CEOI 2013 – Tasks and Solutions

Primošten, Croatia, 2013

PREFACE

The Central European Olympiad in Informatics is an annual programming contest gathering best high-school programmers from the Central European region. The 20th Central European Olympiad in Informatics was held in Primošten, Croatia between October 13th and October 19th of 2013. A total of nine countries participated: Croatia, Czech republic, Germany, Hungary, Poland, Romania, Switzerland, Slovakia and Slovenia – each sending a team of up to 4 students, except the host country Croatia which participated with two teams.

The contest was won by Eduard Batmendijn from Slovakia with Andrei Heidelbacher from Romania and Ivan Lazarić from Croatia finishing second and third and also winning gold medals.

This booklet contains contest tasks and solution descriptions. More contest materials, including the official test data, helper tools and the full contest results are available from the CEOI 2013 website at http://ceoi2013.hsin.hr.

Tasks, solutions and other problem preparation provided by the CEOI 2013 Scientific Committee:

- Adrian Satja Kurdija
- Gustav Matula
- Anton Grbin
- Goran Žužić
- Lovro Pužar
- Luka Kalinovčić
- Ante Đerek

Credits for particular tasks are as follows:

- *Treasure*: Proposed by Adrian Satja Kurdija, algorithm description by Adrian Satja Kurdija
- *Tram*: Proposed by Adrian Satja Kurdija, algorithm description by Luka Kalinovčić and Adrian Satja Kurdija
- *Splot*: Proposed by Anton Grbin and Ante Đerek, algorithm description by Ante Đerek
- *Board*: Proposed by Adrian Satja Kurdija, algorithm description by Gustav Matula
- *Adriatic*: Proposed by Ante Đerek and Luka Kalinovčić, algorithm description by Gustav Matula
- *Watering*: Proposed by Luka Kalinovčić, algorithm description by Adrian Satja Kurdija

The Scientific Committee thanks Ognjen Dragoljević, Stjepan Glavina and Ivan Katanić for providing useful feedback on the early versions of the tasks.

TREASURE

After a recent earthquake, a new island has emerged in the Adriatic sea! The island is mostly barren except for an unusual device that was discovered. The name "oracle" quickly caught on for the device. Although the oracle came with no instruction manual, a crack team of archaeologists and computer experts was able to understand its behavior.

The oracle provides information about the locations of treasure on the island. The island is divided into a grid of cells consisting of *N* rows and *N* columns, with both rows and columns numbered 1 through *N*. Some of the cells in the grid contain treasure. The oracle answers questions of the form "Given a rectangle in the grid, how many cells in the rectangle contain treasure?"

Although the oracle answers questions for rectangles of all sizes, it was found that the more specific the information requested (the smaller the rectangle), the more energy is used by the oracle when answering. More precisely, if a rectangle contains *S* cells, then the oracle uses exactly 1 + *N***N* - *S* units of energy to answer.

TASK

Write a program that will, given the ability to interact with the oracle, find the **locations of all cells on the island that contain treasure**. We do not want to use too much energy in the process – the less energy is used, the better. It is not required that the amount of energy used is the smallest possible – see the 'Grading' section for details on how your solution will be scored.

INTERACTION

This is an interactive task. Your program asks the oracle questions using the standard output, and receives answers by reading from the standard input.

- At the beginning of your program, it should read an integer *N* ($2 \le N \le 100$), the size of the grid.
- To ask the oracle a question, your program should output a line containing four integers, R_1, C_1, R_2 and C_2, separated by spaces such that both $1 \le R_1 \le R_2 \le N$ and $1 \le C_1 \le C_2 \le N$ hold. If the conditions do not hold or the line is incorrectly formatted, your program will receive a score of zero on that test run.
- The oracle will respond with a line containing a single integer – the number of cells in the provided rectangle that contain treasure. More precisely, the number of cells (*R*, *C*) such that $R_1 \le R \le R_2$ and $C_1 \le C \le C_2$ and the cell at row *R*, column *C* contains treasure.
- When your program is done asking questions, it should output 'END' on a single line. After that it should output *N* lines, one for each row in the grid, each containing a string of *N* characters '0' (zero) or '1' (one).The *C*-th character in the *R*-th line is '1' if there is treasure in the cell at row *R*, column *C* or '0' if not. Rows are numbered 1 to *N* top to bottom, columns left to right. The execution of your program will be automatically terminated once your program outputs a solution.
- In order to interact properly with the grader, your program needs to flush the standard output after each question and after writing the solution. The provided code samples show how to do this.

You may assume that, in each test run, the oracle will correctly answer questions and the locations of treasure will be chosen prior to the interaction. In other words, the grader is not adaptive and the answers will not depend on previous questions asked by your program.

CODE SAMPLES

Code samples in all three programming languages are available for download on the 'Tasks' page of the contest system. The purpose of the samples is only to show how to interact with the oracle; these are not the correct solutions and may not score any points.

GRADING

Each test case is worth 10 points. If the output of your program is incorrect, you get zero points for that test case. Otherwise, the number of points awarded to your program depends on the total number of energy units K used by the oracle. More precisely:

- If $K \leq 7/16\ N^4 + N^2$, your score is 10 points.
- Otherwise, if $K \leq 7/16\ N^4 + 2\ N^3$, your score is 8 points.
- Otherwise, if $K \leq 3/4\ N^4$, your score is 4 points.
- Otherwise, if $K \leq N^4$, your score is 1 point.
- Otherwise, your score is 0 points.

Additionally, in test data worth at least 40% of total points, N will be at most 20.

Your output will be considered correct even if your program guesses the correct solution, e.g. if it does not even ask a single question.

EXAMPLE

In the following example, commands are given in the left column, row by row. Feedback is given in the second column of the corresponding row.

output	input
	2
1 1 1 1	0
1 2 1 2	1
2 1 2 2	2
END	
01	
11	

TESTING

There are two ways to test your program, locally or through the contest system. In both cases, you must first create a file that describes the test case. The first line of the test file should contain the integer N. The following N lines should describe the locations of treasure in the same format as the output of your program. For example, this would be the test file for the example in the previous section:

```
2
01
11
```

To test through the contest system it is first necessary to submit the source code for your program using the “Submit” page, and then use the “Test” page. The contest system will tell you if your program produced the correct output and provide information on the number of queries and energy units used.

To test locally, use the `treasure_test` binary, which can be downloaded through the contest system. The binary is used with a command like `./treasure_test ./my_solution input_file`. The output will contain feedback on whether your program has correctly solved the test case while the `treasure.log` file will contain information about the queries issued by your program and responses received.

SOLUTION

An approach for 40 points asks a query for each rectangle with an upper-left corner located in cell $(1,1)$. Now, using the inclusion-exclusion principle, we can easily calculate whether a cell (r,c) contains treasure (see Figure 1 below):

$$\begin{aligned} treasure[r][c] = {} & response(1,1,r,c) - response(1,1,r-1,c) - response(1,1,r,c-1) \\ & + response(1,1,r-1,c-1) \end{aligned}$$

The special case, when the cell (r,c) is in the first row or first column is even easier.

