La última Cena

Leonardo trabajó mucho en su más famoso mural: La Ultima Cena. Una de sus primeras tareas de cada día era decidir que colores iba a usar durante el resto del día. El necesitaba varios colores, pero solo podía mantener un número limitado de colores en su andamio. Leonardo tiene un asistente que le ayuda en esta tarea: Subirse al andamio para llevar colores a Leonardo y después bajar al suelo para ponerlos en un estante.

En esta tarea tienes que escribir dos programas por separado para ayudar al asistente. El primer programa recibe las instrucciones de Leonardo (una secuencia de colores que Leonardo va a necesitar durante el día), y crea un string "corto" de bits, llamado *secuencia de ayuda*. Cuando el asistente este ejecutando las peticiones de Leonardo durante el día, el no tendrá acceso a la lista de peticiones futuras de Leonardo, solo la "secuencia de ayuda" producida por tu primer programa. El segundo programa recibe tu "secuencia de ayuda", y después recibe las peticiones de Leonardo de forma "online fashion" (i.e., una a la vez). Este programa debe saber que significa la "secuencia de ayuda" y hacer decisiones óptimas. Todo está explicado abajo con más detalle.

Moviendo colores entre el estante y el andamio.

Vamos a considerar un escenario simple. Supongamos que hay N colores numerados de 0 a N-1 y que cada día Leonardo pide que le lleve algún color exactamente N veces. Sea C la secuencia de los N colores que pide Leonardo. Entonces podemos pensar en C como una secuencia de N números, cada uno entre 0 y N-1 inclusive. Nótese que no todos los posibles colores tienen que estar en C y algunos colores pueden aparecer múltiples veces.

El andamio esta siempre lleno y contiene K de los N colores con $K < N$. Inicialmente el andamio contiene los colores con índice del 0 al K-1 inclusive.

El asistente procesa los pedidos de Leonardo uno a la vez. Sin embargo, el asistente puede descansar si el color que requiere Leonardo ya se encuentra en el andamio, de lo contrario el asistente tiene que tomar el color del estante y subir a ponerlo en el andamio. Por supuesto, si no hay lugar en el andamio para el nuevo color, el asistente tiene que seleccionar un color del andamio y bajarlo al estante.

La estrategia óptima de Leonardo.

El asistente quiere descansar lo más que sea posible. El número de pedidos en los que puede descansar depende de lo que elija durante el proceso. Leonardo le explicó al asistente como puede completar su meta si sabe C: cuando el color que pide Leonardo no está en el andamio, su mejor opción es retrasar lo más que sea posible el momento en el que esta situación se repita de nuevo. Esto se obtiene examinando los colores que ya se encuentran en el andamio y los colores faltantes

en C. El color (que moverá del andamio al estante) debe ser seleccionado entre los otros del andamio de tal manera que:

- Nunca se va a requerir ese color de nuevo o
- El color va a ser requerido después de cualquier otro color del andamio.(Es decir, de todos los colores del andamio, debe regresar al estante el que va a ser utilizado al último.)

Es posible probar que usando la estrategia óptima de Leonardo, el asistente descansará la mayor cantidad de veces posible.

Ejemplo 1

Sea N=4, es decir tenemos 4 colores (numerados de 0 a 3) y 4 pedidos de Leonardo. Sea la secuencia de pedidos C=(2,0,3,0).También asumimos que K=2, es decir el andamio de Leonardo puede tener 2 colores a la vez. Como se mencionó arriba, el andamio inicialmente contiene los primeros K colores, en este caso 0 y 1. Escribiremos el conjunto de colores que están en andamio de la siguiente manera: [0,1]. Una posible manera de realizar las peticiones de Leonardo es la siguiente:

- El primer color pedido (numero 2) no está en el andamio. El asistente lo pone en el andamio y decide bajar el color 1. El andamio actual es [0, 2].
- El siguiente pedido (numero 0) ya se encuentra en el andamio, el asistente puede descansar.
- Para el tercer pedido (numero 3), el asistente remueve del andamio el color 0, cambiando el andamio a [3,2].
- Finalmente, el último color pedido (numero 0) tiene que ser movido del estante al andamio. El asistente decide quitar del andamio el color 2 y el andamio se convierte en [3, 0].

Nótese que en el ejemplo anterior el asistente no siguió la estrategia óptima de Leonardo. La estrategia óptima de Leonardo sería remover el color 2 en el tercer paso y así el asistente puede descansar en el último paso.

Estrategia del asistente cuando la memoria está limitada.

En la mañana, el asistente le pide a Leonardo que escriba C en una pedazo de papel, así el puede encontrar la estrategia óptima. Sin embargo, Leonardo está obsesionado con mantener sus técnicas en secreto, así que no deja que el asistente se quede con el papel. El solo tiene permitido leer C e intentar recordarlo.

