Ultimate SOFTWARE
People first.

Infinite Energy

J.P.Morgan

2016 University of Florida ACM
High School Programming Contest
Problem Set

Problem ID	Problem Name
deadlock	Avoiding Deadlock
dna	DNA Replication
ekg	Electrokardiogram
integral	Polygonal Integral
kerning	Kerning
maxima	Left-to-Right Maxima
multiply	RSA Cryptosystem
rgb	RGB Mixing
roster	Dispatch Roster
zamboni	Smoothing Ice

Problem ID: deadlock
Avoiding Deadlock

The ecosystem of a computer typically consists of files and processes. *Files* allow for temporary/permanent data storage and *processes* encapsulate functionality related to executing compiled code. In order for operating systems to make it seem like many processes are running simultaneously a technique called *time slicing* is used. A simplistic view of time slicing says that every 20 milliseconds a new process is given 100 milliseconds of time on the CPU (that is, a new process gets to run for 100 milliseconds every 20 milliseconds). Processes on most systems run independently from one another and are therefore not allowed to peek into each other's address space (where program data such as variables are stored). Sometimes, however, it is important for processes to work together to fulfill a common goal and this is where files become very useful.

If process A wants to send the message "Hello World" to process B, A just has to open a file and place the message "Hello World" inside of it. Then process B can simply read the message, clear the file, and wait for another message to be written! Unfortunately there is a huge problem with this method of inter-process communication: when does process B know that process A has finished writing to the file? If process A can only write "Hello W" to the file during its 100 millisecond time slice, then process B won't get the full message! This is where the concept of a *lock* comes into play. A lock is essentially a data structure that supports two operations: $lock(L)$ and $unlock(L)$.

Suppose we have some lock named L. If we call $lock(L)$ for the first time, the process that made this call *acquires* the lock. A call to $unlock(L)$ by the same process will *free* the lock. If a call such as $lock(L)$ is made while L is already acquired, then the process that made the call will wait until the lock is freed. With locks we can now pass the "Hello World" message safely.

Not all processes that use locks work correctly. Suppose there is a process A that has lock L_1 and process B that has lock L_2; after acquiring those locks, the processes then try and acquire the others. At this point, neither process can proceed because they're waiting on locks that can't ever be released! This situation is known as *deadlock*. Deadlock can occur with more than two processes; more formally, deadlock occurs when there is a set of waiting processes $\{P_1, P_2, ..., P_n\}$ such that P_1 is waiting for a lock held by P_2, P_2 is waiting for a lock held by P_3 and so on until P_n is waiting for a lock held by P_1.

Given a set of processes, the locks they're currently holding, and the locks they're waiting on, can you determine if a deadlock is occurring?

Input

The input will begin with a line containing a single positive integer, t, representing the number of test cases to process. Each test case will begin with two space-separated integers N and M ($1 \leq N, M \leq 1,000$): the number of processes running and the number of locks available. Following will be a line containing an integer K. The next K lines will be either of the form "L i j" or "H i j" ($1 \leq i \leq N$, $1 \leq j \leq M$). "L i j" says that process P_i is waiting on lock L_j and "H i j" says that process P_i has lock L_j. A process can have multiple locks but can only wait on a single lock. A process will never wait on a lock that it owns. A lock can only be owned by a single process.

Output

For each test case print "Yes" if the processes are in a deadlock, otherwise "No" (on its own line).

Sample Input	Sample Output
2 3 3 6 H 1 1 H 2 2 H 3 3 L 1 2 L 2 3 L 3 1 3 2 4 H 1 1 H 2 2 L 3 2 L 1 2	Yes No

Problem ID: dna
DNA Replication

Deoxyribonucleic acid (also known as *DNA*) is a molecule that carries most of the genetic instructions used to power life on Earth. At a high level, DNA is composed of the fundamental primitives adenine (A), thymine (T), guanine (G), and cytosine (C) which are collectively known as nucleotides. Nucleotides are essentially the 1s and 0s of life (except DNA is more like base-4 instead of base-2). *DNA sequencing* is the process of determining the precise order of nucleotides within a given DNA molecule; the output of this process is a string containing 'A', 'T', 'G', or 'C'.

