Last Supper

Leonardo was very active when working on the Last Supper, his most famous mural painting: one of his first daily tasks was that of deciding which tempera colors to use during the rest of the working day. He needed many colors but could only keep a limited number of them on his scaffold. His assistant was in charge, among other things, of climbing up the scaffold to deliver colors to him and then getting down to put them back on the suitable shelf on the floor.

In this task, you will need to write two separate programs to help the assistant. The first program will receive Leonardo's instructions (a sequence of colors Leonardo will need during the day), and create a *short* string of bits, called *advice*. While processing Leonardo's requests during the day, the assistant will not have access to Leonardo's future requests, only to the advice produced by your first program. The second program will receive the advice, and then receive and process Leonardo's requests in an online fashion (i.e., one at a time). This program must be able to understand what the advice means and use it to make optimal choices. Everything is explained below in more detail.

Moving colors between shelf and scaffold

We will consider a simplified scenario. Suppose that there are N colors numbered from 0 to N - 1, and that each day Leonardo asks the assistant for a new color exactly N times. Let C be the sequence of the N color requests made by Leonardo. Thus we may think of C as a sequence of N numbers, each being between 0 and N - 1, inclusive. Note that some colors might not occur in C at all, and others may appear multiple times.

The scaffold is always full and contains some K of the N colors, with K < N. Initially, the scaffold contains the colors from 0 to K - 1, inclusive.

The assistant processes Leonardo's requests one at a time. Whenever the requested color is *already on the scaffold*, the assistant can rest. Otherwise, he has to pick up the requested color from the shelf and move it to the scaffold. Of course, there is no room on the scaffold for the new color, so the assistant must then choose one of the colors on the scaffold and take it from the scaffold back to the shelf.

Leonardo's optimal strategy

The assistant wants to rest as many times as possible. The number of requests for which he can rest depends on his choices during the process. More precisely, each time the assistant has to remove a color from the scaffold, different choices may lead to different outcomes in the future. Leonardo explains to him how he can achieve his goal knowing C. The best choice for the color to be removed from the scaffold is obtained by examining the colors currently on the scaffold, and the remaining color requests in C. A color should be chosen among those on the scaffold according to the following rules:

- If there is a color on the scaffold that will never be needed in the future, the assistant should remove such a color from the scaffold.
- Otherwise, the color removed from the scaffold should be the one *that will next be needed furthest in the future*. (That is, for each of the colors on the scaffold we find its first future occurrence. The color moved back to the shelf is the one that will be needed last.)

It can be proved that when using Leonardo's strategy, the assistant will rest as many times as possible.

Example 1

Let N = 4, so we have 4 colors (numbered from 0 to 3) and 4 requests. Let the sequence of requests be C = (2, 0, 3, 0). Also, assume that K = 2. That is, Leonardo has a scaffold capable of holding 2 colors at any time. As stated above, the scaffold initially contains the colors 0 and 1. We will write the content of the scaffold as follows: [0, 1]. One possible way that the assistant could handle the requests is as follows.

- The first requested color (number 2) is not on the scaffold. The assistant puts it there and decides to remove color 1 from the scaffold. The current scaffold is [0, 2].

- The next requested color (number 0) is already on the scaffold, so the assistant can rest.

- For the third request (number 3), the assistant removes color 0, changing the scaffold to [3, 2].

- Finally, the last requested color (number 0) has to be taken from the shelf to the scaffold. The assistant decides to remove color 2, and the scaffold now becomes [3, 0].

Note that in the above example the assistant did not follow Leonardo's optimal strategy. The optimal strategy would remove the color 2 in the third step, so the assistant could then rest again in the final step.

Assistant's strategy when his memory is limited

In the morning, the assistant asks Leonardo to write C on a piece of paper, so that he can find and follow the optimal strategy. However, Leonardo is obsessed with keeping his work techniques secret, so he refuses to let the assistant have the paper. He only allowed the assistant to read C and try to remember it.

Unfortunately, the assistant's memory is very bad. He is only able to remember up to M bits. In general, this might prevent him from being able to reconstruct the entire sequence C. Hence, the assistant has to come up with some clever way of computing the sequence of bits he will remember. We will call this sequence the *advice sequence* and we will denote it A.

Example 2

In the morning, the assistant can take Leonardo's paper with the sequence C, read the sequence, and make all the necessary choices. One thing he might choose to do would be to examine the state of the scaffold after each of the requests. For example, when using the (sub-optimal) strategy given in Example 1, the sequence of scaffold states would be [0, 2], [0, 2], [3, 2], [3, 0]. (Recall that he knows that the initial state of the scaffold is [0, 1].)

