2011 ACM ICPC Southeast USA Regional Programming Contest

29 October, 2011

PROBLEMS

A: Sunday Drive 1
B: Hexagram 3
C: Flooring Tiles 4
D: Vive la Difference! 5
E: Robot Navigation 6
F: Folding Game 8
G: Burnout 10
H: Family Fortune 12
I: Moving Points 14
J: Vampire Numbers 16

Hosted by:

Florida Institute of Technology

Georgia Southern University

University of West Florida

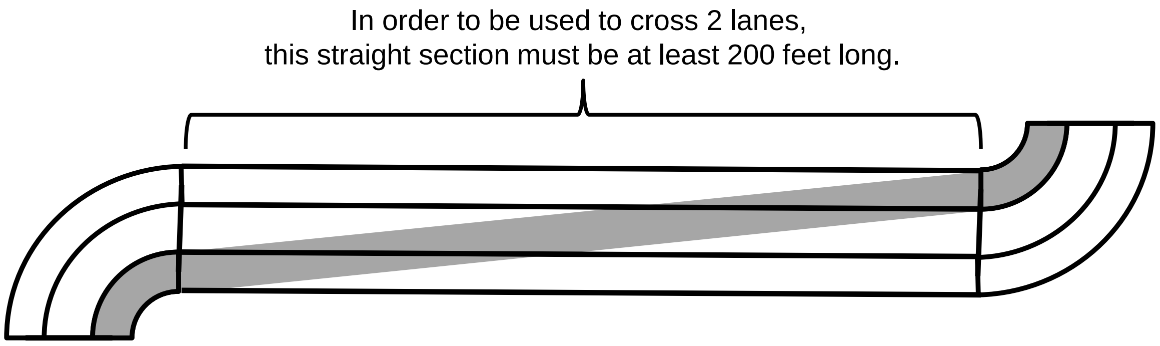
A: Sunday Drive

After wracking your brains at a programing contest on Saturday, you'd like to relax by taking a leisurely Sunday drive. But, gasoline is so expensive nowadays! Maybe, by creatively changing lanes, you can minimize the distance you travel and save some money!

You will be given a description of several sections of a highway. All sections will have the same number of lanes. Think of your car as a point mass, moving down the center of the lane. Each lane will be 10 feet wide. There are two kinds of highway sections: curved and straight. You can only change lanes on straight sections, and it takes a minimum of 100 feet of the straight section to move over one lane. You can take longer than that, of course, if you choose.

All curve sections will make 90 degree turns. You cannot change lanes on a curve section. In addition, you must be driving along the exact middle of a lane during a turn. So during a turn your position will be 5 feet, or 15 feet, or 25 feet from the edge, etc.

Given a description of a highway, compute the minimum total distance required travel the entire highway, including curves and lane changes. You can start, and end, in any lane you choose. Assume that your car is a point mass in the center of the lane. The highway may cross over/under itself, but the changes in elevation are miniscule, so you shouldn't worry about their impact on your distance traveled.


In order to be used to cross 2 lanes,
this straight section must be at least 200 feet long.

Input

There will be several test cases in the input. Each test case will begin with two integers

N M

Where **N** ($1 \le$ **N** $\le 1{,}000$) is the number of segments, and **M** ($2 \le$ **M** ≤ 10) is the number of lanes.

On each of the next **N** lines will be a description of a segment, consisting of a letter and a number, with a single space between them:

T K

The letter **T** is one of **`S`**, **`L`**, or **`R`** (always capital). This indicates the type of the section: a straight section (**`S`**), a left curve (**`L`**) or a right curve (**`R`**). If the section is a straight section, then the number **K** (10 ≤ **K** ≤ 10,000) is simply its length, in feet. If the section is a right or left curve, then the number **K** (10 ≤ **K** ≤ 10,000) is the radius of the inside edge of the highway, again in feet. There will never be consecutive straight sections in the input, but multiple consecutive turns are possible. The input will end with a line with two 0s.

Output

For each test case, print a single number on its own line, indicating the minimum distance (in feet) required to drive the entire highway. The number should be printed with exactly two decimal places, rounded. Output no extra spaces, and do not separate answers with blank lines.

