Problem A. Asynchronous Exceptions

Input file:	*standard input*
Output file:	*standard output*
Time limit:	2 seconds
Memory limit:	256 mebibytes

Your job is to simulate the operation on a machine called ACM (Asynchrounous Calculating Machine) to determine how long each program takes to finish their job. This helps detecting unexpected thread behaviors, like race and deadlock.

ACM has many processing units and can run many threads simultaneously. It runs a global scheduler, which maintains all the threads, assigns a thread on each processing unit and manages all semaphores. Each thread has its own thread id and every thread is one of three states: `RUNNING`, `READY`, and `WAITING`. Global scheduler has one thread queue which contains all the threads in `READY` state.

A program consists of one or more code blocks. Each code block has a list of ACM operations, including the actual computations and thread creation. Each thread runs operations in only one of the given code blocks, and it runs operations from the top of the block to the bottom, unless it receives killThread-signal from other threads.

ACM simulation program simulates 9 operations. Here I describe them all. Please note that word enclosed by '<>' should have some actual name or value.

- `compute <clock>` — spend *clock* clocks for computation. When this operation is interrupted after spending T clocks by other threads or the scheduler and is resumed later, it consumes $clock - T$ clocks.
- `<threadVar> <- forkR <code block>` — generate a native thread running on *code block* and store its id in a thread variable $threadVar$. It can run simultaneously with any threads which are generated in other processing unit. If $threadVar$ is used before in the thread, its previous value is overwritten and lost (this means that we will not be able to refer to the thread although it might run forever).
- `<threadVar> <- forkI <code block>` — generate a virtual thread running on *code block* and store its id in a thread variable $threadVar$. A virtual thread shares some OS resources with its parent thread, so they can't run simultaneously even there are idle processing units. This restriction is applied to any two threads which are directly or indirectly connected by the parent-child relatinship of $forkI$. If $threadVar$ is used before in the thread, its previous value is overwritten and lost.
- `yield` — make the current thread from running state to ready state. The thread goes to the end of the thread queue.
- `killThread <threadVar>` — send a kill signal to the $threadVar$ thread. $threadVar$ guaranteed to be used in either $forkR$ or $forkI$ operations in the same code block before this operation. When a thread receives a kill signal, it immediately cancels its resource requests by lock, changes into ready state if it's in block state, and ends its operation when it's in running state. This operation does not change the semaphore values. If the target thread is already ended, this operation does nothing.
- `lock <semaphore name> <amount>` — request *amount* of *semaphore name*. If the value of *semaphore name* is greater or equal to the *amount*, it just subtracts the *amount* from the semaphore variable. Otherwise, the operation blocks the current thread until the variable is greater or equal to the *amount*. Once the variable gets greater or equal to the *amount*, the thread gets into `READY` state and goes into the thread queue after subtracting *amount* from the variable. If there are more than one thread requesting for the same semaphore, the thread which locked the semaphore first always gets `READY` first. So, the other threads won't get `READY` even if the semaphore variable meets their demands. If there are more than one thread getting `READY`, the thread which locked the semaphore earlier goes into the queue first.
- `unlock <semaphore name> <amount>` — add *amount* to *semaphore name*. Values of semaphores can be larger than initial values. The thread doesn't get blocked at all by this operation.

- `loop <loop count>` — run the code snippet between loop and corresponding `next` command for *loop count* times.
- `next` — this corresponds a loop command in the same code block. The code snippet between `loop` and `next` should be run.
- `end` — end the operation. This command only appears in the end of the code block, and each code block has this operation at the end.

