Seventh Annual
University of Central Florida

High School Programming Tournament: Online Edition

Problems

Problem Name	Filename
Hyenas in the Pridelands	hyenas
Infinite Keyboard	keyboard
Overgrown Mazes	mazes
Perfect Number of Windows	perfect
Sandwich Numbers	sandwich
Johnny's Shapes	shapes
Square Circle	squircle
Stacks on Stacks	stacks

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

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

Hyenas in the Pridelands

Filename: `hyenas`

Zazu, a red-billed hornbill, was watching over the Pridelands from the sky. To make sure the Circle of Life stays in balance, he has to keep tabs on the savannah, with the exception of the shadowy place beyond the borders. Even though news from the underground fuels his morning report, he still likes to keep personal tabs on the buzz from the bees. By the watering hole, Zazu notices zebras being stalked by hyenas, and needs to quickly report to Simba exactly how many hyenas there are. The hyenas are hiding in the tall grass, ready to pounce on the zebras at any moment, and Simba needs to know exactly how many hyenas need to be scared back to the elephant graveyard, so that no hyenas escape.

The Problem:

Given a map of a location, output the total number of hyenas that are visible.

The Input:

The first line will contain a single, positive integer, *n*, representing how many different locations Zazu sees hyenas hiding. Each location is specified by a 2-D map and will then begin on a new line with two positive integers, *r* and *c*, where *r* ($1 < r \leq 100$) represents the number of rows, and *c* ($1 < c \leq 100$) represents the number of columns, in the map, respectively. The next *r* lines will each contain *c* characters of three different capital letters: `H` for a hyena, `Z` for a zebra, and `G` for one spot of tall grass. It is guaranteed that there will always be at least one hyena.

The Output:

For each location, output a single line containing "`Location #`*i*`:` *h*" where *i* is the number of the location in the input (starting at one), and *h* is the number of hyenas in that location.

Sample Input:

```
3
2 2
HG
GZ
3 3
HHG
GGG
ZZZ
2 4
HGHG
GHGH
```

Sample Output:

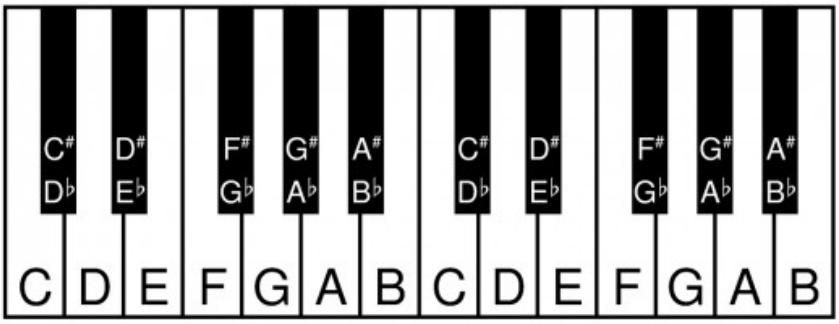
```
Location #1: 1
Location #2: 2
Location #3: 4
```

Infinite Keyboard

Filename: `keyboard`

George often visits the Land of Nowhere, home to all kinds of optical illusions, infinite staircases, and time machines. However, his favorite attraction in the Land of Nowhere is the Infinite Keyboard.

The Infinite Keyboard is a piano-like instrument whose keys continue forever in either direction. See below for the layout of a standard keyboard.



In this problem, we will use the same names for notes as in the image above. Therefore, some notes have two valid names. For example, C♯ (read as "C sharp") and D♭ (read as "D flat") refer to the same note. Moreover, each note appears in equally-spaced intervals, called *octaves* throughout the entire keyboard. Each octave has the same number and sequence of notes inside of it. The image above shows two successive octaves. Since the Infinite Keyboard has an infinite number of octaves, playing the same note in different octaves sounds *exactly the same*!

In musical terms, a *half step* is the interval between two adjacent notes on the keyboard. A white key is adjacent to the two black keys next to it, or if one does not exist, to the closest white key in that direction. For example, A and G♯ form a half step and B and C form a half step, but F and G have two half steps between them.

