National Olympiad in Informatics

Finals Round 2

Important! Read the following:

Hidden Test Cases. Your solution will be checked by running it against one or more (usually several) hidden test cases. You will not have access to these cases, but a correct solution is expected to handle them correctly.

Strict Output Format. The output checker is **strict**. Follow these guidelines strictly:

- It is **space sensitive**. Do not output extra leading or trailing spaces. Do not output extra blank lines unless explicitly stated.
- It is **case sensitive**. So, for example, if the problem asks for the output in lowercase, follow it.
- Do not print any tabs. (No tabs will be required in the output.)
- Do not output anything else aside from what's asked for in the Output section. So, do not print things like "`Please enter t`".

Not following the output format strictly and exactly will likely result in the verdict "*Output isn't correct*".

Use Standard I/O. Do not read from, or write to, a file. You must read from the standard input and write to the standard output.

Submit Code Only. Only include **one** file when submitting: the source code (.cpp, .py, etc.) and nothing else.

No Java Package. For Java submissions, do not include a `package` line.

No Weird Filenames. Only use letters, digits and underscores in your filename. Do not use spaces or other special symbols.

Use Fast I/O. Many problems have large input file sizes, so use fast I/O. For example:

- In C/C++, use `scanf` and `printf`.
- In Python, use `sys.stdin.readline()`

Flush On Interactive Problems. On interactive problems, make sure to **flush** your output stream after printing.

- In C++, use `fflush(stdout);` or `cout << endl;`
- In Python, use `sys.stdout.flush()` or `print(flush=True)`
- For more details, including for other languages, ask a question/clarification through CMS.

Good luck and enjoy the contest! 😃

Contents

Notes

- Many problems have large input file sizes, so use fast I/O. For example:
  - In C/C++, use `scanf` and `printf`.
  - In Python, use `sys.stdin.readline()`
- On interactive problems, make sure to **flush** your output stream after printing.
  - In C++, use `fflush(stdout);` or `cout << endl;`
  - In Python, use `sys.stdout.flush()` or `print(flush=True)`
  - For more details, including for other languages, ask a question/clarification through CMS.

Good luck and enjoy the problems!

Problem A
Incompetent Interactor

Alice has a great idea for an interactive problem for the NOI.PH 2025 Finals!

It goes like this:

Alice is thinking of a secret integer A, from 1 to m (inclusive). Bob's job is to guess what it is.

Bob can ask her n questions, each of the following form:

- "`Is A {cmp} {x} ?`" (without the quotes)

where `x` is some positive integer, and `cmp` is one of

`<`	`<=`	`==`
`>`	`>=`	`!=`

with their usual familiar meanings. Alice will respond "Yes" or "No" to each one.

For example, suppose $m = 10$, and Bob asks the $n = 2$ questions

- `Is A > 6 ?`
- `Is A < 9 ?`

If Alice had chosen $A = 7$, then she should answer "Yes" and then "Yes".

Alice and Bob try playing this game for real. Bob proceeds to ask Alice his n questions. The issue is that Alice had forgotten her initial number A. Oops! So, she just answers Bob's questions randomly. Bad Alice!

Oh well, as long as her answers are consistent with *some* integer from 1 to m, Bob can't call her out on it. If she ends up giving contradictory answers, though, Bob will be furious!

There are 2^n ways that Alice can choose to answer Bob's n Yes/No questions—call one such way a *response set*.

The **narrowness** of a response set is equal to the number of integers from 1 to m such that if that were the value of A, then this response set would be how Alice should have answered Bob's questions. Then, a response set is called **valid** if its narrowness is nonzero.

In the example from earlier, the narrowness of the response set (Yes, Yes) is 2, since Alice should answer "Yes" and then "Yes" if she had chosen $A = 7$, and also if she had chosen $A = 8$. Since $2 \neq 0$, this response set is valid.

Please answer these two questions:

- How many of these 2^n response sets are valid?
- What is the *sum* of the narrownesses across all 2^n response sets?

Output each answer modulo $10^9 + 7$.

Input Format

The first line of input contains the two space-separated integers n and m.

Then, n lines follow, the ith of which contains Bob's ith question, in the format described above.

