

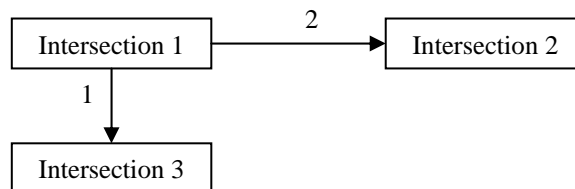
Problem 1: The Longest Walk

Jim Hikerlink enjoys a good walk around his hometown of Browserville. Browserville consists of several intersections connected by roads of varying length. Jim wants to take the longest walk possible but does not want to visit an intersection twice. He doesn't care what intersection he starts at or ends at, but the start and finish are visits to these intersections; therefore, Jim's path can't be a closed loop (because the starting intersection would be visited twice).

Intersections in Browserville are either not connected by a road, or are connected by a single road. There will never be more than one road connecting two intersections, and no road runs in a loop (i.e. from an intersection to itself). There are both one-way and two-way roads in Browserville. Jim, being a law-abiding citizen, obeys all one-way signs.

The input data to your program, that describes the roads in Browserville, begins with a single integer n on a line. This integer indicates how many intersections are in the town. The next n lines of input, each contain n non-negative integers each. The numbers on line i indicate the length of the roads from intersection i to each intersection (including itself). A length of 0 indicates no road (or a road going the wrong way). The lengths are listed in order by intersection (i.e., the length of the road to intersection 1 is listed first, followed by the length of the road to intersection 2 ... followed by the length of the road to intersection n).

For example, the town:



Would be represented as:

```
3
0 2 1
0 0 0
0 0 0
```

Your program will produce as output the length of the longest path, and one valid path of that length. The output that would be produced for the input data above would be:

```
The longest path has length 2
Such a path is 1 2
```

If there is no path at all, the output should be:

```
The longest path has length 0
```

Sample Input

```
3
0 2 1
0 0 0
0 0 0
```

Sample Output

```
The longest path has length 2
Such a path is 1 2
```

Problem Number 2: Sichuan MahJong

Sichuan MahJong is played on a rectangular grid where pairs of equal tiles are placed, at most one tile per cell. There can be several pairs of the same kind of tile. Each move is to remove one pair of equal tiles. The game is over if there are no more tiles on the grid or if there are no more possible moves.

A pair of equal tiles can be removed if they are connected by up to three straight lines (parallel to the grid); but you may not move across other tiles (i.e., the path between the pair of tiles must be empty). The path need not be completely in the grid.

The grid can be represented by a sequence of lines of equal length, which contain letters for tiles and blanks for empty cells. A grid consisting of 3 rows and 6 columns is shown below:

```
AA D T
E EXD
FX yFT
```

A move can be represented as a sequence of four positive integers: row and column of the first tile and row and column of the second tile of a pair; row 1 is at the top, and column 1 is at the left. Given the grid above, the move

1 1 1 2

would remove the two tiles labeled 'A' in the upper left hand area of the grid which are connected by a single horizontal line. The move

1 4 2 5

would remove the two tiles labeled 'D' in the upper right hand corner of the grid which are connected by a horizontal and then a vertical line. The move

3 1 3 5

would remove the two tiles labeled 'F' in the last row of the grid. Three lines connect the 'F' tiles; two extend vertically out of the bottom of the grid, and one runs horizontally along the bottom outside the grid. Finally, the move

2 4 3 2

is not valid. There are at least three paths connecting these tiles (two paths of length two, and one path of length three that goes outside the grid), but none of these paths are empty (one is blocked by 'E' and the other two by 'y').

These rules seem to overwhelm some players. Therefore, you are asked to write a program to help play the game. The first line of input to your program will contain a single positive integer that specifies the number of rows in the grid. Your program will then read a grid, represented as described above. The first line after the grid will contain a single nonnegative integer that gives the number of moves. The remaining lines contain the moves, as described above.

Your program will then execute the moves, in the order given. If your program decides that a move is not legal, it should print the following message, and terminate immediately without producing any additional output:

```
Move  $R_1$   $C_1$   $R_2$   $C_2$  is illegal
```

After executing all of the moves, your program will print out one line with the number of tiles remaining on the grid. If there are tiles left on the grid your program will print a list of possible single moves that can be made on the resulting grid, one per line. These moves are not cumulative (i.e., each move is independent of the others) and they need not complete the game.