Each parameter should either be a name or a digit. A name is an alphabetic string (case-sensitive) and its length is at most 200. Here are some explanations and constraints:

- *threadVar* is a name. It only refers to the id in the same thread. Each thread has its own value even it has the same name;
- *clock* is a non-negative integer which is at most 1000;
- *semaphore name* is a name;
- *amount* is a non-negative integer which is at most 1000;
- *loop count* is a non-negative integer which is at most 1000;
- total number of lines that all threads evaluate throughout simulation never exceeds 10^5;
- size of input file never exceeds 100K.
- threre is 1 thread queue;
- the id of the n-th created thread is n;
- CPU ids are assigned from 1 to number of CPUs.

Threads are assigned to CPUs when:

- a simulation starts (A thread is assigned to CPU 1; its code block is the first code block of the input);
- there are more than one threads that are in ready state which can be assigned to a CPU (threads are asssinged in the thread queue order; if there are more than one CPUs that can be assinged to the thread, the smallest CPU is selected);
- time step is a multiple of time slice (if the CPU has a thread that is running state, the thread goes to ready state in asscending order of the CPU id; after this, threads are assigned to each CPU in asscending order of the CPU id).

In each step, the simulator does round-robin preemption (when time step is a multiple of time slice), executes operations of each running threads and increments time step. When a compute operation is executed, the thread gets computing time. A thread that has computing time greater than 0 cannot do any operations. After each step, the simulator decrements positive computing time of each running thread.

Operations executed in the same time step is executed in ascending order of the CPU id. In the same time step, a CPU id of an operation is greater or equal to CPU ids of operations which have already been executed. Time step starts from 0.

Input

Input file begins with a line that contains two integers N and T ($1 \leqslant N, T \leqslant 1000$), separated by a space. They mean the number of time steps of the simulation and the maximum number of threads that ACM

is capable, respectively. The next line contains a single integer C ($1 \leqslant C \leqslant 100$), which indicates the number of CPUs available.

The following line has an integer q ($1 \leqslant q \leqslant 1000$) that means time slice of the scheduler.

Description of semaphores follows.

It begins with the number of semaphores S ($0 \leqslant S \leqslant 1000$) which is followed by information of each semaphore, one per line. The information of a semaphore consists of an alphabet string (case-sensitive) and an integer, which specify the name of the semaphore and its initial value, respectively.

Finally code blocks are given. The number of code blocks B ($1 \leqslant B \leqslant 1000$) comes first, and then each code block is described. The first line of the description of each code block is the name of the block (case-sensitive alphabet string), followed by a colon (‘`:`’). Following lines describe content of the code block. A code block is an array of operations described above, and one operation is written in one line. You may assume that every code block always ends with an ‘`end`’ operation.

Output

If the number of living threads exceeds ACM's capacity, put information of all threads which terminated before exceeding the capacity, and then put “`<<oops>>`”. If not, and if there are one or more threads not finished at the end of the simulation, put information of all threads terminated, and then put “`<<loop>>`”. Otherwise (i. e. when all threads terminates within the time), just put information of all threads. All quotes are for clarity.

Information of threads must be written in increasing order of thread ID, one per line. Information of a single thread is denoted by two integers, the thread ID and the time it terminated, separated by a single space character.

Examples

standard input	*standard output*
50 50 1 10 1 semaphore 1 1 codeBlockA: loop 2 compute 10 next end	1 20
50 50 1 10 1 semaphore 1 2 codeBlockA: hoge <- forkR codeBlockB yield compute 1 killThread hoge lock semaphore 1 compute 1 end codeBlockB: compute 1 lock semaphore 2 end	1 3 2 3
5 5 1 3 1 semaphore 1 1 codeBlockA: hoge <- forkI codeBlockA compute 1 end	1 1 2 2 3 4 4 4 5 5 <<loop>>
5 5 1 2 1 semaphore 1 1 codeBlockA: compute 1 hoge <- forkI codeBlockA hoge <- forkI codeBlockA end	1 1 2 3 3 3 <<oops>>

Problem B. AYBABTU

Input file:	*standard input*
Output file:	*standard output*
Time limit:	2 seconds
Memory limit:	256 mebibytes

There is a tree that has n nodes and $n-1$ edges. There are military bases on t out of the n nodes. We want to disconnect the bases as much as possible by destroying k edges. The tree will be split into $k+1$ regions when we destroy k edges. Given the purpose to disconnect the bases, we only consider to split in a way that each of these $k+1$ regions has at least one base. When we destroy an edge, we must pay destroying cost. Find the minimum destroying cost to split the tree.

