La última cena

Leonardo estaba muy activo cuando trabajaba en la Última Cena, su mural más famoso: una de sus primeras tareas diarias era decidir qué colores usaría durante ese día de trabajo. Necesitaba varios colores, pero sólo podía mantener un número limitado en su andamio. Su asistente estaba al cargo, entre otras cosas, de subir al andamio para darle los colores, y después bajar otra vez para dejarlos en una repisa.

En esta tarea, necesitarás escribir dos programas por separado para ayudar al asistente. El primer programa recibirá los pedidos de Leonardo (una secuencia de colores que Leonardo necesitará ese día), y creará una string *corta* de bits, a la que diremos *advice* (consejo). Mientras procesa los pedidos de Leonardo durante el día, el asistente no tendrá acceso a los pedidos futuros de Leonardo, solo al *advice* producido por tu primer programa. El segundo programa recibirá el advice, y luego recibirá y procesará los pedidos de Leonardo en modo online (o sea, de uno en uno). Este programa debe ser capaz de entender el significado del advice y usarlo para tomar decisiones óptimas. Todo será explicado a continuación de manera más detallada.

Moviendo los colores entre la repisa y el andamio

Consideraremos un escenario simplificado. Imagina que hay N colores numerados de 0 a N - 1, y que cada día Leonardo pide a su asistente un nuevo color exactamente N veces. Sea C la secuencia de los N colores pedidos. Podemos suponer que C es una secuencia de N números, cada uno entre 0 y N-1 (incluídos). Nótese que algunos colores pueden no ocurrir en C, y que otros pueden aparecer más de una vez.

El andamio siempre está lleno y contiene K de los N colores, con $K < N$. Inicialmente, el andamio contiene los colores de 0 a K - 1 (incluídos).

El asistente procesa los pedidos de Leonardo de uno en uno. Cuando el color pedido ya está en el andamio, el asistente descansa. En caso contrario, tiene que coger de la repisa el color pedido, y subirlo al andamio. Como no hay espacio en el andamio para el nuevo color, el asistente debe escoger uno de los colores que estén actualmente allí, y bajarlo a la repisa.

Estrategia óptima de Leonardo

El asistente quiere descansar tanto como pueda. El número de pedidos para los que puede descansar depende de sus elecciones durante el proceso. Concretamente, cada vez que debe bajarse un color, distintas elecciones pueden llevar a resultados diferentes en el futuro. Leonardo le explica que puede conseguir su objetivo si conoce C. La mejor manera de elegir el color a bajar se puede tomar sabiendo los colores en el andamio y los colores que quedan por pedir en C. Un color tendría que ser elegido siguiendo las reglas siguientes:

- Si hay un color en el andamio que no se volverá a usar en el futuro, el asistente debería bajar ese color.
- En caso contrario, el color a bajar debería ser *aquel que se necesite por primera vez más tarde en el futuro*. (O sea, para cada color encontramos la primera ocurrencia en el futuro. El color que se debería bajar es el que se necesitará último.)

Se puede demostrar que cuando se usa la estrategia de Leonardo, el asistente descansa el máximo número de veces.

Ejemplo 1

Sea N = 4, por lo tanto tenemos 4 colores (numerados de 0 a 3) y 4 pedidos. Sea la secuencia de pedidos C = (2, 0, 3, 0). Asume que K = 2. Esto es, Leonardo tiene un andamio capaz de contener 2 colores en cualquier momento. Como se ha descrito anteriormente, el andamio inicialmente tiene los colores 0 y 1. Escribiremos el contenido del andamio así: [0, 1]. Una opción para manejar los pedidos sería la siguiente.

- El primer pedido de color (número 2) no está en el andamio. El asistente lo pone allí y decide en bajar el color 1. El andamio actual contiene [0, 2].
- El siguiente pedido de color (número 0) ya está en el andamio, por lo que el asistente puede descansar.
- Para el tercer pedido (número 3), el asistente baja a la repisa el color 0, y cambia el andamio a [3, 2].
- Finalmente, el último pedido de color (número 0) tiene que sacarse de la repisa y subirse al andamio. El asistente decide bajar el color 2, y el andamio ahora contiene [3, 0].

Fíjate que en el ejemplo anterior el asistente no ha seguido la estrategia óptima, que sería bajar el color 2 en el tercer paso, para que así el asistente pudiera descansar otra vez en el último paso.

Estrategia del asistente cuando tiene memoria limitada

Por la mañana, el asistente pide a Leonardo que escriba C en una página, para que pueda encontrar y seguir la estrategia óptima. Sin embargo, Leonardo está obsesionado con mantener sus técnicas de trabajo secretas, así que se niega a que el asistente se quede con el papel. Sólo le permite leer la lista C e intentar memorizarla.

Desafortunadamente, la memoria del asistente es muy mala. Sólo es capaz de recordar hasta M bits. En general, esto podría evitar que pudiese reconstruir la secuencia C entera. Por tanto, el asistente tiene que encontrar alguna manera inteligente de calcular la secuencia de bits que recordará. Llamaremos a esta secuencia *secuencia advice*, y la denotaremos con una A.