Today, George wants to play his favorite tunes on the Infinite Keyboard. When playing a tune, he can press each note in *any* octave, but he must play all notes in the correct order. In addition, he counts the number of half steps between each pair of consecutive notes he plays. However, George doesn't want to take forever playing a single song. Please tell him the total minimum number of half steps needed to play each tune.

The Problem:

Given George's list of songs, find the minimum number of half steps needed to play each song on the Infinite Keyboard.

The Input:

The first line of input contains a single positive integer, *s*, representing the number of songs that George wants to play. The next *s* lines each contain a single song, consisting of an integer, *n* ($1 \leq n \leq 1000$), the number of notes in the song, followed by a list of *n* musical notes. Each note is described by one or two characters in the same format as described above. That is, the first character is an uppercase letter between A and G, inclusive, and the second character, if there is one, is either '#' to denote a sharp or 'b' (lowercase b) to denote a flat. Each note will be separated from adjacent notes by a single space. Note names that are not in the image (such as C♭ or E♯ will not be used in this problem.

The Output:

For each song, output "`Song #`*i*`:` *k*" on a single line where *i* is the song number in the input (starting at 1), and *k* is the minimum number of half steps needed to play the song on the Infinite Keyboard.

Sample Input:

```
2
5 F B D# D G
4 Db C# C B
```

Sample Output:

```
Song #1: 16
Song #2: 2
```

Overgrown Mazes

Filename: `mazes`

Maize Mazers, Inc. is a company that brings mazes to cornfields around the world. They have created many mazes that have delighted customers in the past, but unfortunately, they have allowed some of their mazes to become overgrown to the point that they don't have solutions anymore. To make matters worse, they don't remember what the solutions were supposed to be.

Each of the Maize Mazers' mazes can be described by a 2-dimensional grid. Each cell in the grid is either open or filled with corn. People can walk through the open cells, but the cells that are filled with corn act as walls for the maze and can't be traversed. When customers travel through a maze, they can only move between adjacent open cells (that is, open cells that share a side). In addition, each maze has a designated start cell and end cell that don't have corn on them.

In their current state, the mazes may not have a path from the start to the end using only open cells. The Maize Mazers would like to reopen their mazes as soon as possible, so they want you to tell them the minimum number of cells that need to be cleared so that there is a path from the start to the end. Note that the new solution doesn't have to be the same as the solution to the original maze. Also, the Maize Mazers don't care how long the path to the exit ends up being (it is a maze, after all) as long as they clear the minimum number of cells to get their mazes re-opened!

The Problem:

Determine the minimum number of corn cells that need to be cleared in each of the Maize Mazers' overgrown mazes so that there is a path from the start to the end using only open cells.

The Input:

The first line of input contains a single, positive integer, *m*, representing the number of overgrown mazes. Following this line are *m* descriptions of mazes. Each maze description begins with a line with two integers, *h* and *w* ($1 \leq h \leq 300$; $1 \leq w \leq 300$; $h * w \geq 2$), representing the height and width of the maze, respectively. The next *h* lines, each containing *w* characters, represent the current state of the overgrown maze. Each character in the maze is one of the following:

- '`.`': An open cell
- '`#`': A cell filled with corn
- '`S`': The start cell
- '`E`': The end cell

Each maze will contain exactly one '`S`' and exactly one '`E`'. In addition, you may assume that customers cannot leave the boundaries of the *w* by *h* grid, even if they are at the edge.

The Output:

For each maze, output "`Maze #`*i*`: `*k*" on a single line where *i* is the number of the maze in the input (starting at 1), and *k* is the minimum number of cells that need to be cleared so that there is a path from the start to the end of the maze using only open cells.

Sample Input:

```
3
7 5
S....
####.
...#.
.###.
.#...
.####
....E
1 3
S.E
5 9
.........
.#######.
.#S###E#.
.#######.
.........
```

Sample Output:

