

Problem 1: Let's create a committee

The Dean of ACM University's School of Programming Contests is responsible for assigning faculty to university committees. He has found that if he assigns faculty from different departments to a committee, they wind up discussing the socioeconomic importance of the number of bits in a byte and similar matters of earth-shaking importance but they never get any work done. On the other hand, if the faculty are all from the same department, they wind up discussing department business instead of university business.

So the Dean wants to know if there is a way of assigning faculty to committees so that a majority (more than 50%) of faculty on each committee are from the same department (not necessarily the same department for different committees), but not all faculty on a committee are from the same department. The Dean thinks faculty ought to serve on at least two committees, but faculty refuse to serve on more than three committees.

The first line of input to your program contains the names of the committees, separated by white space, for example:

```
Curriculum Eisenhart-Awards
```

Each of the remaining lines contains the (unique) name of a faculty member and the name of the department which the faculty member belongs to, separated by white space, for example:

```
Heliotis Software-Engineering  
Tymann Information-Technology  
Vallino Software-Engineering
```

For each committee your program must print out a line with the committee name, followed by one line with name and department for each committee member, such that all the conditions above are met. If there is more than one way to assign faculty, any single way may be printed. If there is no possible way to assign faculty, print `No solution`.

Sample Input

see ~/1.1

```
Curriculum Eisenhart-Awards  
Heliotis Software-Engineering  
Tymann Information-Technology  
Vallino Software-Engineering
```

Sample Output

```
Curriculum  
Heliotis Software-Engineering  
Tymann Information-Technology  
Vallino Software-Engineering
```

```
Eisenhart-Awards  
Heliotis Software-Engineering  
Tymann Information-Technology  
Vallino Software-Engineering
```

Sample Input

see ~/1.2

Library Budget
Etlinger Computer-Science
Strout Computer-Science

Sample Output

No solution

Problem 2: iFlop

A major manufacturer has contracted you to implement a simulator for their new preemptive multi-tasking system iFlop. A run of iFlop is described by a configuration file which contains lines separated by newline; each line contains positive integer numbers and case-sensitive words separated by white space.

A configuration file starts with the following four lines, in order:

```
slice number1
ready name2 name3 ...
wait name4 name5 ...
var name6 number7
```

where *number*₁ is the (positive) number of instructions started within a time slice; *name*₂ etc. specify the user programs which are to be run, each as a separate process; *name*₄ etc. are the names of waiting lists for processes; and *name*₆ is the name and *number*₇ is the initial value of a global integer variable. There is at least one process and one waiting list and `var` lines may be repeated. All waiting list and variable names are different.

Following the variables, the configuration file can specify some subprograms, for example:

```
sub Pfull
0 if f 1 3
1   dec f
2   if TRUE 4 4
3   move ready full
4 end
```

where `sub` starts a subprogram and defines its name; the program ends with `end`. All subprogram names are different.

Within each subprogram, lines are numbered consecutively from 0.

`if` specifies a condition and two existing line numbers. Execution continues with one of these two lines depending on whether the condition is true or false. A condition is either the name of a variable and is true if the variable is not zero, or it is the name of a list (including `ready`) and is true if the list is not empty.

`dec` decrements a variable by one; there is also an `inc` statement to increment a variable by one.

`move` specifies two lists (either one can be `ready`, otherwise names defined with `wait` are used) and takes the first process from the first list (which will be not empty) and appends it as the last process on the second list.

Finally, the configuration file specifies each user program, for example:

```
prog consumer
0 call Pfull
1 print n
2 call Vempty
3 end
```

where `prog` starts a user program and specifies one of the names on the `ready` line; the program

ends with `end`. All program names are different.

Within each program, lines are numbered consecutively from 0.

`print` outputs the value of a variable to standard output.

`call` specifies the name of an existing subprogram to invoke. `call` cannot be used in a subprogram.

For the purposes of `slice`, each line (even `call`) counts as one unit. The program above requires four time units to complete because `call` counts as one unit irrespective of how many statements the subprogram contains.

Your simulator reads a configuration file from standard input and executes the programs for the processes specified with `ready`, one at a time beginning with the first process on the `ready` list.

If a process executes the number of lines specified with `slice` without executing `call` or `end` it is moved from the beginning to the end of the `ready` list and execution continues with the process which is then first on the `ready` list.

After a process executes a `call`, execution continues with the process which is then first on the `ready` list.

If a process reaches the matching `end` in its program the process is removed from the system.

Your simulator ends once the `ready` list is empty. At this point, for each non-empty list you must print one line with the name of the list and the name of the program of each process on the list.

