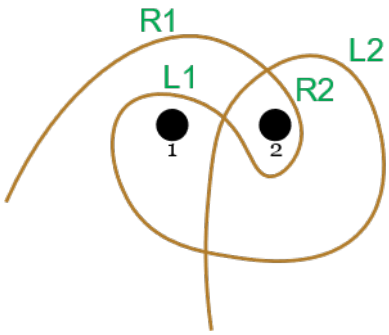
Nailed It!

Filename: `nails`

Ali loves taking photos of the UCF Programming Team! Even more than taking the photos, though, he enjoys hanging them on the wall for all of UCF to marvel. This year, he wants to take both his photo-hanging skills and his appreciation for the team to the next level by turning the wall where the photos hang into a kind of art piece. Rather than hanging each photo with a single nail, he will use a set of nails (he represents team members and coaches each with a nail).

Each picture frame has a string that extends from one corner of the frame to another, but he can cut the length of the string to be as short or as long as he needs. Since Ali believes that each and every person is vital to the team's success, he wants the string to touch all of these nails in some way, and also wants the photo to fall to the ground if any of the nails are taken out of the wall.

You are to provide Ali with a set of instructions for doing this. There are two kinds of instructions you can give him: R*x* and L*x*, which correspond to clockwise (right) and counterclockwise (left) wrappings around nail *x*. The *n* nails are numbered from 1 to *n*.



For example, for $n = 2$, one method for hanging the frame according to Ali's requirements is R1 R2 L1 L2 and is shown to the right.

The Problem:

Giving a number, *n*, corresponding to the number of nails in a wall, find a sequence of instructions for wrapping a string around these nails in such a way that a photo hangs successfully from the string after performing the steps but falls to the ground if any nails are removed.

The Input:

The first line contains a single, positive integer, *t*, representing the number of photos to hang. On each of the following *t* lines, there will be a single positive integer, *n* ($1 \le n \le 80$), representing the number of nails in the wall to use in hanging that photo.

The Output:

For each picture, output a single line. Begin the line with "`Picture #i: d` " where *i* is the picture number in the input (starting with 1) and *d* is the number of commands in the sequence. Then, print *d* commands from the set of commands described above, each separated by a single space. Note that Ali's string has a finite length so it is required that $d \le 10^6$. If there are multiple ways to do this, pick any one (provided the limitation on *d* given).

Sample Input:

```
2
1
2
```

Sample Output:

```
Picture #1: 1 L1
Picture #2: 4 R1 R2 L1 L2
```

Poker Hands

Filename: `poker`

Poker is a very popular card game that requires both skill and luck, and is played with 2 or more people. To start a game, each player is given 2 cards, which are hidden from their opponents. Several cards are also placed face up for everyone to see. First, 3 cards (called the *flop*) are placed on the table, then a 4th (the *turn*), and a 5th (the *river*) are placed.

Before the *flop*, *turn*, and *river*, all players bet on their hands (or potential hands). They can also *check* (do nothing) or *fold* (give up). A player's hand is determined by the best 5 cards among the 5 face up cards and his/her 2 hidden cards. The ranking of the hands (from best to worst) are as follows:

- Straight flush
- Four of a kind
- Full house
- Flush
- Straight
- Three of a kind
- Two pair
- Pair
- High card

Whoever has the better hand at the end of the round wins!

Ryan and Tyler like playing poker. They are great at determining what hands they have, but they always forget the rules for determining the winner. Does a straight beat a flush? Is four of a kind better than a full house? Help Ryan and Tyler by writing a program to solve their problem.

The Problem:

For every game of poker, determine the winner based on Ryan and Tyler's final hand. There will never be a tie (they already know how to handle that).

The Input:

The first line will contain a single, positive integer, *g*, representing the number of games Ryan and Tyler played.

For every game, there will be two lines, each containing one of the nine options given above, without leading or trailing spaces. The spelling and capitalization will be exactly the same as shown above. The first of the two lines will represent the result of Ryan's hand, and the second will represent the result of Tyler's. Ryan's hand will never be the same type as Tyler's hand.

The Output:

For every game, print a single line: "`Game #`*`i`*`:` " where *i* is the game number (starting with 1), followed by the winner: "`Ryan`" if Ryan's hand is better, or "`Tyler`" if Tyler's hand is better.

Sample Input:

```
3
Flush
Straight
High card
Three of a kind
Straight flush
Pair
```

Sample Output:

```
Game #1: Ryan
Game #2: Tyler
Game #3: Ryan
```

Sharon the Slayer

Filename: `slayer`

Sharon is playing a new card game, it is called "Slayers." In this game there is a Slayer and a Monster. Both the Slayer and the Monster have certain properties. The Monster has p hit points and attack power a. The Slayer has his own number of hit points q and n moves at his disposal as well as m energy points.

Each of the slayer's moves fits in one of three categories: Attack, Block, and Charge. A move is described with three integers x, y, z, representing the move's type, power, and energy cost, respectively. The move's type will be either 1, 2, or 3, depending on whether it is an Attack, Block, or Charge move, respectively.

An "Attack" move strikes the Monster, reducing its hit points by y. A "Block" move increases the Slayer's hit points by y. A "Charge" move strikes the Monster before the Monster attacks the Slayer, reducing a monster's hit points by y, similarly to an Attack. The slayer can only use each move once but in any order (though remember, he is still bounded by his energy).

The Monster has only a single move. When the Monster moves, it simply reduces the Slayer's hit points by a. The game contains only a single round (yeah, weird game) and the order that moves are played out is as follows (from first to last): any Block moves, any Charge moves, the Monster, and finally any Attack moves. The slayer can pick any moves from his list, but the total moves he uses must fit within his available energy points, m (but whatever moves he selects are applied given the order of moves given).

If the Slayer's hit points are reduced to zero, the Slayer can no longer defeat the Monster. The Slayer must select a number moves so that he can defeat the monster, given his maximum energy availability. Your job is to determine if defeating the Monster is possible.

The Problem:

Given the properties of the Slayer and the Monster, and the list of moves that the Slayer has, determine if the Slayer can defeat the Monster. The Slayer defeats the Monster by reducing its hit points to zero or less. If the Monster reduces the Slayer's hit points zero or less first, then the Slayer cannot defeat the Monster.

The Input:

The first line will contain a single, positive integer, t, representing the number of scenarios. Each scenario will begin with five integers on a line, q, n, m, p, and a, ($1 \le q \le 100$; $1 \le n \le 100$; $1 \le m \le 100$; $1 \le p \le 100$; $1 \le a \le 100$), representing the Slayer's health, number of moves, energy points, and the Monster's health and attack power, respectively. Then, n lines follow, each with three integers, x ($1 \le x \le 3$), y ($1 \le y \le 100$) and z ($1 \le z \le 100$), representing each move the Slayer has at his disposal, respectively.

The Output:

For each scenario, output "`Fight #`*i*`:`" where *i* is the number of the scenario in the input (starting with 1) followed by a single space and the appropriate outcome, either "`Win`" or "`Lose`" depending on whether or not the Slayer can defeat the monster.

Sample Input:

```
3
10 5 8 20 5
1 6 4
1 3 1
2 8 1
3 5 5
1 14 4
10 5 10 20 16
3 25 100
2 4 1
2 6 10
2 8 12
2 5 1
32 3 2 3 32
3 2 1
3 2 2
1 32 2
```

Sample Output:

```
Fight #1: Win
Fight #2: Lose
Fight #3: Lose
```