Now assume that we have M = 16, so the assistant is able to remember up to 16 bits of information. As N = 4, we can store each color using 2 bits. Therefore 16 bits are sufficient to store the above sequence of scaffold states. Thus the assistant computes the following advice sequence: A = (`0, 0, 1, 0, 0, 0, 1, 0, 1, 1, 1, 0, 1, 1, 0, 0`).

Later in the day, the assistant can decode this advice sequence and use it to make his choices.

(Of course, with M = 16 the assistant can also choose to remember the entire sequence C instead, using only 8 of the available 16 bits. In this example we just wanted to illustrate that he may have other options, without giving away any good solution.)

Statement

You have to write *two separate programs* in the same programming language. These programs will be executed sequentially, without being able to communicate with each other during the execution.

The first program will be the one used by the assistant in the morning. This program will be given the sequence C, and it has to compute an advice sequence A.

The second program will be the one used by the assistant during the day. This program will receive the advice sequence A, and then it has to process the sequence C of Leonardo's requests. Note that the sequence C will only be revealed to this program one request at a time, and each request has to be processed before receiving the next one.

More precisely, in the first program you have to implement a single routine `ComputeAdvice`(C, N, K, M) having as input the array C of N integers (each in 0, …, N - 1), the number K of colors on the scaffold, and the number M of bits available for the advice. This program must compute an advice sequence A that consists of up to M bits. The program must then communicate the sequence A to the system by calling, for each bit of A in order, the following routine:

- `WriteAdvice`(B) — append the bit B to the current advice sequence A. (You can call this routine at most M times.)

In the second program you have to implement a single routine `Assist`(A, N, K, R). The input to this routine is the advice sequence A, the integers N and K as defined above, and the actual length R of the sequence A in bits (R ≤ M). This routine should execute your proposed strategy for the assistant, using the following routines that are provided to you:

- `GetRequest`() — returns the next color requested by Leonardo. (No information about the future requests is revealed.)

- `PutBack`(T) — put the color T from the scaffold back to the shelf. You may only call this routine with T being one of the colors currently on the scaffold.

When executed, your routine `Assist` must call `GetRequest` exactly N times, each time receiving one of Leonardo's requests, in order. After each call to `GetRequest`, if the color it returned is *not* in the scaffold, you *must* also call `PutBack`(T) with your choice of T. Otherwise, you *must not* call `PutBack`. Failure to do so is considered an error and it will cause the termination of your program. Please recall that in the beginning the scaffold contains the colors from 0 to K - 1, inclusive.

A particular test case will be considered solved if your two routines follow all the imposed constraints, and the total number of calls to `PutBack` is *exactly equal* to that of Leonardo's optimal strategy. Note that if there are multiple strategies that achieve the same number of calls to `PutBack`, your program is allowed to perform any of them. (I.e., it is not required to follow Leonardo's strategy, if there is another equally good strategy.)

Example 3

Continuing Example 2, assume that in `ComputeAdvice` you computed A = (`0, 0, 1, 0, 0, 0, 1, 0, 1, 1, 1, 0, 1, 1, 0, 0`). In order to communicate it to the system, you would have to make the following sequence of calls: `WriteAdvice(0)`, `WriteAdvice(0)`, `WriteAdvice(1)`, `WriteAdvice(0)`, `WriteAdvice(0)`, `WriteAdvice(0)`, `WriteAdvice(1)`, `WriteAdvice(0)`, `WriteAdvice(1)`, `WriteAdvice(1)`, `WriteAdvice(1)`, `WriteAdvice(0)`, `WriteAdvice(1)`, `WriteAdvice(1)`, `WriteAdvice(0)`, `WriteAdvice(0)`.

Your second routine `Assist` would then be executed, receiving the above sequence A, and the values N = 4, K = 2, and R = 16. The routine `Assist` then has to perform exactly N = 4 calls to `GetRequest`. Also, after some of those requests, `Assist` will have to call `PutBack`(T) with a suitable choice of T.

The table below shows a sequence of calls that corresponds to the (sub-optimal) choices from Example 1. The hyphen denotes no call to `PutBack`.

GetRequest()	**Action**
2	`PutBack(1)`
0	-
3	`PutBack(0)`
0	`PutBack(2)`

Subtask 1 [8 points]

- N ≤ 5 000.
- You can use at most M = 65 000 bits.

