CEOI 2026

Practice session

(English)

Beautiful Subsets

Time limit: 8 s *Memory limit: 768 MiB*

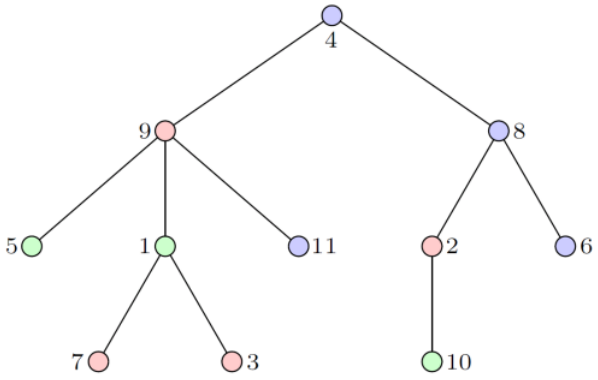
We are given a tree with n vertices, numbered from 1 to n. Each vertex is colored with one of b possible colors (the colors are represented by the integers from 1 to b).

We say that a subset of vertices $A \subseteq \{1, 2, \ldots, n\}$ is **beautiful** if:

- all vertices in A have the same color, and
- there exists a simple path in the tree (that visits no vertex more than once) that visits all vertices in A (it may also visit some vertices that are not in A).

Write a program that, for each color, finds the size of the largest beautiful subset consisting of vertices of that color.

A tree with $n = 11$ vertices and $b = 3$ colors that corresponds to the example input:



Input

The first line contains two integers, n (the number of vertices) and b (the number of colors), separated by a space. The remainder of the input depends on the value of n:

- If $n \leq 200\,000$ (which holds for the first three subtasks), the first line (the one containing n and b) is followed by n more lines. The i-th of these lines contains two integers, p_i and c_i, indicating that vertex i has color c_i and parent p_i. If i is the root of the tree and therefore has no parent, then $p_i = 0$. (For example, in the figure above, vertex 9 is the parent of vertices 1, 5, and 11, while vertex 4 is the root of the tree.)
- If $n > 200\,000$ (which occurs only in the fourth subtask), the first line is followed by a single additional line containing seven numbers separated by spaces: A, B, M, K, A', B', and M'. These are integers between 1 and 10^9. They define two sequences: $r_0 = 0$, $r_{i+1} = (A \cdot r_i + B) \bmod M$ and $r'_0 = 0$, $r'_{i+1} = (A' \cdot r'_i + B') \bmod M'$.
  Such an input represents a tree with the following values. Vertex 1 is the root (so $p_1 = 0$). For each $i = 2, \ldots, n$, vertex i has parent $p_i = i - 1 - \big(r_i \bmod \min(K, i-1)\big)$. For each $i = 1, \ldots, n$, vertex i has color $c_i = 1 + (r'_i \bmod b)$. (The purpose of these formulas is to allow us to provide a large, approximately random tree without requiring you to read an enormous amount of input data.)

Constraints

- $1 \leq b \leq n \leq 5\,000\,000$.
- For every i, we have $1 \leq p_i \leq n$ (except for the root, where $p_i = 0$) and $1 \leq c_i \leq b$.
- The parent data p_i is such that the edges between vertices and their parents indeed form a tree.

Subtasks

- Subtask 1 (25 points): $n \leq 20$.
- Subtask 2 (25 points): $n \leq 1000$.
- Subtask 3 (30 points): $n \leq 200\,000$.
- Subtask 4 (20 points): No additional constraints.

Output

Let ℓ_i denote the size of the largest beautiful subset containing vertices of color i. The required output of your program depends on the value of n:

- If $n \leq 200\,000$, print b lines. On the i-th line (for each $i = 1, \ldots, b$), print the value ℓ_i (i.e., the size of the largest such beautiful subset containing vertices of color i).
- If $n > 200\,000$, print a single line containing the sum $\ell_1 + \ell_2 + \cdots + \ell_b$. (The purpose of this requirement is to avoid producing a huge amount of output for large test cases.)

