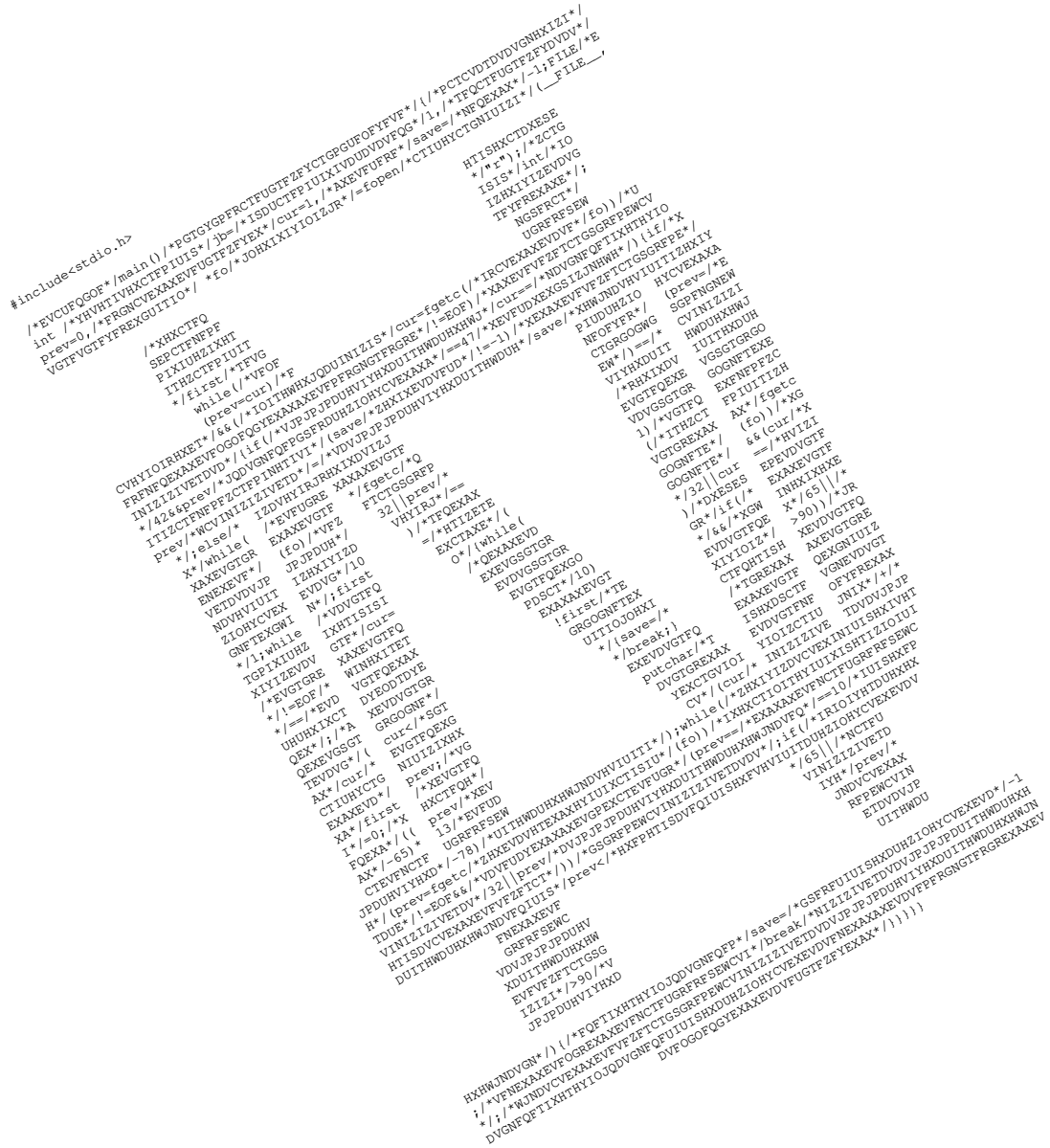
ACM East Central Region
1997 Regional Programming Contest

November 8, 1997

University of Notre Dame

Problem Set
Version 2.0



Instructions

There are 7 problems, given in no particular order of difficulty. There are 15 labeled pages in this handout. Notify a consultant immediately if you do not have all 7 problems or if you do not have all 15 pages.

