Odprto ekipno prvenstvo UNIVERZE V LJUBLJANI v programiranju

2003

4. krog

Glavni pokrovitelji

UNIVERZA V LJUBLJANI

Fakulteta za matematiko in fiziko

Pokrovitelji

Šolski servis
Janez Vrankar

NAVODILA

Prijava: Prijavite se lahko le na en računalnik. Če bodo med tekmovanjem z njim slučajno težave, javite vodstvu tekmovanja. V primeru, da boste hkrati prijavljeni na dveh računalnikih, bo ekipa diskvalificirana.

Uporabniško ime je sestavljeno iz besede acm in številke računalnika.

Primer: uporabniško ime na računalniku št. 1 je: acm01

uporabniško ime na računalniku št. 12 je: acm12

Geslo je sporočeno pred začetkom tekmovanja.

Vaše domače področje na disku je `/home/acmXX` kjer je XX številka računalnika

Priprava delovnega okolja

Po uspešni prijavi morate odpreti dva terminala in program Netscape.

Terminal št.1: v njem boste pisali programe, prevajali programe in pošiljali programe sodniku.

Terminal št.2: v njem boste dobivali obvestila o pravilnosti rešitev in pošiljali morebitna vprašanja sodniku.

Program Netscape: bo omogočal pregled trenutne uspešnosti ekip

Pisanje programov:

Na voljo so urejevalniki besedil: *vi, kwrite, mcedit, emacs*, *gedit in kedit*

Ime programov je točno določeno pri opisu nalog. Pazite na `return 0,` ki mora zaključiti vsak uspešno izveden C program.

Prevajanje programov:

Za prevajanje programov morate uporabljati naslednje prevajalnike:

```
C       (*.c)        gcc    primer:   gcc -o program program.c
C++     (*.cpp)      g++    primer:   g++ -o program program.cpp
Pascal  (*.pas)      gpc    primer:   gpc -o program program.pas
Java    (*.java)     gcj    primer:   gcj -o program --main=program program.pas
```

Pošiljanje programov:

Za pošiljanje programov sodniku uporabljate ukaz `submit`.

Primer: `submit program.c`

Sporočila o napakah – PE in WA

Zaradi različnih razlogov je zelo težko razlikovati med `Presentation Error` in `Wrong Answer`. Ker preverjanje poteka več ali manj avtomatsko, program pogosto ne zna razlikovati med tem, ali ste le pozabili piko v odgovoru, ali pa ste napačno zračunali. Zato tudi `Wrong Answer` včasih lahko pomeni le manjkajoč znak v odgovoru. Praviloma naj bi sporočilo `PE` dobili le v primeru napačno postavljenih presledkov ali ločil in praznih vrstic, ni pa to nujno!

Komunikacija sodniki – ekipa:

V terminalu 2 izvedete ukaz: `ssh svarun.fmf.uni-lj.si`

Za prijavo na računalnik svarun uporabite isto uporabniško ime in geslo, kot ste ga uporabili za prijavo na vaš računalnik. V tem terminalskem oknu se bodo izpisovala sodnikova sporočila vam. Če želite poslati sporočilo sodniku, morate v tem terminalu napisati ukaz:

```
write sodnik
```

in potem sporočilo, ki ga zaključite s kombinacijo tipk CTRL + D na začetku prazne vrstice. Prosimo, da **ne** pošiljate sporočil tipa `... Sedaj smo poslali program ...` in `... Naš program zagotovo dela, pa dobimo WA ...` Ekipa, ki bo pošiljala tovrstna sporočila, je lahko tudi diskvalificirana.

Trenutni rezultati:

Program Netscape: bo omogočal pregled trenutne uspešnosti ekip

Po zagonu programa greste na naslov:

http://svarun.fmf.uni-lj.si/index.html

Kadar želite videti trenutno stanje, morate narediti »reload«.

Pritožbe:

So možne le takoj po zaključku tekmovanja. Kasneje se rezultati lahko spremenijo le v primeru drastične sodniške napake.