Output Format

Output two space-separated integers: the number of valid response sets, and the sum of the narrownesses across all response sets (each modulo $10^9 + 7$).

Constraints

For all subtasks
$1 \leq n \leq 10^5$ $1 \leq m \leq 10^9$ $1 \leq x \leq 10^9$ in each of Bob's questions.

Subtask	Points	Constraints
1	**11**	All questions use `<`
2	**17**	There are no `==` or `!=` questions.
3	**13**	$m \leq 100$ $n \leq 15$
4	**19**	$m \leq 2 \times 10^5$ $n \leq 15$
5	**19**	$n \leq 15$
6	**21**	No further constraints.

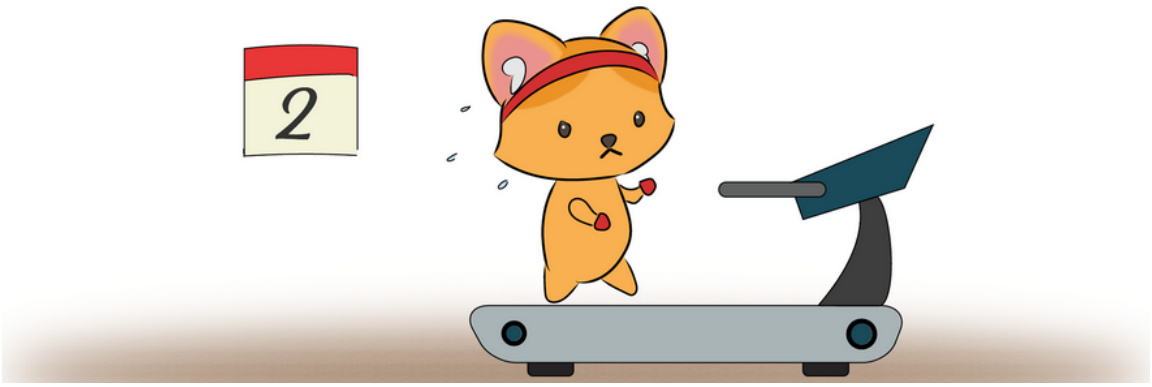
Sample I/O

Input 1	Output 1
4 10 Is A <= 7 ? Is A >= 2 ? Is A < 13 ? Is A != 5 ?	4 10

Input 2	Output 2
6 144 Is A <= 1 ? Is A >= 2 ? Is A < 3 ? Is A > 5 ? Is A == 8 ? Is A != 13 ?	6 144

Input 3	Output 3
5 12 Is A == 1 ? Is A == 2 ? Is A == 4 ? Is A == 8 ? Is A == 16 ?	5 12

Problem B
The Astounding Race



Abby is working hard to live an Active Lifestyle, so that she might lower her Body Mass Index. Why? Well you see...

We are proud to announce our new contest, the National Olympiad in Informatics and PHysicaleducation - PHilippines (NOI.PH.PH). Rather than being confined to one seat for five hours, participants in the NOI.PH.PH must traverse perilous terrain and complete various physical challenges, in between the coding tasks!

This innovation is sure to make comp prog more exciting as a spectator sport!

NOI.PH.PH will involve n problems, which we label from 1 to n. We will also set up n *workstations*, arranged in a row and labeled 1 to n from left to right.

Obstacle courses have been set up between workstations such that traveling between adjacent workstations (that is, between workstations i and $i+1$, for $1 \leq i < n$) always takes *exactly one minute* in either direction. Contestants may *not* move between workstations by any means other than through the obstacle courses.

Contestants are *forced* to answer the problems in the order $1, 2, 3, \ldots, n$—that is, you may not initiate problem i until all of problems 1 through $i-1$ have already been answered. Also, you aren't allowed to just answer any problem anywhere. For each i from 1 to n, problem i was assigned a distinct workstation w_i, meaning you can only solve problem i when you're at workstation w_i.

Your time starts when you start the first problem, and your time ends when you finish the last problem. A contestant's *physical time penalty* is computed by totalling the amount of time they spent (in minutes) moving between workstations.