Sample Input

see ~/2.1

```
slice 10
ready consumer producer
wait full empty
var n 0
var f 0
var e 1
var TRUE 1
sub Pfull
0 if f 1 3
1 dec f
2 if TRUE 4 4
3 move ready full
4 end
sub Vfull
0 if full 1 3
1 move full ready
2 if TRUE 4 4
3 inc f
4 end
sub Pempty
0 if e 1 3
1 dec e
2 if TRUE 4 4
3 move ready empty
4 end
sub Vempty
```

```
0 if empty 1 3
1   move empty ready
2   if TRUE 4 4
3   inc e
4 end
prog consumer
0 call Pfull
1 print n
2 call Vempty
3 end
prog producer
0 call Pempty
1 inc n
2 call Vfull
3 end
```

Sample Output

1

Sample Input

see ~/2.2

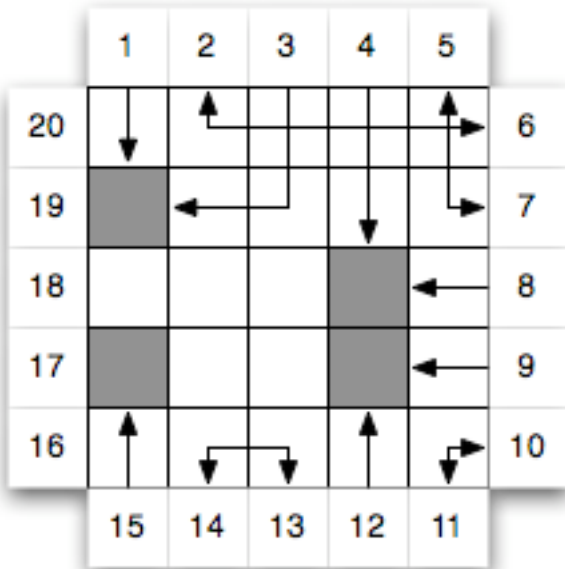
```
slice 1
ready bad good bad good
wait blocked
sub block
0 move ready blocked
1 end
prog bad
0 call block
1 end
prog good
0 end
```

Sample Output

blocked bad bad

Problem 3: BlackBox

In Eric Solomon's game BlackBox a few marbles are hidden, each in one cell on a square grid. Your program should locate the marbles by shooting rays into the box.



The first line of input contains two positive numbers, the side length of the square and the number of hidden marbles (separated by white space and not necessarily equal); in this case:

5 4

The edges of the square are numbered, starting with 1, as shown above. To compute deflection, as discussed below, the edges are treated as if they were squares adjacent to the board. Board positions are identified by the ordered sequence of smallest edge numbers; for example, the top left gray box is at position 1, 7.

The remaining input lines describe what happens to search rays which enter the square from each side, proceed horizontally and/or vertically, and are either deflected or absorbed by the marbles.

A ray is absorbed if it hits a marble directly. A line describing an absorbed ray contains a single positive number identifying the edge position where the ray enters the square; in this case:

1
3
4
8
9
12
15
17
19

A ray is deflected if it hits one of the diagonal neighbors of a marble: it is turned 90 degrees away from the marble and from its current direction. A ray can be deflected more than once. A line describing a deflected ray contains two positive numbers identifying the edge positions where the ray enters and leaves the square (separated by white space); in this case:

```
2 6
5 7
10 11
13 14
18 18
```

Note that input and output edge of a ray can always be interchanged.

The last line is a special case: the ray starts at 18 and this edge is considered the diagonal neighbor of the marbles at 1,9 and 1,7. The ray starts out going to the right. The first neighbor turns the ray up, the second neighbor turns the ray to the left. In summary, the ray is deflected twice and returns to its starting point 18.

One would expect that the list of absorbed and deflected rays contains each edge at least once. However, this example shows that that need not be the case: A ray starting at 16 is turned from going right to going down and never even reaches the board. Similarly, a ray starting at 20 is immediately turned up and misses the board, too.

Your program should output one line with cell coordinates for each hidden marble. The coordinates should be the *smaller* edge numbers identifying column and row of the marble.