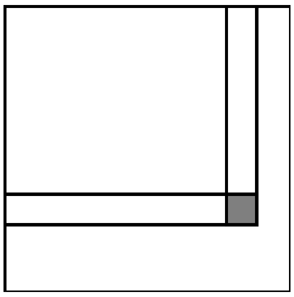


Figure 1

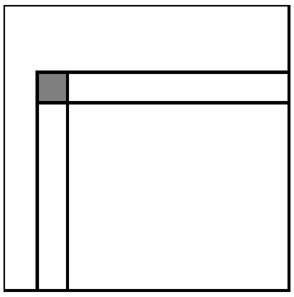


Figure 2

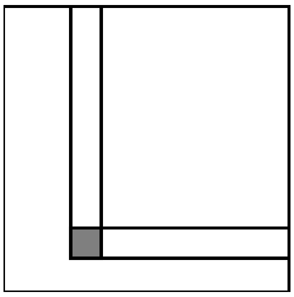


Figure 3

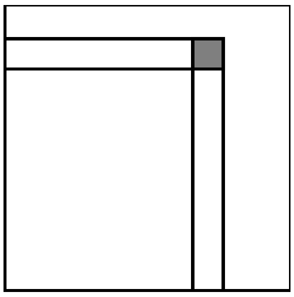


Figure 4

For the cells (r,c) belonging to the upper-left quarter of the island, that approach asks queries for some very small rectangles, which is expensive. For these cells, instead of asking for the rectangles with a corner located in $(1,1)$, we can ask for the rectangles with a corner located in (N,N) and then use a similar formula (see Figure 2 above):

$$\begin{aligned} treasure[r][c] = {} & response(r,c,N,N) - response(r,c+1,N,N) - response(r+1,c,N,N) \\ & + response(r+1,c+1,N,N). \end{aligned}$$

Analogously, for (r,c) in the upper-right quarter of the island, we can ask for the rectangles with a corner located in $(N,1)$. For (r,c) in the lower-left quarter of the island, we can ask for the rectangles with a corner located in $(1,N)$ (see Figures 3 and 4 above).

The algorithm, therefore, does the following: for every cell (r,c), find the largest among four rectangles beginning at (r,c) and ending in a corner of the island, and ask the oracle for this rectangle. Finally, calculate the values of all cells using the formulas as above.

You might notice that, in this approach, tricky situations appear for the cells in the middle row/column if N is odd or two middle rows/columns if N is even. In these situations, we do not have responses for some of the rectangles appearing in the formulas above (because they are a bit smaller), but we can calculate their values using other, larger rectangles that we do have responses for. These observations make a difference between 80% and 100% of the points.

TRAM

Seats in a new tram operating in Zagreb are organized into a grid consisting of N rows numbered 1 through N and two columns numbered 1 and 2. The distance between two seats, one at row R_A, column C_A and other at row R_B, column C_B is the Euclidean distance between the centers of the corresponding grid squares – namely $\sqrt{(R_A - R_B)^2 + (C_A - C_B)^2}$.

Most passengers prefer solitude when using public transportation and they always try to choose a seat that is as far away from other passengers as possible. More precisely, when a passenger enters the tram he or she will choose a free seat whose **distance from the closest occupied seat is the highest possible**. If there is more than one such seat, they will always choose one with the lower row number and if there is still more than one such seat, they will choose the one with the lower column number. After the passenger chooses a seat, he or she will sit there until leaving the tram. If the tram is empty, the next passenger to enter will always choose the seat in row 1 and column 1.

TASK

Write a program that will, given a sequence of events, each event either a passenger entering or leaving the tram, determine where each of the passengers was sitting. The tram is initially empty.

There are M events in the input numbered 1 through M in the order in which they occurred. There are two kinds of events: event of type 'E' corresponds to a passenger entering the tram, while the event of type 'L' corresponds to a passenger leaving the tram. For an event of type 'L', an integer P is also given – it specifies that the passenger leaving in this event is the one that entered at event P.

Test data will be such that there will always be at least one free seat in the tram whenever a passenger is trying to enter.

INPUT

The first line of input contains two integers N and M ($1 \le N \le 150\ 000$, $1 \le M \le 30\ 000$), the number of rows in the tram and the number of events. The following M lines contain the description of the events, K-th of those M lines contains the description of event K – either the character 'E', or the character 'L' followed by a single space and the integer P_K ($1 \le P_K < K$). Each P_K will be valid- event P_K is of type 'E' and no passenger will try to leave twice.

OUTPUT

The number of lines in the output should be equal to the number of events of type 'E' in the input. For each event of type 'E', in the order in which they occurred, output on a single line the row and the column number of the seat chosen by the passenger, separated by a single space.

GRADING

- In test cases worth a total of 25 points, it holds $N \le 150$ and $M \le 150$.
- In test cases worth a total of 45 points, it holds $N \le 1500$ and $M \le 1500$.
- In test cases worth a total of 65 points, it holds $N \le 150\ 000$ and $M \le 1500$.

DETAILED FEEDBACK WHEN SUBMITTING

During the contest, you may select up to 50 submissions for this task to be evaluated on a part of the official test data. When the results are ready, a summary of the results will be available on the contest system.

EXAMPLES

input

```
3 7
E
E
E
L 2
E
L 1
E
```

output

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

input

```
13 9
E
E
E
E
E
E
E
E
E
```

output

```
1 1
13 2
7 1
4 2
10 1
2 2
3 1
5 1
6 2
```

input

```
10 9
E
E
E
E
L 3
E
E
L 6
E
```

output

```
1 1
10 2
5 2
7 1
4 2
2 2
4 1
```

SOLUTION

First, we define *interesting* rows. The first and the last row are always interesting; any other row is interesting when one or both of its seats are occupied.

If r_1 and r_2 are interesting rows with no other interesting rows between them, they define the interval $[r_1,\ r_2]$. This interval includes r_1 and r_2, which means that an interesting row can belong to two intervals, i.e. the intervals are not all disjoint.

There are at most four occupied seats in each interval $[r_1,\ r_2]$, namely: $(r_1, 1)$, $(r_1, 2)$, $(r_2, 1)$ or $(r_2, 2)$. Hence, the best seat in the interval is always in one of the following eight locations:

$r_1, 1$	$r_1, 2$
…	…
$floor((r_1\ +\ r_2)\ /\ 2), 1$	$floor((r_1\ +\ r_2)\ /\ 2), 2$
$ceil((r_1\ +\ r_2)\ /\ 2), 1$	$ceil((r_1\ +\ r_2)\ /\ 2), 2$
…	…
$r_2, 1$	$r_2, 2$

Therefore, to evaluate an interval, we can use a short loop to find the best empty seat among these eight (we only need to check the distances to the occupied seats in rows r_1 and r_2), as well as the best distance for this interval.