The minimum possible physical time penalty depends on how the judges decide to assign the workstations to the problems.

For example, suppose $n = 5$ and $w = [3, 1, 4, 5, 2]$. Then, the minimum physical time penalty would be 9:

- We can take our sweet time getting to workstation $w_1 = 3$.
- After that, we spend 2 minutes moving to workstation $w_2 = 1$.
- After that, we spend 3 minutes moving to workstation $w_3 = 4$.
- After that, we spend 1 minute moving to workstation $w_4 = 5$.
- Finally, we spend 3 minutes moving to workstation $w_5 = 2$.
- Once we finish the last problem, our time ends.

Since $2 + 3 + 1 + 3 = 9$, that is the minimum possible physical time penalty.

Given n and k, determine if there exist workstation assignments $w_1, w_2, \ldots, w_n$ which make it so that the minimum physical time penalty is exactly equal to k. If yes, we would *prefer* if you are also able to actually construct such an assignment. If not, but your Yes/No answers were all correct, then you can still earn partial points.

Also, there will be T test cases.

Input Format

The first line of input contains a single integer T, the number of test cases. The descriptions of the T test cases follow.

Each test case is described by a line containing the two space-separated integers n and k for that test case.

Output Format

For each test case:

- Output a line containing either `YES` or `NO`, depending on if the task is possible.
- If `YES`, also output a line containing n space-separated integers, your proposed values for $w_1, w_2, w_3, \ldots, w_n$, which should be some permutation of the integers from 1 to n.

Note that if you want the partial points, you **must** output a permutation of 1 to n after every `YES` response, even if it's incorrect.

Constraints

Let N be the sum of n across all test cases.

For all subtasks
$1 \leq T \leq 10^5$ $2 \leq n \leq 5 \times 10^5$ $0 \leq k \leq 10^{18}$ $N \leq 10^6$

Subtask	Points	Constraints
1	**12**	$n \leq 9$
2	**19**	$n \leq 15$
3	**22**	$k \leq 2n$
4	**35**	$N \leq 5000$
5	**12**	No further constraints.

There is also partial scoring:

- You get 0 points for this subtask if there exists a test file under this subtask for which you gave an incorrect `YES` or `NO` answer in some test case.
- If not, you get 40% of the points for this subtask if there exists a test file under this subtask such that: all your `YES` and `NO` answers were valid, but there exists some `YES` test case where the provided assignments would not yield a minimum physical time penalty of k.
- Otherwise, you get 100% of the points for this subtask.

Sample I/O

Input 1	Output 1
3 12 38 7 14 2 1000	YES 12 10 9 11 6 1 8 5 2 7 3 4 YES 3 1 2 7 4 6 5 NO

Problem C
Fast Threerier Transform

Hello everyone, welcome back to another episode of 2Blue2Brown - Tokyo Drift. As you recall, in last week's episode, we learned about how a fourier transform is a process which allows us to decompose an arbitrary signal into square waves (because squares have four *sides)*[citation needed].

This week, we're going to learn about a natural extension, the threerier *transform, which allows us to decompose a signal into* triangle waves.

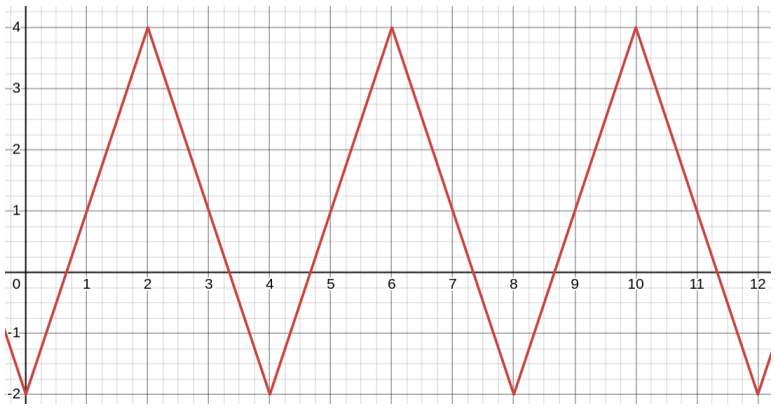
A **triangle wave** is a real-valued function described by three real-number values:

- b, the y-intercept
- m, the slope, which **must be nonzero**
- p, the half-period, which **must be positive**

It is a piecewise-linear function, and we define the triangle wave given by parameters (b, m, p) to be the unique *continuous* function T such that:

- $T(0) = b$
- If $0 \le x < p$, then $T(x)$ is a linear function[1] with a slope of m.
- If $p \le x < 2p$, then $T(x)$ is a linear function with a slope of $-m$.
- $T(x) = T(x - 2p)$ for all $x \in \mathbb{R}$.

