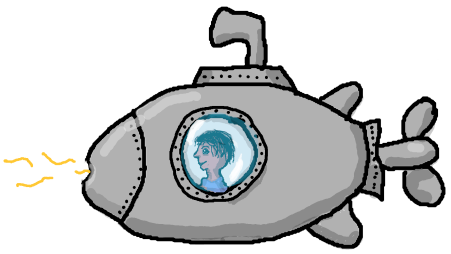
Flight to the Ford (communication)

Every break-in, even if hypothetical, needs a good escape plan. Thus, you have hired an assistant to help you in your escape from the underwater vault you discovered yesterday.

In order for your plan to work, it will be important to communicate with your assistant. More precisely, you will need to send them one of N distinct messages (conveniently numbered from 1 to N). Unfortunately, your possibilities to send a message are very limited once you're in your fancy new submarine: There are only two different types of signals you can send. Therefore, you have to encode your message as a sequence of these signals.

However, sending such a signal is a complicated process* and can even fail; in this case, the *wrong* signal is sent. *At least you can be sure that this will never happen twice in a row.* Moreover, you will always know which signal was actually sent, and will be able to react accordingly.



Obviously the signal displayed in the picture is to be interpreted as the bit 1

You already noticed that it might be impossible to unambiguously communicate a message under these circumstances. Hence, you will be happy when your assistant can *determine at most two messages that you possibly wanted to communicate*, i.e. such that your original message is one of them. Remembering that you are a talented programmer, you now want to write a program which

- you can use to decide which signals to send to your assistant, *and*
- your assistant can use to determine the two possible messages.

As sending signals from your submarine might raise suspicion,† you can send at most 250 of them. Beware moreover that your assistant will have to react to your message quickly. *Thus, they must notice when the communication ends without waiting for further signals!*

Communication

This is a communication task, in which your program is run several times for each testcase. You have to write the following functions; in each run of your submission, precisely one of them will be called, *multiple times* and possibly with different parameters:

- The function **void** *encode*(**int** N, **int** X) where as above N denotes the number of distinct messages and X is the message you want to communicate, where $1 \le X \le N$. Inside *encode* you may make up to 250 calls to the grader function **int** *send*(**int** s); s must be either 0 or 1, meaning you want to send signal s. The return value of this function tells you which signal was actually sent. This value may differ from s, but for any two consecutive calls to *send* inside the same call to *encode* this will happen at most once.
- The function **std**::**pair**⟨**int**, **int**⟩ *decode*(**int** N) where N is the same as in *encode*. For each call to *encode*, there is one call to *decode*. Inside *decode* you may call the grader function **int** *receive*() which returns the next signal sent during the corresponding call of *encode*. In the end, *decode* should return a pair of two integers $1 \le a, b \le N$ (the case $a = b$ is allowed) such that the original message was one of a or b.

* Involving the submarine's torpedo tube, a surprising and elegant application of Dijkstra's algorithm, and a family size spaghetti pack

† It *does* lead to major radio interference as well as deeply disturbed local wildlife after all.

You will automatically fail a testcase if you call *receive* inside *decode* more often than *send* was called in the respective run of *encode*. However, you are allowed to call *receive* fewer times than that.

If any of your function calls does not satisfy the above constraints, your program will be immediately terminated and judged as **Not correct** for the respective testcase. You must not write to standard output or read from standard input; otherwise you may receive the verdict **Security violation!**.

You must include the file `communication.h` in your source code. To test your program locally, you can link it with `sample_grader.cpp`, which can be found in the attachment for this task in CMS (see below for a description of the sample grader, and see `sample_grader.cpp` for instructions on how to run it with your program). Beware that for simplicity this sample grader does *not* run your program twice but instead calls both *encode* and *decode* (exactly once) as part of a single run. The attachment also contains a sample implementation as `communication_sample.cpp`.

Important technical remarks

As mentioned above, the functions *encode* and *decode* can be called several times per run. Please note the following:

1. It is *not* guaranteed that the calls to *decode* will appear in the same order as the calls to *encode*.
2. The time limit below and the runtime displayed inside CMS refer to the *average runtime* of all calls to *encode* and *decode* in a given run. Put differently, when there are K calls to *encode* or K calls to *decode* in a given run, then your program must not take more than $K \cdot 0.005\,\text{s}$ for this run. It is guaranteed that there at least 50 calls to *encode* or *decode* in each run.
3. As usual, the memory limit refers to the maximum memory consumption at any point in time during execution.