When a person comes in, which interval should we choose? It should be the one with the maximal distance (calculated as described above), and if there are more such intervals, the one with the best seat closest to the entrance. Therefore, we will keep the intervals in priority queue, ordered according to these criteria. When a person comes in or out, we might need to update some intervals: either reevaluate an interval, or split one interval into two intervals, or connect two intervals into one. In all of these cases, we need to erase an interval (or two) from the priority queue and insert a new one (or two). A priority queue with the ability to erase elements can be implemented using binary heap or C++/STL's set.

To maintain the interesting rows, we only need a few arrays: for each row we keep track of who is sitting there and, if the row is interesting, what the previous and the next interesting row is, and which intervals it belongs to. This information is sufficient to find all intervals that are interesting for the event and to update them successfully.

SPLOT

The Adriatic coast and islands are lined with amazing beaches of all shapes and sizes. However, many beaches are not accessible by car. To accommodate the growing demand, a giant field near the coast has been converted into a parking lot. Since all of the architects involved have electrical engineering backgrounds, the layout of the parking lot resembles a series-parallel graph often used when designing electrical circuits.

The parking lot consists of parking spaces and two-way roads connecting them. Each road connects two different parking spaces and each pair of parking spaces is connected by at most one road. At any moment, each parking space can be occupied by **at most one car.** No other cars can travel through an occupied parking space.

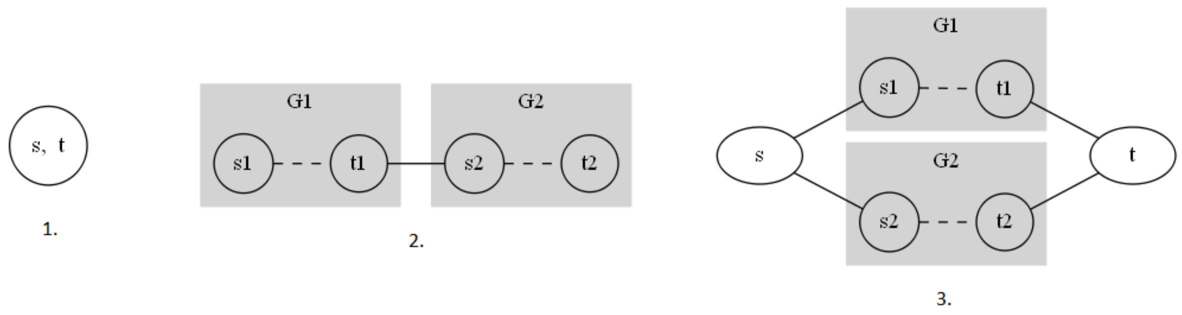


Figure 5: Rules for building splots, each described in the corresponding item below

A series-parallel parking lot (also called *splot*) is a parking lot with two distinguished parking spaces called *source* and *terminal* that is constructed from individual parking spaces using the rules of serial and parallel composition. Each splot can be specified by its encoding – a sequence of characters describing its structure and the locations of parked cars. The valid splots and their encoding are defined recursively as follows:

1. A parking lot consisting of a single parking space and no roads is a valid splot. This single parking space is both the source and the terminal of the splot. The encoding of this splot is simply the lowercase letter '`o`' if the parking space is empty and the lowercase letter '`x`' if the parking space is occupied by a car.
2. If G_1 and G_2 are two valid splots, their *serial composition* G is also a splot. The serial composition is obtained by adding a road between the terminal of G_1 and the source of G_2. The source of the newly obtained splot G is the source of G_1 while the terminal of G is the terminal of G_2. If E_1 and E_2 are encodings of splots G_1 and G_2 respectively then the encoding of G is '`S`E_1E_2`#`'. In other words, the encoding is obtained by concatenating the uppercase letter '`S`', the encodings of splots being composed and the hash character '`#`'.
3. If G_1 and G_2 are two valid splots, their *parallel composition* G is also a valid splot. The parallel composition is obtained by adding two new parking spaces called s and t, adding the roads between s and the sources of both G_1 and G_2, as well as roads between t and the terminals of both G_1 and G_2. The source of the newly obtained splot G is the newly added parking space s while the terminal of G is the newly added parking space t. If E_1 and E_2 are encodings of splots G_1 and G_2 respectively and E_s and E_t are encodings of the source s and terminal t (lowercase letter '`o`' if the corresponding space is empty and lowercase letter '`x`' otherwise) then the encoding of G is '`P`E_s`|`E_1E_2`|`E_t`#`'. In other words, the encoding is obtained by concatenating the uppercase letter '`P`', the encoding of the source parking space, the pipe character '`|`', the encodings of splots being composed, another pipe character '`|`', the encoding of the terminal parking space, and finally the hash character '`#`'.

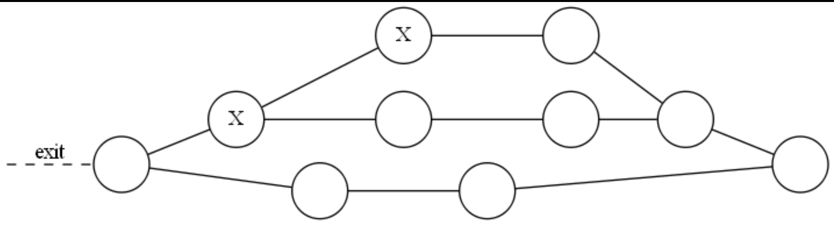


Figure 6: Splot corresponding to the first test example below

For example, the encoding of the splot given in the figure above is '`Po|Px|Sxo#Soo#|o#Soo#|o#`'. Notice that number of lowercase letters in the encoding of a splot *G* is always the same as the number of parking spaces in *G* and that there is a one-to-one correspondence between parking spaces in *G* and the lowercase letters in its encoding.

There is **exactly one exit** from the parking lot and it is directly connected to the source parking space of the overall splot. We say that the car is **not blocked** if it can **exit the splot**, i.e. it can reach the source parking space **via some sequence of roads and empty parking spaces**. For example, in the splot above neither of the cars is blocked, but if we were to park a car in the terminal of the splot (the rightmost node) then one of the cars would become blocked. It is allowed to park a car in the source parking space of the splot, however, if we were to do that, then all other cars in the splot would be blocked.

TASK

The operators of the parking lot would like to arrange incoming cars in such a fashion that **no car is blocked**. Suppose we are given a splot that may already contain some cars but none of those cars are blocked. Write a program that will calculate the **largest total number of cars** that can be parked in the given splot, including the cars already there, without **any of the cars being blocked** and without moving any of the cars already there. Additionally, your program should find one way to arrange this largest number of cars in the splot.

INPUT

The first line of input contains a sequence of at least 1 and at most 100 000 characters – the encoding of the given splot. The only characters appearing in the sequence will be the uppercase letters '`P`' and '`S`', the lowercase letters '`o`' and '`x`', and characters '`#`' (ASCII 35) and '`|`' (ASCII 124). The input will be an encoding of a splot according to the rules above. None of the cars already in the splot will be blocked.

OUTPUT

The output should contain 2 lines. The first line should contain a single integer *M* – the largest number of cars that can be parked in the splot as described above.