Informally, it's a function whose plot has a shape that looks like this (here, $b = -2$, $m = 3$, and $p = 2$):



Note that the reason we disallowed $m = 0$ is because we wouldn't get these nice triangle shapes... we'd just get a line.

[1] Linear as in "functions whose plot is a line", like $y = mx + b$—nothing to do with the linear transformations we covered in our *Essence of Linear Algebra* series

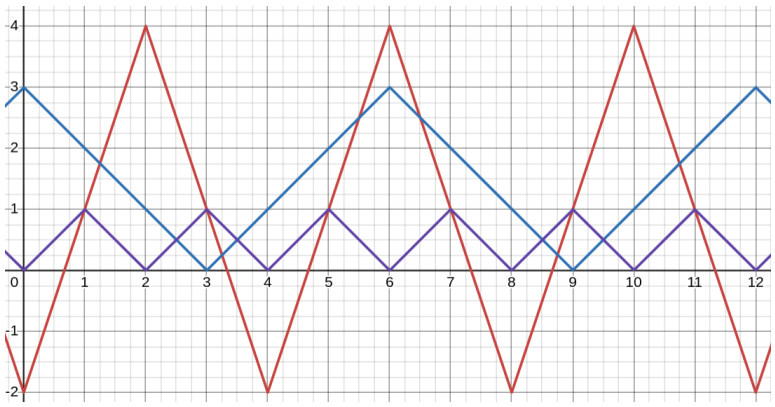
Your exercise for today is to reinvent the process known as the *discrete threerier transform*. Suppose we have n arbitrary data points, call them $a[0], a[1], a[2], \ldots, a[n-1]$, representing an arbitrary signal.

What we need to do today is determine whether there exists a **non-empty** collection of triangle waves $T_1, T_2, \ldots, T_k$ such that:

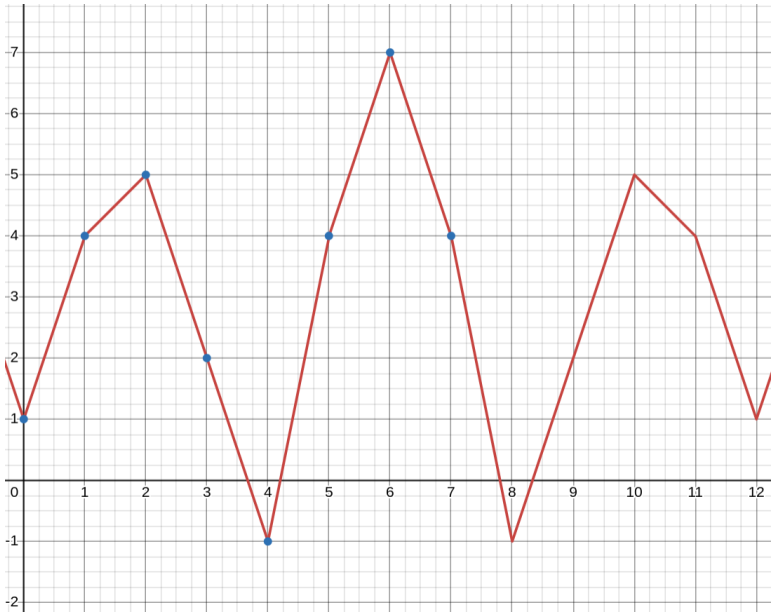
- $a[j] = T_1(j) + T_2(j) + \cdots + T_k(j)$, for all j from 0 to $n-1$.
- If T_i is the triangle wave described by the triple of integers (b_i, m_i, p_i), then b_i and m_i and p_i are all *integers*, for all i from 1 to k.