Input

The first line of input file consists of three integers n, t, and k ($1 \leqslant n \leqslant 10^4$, $1 \leqslant t \leqslant n$, $0 \leqslant k \leqslant t-1$). Each of the next $n-1$ lines consists of three integers representing an edge. The first two integers represent node numbers connected by the edge. A node number is a positive integer less than or equal to n. The last one integer represents destroying cost. Destroying cost is a non-negative integer less than or equal to 10^4. The next t lines contain a distinct list of integers one in each line, and represent the list of nodes with bases.

Output

Print one integer — minimum destroying cost to split the tree.

Examples

standard input	*standard output*
2 2 1 1 2 1 1 2	1
4 3 2 1 2 1 1 3 2 1 4 3 2 3 4	3

Problem C. Billiards Sorting

Input file:	*standard input*
Output file:	*standard output*
Time limit:	6 seconds (10 seconds for Java)
Memory limit:	256 mebibytes

Rotation is one of several popular pocket billiards games. It uses 15 balls numbered from 1 to 15, and set them up as illustrated in the following figure at the beginning of a game. (Note: the ball order is modified from real-world Rotation rules for simplicity of the problem.)

```
        ( 1)
      ( 2)( 3)
    ( 4)( 5)( 6)
  ( 7)( 8)( 9)(10)
(11)(12)(13)(14)(15)
```

You are an engineer developing an automatic billiards machine. For the first step you had to build a machine that sets up the initial condition. This project went well, and finally made up a machine that could arrange the balls in the triangular shape. However, unfortunately, it could not place the balls in the correct order.

So now you are trying to build another machine that fixes the order of balls by swapping them. To cut off the cost, it is only allowed to swap the ball numbered by 1 with its neighboring balls that are not in the same row. For example, in the case below, only the following pairs can be swapped: $(1, 2)$, $(1, 3)$, $(1, 8)$, and $(1, 9)$.

```
        ( 5)
      ( 2)( 3)
    ( 4)( 1)( 6)
  ( 7)( 8)( 9)(10)
(11)(12)(13)(14)(15)
```

Write a program that calculates the minimum number of swaps required.

Input

The first line of the input file has an integer N ($1 \leqslant N \leqslant 5$), which is the number of the rows.

The following N lines describe how the balls are arranged by the first machine; the i-th of them consists of exactly i integers, which are the ball numbers.

Output

Print the minimum number of swaps. You can assume that any arrangement can be fixed by not more than 45 swaps.

Examples

standard input	*standard output*
2 3 2 1	1
4 9 2 4 8 5 3 7 1 6 10	13

Problem D. Digit

Input file:	*standard input*
Output file:	*standard output*
Time limit:	2 seconds
Memory limit:	256 mebibytes

For a positive integer a, let $S(a)$ be the sum of its digits in base l, $S^2(a)$ be the sum of digits of $S(a)$ in base l etc. Also let $L(a)$ be the minimum k such that $S^k(a)$ is less than or equal to $l - 1$. Find the minimum x such that $L(x) = N$ for a given N, and print it modulo m.

Input

Input is given by a line with three integers N, m, l ($0 \leqslant N \leqslant 10^5$, $2 \leqslant m \leqslant 10^9$, $2 \leqslant l \leqslant 10^9$).

Output

Print the minimum x modulo m as described above.