Preklapljanje med angleško in slovensko tipkovnico:

Z ukazom v terminalu:

sexkbmap si

setxkbmap us

Nastavitve veljajo za novo odprte programe.

Sorting It All Out

Program sorting.c, sorting.cpp, sorting.java, sorting.pas

An ascending sorted sequence of distinct values is one in which some form of a less-than operator is used to order the elements from smallest to largest. For example, the sorted sequence *A*, *B*, *C*, *D* implies that $A < B$, $B < C$ and $C < D$. In this problem, we will give you a set of relations of the form $A < B$ and ask you to determine whether a sorted order has been specified or not.

Input

Input consists of multiple problem instances. Each instance starts with a line containing two positive integers *n* and *m*. The first value indicates the number of objects to sort, where $2 \le n \le 26$. The objects to be sorted will be the first *n* characters of the uppercase alphabet. The second value m indicates the number of relations of the form $A < B$ which will be given in this problem instance. Next will be *m* lines, each containing one such relation consisting of three characters: an uppercase letter, the character "<" and a second uppercase letter. No letter will be outside the range of the first n letters of the alphabet. Values of $n = m = 0$ indicate end of input.

Output

For each problem instance, output consists of one line. This line should be one of the following three:

```
Sorted sequence determined after xxx relations: yyy...y.
```

```
Sorted sequence cannot be determined.
```

```
Inconsistency found after xxx relations.
```

where xxx is the number of relations processed at the time either a sorted sequence is determined or an inconsistency is found, whichever comes first, and yyy...y is the sorted, ascending sequence.

Sample Input

```
4 6
A<B
A<C
B<C
C<D
B<D
A<B
3 2
A<B
B<A
26 1
A<Z
0 0
```

Sample Output

```
Sorted sequence determined after 4 relations: ABCD.
```

```
Inconsistency found after 2 relations.
Sorted sequence cannot be determined.
```

Stacking Cubes [KCi naloga]

Program cubes.c, cubes.cpp, cubes.java, cubes.pas

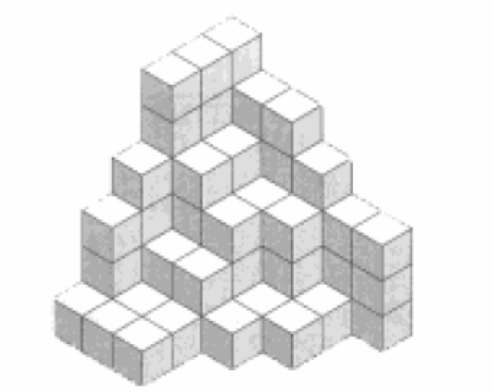
Consider the following pattern of positive integers:

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

Note that each row is left-justified and no longer than its preceding row. Also, the entries in each row, when read left to right, are non-increasing and the entries in each column, when read top to bottom are non-increasing. We will call such a pattern a stacking pattern (SP) because such a pattern can represent a way of stacking cubes in a corner in the following way: if you consider placing the topmost row and leftmost column against walls, then the SP gives a bird's-eye view of how many cubes are stacked vertically. The SP above represents the following corner stacking:

We will call the wall against the topmost row the right wall, and the wall against the leftmost column the left wall. Here is another SP and the corner stacking it represents:

```
6 5 5 4 3 3
6 4 3 3 1
6 4 3 1 1
4 2 2 1
3 1 1
1 1 1
```



Note that if you rotate a corner stacking so the left wall becomes the floor and the floor becomes the right wall, you still have a corner stacking. (We will call this a left rotation.) Likewise, you would still have a corner stacking if you rotate so the right wall becomes the floor and the floor becomes the left wall. (We will call this a right rotation.) So the SP of the left and right rotations of the first SP given above are

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

```
2 1 1                  2 1 1
2 1                    1
```

You should check that both the left and right rotations of the second example SP are identical to the original SP.