Sample Input

```
3 3
R 100
S 1000
L 100
9 5
S 2500
L 500
S 2000
L 500
S 5000
L 500
S 2000
L 500
S 2500
5 4
L 100
L 100
L 100
L 100
L 100
0 0
```

Sample Output

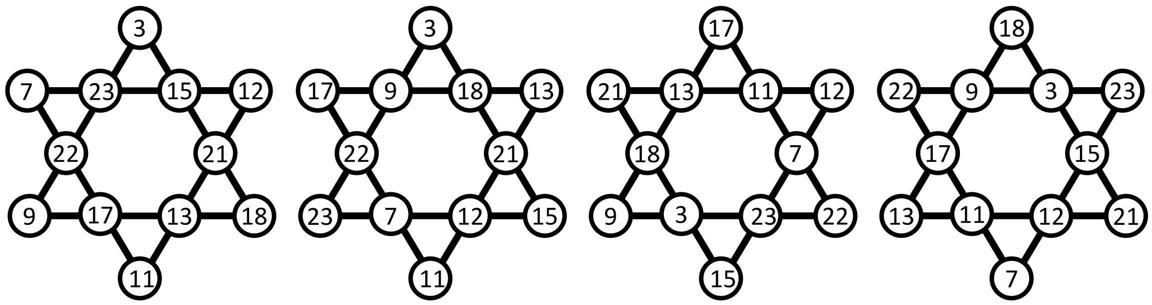
```
1330.07
17173.01
824.67
```

B: Hexagram

A Hexagram is a 6-pointed star, sometimes called the Star of David. Given these numbers:

```
3 17 15 18 11 22 12 23 21 7 9 13
```

There are four unique ways of assigning the numbers to vertices of the hexagram such that all of the sets of four numbers along the lines have the same sum (57 in this case). All other ways may be obtained from these by rotation and/or reflection.



Given 12 distinct numbers, in how many ways, disregarding rotations and reflections, can you assign the numbers to the vertices such that the sum of the numbers along each of 6 straight lines passing through 4 vertices is the same?

The Input

There will be several test cases in the input. Each test case will consist of twelve unique positive integers on a single line, with single spaces separating them. All of the numbers will be less than 1,000,000. The input will end with a line with twelve 0s.

The Output

For each test case, output the number of ways the numbers can be assigned to vertices such that the sum along each line of the hexagram is the same. Put each answer on its own line. Output no extra spaces, and do not separate answers with blank lines.

Sample Input

```
3 17 15 18 11 22 12 23 21 7 9 13
1 2 3 4 5 6 7 8 9 10 11 13
0 0 0 0 0 0 0 0 0 0 0 0
```

Sample Output

```
4
0
```

C: Flooring Tiles

You want to decorate your floor with square tiles. You like rectangles. With six square flooring tiles, you can form exactly two unique rectangles that use all of the tiles: 1x6, and 2x3 (6x1 is considered the same as 1x6. Likewise, 3x2 is the same as 2x3). You can also form exactly two unique rectangles with four square tiles, using all of the tiles: 1x4, and 2x2.

Given an integer **N**, what is the smallest number of square tiles needed to be able to make exactly **N** unique rectangles, and no more, using all of the tiles? If **N**=2, the answer is 4.

Input

There will be several test cases in the input. Each test case will consist of a single line containing a single integer **N** ($1 \le$ **N** ≤ 75), which represents the number of desired rectangles. The input will end with a line with a single 0.

Output

For each test case, output a single integer on its own line, representing the smallest number of square flooring tiles needed to be able to form exactly **N** rectangles, and no more, using all of the tiles. The answer is guaranteed to be at most 10^{18}. Output no extra spaces, and do not separate answers with blank lines.

Sample Input

```
2
16
19
0
```

Sample Output

```
4
840
786432
```

D: Vive la Difference!

Take any four positive integers: **a**, **b**, **c**, **d**. Form four more, like this:

|a-b| |b-c| |c-d| |d-a|

That is, take the absolute value of the differences of **a** with **b**, **b** with **c**, **c** with **d**, and **d** with **a**. (Note that a zero could crop up, but they'll all still be non-negative.) Then, do it again with these four new numbers. And then again. And again. Eventually, all four integers will be the same. For example, start with `1,3,5,9`:

```
1 3 5 9
2 2 4 8 (1)
0 2 4 6 (2)
2 2 2 6 (3)
0 0 4 4 (4)
0 4 0 4 (5)
4 4 4 4 (6)
```

In this case, the sequence converged in 6 steps. It turns out that in all cases, the sequence converges very quickly. In fact, it can be shown that if all four integers are less than 2^n, then it will take no more than 3*n steps to converge!

Given **a**, **b**, **c** and **d**, figure out just how quickly the sequence converges.

Input

There will be several test cases in the input. Each test case consists of four positive integers on a single line ($1 \le$ **a**,**b**,**c**,**d** $\le$ 2,000,000,000), with single spaces for separation. The input will end with a line with four 0s.