Examples

standard input	*standard output*
0 1000 10	1
1 1000 10	10

Problem E. Dungeon Creation

Input file:	*standard input*
Output file:	*standard output*
Time limit:	2 seconds
Memory limit:	256 mebibytes

The king demon is waiting in his dungeon to defeat a brave man. His dungeon consists of $H \times W$ grids. Each cell is connected to four (i. e. north, south, east and west) neighboring cells and some cells are occupied by obstacles.

To attack the brave man, the king demon created and sent a servant that walks around in the dungeon. However, soon the king demon found that the servant does not work as he wants. The servant is too dumb. If the dungeon had cyclic path, it might keep walking along the cycle forever.

In order to make sure that the servant eventually find the brave man, the king demon decided to eliminate all cycles by building walls between cells. At the same time, he must be careful so that there is at least one path between any two cells not occupied by obstacles.

Your task is to write a program that computes in how many ways the king demon can build walls.

Input

The first line of the input file has two integers H and W ($1 \leqslant H \leqslant 500$, $1 \leqslant W \leqslant 15$), which are the height and the width of the dungeon. Following H lines consist of exactly W letters each of which is '`.`' (there is no obstacle on the cell) or '`#`' (there is an obstacle). You may assume that there is at least one cell that does not have an obstacle.

Output

Print the number of ways that walls can be built in one line. Since the answer can be very big, output it modulo $10^9 + 7$.

Examples

standard input	*standard output*
2 2	4
3 3 ... .#	56

Problem F. Longest Lane

Input file:	*standard input*
Output file:	*standard output*
Time limit:	2 seconds
Memory limit:	256 mebibytes

Mr. KM, the mayor of KM city, decided to build a new elementary school. The site for the school has an awkward polygonal shape, which caused several problems. The most serious problem was that there was not enough space for a short distance racetrack. Your task is to help Mr. KM to calculate the maximum possible length for the racetrack that can be built in the site.

The track can be considered as a straight line segment whose width can be ignored. The boundary of the site has a simple polygonal shape without self-intersection, and the track can touch the boundary.

Note that the boundary might not be convex.

Input

The first line of the input file contains an integer N ($3 \leqslant N \leqslant 100$). Each of the following N lines contains two integers x_i and y_i ($-1000 \leqslant x_i, y_i \leqslant 1000$), which describe the coordinates of a vertex of the polygonal border of the site, in counterclockwise order.

Output

Print the maximum possible length of the track. The answer should be given as a floating point number with an absolute error of at most 10^{-6}.

Examples

standard input	*standard output*
4 0 0 10 0 10 10 0 10	14.142135624
3 0 0 1 0 0 1	1.41421356

Problem G. PLAY in BASIC

Input file:	*standard input*
Output file:	*standard output*
Time limit:	2 seconds
Memory limit:	256 mebibytes

One sunny day, Dick T. Mustang found an ancient personal computer in the closet. He brought it back to his room and turned it on with a sense of nostalgia, seeing a message coming on the screen:

```
READY?
```

It was BASIC — a programming language designed for beginners, and was widely used from 1980's to 1990's. There have been quite a few BASIC dialects. Some of them provide the PLAY statement, which plays music when called with a string of a musical score written in Music Macro Language (MML). Dick found this statement is available on his computer and accepts the following commands in MML:

- **Notes:** `Cn`, `C+n`, `C-n`, `Dn`, `D+n`, `D-n`, `En`, ... ($n = 1, 2, 4, 8, 16, 32, 64, 128 + dots$)

  Each note command consists of a musical note followed by a duration specifier

  Each musical note is one of the seven basic notes: '`C`', '`D`', '`E`', '`F`', '`G`', '`A`', and '`B`'. It can be followed by either '`+`' (indicating a sharp) or '`-`' (a flat). The notes '`C`' through '`B`' form an octave as depicted in the figure below. The octave for each command is determined by the current octave, which is set by the octave commands as described later. It is not possible to play the note '`C-`' of the lowest octave (1) nor the note '`B+`' of the highest octave (8).

