La Última Cena

Leonardo era muy activo cuando trabajaba en su Última Cena, su más famoso mural: una de sus tareas diarias era decidir qué colores de témpera iba a usar durante el resto del día. Él necesitaba muchos colores pero sólo podía tener una cantidad limitada con él en el andamio. Su asistente estaba a cargo, entre otras cosas, de escalar el andamio para entregarle sus colores y después bajar a dejar el resto en una mesa cerca al piso.

En esta tarea, usted debe escribir dos programas distintos para ayudar al asistente. El primer programa recibirá las instrucciones de Leonardo (una secuencia de colores que Leonardo necesitará durante el día), y creará una *corta* secuencia de bits, llamada *ayuda*. Cuando procese las solicitudes de Leonardo durante el día, el asistente no tendrá acceso a la lista de pedidos futuros de Leonardo, solo a la ayuda producida por su primer programa. El segundo programa recibirá la ayuda, y luego recibirá y procesará las solicitudes de Leonardo, una a la vez. Este programa debe ser capaz de entender lo que la ayuda significa y usarla para tomar decisiones óptimas. Todo está explicado en más detalle a continuación.

Moviendo los colores entre la mesa y el andamio

Cosideraremos un escenario simplificado. Suponga que hay exactamente N colores numerados de 0 a N - 1, y que cada día Leonardo le pide a su asistente un nuevo color exactamente N veces. Sea C la secuencia de N peticiones de color hechas por Leonardo. Podríamos considerar C como una secuencia de N números, cada uno entre 0 y N - 1, inclusive. Note que algunos colores pueden no aparecer en C y otros pueden aparecer más de una vez.

El andamio siempre está lleno y contiene algunos K de los N colores, donde $K < N$. Inicialmente, el andamio contiene los colores del 0 al K - 1, inclusive.

El asistente procesa las peticiones de Leonardo una por una. Cuando el color pedido ya *está en el andamio* el asistente puede descansar. En caso contrario, él debe recoger el color pedido de la mesa y ponerlo en el andamio. Por supuesto, no hay espacio en el andamio para el nuevo color, por lo que el asistente debe escoger uno de los colores del andamio y debe ponerlo de vuelta en la mesa.

La estrategia óptima de Leonardo

El asistente quiere descansar lo más que pueda. El número de pedidos en los cual él puede descansar depende de cómo escoja los colores durante el proceso. Más precisamente, cada vez que el asistente remueva un color del andamio, decisiones diferentes pueden tener resultados diferentes en el futuro. Leonardo le explica cómo puede lograr esto conociendo C. La mejor opción para remover el color del andamio se obtiene examinando los colores actualmente en el andamio y las peticiones restantes en C. Un color debería ser escogido entre los que están en el andamio

siguiendo las siguientes reglas:

- Si hay un color en el andamio que nunca va a ser necesitado en el futuro, el asistente debería remover dicho color del andamio.
- De otra forma el color removido del andamio debería ser aquél que *será necesitado lo más adelante posible en el futuro*. (Es decir, por cada uno de los colores en el andamio encontramos su primera ocurrencia en el futuro. El color que se devuelve a la mesa es el que será necesitado de último.)

Se puede probar que usando la estrategia de Leonardo el asistente descansará la mayor cantidad de veces posible.

Ejemplo 1

Sea N = 4, entonces tenemos cuatro colores (numerados de 0 a 3) y cuatro pedidos. Sea la secuencia de pedidos C = {2, 0, 3, 0}. Además, suponga que K = 2. Es decir, Leonardo tiene un andamio capaz de tener 2 colores al mismo tiempo. Como fue descrito anteriormente, el andamio comienza con los colores 0 y 1. Describiremos los colores del andamio de la siguiente forma: [0, 1]. Una forma en la cual el asistente puede manejar los pedidos es la siguiente.

- El primer color pedido (el número 2) no está en el andamio. El asistente lo pone ahí y decide quitar el color 1 del andamio. El andamio entones queda [0, 2].
- El siguiente color pedido (el número 0), ya está en el andamio, por lo que el asistente puede descansar.
- Para el tercer pedido (el número 3), el asistente remueve el color 0, el andamio entonces queda [3, 2].
- Por último, el último color pedido (el número 0) tiene que llevarse de la mesa al andamio. El asistente decide quitar el color 2, el andamio entonces queda [3, 0].

Note que en el ejemplo anterior el asistente no sigió la estrategia óptima de Leonardo. La estrategia óptima hubiera sido haber quitado el color 2 en el tercer pedido para que el asistente hubiera podido descansar en el último pedido.