A *repeated substring* is a substring that has at least two occurrences in the given string (occurrences can overlap). Repeated substrings of a DNA sequence are important because they often indicate the start/stop of an important sequence. If a repeated substring is the longest out of all of them we call it a *longest repeated substring*. It is possible for there to be multiple longest repeated substrings.

Given a string corresponding to a sequenced DNA molecule, can you determine the length of a longest repeated substring as well as the number of distinct repeated substrings that have this length?

Input

The input will begin with a line containing a single positive integer, t, representing the number of test cases to process. Each test case will consist of a single line containing the characters 'A', 'T', 'G', or 'C'. The input string is guaranteed to be at most 50 characters long.

Output

For each test case print two space-separated integers on their own line: the length of a longest repeated substring and the number of distinct repeated substrings that have this length. Output "0 0" if there is no repeated substring.

Sample Input	**Sample Output**
6 ATATA ATATAT CCCCC ACTGGGACT ACTG CCCAGGG	3 1 4 1 4 1 3 1 0 0 2 2

Problem ID: ekg
Electrokardiogram

An *electrokardiogram* is a graph of the electrical activity of a person's heart. Every millisecond, the current voltage going through the heart is recorded and plotted.

Figure 1: An example electrokardiogram (EKG).

A healthy heartbeat has a fairly regular EKG. As Figure 1 shows, there is a pattern of a quick high voltage followed by small bumps of low voltage. High voltages indicate that the heart is being electrically stimulated, whilst low voltages indicate that the heart is preparing for recovery.

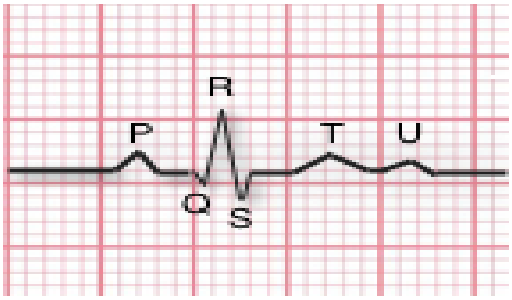

Figure 2: An electrokardiogram (EKG) with its waveforms labelled.

Figure 2 labels the specific patterns of electrical activity found on an EKG. For example, the sequence "QRS" (also known as the QRS-complex) indicates the ventricles of the heart are being stimulated to contract. Sometimes, however, it's possible for the heart to not actually contract during a QRS phase — a condition known as *electromechanical dissociation*.

A simplified view treats an EKG as a sequence of the characters 'P', 'Q', 'R', 'S', 'T', 'U', and '-' (where '-' indicates no observable voltage) corresponding to the voltage measurements every millisecond. Then, Figure 2 would be written as the sequence 'P-QRS-TU-'. We say that a heartbeat is *regular* if the time between consecutive R voltages is the same; otherwise, the heartbeat is called *irregular*.

Given an EKG, can you determine if it corresponds to a heartbeat that is *regular* or *irregular*?

Input

The input will begin with a line containing a single positive integer, t, representing the number of test cases to process. Each test case will consist of a single line containing the characters 'P', 'Q', 'R', 'S', 'T', 'U', and '-'. Every EKG is guaranteed to have at least three R voltages.

Output

For each test case print "Regular" or "Irregular" on its own line depending on whether the EKG corresponds to a regular or irregular heartbeat.

Sample Input	Sample Output
3 -P-QRSTU-P-QRSTU-P-QRSTU-P-QRSTU -R---R--R---R-R------- -R--R--R-----------	Regular Irregular Regular

Problem ID: integral
Polygonal Integral

The *integral* of a function $f(x)$ from a to b, denoted

$$\int_a^b f(x)dx,$$

is a mathematical object that represents the area between the x-axis, $f(x)$, and the lines $x = a$ and $x = b$. Pictured below is the the graph $f(x) = x^2$ along with the integral $\int_{0.5}^{1.5} x^2 dx$ shaded in red.

