Task: TRE
Treasure Hunt

CEOI 2011, Day 1. Source file `tre.*` Available memory: 256 MB. *9.07.2011*

Ahoy! Have you ever heard about pirates and their treasures? Bytie has found an old bottle while having a walk along the beach in Gdynia. The letter inside gives instructions on how to find a hidden treasure, but it is quite difficult to decipher. One thing Bytie knows for sure is that he needs to find two special points in the park nearby and the treasure will be in the middle of the path connecting them.

There are several trails in the park. Apart from that, the forest in the park is very dense, so only positions on the trails are reachable for human beings. The structure of the trails has an interesting property: for any two points lying on the trails there is a unique path connecting them. The path may lead along multiple trails, but it never visits any point more than once.

Bytie asked his friends for help in exploring the park. They will start the treasure hunt in some point of the park, located on one of the trails. They will explore the park in phases. In each phase, one of the friends chooses one point that was already explored and walks a number of steps from that point along a trail, visiting only points which were never reached by any of the friends before.

During the exploration, Bytie will be analysing the structure of the park carefully. From time to time, Bytie may guess the two special points which determine the location of the treasure. For each such guess, he wants to know the point located in the middle of the only path connecting them. Your task is to help Bytie in determining these middle points.

Communication

You should write a library which interacts with the grading program. The library should contain the following three functions which will be called by the grader (and any more functions if you like):

- `init` — this function will be called exactly once, in the beginning of the execution. It is for you to initialize your data structures etc.
  - C/C++: `void init();`
  - Pascal: `procedure init();`

  When this function is called, you should assume that there is exactly one point already explored in the park, marked with the number 1.
- `path` — stating that one of the friends explored a new path in the park. This function is for you to build your data structures representing trails.
  - C/C++: `void path(int a, int s);`
  - Pascal: `procedure path(a, s: longint);`

  The path starts in the point number a (which was already explored) and takes s steps along a trail ($s > 0$). After each step, the current point is assigned a new number: the smallest positive integer not yet used for this purpose. This function will be called at least once.
- `dig` — asking where to dig for the treasure.
  - C/C++: `int dig(int a, int b);`
  - Pascal: `function dig(a, b: longint) : longint;`

  It should return the number assigned to the point located in the middle of the path connecting the points marked with the numbers a and b. You can assume that the points a and b have already been explored and that $a \neq b$. If the middle of the path is not a point with an assigned number (because the path has an odd length), the function should return the number assigned to one of the two middle points of the path — the one that is closer to a (see also the example on the next page). This function will be called at least once.

Your library **may not** read anything, neither from the standard input nor from any file, and **may not** write anything, neither to the standard output nor to any file. Your library **may** write to the standard error stream/file (`stderr`). You should be aware, however, that this consumes time during the execution of the program.

If you are writing in C/C++, your library **may not** contain the `main` function. If you are using Pascal, you have to provide a unit (see the sample programs on your disk).

Compilation

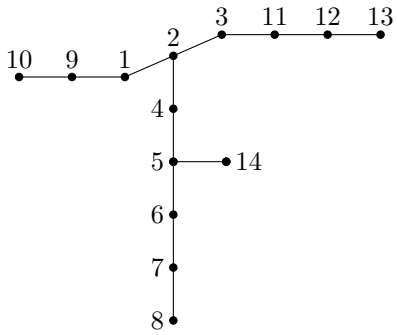
Your library — `tre.c`, `tre.cpp`, or `tre.pas` — will be compiled with the grader using the following commands:

- C: `gcc -O2 -static -lm tre.c tregrader.c -o tre`
- C++: `g++ -O2 -static -lm tre.cpp tregrader.cpp -o tre`
- Pascal:
```
ppc386 -O2 -XS -Xt tre.pas
ppc386 -O2 -XS -Xt tregrader.pas
mv tregrader tre
```

Sample execution

The table below shows a sample sequence of calls to the functions and the correct results of the `dig` calls. The structure of the trails in the park corresponding to this example run is shown in the figure.

Function call	Result	New points added
`init();`		1
`path(1, 2);`		2, 3
`dig(1, 3);`	2	
`path(2, 5);`		4, 5, 6, 7, 8
`dig(7, 3);`	5	
`dig(3, 7);`	4	
`path(1, 2);`		9, 10
`path(3, 3);`		11, 12, 13
`dig(10, 11);`	1	
`path(5, 1);`		14
`dig(14, 8);`	6	
`dig(2, 4);`	2	



Constraints

- There will be at most 400 000 calls to the functions (`init`, `path`, and `dig`) and at most 1 000 000 000 points explored by Bytie's friends.
- In test cases worth 50 points in total, there will be no more than 400 000 points explored.
- In test cases worth 20 points in total, there will be no more than 5 000 points explored and at most 5 000 calls to the functions `init`, `path`, and `dig`.

Experimentation

To let you experiment with your solution, you are given a sample grading program in the file

`tregrader.c`, `tregrader.cpp`, and `tregrader.pas`

located in the directory **/home/zawodnik/tre/** on your machine. To experiment with the grading program, you should put your solution in the file

`tre.c`, `tre.cpp`, or `tre.pas`

in the respective directory (`c`, `cpp`, or `pas`). In the beginning of the contest in each of these files you can find a sample incorrect solution to the problem. You can compile your solution using the command

```
make tre
```

which works exactly as described in the Compilation section, provided that you compile your program in the respective directory. The C/C++ compilation requires the file `treinc.h`, which is also located in the respective directories.

The resulting binary reads a list of function names and arguments from the standard input, calls the corresponding functions from your solution and writes the results of the calls of the `dig` function to the standard output. The list of functions in the input should be formatted as follows: the first line contains the

number of instructions, q. Then q lines follow, each containing a character `i`, `p`, or `d` followed by two non-negative integers. The character determines which function should be called: `i` for `init`, `p` for `path`, and `d` for `dig`. The integers denote the arguments of the function: a and s for `path`, a and b for `dig`. If the character is `i`, both integers are equal to 0. Note that the sample grader **does not** check if the input is formatted correctly or if it satisfies the requirements listed in the Communication & Constraints sections.

You are given the file `tre0.in` which represents the sample execution described above:

```
12
i 0 0
p 1 2
d 1 3
p 2 5
d 7 3
d 3 7
p 1 2
p 3 3
d 10 11
p 5 1
d 14 8
d 2 4
```

To read the data from this file, use the following command:

```
./tre < tre0.in
```

The results of the `dig` calls returned by your solution will be written to the standard output. The correct result for the aforementioned input, written also in the file `tre0.out`, is:

```
2
5
4
1
6
2
```

You can verify if the output of your solution to the sample test case is correct by submitting your solution to the grading system (SIO).