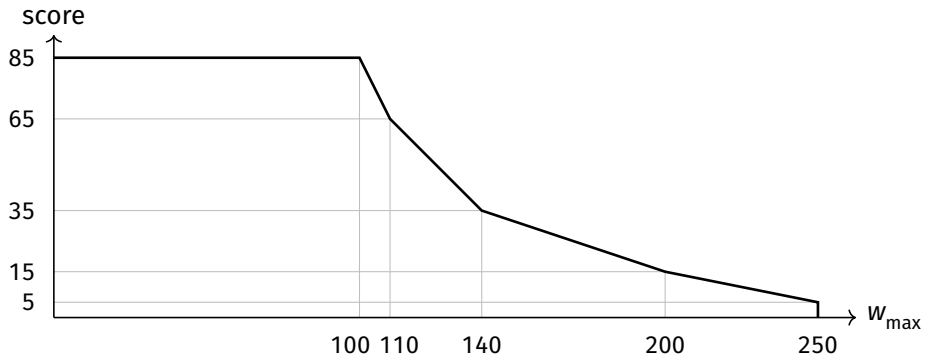
Constraints

We always have $3 \le N \le 10^9$.

Subtask 1 (15 points). $N = 3$

Subtask 2 (85 points). No further constraints.

Your actual score in Subtask 2 depends on the maximum number $w_{\max}$ of signals sent over all messages in testcases in this subtask according to the following piecewise linear function (rounded to the unique nearest integer):



In particular, to get full score you must not make more than 100 calls to *send* per testcase.

Example Interaction

Consider a testcase with $N = 1337$ and $X = 42$.

First, the grader calls your function *encode* as *encode*(1337, 42). Then, a possible interaction between your program and the grader could look as follows:

Your program	Return value	Explanation
send(1)	0	the wrong signal was sent
send(0)	0	the correct signal was sent (as was guaranteed)
send(1)	1	the correct signal was sent again
send(1)	0	the wrong signal was sent

Afterwards (in a new run of your program) the grader calls your function *decode* as *decode*(1337). Here a possible interaction could look as follows:

Your program	Return value	Explanation
receive()	0	the first call to *send* returned 0 (although this wasn't the parameter you called it with)
receive()	0	the second call to *send* returned 0
receive()	1	the third call to *send* returned 1
return {1337, 42}	—	your solution is correct and is accepted

Note that your program would have been allowed to call *receive* one more time.

Grader

The sample grader first expects on standard input the integers N and X ($1 \le X \le N$). Then, the grader calls *encode*(N, X) and writes to standard output a protocol of all calls to *send* by your program. For each call to *send* it expects the return value on standard input.

Afterwards the grader calls *decode*(N) and writes to standard output a protocol of all calls to *receive* by your program. Upon termination, it writes one of the following messages to standard output:

Invalid input. The input to the grader via standard input was not of the above form.

Invalid send. You called *send* inside *decode* or you called *send* with a parameter different from 0 or 1.

Invalid reply to send. The return value to *send* given on standard input was neither 0 nor 1, or it differed from the argument to *send* twice in a row.

Looks (and smells) fishy. You called *send* more than 250 times.

Invalid receive. You called *receive* inside *encode*.

Assistant waiting for Godot. You called *receive* more often than *send*.

Invalid answer. The function *decode* did not return a pair of integers between 1 and N.

Wrong answer. The pair returned by *decode* does not contain the original message X.

Correct: *w* signal(s) sent. The pair returned by *decode* contains the original message X and there were w calls to *send*.

In contrast, the grader actually used to judge your program will only output **Not correct** (for any of the above errors) or **Correct: *w* signal(s) sent**. Moreover, it is *adaptive*, i.e. the parameters N and X as well as the return values of *send* can depend on the behaviour of your program in the current as well as other runs. Both the sample grader and the grader used to judge your program will terminate your program automatically whenever one of the above errors occurs.

Limits

Time: 0.005 s
Memory: 8 MiB