Input

This problem will consist of multiple problem instances. Each problem instance will consist of a positive integer $n \leq 11$ indicating the number of rows in the SP that follows. ($n = 0$ indicates the end of input.) The rows of the SP will follow, one per line with entries separated by single spaces, delimited by a trailing 0. (The trailing 0 is, of course, not part of the input data proper and you may assume that each row given has at least one cube.) Each entry in the pattern proper will be a positive integer less than or equal to 20 and there will be no more than 20 entries in any row.

Output

For each input SP you should produce two stacking patterns corresponding to the left rotation and the right rotation (in that order). Rows of the SP should be left-justified with entries separated by a single space. One blank line should separate the left and right rotations of the given SP and two blank lines should separate output for different problem instances.

Sample Input

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

Sample Output

```
3 2 1
2 1 1
2 1
3 3 2
2 1 1
1
6 5 5 4 3 3
6 4 3 3 1
6 4 3 1 1
4 2 2 1
3 1 1
1 1 1
6 5 5 4 3 3
6 4 3 3 1
6 4 3 1 1
4 2 2 1
3 1 1
1 1 1
```

Quadtrees

Program quadtrees.c, quadtrees.cpp, quadtrees.java, quadtrees.pas

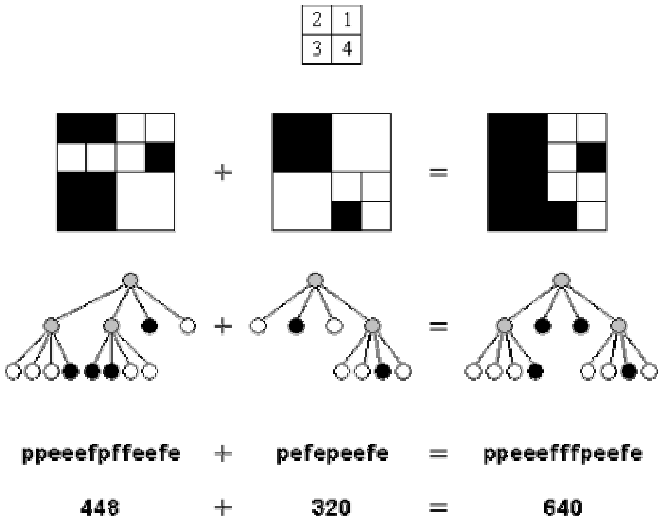
A quadtree is a representation format used to encode images. The fundamental idea behind the quadtree is that any image can be split into four quadrants. Each quadrant may again be split in four sub quadrants, etc. In the quadtree, the image is represented by a parent node, while the four quadrants are represented by four child nodes, in a predetermined order.

Of course, if the whole image is a single color, it can be represented by a quadtree consisting of a single node. In general, a quadrant needs only to be subdivided if it consists of pixels of different colors. As a result, the quadtree need not be of uniform depth.

A modern computer artist works with black-and-white images of 32 × 32 units, for a total of 1024 pixels per image. One of the operations he performs is adding two images together, to form a new image. In the resulting image a pixel is black if it was black in at least one of the component images, otherwise it is white.

This particular artist believes in what he calls the *preferred fullness*: for an image to be interesting (i.e. to sell for big bucks) the most important property is the number of filled (black) pixels in the image. So, before adding two images together, he would like to know how many pixels will be black in the resulting image. Your job is to write a program that, given the quadtree representation of two images, calculates the number of pixels that are black in the image, which is the result of adding the two images together.

In the figure, the first example is shown (from top to bottom) as image, quadtree, pre-order string (defined below) and number of pixels. The quadrant numbering is shown at the top of the figure.



Input

The first line of input specifies the number of test cases (*N*) your program has to process.

The input for each test case is two strings, each string on its own line. The string is the pre-order representation of a quadtree, in which the letter 'p' indicates a parent node, the letter 'f' (full) a black quadrant and the letter 'e' (empty) a white quadrant. It is guaranteed that each string represents a valid quadtree, while the depth of the tree is not more than 5 (because each pixel has only one color).

Output

For each test case, print on one line the text '`There are X black pixels.`', where *X* is the number of black pixels in the resulting image.