Ejemplo 2

Por la mañana, el asistente puede cojer la página de Leonardo con la secuencia C, leerla, y tomar todas las decisiones que quiera. Una elección posible sería examinar el estado del andamio después de cada pedido. Por ejemplo, cuando usase la estrategia (sub-óptima) dada en el Ejemplo 1, la secuencia de los estados del andamio sería: [0, 2], [0, 2], [3, 2], [3, 0]. (Recuerda que ya sabe que el

estado inicial del andamio es [0, 1].)

Asume ahora que tenemos M = 16, por lo que el asistente es capaz de recordar hasta 16 bits de información. Como N = 4, podemos guardar cada color usando 2 bits. Por lo tanto, los 16 bits son suficientes para guardar la secuencia anterior de los estados del andamio. Así, el asistente calcula la secuencia de advice siguiente: A = (0, 0, 1, 0, 0, 0, 1, 0, 1, 1, 1, 0, 1, 1, 0, 0).

Más tarde en ese día, el asistente puede descodificar esta secuencia de advice y usarla en sus propias elecciones.

(Por supuesto, con M = 16 el asistente puede escoger recordar toda la secuencia C, usando sólo 8 de los 16 bits disponibles. Con este ejemplo, simplemente queríamos mostrarte que tiene otras opciones, sin descubrirte ninguna buena solución.)

Enunciado

Tienes que escribir *dos programas* en el mismo lenguaje de programación. Estos programas seran ejecutados secuencialmente, sin que se puedan comunicar el uno con el otro durante la ejecución.

El primer programa será el que el asistente usará por la mañana. Este programa recibirá la secuencia C, y tiene que calcular la secuencia advice A.

El asistente usará el segundo programa durante el día. Este programa recibirá la secuencia advice A, y luego tendrá que procesar la secuencia C de los pedidos de Leonardo. Fíjate que cada valor de la secuencia C será revelado de uno en uno, y que cada pedido debe ser procesado antes de recibir el siguiente.

Más precisamente, en el primer programa tienes que implementar la rutina `ComputeAdvice`(C, N, K, M) que tenga como entrada el array C de N enteros (cada uno entre 0, ..., N - 1), el número K de colores en el andamio, y el número M de bits disponibles para la secuencia advice. Este programa tiene que calcular la secuencia advice A, que consiste de hasta M bits. El programa debe comunicar la secuencia A al sistema llamando, para cada bit de A en orden, la rutina siguiente:

- `WriteAdvice`(B) — concatena el bit B a la secuencia advice A. (Puedes llamar a esta rutina como mucho M veces.)

En el segundo programa, tienes que implementar la rutina `Assist`(A, N, K, R). La entrada es la secuencia advice A, los enteros N y K definidos anteriormente, y el tamaño R de la secuencia A (R ≤ M). Esta rutina debe ejecutar tu estrategia para el asistente, usando las siguientes rutinas disponibles:

- `GetRequest`() — devuelve el siguiente color pedido por Leonardo. (No se revela ninguna información sobre los pedidos siguientes.)

- `PutBack`(T) — baja el color T del andamio a la repisa. Tienes que llamar a esta rutina con un T que sea uno de los colores que esté actualmente en el andamio.

Cuando se ejecuta, tu rutina `Assist` tiene que llamar a `GetRequest` exactamente N veces, cada vez recibiendo uno de los pedidos de Leonardo, y en orden. Después de cada llamada a

`GetRequest`, si el color pedido *no* está en el andamio, *tienes* que llamar a `PutBack`(T) con un color T a tu elección. En caso contrario, *no puedes* llamar a `PutBack`. Si tu rutina no cumple este requerimiento, se considerará un error y tu programa se abortará. Por favor, recuerda que el andamio contiene inicialmente los colores de 0 a K - 1, incluídos.

Un juego de pruebas se considerará solucionado si tus dos rutinas siguen las restricciones, y el número total de llamadas a `PutBack` es *exactamente igual* a la estrategia óptima de Leonardo. Fíjate que si hay maneras distintas de llegar al mismo número de llamadas a `PutBack`, tu programa puede realizar cualquiera de ellas. (O sea, no se requiere seguir la estrategia de Leonardo, si hay una estrategia igualmente buena)

Ejemplo 3

Continuando con el Ejemplo 2, asume que con `ComputeAdvice` hemos calculado A = (`0`, `0`, `1`, `0`, `0`, `0`, `1`, `0`, `1`, `1`, `1`, `0`, `1`, `1`, `0`, `0`). Para comunicarla al sistema, tenemos que hacer la secuencia de llamadas siguiente: `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)`.

Tu segunda rutina `Assist` sería entonces ejecutada, recibiendo la secuencia A anterior, y los valores N = 4, K = 2, y R = 16. Tu rutina `Assist` tendría que llamar exactamente N = 4 veces a `GetRequest`. También, después de alguno de los pedidos, `Assist` tendría que llamar a `PutBack`(T) con una elección correcta de T.