The second line should contain a sequence of characters - the encoding of the splot with an optimal arrangement of cars. The sequence should contain exactly *M* occurrences of the letter '`x`' and be obtainable from the input sequence by replacing some of the letters '`o`' by '`x`'.

There may be more than one optimal arrangement and your program may output any one of them.

GRADING

- If the output is incorrect or incomplete, but the first line of output (the maximal number of cars) is correct, the solution will be awarded 80% of the points for the corresponding test case.
- In test cases worth a total of 30 points, it holds that the total number of parking spaces in the splot is at most 20.
- In additional test cases worth a total of 40 points, the splot is empty, i.e. the input does not contain the letter '`x`'.

EXAMPLES

input

```
Po|Px|Sxo#Soo#|o#Soo#|o#
```

output

```
3
Po|Px|Sxo#Sox#|o#Soo#|o#
```

input

```
Po|SPo|oo|o#Px|oo|o##Po|Sxo#Po|ox|o#|o#|o#
```

output

```
7
Po|SPo|xx|o#Px|ox|o##Po|Sxx#Po|ox|o#|o#|o#
```

TOOL SUPPORT

A binary called `splot_tool` (available for download through the contest system) can help you visualize the test inputs and outputs. The tool takes a filename as an argument - either a validly formatted input file or an output file for this tasks and generates an svg image depicting the splot encoded in the file. The generated image can be viewed using a web browser. Sample usage of the tool is:

```
$ ./splot_tool splot.dummy.out.1
splot.dummy.out.1 parsed (10 parking spaces).
splot.dummy.out.1.svg created.
$ chromium splot.dummy.out.1.svg
```

The tool will check that the splot is properly formatted and give an appropriate error message otherwise. It **will not** perform any other validity checks - the files may render properly even if they are incorrect solutions.

The tool will only work for splots whose encoding is at most 200 characters long.

SOLUTION

Since the splots are defined recursively, it makes sense to try to find a recursive formula for the answer to the problem. After considering the serial and parallel composition, it becomes necessary to introduce additional functions that we need to compute. The solution will, for all sub-splots G of the given splot, calculate several functions – each denoting the total number of cars that can be parked in the splot G (including the cars already there) so that each parked car can exit G somehow – the functions mostly differ on how exactly the car can exit G:

- $FORWARD[G]$: Each parked car can exit via the source node of G.
- $BACKWARD[G]$: Each parked car can exit via the terminal node of G.
- $BOTH[G]$: Each parked car can exit either via the source node of G or the terminal node of G.
- $THROUGH[G]$: Same as the previous case, but, additionally, it is possible to travel through G from the source to the terminal, i.e. when all cars are parked, there is a path between s and t consisting of empty parking spaces.

In all the functions above, we will use a special symbol (e.g $-\infty$) to denote that it is not possible to reach any such arrangement due to cars already parked in the splot.

Additionally, to make the formulas simpler, we introduce the following function for each sub-splot:

- $NONE[G]$: is zero if the splot is empty and $-\infty$ otherwise.

Now, we need to find recursive formulas for each of the functions above. In this algorithm description, we will only explain the recursive formula for the function $FORWARD$ and just list the formulas for the other functions. We leave it to the reader to verify the other formulas. In addition, we will use the fact that it always holds $THROUGH \le FORWARD, BACKWARD \le BOTH$ in order to reduce the number of cases that need to be considered.

1. If the splot consists of one node then the values of $FORWARD$, $BACKWARD$, and $BOTH$ are always 1, while the values of $THROUGH$ and $NONE$ are either 0 (if the splot is empty) or $-\infty$ otherwise.
2. Suppose we are serialy composing splots G_1 and G_2 and want to calculate the function $FORWARD$ for the resulting splot G. There are two possibilities: a) at least one car is parked in G_2 and b) G_2 is empty. In the first case, there has to be a path through G_1 in order for cars from G_1 to exit foward, hence largest number of cars in this case will be $THROUGH[G_1] + FORWARD[G_2]$. In the second case, the largest number of cars is simply $FORWARD[G_1] + NONE[G_2]$ as G_2 has to be empty and the $FORWARD[G_1]$ exactly calculates the best arrangement in this case.
3. Suppose now G is the parallel composition of G_1 and G_2 and, again, we are calculating the function $FORWARD$ for G. Parallel composition is more complicated since we need to also consider the configuration of the newly added source and terminal. We distinguish 3 cases depending on the final configuration of source and terminal
   a. *Source is occupied, terminal is free*: Both G_1 and G_2 have to be empty in this case. Value of $FORWARD$ is $1 + NONE[G1] + NONE[G2]$.
   b. *Both source and terminal are free*: Now either cars from both G_1 and G_2 can exit directly forward or they can exit on both sides from one of them, while the other one has a path from the terminal to sink. Value of $FORWARD$ is, therefore, the largest of $FORWARD[G_1] + FORWARD[G_2], BOTH[G_1] + THROUGH[G_2], THROUGH[G_1] + BOTH[G_2]$.
   c. Source is free, terminal is occupied: One of the sub-splots has to be passable for the car park in terminal to exit, all cars from the other one need to exit forward. Value of $FORWARD$ is therefore larger of $1 + THROUGH[G_1] + FORWARD[G_2], 1 + FORWARD[G_1] + THROUGH[G_2]$.

Even when all possible rules for recursively evaluating the functions are found, the implementation (especially with reconstruction of the optimal arrangement) remains challenging and error prone due to the sheer number of cases that need to be considered. One nifty trick that greatly simplifies the implementation is to use constants to define recursive rules described above.

```
const int S_RULE[][3] = {
  {FORWARD, FORWARD, NONE},
  {FORWARD, THROUGH, FORWARD},
  {BACKWARD, NONE, BACKWARD},
  {BACKWARD, BACKWARD, THROUGH},
  {BOTH, FORWARD, BACKWARD},
  {BOTH, BOTH, THROUGH},
  {BOTH, THROUGH, BOTH},
  {THROUGH, THROUGH, THROUGH},
  {NONE, NONE, NONE},
  {-1, -1, -1}
};
```

```
const int P_RULE[][5] = {
  {FORWARD, 1, 0, NONE, NONE},
  {FORWARD, 0, 0, FORWARD, FORWARD},
  {FORWARD, 0, 0, THROUGH, BOTH},
  {FORWARD, 0, 0, BOTH, THROUGH},
  {FORWARD, 0, 1, THROUGH, FORWARD},
  {FORWARD, 0, 1, FORWARD, THROUGH},
  {BACKWARD, 0, 1, NONE, NONE},
  {BACKWARD, 0, 0, BACKWARD, BACKWARD},
  {BACKWARD, 0, 0, THROUGH, BOTH},
  {BACKWARD, 0, 0, BOTH, THROUGH},
  {BACKWARD, 1, 0, THROUGH, BACKWARD},
  {BACKWARD, 1, 0, BACKWARD, THROUGH},
  {BOTH, 1, 1, NONE, NONE},
  {BOTH, 1, 0, BACKWARD, BACKWARD},
  {BOTH, 0, 1, FORWARD, FORWARD},
  {BOTH, 0, 0, BOTH, BOTH},
  {THROUGH, 0, 0, THROUGH, BOTH},
  {THROUGH, 0, 0, BOTH, THROUGH},
  {NONE, 0, 0, NONE, NONE},
  {-1, -1, -1, -1}
};
```

Table 1: Recursive rules described as constants

The table above encodes rules for recursevily calculating functions as simple integer arrays in C++. For example the `{FORWARD, 0, 1, THROUGH, FORWARD}` entry in the second column says that when G is a parallel composition of G_1 and G_2 and function being calculated is $FORWARD$, we need to consider the case where sink is empty, terminal is occupied, G_1 is in the optimal $THROUGH$ configuration and G_2 is in the optimal $FORWARD$ configuration.