Examples

Input	Output
11 3 9 2 8 1 1 1 0 3 9 2 8 3 1 1 4 3 4 1 2 2 9 3	3 2 4

Input	Output
200001 123 7 17 11 2 19 3 13	120010

Comment

The first example corresponds to the figure above. For color 1, for instance, there exists a beautiful subset $\{2, 7, 9\}$, which can be visited by the path $2 \to 8 \to 4 \to 9 \to 1 \to 7$.

Cave

Time limit: 4 s *Memory limit: 256 MiB*

Cave explorers have discovered a long, low cave with many mineral deposit formations called stalagmites (growing from the ground up) and stalactites (hanging from the ceiling down). The ground and the ceiling of the cave are parallel. The cave has height v meters and length n meters, and at each meter of the cave either a stalagmite grows from the ground or a stalactite hangs from the ceiling. For each formation, we know its type (stalagmite or stalactite) and its size k_i ($1 \leq k_i < v$).

In the cave, they want to install a tourist railway that will run parallel to the ground and the ceiling at some integer height y ($1 \leq y \leq v$). Since stalagmites with size $k_i \geq y$ and stalactites with size $k_i > v - y$ would obstruct the railway, they will need to be removed.

Write a program that finds the heights y of the railway such that the minimum number of formations (stalagmites and stalactites) would need to be removed for constructing the railway at that height.

Input

In the first line, two integers v and n are given, separated by a space.

The second line contains a string of length n consisting of the characters `'M'` or `'T'`, where the i-th character in the string represents the type of the formation at the i-th meter of the cave. If it is `'M'`, it is a stalagmite growing from the ground; if it is `'T'`, it is a stalactite hanging from the ceiling.

The third line contains n space-separated integers k_i, where the i-th number represents the size of the i-th formation.

Constraints

- $1 \leq v \leq 10^{18}$
- $1 \leq n \leq 10^5$

Subtasks

- Subtask 1 (20 points): $n \leq 1000$ and $v \leq 1000$.
- Subtask 2 (40 points): $v \leq 10^6$.
- Subtask 3 (40 points): No additional constraints.

Output

Output the minimum number of formations that need to be removed, and the number of railway heights at which this minimum can be achieved. Print both values on the same line, separated by a single space.

Example

Input

```
8 9
TTMTMMTTM
2 1 6 5 2 5 3 7 2
```

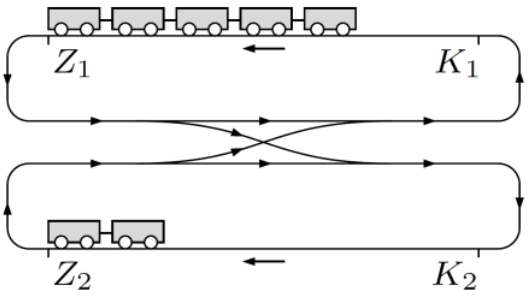
Output

```
3 1
```

Wagon Sorting

Time limit: 5 s *Memory limit: 256 MiB*

We have two sequences (or *queues*, to use a term familiar from computer science) of railroad wagons, one starting at Z_1 and ending at K_1 and the other starting at Z_2 and ending at K_2, as shown by the figure below. The arrows indicate the direction into which the wagons can move.



Each wagon has a unique numerical value, but we cannot access these values directly. We can only deal with the wagons using a very limited set of operations: we can check whether a queue is empty; we can move a wagon from the start of one queue to the end of another queue, or even of the same queue; and we can compare the values of the wagons at the start of both queues (and find out which of them has the smaller value of the two).

Initially the first queue contains some unknown number ($n > 0$) of wagons in an unknown order, while the second queue is empty. Our task is to use the aforementioned operations so that at the end, all wagons will again be in the first queue, but this time sorted in increasing order of their values.