And, if a solution exists, you need to provide one of minimal size (i.e. the positive integer k, the number of triangle waves in the collection, must be as small as possible).

For example, if $a = [1, 4, 5, 2, -1, 4, 7, 4]$, we can decompose it into three triangle waves described by parameters $(-2, 3, 2)$, $(3, -1, 3)$, and $0, 1, 1$, illustrated below.



Summing up these three triangle waves gives us the following function, and we can see that the values it attains at inputs $x = 0, 1, 2, \ldots, n-1$ match the desired values from our given signal.



Input Format

The first line of input contains the single integer n.

The second line contains the n space-separated integers $a[0], a[1], \ldots, a[n-1]$.

Output Format

If a discrete threerier transform does not exist for the given input, output a line containing the integer -1.

If one does exist, instead output a line containing a positive integer k, the size of the smallest non-empty collection of triangle waves which satisfies the conditions described above.

Then, output k lines, each containing three space-separated integers, describing the triangle waves in the collection. The ith line should contain the three parameters b_i, m_i, and p_i. Each of these values should be between -10^{18} and 10^{18}, and also $m_i \neq 0$ and $p_i > 0$ must hold. It can be shown that if a solution exists, then one exists with these constraints.

If multiple answers exist, any will be accepted.

Constraints

For all subtasks
$1 \leq n \leq 2 \times 10^5$ $\lvert a[i] \rvert \leq 998244353$ for all i.

Subtask	Points	Constraints
1	**15**	If an answer exists, it uses at most 1 triangle wave.
2	**25**	If an answer exists, it uses at most 2 triangle waves.
3	**11**	$\lvert a[i] \rvert \leq 5$ for all i
4	**25**	$n \leq 2000$
5	**24**	No further constraints.

Sample I/O

Input 1	Output 1
8 1 4 5 2 -1 4 7 4	3 -2 3 2 3 -1 3 0 1 1

Problem D
CJ's Marble Runs

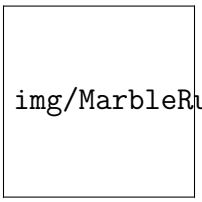
CJ noticed that the "simulation" genre of game is popping off on Steam right now, turning otherwise banal premises into highly addictive and satisfying gameplay. He has a great idea for a simulation game, he just hopes you won't think he's lost his marbles.

CJ's Marble Runs *is a brand new simulation game, all about setting up elaborate marble networks and then wasting hours watching marbles whizz by in your grand creations. The fun thing about video games though... is that they're not constrained by the limitations of reality.*

A **marble network** is a set of nodes and pipes. Marbles go between nodes using pipes.

Let's examine the types of nodes that a marble network might have.

- n **router nodes** numbered 1, 2, ..., n;
- one or more **input nodes** labeled a, b, ...;
- one or more **output nodes** labeled A, B, ...

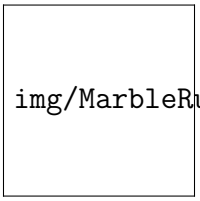


A marble network also has m pipes, labeled 1, 2, ..., m. Each pipe has a **source node** and a **target node**. We can draw this with an arrow going from the source node to the target node.

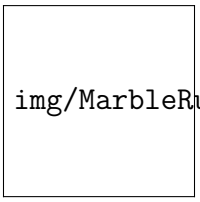
Marbles are placed into our network from the input nodes, and cause output to be printed when they enter an output node. Router nodes are just additional extra nodes that our marbles might pass through along the way.

Whenever a marble is sent to some node, a marble then enters *each pipe* that has that node as a source node. Whenever a marble enters a pipe, the pipe *may* send a marble to its target node, depending on the kind of pipe.

There are three kinds of pipes in CJ's Marble Runs. Let's familiarize ourselves with all of them!

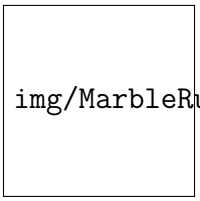


Repeaters, denoted by `-` characters: For very marble sent to the source node, the repeater sends a marble to the target node.