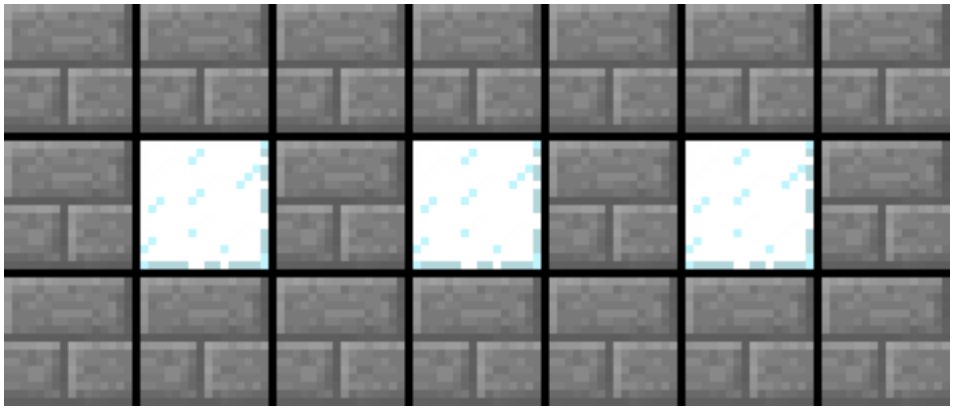
```
Maze #1: 1
Maze #2: 0
Maze #3: 2
```

Perfect Number of Windows

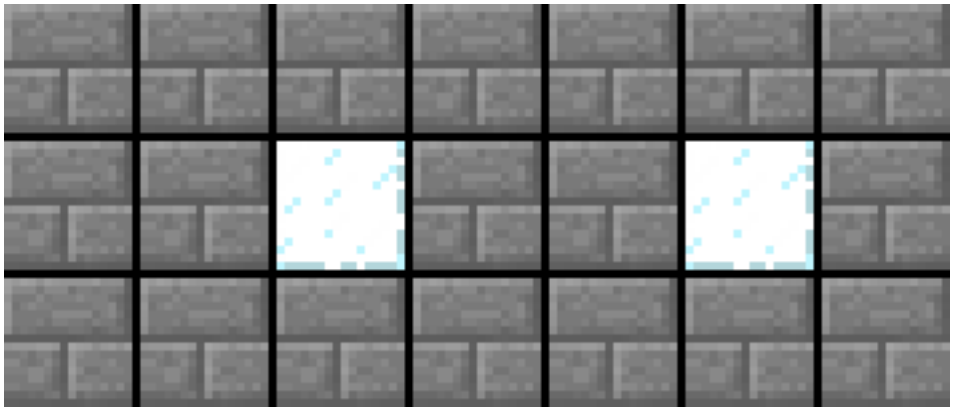
Filename: `perfect`

Minecraft Steve was planning his house one day and he started thinking about placements of windows along the walls. His walls are built with a 2-D array of blocks. He noticed that depending on the wall's width and the number of windows he wanted, it might not be possible to make it so that every window had the same number of blocks between them and between the sides of the wall.

As an example, let's say Steve has a wall that is 7 blocks wide and wanted windows that are 1 block wide. A perfect window plan would be to have gaps of 1 block, which would allow for 3 windows:



If Steve were to try to have 2 windows on the same wall, the gaps would not all be the same number of blocks. The last gap is only 1 block wide while the other gaps are 2 blocks wide:



This is not a perfect window plan.

The Problem:

Given the width of the wall, the number of blocks per window, and a range for the desired amount of windows, determine the perfect window plans that will work for Steve's house. A window plan consists of a number of windows and the number of blocks per gap. The first gap is between the start of the wall and the first window, and the last gap is between the last window and the end of the wall. All windows are one block tall and a gap must be at least one block wide.

The Input:

The input will begin with a positive integer, *t*, indicating the number of walls. Each wall will be on a new line and will contain four integers, *w*, *s*, *a*, and *b* ($3 \leq w \leq 1{,}000$; $1 \leq s < w$; $1 \leq a \leq b \leq w$), where *w* is the width of the wall, *s* is the number of blocks per window, *a* is the smallest desired number of windows, and *b* is the largest desired number of windows.

The Output:

The output for each wall should begin with "`Wall #`*w*`: `*p*" where *w* is the wall number in the input (starting with 1), and *p* is the number of perfect window plans Steve can use on his house. The next *p* lines should contain two integers each, *n* and *m*, separated by a single space where *n* is the number of windows and *m* is the size of the gaps between each window and the edges of the wall. Output each perfect window plan in increasing order of the number of windows. If no perfect window plans can be constructed given the desired range of windows, output only "`Wall #`*w*`: 0`" where *w* is the wall number. Output a blank line after the output for each wall.