La estrategia del asistente cuando su memoria está limitada

En la mañana, el asistente le pide a Leonardo que escriba C en una hoja de papel, para que él pueda encontrar y seguir la estrategia óptima. Sin embargo, Leonardo está obsesionado con dejar en secreto sus inventos, por lo cual se rehusa a dejarle el papel a su asistente. Sólo le permite leer C e intentar recordarlo.

Desafortunadamente, la memoria del asistente es muy mala. Él solo puede recordar hasta M bits. En general, esto puede evitar que él reconstruya completamente la secuencia C. Entonces el asistente se ha inventado una manera inteligente de calcular la secuencia de bits que él recordará. Él llamará esta secuencia *secuencia de ayuda* y la denotará con la letra A.

Ejemplo 2

En la mañana, el asistente puede llevarse el papel de Leonardo con la secuencia C, leer la

secuencia, y hacer todas las decisiones necesarias. Una cosa que él puede hacer es examinar el estado del andamio después de cada uno de los pedidos. Por ejemplo, cuando usa la estrategia (sub-óptima) del Ejemplo 1, la secuencia de estados del andamio sería [0, 2], [0, 2], [3, 2], [3, 0]. (Recuerde que él sabe que el estado inicial es [0, 1].)

Ahora asuma que M = 16, entonces el asistente puede recordar hasta 16 bits de información. Como N = 4, podemos guardar cada color usando 2 bits. Por lo que 16 bits es suficiente para almacenar la anterior secuencias de estados del andamio. Por lo que el asistente calcula la siguiente secuencia de ayuda: A = (`0`, `0`, `1`, `0`, `0`, `0`, `1`, `0`, `1`, `1`, `1`, `0`, `1`, `1`, `0`, `0`).

Más tarde durante el día, el asistente puede decodificar esta secuencia y usarla para tomar sus decisiones.

(Por supuesto, con M = 16, el asistente puede recordar toda la secuencia C, usando solo 8 de los 16 bits. En este ejemplo queríamos ilustrar que él puede tener otras opciones, sin dar una buena solución.)

Enunciado

Tiene que escribir *dos programas distintos* en el mismo lenguaje de programación. Estos programas se ejecutarán secuencialemente, sin tener forma de comunicarse entre ellos durante la ejecución.

El primer programa será usado por el asistente en la mañana. A este programa se le dará la secuencia C y deberá calcular la secuencia de ayuda A.

El segundo programa será el que use el asistente durante el día. Este programa recibirá la secuencia de ayuda A, y debe procesar la secuencia de pedidos de Leonardo C. Note que la secuencia C sólo le será revelada a este programa un pedido a la vez, cada pedido debe ser procesado antes de recibir el siguiente.

Más precisamente, el primer programa debe implementar una sola subrutina `ComputeAdvice`(C, N, K, M) que tiene como entrada un arreglo 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 ayuda. Este programa debe calcular una secuencia de ayuda A que consiste de a lo mucho M bits. El programa entonces debe comunicar esta secuencia A al sistema llamando, para cada bit de A, en orden, la siguiente rutina:

- `WriteAdvice`(B) — añada el bit B a la secuencia actual de ayuda A. (puede invocar esta subrutina a lo mucho M veces).

En el segundo programa usted debe implementar una única rutina `Assist`(A, N, K, R). La entrada de esta rutina es la secuencia de ayuda A, los enteros N y K definidos anteriormente, y la longitud real de la secuencia A en bits ($R \leq M$). Esta rutina debería ejecutar su estrategia propuesta para el asistente usando las siguientes rutinas que se le proveen:

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

- `PutBack`(T) — pone el color T del andamio de vuelta en la mesa. Solo puede llamar esta rutina con T siendo uno de los colores actualmente en el andamio.

Cuando se ejecuta su rutina `Assist` debe llamar `GetRequest` exactamente N veces, cada vez se recibirá una orden de Leonardo. Después de cada llamada a `GetRequest`, si el color que retornó *no* está en el andamio, usted *debe* llamar `PutBack`(T) con su elección de T. En caso contrario usted *no debe* llamar `PutBack`. Algún fallo con esto será considerado un error y causará la finalización de su programa. Recuerde que al principio el andamio contiene los colores numerados de 0 a K - 1, inclusive

Un caso de prueba particular se considerará resuelto si sus dos rutinas siguen las restricciones descritas y en total el número de llamadas a `PutBack` es *exactamente igual* al número de llamados de la estrategia óptima de Leonardo. Note que si hay más de una estrategia que logre el mismo número de llamadas a `PutBack`, su programa puede realizar cualquiera de ellas. (Es decir, no es obligatorio seguir la estrategia si hay otra estrategia igual de buena.)