With the constants defined as above the solution is easy but still long, a parser needs to be implemented that converts the encoding to a tree-like structure. The tree is then recursively traversed calulating each of the functions. In order to reconstruct the optimal arrangement we need to remember the index of the rule that was used to obtain the maximal value of the corresponding function.

BACKGROUND

In the original version of this problem, the parking lot was a rectangular grid, but we could not solve this variant. It turns out that the problem is closely related to the *minimum connected dominating set* problem and its dual, the *maximum leaf spanning tree* problem. Both problems are NP-complete on arbitrary graphs but, as demonstrated, can be solved efficiently on SP-graphs.

We believe there is a linear solution to the problem even if arbitrary number of splots can be composed in each step and even if there is an arbitrary number of exists in the splot. We leave the details to the reader.

BOARD

Mirko and Slavko have a new board game. The game board resembles a complete infinite binary tree. More precisely, the board consists of nodes and two-way roads connecting them. The root node is located at the top of the board and we say it is at *level zero*. Each node has exactly two children, the *left child* and the *right child*, located in the lower-left and the lower-right directions of the parent node. The level of a child node is one greater than the level of the parent node. In addition to roads connecting a parent node with its children, there are roads connecting all of the nodes at a particular level – for each level, starting from the leftmost node, there is a road connecting each node to the next node to the right on the same level.

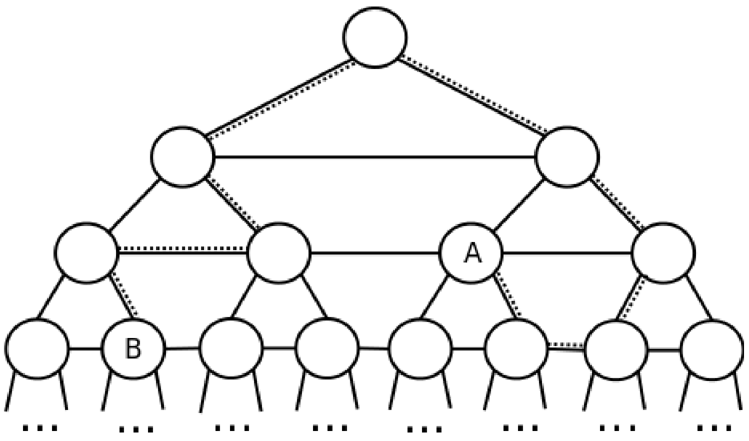


Figure 1: The second test example below

Each *path* through the game board is a sequence of steps, each moving from a node to a different node via a single road. Each step can be described by a single character as follows:

- character '1' describes moving from a node to its left child,
- character '2' describes moving from a node to its right child,
- character 'U' describes moving from a node to its parent,
- character 'L' describes moving from a node to the next node to the left on the same level,
- character 'R' describes moving from a node to the next node to the right on the same level.

For example, if we were to start at the root node and take the sequence of steps '221LU' we would end up in the node denoted with the letter 'A' in the figure above.

TASK

Write a program that will, given two nodes on the board, find the smallest number of steps needed to go from one node to the other. The two nodes are given by specifying paths from the root node to them. If the two paths lead to the same node, the answer is zero.

INPUT

The first line of input contains a sequence of at most 100 000 characters – the path from the root to the first node.

The second line of input contains a sequence of at most 100 000 characters – the path from the root to the second node.

The two paths will be valid (it will be possible to make every move in both sequences).

OUTPUT

The first and only line of output should contain a single integer - the smallest number of steps needed to go from one node to the other.

GRADING

Let *D* be the smallest integer such that both input paths only visit nodes whose levels are at most *D*.

- In test cases worth a total of 20 points, *D* is at most 10.
- In test cases worth a total of 40 points, *D* is at most 50.
- In test cases worth a total of 70 points, *D* is at most 1000.

DETAILED FEEDBACK WHEN SUBMITTING

During the contest, you may select up to 50 submissions for this task to be evaluated on a part of the official test data. When the results are ready, a summary of the results will be available on the contest system.

EXAMPLES

input

```
111RRRRRRR
222
```

output

```
0
```

input

```
221LU
12L2
```

output

```
3
```

input

```
11111
222222
```

output

```
10
```

SOLUTION

Let us look at the *infinite binary tree* of the graph and index its vertices starting at the root with index 1. For some node with index n, we index its left child (in the tree) with $2 * n$, and right child with $2 * n + 1$:

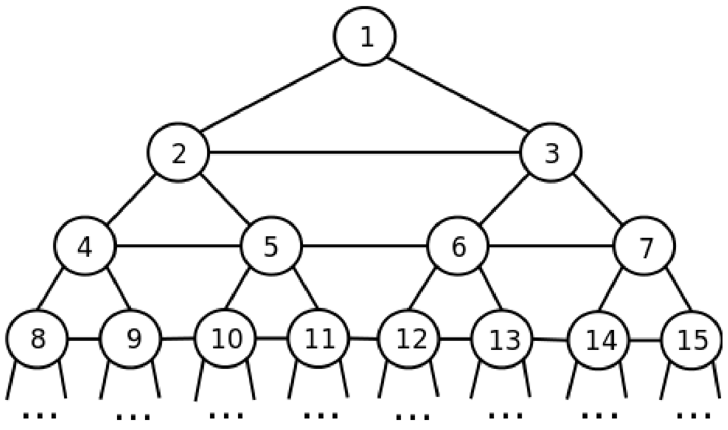


Figure 7: Indexing of board nodes

Note that if two nodes, a and b, are on the same level, the horizontal distance between them is $|a - b|$.

Now, let us talk about the shortest path between two nodes, indexed a and b, and suppose a is to the left of b (meaning that if we were to climb from a and b up to the common parent, a would be to the left of b). We will show that there always exists an optimal path starting at a and going up to some level, then zero or more steps to the right, then down to b as in the Figure 8 below.