  Each duration specifier is basically one of the following numbers: 1, 2, 4, 8, 16, 32, 64, and 128, where 1 denotes a whole note, 2 a half note, 4 a quarter note, 8 an eighth note, and so on. This specifier is *optional*; when omitted, the duration will be the default one set by the '`L`' command (which is described below). In addition, duration specifiers can contain *dots* next to the numbers. A dot adds the half duration of the basic note. For example, '`4.`' denotes the duration of 4 (a quarter) plus 8 (an eighth, i. e. half of a quarter), or as 1.5 times long as 4. It is possible that a single note has more than one dot, where each extra dot extends the duration by half of the previous one. For example, '`4..`' denotes the duration of 4 plus 8 plus 16, '`4...`' denotes the duration of 4 plus 8 plus 16 plus 32, and so on. The duration extended by dots cannot be shorter than that of 128 due to limitation of Dick's computer; therefore neither '`128.`' nor '`32...`' will be accepted. The dots specified without the numbers extend the default duration. For example, '`C.`' is equivalent to '`C4.`' when the default duration is 4. Note that '`C4C8`' and '`C4.`' are *unequivalent*; the former contains two distinct notes, while the latter just one note.

- **Rest:** `Rn` ($n = 1, 2, 4, 8, 16, 32, 64, 128 + dots$)

  The '`R`' command rests for the specified duration. The duration should be specified in the same way as the note commands, and it can be omitted, too. Note that '`R4R8`' and '`R4.`' are *equivalent*, unlike '`C4C8`' and '`C4.`', since the both rest for the same duration of 4 plus 8.

- **Octave:** `On` ($n = 1 \ldots 8$), `<`, `>`

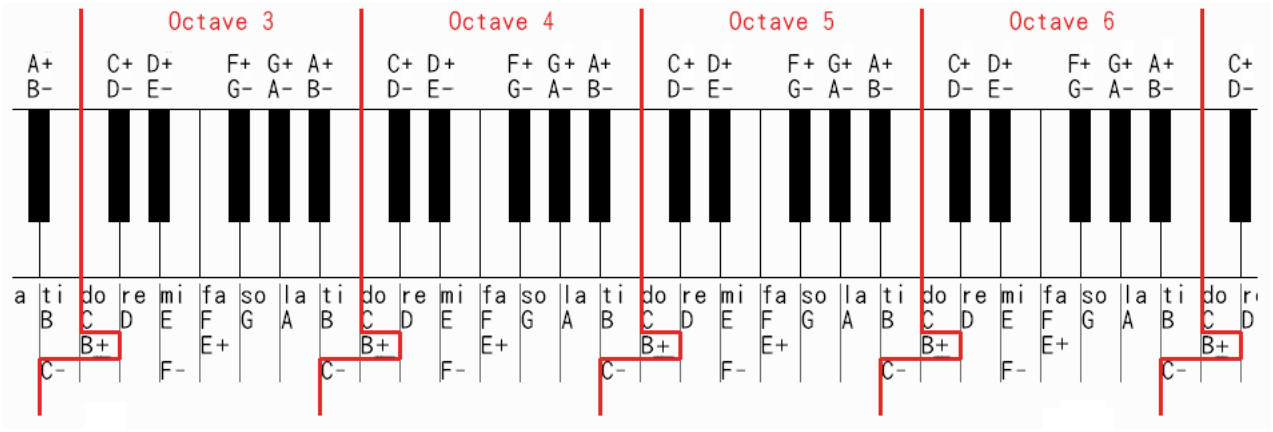
  The '`O`' command sets the current octave to the specified number. '`>`' raises one octave up, and '`<`' drops one down. It is not allowed to set the octave beyond the range from 1 to 8 by these commands. The octave is initially set to 4.

- **Default duration:** `Ln` ($n = 1, 2, 4, 8, 16, 32, 64, 128$)

  The L command sets the default duration. The duration should be specified in the same way as the note commands, but *cannot be omitted nor followed by dots*. The default duration is initially set to 4.