Ejemplo 3

Siguiendo el Ejemplo 2, asuma que en `ComputeAdvice` usted calculó A = (`0`, `0`, `1`, `0`, `0`, `0`, `1`, `0`, `1`, `1`, `1`, `0`, `1`, `1`, `0`, `0`). Para comunicárselo al sistema usted debería 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`).

Su segunda rutina `Assist` será luego ejecutada, recibiendo la anterior secuencia A, y los valores N = 4, K = 2 y R = 16. La rutina `Assist` tiene que entonces hacer exactamente N = 4 llamadas a `GetRequest`. Además, después de esos pedidos, `Assist` tendrá que llamar a `PutBack`(T) con un T apropiado.

La siguiente tabla muestra una secuencia de llamadas correspondientes a las elecciones (sub-óptimas) del Ejemplo 1. El guión representa que no hubo llamada a `PutBack`.

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

Subtarea 1 [8 puntos]

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

Subtarea 2 [9 puntos]

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

Subtarea 3 [9 puntos]

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

Subtarea 4 [35 puntos]

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

Subtarea 5 [hasta 39 puntos]

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

El puntaje para esta subtarea depende de la longitud R de la ayuda que su programa comunica. Más precisamente, si R_{max} es el máximo (sobre todos los casos de prueba) de la longitud de la secuencia de ayuda que provee su rutina `ComputeAdvice`, su 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

Usted deberá enviar exactamente dos archivos *en el mismo lenguaje de programación*.

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

Los encabezados para todas las rutinas son los siguientes.

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;
```

Esta rutinas deben comportarse como fue descrito anteriormente. Por supuesto, usted es libre de implementar otras rutinas para uso interno. Para programas en C/C++, sus rutinas deben ser declaradas como `static`, ya que el calificador proveido las unirá. Alternativamente, basta con evitar tener dos rutinas (cada una en un programa) con el mismo nombre. Sus envíos no deben interactuar de ninguna forma con la entrada o salida estándar, o con algún otro archivo.

Cuando programe su solución, usted también debe seguir las siguientes instrucciones (las plantillas que encuentra en su ambiente ya satisfacen los siguientes requerimientos).

Programas en C/C++

Al principio de su solución, usted debe incluir el archivo `advisor.h` y `assistant.h`, respectivamente, en la ayuda y en el assistente. Esto se hace incluyendo en su archivo la línea:

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

o

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

Los dos archivos `advisor.h` y `assistant.h` se le proveerán en un directorio dentro de su ambiente y también será proveido por la Interfaz Web. A usted también se le proveerá (por los mismos medios) con códigos para compilar y probar su solución. Especificamente, después de copiar su solución al directorio con estos códigos, usted tendrá que correr `compile_c.sh` o `compile_cpp.sh` (dependiendo del lenguaje de su código).

Programas en Pascal

Usted debe usar las unidades `advisorlib` y `assitantlib`, respecivamente, en el ayudante y el asistente. Esto se hace incluyendo en su código la línea:

```
uses advisorlib;
```

o

```
uses assistantlib;
```

Los dos archivos `advisorlib.pas` y `assitantlib.pas` se le proveerán en un directorio en su ambiente y también estarán disponibles en la interfaz web. A usted también se le proveerá (por los mismos medios) con código para compilar y probar su solución. Especificamente, después de copiar su solución en un directorio con estos códigos, usted deberá correr `compile_pas.sh`.

Calificador ejemplo

El calificador ejemplo aceptará entradas en el siguiente formato:

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

El calificador primero ejecutará la rutina `ComputeAdvice`. Esto generará un archivo `advice.txt`, con los bits individuales de la secuencia ayuda, separados por espacios y terminando con un 2.

Luego procederá a ejecutar su rutina `Assist`, y a generar una salida donde cada línea es de la forma "`R [número]`", o de la forma "`P [número]`". Las líneas del primer tipo indican llamadas a `GetRequest()` y las respuestas recibidas. Las líneas del segundo tipo representan llamadas a `PutBack`() y los colores que escogen devolver. La salida se termina con una línea de la forma "`E`".

Note que en el calificador oficial el tiempo de ejecución puede diferir del tiempo que tome en su computador. La diferencia no debería ser significativa. Sin embargo, se le invita a usar la interfaz de prueba para verificar si su solución corre dentro de los límites de tiempo

Límites de tiempo y memoria

- Límite de tiempo: 7 segundos.
- Límite de memoria: 256 MiB.