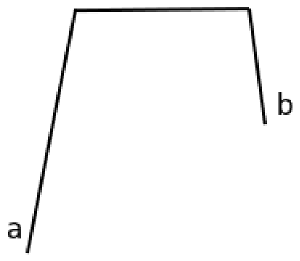


Figure 8

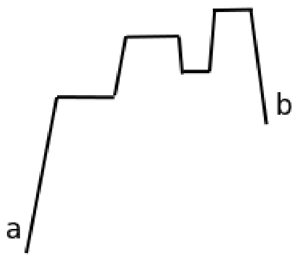


Figure 9

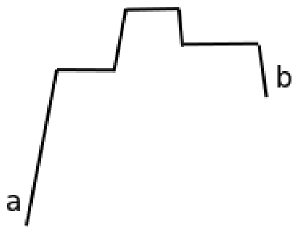


Figure 10

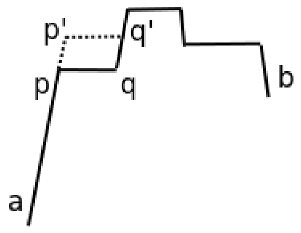


Figure 11

First, it is clear that we will not go left on this path, because we would surely go away from b.

Also, going down and then up (Figure 9) makes no sense. That means we will only go up/right and then down/right (Figure 10).

Let us look at nodes p and q in the Figure 11: The distance between the nodes is $|p - q|$, but if we move them up one step, it will be $|floor(p/2) - floor(q/2)| \leq |p - q|$, here $p' = floor(p/2)$ and $q' = floor(q/2)$.

Therefore, we can always transform an optimal path into one that goes up, then right, and then down.

This leads us to the following algorithm: try all paths of the shape stated above.

Now let us look at a possible implementation.

We can calculate the node index from the path by tracing it from the root.

The initial index is set to 1 (the root), and each character of the path description represents some operation applied to the index:

- '1': multiply by 2
- '2': multiply by 2 and add 1
- 'U': divide by 2 (integer division)
- 'L': subtract 1
- 'R': add 1

The operations can be implemented using an integer array that holds the binary representation of the node index. Increment and decrement correspond to incrementing/decrementing the last element and resolving any carries. Multiplication by 2 corresponds to adding a zero element to the end of the array. Division by 2 corresponds to removing the last element of the array.

With some care, carries can be resolved using lazy evaluation and a single pass through the array after all the operations have been applied. The only tricky operation is division by two, where we have to pass the carry from the last element to the one next to it before removing the element itself. The resulting number can have leading zeroes, so it might need to be shifted to the left.

After obtaining the indices, the rest is quite easy.

Let len_1 be the length (in bits) of the first index (a), and len_2 the same for the second (b). Let's denote by $pref(n, len)$ the prefix of length len of number n in binary representation. It is easy to see that

$$(len_1 - L) + |pref(a, L) - pref(b, L)| + (len_2 - L)$$

is the length of the path going from a to level L, then right and down to b. The first summand corresponds to the part leading up from node a to level L, the second is the length of the horizontal segment of the path, and the last is the length of the path leading from level L down to node b. Since

$$\begin{aligned} pref(a, level) &- pref(b, level) \\ &= (pref(a, level - 1) - pref(b, level - 1)) * 2 + a[level - 1] - b[level - 1] \end{aligned}$$

($a[i]$ is the 0-indexed i-th most significant bit of a), we can do the calculation using a simple loop over the possible levels. We stop when the absolute value of the difference becomes too big, or we reach $level = min(len_1, len_2)$.

ADRIATIC

"The land of a thousand islands" was an official motto of Croatian tourism in the mid nineteen-nineties. While the motto is technically inaccurate (there are slightly more than 1000 islands in Croatia), it is true that island hopping (sailing from island to island) is a popular summer activity.

For the purpose of this task, the map of the Adriatic sea is a grid consisting of unit squares organized into 2500 rows and 2500 columns. Rows are numbered 1 through 2500, north to south, while columns are numbered 1 through 2500, west to east. There are N islands in the sea, numbered 1 through N and each island is located inside some unit square of the grid. The location of island K is given by the coordinates of the corresponding grid square – its row number R_K and its column number C_K. Finally, no two islands have the same location.

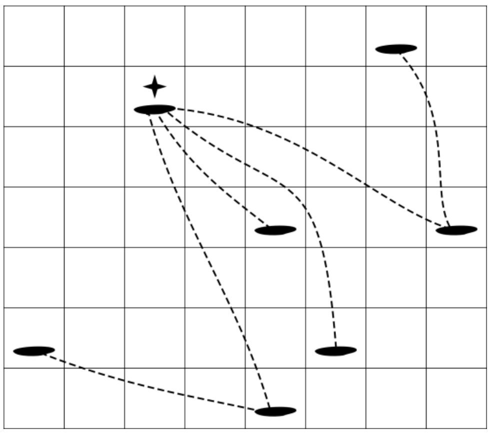


Figure 12: Map corresponding to the first test example below

Due to winds and sea currents, it is possible to sail directly from an island only to those islands that are located in the general northwest or southeast directions. More precisely, it is possible to sail from island A to island B **in one hop** if either both $R_A < R_B$ and $C_A < C_B$ hold or both $R_A > R_B$ and $C_A > C_B$ hold. Note that the distance between the two islands or the presence of other islands between them does not affect the possibility of hopping from one to the other. If it is not possible to hop directly from A to B, it might be possible to sail from A to B via other islands using some sequence of hops. The *sailing distance* from A to B is defined as the smallest number of hops required to sail from A to B.

For example, in the figure above, starting from the island at row 2, column 3, we can hop to four other islands while the sailing distance to the remaining two islands is two.

TASK

A sailing congress is being planned and the organizers are considering each of the islands as a possible location for the congress. When considering a candidate island they would like to know: if every other island sends a single sailboat, what is the **smallest total number of hops** required in order for **all sailboats to reach the candidate island**, or equivalently, what is the sum of sailing distances from all other islands to the candidate island. Write a program that will, given the locations of N islands, for each island K, calculate the sum of sailing distances from all other islands to island K.

Test data will be such that, for all islands A and B it is possible to sail from A to B using some sequence of hops.

INPUT

The first line of input contains an integer *N* (3 ≤ *N* ≤ 250 000), the number of islands. The following *N* lines contain the locations of the islands. Each location is a pair of integers between 1 and 2500 (inclusive), the row and column numbers, respectively.

OUTPUT

The output should contain *N* lines. For each island, in the same order they were given in the input, output the sum of sailing distances from all other islands on a single line.

GRADING

- In test cases worth a total of 25 points, N will be at most 100.
- In test cases worth a total of 50 points, N will be at most 1500.
- In test cases worth a total of 60 points, N will be at most 5000.
- In test cases worth a total of 80 points, N will be at most 25000.

EXAMPLES

input

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

output

```
16
11
12
11
12
16
8
```

input

```
4
1 1
2 3
3 2
4 4
```

output

```
3
4
4
3
```

SOLUTION

First, instead of going from every other island to some destination island, we can equivalently go from that destination island to every other island, which is somewhat easier to think about.

For a small number of points we can use a BFS (breadth-first-search) from each island to find the distances to all the other islands and then sum them up.