Dividers, denoted by `/` characters: For every *other* marble send to the source node, the divider sends a marble to the target node. So...

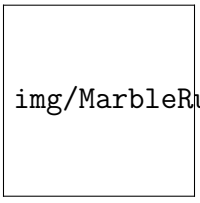
- the first marble sent to the source does nothing,
- the second marble sent to the source sends a marble to the target,
- the third marble sent to the source does nothing,
- the fourth marble sent to the source sends a marble to the target, etc.



Sinks, denoted by `X` characters: For *only the first* marble sent to the source node does the sink send a marble to the target node. So...

- the first marble sent to the source sends a marble to the target,
- the second marble sent to the source does nothing,
- the third marble sent to the source does nothing, etc.

Putting everything together, here's one possible example of a marble network:



Recall that the pipes are *labeled* from 1 to m, as indicated in the above completed network. Why? Well...

One by one, we will send marbles into input nodes and record the results as the simulation plays out. We write the label of an input node whenever we send a marble to it, and write the label of an output node when a marble is sent to it. We always wait for the current marble run to finish completely before manually sending another marble.

The pipes' labels act as a "tie-breaker" to unambiguously resolve what would happen if multiple marbles enter different output nodes "at the same time", where the pipes with smaller labels trigger first.

Note that input nodes are not disallowed from being target nodes of some pipe, but we only write an input node's label whenever *we* manually send a marble to it, **never** when a pipe sends a marble to it.

So, a sequence of input nodes (meaning we sent marbles into those input nodes in that order) defines a **simulation string**, the string that would be produced by our marble network.

You can verify that in the network illustrated above, the sequence $[a, a, a, a]$ (meaning we send a marble to input node a four times) would produce the simulation string `aBaACaaA` (color only added for illustration).

- We send a marble to a, which sends a marble to repeater pipe 1.
  - This sends a marble to node 1, which sends marbles to divider pipe 2 and sink pipe 3.
  - Divider pipe 2 does not send out a marble (it only does every *other* time). Sink pipe 3 sends a marble to node B.
- We send a marble to a, which sends a marble to repeater pipe 1.
  - This sends a marble to node 1, which sends marbles to divider pipe 2 and sink pipe 3.
  - Divider pipe 2 now *does* send a marble to node A, which sends a marble to sink pipe 4. Sink pipe 3 does not send out a marble anymore (it's been "used up").
  - Sink pipe 4 sends a marble to node C.
- We send a marble to a, which sends a marble to repeater pipe 1.
  - This sends a marble to node 1, which sends marbles to divider pipe 2 and sink pipe 3.
  - Neither divider pipe 2 nor sink pipe 2 send out a marble.
- We send a marble to a, which sends a marble to repeater pipe 1.
  - This sends a marble to node 1, which sends marbles to divider pipe 2 and sink pipe 3.

- Divider pipe 2 now *does* send a marble to node A, which sends a marble to sink pipe 4. Sink pipe 3 does not send out a marble anymore.
- Sink pipe 4 does not send out a marble anymore (it's been "used up").

Strictly formally speaking (i.e. how you might actually implement this in code, or in a video game), nothing happens "at the same time". Instead, we would implement an *event queue*, and have a source node send marbles to all its pipes by enqueuing them into the event queue in order of smallest label first. In an appendix below, we describe this idea in more detail, and show the example simulation from earlier in terms of this event queue.

Here's the puzzle for this task. Given a simulation string (and the number of input and output nodes, as mandated by that simulation string), construct *any* marble network that would produce it. It is guaranteed that this task will be possible for all simulation strings given as input.

You may use as many router nodes and pipes as you like (within reason). The fewer pipes you use, the higher your score will be!

Input Format

The first line of input contains three space-separated integers: the number of input nodes, the number of output nodes, and the length of the simulation string.

The second line of input contains the simulation string.

Output Format

First, output two space-separated integers n and m: the number of router nodes in your solution, and the number of pipes.

Then, output m lines, each in the form of three space-separated tokens,

```
<source_node> <pipe_kind> <target_node>
```

where each node is a valid node in the network (a letter of an existing input/output node, or an integer from 1 to n), and `pipe_kind` is one of `-` or `/` or `X` for a repeater, divider, or sink pipe, respectively.