Output

For each test case, output a single integer on its own line, indicating the number of steps until convergence. Output no extra spaces, and do not separate answers with blank lines.

Sample Input

```
1 3 5 9
4 3 2 1
1 1 1 1
0 0 0 0
```

Sample Output

```
6
4
0
```

E: Robot Navigation

A robot has been sent to explore a remote planet. To specify a path the robot should take, a program is sent each day. The program consists of a sequence of the following commands:

- `FORWARD` *`X`*: move forward by *X* units.
- `TURN LEFT`: turn left (in place) by 90 degrees.
- `TURN RIGHT`: turn right (in place) by 90 degrees.

The robot also has sensor units which allow it to obtain a map of its surrounding area. The map is represented as a grid. Some grid points contain hazards (e.g. craters) and the program must avoid these points or risk losing the robot.

Naturally, if the initial location of the robot, the direction it is facing, and its destination position are known, it is best to send the shortest program (one consisting of the fewest commands) to move the robot to its destination (we do not care which direction it faces at the destination). You are more interested in knowing the number of different shortest programs that can move the robot to its destination. However, the number of shortest programs can be very large, so you are satisfied to compute the number as a remainder modulo 1,000,000.

Input

There will be several test cases in the input. Each test case will begin with a line with two integers

N M

Where **N** is the number of rows in the grid, and **M** is the number of columns in the grid (2 ≤ **N, M** ≤ 100). The next **N** lines of input will have **M** characters each. The characters will be one of the following:

`'.'` Indicating a navigable grid point.

`'*'` Indicating a crater (i.e. a non-navigable grid point).

`'X'` Indicating the target grid point. There will be exactly one **`'X'`**.

`'N'`, **`'E'`**, **`'S'`**, or **`'W'`** Indicating the starting point and initial heading of the robot. There will be exactly one of these. Note that the directions mirror compass directions on a map: **`N`** is North (toward the top of the grid), **`E`** is East (toward the right of the grid), **`S`** is South (toward the bottom of the grid) and **`W`** is West (toward the left of the grid).

There will be no spaces and no other characters in the description of the map. The input will end with a line with two 0s.

Output

For each test case, output two integers on a single line, with a single space between them. The first is the length of a shortest possible program to navigate the robot from its starting point to the target, and the second is the number of different programs of that length which will get the robot to the target (modulo 1,000,000). If there is no path from the robot to the target, output two zeros separated by a single space. Output no extra spaces, and do not separate answers with blank lines.

Sample Input

```
5 6
*....X
.....*
.....*
.....*
N....*
6 5
....X
.****
.****
.****
.****
N****
3 3
.E.
***
.X.
0 0
```

Sample Output

```
6 4
3 1
0 0
```

F: Folding Game

Alice and Bob are playing a game. Alice places a rectangular piece of paper in front of Bob with width **W** and height **H**. Then she proceeds to fold the paper **N** times. Each fold is either horizontal or vertical. Folding the paper horizontally leaves another rectangle of the same width **W** and smaller height **h**. Similarly, a vertical fold leaves a rectangle with same height **H** and smaller width **w**.

In the end, Alice puts her finger on some point on the resulting rectangle and asks 'Bob, how many layers of paper are directly beneath my finger?'.

Input

There will be several test cases in the input. Each test case will begin with a line with three integers:

W H N

Where **W** and **H** ($0 < $ **W**,**H** $\leq$ 1,000,000) are the width and height of the paper, and **N** ($0 \leq$ **N** ≤ 20) is the number of folds. **W** and **H** are guaranteed to be even. On each of the subsequent **N** lines there will be a letter and a number, separated by a single space:

D K

The letter **D** is one of { **`'T'`**, **`'B'`**, **`'L'`**, **`'R'`** } indicating whether the fold is from the **`T`**op, **`B`**ottom, **`L`**eft or **`R`**ight. It will always be capital. The number **K** indicates where Alice makes the fold, measured from the given edge. For example, if **D** is `'T'`, then Alice starts with the paper lying flat, lifts the TOP edge and folds it downward. **K** is guaranteed to be on the paper, and it is guaranteed to be even.

On the final line of each case there will be two integers:

X Y

Which indicate the point where Alice puts her finger. This is measured from the bottom left corner, with **X** being the distance towards the right, and **Y** being the distance towards the top. The point (**X**,**Y**) is guaranteed to be on the fully folded paper. Both **X** and **Y** are also guaranteed to be odd. Since **W**, **H** and **K** are all even, this assures that the point (**X**,**Y**) will not be over any edge or fold.

The input ends with a line with three 0s.