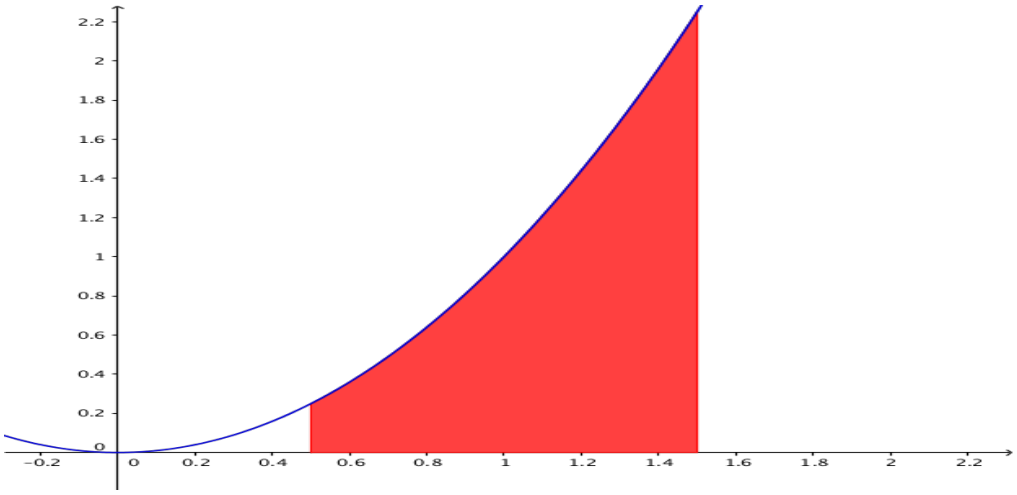

Figure 1: An example integral of a function.

In general it is fairly difficult to compute integrals without a smorgasbord of algebraic transformations. However, it is possible to generalize the integral and apply it to mathematical objects other than functions (some of which are actually easier to compute algorithmically)! A *polygonal chain* is a sequence of points $(x_0, y_0), (x_1, y_1), ..., (x_N, y_N)$; every consecutive pair of points gives us a line segment. For example, we draw the the polygonal chain $(1, 5), (3, 7), (5, 2), (10, 9)$ as follows:

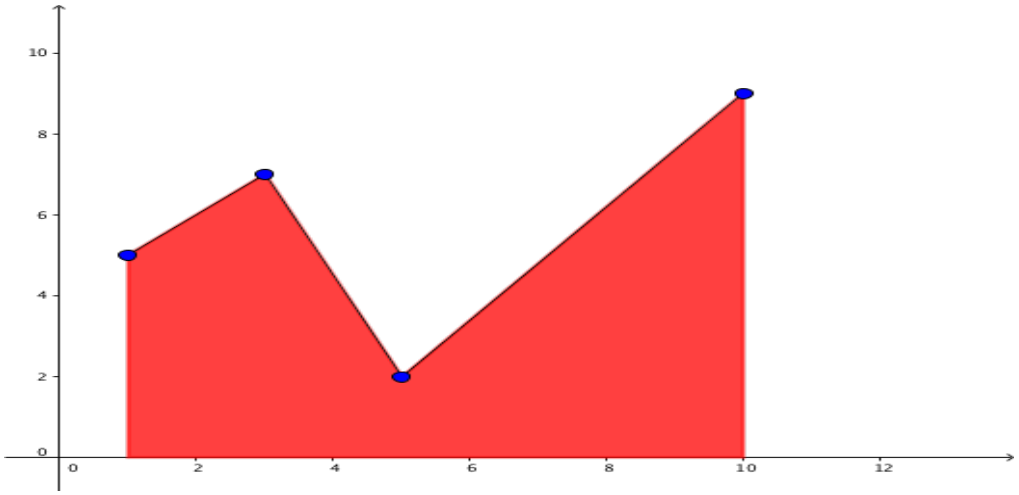

Figure 2: An example integral of a polygonal chain.

If we restrict ourselves to looking at polygonal chains where $x_0 < x_1 < ... < x_N$, we can then define the integral of this polygonal chain as the area between the x-axis and the polygonal chain between x_0 and x_N.

Given a polygonal chain, can you compute its integral?

Input

The input will begin with a line containing a single positive integer, t, representing the number of test cases to process. Each test case will begin with an integer N ($2 \leq N \leq 10,000$), the number of points in the polygonal chain. Following will be N lines giving the polygonal chain. The i-th line will be of the form "x_i y_i" ($0 \leq x_i, y_i \leq 10,000$, x_i and y_i are both integers). It is guaranteed that $x_i < x_{i+1}$ for all i.