Getting Help

If you have a problem with your computer or computing environment, please notify a student consultant immediately. Consultants will not give any help on the problem set.

Judges will **not** give out helpful hints on any of the programs. We will clarify any questions that you have about the problem, but no help will be given.

Input and Output

For programming purposes, you may assume that all data will be of the format specified in the problem specification. That is, no invalid data will be used to judge your program.

Unless otherwise specified in the instructions for a particular problem, you should output numerical values as indicated below. I.e., do not attempt to specially format output data unless you are asked to.

You should use double precision floating point for real numbers. For C/C++ use type `double`. For Pascal, the `real` data type corresponds to double precision in the Solaris Pascal compiler on this computing environment. Typically, you will be asked to output the floating point numbers to three decimal places, with no extra white space before or after the number. The corresponding formatting instructions for C, C++, and Pascal are as follows:

```
C       printf("%.3lf", num);
C++     printf("%.3lf", num);
Pascal  write(num :1:3);
```

Integer values are printed out normally in C and C++. However, in Pascal, the default behavior is to pad the integer with spaces. For all output involving integers, do not pad with extra space (unless asked to do so specifically by the problem). For Pascal, you can format the output like this:

```
Pascal  write(num :1);
```

Examples

Unless otherwise specified, the example output in each problem is assumed to have been produced by a program written in C, C++, or Pascal.

1 Polygon Puzzler

We define a *simple polygon* as an area enclosed by endpoint-connected line segments such that no line segment intersects another (except for adjoining segments at their endpoints). A simple polygon can thus be defined by an ordered list of its vertices (the endpoints of the enclosing line segments). A *planar polygon* is a polygon whose vertices all lie in the same plane.

For this problem you are asked to compute the area of a simple planar polygon oriented in three space. That is, although the vertices of the polygon lie in some two-dimensional plane, the vertices are specified in three-dimensional Cartesian coordinates.

1.1 Input

The input will consist of an ordered sequence of coordinates for the vertices of the polygon. Each line of the input will contain the three-dimensional Cartesian coordinates for a single vertex in the order x, y, z. The values for the x, y, z components will be separated by a single space. Input values should be considered to be double precision floating point and may be positive or negative. The coordinates of the final line of input input will be the same as the coordinates on the first line of input. No polygon will have more than 1024 vertices.

1.2 Output

The output should be the area of the polygon specified by the input and should be rounded to the nearest 1/1000 (i.e., three places after the decimal point should be printed).

1.3 Example

The following is sample input for this problem.

```
-1.401117996399998e+00 1.509291958378880e-01 1.186959898555237e-01
1.918738650437130e-01 1.067473024933127e+00 9.075713530920345e-01
1.401117996399998e+00 -1.509291958378880e-01 -1.186959898555237e-01
-1.918738650437130e-01 -1.067473024933127e+00 -9.075713530920345e-01
-1.401117996399998e+00 1.509291958378880e-01 1.186959898555237e-01
```

The following is the corresponding output for the input above.

```
4.000
```

2 Clock Hands

The medieval interest in mechanical contrivances is well illustrated by the development of the mechanical clock, the oldest of which is driven by weights and controlled by a verge, an oscillating arm engaging with a gear wheel. It dates back to 1386.

Clocks driven by springs had appeared by the mid-15th century, making it possible to construct more compact mechanisms and preparing the way for the portable clock.

English spring-driven pendulum clocks were first commonly kept on a small wall bracket and later on a shelf. Many bracket clocks contained a drawer to hold the winding key. The earliest bracket clocks, made for a period after 1660, were of architectural design, with pillars at the sides and a pediment on top.

In 17th- and 18th-century France, the table clock became an object of monumental design, the best examples of which are minor works of sculpture.

The longcase clocks (also called grandfather clocks) are tall pendulum clock enclosed in a wooden case that stands upon the floor and is typically from 6 to 7.5 feet (1.8 to 2.3 m) in height. Later, the name “grandfather clock” became popular after the popular song "My Grandfather’s Clock," written in 1876 by Henry Clay Work.

One of the first atomic clocks was an ammonia-controlled clock. It was built in 1949 at the National Bureau of Standards, Washington, D.C.; in this clock the frequency did not vary by more than one part in 10^8

Nuclear clocks are built using two clocks. The aggregate of atoms that emit the gamma radiation of precise frequency may be called the emitter clock; the group of atoms that absorb this radiation is the absorber clock. One pair of these nuclear clocks can detect energy changes of one part in 10^{14} , being about 1,000 times more sensitive than the best atomic clock.

