E. Coprocessor

time limit per test: 1.5 seconds
memory limit per test: 256 megabytes
input: standard input
output: standard output

You are given a program you want to execute as a set of tasks organized in a dependency graph. The dependency graph is a directed acyclic graph: each task can depend on results of one or several other tasks, and there are no directed circular dependencies between tasks. A task can only be executed if all tasks it depends on have already completed.

Some of the tasks in the graph can only be executed on a coprocessor, and the rest can only be executed on the main processor. In one coprocessor call you can send it a set of tasks which can only be executed on it. For each task of the set, all tasks on which it depends must be either already completed or be included in the set. The main processor starts the program execution and gets the results of tasks executed on the coprocessor automatically.

Find the minimal number of coprocessor calls which are necessary to execute the given program.

Input

The first line contains two space-separated integers N ($1 \le N \le 10^5$) — the total number of tasks given, and M ($0 \le M \le 10^5$) — the total number of dependencies between tasks.

The next line contains N space-separated integers . If $E_i = 0$, task i can only be executed on the main processor, otherwise it can only be executed on the coprocessor.

The next M lines describe the dependencies between tasks. Each line contains two space-separated integers T_1 and T_2 and means that task T_1 depends on task T_2 ($T_1 \neq T_2$). Tasks are indexed from 0 to N - 1. All M pairs (T_1, T_2) are distinct. It is guaranteed that there are no circular dependencies between tasks.

Output

Output one line containing an integer — the minimal number of coprocessor calls necessary to execute the program.

Examples

input
4 3 0 1 0 1 0 1 1 2 2 3

output
2

input
4 3 1 1 1 0 0 1 0 2 3 0

output
1

Note

In the first test, tasks 1 and 3 can only be executed on the coprocessor. The dependency graph is linear, so the tasks must be executed in order 3 -> 2 -> 1 -> 0. You have to call coprocessor twice: first you call it for task 3, then you

execute task 2 on the main processor, then you call it for for task 1, and finally you execute task 0 on the main processor.

In the second test, tasks 0, 1 and 2 can only be executed on the coprocessor. Tasks 1 and 2 have no dependencies, and task 0 depends on tasks 1 and 2, so all three tasks 0, 1 and 2 can be sent in one coprocessor call. After that task 3 is executed on the main processor.