Sample Input

see ~/3.1

```
5 5
1
2 6
3
4
5 7
8
9
10 11
12
13 14
15
17
18 18
19
```

Sample Output

```
1 7
4 8
1 9
4 9
```

Problem 4: Bioinformatics Gone Wild

The DNA in your cells contains the instructions that specify the proteins that are synthesized by your body. DNA consists of a sequence of 4 nucleotides which can be represented by the letters A, T, G, and C. For example the string GCTGCTCGCGATAATAATGAA would represent a short DNA sequence consisting of 21 nucleotides. A protein consists of a sequence of amino acids, and like nucleotides, each amino acid is represented by a letter. There are 20 amino acids and they are represented by the characters: G, A, V, L, I, M, F, W, P, S, T, C, Y, N, Q, D, E, K, R, and H. The string AARDNNE would represent a protein consisting of 7 amino acids.

Genes, which can be viewed as substrings of the DNA sequence, direct the synthesis of proteins. The nucleotides in a gene are read in sequential order in groups of three. Each group of three nucleotides encodes a particular amino acid. A genetic code indicates what three nucleotides encode each amino acid. A subset of a genetic code is given below:

A	GCT GCG GCC GCA
R	CGT CGG CGC CGA AGG AGA
D	GAT GAC
N	AAT AAC
C	TGC TGT
E	GAG GAA
W	TTT

Note that there can be multiple nucleotide triplets which encode a single amino acid. For example, in the genetic code subset above, only the amino acid W is encoded by a single triplet. Using this genetic code subset, the DNA sequence GCTGCTCGCGATAATAATGAA encodes the protein AARDNNE.

Unfortunately molecular biology is nowhere near as simple as we have just described. As it turns out the DNA in your cells contains over 3 billion nucleotides, and only a fraction of your DNA actually contains genes that encode proteins. This means that genes are often embedded in long sequences of nucleotides (which need not start with triplets). So for example, the DNA sequence

AAATTGCAGCTGCTCGCGATAATAATGAATTTTCGCGCGCGCGCGC

contains the gene that encodes the amino acid sequence AARDNNE.

To make matters even worse, DNA is actually double stranded, which means when you are working with a DNA sequence the gene might be coded in the forward direction (as in the example above) or in the reverse direction. Again for example, the DNA sequence

CGCGCGCGCGCGCTTTAAGTAATAATAGCGCTCGTCGACGTTAA

also encodes the amino acid sequence AARDNNE. This time the amino acid sequence is encoded by the reverse strand.

Given a strand of DNA, a protein sequence, and a genetic code, you are to determine if the complete protein sequence is encoded by all or some part of the DNA sequence. The input to your program will consist of a line containing a string that gives the DNA strand, for example:

AAATTGCAGCTGCTCGCGATAATAATGAATTTTCGCGCGCGCGCGC

The second line will consist of a string of alphabetic characters that gives the protein sequence, for example:

AARDNNE

The remaining lines of input will give a subset of the genetic code. Each line will begin with a single character that specifies the amino acid being encoded, followed by the groups of three nucleotide sequences that encode the amino acid. All elements on the line will be separated by whitespace, for example:

A GCT GCG GCC GCA
R CGT CGG CGC CGA AGG AGA
D GAT GAC
N AAT AAC
C TGC TGT
E GAG GAA
W TTT

Your program will print the word *yes* if the given DNA sequence encodes the amino acid sequence and the word *no* if it does not.

Sample Input

see ~/4.1

AAATTGCAGCTGCTCGCGATAATAATGAATTTTCGCGCGCGCGCGC
AARDNNE
A GCT GCG GCC GCA
R CGT CGG CGC CGA AGG AGA
D GAT GAC
N AAT AAC
C TGC TGT
E GAG GAA
W TTT

Sample Output

yes

Sample Input

see ~/4.2

AAATTGCAGCTGCTCGCGAT
AARDNNE
A GCT GCG GCC GCA
R CGT CGG CGC CGA AGG AGA
D GAT GAC
N AAT AAC
E GAG GAA

Sample Output

no

Problem 5: Sokoban

Dezider works in a large factory as a box-mover. A box-mover's job is relatively simple: every day he is presented with a huge box which he can only push but not lift. His task is to push the box from its current position to a desired location. The factory floor is covered with square tiles and from a bird's perspective it would look like a grid. The box and Dezider each occupy exactly one square. As Dezider cannot lift the box, they cannot be in the same square at the same time. Apart from the box and Dezider, some squares contain unmovable obstacles, such as walls or very heavy furniture. Neither Dezider, nor a box can walk through or step on the obstacles. Dezider can move freely on the unblocked floor — his possible moves are “turn right”, “turn left”, and “make a step”. If there is an obstacle in front of him while attempting to make a step, he simply bounces off the obstacle and lands at the same spot where he started doing the step. If there is a box in front of him while making a step and there is no obstacle in front of the box, Dezider pushes the box (as if the box made a step forward in sync with Dezider, so he ends up facing the box again but both moved). Dezider cannot pull the box — it is too heavy and does not have any handles.