The cesium clock is the most accurate type of clock yet developed. This device makes use of transitions between the spin states of the cesium nucleus and produces a frequency which is so regular that it has been adopted for establishing the time standard.

The history of clocks is fascinating, but unrelated to this problem. In this problem, you are asked to find the angle between the minute hand and the hour hand on a regular analog clock.[1] Assume that the second hand, if there were one, would be pointing straight up at the 12. Give all angles as the smallest positive angles. For example 9:00 is 90 degrees; not -90 or 270 degrees.

2.1 Input

The input is a list of times in the form `H:D`, each on their own line, with $1 \leq H \leq 12$ and $00 \leq M \leq 59$. The input is terminated with the time `0:00`. Note that `H` may be represented with 1 or 2 digits (for 1–9 or 10–12, respectively); `M` is always represented with 2 digits.[2]

2.2 Output

The output displays the smallest positive angle in degrees between the hands for each time. The answer should between `0` degrees and `180` degrees for all input times. Display each angle on a line by itself in the same order as the input. The output should be rounded to the nearest 1/1000, i.e., three places after the decimal point should be printed.

[1] Ask one of the consultants to show you what this type of clock looks like, if you are not sure.

[2] The input times are what you typically see on a digital clock.

2.3 Example

The following is sample input for this problem.

```
12:00
9:00
8:10
0:00
```

The following is the corresponding output for the input above.

```
0.000
90.000
175.000
```

3 Critical Mass

During the early stages of the Manhattan Project, the dangers of the new radioctive materials were not widely known. Vast new factory cities were built to manufacture uranium and plutonium in bulk. Compounds and solutions of these substances were accumulating in metal barrels, glass bottles and cardboard box piles on the cement floors of store rooms. Workers did not know that the substances they were handling could result in sickness, or worse, an explosion. The officials who new the danger assumed that they could ensure safety by never assembling any amount close to the critical mass estimated by the physicists. But mistakes were made. The workers, ignorant of the the dangers, often did not track these materials carefully, and in some cases, too much material was stored together – an accident was waiting to happen.

Fortunately, the dangers were taken seriously by a few knowledgeable physicists. They drew up guidelines for how to store the materials to eliminate the danger of critical mass accumulations. The system for handling uranium was simple. Each uranium cube was marked "U". It was to be stacked with lead cubes (marked "L") interspersed. No more than two uranium cubes could be next to each other on a stack. With this simple system, a potential for the uranium reaching critical mass (three stacked next to each other) was avoided. The second constraint is that no more than thirty cubes can be stacked on top of each other, since the height of the storage room can only accommodate that many.

One of the physicists was still not completely satisfied with this solution. He felt that a worker, not paying attention or not trained with the new system, could easily cause a chain reaction. He posed the question: consider a worker stacking the radioactive cubes and non radioactive cubes at random on top of each other to a height of n cubes; how many possible combinations are there for a disaster to happen?

For example, say the stack is of size 3. There is one way for the stack to reach critical mass – if all three cubes are radioactive.

```
1: UUU
```

However, if the size of the stack is 4, then there are three ways:

```
1: UUUL
2: LUUU
3: UUUU
```

3.1 Input

The input is a list of integers on separate lines. Each integer corresponds to the size of the stack and is always greater than 0. The input is terminated with a integer value of 0.

3.2 Output

For each stack, compute the total number of dangerous combinations where each cube position in the linear stack can either be "L" for lead, or "U" for uranium. Output your answer as a single integer on a line by itself.

3.3 Example

The following is sample input for this problem.

```
4
5
0
```

The following is the corresponding output for the input above.

```
3
8
```

4 Word Search Wonder

Word search is a game enjoyed by all ages. The basic idea is to find certain specified words within a given matrix of letters.

One interesting means of constructing a word search matrix is to take some written text (in upper case), strip out all but the characters A–Z, and format the resulting sequence of characters into columns of s characters. That is, taking $s = 9$ and given the text

> Our revels now are ended. These our actors,
> As I foretold you, were all spirits and
> Are melted into air, into thin air:
> And, like the baseless fabric of this vision,
> The cloud-capp'd towers, the gorgeous palaces,
> The solemn temples, the great globe itself,
> Ye all which it inherit, shall dissolve
> And, like this insubstantial pageant faded,
> Leave not a rack behind. We are such stuff
> As dreams are made on, and our little life
> Is rounded with a sleep.