La tabla siguiente muestra una secuencia de llamadas correspondiente a la elección (sub-óptima) del ejemplo 1. El guión marca que no se ha llamado a `PutBack`.

`GetRequest()`	Acción
2	`PutBack`(1)
0	-
3	`PutBack`(0)
0	`PutBack`(2)

Subtarea 1 [8 puntos]

- $N \leq 5\,000$.
- Puedes usar como mucho M = 65 000 bits.

Subtarea 2 [9 puntos]

- $N \leq 100\,000$.
- Puedes usar como mucho M = 2 000 000 bits.

Subtarea 3 [9 puntos]

- $N \leq 100\,000$.
- $K \leq 25\,000$.
- Puedes usar como mucho $M = 1\,500\,000$ bits.

Subtarea 4 [35 puntos]

- $N \leq 5\,000$.
- Puedes usar como mucho $M = 10\,000$ bits.

Subtarea 5 [hasta 39 puntos]

- $N \leq 100\,000$.
- $K \leq 25\,000$.
- Puedes usar como mucho $M = 1\,800\,000$ bits.

La puntuación de esta subtarea depende de la longitud R de la secuencia advice que tu programa comunica. Más precisamente, si R_{max}</suv> es la longitud maxima (en todos los casos) de la secuencia advice producida por tu rutina `ComputeAdvice`, tu puntuación será:

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

Detalles de implementación

Tienes que enviar dos archivos *en el mismo lenguaje de programación*.

El primer archivo se debe llamar `advisor.c`, `advisor.cpp` o `advisor.pas`. Este archivo tiene que implementar la rutina `ComputeAdvice` descrita anteriormente y puede llamar a la rutina `WriteAdvice`. El segundo archivo se llama `assistant.c`, `assistant.cpp` o `assistant.pas`. Este archivo tiene que implementar la rutina `Assist` como hemos descrito anteriormente, y puede llamar a las rutinas `GetRequest` y `PutBack`.

Las cabeceras para todas las rutinas.

Programas en C/C++

```
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();
```

Programas en Pascal

```
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;
```

Estas rutinas tienen que comportarse como se ha descrito anteriormente. Por supuesto, estas rutinas pueden llamar a otras rutinas que hayas programado. Para los programas en C/C++, tienes que declarar tus rutinas internas con `static` , ya que el sample grader las enlazará juntas. Alternativamente, simplemente evita tener dos rutinas (una en cada programa) con el mismo nombre. Tu envío no debe interactuar de ningún modo con la entrada/salida estándar, ni con ningún otro fichero.

Cuando programes tu solución, tienes que tener en cuenta las siguientes instrucciones (los templates que puedes encontrar en tu entorno de competición ya cumplen los requisitos que siguen).

Programas en C/C++

Al principio de tu solución, tienes que incluir el archivo `advisor.h` y `assistant.h`, respectivamente en el advisor y el assistant. Esto se hace añadiendo en tu código la línea:

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

o

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

Los dos archivos `advisor.h` y `assistant.h` te serán entregados dentro de una carpeta en tu entorno de competición, y también serán ofrecidos en el interfaz Web. También se te proveerá (por los mismos canales) el código y scripts para compilar y testear tu solución. Específicamente, despues de copiar tu solución en la carpeta con estos scripts, tienes que ejecutar `compile_c.sh` o `compile_cpp.sh` (dependiendo del lenguaje de tu código).

Programas en Pascal

Tienes que usar las unidades `advisorlib` y `assistantlib`, respectivamente, en el advisor y en el assistant. Esto se hace incluyendo en tu código la línea:

```
uses advisorlib;
```

o

```
uses assistantlib;
```

Los dos archivos `advisorlib.pas` y `assistantlib.pas` te serán entregados dentro de un directorio en tu entorno de competición y también serán ofrecidos en el interfaz Web. También se te proveerá (por los mismos canales) con el código y scripts para compilar y testear tu solución. Específicamente, despues de copiar tu solución en el directorio con estos scripts, tienes que ejecutar `compile_pas.sh`.

Sample grader

El sample grader aceptará entrada con el formato siguiente:

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

El grader primero ejecutará la rutina `ComputeAdvice`. Esto generará un fichero `advice.txt`, que contendrá los bits individuales de la secuencia advice, separados con espacios y acabados en un 2.

Entonces procederá a ejecutar tu rutina `Assist`, y generará una salida en la cual cada línea es o de la forma "`R [number]`", o de la forma "`P [number]`". Líneas del primer tipo indican llamadas a `GetRequest()` y las respuestas recibidas. Líneas del segundo tipo son llamadas a `PutBack`() y los colores escogidos para put back. La salida acaba con una línea de la forma "`E`".

En el grader oficial el tiempo de ejecución de tu programa puede ser ligeramente diferente que en tu ordenador. Esta diferencia no debería ser significativa. A pesar de esto, te invitamos a usar la interfaz de test para verificar si tu solución se ejecuta dentro del time limit.

Requisitos de tiempo y memoria

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