Output

For each case output a single integer on its own line, indicating the number of layers of paper at the given point (**X**,**Y**). Output no extra spaces, and do not separate answers with blank lines.

Sample Input

```
10 10 1
B 4
5 1
10 10 1
B 4
7 5
10 10 1
T 6
3 1
10 10 1
T 6
9 3
14 10 2
L 4
R 4
3 3
0 0 0
```

Sample Output

```
2
1
1
2
3
```

G: Burnout

Whenever you buy a light bulb, the package will state an amount of time that the bulb should last. But, that's assuming the bulb is on continuously. Most people turn their light bulbs on and off. How does this affect their lifetime?

A friend of yours has built a rig to test this. Given a simple pattern, his rig will turn the bulb on and off, and a light sensor will tell when the bulb burns out. The pattern has a very simple syntax. It consists of a list of elements, separated by spaces. Each element is either a number **M** (1 ≤ **M** ≤ 1,000,000), indicating a number of milliseconds, or a repeating list. A repeating list consists of one or more elements, surrounded by parentheses, and followed by a `'*'` and then an integer **K** (1 ≤ **K** ≤ 100). This integer **K** indicates how many times the list should be repeated. Your friend's machine always starts with the bulb on, and then goes through the pattern. For every integer, it waits that number of milliseconds, and then switches the bulb. If the bulb is on, it's turned off, and if it's off, it gets turned back on. If the machine reaches the end of the pattern, it will start again from the beginning.

For example, with this pattern:

```
1 3 5
```

The machine will start by turning the bulb on.
- It will wait 1 millisecond and turn the bulb off.
- It will then wait 3 milliseconds and turn the bulb on.
- It will then wait 5 milliseconds and turn the bulb off.
- It will then wait 1 millisecond and turn the bulb on.
- It will then wait 3 milliseconds and turn the bulb off.

It will continue like this until the bulb burns out.

Here's an example of a pattern with a repeating section:

```
1 (3 5)*2 7
```

The machine will start by turning the bulb on, and will change the bulb's state after 1 millisecond, then 3, then 5, then 3, then 5, then 7, then 1.....

Note that repeating sections can be nested inside of repeating sections.

Your friend is not good with programming, though, so he's asked you to help. He can easily measure how long it takes for the bulb to actually burn out - but, how long SHOULD it have taken? Assume that turning the bulb on and off does NOT affect its life. Also assume that changing the bulb's state takes no time.

Given a life **N** in milliseconds, and a pattern of turning the bulb on and off, how many actual milliseconds would elapse before the bulb is on for exactly **N** milliseconds?

Input

There will be several test cases in the input. Each test case will consist of two lines. The first line will contain an integer **N** (1 ≤ **N** ≤ 1,000,000,000) which is the expected life of the bulb in milliseconds.

The second line will contain a string, which is the pattern. The pattern will not be longer than 500 characters. The pattern consists of a list of elements, separated by single spaces. Each element is either a number **M** (1 ≤ **M** ≤ 1,000,000), indicating a number of milliseconds, or a repeating list. A repeating list consists of one or more elements, surrounded by parentheses, and followed by a `'*'` and then an integer **K** (1 ≤ **K** ≤ 100). There will be single spaces separating elements of a list, but nowhere else. In particular, note that there will not be any spaces surrounding the `'*'` at the end of a repeating list, nor immediately following an opening parenthesis, nor immediately before a closing parenthesis. The total amount of time represented by a pattern, including all repetition, will be no greater than 1,000,000,000.

The input will terminate with a line with a single 0.

Output

For each test case, output a single integer on its own line, indicating the number of milliseconds of total elapsed time until the bulb has been lit for **N** milliseconds. Output no extra spaces, and do not separate answers with blank lines.

Sample Input

```
100
20 30 40
1000
1 3 5
1000
1 (3 (5)*3 2)*2 7
0
```

Sample Output

```
190
1999
2279
```

H: Family Fortune

While studying the history of wealthy families, researchers want to know just how much of a fortune each family has amassed. There are various 'net worth' figures for each individual throughout history, but totaling them is complicated by double counting, caused by inheritance. One way to estimate the wealth of a family is to sum up the net worth of a set of **K** people such that no one in the set is an ancestor or a descendant of anyone else in the set. The wealth of the family is the maximum sum achievable over all such sets of **K** people. Since historical records contain only the net worth of male family members, the family tree is a simple tree in which every male has exactly one father and a non-negative number of sons. Also, there is exactly one person who is an ancestor of all other family members.

Given information about a family tree, what is the wealth of the family, by this measure?