For full points we need some insight on how the BFS operates on this specific type of graph. Let us look at the following figure:

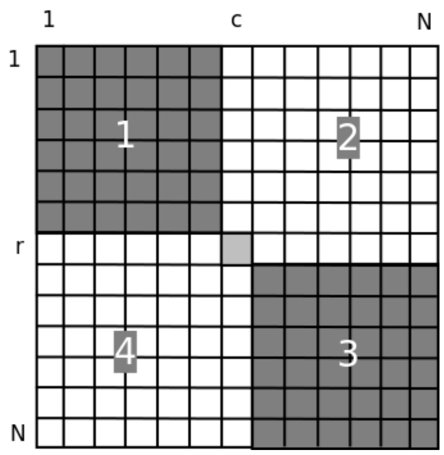


Figure 13: Dividing the grid into regions

Figure 13 shows a single starting island, and how the rest of the matrix is divided into four regions, (we do not count the starting island in any of those regions). Rules from the task description tell us we can reach all the islands in regions 1 and 3. Note that if we want to reach an island in region 2, we never need to go to region 4, since if we went there we would have to go to regions 1 or 3, and we could have went there in the first place.

This means that we can look at regions 2 and 4 separately. The problem is thus to compute the sailing distances to the islands in region 2 (region 4 will be completely analogous).

Let's see what the picture looks like when the BFS reaches islands at distance 2 (two hops away). Note that there are at most 4 islands at distance 1 we need to hop to in order to reach all the islands at distance 2. Those are the topmost, leftmost, bottommost and the rightmost island. Figure 14 shows the example four islands. Figure 15 shows the region of the matrix we can reach in two or less hops (note how the region only depends on the four *significant* islands):

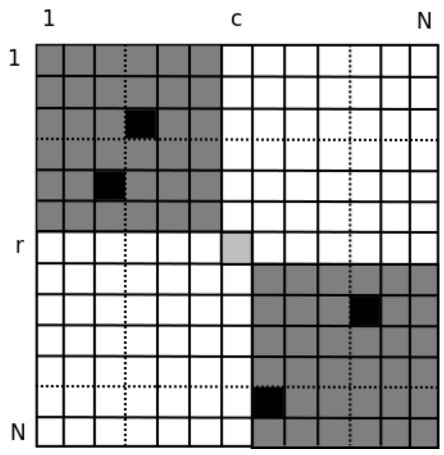


Figure 14: Significant islands at distance 1

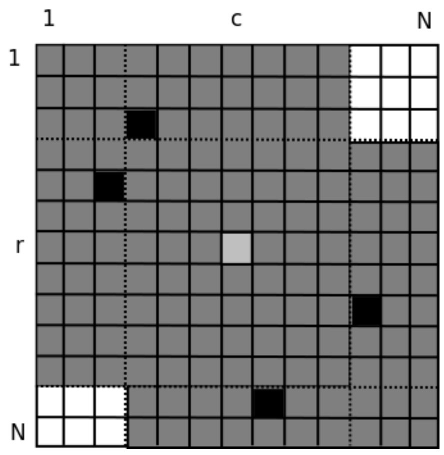


Figure 15: Cells that can be reached in two hops or less

We see that rectangle representing region 2 has shrunk compared to the first figure. The same holds for region 4. As the BFS progresses to islands 3, 4 or more hops away, the rectangle will keep shrinking until there are no more islands inside it to visit.

Note that the rectangle is defined by its bottom-left corner, and that all we need to know about the way it will shrink in the next step of the BFS is the topmost island to the left of it, and the rightmost island under it (those are marked light grey in the following figure):

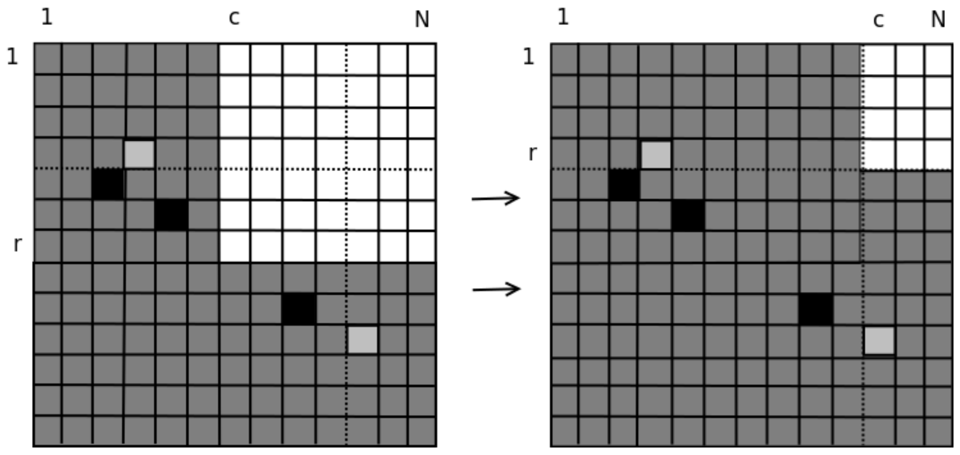


Figure 16: Shrinking the region rectangle

Let $dp(r,c)$ be the remaining number of hops required visit the islands in rectangle $(r,c) - (N,1)$.

Let $cnt(r,c)$ be the number of islands in the rectangle, and can be precomputed for all rectangles in $O(N^2)$ time using simple dynamic programming.

Now, if there are no islands in the rectangle ($cnt(r,c) = 0$), then $dp(r,c) = 0$ as we have nothing to hop to. Otherwise, when the BFS progresses the rectangle shrinks to $(min_r(r,c), max_c(r,c)) - (N,1)$, where $min_r(r,c)$ is the minimum row index over all islands to the left of the rectangle, and $max_c(r,c)$ is the maximum column index over all islands under the rectangle. Both of those can be precomputed for all rectangles in $O(N^2)$ time using simple dynamic programming. It is now easy to express $dp(r,c)$ as $cnt(r,c) + dp(min\_r(r,c), max\_c(r,c))$.

Applying the analogous idea for the bottom-left rectangle $(r,c) - (1,N)$ and combining the results gives us the total number of hops from a starting island (r,c).

BACKGROUND

The basic idea for the task is related to so called *permutation graphs* and the fact that there exists a O(n log n) algorithm for the single-source shortest path problem in such graphs (e.g. see [1]). A permutation graph is obtained if we consider a permutation of n numbers, construct a node for each number and add an edge between two nodes that are not in the correct order in the permutation (the larger one is in front of the smaller one). Points in a plane with norwest-southeast edges are a special case if all coordinates are different. When the coordinates can be same and they are additionally, bounded, we obtain this task which can, somewhat surprisingly, be solved with direct dynamic programming without any advanced data structures.

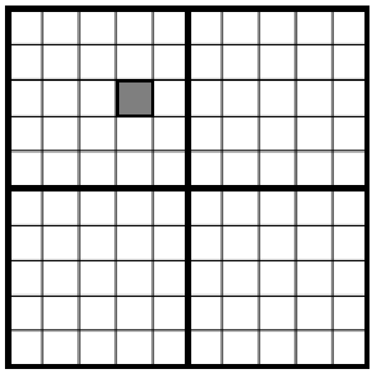
[1] O.H. Ibarra, Q. Zheng, Some Efficient Algorithms for Permutation Graphs, Journal of Algorithms, Volume 16, Issue 3, May 1994, Pages 453-469, ISSN 0196-6774,

(http://www.sciencedirect.com/science/article/pii/S0196677484710212)

WATERING

Sara is a dedicated farmer owning a large rectangular piece of land. Her land has a number of *cells* nicely arranged into a grid consisting of 5**R* rows and 5**C* columns. Furthermore, there is a horizontal fence across the width of the board after every fifth row and a vertical fence across the height of the board after every fifth column. The fences divide the land into *R***C* 5x5 areas called *fields*.