Sample Input:

```
3
7 1 1 7
6 2 2 2
10 4 1 10
```

Sample Output:

```
Wall #1: 2
1 3
3 1

Wall #2: 0

Wall #3: 1
1 3
```

Sandwich Numbers

Filename: `sandwich`

Researchers at the Hidalgo Site of Peculiar Testing (HSPT) are very interested in a class of numbers called sandwich numbers. A sandwich number is a non-negative integer with a common non-overlapping prefix and suffix. For example, 101 is a sandwich number since it shares a non-overlapping prefix (1) and suffix (1). 2323 is also a sandwich number since it shares a non-overlapping prefix (23) and suffix (23). However, 123456 is not a sandwich number since it does not have a common prefix and suffix, and 2 is not a sandwich number because its prefix and suffix, though they are the same, overlap. The researchers are particularly interested, however, in determining the number of sandwich numbers less than or equal to another number. Can you help them determine this?

The Problem:

Given a non-negative integer, determine how many sandwich numbers are less than or equal to it.

The Input:

The input will begin with a single, positive integer, *n*, denoting the number of test numbers the researchers will give you. The next *n* lines each contain a single integer, *m* ($0 \leq m \leq 10^{12}$), the number they are interested in.

The Output:

For each test number, output a single line in the format "`Number #`*`i`*`: There are `*`s`*` sandwich numbers that meet our criteria.`" where *i* is the i^{th} test number under consideration (starting at 1), and *s* is the number of sandwich numbers less than or equal to the number given to you.

Sample Input:

```
3
10
100
500
```

Sample Output:

```
Number #1: There are 0 sandwich numbers that meet our criteria.
Number #2: There are 9 sandwich numbers that meet our criteria.
Number #3: There are 49 sandwich numbers that meet our criteria.
```

Johnny's Shapes

Filename: `shapes`

Johnny, a very talkative kindergartener, learned all about shapes yesterday during class and became entranced by triangles and quadrilaterals. He spent the rest of the day yapping about his new obsession to his classmates, his teachers, his mother – anyone who would listen. When it continued the next day, his mother decided that she needed a way to calm his seemingly irrepressible chattiness. She has devised a plan to preoccupy Johnny with books about shapes from the local library.

Johnny's mother would have no problem choosing the first few books available in the library – if it weren't for a peculiar idiosyncracy of his. You see, Johnny is a very emotional five-year-old. So deeply did he fall in love with his beloved shapes that he began to indiscriminately despise all shapes with more than four sides! If his mother were to accidentally include a book with a shape of this kind, she fears that he would become distraught, leaving her with an even worse situation than before. To complicate matters even further, she has discovered that the library has thousands of lengthy titles on shapes for her to choose from, making the task of locating books with only triangles and quadrilaterals rather significant. She has decided that she will need a program that determines whether or not a shape in a book will be safe to show to Johnny, and you have offered your assistance in writing it.

The Problem:

Given the number of sides of a particular shape, determine if the shape formed by those sides is one that Johnny will be happy to see.

The Input:

The input will begin with a single, positive integer, *n*, denoting the number of shapes to check. Following that will be *n* lines, with each line containing one integer, *m* ($3 \leq m \leq 500$), for the number of sides of that shape to check.

The Output:

For each shape, output a single line with one of the following, where *i* is the number of the shape corresponding to the i^{th} line:

"`Shape #i: Johnny's favorite!`" if the shape is one that Johnny likes.

"`Shape #i: Johnny will not be pleased with this one.`" otherwise.