Sample Input

```
3
ppeeefpffeefe
pefepeefe
peeef
peefe
peeef
peepefefe
```

Sample Output

```
There are 640 black pixels.
There are 512 black pixels.
There are 384 black pixels.
```

Picnic Planning

Program picnic.c, picnic.cpp, picnic.java, picnic.pas

The Contortion Brothers are a famous set of circus clowns, known worldwide for their incredible ability to cram an unlimited number of themselves into even the smallest vehicle. During the off-season, the brothers like to get together for an Annual Contortionists Meeting at a local park. However, the brothers are not only tight with regard to cramped quarters, but with money as well, so they try to find the way to get everyone to the party which minimizes the number of miles put on everyone's cars (thus saving gas, wear and tear, etc.). To this end they are willing to cram themselves into as few cars as necessary to minimize the total number of miles put on all their cars together. This often results in many brothers driving to one brother's house, leaving all but one car there and piling into the remaining one. There is a constraint at the park, however: the parking lot at the picnic site can only hold a limited number of cars, so that must be factored into the overall miserly calculation. Also, due to an entrance fee to the park, once any brother's car arrives at the park it is there to stay; he will not drop off his passengers and then leave to pick up other brothers. Now for your average circus clan, solving this problem is a challenge, so it is left to you to write a program to solve their mileage minimization problem.

Input

Input consists of multiple problem instances. The first line of every instance contains a single integer *n* indicating the number of highway connections between brothers or between brothers and the park. On the second line there is an integer *s* which specifies the number of cars which can fit in the parking lot of the picnic site. The last *n* lines contain one connection per line, of the form *name1 name2 dist*, where *name1* and *name2* are either the names of two brothers or the word `Park` and a brother's name (in either order), and *dist* is the integer distance between them. The roads are all 2-way roads, and *dist* is always positive. The maximum number of brothers is 20 and the maximum length of any name is 10 characters. You may assume that there is a path from every brother's house to the park and that a solution exists for each problem instance. The input ends with $n = 0$.

Output

Output should consist of one line per case. Lines have the form

```
Total miles driven: xxx
```

where `xxx` is the total number of miles driven by all the brothers' cars.

Sample Input

```
10
3
Alphonzo Bernardo 32
Alphonzo Park 57
Alphonzo Eduardo 43
Bernardo Park 19
Bernardo Clemenzi 82
Clemenzi Park 65
Clemenzi Herb 90
Clemenzi Eduardo 109
Park Herb 24
```

```
Herb Eduardo 79
10
1
Alphonzo Bernardo 32
Alphonzo Park 57
Alphonzo Eduardo 43
Bernardo Park 19
Bernardo Clemenzi 82
Clemenzi Park 65
Clemenzi Herb 90
Clemenzi Eduardo 109
Park Herb 24
Herb Eduardo 79
0
```

Sample Output

```
Total miles driven: 183
Total miles driven: 255
```

Book Pages

Program pages.c, pages.cpp, pages.java, pages.pas

For the purposes of this problem, we will assume that every page in an Acmonian book is numbered sequentially, and that the first page is numbered 1.

How many digits would you need to use to number the pages of a 10 page book? Pages 1 to 9 would require 1 digit each (total 9), and page 10 would require 2 digits. This makes 11 digits. Similarly, a book of 34 pages would require 59 digits.

Can we work backwards? If you are told that a book requires 13 digits to number its pages, can you work out how many pages the book has? I hope so, because that is all you have to do for this problem. Each line in the input file represents the number of digits used in numbering a book. Your answer will be the number of pages the book has. If the number supplied cannot possibly be valid, your answer should be "Impossible!" Beware that Acmonian books can be quite large, and the number of digits required for a given Acmonian book can reach 2,000,000,000.

Input

Each line in the input file contains a single integer, between 1 and 2,000,000,000, representing a number of digits used in numbering the pages of a book. A single 0 on a line indicates the end of input.

Output

Output for each input number must be on a single line. If the input value is valid, output the number of pages in the book. Otherwise, output "Impossible!"

Sample Input:

```
11
13
59
60
1999999998
0
```

Sample Output:

```
10
11
34
Impossible!
234567900
```