Input

There will be several test cases in the input. Each test case will begin with two integers:

N K

Where **N** (1 ≤ **N** ≤ 100,000) is the total number of family members in the records, and **K** (1 ≤ **K** ≤ 1,000) is the size of the desired set of people.

Each of the next **N** lines will hold two integers:

P W

Where **P** (0 ≤ **P** ≤ **N**) indicates the parent of that family member. The family members are numbered from 1 to **N**, with the parent and fortune of family member *i* appearing on the i^{th} line. There will be a single root, with **P**=0. The tree will not have a depth greater than 1,000, and, of course, it won't have any cycles. **W** (1 ≤ **W** ≤ 1,000) is the wealth (in millions) of that family member.

The input will end with a line with two 0s.

Output

For each test case, output a single integer on its own line, which is the maximum sum (in millions) of the fortunes of a set of **K** family members in the tree, where no member of the set is an ancestor or descendant of any other member of the set. If you cannot find **K** such nodes, then output 0. Output no extra spaces, and do not separate answers with blank lines.

Sample Input

```
11 5
0 1
1 1
1 1
2 1
2 1
3 1
3 1
3 1
5 1
7 1
7 1
11 5
11 3
1 1
4 1
1 2
10 2
10 2
6 2
6 1
10 2
11 3
0 4
7 3
0 18
1 20
1 15
2 12
2 6
3 8
3 8
0 0
```

Sample Output

```
5
10
36
```

I: Moving Points

Consider a number of Target points in a plane. Each Target point moves in a straight line at a constant speed, and do not change direction. Now, consider a Chaser point that starts at the origin, and moves at a speed faster than any of the Target points. The Chaser point moves at a constant speed, but it is capable of changing direction at will. It will 'Catch' a Target point, and then move from there to catch another Target point, and so on.

Given the parameters of the Chaser point and the Target points, what is the least amount of time it takes the Chaser point to catch all of the Target points? 'Catch' simply means that the Catcher and the Target occupy the same point in the plane at the same time. This can be instantaneous; there's no need for the Catcher to stay with the Target for any non-zero length of time.

Input

There will be several test cases in the input. Each test case will begin with two integers

N C

Where **N** (1 ≤ **N** ≤ 15) is the number of Target points, and **C** (0 < **C** ≤ 1,000) is the speed of the Chaser point. Each of the next **N** lines will have four integers, describing a Target point:

X Y D S

Where (**X**,**Y**) is the location in the plane (-1000 ≤ **X**,**Y** ≤ 1,000) of that Target point at time 0, **D** (0 ≤ **D** < 360) is the direction of movement in Degrees (0 degrees is the positive X axis, 90 degrees is the positive Y axis), and **S** (0 ≤ **S** < **C**) is the speed of that Target point. It is assumed that all Target points start moving immediately at time 0.

The input will end with a line with two 0s.

Output

For each test case, output a single real number on its own line, representing the least amount of time needed for the Chaser point to catch all of the Target points. Print this number to exactly 2 decimal places, rounded. Output no extra spaces, and do not separate answers with blank lines.

Sample Input

```
2 25
19 19 32 10
6 45 133 19
5 10
10 20 45 3
30 10 135 4
100 100 219 5
10 100 301 4
30 30 5 3
0 0
```

Sample Output

```
12.62
12.54
```

J: Vampire Numbers

The number `1827` is an interesting number, because `1827=21*87`, and all of the same digits appear on both sides of the '='. The number `136948` has the same property: `136948=146*938`.

Such numbers are called *Vampire Numbers*. More precisely, a number **v** is a Vampire Number if it has a pair of factors, **a** and **b**, where **a*****b**=**v**, and together, **a** and **b** have exactly the same digits, in exactly the same quantities, as **v**. None of the numbers **v**, **a** or **b** can have leading zeros. The mathematical definition says that **v** should have an even number of digits and that **a** and **b** should have the same number of digits, but for the purposes of this problem, we'll relax that requirement, and allow **a** and **b** to have differing numbers of digits, and **v** to have any number of digits. Here are some more examples:

```
    126    = 6  * 21
    10251  = 51 * 201
    702189 = 9  * 78021
    29632  = 32 * 926
```

Given a number **X**, find the smallest Vampire Number which is greater than or equal to **X**.

Input

There will be several test cases in the input. Each test case will consist of a single line containing a single integer **X** (10 ≤ **X** ≤ 1,000,000). The input will end with a line with a single 0.

Output

For each test case, output a single integer on its own line, which is the smallest Vampire Number which is greater than or equal to **X**. Output no extra spaces, and do not separate answers with blank lines.

Sample Input

```
10
126
127
5000
0
```

Sample Output

```
126
126
153
6880
```