Output

For each test case print the area between the given polygonal chain and the x-axis rounded to two decimal places on its own line.

Sample Input	Sample Output
1 4 1 5 3 7 5 2 10 9	48.50

Problem ID: kerning
Kerning

In typography, *kerning* is the process of adjusting the spacing between characters (also known as *glyphs*) in a font. A well-kerned font is one where the spacing between individual characters is visually appealing. Fonts are typically represented as a two dimensional array of bit pixels (black or white) where glyphs are placed side by side and are kerned appropriately.

Figure 1: A well(ish)-kerned font represented as a bitmap.

00011111
00110000
01000000
00110000
00011111

Figure 2: A 5x8 bitmap. The glyph is highlighted in red.

Let's say that '1' represents a bit pixel that is part of a glyph and '0' represents a bit pixel that is part of the background. Then, we formally define a glyph as a maximal set of '1' bit pixels with the property that we can find a path from every bit pixel in the set to every other one (a path consisting of only up/down, left/right, and diagonal moves). Suppose we are given a bitmap that contains exactly two glyphs. The *level of kerning* between these two glyphs is the minimum distance between any point of the first and second glyph (where distance is measured using the square of the euclidean distance). More specifically, the distance between two points (x_0, y_0) and (x_1, y_1) is $(x_0 - x_1)^2 + (y_0 - y_1)^2$; the first coordinate is the row number (starting at the top as 0) and the second coordinate is the column number (starting from the left as 0).

Input

The input will begin with a line containing a single positive integer, t, representing the number of test cases to process. Each test case will begin with a single line containing two space-separated integers N and M ($1 \leq N, M \leq 100$) each representing number of rows and columns in the bitmap, respectively. Following will be N lines each containing M characters (which will be either '0' or '1') which collectively are the bitmap for this test case. It is guaranteed that each bitmap will contain *exactly* two glyphs.

Output

For each test case print the level of kerning between the two glyphs on its own line.

Sample Input	Sample Output
2 6 20 00000000000000000000 11111111000011000000 00011000000011000000 00011000000011100000 00011000000011111110 00000000000000000000 5 10 0000111000 0010001000 1110001000 0010001000 0000000100	25 5

Problem ID: maxima
Left-to-Right Maxima

A *permutation* of length N is an ordering of the numbers $1, 2, ..., N$. For example, 321 and 213 are both permutations of length 3. A number in a permutation that's greater than everything to its left is called a *left-to-right maxima*. The permutation 213 has two left-to-right maxima (2 and 3), but the permutation 321 only has one (3).

We can put a permutation into *canonical cycle form* by breaking it into groups starting at each left-to-right maxima. The canonical cycle form of 213 is then (21)(3) (2 and 1 are in the same group, 3 is in a group by itself). Here's another more complicated permutation written in canonical cycle form (note that the left-to-right maxima are all bolded):

312548976	(**3**12)(**5**4)(**8**)(**9**76)

The canonical cycles forms of length 3 permutations are:

123	(**1**)(**2**)(**3**)
132	(**1**)(**3**2)
312	(**3**12)
321	(**3**21)
213	(**2**1)(**3**)
231	(**2**)(**3**1)

For this problem, we would like to know how many permutations of length N have both i and j in the same group.

Input

The input will begin with a line containing a single positive integer, t, representing the number of test cases to process. Each test case will consist of three space-separated integers N, i, and j ($1 \leq i, j \leq N \leq 19$).

Output

For each test case print the number of permutations of length N that have i and j in the same groups when put into canonical cycle form, on its own line.

Sample Input	Sample Output
2 2 1 2 3 1 1 4 3 4	1 6 12

Problem ID: multiply
RSA Cryptosystem

Your friends Alice and Bob are very secretive people. Whenever they send a message to each other they encrypt it using the *RSA* algorithm. For the algorithm to work, Alice and Bob must each pick two very large prime numbers p and q and from these two numbers they can follow the RSA procedure to generate their own public-key and private-key.

To send an encrypted message to Alice, Bob would have to take her public-key and encrypt his message with it; the only way to decrypt this message is with the private-key that Alice has. One part of the public-key that Alice releases is the cryptographic modulus: $n = p * q$. Alice and Bob have decided to enlist your efforts to help them compute this cryptographic modulus.

