A. Cache Hit Optimization

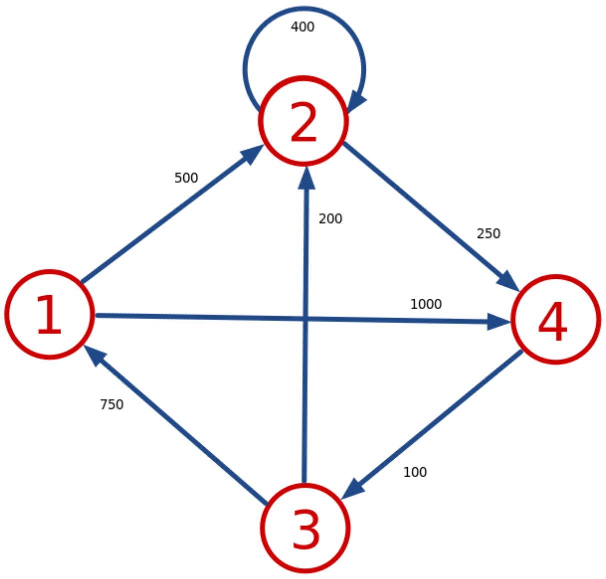
Limits: 2 sec., 1024 MiB

Modern data-center applications often comprise a large amount of code with substantial working sets making them good candidates for code-layout optimizations. Although recent work has evaluated the impact of profile-guided intramodule optimizations and some cross-module optimizations, no recent study has evaluated the benefit of function placement for such large-scale applications.

The embedded software also typically has a very large set of functions, in the order of hundreds of thousands or more. Finding an optimal data or code placement that minimizes cache misses is an NP-hard problem [1]. Moreover, this problem is unlikely to have an efficient approximate solution either. Therefore, in practice, we need some creative solutions to these problems.

You are given N functions as well as their call graph which consists of M directed edges. The functions are numbered from 1 to N and the graph edges are numbered from 1 to M. The instruction size of the k-th function is F_k bytes. The k-th graph edge represents a call of function B_k from function A_k and has a weight of W_k.

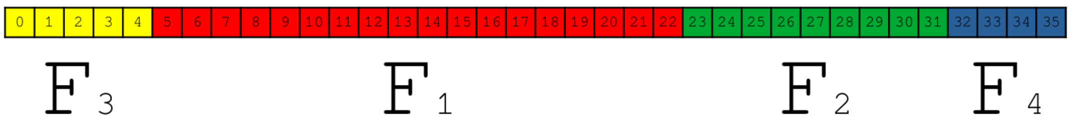
In the sample test case, $N = 4$, $M = 7$, $F_1 = 18$, $F_2 = 9$, $F_3 = 5$, $F_4 = 4$. Its function call graph illustration is provided below. Note that an edge can connect a function to itself, e.g. edge 7 represents a recursive call of function 2.



The function instructions are continuously located in memory from a relative address 0 to F (exclusive) according to some permutation of the functions. Here

$$F = \sum_{k=1}^{N} F_k.$$

The sample test case output corresponds to permutation (3, 2, 1, 4). The corresponding memory allocation is presented below. Note that $F = 36$ for the sample.



[1] E. Petrank and D. Rawitz, "The hardness of cache conscious data placement" in Proceedings of the ACM Symposium on Principles of Programming Languages, pp. 101–112, 2002

The cache consists of C cache lines each of size S bytes. Initially, all cache lines are empty. A cache line can cover any memory range S bytes long.

Before a function call, its instructions have to be processed in the cache according to the following procedure:

1. find the smallest function memory address not yet processed, denote it with Z. If all function instructions have been processed, then exit
2. find a cache line containing Z, denote its start address with Y. If there are multiple such lines, choose one with the smallest start address
3. if the cache line is found at step 2, process function instructions loaded within $[Y,\ Y+S]$, i.e. all between Z and $Y+S$ (exclusive), and go to 1
4. otherwise, if there is no empty cache line, clear one according to Least Recently Used algorithm
5. cover range $[Z,\ Z+S]$ by an available empty cache line
6. process function instructions within range $[Z,\ Z+S]$ and go to 1

The situation at the procedure step 3 is the cache hit, while the one at step 4 is a cache miss. Your goal is to minimize the relative number of such misses among all step 3 and step 4 events by providing a function permutation used for the placement in memory.

During a function call for each corresponding outgoing edge (in order of appearance in the input) the function call it represents takes place with a probability of $\frac{W}{1000}$. Here W is the weight of the call edge.

For scoring purposes, the following algorithm is used:

1. empty the cache
2. iterate over all functions from 1 to N and make a call
3. if the total number of function calls reaches 10^5, then exit

Note, that the algorithm stops as soon as the total number of function calls reaches 10^5. Finally, your score depends on a ratio between cache hits and a sum of hits and misses. For more details, please, refer to the provided scorer source code.

Input

The first line contains four integer numbers N, M, C, and S separated by single spaces. The next N lines contain function instruction lengths F_1, F_2, ..., F_N. Each of the following M lines contains three space-separated integers A_k, B_k, and W_k.

Output

Print N lines each containing one integer. It must be a permutation of integers $1, \ldots, N$ representing a function order in memory.

Constraints

$1 \le N \le 10^4$,
$1 \le M \le 2 \cdot 10^4$,
$1 \le C \le 2000$,
$1 \le S \le 10^4$,
$1 \le F_k \le 10^4$,
$1 \le A_k, B_k \le N$,
$1 \le W_k \le 1000$.

Samples

Input (*stdin*)	Output (*stdout*)
`4 7 4 8`	`3`
`18`	`1`
`9`	`2`
`5`	`4`
`4`	
`1 2 500`	
`2 4 250`	
`1 4 1000`	
`3 1 750`	
`4 3 100`	
`3 2 200`	
`2 2 400`	

Notes

You can print any valid function permutation your solution can find, not necessarily the most optimal one.

Scoring

Your score for a test case is

$$\left\lfloor \frac{hits}{hits + misses} \cdot 10^7 \right\rfloor .$$

Here *hits* is the number of hits and *misses* is the number of misses according to the scorer. Finally, your overall score is the sum of your scores over all test cases.

Scorer

For each test case, the scorer will use:

- the same input provided to your solution
- your output
- random seed parameter

The seed parameter is unknown to solvers and is different for different test cases. However, to ensure fair scoring, it is the same for all simulations of a particular test case. The scorer source code is available for local testing.

Submissions

- The execution time limit is 2 seconds per test case and the memory limit is 1024 mebibytes.
- The code size limit is 64 kibibytes.
- The compilation time limit is 1 minute.
- There are 50 provisional test cases. Your submissions will be evaluated on the provisional set during the submission phase.
- You can submit your code once per 10 minutes and you will get feedback with your normalized score for each of the provisional tests.

- There will be 500 test cases in the final testing after the submission phase is over. The final results will be announced in one week.

Quick start

Check the sample solution which simply orders function according to their incoming edge accumulated weight. The source code is available for some of the contest programming languages:

- C++
- Java
- C#
- Python

Tests

All test cases including provisional and final sets are generated by the generator. Please, check the generator source code and feel free to use it for local testing.