Sample Input:

```
3
AA D T
E EXD
FX yFT
4
1 1 1 2
1 4 2 5
3 1 3 5
2 4 3 2
```

Sample Output:

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

Problem Number 3: Wolf and Sheep

Wolf and Sheep is played on the black squares of a chessboard (row 0 is at the top, column 0 is at the left). The sheep start in row 0 as shown below:

	0		1		2		3
■		■		■		■	
	■		■		■		■
■		■		■		■	
	■		■		■		■
■		■		■		■	
	■		■		■		■
■		■		W		■	

There are four sheep on one side and one wolf somewhere on the opposite side. A move is on a diagonal, from a black square to a diagonally adjacent square. In each round, the wolf moves, forward or backward, and then one sheep moves, only forward. The sheep win if they can immobilize the wolf (i.e., the wolf can no longer move), the wolf wins if it reaches the sheep's original side. The wolf moves first.

You are to write a program that moves the sheep. Your program will be judged correct if immobilizes the wolf before it reaches the sheep's original side of the board. Your program will write either "Sheep won" or "Wolf won" to standard error depending on whether or not the wolf was immobilized, and then terminate.

Your program will start by reading one line of input containing two positive numbers separated by a single blank: the row and column of the wolf's first move. The wolf can start anywhere in row 7; therefore, this first line of input contains 6 and any odd column number.

Your program will now move one of the sheep. When your program takes a turn, it has to write one line of output containing three positive numbers separated by single blanks and terminated by a *linefeed*: the number of a sheep (0 to 3) and the row and column to which this sheep moves; for the sheep's first move this will be row 1 and the appropriate even-numbered column. Your program must flush the output stream after every move.

After moving one of the sheep, your program will read another line of input (using the same format as before) representing the wolf's next move. The program will repeat this cycle until it terminates.

Your program is not allowed to peek at even a single byte beyond the linefeed terminating this line.

We have provided a graphical user interface that you can use to execute your program. You execute this program as follows:

```
java -jar ~/was.jar command-line-for-your-program
```

Problem Number 4: Arnie versus the IRS

Arnie's bakery is the leading producer of gourmet dog treats in the world. To comply with FDA requirements, Arnie etches the current date on each treat he produces. The date is etched using the format *YYYYMMDD*, where *YYYY* represents the year, *MM* represents the month, and *DD* represents the date (these values are padded with zeros if necessary). Arnie is very particular about the work that he does, and takes his time making treats, and as a result he only produces one treat a day.

The IRS has been after Arnie for some time, and this year they have decided to audit him. After working with the auditor for several days, the only remaining bone of contention is the writeoff that Arnie claims for etching the date on his products. In order to satisfy the auditor, and get back to work, Arnie must determine the number of each of the digits he engraved in his treats during a specific period of time.

For example, during the period 11/01/2004 through 11/02/2004 Arnie produced two treats and etched a total of 6 zeros, 5 ones, 3 twos, and 2 fours.

The input to your program will consist of two lines, each containing a date formatted as *MM/DD/YYYY*. The first line specifies the start of the period, and the second the end of the period (inclusive). You may assume that the first date occurs before or on the second and that the year in both dates is greater than or equal to 1920. Your program will produce as output a list that contains two columns. The first column contains the digits 0 to 9, in ascending order. The second column contains the number of times the corresponding digit appeared. Note that Arnie enjoys his free time (he likes to take his humans on long walks in the woods) and therefore does not work on Saturdays and Sundays.

Sample Input:

```
10/31/2004
11/02/2004
```

Sample Output:

```
0 6
1 5
2 3
3 0
4 2
5 0
6 0
7 0
8 0
9 0
```

Problem Number 5: Harry and the Dementors

Harry is trying to escape from a castle that is inhabited by a number of Dementors. If the Dementors are able to combine their powers they will be able to suck out Harry's soul, leaving him an empty shell, alive but completely, and irretrievably "gone".

The Dementors combine their powers by forming a network. When two Dementors can see each other in a hallway, they form a network and combine their powers. If any one of these Dementors can see another Dementor (i.e., they are already a member of another network), the networks are combined to form a larger, and even more powerful, network. While Harry is trying to escape from the castle, the Dementors move to create a network that contains every Dementor in the castle. If any Dementor in the final network can see Harry, the Dementors are able to overpower him.