Alice and Bob will provide two numbers, p and q ($1 \leq p, q \leq 10^{1000}$). Fortunately, they aren't very good at math and don't really know what prime numbers are; instead they will provide you numbers of a very specific form. Both numbers will be a single non-zero digit followed by zeros (it's possible for there to be no zeros following the non-zero digit, but there will always be a non-zero digit that starts).

Input

The input will begin with a line containing a single positive integer t representing the number of modulus values you must compute. Following will be t lines each containing two space-separated integers p and q ($1 \leq p, q \leq 10^{1000}$).

Output

For each test case, print the modulus ($p * q$) on its own line.

Sample Input	**Sample Output**
4 5 9 700000 700000 10 1 1000000 9	45 490000000000 10 9000000

Problem ID: rgb
RGB Mixing

On many systems, colors are represented using the *RGB* color scheme. The basic idea behind this scheme is that every color is composed of a mixture of the colors red, green, and blue. More specifically, each color is assigned a value between $0 - 255$ representing the amount of either red, green, or blue that is part of the color. The color violet is represented by the RGB values R: 200 G: 100 B: 255.

If we're given two colors with their associated RGB values (lets say color 1 has R_1, G_1, B_1 and color 2 has R_2, G_2, B_2) we can add them together to create a new color with RGB values $R_1 + R_2$, $G_1 + G_2$, $B_1 + B_2$ (of course, their values can't exceed 255).

Suppose you're given a set of N colors which you can use to compose new colors with. Can you determine if it's possible to create a color with RGB values R, G, and B?

Input

The input will begin with a line containing a single positive integer, t, representing the number of test cases to process. Each test case will begin with a single line containing an integer N ($1 \leq N \leq 10$). The next line will contain three space-separated integers R, G, and B ($0 \leq R, G, B \leq 255$). Following will be N lines, the i-th of which will be of the form "R_i G_i B_i" which corresponds to the i-th color ($0 \leq R_i, G_i, B_i \leq 255$).

Output

For each test case print "Yes" if you can create the color using the component colors provided; "No" otherwise. Each color may only be used once.

Sample Input	**Sample Output**
3 2 255 255 255 150 150 150 175 175 175 3 127 239 119 100 200 100 12 15 19 15 24 0 2 200 100 255 200 100 250 5 5 5	Yes Yes No

Note: For the first test case, we mix together both of the colors provided. Since the sum of each individual component is maxed out by 255, summing these two colors together gives us the goal value of 255 for each component.

Problem ID: roster
Dispatch Roster

In another effort to enlist your services, the Gainesville Computer Company (GCC) decided that their truck dispatch roster needs to be computerized.

Every day N trucks are dispatched from the GCC warehouse (at different times), each headed to a different location; additionally, trucks may hold different amount of computers (some stores order different amounts!). More specifically, the i-th truck is dispatched at minute t_i with a load of k_i computers and takes exactly s_i minutes to reach its target location.

Now, the Gainesville Computer Company is interested in knowing how many computers are on the road at every point in time. Can you help them?

Input

The input will begin with a line containing a single positive integer, t, representing the number of test cases to process. Each test case will begin with two integers N and M ($1 \leq N \leq 100,000$, $1 \leq M \leq 100,000$); N is the number of trucks that are going to be dispatched and M is the amount of minutes the dispatch roster will keep track of. Following will be N lines, each corresponding to one truck that is going to be dispatched. The i-th line is of the form "t_i s_i k_i" ($s_i \geq 1$, $t_i \geq 1$, $s_i + t_i \leq M + 1$, and $k_i \leq 10$), which have the same meaning as defined above.

Output

For each test case print M space-separated integers on a single line, the i-th integer of which should be the number of computers that are on the road at minute i. The output for each test case should be on its own line.

Sample Input	**Sample Output**
2 4 5 1 1 3 2 1 5 3 1 6 1 5 10 1 10 4 2 10	13 15 16 10 10 0 0 0 10 10 0 0 0 0 0

Problem ID: zamboni
Smoothing Ice

An ice resurfacer (also known as a *Zamboni*) is a vehicle used to create a smooth layer of ice in an ice rink. We will model an ice rink as a two-dimensional grid with N rows and M columns (where the top-left most position is designated as $(0, 0)$ and the bottom-right most position is $(N - 1, M - 1)$). Each spot on the grid represents a patch of ice in the ice rink; a patch can be either *rough* or *smooth*. Initially each patch of ice is rough.

