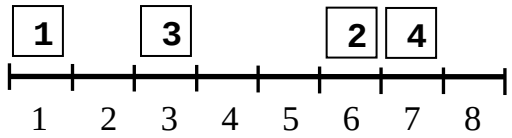
boxes 100 points

Source code: **boxes.c, boxes.cpp, boxes.pas**
Time limit **10 s**
Memory limit: **32 MB**

Task

A circular belt has **N** positions (numbered from **1** to **N**), each of which can hold a single box. At any point in time, some positions have boxes, and the remaining are empty. You can move a box by pushing it on the belt to any position, provided you do not skip over a box. Pushing is circular (a box at position **N** can be pushed to position **1** if it's free).
Below you have an example belt with **N**=8 and 4 boxes (each with an id from 1 to 4). In this example, you can push box 3 to positions 2, 4 or 5, box 1 to positions 2 and 8, box 4 to position 8, and box 2 to positions 4 and 5.



Your task is to support a number of box insertions on the belt. Each new box comes with a requirement for its location: it must be placed after a specific box $\mathbf{b_i}$ and before the next one on the belt (in circular order). Before placing the new box, you can move the boxes already on the belt to make space. In the above example, to insert a new box between 2 and 4, at least one of them has to be moved.

Boxes are identified by id numbers starting at 1, in order of insertion. Initially, there are **N/4** boxes on the belt that your program can place at its discretion. Thereafter, **N/4** more boxes will be inserted. Before inserting each new box, you are allowed to make at most **200** moves (you can receive partial credit if you make at most **500** moves).

Interaction

Your program should not read or write any files. Instead, it will interact with another program provided by the scientific committee that will run at the same time with your program. The interaction will take place in the following way:

1. Your program will read one line from standard input containing an integer **N.**
2. Initially, you will place **N/4** boxes on the band as you wish. Your program should write **N/4** lines to the standard output, in the following format: **I p**
   The *k*-th line specifies the position **p** on the belt, where you want to place box number *k*. All positions must be distinct.
3. You will be required to insert **N/4** more boxes, following this protocol:
   - Your program will read one line from standard input containing an integer $\mathbf{b_i}$. The new box must be inserted on any position after box number $\mathbf{b_i}$ and before the next box on the belt, and it will have the next available box number.

- Your program will write several lines to standard output, each describing a move operation in the following format: **`M b p`**
  Such a line means "move box number **`b`** to position **`p`**." Remember that there should be an empty path from the current position of box **`b`** to the final position **`p`**.
- When you are done moving boxes, your program will write one line to standard output in the following format: **`I p`**
  Such a line means "insert the new box on position **`p`**"; remember that **`p`** should be after the position of box b_i and before the next box in circular order.

4. After exactly **`N/4`** insertions your program must finish without any further interaction.

Constraints

- In all test cases, **`N = 20 000`**
- The committee's program will use various strategies for deciding the position of new boxes.
- Your score for each test is based on the **maximum** number of moves made before any insertions:
  - You gain 100% if you never use more than 200 moves
  - You gain 70% if you never use more than 300 moves
  - You gain 40% if you never use more than 500 moves

Programming instructions

After every complete line written to the standard output, C programmers must use **`fflush(stdout)`** function while Pascal programmers must use **`flush(output)`** procedure.

C	*C++*	*Pascal*
`printf("I %d\n", p);` `fflush(stdout);`	`cout<<"I "<<p<< '\n';` `cout.flush();`	`writeln('I ', p);` `flush(output);`

Submission tests

At submission time, your program will be evaluated with three different interactive programs. The behavior of these is as follows:

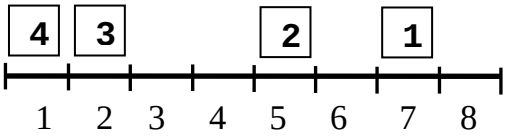
1. All insertions are after box **`1`**
2. Insertions are made after randomly selected boxes
3. Finds an interval of the belt that contains many boxes, and inserts inside it.

The interactive programs used during official grading will use other strategies.

Example

Read **N**	**8**
Write **I 1** Write **I 5**	
Read **b1** Write **I 2**	**1**
Read **b2** Write **M 1 7** Write **I 1**	**1**

The belt will have the following arrangement:



harbingers 100 points

Source code:	**harbingers.c, harbingers.cpp, harbingers.pas**
Input files:	**harbingers.in**
Output files:	**harbingers.out**
Time limit:	**1 s**
Memory limit:	**32 MB**

Task

Once upon a time, there were **N** medieval towns in the beautiful Moldavian territory, uniquely numbered from **1** through **N**. The town numbered with **1** was the capital city. The towns were connected by **N-1** bidirectional roads, each road having a length expressed in kilometers. There was a unique way to travel between any pair of towns without going through a town twice (i.e. the graph of roads was a tree).