Subtask 2 [9 points]

- N ≤ 100 000.
- You can use at most M = 2 000 000 bits.

Subtask 3 [9 points]

- $N \leq 100\,000$.
- $K \leq 25\,000$.
- You can use at most M = 1 500 000 bits.

Subtask 4 [35 points]

- $N \leq 5\,000$.
- You can use at most M = 10 000 bits.

Subtask 5 [up to 39 points]

- $N \leq 100\,000$.
- $K \leq 25\,000$.
- You can use at most M = 1 800 000 bits.

The score for this subtask depends on the length R of the advice your program communicates. More precisely, if R_{max} is the maximum (over all test cases) of the length of the advice sequence produced by your routine `ComputeAdvice`, your score will be:

- 39 points if $R_{max} \leq 200\,000$;
- 39 (1 800 000 - R_{max}) / 1 600 000 points if $200\,000 < R_{max} < 1\,800\,000$;
- 0 points if $R_{max} \geq 1\,800\,000$.

Implementation details

You should submit exactly two files *in the same programming language.*

The first file is called `advisor.c`, `advisor.cpp` or `advisor.pas`. This file must implement the routine `ComputeAdvice` as described above and can call the routine `WriteAdvice`. The second file is called `assistant.c`, `assistant.cpp` or `assistant.pas`. This file must implement the routine `Assist` as described above and can call the routines `GetRequest` and `PutBack`.

The signatures for all the routines follow.

C/C++ programs

```
void ComputeAdvice(int *C, int N, int K, int M);
void WriteAdvice(unsigned char a);
void Assist(unsigned char *A, int N, int K, int R);
void PutBack(int T);
int GetRequest();
```

Pascal programs

```
procedure ComputeAdvice(var C : array of LongInt; N, K, M : LongInt);
procedure WriteAdvice(a : Byte);
```

```
procedure Assist(var A : array of Byte; N, K, R : LongInt);
procedure PutBack(T : LongInt);
function GetRequest : LongInt;
```

These routines must behave as described above. Of course you are free to implement other routines for their internal use. For C/C++ programs, your internal routines should be declared `static`, as the sample grader will link them together. Alternately, just avoid having two routines (one in each program) with the same name. Your submissions must not interact in any way with standard input/output, nor with any other file.

When programming your solution, you also have to take care of the following instructions (the templates you can find in your contest environment already satisfy the requirements listed below).

C/C++ programs

At the beginning of your solution, you have to include the file `advisor.h` and `assistant.h`, respectively, in the advisor and in the assistant. This is done by including in your source the line:

```
#include "advisor.h"
```

or

```
#include "assistant.h"
```

The two files `advisor.h` and `assistant.h` will be provided to you in a directory inside your contest environment and will also be offered by the contest Web interface. You will also be provided (through the same channels) with code and scripts to compile and test your solution. Specifically, after copying your solution into the directory with these scripts, you will have to run `compile_c.sh` or `compile_cpp.sh` (depending on the language of your code).

Pascal programs

You have to use the units `advisorlib` and `assistantlib`, respectively, in the advisor and in the assistant. This is done by including in your source the line:

```
uses advisorlib;
```

or

```
uses assistantlib;
```

The two files `advisorlib.pas` and `assistantlib.pas` will be provided to you in a directory inside your contest environment and will also be offered by the contest Web interface. You'll also be provided (through the same channels) with code and scripts to compile and test your solution. Specifically, after copying your solution into the directory with these scripts, you will have to run `compile_pas.sh`.

Sample grader

The sample grader will accept input formatted as follows:

- line 1: N, K, M;
- lines 2, …, N + 1: C[i].

The grader will first execute the routine `ComputeAdvice`. This will generate a file `advice.txt`, containing the individual bits of the advice sequence, separated by spaces and terminated by a 2.

Then it will proceed to execute your `Assist` routine, and generate output in which each line is either of the form "`R [number]`", or of the form "`P [number]`". Lines of the first type indicate calls to `GetRequest()` and the replies received. Lines of the second type represent calls to `PutBack`() and the colors chosen to put back. The output is terminated by a line of the form "`E`".

Please note that on the official grader the running time of your program may differ slightly from the time on your local computer. This difference should not be significant. Still, you are invited to use the test interface in order to verify whether your solution runs within the time limit.

Time and Memory limits

- Time limit: 7 seconds.
- Memory limit: 256 MiB.