We obtain the following word-search matrix:

```
ONERIYLNEIITSFODEGCESGLHEDNSNAEAWSEDRFEE
UODAFOSDDNRHSTNCREEMTLFIRIDITNACETAELEDE
RWTCOUPAITAEFHTASOSNHOYCISLNITVKAUMOIIWP
RAHTRWIRNONBAIHPTUTTEBEHTSISAFEBRFSNTSI
EREOEERETTDABSEPHSHEGEAISOKULANEEFAATRT
VESRTRIMOHLSRVCDEPEMRIATHLEBPDOHSARNLOH
EEESOETEAIIEIILTGASPETLIAVTSAETIUSEDEUA
LNOALASLINKLCSOOOLOLASLNLEHTGDANCDMOLNS
SDUSDLATRAEEOIUWRALETEWHLAIAELRDHRAUIDL
```

We can now search this matrix for words. For example, the word "SAFE" can be found starting at lexicographic position 246 and running eastward in the text.[3] Since, running row-wise, we are looking at every s'th character, the value of s is sometimes called a *skip* value.

For this problem, you will be given search words, a search text, and a skip value. Your job (well, really, your program's job) is to find the location of each of the search words within the search text, formatted into a search matrix according to the given skip value.

4.1 Input

The first line of input will contain a single integer specifying the skip value to use in doing the word search. The second line will contain a single integer m specifying the number of words to search for. The next m lines will contain the words to search for, one word per line. The words will be given in all uppercase. No word will be more than 32 characters in length. There will be a maximum of 128 search words. After the search words will be a line containing a single integer n giving the number of lines in the search text. The next n lines contain the search text.

[3] By *lexicographic position* we mean the numerical position with the search matrix when counting consecutively and column-wise from the first character, which is position 0.

No line of search text will be more than 128 characters long. All characters in the text will be upper case. There will be a maximum of 5452 lines.[4]

4.2 Output

Each line of output should contain a search word, the lexicographical location of the first character of the word in the search text, and its orientation (the direction in which the rest of characters are oriented with respect to the first character). Valid directions are *N*, *NE*, *E*, *SE*, *S*, *SW*, *W*, and *NW*, indicating text running in the corresponding directions within the word search text matrix. If a word occurs more than once in the search matrix, you should print out only the location of its first occurence. The words in the output should be ordered as given in the input. Each line of output should contain the search word, followed by a space, then the lexicographic position of the word, followed by another space, and finally the direction that word runs in the search text. If a word is not found in the search matrix, the line should contain the word followed by a space and then the text

```
NOT FOUND
```

4.3 Example

The following is sample input for this problem.

```
9
5
SAFE
TOES
SHAKE
SPEARE
GLOBE
11
OUR REVELS NOW ARE ENDED. THESE OUR ACTORS,
AS I FORETOLD YOU, WERE ALL SPIRITS, AND
ARE MELTED INTO AIR, INTO THIN AIR:
AND, LIKE THE BASELESS FABRIC OF THIS VISION,
THE CLOUD-CAPP'D TOWERS, THE GORGEOUS PALACES,
THE SOLEMN TEMPLES, THE GREAT GLOBE ITSELF,
YEA, ALL WHICH IT INHERIT, SHALL DISSOLVE,
AND, LIKE THIS INSUBSTANTIAL PAGEANT FADED,
LEAVE NOT A RACK BEHIND. WE ARE SUCH STUFF
AS DREAMS ARE MADE ON; AND OUR LITTLE LIFE
IS ROUNDED WITH A SLEEP.
```

The following is the corresponding output for the input above.

```
SAFE 246 E
TOES 85 SW
SHAKE NOT FOUND
SPEARE NOT FOUND
GLOBE 189 S
```

[4]The number of lines in *Hamlet*.