Your task is to help Dezider to push the box to its target location. Dezider does not mind walking around the floor but pushing the box is a real pain in the neck. So he asks you to minimize the number of steps when he pushes the box. And — Dezider is especially interested in finding out if it is possible to move the box to its target location at all! (Sometimes, after Dezider has pushed the box around for a few days, his boss wears a rather curious expression...)

The first line of the input file contains m and n , the number of columns and rows of the grid, separated by a space, for example:

```
15 5
```

The next n lines describe the grid. Each line contains a string of length m . Within the strings is a single `x` (the target location for the box), a single `B` (the box), a single `D` (Dezider). The other characters in the strings are either `o` (obstacle) or period (empty tile). The four edges of the grid are always filled with obstacles. For example:

```
ooooooooooooooooo
O...D.O.....O
O.....O.B....O
O.....X..O
ooooooooooooooooo
```

If it is not possible to get the box to its target location your program should output 0. Otherwise, it should output the minimum number of box-pushing steps Dezider needs to make in order to get the box to its target location.

Sample Input

see ~/5.1

```
15 5
ooooooooooooooooo
O...D.O.....O
O.....O.B....O
O.....X..O
ooooooooooooooooo
```

Sample Output

```
3
```

Problem 6: Allergic Robot

Kazimir is a trash-sorting robot. Everybody (humans, that is) think he is very useful: you come to him with a list of parts needed for your new invention and he then sits by a conveyor belt which transports trash and whenever he spots the next part on your list, he picks it up, cleans it to perfection, and attaches it to the already obtained parts. Of course, every object on the conveyor belt has a different level of dirtiness. Some things take only a minute to clean while others take an hour. The conveyor belt is behind a glass wall with a window at the very front, so Kazimir can see all the objects (and their level of dirtiness) but he can reach only the very first object.

Kazimir's problem is that he is allergic to dirt. He strongly suspects that his current program does not minimize his exposure to the dirt — often he finds himself cleaning an object for an hour whereas if he only waited a few minutes, he could have gotten the same object in an almost clean condition. So he asked you to reprogram him so that he spends the least possible time cleaning the objects on people's lists.

Kazimir has a list of m wanted parts which he has to get in the specified order.

The conveyor belt is n feet long and every object occupies one foot on the belt.

The speed of the belt is one foot per minute. If Kazimir spends x minutes cleaning an object, the belt shifts by x feet and the objects that fall off the end of the belt in the meantime are loaded onto trucks which drive away to some unknown landfill, never to be seen again.

Your task is to tell Kazimir the specific objects he should pick up and clean in order to minimize the total cleaning time. You can assume that Kazimir can do everything but cleaning in a flash (0 minutes).

The first line contains m and n , separated by white space, for example:

```
3 8
```

The next line contains m integers specifying the ordered list of the wanted objects (separated by white space), for example:

```
6 7 5
```

It is possible for an object to be repeated several times on the list. The list must be filled exactly in order as written.

The following n lines describe the conveyor belt. Each line contains two integers separated by white space; the first specifies the next object on the conveyor belt, the second specifies the number of minutes it takes to clean the object, for example:

```
5 2
6 20
5 3
6 1
7 2
6 15
3 2
5 20
```

If it is not possible to collect all objects on the wish list, your program should output -1. Otherwise you should output the minimal number of minutes needed to clean the objects on the list in the specified order, for example:

23

In the example above, Kazimir can wait 3 minutes, clean the second object 6 in one minute, clean the object 7 in 2 minutes, wait another minute, and then clean the last object 5 in 20 minutes for a total of 23 minutes cleaning time. He is not allowed to clean object 5 first.

Sample Input

see ~/6.1

3 8
6 7 5
5 2
6 20
5 3
6 1
7 2
6 15
3 2
5 20

Sample Output

23

Problem 7: Intergalactic Collegiate Programming Contest

The contest judges are frantic: the contest is about to start and the planet colonies have just decided to compete after all. The judges were planning to use the old scoreboard program from the intercontinental days but it cannot handle the different time systems used on the planets.

Clearly, you would like to come to their rescue and write a new scoring program. The input is a list of teams and their raw performance on the problems, the output is a sorted list of teams with summary information about their performance.

