Register Allocation

Modern computer processors have a limited number of *registers* — general purpose storage locations that are substantially faster than the main memory. The operations that perform computations (e.g. addition, multiplication, etc.) expect their arguments in the registers and return the result in a register.

In this task we consider the problem of *register allocation* for expression evaluation. Inside a compiler, an expression is represented as a tree. Leaf nodes of the tree correspond to values that have to be loaded from the main memory. Intermediate (non-leaf) nodes of the tree correspond to the operations and each of them has as many children as there are arguments to the operation. Obviously, the values of all arguments must be available before an operation can be performed.

Because there is only a limited number of registers, the compiler has to decide which of the intermediate results to keep in the registers (these are available immediately when they are needed) and which ones to store into the main memory (these must be loaded back when they are needed). It may also be useful to change the order of evaluation of the arguments for an operation (this is why most high-level languages do not guarantee any particular order).

Your task is to write a program that, given an expression tree, finds the register allocation plan and evaluation order with the minimal total cost.

Input. The first line of the input file `REGS.IN` contains the number of registers, N ($1 \le N \le 100$). The second line contains two integers: the cost of loading a value from the main memory into a register, C_l ($1 \le C_l \le 100$), and the cost of storing a value from a register into the main memory, C_s ($1 \le C_s \le 100$). The rest of the file describes the expression tree, starting from the root node:

- the first line contains the number of children of the node, K_x ($0 \le K_x \le 10$ and $K_x \le N$);
- if $K_x = 0$, this is a leaf node and the description is over;
- if $K_x > 0$, this is an intermediate node, and:
  - the next line contains one integer: the cost of the operation that the node represents, C_x ($1 \le C_x \le 100$);
  - this is followed by the descriptions of the K_x subtrees according to the same scheme.

The nodes are numbered from 1 to M in the order in which they appear in the input file. It can be assumed that $M \le 10\,000$.

Output. The first line of output file `REGS.OUT` must contain the minimal cost of evaluating the expression. The rest of the file must contain one line for each intermediate node of the expression tree. Each line must contain two integers: the first is the number of the node to be evaluated, the second is 1 if the result should be kept in a register, or 0 if it should be stored into the main memory (with the cost C_s also added to the total cost of evaluation).

The operations must be listed in the order in which they should be performed to ensure that the total cost of evaluating the expression is the least possible, under the following conditions:

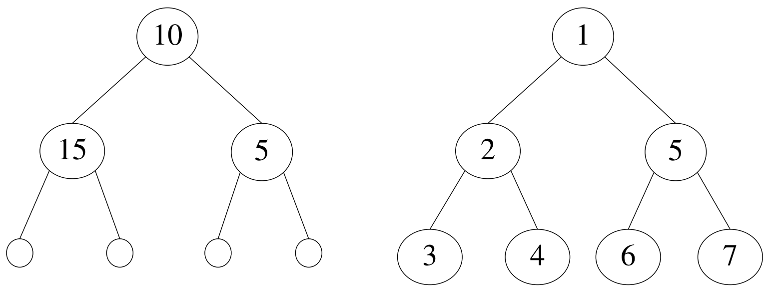
- a node can only be evaluated after all its children have been evaluated;
- all arguments of an operation to be performed that are not in the registers have to be loaded from the main memory (with the cost C_l per value);
- registers containing the arguments of the operation performed can be immediately reused, in particular for storing the result of the operation.

If there are several plans with minimal cost, output any one of them.

Sample.

REGS.IN	REGS.OUT
2	47
3 2	2 0
2	5 1
10	1 1
2	
15	
0	
0	
2	
5	
0	
0	

Remark. The figure below illustrates the input file above: both trees correspond to the expression tree, whereas the tree on the left shows the costs of the intermediate nodes and the tree on the right shows the numbering of the nodes.



The cost of the evaluation plan in the output file above is

$$(C_l + C_l + 15 + C_s) + (C_l + C_l + 5) + (C_l + 10) = 47.$$

Remark. You will be given a program REGSCHECK which verifies the correctness (but not optimality) of the REGS.OUT with respect to the REGS.IN and gives informative error messages.