5 Randomly Wired Neural Nets

Introduction

The following story ("AI Koan") is from *The Hacker's Dictionary:*

> In the days when Sussman was a novice, Minsky once came to him as he sat hacking at the PDP-6.
>
> "What are you doing?", asked Minsky.
>
> "I am training a randomly wired neural net to play Tic-Tac-Toe" Sussman replied.
>
> "Why is the net wired randomly?", asked Minsky.
>
> "I do not want it to have any preconceptions of how to play", Sussman said.
>
> Minsky then shut his eyes.
>
> "Why do you close your eyes?", Sussman asked his teacher.
>
> "So that the room will be empty."
>
> At that moment, Sussman was enlightened.

A Graph Model

Graphs are powerful tools for modeling connected entities of all kinds (including randomly wired neural nets). In this problem, we wish to prune unnecessary nodes from a graph that is modeling a particular kind of neural net. In particular, we wish to remove neurons from the neural net that do not have any potential across them (or, equivalently, any flow through them) when we introduce a potential across two given nodes of the network.

Let $G = (V, E)$ be an undirected graph with $|V| = n$ vertices and $|E| = m$ edges. For two given distinct vertices v and w of G, a path $P(v, w)$ in G between v and w is called a *simple path* between v and w if no vertex of G appears in $P(v, w)$ more than once.

It is straightforward to show that a neuron in a neural net will support flow if and only if the corresponding edge in the graph model of the neural net lies on some simple path between v and w. Thus, the set of all vertices of G that are on at least one simple path between v and w represent the vertices that would be present in a model of the neural net with the unnecessary neurons removed.

Your program is to print out a modified graph G such that each vertex in the graph is present on at least one simple path between v and w.

5.1 Input

The input will consist of a graph G described as an adjacency list and two distinguished vertices, v and w. The first line of the input will contain two numbers separated by a space, indicating the distinguished vertices, v and w. The second line will contain one number n, specifying the number of vertices in G. The next n lines specify the adjacency list for the graph, one line for each vertex, ordered from vertex 0 to vertex $n - 1$. The line corresponding to vertex q (i.e., the qth line in the adjacency list portion of the input file) consists of an integer k specifying the degree of q (the number of edges connected to q), followed by k integers specifying the vertices connected to vertex q. All of the numbers in one line will be separated by a single space.

The maximum number of vertices is 1024.

5.2 Output

The output of the program will consist of an ordered list of all vertices of G such that each of the identified vertices is on some simple path in G between v and w. Lines 0 through $n-1$ should specify the connectivity for the vertices 0 through $n-1$, respectively. If a given vertex is not present in the modified graph, an X should be printed on that line. Otherwise, the vertices (in the modified graph) to which the given vertex is connected should be printed.

5.3 Example

The following is sample input for this problem.

```
0 6
8
2 1 4
5 0 2 3 5 7
1 1
1 1
2 0 5
3 4 1 7
1 7
3 1 5 6
```

The following is the corresponding output for the input above.

```
1 4
0 5 7
X
X
0 5
4 1 7
7
1 5 6
```

6 Prime Factors

Webster defines *prime* as:

prime (prīm) *n.* [**ME**, fr. **MF**, fem. of *prin* first, **L** *primus*; akin to **L** *prior*] **1** : first in time : **original 2 a** : having no factor except itself and one ⟨3 is a ~ number ⟩ **b** : having no common factor except one ⟨ 12 and 25 are relatively ~⟩ **3 a** : first in rank, authority or significance : **principal b** : having the highest quality or value ⟨~ television time ⟩ [from *Webster's New Collegiate Dictionary*]

The most relevant definition for this problem is 2a: An integer $g > 1$ is said to be *prime* if and only if its only positive divisors are itself and one (otherwise it is said to be *composite*). For example, the number 21 is composite; the number 23 is prime. Note that the decompositon of a positive number g into its prime factors, i.e.,

$$g = f_1 \times f_2 \times \ldots \times f_n$$

is unique if we assert that $f_i > 1$ for all i and $f_i \leq f_j$ for $i < j$.

One interesting class of prime numbers are the so-called *Mersenne* primes which are of the form $2^p - 1$. Euler proved that $2^{31} - 1$ is prime in 1772 — all without the aid of a computer.

6.1 Input

The input will consist of a sequence of numbers. Each line of input will contain one number g in the range $-2^{31} < g < 2^{31}$. The end of input will be indicated by an input line having a value of zero.