- **Volume:** `Vn` ($n = 1 \ldots 255$)

  The '`V`' command sets the current volume. Larger is louder. The volume is initially set to 100.



As an amateur composer, Dick decided to play his pieces of music by the PLAY statement. He managed to write a program with a long MML sequence, and attempted to run the program to play his music — but unfortunately he encountered an unexpected error: the MML sequence was too long to be handled in his computer's small memory!

Since he didn't want to give up all the efforts he had made to use the PLAY statement, he decided immediately to make the MML sequence shorter so that it fits in the small memory. It was too hard for him, though. So he asked you to write a program that, for each given MML sequence, prints the shortest MML sequence (i. e. MML sequence containing minimum number of characters) that expresses the same music as the given one. Note that the final values of octave, volume, and default duration can be different from the original MML sequence.

Input

The input is given by a single line that contains an MML sequence up to 100 000 characters. All sequences only contain the commands described above. Note that sequence contains at least one note, and there is no rest before the first note and after the last note.

Output

Print the shortest MML sequence, equivalent to a given one. If there are multiple solutions, print any of them.

Examples

standard input	*standard output*
C4C4G4G4A4A4G2F4F4E4E4D4D4C2	CCGGAAG2FFEEDDC2
O4C4.C8F4.F8G8F8E8D8C2	C.C8F.L8FGFEDC2
B-8>C8<B-8V40R2R..R8.V100EV50L1CG	L8B-B+B-RL1RE4V50CG

Problem H. Skyland

Input file:	*standard input*
Output file:	*standard output*
Time limit:	2 seconds
Memory limit:	256 mebibytes

Somewhere in the sky, KM kingdom built n floating islands by their highly developed technology. The islands are numbered from 1 to n.

The king of the country, Kita-masa, can choose any non-negative real number as the altitude for each island, as long as the sum of the altitudes is greater than or equals to H. For floating the island i to the altitude h_i, it costs $b_i \cdot h_i$ Besides, it costs $|h_i - h_j| \cdot c_{i,j}$ for each pair of islands i and j since there are communications between these islands.

Recently, energy prices are rising, so the king Kita-masa, wants to minimize the sum of the costs. The king ordered you, a court programmer, to write a program that finds the altitudes of the floating islands that minimize the cost.

Input

Input starts with a line containing two integers n ($1 \leqslant n \leqslant 100$) and H ($0 \leqslant H \leqslant 1000$), separated by a single space. The next line contains n integers $b_1, b_2, \ldots, b_n$ ($0 \leqslant b_i \leqslant 1000$).

Each of the next n lines contains n integers $c_{i,j}$ ($0 \leqslant c_{i,j} \leqslant 1000$). You may assume $c_{i,i} = 0$ and $c_{i,j} = c_{j,i}$.

Output

Print a line containing a space-separated list of the altitudes of the islands that minimizes the sum of the costs. If there are several possible solutions, print any of them. Your answer will be accepted if the altitude of each island is non-negative, sum of the altitudes is greater than $(1 - 10^{-9}) \cdot H$, and the cost calculated from your answer has an absolute or relative error less than 10^{-9}.

Examples

standard input	*standard output*
2 1 1 3 0 1 1 0	0.75 0.25
3 3 1 2 4 0 2 0 2 0 1 0 1 0	1.5 1.5 0.0

Problem I. Three-way Branch

Input file:	*standard input*
Output file:	*standard output*
Time limit:	2 seconds
Memory limit:	256 mebibytes

There is a grid that consists of $W \times H$ cells. The upper-left-most cell is $(1, 1)$ You are standing on the cell of $(1, 1)$ and you are going to move to cell of (W, H). You can only move to adjacent lower-left, lower or lower-right cells.