These labels will be labeled 1 to m in the order you give them in the output. If there are multiple answers, any will be accepted within the following constraints:

- You may only use at most 10^4 router nodes.
- You may only use at most 10^5 pipes.
- There must only be at most 10^3 pipes in the event queue at any given time (see appendix for a formal description of the event queue).
- The pipes "act" (triggered by receiving a marble, whether or not it will then send one out) no more than 10^6 times in total.

Constraints

The test cases are publicly available to you, included as the attachment `marble-network.zip` to this problem.

Each test case is worth up to 10 points, where your score is determined by the following formula.

If your program does not produce the indicated simulation string, you get 0 points. Otherwise, let m be the number of pipes in your solution for that test case, and let M be the number of pipes in the judge's best solution. The scoring formula is as follows.

$$\text{score}(m, M) = \begin{cases} 0 & \text{if } 10^5 < m \\ 3 + 5 \cdot \left(1 - \frac{\log_{3M}(m-M)-1}{\log_{3M}(10^5-M)-1}\right)^3 & \text{if } 4M < m \leq 10^5 \\ 8 + \frac{1.9}{3} \cdot \left(4 - \frac{m}{M}\right) & \text{if } M < m \leq 4M \\ 10 & \text{if } m \leq M \end{cases}$$

Sample I/O

Input 1	Output 1
1 3 8 aBaACaaA	1 4 a - 1 1 / A 1 X B A X C

Formalization of Event Queue

Formally, we consider the pipes to act in a queue, which initially starts empty.

- Whenever a marble enters a pipe, it is appended to the queue.
- Then, the first pipe in the queue possibly sends a marble to its target node, before being removed from the queue.
- Nodes always act instantly, and send their marbles to pipes in order from smallest to largest index.

Marbles can only be sent to an input node if the queue is empty, i.e. you can't send another marble into an input node until the current marble run has finished resolving.

Suppose we consider the sequence of input nodes $[a, a, a, a]$. Then, the simulation string produced by that sequence using the above illustrated marble network would be `aBaACaaA`, describing the following simulation:

- We send a marble to a, which sends a marble to pipe 1. The queue is $[1]$.
  - First in the queue is pipe 1, a repeater. It sends a marble to node 1, and marbles enter pipes 2 and 3. The queue is $[2, 3]$.
  - Next in the queue is pipe 2, a divider. It does nothing. The queue is $[3]$.
  - Next in the queue is pipe 3, a sink. It sends a marble to node B, which is source node to no pipe.
  - The queue is empty, so we may send another marble.
- We send a marble to a. The queue is $[1]$.
  - First in the queue is pipe 1, a repeater. It sends a marble to node 1, and marbles enter pipes 2 and 3. The queue is $[2, 3]$.
  - Next in the queue is pipe 2, a divider. It sends a marble to node A, and a marble enters pipe 4. The queue is $[3, 4]$.
  - Next in the queue is pipe 3, a sink. It does nothing. The queue is $[4]$.
  - Next in the queue is pipe 4, a sink. It sends a marble to node C, which is source node to no pipe.
  - The queue is empty, so we may send another marble.
- We send a marble to a. The queue is 1.
  - First in the queue is pipe 1, a repeater. It sends a marble to node 1, and marbles enter pipes 2 and 3. The queue is $[2, 3]$.
  - Next in the queue is pipe 2, a divider. It does nothing. The queue is $[3]$.

- Next in the queue is pipe 3, a sink. It does nothing.
- The queue is empty, so we may send another marble.

- We send a marble to a. The queue is 1.
  - First in the queue is pipe 1, a repeater. It sends a marble to node 1, and marbles enter pipes 2 and 3. The queue is $[2, 3]$.
  - Next in the queue is pipe 2, a divider. It sends a marble to node A, and a marble enters pipe 4. The queue is $[3, 4]$.
  - Next in the queue is pipe 3, a sink. It does nothing.
  - Next in the queue is pipe 4, a sink. It does nothing.
  - The queue is empty, so we may send another marble.