Desafortunadamente, la memoria del asistente es muy mala. El solo puede recordar a lo más M bits. En general, este puede causar que no pueda reconstruir toda la secuencia C. Por lo tanto, el asistente tiene que encontrar una forma inteligente de procesar la secuencia de bits que él va a recordar. Llamaremos a esta secuencia “secuencia de ayuda” y la denotaremos por A.

Ejemplo 2

En la mañana, el asistente puede tomar el papel de Leonardo con la secuencia C, leer la secuencia, y tomar todas las decisiones necesarias. Una de las cosas que el asistente puede decidir hacer es

escribir el estado del andamio para cada pedido de Leonardo. Por ejemplo usando la (sub-optima) estrategia del ejemplo 1, la secuencia de estados del andamio sería [0, 2], [0, 2], [3, 2], [3, 0]. (Recordemos que el estado inicial del andamio es conocido [0, 1], no hay necesidad de escribirlo.)

Ahora asumamos que tenemos M=16, entonces el asistente puede recordar hasta 16 bis de información. Como N=4, podemos guardar cada color usando 2 bits. Por lo tanto 16 bits es suficiente para guardar toda la secuencia de estados de andamio anterior. Entonces el asistente genera la siguiente "secuencia de ayuda"; A=

```
(0,  0,  1,  0,  0,  0,  1,  0,  1,  1,  1,  0,  1,  1,  0,  0).
```

Más tarde en el mismo día, el asistente puede decodificar la "secuencia de ayuda" y usar todas las decisiones que guardó en ella —incluso si en ese momento no recuerda la secuencia C.

(Por supuesto, con M=16 el asistente puede recordar la secuencia entera C usando solo 8 bits de los 16 disponibles. En este ejemplo solo quiere ilustrar que se tienen otras opciones, sin revelar alguna buena solución.)

Problema

Tienes que escribir "dos problemas separados" en el mismo lenguaje de programación. Estos programas serán ejecutados secuencialmente, sin la posibilidad de que se comuniquen durante la ejecución.

El primer programa es el que usará el asistente en la mañana. Este programa recibirá la secuencia C, y tiene que generar una "secuencia de ayuda" A.

El segundo programa es el que usara el asistente durante el resto del día. Este programa recibirá la "secuencia de ayuda" A generada por el programa anterior y después tiene que procesar la secuencia C de pedidos de Leonardo. Nótese que la secuencia C, será revelada paso por paso y cada paso tiene que ser procesado (decidir qué hacer con las pinturas) antes de recibir el siguiente pedido.

De forma más precisa, en el primer programa tienes que implementar solo la función `ComputeAdvice`(C, N, K, M) que tiene como entrada el arreglo C de N enteros (cada uno entre 0 y N-1), el numero K de colores en el andamio y el máximo número de bits M disponibles para la "secuencia de ayuda". El programa debe generar una "secuencia de ayuda" A que consiste de a lo más M bits. Después tu programa debe comunicar al sistema la secuencia A, mandando cada bit de A en orden, con la siguiente función:

- `WriteAdvice`(B) — Agrega al final de la "secuencia de ayuda" A el bit B. (Puedes llamar esta función a lo mas M veces).

En el segundo programa tienes que implementar una sola función `Assist`(A, N, K, R). La entrada de esta rutina es la "secuencia de ayuda" A, los enteros N y K definidos anteriormente y R: la cantidad de bits de la secuencia A ($R \leq M$). Esta función debe ejecutar tu estrategia propuesta usando las siguientes funciones:

- `GetRequest`() — regresa el siguiente color pedido por Leonardo (No se revela información sobre los siguientes pedidos)
- `PutBack`(T) — pone el color T del andamio en el estante. Solo puedes llamar esta función si T es uno de los colores que actualmente se encuentran en el andamio.

Durante la ejecución, tu programa `Assist` debe llamar `GetRequest` exactamente N veces, en cada una recibe una de las peticiones de Leonardo en orden. Después de cada llamada `GetRequest`, si el color requerido "no está" en el andamio, tu "tienes que" mandar llamar la función `PutBack`(T) con el color T que tu escojas. De lo contrario, "no debes" llamar `PutBack`. Si fallas al hacerlo, será considerado como error y causara que tu programa termine. Recuerda que al inicio el andamio contiene los colores con índice de 0 a K-1, inclusive.

Un caso de prueba particular será considerado resuelto si tus dos funciones siguen todas las restricciones ateriores, y el número total de llamadas de `PutBack` es "exactamente igual" al de la estrategia óptima de Leonardo. Si existen varias estrategias optimas que logren el mismo número de llamadas a `PutBack`, tu programa puede realizar cualquiera de ellas. (I.e., no es necesario seguir la estrategia óptima de Leonardo si existe otra estrategia que es igual de buena que la de Leonardo.)