There are obstructions on several cells. You can not move to it. You cannot move out the grid, either. Write a program that outputs the number of ways to reach (W, H) modulo $10^9 + 9$. You can assume that there is no obstruction at $(1, 1)$.

Input

The first line of the input file contains three integers, the width W, the height H, and the number of obstructions N ($1 \leqslant W \leqslant 75$, $2 \leqslant H \leqslant 10^{18}$, $0 \leqslant N \leqslant 30$). Each of following N lines contains 2 integers, denoting the position of an obstruction (x_i, y_i) ($1 \leqslant x_i \leqslant W$, $1 \leqslant y_i \leqslant H$).

Output

Print the number of ways to reach (W, H) modulo $10^9 + 9$.

Examples

standard input	*standard output*
2 4 1 2 1	4
2 2 1 2 2	0

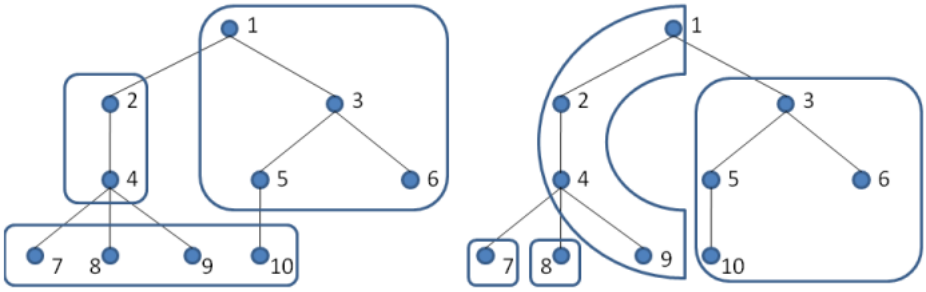
Problem J. Tree Allocation

Input file:	*standard input*
Output file:	*standard output*
Time limit:	2 seconds
Memory limit:	256 mebibytes

A tree is one of the most popular data structures in computer science. Tree itself is very elegant, but it is incompatible with current computer architecture. In the current architecture, memory can be regarded as a one-dimensional array. Since a tree is not a one-dimensional structure, cache efficiency may come into question when we allocate a tree on memory. We can access the data faster if it is contained in cache. Let us consider the allocation of nodes of a tree which achieves high cache performance.

We define the cost of allocation for a tree as follows. For simplicity, we regard memory to be separated to blocks each can store at most B nodes of a tree. When we access data in a block, all data in the block are stored in cache and access request to data in the block will be faster. The cost of an access to node u after node v is 0 if u and v are in same block, and 1 otherwise. Initially, cache has no data and the cost of an access to a node is always 1. The cost of the path v_1, v_2, ..., v_n is sum of the cost when we access the nodes v_1, v_2, ..., v_n in this order. For a tree, we define the cost of allocation as a maximum cost of the paths from root node to each terminal node.

The figures below show examples of allocation when $B = 4$ and node 1 is the root node (it corresponds to the tree described in the third sample input). Each frame represents a block. The left figure is an example of allocation whose cost is 3. It is not optimal, because the right example achieves the optimal allocation cost 2.



Given a tree, for each node i in the tree, calculate the minimum cost of allocation when the node i is the root node.

Input

Input file starts with a line containing two integers N ($1 \leqslant N \leqslant 10^5$) and B ($1 \leqslant B \leqslant N$), separated by a single space. Each of the next $N - 1$ lines contains an integer. The i-th integer p_i ($1 \leqslant p_i \leqslant i$) denotes that the node $i + 1$ and the node p_i are connected. The nodes are numbered from 1 to N.

Output

Print N lines. The i-th line should contain an integer which denotes the minimum cost of allocation of the given tree when node i is the root node.

Examples

standard input	*standard output*
3 1 1 2	3 2 3
3 2 1 1	2 2 2
10 4 1 1 2 3 3 4 4 4 5	2 2 2 2 2 2 2 2 2 3