The two most common problems Sara faces are birds and droughts. To combat annoying crop-eating birds, some fields are equipped with a scarecrow. A scarecrow (if present) occupies a **single cell** and there can be **at most one scarecrow in each 5x5 field**.

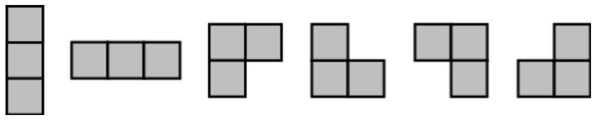


```
.....|.....
.....|.....
...#.|.....
.....|.....
.....|.....
-----+-----
.....|.....
.....|.....
.....|.....
.....|.....
.....|.....
```

```
aaacc|dxxxa
bbbce|dyyya
ddd#e|dzzza
ccbae|fccbb
cbbaa|ffcdb
-----+---_-
ssrrr|tttdd
saaax_xxeee
yxbbb|zdaaa
yxccc|zdbbb
yxddd|zdccc
```

Figure 1: Example layout of Sara's land, its textual representation and a valid sprinkler arrangement

During droughts, which can last for months, Sara uses sprinklers to keep her crops watered. Each of her sprinklers has three nozzles: one main nozzle and two side nozzles. It occupies **exactly three cells** and waters all of them. The side nozzles always occupy exactly **two cells that are adjacent** (up, down, left or right) to the main nozzle. Hence, a single sprinkler is always in one of the following configurations:



Sara wants to put sprinklers on her land in such a way that there is **exactly one sprinkler** on every cell that is not occupied with a scarecrow. A cell containing a scarecrow **must not** contain a sprinkler nozzle. In addition, nozzles must not be placed outside Sara's land.

The three cells watered by a single sprinkler need not belong to the same 5x5 field: they may also belong to the neighboring fields. In that case, Sara has to **drill a hole in the fence**, **between the two cells in neighboring fields watered by the same sprinkler**. Drilling a hole is difficult for Sara: she does not want to drill many.

TASK

Given the description of Sara's land you need to produce a valid configuration of sprinklers to water it. If you succeed, your score will depend on the total number of holes that need to be made in the fences – see the Grading section for details.

This is an output only task. You will be given 10 input files and you only need to produce the matching output files. You may download the input files from the contest system, on the page labeled 'Tasks'.

You need to submit each output file separately using the contest system. When submitting, the contest system will check the format of your output file. If the format is valid, the output file will be **graded and the score reported**; otherwise, the contest system will report an error. Hence, you will get **full feedback** for the output files submitted for this task.

Test data will be such that a solution always exists. If there is more than one solution, you may submit any one.

INPUT

The first line of the input contains a pair of integers R and C ($1 \le R, C \le 100$) – the size of Sara's land as described above.

$6*R$-1 lines follow, each containing a sequence $6*C$-1 characters. They represent Sara's fields and fences between them. The fence itself is also represented with characters even though the fence is actually infinitely thin.

A single cell is represented by a single character. The dot character '.' represents an empty cell, while the '#' character (ASCII 35) represents a scarecrow. Vertical fences are represented by the '|' character (ASCII 124) and the horizontal fences with the '–' character (minus). The '+' character denotes an intersection of fences.

OUTPUT

The output file should contain the textual representation of the field with a valid sprinkler arrangement in the same format as the input file. Each hole in the fence should be denoted by the underscore character '_'. All empty cells (dots) from the input file should be replaced by lowercase letters 'a' – 'z' so that the following rules are satisfied:

1. Any three cells watered by the same sprinkler are denoted by the same letter, even if not all of them are in the same 5x5 field.
2. If two adjacent cells **in the same field** are watered by different sprinklers, they must be denoted by different letters.
3. If two adjacent cells **in different fields** are watered by **different sprinklers** and there is a **hole in the fence** between them, they must be denoted by different letters.
4. It is allowed for adjacent cells that belong to different fields to be denoted by the same letter, as long as all the previous rules are satisfied.

GRADING

Each test case is worth 10 points. If the watering configuration is not valid, you will get zero points for that test case. If the configuration is valid, the solution will be graded as follows:

- If the number of holes in fences is not greater than $R*C$, your score is 10 points.
- Otherwise, your score is 5 points.
- In 4 out of 10 official test inputs, there will be a scarecrow in every field.

EXAMPLE

input

```
2 2
.....|.....
.....|.....
...#.|.....
.....|.....
.....|.....
-----+-----
.....|.....
.....|.....
.....|.....
.....|.....
.....|.....
```

output

```
aaacc|dxxxa
bbbce|dyyya
ddd#e|dzzza
ccbae|fccbb
cbbaa|ffcdb
-----+---_-
ssrrr|tttdd
saaax_xxeee
yxbbb|zdaaa
yxccc|zdbbb
yxddd|zdccc
```

SOLUTION

Notice that a 5 x 5 field without a scarecrow cannot be completely tiled with sprinklers because the number of cells is not divisible by 3. However, if there is a scarecrow in the field, there are 24 (8 * 3) cells and such field can be completely tiled with sprinklers regardless of the scarecrow's position. This is quite intuitive, and could be proven by executing a simple recursive tiling algorithm to tile the field for every possible scarecrow's position.

This means that it is not difficult to solve a test case where each field has a scarecrow. What should we do with other, empty fields? Obviously, we should drill a hole on the edge of each of them. Drilling a hole enables us to make the number of empty cells in this field divisible by 3: if we put an appropriate tile through the hole, it can reduce the number of empty cells in the field by 1 or 2. Then we can tile the field using the simple recursive tiling. From this observation we derive the algorithm for solving each test case, described in the next paragraph.

It is easy to arrange all fields in a sequence such that any two adjacent fields in the sequence are also neighboring fields in Sara's land (for example, we enumerate the fields spirally). We pass along this sequence, field by field. If a current field has a number of empty cells divisible by 3, we tile it. Otherwise, we drill a hole between this field and the next one, and we put an appropriate tile through this hole, so that the number of empty cells in the current field becomes divisible by 3. Then we tile the current field and move on to the next field (which now contains a piece of the tile). This algorithm will end successfuly because the initial total number of empty cells is divisible by 3 (the input file is solvable).

Implementing all of this is tricky: usually there will remain some situations where the hole and the corresponding tile is chosen in a way that makes the tiling of that particular field impossible (for example, the scarecrow and the tile through the hole might leave one empty cell isolated). Fortunately, since the task is output-only, we can manually fix the fields that our algorithm did not manage to tile.