The Zamboni wants to clean the ice as fast as possible, but exactly how long will it take? Initially, the Zamboni starts on position (x, y) facing north. Whenever the Zamboni is on top of a patch of ice, the ice gets smoothed. The Zamboni follows a very regular pattern to clean the ice; at each step, the Zamboni moves t times in its current direction (initially $t = 1$), turns 90 degrees clockwise, and then increments t (that is, the next step the Zamboni will move over $t + 1$ patches of ice). *Note: a step is defined as the whole process of moving t steps, changing direction, and incrementing the counter. Namely, the Zamboni may move over multiple patches of ice in the same step!* There's one more catch, however... the grid wraps around! If the Zamboni moves off an edge, it emerges on the opposite side of the grid. The figures below show the steps that the Zamboni must go through to clean a 2x2 grid starting at position $(0, 0)$.

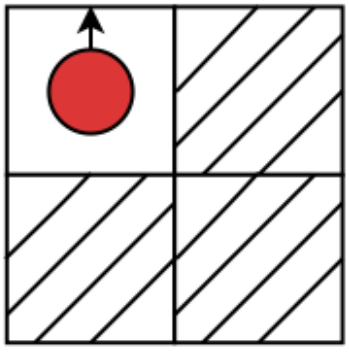

Figure 1: Initially the Zamboni faces upward.

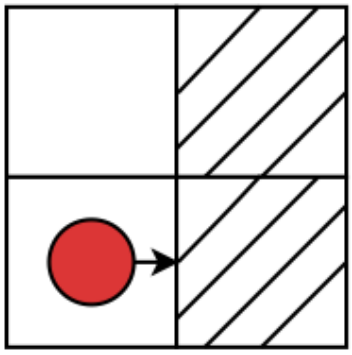

Figure 2: This is the configuration after step one.

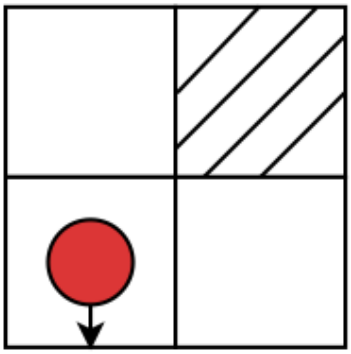

Figure 3: This is the configuration after step two.

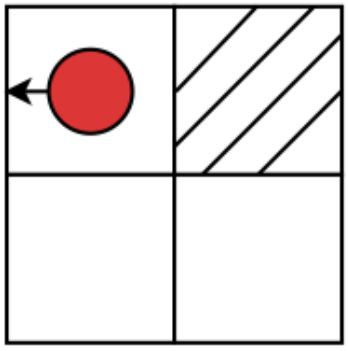

Figure 4: This is the configuration after step three.

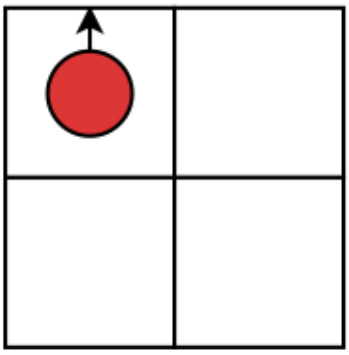

Figure 5: This is the configuration after step four.

Given the size of the ice rink and the starting position of the Zamboni, can you determine how many steps it will take to smooth all of the ice in the rink?

Input

The input will begin with a line containing a single positive integer, t, representing the number of test cases to process. Each test case will consist of the four space separated integers N, M, x, and y ($1 \leq N, M \leq 100$, $0 \leq x < N$, $0 \leq y < M$), each of which has the same meaning as defined above.

Output

For each test case print (on its own line) the number of steps it takes for the Zamboni to finish cleaning the ice.

Sample Input	**Sample Output**
4 2 2 0 0 55 55 37 37 100 100 0 0 1 1 0 0	4 164 298 0