Harry has outsourced his escape plan to you. A rectangular grid will be used to represent the castle that Harry is in. Each cell is either empty (a blank), contains Harry (H), contains a Dementor (a lowercase letter), or is made from stone (#). Your program will first read a line with a positive integer n , the number of rows of the mansion's floor plan. The next n lines contain the floor plan as strings of equal length, for example:

```
6
#####
#
# #a#
#H# #
##b #
#####
```

Harry escapes from the castle if he can get to any unoccupied cell on any edge of the grid. In this example Harry can escape in seven moves and there are two Dementors sitting in the middle of the mansion. There can be more than one exit from the castle.

On each turn, first Harry and then one Dementor take a single step in one of the four compass directions (N E W S). For example, the move:

```
H N a N
```

results in the following situation:

```
#####  
#  a  
#H# #  
# # #  
##b #  
#####
```

Nobody in the castle (Harry or any of the Dementors) can move through walls or squeeze through diagonal cracks. However, Harry and a Dementor can occupy the same cell (as long as the Dementors have not finished forming their network).

The following move:

```
H N b E
```

places Harry within five steps of the exit:

```
#####  
#H a  
# # #  
# # #  
## b#  
#####
```

However, Harry is in big trouble. The Dementors have completed their network (i.e., they have formed a connected graph whose edges only travel along the hallways) and Dementor a can see Harry, which means Harry is doomed.

Your program, after reading the initial grid, will determine whether or not Harry can escape from the castle given any possible sequence of moves made by the Dementors. If Harry cannot escape from the castle, your program will print the message:

```
There is no escape
```

If there is an escape, your program must output Harry's moves to freedom. You will print Harry's moves as a single string consisting of the letters, N, E, W, or S, that when executed in the order specified show how Harry escapes from the castle.

Sample Input:

```
9
#####
#
#   ## #
#  #a  #
#   ## #
#   H# #
#   ##
##b   #
#####
```

Sample Output:

```
WWNNNNW
```

Problem 6: MOAL Interpreter

You are required to write an interpreter to evaluate a program as described below.

A program consists of an ordered list of zero or more abbreviations, a period, and one final computation. An abbreviation consists of an identifier (key) followed by a computation (value) followed by a period. White space is completely ignored; identifiers are single letters or single digits. A computation has one of the following three forms:

```
identifier  
: identifier computation  
! computation computation
```

Here is an example of a program:

```
T : x : y x .  
I : c : t : e ! ! c t e .  
.  
! ! ! I T a b
```

Your interpreter would evaluate this to *a*.

Evaluation of a program is concerned with the final computation and has two phases, expansion, and simplification.

Expansion: Look at each identifier in the final computation. If the identifier is a key in an abbreviation, replace the identifier in the final computation by the value from the abbreviation. This will result in a longer final computation; start over and expand the new final computation. We guarantee that there will not be cyclic references in the abbreviations.

Simplification: You can only repeatedly and top-down simplify the following situation:

```
! : i a b
```

where *i* is an identifier and *a* and *b* are computations. Simplification consists of replacing declared occurrences of *i* in *a* by *b*. Such a declared occurrence is any *i* in *a* which does not occur inside a computation of the form

```
: i x
```

which itself is a computation contained in *a* and where *x* is another computation.

As an example, here is how the program above would be interpreted. Expansion produces

```

!
!
! :c
    :t
        :e
            !
                ! c t
                    e
            :x
        :y x
    a
b

```

The first simplification replaces *c* and produces

```

!
!
! :t
    :e
        !
            !:x
                :y x
                    t
            e
    a
b

```

The second simplification replaces *t* and produces

```

! :e
!
! :x
    :y x
    a
    e
b

```

The third simplification replaces *e* and produces

```

!
! :x
    :y x
    a
b

```

The fourth simplification replaces x and produces

$$\frac{! : y a}{b}$$

The last simplification replaces y and produces the correct output

a

Sample Input:

```
T : x : y x .  
I : c : t : e ! ! c t e .  
.  
! ! ! I T a b
```

Sample Output:

a

Problem 7: Intergalactic Licenses

Since the formation of the Intergalactic Nation of Solar Systems (INS), citizens of the various planets in the nation have to apply for an intergalactic driver's license in order to operate a vehicle on a planet. These licenses are valid from the date the application is submitted until the end of the driver's next birthday on the planet the license is issued for; relative to a planet, birthdays repeat once a year. Calculating an expiration date is complicated by the fact that different planets have days of different length and different calendars and everything is calculated as precisely as possible.