6.2 Output

For each line of input, your program should print a line of output consisting of the input number and its prime factors. For an input number $0 < g = f_1 \times f_2 \times \ldots \times f_n$, where each f_i is a prime number greater than unity (with $f_i \leq f_j$ for $i < j$), the format of the output line should be

$\langle g \rangle$ = $\langle f_1 \rangle$ x $\langle f_2 \rangle$ x ... x $\langle f_n \rangle$

If $0 > g = f_1 \times f_2 \times \ldots \times f_n$, the format of the output line should be

$\langle g \rangle$ = -1 x $\langle f_1 \rangle$ x $\langle f_2 \rangle$ x ... x $\langle f_n \rangle$

6.3 Example

The following is sample input for this problem.

```
-190
-191
-192
-193
-194
195
196
197
198
```

```
199
200
0
```

The following is the corresponding output for the input above.

```
-190 = -1 x 2 x 5 x 19
-191 = -1 x 191
-192 = -1 x 2 x 2 x 2 x 2 x 2 x 2 x 3
-193 = -1 x 193
-194 = -1 x 2 x 97
195 = 3 x 5 x 13
196 = 2 x 2 x 7 x 7
197 = 197
198 = 2 x 3 x 3 x 11
199 = 199
200 = 2 x 2 x 2 x 5 x 5
```

7 Bowling

History

Bowling has been traced to articles found in the tomb of an Egyptian child buried in 5200 BC. The primitive implements included nine pieces of stone at which a stone "ball" was rolled, the ball having first to roll through an archway made of three pieces of marble.

Another ancient discovery was the Polynesian game of *ula maika*, which also used pins and balls of stone. The stones were to be rolled at targets 60 feet away, a distance which is still one of the basic regulations of tenpins.

Bowling at tenpins probably originated in Germany, not as a sport but as a religious ceremony. Martin Luther is credited with settling on nine as the ideal number of pins.

Tracing history reveals the game moved through Europe, the Scandinavian countries and finally to the United States, where the earliest known reference to bowling at pins in America was made by author Washington Irving about 1818 in *Rip Van Winkle*.

Although the game was being played throughout the world, rules were different almost everywhere, and even basic equipment was not the same. In fact, why and when the 10th pin was added from the European game of ninepins to the American game of tenpins is still a mystery.

Rules

A single bowling game consists of ten *frames*. The object in each frame is to roll a ball at ten bowling pins arranged in an equilateral triangle and to knock down as many pins as possible.

For each frame, a bowler is allowed a maximum of two *rolls* to knock down all ten pins. If the bowler knocks them all down on the first attempt, the frame is scored as a *strike*. If the bowler does not knock them down on the first attempt in the frame the bowler is allowed a second attempt to knock down the remaining pins. If the bowler succeeds in knocking the rest of the pins down in the second attempt, the frame is scored as a *spare*.

The score for a bowling game consists of sum of the scores for each frame. The score for each frame is the total number of pins knocked down in the frame, plus bonuses for strikes and spares. In particular, if a bowler scores a strike in a particular frame, the score for that frame is ten plus the sum of the next two rolls. If a bowler scores a spare in a particular frame, the score for that frame is ten plus the score of the next roll. If a bowler scores a strike in the tenth (final) frame, the bowler is allowed two more rolls. Similarly, a bowler scoring a spare in the tenth frame is allowed one more roll.

The maximum possible score in a game of bowling (strikes in all ten frames plus two extra strikes for the tenth frame strike) is 300.

7.1 Input

The input will consist of a sequence of bowling game scores. Each line will contain the scores for a single game, with the scores for each roll of the ball separated by a single space. The score for a single roll will be represented by a single character — either a number indicating the number of pins knocked down, a '/' for a spare or a 'X' for a strike.

The end of input is indicated by a single line containing the text `Game Over` (terminated with a newline).

7.2 Output

Your program should output the total game score for each game in the input file. The game scores should be left justified and each score should be printed on a separate line. The order of the scores on the output should correspond to the order of the games on the input.

7.3 Example

The following is sample input for this problem.

```
1 0 1 / 2 2 X 3 3 X 1 / 3 / X 1 2
1 0 1 / 2 2 X 3 3 X 1 / 3 / 1 / X 8 0
1 0 1 / 2 2 X 3 3 X 1 / 3 / 1 / 8 / 9
Game Over
```

The following is the corresponding output for the input above.

```
108
121
120
```