When a town was attacked, the situation had to be reported as soon as possible to the capital. The message was carried by harbingers, one of which resided in each town. Each harbinger was characterized by the amount of time required to start the journey and by his constant speed (expressed in minutes per kilometer) after departure.

The message from a town was always carried on the unique shortest path to the capital. Initially, the harbinger from the attacked town carried the message. In each town that he traversed, a harbinger had two options: either go to the next town towards the capital, or leave the message to the harbinger from this town. The new harbinger applied the same algorithm as above. Overall, a message could be carried by any number of harbingers before arriving in the capital.

Your task is to find, for each town, the minimum time required to send a message from that town to the capital.

Description of input

The first line of the input file **harbingers.in** contains one integer **N**, the number of towns in Moldavia. Each of the following **N-1** lines contains three integers **u v d**, separated by one space, describing a road of length **d** kilometers between towns numbered with **u** and **v**. Subsequently, **N-1** pairs of integers follow, one per line. The **i**th pair, $\mathbf{S_i}$ $\mathbf{V_i}$, describes the characteristics of the harbinger in the **(i+1)**th town: $\mathbf{S_i}$ is the number of minutes to prepare for the journey, and $\mathbf{V_i}$ is the number of minutes needed to travel one kilometer. There is no harbinger in the capital.

Description of output

The output file **harbingers.out** should consist of exactly one line containing **N-1** integers. The **i**th number represents the minimum time, in minutes, required to send a message from the **(i+1)**th town to the capital.

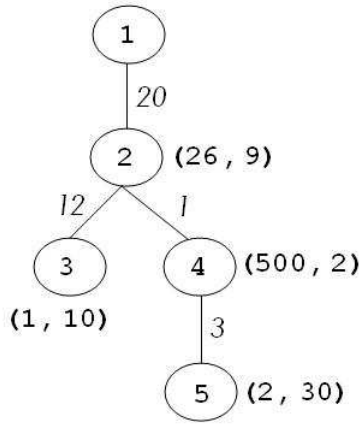
Constraints

- **3 ≤ N ≤ 100 000**
- **0 ≤ S_i ≤ 10^9**
- **1 ≤ V_i ≤ 10^9**
- The length of each road will not exceed **10 000**
- For 20% of the tests**, N ≤ 2 500**
- For 50% of the tests, each town will have at most 2 adjacent roads (i.e., the graph of roads will be a *line*)

Example

harbingers.in	harbingers.out
5 1 2 20 2 3 12 2 4 1 4 5 3 26 9 1 10 500 2 2 30	206 321 542 328

Explanation



The roads and their lengths are shown in the image on the left. The start-up time and speed of the harbingers are written between brackets.

The minimum time to send a message from town 5 to the capital is achieved as follows. The harbinger from town 5 takes the message and leaves the town after 2 minutes. He walks 4 kilometers in 120 minutes before arriving in town 2. There he leaves the message to the harbinger from that town. The second harbinger requires 26 minutes to start the journey and walks for 180 minutes before arriving to the capital.

The total time is therefore 2 + 120 + 26 + 180 = 328.

photo 100 points

Source code:	`photo.c, photo.cpp, photo.pas`
Input files:	`photo.in`
Output files:	`photo.out`
Time limit:	`1.0 s`
Memory limit:	`16 MB`

Task

You are given a photo of the skyline of Târgu-Mureş taken during the night. Some rooms still have the light on. You know that all the buildings can be modeled by rectangles of surface area at most **A**. Find the minimum number of buildings that can lead to the picture.

Specifically, you are given an integer **A**, and **N** points at integer coordinates (**x,y**). You must find a minimum number of rectangles that have one side on the *x*-axis and area at most **A,** which cover all points. The rectangles may overlap.

Description of input

The first line of the input file **photo.in** will contain two integers **N** and **A**, separated by a single space. The next **N** lines will contain two integers **x** and **y**, representing the coordinates of each point.

Description of output

The output file **photo.out** should consist of exactly one line containing the minimum number of rectangles.

Constraints

- $1 \le N \le 100$
- $1 \le A \le 200\,000$
- Each point has $0 \le x \le 3\,000\,000$ and $1 \le y \le A$
- For **30**% of the test cases, $1 \le N \le 18$

Example

photo.in	photo.out	Here is one possible picture that explains the example:
6 4 2 1 4 1 5 1 5 4 7 1 6 4	3