In an attempt to make interplanetary travel easier, the INS passed a law that standardized the way in which all of the planets in the nation denote a day and time. Time is measured by counting the number of standard time units (STU) that have passed since the start of a day. The length of a STU is the same on every planet - however, the number of STUs in a day may vary from planet to planet. The length of a STU was carefully selected so that the number of STUs in a day, on any planet, is always a positive integer. INS timestamps are written in the following format: *year/month/day+stu*. After the passage of this law, the time on all planets was set to 0/0/0+0. Years, months, and days start at 0.

The INS also passed a law that standardized calendars. The law eliminated leap years and specified that a planetary calendar be represented by a series of positive integers, separated by blanks, on a single line. The first number in the calendar specifies the number of STUs in a single day on that planet. The remaining numbers on the line define the number of days in each month on the planet (note that the number of these remaining numbers specifies how many months are in a year). The following line specifies the calendar used on the planet Earth:

```
86400 31 28 31 30 31 30 31 31 30 31 30 31
```

When applying for an intergalactic driver's license, travelers specify three planetary calendars: the calendar for the planet where they were born, the calendar for the planet where they live (their home planet), and the calendar for the planet where they want to operate a vehicle. They then supply two dates: their birth date and the date of the application.

You are to write a program that computes the expiration date for the license. The program will read 5 lines of input: the birth planet's calendar, the home planet's calendar, the calendar for the planet where a vehicle is to be operated, the birth date (on the birth planet), and the application date (on the home planet). Your program will produce as output a single line that contains the expiration date of the license on the home planet.

Sample Input

```
1000 10 10
100 200 200
10 3000 3000
0/1/2+3
1/0/5+0
```

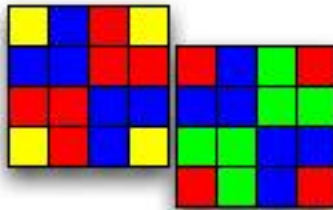
Sample Output

```
4/0/0+209
```

Problem 8: Continuo

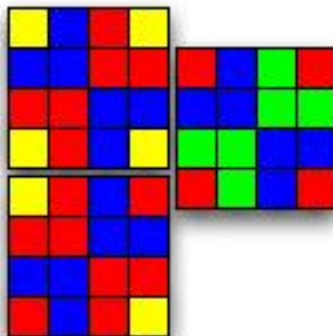
Continuo, developed by Maureen Hiron, was game of the year 1987 in England. The game is played with a subset of a deck of 42 colored cards (the complete set of cards is attached to this question), which are placed faced-up on a playing surface. The objective is to arrange the cards so that each card touches at least one other card and forms at least one uni-colored band into each card that it touches. A card can be rotated; however, it may not overlap another card. The objective of the game is to build long uni-colored bands. The game is scored by counting the total number of cells in all of the bands that involve two or more cards.

For example, here cards 26 and 19 have been arranged to form two bands:



The value of the cards in the example above would be ten, four cells for red, and six cells for blue. The only bands that are included in the count are the ones that span across two (or more) cards.

Card 34 could be turned counterclockwise by 90 degrees and added to the example above at the bottom left, resulting in the following configuration:



The score of these three cards is 23, twelve red, two yellow, and nine blue.

The input to your program will describe the subset of cards that should be placed. The first line of input will contain a single positive integer n that specifies the number of cards your program will read. The remaining input will specify the cards. Each card is represented by one line of input for each row of cells in the card. Each line will contain a single 4-character string that specifies the color of

the cells in the corresponding row of the card. The string will consist of a combination of the characters *r*, *g*, *b*, and *y*, representing the colors red, green, blue, and yellow respectively.

For example, card number 26 alone would be represented as:

```
1
ybry
bbrr
rrbb
yrby
```

Your program will produce as output the best score that can be achieved with the cards given, where you may not need to use all of them. Your program will print the total number of red, green, blue, and yellow cells in the best score, and an arrangement of the cards that has the best score. You may assume that the display device is capable of displaying all of the characters in the longest line of your output.

Sample Input:

```
2
ybry
bbrr
rrbb
yrby
rbgr
bbgg
ggbb
rgrb
```

Sample Output:

```
Total score: 10 (4 red, 0 green, 6 blue, 0 yellow)

ybry
bbrrrbgr
rrbbbbgg
yrbyggbb
  rgrb
```