Task

This is an interactive task where your program will work with the wagons by calling the following functions from a library provided by the organizers:

- **void** *BeginSorting*() — you must call this function at the beginning, before calling any other functions from the library.
- **bool** *IsEmpty*(**int** q) — returns a logical value which indicates whether the queue q is empty or not. The value q must be either 1 or 2.
- **void** *Move*(**int** q, **int** r) — removes the first wagon from the start of the queue q and transfers it to the end of the queue r. The values q and r must be from $\{1, 2\}$ and they are also allowed to be equal to each other. Both queues are large enough to accommodate all wagons, so there is never any question of the destination queue r being too full or anything of that sort.
- **int** *Compare*() — compares the values of the wagons at the start of both queues, finds out which of them has the smaller value (note that no two wagons have the same value), and returns the number of the queue containing it. Hence this function always returns 1 or 2.

- **void** *DoneSorting*() — you must call this function after you have finished sorting the wagons; after this you must not make any more calls to any functions from this library.

Your program must make exactly one call to *BeginSorting*, then 0 or more calls to *IsEmpty*, *Move*, and *Compare*, and finally exactly one call to *DoneSorting*.

If your program doesn't call these functions in the correct order, or if it passes values other than 1 and 2 as the arguments q and r, or if it calls *Move*(q, r) when the queue q is empty, or if it calls *Compare* when at least one of the queues is empty, the library will terminate your program and return a run-time-error (RTE) verdict for the current test case. If your program uses the library correctly but the wagons are not sorted correctly at the time of its call to *DoneSorting*, this will result in a wrong-answer (WA) verdict for the current test case.

To access the library, your program should include the header file `wagonslib.h`:

```
#include "wagonslib.h"
```

You can download this header file here: `wagonslib.h`.

To help you in developing your solution, a simple implementation of the library is available here: `wagonslib-public.cpp`. To compile it together with your program, simply add its name as a parameter to the compiler, e.g.:

```
g++ foo.cpp wagonslib-public.cpp
```

if `foo.cpp` is the name of the file containing your solution.

On the evaluation server, a different implementation of the library will be used, so you should not make any assumptions about how exactly the implementation works. You may, however, assume that it does not pollute the global namespace with any declarations other than *BeginSorting*, *IsEmpty*, *Move*, *Compare*, and *DoneSorting*.

Your code must not read from the standard input or write to the standard output, because those will be used by our implementation of the library on the evaluation server to communicate with the rest of the evaluation environment.

Constraints

- The number of wagons n is at least 1 and at most 10 000.

Subtasks

- Subtask 1 (30 points): $30 \leq n \leq 100$.
- Subtask 2 (35 points): $800 \leq n \leq 1000$.
- Subtask 3 (35 points): $8000 \leq n \leq 10\,000$.

Scoring

On a particular run of your program, let n be the number of wagons, and let m be the total number of calls your program made to the functions *IsEmpty*, *Move*, and *Compare*; then the performance of your program on this run is evaluated by the value of $f_n(m)$, where f_n is a piecewise linear function defined as follows:

m	$f_n(m)$
$\leq 2n\lceil \log_2 n \rceil$	1
$3n\lceil \log_2 n \rceil$	0.8
$\geq n^2$	0.3

In each of the ranges $[2n\lceil \log_2 n \rceil, 3n\lceil \log_2 n \rceil]$ and $[3n\lceil \log_2 n \rceil, n^2]$, the function $f_n(m)$ is a linear function in m.

In each subtask, your program will be run several times (on different initial sequences of wagons); the number of points it will be awarded is defined as the total number of points for that subtask multiplied by the smallest value of $f_n(m)$ that your program will have achieved over all the runs in that subtask.

If your program doesn't sort the wagons correctly or doesn't adhere to the aforementioned protocol in using the library, it will get 0 points for the whole subtask.

Example

Call	Return value
BeginSorting()	
Move(1, 2)	
Compare()	2
Move(1, 2)	
IsEmpty(1)	true
Move(2, 1)	
Move(2, 1)	
DoneSorting()	

Comment

If the sequence of calls and return values took place as shown above, the program will have sorted the two wagons correctly; but note that it has behaved in a very risky fashion: if there had been just one wagon, the call to *Compare* would have caused a runtime error!