Sample Input:

```
3
3
4
5
```

Sample Output:

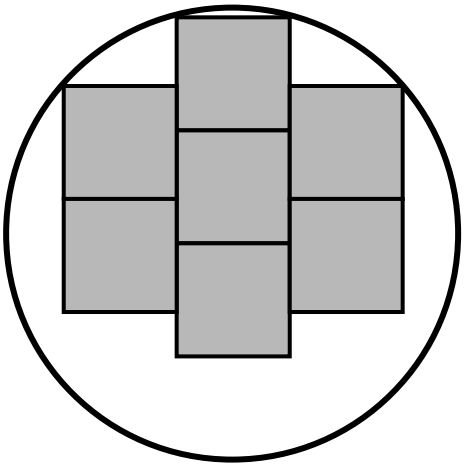
```
Shape #1: Johnny’s favorite!
Shape #2: Johnny’s favorite!
Shape #3: Johnny will not be pleased with this one.
```

Square Circle

Filename: `squircle`

Jimmy has recently become obsessed with circles. However, he does not quite realize that circles are circular, not square. He, thus, is trying to measure the area of his circles by using nothing but squares. Obviously, this is not doing him any good because he seems to always be underestimating the areas of his circles, but he does not know any better!

Jimmy takes his white circle and begins to cover it with his gray squares (all of the squares he uses are the same size) in the following way: first, he places a square at the very topmost possible location in his circle and then continues to place squares directly under it until no more can fit. Then, if possible, he places more squares directly to the left and right of his stack of squares as high as possible and again puts more squares underneath. He repeats this process until no more squares can fit. Below is an example of what the final product looks like.



Jimmy then counts how many squares fit into the circle and multiplies that by the area of one square (he knows how to find the area of a square) to get his estimate for the area of the circle.

The Problem:

Your task is to create a program that, given the radius of a circle and the side length of the square tiles Jimmy has, calculate how much Jimmy will estimate when calculating the area of the circle using his method. Jimmy has a vast collection of square tiles, and, thus, he will never run out of squares. If you need the value of π, use 3.141592653589793.

The Input:

The first line of the input will contain a single, positive integer, *n*, representing the number of times Jimmy will aimlessly try to measure the area of circles with squares. Each of the next *n* lines will contain two positive integers, *r* and *s* ($1 \leq r \leq 1{,}000$; $1 \leq s \leq 1{,}000$), representing the radius of the circle being measured and the side length of the squares Jimmy is using, respectively.

The Output:

For each measure, output "`Measure #i: a`" where *i* is the number of the test case in the order given in the input (starting with 1) and *a* is the area (in units squared) Jimmy estimates the circle to be using his squares.

Sample Input:

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

Sample Output:

```
Measure #1: 28
Measure #2: 1
Measure #3: 0
```

Stacks on Stacks

Filename: `stacks`

Jerry was bored, so he decided to create one long row of stacks of coins from his giant coin collection. Jerry likes it when the number of coins in each stack has a different parity (odd vs. even) than the number of coins in each adjacent stack. For example, if one stack had an even number of coins, the stacks to its left and right would have an odd number of coins.

Unfortunately, after Jerry had created his row of coin stacks, his little brother came in and added more coins to the stacks, but did not necessarily keep the same pattern Jerry had used (luckily, his little brother did not remove any and only added new coins!). Now Jerry has to fix his coin stacks! He does not want to remove coins, but he also does not want to add a lot of extra coins to fix his stacks, so he's asked you to write a program to determine the minimum number of coins he needs to add to his stacks to guarantee that each adjacent stack of coins has a different parity.

The Problem:

Given the number of coins in each stack after Jerry's little brother added extra coins, determine the minimum number of coins which need to be added to some of the stacks to guarantee each adjacent stack of coins has a different parity.

The Input:

The input will begin with a single, positive integer, n, denoting the number of days Jerry's little brother adds more coins to Jerry's coin stacks. For each of these instances, on a new line you will be given a single integer, m ($2 \leq m \leq 10{,}000$), representing the number of stacks of coins Jerry has laid out in a row. The following line then contains m space-separated integers, c_i ($1 \leq c_i \leq 100{,}000$), which represent the number of coins in each tower in order from beginning to end after Jerry's brother added extra coins.

The Output:

For each day's row, output a single line in the format "`Row #`*`i`*`: Jerry needs to add a minimum of `*`c`*` coins`" where i is the row of stacks of coins under consideration in the input (starting at 1), and c is the minimum number of coins needed to add to the stacks to guarantee each adjacent stack of coins has a different parity.

Sample Input:

```
2
8
1 18 17 20 25 100 203 44
4
26 26 28 28
```

Sample Output:

```
Row #1: Jerry needs to add a minimum of 0 coins
Row #2: Jerry needs to add a minimum of 2 coins
```