More precisely, the input starts with a line with the time stamp for the start of the contest. Time stamps use 2 digits each for the hour and for the minute, separated by a colon, on a 24 hour clock, for example:

```
10:00
```

The start line is followed by a sequence of raw team performances. Each performance starts with one line with the team name and the planet name where the team competes, both in lower-case letters and separated by white space, for example:

```
rit earth
```

This line is followed by one line for each problem (out of the eight) which the team attempted. Each problem line starts with the problem number. The number is followed by the time stamp for each attempt. If the team actually solved the problem, the last time stamp is followed by the word `solved`. You can assume that the contest ends before `24:00`. All items on each problem line are separated by white space, for example:

```
1 10:01 10:02 10:03 15:03
2 10:05 solved
```

Of course, the planet name determines what the time stamps actually mean. A day is defined as the time it takes for the sun to appear again in the same place in the sky. This is quite different for the different planets:

```
mercury      59 earth days
venus        117 earth days
earth         1 earth day
mars          1 earth day
jupiter      10/24 earth days
saturn        10/24 + 39/1440 earth days
uranus        17/24 earth days
neptune       16/24 earth days
```

For historic reasons (and this may or may not be rooted in reality) each planet divides its day into 24 planet hours with 60 planet minutes each. The contest starts on each planet when the planet clock shows the start time reported in the input data as described above. And — you guessed it — the time stamps reported for each problem for each team are read off the clock on the team's planet. For a fair comparison the cumulative times will have to be converted to a common time scale and rounded (up beginning with 0.5 and down otherwise) to full minutes.

The output from your program is a sorted list of performances, one performance per line. Information within a performance must be separated by blanks.

Each performance line starts with an alphabetized list of team names of the teams which achieved

the same result. The names are followed by the (single) number of problems solved and cumulative time needed by each of these teams. Performances with more problems must be output first; within the same number of problems solved performances with shorter cumulative time must be output first. In case of equal times the performance with the fewer attempts at the solved problems must be output first.

The judges sit on Jupiter — no way to stand up without an exo-skeleton there — and expect the output to use Jupiter's clock. Therefore, the cumulative time must be converted from each team's planet to Jupiter days, hours, and minutes for output, separated by colons.

Sample Input

see ~/7.1

```
10:00
rit earth
1 10:01 10:02 10:03 15:03
2 10:05 solved
mit venus
1 11:10 solved
2 14:59 solved
brown venus
1 10:15 10:29
2 12:20 solved
3 13:49 solved
ur earth
4 10:04 solved
sunny jupiter
2 10:04 10:12 solved
```

Sample Output

```
brown mit 2 71:22:55
ur 1 0:0:10
rit 1 0:0:12
sunny 1 0:0:12
```

Problem 8: Doggone Air

After running a successful bakery for years and avoiding a tax audit last year, Arnie has discovered that he has accumulated billions of dollars. In the interest of trying new things Arnie has decided to start up a new airline "Doggone Air" that caters exclusively to dogs. Arnie being the smart business dog has decided to hire greyhounds for sleek and speedy in-flight Beggin' Strip service, as well as seasoned pit bulls to maintain order, and limit his flights so that they all take off and land at locations in the same time zone and on the same day.

When selecting a flight dogs, unlike humans, do not care about the actual time a flight departs or arrives at its destination, they only care about the total elapsed time of a flight plan consisting of one or more flights. Second, dogs cannot travel for more than 3 hours without taking a break. So, for destinations that are further than 3 hours away, the flight plan must include a layover of at least 30 minutes, to give the passengers time to stretch their legs and find suitable trees. Finally, dogs insist on leaving and arriving on the same day.

You are to write a program that, given the flights offered by Doggone, along with a source and destination airport, will determine whether or not a flight plan exists that meets the selection criteria in the previous paragraph.

The input to your program will consist of several lines. The first line will contain a single positive integer n that gives the number of flights offered by Doggone Air. The next n lines of input will be flights, in the following format:

```
source destination dd:dd aa:aa
```

where *source* and *destination* are the source and destination airport, *dd:dd* is the time the flight departs (in 24-hour format) and *aa:aa* is the time the flight is scheduled to arrive at the destination. Note that these times may or may not correspond to what people would term real life.

The last line of input will be in the following format:

```
source destination
```

where *source* is the airport from which the passenger wishes to depart and *destination* is the airport at which the passenger wishes to arrive. All items are separated by white space.

Your program will either output *yes* if a valid flight path exists or *no* if a valid flight path cannot be found.

Sample Input

see ~/8.1

```
5
Rochester Toronto 07:45 10:46
Rochester Detroit 07:45 09:13
Detroit London 09:49 12:17
London Montreal 18:48 20:57
Montreal Toronto 22:45 23:15
Rochester Toronto
```

Sample Output

```
yes
```