Ejemplo 3

Continuando con el ejemplo 2, asumamos que en `ComputeAdvice` tu generas A = (`0`, `0`, `1`, `0`, `0`, `0`, `1`, `0`, `1`, `1`, `1`, `0`, `1`, `1`, `0`, `0`). Para poder comunicarlo al sistema, debes hacer la siguiente secuencia de llamadas: `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 función `Assist` sería ejecutada, recibiendo la "secuencia de ayuda" A anterior, y los valores N = 4, K = 2, y R = 16. La función `Assist` tiene que hacer exactamente N = 4 llamadas a `GetRequest`. También, después de algunas peticiones, `Assist` tendrá que llamar `PutBack`(T) con una elección adecuada de T.

La tabla inferior muestra la secuencia de llamadas correspondientes a las elecciones (sub-optimas) del Ejemplo 1. El guion denota que no se llamó a `PutBack`.

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

Sub-tarea 1 [8 puntos]

- $N \leq 5\,000$.
- Puedes usar a lo mas M = 65 000 bits.

Sub-tarea 2 [9 puntos]

- $N \leq 100\,000$.
- Puedes usar a lo mas M = 2 000 000 bits.

Sub-tarea 3 [9 puntos]

- $N \leq 100\,000$.
- $K \leq 25\,000$.
- Puedes usar a lo mas M = 1 500 000 bits

Sub-tarea 4 [35 puntos]

- $N \leq 5\,000$.
- Puedes usar a lo mas M = 10 000 bits.

Sub-tarea 5 [hasta 39 puntos]

- $N \leq 100\,000$.
- $K \leq 25\,000$.
- Puedes usar a lo mas M = 1 800 000 bits.

El puntaje para esta sub-tarea depende de la longitud de R de la "secuencia de ayuda" que tu programa genera. De forma más precisa, si R_{max} el máximo (sobre todos los casos) de la longitud de de la "secuencia de ayuda" que genera tu programa `ComputeAdvice`, entonces tu puntaje 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

Debes mandar exactamente dos archivos *en el mismo lenguaje de programación*.

El primer archivo debe ser llamado `advisor.c`, `advisor.cpp` o code>advisor.pas</code>. En este archive debe estar implementada la función `ComputeAdvice` como se describe arriba y puede llamar a la función `WriteAdvice`. El segundo archivo debe ser llamado`assistant.c`, `assistant.cpp` o `assistant.pas`. En este archive debe estar implementada la función `Assist` como se describe arriba y puede llamar a la funciones `GetRequest` y `PutBack`.

Las firmas para todas las rutinas siguen lo siguiente.

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 funciones se deben comportar como se describe anteriormente. Por supuesto, puedes implementar otras funciones para uso interno. Para programas en C/C++, tus funciones deben ser declaradas estáticas, así el "simple grader" las puede linekar juntas. Tus envíos no pueden interactuar de ninguna manera con la entrada y salida estándar o con algún otro archivo.

Cuando estés programando tu solución, tienes que tener cuidado con las siguientes instrucciones (los templates que puedes encontrar en el sistema ya satisfacen los requerimientos de abajo).

Programas en C/C++

Al principio de tu solución, tienes que incluir el archivo `advisor.h` y `assistant.h`, respectivamente, en el primer y segundo programa. Esto se hace incluyendo lo siguiente en tu código:

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

o

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

Los dos archivos `advisor.h` y `assistant.h` los podrás encontrar en un directorio dentro de tu sistema y también los podrás encontrar en la página Web del concurso. También tendrás (de la misma forma que los archivos anteriores) código y scripts para probar tus soluciones. Específicamente, después de copiar tus soluciones a las carpeta que contienen los scripts, tienes que correr `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. Esto se hace incluyendo las siguientes líneas en tu código:

```
uses advisorlib;
```

o

```
uses assistantlib;
```

Los dos archivos `advisorlib.pas` y `assistantlib.pas` los podrás encontrar en un directorio dentro de tu sistema y también los podrás encontrar en la página Web del concurso. También tendrás (de la misma forma que los archivos anteriores) código y scripts para probar tus soluciones. Específicamente, después de copiar tus soluciones a las carpeta que contienen los scripts, tienes que correr `compile_c.sh` o `compile_cpp.sh` (dependiendo del lenguaje de tu código).

Evaluador de prueba (Sample grader)

El evaluador de prueba aceptará entradas con el siguiente formato.

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

El evaluador ejecutará la siguiente función `ComputeAdvice`. Esto generará un archivo `advice.txt`, que contiene los bits individuales de la “secuencia de ayuda”, separada por espacios y termina con un 2.

Después ejecutara tu función `Assist`, y generará una salida donde cada línea contiene "`R [numero]`", o de la forma "`P [numero]`". Líneas del primer tipo indican llamadas a `GetRequest()` y la respuesta recibida. Líneas el segundo tipo representan llamadas a `PutBack`() y los colores regresados. La salida es terminada por una línea de la forma "`E`".

El evaluador oficial puede trabajar diferente. En particular la forma substancial de trabajar puede ser diferente. Estas invitado a usar el test del evaluador oficial en la página web.

Tiempo y memoria limite

- Tiempo límite: 7 segundos.
- Memoria límite: 256 MiB.